yAxis contest
Findings & Analysis Report

2021-11-10

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 yAxis smart contract system written in Solidity. The code contest took place between September 9—September 15 2021.

Wardens

13 Wardens contributed reports to the yAxis code contest:

1. cmichel
2. jonah1005
3. 0xRajeev
4. 0xsanson
5. hickuphh3
6. WatchPug
    - jtp
    - ming
7. hrkrshnn
8. itsmeSTYJ
9. pauliax
10. defsec
11. gpersoon
12. verifyfirst

This contest was judged by AlexTheEntreprenerd.

C4 Warden gpersoon also contributed to mitigating issues after the contest closed, hence their handle appears in this report as a contributor to the yAxis repo, as well

as alongside their vulnerability reports.

Final report assembled by moneylegobatman and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 50 unique vulnerabilities and 88 total findings. All of the issues presented here are linked back to their original submissions.

Of these vulnerabilities, 10 received a risk rating in the category of HIGH severity, 15 received a risk rating in the category of MEDIUM severity, and 25 received a risk rating in the category of LOW severity.

C4 analysis also identified 11 non-critical recommendations and 27 gas optimizations.

Scope

The code under review can be found within the C4 yAxis code contest repository and is composed of 90 smart contracts written in the Solidity programming language and includes 4,933 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 (10)

[H-01] `Controller.setCap` sets wrong vault balance

Submitted by cmichel

The `Controller.setCap` function sets a cap for a strategy and withdraws any excess amounts (`_diff`). The vault balance is decreased by the entire strategy balance instead of by this `_diff` :

```
// @audit why not sub _diff?
_vaultDetails[_vault].balance = _vaultDetails[_vault].balance.su
```

Impact

The `_vaultDetails[_vault].balance` variable does not correctly track the actual vault balances anymore, it will usually **underestimate** the vault balance. This variable is used in `Controller.balanceOf()` , which in turn is used in `Vault.balance()` , which in turn is used to determine how many shares to mint / amount to receive when redeeming shares. If the value is less, users will lose money as they can redeem fewer tokens. Also, an attacker can `deposit` and will receive more shares than they should receive. They can then wait until the balance is correctly updated again and withdraw their shares for a higher amount than they deposited. This leads to the vault losing tokens.

Recommended Mitigation Steps

Sub the `_diff` instead of the `balance` : `_vaultDetails[_vault].balance = _vaultDetails[_vault].balance.sub(_diff);`

[Haz077 (yAxis) confirmed and patched](#):

> Already fixed in code-423n4/2021-09-yaxis#1

GalloDaSballo (judge) commented:

> Finding is valid, has been mitigated by sponsor as of 14 days ago

[H-02] set cap breaks vault's Balance

Submitted by jonah1005, also found by 0xsanson

Impact

In controller.sol's function `setCap`, the contract wrongly handles `_vaultDetails[_vault].balance`. While the balance should be decreased by the difference of strategies balance, it subtracts the remaining balance of the strategy. See Controller.sol L262-L278. `_vaultDetails[_vault].balance = _vaultDetails[_vault].balance.sub(_balance);`

This would result in `vaultDetails[_vault].balance` being far smaller than the strategy's value. A user would trigger the assertion at Controller.sol#475 and the fund would be locked in the strategy.

Though `setCap` is a permission function that only the operator can call, it's likely to be called and the fund would be locked in the contract. I consider this a high severity issue.

Proof of Concept

We can trigger the issue by setting the cap 1 wei smaller than the strategy's balance.

```
strategy_balance = strategy.functions.balanceOf().call()
controller.functions.setCap(vault.address, strategy.address, str

## this would be reverted
vault.functions.withdrawAll(dai.address).transact()
```

Tools Used

Hardhat

Recommended Mitigation Steps

I believe the dev would spot the issue in the test if `_vaultDetails[_vault].balance` is a public variable.

One possible fix is to subtract the difference of the balance.

```
uint previousBalance = IStrategy(_strategy).balanceOf();
_vaultDetails[_vault].balance.sub(previousBalance.sub(_amount));
```

transferAndCall (yAxis) confirmed and patched:

> Please review https://github.com/code-423n4/2021-09-yaxis/pull/1 to verify resolution.

GalloDaSballo (judge) commented:

> High risk vulnerability due to incorrect logic which can impact protocol functionality

> Sponsor has mitigated

[H-03] No safety check in `addToken`

Submitted by jonah1005, also found by hrkrshnn and 0xRajeev

Impact

There's no safety check in `Manager.sol addToken`. There are two possible cases that might happen.

1. One token being added twice in a Vault. Token would be counted doubly in the vault. Ref: Vault.sol#L293-L303. There would be two item in the array when querying `manager.getTokens(address(this));`.
2. A token first being added to two vaults. The value calculation of the first vault would be broken. As `vaults[_token] = _vault;` would point to the other vault.

Permission keys should always be treated cautiously. However, calling the same initialize function twice should not be able to destroy the vault. Also, as the protocol develops, there's likely that one token is supported in two vaults. The DAO may mistakenly add the same token twice. I consider this a high-risk issue.

Proof of Concept

Adding same token twice would not raise any error here.

```
manager.functions.addToken(vault.address, dai.address).trans
manager.functions.addToken(vault.address, dai.address).trans
```

Tools Used

Hardhat

Recommended Mitigation Steps

I recommend to add two checks

```
require(vaults[_token] == address(0));
bool notFound = True;
for(uint256 i; i < tokens[_vault].length; i++) {
    if (tokens[_vault] == _token) {
        notFound = False;
    }
}
require(notFound, "duplicate token");
```

transferAndCall (yAxis) confirmed and patched:

> Please review https://github.com/code-423n4/2021-09-yaxis/pull/2 to verify resolution.

GalloDaSballo (judge) commented:

> Lack of check for duplicates can cause undefined behaviour, sponsor mitigated by adding a require check

[H-04] Controller does not raise an error when there's insufficient liquidity

Submitted by jonah1005, also found by 0xRajeev and WatchPug

Impact

When a user tries to withdraw the token from the vault, the vault would withdraw the token from the controller if there's insufficient liquidity in the vault. However, the controller does not raise an error when there's insufficient liquidity in the controller/ strategies. The user would lose his shares while getting nothing.

An MEV searcher could apply this attack on any withdrawal. When an attacker found an unconfirmed tx that tries to withdraw 1M dai, he can do such sandwich attack.

1. Deposits USDC into the vault.
2. Withdraw all dai left in the vault/controller/strategy.
3. Place the vitims tx here. The victim would get zero dai while burning 1 M share. **This would pump the share price.**
4. Withdraw all liquidity.

All users would be vulnerable to MEV attackers. I consider this is a high-risk issue.

Proof of Concept

Here's web3.py script to reproduce the issue.

```
deposit_amount = 100000 * 10**18
user = w3.eth.accounts[0]
get_token(dai, user, deposit_amount)
dai.functions.approve(vault.address, deposit_amount + margin_dep
vault.functions.deposit(dai.address, deposit_amount).transact()
vault.functions.withdrawAll(usdt.address).transact()

#
print("usdt amount: ", usdt.functions.balanceOf(user).call())
```

Recommended Mitigation Steps

There are two issues involved. First, users pay the slippage when they try to withdraw. I do not find this fair. Users have to pay extra gas to withdraw liquidity from strategy, convert the token, and still paying the slippage. I recommend writing a view function for the frontend to display how much slippage the user has to pay (Controler.sol L448-L479).

Second, the controller does not revert the transaction there's insufficient liquidity (Controller.sol#L577-L622).

Recommend to revert the transaction when `_amount` is not equal to zero after the loop finishes.

GainsGoblin (yAxis) acknowledged

GalloDaSballo (judge) commented:

> Agree with warden finding, this shows the path for an attack that is based on the Vault treating all tokens equally Since the finding shows a specific attack, the finding is unique

> Recommend the sponsor mitigates Single Sided Exposure risks to avoid this attack

BobbyYaxis (yAxis) noted:

> We have mitigated by deploying vaults that only accept the Curve LP token itself used in the strategy. There is no longer an array of tokens accepted. E.g Instead of a wBTC vault, we have a renCrv vault. Or instead of 3CRV vault, we have a mimCrv vault. The strategy want token = the vault token.

[H-05] Vault treats all tokens exactly the same that creates (huge) arbitrage opportunities.

Submitted by jonah1005, also found by cmichel and itsmeSTYJ

Impact

The v3 vault treats all valid tokens exactly the same. Depositing 1M DAI would get the same share as depositing 1M USDT. User can withdraw their share in another token.

Though there's `withdrawalProtectionFee` (0.1 percent), the vault is still a no slippage stable coin exchange.

Also, I notice that 3crv*token is added to the vault in the test. Treating 3crv*token and all other stable coins the same would make the vault vulnerable to flashloan attack. 3crv_token is an lp token and at the point of writing, the price of it is 1.01. The arbitrage space is about 0.8 percent and makes the vault vulnerable to flashloan attacks.

Though the team may not add crv_token and dai to the same vault, its design makes the vault vulnerable. Strategies need to be designed with super caution or the vault would be vulnerable to attackers.

Given the possibility of a flashloan attack, I consider this a high-risk issue.

Proof of Concept

The issue locates at the deposit function (Vault.sol#L147-L180). The share is minted according to the calculation here

```
_shares = _shares.add(_amount);
```

The share is burned at Vault.sol L217

```
uint256 _amount = (balance().mul(_shares)).div(totalSupply());
```

Here's a sample exploit in web3.py.

```
deposit_amount = 100000 * 10**6
user = w3.eth.accounts[0]
get_token(usdt, user, deposit_amount)
usdt.functions.approve(vault.address, deposit_amount).transact()
vault.functions.deposit(usdt.address, deposit_amount).transact()
vault.functions.withdrawAll(t3crv.address).transact()
# user can remove liquiditiy and get the profit.
```

Tools Used

Hardhat

Recommended Mitigation Steps

Given the protocols' scenario, I feel like we can take iearn token's architect as a Ref. yDdai

yDai handles multiple tokens (cDai/ aDai/ dydx/ fulcrum). Though four tokens are pretty much the same, the contract still needs to calculate the price of each token.

Or, creating a vault for each token might be an easier quick fix.

Haz077 (yAxis) acknowledged

transferAndCall (yAxis) commented:

> The design of the v3 vaults is to intentionally assume that all allowed tokens are of equal value. I do not see us enabling the 3CRV token in our Vault test, though if we did, that doesn't mean we would in reality. Using a separate vault per token is an architecture we want to avoid.

GalloDaSballo (judge) commented:

> Anecdotal example from warden makes sense.

> Assuming that 3CRV is worth the same as a stablecoin is in principle very similar to assuming that a swap between each stable on curve will yield a balanced trade

> This reminds me of the Single Sided Exposure Exploit that Yearn Suffered, and would recommend mitigating by checking the virtual_price on the 3CRV token

GalloDaSballo (judge) commented:

> TODO: Review and check duplicates, need to read yaxis vault code and use cases before can judge this

GalloDaSballo (judge) commented:

> After reviewing the code and the submissions, I have to agree that the vault creates arbitrage opportunities, since it heavily relies on 3CRV you may want to use it's `virtual_price` as a way to mitigate potential exploits, alternatively you can roll your own pricing oracle solution

> Not mitigating this opportunity means that an attacker will exploit it at the detriment of the depositors

BobbyYaxis (yAxis) noted:

> We have mitigated by deploying vaults that only accept the Curve LP token itself used in the strategy. There is no longer an array of tokens accepted. E.g Instead of a wBTC vault, we have a renCrv vault. Or instead of 3CRV vault, we have a mimCrv vault. The strategy want token = the vault token.

[H-06] earn results in decreasing share price

Submitted by jonah1005

Impact

For a dai vault that pairs with `NativeStrategyCurve3Crv`, every time `earn()` is called, shareholders would lose money. (about 2%)

There are two issues involved. The `Vault` contract and the `controller` contract doesn't handle the price difference between the want token and other tokens.

At Vault.sol L293, when a vault calculates its value, it sums up all tokens balance. However, when the controller calculates vaults' value (at Controller.sol L410-L436), it only adds the amount of `strategy.want` it received. (in this case, it's t3crv).

Under the current design, users who deposit dai to the vault would not get yield. Instead, they would keep losing money. I consider this a high-risk issue

Proof of Concept

I trigger the bug with the following web3.py script:

```
previous_price = vault.functions.getPricePerFullShare().call()
vault.functions.available(dai.address).call()
vault.functions.earn(dai.address, strategy.address).transact()
current_price = vault.functions.getPricePerFullShare().call()
print(previous_price)
print(current_price)
```

Tools Used

Hardhat

Recommended Mitigation Steps

The protocol should decide what the balance sheet in each contract stands for and make it consistent in all cases. Take, for example, if `_vaultDetails[_vault].balance;` stands for the amount of 'want' token the vault owns, there shouldn't exist two different want in all the strategies the vault has. Also, when the vault queries controllers `function balanceOf()`, they should always multiply it by the price.

transferAndCall (yAxis) acknowledged

gpersoon commented:

> I think this is also related to the underlying problem that all coins are assumed to have the same value. See also #2, #8 and #158

GalloDaSballo (judge) commented:

> Agree with wardens finding and acknowledge it's similitude with other issues

> Personally this is a different vulnerability that can be solved by solving the same underlying problem

> Marking this as unique finding as it's a specific exploit the protocol could face

BobbyYaxis (yAxis) noted:

> We have mitigated by deploying vaults that only accept the Curve LP token itself used in the strategy. There is no longer an array of tokens accepted. E.g Instead of a wBTC vault, we have a renCrv vault. Or instead of 3CRV vault, we have a mimCrv vault. The strategy want token = the vault token.

[H-07] `Vault.balance()` mixes normalized and standard amounts

Submitted by cmichel

The `Vault.balance` function uses the `balanceOfThis` function which scales ("normalizes") all balances to 18 decimals.

```
    for (uint8 i; i < _tokens.length; i++) {
        address _token = _tokens[i];
        // everything is padded to 18 decimals
        _balance = _balance.add(_normalizeDecimals(_token, IERC20(_t
    }
```

Note that `balance()`'s second term `IController(manager.controllers(address(this))).balanceOf()` is not normalized. The code is adding a non-normalized amount (for example 6 decimals only for USDC) to a normalized (18 decimals).

Impact

The result is that the `balance()` will be under-reported. This leads to receiving wrong shares when `deposit`ing tokens, and a wrong amount when redeeming `tokens`.

Recommended Mitigation Steps

The second term `IController(manager.controllers(address(this))).balanceOf()` must also be normalized before adding it.

`IController(manager.controllers(address(this))).balanceOf()` uses `_vaultDetails[msg.sender].balance` which directly uses the raw token amounts which are not normalized.

GainsGoblin (yAxis) acknowledged

GalloDaSballo (judge) commented:

> `balance` and `balanceOfThis` mixes the usage of decimals by alternatingly using `_normalizeDecimals` This can break accounting as well as create opportunities for abuse A consistent usage of `_normalizeDecimals` would mitigate

BobbyYaxis (yAxis) noted:

> Mitigated in PR 114: https://github.com/yaxis-project/metavault/pull/114/commits/b3c0405640719aa7d43560f4b4b910b7ba88170b

[H-08] `Vault.withdraw` mixes normalized and standard amounts

Submitted by cmichel, also found by hickuphh3 and jonah1005

The `Vault.balance` function uses the `balanceOfThis` function which scales ("normalizes") all balances to 18 decimals.

```
    for (uint8 i; i < _tokens.length; i++) {
        address _token = _tokens[i];
        // everything is padded to 18 decimals
        _balance = _balance.add(_normalizeDecimals(_token, IERC20(_t
    }
```

Note that `balance()`'s second term `IController(manager.controllers(address(this))).balanceOf()` is not normalized, but it must be.

This leads to many issues through the contracts that use `balance` but don't treat these values as normalized values. For example, in `Vault.withdraw`, the computed `_amount` value is normalized (in 18 decimals). But the `uint256 _balance = IERC20(_output).balanceOf(address(this));` value is not normalized but compared to the normalized `_amount` and even subtracted:

```
    // @audit compares unnormalzied output to normalized output
    if (_balance < _amount) {
        IController _controller = IController(manager.controllers(ac
        // @audit cannot directly subtract unnormalized
        uint256 _toWithdraw = _amount.sub(_balance);
        if (_controller.strategies() > 0) {
            _controller.withdraw(_output, _toWithdraw);
        }
        uint256 _after = IERC20(_output).balanceOf(address(this));
        uint256 _diff = _after.sub(_balance);
        if (_diff < _toWithdraw) {
            _amount = _balance.add(_diff);
        }
    }
```

Impact

Imagine in `withdraw`, the `output` is USDC with 6 decimals, then the normalized `_toWithdraw` with 18 decimals (due to using `_amount`) will be a huge number and attempt to withdraw an inflated amount. An attacker can steal tokens this way by withdrawing a tiny amount of shares and receive an inflated USDC or USDT amount (or any `_output` token with less than 18 decimals).

Recommended Mitigation Steps

Whenever using anything involving `vault.balanceOfThis()` or `vault.balance()` one needs to be sure that any derived token amount needs to be denormalized again before using them.

GalloDaSballo (judge) commented:

> An inconsistent usage of `_normalizeDecimals` will cause accounting issues and potentially paths for an exploit

BobbyYaxis (yAxis) noted:

> Mitigated in PR 114: https://github.com/yaxis-project/metavault/pull/114/commits/b3c0405640719aa7d43560f4b4b910b7ba88170b

[H-09] `removeToken` would break the vault/protocol.

Submitted by jonah1005

Impact

There's no safety check in Manager.sol's `removeToken`. Manager.sol#L454-L487

1. The token would be locked in the original vault. Given the current design, the vault would keep a ratio of total amount to save the gas. Once the token is removed at manager contract, these token would lost.
2. Controller's `balanceOf` would no longer reflects the real value. Controller.sol#L488-L495 While `_vaultDetails[msg.sender].balance;` remains the same, user can nolonger withdraw those amount.
3. Share price in the vault would decrease drastically. The share price is calculated as `totalValue / totalSupply` Vault.sol#L217. While the `totalSupply` of the share remains the same, the total balance has drastically decreased.

Calling `removeToken` way would almost break the whole protocol if the vault has already started. I consider this is a high-risk issue.

Proof of Concept

We can see how the vault would be affected with below web3.py script.

```
print(vault.functions.balanceOfThis().call())
print(vault.functions.totalSupply().call())
manager.functions.removeToken(vault.address, dai.address).transa
print(vault.functions.balanceOfThis().call())
print(vault.functions.totalSupply().call())
```

output

```
100000000000000000000000
100000000000000000000000
0
100000000000000000000000
```

Tools Used

Hardhat

Recommended Mitigation Steps

Remove tokens from a vault would be a really critical job. I recommend the team cover all possible cases and check all components' states (all vault/ strategy/ controller's state) in the test.

Some steps that I try to come up with that is required to remove TokenA from a vault.

1. Withdraw all tokenA from all strategies (and handle it correctly in the controller).
2. Withdraw all tokenA from the vault.
3. Convert all tokenA that's collected in the previous step into tokenB.
4. Transfer tokenB to the vault and compensate the transaction fee/slippage cost to the vault.

transferAndCall (yAxis) acknowledged:

> Removing a token is understood as a critical (and possibly nuclear) operation within this architecture. We knew we would have to first withdraw all of the identified token from strategies, but what was missed was converting that token to another (without withdrawing, as that would be too much centralization).

> Proposed method of resolution:

- Withdraw all tokenA from all strategies (this sends it to the vault)
- Swap tokenA for tokenB in the vault (requires implementing a new function to be called by the strategist)
- Remove the token via the Manager function

transferAndCall (yAxis) confirmed and patched:

> Please review https://github.com/code-423n4/2021-09-yaxis/pull/5 to check resolution.

GalloDaSballo (judge) commented:

> Removing a token can cause accounting errors, stuck funds and break some of the functionality

> Adding additional checks to prevent removing the token until all tokens have been migrated may be the simplest path forward

> Sponsor has mitigated by adding custom functionality, however it is up to them to enforce that the vault has no token left before removing it, adding a couple extra checks may provide a guarantee against admin privileged abuses

[H-10] An attacker can steal funds from multi-token vaults

Submitted by WatchPug, also found by cmichel and jonah1005

The total balance should NOT be simply added from different tokens' tokenAmounts, considering that the price of tokens may not be the same.

Vault.sol L324

```
    function balanceOfThis()
        public
        view
        returns (uint256 _balance)
    {
        address[] memory _tokens = manager.getTokens(address(this));
        for (uint8 i; i < _tokens.length; i++) {
            address _token = _tokens[i];
            _balance = _balance.add(_normalizeDecimals(_token, IERC2
        }
    }
```

Controller.sol L396

```
    function harvestStrategy(
        address _strategy,
        uint256 _estimatedWETH,
        uint256 _estimatedYAXIS
    )
        external
```

```
        override
        notHalted
        onlyHarvester
        onlyStrategy(_strategy)
    {
        uint256 _before = IStrategy(_strategy).balanceOf();
        IStrategy(_strategy).harvest(_estimatedWETH, _estimatedYAXIS
        uint256 _after = IStrategy(_strategy).balanceOf();
        address _vault = _vaultStrategies[_strategy];
        _vaultDetails[_vault].balance = _vaultDetails[_vault].balanc
        _vaultDetails[_vault].balances[_strategy] = _after;
        emit Harvest(_strategy);
    }
```

Vault.sol L310

```
    /**
     * @notice Returns the total balance of the vault, including str
     */
    function balance()
        public
        view
        override
        returns (uint256 _balance)
    {
        return balanceOfThis().add(IController(manager.controllers(a
    }
```

Impact

An attacker can steal funds from multi-token vaults. Resulting in fund loss of all other users.

Proof of Concept

If there is a multi-token vault with 3 tokens: DAI, USDC, USDT, and their price in USD is now 1.05, 0.98, and 0.95. If the current balances are: 2M, 1M, and 0.5M.

An attacker may do the following steps:

1. Deposit 3M of USDT;

2. Withdraw 3M, receive 2M in DAI and 1M in USDC.

As 2M of DAI + 1M of USDC worth much more than 3M of USDT. The attacker will profit and all other users will be losing funds.

Recommended Mitigation Steps

Always consider the price differences between tokens.

BobbyYaxis (yAxis) acknowledged

GalloDaSballo (judge) commented:

> Fully agree with the finding, assuming price of tokens is the same exposes the Vault and all depositors to risk of Single Sided Exposure
>
> This risk has been exploited multiple times, notably in the Yearn Exploit
>
> The solution for for managing tokens with multiple values while avoiding being rekt is to have an index that ensures your LP Token maintains it's peg, curve's solution is called `virtual_price`
>
> Having a virtual price would allow to maintain the Vault Architecture, while mitigating exploits that directly use balances

BobbyYaxis (yAxis) noted:

> We have mitigated by deploying vaults that only accept the Curve LP token itself used in the strategy. There is no longer an array of tokens accepted. E.g Instead of a wBTC vault, we have a renCrv vault. Or instead of 3CRV vault, we have a mimCrv vault. The strategy want token = the vault token.

Medium Risk Findings (15)

[M-01] `VaultHelper` deposits don't work with fee-on transfer tokens

Submitted by cmichel, also found by 0xsanson

There are ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. Others are rebasing tokens that increase in value over time like Aave's aTokens (`balanceOf` changes over time).

Impact

The `VaultHelper`'s `depositVault()` and `depositMultipleVault` functions transfer `_amount` to `this` contract using `IERC20(_token).safeTransferFrom(msg.sender, address(this), _amount);`. This could have a fee, and less than `_amount` ends up in the contract. The next actual vault deposit using `IVault(_vault).deposit(_token, _amount);` will then try to transfer more than the `this` contract actually has and will revert the transaction.

Recommended Mitigation Steps

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. This is already done correctly in the `Vault.deposit` function.

GalloDaSballo (judge) commented:

> Agree with finding, checking actual balance of contract would mitigate vulnerability Additionally ensuring the protocol never uses rebasing or tokens with `feeOnTransfer` can be enough to mitigate

> The vulnerability can brick the protocol However it can be sidestepped by simply not using `feeOnTransfer` tokens Downgrading to medium

[M-02] ERC20 return values not checked

Submitted by cmichel, also found by defsec and jonah1005

The `ERC20.transfer()` and `ERC20.transferFrom()` functions return a boolean value indicating success. This parameter needs to be checked for success. Some tokens do **not** revert if the transfer failed but return `false` instead.

The `Manager.recoverToken` function does not check the return value of this function.

Impact

Tokens that don't actually perform the transfer and return `false` are still counted as a correct transfer. Furthermore, tokens that do not correctly implement the EIP20 standard, like USDT which does not return a success boolean, will revert.

Recommended Mitigation Steps

We recommend using OpenZeppelin's `SafeERC20` versions with the `safeTransfer` and `safeTransferFrom` functions that handle the return value check as well as non-standard-compliant tokens.

GalloDaSballo (judge) commented:

> Agree with finding, using a non reverting token can potentially cause issues to the protocol accounting

> Sponsor can check each token on a case by case basis, or simply use OpenZeppelin's `safeERC20`

[M-03] `Vault.withdraw` sometimes burns too many shares

Submitted by cmichel, also found by 0xsanson and 0xRajeev

The `Vault.withdraw` function attempts to withdraw funds from the controller if there are not enough in the vault already. In the case the controller could not withdraw enough, i.e., where `_diff < _toWithdraw`, the user will receive **less** output tokens than their fair share would entitle them to (the initial `_amount`).

```
    if (_diff < _toWithdraw) {
        // @audit burns too many shares for a below fair-share amour
        _amount = _balance.add(_diff);
    }
```

Impact

The withdrawer receives fewer output tokens than they were entitled to.

Recommended Mitigation Steps

In the mentioned case, the `_shares` should be recomputed to match the actual withdrawn `_amount` tokens:

```
    if (_diff < _toWithdraw) {
        _amount = _balance.add(_diff);
        // recompute _shares to burn based on the lower payout
        // should be something like this, better to cache balance()
        _shares = (totalSupply().mul(_amount)).div(_balance);
    }
```

Only these shares should then be burned.

uN2RVw5q commented:

> Duplicate of https://github.com/code-423n4/2021-09-yaxis-findings/issues/41 and https://github.com/code-423n4/2021-09-yaxis-findings/issues/136

GalloDaSballo (judge) commented:

> Agree with the finding
>
> Anytime the strategy incurs a loss during withdrawal, the person that triggered that withdrawal will get less for their shares than what they may expect.
>
> Since amount of shares is computed by checking balance in strategy, and controller enacts this withdrawal, adding a check in the controller to compare expected withdrawal vs actual shares received would be a clean way to mitigate

BobbyYaxis (yAxis) noted:

> We have mitigated by deploying vaults that only accept the Curve LP token itself used in the strategy. There is no longer an array of tokens accepted. E.g Instead of

> a wBTC vault, we have a renCrv vault. Or instead of 3CRV vault, we have a mimCrv vault. The strategy want token = the vault token.

[M-04] Adding asymmetric liquidity in `_addLiquidity` results in fewer LP tokens minted than what should be wanted

Impact

Because the call in `_addLiquidity` forwards the entire balances of the 3 stablecoins without checking the ratio. between the 3, less liquidity is minted than what should be wanted. Furthermore, an attacker can abuse this arbitrage the forwarded balances if the discrepancy is large enough.

For example, suppose the contract holds $10K each of usdc, usdt, dai. An attacker deposits $100K worth of DAI and get credited with $100K worth of shares in the protocol. Liquidity is added, but since the ratio is now skewed 11:1:1, a lot less liquidity is minted by the stableswap algorithm to the protocol. The attacker can now arbitrage the curve pool for an additional profit.

There doesn't even need to be an attacker, just an unbalanced amount of user deposits will also lead to lower liquidity minted.

Proof of Concept

- `NativeStrategyCurve3Crv.sol` L73

Recommended Mitigation Steps

Adding liquidity should probably be managed more manually, it should be added in equal proportion to the curve pool balances, not the contract balances.

gpersoon commented:

> Seems the same as #2

GalloDaSballo (judge) commented:

> Agree on the finding This finding claims that adding liquidity on Curve while treating each token to have the same weight is a surefire way to get less tokens than expected

> While #2 addresses a similar (IMO higher risk) vulnerability

> This finding shows how the vault can have a loss of value through how it deals with token accounting

> To me this is a unique finding, however am downgrading it to medium

BobbyYaxis (yAxis) noted:

> We have mitigated by deploying vaults that only accept the Curve LP token itself used in the strategy. There is no longer an array of tokens accepted. E.g Instead of a wBTC vault, we have a renCrv vault. Or instead of 3CRV vault, we have a mimCrv vault. The strategy want token = the vault token.

[M-05] Vault: Swaps at parity with swap fee = withdrawal fee

Submitted by hickuphh3

Impact

The vault treats all assets to be of the same price. Given that one can also deposit and withdraw in the same transaction, this offers users the ability to swap available funds held in the vault at parity, with the withdrawal protection fee (0.1%) effectively being the swap fee.

Due care and consideration should therefore be placed if new stablecoins are to be added to the vault (eg. algorithmic ones that tend to occasionally be off-peg), or when lowering the withdrawal protection fee.

Recommended Mitigation Steps

- Prevent users from depositing and withdrawing in the same transaction. This would help prevent potential flash loan attacks as well
- `setWithdrawalProtectionFee()` could have a requirement for the value to be non-zero. Zero withdrawal fee could be set in `setHalted()` whereby only

withdrawals will be allowed.

- Use price oracles to accurately calculate the shares to be transferred to users for deposits, or token amounts to be sent for withdrawals

GalloDaSballo (judge) commented:

> Agree with finding, this vault accounting can be used for arbitrage opportunities as tokens are treated at exact value while they may have imbalances in price

> This is not a duplicate as it's explaining a specific attack vector

GalloDaSballo (judge) commented:

> Also raising risk valuation as this WILL be used to extract value from the system

BobbyYaxis (yAxis) noted:

> We have mitigated by deploying vaults that only accept the Curve LP token itself used in the strategy. There is no longer an array of tokens accepted. E.g Instead of a wBTC vault, we have a renCrv vault. Or instead of 3CRV vault, we have a mimCrv vault. The strategy want token = the vault token.

[M-06] # Controller is vulnerable to sandwich attack

Submitted by jonah1005

Impact

The protocol frequently interacts with crv a lot. However, the contract doesn't specify the minimum return amount. Given the fact that there's a lot of MEV searchers, calling swap without specifying the minimum return amount really puts user funds in danger.

For example, controller's `withdrawAll` is designed to transfer all the funds in a strategy.Controller.sol#L360 The arbitrage space is enough for a searcher to sandwich this trade.

Proof of Concept

Manager.sol#L442-L452

Controller.sol#L273

Recommended Mitigation Steps

Always calculates an estimate return when calling to crv.

transferAndCall (yAxis) acknowledged

GalloDaSballo (judge) commented:

> Agree with finding, agree with severity as this allows to "leak value"

[M-07] Vault: Withdrawals can be frontrun to cause users to burn tokens without receiving funds in return

Submitted by hickuphh3

Impact

Let us assume either of the following cases:

1. The vault / protocol is to be winded down or migrated, where either the protocol is halted and `withdrawAll()` has been called on all active strategies to transfer funds into the vault.
2. There are 0 strategies. Specifically, `_controller.strategies() = 0`

Attempted withdrawals can be frontrun such that users will receive less, or even no funds in exchange for burning vault tokens. This is primarily enabled by the feature of having deposits in multiple stablecoins.

Proof of Concept

1. Assume `getPricePerFullShare()` of `1e18` (1 vault token = 1 stablecoin). Alice has 1000 vault tokens, while Mallory has 2000 vault tokens, with the vault holdings being 1000 USDC, 1000 USDT and 1000 DAI.
2. Alice attempts to withdraw her deposit in a desired stablecoin (Eg. USDC).

3. Mallory frontruns Alice's transaction and exchanges 1000 vault tokens for the targeted stablecoin (USDC). The vault now holds 1000 USDT and 1000 DAI.
4. Alice receives nothing in return for her deposit because the vault no longer has any USDC. `getPricePerFullShare()` now returns `2e18`.
5. Mallory splits his withdrawals evenly, by burning 500 vault tokens for 1000 USDT and the other 500 vault tokens for 1000 DAI.

Hence, Mallory is able to steal Alice's funds by frontrunning her withdrawal transaction.

Recommended Mitigation Steps

The withdrawal amount could be checked against `getPricePerFullShare()`, perhaps with reasonable slippage.

GainsGoblin (yAxis) commented:

> Duplicate of #28

GalloDaSballo (judge) commented:

> Disagree with duplicate label as this shows a Value Extraction, front-running exploit. Medium severity as it's a way to "leak value"

> This can be mitigated through addressing the "Vault value all tokens equally" issue

GainsGoblin (yAxis) commented:

> The issue is exactly the same as #28. Both issues present the exact same front-running example.

BobbyYaxis (yAxis) noted:

> We have mitigated by deploying vaults that only accept the Curve LP token itself used in the strategy. There is no longer an array of tokens accepted. E.g Instead of a wBTC vault, we have a renCrv vault. Or instead of 3CRV vault, we have a mimCrv vault. The strategy want token = the vault token.

[M-08] `Controller.inCaseStrategyGetStuck` does not update balance

Submitted by cmichel

The `Controller.inCaseStrategyGetStuck` withdraws from a strategy but does not call `updateBalance(_vault, _strategy)` afterwards.

Impact

The `_vaultDetails[_vault].balances[_strategy]` variable does not correctly track the actual strategy balance anymore. I'm not sure what exactly this field is used for besides getting the withdraw amounts per strategy in `getBestStrategyWithdraw`. As the strategy contains a lower amount than stored in the field, `Controller.withdraw` will attempt to withdraw too much.

Recommended Mitigation Steps

Call `updateBalance(_vault, _strategy)` in `inCaseStrategyGetStuck`.

Haz077 (yAxis) acknowledged

GalloDaSballo (judge) commented:

> Agree with finding, I also believe `inCaseStrategyGetStuck` and `inCaseTokenGetStuck` are vectors for admin rugging, may want to add checks to ensure only non strategy token can be withdrawn from the vaults and strats

BobbyYaxis (yAxis) noted:

> It's a needed function for the strategist. The risk of these functions are mitigated as the strategies and controller should never have a balance of any tokens regardless. So there should be nothing/meaningful for the strategist to ever "rug" in that sense. But we can make this a governance-only feature, rather than strategist.

[M-09] token -> vault mapping can be overwritten

Submitted by cmichel

One vault can have many tokens, but each token should only be assigned to a single vault. The `Manager` contract keeps a mapping of tokens to vaults in the `vaults[_token] => _vault` map, and a mapping of vault to tokens in `tokens[vault] => token[]`.

The `addToken` function can assign any token to a single vault and allows overwriting an existing `vaults[_token]` map entry with a different vault. This indirectly disassociates the previous vault for the token. Note that the previous vault's `tokens[_previousVault]` map still contains the `token`.

Impact

The token disappears from the system for the previous vault but the actual tokens are still in there, getting stuck. Only the new vault is considered for the token anymore, which leads to many issues, see `Controller.getBestStrategyWithdraw` and the `onlyVault` modifier that doesn't work correctly anymore.

Recommended Mitigation Steps

It should check if the `token` is already used in a map, and either revert or correctly remove the token from the vault - from the `tokens` array. It should do the same cleanup procedure as in `removeToken`:

```
    if (found) {
        // remove the token from the vault
        tokens[_vault][index] = tokens[_vault][k-1];
        tokens[_vault].pop();
        delete vaults[_token];
        emit TokenRemoved(_vault, _token);
    }
```

`addToken` should also check that the token is not already in the `tokens[_vault]` array.

GalloDaSballo (judge) commented:

> Mapping mismatch can cause undefined behaviour

> Recommend having one source of truth to keep things simple

[M-10] `YAxisVotePower.balanceOf` can be manipulated

Submitted by cmichel

The `YAxisVotePower.balanceOf` contract uses the Uniswap pool reserves to compute a `_lpStakingYax` reward:

```
(uint256 _yaxReserves,,) = yaxisEthUniswapV2Pair.getReserves();
int256 _lpStakingYax = _yaxReserves
    .mul(_stakeAmount)
    .div(_supply)
    .add(rewardsYaxisEth.earned(_voter));
```

The pool can be temporarily manipulated to increase the `_yaxReserves` amount.

Impact

If this voting power is used for governance proposals, an attacker can increase their voting power and pass a proposal.

Recommended Mitigation Steps

One could build a TWAP-style contract that tracks a time-weighted-average reserve amount (instead of the price in traditional TWAPs). This can then not be manipulated by flashloans.

uN2RVw5q commented:

> I disagree with the "sponsor disputed" tag.

> I think this is a valid issue and makes `balanceOf(_voter)` susceptible to flashloan attacks. However, as long as `balanceOf(_voter)` is always called by a trusted EOA during governance vote counts, this should not be a problem. I assume this is the case for governance proposals. If that is not the case, I would recommend changing the code. Otherwise, changing the risk to "documentation" would be reasonable.

GalloDaSballo (judge) commented:

> Agree with original warden finding, as well as severity

> The ability to trigger the count at any time does prevent a flashloan attack (as flashloans are atomic) It would allow the privilege of the flashloan attack to the trusted EOA (admin privilege)

> Additionally the voting power can still be frontrun, while you cannot manipulate that voting power via a flashloan, you can just buy and sell your position on the same block as when the count is being taken

> Due to this I will up the severity back to medium as this is a legitimate vector to extract value

[M-11] wrong YAXIS estimates

Submitted by cmichel

The `Harvester.getEstimates` contract tries to estimate a `YAXIS` amount but uses the wrong path and/or amount.

It currently uses a `WETH` **input** amount to compute a `YAXIS -> WETH` trade.

```
address[] memory _path;
_path[0] = IStrategy(_strategy).want();
_path[1] = IStrategy(_strategy).weth();
// ...

_path[0] = manager.yaxis();
// path is YAXIS -> WETH now
// fee is a WETH precision value
uint256 _fee = _estimatedWETH.mul(manager.treasuryFee()).div(ONE
// this will return wrong trade amounts
_amounts = _router.getAmountsOut(_fee, _path);
_estimatedYAXIS = _amounts[1];
```

Impact

The estimations from `getEstimates` are wrong. They seem to be used to provide min. amount slippage values `(_estimatedWETH, _estimatedYAXIS)` for the harvester when calling `Controller._estimatedYAXIS`. These are then used in `BaseStrategy._payHarvestFees` and can revert the harvest transactions if the wrongly computed `_estimatedYAXIS` value is above the actual trade value. Or they can allow a large slippage if the `_estimatedYAXIS` value is below the actual trade value, which can then be used for a sandwich attack.

Recommended Mitigation Steps

Fix the estimations computations.

As the estimations are used in `BaseStrategy._payHarvestFees`, the expected behavior seems to be trading `WETH` to `YAXIS`. The path should therefore be changed to `path[0] = WETH; path[1] = YAXIS` in `Harvester.getEstimates`.

Haz077 (yAxis) acknowledged

GalloDaSballo (judge) commented:

> Price estimates on Uniswap are dependent on which side of the swap you're making

> Sponsor has mitigated in later PR

[M-12] Harvest can be frontrun

Submitted by Oxsanson

Impact

In the `NativeStrategyCurve3Crv._harvest` there are two instances that a bad actor could use to frontrun the harvest.

First, when we are swapping WETH to a stablecoin by calling `_swapTokens(weth, _stableCoin, _remainingWeth, 1)` the function isn't checking the slippage, leading to the risk to a frontun (by imbalancing the Uniswap pair) and losing part of the harvesting profits.

Second, during the `_addLiquidity` internal function: this calls `stableSwap3Pool.add_liquidity(amounts, 1)` not considering the slippage when minting the 3CRV tokens.

Proof of Concept

`NativeStrategyCurve3Crv.sol` L108

Tools Used

editor

Recommended Mitigation Steps

In the function `_harvest(_estimatedWETH, _estimatedYAXIS)` consider adding two additional estimated quantities: one for the swapped-out stablecoin and one for the minted 3CRV.

BobbyYaxis (yAxis) acknowledged

uN2RVw5q commented:

> On second thought, I think this is a valid issue.
>
> consider adding two additional estimated quantities: one for the swapped-out stablecoin and one for the minted 3CRV.
>
> This suggestion should be considered.

GalloDaSballo (judge) commented:

> Warden identified two paths for front-running
>
> Since these are ways to extract value, severity is Medium

BobbyYaxis (yAxis) noted:

> Mitigated in PR 114: https://github.com/yaxis-project/metavault/pull/114

[M-13] `manager.allowedVaults` check missing for add/remove strategy

Submitted by 0xRajeev

Impact

The `manager.allowedVaults` check is missing for add/remove strategy like how it is used in `reorderStrategies()`. This will allow a strategist to accidentally/maliciously add/remove strategies on unauthorized vaults.

Given the critical access control that is missing on vaults here, this is classified as medium severity.

Proof of Concept

- `Controller.sol#L101` **L130**
- `Controller.sol#L172` **L207**
- `Controller.sol` **L224**
- `Manager.sol#L210` **L221**

Tools Used

Manual Analysis

Recommended Mitigation Steps

Add `manager.allowedVaults` check in `addStrategy()` and `removeStrategy()`

GainsGoblin (yAxis) acknowledged

Haz077 (yAxis) confirmed

uN2RVw5q commented:

> Implemented in https://github.com/code-423n4/2021-09-yaxis/pull/36

GalloDaSballo (judge) commented:

> Sponsor has acknowledged and mitigated by adding further access control checks

[M-14] Halting the protocol should be `onlyGovernance` and not `onlyStrategist`

Submitted by 0xRajeev

Impact

A malicious strategist can halt the entire protocol and force a permanent shutdown once they observe that governance is trying to set a new strategist and they do not agree with that decision. They may use the 7 day window to halt the protocol. The access control on `setHalted()` should be `onlyGovernance`.

Proof of Concept

- `Manager.sol#L515` **L522**
- `Manager.sol#L333` **L345**

Tools Used

Manual Analysis

Recommended Mitigation Steps

Change access control to `onlyGovernance` from `onlyStrategist` for `setHalted()`

GainsGoblin (yAxis) acknowledged

uN2RVw5q commented:

> Implemented in https://github.com/code-423n4/2021-09-yaxis/pull/35

GalloDaSballo (judge) commented:

> Agree that such critical functionality should be limited to the highest permission access role. Sponsor has mitigated

Low Risk Findings (25)

- [L-01] Missing support/documentation for use of deflationary tokens in protocol *Submitted by 0xRajeev, also found by itsmeSTYJ*
- [L-02] `getMostPremium()` can be wrong *Submitted by 0xsanson*
- [L-03] getMostPremium() does not necessarily return the best asset to trade for.
- [L-04] Be aware that transactions can be frontrun to exactly the estimated amount.
- [L-05] Missing zero-address checks *Submitted by 0xRajeev*
- [L-06] Decimals of upgradeable tokens may change *Submitted by pauliax*
- [L-07] Missing sanity/threshold check on totalDepositCap may cause DoS *Submitted by 0xRajeev*
- [L-08] No use of notHalted in LegacyController functions *Submitted by 0xRajeev*
- [L-09] onlyEnabledConverter modifier is not used in all functions *Submitted by 0xRajeev*
- [L-10] safeApprove may revert for non-zero to non-zero approvals *Submitted by 0xRajeev, also found by 0xsanson*
- [L-11] Max approvals are risky if contract is malicious/compromised *Submitted by 0xRajeev*
- [L-12] `cap` isn't enforced *Submitted by 0xsanson*
- [L-13] Unclear `totalDepositCap` *Submitted by 0xsanson*
- [L-14] Missing check in `reorderStrategies` *Submitted by 0xsanson*
- [L-15] `maxStrategies` can be lower than existing strategies *Submitted by 0xsanson*
- [L-16] Harvesting and Funding Is Not Checked When the Contract Is Halted *Submitted by defsec, also found by 0xsanson*
- [L-17] Unbounded iterations over strategies or tokens *Submitted by cmichel*
- [L-18] Withdraw event uses wrong parameter *Submitted by cmichel*
- [L-19] Vault: Zero Withdrawal Fee If Protocol Halts *Submitted by hickuphh3*

- [L-20] The `sqrt` function can overflow execute invalid operation *Submitted by hrkrshnn*
- [L-21] missing safety check in addStrategy *Submitted by jonah1005*
- [L-22] vault cap's at totalSupply would behave unexpectedly *Submitted by jonah1005, also found by pauliax*
- [L-23] Removed tokens can't be withdrawn from vault *Submitted by hickuphh3*
- [L-24] No slippage checks can lead to sandwich attacks *Submitted by cmichel*
- [L-25] hijack the vault by pumping vault price. *Submitted by jonah1005*

Non-Critical Findings (11)

- [N-01] Change public visibility to external *Submitted by 0xRajeev*
- [N-02] Unused imports *Submitted by pauliax*
- [N-03] Style issues *Submitted by pauliax*
- [N-04] setMinter should check that _minter is not empty *Submitted by pauliax*
- [N-05] Tokens with > 18 decimals will break logic *Submitted by 0xRajeev, also found by hrkrshnn, and pauliax*
- [N-06] Unused event may be unused code or indicative of missed emit/logic *Submitted by 0xRajeev, also found by cmichel*
- [N-07] Missing parameter validation *Submitted by cmichel*
- [N-08] shadowing of strategies *Submitted by gpersoon*
- [N-09] VaultHelper contract should never have tokens at the end of a transaction *Submitted by hrkrshnn*
- [N-10] Safety of the Vyper compiler *Submitted by hrkrshnn*
- [N-11] Single-step change of governance address is extremely risky *Submitted by 0xRajeev*

Gas Optimizations (27)

- [G-01] Checking for zero amounts can save gas by preventing expensive external calls *Submitted by 0xRajeev*
- [G-02] extra array length check in depositMultipleVault *Submitted by gpersoon, also found by 0xRajeev*

- [G-03] Rearranging declaration of state variables will save storage slots because of packing *Submitted by 0xRajeev*
- [G-04] Removal of last token in the array can be optimized *Submitted by 0xRajeev, also found by hickuphh3*
- [G-05] `tokens[i]` can be memorized *Submitted by 0xsanson, also found by pauliax*
- [G-06] Removing unused parameter and modifier can save gas *Submitted by 0xRajeev*
- [G-07] Earn process emits two events that can be arranged into one *Submitted by 0xsanson*
- [G-08] Unnecessary `balanceOfWant() > 0` *Submitted by 0xsanson*
- [G-09] `harvestNextStrategy` can be optimized *Submitted by 0xsanson*
- [G-10] Gas: `removeToken` iteration over all tokens can be avoided *Submitted by cmichel, also found by itsmeSTYJ*
- [G-11] Gas: `removeStrategy` iteration over all strategies can be avoided *Submitted by cmichel, also found by itsmeSTYJ*
- [G-12] Gas: Unnecessary addition in `Vault.deposit` *Submitted by cmichel, also found by jonah1005, hickuphh3, and pauliax*
- [G-13] Vault: Redundant notHalted modifier in depositMultiple() *Submitted by hickuphh3, also found by cmichel, hrkrshnn, and pauliax*
- [G-14] Gas: Loop in `StablesConverter.convert` can be avoided *Submitted by cmichel*
- [G-15] Gas: Loop in `StablesConverter.expected` can be avoided *Submitted by cmichel*
- [G-16] Gas: Timestamp in router swap can be hardcoded *Submitted by cmichel*
- [G-17] Save a step in withdraw of Vault.sol *Submitted by gpersoon*
- [G-18] Controller: Extra sload of *vaultDetails*[vault].balance *Submitted by hickuphh3*
- [G-19] Harvester: Simpler implementation for canHarvest() *Submitted by hickuphh3*

- [G-20] Consider making some constants as non-public to save gas *Submitted by hrkrshnn*
- [G-21] Caching the length in for loops *Submitted by hrkrshnn*
- [G-22] Upgrade to at least 0.8.4 *Submitted by hrkrshnn, also found by 0xRajeev*
- [G-23] uint8 is less efficient than uint256 in loop iterations *Submitted by pauliax, also found by hrkrshnn*
- [G-24] Dead code *Submitted by pauliax*
- [G-25] Join _checkToken function and modifier together *Submitted by pauliax*
- [G-26] The function `removeToken` can get prohibitively expensive *Submitted by hrkrshnn*
- [G-27] VaultHelper could validate that amount is greater than 0 *Submitted by pauliax*

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