Sandclock contest
Findings & Analysis Report

2022-02-23

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 Sandclock contest smart contract system written in Solidity. The code contest took place between January 6—January 12 2022.

Wardens

36 Wardens contributed reports to the Sandclock contest:

1. WatchPug (jtp and ming)
2. camden
3. jayjonah8
4. pauliax
5. Dravee
6. harleythedog
7. kenzo
8. leastwood
9. cmichel
10. hickuphh3
11. palina
12. defsec
13. danb
14. sirhashalot
15. pedroais
16. 0x1f8b
17. hyh
18. gzeon
19. Ruhum
20. Tomio

21. bugwriter001
22. shenwilly
23. cccz
24. p4st13r4 (0xb4bb4 and 0x69e8)
25. hubble (ksk2345 and shri4net)
26. ACai
27. pmerkleplant
28. ye0lde
29. Fitraldys
30. onewayfunction
31. certora
32. robee
33. tqts

This contest was judged by LSDan (ElasticDAO).

Final report assembled by itsmetechjay, CloudEllie, and liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 41 unique vulnerabilities and 58 total findings. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 5 received a risk rating in the category of HIGH severity, 15 received a risk rating in the category of MEDIUM severity, and 21 received a risk rating in the category of LOW severity.

C4 analysis also identified 17 non-critical recommendations.

Scope

The code under review can be found within the C4 Sandclock contest repository, and is composed of 9 smart contracts written in the Solidity programming language and includes 1400 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 (5)

[H-01] `forceUnsponsor()` may open a window for attackers to manipulate the `_totalShares` and freeze users' funds at a certain deposit amount

Submitted by WatchPug

https://github.com/code-423n4/2022-01-sandclock/blob/a90ad3824955327597be00bb0bd183a9c228a4fb/sandclock/contracts/Vault.sol#L390-L401

```
    if (_force && sponsorAmount > totalUnderlying()) {
        sponsorToTransfer = totalUnderlying();
    } else if (!_force) {
        require(
            sponsorToTransfer <= totalUnderlying(),
            "Vault: not enough funds to unsponsor"
```

```
        );
    }

    totalSponsored -= sponsorAmount;

    underlying.safeTransfer(_to, sponsorToTransfer);
```

When `sponsorAmount > totalUnderlying()` , the contract will transfer `totalUnderlying()` to `sponsorToTransfer` , even if there are other depositors and `totalShares` > 0.

After that, and before others despoiting into the Vault, the Attacker can send `1 wei` underlying token, then cal `deposit()` with 0.1 * 1e18 , since `newShares = (_amount * _totalShares) / _totalUnderlyingMinusSponsored` and `_totalUnderlyingMinusSponsored` is `1` , with a tiny amount of underlying token, `newShares` will become extremly large.

As we stated in issue **#166**, when the value of `totalShares` is manipulated precisely, the attacker can plant a bomb, and the contract will not work when the deposit/withdraw amount reaches a certain value, freezing the user's funds.

However, this issue is not caused by lack of reentrancy protection, therefore it cant be solved by the same solution in issue **#166**.

Recommendation

Consider adding a minimum balance reserve (eg. 1e18 Wei) that cannot be withdrawn by anyone in any case. It can be transferred in alongside with the deployment by the deployer.

This should make it safe or at least make it extremely hard or expensive for the attacker to initiate such an attack.

naps62 (Sandclock) confirmed and commented:

> @gabrielpoca @ryuheimat is this new?

ryuheimat (Sandclock) commented:

> it's new

gabrielpoca (Sandclock) commented:

> yap, it's interesting. The sponsor really is an issue

[H-02] Withdrawers can get more value returned than expected with reentrant call

Submitted by camden, also found by cmichel and harleythedog

The impact of this is that users can get significantly more UST withdrawn than they would be alotted if they had done non-reentrant withdraw calls.

Proof of Concept

Here's an outline of the attack:

Assume the vault has 100 UST in it. The attacker makes two deposits of 100UST and waits for them to be withdrawable. The attacker triggers a withdraw one of their deposit positions. The vault code executes until it reaches this point: https://github.com/code-423n4/2022-01-sandclock/blob/a90ad3824955327597be00bb0bd183a9c228a4fb/sandclock/contracts/Vault.sol#L565 Since the attacker is the claimer, the vault will call back to the attacker. Inside `onDepositBurned`, trigger another 100 UST deposit. Since `claimers.onWithdraw` has already been called, reducing the amount of shares, but the UST hasn't been transferred yet, the vault will compute the amount of UST to be withdrawn based on an unexpected value for `_totalUnderlyingMinusSponsored` (300). https://github.com/code-423n4/2022-01-sandclock/blob/a90ad3824955327597be00bb0bd183a9c228a4fb/sandclock/contracts/Vault.sol#L618

After the attack, the attacker will have significantly more than if they had withdrawn without reentrancy.

Here's my proof of concept showing a very similar exploit with `deposit`, but I think it's enough to illustrate the point. I have a forge repo if you want to see it, just ping

me on discord.
https://gist.github.com/CamdenClark/abc67bc1b387c15600549f6dfd5cb27a

Tools Used

Forge

Recommended Mitigation Steps

Reentrancy guards.

Also, consider simplifying some of the shares logic.

ryuheimat (Sandclock) confirmed

naps62 (Sandclock) resolved:

> Fixed in https://github.com/sandclock-org/solidity-contracts/pull/75

[H-03] Vaults with non-UST underlying asset vulnerable to flash loan attack on curve pool

Submitted by camden, also found by cccz, cmichel, danb, defsec, harleythedog, hyh, kenzo, leastwood, palina, pauliax, pmerkleplant, Ruhum, WatchPug, and yeOlde

In short, the `NonUSTStrategy` is vulnerable to attacks by flash loans on curve pools.

Here's an outline of the attack:

- Assume there is a vault with DAI underlying and a `NonUSTStrategy` with a DAI / UST curve pool
- Take out a flash loan of DAI
- Exchange a ton of DAI for UST
- The exchange rate from DAI to UST has gone up (!!)
- Withdraw or deposit from vault with more favorable terms than market
- Transfer back UST to DAI
- Repay flash loan

Proof of Concept

Here is my proof of concept:
https://gist.github.com/CamdenClark/932d5fbeecb963d0917cb1321f754132

I can provide a full forge repo. Just ping me on discord.

Exploiting this line: https://github.com/code-423n4/2022-01-sandclock/blob/a90ad3824955327597be00bb0bd183a9c228a4fb/sandclock/contracts/strategy/NonUSTStrategy.sol#L135

Tools Used

Forge

Recommended Mitigation Steps

Use an oracle

naps62 (Sandclock) confirmed

[H-04] deposit() function is open to reentrancy attacks

Submitted by jayjonah8, also found by bugwriter001, camden, cccz, cmichel, danb, defsec, Fitraldys, harleythedog, hickuphh3, jayjonah8, kenzo, leastwood, onewayfunction, pedroais, and WatchPug

In `Vault.sol` the `deposit()` function is left wide open to reentrancy attacks. The function eventually calls `\_createDeposit()` => `\_createClaim()` which calls `depositors.mint()` which will then mint an NFT. When the NFT is minted the sender will receive a callback which can then be used to call the `deposit()` function again before execution is finished. An attacker can do this minting multiple NFT's for themselves. `claimers.mint()` is also called in the same function which can also be used to call back into the deposit function before execution is complete. Since there are several state updates before and after NFT's are minted this can be used to further manipulate the protocol like with `newShares` which is called before minting. This is not counting what an attacker can do with cross function reentrancy entering into several other protocol functions (like withdraw) before code execution is complete further manipulating the system.

Proof of Concept

- https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/Vault.sol#L160
- https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/Vault.sol#L470
- https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/Vault.sol#L476

Recommended Mitigation Steps

Reentrancy guard modifiers should be placed on the `deposit()`, `withdraw()` and all other important protocol functions to prevent devastating attacks.

ryuheimat (Sandclock) confirmed

[H-05] sponsor() function in open to reentrancy attacks

Submitted by jayjonah8, also found by camden

In `Vault.sol` the `sponsor()` function does not have a reentrancy guard allowing an attacker to reenter the function because the `depositors.mint()` function has as callback to the msg.sender. Since there are state updates after the call to `depositors.mint()` function this is especially dangerous. An attacker can make it so the totalSponsored amount is only updated once after calling `mint()` several times since the update takes place after the callback. The same will be true for the Sponsored event that is emitted.

Proof of Concept

https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/Vault.sol#L244

Recommended Mitigation Steps

A reentrancy guard modifier should be added to the sponsor() function in Vault.sol

naps62 (Sandclock) confirmed and resolved:

> Fixed in https://github.com/sandclock-org/solidity-contracts/pull/75

Medium Risk Findings (15)

[M-01] Late users will take more losses than expected when the underlying contract (`EthAnchor`) suffers investment losses

Submitted by WatchPug

Even though it's unlikely in practice, but in theory, the underlying contract (`EthAnchor`) may suffer investment losses and causing decreasing of the PPS of AUST token. (There are codes that considered this situation in the codebase. eg. handling of `depositShares > claimerShares`).

However, when this happens, the late users will suffer more losses than expected than the users that withdraw earlier. The last few users may lose all their funds while the first users can get back 100% of their deposits.

Proof of Concept

```
// ### for deposits: d1, d2, d3, the beneficiary are: c1, c2, c2
    depositAmount          claimerShares
d1: + 100e18           c1: + 100e36
d2: + 100e18           c2: + 100e36
d3: + 100e18           c2: + 100e36

depositAmount of d1, d2, d3 = 100e18
c1 claimerShares: 100e36
c2 claimerShares: 200e36
total shares: 300e36


// ### when the PPS of AUST drop by 50%
totalUnderlyingMinusSponsored: 300e18 -> 150e18

// ### d2 withdraw
c2 claimerShares: 200e36
d2 depositAmount: 100e18
```

```
d2 depositShares: 300e36 * 100e18 / 150e18 = 200e36

Shares to reduce: 200e36
c2 claimerShares: 200e36 -> 0
c2 totalPrincipal: 200e18 -> 100e18
totalShares: 300e36 -> 100e36

underlying.safeTransfer(d2, 100e18)
totalUnderlyingMinusSponsored: 150e18 -> 50e18
```

Root Cause

When the strategy is losing money, `share / underlying` increases, therefore the computed `depositShares` : `depositAmount * share / underlying` will increase unexpectedly.

https://github.com/code-423n4/2022-01-sandclock/blob/a90ad3824955327597be00bb0bd183a9c228a4fb/sandclock/contracts/Vault.sol#L544-L548

While `totalShares` remain unchanged, but the computed `depositShares` is increasing, causing distortion of `depositShares / totalShares` , eg, ∑ `depositShares > totalShares` .

Recommendation

In order to properly handle the investment loss of the strategy, consider adding a new storage variable called `totalLoss` to maintain a stable value of `share / adjustedUnderlying` .

```
adjustedUnderlying = underlying + totalLoss
```

CrisBRM (Sandclock) confirmed and disagreed with severity

dmvt (judge) changed severity and commented:

> This is a classic medium risk when using the definition provided by Code4rena:

> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.

[M-02] `NonUSTStrategy.sol` Improper handling of swap fees allows attacker to steal funds from other users

Submitted by WatchPug

https://github.com/code-423n4/2022-01-sandclock/blob/a90ad3824955327597be00bb0bd183a9c228a4fb/sandclock/contracts/strategy/NonUSTStrategy.sol#L66-L69

`NonUSTStrategy` will swap the deposited non-UST assets into UST before depositing to EthAnchor. However, the swap fee is not attributed to the depositor correctly like many other yield farming vaults involving swaps (`ZapIn`).

An attacker can exploit it for the swap fees paid by other users by taking a majority share of the liquidity pool.

Root Cause

The swap fee of depositing is not paid by the depositor but evenly distributed among all users.

Proof of Concept

Given:

- A NonUST vault and strategy is created for `FRAX` ;
- The liquidity in FRAX-UST curve pool is relatively small (<$1M).

The attacker can do the following:

1. Add $1M worth of liquidity to the FRAX-UST curve pool, get >50% share of the pool;
2. Deposit 1M FRAX to the vault, get a `depositAmount` of 1M;

3. The strategy will swap 1M FRAX to UST via the curve pool, paying a certain amount of swap fee;
4. Withdraw all the funds from the vault.
5. Remove the liquidity added in step 1, profit from the swap fee. (A majority portion of the swap fee paid in step 3 can be retrieved by the attacker as the attacker is the majority liquidity provider.)

If the vault happens to have enough balance (from a recent depositor), the attacker can now receive 1M of FRAX.

A more associated attacker may combine this with issue #160 and initiate a sandwich attack in step 3 to get even higher profits.

As a result, all other users will suffer fund loss as the swap fee is essentially covered by other users.

Recommendation

Consider changing the way new shares are issued:

1. Swap from Vault asset (eg. FRAX) to UST in `deposit()`;
2. Using the UST amount out / total underlying UST for the amount of new shares issued to the depositor.

In essence, the depositor should be paying for the swap fee and slippage.

CrisBRM (Sandclock) confirmed and disagreed with severity:

> This is only an issue if we support low liquidity Curve pools We are also adding slippage control as per some other issue which would cause massive transfers using low liquidity pools to revert, fully mitigating this. Likelihood of this happening would also be quite low given that profitability would go down tremendously as curve LPs would move to that pool in order to capture higher base fees, dissuading the attacker from continuing.

> That being said, I do agree that the curve swap fee (0.04%) should be paid by each individual depositor.

dmvt (judge) changed severity and commented:

> This requires a number of external factors to line up just right. It is a medium risk according to the definition provided by Code4rena.

> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

[M-03] Centralization Risk: Funds can be frozen when critical key holders lose access to their keys

Submitted by WatchPug

The current implementation requires trusted key holders (`isTrusted[msg.sender]`) to send transactions (`initRedeemStable()`) to initialize withdrawals from `EthAnchor` before the users can withdraw funds from the contract.

- https://github.com/code-423n4/2022-01-sandclock/blob/a90ad3824955327597be00bb0bd183a9c228a4fb/sandclock/contracts/strategy/BaseStrategy.sol#L214-L223
- https://github.com/code-423n4/2022-01-sandclock/blob/a90ad3824955327597be00bb0bd183a9c228a4fb/sandclock/contracts/strategy/BaseStrategy.sol#L163-L170

This introduces a high centralization risk, which can cause funds to be frozen in the contract if the key holders lose access to their keys.

Proof of Concept

Given:

- `investPerc` = 80%
- 1,000 users deposited 1M UST in total ($1000 each user in avg), 800k invested into AUST (`EthAnchor`)

If the key holders lose access to their keys (“hit by a bus”). The 800k will be frozen in `EthAnchor` as no one can `initRedeemStable()`.

Recommendation

See the recommendation on issue **#157**.

CrisBRM (Sandclock) confirmed and disagreed with severity:

> Agree that there should be a way for users to call the uninvest functions themselves, subject to certain rules. Again, not sure I agree with the severity given the likelihood of the event transpiring.

> Consensus is for UST vaults, allow depositors to call uninvest. For nonUST vaults that pay per curve swap, add trusted multisig instead of just the backend’s EOA.

dmvt (judge) changed severity and commented:

> This issue requires external factors to align in a very negative way, but it would result in a potentially significant loss of funds. Because there is no direct attack path, it doesn’t qualify as a high risk issue, but a medium risk per Code4rena definitions.

> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

[M-04] unsponsor, claimYield and withdraw might fail unexpectedly

Submitted by danb, also found by ACai, cmichel, harleythedog, leastwood, palina, pedroais, and WatchPug

`totalUnderlying()` includes the invested assets, they are not in the contract balance.

when a user calls withdraw, claimYield or unsponsor, the system might not have enough assets in the balance and the transfer would fail.

especially, force unsponsor will always fail, because it tries to transfer the entire `totalUnderlying()`, which the system doesn't have:

https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/Vault.sol#L391

Recommended Mitigation Steps

when the system doesn't have enough balance to make the transfer, withdraw from the strategy.

gabrielpoca (Sandclock) confirmed:

> I'm not sure this is an issue. We are aware of it, and redeeming from the strategy won't fix it because it is asynchronous. This is why we have an investment percentage.

dmvt (judge) changed severity and commented:

> This one is a hard issue to size, but I'm going to go with the medium risk rating provided by other wardens reporting this issue. This seems to amount to a bank run like issue similar to what can happen with DeFi lending protocols.

> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

> If the invested assets are compromised or locked, this could result in a loss of funds. Users of the protocol should be made aware of the risk. This risk exists with many DeFi protocols and probably shouldn't be a surprise to most users.

[M-05] Add a timelock to `BaseStrategy:setPerfFeePct`

Submitted by Dravee

To give more trust to users: functions that set key/critical variables should be put behind a timelock.

Proof of Concept

https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/strategy/BaseStrategy.sol#L249-L253

Tools Used

VS Code

Recommended Mitigation Steps

Add a timelock to setter functions of key/critical variables.

naps62 (Sandclock) acknowledged:

> While this is a valid suggestion, it doesn't necessarily indicate a vulnerability in the existing approach. A timelock can indeed increase trust, but it never truly eliminates the same risk (i.e.: once the timelock finishes, the same theoretical attacks from a malicious operator could happen anyway)

ryuheimat (Sandclock) commented:

> We will set admin as a timelock

[M-06] `totalUnderlyingMinusSponsored()` may revert on underflow and malfunction the contract

Submitted by WatchPug

https://github.com/code-423n4/2022-01-sandclock/blob/a90ad3824955327597be00bb0bd183a9c228a4fb/sandclock/contracts/Vault.sol#L290-L293

```
    function totalUnderlyingMinusSponsored() public view returns (ui
        // TODO no invested amount yet
        return totalUnderlying() - totalSponsored;
```

```
    }
```

As a function that many other functions depended on, `totalUnderlyingMinusSponsored()` can revert on underflow when `sponsorAmount > totalUnderlying()` which is possible and has been considered elsewhere in this contract:

https://github.com/code-423n4/2022-01-sandclock/blob/a90ad3824955327597be00bb0bd183a9c228a4fb/sandclock/contracts/Vault.sol#L390-L392

```
    if (_force && sponsorAmount > totalUnderlying()) {
        sponsorToTransfer = totalUnderlying();
    }
```

Proof of Concept

- Underlying token = USDT
- Swap Fee = 0.04%
- Sponsor call `sponsor()` and send 10,000 USDT
- totalSponsored = 10,000
- `NonUSTStrategy.sol#doHardWork()` swapped USDT for UST
- pendingDeposits = 9,996
- totalUnderlying() = 9,996
- Alice tries to call `deposit()`, the tx will revet due to underflow in `totalUnderlyingMinusSponsored()`.

Recommendation

Change to:

```
    function totalUnderlyingMinusSponsored() public view returns (ui
        uint256 _totalUnderlying = totalUnderlying();
        if (totalSponsored > _totalUnderlying) {
            return 0;
```

```
        }
        return _totalUnderlying - totalSponsored;
    }
```

naps62 (Sandclock) confirmed

[M-07] Vault can't receive deposits if underlying token charges fees on transfer

Submitted by Ruhum, also found by harleythedog, Tomio, and WatchPug

Some ERC20 tokens charge a fee for every transfer. If the underlying token of a vault is such a token any deposit to the protocol will fail.

Some tokens have the possibility of adding fees later on, e.g. USDT. So those have to be covered too.

Generally, the user would also receive fewer tokens on withdrawing in such a scenario but that's not the protocol's fault.

I rated the issue as medium since part of the protocol become unavailable in such a situation.

Proof of Concept

https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/Vault.sol#L583-L585

`_transferAndCheckUnderlying()` is used to deposit and sponsor the vault. It checks that after a `safeTransferFrom()` the same exact amount is sent to the balance of the vault. But, if fees are enabled the values won't match, causing the function to revert. Thus, it won't be able to deposit or sponsor the vault in any way.

- https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/Vault.sol#L162
- https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/Vault.sol#L266

Recommended Mitigation Steps

One possibility would be to simply not use ERC20 tokens with fees.

ryuheimat (Sandclock) disputed:

> We don't use tokens with fees

naps62 (Sandclock) commented:

> The only place where we mention USDT is on an old pitch deck (not up to date anymore). The codebase itself doesn't mention it, and all tests are done with USDC and DAI as examples

dmvt (judge) commented:

> I'm going to let this issue stand given that #164 is also valid. Supported or not, fee on transfer tokens would cause a loss of funds in the scenario described. As the USDT example shows (in both issues), many stables can be upgraded and add a fee later.

[M-08] Medium: Consider alternative price feed + ensure _minLockPeriod > 0 to prevent flash loan attacks

Submitted by hickuphh3, also found by 0x1f8b

It is critical to ensure that `_minLockPeriod > 0` because it is immutable and cannot be changed once set. A zero `minLockPeriod` will allow for flash loan attacks to occur. Vaults utilising the nonUST strategy are especially susceptible to this attack vector since the strategy utilises the spot price of the pool to calculate the total asset value.

Proof of Concept

Assume the vault's underlying token is MIM, and the curve pool to be used is the MIM-UST pool. Further assume that both the vault and the strategy holds substantial funds in MIM and UST respectively.

1. Flash loan MIM from the Uniswap V3 MIM-USDC pool (currently has ~$3.5M in MIM at the time of writing).
2. Convert half of the loaned MIM to UST to inflate and deflate their prices respectively.
3. Deposit the other half of the loaned MIM into the vault. We expect `curvePool.get_dy_underlying(ustI, underlyingI, ustAssets);` to return a smaller amount than expected because of the previous step. As a result, the attacker is allocated more shares than expected.
4. Exchange UST back to MIM, bringing back the spot price of MIM-UST to a normal level.
5. Withdraw funds from the vault. The number of shares to be deducted is lower as a result of (4), with the profit being accounted for as yield.
6. Claim yield and repay the flash loan.

Recommended Mitigation Steps

Ensure that `_minLockPeriod` is non-zero in the constructor. Also, given how manipulatable the spot price of the pool can be, it would be wise to consider an alternative price feed.

```
// in Vault#constructor
require(_minLockPeriod > 0, 'zero minLockPeriod');
```

ryuheimat (Sandclock) disputed:

> we don't think it's an issue.

dmvt (judge) commented:

> This does potentially open assets up to flash loan risk. It is probably a good idea to have this variable guarded.

[M-09] no use of safeMint() as safe guard for users

Submitted by jayjonah8, also found by bugwriter001, camden, palina, and sirhashalot

In `Vault.sol` the `deposit()` function eventually calls claimers.mint() and depositers.mint(). Calling mint this way does not ensure that the receiver of the NFT is able to accept them. `\_safeMint()` should be used with reentrancy guards as a guard to protect the user as it checks to see if a user can properly accept an NFT and reverts otherwise.

Proof of Concept

- https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/Vault.sol#L470
- https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/Vault.sol#L256
- https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC721/ERC721.sol#L248

Recommended Mitigation Steps

Use `\_safeMint()` instead of `mint()`

ryuheimat (Sandclock) disagreed with severity:

> I think _safeMint check if the recipient contract is able to accept NFT, it does not involves any issues. However we will use _safeMint.

gabrielpoca (Sandclock) commented:

> @ryuheimat this is a non-issue. The mint functions called in the Vault's deposit function are implemented by us, they just happen to be called mint.

dmvt (judge) commented:

> The Depositors contract does use `_safeMint`, but the Claimers contract does not.

> See: https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/vault/Claimers.sol#L63
> https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/vault/Depositors.sol#L53

> The `deposit` function on Vault also appears to lack reentrancy guards. The issue is valid and should be addressed, despite the fact that the warden clearly did not look at the Depositors contract to see that it already used `_safeMint`.

[M-10] No setter for exchangeRateFeeder, whose address might change in future

Submitted by kenzo

EthAnchor's docs state that "the contract address of ExchangeRateFeeder may change as adjustments occur". BaseStrategy does not have a setter to change exchangeRateFeeder after deployment.

Impact

Inaccurate/unupdated values from exchangeRateFeeder when calculating vault's total invested assets.

While the strategy's funds could be withdrawn from EthAnchor and migrated to a new strategy with correct exchangeRateFeeder, during this process (which might take time due to EthAnchor's async model) the wrong exchangeRateFeeder will be used to calculate the vault's total invested assets. (The vault's various actions (deposit, claim, withdraw) can not be paused.)

Proof of Concept

The exchangeRateFeeder is being used to calculate the vault's invested assets, which is used extensively to calculate the correct amount of shares and amounts: **(Code ref)**

```
    function investedAssets() external view virtual override(IStrate
        uint256 underlyingBalance = _getUnderlyingBalance() + pendir
        uint256 aUstBalance = _getAUstBalance() + pendingRedeems;

        return underlyingBalance + ((exchangeRateFeeder.exchangeRate
                * aUstBalance) / 1e18);
    }
```

EthAnchor documentation states that unlike other contracts, exchangeRateFeeder is not proxied and it's address may change in future: "the contract address of ExchangeRateFeeder may change as adjustments occur. " (ref)

Recommended Mitigation Steps

Add a setter for exchangeRateFeeder.

ryuheimat (Sandclock) confirmed

[M-11] Changing a strategy can be bricked

Submitted by kenzo, also found by danb and harleythedog

A vault wouldn't let the strategy be changed unless the strategy holds no funds.

Since anybody can send funds to the strategy, a griefing attack is possible.

Impact

Strategy couldn't be changed.

Proof of Concept

`setStrategy` requires `strategy.investedAssets() == 0`. (Code ref)

`investedAssets` contains the aUST balance and the pending redeems: (Code ref)

```
    uint256 aUstBalance = _getAUstBalance() + pendingRedeems;
```

So if a griefer sends 1 wei of aUST to the strategy before it is to be replaced, it would not be able to be replaced. The protocol would then need to redeem the aUST and wait for the process to finish - and the griefer can repeat his griefing. As they say, griefers gonna grief.

Recommended Mitigation Steps

Consider keeping an internal aUST balance of the strategy, which will be updated upon deposit and redeem, and use it (instead of raw aUST balance) to check if the strategy holds no aUST funds.

Another option is to add capability for the strategy to send the aUST to the vault.

ryuheimat (Sandclock) confirmed

CloudEllie (C4) commented:

> Warden kenzo requested that I add the following:
>
> "Additionally, impact-wise: EthAnchor does not accept redeems of less than 10 aUST. This means that if a griefer only sends 1 wei aUST, the protocol would have to repeatedly send additional aUST to the strategy to be able to redeem the griefer's aUST."

[M-12] `investedAssets()` Does Not Take Into Consideration The Performance Fee Charged On Strategy Withdrawals

Submitted by leastwood, also found by danb

The `investedAssets()` function is implemented by the vault's strategy contracts as a way to express a vault's investments in terms of the underlying currency. While the implementation of this function in `BaseStrategy.sol` and `NonUSTStrategy.sol` is mostly correct. It does not account for the performance fee charged by the treasury as shown in `finishRedeemStable()`.

Therefore, an attacker could avoid paying their fair share of the performance fee by withdrawing their assets before several calls to `finishRedeemStable()` are made and reenter the vault once the fee is charged.

Proof of Concept

https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/strategy/BaseStrategy.sol#L180-L204

```
function finishRedeemStable(uint256 idx) public virtual {
    require(redeemOperations.length > idx, "not running");
    Operation storage operation = redeemOperations[idx];
    uint256 aUstBalance = _getAUstBalance() + pendingRedeems;
    uint256 originalUst = (convertedUst * operation.amount) / aU
```

```
        uint256 ustBalanceBefore = _getUstBalance();

        ethAnchorRouter.finishRedeemStable(operation.operator);

        uint256 redeemedAmount = _getUstBalance() - ustBalanceBefore
        uint256 perfFee = redeemedAmount > originalUst
            ? (redeemedAmount - originalUst).percOf(perfFeePct)
            : 0;
        if (perfFee > 0) {
            ustToken.safeTransfer(treasury, perfFee);
            emit PerfFeeClaimed(perfFee);
        }
        convertedUst -= originalUst;
        pendingRedeems -= operation.amount;

        operation.operator = redeemOperations[redeemOperations.lengt
            .operator;
        operation.amount = redeemOperations[redeemOperations.length
        redeemOperations.pop();
    }
```

https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/strategy/BaseStrategy.sol#L263-L277

```
    function investedAssets()
        external
        view
        virtual
        override(IStrategy)
        returns (uint256)
    {
        uint256 underlyingBalance = _getUnderlyingBalance() + pendir
        uint256 aUstBalance = _getAUstBalance() + pendingRedeems;

        return
            underlyingBalance +
            ((exchangeRateFeeder.exchangeRateOf(address(aUstToken),
                aUstBalance) / 1e18);
    }
```

https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/strategy/NonUSTStrategy.sol#L120-

L136

```
    function investedAssets()
        external
        view
        override(BaseStrategy)
        returns (uint256)
    {
        uint256 underlyingBalance = _getUnderlyingBalance();
        uint256 aUstBalance = _getAUstBalance() + pendingRedeems;

        uint256 ustAssets = ((exchangeRateFeeder.exchangeRateOf(
            address(aUstToken),
            true
        ) * aUstBalance) / 1e18) + pendingDeposits;
        return
            underlyingBalance +
            curvePool.get_dy_underlying(ustI, underlyingI, ustAssets
    }
```

Tools Used

Manual code review. Discussions with the Sandclock team (mostly Ryuhei).

Recommended Mitigation Steps

When calculating the `investedAssets()` amount (expressed in the underlying currency), consider calculating the expected performance fee to be charged if all the strategy's assets are withdrawn from the Anchor protocol. This should ensure that `investedAssets()` returns the most accurate amount, preventing users from gaming the protocol.

ryuheimat (Sandclock) confirmed

[M-13] Incompatibility With Rebasing/Deflationary/Inflationary tokens

Submitted by defsec

The Strategy contracts do not appear to support rebasing/deflationary/inflationary tokens whose balance changes during transfers or over time. The necessary checks include at least verifying the amount of tokens transferred to contracts before and after the actual transfer to infer any fees/interest.

Proof of Concept

- https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/strategy/BaseStrategy.sol#L239
- https://github.com/code-423n4/2022-01-sandclock/blob/main/sandclock/contracts/strategy/BaseStrategy.sol#L221

Recommended Mitigation Steps

- Make sure token vault accounts for any rebasing/inflation/deflation
- Add support in contracts for such tokens before accepting user-supplied tokens
- Consider to check before/after balance on the vault.

naps62 (Sandclock) disputed:

> we did not intend to support those currencies in the first place

dmvt (judge) commented:

> As with issues #55 and #164, this oversight can cause a loss of funds and therefor constitutes a medium risk. Simply saying you don't support something does not mean that thing doesn't exist or won't cause a vulnerability in the future.

[M-14] A Single Malicious Trusted Account Can Takeover Parent Contract

Submitted by leastwood, also found by hickuphh3

The `requiresTrust()` modifier is used on the strategy, vault and factory contracts to prevent unauthorised accounts from calling restricted functions. Once an account is considered trusted, they are allowed to add and remove accounts by calling `setIsTrusted()` as they see fit.

However, if any single account has its private keys compromised or decides to become malicious on their own, they can remove all other trusted accounts from the `isTrusted` mapping. As a result, they are effectively able to take over the trusted group that controls all restricted functions in the parent contract.

Proof of Concept

```
abstract contract Trust {
    event UserTrustUpdated(address indexed user, bool trusted);

    mapping(address => bool) public isTrusted;

    constructor(address initialUser) {
        isTrusted[initialUser] = true;

        emit UserTrustUpdated(initialUser, true);
    }

    function setIsTrusted(address user, bool trusted) public vir
        isTrusted[user] = trusted;

        emit UserTrustUpdated(user, trusted);
    }

    modifier requiresTrust() {
        require(isTrusted[msg.sender], "UNTRUSTED");

        _;
    }
}
```

Recommended Mitigation Steps

Consider utilising Rari Capital's updated `Auth.sol` contract found here. This updated contract gives the `owner` account authority over its underlying trusted accounts, preventing any single account from taking over the trusted group. The `owner` account should point to a multisig managed by the Sandclock team or by a community DAO.

naps62 (Sandclock) confirmed

dmvt (judge) changed severity and commented:

> If this were to happen, funds would definitely be lost. Accordingly, this is a medium risk issue.

> `2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements.`

[M-15] Check _to is not empty

Submitted by pauliax

functions `claimYield`, `\_withdraw`, and `\_unsponsor` should validate that `\_to` is not an empty 0x0 address to prevent accidental burns.

Recommended Mitigation Steps

Consider implementing the proposed validation: require `\_to != address(0)`

gabrielpoca (Sandclock) confirmed

dmvt (judge) commented:

> In this case assets are at risk due to external factors. A zero address check makes sense.

Low Risk Findings (21)

- [L-01] `NonUSTStrategy.sol` A malicious user/attacker can game the system by `claimYield()` or `withdraw()` based on price changes *Submitted by WatchPug*
- [L-02] Incorrect share accounting *Submitted by gzeon*
- [L-03] Some Strategy functions can't be called from the Vault *Submitted by palina*

- [L-04] Lack of checks *Submitted by 0x1f8b, also found by hubble, leastwood, and pedroais*
- [L-05] Unclear require statement *Submitted by Dravee*
- [L-06] `BaseStrategy:perfFeePct` can be 100% *Submitted by Dravee*
- [L-07] Open TODOs *Submitted by Dravee, also found by 0x1f8b, camden, cccz, certora, defsec, jayjonah8, kenzo, p4st13r4, palina, pauliax, and robee*
- [L-08] Incorrect comment in BaseStrategy *Submitted by camden*
- [L-09] NonUST strategies lose value with swap fees *Submitted by cmichel*
- [L-10] NonUSTStrategy: Ensure correct UST index *Submitted by hickuphh3*
- [L-11] IVault underlying() description is wrong *Submitted by hyh*
- [L-12] Use of floating pragma statement *Submitted by jayjonah8, also found by palina*
- [L-13] `Vault.sponsor()` Does Not Prevent Sponsoring The Zero Amount *Submitted by leastwood*
- [L-14] Missing validation in constructors *Submitted by palina*
- [L-15] _lockedUntil is not deterministic and does not have an upper boundary *Submitted by pauliax*
- [L-16] A precision loss when creating deposits *Submitted by pauliax*
- [L-17] No input check : claim.pct = 0 *Submitted by pedroais*
- [L-18] Wrong comment on `IVault.sol` *Submitted by shenwilly, also found by pedroais*
- [L-19] Incorrect require message *Submitted by sirhashalot*
- [L-20] Inaccurate comment in Depositors.sol *Submitted by sirhashalot*
- [L-21] Wrong revert message at Depositors' onlyVault modifier *Submitted by kenzo, also found by p4st13r4 and palina*

Non-Critical Findings (17)

- [N-01] BaseStrategy implements USTStrategy and is risky to inherit from *Submitted by palina*
- [N-02] Critical operations should emit events *Submitted by WatchPug, also found by palina*

- [N-03] redundant variable *Submitted by danb, also found by hyh and palina*
- [N-04] BaseStrategy.finishRedeemStable can produce a low-level division revert *Submitted by hyh*
- [N-05] `Depositor` and `Sponsor` are used interchangeably *Submitted by palina*
- [N-06] Unused imports *Submitted by robee, also found by shenwilly and tqts*
- [N-07] safeApprove of openZeppelin is deprecated *Submitted by robee, also found by sirhashalot and WatchPug*
- [N-08] Lack of inputs *Submitted by 0x1f8b*
- [N-09] Incorrect use of modifier *Submitted by ACai*
- [N-10] typo miskate in comment *Submitted by certora, also found by ye0lde*
- [N-11] Vault: Spelling Error in comment *Submitted by hickuphh3*
- [N-12] Incomplete natspec comments *Submitted by kenzo*
- [N-13] Inconvenient retrieval of depositors token IDs *Submitted by kenzo*
- [N-14] Account for future reentrancy attacks *Submitted by p4st13r4*
- [N-15] Event ProfitShared in IStrategy is never used *Submitted by palina*
- [N-16] Missing topic in Unsponsored event *Submitted by palina*
- [N-17] Deprecated Trust.sol dependency *Submitted by sirhashalot*

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