bunker.finance contest Findings & Analysis Report

2022-07-25

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the bunker.finance smart contract system written in Solidity. The audit contest took place between May 3—May 7 2022.

Wardens

54 Wardens contributed reports to the bunker.finance contest:

1. leastwood
2. hubble (ksk2345 and shri4net)
3. BowTiedWardens (BowTiedHeron, BowTiedPickle, m4rio_eth, Dravee, and BowTiedFirefox)
4. sorrynotsorry
5. IIIIIII
6. 0xDjango
7. GimelSec (rayn and sces60107)
8. oyc_109

9. 0x1f8b
10. throttle
11. robee
12. kebabsec (okkothejawa and FlameHorizon)
13. 0xNazgul
14. ellahi
15. hake
16. Ruhum
17. tintin
18. cccz
19. joestakey
20. Picodes
21. TerrierLover
22. delfin454000
23. samruna
24. 0x4non
25. fatherOfBlocks
26. simon135
27. Funen
28. ilan
29. 0x1337
30. dirk_y
31. hyh
32. bobi
33. David_
34. WatchPug (jtp and ming)
35. cryptphi
36. csanuragjain
37. jayjonah8

38. slywaters
39. 0v3rf10w
40. 0xkatana
41. Cityscape
42. hansfriese
43. rfa
44. Tomio
45. MaratCerby
46. Fitraldys

This contest was judged by gzeon.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 4 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 4 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 30 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 29 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 bunker.finance contest repository, and is composed of 9 smart contracts written in the Solidity programming language and includes 3,214 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

Medium Risk Findings (4)

[M-01] `CNft.sol` - revert inside `safeTransferFrom` will break composability & standard behaviour

Submitted by hubble

CNft.sol#L204

The function safeTransferFrom is a standard interface in ERC1155, and its expected to succeed if all the parametes are valid, and revert on error, which is not the case here so its a deviation.

Refer to the EIP-1155 safeTransferFrom rules:

> MUST revert if _to is the zero address.
> MUST revert if balance of holder for token _id is lower than the _value sent to the recipient.
> MUST revert on any other error.

There is no loss of assets, but the assets or tokens and CNft contract can be unusable by other protocols, and likelihood & impact of this issue is high.

Impact

If other protocols want to integrate CNft, then in that case just for CNft Contract / tokens, they have to take exception and use safeBatchTransferFrom, instead of safeTransferFrom. If they dont take care of this exception, then their protocol functions will fail while using CNft, even if valid values are given.

Proof of Concept

Contract : CNft.sol
Function : safeTransferFrom

> Line 204 revert("CNFT: Use safeBatchTransferFrom instead");

Recommended Mitigation Steps

Instead of revert, call function safeBatchTransferFrom with 1 item in the array, e.g.,

> safeBatchTransferFrom(from, to, [id], [amount], data)

bunkerfinance-dev (bunker.finance) confirmed, but disagreed with High severity and commented:

> We can fix this, but we do not feel like this is high severity at all.

gzeon (judge) decreased severity to Medium and commented:

> I think this is a Med Risk issue as it impacts the function of the protocol.

[M-02] Chainlink pricer is using a deprecated API

Submitted by cccz, also found by 0x1f8b, 0xDjango, 0xNazgul, GimelSec, hake, IIIIIII, kebabsec, oyc_109, Ruhum, sorrynotsorry, throttle, and tintin

According to Chainlink's documentation, the latestAnswer function is deprecated. This function might suddenly stop working if Chainlink stop supporting deprecated APIs. And the old API can return stale data.

Proof of Concept

PriceOracleImplementation.sol#L29-L30

Recommended Mitigation Steps

Use the latestRoundData function to get the price instead. Add checks on the return data with proper revert messages if the price is stale or the round is uncomplete
https://docs.chain.link/docs/price-feeds-api-reference/

bunkerfinance-dev (bunker.finance) confirmed

[M-03] `call()` should be used instead of `transfer()` on an `address payable`

Submitted by BowTiedWardens, also found by leastwood and sorrynotsorry

This is a classic Code4rena issue:

- https://github.com/code-423n4/2021-04-meebits-findings/issues/2
- https://github.com/code-423n4/2021-10-tally-findings/issues/20
- https://github.com/code-423n4/2022-01-openleverage-findings/issues/75

Impact

The use of the deprecated `transfer()` function for an address will inevitably make the transaction fail when:

1. The claimer smart contract does not implement a payable function.
2. The claimer smart contract does implement a payable fallback which uses more than 2300 gas unit.
3. The claimer smart contract implements a payable fallback function that needs less than 2300 gas units but is called through proxy, raising the call's gas usage above 2300.

Additionally, using higher than 2300 gas might be mandatory for some multisig wallets.

Impacted lines

```
CEther.sol:167:         to.transfer(amount);
```

Recommended Mitigation Steps

Use `call()` instead of `transfer()`, but be sure to implement CEI patterns in CEther and add a global state lock on the comptroller as per Rari.

THIS HAS REKT COMPOUND FORKS BEFORE!!!

Relevant links:
https://twitter.com/hacxyk/status/1520715516490379264?s=21&t=fnhDkcC3KpE_kJE8eLiE2A
https://twitter.com/hacxyk/status/1520715536325218304?s=21&t=fnhDkcC3KpE_kJE8eLiE2A
https://twitter.com/hacxyk/status/1520370441705037824?s=21&t=fnhDkcC3KpE_kJE8eLiE2A
https://twitter.com/Hacxyk/status/1521949933380595712

bunkerfinance-dev (bunker.finance) acknowledged and commented:

> We agree that this can make the protocol hard to use if the claimer is a smart contract. This bug needs to be fixed with great care, so we will hold off on fixing this for now.

[M-04] `COMP` Distributions Can Be Manipulated And Duplicated Across Any Number Of Accounts

Submitted by leastwood

Comptroller.sol#L240-L242
Comptroller.sol#L260-L262
Comptroller.sol#L469-L472
Comptroller.sol#L496-L499

[Comptroller.sol#L1139-L1155](#)
[Comptroller.sol#L1222-L1243](#)

The `updateCompSupplyIndex()` and `distributeSupplierComp()` functions are used by Compound to track distributions owed to users for supplying funds to the protocol. Bunker protocol is a fork of compound with NFT integration, however, part of the original functionality appears to have been mistakenly commented out. As a result, whenever users enter or exit the protocol, `COMP` distributions will not be correctly calculated for suppliers. At first glance, its possible that this was intended, however, there is nothing stated in the docs that seems to indicate such. Additionally, the `COMP` distribution functionality has not been commented out for borrowers. Therefore, tokens will still be distributed for borrowers.

Both the `updateCompSupplyIndex()` and `updateCompBorrowIndex()` functions operate on the same `compSpeeds` value which dictates how many tokens are distributed on each block. Therefore, you cannot directly disable the functionality of supplier distributions without altering how distributions are calculated for borrowers. Because of this, suppliers can manipulate their yield by supplying tokens, calling `updateCompSupplyIndex()` and `distributeSupplierComp()`, removing their tokens and repeating the same process on other accounts. This completely breaks all yield distributions and there is currently no way to upgrade the contracts to alter the contract's behaviour. Tokens can be claimed by redepositing in a previously "checkpointed" account, calling `claimComp()` and removing tokens before re-supplying on another account.

Recommended Mitigation Steps

Consider commenting all behaviour associated with token distributions if token distributions are not meant to be supported. Otherwise, it is worthwhile uncommenting all occurrences of the `updateCompSupplyIndex()` and `distributeSupplierComp()` functions.

[bunkerfinance-dev (bunker.finance) acknowledged, but disagreed with High severity and commented](#):

> We are not going to use the COMP code. We could fix documentation or comment more code to make this clearer though.

gzeon (judge) decreased severity to Medium and commented:

> Comptroller.sol is in scope of this contest, and there are no indication that token distribution will be disabled despite the sponsor claim they are not going to use the $COMP code. However, it is also true the deployment setup within contest repo lack the deployment of$COMP and its distribution. I believe this is a valid Med Risk issue given fund(reward token) can be lost in certain assumptions.

Low Risk and Non-Critical Issues

For this contest, 30 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **BowTiedWardens** received the top score from the judge.

The following wardens also submitted reports: lllllll, sorrynotsorry, robee, ellahi, 0xDjango, GimelSec, Picodes, 0x1337, dirk_y, hyh, leastwood, samruna, TerrierLover, 0x4non, bobi, David_, fatherOfBlocks, WatchPug, 0x1f8b, cryptphi, csanuragjain, delfin454000, Funen, ilan, jayjonah8, kebabsec, oyc_109, simon135, *and* throttle.

Table of Contents

[L-01] Add constructor initializers

As per OpenZeppelin's (OZ) recommendation, "The guidelines are now to make it impossible for *anyone* to run `initialize` on an implementation contract, by adding an empty constructor with the `initializer` modifier. So the implementation contract gets initialized automatically upon deployment."

Note that this behaviour is also incorporated the OZ Wizard since the UUPS vulnerability discovery: "Additionally, we modified the code generated by the Wizard 19 to include a constructor that automatically initializes the implementation when deployed."

Furthermore, this thwarts any attempts to frontrun the initialization tx of these contracts:

```
contracts/CErc20.sol:
  25:     function initialize(address underlying_,

contracts/CNft.sol:
  17:     function initialize (
```

[L-02] Missing address(0) checks

Consider adding an `address(0)` check here:

```
- underlying = underlying_ (contracts/CErc20.sol#36)
- pendingAdmin = newPendingAdmin (contracts/CToken.sol#1216)
- admin = admin_ (contracts/CEther.sol#34)
- admin = newAdmin (contracts/Comptroller.sol#733)
- borrowCapGuardian = newBorrowCapGuardian (contracts/Comptrol
- pauseGuardian = newPauseGuardian (contracts/Comptroller.sol#
- admin = _admin (contracts/Oracles/CNftPriceOracle.sol#48)
- uniswapV2Factory = _uniswapV2Factory (contracts/Oracles/CNft
- baseToken = _baseToken (contracts/Oracles/CNftPriceOracle.sc
- admin = newAdmin (contracts/Oracles/CNftPriceOracle.sol#55)
```

[L-03] `Comptroller.sol#allMarkets` : an unbounded loop on array can lead to DoS

`CToken[] public allMarkets;` in `contract ComptrollerV3Storage` is an array where there are just pushes. No upper bound, no pop.

As this array can grow quite large, the transaction's gas cost could exceed the block gas limit and make it impossible to call this function at all here:

```
File: Comptroller.sol
927:     function _addMarketInternal(address cToken) internal {
928:         for (uint i = 0; i < allMarkets.length; i ++) { //
929:             require(allMarkets[i] != CToken(cToken), "marke
930:         }
```

```
931:         allMarkets.push(CToken(cToken));
932:     }
```

Consider introducing a reasonable upper limit based on block gas limits and adding a method to remove elements in the array.

[L-04] `CNft.sol` should implement a 2-step ownership transfer pattern

This contract inherits from OpenZeppelin's library and the `transferOwnership()` function is the default one (a one-step process). It's possible that the `onlyOwner` role mistakenly transfers ownership to a wrong address, resulting in a loss of the `onlyOwner` role (which is quite powerful given the power from `L274:function call`). Consider overriding the default `transferOwnership()` function to first nominate an address as the pending owner and implementing an `acceptOwnership()` function which is called by the pending owner to confirm the transfer.

[N-01] Comment says "public" instead of "external"

```
File: CErc20.sol
131:     /**
132:      * @notice A public function to sweep accidental ERC-20(
133:      * @param token The address of the ERC-20 token to swee
134:      */
135:     function sweepToken(EIP20NonStandardInterface token) ex
136:      require(address(token) != underlying, "CErc20::sweepTo
137:      uint256 balance = token.balanceOf(address(this));
138:      token.transfer(admin, balance);
139:     }
```

[N-02] Prevent accidentally burning tokens

Transferring tokens to the zero address is usually prohibited to accidentally avoid "burning" tokens by sending them to an unrecoverable zero address.

Consider adding a check to prevent accidentally burning tokens here:

```
File: CErc20.sol
207:     function doTransferOut(address payable to, uint amount)
208:         EIP20NonStandardInterface token = EIP20NonStandardI
209:         token.transfer(to, amount); //@audit low: avoid bur
```

[N-03] `require()` should be used for checking error conditions on inputs and return values while `assert()` should be used for invariant checking

Properly functioning code should **never** reach a failing assert statement, unless there is a bug in your contract you should fix. Here, I believe the assert should be a require or a revert:

```
contracts/Comptroller.sol:
  207:         assert(assetIndex < len);
  333:             assert(markets[cToken].accountMembership[borr
```

As the Solidity version is < 0.8.* the remaining gas would not be refunded in case of failure.

[N-04] Avoid floating pragmas: the version should be locked

```
contracts/CErc20.sol:
  1: pragma solidity ^0.5.16;

contracts/CEther.sol:
  1: pragma solidity ^0.5.16;

contracts/CNft.sol:
  2: pragma solidity ^0.8.0;

contracts/Comptroller.sol:
  1: pragma solidity ^0.5.16;

contracts/CToken.sol:
  1: pragma solidity ^0.5.16;

contracts/ERC1155Enumerable.sol:
```

```
  2: pragma solidity ^0.8.0;

contracts/PriceOracleImplementation.sol:
  1: pragma solidity ^0.5.16;

contracts/Oracles/CNftPriceOracle.sol:
  2: pragma solidity ^0.8.0;

contracts/Oracles/UniswapV2PriceOracle.sol:
  2: pragma solidity ^0.8.0;
```

Gas Optimizations

For this contest, 29 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **BowTiedWardens** received the top score from the judge.

The following wardens also submitted reports: joestakey, lllllll, robee, 0xNazgul, delfin454000, ellahi, slywaters, TerrierLover, 0v3rf10w, 0x4non, 0xkatana, Cityscape, fatherOfBlocks, hansfriese, oyc_109, rfa, samruna, simon135, Tomio, MaratCerby, 0x1f8b, 0xDjango, Fitraldys, Funen, GimelSec, ilan, Picodes, *and* throttle.

Table of Contents

See original submission.

[G-01] Copying a full array from storage to memory isn't optimal

Here, what's happening is a full copy of a storage array in memory, and then a second copy of each memory element in a CToken struct:

```
File: Comptroller.sol
590:         CToken[] memory assets = accountAssets[account]; //
591:         for (uint i = 0; i < assets.length; i++) {
592:             CToken asset = assets[i]; //@audit here is a cc
```

The code should be optimized that way:

```
CToken[] storage assets = accountAssets[account]; //@auc
```

This way, the amount of MSTOREs gets divided by 2 and no MLOADs are then necessary

[G-02] Help the optimizer by saving a storage variable's reference instead of repeatedly fetching it

To help the optimizer, declare a `storage` type variable and use it instead of repeatedly fetching the reference in a map or an array.

The effect can be quite significant.

As an example, instead of repeatedly calling `someMap[someIndex]`, save its reference like this: `SomeStruct storage someStruct = someMap[someIndex]` and use it.

Instances include (check the `@audit` tags):

```
contracts/Comptroller.sol:
   268:         if (!markets[cToken].isListed) { //@audit gas: S
   273:         if (cToken != address(nftMarket) && !markets[cTo
   318:         if (!markets[cToken].isListed) { //@audit gas: S
   322:         if (!markets[cToken].accountMembership[borrower]
   333:             assert(markets[cToken].accountMembership[bor
  1067:             if (compSupplyState[address(cToken)].index =
  1068:                 compSupplyState[address(cToken)] = CompM
  1074:             if (compBorrowState[address(cToken)].index =
  1075:                 compBorrowState[address(cToken)] = CompM
  1345:         if (markets[address(cNft)].isListed) { //@audit
  1355:         markets[address(cNft)] = Market({isListed: true,
```

[G-03] Caching storage values in memory

The code can be optimized by minimising the number of SLOADs. SLOADs are expensive (100 gas) compared to MLOADs/MSTOREs (3 gas). Here, storage values should get cached in memory (see the `@audit` tags for further details):

```
contracts/CErc20.sol:
   37:         EIP20Interface(underlying).totalSupply(); //@audi
  172:         EIP20NonStandardInterface token = EIP20NonStandar
  173:         uint balanceBefore = EIP20Interface(underlying).b
  193:         uint balanceAfter = EIP20Interface(underlying).ba

contracts/CNft.sol:
   64:                     (bool checkSuccess, bytes memory resu
   68:                     (bool buyPunkSuccess, ) = underlying.
   73:                     IERC721(underlying).safeTransferFrom(
  147:                     (bool transferPunkSuccess, ) = underl
  152:                     IERC721(underlying).safeTransferFrom(

contracts/Comptroller.sol:
   328:             if (err != Error.NO_ERROR) { //@audit gas: E
   350:         if (err != Error.NO_ERROR) {  //@audit gas: Erro
   362:         return uint(Error.NO_ERROR); //@audit gas: Error
   603:             vars.oraclePriceMantissa = oracle.getUnderly
   630:         uint256 nftBalance = nftMarket.totalBalance(acco
   633:             vars.nftOraclePriceMantissa =  nftOracle.get
   638:             vars.nftCollateralFactor = Exp({mantissa: ma
   641:             vars.sumCollateral = mul_ScalarTruncateAddU1
   642:             if (cAssetModify == address(nftMarket)) {  /
   645:                 vars.sumBorrowPlusEffects = mul_ScalarTr
   667:         uint priceBorrowedMantissa = oracle.getUnderlyin
   668:         uint priceCollateralMantissa = oracle.getUnderly
   702:         require(cNftCollateral == address(nftMarket), "c
   706:         uint priceCollateralMantissa = nftOracle.getUnde
   788:         emit NewCloseFactor(oldCloseFactorMantissa, clos

contracts/CToken.sol:
   976:         if (repayBorrowError != uint(Error.NO_ERROR)) {
   986:         require(amountSeizeError == uint(Error.NO_ERROR)
  1000:         require(seizeError == uint(Error.NO_ERROR), "tok
  1005:         return (uint(Error.NO_ERROR), actualRepayAmount)
  1073:         if (repayBorrowError != uint(Error.NO_ERROR)) {/
  1083:         require(amountSeizeError == uint(Error.NO_ERROR)
  1101:         return (uint(Error.NO_ERROR), actualRepayAmount)

contracts/Oracles/UniswapV2PriceOracle.sol:
```

```
26:             numPairObservations[pair] > 0 && //@audit shc
27:             (block.timestamp - pairObservations[pair][(nu
32:         pairObservations[pair][numPairObservations[pair]+
130:        if (lastObservation.timestamp > block.timestamp -
136:            block.timestamp - lastObservation.timestamp >
142:            return (px0Cumulative - lastObservation.price
151:        if (lastObservation.timestamp > block.timestamp -
157:            block.timestamp - lastObservation.timestamp >
163:            return (px1Cumulative - lastObservation.price
```

[G-04] Unchecking arithmetics operations that can't underflow/overflow

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: https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

I suggest wrapping with an `unchecked` block here (see `@audit` tags for more details):

```
contracts/Oracles/UniswapV2PriceOracle.sol:
128  129:          Observation storage lastObservation = pairObs
131  132:              lastObservation = pairObservations[pair][
149  150:          Observation storage lastObservation = pairObs
152  153:              lastObservation = pairObservations[pair][
```

[G-05] Boolean comparisons

Comparing to a constant (`true` or `false`) is a bit more expensive than directly checking the returned boolean value. I suggest using `if(directValue)` instead of `if(directValue == true)` here (same for require statements):

```
Comptroller.sol:142:          if (marketToJoin.accountMembership[k
Comptroller.sol:997:          require(msg.sender == admin || state
Comptroller.sol:1011:          require(msg.sender == admin || stat
```

```
Comptroller.sol:1020:        require(msg.sender == admin || stat
Comptroller.sol:1029:        require(msg.sender == admin || stat
Comptroller.sol:1065:            require(market.isListed == true
Comptroller.sol:1226:            if (borrowers == true) {
Comptroller.sol:1233:            if (suppliers == true) {
Comptroller.sol:1325:            borrowGuardianPaused[address(cT
```

[G-06] `> 0` is less efficient than `!= 0` for unsigned integers (with proof)

`!= 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer at 10k AND you're in a `require` statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

I suggest changing `> 0` with `!= 0` here:

```
Oracles/CNftPriceOracle.sol:63:            cNfts.length > 0 && c
Oracles/UniswapV2PriceOracle.sol:67:                    reserve(
Oracles/UniswapV2PriceOracle.sol:91:                    reserve(
Oracles/UniswapV2PriceOracle.sol:115:                    reserve
Oracles/UniswapV2PriceOracle.sol:128:        require(length > 0,
Oracles/UniswapV2PriceOracle.sol:149:        require(length > 0,
CToken.sol:37:        require(initialExchangeRateMantissa > 0, '
```

Also, please enable the Optimizer.

[G-07] Splitting `require()` statements that use `&&` saves gas

If you're using the Optimizer at 200, instead of using the `&&` operator in a single require statement to check multiple conditions, I suggest using multiple require statements with 1 condition per require statement:

```
contracts/CNft.sol:
  66:                     require(checkSuccess && nftOwner == ms

contracts/Comptroller.sol:
  947:          require(numMarkets != 0 && numMarkets == numBorrc

contracts/CToken.sol:
  33:          require(accrualBlockNumber == 0 && borrowIndex ==

contracts/Oracles/CNftPriceOracle.sol:
  63:              cNfts.length > 0 && cNfts.length == nftxTokens

contracts/Oracles/UniswapV2PriceOracle.sol:
  67:                     reserve0 > 0 && reserve1 > 0,
```

[G-08] Usage of assert() instead of require()

Between solc 0.4.10 and 0.8.0, `require()` used `REVERT` (0xfd) opcode which refunded remaining gas on failure while `assert()` used `INVALID` (0xfe) opcode which consumed all the supplied gas. (see https://docs.soliditylang.org/en/v0.8.1/control-structures.html#error-handling-assert-require-revert-and-exceptions).
`require()` should be used for checking error conditions on inputs and return values while `assert()` should be used for invariant checking (**properly functioning code should never reach a failing assert statement, unless there is a bug in your contract you should fix**).
From the current usage of `assert`, my guess is that they can be replaced with `require`, unless a `Panic` really is intended.

Here are the `assert` locations:

```
Comptroller.sol:207:         assert(assetIndex < len);
Comptroller.sol:333:             assert(markets[cToken].accountMe
```

[G-09] Amounts should be checked for 0 before calling a transfer

Checking non-zero transfer values can avoid an expensive external call and save gas.

I suggest adding a non-zero-value check here:

```
File: CErc20.sol
135:     function sweepToken(EIP20NonStandardInterface token) ex
136:      require(address(token) != underlying, "CErc20::sweepTo
137:      uint256 balance = token.balanceOf(address(this));
138:      token.transfer(admin, balance); //@audit gas: should c
139:     }
```

[G-10] An array's length should be cached to save gas in for-loops

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
Oracles/CNftPriceOracle.sol:66:         for (uint256 i = 0; i < c
Oracles/UniswapV2PriceOracle.sol:42:         for (uint256 i = 0;
CEther.sol:178:         for (i = 0; i < bytes(message).length; i+
CNft.sol:176:         for (uint256 i; i < vars.length; ++i) {
Comptroller.sol:591:         for (uint i = 0; i < assets.length;
Comptroller.sol:928:         for (uint i = 0; i < allMarkets.lenc
Comptroller.sol:1223:         for (uint i = 0; i < cTokens.length
Comptroller.sol:1229:                 for (uint j = 0; j < holder
Comptroller.sol:1235:                 for (uint j = 0; j < holder
Comptroller.sol:1240:         for (uint j = 0; j < holders.length
ERC1155Enumerable.sol:51:         for (uint256 i; i < ids.length;
```

[G-11] `++i` costs less gas compared to `i++` or `i += 1`

`++i` costs less gas compared to `i++` or `i += 1` for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration). This statement is true even with the optimizer enabled.

The same is also true for `i--`.

`i++` increments `i` and returns the initial value of `i`. Which means:

```
uint i = 1;
i++; // == 1 but i == 2
```

But `++i` returns the actual incremented value:

```
uint i = 1;
++i; // == 2 and i == 2 too, so no need for a temporary variable
```

In the first case, the compiler has to create a temporary variable (when used) for returning `1` instead of `2`

Instances include:

```
CEther.sol:178:         for (i = 0; i < bytes(message).length; i+
Comptroller.sol:119:        for (uint i = 0; i < len; i++) {
Comptroller.sol:199:        for (uint i = 0; i < len; i++) {
Comptroller.sol:212:        storedList.length--; //@audit use --
Comptroller.sol:591:        for (uint i = 0; i < assets.length;
Comptroller.sol:928:        for (uint i = 0; i < allMarkets.lenc
Comptroller.sol:949:        for(uint i = 0; i < numMarkets; i++)
Comptroller.sol:1223:        for (uint i = 0; i < cTokens.length
Comptroller.sol:1229:                for (uint j = 0; j < holder
Comptroller.sol:1235:                for (uint j = 0; j < holder
Comptroller.sol:1240:        for (uint j = 0; j < holders.length
```

I suggest using `++i` instead of `i++` to increment the value of an uint variable.

[G-12] Do not pre-declare variable with default values

One of the practices in the project is to pre-declare variables before assigning a value to them. This is not necessary and actually costs some gas (MSTOREs and MLOADs).

As an example, consider going from:

```
File: Comptroller.sol
715:         uint seizeTokens;
716:         Exp memory numerator;
717:         Exp memory denominator;
718:         Exp memory ratio;
719:
720:         numerator = mul_(Exp({mantissa: liquidationIncentiv
721:         denominator = Exp({mantissa: priceCollateralMantiss
722:         ratio = div_(numerator, denominator);
723:
724:         seizeTokens = truncate(mul_(ratio, Exp({mantissa: a
```

to:

```
        Exp memory numerator = mul_(Exp({mantissa: liquidationIn
        Exp memory denominator = Exp({mantissa: priceCollateralM
        Exp memory ratio = div_(numerator, denominator);

        uint seizeTokens = truncate(mul_(ratio, Exp({mantissa: a
```

Same for the following code:

```
File: Comptroller.sol
680:         uint seizeTokens;
681:         Exp memory numerator;
682:         Exp memory denominator;
683:         Exp memory ratio;
684:
685:         numerator = mul_(Exp({mantissa: liquidationIncentiv
686:         denominator = mul_(Exp({mantissa: priceCollateralMa
687:         ratio = div_(numerator, denominator);
688:
689:         seizeTokens = mul_ScalarTruncate(ratio, actualRepay
```

[G-13] Increments can be unchecked

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

Instances include:

```
Oracles/CNftPriceOracle.sol:66:        for (uint256 i = 0; i < c
Oracles/UniswapV2PriceOracle.sol:42:        for (uint256 i = 0;
CNft.sol:50:        for (uint256 i; i < length; ++i) {
CNft.sol:62:                for (uint256 i; i < length; ++i) {
CNft.sol:72:                for (uint256 i; i < length; ++i) {
CNft.sol:98:        for (uint256 i; i < length; ++i) {
CNft.sol:122:        for (uint256 i; i < length; ++i) {
CNft.sol:145:                for (uint256 i; i < length; ++i) {
CNft.sol:151:                for (uint256 i; i < length; ++i) {
CNft.sol:176:        for (uint256 i; i < vars.length; ++i) {
ERC1155Enumerable.sol:51:        for (uint256 i; i < ids.length;
```

The code would go from:

```
for (uint256 i; i < numIterations; i++) {
 // ...
}
```

to:

```
for (uint256 i; i < numIterations;) {
 // ...
 unchecked { ++i; }
}
```

The risk of overflow is inexistant for `uint256` here.

[G-14] Public functions to external

The following functions could be set external to save gas and improve code quality. External call cost is less expensive than of public functions.

```
_setInterestRateModel(InterestRateModel) should be declared exte
        - CToken._setInterestRateModel(InterestRateModel) (contr

enterMarkets(address[]) should be declared external:
        - Comptroller.enterMarkets(address[]) (contracts/Comptro
getAccountLiquidity(address) should be declared external:
        - Comptroller.getAccountLiquidity(address) (contracts/Co
getHypotheticalAccountLiquidity(address,address,uint256,uint256)
        - Comptroller.getHypotheticalAccountLiquidity(address,ad
_setPriceOracle(PriceOracle) should be declared external:
        - Comptroller._setPriceOracle(PriceOracle) (contracts/Co
_setNftPriceOracle(NftPriceOracle) should be declared external:
        - Comptroller._setNftPriceOracle(NftPriceOracle) (contra
_setPauseGuardian(address) should be declared external:
        - Comptroller._setPauseGuardian(address) (contracts/Comp
_setMintPaused(address,bool) should be declared external:
        - Comptroller._setMintPaused(address,bool) (contracts/Co
_setBorrowPaused(CToken,bool) should be declared external:
        - Comptroller._setBorrowPaused(CToken,bool) (contracts/C
_setTransferPaused(bool) should be declared external:
        - Comptroller._setTransferPaused(bool) (contracts/Comptr
_setSeizePaused(bool) should be declared external:
        - Comptroller._setSeizePaused(bool) (contracts/Comptroll
_become(Unitroller) should be declared external:
        - Comptroller._become(Unitroller) (contracts/Comptroller
claimComp(address) should be declared external:
        - Comptroller.claimComp(address) (contracts/Comptroller.
_grantComp(address,uint256) should be declared external:
        - Comptroller._grantComp(address,uint256) (contracts/Com
_setCompSpeed(CToken,uint256) should be declared external:
        - Comptroller._setCompSpeed(CToken,uint256) (contracts/C
_setContributorCompSpeed(address,uint256) should be declared ext
        - Comptroller._setContributorCompSpeed(address,uint256)
getAllMarkets() should be declared external:
        - Comptroller.getAllMarkets() (contracts/Comptroller.sol
```

[G-15] No need to explicitly initialize variables with default values

If a variable is not set/initialized, it is assumed to have the default value (`0` for `uint` , `false` for `bool` , `address(0)` for address...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

As an example: `for (uint256 i = 0; i < numIterations; ++i) {` should be replaced with `for (uint256 i; i < numIterations; ++i) {`

Instances include:

```
Oracles/CNftPriceOracle.sol:66:        for (uint256 i = 0; i < c
Oracles/UniswapV2PriceOracle.sol:41:        uint256 numberUpdate
Oracles/UniswapV2PriceOracle.sol:42:        for (uint256 i = 0;
CEther.sol:178:        for (i = 0; i < bytes(message).length; i+
CNft.sol:49:        uint256 totalAmount = 0;
CNft.sol:97:        uint256 totalAmount = 0;
CNft.sol:119:        uint256 totalAmount = 0;
Comptroller.sol:119:        for (uint i = 0; i < len; i++) {
Comptroller.sol:199:        for (uint i = 0; i < len; i++) {
Comptroller.sol:591:        for (uint i = 0; i < assets.length;
Comptroller.sol:928:        for (uint i = 0; i < allMarkets.lenc
Comptroller.sol:949:        for(uint i = 0; i < numMarkets; i++)
Comptroller.sol:1223:        for (uint i = 0; i < cTokens.length
Comptroller.sol:1229:                for (uint j = 0; j < holder
Comptroller.sol:1235:                for (uint j = 0; j < holder
Comptroller.sol:1240:        for (uint j = 0; j < holders.length
CToken.sol:81:        uint startingAllowance = 0;
```

I suggest removing explicit initializations for default values.

[G-16] Upgrade pragma to at least 0.8.4

Using newer compiler versions and the optimizer give gas optimizations. Also, additional safety checks are available for free.

The advantages here are:

- **Low level inliner** (>= 0.8.2): Cheaper runtime gas (especially relevant when the contract has small functions).
- **Optimizer improvements in packed structs** (>= 0.8.3)

- **Custom errors** (>= 0.8.4): cheaper deployment cost and runtime cost. *Note*: the runtime cost is only relevant when the revert condition is met. In short, replace revert strings by custom errors.

Consider upgrading pragma to at least 0.8.4:

```
Oracles/CNftPriceOracle.sol:2:pragma solidity ^0.8.0;
Oracles/UniswapV2PriceOracle.sol:2:pragma solidity ^0.8.0;
CNft.sol:2:pragma solidity ^0.8.0;
ERC1155Enumerable.sol:2:pragma solidity ^0.8.0;
```

[G-17] `PriceOracleImplementation.cEtherAddress` variable should be immutable

This variable is only set in the constructor and is never edited after that:

```
File: PriceOracleImplementation.sol
10:     address public cEtherAddress; //@audit gas: should be im
```

Consider marking it as immutable.

[G-18] Reduce the size of error messages (Long revert Strings)

Shortening revert strings to fit in 32 bytes will decrease deployment time gas and will decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead for computing memory offset, etc.

Revert strings > 32 bytes:

```
Oracles/CNftPriceOracle.sol:64:            "CNftPriceOracle: `cN
Oracles/CNftPriceOracle.sol:70:                "CNftPriceOracle:
Oracles/CNftPriceOracle.sol:90:            "CNftPriceOracle: No
Oracles/UniswapV2PriceOracle.sol:68:                    "Uniswap
Oracles/UniswapV2PriceOracle.sol:92:                    "Uniswap
```

```
Oracles/UniswapV2PriceOracle.sol:116:                    "Uniswa
Oracles/UniswapV2PriceOracle.sol:128:        require(length > 0,
Oracles/UniswapV2PriceOracle.sol:131:            require(length
Oracles/UniswapV2PriceOracle.sol:137:            "UniswapV2Price
Oracles/UniswapV2PriceOracle.sol:149:        require(length > 0,
Oracles/UniswapV2PriceOracle.sol:152:            require(length
Oracles/UniswapV2PriceOracle.sol:158:            "UniswapV2Price
CErc20.sol:136:     require(address(token) != underlying, "CErc2
CErc20.sol:234:        require(msg.sender == admin, "only the ac
CNft.sol:24:        require(_underlying != address(0), "CNFT: As
CNft.sol:25:        require(ComptrollerInterface(_comptroller).i
CNft.sol:52:                require(amounts[i] == 1, "CNFT: Amou
CNft.sol:69:                    require(buyPunkSuccess, "CNFT: C
CNft.sol:93:        require(borrower != liquidator, "CNFT: Liqui
CNft.sol<img class="emoji-icon" alt="emoji-100" data-icon="emoji
CNft.sol:124:                require(amounts[i] == 1, "CNFT: Amo
CNft.sol:148:                    require(transferPunkSuccess, "C
CNft.sol:204:            revert("CNFT: Use safeBatchTransferFrom
CNft.sol:208:        require(msg.sender == underlying, "CNFT: Th
CNft.sol:209:        require(operator == address(this), "CNFT: C
CNft.sol:279:        require(to != underlying, "CNFT: Cannot mak
Comptroller.sol:171:            require(oErr == 0, "exitMarket: getA
Comptroller.sol:420:                require(borrowBalance >= repayAm
Comptroller.sol:702:            require(cNftCollateral == address(nf
Comptroller.sol:942:        require(msg.sender == admin || msg.senc
Comptroller.sol:960:            require(msg.sender == admin, "only a
Comptroller.sol:995:            require(markets[cAsset].isListed, "C
Comptroller.sol:996:            require(msg.sender == pauseGuardian
Comptroller.sol:1009:           require(markets[address(cToken)].is
Comptroller.sol:1010:           require(msg.sender == pauseGuardian
Comptroller.sol:1019:           require(msg.sender == pauseGuardian
Comptroller.sol:1028:           require(msg.sender == pauseGuardian
Comptroller.sol:1037:           require(msg.sender == unitroller.ac
Comptroller.sol:1338:           require(address(nftMarket) == addre
Comptroller.sol:1339:           require(address(cNft) != address(0)
CToken.sol:32:          require(msg.sender == admin, "only admin m
CToken.sol:33:          require(accrualBlockNumber == 0 && borrowI
CToken.sol:37:          require(initialExchangeRateMantissa > 0, "
CToken.sol:49:          require(err == uint(Error.NO_ERROR), "sett
CToken.sol:271:          require(err == MathError.NO_ERROR, "borrC
CToken.sol:328:          require(err == MathError.NO_ERROR, "excha
CToken.sol:542:          require(vars.mathErr == MathError.NO_ERRO
CToken.sol:545:          require(vars.mathErr == MathError.NO_ERRO
CToken.sol:609:          require(redeemTokensIn == 0 || redeemAmou
CToken.sol:891:          require(vars.mathErr == MathError.NO_ERRO
CToken.sol:894:          require(vars.mathErr == MathError.NO_ERRO
```

```
CToken.sol:986:        require(amountSeizeError == uint(Error.NO
CToken.sol:1083:        require(amountSeizeError == uint(Error.N
CToken.sol:1093:        require(seizeTokens == 0, "LIQUIDATE_SEI
CToken.sol:1433:        require(totalReservesNew <= totalReserve
```

I suggest shortening the revert strings to fit in 32 bytes, or using custom errors as described next.

[G-19] Use Custom Errors instead of Revert Strings to save Gas

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met)

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

> Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

Instances include:

```
Oracles/CNftPriceOracle.sol:31:        require(msg.sender == adm
Oracles/CNftPriceOracle.sol:62:        require(
Oracles/CNftPriceOracle.sol:68:            require(
Oracles/CNftPriceOracle.sol:88:        require(
Oracles/UniswapV2PriceOracle.sol:66:                require(
Oracles/UniswapV2PriceOracle.sol:90:                require(
Oracles/UniswapV2PriceOracle.sol:114:                 require(
Oracles/UniswapV2PriceOracle.sol:128:        require(length > 0,
Oracles/UniswapV2PriceOracle.sol:131:            require(length
Oracles/UniswapV2PriceOracle.sol:135:        require(
Oracles/UniswapV2PriceOracle.sol:149:        require(length > 0,
```

```
Oracles/UniswapV2PriceOracle.sol:152:            require(length
Oracles/UniswapV2PriceOracle.sol:156:        require(
CNft.sol:24:        require(_underlying != address(0), "CNFT: As
CNft.sol:25:        require(ComptrollerInterface(_comptroller).i
CNft.sol:40:        require(tokenIds.length == amounts.length, '
CNft.sol:45:        require(mintAllowedResult == 0, "CNFT: Mint
CNft.sol:52:                require(amounts[i] == 1, "CNFT: Amou
CNft.sol:66:                    require(checkSuccess && nftOwner
CNft.sol:69:                    require(buyPunkSuccess, "CNFT: C
CNft.sol:85:        require(seizeIds.length == seizeAmounts.leng
CNft.sol:90:        require(siezeAllowedResult == 0, "CNFT: Seiz
CNft.sol:93:        require(borrower != liquidator, "CNFT: Liqui
CNft.sol<img class="emoji-icon" alt="emoji-100" data-icon="emoji
CNft.sol:116:        require(tokenIds.length == amounts.length,
CNft.sol:124:                require(amounts[i] == 1, "CNFT: Amo
CNft.sol:127:            require(balanceOf(msg.sender, tokenIds[
CNft.sol:132:        require(redeemAllowedResult == 0, "CNFT: Re
CNft.sol:148:                    require(transferPunkSuccess, "C
CNft.sol:182:        require(transferAllowedResult == 0, "CNFT:
CNft.sol:208:        require(msg.sender == underlying, "CNFT: Th
CNft.sol:209:        require(operator == address(this), "CNFT: C
CNft.sol:279:        require(to != underlying, "CNFT: Cannot mak
```

I suggest replacing revert strings with custom errors.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top