Possum Security Review

Version 1.0

22-12-2023

Conducted by:

Solthodox and MaslarovK, Independent Security Researchers

Table of Contents

1 About Solthodox

Solthodox is a smart contract developer and independent security researcher experienced in Solidity smart contract development and transitioning to security. With +1 year of experience in the development side, he has been joining security contests in the last few months. He also serves as a smart contract developer at Unlockd Finance, where he has been involved in building defi yield farming strategies to maximze the APY of it's users.

2 About MaslarovK

MaslarovK is an independent security researcher from Bulgaria with 3 years of experience in Web2 development. His curiosity and love for decentralisation and transparency made him transition to Web3. He has secured various protocols through public contests and private audits.

3 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**.

4 Risk classification

- **High** - Issues that lead to the loss of user funds.Such issues include:
  - Direct theft of any user funds, whether at rest or in motion.
  - Long-term freezing of user funds.
  - Theft or long term freezing of unclaimed yield or other assets.
  - Protocol insolvency
- **Medium** - Issues that lead to an economic loss but do not lead to direct loss of on-chain assets.Examples are:
  - Gas griefing attacks (make users overpay for gas)
  - Attacks that make essential functionality of the contracts temporarily unusable or inaccessible.
  - Short-term freezing of user funds.
- **Low** - Issues where the behavior of the contracts differs from the intended behavior (as described in the docs and by common sense), but no funds are at risk.

4.1 Actions required by severity level

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

5 Executive summary

Overview

Project Name	Possum TimeRift
Repository	https://github.com/PossumLabsCrypto/TimeRift
Commit hash	c18c975291d14d5f62bab94308f4b0a20d560565
Documentation	Not Provided
Methods	Manual review & testing

Scope

contracts/TimeRift.sol

Issues Found

High risk	0
Medium risk	0
Low risk	0
Informational	5

6 Findings

6.1 Informational

6.1.1 Users could distribute an unwanted amount

Severity: *Informational*

Context: TimeRift.sol#266

Description: The `distributeEnergyBolts` function will distribute all of the user's energy bolts balance when the input `amount` is greater than the user's bolt balance, potentially resulting in a unwanted behaviour.

Recommendation: Revert when the input amount is greater than the user's bolt balance, and reserve the `type(uint256).max` input for the case where user intends to withdraw all of his balance, this will help prevent unwanted behaviours:

```
    if(_amount == type(uint256).max){
        _amount = userStake.energyBolts;
    }
    else if (userStake.energyBolts < _amount) {
       revert InsufficientBoltBalance();
    }
```

Resolution: Not resolved.

6.1.2 Consider hardcoding predefined values

Severity: *Informational*

Context: TimeRift.sol#L34

Description: There are some values that are predefined and don't need to be defined in the constructor, and can be hardcoded instead, and avoid incorrect constructor values. This are the values that are known before deployment: - FLASH token address: 0xc628534100180582E43271448098cb2c185795BD - PSM token address: 0x17A8541B82BF67e10B0874284b4Ae66858cb1fd5 - Withdraw penaly percentage: 2% - Energey bolts accrual rate: 150%

Additionally, if this values are hardcoded as `constant` they are gonna be directly written to the contract bytecode saving gas when reading them on-chain.

Recommendation: Harcode this values as `constant` instead of initializing them in the constructor.

Resolution: Not resolved.

6.1.3 Use a `rewardRatePerSecond` to calculate rewards

Severity: *Informational*

Context: TimeRift.sol#L216

Description: the contract calculates the accrued rewards by using a overcomplex formula, that calculates them by multiplying `userStake.stakedTokens` by the time ellapsed from `userStake.lastCollectTime` and then multiplies by the APR and divides again by `SECONDS_PER_YEAR`.

```
uint256 energyBoltsCollected = ((time - userStake.lastCollectTime) *
    userStake.stakedTokens *
    ENERGY_BOLTS_ACCRUAL_RATE) / (100 * SECONDS_PER_YEAR);
```

Recommendation: This could be simplified calculating a `rewardRatePerSecond` value that defines the number of reward tokens per staked tokens per second, and to calculate the contract only needs to multiply the staked amount by the time elapsed and rewards per second. We highly recommend using 30 decimals precision for this matter, so no precision is lost, and using a `mulDiv` function from Openzeppelin's Math library to prevent "phantom overflow":

```
using Math for uint256;

//...

// 30 decimals precision
uint256 constant PRECISION = 1e30;
// 1.5 / seconds per year scaled to 30 decimals
uint256 constant REWARD_RATE_PER_SECOND = 317097919837645865O431;

//...

// calculations will be simpler and more efficient
uint256 elapsed = block.timestamp - userStake.lastCollectTime;
uint256 energyBoltsCollected = (userStake.stakedTokens).mulDiv(
   REWARD_RATE_PER_SECOND * elapsed, PRECISION)
```

Resolution: Not resolved.

6.1.4 Use an optimized library for transfers

Severity: *Informational*

Context: TimeRift.sol#82

Description: The contract uses OZ `SafeERC20` to securely interact with the tokens involved, but for this specific case it's not necessary since the tokens used by this contract are trusted. The contract could use other gas-optimized libraries for the transfers, and reduce gas costs.

Recommendation: Use an optimized transfer libraru such as Solady's SafeTransferLib.

Resolution: Not resolved.

6.1.5 Consider adding a no-penalization exit period

Severity: *Informational*

Context: TimeRift.sol#186

Description: The contract penalizes users(2% of the staked amounts) that withdraw and exit before the `_MINIMUM_STAKE_DURATION` has passed for allocating some exchange balance that will not be used to them. If a user wants to withdraw short after depositing he will still be penalized, even he didn't cause much struggle to the contract.

Recommendation: Consider adding a `regret period` where users can withdraw with no penalization.

Resolution: Not resolved.

6.2 Developer recommendations

6.2.1 Use `vm.expectRevert` to test reverting cases

Severity: *Informational*

Context: Out of scope

Description: In the tests the dev has used raw calls to the contract and checked the success status to ensure it failed when expecting it to revert. This unusual approach loses the opportunity of using Foundry's `expectRevert`cheat, that even allows to expect an specific error message giving more accuracy to the tests.

Recommendation: Check this reference in the Foundry documentation.

Resolution: Not resolved.