Mochi contest
Findings & Analysis Report

2021-11-23

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 Mochi smart contract system written in Solidity. The code contest took place between October 21—October 27 2021.

Wardens

16 Wardens contributed reports to the Mochi code contest:

1. jonah1005
2. leastwood
3. WatchPug (jtp and ming)
4. cmichel
5. gpersoon
6. harleythedog
7. gzeon
8. pauliax
9. defsec
10. yeOlde
11. nikitastupin
12. loop
13. pants

14. 0x0x0x
15. hyh

This contest was judged by Ghoul.sol.

Final report assembled by CloudEllie and moneylegobatman.

Summary

The C4 analysis yielded an aggregated total of 38 unique vulnerabilities and 89 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 13 received a risk rating in the category of HIGH severity, 15 received a risk rating in the category of MEDIUM severity, and 10 received a risk rating in the category of LOW severity.

C4 analysis also identified 18 non-critical recommendations and 33 gas optimizations.

Scope

The code under review can be found within the C4 Mochi contest repository, and is composed of 70 smart contracts written in the Solidity programming language and includes 3,828 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 (13)

[H-01] Vault fails to track debt correctly that leads to bad debt

Submitted by jonah1005, also found by WatchPug

Impact

It's similar to the issue "misuse amount as increasing debt in the vault contract". Similar issue in a different place that leads to different exploit patterns and severity.

When users borrow usdm from a vault, the debt increases by the amount * 1.005.

```
        uint256 increasingDebt = (_amount * 1005) / 1000;
```

However, when the contract records the total debt it uses `_amount` instead of `increasingDebt`.

```
    details[_id].debtIndex =
        (details[_id].debtIndex * (totalDebt)) /
        (details[_id].debt + _amount);
    details[_id].debt = totalDebt;
    details[_id].status = Status.Active;
    debts += _amount;
```

[MochiVault.sol L242-L249](#)

The contract's debt is inconsistent with the total sum of all users' debt. The bias increases overtime and would break the vault at the end.

For simplicity, we assume there's only one user in the vault. Example:

1. User deposits 1.2 M worth of BTC and borrows 1M USDM.
2. The user's debt (`details[_id].debt`) would be 1.005 M as there's a .5 percent fee.
3. The contract's debt is 1M.
4. BTC price decrease by 20 percent
5. The liquidator tries to liquidate the position.
6. The liquidator repays 1.005 M and the contract tries to sub the debt by 1.005 M
7. The transaction is reverted as `details[_id].debt -= _usdm;` would raise exception.

inaccurate accounting would lead to serious issues. I consider this a high-risk issue.

Proof of Concept

This is a web3.py script that a liquidation may fail.

```
deposit_amount = 10**18
big_deposit = deposit_amount * 100000
minter.functions.mint(user, big_deposit).transact()

dai.functions.approve(vault.address, big_deposit + deposit_amount).tr

# create two positions
vault.functions.mint(user, zero_address).transact()
vault.functions.mint(user, zero_address).transact()

# # borrow max amount
vault.functions.increase(0, big_deposit, big_deposit, zero_address, '
vault.functions.increase(1, deposit_amount, deposit_amount, zero_addr

vault_debt = vault.functions.debts().call()

# ## This would clear out all debt in vault.
repay_amount = vault_debt + 10**18
usdm.functions.approve(vault.address, repay_amount).transact()

vault.functions.repay(0, repay_amount).transact()
```

```
print('debt left:', vault.functions.debts().call())
# ## All the positions would not be liquidated from now on

dai_price = cssr_factory.functions.getPrice(dai.address).call()
cssr_factory.functions.setPrice(dai.address, dai_price[0] // 10).tran

## this would revert
liquidator.functions.triggerLiquidation(dai.address, 1).transact()
```

Recommended Mitigation Steps

I believe this is a mistake. Recommend to check the contract to make sure `increasingDebt` is used consistently.

[H-02] `FeePoolV0.sol#distributeMochi()` will unexpectedly flush `treasuryShare`, causing the protocol fee cannot be properly accounted for and collected

Submitted by WatchPug

`distributeMochi()` will call `_buyMochi()` to convert `mochiShare` to Mochi token and call `_shareMochi()` to send Mochi to vMochi Vault and veCRV Holders. It wont touch the `treasuryShare`.

However, in the current implementation, `treasuryShare` will be reset to `0`. This is unexpected and will cause the protocol fee can not be properly accounted for and collected.

[`FeePoolV0.sol#L79` L95]

```
function _shareMochi() internal {
    IMochi mochi = engine.mochi();
    uint256 mochiBalance = mochi.balanceOf(address(this));
    // send Mochi to vMochi Vault
    mochi.transfer(
        address(engine.vMochi()),
        (mochiBalance * vMochiRatio) / 1e18
    );
    // send Mochi to veCRV Holders
```

```
        mochi.transfer(
            crvVoterRewardPool,
            (mochiBalance * (1e18 - vMochiRatio)) / 1e18
        );
        // flush mochiShare
        mochiShare = 0;
        treasuryShare = 0;
    }
```

Impact

Anyone can call `distributeMochi()` and reset `treasuryShare` to `0`, and then call `updateReserve()` to allocate part of the wrongfuly resetted `treasuryShare` to `mochiShare` and call `distributeMochi()`.

Repeat the steps above and the `treasuryShare` will be consumed to near zero, profits the vMochi Vault holders and veCRV Holders. The protocol suffers the loss of funds.

Recommendation

Change to:

```
    function _buyMochi() internal {
        IUSDM usdm = engine.usdm();
        address[] memory path = new address[](2);
        path[0] = address(usdm);
        path[1] = address(engine.mochi());
        usdm.approve(address(uniswapRouter), mochiShare);
        uniswapRouter.swapExactTokensForTokens(
            mochiShare,
            1,
            path,
            address(this),
            type(uint256).max
        );
        // flush mochiShare
        mochiShare = 0;
    }

    function _shareMochi() internal {
        IMochi mochi = engine.mochi();
        uint256 mochiBalance = mochi.balanceOf(address(this));
        // send Mochi to vMochi Vault
        mochi.transfer(
            address(engine.vMochi()),
            (mochiBalance * vMochiRatio) / 1e18
        );
        // send Mochi to veCRV Holders
        mochi.transfer(
            crvVoterRewardPool,
            (mochiBalance * (1e18 - vMochiRatio)) / 1e18
        );
    }
```

[ryuheimat (Mochi) confirmed]

[H-03] `ReferralFeePoolV0.sol#claimRewardAsMochi()` Array out of bound exception

Submitted by WatchPug, also found by pauliax

[`ReferralFeePoolV0.sol#L28`] [L42]

```
    function claimRewardAsMochi() external {
        IUSDM usdm = engine.usdm();
        address[] memory path = new address[](2);
        path[0] = address(usdm);
        path[1] = uniswapRouter.WETH();
        path[2] = address(engine.mochi());
        usdm.approve(address(uniswapRouter), reward[msg.sender]);
        // we are going to ingore the slippages here
        uniswapRouter.swapExactTokensForTokens(
            reward[msg.sender],
            1,
            path,
            address(this),
            type(uint256).max
        );
```

In `ReferralFeePoolV0.sol#claimRewardAsMochi()` , `path` is defined as an array of length 2 while it should be length 3.

As a result, at L33, an out-of-bound exception will be thrown and revert the transaction.

Impact

`claimRewardAsMochi()` will not work as expected so that all the referral fees cannot be claimed but stuck in the contract.

[ryuheimat (Mochi) confirmed]

[H-04] `registerAsset()` can overwrite `_assetClass` value

Submitted by gpersoon, also found by jonah1005 and leastwood

Impact

Everyone can call the function `registerAsset()` of MochiProfileV0.sol Assuming the liquidity for the asset is sufficient, `registerAsset()` will reset the _assetClass of an already registered asset to `AssetClass.Sigma`.

When the _assetClass is changed to `AssetClass.Sigma` then `liquidationFactor()`, `riskFactor()`, `maxCollateralFactor()`, `liquidationFee()` `keeperFee()` `maxFee()` will also return a different value. Then the entire vault will behave differently. The threshold for liquidation will also be different, possibly leading to a liquidation that isn't supposed to happen.

Recommended Mitigation Steps

Add the following in function `registerAsset()`:

```
    require(\_assetClass\[\_asset] ==0,"Already exists");
```

[ryuheimat (Mochi) confirmed]

[H-05] `debts` calculation is not accurate

Submitted by gpersoon

Impact

The value of the global variable `debts` in the contract `MochiVault.sol` is calculated in an inconsistent way.

In the function `borrow()` the variable `debts` is increased with a value excluding the fee. However in `repay()` and `liquidate()` it is decreased with the same value as `details\[\_id].debt` is decreased, which is including the fee.

This would mean that `debts` will end up in a negative value when all debts are repay-ed. Luckily the function `repay()` prevents this from happening.

In the meantime the value of `debts` isn't accurate. This value is used directly or indirectly in:

- `utilizationRatio()` , `stabilityFee()` `calculateFeeIndex()` of `MochiProfileV0.sol`
- `liveDebtIndex()` , `accrueDebt()` , `currentDebt()` of `MochiVault.sol`

This means the entire debt and claimable calculations are slightly off.

Proof of Concept

[`vault/MochiVault` sol]()

```
    function borrow(..)
    details\[\_id].debt = totalDebt; // includes the fee
    debts += \_amount;     // excludes the fee

    function repay(..)
    debts -= \_amount;\
    details\[\_id].debt -= \_amount;

    function liquidate(..)
    debts -= \_usdm;
    details\[\_id].debt -= \_usdm;
```

see [issue page]() for referenced code.

Recommended Mitigation Steps

In function `borrow()` : replace `debts += \_amount;` with `debts += totalDebt`

ryuheimat (Mochi) confirmed

[H-06] Referrer can drain `ReferralFeePoolV0`

Submitted by gzeon

Impact

function `claimRewardAsMochi` in `ReferralFeePoolV0.sol` did not reduce user reward balance, allowing referrer to claim the same reward repeatedly and thus draining the fee pool.

Proof of Concept

Did not reduce user reward balance at L28-47 in ReferralFeePoolV0.sol

Recommended Mitigation Steps

Add the following lines

rewards -= reward[msg.sender]; reward[msg.sender] = 0;

ryuheimat (Mochi) confirmed

[H-07] Liquidation will never work with non-zero discounts

Submitted by harleythedog

Impact

Right now, there is only one discount profile in the github repo: the " `NoDiscountProfile` " which does not discount the debt at all. This specific discount profile works correctly, but I claim that any other discount profile will result in liquidation never working.

Suppose that we instead have a discount profile where `discount()` returns any value strictly larger than 0. Now, suppose someone wants to trigger a liquidation on a position. First, `triggerLiquidation` will be called (within `DutchAuctionLiquidator.sol`). The variable "debt" is initialized as equal to `vault.currentDebt(\_nftId)` . Notice that `currentDebt(\_ndfId)` (within

`MochiVault.sol`) simply scales the current debt of the position using the `liveDebtIndex()` function, but there is no discounting being done within the function - this will be important.

Back within the `triggerLiquidation` function, the variable “collateral” is simply calculated as the total collateral of the position. Then, the function calls `vault.liquidate(\_nftId, collateral, debt)` , and I claim that this will never work due to underflow. Indeed, the liquidate function will first update the debt of the position (due to the `updateDebt(\_id)` modifier). The debt of the position is thus updated using lines 99-107 in `MochiVault.sol` . We can see that the `details\[\_id].debt` is updated in the exact same way as the calculations for `currentDebt(\_nftId)` , however, there is the extra subtraction of the `discountedDebt` on line 107.

Eventually we will reach line 293 in `MochiVault.sol` . However, since we discounted the debt in the calculation of `details\[\_id].debt` , but we did not discount the debt for the passed in parameter _usdm (and thus is strictly larger in value), line 293 will always error due to an underflow. In summary, any discount profile that actually discounts the debt of the position will result in all liquidations erroring out due to this underflow. Since no positions will be liquidatable, this represents a major flaw in the contract as then no collateral can be liquidated so the entire functionality of the contract is compromised.

Proof of Concept

- [Liquidate function in `MochiVault.sol`]
- [`triggerLiquidation` function in `DutchAuctionLiquidator.sol`]

Retracing the steps as I have described above, we can see that any call to `triggerLiquidation` will result in:

```
details\[\_id].debt -= \_usdm;
```

throwing an error since _usdm will be larger than `details\[\_id].debt` .

Recommended Mitigation Steps

An easy fix is to simply change:

```
details\[\_id].debt -= \_usdm;
```

to be:

```
details\[\_id].debt = 0;
```

as liquidating a position should probably just be equivalent to repaying all of the debt in the position.

Side Note: If there are no other discount profiles planned to be added other than " `NoDiscountProfile` ", then I would recommend deleting all of the discount logic entirely, since `NoDiscountProfile` doesn't actually do anything.

[ryuheimat (Mochi) confirmed]()

[H-08] Anyone can extend withdraw wait period by depositing zero collateral

Submitted by harleythedog, also found by WatchPug

Impact

In `MochiVault.sol` , the deposit function allows anyone to deposit collateral into any position. A malicious user can call this function with amount = 0, which would reset the amount of time the owner has to wait before they can withdraw their collateral from their position. This is especially troublesome with longer delays, as a malicious user would only have to spend a little gas to lock out all other users from being able to withdraw from their positions, compromising the functionality of the contract altogether.

Proof of Concept

the `deposit` function [here]()

Notice that calling this function with amount = 0 is not disallowed. This overwrites `lastDeposit\[\_id]` , extending the wait period before a withdraw is allowed.

Recommended Mitigation Steps

I would recommend adding:

```
require(amount > 0, "zero")
```

at the start of the function, as depositing zero collateral does not seem to be a necessary use case to support.

It may also be worthwhile to consider only allowing the owner of a position to deposit collateral.

ryuheimat (Mochi) confirmed

[H-09] treasury is vulnerable to sandwich attack

Submitted by jonah1005

Impact

There's a permissionless function `veCRVlock` in `MochiTreasury`. Since everyone can trigger this function, the attacker can launch a sandwich attack with flashloan to steal the funds. MochiTreasuryV0.sol#L73-L94

Attackers can possibly steal all the funds in the treasury. I consider this is a high-risk issue.

Proof of Concept

MochiTreasuryV0.sol#L73-L94

Here's an exploit pattern

1. Flashloan and buy CRV the uniswap pool
2. Trigger `veCRVlock()`
3. The treasury buys CRV at a very high price.
4. Sell CRV and pay back the loan.

Recommended Mitigation Steps

Recommend to add `onlyOwner` modifier.

ryuheimat (Mochi) confirmed

[H-10] Changing NFT contract in the `MochiEngine` would break the protocol

Submitted by jonah1005

Impact

`MochiEngine` allows the operator to change the NFT contract in MochiEngine.sol#L91-L93

All the vaults would point to a different NFT address. As a result, users would not be access their positions. The entire protocol would be broken.

IMHO, A function that would break the entire protocol shouldn't exist.

I consider this is a high-risk issue.

Proof of Concept

MochiEngine.sol#L91-L93

Recommended Mitigation Steps

Remove the function.

ryuheimat (Mochi) confirmed

[H-11] `treasuryShare` is Overwritten in `FeePoolV0._shareMochi()`

Submitted by leastwood

Impact

The `FeePoolV0.sol` contract accrues fees upon the liquidation of undercollaterised positions. These fees are split between treasury and `vMochi` contracts. However, when `distributeMochi()` is called to distribute `mochi` tokens to `veCRV` holders, both `mochiShare` and `treasuryShare` is flushed from the contract when there are still `usdm` tokens in the contract.

Proof of Concept

Consider the following scenario:

- The `FeePoolV0.sol` contract contains 100 `usdm` tokens at an exchange rate of 1:1 with `mochi` tokens.
- `updateReserve()` is called to set the split of `usdm` tokens such that `treasuryShare` has claim on 20 `usdm` tokens and `mochiShare` has claim on the other 80 tokens.
- A `veCRV` holder seeks to increase their earnings by calling `distributeMochi()` before `sendToTreasury()` has been called.
- As a result, 80 `usdm` tokens are converted to `mochi` tokens and locked in a curve rewards pool.
- Consequently, `mochiShare` and `treasuryShare` is set to `0` (aka flushed).
- The same user calls `updateReserve()` to split the leftover 20 `usdm` tokens between `treasuryShare` and `mochiShare`.
- `mochiShare` is now set to 16 `usdm` tokens.
- The above process is repeated to distribute `mochi` tokens to `veCRV` holders again and again.
- The end result is that `veCRV` holders have been able to receive all tokens that were intended to be distributed to the treasury.

`FeePoolV0.sol` L94

Tools Used

- Manual code review
- Discussions with the Mochi team.

Recommended Mitigation Steps

Consider removing the line in `FeePoolV0.sol` (mentioned above), where `treasuryShare` is flushed.

ryuheimat (Mochi) confirmed

[H-12] feePool is vulnerable to sandwich attack.

Submitted by jonah1005

Impact

There's a permissionless function `distributeMochi` in [FeePoolV0.sol L55-L62](). Since everyone can trigger this function, an attacker can launch a sandwich attack with flashloan to steal the funds.

The devs have mentioned this concern in the comment. An attacker can steal the funds with a flash loan attack.

Attackers can steal all the funds in the pool. I consider this is a high-risk issue.

Proof of Concept

[FeePoolV0.sol#L55-L62]()

Please refer to [yDai Incident]() to check the severity of a `harvest` function without slippage control.

Please refer to [Mushrooms-finance-theft]() to check how likely this kind of attack might happen.

Recommended Mitigation Steps

If the dev wants to make this a permissionless control, the contract should calculate a min return based on TWAP and check the slippage.

Comments:

[ryuheimat (Mochi) disputed]():

I think this is same case as https://github.com/code-423n4/2021-10-mochi-findings/issues/60

[ghoul-sol (judge) commented]():

The same attack, different part of the code. I'll keep them both.

[H-13] Tokens Can Be Stolen By Frontrunning `VestedRewardPool.vest()` and `VestedRewardPool.lock()`

Submitted by leastwood

Impact

The `VestedRewardPool.sol` contract is a public facing contract aimed at vesting tokens for a minimum of 90 days before allowing the recipient to withdraw their `mochi`. The `vest()` function does not utilise `safeTransferFrom()` to ensure that vested tokens are correctly allocated to the recipient. As a result, it is possible to frontrun a call to `vest()` and effectively steal a recipient's vested tokens. The same issue applies to the `lock()` function.

Proof of Concept

- `VestedRewardPool.sol#L36` L46
- `VestedRewardPool.sol#L54` L64

Tools Used

Manual code review Discussions with the Mochi team

Recommended Mitigation Steps

Ensure that users understand that this function should not be interacted directly as this could result in lost `mochi` tokens. Additionally, it might be worthwhile creating a single externally facing function which calls `safeTransferFrom()`, `vest()` and `lock()` in a single transaction.

ryuheimat (Mochi) confirmed

Medium Risk Findings (15)

[M-01] liquidation factor < collateral factor for Sigma type

Submitted by cmichel, also found by gzeon

The `MochiProfileV0` defines liquidation and collateral factors for different asset types. For the `AssetClass.Sigma` type, the liquidation factor is *less* than the collateral factor:

```
    function liquidationFactor(address _asset)
        public
        view
        override
        returns (float memory)
    {
        AssetClass class = assetClass(_asset);
        if (class == AssetClass.Sigma) { // } else if (class == AssetClas
            return float({numerator: 40, denominator: 100});
        }
    }

    function maxCollateralFactor(address _asset)
        public
        view
        override
        returns (float memory)
    {
        AssetClass class = assetClass(_asset);
        if (class == AssetClass.Sigma) {
            return float({numerator: 45, denominator: 100});
        }
    }
```

This means that one can take a loan of up to 45% of their collateral but then immediately gets liquidated as the liquidation factor is only 40%. There should always be a buffer between these such that taking the max loan does not immediately lead to liquidations:

A safety buffer is maintained between max CF and LF to protect users against liquidations due to normal volatility. [Docs]

Recommended Mitigation Steps

The max collateral factor for the Sigma type should be higher than its liquidation factor.

[ryuheimat (Mochi) confirmed]

[M-02] `regerralFeePool` is vulnerable to MEV searcher

Submitted by jonah1005, also found by cmichel

Impact

`claimRewardAsMochi` in the `ReferralFeePoolV0` ignores slippage. This is not a desirable design. There are a lot of MEV searchers in the current network. Swapping assets with no slippage control would get rekted. Please refer to https://github.com/flashbots/pm.

Given the current state of the Ethereum network, users would likely be sandwiched. I consider this is a high-risk issue.

Proof of Concept

ReferralFeePoolV0.sol#L28-L48 Please refer to Mushrooms Finance Theft of Yield Bug Fix Postmortem | by Immunefi | Immunefi | Medium to see a possible attack pattern.

Recommended Mitigation Steps

I recommend adding `minReceivedAmount` as a parameter.

```
    function claimRewardAsMochi(uint256 _minReceivedAmount) external {
        // original logic here
        require(engine.mochi().balanceOf(address(this)) > _minReceivedAmo
        engine.mochi().transfer(
            msg.sender,
            engine.mochi().balanceOf(address(this))
        );
    }
```

Also, the front-end should calculate the min amount with the current price.

ryuheimat (Mochi) confirmed

[M-03] A malicious user can potentially escape liquidation by creating a dust amount position and trigger the liquidation by themself

Submitted by WatchPug, also found by jonah1005

In the current implementation, a liquidated position can be used for depositing and borrowing again.

However, if there is a liquidation auction ongoing, even if the position is now `liquidatable`, the call of `triggerLiquidation()` will still fail.

The liquidator must `settleLiquidation` first.

If the current auction is not profitable for the liquidator, say the value of the collateral can not even cover the gas cost, the liquidator may be tricked and not liquidate the new loan at all.

Considering if the liquidator bot is not as small to handle this situation (take the profit of the new liquidation and the gas cost loss of the current auction into consideration), a malicious user can create a dust amount position trigger the liquidation by themself.

Since the collateral of this position is so small that it can not even cover the gas cost, liquidators will most certainly ignore this auction.

The malicious user will then deposit borrow the actual loan.

When this loan becomes `liquidatable`, liquidators may:

1. confuse the current dust auction with the `liquidatable` position;
2. unable to proceed with such a complex liquidation.

As a result, the malicious user can potentially escape liquidation.

Recommendation

Consider making liquidated positions unable to be used (for depositing and borrowing) again.

ryuheimat (Mochi) confirmed

[M-04] Unchecked ERC20 transfer calls

Submitted by loop, also found by cmichel, defsec, gzeon, leastwood, nikitastupin, pants, and WatchPug

ERC20 `transfer` and `transferFrom` calls normally return `true` on a succesful transfer. In DutchAuctionLiquidator the call `asset.transfer(msg.sender, _collateral);` is made. `asset` refers to whichever ERC20 asset is used for the vault of that auction. If `asset` is an ERC20 token which does not comply with the EIP-20 standard it might return `false` on a failed transaction rather than revert. In this case it would count as a valid transaction even though it is not. If a vault would be making use of USDT the transfer call would always revert as USDT returns `void` on transfers.

There are a few more transfer(From) calls which are unchecked, these are however all on a predetermined asset (mochi, usdM and crv) and unlikely to cause problems.

Proof of Concept

See issue page for referenced code.

Tools Used

Slither

Recommended Mitigation Steps

In other contracts the functions `cheapTransfer` and `cheapTransferFrom` are used which are part of the mochifi cheapERC20 library. These functions do check for a return value and could be used rather than `transfer` and `transferFrom`.

Comments:

ryuheimat (Mochi) confirmed:

`transferFrom` and `transfer` functions are used for mochi and usdm tokens which are standard EIP-20 tokens.

[M-05] Chainlink's `latestRoundData` might return stale or incorrect results

Submitted by nikitastupin, also found by cmichel, defsec, leastwood, and WatchPug

Proof of Concept

[`ChainlinkAdapter.sol` L49]

The `ChainlinkAdapter` calls out to a Chainlink oracle receiving the `latestRoundData()` . If there is a problem with Chainlink starting a new round and finding consensus on the new value for the oracle (e.g. Chainlink nodes abandon the oracle, chain congestion, vulnerability/attacks on the chainlink system) consumers of this contract may continue using outdated stale or incorrect data (if oracles are unable to submit no new round is started).

Recommended Mitigation Steps

Recommend adding the following checks:

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

References

- https://consensys.net/diligence/audits/2021/09/fei-protocol-v2-phase-1/#chainlinkoraclewrapper-latestrounddata-might-return-stale-results
- https://github.com/code-423n4/2021-05-fairside-findings/issues/70

ryuheimat (Mochi) confirmed

[M-06] Debt accrual is path-dependant and inaccurate

Submitted by cmichel

The total `debt` in `MochiVault.accrueDebt` increases by the current `debt` times the debt index growth. This is correct but the total `debt` is then *reduced* again by the calling *user's* discounted debt, meaning, the total debt depends on which specific user performs the debt accrual.

This should not be the case.

POC

Assume we have a total debt of `2000`, two users A and B, where A has a debt of 1000, and B has a debt of 100. The (previous) `debtIndex = 1.0` and accruing it now would increase it to `1.1`.

There’s a difference if user A or B first does the accrual.

User A accrues first

User A calls `accrueDebt`: `increased = 2000 * 1.1/1.0 - 2000 = 200`. Thus `debts` is first set to `2200`. The user’s `increasedDebt = 1000 * 1.1 / 1.0 - 1000 = 100` and assume a discount of `10%`, thus `discountedDebt = 100 * 10% = 10`. Then `debts = 2200 - 10 = 2190`.

The next accrual will work with a total debt of `2190`.

User B accruess first

User B calls `accrueDebt`: `increased = 2000 * 1.1/1.0 - 2000 = 200`. Thus `debts` is first set to `2200`. The user’s `increasedDebt = 100 * 1.1 / 1.0 - 100 = 10` and assume a discount of `10%`, thus `discountedDebt = 10 * 10% = 1`. Then `debts = 2200 - 1 = 2199`.

The next accrual will work with a total debt of `2199`, leading to more debt overall.

Impact

The total debt of a system depends on who performs the accruals which should ideally not be the case. The discrepancy compounds and can grow quite large if a whale always does the accrual compared to someone with almost no debt or no discount.

Recommended Mitigation Steps

Don’t use the discounts or track the weighted average discount across all users that is subtracted from the increased total debt each time, i.e., reduce it by the discount of **all users** (instead of current caller only) when accruing to correctly track the debt.

ryuheimat (Mochi) confirmed

[M-07] Changing engine.nft contract breaks vaults

Submitted by cmichel

Governance can change the `engine.nft` address which is used by vaults to represent collateralized debt positions (CDP). When minting a vault using `MochiVault.mint` the address returned ID will be used and overwrite the state of an existing debt position and set its status to `Idle` .

Impact

Changing the NFT address will allow overwriting existing CDPs.

Recommended Mitigation Steps

Disallow setting a new NFT address. or ensure that the new NFT's IDs start at the old NFT's IDs.

ryuheimat (Mochi) confirmed

[M-08] `UniswapV2/SushiwapLPAdapter` update the wrong token

Submitted by cmichel

The `UniswapV2LPAdapter/SushiswapV2LPAdapter.update` function retrieves the `underlying` from the LP token pair (`_asset`) but then calls `router.update(_asset, _proof)` which is the LP token itself again. This will end up with the router calling this function again recursively.

Impact

This function fails as there's an infinite recursion and eventually runs out of gas.

Recommendation

The idea was most likely to update the `underlying` price which is used in `_getPrice` as `uint256 eAvg = cssr.getExchangeRatio(_underlying, weth);` .

Call `router.update(underlying, _proof)` instead. Note that the `_proof` does not necessarily update the `underlying <> WETH` pair, it could be any `underlying <> keyAsset` pair.

ryuheimat (Mochi) confirmed

[M-09] `UniswapV2TokenAdapter` does not support Sushiswap-only assets

Submitted by cmichel

The `UniswapV2TokenAdapter.supports` function calls its `aboveLiquidity` function which returns the UniswapV2 liquidity if the pair exists. If this is below `minimumLiquidity`, the `supports` function will return `false`.

However, it could be that the `Sushiswap` pair has lots of liquidity and could be used.

```
    try uniswapCSSR.getLiquidity(_asset, _pairedWith) returns (
                uint256 liq
            ) {
        float memory price = cssrRouter.getPrice(_pairedWith);
        // @audit this returns early. if it's false it should check sushi
        return convertToValue(liq, price) >= minimumLiquidity;
    } catch {
        try sushiCSSR.getLiquidity(_asset, _pairedWith) returns (
            uint256 liq
        ) {
            float memory price = cssrRouter.getPrice(_pairedWith);
            return convertToValue(liq, price) >= minimumLiquidity;
        } catch {
            return false;
        }
    }
```

Impact

Suppose the `UniswapV2TokenAdapter` wants to be used as an adapter for a Sushiswap pool. An attacker creates a UniswapV2 pool for the same pair and does not provide liquidity. The `Router.setPriceSource` calls `UniswapV2TokenAdapter.supports` and returns false as the Uniswap liquidity is too low, without checking the Sushiswap liquidity.

Recommendation

In `aboveLiquidity` , if the UniswapV2 liquidity is *less* than the minimum liquidity, instead of returning, compare the Sushiswap liquidity against this threshold.

[ryuheimat (Mochi) confirmed]()

[M-10] griefing attack to block withdraws

Submitted by gpersoon

Impact

Every time you deposit some assets in the vault (via `deposit()` of `MochiVault.sol`) then "lastDeposit[_id]" is set to `block.timestamp` . The modifier `wait()` checks this value and makes sure you cannot withdraw for " `delay()` " blocks. The default value for `delay()` is 3 minutes.

Knowing this delay you can do a griefing attack: On chains with low gas fees: every 3 minutes deposit a tiny amount for a specific NFT-id (which has a large amount of assets). On chains with high gas fees: monitor the mempool for a `withdraw()` transaction and frontrun it with a `deposit()`

This way the owner of the NFT-id can never withdraw the funds.

Proof of Concept

- [`MochiVault.sol#L47` L54]()
- [`vault/MochiVault.sol` L171]()
- [`profile/MochiProfileV0.sol` L33]()

Recommended Mitigation Steps

Create a mechanism where you only block the withdraw of recently deposited funds

[ryuheimat (Mochi) confirmed]():

Will update deposit function to allow only NFT owner to deposit

[M-11] borrow function will borrow max cf when trying to borrow > cf

Submitted by gzeon

Impact

Borrow function in `MochiVault` will borrow to max cf when trying to borrow > cf instead of revert with ">cf" as specified in the supplied test. The difference in behavior may cause user to borrow at dangerous collateral level, and receive less than the amount requested.

Proof of Concept

- `MochiVault` .sol

Recommended Mitigation Steps

Revert if `details\[\_id].debt + \_amount > maxMinted` with ">cf"

ryuheimat (Mochi) conirmed

[M-12] anyone can create a vault by directly calling the factory

Submitted by jonah1005

Impact

In MochiVaultFactory.sol#L26-L37, there's no permission control in the `vaultFactory`. Anyone can create a vault. The transaction would be reverted when the government tries to deploy such an asset.

As the protocol checks whether the vault is a valid vault by comparing the contract's address with the computed address, the protocol would recognize the random vault as a valid one.

I consider this is a medium-risk issue.

Proof of Concept

Here's a web3.py script to trigger the bug.

```
vault_factory.functions.deployVault(usdt.address).transact()
## this tx would be reverted
profile.functions.registerAssetByGov([usdt.address], [3]).transact()
```

Recommended Mitigation Steps

Recommend to add a check.

```
    require(msg.sender == engine, "!engine");
```

[ryuheimat (Mochi) confirmed](#)

[M-13] Improper Validation Of `create2` Return Value

Submitted by leastwood

Impact

The `BeaconProxyDeployer.deploy()` function is used to deploy lightweight proxy contracts that act as each asset's vault. The function does not revert properly if there is a failed contract deployment or revert from the `create2` opcode as it does not properly check the returned address for bytecode. The `create2` opcode returns the expected address which will never be the zero address (as is what is currently checked).

Proof of Concept

- [`BeaconProxyDeployer.sol` L31](#)

Tools Used

- Manual code review
- Discussions with the Mochi team
- Discussions with library dev

Recommended Mitigation Steps

The recommended mitigation was to update `iszero(result)` to `iszero(extcodesize(result))` in the line mentioned above. This change has already been made in the corresponding library which can be found [here](#), however, this needs to also be reflected in Mochi's contracts.

[ryuheimat (Mochi) confirmed](#)

[M-14] `MochiTreasuryV0.withdrawLock()` Is Callable When Locking Has Been Toggled

Submitted by leastwood

Impact

`withdrawLock()` does not prevent users from calling this function when locking has been toggled. As a result, withdraws may be made unexpectedly.

Proof of Concept

- `MochiTreasuryV0.sol#L40` L42

Tools Used

Manual code review

Recommended Mitigation Steps

Consider adding `require(lockCrv, "!lock");` to `withdrawLock()` to ensure this function is not called unexpectedly. Alternatively if this is intended behaviour, it should be rather checked that the lock has not been toggled, otherwise users could maliciously relock tokens.

ryuheimat (Mochi) confirmed

[M-15] `MochiTreasuryV0.sol` Is Unusable In Its Current State

Submitted by leastwood

Impact

`MochiTreasuryV0.sol` interacts with Curve's voting escrow contract to lock tokens for 90 days, where it can be later withdrawn by the governance role. However, `VotingEscrow.vy` does not allow contracts to call the following functions; `create_lock()`, `increase_amount()` and `increase_unlock_time()`. For these functions, `msg.sender` must be an EOA account or an approved smart wallet. As a result, any attempt to lock tokens will fail in `MochiTreasuryV0.sol`.

Proof of Concept

- `VotingEscrow.vy` L418

- `VotingEscrow.vy` L438
- `VotingEscrow.vy` L455

Tools Used

- Manual code review
- Discussions with the Mochi team

Recommended Mitigation Steps

Consider updating this contract to potentially use another escrow service that enables `msg.sender` to be a contract. Alternatively, this escrow functionality can be replaced with an internal contract which holds `usdm` tokens instead, removing the need to convert half of the tokens to Curve tokens. Holding Curve tokens for a minimum of 90 days may overly expose the Mochi treasury to Curve token price fluctuations.

ryuheimat (Mochi) confirmed

Low Risk Findings (10)

- [L-01] `MochiVault.flashFee()` May Truncate Result *Submitted by leastwood*
- [L-02] `FeePoolV0.sol` Lack of input validation *Submitted by WatchPug, also found by cmichel and pauliax*
- [L-03] Minor precision loss *Submitted by WatchPug*
- [L-04] Key currencies can be double counted *Submitted by cmichel*
- [L-05] Mochi fees can be accidentally burned *Submitted by cmichel*
- [L-06] Flashloan fee griefing attack for existing approvals *Submitted by cmichel*
- [L-07] Unsafe `int256` casts in `accrueDebt` *Submitted by cmichel*
- [L-08] Unchecked low level call *Submitted by loop, also found by cmichel*
- [L-09] `UniswapV2CSSR` assumes data was observed when block state was inserted *Submitted by cmichel*
- [L-10] FRONT-RUNNABLE INITIALIZERS *Submitted by defsec, also found by pants*

Non-Critical Findings (18)

- [N-01] Missing events for governor only functions that change critical parameters *Submitted by defsec, also found by hyh, leastwood, nikitastupin, pants, and WatchPug*
- [N-02] borrow function will underflow when total debt > creditCap *Submitted by gzeon*
- [N-03] Vault status is not set to Liquidated after liquidation *Submitted by gzeon, also found by cmichel, gpersoon, harleythedog, and WatchPug*
- [N-04] `MochiTreasuryV0.sol` Implements `receive()` Function With No Withdraw Mechanism *Submitted by leastwood*
- [N-05] `MochiTreasuryV0.claimOperationCost()` Writes State Variable After An External Call Is Made *Submitted by leastwood*
- [N-06] Mochi Protocol Is Lacking Extensive Test Coverage *Submitted by leastwood*
- [N-07] `flashLoan()` is Lacking Protections Against Reentrancy *Submitted by leastwood*
- [N-08] Lack of input validation of arrays *Submitted by gzeon, also found by WatchPug*
- [N-09] Typos *Submitted by WatchPug*
- [N-10] ERC20 approve method missing return value check *Submitted by defsec*
- [N-11] Not all functions of `DutchAuctionLiquidator.sol` check the auction state *Submitted by gpersoon*
- [N-12] Missing zero-address checks *Submitted by loop, also found by pants*
- [N-13] flashFee lack of precision *Submitted by pants*
- [N-14] Unable to deposit to liquidated vault as specified *Submitted by gzeon*
- [N-15] Misspelling: Collaterized should be Collateralized *Submitted by harleythedog*
- [N-16] Unlocked pragma version *Submitted by loop*
- [N-17] Comment typos *Submitted by yeOlde*
- [N-18] Open TODOs/questions *Submitted by yeOlde*

Gas Optimizations (33)

- [G-01] debts <= _amount *Submitted by pauliax*
- [G-02] Unchecked math *Submitted by pauliax*
- [G-03] Gas optimizations (code simplification, memory variables addition or removal) *Submitted by hyh, also found by 0x0x0x*
- [G-04] Cached length of arrays to avoid loading them repeadetly *Submitted by 0x0x0x, also found by WatchPug*
- [G-05] Initialization of `pair` can be done later to save gas *Submitted by WatchPug*
- [G-06] `VestedRewardPool.sol#checkClaimable()` Add `vesting[recipient].ends > 0` to the condition can save gas *Submitted by WatchPug*
- [G-07] `DutchAuctionLiquidator.sol#triggerLiquidation()` Adding precondition check can save gas *Submitted by WatchPug*
- [G-08] Variable `liquidated` in MochiVault is never used *Submitted by loop, also found by defsec, gzeon, harleythedog, and WatchPug*
- [G-09] Avoid unnecessary storage read can save gas *Submitted by WatchPug*
- [G-10] Simplify `sqrt()` can save gas *Submitted by WatchPug*
- [G-11] Declaring unnecessary immutable variables as constant can save gas *Submitted by WatchPug*
- [G-12] `MochiVault.sol` Remove redundant check can save gas *Submitted by WatchPug*
- [G-13] Save Gas With The Unchecked Keyword (MochiVault.sol) *Submitted by yeOlde, also found by WatchPug*
- [G-14] multiple inter-contract references in the same function *Submitted by jonah1005, also found by WatchPug*
- [G-15] Replace engine.nft().ownerOf(_id) with msg.sender in withdraw function *Submitted by harleythedog, also found by WatchPug*
- [G-16] Upgrade pragma to at least 0.8.4 *Submitted by defsec*
- [G-17] Gas Optimization on the Public Function *Submitted by defsec*

- [G-18] Gas optimization: Placement of require statements in MochiVault.sol *Submitted by gzeon, also found by harleythedog*
- [G-19] Gas optimization: Caching variables *Submitted by gzeon, also found by pauliax and yeOlde*
- [G-20] Gas optimization: Struct layout *Submitted by gzeon*
- [G-21] Gas optimization: Struct layout in `DutchAuctionLiquidator.sol` *Submitted by gzeon*
- [G-22] Unused storage variable in UsdmMinter.sol *Submitted by harleythedog*
- [G-23] Unnecessary require in settleLiquidation *Submitted by harleythedog*
- [G-24] Calling `sushiCSSR.getLiquidity` two times leads to increased gas cost without benefits *Submitted by nikitastupin*
- [G-25] Improve precision and gas costs in _shareMochi *Submitted by pauliax*
- [G-26] Cache the results of duplicate external calls *Submitted by pauliax*
- [G-27] Pack structs tightly *Submitted by pauliax*
- [G-28] Useless imports *Submitted by pauliax*
- [G-29] Duplicate math operations *Submitted by pauliax*
- [G-30] Duplicate check of supported token on flash loan *Submitted by pauliax*
- [G-31] Long Revert Strings *Submitted by yeOlde*
- [G-32] Reduce State Variable Use in VestedRewardPool.sol *Submitted by yeOlde*
- [G-33] Remove extra calls in updateReserve (FeePoolV0.sol) *Submitted by yeOlde*

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