Possum Security Review

Version 2.0

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
Resolution 1	8a21bddea6ec6b7807dc7bab1c60e3ab03263d8f
Documentation	Not Provided
Methods	Manual review & testing

Scope

contracts/TimeRift.sol

Issues Found

High risk	0
Medium risk	0
Low risk	0
Informational	7

6 Findings

6.1 Informational

6.1.1 Off by one in `distributeEnergyBolts`

Severity: *Low risk*

Context: TimeRift.sol#272

Description: The `distributeEnergyBolts` function handles rewards distributed to the destination and the sender based on the current PSM token balance. The total amount of PSM tokens distributed each time equals `_amount * 2`. However, when `available_PSM` equals this value (there are still enough tokens available), the contract takes a different path when `available_PSM <= _amount * 2`. This has no impact on the final amounts distributed, as the contract still distributes `rest` to the destination and `_amount` to the sender, and both amounts end up being the same.

```
// if available_PSM = _amount * 2
if (available_PSM <= _amount * 2) {
    //...
    // we enter here
    else {
        // rest = _amount so no impact
        uint256 rest = available_PSM - _amount;

        userStake.exchangeBalance += _amount;
        exchangeBalanceTotal += _amount;
        userStake.energyBolts -= _amount;
        PSM_distributed += rest;
        IERC20(PSM_ADDRESS).safeTransfer(_destination, rest);

        emit EnergyBoltsDistributed(
            msg.sender,
            _amount,
            _destination,
            rest
        );
    }
    //...
```

Recommendation: Correct the off-by-one issue in the “if” statement to ensure accurate conditional checking.

```
if (available_PSM < _amount * 2) {
    //...
```

Resolution: Resolved.

Correcting this off-by-one issue ensures that the conditional check accurately reflects the intended logic of the contract, potentially preventing confusion and enhancing code readability.

6.1.2 Users could distribute an undesired amount to a destination

Severity: *Informational*

Context: TimeRift.sol#266

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

Recommendation: To mitigate this risk, it is advised to implement proper input validation. Specifically, revert the transaction if the input `_amount` surpasses the user's available bolt balance. Additionally, to allow users to withdraw their entire balance, reserve the `type(uint256).max` input for this purpose. The suggested code modification as follows:

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

Resolution: Resolved.

6.1.3 Users could stake an undesired amount

Severity: *Informational*

Context: TimeRift.sol#147

Description: The `stake` function currently allows staking only a portion of the desired amount if `available_PSM` is less than the specified amount. This behavior may lead to an undesired outcome, as it locks the user's funds unless they choose to withdraw with a penalty.

Recommendation: To prevent this undesired behavior, it is recommended to add a check within the `stake` function and revert if the specified amount exceeds the available PSM balance.

```
if (_amount > available_PSM) {
    revert InsufficientAvailablePSM();
}
```

Resolution: Resolved.

Implementing this recommendation ensures that users cannot stake an amount greater than the available PSM balance, reducing the risk of unintended consequences and providing a better input sanity.

6.1.4 Consider hardcoding predefined values

Severity: *Informational*

Context: TimeRift.sol#L34

Description: Certain parameters within the smart contract have predefined values known prior to deployment, eliminating the necessity for dynamic assignment in the constructor. To enhance efficiency and eliminate the risk of incorrect constructor values, consider hardcoding the following constants: - FLASH token address: `0xc628534100180582E43271448098cb2c185795BD` - PSM token address: `0x17A8541B82BF67e10B0874284b4Ae66858cb1fd5` - Withdraw penaly percentage: `2` - Energey bolts accrual rate: `150`

Hardcoding these values as `constant` not only reflects their static nature but also results in direct inclusion within the contract bytecode. This approach offers gas savings during on-chain retrieval.

Recommendation: To optimize gas usage and streamline contract deployment, it is advisable to replace the constructor initialization with hardcoded constants.

Resolution: Resolved.

6.1.5 Use a `rewardRatePerSecond` to calculate rewards

Severity: *Informational*

Context: TimeRift.sol#L216

Description: The current implementation of the contract employs a complex formula to calculate accrued rewards by multiplying `userStake.stakedTokens` with the time elapsed since `userStake.lastCollectTime`. This result is then further multiplied by the ENERGY_BOLTS_ACCRUAL_RATE and divided by `SECONDS_PER_YEAR`.

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

Recommendation: To simplify and improve efficiency, it is suggested to introduce a `rewardRatePerSecond` variable. This variable represents the number of reward tokens per staked token per second.It is highly recommended using 30 decimals precision for this variable. By using OpenZeppelin's `mulDiv` function from the Math library to prevent "phantom overflow," the contract can then calculate rewards more efficiently with increased precision.

```
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 = 3170979198376458650431;

//...

// Simplified calculation for more efficiency
uint256 elapsed = block.timestamp - userStake.lastCollectTime;
uint256 energyBoltsCollected = (userStake.stakedTokens).mulDiv(
    REWARD_RATE_PER_SECOND * elapsed, PRECISION);
```

Resolution: Not resolved.

By implementing this recommendation, the contract can streamline reward calculations, enhance precision, and potentially reduce gas costs associated with the reward calculation process.

6.1.6 Use an optimized library for transfers

Severity: *Informational*

Context: TimeRift.sol#L82

Description: The contract currently utilizes OpenZeppelin's `SafeERC20` to ensure secure interactions with tokens. However, considering that the tokens involved in this contract are trusted, the use of `SafeERC20` may result in unnecessary gas costs. Alternative, gas-optimized libraries, such as Solady's SafeTransferLib, can be considered for more efficient transfers.

Recommendation: To potentially reduce gas costs associated with token transfers, it is recommended to explore alternative gas-optimized libraries for transfers. Solady's SafeTransferLib is mentioned as an example. However, the specific library chosen should be reviewed for compatibility and thoroughly tested before implementation.

Resolution: Not resolved.

By adopting a more gas-optimized library for token transfers, the contract may benefit from reduced gas costs, especially in scenarios where the trustworthiness of the involved tokens allows for such optimizations.

6.1.7 Use `uint96` types in the `Stake` struct

Severity: *Informational*

Context: TimeRift.sol#L99

Description: The `Stake` struct in the contract currently uses `uint256` types, each occupying a full 32 bytes storage slot. This storage packing results in inefficiency, utilizing 5 words of storage and increasing gas costs for accessing these variables. Given that the maximum potential amount of rewards a user could receive is `500_000_000 * 10 ** 18`, a 96-bit value would be more than sufficient for this purpose.

```
// 5 slots
struct Stake {
    uint256 lastStakeTime; // 32 bytes
    uint256 lastCollectTime; // 32 bytes
    uint256 stakedTokens; // 32 bytes
    uint256 energyBolts; // 32 bytes
    uint256 exchangeBalance; // 32 bytes
}
```

Recommendation: To optimize gas usage and reduce storage costs, consider using `uint96` types to represent the values of the struct. This alternative storage packing reduces the storage slots used from 5 to less than 2.

```
// < 2 slots
struct Stake {
    uint96 lastStakeTime; // 12 bytes
    uint96 lastCollectTime; // 12 bytes
    uint96 stakedTokens; // 12 bytes
    uint96 energyBolts; // 12 bytes
    uint96 exchangeBalance; // 12 bytes
}
```

Resolution: Not resolved.

Implementing this recommendation would lead to reduced gas costs and more efficient storage utilization, particularly beneficial in scenarios where gas optimization is crucial, such as high-frequency contract interactions.

6.2 Developer recommendations

6.2.1 Use `vm.expectRevert` to test reverting cases

Severity: *Informational*

Context: Out of scope

Description: In the tests, raw calls to the contract are made, and the success status is checked to ensure failure when expecting a revert. This approach deviates from the conventional usage of Foundry's `expectRevert` cheat, which offers additional features such as the ability to expect specific error messages, providing more accuracy in testing scenarios.

Recommendation: Consider leveraging Foundry's `expectRevert` cheat for testing scenarios that involve expecting reverts. This cheat enhances the precision of tests and allows for more detailed error message checking. Refer to the Foundry documentation for guidance on its usage.

Resolution: Not resolved.

By adopting `vm.expectRevert` and related Foundry cheats, developers can enhance the quality and accuracy of their tests, leading to more robust and reliable smart contract testing practices.