Union Finance contest
Findings & Analysis Report

2021-12-08

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 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 Union Finance smart contract system written in Solidity. The code contest took place between October 14—October 20 2021.

Wardens

14 Wardens contributed reports to the Union Finance contest:

1. cmichel
2. WatchPug (jtp and ming)
3. csanuragjain
4. itsmeSTYJ
5. kenzo
6. gpersoon
7. pants
8. pmerkleplant
9. pauliax
10. hyh
11. defsec
12. ye0lde
13. loop

This contest was judged by AlexTheEntreprenerd.

Final report assembled by moneylegobatman and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 25 unique vulnerabilities and 81 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity, 11 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 13 non-critical recommendations and 43 gas optimizations.

Scope

The code under review can be found within the C4 Union Finance contest repository, and is composed of 98 smart contracts written in the Solidity programming language and includes 4,834 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.

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 (2)

[H-01] `borrow` must `accrueInterest` first

Submitted by cmichel

The `UToken.borrow` function first checks the borrowed balance and the old credit limit *before* accruing the actual interest on the market:

```
// @audit this uses the old value
require(borrowBalanceView(msg.sender) + amount + fee <= maxBorrc

require(
    // @audit this calls uToken.calculateInterest(account) which
    uint256(_getCreditLimit(msg.sender)) >= amount + fee,
    "UToken: The loan amount plus fee is greater than credit lim
);

// @audit accrual only happens here
```

```
require(accrueInterest(), "UToken: accrue interest failed");
```

Thus the borrowed balance of the user does not include the latest interest as it uses the old global `borrowIndex` but the new `borrowIndex` is only set in `accrueInterest`.

Impact

In low-activity markets, it could be that the `borrowIndex` accruals (`accrueInterest` calls) happen infrequently and a long time is between them. A borrower could borrow tokens, and borrow more tokens later at a different time without first having their latest debt accrued. This will lead to borrowers being able to borrow more than `maxBorrow` and **more than their credit limit** as these checks are performed before updating accruing interest.

Recommended Mitigation Steps

The `require(accrueInterest(), "UToken: accrue interest failed");` call should happen at the beginning of the function.

GeraldHost (Union Finance) confirmed

GalloDaSballo (judge) commented:

> Agree with the finding, this fundamentally breaks the accounting of the protocol
>
> In protocols that calculate interest, and that have to recalculate state after something changed, it is vital that you accrue all changes up to this point before proceeding with any other state-changing logic

[H-02] Wrong implementation of `CreditLimitByMedian.sol#getLockedAmount()` makes it unable to unlock `lockedAmount` in `CreditLimitByMedian` model

Submitted by WatchPug

CreditLimitByMedian.sol L27-L78

```
    function getLockedAmount(
        LockedInfo[] memory array,
        address account,
        uint256 amount,
        bool isIncrease
    ) public pure override returns (uint256) {
        if (array.length == 0) return 0;

        uint256 newLockedAmount;
        if (isIncrease) {
            ...
        } else {
            for (uint256 i = 0; i < array.length; i++) {
                if (array[i].lockedAmount > amount) {
                    newLockedAmount = array[i].lockedAmount - 1;
                } else {
                    newLockedAmount = 0;
                }

                if (account == array[i].staker) {
                    return newLockedAmount;
                }
            }
        }

        return 0;
    }
```

`getLockedAmount()` is used by `UserManager.sol#updateLockedData()` to update locked amounts.

Based on the context, at L66, `newLockedAmount = array[i].lockedAmount - 1;` should be `newLockedAmount = array[i].lockedAmount - amount;`.

The current implementation is wrong and makes it impossible to unlock `lockedAmount` in `CreditLimitByMedian` model.

Recommendation

Change to:

```
newLockedAmount = array[i].lockedAmount - amount;
```

kingjacob (Union) acknowledged

GalloDaSballo (judge) commented:

> The warden identified a mistake in the accounting that would make it impossible to unlock funds, mitigation seems to be straightfoward

Medium Risk Findings (11)

[M-01] Wrong implementation of `CreditLimitByMedian.sol#getLockedAmount()` will lock a much bigger total amount of staked tokens than expected

Submitted by WatchPug, also found by itsmeSTYJ

`CreditLimitByMedian.sol` L27-L63

```
    function getLockedAmount(
        LockedInfo[] memory array,
        address account,
        uint256 amount,
        bool isIncrease
    ) public pure override returns (uint256) {
        if (array.length == 0) return 0;

        uint256 newLockedAmount;
        if (isIncrease) {
            for (uint256 i = 0; i < array.length; i++) {
                uint256 remainingVouchingAmount;
                if (array[i].vouchingAmount > array[i].lockedAmount)
                    remainingVouchingAmount = array[i].vouchingAmour
                } else {
                    remainingVouchingAmount = 0;
                }

                if (remainingVouchingAmount > array[i].availableStak
```

```
                    if (array[i].availableStakingAmount > amount) {
                        newLockedAmount = array[i].lockedAmount + am
                    } else {
                        newLockedAmount = array[i].lockedAmount + ar
                    }
                } else {
                    if (remainingVouchingAmount > amount) {
                        newLockedAmount = array[i].lockedAmount + am
                    } else {
                        newLockedAmount = array[i].lockedAmount + re
                    }
                }

                if (account == array[i].staker) {
                    return newLockedAmount;
                }
            }
        } else {
        ...
```

`getLockedAmount()` is used by `UserManager.sol#updateLockedData()` to update locked amounts.

The current implementation is wrong and locks every staker for the amount of the borrowed amount or all the `vouchingAmount` if the `vouchingAmount` is smaller than the borrowed amount in `CreditLimitByMedian` model.

PoC

- 10 stakers each give `100` of `vouchingAmount` to Alice;
- Alice borrows `100`;

The protocol will now lock `100` of each of the 10 stakers, making the total locked amount being `1000`.

kingjacob (Union) disputed:

> This is as designed.

GalloDaSballo (judge) commented:

> With the evidence shown and the comment of the sponsor, it seems to me like this is a bad idea, the protocol will lock funds at a rate of N * X, where X was the original amount borrowed and N is the number of stakers that are covering for it.

> This seems to be a surefire way for griefing, DOS attacks, and potentially to block other people funds in the protocol

> I would highly recommend the sponsor to consider the possibility that this behaviour can be used against the users of the protocol and can potentially kill the protocol in it's infancy

kingjacob (Union) commented:

> While creditlimitbymedian seems inefficient from a capital locked per loans outstanding view, it is a valid if agressive model. We've run agent simulation on both it and sumoftrust and theres tradeoffs with both.

> For clarity of whats deployed, we'll be deleting creditlimitbymedian from the repo but it is a valid implementation.

kingjacob (Union) commented:

> @GalloDaSballo this issue id argue is invalid as the implementation is correct and locks funds as instructed by creditlimitbymedian. Which is as designed.

> Locking > $M in underwriting per $M in borrowed funds is also not a "loss of funds" its common practice in credit markets, and might actually end up be required if default rates are high enough. And in the above model, the only one who can lock funds is the person the user chose to give the right to borrow (and lock the funds). Which I'd argue is distinct from an actual grief or DoS. (Unless you had something else in mind?)

> But all thats a bit offtopic, as this issue is reporting a wrong implementation, not an inefficient underwriting model.

GalloDaSballo (judge) commented:

> @kingjacob (Union) For the sake of understarding, let's assume I convince 10 people to each put 1MLN so I can borrow 1 MLN. If I then run away with the money

> and default, their 10 MLN would get locked.

> What would happen after?

kingjacob (Union) commented:

> @GalloDaSballo nothing. Stake stays locked earning interest which slowly fills the whole left by the default. Unless one of the stakers calls writeoffdebt.

> We dont solve the problem of what if someone trusts someone they shouldnt. Email me at jacob@union.finance, can hop on a call to explain in more detail.

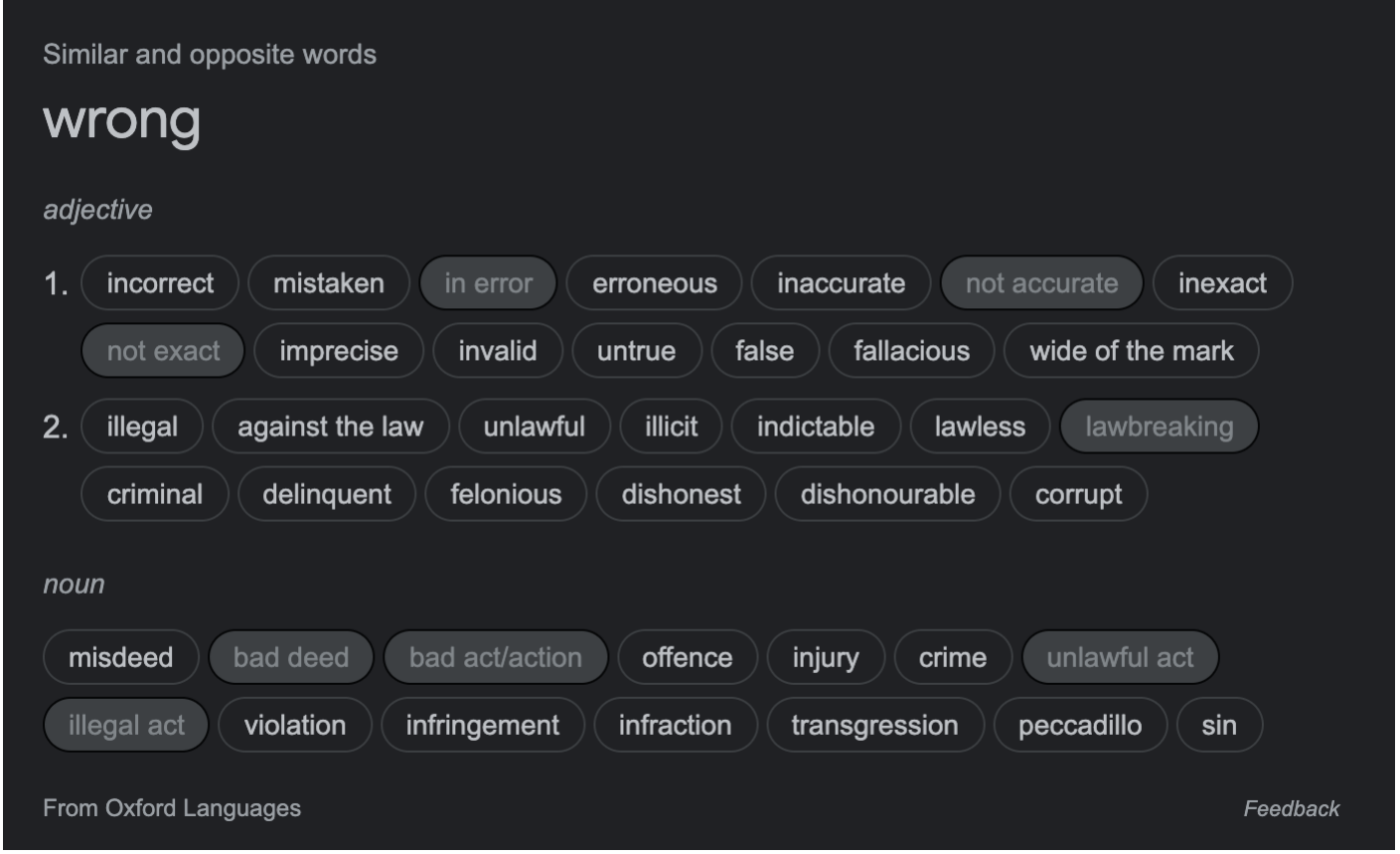
GalloDaSballo (judge) commented:

> Me and the sponsor have scheduled a clarification call, the conversation should help clarify our different opinions

> Before the call I'd like to record my current opinion:

- The system does what the sponsor designed (locks funds on default)
- To my understanding the system will lock N times the tokens necessary (again by design)
- By my experience in DeFi this means that potentially a lot of people can get locked
- There is a technicality with the findings that says "Wrong Implementation", the implementation is as the sponsor specified, however, for a myriad of reasons, `wrong` is a synonym with "fallacious" / "errouneous", hence my interpretation of the finding at this time is that it's not a warning for the code not being to spec, but rather the code allowing for a risky situation (tons of people with funds locked)

> Ref: Google's synonims for `wrong`



GalloDaSballo (judge) commented:

> Given the specifics of this risk I am conflicted between a High (can lock arbitrary funds for an arbitrary amount of people) and Medium (they agreed to the dynamic, the time being locked is proportional to yield, higher yield = less time) severity finding, however believe it is correct for me to talk with the sponsor to hear their side of the story

GalloDaSballo (judge) commented:

> After the conversation and the clarifications from the sponsor we both agree that the finding is of medium severity. Allowing someone to borrow is a situation closer to giving your allowance, more of a feature than a risk. However, locking a disproportional amount of people on default can be problematic and I think there should be a cap on that.

> The sponsor will mitigate by removing `CrediLimitByMedian` from the codebase

GalloDaSballo (judge) commented:

> For the sake of transparency, we have recorded the call here https://us02web.zoom.us/rec/share/kx789PdETc3NqJ7J9gy1tNHazJAmYufFfwKD-ob_Ct6akpnC-sZPp84cLozLOlpm.khrAW4nfFCPxJZCx Passcode: 2A07bsS@

GalloDaSballo (judge) commented:

> I've re-rated the severity of the issue given the specifics of the risk, as per CodeArena docs: 2 — Med: 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.

[M-02] Rebalance will fail due to low precision of percentages

Submitted by cmichel, also found by hyh

The `AssetManager.rebalance` function has a check at the end to ensure that all tokens are deposited again:

```
require(token.balanceOf(address(this)) == 0, "AssetManager: ther
```

The idea is that the last market deposits all `remainingTokens` but the last market does not have to support the token in which case the transaction will fail, or the `percentages` parameter needs to be chosen to distribute all tokens before the last one (they need to add up to `1e4`). However, these percentages have a low precision as they are in base points, i.e, the lowest unit is `1 = 0.01%`. This will leave dust in the contract in most cases as the tokens have much higher precision.

POC

Assume the last market does not support the token and thus `percentages` are chosen as `[5000, 5000]` to rebalance the first two markets. Withdrawing all tokens form the markets leads to a `tokenSupply = token.balanceOf(address(this)) = 10,001`:

Then the deposited amount is `amountToDeposit = (tokenSupply * percentages[i]) / 10000 = 10,001 * 5,000 / 10,000 = 5,000`. The two deposits will leave dust of `10,001 - 2 * 5,000 = 1` in the contract and the `token.balanceOf(address(this)) == 0` balance check will revert.

Impact

Rebalancing will fail in most cases if the last market does not support the token due to precision errors.

Recommended Mitigation Steps

Remove the final zero balance check, or make sure that the last market that is actually deposited to receives all remaining tokens.

kingjacob (Union) confirmed

[M-03] `UnionToken` should check whitelist on `from`?

Submitted by cmichel

The `UnionToken` can check for a whitelist on each transfer in `_beforeTokenTransfer`:

```
    if (whitelistEnabled) {
        require(isWhitelisted(msg.sender) || to == address(0), "Whit
    }
```

This whitelist is checked on `msg.sender` not on `from`, the token owner.

Impact

A single whitelisted account can act as an operator (everyone calls `unionToken.allow(operator, max)` where the operator is a whitelisted trusted smart contract) for all other accounts. This essentially bypasses the whitelist.

Recommended Mitigation Steps

Think about if the whitelist on `msg.sender` is correct or if it should be on `from`.

GeraldHost (Union Finance) confirmed

GalloDaSballo (judge) commented:

> Agree with the warden findings, since `_beforeTokenTransfer` is called on all transfers (transfer and transferFrom) [https://github.com/OpenZeppelin/openzeppelin-contracts/blob/e63b09c9ad3a45484b6dc304e0e99640a9dc3036/contracts/token/ERC20/ERC20.sol#L229]

> In order to enforce the whitelist you need to check against from and not msg.sender

> msg.sender could be a relayer or another contract, while `from` will be the account the tokens are being moved from

> Given the context and info I have this is a way to sidestep the guestList, hence a medium severity attack

[M-04] Change in interest rate can disable repay of loan

Submitted by pmerkleplant

Impact

The ability of a borrower to repay a loan is disabled if the interest rate is set too high by the `InterestRateModel`.

However, there is neither a check when setting the interest rate nor an indication in the `IInterestRateModel`'s specs of this behavior.

But this issue could also be used in an adversarial fashion by the `FixedInterestRateModel`-owner if he/she would disable the repay functionality for some time and enables it at a later point again with the demand of a higher interest to be paid by the borrower.

Proof of Concept

If an account wants to repay a loan, the function `UToken::_repayBorrowFresh()` is used. This function calls `UToken::accrueInterest()` (line 465) which fetches the current borrow rate of the interest rate model (line 546 and line 330).

The function `UToken::borrowRatePerBlock()` requires an not “absurdly high” rate, or fails otherwise (line 331).

However, there is no check or indicator in `FixedInterestRateModel.sol` to prevent the owner to set such a high rate that effectively disables repay of borrowed funds (line 36).

Recommended Mitigation Steps

Disallow setting the interest rate too high with a check in `FixedInterestRateModel::setInterestRate()`.

kingjacob (Union) confirmed

GalloDaSballo (judge) commented:

> Agree with the need for a check on the `setInterestRate` function Since the warden showed a specific way to negate certain protocol functionality, under specific assumptions, the finding is of medium severity

[M-05] Comptroller rewards can be artificially inflated and drained by manipulating [totalStaked - totalFrozen] (or: wrong rewards calculation)

Submitted by kenzo

By adding a small of amount of staking to a normal user scenario, and not approving this small amount as a loan for anybody, a staker can gain disproportionate amounts of comptroller rewards, even to the point of draining the contract. For example: Stakers A,B,C stake 100, 65, 20, approve it for borrower Z, then staker B stakes an additional 0.07 DAI, and borrower Z borrows 185. This will result in disproportionate amount of rewards.

As far as I see, this is the main line that causes the inflated amount (*deep breath*): In `calculateRewardsByBlocks`, you set:

```
userManagerData.totalStaked = userManagerContract.totalStaked()
```

`Comptroller.sol` **L140**

Note that a staker can make this amount very small (depending of course on the current numbers of the protocol). (A more advanced attacker might diminish the effect of the current numbers of the protocol by initiating fake loans to himself and not paying them.) This field is then passed to `calculateRewards`, and passed further to `getInflationIndexNew`, and further to `getInflationIndex`. passed to `calculateRewards` : `Comptroller.sol` **L167**

passed to `getInflationIndexNew` : `Comptroller.sol` **L259**

passed to `getInflationIndex` : `Comptroller.sol` **L238**

Now we actually use it in the following line (as `effectiveAmount`):

```
return blockDelta * inflationPerBlock(effectiveAmount).wadDiv(ef
```

`Comptroller.sol` **L315**

So 2 things are happening here:

1. mul by `inflationPerBlock(effectiveAmount)` - uses the lookup table in Comptroller. This value gets bigger as effectiveAmount gets smaller, and if effectiveAmount is in the area of 10**18, we will get the maximum amount of the lookup.
2. div by `effectiveAmount` - as we saw, this can be made small, thereby enlarging the result.

All together, this calculation will be set to `curInflationIndex` and then used in the following line:

```
return (curInflationIndex - startInflationIndex).wadMul(effectiv
```

`Comptroller.sol` **L263**

Note the `curInflationIndex - startInflationIndex` : per my POC (see below), this can result in a curInflationIndex which is orders of magnitude larger (200x) than startInflationIndex. This creates a huge inflation of rewards.

Impact

Comptroller rewards can be drained.

Proof of Concept

See the following script for a POC of reward drainage. It is based on the scenario in test/integration/ `testUserManager` :

Stakers A,B,C stake 100, 65, 20, and borrower Z borrows 185. But the difference in my script is that just before borrower Z borrows 185, staker B stakes an additional 0.07 DAI. (This will be the small amount that is `totalStaked - totalFrozen`).

Then, we wait 11 blocks to make the loan overdue, call `updateOverdueInfo` so `totalFrozen` would be updated, and then staker B calls `withdrawRewards` . He ends up with 873 unionTokens out of the 1000 the Comptroller has been seeded with. And this number can be enlarged by changing the small additional amount that staker B staked.

In this scenario, when calling `withdrawRewards` , the calculated `curInflationIndex` will be 215 WAD, while `startInflationIndex` is 1 WAD, and this is the main issue as I understand it.

File password: “union”. https://pastebin.com/3bJF8mTe

Tools Used

Manual analysis, hardhat

Recommended Mitigation Steps

Are you sure that this line should deduct the `totalFrozen` ?

```
userManagerData.totalStaked = userManagerContract.totalStaked()
```

`Comptroller.sol` **L140**

Per my tests, if we change it to just

```
userManagerData.totalStaked = userManagerContract.totalStaked();
```

Then we are getting normal results again and no drainage. And the var *is* called just `totalStaked` ... So maybe this is the change that needs to be made? But maybe you have a reason to deduct the `totalFrozen`. If so, then a mitigation will perhaps be to limit curInflationIndex somehow, maybe by changing the lookup table, or limiting it to a percentage from startInflationIndex ; but even then, there is also the issue of dividing by `userManagerData.totalStaked` which can be made quite small as the user has control over that.

kingjacob (Union) confirmed

GalloDaSballo (judge) commented:

> Agree with the finding, the warden has found a specific attack that can leak value, while the leak value marks it as Medium Severity, will think over if the economic exploit is big enough to warrant a high severity

[M-06] debtWriteOff updates `totalFrozen` immaturely, thereby losing staker rewards

Submitted by kenzo, also found by itsmeSTYJ

`debtWriteOff` updates `totalFrozen` before withdrawing `unionToken` rewards. As the borrower is overdue, this means the staker calling debtWriteOff will lose his rewards if for example `totalStaked == totalFrozen`. (Note: If the borrower would to first call `withdrawRewards` /stake/unstake before calling debtWriteOff, he would get the rewards.)

Impact

Staker loses rewards. (Or at the very least, inconsistency at rewards calculation between debtWriteOff and stake/unstake/ `withdrawRewards`.)

Proof of Concept

`debtWriteOff` first calls `updateTotalFrozen`, and then `comptroller.withdrawRewards`: `L710:` L712

`updateTotalFrozen` can update `totalFrozen` to be same as `totalStaked`.
`comptroller.withdrawRewards` calls `calculateRewardsByBlocks`: `Comptroller.sol` L98

`calculateRewardsByBlocks` is calculating `userManagerContract.totalStaked() - userManagerData.totalFrozen`, which can be 0 in certain cases, `Comptroller.sol` L140

and passing it as the third parameter to `calculateRewards`: `Comptroller.sol` L167

In `calculateRewards`, if the third parameter is 0, the user won't get any rewards.

`Comptroller.sol` L253

So in this scenario the user won't get any rewards after calling `debtWriteOff`.

If we were to call `updateTotalFrozen` *after* the withdrawing of the rewards, or if the staker would call `withdrawRewards` before calling `debtWriteOff`, the `totalFrozen` would not have been updated, and the user would get his rewards by calling `debtWriteOff`. As I mentioned earlier, if there's a reason I'm not seeing as to why `updateTotalFrozen` is updated in `debtWriteOff` before `withdrawRewards` is called, then it is not consistent with stake/unstake/ `withdrawRewards` functions.

I have a created an (admittedly hacky) script to show the bug. It will run two scenarios which are almost the same (based on integration/ `testUserManager`.js). In the first the staker will call `debtWriteOff` at some point, and at the second the staker will call `withdrawRewards` at the same point, and the test will print the difference in unionToken balance after each call. File password: "union".
https://pastebin.com/xkS0PXtq

Tools Used

Manual analysis, hardhat.

Recommended Mitigation Steps

If I am not missing anything, in `debtWriteOff` I would move the `withdrawRewards` to before `updateTotalFrozen`.

`UserManager.sol` L710:L712

kingjacob (Union) acknowledged

GalloDaSballo (judge) commented:

> As described by the warden, the `debtWriteOff` function causes users to loose rewards, since this is a loss of yield will maintain a Med Risk severity

[M-07] `UserManager: totalStaked ≥ totalFrozen` should be checked before and after totalFrozen is updated

Submitted by itsmeSTYJ

The require statement in `updateTotalFrozen` and `batchUpdateTotalFrozen` to check that `totalStaked ≥ totalFrozen` should be done both before and after `_updateTotalFrozen` is called to ensure that totalStake is still ≥ `totalFrozen`. This will serve as a sanity check to ensure that the integrity of the system is not compromised.

kingjacob (Union) confirmed

GalloDaSballo (judge) commented:

> Agree with the finding and the recommendation of adding an additional require to ensure protocol invariants aren't broken

[M-08] `MAX_TRUST_LIMIT` might be too high

Submitted by gpersoon

Both `SumOfTrust.sol` and `CreditLimitByMedian.sol` contain an expensive sort function. This is used by `UserManager.sol` via the functions `getLockedAmount` and `getCreditLimit`.

If the list of stakers would be very long then the sort would take up all the gas and revert. Attackers could make the list of stakers longer by voting in themselves (as soon as they have 3 accounts voted in), this would result in a griefing attack.

Luckily the number of stakers and borrowers is limited in the function updateTrust by applying a limit of `MAX_TRUST_LIMIT`.

However this limit is quite high (100), if that amount of stakers would be present then an out of gas error would probably occur with the sort. Note: there are also other for loops in the code that could have a similar problem, however sort is the most expensive.

Proof of Concept

- https://github.com/code-423n4/2021-10-union/blob/4176c366986e6d1a6b3f6ec0079ba547b040ac0f/contracts/user/SumOfTrust.sol#L98-L121
- https://github.com/code-423n4/2021-10-union/blob/4176c366986e6d1a6b3f6ec0079ba547b040ac0f/contracts/user/CreditLimitByMedian.sol#L107-L122
- https://github.com/code-423n4/2021-10-union/blob/4176c366986e6d1a6b3f6ec0079ba547b040ac0f/contracts/user/UserManager.sol#L594
- https://github.com/code-423n4/2021-10-union/blob/4176c366986e6d1a6b3f6ec0079ba547b040ac0f/contracts/user/UserManager.sol#L368
- https://github.com/code-423n4/2021-10-union/blob/4176c366986e6d1a6b3f6ec0079ba547b040ac0f/contracts/user/UserManager.sol#L50
- https://github.com/code-423n4/2021-10-union/blob/4176c366986e6d1a6b3f6ec0079ba547b040ac0f/contracts/user/UserManager.sol#L423-L427

Recommended Mitigation Steps

Do a test with a `MAX_TRUST_LIMIT` number of stakers and borrowers and check if the code still works.

Set the `MAX_TRUST_LIMIT` so that everything still works, probably include a margin for future changes in gas costs.

GeraldHost (Union Finance) acknowledged:

> yes we have tested for this and landed on the 100 limit. We have plans to improve gas efficiency considerably in future.

GalloDaSballo (judge) commented:

> The warden has indentify a griefing exploit that can be applied only under specific circumstances, I agree with the severity

[M-09] Duplicate `utoken` and `usermanager` can be added which cannot be deleted

Submitted by csanuragjain

If Admin decides to delete the market, only the first instance of `utoken` and `usermanager` gets deleted. This means duplicate instance remains and Admin has actually not deleted the market

Proof of Concept

1. Navigate to https://github.com/code-423n4/2021-10-union/blob/main/contracts/market/MarketRegistry.sol
2. Check the `addUToken` function

```
function addUToken(address token, address uToken) public newToke
    uTokenList.push(uToken);
    tokens[token].uToken = uToken;
    emit LogAddUToken(token, uToken);
}
```

3. As we can see there is no check to see if the utoken already existed in uTokenList which means now uTokenList can have now duplicate entries
4. Same case goes for userManagerList

Recommended Mitigation Steps

Modify `addUToken` and `addUserManager` function to check if the usermanager/utoken already existed

GalloDaSballo (judge) commented:

> Agree with the finding, since the warden showed a specific attack with stated conditions, this a medium severity finding

[M-10] User Fund loss in case of Unsupported Market token deposit

Submitted by csanuragjain

Impact

User funds can be lost as current logic cannot withdraw unsupported market token even though deposit can be done for same

Proof of Concept

1. Navigate to https://github.com/code-423n4/2021-10-union/blob/main/contracts/asset/AssetManager.sol
2. Check the function deposit

```
function deposit(address token, uint256 amount)
    external
    override
    whenNotPaused
    onlyAuth(token)
    nonReentrant
    returns (bool)
{
    ...
```

```
    bool remaining = true;
    if (isMarketSupported(token)) {
        ...
    }

    if (remaining) {
        poolToken.safeTransferFrom(msg.sender, address(this), am
    }

    ...
}
```

3. If this function was called with unsupported market token then `isMarketSupported(token)` will result in false. Although remaining remains true. This means `poolToken.safeTransferFrom(msg.sender, address(this), amount);` will get executed and user money will be debited

```
if (remaining) {
    poolToken.safeTransferFrom(msg.sender, address(this), amount
}
```

4. Lets say user decides to withdraw using withdraw function

```
function withdraw(
    address token,
    address account,
    uint256 amount
) external override whenNotPaused nonReentrant onlyAuth(token) r
    ...

    if (isMarketSupported(token)) {
        ...
    }

    if (!_isUToken(msg.sender, token)) {
        balances[msg.sender][token] = balances[msg.sender][toker
        totalPrincipal[token] = totalPrincipal[token] - amount +
    }

    emit LogWithdraw(token, account, amount, remaining);
```

```
        return true;
    }
```

5. As User is trying to withdraw unsupported market token so `isMarketSupported` function will return false. This means user wont be able to withdraw the funds

Recommended Mitigation Steps

`Deposit` function should revert in case of non supported market tokens

kingjacob (Union) acknowledged and disagreed with severity

GalloDaSballo (judge) commented:

> Agree with the adjusted severity of Medium, this is a loss of funds that can happen under specific conditions

> That said, the finding is valid, a simple mitigation would be to return / stop the execution if the token is not supported (or revert if you prefer that)

[M-11] Rebalance will fail if a market has high utilization

Submitted by cmichel

The `AssetManager.rebalance` function iterates through the markets and withdraws **all** tokens in the `moneyMarkets[i].withdrawAll` call.

Note that in peer-to-peer lending protocols like Compound/Aave the borrower takes the tokens from the supplier and it might not be possible for the supplier to withdraw all tokens if the utilisation ratio of the market is high.

See this check for example in **Compound's cToken**.

Impact

Rebalancing will fail if a single market has a liquidity crunch.

Recommended Mitigation Steps

Withdraw only what's available and rebalance on that instead of trying to pull all tokens from each market first. Admittedly, this might be hard to compute for some protocols.

kingjacob (Union) acknowledged

GalloDaSballo (judge) commented:

> Agree with the finding, at this time this potential vulnerability is a feature of the protocol

> I recommend in the long term, that the sponsor rewrites the `rebalance` function to account for liquidity crunches

Low Risk Findings (12)

- **[L-01] Zero-address checks are missing** *Submitted by defsec, also found by pants and pauliax*
- **[L-02] Lack of precision in `wadDiv`** *Submitted by pants*
- **[L-03] `setHalfDecayPoint` check allowed values** *Submitted by gpersoon*
- **[L-04] `withdrawRewards` should send remaining balance** *Submitted by cmichel*
- **[L-05] `repayBorrowWithPermit` is missing `nonReentrant`** *Submitted by cmichel*
- **[L-06] Unbounded iteration in `deleteMarket`** *Submitted by cmichel, also found by pauliax*
- **[L-07] `withdrawSeq` might not be set** *Submitted by cmichel*
- **[L-08] UToken._*UToken*init can be frontrun** *Submitted by pants*
- **[L-09] `UToken.sol` should inherits and complies with `IUToken.sol`** *Submitted by WatchPug*
- **[L-10] Improper Upper Bound Definition on the New Member Fee** *Submitted by defsec*

- [L-11] Governor contract is not matching Contract source *Submitted by csanuragjain*
- [L-12] Two-step change of a critical parameter *Submitted by pauliax, also found by pants*

Non-Critical Findings (13)

- [N-01] Unneeded Named Returns (UToken.sol) *Submitted by yeOlde*
- [N-02] list of _admins *Submitted by pauliax*
- [N-03] WadRayMath state variables *Submitted by pants*
- [N-04] UserManager: _getFrozenCoinAge is not used *Submitted by itsmeSTYJ*
- [N-05] AssetManager: getLoanableAmount() can be made more readable *Submitted by itsmeSTYJ*
- [N-06] Code Style: constants should be named in all caps *Submitted by WatchPug*
- [N-07] Unused imports *Submitted by WatchPug*
- [N-08] Code Style: consistency *Submitted by WatchPug*
- [N-09] deposit onlyAssetManager *Submitted by pauliax*
- [N-10] Open TODOs in `Treasury.sol` *Submitted by pants*
- [N-11] Missing events for owner only functions that change critical parameters *Submitted by defsec*
- [N-12] Inconsistent use of `UToken::getBorrowed()` *Submitted by pmerkleplant*
- [N-13] Inconsistent use of `UToken::getLastRepay()` *Submitted by pmerkleplant*

Gas Optimizations (43)

- [G-01] For Loops Need Break Statements (UserManager.sol) *Submitted by yeOlde*
- [G-02] Function getFrozenCoinAge Can Be Made More Efficient (UserManager.sol) *Submitted by yeOlde*

- [G-03] Function checkIsOverDue Can Be Made More Efficient (UToken.sol) *Submitted by yeOlde*
- [G-04] Functions TotalSupplyView/TotalSupply Can Be Made More Efficient (AssetManager.sol) *Submitted by yeOlde*
- [G-05] Long Revert Strings *Submitted by yeOlde*
- [G-06] Unchecked math operations *Submitted by pauliax*
- [G-07] .length in a loop *Submitted by pauliax, also found by pants*
- [G-08] Zero transfers *Submitted by pauliax*
- [G-09] Pre-calculate known values *Submitted by pauliax*
- [G-10] Struct with only 1 element *Submitted by pauliax*
- [G-11] Immutable variables *Submitted by pauliax*
- [G-12] getSupply and getSupplyView are identical *Submitted by pauliax*
- [G-13] UToken.uErc20 field could be immutable *Submitted by pants*
- [G-14] UToken.sol _redeemFresh could be set private instead internal *Submitted by pants*
- [G-15] caching multiple used variables *Submitted by pants*
- [G-16] double reading from memory inside a for loop. *Submitted by pants*
- [G-17] —j is more gas efficient than j—. *Submitted by pants*
- [G-18] More efficient loops *Submitted by pants*
- [G-19] stake function in `UserManager` checks for allowance, which is also done in ERC20 transferFrom *Submitted by loop*
- [G-20] Tautologies in require statements *Submitted by loop*
- [G-21] `UserManager` : debtWriteOff() doesn't need if borrower has sufficient assets frozen before subtracting *Submitted by itsmeSTYJ*
- [G-22] `UserManager` : registerMember() can be optimized further *Submitted by itsmeSTYJ*
- [G-23] `UserManager` : cancelVouch() should break from loop when address is found. *Submitted by itsmeSTYJ*
- [G-24] `UserManager` : use mapping to avoid iteration *Submitted by itsmeSTYJ*
- [G-25] `UserManager` : addMember() contains redundant require check *Submitted by itsmeSTYJ*

- [G-26] `UserManager` : getCreditLimit() can be optimized further *Submitted by itsmeSTYJ*
- [G-27] `UserManager` : getTotalLockedStake() redundant assignment *Submitted by itsmeSTYJ*
- [G-28] CreditLimitByMedian: getLockedAmount() can be optimized further. *Submitted by itsmeSTYJ*
- [G-29] UToken: revert on over/underflow checks in addReserve() and removeReserve() are unnecessary *Submitted by itsmeSTYJ*
- [G-30] UToken: _repayBorrowFresh() function can be optimized further *Submitted by itsmeSTYJ*
- [G-31] AssetManager: Deposit() function has redundant continue statement. *Submitted by itsmeSTYJ, also found by cmichel*
- [G-32] Gas efficiency suggestions *Submitted by hyh, also found by csanuragjain*
- [G-33] Overusage of gas due to non needed loop *Submitted by csanuragjain, also found by WatchPug*
- [G-34] Gas: Explicit overflow checks even though solidity 0.8 is used (2) *Submitted by cmichel*
- [G-35] Gas: Explicit overflow checks even though solidity 0.8 is used (1) *Submitted by cmichel*
- [G-36] Gas: `AssetManager.getMoneyMarket` use assignment *Submitted by cmichel*
- [G-37] Gas: `AssetManager.rebalance` cache last market *Submitted by cmichel*
- [G-38] Cache array length in for loops can save gas *Submitted by WatchPug*
- [G-39] Cache and read storage variables from the stack can save gas *Submitted by WatchPug*
- [G-40] Adding unchecked directive can save gas *Submitted by WatchPug*
- [G-41] Avoid unnecessary code execution can save some gas in edge cases *Submitted by WatchPug*
- [G-42] Use short circuiting can save gas *Submitted by WatchPug*

- [G-43] `UserManager` : _updateTotalFrozen can be optimized further

  Submitted by itsmeSTYJ

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