Wild Credit
Findings & Analysis Report

2021-08-17

Table of contents

Overview

About C4

Code 432n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code 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 code contest outlined in this document, C4 conducted an analysis of the Wild Credit smart contract system written in Solidity. The code contest took place between July 7—July 14 2021.

Wardens

12 Wardens contributed reports to the Wild Credit code contest:

- cmichel
- greiart
- paulius.eth
- 0xRajeev
- shw
- Jmukesh
- gpersoon
- defsec
- toastedsteaksandwich
- a_dlamo
- jonah1005
- hrkrshnn
- 0xsanson

This contest was judged by ghoul.sol.

Final report assembled by moneylegobatman and ninek.

Summary

The C4 analysis yielded an aggregated total of 19 unique vulnerabilities. All of the issues presented here are linked back to their original finding

Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity, 4 received a risk rating in the category of MEDIUM severity, and 12 received a risk rating in the category of LOW severity.

C4 analysis also identified 40 non-critical recommendations.

Scope

The code under review can be found within the C4 Wild Credit code contest repository and is comprised of 29 smart contracts written in the Solidity programming language.

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.

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.

High Risk Findings

[H-01] Reward computation is wrong

Submitted by cmichel

The `LendingPair.accrueAccount` function distributes rewards **before** updating the cumulative supply / borrow indexes as well as the index + balance for the user (by minting supply tokens / debt). This means the percentage of the user's balance to the total is not correct as the total can be updated several times in between.

```
function accrueAccount(address _account) public {
  // distributes before updating accrual state
  _distributeReward(_account);
  accrue();
  _accrueAccountInterest(_account);

  if (_account != feeRecipient()) {
    _accrueAccountInterest(feeRecipient());
  }
}
```

Example: Two users deposit the same amounts in the same block. Thus, after some time they should receive the same tokens.

1. User A and B deposit 1000 tokens (in the same block) and are minted 1000 tokens in return. Total supply = `2000`
2. Assume after 50,000 blocks, `A` calls `accrueAccount(A)` which first calls `_distributeReward`. A is paid out 1000/2000 = 50% of the 50,000 blocks reward since deposit. Afterwards, `accrue` + `_accrueAccountInterest(A)` is called and `A` is minted 200 more tokens due to supplier lending rate. The supply **totalSupply is now 2200**.
3. After another 50,000 blocks, `A` calls `accrueAccount(A)` again. which first calls `_distributeReward`. A is paid out 1200/2200 = **54.5454% of the 50,000 blocks reward since deposit.**

From here, you can already see that `A` receives more than 50% of the 100,000 block rewards although they deposited at the same time as `B` and didn't deposit or withdraw any funds. `B` will receive `~1000/2200 = 45%` (ignoring any new LP supply tokens minted for `A`'s second claim.)

The impact is that wrong rewards will be minted users which do not represent their real fair share. Usually, users will get fewer rewards than they should receive, as their individual interest was not updated yet, but the totals (total debt and total supply) could have been updated by other accounts in between.

There are two issues that both contribute to it:

- total LP supply and total debt must be updated by the **total new interest** when `accrue` is called, not only increased by an **individual user**'s interest. See my other issue "Reward computation is wrong" that goes into more depth
- Lending/borrow accrual must happen before reward distribution

[talegift (Wild Credit) acknowledged but disagreed with severity]:

> *Assets not at direct risk, but the function of the protocol or its availability could be impacted, or* **leak value** *with a hypothetical attack path with stated assumptions, but* **external requirements.**

> Update to severity - 2

[ghoul-sol (Judge) commented]:

> Disagree with sponsor about severity, this is significant accounting error.

[H-02] `LendingPair.liquidateAccount` does not accrue and update `cumulativeInterestRate`

Submitted by cmichel

The `LendingPair.liquidateAccount` function does not accrue and update the `cumulativeInterestRate` first, it only calls `_accrueAccountInterest` which does not update and instead uses the old `cumulativeInterestRate`.

The liquidatee (borrower)'s state will not be up-to-date. I could skip some interest payments by liquidating myself instead of repaying if I'm under-water. As the market interest index is not accrued, the borrower does not need to pay any interest accrued from the time of the last accrual until now.

Recommend calling `accrueAccount` instead of `_accrueAccountInterest`

[talegift (Wild Credit) confirmed but disagreed with severity]:

> *Assets not at direct risk, but the function of the protocol or its availability could be impacted, or* **leak value** *with a hypothetical attack path with stated assumptions, but* **external requirements.**

> Update to severity - 2

ghoul-sol (Judge) commented:

> No funds are lost however a user can steal “unpaid interest” from the protocol. Keeping high risk.

[H-03] `LendingPair.liquidateAccount` fails if tokens are lent out

Submitted by cmichel

The `LendingPair.liquidateAccount` function tries to pay out underlying supply tokens to the liquidator using `_safeTransfer(IERC20(supplyToken), msg.sender, supplyOutput)` but there's no reason why there should be enough `supplyOutput` amount in the contract, the contract only ensures `minReserve`.

As a result, no liquidations can be performed if all tokens are lent out. **Example:** User A supplies 1k$ WETH, User B supplies 1.5k$ DAI and borrows the ~1k$ WETH (only leaves `minReserve`). The ETH price drops but user B cannot be liquidated as there's not enough WETH in the pool anymore to pay out the liquidator.

Recommend minting LP supply tokens to `msg.sender` instead, these are the LP supply tokens that were burnt from the borrower. This way the liquidator basically seizes the borrower's LP tokens.

talegift (Wild Credit) confirmed but disagreed with severity:

> *Assets not at direct risk, but the function of the protocol or its availability could be impacted, or* **leak value** *with a hypothetical attack path with stated assumptions, but* **external requirements**.

> Update to severity - 2

ghoul-sol (Judge) commented:

> If liquidation is impossible, there's insolvency risk and that creates a risk to lose user funds. Keeping high severity.

Medium Risk Findings

[M-01] Chainlink - Use `latestRoundData` instead of `latestAnswer` to run more validations

*Submitted by a*delamo, also found by 0xRajeev, cmichel, greiart, and shw_

`UniswapV3Oracle.sol` is calling `latestAnswer` to get the last WETH price. This method will return the last value, but you won't be able to check if the data is fresh. On the other hand, calling the method `latestRoundData` allow you to run some extra validations

```
        (
          roundId,
          rawPrice,
          ,
          updateTime,
          answeredInRound
        ) = AggregatorV3Interface(XXXXX).latestRoundData();
        require(rawPrice > 0, "Chainlink price <= 0");
        require(updateTime != 0, "Incomplete round");
        require(answeredInRound >= roundId, "Stale price");
```

See the chainlink documentation for more information.

talegift (Wild Credit) confirmed

ghoul-sol (Judge) commented:

> Since slate prices could have quite serious consequences, I'll bump it to medium risk.

[M-02] `safeTransferFrom` in `TransferHelper` is not `safeTransferFrom`

Submitted by jonah1005, also found by 0xRajeev, shw, JMukesh, and cmichel

A non standard ERC20 token would always raise error when calling `_safeTransferFrom`. If a user creates a USDT/DAI pool and deposit into the pool he would find out there's never a counterpart deposit. See `TransferHelper.sol #L19`.

`TransferHelper` does not uses `SafeERC20` library as the function name implies.

A sample POC:

```
usdt.functions.approve(lending_pair.address, deposit_amount).tra
lending_pair.functions.deposit(w3.eth.accounts[0], usdt.address,
```

Error Message:

```
Error: Transaction reverted: function returned an unexpected a
    at LendingPair._safeTransferFrom (contracts/TransferHelper
    at LendingPair.deposit (contracts/LendingPair.sol:95)
```

Recommend using `openzeppelin SafeERC20` in `transferHelper` (and any other contract that uses IERC20).

talegift (Wild Credit) confirmed

ghoul-sol (Judge) commented:

> This can effect deposits so it's a medium risk.

[M-03] `_wethWithdrawTo` is vulnerable re-entrancy

Submitted by pauliax

The function `withdrawBorrowETH` invokes `_wethWithdrawTo` and later `_checkMinReserve`, however, the check of reserve is not necessary here, as function `_wethWithdrawTo` also does that after transferring the ether. However, this reserve check might be bypassed as `TransferHelper._wethWithdrawTo` uses a

low level call that is vulnerable to re-entrancy attacks. As this `MIN_RESERVE` sounds like an important value, you should consider preventing re-entrancy attacks here.

```
    // Prevents division by zero and other undesirable behavior
    uint public constant MIN_RESERVE = 1000;
```

Recommend considering using [re-entrancy guard] on all main action functions (e.g. `deposit`, `withdraw`, `borrow`, `repay`, etc...):

talegift (Wild Credit) confirmed

[M-04] Total LP supply & total debt accrual is wrong

Submitted by cmichel

The total debt and total supply only increase when debt/supply is minted to the user when it should increase by the entire new interest amount on each accrual.

```
  function accrueAccount(address _account) public {
    _distributeReward(_account);
    // accrue only updates cumulativeInterestRate to the newIntere
    // newInterest is not added to total debt or total LP supply!
    accrue();
    // instead total debt / total LP supply is increased here by a
    _accrueAccountInterest(_account);

    if (_account != feeRecipient()) {
      _accrueAccountInterest(feeRecipient());
    }
  }
```

The borrow rates (see `borrowRatePerBlock`) are wrong due to the wrong utilization ratio: The borrow utilization rate uses `LPToken.totalSupply`. Assume there's a single lender supplying $100k, another single borrower borrows $70k (ignoring irrelevant details like liquidation and the borrower not putting up collateral for the sake of the argument). After some time debt accrued and the supplier "updates" by calling `accrue` (but the borrower does not update), this increases the LP total supply to, say, $110k, while total debt is not updated. The utilization rate and thus the

borrow rate is now less than before (from 70/100=70% to 70/110=63%). In reality, it should have increased as the supplier interest is only a fraction of the borrower accumulated debt. From now on less debt than expected accrues until the borrower is updated and total debt is increased. To get the correct borrow rates in the current system, every borrower and every supplier would need to be updated on every accrual which is infeasible.

Recommend doing it like Compound/Aave, increase total debt and total supply on each accrual by the **total** new interest (not by the specific user's interest only). This might require a bigger refactor because the LP token is treated as a ("lazy-evaluated") rebasing token and the total supply is indeed the total tokens in circulation `LP.totalSupply()` and one cannot mint the new interest to all users at once in each `accrue`. That's why Compound uses an interest-bearing token and tracks total supply separately and Aave uses a real rebasing token that dynamically scales the balance in `balanceOf` and must not be updated individually for each user.

talegift (Wild Credit) confirmed but disagreed with severity:

> The issue seems to only impact interest rate calculation. It doesn't allow anyone to steal the funds.

> Severity should be set to 1.

ghoul-sol (Judge) commented:

> I'm making this medium risk as no funds are lost but the accounting is basically incorrect.

Low Risk Findings

[L-01] No check of `MAX_LIQ_FEES` in contructor of `Controller`

Submitted by gpersoon, also found by 0xRajeev

Both the functions `setLiqParamsToken` and `setLiqParamsDefault` have a check to make sure that `_liqFeeCaller + _liqFeeSystem <= MAX_LIQ_FEES`

However the constructor of `Controller` sets the same parameters and doesn't have this check. It seems logical to also do the check in the `controller`, otherwise the parameters could be set outside of the wanted range.

`Controller.sol` **#L49**

```
    constructor( address _interestRateModel, uint _liqFeeSystemDefau
        ...
        liqFeeSystemDefault = _liqFeeSystemDefault;
        liqFeeCallerDefault = _liqFeeCallerDefault;

    function setLiqParamsToken( address _token, uint    _liqFeeSyste
        require(_liqFeeCaller + _liqFeeSystem <= `MAX_LIQ_FEES`, "Cc
    ...
        liqFeeSystemToken[_token] = _liqFeeSystem;
        liqFeeCallerToken[_token] = _liqFeeCaller;

    function setLiqParamsDefault( uint    _liqFeeSystem, uint    _li
        require(_liqFeeCaller + _liqFeeSystem <= MAX_LIQ_FEES, "Cont
        liqFeeSystemDefault = _liqFeeSystem;
        liqFeeCallerDefault = _liqFeeCaller;
```

Recommend adding something like the following in the constructor of `Controller`

```
    require(liqFeeCallerDefault + liqFeeSystemDefault <= MAX_LIQ_FEE
```

- **talegift (Wild Credit) confirmed**

[L-02] Uniswap oracle assumes `PairToken <> WETH` liquidity

Submitted by cmichel

The `UniswapV3Oracle.tokenPrice` function gets the price by combining the chainlink ETH price with the TWAP prices of the `token <> pairToken` and

`pairToken <> WETH` pools. It is therefore required that the `pairToken <> WETH` pool exists and has sufficient liquidity to be tamper-proof.

When listing lending pairs for tokens that have a WETH pair with low liquidity (at 0.3% fees) the prices can be easily manipulated leading to liquidations or underpriced borrows. This can happen for tokens that don't use `WETH` as their default trading pair, for example, if they prefer a stablecoin, or `WBTC`.

Recommend ensuring there's enough liquidity on the `pairToken <> WETH` Uniswap V3 0.3% pair, either manually or programmatically.

talegift (Wild Credit) acknowledged:

> This is a known & documented risk: **https://wild-credit.gitbook.io/wild-credit/advanced-concepts/price-oracles**

[L-03] Simple interest formula is used

Submitted by cmichel

The `_accrueInterest` function uses a simple interest formula to compute the accrued debt, instead of a compounding formula. This means the actual borrow rate and interest for suppliers depend on how often updates are made. This difference should be negligible in highly active markets, but it could lead to a lower borrow rate in low-activity markets.

Recommend ensuring that the lending pairs is accrued regularly, or switching to a compound interest formula (which has a higher gas cost due to exponentiation, but can be approximated, see Aave).

- talegift (Wild Credit) acknowledged

[L-04] `LendingPair.pendingSupplyInterest` is not accurate

Submitted by cmichel

The `LendingPair.pendingSupplyInterest` does not accrue the new interest since the last update and therefor the returned value is not accurate.

Recommend accruing it first such that `cumulativeInterestRate` updates and `_newInterest` returns the updated value.

- talegift (Wild Credit) confirmed

[L-05] Migrate Rewards Without Distribution

Submitted by defsec

With the `migrateRewards` function, owner can transfer all reward token into other address. Owner should distribute reward balance before migrating rewards.

1. Navigate to "https://github.com/code-423n4/2021-07-wildcredit/blob/82c48d73fd27a9d4d5d4a395b3affcef4ef6c5c8/contracts/RewardDistribution.sol"
2. See functionality on the " `migrateRewards` " function.
3. According to function, owner can transfer all reward balance to another address.

Recommend that owner should distribute reward balance before migrating rewards.

talegift (Wild Credit) acknowledged:

> This is technically not possible to distribute all pending rewards to all users due to the block gas limit and the fact that rewards accrue with each new block.

> If the rewards are ever migrated (and it's not urgent), we would likely notify users upfront and give them some time to claim & unstake.

[L-06] `minBorrowUSD` not initialized in the contract

Submitted by gpersoon

The parameter `minBorrowUSD` of the contract `Controller` isn't initialized. If someone is able to Borrow before the function `setMinBorrowUSD` is called, he might be able to borrow a very small amount. This might be unwanted.

`Controller.sol` **#L27**

```
    uint public minBorrowUSD;

    function setMinBorrowUSD(uint _value) external onlyOwner {
      minBorrowUSD = _value;
    }
```

`LendingPair.sol` **#L553**

```
    function _checkBorrowLimits(address _token, address _account) ir
      ...
      require(accountBorrowUSD >= controller.minBorrowUSD(), "Lendir
```

Recommend Initializing `minBorrowUSD` via the constructor or set a reasonable default in the contract.

- talegift (Wild Credit) confirmed

[L-07] `LendingPair` : Avoid rounding error in `_accrueAccountSupply()`

Submitted by greiart

`supplyInterest` is split between the account and the system, ie. `supplyInterest = newSupplyAccount + newSupplySystem`. Hence, an additional call to the `_systemRate` can be avoided in the calculation of `newSupplySystem`, as well as potential rounding errors. See L401-L403 in `LendingPair.sol`.

Recommend that L403 be changed to `newSupplySystem = supplyInterest - newSupplyAccount;`

- talegift (Wild Credit) acknowledged

[L-08] `UniswapV3Oracle` : Reduce `minObservations` to uint16

Submitted by greiart

Reducing `minObservations` to uint16 will help prevent erroneous `minObservations` values from being set (ie. `> 65535`) by the owner without needing checks. Otherwise, the `isPoolValid` will always return false, causing reverts in calling `tokenPrice` and `addPool` functions (and other functions calling these). See L25 and L101 of `UniswapV3Oracle.sol`.

The maximum number of observations available is 65535 (see `UniswapV3Pool.sol` L39), which is equivalent to `type(uint16).max`.

Hence,

- `uint public minObservations` can be reduced to `uint16 public minObservations`
- `(, , , , uint observationSlots , ,) = IUniswapV3Pool(poolAddress).slot0();` becomes `(, , , , uint16 observationSlots , ,) = IUniswapV3Pool(poolAddress).slot0();`

- talegift (Wild Credit) confirmed

[L-09] `repayAll()` and `repayAllETH()` vulnerable to front-running

Submitted by toastedsteaksandwich

The `repayAll()` and `repayAllETH()` functions allow any user to pay off debt of another user. Since all of the debt is going to be paid, no amount is specified, allowing the recipient of the repayment to front-run the transaction to increase their debt. The risk of this issued was lowered as it depended on the user having enough

tokens and allowance in the case of `repayAll()`, or having a `msg.sender` higher than the current debt in the case of `repayAllEth()`.

The affected lines are `LendingPair.sol` #L14 and #L156.

The scenario for `repayAll()` is the following:

1. Alice pays off 5 of Bob's Dai debt using `repayAll()`.
2. Bob monitors the mempool for Alice's transaction, and front-runs it by taking out as much debt as Alice's allowance (and therefore balance) to the contract.
3. `debtOf[_token][_account]` now returns the higher amount and pays off Bob's new debt.

The scenario for `repayAllEth()` is similar:

1. Alice pays off 0.5 of Bob's Weth debt using `repayAllEth()`.
2. Bob monitors the mempool for Alice's transaction, and front-runs it by taking out as much debt as Alice's `msg.value` amount used.
3. `debtOf[address(WETH)][_account]` now returns the higher amount and pays off Bob's new debt.

This issue can be mitigated by enforcing a minimum time to hold debt - e.g. not allowing to repay debt for at least 6 blocks. Alternatively, the `repay()` function could be used to replace the 2 affected functions by passing in the `_amount` as the total debt (looked up off-chain and used in the dapp, for example) so that only up to a certain amount of debt is paid. This also means the `repay()` function would need to be made `payable`, and that the `msg.value` is validated to be equal to the `_amount` parameter.

- talegift (Wild Credit) confirmed

[L-10] Critical protocol parameter configuration/changes should have sanity/threshold checks

Submitted by 0xRajeev

Input validation on key function parameters is a best-practice. Not applying sanity/threshold checks will allow incorrect values to be set accidentally/maliciously and may significantly affect the security/functionality of the protocol.

Except for checks on liquidation fees and collaterization factor, the codebase does not have input validation (sanity/threshold checks) on key protocol parameters in setter functions that are callable only by contract owners. Given that the fundamental value proposition of the protocol is to address risk management using isolated lending pairs, it becomes even more important to enforce sanity/threshold validation on protocol parameters in order to increase confidence in them, reduce risk and prevent accidental/malicious changes that increases risk significantly. The sanity/threshold values may be configurable in a time-delayed manner by owner/governance instead of hardcoding and enforcing unilaterally.

Scenario: Protocol is initialized/configured with absurd/unreasonable (i.e. very low/high) values of critical parameters such as `depositLimit`, `borrowLimit`, `minBorrowUSD`, pool allocation points, `totalRewardPerBlock`, `poolFee`, `twapPeriod` or `minObservations`. Owners/users fail to check or understand the impact of these absurd values until the protocol's functionality or profitability is affected significantly.

See similar high-severity finding from Consensys Diligence Audit of Shell Protocol. See issue page for referenced code.

Recommend enforcing sanity/threshold checks in all `onlyOwner` setters.

talegift (Wild Credit) acknowledged but disagreed with severity:

> Severity should be 1.

ghoul-sol (Judge) commented:

> Agree with sponsor, this is low risk at most.

[L-11] Erc20 Race condition for allowance

Submitted by JMukesh

Due to the implementation of the `approve()` function in `LPTokenMaster.sol` it's possible for a user to over spend their allowance in certain situations. See unboxing erc20 approve issues.

Recommend that, instead of having a direct setter for allowances, `decreaseAllowance` and `increaseAllowance` functions should be exposed which decreases and increases allowances for a recipient respectively.

talegift (Wild Credit) disputed:

> This is not possible. Solidity 8 checks for underflow.

ghoul-sol (Judge) commented:

> Warden, please provide more explicit explanation of the exploit in the future so we don't have to guess. Providing an article link is not the way to do it. Making this a low risk as this is a well known issue and the protocol doesn't really depend on it.

[L-12] The interest rate is calculated based on assumptions on the block time

Submitted by shw

The `InterestRateModel` contract assumes that the average block time is 13.2 seconds. However, the block time could range from 13.0 to 30.27, as we have seen in the past. The use of inaccurate block time could cause inaccurate borrow and supply rates. See `InterestRateModel.sol` #L17-L18 and Ethereum Average Block Time Chart (Etherscan.io)

Recommend allowing authorized parties to set the values of the `LOW_RATE` and `HIGH_RATE` variables. Or, designing them to be dynamically adjusted when the block time changes drastically. Alternatively, also recommend calculating the interests and rewards based on `block.timestamp` instead of `block.number`.

talegift (Wild Credit) acknowledged but disagreed with severity:

> `block.timestamp` could be manipulated by miners.

> This can be considered an acceptable risk. If the block time changes, we can swap the interest rate model in the controller quickly.

> Severity should be set to 1.

ghoul-sol (Judge) commented:

> I don't see a direct exploit here. If blocks are produced faster, interest rate is calculated faster, if slower it's lower. This can be an issue however I agree it's a low risk.

Non-Critical Findings (24)

- [N-01] Unused imported interface in `LendingPair`
- [N-02] typo: `totalAccountBorrrow`
- [N-03] typo in revert
- [N-04] Boolean to constant comparison
- [N-05] `Math.max` can be used
- [N-06] when setting new value for `feeRecepient` / `totalRewardPerBlock` ensure that new value is different from old one
- [N-07] Undefined Event
- [N-08] Use immutable keyword
- [N-09] difference between `_safeTransferFrom` and `_safeTransfer`
- [N-10] `InterestRateModel` : Use constant for 100e18
- [N-11] `LendingPair` : Error Messages can be improved
- [N-12] UniswapV3Oracle: No events emitted for setUniPriceConverter, setTwapPeriod, setMinObservations
- [N-13] `LendingPair` : Missing validation check for ETH methods [Updated]
- [N-14] setReward does not check if pid exists
- [N-15] LPTokenMaster: CEI Pattern
- [N-16] Lack of zero address validation
- [N-17] Use of Floating Pragma
- [N-18] LPTokenMaster does not implement `IERC20`

- [N-19] Code size exceed limit
- [N-20] addPool emits PoolUpdate with wrong pid
- [N-21] Add a proper revert message in `transferFrom` of `LPTokenMaster`
- [N-22] Unimplemented methods in several interfaces
- [N-23] Single-step process for critical ownership transfers is risky
- [N-24] Interest model is non-continuous

Gas Optimizations (16)

- [G-01] `LendingPair` : Unnecessary Casting
- [G-02] redundant call to `_checkMinReserve` in `withdrawBorrowETH`
- [G-03] External call does not have amount check in `TransferHelper`
- [G-04] `InterestRateModel` : Infallible logic
- [G-05] Packing of variable in controller.sol
- [G-06] Gas optimizations - Use bytesX instead of string
- [G-07] Gas optimizations - Check first If `blocksElapsed == 0` in `_pendingRewardPerToken`
- [G-08] Gas optimizations - optimize reads in `_distributeReward`
- [G-09] `LendingPair` : Cache token decimals
- [G-10] `LendingPair` : Optimise liquidation parameter calculations
- [G-11] `LPTokenMaster` : `underlying()` → address `underlyingToken`
- [G-12] `RewardDistribution` : Avoid 0 pid to drop boolean flag use
- [G-13] `RewardDistribution` : Redundant boolean flag check in `_getPid()`
- [G-14] `RewardDistribution` : Optimise `_getPid`
- [G-15] `RewardDistribution` : Optimise `pendingSupplyReward`, `pendingBorrowReward`, `_distributeReward` and `_poolRewardRate`
- [G-16] Variables that can be converted into immutables

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