FairSide contest
Findings & Analysis Report

2021-12-16

Table of contents

- Overview
  - About C4
  - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (4)
  - [H-01] Anyone Can Arbitrarily Call `FSDVesting.updateVestedTokens()`
  - [H-02] `FSDVesting` : Claiming tributes should call FSD token's corresponding functions
  - [H-03] Beneficiary cant get `fairSideConviction` NFT unless they only claim once, and only after it's fully vested
  - [H-04] ERC20ConvictionScore._writeCheckpoint does not write to storage on same block
- Medium Risk Findings (2)
  - [M-01] `TributeAccrual.availableTribute()` & `TributeAccrual.availableGovernanceTribute()` Distributes Tributes

Unfairly
    - [M-02] `user.creation` is updated incorrectly when the user tries to extend membership
- Low Risk Findings (6)
- Non-Critical Findings (13)
- Gas Optimizations (21)
- Disclosures

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 FairSide smart contract system written in Solidity. The code contest took place between November 9—November 11 2021.

Wardens

18 Wardens contributed reports to the FairSide contest:

1. cmichel
2. leastwood
3. WatchPug (jtp and ming)
4. hickuphh3
5. hyh
6. rfa
7. gzeon
8. Ruhum

9. 0x0x0x
10. yeOlde
11. pants
12. JMukesh
13. TomFrench
14. Reigada
15. nathaniel
16. elprofesor
17. pmerkleplant

This contest was judged by pauliax.

Final report assembled by moneylegobatman and CloudEllie.

Summary

The C4 analysis yielded an aggregated total of 12 unique vulnerabilities and 46 total findings. 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, 2 received a risk rating in the category of MEDIUM severity, and 6 received a risk rating in the category of LOW severity.

C4 analysis also identified 13 non-critical recommendations and 21 gas optimizations.

Scope

The code under review can be found within the C4 FairSide contest repository, and is composed of 43 smart contracts written in the Solidity programming language and includes 5,254 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 (4)

[H-01] Anyone Can Arbitrarily Call `FSDVesting.updateVestedTokens()`

Submitted by leastwood, also found by cmichel, hickuphh3, hyh, rfa, and WatchPug

Impact

The `updateVestedTokens()` function is intended to be called by the `FSD.sol` contract when updating a user's vested token amount. A check is performed to ensure that `_user == beneficiary`, however, as `_user` is a user controlled argument, it is possible to spoof calls to `updateVestedTokens()` such that anyone can arbitrarily add any amount to the vested contract. Additionally, there is no check to ensure that the call originated from a trusted/whitelisted source.

There are two main reasons as to why the beneficiary or an attacker would want to call this function:

- To increase the vested amount such that `calculateVestingClaim()` allows them to withdraw their entire vested amount without waiting the entire duration.
- An attacker wishes to block withdrawals from other vested contracts by preventing successful calls to `claimVestedTokens()` by the beneficiary

account. This can be done by increasing the vested amount such that `safeTransfer()` calls fail due to insufficient token balance within the contract.

Proof of Concept

- https://github.com/code-423n4/2021-11-fairside/blob/main/contracts/token/FSDVesting.sol#L147-L161
- https://github.com/code-423n4/2021-11-fairside/blob/main/contracts/token/FSDVesting.sol#L100-L115
- https://github.com/code-423n4/2021-11-fairside/blob/main/contracts/token/FSDVesting.sol#L125
- https://github.com/code-423n4/2021-11-fairside/blob/main/contracts/token/FSDVesting.sol#L134

Tools Used

Manual code review. Discussions with dev.

Recommended Mitigation Steps

Ensure that the `updateVestedTokens()` function is only callable from the `FSD.sol` contract. This can be done by implementing an `onlyFSD` role.

YunChe404 (FairSide) confirmed

[H-02] `FSDVesting` : Claiming tributes should call FSD token's corresponding functions

Submitted by hickuphh3, also found by leastwood

Impact

The claiming of staking and governance tributes for the a beneficiary's vested tokens should be no different than other users / EOAs. However, the `claimTribute()` and `claimGovernanceTribute()` are missing the actual claiming calls to the corresponding functions of the FSD token contract. As a result, the accrued rewards are taken from the beneficiary's vested token while not claiming (replenishing) from the FSD token contract.

Recommended Mitigation Steps

In addition to what has been mentioned above, the internal accounting for claimedTribute states can be removed because they are already performed in the FSD token contract.

```
// TODO: Remove _claimedTribute and _claimedGovernanceTribute ma

/**
* @dev Allows claiming of staking tribute by `msg.sender` during
* It updates the claimed status of the vest against the tribute
* being claimed.
*
* Requirements:
* - claiming amount must not be 0.
*/
function claimTribute(uint256 num) external onlyBeneficiary {
    uint256 tribute = fsd.availableTribute(num);
    require(tribute != 0, "FSDVesting::claimTribute: No tribute
                fsd.claimTribute(num);
    fsd.safeTransfer(msg.sender, tribute);
    emit TributeClaimed(msg.sender, tribute);
}

/**
* @dev Allows claiming of governance tribute by `msg.sender` dur
* It updates the claimed status of the vest against the tribute
* being claimed.
*
* Requirements:
* - claiming amount must not be 0.
*/
function claimGovernanceTribute(uint256 num) external onlyBenefi
  uint256 tribute = fsd.availableGovernanceTribute(num);
  require(
    tribute != 0,
    "FSDVesting::claimGovernanceTribute: No governance tribute t
  );
  fsd.claimGovernanceTribute(num);
  fsd.safeTransfer(msg.sender, tribute);
  emit GovernanceTributeClaimed(msg.sender, tribute);
}
```

YunChe404 (FairSide) confirmed

[H-03] Beneficiary cant get `fairSideConviction` NFT unless they only claim once, and only after it's fully vested

Submitted by WatchPug, also found by cmichel

Based on the context, once the beneficiary claimed all their vesting tokens, they should get the `fairSideConviction` NFT.

However, in the current implementation, if the beneficiary has claimed any amounts before it's fully vested, then they will never be able to get the `fairSideConviction` NFT, because at L138, it requires the `tokenbClaim` to be equal to the initial vesting amount.

`FSDVesting.sol` **L124-L142**

```
    function claimVestedTokens() external override onlyBeneficiary {
        uint256 tokenClaim = calculateVestingClaim();
        require(
            tokenClaim > 0,
            "FSDVesting::claimVestedTokens: Zero claimable tokens"
        );

        totalClaimed = totalClaimed.add(tokenClaim);
        lastClaimAt = block.timestamp;

        fsd.safeTransfer(msg.sender, tokenClaim);

        emit TokensClaimed(msg.sender, tokenClaim, block.timestamp);

        if (amount == tokenClaim) {
            uint256 tokenId = fsd.tokenizeConviction(0);
            fairSideConviction.transferFrom(address(this), msg.sende
        }
    }
```

Recommendation

Change to:

```
function claimVestedTokens() external override onlyBeneficiary {
    uint256 tokenClaim = calculateVestingClaim();
    require(
        tokenClaim > 0,
        "FSDVesting::claimVestedTokens: Zero claimable tokens"
    );

    totalClaimed = totalClaimed.add(tokenClaim);
    lastClaimAt = block.timestamp;

    fsd.safeTransfer(msg.sender, tokenClaim);

    emit TokensClaimed(msg.sender, tokenClaim, block.timestamp);

    if (amount == totalClaimed) {
        uint256 tokenId = fsd.tokenizeConviction(0);
        fairSideConviction.transferFrom(address(this), msg.sende
    }
}
```

YunChe404 (FairSide) confirmed and disagreed with severity

[H-04] ERC20ConvictionScore._writeCheckpoint does not write to storage on same block

Submitted by cmichel

In `ERC20ConvictionScore._writeCheckpoint`, when the checkpoint is overwritten (`checkpoint.fromBlock == blockNumber`), the new value is set to the `memory checkpoint` structure and never written to storage.

```
// @audit this is MEMORY, setting new convictionScore doesn't wr
Checkpoint memory checkpoint = checkpoints[user][nCheckpoints -

if (nCheckpoints > 0 && checkpoint.fromBlock == blockNumber) {
    checkpoint.convictionScore = newCS;
}
```

Users that have their conviction score updated several times in the same block will only have their first score persisted.

POC

- User updates their conviction with `updateConvictionScore(user)`
- **In the same block**, the user now redeems an NFT conviction using `acquireConviction(id)`. This calls `_increaseConvictionScore(user, amount)` which calls `_writeCheckpoint(..., prevConvictionScore + amount)`. The updated checkpoint is **not** written to storage, and the user lost their conviction NFT. (The conviction/governance totals might still be updated though, leading to a discrepancy.)

Impact

Users that have their conviction score updated several times in the same block will only have their first score persisted.

This also applies to the total conviction scores `TOTAL_CONVICTION_SCORE` and `TOTAL_GOVERNANCE_SCORE` (see `_updateConvictionTotals`) which is a big issue as these are updated a lot of times each block.

It can also be used for inflating a user's conviction by first calling `updateConvictionScore` and then creating conviction tokens with `tokenizeConviction`. The `_resetConviction` will not actually reset the user's conviction.

Recommended Mitigation Steps

Define the `checkpoint` variable as a `storage` pointer:

```
Checkpoint storage checkpoint = checkpoints[user][nCheckpoints -
```

YunChe404 (FairSide) confirmed

Medium Risk Findings (2)

[M-01] `TributeAccrual.availableTribute()` &

`TributeAccrual.availableGovernanceTribute()` Distributes Tributes Unfairly

Submitted by leastwood

Impact

Conviction scores are calculated by taking the user's balance and multiplying it by the time elapsed. This score is updated upon each token transfer, or alternatively by directly calling `ERC20ConvictionScore.updateConvictionScore()`. The `availableTribute()` and `availableGovernanceTribute()` functions calculate a user's share by calculating their percentage of the total conviction score to find the amount owed to them.

This is shown in the following statement where `userCS` is the user's conviction score and `totalCS` is the protocol's total conviction score:

```
return amount.mul(userCS).div(totalCS);
```

The tribute amount that can be successfully claimed disproportionally favours users who have updated their conviction score more recently and who are early to claim their allocated tribute.

Proof of Concept

Consider the following POC:

- Two users Alice and Bob both have a conviction score of 5 at time = 1.
- Time advances such that time = 2.
- Bob calls `updateConvictionScore()` which sets his conviction score to `5 * time = 10` where time = 2.
- Therefore, the total conviction score is now 15 and Alice and Bob's individual conviction scores are 5 and 10 respectively.
- Bob attempts to call `claimTribute()` where the tribute to claim is of size 15 tokens. As a result, Bob receives 10 tokens according to the calculation mentioned above (`(15*10)/15`).

- Alice updates her conviction score and also attempts to call `claimTribute()`, receiving 7.5 tokens in the process.
- As a result, a total of 22.5 tokens were transferred out despite the tribute amount being only 15 tokens in total.

As you can see from the above example, the amount of tokens transferred favours user's who claim early and update their conviction score. It is entirely possible that due to the finite amount of tokens stored in the contract, a number of user's won't be able to successfully claim their allocated tribute.

- `TributeAccrual.sol#L93` **L112**
- `TributeAccrual.sol#L117` **L136**
- `dependencies/TributeAccrual.sol` **L156**
- `dependencies/TributeAccrual.sol` **L179**
- `ERC20ConvictionScore.sol#L476` **L479**

Tools Used

Manual code review.

Recommended Mitigation Steps

There is no easy solution to the mentioned issue. Ideally, all users should have an up to date conviction score whenever `claimTribute()` or `claimGovernanceTribute()` are called, although this would be an incredibly gas intensive mitigation. Alternatively, the protocol's total conviction score can be calculated by tracking the total minted balance of `FSD.sol`'s token and multiplying this by a running total of time elapsed. This will need to be adjusted whenever tokens are minted or burned and updated during tribute claims.

YunChe404 (FairSide) disputed and disagreed with severity

[M-02] `user.creation` is updated incorrectly when the user tries to extend membership

Submitted by WatchPug

`FSDNetwork.sol` **L274-L291**

https://github.com/code-423n4/2021-11-fairside/blob/20c68793f48ee2678508b9d3a1bae917c007b712/contracts/network/FSDNetwork.sol#L274-L291

```
if (user.creation == 0) {
    user.creation = block.timestamp;
    user.gracePeriod =
        membership[msg.sender].creation +
        MEMBERSHIP_DURATION +
        60 days;
} else {
    uint256 elapsedDurationPercentage = ((block.timestamp -
        user.creation) * 1 ether) / MEMBERSHIP_DURATION;
    if (elapsedDurationPercentage < 1 ether) {
        uint256 durationIncrease = (costShareBenefit.mul(1 ether
            (totalCostShareBenefit - costShareBenefit)).mul(
                MEMBERSHIP_DURATION
            ) / 1 ether;
        user.creation += durationIncrease;
        user.gracePeriod += durationIncrease;
    }
}
```

PoC

1. Alice calls `function purchaseMembership()` and adds 20 ether of `costShareBenefit` on day 1:

```
alice.creation = day 1 timestamp;
alice.gracePeriod = day 791 timestamp;
```

2. Alice calls `function purchaseMembership()` again and adds 20 ether of `costShareBenefit` on day 2:

```
elapsedDurationPercentage = 1/720
durationIncrease = 730 day
```

```
alice.creation = day 731 timestamp;
alice.gracePeriod = day 1521 timestamp;
```

Making Alice unable to use any membership features until two years later.

YunChe404 (FairSide) confirmed and disagreed with severity

Low Risk Findings (6)

- [L-01] Offchain voting can't be renabled in DAO *Submitted by Ruhum*
- [L-02] `FSDNetwork.getFSDPrice()` Is Vulnerable To Flash Loan Attacks *Submitted by leastwood*
- [L-03] Possible DOS by 1 of 3 assessors by replay attack *Submitted by gzeon*
- [L-04] Open TODOs *Submitted by ye0lde, also found by gzeon, JMukesh, leastwood, and pants*
- [L-05] `FSD.mintBeta` function has potentially blocking condition unrelated with documented logic *Submitted by hyh, also found by 0x0x0x*
- [L-06] Missing SafeMath & SafeCasts *Submitted by cmichel, also found by rfa, Ruhum, and ye0lde*

Non-Critical Findings (13)

- [N-01] Missing events for critical operations *Submitted by WatchPug, also found by pmerkleplant and ye0lde*
- [N-02] `FSDNetwork` should inherit from interface `IFSDNetwork` *Submitted by pmerkleplant*
- [N-03] Faulty comments in `dao/FairSideDAO.sol` *Submitted by pmerkleplant*
- [N-04] Incorrect comment in function rmul (DSMath.sol) *Submitted by ye0lde*
- [N-05] Various typos *Submitted by ye0lde*
- [N-06] does not check the existence of address while using it in low level call *Submitted by JMukesh*
- [N-07] safeApprove is deprecated. *Submitted by pants*
- [N-08] `FairSideDAO.SECS_PER_BLOCK` is inaccurate *Submitted by cmichel*

- [N-09] FSD.sol does not implement transfer-accept ownership pattern *Submitted by elprofesor*
- [N-10] Underflow in `ERC20ConvictionScore._writeCheckpoint` *Submitted by cmichel*
- [N-11] Improper Validation Of Chainlink `latestRoundData()` Function *Submitted by leastwood*
- [N-12] Use of transfer function for transferring NFTs *Submitted by Reigada, also found by Ruhum*
- [N-13] Missing parameter validation *Submitted by cmichel, also found by pants, pmerkleplant, Reigada, and yeOlde*

Gas Optimizations (21)

- [G-01] Assigning keccak operations to constant variables results in extra gas costs *Submitted by TomFrench*
- [G-02] Long revert strings *Submitted by yeOlde, also found by elprofesor, pants, and WatchPug*
- [G-03] Using ++i consumes less gas than i++ *Submitted by Reigada, also found by pants*
- [G-04] redundant named return issue *Submitted by pants, also found by yeOlde*
- [G-05] Calling require on a tautology *Submitted by 0x0x0x*
- [G-06] `!= 0` costs less gass compared to `> 0` for unsigned integer *Submitted by 0x0x0x*
- [G-07] Double update on storage pointer can lead to massive gas consumption *Submitted by rfa, also found by cmichel*
- [G-08] Gas: Reorder conditions in `claimGovernanceTribute` *Submitted by cmichel*
- [G-09] Use existing memory version of state variables (Timelock.sol) *Submitted by yeOlde*
- [G-10] `_calculateDeltaOfFSD(ABC.sol)` can be optimized *Submitted by 0x0x0x*

- [G-11] The function `propose(FairSideDAO.sol)` can be optimized *Submitted by 0x0x0x*
- [G-12] FSDVesting: Define new constant LINEAR*VEST*AFTER_CLIFF *Submitted by hickuphh3*
- [G-13] FSDVesting: Redundant _start input parameter in initiateVesting() *Submitted by hickuphh3*
- [G-14] FSDVesting: Optimise updateVestedTokens() *Submitted by hickuphh3, also found by nathaniel*
- [G-15] Several public functions can be declared as external *Submitted by Reigada, also found by hyh and pants*
- [G-16] Remove redundant check can save gas *Submitted by WatchPug, also found by gzeon*
- [G-17] Using fixed length array as parameter type can avoid checks to save gas and improve consistency *Submitted by WatchPug*
- [G-18] Avoid unnecessary external calls can save gas *Submitted by WatchPug*
- [G-19] Use `else if` in for loops can save gas and simplify code *Submitted by WatchPug*
- [G-20] Avoid unnecessary storage reads in for loops can save gas *Submitted by WatchPug*
- [G-21] FSDVesting: Constants can be made internal / private *Submitted by hickuphh3*

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