Visor Finance
Findings & Analysis Report

2021-08-30

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 o the Visor Finance smart contract system written in Solidity. The code contest took place between May 12 — May 19, 2021.

Wardens

12 Wardens contributed reports to the Visor Finance code contest:

- 0xRajeev
- shw
- pauliax
- gpersoon
- Sherlock (team of 4)
- cmichel
- toastedsteaksandwich
- Jmukesh

- a_delamo

This contest was judged by ghoulsol.

Final report assembled by ninek and moneylegobatman.

Summary

The C4 analysis yielded an aggregated total of 30 unique vulnerabilities. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 4 received a risk rating in the category of HIGH severity, 3 received a risk rating in the category of MEDIUM severity, and 8 received a risk rating in the category of LOW severity.

C4 analysis also identified 15 non-critical recommendations.

Scope

The code under review can be found within the C4 Visor Finance code contest repository and is comprised of 24 smart contracts written in the Solidity programming language.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings

[H-01] A previously timelocked NFT token becomes permanently stuck in vault if it's ever moved back into the vault

Submitted by 0xRajeev, also found by pauliax

Let's consider a scenario where a particular NFT token was timelocked for a certain duration by the owner using `timeLockERC721()` with a delegate as the recipient and then transferred out of the vault by the delegate via `transferERC721()` but without unlocking it explicitly using `timeUnlockERC721()`.

This is possible because `transferERC721()` does all the timelock checks on `expires/block.timestamp` and `recipient/msg.sender` as is done in `timeUnlockERC721()`. But it misses deleting `timelockERC721s[key]` for that NFT `tokenID` (as done in L572 of `timeUnlockERC721()`).

Because of this missing deletion, if that same NFT is ever put back into the vault later but this time without a timelock, the vault logic still thinks it is a timelocked NFT with the older/stale recipient from earlier because of the missing deletion. So now the owner who makes the `transferERC721()` call will not match the older/stale recipient address and will fail the check on L510 (unless they control that stale recipient address from the earlier timelock).

The impact is that, without access/control to the earlier timelock recipient, this NFT token is now locked in the vault forever.

1. Alice time locks a particular NFT token with delegate Eve as recipient using `timeLockERC721()`
2. Eve transfers NFT to Bob using `transferERC721()` but without calling `timeUnlockERC721()` first

3. Alice buys the same NFT back from Bob (e.g. because it is now considered rare and more valuable) and again puts it back in her vault but this time without locking/delegating it to any recipient i.e. intending to control it herself.
4. Because this NFT's timelock data and delegate approval for Eve is never removed after Step 2, the NFT is still treated as timelocked in the vault with previous delegate Eve as the recipient (because of stale data in `timelockERC721s` and `nftApprovals`)
5. Alice now cannot withdraw her own NFT without Eve's help because the check on L510 will only allow Eve to transfer this NFT out of the vault.
6. If Eve is no longer trusted/accessible then the NFT is locked in the vault forever.

Recommend adding `delete timelockERC721s[timelockERC721Keys[nftContract][i]];` after L510.

xyz-ctrl (Visor) confirmed:

ztcrypto (Visor) patched:

> patch link

[H-02] NFT transfer approvals are not removed and cannot be revoked thus leading to loss of NFT tokens

Submitted by 0xRajeev, also found by shw

NFT transfer approvals that are set to true in `approveTransferERC721()` are never set to false and there is no way to remove such an nft approval.

Impact 1: The approval is not removed (set to false) after a transfer in `transferERC721()`. So if the NFT is ever moved back into the owner's vault again, then the previous/compromised delegate can again transfer it to any address of choice without requiring a new approval.

Impact 2: If a delegate becomes compromised/untrustworthy after granting approval but before transfer then the owner will lose its NFT because there is no mechanism to revoke the approval that was granted earlier.

PoC-1:

- Alice grants Eve approval to transfer a particular NFT out of its vault using `approveTransferERC721()`
- Eve, who has transfer rights to that NFT from Alice's vault, transfers that NFT to Bob using `transferERC721()`
- Alice decides to buy back that NFT (e.g. because it is now considered rare and more valuable) from Bob and transfers it back to its vault
- Eve, who continues to have transfer rights to that NFT from Alice's vault, can steal that NFT and transfer to anyone

PoC-2:

- Alice grants Eve approval to transfer a particular NFT out of its vault using `approveTransferERC721()`
- Alice learns that Eve's keys are compromises or that Eve is malicious and wants to revoke the approval but there is no mechanism to do so
- Eve (or whoever stole her credentials) has transfer rights to that NFT from Alice's vault and can steal that NFT and transfer to anyone

Recommend adding a boolean parameter to `approveTransferERC721()` and set the `nftApprovals` to that parameter which can be true for giving approval and false for removing/revoking approval If `msg.sender != _getOwner()`, call `approveTransferERC721()` with the boolean false to remove approval before making a transfer in `transferERC721()` on L515.

xyz-ctrl (Visor) commented:

> duplicate https://github.com/code-423n4/2021-05-visorfinance-findings/issues/35

ghoul-sol (Judge) commented:

35 is about token being stuck in the vault. This issue is about not being able to revoke approval. Marking this as separate.

ztcrypto (Visor) patched:

> patch link

[H-03] Approval for NFT transfers is not removed after transfer

Submitted by cmichel, also found by gpersoon, and pauliax

The `Visor.transferERC721` does not reset the approval for the NFT.

An approved delegatee can move the NFT out of the contract once. It could be moved to a market and bought by someone else who then deposits it again to the same vault. The first delegatee can steal the NFT and move it out of the contract a second time.

Recommend resetting the approval on transfer.

xyz-ctrl (Visor) confirmed:

> We will be mitigating this issue for our next release and before these experimental features are introduced in platform. PR pending

ztcrypto (Visor) commented:

> duplicate of above ones and fixed

[H-04] Unbounded loop in `_removeNft` could lead to a griefing/DOS attack

Submitted by shw, also found by cmichel, gpersoon, pauliax, Sherlock, and toastedsteaksandwhich

Griefing/DOS attack is possible when a malicious NFT contract sends many NFTs to the vault, which could cause excessive gas consumed and even transactions reverted when other users are trying to unlock or transfer NFTs.

The function `_removeNft` uses an unbounded loop, which iterates the array nfts until a specific one is found. If the NFT to be removed is at the very end of the nfts array, this function could consume a large amount of gas. The function `onERC721Received` is permission-less. The vault accepts any NFTs from any NFT contract and pushes the received NFT into the array nfts. A malicious user could write an NFT contract, which calls `onERC721Received` of the vault many times to make the array nfts grow to a large size. Besides, the malicious NFT contract reverts when anyone tries to transfer (e.g., `safeTransferFrom`) its NFT. The vault then has no way to remove the transferred NFT from the malicious NFT contract. The two only functions to remove NFTs, `transferERC721` and `timeUnlockERC721`, fail since the malicious NFT contract reverts all `safeTransferFrom` calls. As a result, benign users who unlock or transfer NFTs would suffer from large and unnecessary gas consumption. The consumed gas could even exceed the block gas limit and cause the transaction to fail every time.

Recommend using a mapping (e.g., `mapping(address=>Nft[]) nfts`) to store the received NFTs into separate arrays according to `nftContract` instead of putting them into the same one. Or, add a method specifically for the owner to remove NFTs from the nfts array directly.

xyz-ctrl (Visor) confirmed:

> sponsor confirmed We are working to mitigate this issue in our next upgrade

ghoul-sol (Judge) commented:

> I'm making this high severity because it doesn't need a malicious actor to happen. This can happen by simply being very successful

xyz-ctrl (Visor) commented:

> Agreed. We merged a pr with mitigations a few weeks ago here VisorFinance/visor-core#2

ghoul-sol (Judge) commented:

> Duplicate of **#66**

ztcrypto (Visor) patched:

> duplicated and patched link

Medium Risk Findings

[M-01] Unhandled return value of `transferFrom` in `timeLockERC20()` could lead to fund loss for recipients

Submitted by OxRajeev, also found by Sherlock, pauliax, shw, and JMukesh

ERC20 implementations are not always consistent. Some implementations of `transfer` and `transferFrom` could return 'false' on failure instead of reverting. It is safer to wrap such calls into `require()` statements or use safe wrapper functions implementing return value/data checks to handle these failures. For reference, see similar Medium-severity finding from Consensys Diligence Audit of Aave Protocol V2.

While the contract uses Uniswap's `TransferHelper` library function `safeTransfer` in other places for ERC20 tokens, or OpenZeppelin's `saferTransferFrom` for ERC721 tokens (both of which call the token's `transfer` / `transferFrom` functions and check return value for success and return data), it misses using `TransferHelper.safeTransferFrom` in this one case on L610 in `timeLockERC20()` when tokens are transferred from owner to the vault and instead directly uses the token's `transferFrom()` call without checking for its return value.

The impact can be that for an arbitrary ERC20 token, this `transferFrom()` call may return failure but the vault logic misses that, assumes it was successfully transferred into the vault and updates the `timelockERC20Balances` accounting accordingly. The `timeUnlockERC20()`, `transferERC20()` or `delegatedTransferERC20()` calls for that token will fail because the vault contract balance would have less tokens than accounted for in `timelockERC20Balances` because of the previously failed (but ignored) `transferFrom()` call.

1. Let's say Alice owes Bob 100 USD after a week, for which they agree that Alice will pay in 100 tokens of USD stablecoin tokenA.

2. Alice, the vault owner, calls `timeLockERC20()` for recipient=Bob, token=tokenA, amount=100 and expiry=1-week-from-then (corresponding Unix timestamp) but tokenA's implementation does not revert on failure but instead returns true/false. If the `transferFrom` failed, say because Alice did not have those 100 tokenAs, the return value is ignored on L610 in `timeLockERC20()` and vault logic considers that it indeed has 100 tokenAs locked for Bob.
3. Bob looks at the `TimeLockERC20` event emitted in the successful `timeLockERC20()` transaction from Alice and assumes 100 tokenAs are indeed locked by Alice in the vault for him which can be withdrawn after expiry.
4. After timelock expiry, Bob tries to transfer the 100 tokenAs Alice locked in the vault for him. The `TransferHelper.safeTransfer()` call on L637 in `timeUnlockERC20()` fails because the vault has 0 tokenAs because they were never successfully transferred in Step 2.
5. Bob could thus be tricked into thinking that 100 tokenAs are locked in the vault for him by Alice but they never were. This leads to loss of funds for Bob.

Recommend replacing use of

```
IERC20(token).transferFrom(msg.sender, address(this), amount);
```

with

```
TransferHelper.safeTransferFrom(token, msg.sender, address(this)
```

This will revert on transfer failure for e.g. if `msg.sender` does not have a token balance >= amount.

xyz-ctrl (Visor) acknowledged:

> sponsor acknowledged disagree with severity 0 While we may include refactor in next version, this is all foreseen behavior and is component of many stable ethereum project. The onus here is on client

ghoul-sol (Judge) commented:

> I'm going to make it medium as the risk is there but it could be mitigated by UI and tokens that are used.

[ztcrypto (Visor) patched](#):

> fixed patch [link](#)

[M-02] `transferERC721` doesn't clean `timelockERC721s`

Submitted by gpersoon, also found by shw

The function `transferERC721` works similar to the functions `timeUnlockERC721` with timelocked NFT's. However `timeUnlockERC721` cleans `timelockERC721s` (delete `timelockERC721s[key]` ;), while `transferERC721` doesn't clean `timelockERC721s`

This could mean that timelock keys could be used later on (when the NFT would have been transferred to the contract on a later moment in time). Also, the administration doesn't correspond to the available NFT's. Additionally doing a delete gives backs some gas (at least for now).

See [Issue #19](#) for code referenced in proof of concept

Recommend checking if the `timelockERC721s` mapping should also be cleaned from `transferERC721` , if so adapt the code accordingly.

[xyz-ctrl (Visor) confirmed](#):

[ztcrypto (Visor) patched](#):

> patch [link](#)

[M-03] `timelockERC721Keys` could exceed the block size limit

Submitted by Sherlock, also found by shw

On line 504 of `Visor.sol`, looping through the `timelockERC721Keys` could exceed the block size limit

Recommend transfer by index instead of token ID

[xyz-ctrl (Visor) acknowledged]():

> sponsor acknowledged We will be significantly refactoring experimental nft functionality in our next version before exposing to users of platform. In this refactor we will cap size of nft collection

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

> I'm going to bump it to medium severity because this may happen if project is very successful

[ztcrypto (Visor) patched]():

> patch [link]()

> By using EnumerableSets

Low Risk Findings

[L-01] sandwich `approveTransferERC20`

Submitted by pauliax, also found by 0xRajeev, and shw

Function `approveTransferERC20` is vulnerable to the sandwich attack. Similar to the ERC20 approve issue described [here](). A malicious delegate can scout for a `approveTransferERC20` change and sandwich that (`delegatedTransferERC20` amount A, `approveTransferERC20` amount A->B, `delegatedTransferERC20` amount B). It is more of a theoretical issue and mostly depends on the honesty of the delegators. If we can assume that delegators are trustable actors, then this is very unlikely to happen.

Possible mitigation could be to replace `approveTransferERC20` with increasing/decreasing functions.

xyz-ctrl (Visor) acknowledged:

ztcrypto (Visor) commented:

> this is skiped for now

[L-02] Wrong `TimeLockERC20` event emitted

Submitted by cmichel, also found by 0xRajeev, pauliax

The `Visor.timeLockERC721` function emits the `TimeLockERC20` event but should emit `TimeLockERC721` instead.

It allows tricking the backend into registering ERC20 token transfers that never happened which could lead to serious issues when something like an accounting app uses this data.

Recommend emitting the correct event.

xyz-ctrl (Visor) confirmed but disputed severity:

ghoul-sol (Judge) comment:

> Agree with sponsor. Even though it's obviously wrong event, there is no obvious high security risk here.

ztcrypto (Visor) patched:

> patch link

[L-03] Timelock keys are never removed after unlocks

Submitted by 0xRajeev, also found by shw `timelockERC20Keys` and `timelockERC721Keys` are used to keep track of number of timelocks for ERC20 and ERC721 tokens. While `timelockERC20()` and `timelockERC721()` functions update these data structures to add the new timelocks, the corresponding unlock functions do not remove the expired timelocks.

This results in their getter functions `getTimeLockCount()` and `getTimeLockERC721Count()` returning the number of all timelocks ever held instead of the expected number of timelocks that are currently active.

Let's say 5 timelocks are created for a specific ERC20 token of which 3 have been unlocked after expiry. The getter function `getTimeLockCount()` incorrectly reports 5 instead of 2.

Recommend removing unlocked keys from `timelockERC20Keys` and `timelockERC721Keys` in `timeUnlockERC20()` and `timeUnlockERC721()` functions.

xyz-ctrl (Visor) confirmed:

ztcrypto patched:

> patch link

[L-04] The function `onERC721Received()` allows writing duplicates in the array "nfts". Another functions dealing with this array do not expect duplicates met.

Submitted by Sherlock

Duplicates can be written accidentally. `If_removeNft()` function is running, it will break when meeting the first match, not trying to remove other duplicates. Thus a caller should call removing a few times.

```
    function onERC721Received(address operator, address from, uint25
          _addNft(msg.sender, tokenId);

    function _addNft(address nftContract, uint256 tokenId) internal
```

```
        nfts.push(
        Nft({
            tokenId: tokenId,
            nftContract: nftContract
        })
        );
```

Recommend that `In _addNft()` to check if an inputted nft is existing in the "nfts" array. Do not push inputted nft if already added.

xyz-ctrl (Visor) confirmed:

ztcrypto (Visor) patched:

> patch link

[L-05] `delegatedTransferERC20` can revert when called by owner

Submitted by gpersoon, also found by cmichel and pauliax

If the function `delegatedTransferERC20` is called from the owner (e.g. msg.sender == _getOwner) then `erc20Approvals[keccak256(abi.encodePacked(msg.sender, token))]` doesn't have to set, so it can have the value of 0.

If you then subtract the amount, you will get an error and the code will revert:

```
    erc20Approvals[keccak256(abi.encodePacked(msg.sender, token))] =
```

A workaround would be to call `approveTransferERC20` also for the owner.

```
    function delegatedTransferERC20(address token,address to,uint25
        if(msg.sender != _getOwner()) {
            require(erc20Approvals[keccak256(abi.encodePacked(msg.se
        }
        // check for sufficient balance
        require(IERC20(token).balanceOf(address(this)) >= (getBalanc

        erc20Approvals[keccak256(abi.encodePacked(msg.sender, token)

        // perform transfer
        TransferHelper.safeTransfer(token, to, amount);
    }
```

Recommend also adding `if(msg.sender != _getOwner())` before

```
        erc20Approvals[keccak256(abi.encodePacked(msg.sender, token))
```

xyz-ctrl (Visor) acknowledged:

> dispute severity 0 The owner does not need to call this function to transferERC20. These does not occur in our platforms context.

ghoul-sol (Judge) commented:

> If owner doesn't call this function it should be refactored. I'm going to stick with warden on this one because code explicitly suggests it's going to be used by owner.

ztcrypto (Visor) patched:

> patch link

[L-06] Locking the same funds twice in `lock()` on line 269 of `Visor.sol`

Submitted by Sherlock

Two different addresses (Alice and Bob) could get credit for locking up the same funds because a user is able to lock without depositing.

Recommend implementing additional checks to force users to have deposited before they are able to lock tokens [xyz-ctrl (Visor) disputed](#):

> disagree with severity 0 These locks are meant to be operated by 3rd party contract and the locks are only as meaningful as this 3rd party contract context allows them to be. The unit test is well put together but exhibits expected behavior

[ghoul-sol (Judge) commented](#):

> In context with 3rd contract this is a non-critical issue but I'll keep low severity because this is extremely confusing and not well documented

[L-07] Deflationary tokens are not considered in time-locked ERC20 functions

Submitted by shw

The functions `timeLockERC20` and `timeUnlockERC20` do not consider deflationary tokens, which burn a percentage of the transferred amount during transfers. In that case, time-locked deflationary ERC20 tokens cannot be unlocked (by `timeUnlockERC20`) nor transferred out of the vault (by `transferERC20`), since the transferred amount exceeds the vault's balance.

Recommend that in function `timeLockERC20`, after the function `transferFrom`, the vault should get the actual received amount by `token.balanceOf(address(this)).sub(tokenAmountBeforeTransfer)`.

[xyz-ctrl (Visor) acknowledged](#):

> sponsor acknowledged True. We are not planning on serving rebasing tokens in this case

[L-08] missing condition in `addTemplate(bytes32 name, address template)`, `visorFactory.sol`

Submitted by JMukesh, also found by 0xRajeev

In `require()` of function `addTemplate(bytes32 name, address template)`, we check if a given name has been allotted or not. But, it misses checking the second parameter of function that is template. Without checking template address, an unintended address can be set for given name.

Recommend adding one more condition in `require()` for checking of template address.

xyz-ctrl (Visor) acknowledged:

ztcrypto (Visor) commented:

> this address check is not critical for now which is only called by the owner

Non-Critical Findings

Gas Optimizations

[G-01] Change function visibility from public to external

Submitted by OxRajeev

Functions `getTimeLockCount()`, `getTimeLockERC721Count()`, `timeLockERC721()`, `timeUnlockERC721()`, `timeLockERC20()` and `timeUnlockERC20()` are never called from within contracts but yet declared public. Their visibility can be made external to save gas.

As described in https://mudit.blog/solidity-gas-optimization-tips/:

> "For all the public functions, the input parameters are copied to memory automatically, and it costs gas. If your function is only called externally, then you should explicitly mark it as external. External function's parameters are not copied into memory but are read from `calldata` directly. This small optimization in your solidity code can save you a lot of gas when the function input parameters are huge."

Recommend changing function visibility from public to external.

xyz-ctrl (Visor) confirmed:

ztcrypto (Visor) commented:

> patch link

[G-02] Unused state variable and associated setter function

Submitted by OxRajeev, also found by gpersoon

The uri state variable is never used anywhere but has an associated setter function `setURI()` . Removing the state variable and its associated setter function will save both storage slot and contract deployment cost because of reduced size.

Recommend removing unused state variable and associated setter function, or adding missing code to use them.

xyz-ctrl (Visor) disputed:

> sponsor disputed this is a feature

ghoul-sol (Judge) commented:

> I'm guessing that this is going to be used in the future or in some other creative way, however warden is right and without more context I don't see why this is needed.

ztcrypto (Visor) commented:

> path link

[G-03] Use a temporary variable to cache repetitive complex calculation

Submitted by OxRajeev, also found by gpersoon

In function `delegatedTransferERC20()` , the complex calculation `keccak256(abi.encodePacked(msg.sender, token))` is performed three times in three different places in the function. This consumes a lot of unnecessary gas which can be saved by saving the calculation in a temporary bytes32 variable and using that instead.

Recommend saving `keccak256(abi.encodePacked(msg.sender, token))` in a temporary bytes32 variable and use that in all places.

xyz-ctrl (Visor) confirmed:

ztcrypto (Visor) commented:

> patch link

[G-04] Use a temporary variable to cache repetitive storage reads

Submitted by 0xRajeev

In function `transferERC721()`, the array value stored in a mapping `timelockERC721Keys[nftContract][i]` is read three times in three different places within the loop iteration. This consumes a lot of unnecessary gas because SLOADs are expensive. This can be prevented by saving the value `timelockERC721Keys[nftContract][i]` in a temporary bytes32 variable at the beginning of the iteration and using that instead.

See issue page for proof of concept

Recommend saving the value `timelockERC721Keys[nftContract][i]` in a temporary bytes32 variable at the beginning of the iteration and using that instead

xyz-ctrl (Visor) confirmed:

ztcrypto (Visor) commented:

> path link

[G-05] Breaking out of loop can save gas

Submitted by OxRajeev

In function `transferERC721()` , the for loop iterates over all the time locked keys for the nftContract `timelockERC721Keys[nftContract].length` times. Given that there will only be a maximum of one `tokenID` that will match (because of unique NFT tokenIDs), if any, we can break from iterating the rest of the loop after a match on L505 and the checks within the if body. This will prevent iterating the rest of the loop and trying to match the if condition on L505 after a match has already happened.

Recommend adding a break statement after L510 within the if body.

xyz-ctrl (Visor) confirmed:

ztcrypto (Visor) commented:

> path link

[G-06] Gas optimizations by using external over public

Submitted by adelamo_

Using public over external has an impact on execution cost.

If we run the following methods on Remix, we can see the difference

```
    //  transaction cost    21448 gas
    //  execution cost      176 gas
    function tt() external returns(uint256) {
        return 0;
    }

    //  transaction cost    21558 gas
    //  execution cost      286 gas
    function tt_public() public returns(uint256) {
        return 0;
    }
```

See issue page for list of methods currently using public that should be declared external.

Recommend just changing from public to external if possible.

xyz-ctrl (Visor) confirmed:

ztcrypto (Visor) commented:

> duplicate of #27 and fixed

[G-07] Gas optimization storage NFTs

*Submitted by a*delamo_

In `Visor.sol` , NFTs are being stored using an array `Nft[] public nfts;`

This seems an optimal structure, but when needing to remove an NFT or look for an NFT by contract and id, we need to do `O(n)` iterations.

See issue page for referenced code.

Checking the function `getNftById` , seems like a lookup by id (mapping(uint=> NFT)) should be fine. In case we need more, we could do mapping(uint => mapping(address => NFT)), but doesn’t seems necessary.

Recommend providing direct links to all referenced code in GitHub. Add screenshots, logs, or any other relevant proof that illustrates the concept.

xyz-ctrl (Visor) confirmed:

ztcrypto (Visor) commented:

> patch link

[G-08] Gas optimizations - storage over memory

*Submitted by a*delamo_

In `Visor.sol`, the functions are using memory keyword, but using storage would reduce the gas cost.

```
    function _removeNft(address nftContract, uint256 tokenId) interr
        uint256 len = nfts.length;
        for (uint256 i = 0; i < len; i++) {
            Nft memory nftInfo = nfts[i];
            if (
                nftContract == nftInfo.nftContract && tokenId == nft
            ) {
                if (i != len - 1) {
                    nfts[i] = nfts[len - 1];
                }
                nfts.pop();
                emit RemoveNftToken(nftContract, tokenId);
                break;
            }
        }
    }
```

xyz-ctrl (Visor) confirmed:

ztcrypto (Visor) commented:

> patch link

[G-09] Gas optimizations - calculation `getBalanceLocked`

*Submitted by adelamo_*

In `Visor.sol`, the function `getBalanceLocked` returns the amount locked. Doing this method will cause O(n) to return the highest locked value.

But this method is only being used to verify that there is an X amount of tokens locked. (See issue page for referenced code)

Recommend that instead of doing O(n), we could just exit when we found that balance >= amount requested. Something like:

xyz-ctrl (Visor) confirmed:

ztcrypto (Visor) commented:

> patch link

[G-10] Missing events

Submitted by cmichel

The following events are not used:

- `IInstanceRegistry.InstanceRemoved`

Unused code can hint at programming or architectural errors. Recommend using it or removing it.

xyz-ctrl (Visor) acknowledged but disputed severity:

ghoul-sol (Judge) commented:

> Agree with sponsor, it doesn't present a security issue it's a non-critical issue.

ztcrypto (Visor) commented:

> patch link

[G-11] `getNftById` is querying against the index not id

Submitted by pauliax

`getNftById` is actually 'get NFT by index' as it queries the element from the array by index, not by `tokenId`. The index may not always equal id as `_addNft` does not automatically assign index incrementally but rather use a parameter's value. Same with `getNftIdByTokenIdAndAddr`, it returns index, not token id.

Recommend either renaming functions to distinguish between id and index or refactoring the function to suit its name.

xyz-ctrl (Visor) acknowledged:

ghoul-sol (Judge) commented:

> This is very similar to #26 but I'll keep it as sponsor finds it a valuable suggestion

ztcrypto (Visor) commented:

> fixed patch link

[G-12] introduce a max lock time limit

Submitted by pauliax

I suggest introducing a max lock time limit, so it won't be possible to accidentally lock tokens forever. As of right now there is no limit on when the timelock expires, so theoretically it is possible to set it to hundreds of years which I think in practice wouldn't make sense.

Even though this is more of a theoretical issue, I recommend introducing a reasonable upper limit for the timelock period.

xyz-ctrl (Visor) acknowledged:

ztcrypto (Visor) commented:

> this is not critical for now by adding additional check

[G-13] Internal `GetBalanceLocked` call can exceed block size limit

Submitted by Sherlock

On line 202 it loops over all the Lock sets. Transaction can run out of gas if this is an extreme size. Precautions have already been taken by restricting adding entries to this array.

Recommend keeping an internal accounting of the `balanceLocked`, updating on every lock call.

[xyz-ctrl (Visor) acknowledged]():

> In practice of our platform context this will never grow large

[ztcrypto (Visor) commented]():

> patch [link]()

[G-14] Events are not indexed

Submitted by shw

The emitted events are not indexed, making off-chain scripts such as front-ends of dApps difficult to filter the events efficiently.

Recommend adding the `indexed` keyword in each event, e.g., `event AddNftToken(address indexed nftContract, uint256 tokenId);`.

[xyz-ctrl (Visor) acknowledged]():

[ghoul-sol (Judge) changed severity]():

> While true, it’s non-critical issue

[ztcrypto (Visor) commented]():

> patch [link]()

[G-15] Unused imported interface `IVisorService`

Submitted by shw

The imported interface `IVisorService` in the contract `Visor` is not used.

Recommend considering removing this import.

xyz-ctrl (Visor) acknowledged:

> sponsor acknowledged, dispute severity 0

ghoul-sol (Judge) commented:

> Agreed, non-critical issue

ztcrypto (Visor) commented:

> patch link

Disclosures