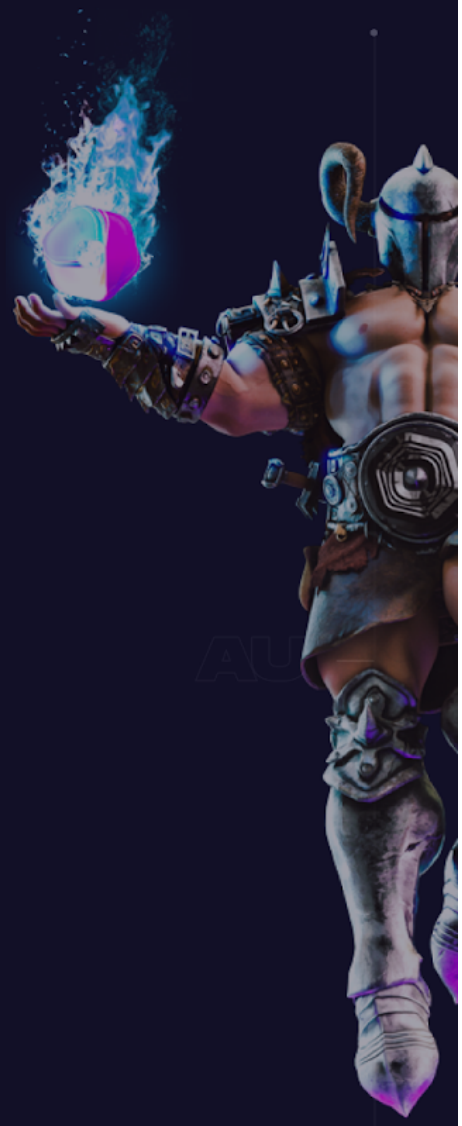
UnoRe

Insured Audit: Code Review & Protocol Security Report



wefi

Protocol

WeFi

Date

1st July 2023

The UnoRe security research team has completed an initial time-boxed security review of the **WeFi (Previous Paxos Finance)** Protocol contract code, with a focus on the security aspects of the application's implementation.

Disclaimer

This report containing confidential information can be used internally by the Customer, or it can be disclosed publicly after all the vulnerabilities are fixed - upon the decision of the Customer.

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts.

Document Changelog:

13th June 2023	**Audit Kickoff and Scoping**
20th June 2023	**Active Monitoring Notes**
1st July 2023	**WeFi Post-Triage Insured Audit Report**

Technical Overview

WeFi is a decentralized money market protocol that opens up investment loan options in DeFi for global users to invest in digital assets via multi-pool borrowing. WeFi allows cryptocurrency lenders to enjoy attractive, sustainable yields. Conversely, borrowers access predictable investment loans empowering them to invest in the lucrative digital asset market. WeFi supports lending borrowing and settling. You can read more about the protocol in the full documentation at https://docs.wefi.xyz/

Threat Model

Internal Security QA

1. Q: When adding new markets to the protocol, what precautionary steps are you taking to ensure the supply never goes to zero? Are you guys minting a minimum amount of tokens or potentially even burning it afterward - so the supply never goes to zero?

   A: We mint some tokens as market goes live. We haven't burn any will do that later

2. Q: Are you currently using any rate controllers to regulate the market, or are you planning to use one anytime soon? Is it custom-built, or are you using a 3rd party service provider?

   A: Right now we are dependent on chain link price feeds. To align with your policy we can work together to handle excessive rate fluctutations.

3. Q: What markets do you plan to include initially, and on which chains? Is it only the ones at the front end (wmatic, weth, wbtc, etc.)? Are chainlink feeds available for all of them?

   A: USDT, USDC, WMATIC, WETH, WTBC, AAVE, QUICKSWAP, CHAINLINK, UNISWAP TOKENS. Yes chainlink feeds area available

Severity classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack
Likelihood - the chance that a particular vulnerability gets discovered and exploited
Severity - the overall criticality of the risk

Security Review Summary

Last Review commit hash WeFi - 70842cd2b30df121f496963bedfbd52a497542a8

Audit Scope

The following smart contracts were in scope of the audit:

Precommit hash: 268cec6acdade1da4f0911d125760f2195d8df12

- DexAggregator.sol
- PaxoPriceFeed.sol
- CErc20.sol
- CToken.sol
- comptroller.sol
- ErrorReporter.sol
- Unitroller.sol
- WhitePaperInterestRateModel.sol
- ComptrollerStorage.sol
- CarefulMath.sol
- ExponentialNoError.sol
- Exponential.sol

The following number of issues were found, categorized by their severity:

- Critical: 1 issue
- High: 2 issues
- Medium: 1 issue
- Low: 2 issues
- Informational: 7 issues
- Gas Optimization: 10 issues

Note: The above summary of report findings at the Pre-Triage stage, most of these issues were addressed/fixed in consecutive stages.

Summary Table of Our Findings

ID	TITLE	Severity	Status
[C-01]	claimComp() of Comptroller wont work on actual mainnet, freezing user's yields	Critical	Fixed
[H-01]	accountAssets is not capped properly and may lock user funds	High	Fixed
[H-02]	Incorrect assumption of USD price feed being 8 decimals always, can lead to loss of funds	High	Fixed
[M-01]	Chainlink's latestRoundData() can return stale or incorrect result	Medium	Fixed
[L-01]	Vulnerable versions of dependencies are used	Low	Fixed
[L-02]	Return value for doTransferIn is ignored for DexAggregator	Low	Fixed
[I-01]	DexAggregator doesn't compile due to	Informational	Fixed

	incorrect file extension for QuickSwapRouter		
[I-02]	Multiple SPDX licenses used in the same contract	Informational	Fixed
[I-03]	Unsafe pragma statement	Informational	Fixed
[I-04]	Unclear error messages	Informational	Fixed
[I-05]	Unused import in multiple contracts	Informational	Fixed
[I-06]	Unnecessary NonReentrant modifier	Informational	Fixed
[I-07]	Open TODOs and Typos	Informational	Fixed
[G-01]	Use unchecked in for loops	Gas Optimization	Fixed
[G-02]	Cache array length outside of loop	Gas Optimization	Fixed
[G-03]	Using private rather than public for constants, saves gas	Gas Optimization	Fixed
[G-04]	Use x != 0 instead of x > 0 for uint types	Gas Optimization	Fixed
[G-05]	Use require instead of assert	Gas Optimization	Fixed
[G-06]	Splitting require() statements that use && saves gas	Gas Optimization	Fixed
[G-07]	Constructors can be marked payable	Gas Optimization	Fixed
[G-09]	Change all require across all contracts to Custom Errors	Gas Optimization	Fixed

Triage Fix Comments

1. **[C-01] claimComp() of Comptroller wont work on actual mainnet, freezing user's yields**

   This issue is known to us. Recently we had our token launch. We will upgrade the wefi token contract address at getCompAddress(). We can't verify contracts on mainnet at this stage now. Once audit and insurance is completed we can think of it.

2. **[H-01] accountAssets is not capped properly and may lock user funds**

   Yes, the pools/markets will be keep on adding to the protocol with time.
   Having a large array of assets will end up gas errors for user.
   We will proceed with "maxAssets" implementation.

3. **[M-01] Chainlink's latestRoundData() can return stale or incorrect result**
   I went through the recommendation. We can proceed implementing points below.

   a. **Sanity Check on latestRoundData() output.**
   b. **Staleness Check. Suggest us what should be ideal timeout for it.**
   c. **Chainlink L2 Sequencer Uptime Feed and Grace Period.**

   It will revert if any of them fails, will prevent borrow, withdraw and liquidations.

4. **[L-01] Vulnerable versions of dependencies are used**

   Yes, make sense to upgrade it to latest. Need to check dependency upgrade compatibility.

Centralization Risk Areas:

We have also identified several key areas within the protocol which contains centralization risks which needs to be made aware to the community and have highlighted them below:

Price Feed (0xe696dD106ad4C0e541De30a7820409dB95d1e7cF)

- setpricefeed

Unitrollller (0x1eDf64B621F17dc45c82a65E1312E8df988A94D3)

- _setpendingadmin
- _acceptAdmin
- _setPendingImplementation

Comptroller (0x303A22423B1B8CFA5fb00988ade50c2618D8721D)

- _setpriceOracle
- _setCloseFactor
- _setCollateralFactor
- _setLiquidationIncentive
- _supportMarket
- _setMarketBorrowCaps
- _setBorrowCapGuardian
- _setPauseGuardian
- _setMintPaused
- _setBorrowPaused
- _setTransferPaused
- _setSeizePaused
- _become
- _setCompSpeeds
- _supportBuy
- _setBuyFactor
- _supportSettle

Dex Aggregator (0xA42e5d2A738F83a1e1a907eB3aE031e5A768C085)

- setDexRouter

POOLS/MARKETS

- Admin
- changeAdmin
- setDexAggregator
- _setPendingAdmin
- _acceptAdmin()
- _setReserveFactorFresh
- _reduceReservesFresh
- _setInterestRateModelFresh

Listed Below are all the addresses for markets

USDT (Borrow Vault),0x78718e0cADA1CA39fA23FC710e4accfd0FF9a5E9
USDC (Borrow Vault),0x34Aa3Fb5F02df5FBD03545ED3b35b90E51eEE88D
WETH (Borrow Vault),0x1e8Cd6843Db6286F4c7cF86EFDA572ECcC4DFF05
WMATTIC (Buy/Investment Vault),0x2588c9214376dDb524DCfa72859B522E745283Bf
WETH (Buy/Investment Vault),0xd5700c677Ddc823B0b6cC9a5Dc8415C07842Ef35
WBTC (Buy/Investment Vault),0x8f2DaDC86e7441B613ba1ab6A42193b1930E15ac

USDC (Buy/Investment Vault),0xef0b81a38a3e61c94D1c3F9D24c22FAa9772Fa95
Aave (Buy/Investment Vault),0x6dC71aC27CEE3eaAE4765258C29C427687974002
Quickswap (Buy/Investment Vault),0x2a500d5707F8FbBaeD35b99867AF94DfE86F36D6
ChainLink Token (Buy/Investment Vault),0xD0B5A656Fb3Fa2B5CC61cB5F2aD7904C39D75D50
Uniswap (Buy/Investment Vault),0x12F38c40D605e5Add1D5b92359B3be8157ABDB6b
WeFi Token Vault (Buy/Investment Vault),0xB50FF38F0fB98Dd5f9FB76ea333927BeB8F1920c

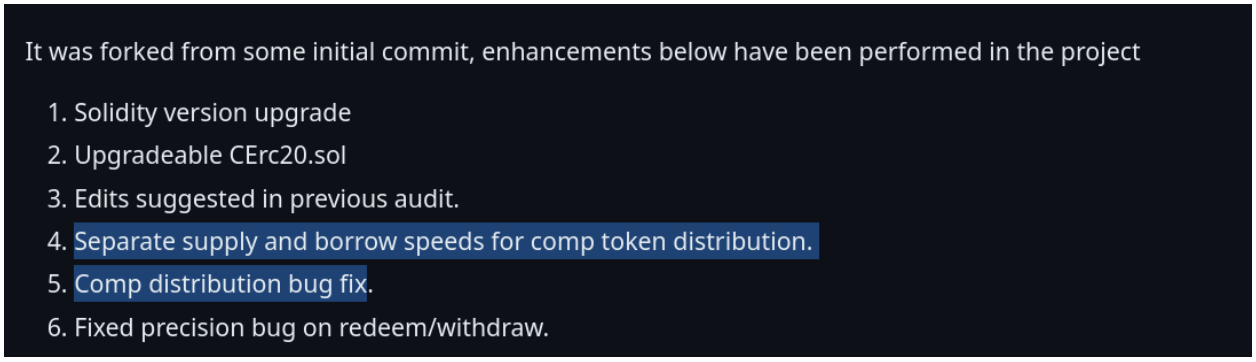
Active Monitoring and Realtime Tracking Scope:

1. **PaxoPriceFeed.sol**
    a. **setPriceFeed (function)**
2. **Unitroller.sol**
    a. **newAdmin(event)**
    b. **new PendingImplementation (event)**
    c. **NewImplementation (event)**
3. **Comptroller.sol**
    a. **NewBorrowCap (event)**
    b. **NewPriceOracle (event)**
    c. **CompBorrowSpeedUpdated (event)**
    d. **CompSupplySpeedUpdated (event)**
4. **DexAggregator.sol**
    a. **setDexRouter**
5. **vaults/pools (all)**
    a. **redeem(event)**
    b. **SetDexAggregator(function)**
    c. **NewReserveFactor(event)**
    d. **NewMarketInterestRateModel (event)**
    e. **Mint (function)**
    f. **ProxyMint (function)**

Initial Report Detailed Findings

[C-01] claimComp() of Comptroller wont work on actual mainnet, freezing user's yields

WeFi has forked CompoundV2, and has made some changes to supply and borrow speeds for COMP token distribution, they have even added a fix for COMP distribution issue.
https://github.com/Uno-Re/unore-WeFi-audit#link-to-documentation



For Compound, COMP is their own token used to incentivize use, In the case of WeFI it represents their own token.
However, none of this would work on the actual mainnet, due to the wrong configuration for the COMP address.

As you can see on the following line, COMP address is not configured to deployed instance and is kept address(0) only.

unore-WeFi-audit/contracts/Comptroller.sol

Line 1368 in f40d147

```
function getCompAddress() public pure returns (address) {
```

Hence whenever someone calls claimComp to claim their accused yield,

claimComp > grantCompInternal > getCompAddress
it would revert inside grantCompInternal, denying users their yield forever.

unore-WeFi-audit/contracts/Comptroller.sol

Lines 1271 to 1280 in f40d147

```
 function grantCompInternal(address user, uint amount) internal returns
(uint) {
     user;
     EIP20Interface comp = EIP20Interface(getCompAddress());
     uint compRemaining = comp.balanceOf(address(this));
     if (amount > 0 && amount <= compRemaining) {
         comp.transfer(user, amount);
         return 0;
     }
     return amount;
 }
```

Recommendation to consider

Consider making comp address a constructor argument OR correct it in getCompAddress

[H-01] accountAssets is not capped properly and may lock user funds

Severity

Impact:
Likelihood:

In ComptrollerStorage.sol, two storage variables are created and documented as follows:

```

/**
     * @notice Max number of assets a single account can participate in
(borrow or use as collateral)
     */
uint public maxAssets;

/**
 * @notice Per-account mapping of "assets you are in", capped by maxAssets
 */
mapping(address => CToken[]) public accountAssets;

```

maxAssets is a variable that is supposed to be used to cap the max number of assets an account can participate at some time.
However, this variable is never set and properly used for the intended purpose.
Currently, the protocol supports a limited number of markets, but as the supported markets count increases, users of the protocol will be subjected to the risk of getting blocked from using important features of the protocol.

The accountAssets mapping holds an array for each account that uses the protocol, and later that array is iterated in various gas-intensive actions such as swapAndSettleCalculateRepayAmount, redeemAllowed, borrowAllowed and exitMarket.
Not capping the number of assets in the accountAssets makes users vulnerable to possible OOG (OUT OF GAS) errors.

When accounts that are active in a high number of markets, use the functions that iterate over the array of accountAssets , their transactions will revert with OOG errors, blocking them from using the most important features of the protocols. Since exitMarket also iterates over the same array, it is possible to end up in a state where the user can not reduce number of the markets they are involved in anymore, effectively bricking the user account without a remediation.

It is important to note here that compounds do have this implemented correctly and WeFI doesn't.
https://github.com/compound-finance/compound-protocol/blob/db1df2b9722fbe4c793576a29fbddbe68205ae92/contracts/ComptrollerG2.sol#L923-L934
https://github.com/compound-finance/compound-protocol/blob/db1df2b9722fbe4c793576a29fbddbe68205ae92/contracts/ComptrollerG2.sol#L150

Even if you consider an onchain instance of the compound, max assets are set as 20 there.
https://etherscan.io/address/0x3d9819210A31b4961b30EF54bE2aeD79B9c9Cd3B#readProxyContract

Remediation:

Consider adding a function to set maxAssets and use it to cap the number of markets and accounts that can be active simultaneously, similar to how compound does.

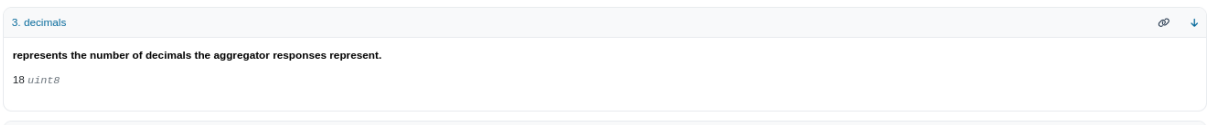
[H-02] Incorrect assumption of USD price feed being 8 decimals always, can lead to loss of funds

WeFI makes an incorrect assumption of chainlink USD price feed always having decimals as 8, inside their PaxosPriceFeed.

unore-WeFi-audit/contracts/oracle/PaxoPriceFeed.sol
Lines 45 to 54 in 9559f17

```
 function getUnderlyingPrice(address pToken) external view returns (uint) {
     int price = getPrice(pToken);
     require(price >= 0, "Price cannot be negative");
     uint256 unsignedPrice = uint256(price);
     // Comptroller needs prices in the format: ${raw price} * 1e36 /
baseUnit
     // The baseUnit of an asset is the amount of the smallest denomination
of that asset per whole.
     // For example, the baseUnit of ETH is 1e18.
     // Since the prices from chainlink for token/usd feed will have 8
decimals, we must scale them by 1e(36 - 8)/baseUnit
     return mul(1e28, unsignedPrice) / tokenConfigs[pToken].baseUnit;
 }
```

There are some existing chainlink price feeds with different decimals than 8.
AMPL/USD with 18 decimals for example.
https://etherscan.io/address/0x492575FDD11a0fCf2C6C719867890a7648d526eB#readContract



It is true that protocol can choose consciously and not include AMPL as a supported asset, however, it is also true that the decimals = 8 constraint may or may not hold true anymore for price feeds, which can have a severe impact due to incorrect evaluation.

Recommendation to consider

Consider fetching decimals from the aggregator only inside setPriceFeed and then match respective calculations, instead to keep them static .

[M-01] Chainlink's latestRoundData() can return stale or incorrect result

Severity

Impact:
Likelihood:

Description

For PaxoPriceFeed.sol.sol, you are using latestRoundData to fetch the current price from chainlink, however, there is no check if the return value indicates stale data.

https://github.com/Uno-Re/unore-WeFi-audit/blob/f40d147efafacd5cc064f54347a04ed2b9aff751/contracts/oracle/PaxoPriceFeed.sol#L59-#L72

In the past lending protocols have suffered in cases where chainlink failed to update price feed either due to market volatility or incorrect configuration.
https://twitter.com/0xngmi/status/1524894553068691456

Recommendation

Consider adding appropriate circuit breakers to handle these cases.
More details: https://0xmacro.com/blog/how-to-consume-chainlink-price-feeds-safely/

[L-01] Vulnerable versions of dependencies are used

As can be seen from the package.json file, the protocol is currently using an older version of OpenZeppelin contracts:

```
  "@openzeppelin/contracts": "^4.8.1",
    "@openzeppelin/contracts-upgradeable": "^4.8.0-rc.1",
```

The problem is that OpenZeppelin are continuously updating their contracts to fix any bugs. You can find more detailed information about all the bugs related to specific versions here. As the latest available version is 4.9.1, consider upgrading the dependencies.

[L-02] Return value for doTransferIn is ignored for DexAggregator

doTransferIn returns a boolean value based on whether the transfer was successful or not.

unore-WeFi-audit/contracts/dex/DexAggregator.sol

Lines 93 to 95 in 9559f17

```
            }
        }
         return result ;
```

However its return value is not checked inside swapExactIn, leading to a false assumption of all token transfers being successful always.

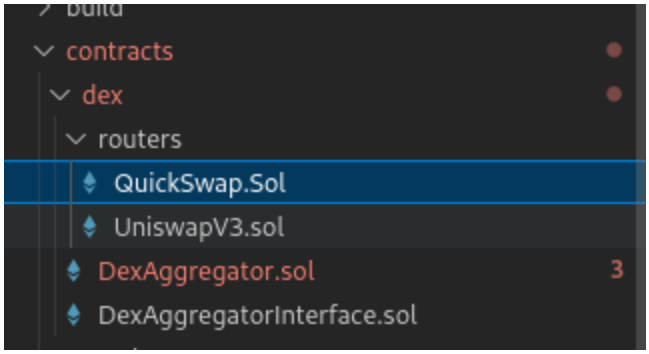
unore-WeFi-audit/contracts/dex/DexAggregator.sol

Lines 36 to 42 in 9559f17

```
```

```
        function swapExactIn(SwapInput calldata params) override external
        returns(uint256 outAmount) {

         (DexData memory dexData) = abi.decode(params.dexData, (DexData));
         if(dexData.dex == DEX_UNI_V3) {
            doTransferIn(msg.sender, params.tokenIn, params.amountIn);
            setMaxAllowanceIfRequired(params.tokenIn,
        params.amountIn,dexRouter[DEX_UNI_V3]);

        outAmount =
        UniswapV3Router(dexRouter[DEX_UNI_V3]).exactInputSingle(UniswapV3Router.
        ExactInputSingleParams(
```

```

The impact is not severe since DexAggregator is not designed to hold any funds, however, consider correcting it.

### Recommendation To Consider

Consider adding a revert if doTransferIn() returns false

## [I-01] DexAggregator doesn't compile due to incorrect file extension for QuickSwapRouter

QuickSwap router is defined as QuickSwap.Sol instead of the correct extension "QuickSwap.sol" (.Sol to .sol)
```



Hence no file importing QuickSwapRouter would be compilable.

Consider correcting the file extension.

[I-02] Multiple SPDX licenses used in the same contract

In Migrations.sol, there are two SPDX license identifiers. However, there should be only a single license identifier per contract.

[I-03] Unsafe pragma statement

Currently, all of the contracts are using a floatable ^0.8.2; compiler version. It is a best practice to always use a newer version as it comes with a lot of new features and bug optimizations. Consider using a stable 0.8.19 version to make sure it is up to date and all of the contracts are compiled with the same compiler version.

[I-04] Unclear error messages

There are a lot of require statements that return an error message "F":

```
   require(markets[address(cToken)].isListed, "F");
```

As this error message is unclear and might be confusing for users, consider replacing it with a more descriptive one. Furthermore, since version 0.8.4 of the compiler, custom errors can be used. Using Solidity Custom Errors has the benefits of less gas spent in reverted transactions, better interoperability of the protocol as clients of it can catch the errors easily on-chain, as well

as you can give descriptive names of the errors without having a bigger bytecode or transaction gas spending, which will result in a better UX as well.

[I-05] Unused import in multiple contracts

```
    import './CTokenUInterface.sol';
```

The above mentioned interface is imported in the following contracts and not used:

CErc20,
Comptroller,
MockPaxoPriceFeed,
PaxoPriceFeed

Consider removing the import from the contracts.

[I-06] Unnecessary NonReentrant modifier

borrowBalanceCurrent() can not be marked as view, it calls accrueInterest() which writes to state.

[I-07] Open TODOs and Typos

There are couple of open TODOs in Comptroller and CToken. This implies changes that might not be audited. Resolve it or remove it.

Also there are a lot of typos in CErc20.sol:

borow -> borrow

prtocol -> protocol

underllying -> underlying , also present in CToken.sol

taget -> target

selleing -> selling , also present in Comptroller.sol

reapy -> repay

[G-01] Use unchecked in for loops

Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation), some gas can be saved by using an unchecked block. This is the case in for loops where unit256 are used. See resource here.

Consider implementing unchecked blocks:

```

```

```
for(uint256 i; i < 10;) {
  // loop logic
  unchecked { i++; }
}
```

This saves 40-50 gas per loop.

[G-02] Cache array length outside of loop

Caching the array length outside a loop saves reading it on each iteration, as long as the array's length is not changed during the loop. The project has many instances of this issue. Cache the array outside the loop.

With this optimization you can potentially save 20 gas units per loop iteration.

The issue is present in Comptroller, DataUtility, DexAggregator. Here is an example instance from Comptroller:

```
function _addMarketInternal(address cToken) internal {
-       for (uint i = 0; i < allMarkets.length; i ++) {
+       uint256 allMarketsLength = allMarkets.length;
+       for (uint i = 0; i < allMarketsLength; i ++) {
            require(allMarkets[i] != CToken(cToken), "F");
```

```
            //market already added
        }
        allMarkets.push(CToken(cToken));
    }
```

Cache all array length outside a loop instances in the above mentioned contracts.

[G-03] Using private rather than public for constants, saves gas

If needed, the values can be read from the verified contract source code, or if there are multiple values there can be a single getter function that returns a tuple of the values of all currently-public constants. Saves 3406-3606 gas in deployment gas due to the compiler not having to create non-payable getter functions for deployment calldata, not having to store the bytes of the value outside of where it's used, and not adding another entry to the method ID table.

Change all instances of public constant to private constant.

[G-04] Use x != 0 instead of x > 0 for uint types

The != operator costs less gas than > and for uint types you can use it to check for non-zero values to save gas.

This change saves 6 gas per instance.

There are 20 instances of this issue. Change them to x != 0, instead of x > 0.

[G-05] Use require instead of assert

The assert() and require() functions are a part of the error handling aspect in Solidity. Solidity makes use of state-reverting error handling exceptions. This means all changes made to the contract on that call or any sub-calls are undone if an error is thrown. It also flags an error.

6 instances of assert in the codebase. Use require instead.

[G-06] Splitting require() statements that use && saves gas

Instead of using the && operator in a single require statement to check multiple conditions,using multiple require statements with 1 condition per require statement will save 3 GAS per &&:

There are two instances of this issue in Comptroller:

```
require(numMarkets != 0 && numMarkets == numBorrowCaps, "F2");
 //invalid input
```
```
require(numTokens == supplySpeeds.length && numTokens ==
borrowSpeeds.length, "F");
```

[G-07] Constructors can be marked payable

Payable functions cost less gas to execute, since the compiler does not have to add extra checks to ensure that a payment wasn't provided. A constructor can safely be marked as payable, since only the deployer would be able to pass funds, and the project itself would not pass any funds.

Making the constructor payable eliminates the need for an initial check of msg.value == 0 and saves 21 gas on deployment with no security risks.

[G-09] Change all require across all contracts to Custom Errors

Custom errors are significantly cheaper than require statements in runtime.

Consider these two functions:

```
error Error();

function test(bool inp)
  external
  view
{
  if (!inp) {
    revert Error();
  }
}

function test2(bool inp)
  external
  view
  returns (uint256 redemptionFee)
{
  require(inp, "Error");
}

```
Gas cost for test(): 440

Gas cost for test2(): 719

Gas Saved: >300 gas can be saved for each reverted calls from contract. In total 80 appearances = 24_000