Lotus

Security review

Version 1.0

Reviewed by

nmirchev8
deth

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.

The team has a proven track record on public auditing platforms like Code4rena, Sherlock, and Cantina, earning top placements and rewards exceeding $170,000. They have identified over 150 high and medium-severity vulnerabilities in both public contests and private 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	Lotus
Repository	Private
Commit hash	8ef213544af20f829b66810b0697304ae282691e
Resolution	b34265e348fa210c9356ab10420522a01b58db59
Documentation	Private
Methods	Manual review

Scope

src/BuyAndBurn.sol
src/Lotus.sol
src/Mining.sol
src/Staking.sol
src/utils/*
src/utils/*
src/libs/*
src/const/*
src/actions/*

Issues Found

Critical risk	0
High risk	2
Medium risk	4
Low risk	3
Informational	8

5 Findings

5.1 High risk

5.1.1 Staking on multiple instances from different accounts is beneficial for exploiter

Severity: *High risk*

Context: `LotusBloom.sol`

Description: In `Staking.stake`, there are no minimum shares required to participate—users with as little as 1 share will be included (= 1 wei of Lotus). This creates a negative experience for honest stakers, as their chances will plummet significantly for each account with as little as 1 share.

Attackers can skew odds and grief honest stakers by staking 1 lotus from a substantial number of addresses in order to gain an unfair advantage on the raffle draw.

Additionally, even if there is a minimum imposed, honest users would still be incentivized to stake at smaller batches from diff addresses instead of 1 big stake from one address, since there are no drawbacks.

Recommendation: Make odds for someone to win to be related to his stake shares relative to all shares, or implement a minimum stake size to participate in the raffle.

Resolution: Fixed

5.1.2 `Mining:: _distribute()` - Perentages don't add to 100%

Severity: *High risk*

Context: `Constants.sol#L17`

Description: When `_distribute` is called, the TitanX will be distributed to other parts of the TitanX ecosystem and to some parts of Lotus as well.

If we look at the percentages:

```
uint64 constant TO_DRAGON_X = 0.04e18; // 4%
uint64 constant TO_VOLT_LIQUIDITY_BONDING = 0.04e18; // 4%
uint64 constant TO_LOTUS_LIQUIDTY_BONDING = 0.04e18; // 4%
uint64 constant TO_LOTUS_BUY_AND_BURN = 0.48e18; // 48%
uint64 constant TO_REWARD_POOLS = 0.28e18; // 28%
uint64 constant TO_GENESIS = 0.08e18; // 8%
```

They don't add up to 100%, but to 96%. This results in a loss of funds. Regarding the docs, Lotus Liquidity Bonding should be 8%, instead of 4%

Recommendation: Make `TO_LOTUS_LIQUIDTY_BONDING = 0.08e18`

Resolution: Fixed

5.2 Medium risk

5.2.1 Miner cost compounding daily

Severity: *Medium risk*

Context: `Mining.sol#L251C29-L251C38`

Description: As per the docs, the cost for mining increases by 1.75% daily, however the `minerCost` formula compounds this increase as 1.0175 ^ daysSinceStartTimestamp which in numbers is:

- 562x in the first year
- 316_318x in the second year

In around 4.5 years, the `cost` variable will reach it's max which is type(uint128).max which is a pretty hefty number ~3e38.

Recommendation: Consider if such aggressive behaviour is the intended design and rework the formula if you expect mints after 4.5 years.

Resolution: Acknowledged

5.2.2 If we pick a winner, when all stakers have exited, `LotusBloomPool` will be DoSed

Severity: *Medium risk*

Context: `LotusBloom.sol#L171`

Description: The probability of having such a scenario is pretty low, but the impact would be permanent DoS and stucked funds. If we have no stakers, but someone malicious calls `pickWinner` and then the verifier calls `fulfillRandomWords`, the tx will revert on the following line: `address winner = participants.at(randomness % participants.length());` because we try to divide by 0 (`participants.length()`) The following will leave pending randomness and the contract functionality is forever bricked.

The odds of this happening could be slightly higher due to:

1. LotusBloom is initialized during the constructor execution of `Staking.sol` with an arbitrary `startTimestamp`
2. If the startTimestamp was chosen 2 weeks (or more) in the past, then the 2nd require check in `pickWinner` will be bypassed

```
require(lastIntervalCall + INTERVAL_TIME <= Time.blockTs(), OnlyAfterIntervalTime())
    ;
```

3. The other require check can be instantly bypasssed with a donation through `Staking.distribute` to increment `upForGrabs` to be above 0
4. Given these circumstances, the lotus bloom can be bricked right after deployment

Recommendation: In `fulfillRandomWords` check if `participants.length()= 0`. If so mark, `_winnerReq.fulfilled = true;` and return

Resolution: Fixed

5.2.3 L-rank percentage bonus values are different in the docs

Severity: *Medium risk*

Context: `Constants.sol#L32-L33`

Description: In docs we have:

- 31-60 days: +8%
- 61-120 days: +13%

While in the code it is:

- 31-60 days: +7%
- 61-120 days: +12%

Recommendation: Change code to follow the docs.

Resolution: Fixed

5.2.4 Staking bonus is applied on an entire duration range

Severity: *Medium risk*

Context: `Constants.sol#L32-L33`

Description: Based on how long a user stakes for, they receive a bonus % on their stake. However, the bonus is applied universally across an entire range. For example:

Staking for any duration between 91-365 days will yield 10% bonus, staking a day extra or less has no good/bad consequences. Users are not incentivized to stake for anything than `minDuration` for a given bonus % e.g

```
5% bonus - stake for 40 days
10% bonus - stake for 91 days
20% bonus - stake for 366 days
30% bonus - stake for 731 days
```

Staking for 2 years (731 days) yields the same bonus as staking for 4 years (1480 days)

Recommendation: Implement linear interpolation formula for the staking period from the lowest % to the highest based on the staking time.

Resolution: Fixed

5.3 Low risk

5.3.1 `swapTitanXForLotusAndBurn` prioritizes incentive fee over treasury allocation

Severity: *Low risk*

Context: `BuyAndBurn.sol#L151`

Description: In `BuyAndBurn.swapTitanXForLotusAndBurn` there is a user incentive of 1.5% that is being deducted from the `amountAllocated`, afterward a volt treasury % (16.7) is taken out of which performs a titan->volt swap and the volt is sent to a treasury contract.

The issue here is that the 2 fees are applied one after the other instead of on the entire `amountAllocated`

Both fees (presumably) should be applied on the entire sum and subtracted from it instead of applying a tax on an already taxed amount The following will result in a slight % discrepancy from the described in the docs, where it is stated that 40% from mint titanX distribution buys and burns lotus and another 8% fills volt treasury.

Recommendation: Consider first calculating `forVoltTreasury` and then on the rest `incentive` and `titanXToSwapAndBurn`.

Resolution: Acknowledged

5.3.2 `Lotus::_createUniswapV3Pool()` - Lotus/Volt pool could be created with a manipulated/skewed price

Severity: *Low risk*

Context: `Lotus.sol#L100`

Description: The protocol calculates the initial price for the Volt/Lotus UniV3 pool by using the output of the Quoter#quoteExactInputSingle() function in `Lotus#_createUniswapV3Pool()` to calculate the volt amount for the initial price. However, the small liquidity in the TITANX/Volt pool makes it susceptible to price manipulation. A malicious actor can perform a sandwich attack by devaluing TITANX relative to Volt just before the `quoteExactInputSingle()` call. This manipulation results in the protocol receiving fewer Volt tokens than expected, leading to the Volt/Lotus pool being initialized with a skewed and unintended price.

Recommendation: Consider providing the price from outside.

Resolution: Acknowledged

5.3.3 Consider emitting events on important state changes

Severity: *Low risk*

Context: Everywhere

Description: Places: - `Lotus.sol` - `setBnB` - `setStaking` - `setMining` - `SwapActions`: - `changeSlippageAdmin` - `changeSlippageConfig` - `Mining`: - `changeLpSlippage`

Recommendation: Emit event on important state changes.

Resolution: Fixed

5.4 Informational

5.4.1 Move `MIN_DURATION` and `MAX_DURATION` to `Constants.sol`

Severity: *Informational*

Context: `Staking.sol`

Resolution: Acknowledged

5.4.2 `swapExactInput` - `getTwapAmount(tokenIn, tokenOut, tokenInAmount)` should only be called if `minAmountOut = 0`

Severity: *Informational*

Context: `SwapActions.sol`

Description:
`swapExactInput` - `getTwapAmount(tokenIn, tokenOut, tokenInAmount)` should only be called if `minAmountOut = 0` since that's the only time twapAmount and slippage are used.

Resolution: Acknowledged

5.4.3 `if (isParticipant(_participant)) return;` check in `LotusBloom::participate` function is redundant

Severity: *Informational*

Context: `LotusBloom.sol`

Description:
`if (isParticipant(_participant))return;` check in `LotusBloom::participate` function is redundant. `EnumerableSet.add` function check it internally.

Resolution: Fixed

5.4.4 In `getDayEnd` instead of hardcoding 14 hours, use the constant `TURN_OVER_TIME`, so it mimics the rest of the code

Severity: *Informational*

Context: `Time.sol`

Resolution: Acknowledged

5.4.5 Consider switching places of `ACTIVE` and `CLAIMED` in `MinerStatus` enum

Severity: *Informational*

Context: `Mining.sol#L39-L42`

Description:
Consider switching places of `ACTIVE` and `CLAIMED` in `MinerStatus` enum, because currently default value is `ACTIVE` and user can claim non-existent miners.

Resolution: Fixed

5.4.6 Wrong natspecs comments

Severity: *Informational*

Context: `Errors.sol#L50`, `BuyAndBurn.sol#L138`

Description:

- Wrong natspec for `notGt` modifier in `Errors.sol#L50`
  - The natspec states `Modifier to ensure the second value is not greater than the first value.`, but in reality we ensure that first value is not greater than second value.
- For `swapTitanXForLotusAndBurn` -> `@notice Swaps TitanX for Volt and burns the Volt tokens` is wrong, because we burn Lotus.

Resolution: Fixed

5.4.7 `Mining::_claim` - `noAddress(0)` modifier is redundant

Severity: *Informational*

Context: `Mining.sol#L337`

Description:
`Mining::_claim` has a useless `noAddress(0)` check, it's impossible for `_user == address(0)` as `_claim` is always called with `msg.sender`

Resolution: Fixed

5.4.8 Consider making keyHash and introducing a setter

Severity: *Informational*

Context: `Mining.sol#L337`

Description:
Consider making `keyHash` a storage variable and introducing a setter because currently, the contract will rely on on one VRF gas lane, which may delay randomness requests in times of high network activity.

Resolution: Acknowledged