Possum Core Audit Report

Version 1.0

Mahdi Rostami

July 29, 2024

Possum Core Audit Report

Mahdi Rostami

July 18, 2024

Prepared by: Mahdi Rostami [Twitter]

Table of Contents

Possum Core Security Review

A security review of the Possum-core, GitHub smart contracts protocol was done by mahdi rostami . This audit report includes all the vulnerabilities, issues and code improvements found during the security review.

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 find as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**."

- Secureum

Impact

- **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.

Actions required by severity level

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

Executive summary

Overview

Project Name	PossumCore
Repository	Link
Commit Hash	e386112
Docs	No docs
Methods	Manual Review

Scope

File
PossumCore

Compatibilities

- Solc Version: 0.8.19
- Chain(s) to deploy contract to: Arbitrum

Known Issues

None

Issues found

Severity	Count
High	0
Medium	3
Low	0
Informational	1
Enhancement	1
Info/Gas	1

Findings

Medium Severity

[M-1] unstakeAndClaim Function Fails to Reset stakeEndTime on Zero Balance

Description: The unstakeAndClaim function does not reset the stakeEndTime for a user when their balance becomes zero. This oversight can cause issues if the user wants to stake again.

Scenario:

User A stakes an amount for 1 year. After 1 month, User A earns some rewards and decides to forfeit the rewards and unstake. User A's stake balance becomes zero. When User A attempts to stake again, the lock time is unnecessarily set based on the previous stakeEndTime (in this case, 9 months remaining).

Impact: Users face unnecessary minimum lock times for new stakes due to the residual stakeEndTime.

POC:

```
function test_reStake() public {
    uint256 amount = 1e18;
    uint256 duration = 60 * 60 * 24 * 365; // 1 year
    vm.startPrank(Alice);
    psm.approve(address(coreContract), 1e55);
    coreContract.stake(amount, duration);
    vm.warp(block.timestamp + (60 * 60 * 24 * 30)); // after 1 month
    coreContract.unstakeAndClaim(amount);

    // stake again for 1 month
    coreContract.stake(amount, (60 * 60 * 24 * 30)); // 1 month
    (, uint256 stakeEndTime,,,,) = coreContract.stakes(Alice);
    stakeEndTime = (stakeEndTime - block.timestamp) / (60 * 60 * 24);
    assert(stakeEndTime == 335);
}
```

Logs:

```
[PASS] test_reStake() (gas: 244652)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 529.20ms
    (1.35ms CPU time)
```

Mitigation: Add a check for the user's balance in unstakeAndClaim and reset the stakeEndTime if the balance is zero.

Remark: Fixed. commit

[M-2] Users Can Exploit High APR Without Locking by Distributing Fragments After Unstake and Claim

Description: Users can manipulate the current implementation to earn the highest APR and accumulate rewards without actually locking their tokens. This is achieved by staking, gaining fragments, unstaking and then distributing those fragments to earn rewards.

Impact: This vulnerability allows users to unfairly accumulate rewards by bypassing the intended locking mechanism, leading to an uneven distribution of rewards and potentially draining the reward pool.

Scenario: - User stakes a amount for a long duration (e.g., 1 year) to get the highest APR. - After a short period (e.g., 1 day), the user calls `unstakeAndClaim`. - The user gains fragments with a high APR and distributes those fragments. - The user then stakes a minimal amount (e.g., 1 PSM) with zero duration and quickly unstakes to claim the rewards.

POC:

```
function test_test() public {
    vm.startPrank(Bob);
    psm.approve(address(coreContract), 1e55);
    uint256 amount = 1e24;
    uint256 oneDay = 60 * 60 * 24;
    uint256 i;
    uint256 fragments;
    while (i < 3) {
        uint256 balance = psm.balanceOf(Bob);
        console.log("balance", balance);
        coreContract.stake(amount + fragments, SECONDS_PER_YEAR); //
            Highest APR
        vm.warp(block.timestamp + oneDay);
        coreContract.unstakeAndClaim(amount + fragments);
```

```
            uint256 fragmentsSaved = coreContract.getFragments(Bob);
            fragments = fragmentsSaved;
            coreContract.distributeCoreFragments(PERMANENT_I,
                fragmentsSaved);
            coreContract.stake(1, 0);
            coreContract.unstakeAndClaim(fragments + 1);
            console.log("balance", psm.balanceOf(Bob));
            console.log("gain", psm.balanceOf(Bob) - balance);
            i = i + 1;
        }
}
```

Logs:

```
Ran 1 test for test/PossumCoreTest.t.sol:PossumCoreTest
[PASS] test_test() (gas: 819144)
Logs:
  balance 100000000000000000000000
  balance 100019726027397260273972 60
  gain 19726027397260273972 60
  balance 100019726027397260273972 60
  balance 100039490966410208294239 06
  gain 19764939012948020266 46
  balance 100039490966410208294239 06
  balance 100059255982180316027320 13
  gain 19765015770107733081 07
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 1.16s
    (4.15ms CPU time)
```

Mitigation: Reset users accumulated fragments if they unstake before the end of the lock commitment (just like rewards). This ensures that users cannot exploit the system by claiming rewards without fulfilling their lock duration.

Remark: Fixed. commit & commit2

[M-3] Inaccurate Calculation Results in Unfair APR

Description: The current implementation of the weighted average APR calculation results in unfair distribution of rewards due to inaccuracies in the logic.

Impact: This causes an unfair distribution of rewards among users.

Proof of Concept (PoC):

To illustrate the issue, change the `_getFragmentsAPR` function to a public function and add the following test:

```
function test_apr1() public {
    uint256 duration = 60 * 60 * 24 * 100; // 100 days
    uint256 pastAprLesserThanNow = 1200;
    uint256 pastAprBiggerThanNow = 1300;
    uint256 pastAmount = 10e22;
    uint256 amount = 5e20;
    uint256 apr = coreContract._getFragmentsAPR(pastAmount, amount,
        duration, pastAprLesserThanNow);
    console.log("frag apr last APR is less than now", apr);

    apr = coreContract._getFragmentsAPR(pastAmount, amount, duration,
        pastAprBiggerThanNow);
    console.log("frag apr last APR is bigger than now", apr);
    uint256 totalAmount = pastAmount + amount;
    apr = coreContract._getFragmentsAPR(0, totalAmount, duration, 0);
    console.log("frag apr", apr);
}
```

Log:

```
[PASS] test_apr1() (gas: 16467)
Logs:
  frag apr last APR is less than now 1208
  frag apr last APR is bigger than now 1307
  frag apr 2843
```

As shown in the log, the APR values differ even though the duration is the same(funds will be locked for 100 days from now), leading to unfair reward distribution.

Mitigation: The weighted apr, is designed to incentivise users. it has a downside for users with lower APR who want to lock in longer to gain a higher APR.

```
        /// @dev Calculate APR for the new stake amount
        uint256 newAPR = MIN_APR + (((MAX_APR - MIN_APR) * _newDuration
            ) / MAX_STAKE_DURATION);

        /// @dev Calculate the APR weight of old and new stake
        uint256 newStakeWeight = newAPR * _newAmount;
        uint256 oldStakeWeight = _currentAPR * _earningBalance;

       /// @dev Calculate weighted average of both APRs
        fragmentsAPR = (newStakeWeight + oldStakeWeight) / (_newAmount
            + _earningBalance);
+      fragmentsAPR  = fragmentsAPR  ? fragmentsAPR  > newAPR : newAPR;
```

this line just ensures that users who are willing to lock longer will not affected by their past low APR

Remark: Fixed. commit

Informational Severity

Rounding Errors Cause No APR Difference for Durations with a Difference of Less than 85 Minutes

Description: Due to rounding errors in the `_getFragmentsAPR` function, there is no differences in the calculated APRs for durations with a difference of less than 85 minutes. This can lead to unfair distribution of rewards.

Impact: This issue results in unfair distribution of rewards, as the APR differences may not accurately reflect the actual staking durations for periods less than 85 minutes apart.

Proof of Concept (PoC):

To illustrate the issue, change the `_getFragmentsAPR` function to a public function and add the following test:

```
function test_apr2(uint256 duration, int256 min) public {
    vm.assume(duration > 1 && duration < 60 * 60 * 24 * 365);
    vm.assume(min > 5120);
    if ((uint256(min) + duration) < 60 * 60 * 24 * 365) {
        uint256 amount = 1e18;
        uint256 apr1 = coreContract._getFragmentsAPR(0, amount,
            duration, 0);
        duration += uint256(min);
        uint256 apr2 = coreContract._getFragmentsAPR(0, amount,
            duration, 0);
        assertNotEq(apr1, apr2);
    }
}
```

Mitigation: To mitigate this issue, increase the precision of the APR calculations by adjusting the constants used in the function. Specifically, add one more digit to the following variables:

```
- uint256 public constant MAX_APR = 7200; // Accrual rate of CF at
   maximum stake duration (10000 = 100%)
- uint256 public constant MIN_APR = 1200; // Accrual rate of CF at
   stake duration = 0 (10000 = 100%)
- uint256 private constant APR_SCALING = 10000;
+ uint256 public constant MAX_APR = 72_000; // Accrual rate of CF at
   maximum stake duration (100,000 = 100%)
+ uint256 public constant MIN_APR = 12_000; // Accrual rate of CF at
   stake duration = 0 (100,000 = 100%)
+ uint256 private constant APR_SCALING = 100_000;
```

By increasing the precision of these variables, the APR calculations will become more accurate, resulting in fairer reward distributions for all stakers.

Remark: Acknowledged.

Enhancement Severity

Unnecessary Restriction on Disabling Permanent Whitelist Destinations

Description: The `updateWhitelist` function currently does not allow for disabling the `PERMANENT_I`, `PERMANENT_II`, and `PERMANENT_III` addresses. This restriction is unnecessary and prevents the owner from managing the distribution of funds effectively.

Impact: This restriction can lead to inefficient fund management. For instance, if most users choose `PERMANENT_I`, the owner cannot redistribute funds to other destinations.

Mitigation: Allow the owner to disable any two of the permanent addresses, ensuring at least one remains active to manage the flow of funds effectively.

```
 function updateWhitelist(address _destination, bool _listed) external onlyGuardian {
     if (_destination == address(0)) {
         revert InvalidAddress();
     }

     whitelist[_destination] = _listed;

     /// @dev Emit the event that the whitelist was updated
     emit WhitelistUpdated(_destination, _listed);

+    /// @dev Prevent disabling all permanent destinations
+    if (!whitelist[PERMANENT_I] && !whitelist[PERMANENT_II] && !whitelist[PERMANENT_III]) {
+        revert PermanentDestination();
+    }
 }
```

This modification allows the owner to disable any two of the permanent addresses while ensuring at least one remains active, enabling better fund management.

Remark: Implemented. commit

Info Gas

- No reentrancy path found, delete import reentrancy and nonReentrant modifier https://github.com/PossumLabsC
  https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/src/PossumC
  https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/src/PossumC
- redundant error https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496
- redundant import of SafeERC20 and ERC20 https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd

- redundant complete import of IERC20 contract, contract just use transferFrom, transfer, balanceOf remove https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d
- redundant varibale remove https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365
- name https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/src/Po CoreFragmentsAPR must be coreFragmentsAPR
- wrong docs https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/s L28
- wrong error https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/s https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/src/PossumC L179
- check if first https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/s https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/src/PossumC L179 https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/src/Pos https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/src/PossumC L250 https://github.com/PossumLabsCrypto/Core/blob/d2fe070c770d46dd4d2cca6b1365e039df0496d9/src/Pos
- (MAX_APR - MIN_APR) could be immutable or constant

Remark: Cleaned. commit & commit