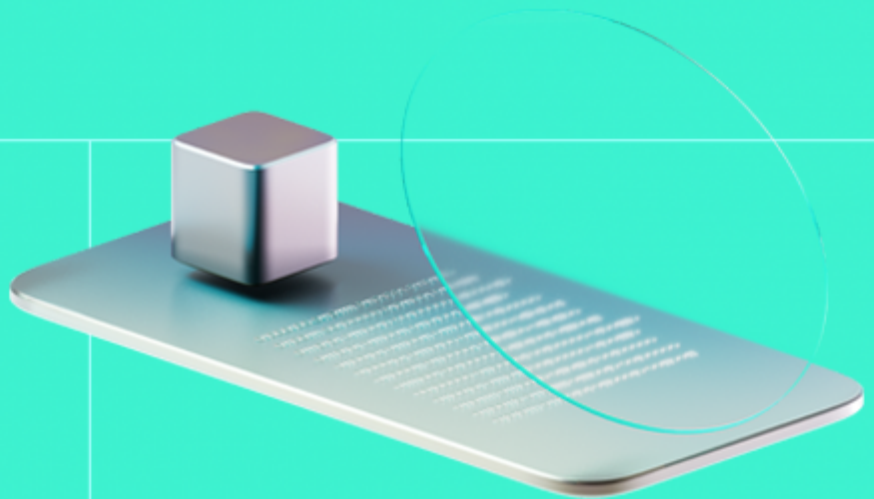




Smart Contract Code Review And Security Analysis Report

Customer: RAIN

Date: 22/08/2025



We express our gratitude to the RAIN team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Rain is a smart contract for the RAIN token, an ERC-20 upgradeable token with controlled inflation mechanics and governance features.

Document

Name	Smart Contract Code Review and Security Analysis Report for RAIN
Audited By	Viktor Lavrenenko, Panagiotis Konstantinidis
Approved By	Ivan Bondar
Website	http://rain.one/
Changelog	19/08/2025 - Preliminary Report
	22/08/2025 - Final Report
Platform	Arbitrum
Language	Solidity
Tags	ERC20; Proxy; Upgradeable; UUPS; Oracle
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/rain1-labs/rain-token
Commit	f439c92
Remediation Commit	7b397e8

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

7	5	1	1
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	0
High	0
Medium	1
Low	0

Vulnerability	Severity
F-2025-12213 - PriceFeed Integration Lacks Safeguards Compromising Minting Policy	Medium
F-2025-12209 - Floating Pragma	Info
F-2025-12210 - Missing Two-Step Ownership Transfer Increases Risk of Permanent Owner Loss	Info
F-2025-12211 - Missing Visibility on totalInflationMinted Prevents Tracking of Total Minted Inflation	Info
F-2025-12214 - Duplicated Token Name and Symbol Can Lead to Confusion	Info
F-2025-12222 - Redundant State Initialization	Info
F-2025-12241 - Misaligned USD Cap Calculation Due to USDT Peg Assumption	Info

Documentation quality

- Functional requirements are provided:
 - The project's purpose is described.
 - Business logic is provided.
 - Use cases are described.
 - Project's features are described.
- Technical description is partially provided:
 - Key function descriptions are described for the files in the scope.
 - Architectural overview is missing.
 - Roles and authorization are provided.
 - No information on used technologies provided.

Code quality

- Best practices are followed.
- The development environment is configured.

Test coverage

Code coverage of the project is **33.33%** (branch coverage).

- Deployment and basic user interactions are partially covered with tests.
- Negative cases coverage is partially missed.
- Interactions by several users are not tested thoroughly.

Table of Contents

System Overview	6
Privileged Roles	6
Potential Risks	7
Findings	8
Vulnerability Details	8
Disclaimers	22
Appendix 1. Definitions	23
Severities	23
Potential Risks	23
Appendix 2. Scope	24
Appendix 3. Additional Valuables	25

System Overview

RAIN is an ERC20 token following contracts:

- **RainToken.sol** - an upgradeable ERC-20 token that implements a controlled inflation system. The contract mints a fixed USD value of tokens (\$50,000) daily to a designated treasury address, with the actual token amount calculated based on real-time price data from an oracle.

It has the following attributes:

- Name: RAIN.
- Symbol: RAIN.
- Decimals: 18.
- Total supply: Not limited.

Privileged roles

- The owner of the `RainToken` can:
 - execute the daily minting via the `dailyMinting()` function.
 - set the address of the price oracle feed via the `setOraclePriceFeed()` function.
 - update the treasury address via the `updateTreasury()` function.
 - upgrade the proxy's implementation via the `upgradeToAndCall()` function.
 - transfer the ownership to another address via the `transferOwnership()` function.

Potential Risks

- The implemented `dailyMinting()` function highly depends on the external RAIN-USD price oracle not covered by the audit. This reliance introduces risks if these external contract is compromised or contains vulnerabilities, affecting the audited project's integrity.
- The token contract's design allows for centralized control over the minting process, posing a risk of unauthorized token issuance, potentially diluting the token value and undermining trust in the project's economic governance.
- Allowing the admin to set oracle addresses without constraints or verification mechanisms introduces the risk of incorrect or malicious oracle selection, affecting the accuracy of data and potentially leading to financial losses.
- The token ecosystem grants a single entity the authority to implement upgrades or changes. This centralization of power risks unilateral decisions that may not align with the community or stakeholders' interests, undermining trust and security.
- The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.
- The contract suite allows for immediate upgrades without a mandatory review or waiting period, increasing the risk of rapid deployment of malicious or flawed code, potentially compromising the system's integrity and user assets.
- The owner holds unilateral authority over minting, oracle configuration, treasury updates, and contract upgrades. Relying on a single key for these critical operations presents a significant centralization risk, which could be mitigated through the adoption of a multi-signature scheme.
- The project depends on a single oracle price feed to determine mint amounts. This reliance introduces risks of inaccurate issuance or denial of service if the feed is manipulated, malfunctioning, or unavailable, directly affecting the system's integrity.
- The whitepaper indicates that minting is intended to occur daily after the 90-day lock period. However, there is a potential risk that minting may be executed at irregular intervals, deviating from the stated schedule.

Vulnerability Details

[F-2025-12213](#) - PriceFeed Integration Lacks Safeguards Compromising Minting Policy - Medium

Description:

The `dailyMinting()` function mints treasury tokens equivalent to a fixed USDT value by retrieving the token price from `oraclePriceFeed.getPrice()`. The current implementation of the `oraclePriceFeed` contract relies directly on a Uniswap v3 spot tick (`slot0`), without any safeguards such as TWAP averaging, staleness detection, sanity limits, or token/pool validation.

```
// Rain.dailyMinting()
function dailyMinting() external onlyOwner {
    ...
    uint256 rainPriceInUSD = oraclePriceFeed.getPrice();
    uint256 amount = (DAILY_INFLATION_USD * WEI) / rainPriceInUSD;
    // manipulable price; risk of division by zero
    ...
}
```

As a result, the mint process can be manipulated by external actors, or fail entirely, leading to inflated token issuance, reduced emissions, or a complete denial of service, specifically:

- **Spot price manipulability:** The oracle fetches the current tick (`slot0`) from Uniswap v3, which is highly sensitive to intra-block price swings and can be manipulated with flash loans or relatively small amounts of capital in low-liquidity pools. If the price is skewed upwards, `(DAILY_INFLATION_USD * 1e18) / price` yields a smaller mint amount, under-issuing tokens compared to the intended policy. Conversely, if the price is pushed downwards, the calculation inflates the minting amount, resulting in excessive token issuance and supply inflation.
- **Division by zero risk:** The code does not check that `rainPriceInUSD > 0`. If the oracle returns zero or an invalid value, the mint reverts, blocking token emissions.
- **Lack of freshness/sanity checks:** No validation exists to ensure the reported price is recent, stable, or within a reasonable range of the last accepted value. Extreme deviations could allow unintended mint sizes.

- **Pool/token mismatch:** The feed allows arbitrary pool/token configuration without enforcing that the pool actually corresponds to `(baseToken, USDT)`. Misconfiguration silently breaks price correctness.
- **Decimal mismatch:** The calculation assumes the PriceFeed's `baseToken` has 18 decimals (1e18). If the `baseToken` in the PriceFeed uses a different decimals value, the quoted price and resulting mint amount will be skewed.

Although only the owner can call `dailyMinting()`, an external attacker can manipulate the pool price right before the owner's transaction to bias mint results. A compromised owner key can trivially abuse this design.

The reliance on an unprotected spot price means the mint process can be deliberately biased or fail entirely, undermining monetary policy and project trust. This results in:

- Over-minting / supply inflation : Manipulated low price yields more tokens minted than intended.
- Under-minting / emission distortion: Inflated price yields fewer tokens, disrupting policy.
- Denial of Service: Zero or invalid price causes minting to revert, halting the minting.
- Loss of trust: A mismatch in the PriceFeed's value causes unit-calculation errors, yielding incorrect mint amounts and undermining confidence in the project's monetary policy.

Assets:

- `src/RainToken.sol` [<https://github.com/rain1-labs/rain-token>]

Status:

Mitigated

Classification

Impact Rate:	4/5
Likelihood Rate:	3/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Medium

Recommendations

Remediation: To make the mint process resilient and align with best practices for oracle integration, we recommend:

- Use a TWAP-based oracle: Have the `PriceFeed` compute and return a time-weighted average price (e.g., 10–30 minutes, liquidity-dependent) rather than an instantaneous quote, and have `Rain` consume that TWAP to mitigate price manipulation.
- Add validation checks: Require `rainPriceInUSD > 0`, enforce freshness/heartbeat requirements, and cap deviations (e.g., $\pm 20\text{--}30\%$) relative to the last accepted price.
- Ensure pool correctness: Verify that the oracle pool matches `(baseToken, USDT)` to prevent misconfiguration.
- Handle decimals properly: Adjust `baseAmount` according to `decimals(baseToken)` to ensure unit consistency.

Resolution:

Mitigated in commit `7b397e8`. The client acknowledged the manipulative nature of spot oracles but only implemented a validation check for zero price, and accepted the remaining risks with the following message:

Since minting can only be called by the owner, if fewer tokens are minted due to a skewed high price, it simply benefits the community and If more tokens are minted than intended, we can simply burn the surplus to keep supply aligned with policy

F-2025-12209 - Floating Pragma - Info

Description:

In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma (e.g., `^0.8.x`) introduces uncertainty, as it allows the contract to be compiled with any version within the specified range. This can result in discrepancies between the compiler used in testing and the one used during deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The project currently uses floating pragma declarations (`^0.8.26`) in its Solidity contracts. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

```
pragma solidity ^0.8.26;
```

Assets:

- `src/RainToken.sol` [<https://github.com/rain1-labs/rain-token>]

Status:

Fixed

Classification

Impact Rate: 2/5

Likelihood Rate: 1/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

It is recommended to lock the pragma version to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler

version, preventing unexpected changes in behavior due to compiler updates. For example, instead of using `^0.8.x`, explicitly define the version with a locked, stable pragma directive.

Before selecting a version, review known bugs and vulnerabilities associated with each Solidity compiler release. This can be done by referencing the official Solidity compiler release notes: [Solidity GitHub releases](#) or [Solidity Bugs by Version](#). Choose a compiler version with a good track record for stability and security.

Resolution:

Fixed in commit `7b397e8`. The pragma version was locked to `0.8.29` to ensure consistent compilation and avoid version related discrepancies.

[F-2025-12210](#) - Missing Two-Step Ownership Transfer Increases Risk of Permanent Owner Loss - Info

Description:

The contract inherits from `OwnableUpgradeable` (via `__Ownable_init(initialOwner)` in the `initialize()` function) to manage administrative control. Ownership can be transferred using the standard `OwnableUpgradeable` pattern, which performs the transfer in a single step without requiring acceptance from the new owner.

```
contract Rain is
    Initializable,
    ERC20Upgradeable,
    OwnableUpgradeable,
    ...
{

    function initialize(
        address initialOwner,
        address treasuryContract
    ) public initializer {
        __ERC20_init("RAIN", "RAIN");
        __Ownable_init(initialOwner);
        ...
    }
}
```

This one-step approach is prone to accidental or misconfigured ownership transfers, such as sending ownership to the wrong address, an address without access, or even a zero address. In such cases, there is no recovery mechanism, leaving the contract permanently inaccessible to its owners.

The lack of a confirmation step leaves the contract vulnerable to irreversible administrative errors, especially during deployment, upgrades, or migration phases where ownership transitions are more likely.

Assets:

- `src/RainToken.sol` [<https://github.com/rain1-labs/rain-token>]

Status:

Fixed

Classification

Impact Rate:

2/5

Likelihood Rate: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation: To mitigate this risk, replace `OwnableUpgradeable` with OpenZeppelin's `Ownable2StepUpgradeable`, which introduces a two-step ownership transfer flow. This pattern requires the new owner to explicitly call `acceptOwnership()` to finalize the transfer, ensuring that only responsive and capable addresses take ownership. This small change significantly improves operational safety without altering core functionality.

Resolution: **Fixed** in commit `7b397e8`. The contract now inherits from `Ownable2StepUpgradeable` instead of `OwnableUpgradeable`, introducing a two-step ownership transfer process that requires the new owner to explicitly accept ownership.

[F-2025-12211](#) - Missing Visibility on totalInflationMinted Prevents Tracking of Total Minted Inflation - Info

Description:

The contract maintains an internal variable `totalInflationMinted`, which is updated each time `dailyMinting()` is called. Its purpose is to track the total number of tokens minted through the inflation mechanism:

```
uint256 totalInflationMinted; // default internal visibility

function dailyMinting() external onlyOwner {
    ...
    uint256 amount = (DAILY_INFLATION_USD * WEI) / rainPriceInUSD;
    ...
    totalInflationMinted += amount;
    ...
}
```

While the variable is correctly updated on each mint operation, it is declared with the default `internal` visibility. As a result, the total minted inflation cannot be directly queried or verified. This reduces visibility into the inflation process, prevents token holders and external users from easily checking the cumulative minted supply, and weakens clarity around the project's monetary policy, undermining the purpose of tracking the total minted supply through this variable.

Assets:

- `src/RainToken.sol` [<https://github.com/rain1-labs/rain-token>]

Status:

Fixed

Classification

Impact Rate:	1/5
Likelihood Rate:	2/5
Exploitability:	Independent
Complexity:	Simple
Severity:	Info

Recommendations

Remediation:

Declare the `totalInflationMinted` variable as `public` so that it can be directly queried on-chain. Exposing this value improves transparency, allows token holders and external users to easily verify the total cumulative minted supply, and ensures the variable serves its intended purpose of tracking overall inflation.

Resolution:

Fixed at commit `7b397e8`. The `totalInflationMinted` variable was made public to allow direct external visibility into the cumulative inflation supply.

[F-2025-12214](#) - Duplicated Token Name and Symbol Can Lead to Confusion - Info

Description:

The contract `Rain` defines identical token name and symbol as `"RAIN"` in the `initialize()` function. While this may be intentional, it can lead to confusion during deployment and interaction, as multiple tokens with identical identifiers complicate differentiation across tools, wallets, or explorers.

```
function initialize(  
    address initialOwner,  
    address treasuryContract  
) public initializer {  
    // Other Code  
    __ERC20_init("RAIN", "RAIN");  
    // Other Code  
}
```

Users often rely on the symbol as a concise identifier and the name for a descriptive label. Having both identical might obscure the token's purpose or identity.

Assets:

- `src/RainToken.sol` [<https://github.com/rain1-labs/rain-token>]

Status:

Accepted

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

To improve clarity and usability, consider using a descriptive `name` and a shorter, unique `symbol` in the `Rain` contract.

Resolution:

Accepted. The client is aware of the finding and accepted the risk.

F-2025-12222 - Redundant State Initialization - Info

Description:

The `initialize` function in the `Rain` smart contract explicitly assigns the value `0` to the `totalInflationMinted` state variable, even though this is the default value for uninitialized storage variables in Solidity.

```
function initialize(  
    address initialOwner,  
    address treasuryContract  
) public initializer {  
    // Other Code  
    totalInflationMinted = 0;  
}
```

Since Solidity automatically initializes storage variables to their default values (e.g., `0` for integers), explicitly setting `totalInflationMinted` to `0` has no functional benefit and only results in additional gas consumption.

Assets:

- `src/RainToken.sol` [<https://github.com/rain1-labs/rain-token>]

Status:

Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

Remediation:

It is recommended to remove the redundant assignment `totalInflationMinted = 0;` from the `initialize()` function to avoid the unnecessary gas usage.

Resolution:

Fixed in commit `7b397e8`. The redundant initialization has been removed from the `initialize()` function.

[F-2025-12241](#) - Misaligned USD Cap Calculation Due to USDT Peg Assumption - Info

Description:

The `Rain` smart contract attempts to enforce a daily minting cap of \$50,000 worth of Rain tokens, as stated in the inline documentation:

```
/// @notice Mints new tokens to the treasury based on a fixed USD value ($50,000)
using the current oracle price.
or
// Calculate tokens needed for $50,000
```

However, the implementation uses USDT tokens as the value representation, converting the \$50,000 cap to `50,000 * 10**6` (USDT with 6 decimals). This logic assumes a strict **1 USDT = 1 USD peg**, which is not guaranteed. In reality, USDT can trade slightly above or below \$1.00 (e.g., \$0.998 or \$1.002), creating a deviation between the intended USD cap and the actual value used for minting calculations.

This misalignment occurs in the function `dailyMinting()` responsible for minting based on the daily cap:

```
uint256 amount = (DAILY_INFLATION_USD * WEI) / rainPriceInUSD; // Assumes 1 U
SDT = 1 USD in the DAILY_INFLATION_USD
```

If USDT depegs or fluctuates slightly, the actual minting amount could exceed or fall below the intended \$50,000 USD cap, violating the expected monetary policy.

Assets:

- `src/RainToken.sol` [<https://github.com/rain1-labs/rain-token>]

Status:

Fixed

Classification

Impact Rate: 1/5

Likelihood Rate: 1/5

Exploitability: Dependent

Complexity: Simple

Severity: Info

Recommendations

- Remediation:** It is recommended to mention **USDT** directly in the comments instead of the **\$** sign since the 1 USDT is not always equal to 1 USD.
- Resolution:** **Fixed** in commit `7b397e8`. The USDT was directly mentioned in the comments as initial value.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

As part of Hacken's ongoing quality assurance process, we may conduct re-audits of select projects. These re-audits are performed independently from the original audit and are intended solely for internal quality control and improvement. Updated reports resulting from such re-audits will be shared privately with the respective clients and may be published on the Hacken website only with their explicit consent.

The sole authoritative source for finalized and most up-to-date versions of all reports remains the Audits section at <https://hacken.io/audits/>.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

[hknio/severity-formula](https://github.com/hacken/severity-formula)

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project's performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/rain1-labs/rain-token
Commit	f439c92ddc28aaa54360ba18e212ca3147a79625
Remediation Commit	7b397e81af0828a1f2c63ea7a0e1f35352e60de3
Whitepaper	https://whitepaper.rain.one/
Requirements	README.md
Technical Requirements	README.md, NatSpec

Asset	Type
src/RainToken.sol [https://github.com/rain1-labs/rain-token]	Smart Contract

Appendix 3. Additional Valuables

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.