Protocol Audit Report

Version 1.0

Dakray Security

November 19, 2025

RaiseBox Faucet Report

Dakray Security

October 16, 2023

Prepared by: [Dakray Security] Lead Security Researcher: - Dakota Rogers

Table of Contents

Protocol Summary

RaiseBox Faucet is a simple token-drip system deployed on Sepolia that distributes 1,000 RaiseBox test tokens every 3 days to users. First-time users also receive 0.005 Sepolia ETH to cover gas fees for interacting with the future RaiseBox testnet protocol, which will require these tokens for participation.

The protocol involves three actors:

Owner – manages token minting, burning, claim limits, and refills ETH to the contract.

Claimer – any user who calls claimFaucetTokens() to receive tokens (and ETH on first claim).

Donators – addresses that send Sepolia ETH directly to the faucet to keep it funded.

Overall, RaiseBox Faucet provides a controlled, rate-limited mechanism for distributing test assets to support development and testing of the upcoming RaiseBox ecosystem.

Disclaimer

The Dakray Security 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

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

Owner: - deploys contract, - mint initial supply and any new token in future, - can burn tokens, - can adjust daily claim limit, - can refill sepolia eth balance Claimer: - can claim tokens by calling the claimFaucetTokens function of this contract. Donators: - can donate sepolia eth directly to contract # Executive Summary

This was a contest on Cyfrin Codehawks Oct 9th - 16th 2025

Issues found

Severity	Number of Issues found
High	1
Medium	3
Info	7

Findings

[H-1] Reentrancy in `claimFaucetTokens` allows attackers to drain faucet tokens and ETH

Description:
The `claimFaucetTokens` function in `RaiseBoxFaucet` sends ETH to the caller before updating internal state. If the caller is a contract, its fallback function can re-enter `claimFaucetTokens` and claim tokens/ETH multiple times in a single transaction, bypassing the intended limits.

```
function claimFaucetTokens() public {
    // ... checks omitted ...
    (success, ) = faucetClaimer.call{value: sepEthAmountToDrip}(); // @> External call before state update
    _transfer(address(this), faucetClaimer, faucetDrip);           // @> State update after external call
    dailyClaimCount++;
    lastClaimTime[faucetClaimer] = block.timestamp;
}
```

External call before internal state update violates the (CEI) pattern, enabling reentrancy. As a result, a malicious contract can repeatedly call `claimFaucetTokens` in a single transaction, draining the faucet.

Impact:
Any attacker can drain the faucet's tokens and ETH by exploiting reentrancy, bypassing daily claim limits and cooldowns. This can result in complete loss of faucet funds.

Proof of Concept:

POC

Place the following into `RaiseBoxFaucet.t.sol`:

```
contract ReentrancyAttacker {
    RaiseBoxFaucet public faucet;
    uint256 public reentrancyCount;

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

    receive() external payable {
        if (reentrancyCount == 0) {
            reentrancyCount++;
            faucet.claimFaucetTokens();
        }
    }

    function attack() external {
        faucet.claimFaucetTokens();
    }
}

function testProvesReentrancy() public {
    ReentrancyAttacker attacker = new ReentrancyAttacker(payable(address(raiseBoxFaucet)));
    vm.deal(address(attacker), 1 ether);

    vm.prank(address(attacker));
    attacker.attack();

    uint256 claimed = raiseBoxFaucet.getBalance(address(attacker));
    uint256 singleClaim = raiseBoxFaucet.faucetDrip();
    assertGt(claimed, singleClaim, "Contract is not vulnerable to
        reentrancy");
}
```

Recommended Mitigation:
Update all state before making any external calls (CEI Pattern), or use OpenZeppelin's `ReentrancyGuard`
.

```
function claimFaucetTokens() public nonReentrant {
-   (success, ) = faucetClaimer.call{value: sepEthAmountToDrip}();
-   _transfer(address(this), faucetClaimer, faucetDrip);
-   dailyClaimCount++;
-   lastClaimTime[faucetClaimer] = block.timestamp;
+   _transfer(address(this), faucetClaimer, faucetDrip);
+   dailyClaimCount++;
+   lastClaimTime[faucetClaimer] = block.timestamp;
+   (success, ) = faucetClaimer.call{value: sepEthAmountToDrip}();
}
```

[M-1] Centralization Risk: Owner Privileges

Description:
The contract owner has significant control over critical functions, including minting and burning tokens, adjusting claim limits, refilling ETH, and pausing ETH drips. If the owner is compromised or acts maliciously, they can drain funds, disrupt faucet operation, or devalue the token supply.

Impact:
Users must trust the owner to act honestly and securely. A compromised or malicious owner can: - Mint excessive tokens, devaluing the faucet - Drain ETH or tokens from the contract - Prevent users from claiming tokens or ETH by pausing drips or setting limits to zero

This centralization creates a medium severity risk for users and the protocol.

Proof of Concept:

- `mintFaucetTokens`, `burnFaucetTokens`, `adjustDailyClaimLimit`, `refillSepEth`
, and `toggleEthDripPause` are all owner-only functions.
- The owner can call these functions at any time, affecting contract state and user access.

Recommended Mitigation:
Consider implementing multi-signature ownership or decentralized governance to reduce single-point-of-failure risk. Clearly document owner privileges for users.

[M-2] Potential for Unintended ETH Locking

Description:
The contract accepts ETH via `receive()` and `fallback()` functions, but only the owner can move ETH out. If the owner loses access to their account, all ETH in the contract is permanently locked and cannot be recovered by donors or users.

Impact:
Permanent loss of donated ETH if the owner is lost or becomes inaccessible. This risk is present in every deployment unless the owner is a contract or multisig wallet with robust recovery mechanisms.

Proof of Concept:

```
receive() external payable { // @> accepts ETH
    emit SepEthDonated(msg.sender, msg.value);
}
// Only owner can move ETH out
function refillSepEth(uint256 amountToRefill) external payable
   onlyOwner { // @> owner-only
    // ...existing code...
}
```

Recommended Mitigation:
Consider using a multi-signature wallet for ownership or implementing a recovery mechanism for owner access. Clearly document the risk for users and donors.

[M-3] No Limit on Owner Minting

Description:
The owner can mint tokens to the contract as long as the contract's balance is below a threshold, but there is no upper bound on the total supply. This allows the owner to repeatedly mint tokens, potentially causing excessive inflation.

Impact:
Potential for excessive inflation and devaluation of faucet tokens if the owner mints repeatedly. This risk is present in every deployment unless a supply cap is enforced.

Proof of Concept:

```
function mintFaucetTokens(address to, uint256 amount) public onlyOwner
    {
    // ...existing code...
    _mint(to, amount); // @> no supply cap
}
```

Recommended Mitigation:
Implement a maximum supply cap for faucet tokens and enforce it in the minting logic.

[I-1] State Variable Not Declared Constant (`blockTime` assignment -> unnecessary gas usage)

Description:
The contract sets the variable `blockTime` to the current block timestamp at deployment, and it never changes after that. Declaring it as `constant` would save gas and make it clear that its value is fixed.

```
uint256 public blockTime = block.timestamp; // @> Should be declared
    constant
```

Impact:
Slightly higher gas costs for reading and storing the variable, and less clarity for users and auditors about its immutability.

Proof of Concept:

POC

After deployment, `blockTime` never changes and always returns the same value.

```
uint256 public blockTime = block.timestamp;
```

Recommended Mitigation:

```
- uint256 public blockTime = block.timestamp;
+ uint256 public constant blockTime = block.timestamp;
```

[I-2] State Variable Not Declared Immutable

Description:
The contract sets the variables `faucetDrip`, `sepEthAmountToDrip`, and `dailySepEthCap` in the constructor, and their values never change after deployment. Declaring these as `immutable` would save gas and make it clear that these values are fixed for the lifetime of the contract.

```
uint256 public faucetDrip;            // @> Should be declared immutable
uint256 public sepEthAmountToDrip;    // @> Should be declared immutable
uint256 public dailySepEthCap;        // @> Should be declared immutable
```

Impact:
Slightly higher gas costs for reading and storing these variables, and less clarity for users and auditors about their mutability.

Proof of Concept:

POC

These variables are only set in the constructor and never changed afterwards.

```
RaiseBoxFaucet.sol:
    uint256 public faucetDrip;
    uint256 public sepEthAmountToDrip;
    uint256 public dailySepEthCap;
```

After deployment, their values never change.

Recommended Mitigation:

```
- uint256 public faucetDrip;
+ uint256 public immutable faucetDrip;

- uint256 public sepEthAmountToDrip;
+ uint256 public immutable sepEthAmountToDrip;

- uint256 public dailySepEthCap;
+ uint256 public immutable dailySepEthCap;
```

[I-3] Public Function Not Used Internally

Description:
Several functions in `RaiseBoxFaucet.sol` are marked as `public` but are not called internally within the contract. In Solidity, functions that are not used internally should be marked as `external` to optimize gas usage and clarify their intended access.

Impact:
Marking functions as `public` when they are only intended to be called externally can result in slightly higher gas costs and may reduce code clarity. This is a minor issue but can be improved for best practices and efficiency.

Proof of Concept:

The following functions are declared as `public` but are not used internally:

- `mintFaucetTokens(address to, uint256 amount)` [Line: 109]
- `burnFaucetTokens(uint256 amountToBurn)` [Line: 127]
- `adjustDailyClaimLimit(uint256 by, bool increaseClaimLimit)` [Line: 142]
- `claimFaucetTokens()` [Line: 161]
- `getBalance(address user)` [Line: 274]
- `getClaimer()` [Line: 278]
- `getHasClaimedEth(address user)` [Line: 284]
- `getUserLastClaimTime(address user)` [Line: 290]
- `getFaucetTotalSupply()` [Line: 295]
- `getContractSepEthBalance()` [Line: 300]
- `getOwner()` [Line: 305]

Recommended Mitigation:
Change the visibility of these functions from **`public`** to `external` where appropriate. This will reduce gas costs and improve code clarity.

[I-4] Unused Error

Description:
The contract declares the custom error `RaiseBoxFaucet_CannotClaimAnymoreFaucetToday` but does not use it anywhere in the code. Unused errors increase code size and reduce clarity.

Impact:
Unused errors can lead to confusion for maintainers and auditors, and slightly increase contract bytecode size.

Proof of Concept:

- `error RaiseBoxFaucet_CannotClaimAnymoreFaucetToday();` [Line: 87]

Recommended Mitigation:
Remove the unused error or use it in relevant require/revert statements to improve code clarity and reduce bytecode size.

[I-5] Unused Import

Description:
The contract imports `IERC20` from OpenZeppelin but does not use it anywhere in the code. Redundant imports increase code size and reduce clarity.

Impact:
Unused imports can lead to confusion for maintainers and auditors, and slightly increase contract bytecode size.

Proof of Concept:

- `import {IERC20} from "@openzeppelin/contracts/token/ERC20/IERC20.sol";` [Line: 4]

Recommended Mitigation:
Remove the unused import statement to improve code clarity and reduce bytecode size.

[I-6] Incorrect SPDX License Identifier

Description:
The contract uses a misspelled SPDX license identifier: `// SPDX-Lincense-Identifier: MIT`. The correct spelling is `// SPDX-License-Identifier: MIT`.

Impact:
Some verification platforms and tooling may fail to recognize the license, potentially causing issues with contract verification or compliance.

Proof of Concept:

```
// SPDX-Lincense-Identifier: MIT // @> typo
```

Recommended Mitigation:
Correct the spelling to `// SPDX-License-Identifier: MIT` at the top of the contract.

[I-7] No Validation on Token Decimals

Description:
The contract assumes 18 decimals for the faucet token in calculations, but does not enforce this. If deployed with a different ERC20 implementation, calculations may be incorrect.

Impact:
Incorrect token calculations and user confusion if the token uses a different number of decimals.

Proof of Concept:

```
uint256 public constant INITIAL_SUPPLY = 1000000000 * 10 ** 18; // @>
    assumes 18 decimals
```

Recommended Mitigation:
Explicitly enforce or validate the token decimals in the contract, or document the requirement for 18 decimals.