plainshift

February 7, 2025

Rainbow Token Launcher

Comprehensive Security Assessment

contact@plainshift.io
plainshift.io

Table of Contents

Disclaimer

This security assessment represents a time-boxed security review using tooling and manual review methodologies. Our findings reflect our comprehensive evaluation of the materials provided in-scope and are specific to the commit hash referenced in this report.

The scope of this security assessment is strictly limited to the code explicitly specified in the report. External dependencies, integrated third-party services, libraries, and any other code components not explicitly listed in the scope have not been reviewed and are excluded from this assessment.

Any modifications to the reviewed codebase, including but not limited to smart contract upgrades, protocol changes, or external dependency updates will require a new security assessment, as they may introduce considerations not covered in the current review.

In no event shall Plainshift's aggregate liability for all claims, whether in contract or any other theory of liability, exceed the Services Fee paid for this assessment. The client agrees to hold Plainshift harmless against any and all claims for loss, liability, damages, judgments and/or civil charges arising out of exploitation of security vulnerabilities in the contracts reviewed.

By accepting this report, you acknowledge that deployment and implementation decisions rest solely with the client. Any reliance upon the information in this report is at your own discretion and risk.
This disclaimer is governed by and construed in accordance with the laws specified in the engagement agreement between Plainshift and the client.

1. Introduction

Plainshift is a full-stack security firm built on the “shift left” security philosophy. We often work with teams early in the product development process to bring security to a greater organizational range than just smart contracts. From the web app, to fuzzing/formal verification, to a team’s operational security, full-stack security can only be achieved by first understanding there is no “scope” to fully protect the users that trust you.

We’re here to meaningfully revolutionize how teams approach security and guide them towards a holistic approach rather than the single sided approach so prevalent today.

Learn more about us at plainshift.io.

1.1. Executive Summary

Plainshift was tasked with reviewing Rainbow’s Token Launcher contracts from February 3rd, 2025 to February 7th, 2025. Within the first 2 days of review, we found and confirmed 2 high, 4 medium, and 1 low severity issues.

1.2. Project Timeline

Date	Phase
`2025-02-03`	Audit Kickoff
`2025-02-07`	End of Audit
`2025-02-07`	Report Delivery

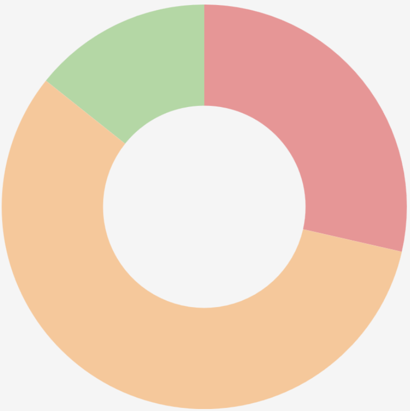
1.3. Scope

Repositories	https://github.com/rainbowfoundation/TokenLauncher
Versions	`534d4ab11bcc74ae62e2de4a4ac0a04bf1c01d89`
Programs	`/src` - `/RainbowSuperToken.sol` - `/RainbowSuperTokenFactory.sol`
Type	Solidity
Platform	EVM-compatible

1.4. Overview of Findings

Our comprehensive review yielded 2 high, 4 medium, and 1 low severity issues.

Severity/Impact Level	Count
High	2
Medium	4
Low	1



2. Detailed Findings

2.1. Unrestricted `launchFromOtherChain` enables frontrunning

Target	`src/RainbowSuperTokenFactory.sol`	Category	Vulnerability
Severity	High	Status	Unresolved

2.1.1. Description

The vulnerability enables any attacker to frontrun a legitimate call to `launchRainbowSuperToken` by invoking `launchWithOtherChain` with identical parameters. Because the deployment uses CREATE2 with a deterministic salt, an attacker can preemptively deploy the token contract at the expected address thereby blocking the intended deployment (which sets up the Uniswap pool).

The factory contract provides two functions for token deployment:

`launchRainbowSuperToken` (lines 305–342):

This function is intended to create a new `rainbowSuperToken` and then set up the associated Uniswap pool.

```
// File: RainbowSuperTokenFactory.sol
     function launchRainbowSuperToken(
...
         string memory tokenURI = string(abi.encode(keccak256(abi.encode(creator, salt,
name, symbol, merkleroot, supply))));

         // Create token
         newToken = new RainbowSuperToken{ salt: keccak256(abi.encode(creator, salt)) }(
             name, symbol, string.concat(baseTokenURI, tokenURI), merkleroot,
airdropAmount, id
         );
...
         IUniswapV3Pool pool =
IUniswapV3Pool(uniswapV3Factory.createPool(address(newToken), address(_pairToken),
POOL_FEE));
...
         newToken.approve(address(nonfungiblePositionManager), lpSupply);
         (uint256 tokenId,,,) = nonfungiblePositionManager.mint(params);
```

`launchFromOtherChain` **(lines 404–429):**

This function is designed to deploy the token on another chain using the same CREATE2 parameters so that the token address remains consistent. However, it does not implement any access control or restrictions.

```
// File: RainbowSuperTokenFactory.sol
     function launchFromOtherChain(
         string memory name,
         string memory symbol,
         bytes32 merkleroot,
         uint256 supply,
         bytes32 salt,
         address creator,
         uint256 originalChainId
     )
         external
         returns (RainbowSuperToken newToken)
     {
         uint256 id;
         assembly {
             id := chainid()
         }

         if (originalChainId == id) revert Unauthorized();

         bool hasAirdrop = merkleroot != bytes32(0);
         (,, uint256 airdropAmount) = calculateSupplyAllocation(supply, hasAirdrop);

         string memory tokenURI = string(abi.encode(keccak256(abi.encode(creator, salt,
name, symbol, merkleroot, supply))));

         newToken = new RainbowSuperToken{ salt: keccak256(abi.encode(creator, salt)) }(
             name, symbol, string.concat(baseTokenURI, tokenURI), merkleroot,
airdropAmount, id
         );     }
```

Because both functions derive the contract address using the same salt (`keccak256(abi.encode(creator, salt))`) and constructor parameters, if an attacker in the same chain calls `launchFromOtherChain` before the legitimate user calls `launchRainbowSuperToken`, the token contract will already exist. Consequently, the subsequent call to `launchRainbowSuperToken` will revert as the contract address is already occupied. This results in the token being deployed without the crucial Uniswap pool initialization, undermining the expected protocol behavior.

Consider the following scenario:

1. On `ChainA`, a user calls `launchRainbowSuperToken`.
2. On the same `ChainA`, an attacker frontruns and calls `launchFromOtherChain`, causing the token deployment to happen on `ChainA`.
3. The transaction from step 1 fails. As a result:
    - The pool is not created by the intended creator.
    - The Uniswap position is not generated.
    - This prevents the ability to claim the fees from the airdrop that was supposed to be distributed to the creator.

```
    function testLaunchTokenRevert() public {
        vm.startPrank(creator1);

        (bytes32 salt,) = findValidSalt(creator1, "Test Token", "TEST", bytes32(0),
INITIAL_SUPPLY);

        // 1. Attacker frontrun and uses launchFromOtherChain to deploy the token
        rainbowFactory.launchFromOtherChain("Test Token", "TEST", bytes32(0),
INITIAL_SUPPLY, salt, address(creator1), 1337);

        // 2. This will revert user call because the token was already deployed by the
frontrunner
        vm.expectRevert();
        RainbowSuperToken token = rainbowFactory.launchRainbowSuperToken("Test Token",
"TEST", bytes32(0), INITIAL_SUPPLY, 200, salt, address(creator1));

        (,,,,,, address creator) = rainbowFactory.tokenFeeConfig(address(token));
        assertEq(creator, address(0));
    }
```

2.1.2. Impact

The legitimate deployment process (which includes setting up the Uniswap liquidity pool) is completely bypassed if an attacker preempts the call. The attack path is straightforward and inexpensive to execute. Since the `launchFromOtherChain` function is public and lacks access control.

2.1.3. Recommendations

Implement access restrictions on the `launchFromOtherChain` function, allowing only the creator to call it.

Regarding this solution, what I'm not 100% convinced about is that deploying tokens on another chain is centralized, allowing only the creator to perform the deployment. This introduces a centralization risk. However, I believe that the solution proposed in the finding "Incorrect chain ID causes token address mismatch" might also address this issue.

```
   function launchFromOtherChain(
        string memory name,
        string memory symbol,
        bytes32 merkleroot,
        uint256 supply,
        bytes32 salt,
        address creator,
        uint256 originalChainId
    )
        external
        returns (RainbowSuperToken newToken)
    {
        uint256 id;
        assembly {
            id := chainid()
        }

        if (originalChainId == id) revert Unauthorized();
+       if (msg.sender != creator) revert Unauthorized();
```

2.1.4. Remediation Status

Unresolved.

2.2. Incorrect chain ID causes token address mismatch

Target	src/RainbowSuperTokenFactory.sol	Category	Vulnerability
Severity	High	Status	Unresolved

2.2.1. Description

The vulnerability affects the deterministic deployment of tokens across chains. The `launchFromOtherChain` function uses the current chain's ID (`id`) as a constructor parameter instead of the `originalChainId`. This causes the token's constructor arguments to differ from the original deployment, leading to a different contract bytecode and therefore a different token address when using CREATE2. This discrepancy prevents the token from being deployed at the expected address across chains, undermining cross-chain interoperability and potentially causing significant operational and economic issues for users who expect to transfer their token to multiple chains.

In the `launchFromOtherChain` function, the contract retrieves the current chain ID and uses it in the token's constructor on line 428:

```
// File: RainbowSuperTokenFactory.sol
     function launchFromOtherChain(
         string memory name,
         string memory symbol,
         bytes32 merkleroot,
         uint256 supply,
         bytes32 salt,
         address creator,
      uint256 originalChainId
     )
         external
         returns (RainbowSuperToken newToken)
     {
       uint256 id;
         assembly {
             id := chainid()
         }

         if (originalChainId == id) revert Unauthorized();

         bool hasAirdrop = merkleroot != bytes32(0);
         (,, uint256 airdropAmount) = calculateSupplyAllocation(supply, hasAirdrop);
```

```
        string memory tokenURI = string(abi.encode(keccak256(abi.encode(creator, salt,
name, symbol, merkleroot, supply))));

       newToken = new RainbowSuperToken{ salt: keccak256(abi.encode(creator, salt))
}(name, symbol, string.concat(baseTokenURI, tokenURI), merkleroot, airdropAmount, id);
```

According to the intended cross-chain deployment process, the token created on the destination chain should mimic the original token deployed on the original chain (using `launchRainbowSuperToken` function), including having the same constructor parameters. However, here the parameter `id` is derived from the current chain's ID (e.g., `ChainB`) instead of using the provided `originalChainId` (e.g., `ChainA`). This discrepancy results in different constructor inputs, leading to a different contract bytecode and thus a different token address on `ChainB`. This misalignment breaks the guarantee of having an identical token address across supported chains.

Consider the following scenario:

1. Alice deploys `TokenA` on `ChainA` using the `launchRainbowSuperToken` function. The function retrieves the chain ID from `ChainA`. For example, assume ChainA's ID is 1. `TokenA` is deployed on `ChainA` with the constructor parameter `id` equal to `1`. The CREATE2 mechanism computes its address based on the provided salt and constructor arguments.

```
        uint256 id;
        assembly {
         id := chainid()
        }

        string memory tokenURI = string(abi.encode(keccak256(abi.encode(creator, salt,
name, symbol, merkleroot, supply))));

        // Create token
        newToken = new RainbowSuperToken{ salt: keccak256(abi.encode(creator, salt)) }(
          name, symbol, string.concat(baseTokenURI, tokenURI), merkleroot,
airdropAmount, id
        );
```

2. Later, Alice attempts to deploy the same `TokenA` address on `ChainB` using the `launchFromOtherChain` function. Inside this function, the contract again retrieves the chain ID—but this time from `ChainB`. Assume ChainB's ID is `42161`. Because the function uses the current chain's ID (`42161`) instead of the `originalChainId` (which should be 1 from ChainA), the constructor parameter for the new token on ChainB is different. The CREATE2 mechanism computes a different token address on `ChainB` due to the altered constructor arguments. **`TokenA` is not deployed at the same address on `ChainB` as it was on `ChainA`.**

```
        uint256 id;
        assembly {
          id := chainid()
        }
...
      newToken = new RainbowSuperToken{ salt: keccak256(abi.encode(creator, salt))
}(name, symbol, string.concat(baseTokenURI, tokenURI), merkleroot, airdropAmount, id)
```

2.2.2. Impact

The token will be deployed at a different address than expected, breaking cross-chain compatibility and interoperability. Token holders and cross-chain applications relying on a consistent token address will face issues such as liquidity fragmentation and inability to transfer tokens across chains.

2.2.3. Recommendations

Modify the function to use `originalChainId` in `launchFromOtherChain` function instead of the current chain’s ID when passing the chain identifier to the token constructor. For example:

```diff
-           newToken = new RainbowSuperToken{ salt: keccak256(abi.encode(creator, salt))
}(name, symbol, string.concat(baseTokenURI, tokenURI), merkleroot, airdropAmount, id);
+           newToken = new RainbowSuperToken{ salt: keccak256(abi.encode(creator, salt))
}(name, symbol, string.concat(baseTokenURI, tokenURI), merkleroot, airdropAmount,
originalChainId);
```

2.2.4. Remediation Status

Unresolved.

2.3. Missing name/ticker checks allow banned tokens

Target	`src/RainbowSuperTokenFactory.sol`	**Category**	Vulnerability
Severity	Medium	**Status**	Unresolved

2.3.1. Description

The factory contract offers two functions for deploying RainbowSuperTokens:

1. **`launchRainbowSuperToken`**

   This function includes checks to prevent the use of restricted or banned names/tickers. For example:

```
// RainbowSuperTokenFactory.sol
         // Name and ticker checks
        if (keccak256(abi.encodePacked(name)) == keccak256(abi.encodePacked("Rainbow")))
{
             revert ReservedName();
         }
         if (keccak256(abi.encodePacked(symbol)) == keccak256(abi.encodePacked("RNBW")))
{
             revert ReservedTicker();
         }
         if (bannedNames[name]) revert BannedName();
         if (bannedTickers[symbol]) revert BannedTicker();
```

2. **`launchFromOtherChain`**

   This function, intended for cross-chain deployments, omits the restricted name/ticker checks. As a result, an attacker can call this function with banned parameters (e.g., `name = "RainbowV2"` and `symbol = "RNBWV2"`) and deploy a token that violates the protocol's naming policies:

```
    /// @return newToken The newly created RainbowSuperToken
    function launchFromOtherChain(
        string memory name,
        string memory symbol,
        bytes32 merkleroot,
        uint256 supply,
```

```
        bytes32 salt,
        address creator,
        uint256 originalChainId
    )
        external
        returns (RainbowSuperToken newToken)
    {
        uint256 id;
        assembly {
            id := chainid()
        }

        if (originalChainId == id) revert Unauthorized();

        bool hasAirdrop = merkleroot != bytes32(0);
        (,, uint256 airdropAmount) = calculateSupplyAllocation(supply, hasAirdrop);

        string memory tokenURI = string(abi.encode(keccak256(abi.encode(creator, salt,
name, symbol, merkleroot, supply))));

        newToken = new RainbowSuperToken{ salt: keccak256(abi.encode(creator, salt)) }(
            name, symbol, string.concat(baseTokenURI, tokenURI), merkleroot,
airdropAmount, id
        );
    }
```

Because `launchFromOtherChain` does not enforce the same name and ticker restrictions as `launchRainbowSuperToken`, an attacker can deploy tokens with banned identifiers, thereby subverting the intended controls.

2.3.2. Impact

The absence of name and ticker validation in the `launchFromOtherChain` function permits an attacker to bypass established restrictions. This may lead to the deployment of tokens with banned or restricted names/tickers.

2.3.3. Recommendations

Implement the same name and ticker restrictions in the `launchFromOtherChain` function as in `launchRainbowSuperToken`.

2.3.4. Remediation Status

Unresolved.

2.4. Lack of `msg.value` check traps ETH

Target	`src/RainbowSuperTokenFactory.sol`	**Category**	Vulnerability
Severity	Medium	**Status**	Unresolved

2.4.1. Description

This vulnerability can lead to significant funds being trapped in the factory contract. If the owner changes `defaultPairToken` using the function `RainbowSuperTokenFactory::setNewPairToken` from WETH to another token (e.g., USDC) and a user inadvertently sends ETH (`msg.value`) with their transaction, the ETH will not be converted or refunded, causing a loss of funds for the user. This undermines user trust and may result in material loss, especially if large amounts of ETH are sent accidentally.

Consider the next scenario:

1. The owner updates `defaultPairToken` from WETH to another token (for example, USDC).
2. A user, unaware of the change, sends ETH (`msg.value > 0`) with their transaction.
3. Because `defaultPairToken` is no longer WETH, the code follows the `else` branch (line 275) and executes a `safeTransferFrom` for USDC. Of course, the user would have to approve the **Factory**, but this could be an issue if the `defaultPairToken` constantly changes from one to another.
4. The sent ETH (`msg.value`) is never processed or refunded, and thus remains trapped in the `RainbowSuperTokenFactory` contract.

```
// File: RainbowSuperTokenFactory.sol
     function launchRainbowSuperTokenAndBuy(
         string memory name,
         string memory symbol,
         bytes32 merkleroot,
         uint256 supply,
         int24 initialTick,
         bytes32 salt,
         address creator,
         uint256 amountIn
     )
         external
         payable
         returns (RainbowSuperToken)
     {
         if (address(defaultPairToken) == address(WETH)) {
             if (msg.value != amountIn) revert InsufficientFunds();
             WETH.deposit{ value: msg.value }();
```

```
        } else {
            defaultPairToken.safeTransferFrom(msg.sender, address(this), amountIn);
        }
```

2.4.2. Impact

Users may accidentally lose their ETH if they send `msg.value` when `defaultPairToken` is not WETH.

Given that the owner can change `defaultPairToken` at any time and users might not be aware of such changes or by chance.

2.4.3. Recommendations

Add a check to ensure that if `msg.value` is nonzero, then `defaultPairToken` must be WETH.

2.4.4. Remediation Status

Unresolved.

2.5. Owner fee config changes cause address inconsistency

Target	`src/RainbowSuperTokenFactory.sol`	**Category**	Vulnerability
Severity	Medium	**Status**	Unresolved

2.5.1. Description

The `launchFromOtherChain` function relies on the mutable `defaultFeeConfig` to compute parameters (specifically the `airdropAmount`) that are passed to the token's constructor, an owner change in `defaultFeeConfig` can alter the constructor parameters. As a consequence, the deployed token's bytecode will differ, and when using CREATE2 with a deterministic salt, this produces a different token address. This breaks the intended cross-chain address consistency, potentially fragmenting liquidity and undermining the protocol's integrity.

The factory contract deploys RainbowSuperTokens using CREATE2 with a salt provided by the user. In the code line 341, the `airdropAmount` is sent:

```
// File: RainbowSuperTokenFactory.sol
         (uint256 lpSupply, uint256 creatorAmount, uint256 airdropAmount) =
calculateSupplyAllocation(supply, hasAirdrop);

         uint256 id;
         assembly {
             id := chainid()
         }

         string memory tokenURI = string(abi.encode(keccak256(abi.encode(creator, salt,
name, symbol, merkleroot, supply))));

         // Create token
         newToken = new RainbowSuperToken{ salt: keccak256(abi.encode(creator, salt)) }(
           name, symbol, string.concat(baseTokenURI, tokenURI), merkleroot,
airdropAmount, id
         );
```

The calculation of `airdropAmount` (and `creatorAmount`) depends on the current `defaultFeeConfig` in `calculateSupplyAllocation` function:

```
             creatorAmount = (totalSupply * defaultFeeConfig.creatorBaseBps) / 10_000;
```

```
            airdropAmount = (totalSupply * defaultFeeConfig.airdropBps) / 10_000;
        } else {
            creatorAmount = (totalSupply * defaultFeeConfig.protocolBaseBps) / 10_000;
           airdropAmount = 0;
```

Additionally, the fee configuration itself can be modified by the owner:

```
function setDefaultFeeConfig(FeeConfig calldata newConfig) external onlyOwner {
        if (newConfig.creatorLPFeeBps > 10_000) revert InvalidFeeSplit();
        if (newConfig.protocolBaseBps + newConfig.creatorBaseBps + newConfig.airdropBps
> 10_000) {
            revert InvalidSupplyAllocation();
        }
        defaultFeeConfig = newConfig;
```

Since `defaultFeeConfig` is mutable, the owner can change it at any time before a cross-chain deployment occurs. As a result, even if the same parameters (creator, salt, name, symbol, merkleroot, supply) are provided, the calculated `airdropAmount` may differ from that used in the original chain deployment. Because the token's constructor arguments affect the contract's deployed bytecode, a change in `airdropAmount` leads to a different computed address via CREATE2. This breaks the intended guarantee that the token will be deployed at the same address across chains.

2.5.2. Impact

The inability to deploy the token at the same address across chains undermines cross-chain interoperability and expected behavior. Since the owner can arbitrarily update `defaultFeeConfig` using the provided setter, an attack or inadvertent change is straightforward to execute.

2.5.3. Recommendations

The user should provide the `airdropAmount` parameter to the `launchFromOtherChain` function as well as the `predictTokenAddress` function. This ensures that the correct address can be calculated properly.

2.5.4. Remediation Status

Unresolved.

2.6. Single-claim restriction blocks multiple allocations

Target	`src/RainbowSuperToken.sol`	**Category**	Vulnerability
Severity	Medium	**Status**	Unresolved

2.6.1. Description

The `claim` function is designed to mint tokens based on a Merkle proof, where the leaf is computed from the `msg.sender` and the claimed `amount`. However, the function uses a boolean flag in the `claimed` mapping to restrict claims on line 094:

```
     function claim(bytes32[] calldata proof, address recipient, uint256 amount)
external onlyOriginalChain {
       if (claimed[msg.sender]) revert AlreadyClaimed();

         claimed[msg.sender] = true;

       bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(msg.sender,
amount))));
         if (!MerkleProofLib.verifyCalldata(proof, merkleRoot, leaf)) {
             revert InvalidProof();
         }
```

Since the Merkle leaf is derived from both the `msg.sender` and `amount` (line 098), if a user were entitled to multiple allocations (i.e., multiple leaf with different amounts due to airdrop dynamics), only the first claim would succeed. Any subsequent claim attempts, even if they pertain to different amounts, will revert due to the already claimed flag.

2.6.2. Impact

Users are prevented from claiming multiple allocations if they are entitled to more than one distribution entry.

2.6.3. Recommendations

Modify the Merkle tree to include a unique identifier for each claim (e.g., an index) so that each claim can be tracked independently:

```
bytes32 leaf = keccak256(abi.encode(msg.sender, amount, claimIndex));
```

Also update the claim tracking mechanism accordingly.

2.6.4. Remediation Status

Unresolved.

2.7. Missing claim event emission in `claim`

Target	`src/RainbowSuperToken.sol`	**Category**	General
Severity	Low	**Status**	Unresolved

2.7.1. Description

The `claim` function mints tokens based on a Merkle proof but does not emit any event after a claim is processed.

```
    function claim(bytes32[] calldata proof, address recipient, uint256 amount)
external onlyOriginalChain {
        if (claimed[msg.sender]) revert AlreadyClaimed();

        claimed[msg.sender] = true;

        bytes32 leaf = keccak256(bytes.concat(keccak256(abi.encode(msg.sender,
amount))));
        if (!MerkleProofLib.verifyCalldata(proof, merkleRoot, leaf)) {
            revert InvalidProof();
        }

        if (amount + totalMintedSupply > maxTotalMintedSupply) {
            amount = maxTotalMintedSupply - totalMintedSupply;
        }

        totalMintedSupply += amount;
        totalSupply += amount;

        // Mint the points to the recipient
        unchecked {
            balanceOf[recipient] += amount;
        }
    }
```

2.7.2. Impact

There is no event emitted to log that a claim/mint has occurred, which means that external observers cannot easily track the action on-chain.

2.7.3. Recommendations

Emit this event at the end of the `claim` function to log the details.

2.7.4. Remediation Status

Unresolved.