EGIS SECURITY

SECURITY REVIEW FOR FLUX

FINDINGS SUMMARY

1 CRITICAL

3 MEDIUM

10 LOW

4 INFO

DATES

06.09.2024 - 13.09.2024

Table of Contents

1 About Egis Security

Egis Security is a team of experienced smart contract researchers, who strive to provide the best smart contract security services possible to DeFi protocols.

Both members of Egis Security have a proven track record on public auditing platforms such as Code4rena, Sherlock & Codehawks, uncovering more than 150 High/Medium severity vulnerabilities, with >1$70,000 in winnings and multiple solo/team audits.

2 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

3 Risk classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

3.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

3.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

Overview

Project Name	Flux
Repository	https://github.com/Kuker-Labs/flux-contracts
Commit hash	e4c6fb70fec5bea16066a975e8b54b8f7426b116
Resolution	33b76781c960fd558ffb8ee906da6a6b9d8b6ef6
Documentation	https://flare-4.gitbook.io/flux-protocol
Methods	Manual review

Scope

/src/**

Issues Found

Critical risk	1
High risk	0
Medium risk	3
Low risk	10
Informational	4

5 Findings

5.1 Critical risk

5.1.1 User can manipulate `Staking#rewardPerShare` value and steal others rewards

Severity: *Critical risk*

Context: Staking.sol#L225-L232

Description: `Staking` contract is supposed to receive `titanX` tokens from `distribute` function. Those tokens should be proportionally distributed across different reward pools. The problem here is that we use `titanX.balanceOf(address(this))` and we increment the current `toDistribute[]` value. This leads to double spending issue because each time `distribute` is called, we count all previous tokens as if they were new, which is not the case. An exploiter may benefit from this by maliciously increasing rewards per share, so he can claim all available tokens in the contract when they are distributed in the corresponding pool.

Imagine the following:

1. We have 10 000 titanX deposited into the staking contract
2. We have user A who has staked 5_000 flux and user B who has staked 5_000 flux at 1:1 shares ratio
3. When `distribute` is called, `toDistribute[POOLS.DAY8] += 10_000 * 0.38 = 3_800`
4. User A call `Staking#distribute` with `_amount = 1` multiple times in a row and here is the result:

- `toDistribute[POOLS.DAY8] += 3_800` = 7_600
- `toDistribute[POOLS.DAY8] += 3_800` = 11_400
- `toDistribute[POOLS.DAY8] += 3_800` = 15_200
- `toDistribute[POOLS.DAY8] += 3_800` = 19_000

5. When the 8th day pass, we update the rewards and `rewardPerShare += 19 000 (toDist[pool])/ 10 000 (totalShares)= 1.9`
6. Now user A calls `claim` with his id, which holds 5K shares: `uint256 amountToClaim = wmul(_rec.shares, rewardPerShare - _rec.rewardDebt)` = `5_000 * 1.9 = 9 500` and he claims 9_500 titanX tokens and Staking is left with only 500 titanX
7. User B is unable to claim because the contract will try to transfer him 9_500 tokens when it has only 500

Recommendation: Instead of using `titanX.balanceOf(address(this))` to determine `forDistribution`, use the amount provided to `distribute` function call.

Resolution: Fixed

5.2 Medium risk

5.2.1 Possibility of DoS in `Staking`, if no one has voluntary staked, or has unstaked

Severity: *Medium risk*

Context: 777Voluntary.sol#L76

Description: `777Voluntary` pool accrues rewards for users, who have staked for maximum duration. Those users receive shares and their reward is calculated using `rewardPerShare * userShares` `rewardPerShare` is calculated by dividing the number of tokens accrued for 777 days by the total shares of users, who are exposed for the reward. The problem is that if there are no such stakes, `totalShares` would be equal to 0, resulting in a revert. We will enter the flow after 777 days:

```
        if (distributeDay777) {
            uint256 forVoluntary = toDistribute[POOLS.VOLUNTARY];

            if (forVoluntary > 0) {
                titanX.transfer(address(voluntary), forVoluntary);
                voluntary.distribute(forVoluntary);
                toDistribute[POOLS.VOLUNTARY] = 0;
            }
        }
```

Because of that the probability for the issue to occur is low, but the impact is critical, as the whole staking functionality is DoSed, because `updateRewardsIfNecessary` is called on stake, unstake and claim. All stakers lose their stakes + rewards. Furthermore, the auction contract is also bricked, because it will try to call `Staking.stake` on each `deposit` call. Additionally, protocol staking tokenomics incentives stakers to stake for the shortest possible period, unstake and stake again, receiving more shares for the same flux amount.

Recommendation:

```
function distribute(uint256 _amount) external onlyStaking {
+ if (totalShares == 0) return;
rewardPerShare += uint112(wdiv(_amount, totalShares));
}
```

Resolution: Fixed

5.2.2 If two or more snapshots are missed in `FluxBuyAndBurn`, distribution accounting will get messed up

Severity: *Medium risk*

Context: FluxBuyAndBurn.sol#L239-L243

Description: `swapTitanXForFluxAndBurn` is responsible for swapping and burning equal amounts titanX tokens. The amount is calculated from `totalTitanXDistributed` variable, which is set for each snapshot (day) to all tokens received for past 24 hours. We have identified a possible issue regarding this invariant if `FluxBuyAndBurn` funcs have not been called for more than 24 hours, which lead to burning all contract balance at once.

Imagine the following scenario:

1. We have `totalTitanXDistributed = 20_000` for snapshot 1 and we are burning.
2. During that period `toDistribute` has become `10 000` (distribute value, which will be burned for the next snapshot)
3. Last call to `distributeTitanXForBurning` has been made on Monday 4:59 PM.
4. Nobody calls the contract for the next 36 hours and on Wednesday at 5:01 AM `distributeTitanXForBurning` is called with value of `10 000` (which should be scheduled for next snapshot burning amounts).
5. Now we have `20_000` titanX in contract (10K, which should have burned last snapshot and the new 10K, which should be scheduled for the next snapshot. **NOTE** For the current snapshot we don't have tokens to burn, because we haven't distributed for past 24 hours)
6. We enter `_intervalUpdate` -> `getCurrentInterval` -> `_calculateIntervals`:

- `timeElapseSinceLastBurn` is 24 hours and we enter **`else`** :

```
            //@note - Calculate the upcoming intervals with the to distribute shares
            uint128 _intervalsForNewDay =
                missedIntervals > accumulatedIntervalsForTheDay ? missedIntervals -
                    accumulatedIntervalsForTheDay : 0;

            _totalAmountForInterval += uint128(toDistribute / INTERVALS_PER_DAY) *
                _intervalsForNewDay;
```

- Here `_intervalsForNewDay` would be intervals for 36 hours, which is > `INTERVALS_PER_DAY`. As a result we will have `(toDistribute(10K)/ 180 )* 270 = 15_000`
- In other words we calculate that we should burn 150% the distribution amount for the missed day.

7. We update `_totalAmountForInterval` to the amount of 15_000 (because we have already transferred the new tokens and current balance is 20_000)
8. We instantly can burn 15_000 tokens (5_000 from those should be scheduled for the next snapshot)
9. We set `toDistribute` to `10 000`, but we leave contract with only `5_000` titanX tokens
10. When next snapshot comes and we update `_totalAmountForInterval = toDistribute`, we will have only half `totalAmountForInterval` available in the contract, which means that if new tokens enter during this snapshot, contract will start using them, messing up the accounting

Recommendation: Implement the following changes in `_updateSnapshot` to ensure that we don't accrue for more intervals than `INTERVALS_PER_DAY`:

```
    if (Time.blockTs() < startTimeStamp) return;

+    if(lastSnapshot + 48 hours < Time.blockTs()){ // If we have missed entire
    snapshot if interacting with the contract
+         toDistribute = 0;
+       }
```

and in `_calculateIntervals`:

```
-     _totalAmountForInterval += uint128(toDistribute / INTERVALS_PER_DAY) *
    _intervalsForNewDay;
+      _totalAmountForInterval +=  (_intervalsForNewDay > INTERVALS_PER_DAY) ?
    uint128(toDistribute) : uint128(toDistribute / INTERVALS_PER_DAY) *
    _intervalsForNewDay;
```

Resolution: Fixed

5.2.3 First day distributed rewards should all go to DAY8 Pool

Severity: *Medium risk*

Context: Staking.sol#L224-L234

Description: Regarding the docs excess titanX from the first day should be distributed only in `[POOLS.DAY8]`, which is not the current behavior. Currently:

- we don't have a guarantee that `Staking.distribute` will be called from the auction contract on that day
- we don't have mechanism to allocate all received rewards on the first day to 8 day pool.

Recommendation: Implement a mechanism to check if the current day is the first day and allocate all amounts to `DAY8 Pool`. Also, call `_distribute` internally in `FluxAuction.deposit`

Resolution: Fixed.

5.3 Low risk

5.3.1 Possibility of DoS in Staking contract, if there are no stakes untill the 8th day

Severity: *Low risk*

Context: Staking.sol#L238-L239

Description: If `_updateRewards` is called and `totalShares = 0`, we will have division by 0, which results in a revert:

```
        rewardPerShare += uint72(wdiv(toDist[pool], totalShares));
```

Due to the very low likelihood of having 0 shares after 8 days, we define the severity to be Low: Likelihood = Low Impact = Medium (Because all staking logic is bricked, because we cannot update the rewards. Also auction `deposit` is bricked, because flow tries to call `Staking#stake`, but no funds are locked)

Recommendation:

```
+      if (totalShares == 0) {
    // Decide whether you will flip rewards for the next cycle, or erase them.
    }
         rewardPerShare += uint72(wdiv(toDist[pool], totalShares));
```

Resolution: Fixed

5.3.2 If `FluxAuction#amountToClaim` is used by integrating parties, it may return wrong data for the current day

Severity: *Low risk*

Context: FluxAuction.sol#L186-L194

Description: Because `stats.titanXDeposited` will most probably change before the end of the day and the value returned from `amountToClaim` cannot be trusted if the id is for the current day.

Recommendation: Consider reverting if the day is still pending, or documenting the behavior (It cannot be trusted for the current day)

Resolution: Acknowledged

5.3.3 If there is no liquidity in the inferno/flux pool after 24 hours, deposit will revert

Severity: *Low risk*

Context: FluxAuction.sol#L131-L136

Description: If after 24 hours from the `startTimestamp`, FluxAuction have not collected `INITIAL_TITAN_X_FOR_LIQ`, or `addLiquidityToInfernoFluxPool` function has not been called, calls to deposit will revert, when we try to swap inferno for flux for auto buy and stake:

```
        uint160 fluxAmount = _swapInfernoForFlux(infernoReceived, _deadline);
```

This may lead to a contract titanX funds block, which means `INITIAL_TITAN_X_FOR_LIQ` may not be reached, and up may not be created. To fix this, someone should donate titanX to the protocol, effectively losing it.

Recommendation: Consider making a `isSecondDay` check to check if lp has been created.

Resolution: Acknowledged

5.3.4 If `startTimeStamp` is in the future, `FluxBuyAndBurn#getCurrentInterval` reverts with underflow

Severity: *Low risk*

Context: FluxBuyAndBurn.sol#L311-L312

Description: If someone calls `getCurrentInterval` before reaching `startTimeStamp`, function will revert with underflow.

Recommendation: Consider handling the case gracefully by either:

- returning zeros
- reverting with detailed error

Resolution: Fixed

5.3.5 `FluxBuyAndBurn#lastBurnedIntervalStartTimestamp` can be set to future timestamp

Severity: *Low risk*

Context: FluxBuyAndBurn.sol#L316

Description: If start timestamp is 5 pm and after 9 minutes we call `updateInterval`: We will calculate `_missedIntervals = uint16(timeElapsedSince / INTERVAL_TIME);` = 1 After that we will increment it with 1:

```
            _missedIntervals += timeElapseSinceLastBurn > INTERVAL_TIME ? 1 : 0;
```

And we will set `lastBurnedIntervalStartTimestamp` to:

```
   lastBurnedIntervalStartTimestamp = _lastIntervalStartTimestamp + (uint32(
       _missedIntervals) * INTERVAL_TIME);
```

Which will result in 5 pm + 2 * 8 = 5:16 pm as `lastBurnedIntervalStartTimestamp`, when we are only 5:09 The following will block `distributeTitanXForBurning` until block.timestamp passes `lastBurnedIntervalStartTimestamp`, because of underflow here:

```
Time.blockTs() - lastBurnedIntervalStartTimestamp > INTERVAL_TIME
```

Recommendation: Inside `getCurrentInterval` increment `_missedIntervals` by:

```
 _missedIntervals += timeElapseSinceLastBurn > INTERVAL_TIME &&
     lastBurnedIntervalStartTimestamp != 0 ? 1 : 0;
```

Resolution: Fixed

5.3.6 If no there are no stakers in `777Voluntary`, tokens distributed will be locked

Severity: *Low risk*

Context: Staking.sol#L192-L193

Description: If after 777 days, there are no stakers in 777Voluntary, because protocol incentiveses short-term staking (by minting more shares for late participants), all titanX tokens that have accrued for that period will be locked in `777Voluntary` pool

Recommendation: You can check if `shares == 0` inside `Staking#updateRewardsIfNecessary` and if so, skip whole pool777 logic:

```
-  if (distributeDay777) {
+  if (distributeDay777 && voluntary.totalShares() > 0)
            uint256 forVoluntary = toDistribute[POOLS.VOLUNTARY];

            if (forVoluntary > 0) {
                titanX.transfer(address(voluntary), forVoluntary);
                voluntary.distribute(forVoluntary);
                toDistribute[POOLS.VOLUNTARY] = 0;
            }
        }
```

Resolution: Fixed

5.3.7 Passing duplicate ids will return incorrect `toClaim`

Severity: *Low risk*

Context: FluxAuction.sol#L180-L184

Description: Currently there is nothing stopping someone from calling `batchClaimableAmount` and passing duplicate id's in the `_ids` array, if duplicate ids are used then each "claim" will be counted twice or more. If an external integrator or front-end relies on the value returned from the function he can get tricked.

Currently this function isn't used anywhere else in the protocol, thus keeping it Low.

Recommendation: Disallow duplicate entries in the `_ids` array.

Resolution: Acknowledged

5.3.8 Making `userDep` storage is unnecessary and increases gas costs

Severity: *Low risk*

Context: FluxAuction.sol#L187-L188

Description: There are no state changes to the `userDep`/`depositOf` values, so there is no point in keeping this variable as `storage`, it just increases the gas costs.

Recommendation: Make the variable `memory`

Resolution: Fixed

5.3.9 `record` in `Staking#unstake` shouldn't be storage

Severity: *Low risk*

Context: Staking.sol#L125-L126

Description: The value of `record` isn't changed anywhere where it's used, so there is no need for it to be `storage`.

Recommendation: Make the variable `memory`

Resolution: Fixed

5.3.10 Event is emitted after the value is reset

Severity: *Low risk*

Context: Staking.sol#L240-L243

Description: `Distributed` event will always be emitted with 0 for amount, because we reset the value on the above line.

```
    function _updateRewards(POOLS pool, mapping(POOLS => uint256) storage toDist)
        internal {
        if (toDist[pool] == 0) return;

        rewardPerShare += uint72(wdiv(toDist[pool], totalShares));

        toDistribute[pool] = 0;

        emit Distributed(pool, toDist[pool]);
    }
```

Recommendation: Use memory var, instead of the storage `toDist[pool]`.

Resolution: Fixed

5.4 Informational

5.4.1 Consider updating `currInterval.amountBurned` in `FluxBuyAndBurn#swapTitanXForFluxAndBurn` to the real burned amount

Severity: *Info risk*

Context: FluxBuyAndBurn.sol#L118

Description:
Consider updating `currInterval.amountBurned` to `currInterval.amountAllocated - incentive`, because 1.5% are incentive and are not burned.

Resolution: Acknowledged

5.4.2 Division before multiplication

Severity: *Info risk*

Context: FluxBuyAndBurn.sol#L222-L223

Description:

We will first divide `_totalTitanXDistributed / INTERVALS_PER_DAY`, which will round down, if `_totalTitanXDistributed` is not a multiple of `INTERVALS_PER_DAY` and then we will multiply the result by `missedIntervals`. We can save some precision if we refactor it as `missedIntervals * (_totalTitanXDistributed / INTERVALS_PER_DAY)`

NOTE The same is present in the `else` branch.

Fix it as follows:

```
uint128(toDistribute / INTERVALS_PER_DAY) * _intervalsForNewDay = ‘toDistribute *
    _intervalsForNewDay / INTERVALS_PER_DAY‘
```

Resolution: Acknowledged

5.4.3 Wrong comments or typos

Severity: *Info risk*

Context: FluxBuyAndBurn.sol#L47

Description:

- FluxBuyAndBurn.sol#L47-L48 - Wrong `@notice` comment. Should be Flux tokens burnt
- FluxBuyAndBurn.sol#L182-L183cut-off hour is 5 PM UTC, instead of 2 (in the comment it is stated that it is 2)
- FluxAuction.sol#L245-L246- `incetive` typo

Resolution: Acknowledged

5.4.4 `ownerOfStake != address(auction)` is redundant

Severity: *Info risk*

Context: Staking.sol#L140-L141

Description: At the beginning of the function we disallow `ownerOfStake == address(auction)`, meaning if the `_tokenId` is owned by the Auction then `unstake` can't be called. Knowing that `unstake` can't be called on tokens which the owner is the Auction, it's redundant to check `ownerOfStake != address(auction)` since we already know that if we hit this part of the code, the owner of the token isn't the Auction, because if it was the tx would have reverted on the above check.

Remove the redundant check.

Resolution: Acknowledged