RaidTreasury Audit Report

Prepared by: yappy-yum

Table of Contents

Protocol Summary

RaidTreasury has 20 Gold. Whoever deposits (can be accrued) the most funds will become the owner of this protocol, thereby withdrawing 20 Gold.

Disclaimer

yappy-yum makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. yappy-yum is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Executive Summary

During the security review, I have discovered multiple vulnerabilities. We identified 1 high-severity vulnarability, 2 low-severiy vulnarabilities and 1 Info. One of the serious vulnarability is that anyone can become owner easily and claim the 20 Gold with just very little investment, while the others are more towards on restriction and gas optimization. Below, I have provided in-depth details of all the code reviews

Issues Found

Severity	Number of issues found
High	1
Medium	0
Low	2
Info	1
Gas Optimizations	0
Total	**4**

Findings

High

[H-1] Missing Ownership Validation in `RaidTreasury::receive`

Description:
The `RaidTreasury::receive` function lacks proper validation when assigning ownership. Ownership is granted to the sender who has contributed the most funds via the `RaidTreasury::contribute` function. Upon contract deployment, the minimum contribution threshold is set to 1e22 wei.

However, the current implementation of the `receive` function only checks that:

- `msg.sender` sends a non-zero amount of Ether
- `msg.sender` has made a prior contribution via contribute.

No further validation is performed before granting ownership. As a result, any user can contribute any amount, then call the receive function again with a trivial amount to be recognized as the owner.

This faulty logic allows the attacker to immediately become the contract owner and call `RaidTreasury::withdraw` to illegitimately claim the 20 Gold.

Impact:

An attacker can become the contract owner with minimal contributions via `RaidTreasury::contribute`, then potentially bypassing the intended logic of rewarding the highest contributor by navigating

`RaidTreasury::receive`. This enables unauthorized withdrawal of the 20 Gold.

Proof of Concept:

1. `Hacker` calls `RaidTreasury::contribute` with `msg.value` set to 1 Wei
2. `Hacker` then sends a small amount of ETH (as little as 1 Wei) directly to the contract, triggering the `RaidTreasury::receive` function, which **erroneously grants ownership** to `Hacker`.
3. As the new owner, `Hacker` is able to call `RaidTreasury::withdraw` function and drain the 20 Gold

```
    function test_steal_Gold() public {
        assertEq(GOLD.balanceOf(Hacker), 0);

        vm.deal(Hacker, 100 ether);
        vm.startPrank(Hacker);

        // 1. Send a small amount via `contribute` function
@>      RT.contribute{value: 1}();

        // 2. Trigger the `receive()` function by sending ETH directly
@>      (bool ok, ) = address(RT).call{value: 1}("");
        assert(ok);

        // 3. The `receive()` function does not properly check conditions
        //    before granting ownership
@>      assertEq(RT.owner(), Hacker);

        // 4. As the new owner, Hacker calls `withdraw` and drains 20 Gold
        //    Hacker just got 20 Gold for 2 wei !!!
@>      RT.withdraw();
        assertEq(GOLD.balanceOf(Hacker), 20 ether);
        assertEq(GOLD.balanceOf(address(RT)), 0 ether);

        vm.stopPrank();
    }
```

This test passes, confirming that the Hacker successfully becomes the owner and withdraws 20 Gold for just 2 Wei.

```
Ran 1 test for test/Lesson-5/RaidTreasury.t.sol:RaidTreasuryT
[PASS] test_steal_Gold() (gas: 104613)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 2.05ms (274.30µs CPU
time)
```

Recommended Mitigation:

Unify the execution logic of the `RaidTreasury::contribute` and `RaidTreasury::receive` functions to ensure proper checks before granding ownership.

1. Create a helper `_contribute()` as shown below:

```
    function _contribute(address _depositor, uint _value) internal {
        require(_value < 0.001 ether);
        require(withdrawn == false, "RaidTreasury: Gold Token Withdrawn");
        treasury[_depositor] += _value;

        if (treasury[_depositor] > treasury[owner]) {
            owner = _depositor;
        }
    }
```

2. Refactor both `contribute()` and `receive()` functions to use `_contribute()`

```diff
    function contribute() public payable {
-       require(msg.value < 0.001 ether);
-       require(withdrawn == false, "RaidTreasury: Gold Token Withdrawn");
-       treasury[msg.sender] += msg.value;

-       if (treasury[msg.sender] > treasury[owner]) {
-           owner = msg.sender;
-       }
+       _contribute(msg.sender, msg.value);
    }

    .
    .
    .

    receive() external payable {
-       require(msg.value > 0 && treasury[msg.sender] > 0);
-       owner = msg.sender;
+       _contribute(msg.sender, msg.value);
    }
```

Low

[L-1] Redundant Calls in `RaidTreasury::withdraw`

Description:

The withdraw function in the `RaidTreasury::withdraw` contract is responsible for transferring the entire balance of the gold token (20 Gold) to the contract owner. However, the function lacks two important safety checks:

1. Unchecked transfer result: The return value of `gold.transfer` is not validated. As per the ERC-20 standard, transfer returns a boolean indicating success or failure. If the token transfer fails silently, the function will still proceed and making unessacery execution.

2. Unchecked withdrawal state: The function does not check whether the funds have already been withdrawn by validating the `RaidTreasury::withdrawn` state variable. As a result, the function can be called multiple times even after the 20 Gold has already been claimed. These repeated calls have no effect but still consume unnecessary gas.

Impact:

Furthur attempt of calling `RaidTreasury::withdraw` after the initial withdrawal will not revert and will effectively perform "no action", wasting gas for the caller. Although this does not lead to loss of funds or a critical vulnerability, it results in inefficient execution.

Proof of Concept:

The following test demonstrates that calling `RaidTreasury::withdraw` multiple times does not revert and continues to execute, despite the 20 Gold having already been withdrawn. This confirms the absence of a state check preventing redundant calls.

```
    function test_unchecked_withdraw() public {
        console.log("Owner Gold Balance Before: ", GOLD.balanceOf(Owner));

        // 1. First call - expected behavior, transfer succeeds
        vm.prank(Owner);
>@      RT.withdraw();
        console.log("First Withdraw Hit, Owner balance: ", GOLD.balanceOf(Owner));

        // 2. Second call - no revert, no change in balance (redundant)
        vm.prank(Owner);
>@      RT.withdraw();
        console.log("Second Withdraw Hit, Owner balance: ",
GOLD.balanceOf(Owner));

        // 3. Third call - same as above, still redundant
        vm.prank(Owner);
>@      RT.withdraw();
        console.log("Third Withdraw Hit, Owner balance: ", GOLD.balanceOf(Owner));
    }
```

below is the output log for the test:

```
[PASS] test_unchecked_withdraw() (gas: 80985)
Logs:
  Deploying to Anvil ...
  Owner Gold Balance Before:  0
  First Withdraw Hit, Owner balance:  20000000000000000000
  Second Withdraw Hit, Owner balance:  20000000000000000000
  Third Withdraw Hit, Owner balance:  20000000000000000000

Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 25.02ms (3.32ms CPU
time)
```

```
Ran 1 test suite in 46.51ms (25.02ms CPU time): 1 tests passed, 0 failed, 0
skipped (1 total tests)
```

Recommended Mitigation:

Add a state check at the beginning of the `RaidTreasury::withdraw` function to prevent redundant execution once the withdrawal has already occurred. Additionally, ensure that the result of the `gold.transfer` call is validated to comply with the ERC-20 standard.

This prevents unnecessary gas usage and ensures the contract state reflects successful transfers.

```diff
    function withdraw() public onlyOwner {
+       require(withdrawn == false, "RaidTreasury: Gold Token Withdrawn");

-       gold.transfer(msg.sender, gold.balanceOf(address(this)));
+       bool ok = gold.transfer(msg.sender, gold.balanceOf(address(this)));
+       require(ok, "RaidTreasury: Withdraw Transfer Failed");

        withdrawn = true;
    }
```

[L-2] Zero Sends in `RaidTreasury::contribute` function

description:

Assuming the changes from [H-1] have been implemented, the `RaidTreasury::_contribute` function currently lacks a check to prevent users from sending zero Kaia contributions. The function only ensures that the contribution is less than 0.001 Kaia, but it does not verify that the value is greater than 0.

```
    function _contribute(address _depositor, uint _value) internal {
        require(_value < 0.001 ether);
        .
        .
        .
```

As a result, users can send zero funds, causing the function to execute without transferring any value.

Impact

This flaw allows users to execute the contribution logic without actually sending any funds, leading to wasted gas usage.

Proof of Conduct:

```
    function test_can_send_zeroKAIA() public {

        RT.contribute(); // no funds sent
        RT.contribute(); // no funds sent
        RT.contribute(); // no funds sent
        RT.contribute(); // no funds sent

    }
```

This test confirms that the function executes successfully without sending any funds, as shown in the following output:

```
[PASS] test_can_send_zeroKAIA() (gas: 20253)
Suite result: ok. 1 passed; 0 failed; 0 skipped; finished in 3.58ms (253.20µs CPU
time)

Ran 1 test suite in 24.07ms (3.58ms CPU time): 1 tests passed, 0 failed, 0 skipped
(1 total tests)
```

Recommended Mitigation:

To prevent this issue, modify the `RaidTreasury::_contribute` function to ensure that the contribution is greater than zero.

```
    function _contribute(address _depositor, uint _value) internal {
-       require(_value < 0.001 ether);
+       require(_value < 0.001 ether && _value > 0);
```

This change ensures that only non-zero contributions are accepted, eliminating the possibility of "empty" transactions.

Informational

[I-1] Redundant Getter for `RaidTreasury::treasury` Mapping

description:

The `RaidTreasury::treasury` mapping is marked as `public`, which automatically generates a getter function allowing external access by key `(address)`. However, a custom getter function `RaidTreasury::getContribution` is also defined, which returns the caller’s contribution `treasury[msg.sender]`.

This introduces redundancy in the contract interface and may slightly increase gas costs or code size unnecessarily.

```
    mapping(address => uint256) public treasury;

    function getContribution() public view returns (uint256) {
        return treasury[msg.sender];
    }
```

Recommended Mitigation:

To reduce redundancy and maintain cleaner encapsulation, mark the `RaidTreasury::treasury` mapping as `private`. This avoids the automatic getter generation and relies solely on the custom getter, which has more specific logic tailored to the caller's context.

```diff
-   mapping(address => uint256) public treasury;
+   mapping(address => uint256) private treasury;
```

This change can slightly reduce contract size and interface clutter, and enforces that contributions can only be read through the defined logic.