FairSide
Findings & Analysis Report

2021-07-07

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 FairSide smart contract system written in Solidity. The code contest took place between May 20 and May 26, 2021.

Wardens

8 Wardens contributed reports to the FairSide code contest:

- cmichel
- shw
- 0xRajeev
- a_delamo
- Thunder
- gpersoon
- Jmukesh
- s1m0

This contest was judged by Cem.

Final report assembled by ninek and moneylegobatman.

Summary

The C4 analysis yielded an aggregated total of 45 unique vulnerabilities. All of the issues presented here are linked back to their original finding.

Of these vulnerabilities, 8 received a risk rating in the category of HIGH severity, 12 received a risk rating in the category of MEDIUM severity, and 7 received a risk rating

in the category of LOW severity.

C4 analysis also identified 18 non-critical recommendations.

Scope

The code under review can be found within the C4 code contest repository and comprises 21 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] Conviction scoring fails to initialize and bootstrap

Conviction scores for new addresses/users fail to initialize+bootstrap in `ERC20ConvictionScore`'s `_updateConvictionScore()` because a new user's `numCheckpoints` will be zero and never gets initialized.

This effectively means that FairSide conviction scoring fails to bootstrap at all, leading to the failure of the protocol's pivotal feature.

When Alice transfers FSD tokens to Bob for the first time, `_beforeTokenTransfer(Alice, Bob, 100)` is triggered which calls `_updateConvictionScore(Bob, 100)` on Line55 of ERC20ConvictionScore.sol.

In function `_updateConvictionScore()`, given that this is the first time Bob is receiving FSD tokens, `numCheckpoints[Bob]` will be 0 (Line116) which will make `ts = 0` (Line120), and Bob's FSD balance will also be zero (Bob never has got FSD tokens prior to this) which makes `convictionDelta = 0` (Line122) and not let control go past Line129.

This means that a new checkpoint never gets written, i.e., conviction score never gets initialized, for Bob or for any user for that matter.

FairSide's adjustment of Compound's conviction scoring is based on time and therefore needs an initialization to take place vs Compound's implementation. Therefore, a new checkpoint needs to be created+initialized for a new user during token transfer.

fairside-core (FairSide) confirmed:

> Fixed in **PR#18**.

[H-02] Locked funds are debited twice from the user during tokenization leading to fund loss

During tokenization of conviction scores, the user can optionally provide FSDs to be locked to let it continue conviction accrual. However, the amount of FSDs specified for locking are debited from the user twice, leading to fund loss.

This, in effect, forces the user to unknowingly and unintentionally lock twice the amount of FSD tokens, leading to a loss of the specified 'locked' number of tokens.

Alice decides to tokenize her conviction score into an NFT and specifies 100 FSD tokens to be locked in her call to `tokenizeConviction(100)`. 100 FSD tokens are transferred from her FSD balance to `FairSideConviction` contract on Line282 of

`ERC20ConvictionScore.sol`. However, in `FairSideConviction.createConvictionNFT()`, the specified locked amount is transferred again from Alice to the contract on Line50 of `FairSideConviction.sol`.

The impact is that Alice wanted to lock only 100 FSD tokens, but the FairSide protocol has debited 200 tokens from her balance leading to a loss of 100 FSD tokens.

Recommend removing the redundant transfer of FSD tokens on Line282 in `tokenizeConviction()` of `ERC20ConvictionScore.sol`.

[H-03] Locked funds from tokenization are credited twice to user leading to protocol fund loss

The tokens optionally locked during tokenization are released twice on acquiring conviction back from an NFT. (The incorrect double debit of locked funds during tokenization has been filed as a separate finding because it is not necessarily related and occurs in different parts of the code.)

When a user wants to acquire back the conviction score captured by an NFT, the FSD tokens locked, if any, are released to the user as well. However, this is incorrectly done twice. Released amount is transferred once on Line123 in `_release()` (via `acquireConviction -> burn`) of FairSideConviction.sol and again immediately after the burn on Line316 in `acquireConviction()` of `ERC20ConvictionScore.sol`.

This leads to loss of protocol funds.

Alice tokenizes her conviction score into an NFT and locks 100 FSDs. Bob buys the NFT from Alice and acquires the conviction score back from the NFT. But instead of 100 FSDs that were supposed to be locked with the NFT, Bob receives 100+100 = 200 FSDs from FairSide protocol.

Recommend removing the redundant transfer of FSD tokens from protocol to the user on Line316 in `acquireConviction()` of `ERC20ConvictionScore.sol`.

fairside-core (FairSide) questioned:

> This is directly related to #29 as it refers to the same workflow, as seen in #74 as a single submission. I believe splitting this into two findings is unfair for the first party, and secondly, it does not make sense because there is a valid argument for disagreeing with the severity seen on #74. Can we close this and merge it with #29?

fairside-core (FairSide) commented:

> Fixed in **PR#3**.

cemozerr (Judge) commented:

> Labeling issues #29 and #30 as separate issues because they both pose major issues, which lead to temporary loss of funds, in two different workflows. One when tokenizing convictions and another when acquiring convictions.

[H-04] `ERC20ConvictionScore`'s `governanceDelta` should be subtracted when user is not a governor anymore

The `TOTAL_GOVERNANCE_SCORE` is supposed to track the sum of the credit scores of all governors.

In `ERC20ConvictionScore._updateConvictionScore`, when the user does not fulfill the governance criteria anymore and is therefore removed, the `governanceDelta` should be negative, but it's positive.

```
    isGovernance[user] = false;
    governanceDelta = getPriorConvictionScore(
        user,
        block.number - 1
    );
```

It then gets added to the new total:

```
    uint224 totalGCSNew =
        add224(
            totalGCSOld,
```

```
            governanceDelta,
            "ERC20ConvictionScore::_updateConvictionTotals: convicti
        );
```

The `TOTAL_GOVERNANCE_SCORE` tracks wrong data leading to issues throughout all contracts like wrong `FairSideDAO.totalVotes` data, which can then be used by anyone to pass proposals in the worst case.

Or `totalVotes` can be arbitrarily inflated and break the voting mechanism as no proposals can reach the quorum (percentage of `totalVotes`) anymore.

Recommend returning a negative signed integer for this case and adding it to the new total.

fairside-core (FairSide) confirmed:

> Fixed in PR#14.

[H-05] `Withdrawable.withdraw` does not decrease `pendingWithdrawals`

The name `pendingWithdrawals` indicates that this storage variable tracks the withdrawals that need yet to be paid out. Furthermore, this matches the behavior in `_increaseWithdrawal`. As such, it should be decreased when withdrawing in `withdraw`, but it is not.

The `getReserveBalance` function consistently under-reports the actual reserve balance, which leads to the wrong mint amounts being used in the `FSD.mint` calculation.

Recommend decreasing `pendingWithdrawals` by the withdrawn amount.

fairside-core (FairSide) confirmed (in duplicate issue #72)

> One of two easter eggs! Fixed in PR#5.

[H-06] Incorrect type conversion in the contract `ABC` makes users unable to burn FSD tokens

The function `_calculateDeltaOfFSD` of contract `ABC` incorrectly converts an `int256` type parameter, `_reserveDelta`, to `uint256` by explicit conversion, which in general results in an extremely large number when the provided parameter is negative. The extremely large number could cause a SafeMath operation `sub` at line 43 to revert, and thus the FSD tokens cannot be burned as `_reserveDelta` is negative when burning FSD tokens.

Simply calling `fsd.burn` after a successful `fsd.mint` will trigger this bug.

Recommend using the solidity function `abs` to get the `_reserveDelta` absolute value.

fairside-core (FairSide) confirmed:

> Fixed in PR#1.

[H-07] `ERC20ConvictionScore._updateConvictionScore` uses stale credit score for `governanceDelta`

In `ERC20ConvictionScore._updateConvictionScore`, when the user does not fulfill the governance criteria anymore, the `governanceDelta` is the old conviction score of the previous block.

```
    isGovernance[user] = false;
    governanceDelta = getPriorConvictionScore(
        user,
        block.number - 1
    );
```

The user could increase their conviction/governance score first (in the same block) and then lose their status in a second transaction. After which, the total governance conviction score would only be reduced by the previous score.

** Example:** Block n - 10000: User is a governor and has a credit score of 1000, which was also contributed to the `TOTAL_GOVERNANCE_SCORE` Block n:

- User updates their own conviction score using the public `updateConvictionScore` function, which increases the credit score by 5000 based on the accumulated time. The total governance credit score increased by 5000, making the user contribute 6000 credit score to governance in total.
- User transfers their whole balance away, the balance drops below `governanceMinimumBalance`, and the user is not a governor anymore. The `governanceDelta` update of the transfer should be 6000 (user's whole credit score), but it's only `1000` because it takes the snapshot of block n - 1.

In this way, the `TOTAL_GOVERNANCE_SCORE` score can be inflated and, in the worst case, break the voting mechanism, as no proposals can reach the quorum (percentage of `totalVotes`) anymore.

Recommend using the current conviction store which should be `governanceDelta = checkpoints[user][userCheckpointsLength - 1].convictionScore`.

fairside-core (FairSide) confirmed but disagreed with severity:

> As with the other governance-related issues, this would once again cause dilution of all users and would not really be a viable attack vector. As such, I believe it is better suited for a medium severity (2) label.

fairside-core (FairSide) disputed:

> This issue is actually quite deeper. When a transaction occurs in the same block, the logic paths within the `if` block will not execute (due to time elapsed being 0), meaning that the conviction score will not be properly accounted for if I have a single normal transaction where I am still governance and consequently lose my governance in a second transaction. As such, the code needs to be adjusted to check governance eligibility outside of the if block as well (if no time has passed -> same block transaction).

> The code highlighted in the finding is actually **correct**. The conviction score should be reduced by the previous blocks as the newly accrued conviction score

> was never accounted for in governance. The proposed solution would lead to more conviction being reduced than it should. However, the finding did point out something wrong, so not sure whether it should be nullified or not.

> I believe it should be awarded as it was on the right track to find the underlying issue!

fairside-core (FairSide) re-confirmed:

> Fixed in **PR#13**.

cemozerr (Judge) commented:

> Labeling this issue as valid because although it wasn't 100% right on suggesting where the code was problematic, it did point out that the users could wrongfully transfer their whole balance and update their conviction score in the same block to keep their conviction score high, and then potentially do harmful things to the protocol by using their wrong conviction scores.

[H-08] Incorrect implementation of arctan in the contract `FairSideFormula`

The current implementation of the arctan formula in the contract `FairSideFormula` is inconsistent with the referenced paper and could cause incorrect results when the input parameter is negative. The erroneous formula affects the function `calculateDeltaOfFSD` and the number of FSD tokens minted or burned.

The function `_arctan` misses two `abs` on the variable à'. The correct implementation should be:

```
    function _arctan(bytes16 a) private pure returns (bytes16) {
        return
            a.mul(PI_4).sub(
                a.mul(a.abs().sub(ONE)).mul(APPROX_A.add(APPROX_B.mu
            );
    }
```

Notice that `_arctan` is called by `arctan`, and `arctan` is called by `arcs` with `ONE.sub(arcInner)` provided as the input parameter. Since `arcInner = MULTIPLIER_INNER_ARCTAN.mul(x).div(fS3_4)` can be a large number (recall that `x` is the capital pool), it is possible that the parameter `a` is negative.

Recommend modifying the `_arctan` function as above.

fairside-core (FairSide) confirmed:

> Fixed in PR#4.

Medium Risk Findings

[M-01] Incorrect use of `_addTribute` instead of `_addGovernanceTribute`

As part of the `purchaseMembership()` function, the `addRegistrationTributeGovernance()` function is called by the FSD network to update tribute when 7.5% is contributed towards governance. However, this function incorrectly calls `_addTribute()` (as is also done in `addRegistrationTribute`) instead of `_addGovernanceTribute()`.

The impact of this is that `governanceTributes` never gets updated, rendering all of the tribute accounting logic incorrect.

Recommend using `_addGovernanceTribute()` instead of `_addTribute` on L140 of FSD.sol

fairside-core (FairSide) confirmed:

> The second of the two easter eggs!

fairside-core (FairSide) commented:

> Fixed in PR#20.

[M-02] Call to `swapExactTokensForETH` in `liquidateDai()` will always fail

`liquidateDai()` calls Uniswap's `swapExactTokensForETH` to swap Dai to ETH. This will work if `msg.sender` (i.e., the FSD contract) has already given the router an allowance amount that is at least as much as the input token Dai.

Given that there is no prior approval, the call to UniswapV2 router for swapping will fail. This is because `msg.sender` has not approved UniswapV2 with an allowance for the tokens that are attempting to be swapped.

The impact is that, while working with the Dai stablecoin, `updateCostShareRequest()` will fail and revert.

Recommend adding FSD approval to UniswapV2 with an allowance for the tokens that are attempting to be swapped.

fairside-core (FairSide) confirmed:

> Fixed in PR#19.

[M-03] Conviction totals not updated during tokenization

`_updateConvictionScore()` function returns `convictionDelta` and `governanceDelta` which need to be used immediately in a call to `_updateConvictionTotals (convictionDelta, governanceDelta)` for updating the conviction totals of conviction and governance-enabled conviction for the entire FairSide network.

This updating of totals after a call to `_updateConvictionScore()` is done on Line70 in `_beforeTokenTransfer()` and on Line367 in `updateConvictionScore()` of ERC20ConvictionScore.sol.

However, the return values of `_updateConvictionScore()` are ignored on Line284 in `tokenizeConviction()` and are not used to update the totals using `_updateConvictionTotals(convictionDelta, governanceDelta)`.

The impact of this is that when a user tokenizes their conviction score, their conviction deltas are updated and recorded (only if the funds locked are zero, which is incorrect and reported separately in a different finding), but the totals are not updated. This leads to incorrect accounting of `TOTAL_CONVICTION_SCORE` and `TOTAL_GOVERNANCE_SCORE`, which are used to calculate tributes, and therefore will lead to incorrect tribute calculations.

EXAMPLE:

> Alice calls `tokenizeConviction()` to convert her conviction score into an NFT. Her conviction deltas (as returned by `_updateConvictionScore()`) are ignored. Furthermore, `TOTAL_CONVICTION_SCORE` and `TOTAL_GOVERNANCE_SCORE` values are not updated. As a result, the tributes rewarded are proportionally more than what they should have been. This is because the conviction score totals are used as the denominator in `availableTribute()` and `availableGovernanceTribute()`.

Recommend using the return values of the `_updateConvictionScore()` function (i.e. `convictionDelta` and `governanceDelta`) on Line284 of `ERC20ConvictionScore.sol`, and then use them in a call to `_updateConvictionTotals(convictionDelta, governanceDelta)`.

fairside-core (FairSide) confirmed:

> Fixed in PR#17.

[M-04] Eth may get stuck in contract

The Istanbul hardfork increases the gas cost of the SLOAD operation and therefore breaks some existing smart contracts.

In file `withdrawable.sol`, contract uses `transfer()` to send eth from contract to EOA due which eth can get stuck.

The reason behind this is that, after the Istanbul hardfork, any smart contract that uses `transfer()` or `send()` is taking a hard dependency on gas costs by

forwarding a fixed amount of gas (2300). This forwards 2300 gas, which may not be enough if the recipient is a contract and the cost of gas changes.

Recommend using `call()` to send eth.

fairside-core (FairSide) confirmed (separately in issue #67):

> Although I am fine with the severity, perhaps it may not be applicable given that even after EIP-3074, transfers will not fail with proper access lists, and I highly doubt the transfer method will fail to work altogether anytime soon.

> Fixed in PR#8.

[M-05] Bug inside ABDKMathQuad library

The `FairSideFormula` library is using the `ABDKMathQuad` library underneath. According to the `ABDKMathQuad` README, the range of values is the following:

> The minimum strictly positive (subnormal) value is 2^−16494 ≈ 10^−4965 and has a precision of only one bit. The minimum positive normal value is 2^−16382 ≈ 3.3621 × 10^−4932 and has a precision of 113 bits, i.e., ±2^−16494 as well. The maximum representable value is 2^16384 − 2^16271 ≈ 1.1897 × 10^4932.

Using Echidna, a fuzzing tool for smart contracts, I found some edge cases in which some of the operations do not work as expected. This is the test code I ran using `echidna-test contracts/TestABDKMathQuad --contract TestABDKMathQuad`. see issue for more details.

If we check in Remix, we can see that there is a small difference when converting from UInt to Bytes16 (and vice versa). This issue is probably the same with all the other operations.

Recommend using some fuzzing tool like Echidna to verify that there are no edge cases.

fairside-core (FairSide) acknowledged:

> I am slightly mixed about this finding. We did employ fuzz tests during the audit we had gone through, and they were unable to pinpoint any issues in the value range we expect the curve to be utilized in. This is definitely a good suggestion and one we will assimilate. However, I am not sure how one would judge the severity of this.

[M-06] pendingWithdrawals just increments

Sponsor commented that this related to another bug and referenced "[H-05] `Withdrawable.withdraw` does not decrease `pendingWithdrawals`" see issue **#48** for more details.

[M-07] NFTs can never be redeemed back to their conviction scores leading to lock/loss of funds

Besides the conviction scores of users, there appears to be tracking of the FairSide protocol's tokenized conviction score as a whole (using `fscAddress = address(fairSideConviction)`). This is evident in the attempted reduction of the protocol's score when a user acquires conviction back from an NFT. However, the complementary accrual of the user's conviction score to `fscAddress` when the user tokenizes their conviction score to mint an NFT is missing in `tokenizeConviction()`.

Because of this missing update of the conviction score to `fscAddress` upon tokenization, there are no checkpoints written for `fscAddress`. There also doesn't appear to be any initialization for bootstrapping this address's conviction score checkpoints. As a result, the `sub224()` on Line350 of `ERC20ConvictionScore.sol` will always fail with an underflow. This is because `fscOld = 0` (because `fscNum = 0`) and `convictionScore > 0`, effectively reverting all calls to `acquireConviction()`.

The impact of this is that all tokenized NFTs can never be redeemed back to their conviction scores leading to a lock/loss of FSD funds for users who tokenized/sold/bought FairSide NFTs.

Proof of Concept:

1. Alice tokenizes her conviction score into an NFT. She sells that NFT to Bob, who pays an amount commensurate with the conviction score captured by that NFT (as valued by the market) and any FSDs locked with the NFT.
2. Bob then attempts to redeem the bought NFT back to the conviction score to use it on the FairSide network. But the call to `acquireConviction()` fails. Bob is never able to redeem Alice's NFT and has lost the funds used to buy it.

Recommend adding appropriate logic to bootstrap, initialize `fscAddress` 's tokenized conviction score checkpoints, and update it during tokenization.

fairside-core (FairSide) confirmed:

> Although the finding is correct, FSDs will not be permanently locked in the NFT as they can still be redeemed via the dedicated `release` function on the conviction NFT implementation. As such, I would label this a medium-level finding, given that the conviction scores will indeed be lost.

fairside-core (FairSide) commented:

> Fixed in **PR#16**.

cemozerr (Judge) commented:

> Labeling this as medium risk as FSDs will not be permanently locked.

[M-08] `ERC20ConvictionScore` allows transfers to special `TOTAL_GOVERNANCE_SCORE` address

The credit score of the special `address(type(uint160).max)` is supposed to represent the sum of the credit scores of all users that are governors.

But, any user can directly transfer to this address, increasing its balance and accumulating a credit score in `_updateConvictionScore(to=address(uint160.max), amount)`.

It'll first write a snapshot of this address' balance, which should be very low:

```
// in _updateConvictionScore
_writeCheckpoint(user, userNum, userNew) = _writeCheckpoint(TOTA
```

This address then accumulates a score based on its balance, which can be updated using `updateConvictionScore(uint160.max)` and breaks the invariant.

Increasing it might be useful for non-governors that don't pass the voting threshold and want to grief the proposal voting system by increasing the `quorumVotes` threshold required for proposals to pass. By manipulating `FairSideDAO.totalVotes`, `totalVotes` can be arbitrarily inflated and break the voting mechanism as no proposals can reach the quorum (percentage of `totalVotes`) anymore.

Recommend disallowing transfers from/to this address. Or better, track the total governance credit score in a separate variable, not in an address.

fairside-core (FairSide) confirmed:

> This is actually what #61 is meant to be used for. I would label this a medium-level finding as it would simply dilute the voting rights of users at the expense of permanently losing FSD, which should not be a viable vector.

fairside-core (FairSide) resolved:

> Indirectly fixed by **PR#10**.

cemozerr (Judge) commented:

> Labeling this as medium risk as it would not pose a threat to user funds yet stall the governance process.

[M-09] Should check return data from Chainlink aggregators

The `getEtherPrice` function in the contract `FSDNetwork` fetches the ETH price from a Chainlink aggregator using the `latestRoundData` function. However, there are no checks on `roundID` nor `timeStamp`, resulting in stale prices.

Recommend adding checks on the return data with proper revert messages if the price is stale or the round is incomplete, for example:

```
(uint80 roundID, int256 price, , uint256 timeStamp, uint80 answe
require(answeredInRound >= roundID, "...");
require(timeStamp != 0, "...");
```

fairside-core (FairSide) confirmed:

> Fixed in **PR#7**.

cemozerr (Judge) commented:

> Labeling this as medium risk as stale ether price could put funds at risk.

[M-10] `gracePeriod` not increased after membership extension

In the function `purchaseMembership` of FSDNetwork.sol, when the membership is extended, `membership[msg.sender].creation` is increased. However, `membership[msg.sender].gracePeriod` is not increased. This might lead to a `gracePeriod` that is lower than expected. It seems logical to also increase the `gracePeriod`.

FSDNetwork.sol:

```
function purchaseMembership(uint256 costShareBenefit) external {
    ...
     if (membership[msg.sender].creation == 0) {
           ...
           membership[msg.sender].creation       = block.timest
           membership[msg.sender].gracePeriod =  membership[msg
       } else {
         ....
         membership[msg.sender].creation += durationIncrease;
  }
```

Recommend checking to see if `gracePeriod` has to be increased and then adding the necessary logic when that is the case.

fairside-core (FairSide) confirmed:

> This should be bumped to a medium severity finding as it actually does not affect the membership duration at all if the `gracePeriod` is not updated.

fairside-core (FairSide) commented:

> Fixed in PR#21.

[M-11] The variable `fShareRatio` is vulnerable to manipulation by flash minting and burning

The variable `fShareRatio` in the function `purchaseMembership` of contract `FSDNetwork` is vulnerable to manipulation by flash minting and burning, which could affect several critical logics, such as the check of enough capital in the pool (line 139-142) and the staking rewards (line 179-182).

The `fShareRatio` is calculated (line 136) by:

```
(fsd.getReserveBalance() - totalOpenRequests).mul(1 ether) / fSh
```

Where `fsd.getReserveBalance()` can be significantly increased by a user minting a large amount of FSD tokens with flash loans. In that case, the increased `fShareRatio` could affect the function `purchaseMembership` results. For Example, the user could purchase the membership even if the `fShareRatio` is < 100% previously, or the user could earn more staking rewards than before to reduce the membership fees. Although performing flash minting and burning might not be profitable overall since a 3.5% tribute fee is required when burning FSD tokens, it is still important to be aware of the possible manipulation of `fShareRatio`.

Recommend forcing users to wait for (at least) a block to prevent flash minting and burning.

fairside-core (FairSide) confirmed:

> I believe this to be a minor (1) or none (0) severity issue given that the manipulation of `fShareRatio` is unsustainable due to the fee, and the Example given is actually not possible. Suppose I affect `fShareRatio` to go above 100% to purchase a membership. In that case, I will be unable to burn the necessary FSD to go below 100% again as burning is disabled when the ratio is or would go to below 100%.

fairside-core (FairSide) resolved:

> Fixed in PR#2.

cemozerr (Judge) commented:

> Labeling this as low risk as a 3.5% tribute fee makes it very unlikely that these flash minting will be profitable.

[M-12] `ERC20ConvictionScore.acquireConviction` implements wrong governance checks

There are two issues with the governance checks when acquiring them from an NFT:

(Issue 1) Missing balance check

The governance checks in `_updateConvictionScore` are:

```
    !isGovernance[user]
    && userConvictionScore >= governanceThreshold
    && balanceOf(user) >= governanceMinimumBalance;
```

Whereas in `acquireConviction`, only `userConvictionScore >= governanceThreshold` is checked but not `&& balanceOf(user) >= governanceMinimumBalance`.

```
    else if (
        !isGovernance[msg.sender] && userNew >= governanceThreshold
    ) {
```

```
        isGovernance[msg.sender] = true;
    }
```

(Issue 2) the `wasGovernance` might be outdated

The second issue is that at the time of NFT creation, the `governanceThreshold` or `governanceMinimumBalance` was different and would not qualify for a governor now. The NFT's governance state is blindly appplied to the new user:

```
    if (wasGovernance && !isGovernance[msg.sender]) {
        isGovernance[msg.sender] = true;
    }
```

This allows a user to circumvent any governance parameter changes by front-running the change with an NFT creation. It's easy to circumvent the balance check to become a governor by minting and redeeming your own NFT. One can also circumvent any governance parameter increases by front-running these actions with an NFT creation and then backrunning with a redemption.

Recommend adding the missing balance check-in `acquireConviction`, removing the `wasGovernance` governance transfer from the NFT, and recomputing it based solely on the current `governanceThreshold` / `governanceMinimumBalance` settings.

fairside-core (FairSide) confirmed:

> The latter of the two issue "types" is actually desired behavior. If a user was historically a governance member, the NFT should boast the exact same rights, and new thresholds should not retroactively apply. The former, however, is a valid issue as it allows circumventing the balance check!

fairside-core (FairSide) resolved:

> Fixed in **PR#12**.

Low Risk Findings

[L-01] Lack of zero-address checks for immutable addresses will force contract redeployment if zero-address used accidentally

Zero-address checks as input validation on address parameters are always a best practice. This is especially true for critical addresses that are immutable and set in the constructor because they cannot be changed later. Accidentally using zero addresses here will lead to failing logic or force contract redeployment and increased gas costs.

Recommend adding zero-address input validation for these addresses in the constructor.

fairside-core (FairSide) acknowledged (in separate issue #56):

> Adding the respective require checks significantly increases the bytecode size of the contract, and all relate to privileged functions (constructor functions or functions voted on by the DAO). As such, I believe this to be a non-critical (0) issue.

[L-02] Dangerous Solidity compiler pragma range that spans breaking versions

All contracts use a Solidity compiler pragma range >=0.6.0 <0.8.0, which spans a breaking change version 0.7.0. This compiler range is very broad and includes many syntactic/semantic changes across the versions. Specifically, see silent changes in https://docs.soliditylang.org/en/v0.7.0/070-breaking-changes.html#silent-changes-of-the-semantics.

For Example, this compiler range allows testing with Solidity compiler version 0.6.x but deployment with 0.7.x. While any breaking syntactic changes will be caught at compile time, there is a risk that the silent change in 0.7.0, which applies to exponentiation/shift operand types, might affect the FairSide formula or other mathematical calculations, thus breaking assumptions and accounting.

The opposite scenario may also happen where testing is performed with Solidity compiler version 0.7.x but deployed with 0.6.x, which may allow bugs fixed in 0.7.x to be present in the deployed code.

Recommend using the same compiler version both for testing and deployment by enforcing this in the pragma itself. An unlocked/floating pragma is risky, especially one that ranges across a breaking compiler minor version.

fairside-core (FairSide) acknowledged (in separate Issue #66):

> he pragma statements were left unlocked to allow flexibility in development. Since this is not a functional finding, it should be marked as 0 (non-critical).

cemozerr (Judge) commented (in separate Issue #66):

> Duplicate of #25. Labeling it as low risk as it could indeed cause the contracts to accidentally be compiled or deployed using an outdated or buggy compiler version

[L-03] Usage of transfer

In `Withdrawable.withdraw` : The `address.transfer` function is used to send ETH to an account. It is restricted to a low amount of gas and might fail if gas costs change in the future or if a smart contract's fallback function handler implements anything non-trivial.

Recommend considering using the lower-level `.call{value: value}` instead and checking its success return value.

fairside-core (FairSide) confirmed and commented (in separate issue #67):

> Although I am fine with the severity, perhaps it may not be applicable given that even after EIP-3074 transfers will not fail with proper access lists, and I highly doubt the transfer method will fail to work altogether anytime soon. Fixed in PR#8.

[L-04] Missing use of DSMath functions may lead to underflows/overflows

The FairSide contracts use DappHub's DSMath safe arithmetic library that provides overflow/underflow protection. But, the safe DSMath functions are not used in many places, especially in the FSD `mint` / `burn` functions.

While there do not appear to be any obvious integer overflows/underflows in the conditions envisioned, there could be exceptional paths where overflows/underflows may be triggered, leading to minting/burning an unexpected number of tokens.

Recommend using DSMath `add` / `sub` functions instead of +/- in all places.

fairside-core (FairSide) acknowledged:

> All linked segments are guaranteed not to overflow / underflow. In detail:
>
> 1. The `getReserveBalance` always takes into account the actual balance of the contract, which will always be greater-than-or-equal to `msg.value`.
> 2. The `bonded` amount is always a percentage of `msg.value`
> 3. The `tribute` amount is always a percentage of `capitalDesired`
> 4. The `reserveWithdrawn` will always be less than or equal to `etherBalanceAtBurn`
>
> Due to the above, I would label the finding as non-critical. In general, SafeMath utilization is avoided in any case that it can be to reduce gas costs.

cemozerr (Judge) commented:

> Labeling this as low risk as not using dsmath might lead to exceptional paths where overflows/underflows may be triggered, even if those paths are not enumerated above.

[L-05] `convictionless` mapping is not used

`convictionless` can be set via function `setConvictionless`; however, it is not used anywhere across the system, thus making it useless. Based on the comment above this variable, I expect to see it used in functions like `_updateConvictionScore`.

Recommend either remove this mapping or use it where intended.

fairside-core (FairSide) confirmed:

> Quite strange no one else identified this one! The absence of usage was a merging mistake; this particular mapping is slightly important to the overall operation of FairSide as certain parties should not accrue conviction, such as the Governance wallet. I believe it should be increased to medium-level severity.

fairside-core (FairSide) resolved:

> Fixed in **PR#10**.

[L-06] Flash minting and burning can reduce the paid fees when purchasing a membership or opening a cost-share request

Users can pay fewer FSD tokens when purchasing a membership or opening a cost-share request by flash minting and burning FSD tokens, which could significantly affect the FSD spot price.

The function `getFSDPrice` returns the current FSD price based on the reserves in the capital pool (see lines 353-364 in contract `FSDNetwork`). Notice that when minting and burning FSD tokens, the `fsd.getReserveBalance()` increases but not the `fShare`. Therefore, according to the pricing formula, `FairSideFormula.f`, the FSD price increases when minting, and vice versa, decreases when burning.

When purchasing a membership, the number of FSD tokens that a user should pay is calculated based on the current FSD price, which is vulnerable to manipulation by flash minting and burning. Consider a user performing the following actions (all are done within a single transaction or flashbot bundle):

1. The user mints a large number of FSD (by using flash loans) to raise the current FSD price.
2. The user purchases a membership by calling `purchaseMembership`. Since the price of FSD is relatively high, the user pays fewer FSD tokens for the membership fee than before.
3. The user burns the previously minted FSD tokens, losing 3.5% of his capital for the tribute fees.

Although the user pays for the 3.5% tribute fees, it is still possible to make a profit. Suppose that the price of FSD to ETH is `p_1` and `p_2` before and after minting, respectively. The user purchases a membership with `x` ETH `costShareBenefit` and uses `y` ETH to flash mint the FSD tokens. In a regular purchase, the user pays `0.04x / p_1` FSD tokens, equivalent to `0.04x` ETH. By performing flash mints and burns, the user pays `0.04x / p_2` FSD tokens, which is, in fact, equivalent to `0.04x * p_1 / p_2` ETH. He also pays `0.035y` ETH for tribute fees. The profit user made is `0.04x * (1 - p1 / p2) - 0.035y` (ETH), where `p2` and `y` are dependent to each other but independent to `x`. Thus, the profit can be positive if `costShareBenefit` is large enough.

The same vulnerability exists when a user opens a cost-share request, where the `bounty` to pay is calculated based on the current price of FSD tokens.

Recommend forcing users to wait for (at least) a block to prevent flash minting and burning.

fairside-core (FairSide) questioned:

> The issue relies on `costShareBenefit` being large enough, which is inherently limited to a % of the capital pool, meaning that the arbitrage opportunity present here is inexistent or highly unlikely to be beneficial. Can we reach out to the submitter to request them to prove that even with the `costShareBenefit` % limit, this is a sustainable attack by providing us with numbers? If no such numbers are present, I would decrease the severity of this to either minor (1) or none (0).

cemozerr (Judge) commented:

> Will wait for a proof from the auditor, shw, for this one.

x9453 commented:

> Hi, thanks for giving me a chance to clarify this finding.

> After realizing that a user's `costShareBenefit` is limited to a % of the capital pool (5% as specified in the code), I would say this attack is not successful according to the following estimation of the upper-bound of user's profit:

```
  User's profit
 = 0.04x * (1 - p1 / p2) - 0.035y
<= 0.04x - 0.035y
<= 0.04 * 0.05 * (z + y) - 0.035y
 = 0.002z - 0.033y
```

> where `z` is the amount of ETH in the capital pool before minting. A negative coefficient of ỳ' implies that using a flash loan does not help to increase the profit but to decrease it.

x9453 commented:

> After some thoughts, I think the estimation should also consider how flash loan affects on the FSD's price to be more accurate. According to the price formula, we have `p1 = A + z^4 / (C * fShare^3)` and `p2 = A + (z + y)^4 / (C * fShare^3)` (assuming the best case, where `fShare` does not increase). Let `r = y / z`, the ratio of flash loan to the capital pool, then we can approximate `p1 / p2 = z^4 / (z + y)^4 = 1 / (r + 1)^4`. Therefore,

```
  User's profit
= 0.04x * (1 - p1 / p2) - 0.035y
= 0.04 * 0.05 * (z + y) * (1 - 1 / (r + 1)^4) - 0.035y
= (0.002 * (r + 1) * (1 - 1 / (r + 1)^4) - 0.035r) * z
= f(r) * z
```

> WolframAlpha tells us that `f(r) < 0` for all `r > 0`, meaning that the user does not make a profit no matter how much flash loan he borrowed.

> It is worth mentioning that different % of withdrawal fee, cost share benefit limit, and tribute fee could lead to different results. That is, the constants, `0.002` and `0.035`, determine whether user's profit can be positive (i.e., there exists `r > 0` s.t. `f(r) > 0`). Further calculation shows that this happens if the product of the withdrawal fee and cost share benefit limit is greater than the tribute fee divided by 4, which is unlikely in normal settings. Please let me know if you need more details or a PoC on this.

[L-07] Check if variables are initialized

A variable named `fairSideConviction` is set in the contract FSD function `setFairSideConviction`. However, functions that use this variable do not check that it is already initialized. For example, function `tokenizeConviction` in contract `ERC20ConvictionScore` may transfer tokens to the 0x0 address:

```
_transfer(msg.sender, address(fairSideConviction), locked);
```

This will make these tokens inaccessible and basically burned. It would be better if the code explicitly checked before that `address(fairSideConviction) != address(0)`

Rating this as low because I expect that, in practice, these variables will be initialized as soon as possible.

Also, this may be an additional small issue. Still, I think it would make sense if functions `setFairSideConviction` and `setFairSideNetwork` explicitly check that the parameter is not 0x0 address as it is theoretically possible to invoke these functions again and again when the address is empty.

Recommend requiring `address(fairSideConviction) != address(0)` where this variable is used. Same can be applied to fsdNetwork variable.

fairside-core (FairSide) acknowledged:

> This function is invoked directly in the deployment script and cannot be raced. As such, I think this should be set as non-critical (0).

cemozerr (Judge) commented:

> Labeling this as low risk, as the issue could pose a problem in this case, the deployment script has a bug.

Non-Critical Findings

[N-01] Use of `ecrecover` is susceptible to signature malleability

The `ecrecover` function is used in `castVoteBySig()` to recover the voter's address from the signature. The built-in EVM precompile `ecrecover` is susceptible to signature malleability, which could lead to replay attacks (references: https://swcregistry.io/docs/SWC-117, https://swcregistry.io/docs/SWC-121, and https://medium.com/cryptronics/signature-replay-vulnerabilities-in-smart-contracts-3b6f7596df57).

While this is not immediately exploitable in the DAO use case because the voter address is checked against `receipt.voted` to prevent re-voting, this may become a vulnerability if used elsewhere.

Recommend considering using OpenZeppelin's ECDSA library (which prevents this malleability) instead of the built-in function.

fairside-core (FairSide) acknowledged:

> While this is a valid finding, it also exists in the Compound codebase and, as mentioned in the description, is not an active issue. I would label as non-critical (0).

cemozerr (Judge) commented:

> Labeling this issue as non-critical, as the issue with `ecrecover` would only be a problem if not aided with another check to prevent re-voting.

[N-02] `FairSideDAO.SECS_PER_BLOCK` is inaccurate

The `SECS_PER_BLOCK` is currently set to 15s on Ethereum, but it's closer to 13.5s on average. The voting period will be shorter than in reality which might lead to users not getting enough time.

Recommend using a more accurate representation of `SECS_PER_BLOCK` for the deployed chain.

fairside-core (FairSide) acknowledged:

> This parameter is meant to be updated prior to deployment and is susceptible to network fluctuations. As such, this is something that will be tuned prior to deployment and should be considered a non-critical issue as there is no on-chain way to reliably calculate the median block time.

cemozerr (Judge) commented:

> Labeling this issue as non-critical as @fairside-core's comments on the constant value being dependent on network conditions is right.

[N-03] Wrong error message in `__castOffchainVotes`

The error message states:

```
    require(
        proposal.offchain,
        "FairSideDAO::__castOffchainVotes: proposal is meant to be v
    );
```

But it should be "... meant to be voted onchain".

fairside-core (FairSide) confirmed:

> The change requested simply changes the text reported to off-chain processes and does not accompany a change in functionality.

fairside-core (FairSide) resolved:

> Fixed in PR#15.

cemozerr (Judge) commented:

> Labeling this as non-critical as the issue does not pose any risk to functionality.

[N-04] non existing function returns

The functions `castVote` and `castVoteBySig` of FairSideDAO.sol have no "returns" parameters, however they do call "return" at the end of the function. This is

confusing for the readers of the code.

```
function castVote(uint256 proposalId, bool support) public {
    return _castVote(msg.sender, proposalId, support);
}

function castVoteBySig( .. ) public {
    ...
    return _castVote(signatory, proposalId, support);
}
```

Recommend removing the “return” statements from `castVote` and `castVoteBySig`.

fairside-core (FairSide) confirmed:

> Fixed in **PR#22**.

[N-05] `totalCostShareBenefit` vs `totalCostShareBenefits`

The function `purchaseMembership` of FSDNetwork.sol contains a variable that is very similar to a global variable. It’s easy to confuse the two, possibly introducing errors in the future. These variables are `totalCostShareBenefit`, and `totalCostShareBenefits`.

FSDNetwork.sol:

```
uint256 public totalCostShareBenefits;
function purchaseMembership(uint256 costShareBenefit) exterr
    uint256 totalCostShareBenefit = membership[msg.sender].a
    ...
    totalCostShareBenefits = totalCostShareBenefits.add(cost
```

Recommend changing one of the variables to an obviously different name.

fairside-core (FairSide) acknowledged

[N-06] Misleading error messages

There are misleading copy-pasted error messages. For Example, function `liquidateEth` has a misleading revert message:

> "FSD::payClaim: Insufficient Privileges"

Same situation with functions `liquidateDai`, `setConvictionless`, `_addGovernanceTribute`. Function `_calculateDeltaOfFSD` has it misspelled. contract `Timelock` constructor uses ' `setDelay` '.

Recommending that it should be `payClaim -> liquidateEth`, etc., to identify the real name of the function where the error happened.

fairside-core (FairSide) confirmed and resolved (in separate issue #64):

> Fixed in PR#9.

[N-07] Revert messages are wrong

The following revert messages refer to a different function instead of the one where they actually are, making it harder to understand the flow of the program in case of error.

- I. 166
- I. 185
- I. 254

Recommend setting the messages with the correct function name.

fairside-core (FairSide) confirmed and resolved:

> Fixed in PR#9.

[N-08] Constant values used inline

In several locations, constant values are used inline in the code. Normally, you would define those as constants to be able to review and update them easier.

Recommend using constants for constant values.

fairside-core (FairSide) acknowledged:

> Similar to #65

[N-09] Events in `FairSideDAO` are not indexed

Events in the `FairSideDAO` contract are not indexed, making it difficult for off-chain scripts (such as the front-ends of dApps) to filter these events efficiently.

Recommend adding the `indexed` keyword to the events. For Example: `event ProposalExecuted(uint256 indexed id);`

fairside-core (FairSide) acknowledged:

> Findings that do not alter the functionality of the contracts should not be labeled as anything else than 0 (Non-Critical). This purely relates to off-chain integration.

cemozerr (Judge) commented:

> Labeling this as non-critical as @fairside-core's comment is correct.

[N-10] lack of input validation of id in `getConvictionScore()`

`tokenId` shouldn't be zero because it is initialized to 1. But due to lack of input validation in `getConvictionScore(uint256 id)`, `tokenId` can be zero.

Recommend adding a condition to check input values.

fairside-core (FairSide) acknowledged:

> Usage of an ID equal to `0` will yield `0` for its conviction score, and it cannot lead to any misbehavior of the contracts to my knowledge.

cemozerr (Judge) commented:

> Labeling this as non-critical as `getConvictionScore` returning 0 seems to have no impact on the protocol.

[N-11] `validateVoteHash` does not confirm the vote result

The `validateVoteHash` function only checks if the individual voting power (conviction score) is indeed correct, but it does not verify if the outcome of the vote is correct, i.e., it is possible for a guardian to submit completely different `forVotes` / `againstVotes` in `__castOffchainVotes` changing the proposal outcome.

The guardian needs to be trusted to submit the correct `forVotes` and `againstVotes` to match the votes in the `voteHash`. The issue is that this cannot be easily verified.

Legitimate users can be tricked into thinking the result is correct by checking if their vote & support is contained in `votes` and recomputing the `voteHash` themselves. They then call `validateVoteHash`, which "confirms" the guardian result. However, in reality, the guardian could have submitted arbitrary `forVotes` / `againstVotes` values.

This makes the current validation system kind of useless.

Recommend summing up the for/against votes in the `votes` array of `validateVoteHash` and check if it matches the `proposal.forVotes/againstVotes`.

fairside-core (FairSide) acknowledged:

> The `VotePack` struct contains a `bool` indicating whether there was support for a proposal or not. The `validateVoteHash` function hashes all submitted votes meaning that it is impossible to obscure the for and against votes as they can be calculated off-chain.

> In any case, this is purely an off-chain utility function, and as such, the severity should be reduced to non-critical (0).

cemozerr (Judge) commented:

> Labeling this as non-critical as `validateVoteHash` is an external function.

Gas Optimizations

[G-01] Gas optimizations - checkpoints from `ERC20ConvictionScore`

In `ERC20ConvictionScore.sol`, we store

```
// Conviction score based on # of days multiplied by # of FS
// @notice A record of conviction score checkpoints for each
mapping(address => mapping(uint32 => Checkpoint)) public che

// @notice The number of checkpoints for each account
mapping(address => uint32) public numCheckpoints;
```

These two state variables are used in the following way: (see Issue #54 for referenced code)

Checking the contract seems like using `mapping(address => Checkpoint[]) public checkpoints;` would provide the same functionality while using less storage.

fairside-core (FairSide) questioned:

> I am unsure what this relates to. Can we have some further information from a_delamo?

[G-02] Reduce reads in `purchaseMembership` method

The method `purchaseMembership` in the `FSDNetwork` contract contains the code below. Inside this method, we are constantly reading from the mapping `membership`, so why not use just one read `Membership userMembership =`

`membership[msg.sender]` and use this instance for everything related to memberships as each read we are currently doing has an impact on the gas cost.

See Issue #55 for referenced code.

fairside-core (FairSide) confirmed and resolved:

> Fixed in PR#11.

[G-03] Use external instead of public methods

The following methods are public and could be external. External is more optimized for gas than public and, as such, should be used as much as possible. See issue #57 for examples and more info at https://ethereum.stackexchange.com/a/19391

fairside-core (FairSide) acknowledged:

> As the Stack Overflow post indicates, the optimization is only really applicable when arrays are involved. We will retain the functions as is and adjust them as necessary further down in the development cycle.

[G-04] Improvements arctan

The performance (gas usage) of the current arctan implementation is: `arctan(ONE)` ~ 5126 Gas (with solidity 0.6.8)

The main cause of the gas usage is the library `ABDKMathQuad` which implements IEEE 754 quadruple-precision binary floating-point numbers. However, the arctan approximation has relatively low precision.

The PDF "higher*order*approximations" in this article shows different formulas for the approximation for arctan, which have higher precision than the current implementation.

The third-order approximation is:

```
arctan(x)  ~ π/2 * sgn(x)*φ(abs(x))
φ(x) = { a*x + x^2 + x^3 } / { 1 + (a+1)x + (a+1)x^2 + x^3 }
```

```
a=0.6399276529
```

I've made an implementation (see below), which takes a lot less gas: arctan_uint(1 * precision) ~ 574 Gas (with solidity 0.6.8)

The implementation takes a different approach to floating points: it multiples all numbers by precision. The precision factor can be adjusted as long as all temporary variables stay below 2^256 (the max value of a uint)

```
pragma solidity 0.6.8;
contract Test{
   uint constant precision=10**30;
   uint constant pi=3.1415926535E30;
   uint constant pidiv2=pi/2;
   uint constant a1=0.6399276529E30;
   uint constant aplus1=1.6399276529E30;

   function arctan_uint(uint x) public pure returns (uint) {
       uint xsquare    = x*x/precision;
       uint xtriple      = xsquare * x/precision;
       uint aplus1x    = aplus1 * x/precision;
       uint top          = a1 * x/precision + xsquare + xtriple;
       uint bottom  = precision + aplus1x + aplus1x*x/precision
       return          pidiv2*top/bottom;
   }

   function test_arctan_uint() public pure returns (uint){
       return arctan_uint(precision);
   }
}
```

Recommend defining which resolution is required and take the necessary formula from the higher*order*approximations.pdf document. Change the math library to a simple "precision" based implementation (as shown above). This will also require adapting other code. Also, set the "precision" constant to the required precision and adjust the constants to the required number of decimals.

fairside-core (FairSide) acknowledged:

> Although the optimization is acknowledged, it will not be applied given that we already use ABDK math across the full codebase.

[G-05] Repetitive storage access

The function `_addTribute` can reuse `lastTribute` to reduce the numbers of storage access: tributes `[totalTributes - 1].amount = add224(...)` can be replaced with `lastTribute.amount = add224(...)` as it is already a storage pointer that can be assigned a value with no need to recalculate the index and access the array again. Same situation with function `_addGovernanceTribute` and `governanceTributes`.

Recommend making `lastTribute.amount = add224(...)`

fairside-core (FairSide) confirmed and resolved:

> Fixed in **PR#23**.

[G-06] Gas optimization for the `rootPows` function in `FairSideFormula`

Gas optimization is possible for the current `rootPows` implementation. The original implementation of `rootPows` requires 4 `mul` and 2 `sqrt`:

```
function rootPows(bytes16 x) private pure returns (bytes16, byte
    // fourth root
    x = x.sqrt().sqrt();
    // to the power of 3
    x = _pow3(x);
    // we offset the root on the second arg
    return (x, x.mul(x));
}
```

However, the calculation process can be simplified to be more gas-efficient than the original with only 1 `mul` and 2 `sqrt` requried:

```
function rootPows(bytes16 x) private pure returns (bytes16, byte
```

```
        bytes16 x1_2 = x.sqrt();
        bytes16 x3_2 = x.mul(x1_2);
        bytes16 x3_4 = x3_2.sqrt();
        return (x3_4, x3_2);
    }
```

Recommend changing the implementation of `rootPows` as mentioned above.

fairside-core (FairSide) questioned:

> Optimization is confirmed (basically constructs x^3/2 then applies root on it). Given that this is a gas optimization, perhaps the severity should be noted down to 1? I'll leave this up to the judges.

fairside-core (FairSide) confirmed and resolved:

> Fixed in **PR#6**.

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