Raisebox Faucet Audit Report

Version 1.0

Aldo Surya Ongko

October 15, 2025

Raisebox Faucet Audit Report

Aldo Surya Ongko

Oct 15, 2025

Prepared by: Aldo Surya Ongko

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

I 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

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

Audit Details

The findings described in this document correspond the following commit hash:

```
daf8826cece87801a9d18745cf77e11e39838f5b
```

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

Issues found

Severity	Number of issues found
High	2
Medium	5

Severity	Number of issues found
Low	8

Findings

High

[H-1] Reentrancy Attack in claimFaucetTokens Function

Description: The `claimFaucetTokens()` function violates the Checks-Effects-Interactions pattern by making an external ETH transfer call before updating critical state variables. This allows malicious contracts to re-enter the function during the ETH transfer and potentially drain both tokens and ETH.

Impact: Attackers can drain the entire faucet balance (both tokens and ETH) through recursive calls, completely breaking the faucet mechanism and causing financial loss.

Proof of Concept:

```
// Malicious contract
contract MaliciousReceiver {
    RaiseBoxFaucet faucet;
    uint256 callCount;

    constructor(address _faucet) {
        faucet = RaiseBoxFaucet(_faucet);
    }

    receive() external payable {
        if (callCount < 5) { // Limit to prevent gas issues
            callCount++;
            faucet.claimFaucetTokens(); // Reentrant call
        }
    }
}
```

Recommended Mitigation: Implement OpenZeppelin's ReentrancyGuard modifier and follow Checks-Effects-Interactions pattern:

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

contract RaiseBoxFaucet is ERC20, Ownable, ReentrancyGuard {
    function claimFaucetTokens() public nonReentrant {
        // All checks first
        // Update state variables before external calls
```

```
7           lastClaimTime[faucetClaimer] = block.timestamp;
8           dailyClaimCount++;
9           // Then external interactions
10      }
11  }
```

[H-2] Block Timestamp Manipulation Vulnerability

Description: The contract relies on `block.timestamp` for cooldown periods and daily resets. Miners can manipulate block timestamps within a ±15 second window, potentially allowing users to bypass cooldown periods or manipulate daily claim resets.

Impact: Users could claim tokens more frequently than intended, potentially draining the faucet faster than designed and breaking the economic model.

Proof of Concept:

```
1  // Current vulnerable code
2  if (block.timestamp < (lastClaimTime[faucetClaimer] + CLAIM_COOLDOWN))
       {
3      revert RaiseBoxFaucet_ClaimCooldownOn();
4  }
5  // Miner can manipulate timestamp to make this check pass early
```

Recommended Mitigation: Use block numbers instead of timestamps with appropriate conversion:

```
1  uint256 public constant CLAIM_COOLDOWN_BLOCKS = 17280; // ~3 days at 15
       s/block
2
3  mapping(address => uint256) private lastClaimBlock;
4
5  if (block.number < (lastClaimBlock[faucetClaimer] +
       CLAIM_COOLDOWN_BLOCKS)) {
6      revert RaiseBoxFaucet_ClaimCooldownOn();
7  }
```

Medium

[M-1] Daily Reset Logic Inconsistency

Description: The contract uses two different mechanisms for daily resets - one for ETH drips using day calculation and another for token claims using timestamp comparison. This creates potential desynchronization between ETH and token claim limits.

Impact: Inconsistent daily limits could allow users to exploit timing differences between the two reset mechanisms, potentially claiming more than intended daily limits.

Proof of Concept:

```
// ETH drip reset
uint256 currentDay = block.timestamp / 24 hours;
if (currentDay > lastDripDay) {
    lastDripDay = currentDay;
    dailyDrips = 0;
}

// Token claim reset (different logic)
if (block.timestamp > lastFaucetDripDay + 1 days) {
    lastFaucetDripDay = block.timestamp;
    dailyClaimCount = 0;
}
```

Recommended Mitigation: Standardize both reset mechanisms to use the same calculation method:

```
uint256 currentDay = block.timestamp / 1 days;
if (currentDay > lastResetDay) {
    lastResetDay = currentDay;
    dailyDrips = 0;
    dailyClaimCount = 0;
}
```

[M-2] Logic Error in Burn Function

Description: The `burnFaucetTokens()` function transfers the entire contract balance to the owner but only burns the specified amount, potentially leaving excess tokens with the owner.

Impact: The owner could accumulate tokens beyond intended limits, potentially affecting the token distribution mechanism and creating centralization risks.

Proof of Concept:

```
function burnFaucetTokens(uint256 amountToBurn) public onlyOwner {
    // If contract has 2000 tokens and amountToBurn is 500:
    _transfer(address(this), msg.sender, balanceOf(address(this))); // Transfers 2000 to owner
    _burn(msg.sender, amountToBurn); // Only burns 500, owner keeps 1500
}
```

Recommended Mitigation: Only transfer the amount to be burned:

```
function burnFaucetTokens(uint256 amountToBurn) public onlyOwner {
    require(amountToBurn <= balanceOf(address(this)), "Insufficient
        balance");
    _transfer(address(this), msg.sender, amountToBurn);
    _burn(msg.sender, amountToBurn);
}
```

[M-3] Centralization Risk

Description: The owner has extensive control over critical contract functions including unlimited token minting (subject to balance check), daily limit adjustments, and ETH management without proper governance mechanisms.

Impact: Single point of failure that could lead to abuse of privileges, unfair token distribution, or complete control over faucet operations.

Proof of Concept: Owner can: - Mint unlimited tokens when balance is low - Arbitrarily adjust daily limits - Pause ETH drips indefinitely - Control all ETH refills

Recommended Mitigation: Implement role-based access control and consider multi-signature requirements:

```
import "@openzeppelin/contracts/access/AccessControl.sol";

contract RaiseBoxFaucet is ERC20, AccessControl {
    bytes32 public constant ADMIN_ROLE = keccak256("ADMIN_ROLE");
    bytes32 public constant MINTER_ROLE = keccak256("MINTER_ROLE");

    // Implement time delays for critical operations
    // Consider multi-signature requirements for high-value operations
}
```

[M-4] Missing Emergency Controls

Description: The contract lacks emergency pause mechanisms that could be crucial during security incidents or when vulnerabilities are discovered.

Impact: Inability to quickly halt operations during emergencies could lead to continued exploitation of vulnerabilities and greater financial losses.

Proof of Concept: No emergency stop mechanism exists to halt `claimFaucetTokens()` operations during security incidents.

Recommended Mitigation: Implement pausable functionality:

```solidity
import "@openzeppelin/contracts/security/Pausable.sol";

contract RaiseBoxFaucet is ERC20, Ownable, Pausable {
    function claimFaucetTokens() public whenNotPaused {
        // Function logic
    }

    function emergencyPause() external onlyOwner {
        _pause();
    }
}
```

[M-5] Gas Limit Vulnerabilities

Description: The `claimFaucetTokens()` function performs multiple state changes, external calls, and complex calculations in a single transaction, potentially consuming excessive gas.

Impact: Under certain conditions, high gas consumption could lead to transaction failures, effectively denying service to legitimate users.

Proof of Concept: Function performs: timestamp checks, balance validations, ETH transfers, state updates, and event emissions all in one call.

Recommended Mitigation: Optimize gas usage and consider splitting complex operations:

```solidity
// Separate ETH claiming from token claiming
function claimTokens() external {
    // Only handle token claims
}

function claimFirstTimeEth() external {
    // Separate function for first-time ETH claims
}
```

Low

[L-1] Unspecific Solidity Pragma

Description: Using `^0.8.30` allows compilation with future compiler versions that may introduce breaking changes or unexpected behavior.

Impact: Potential compatibility issues and unexpected behavior with future compiler versions.

Proof of Concept:

```
pragma solidity ^0.8.30; // Allows any 0.8.x version
```

Recommended Mitigation: Use specific compiler version:

```
pragma solidity 0.8.30;
```

[L-2] PUSH0 Opcode Compatibility

Description: Solidity 0.8.20+ uses Shanghai EVM features including PUSH0 opcode, which may not be supported on all L2 chains or testnets.

Impact: Contract deployment may fail on chains that don't support PUSH0 opcode.

Recommended Mitigation: Specify EVM version in compiler settings or use older Solidity version:

```
// In foundry.toml or hardhat.config.js
evm_version = "london"
```

[L-3] State Variables Should Be Immutable/Constant

Description: Variables `faucetDrip`, `sepEthAmountToDrip`, `dailySepEthCap`, and `blockTime` are only set in constructor or never change but aren't marked as immutable/constant.

Impact: Increased gas costs for storage operations and reduced code clarity.

Proof of Concept:

```
uint256 public faucetDrip; // Only set in constructor
uint256 public blockTime = block.timestamp; // Never changes
```

Recommended Mitigation:

```
uint256 public immutable faucetDrip;
uint256 public immutable sepEthAmountToDrip;
uint256 public immutable dailySepEthCap;
uint256 public constant blockTime = block.timestamp; // If truly
    constant
```

[L-4] Public Functions Not Used Internally

Description: Multiple functions are marked `public` but only called externally, wasting gas on internal call optimizations.

Impact: Slightly increased gas costs and reduced code clarity.

Proof of Concept: Functions like `getBalance()`, `getClaimer()`, etc. are only called externally.

Recommended Mitigation: Change visibility to `external`:

```
function getBalance(address user) external view returns (uint256) {
    return balanceOf(user);
}
```

[L-5] Missing Events for State Changes

Description: Critical state changes like daily limit adjustments don't emit events, making off-chain monitoring difficult.

Impact: Reduced transparency and difficulty in tracking contract state changes.

Proof of Concept:

```
function adjustDailyClaimLimit(uint256 by, bool increaseClaimLimit)
    public onlyOwner {
    // State change without event emission
}
```

Recommended Mitigation: Add events for all state changes:

```
event DailyClaimLimitAdjusted(uint256 newLimit, uint256 adjustment,
    bool increased);

function adjustDailyClaimLimit(uint256 by, bool increaseClaimLimit)
    public onlyOwner {
    // Logic
    emit DailyClaimLimitAdjusted(dailyClaimLimit, by,
        increaseClaimLimit);
}
```

[L-6] Large Numeric Literal

Description: Large number `1000000000 * 10 ** 18` should use scientific notation for better readability.

Impact: Reduced code readability and potential for errors in large numbers.

Proof of Concept:

```
uint256 public constant INITIAL_SUPPLY = 1000000000 * 10 ** 18;
```

Recommended Mitigation:

```
uint256 public constant INITIAL_SUPPLY = 1e27; // 1 billion tokens with 18 decimals
```

[L-7] Unused Code Elements

Description: Unused error `RaiseBoxFaucet_CannotClaimAnymoreFaucetToday` and unused import `IERC20` indicate incomplete code cleanup.

Impact: Increased contract size, potential confusion, and reduced code quality.

Proof of Concept:

```
import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol"; // Unused
error RaiseBoxFaucet_CannotClaimAnymoreFaucetToday(); // Unused
```

Recommended Mitigation: Remove unused code elements or implement their intended functionality.

[L-8] Spelling Error in License

Description: License identifier contains typo: `SPDX-Lincense-Identifier` should be `SPDX-License-Identifier`.

Impact: Incorrect license identification and reduced professionalism.

Proof of Concept:

```
// SPDX-Lincense-Identifier: MIT
```

Recommended Mitigation:

```
// SPDX-License-Identifier: MIT
```