BLOCKSEC

Security Audit

Report for Wrapped Rain Vault

Date: December 11, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Rain Coin
Target	Wrapped Rain Vault

Version History

Version	Date	Description
1.0	December 11, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-12-12 05:57:42+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of Wrapped Rain Vault of Rain Coin.

Wrapped Rain Vault is a cross-chain enabled wrapper for the Rain Coin, designed to integrate seamlessly with Chainlink CCIP. Because Rain Coin uses a reflection-tax mechanism and is renounced, direct bridging is impractical. wRAIN solves this by locking Rain Coin inside a custom Vault and issuing standard ERC-20 wRAIN as a claim on the underlying. All pooled Rain Coin automatically accumulates reflections; these rewards are distributed proportionally whenever users redeem wRAIN back to Rain Coin. Deposits mint wRAIN from post-tax received tokens, based on current NAV, and redemptions follow LP-style proportional share accounting. wRAIN enables secure, flexible, CCIP-compatible cross-chain use of Rain Coin.

Note this audit only focuses on the smart contracts in the following directories/files:

- contracts/WrappedRainVault.sol

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Address
Wrapped Rain Vault	`Version 1`	https://polygonscan.com/address/0x9fcc429f37b175224ada40ede54977d828b1def0
	`Version 2`	https://github.com/werewolfe/wrain-vault/commit/9760db4bd5eaa32dd792e1cad616e81008e7daef

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on,

[1] https://polygonscan.com/address/0x9fcc429f37b175224ada40ede54977d828b1def0

the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency

- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we found **three** potential security issues. Besides, we have **two** recommendations and **three** notes.

- High Risk: 1
- Low Risk: 2
- Recommendation: 2
- Note: 3

ID	Severity	Description	Category	Status
1	High	Share value dilution due to incorrect ratio in function `deposit()`	Security Issue	Fixed
2	Low	Inconsistent logic between function `previewWithdraw()` and `withdraw()`	Security Issue	Fixed
3	Low	Inconsistency in asset conversion logic at empty vault state	Security Issue	Fixed
4	-	Use consistent token symbol naming	Recommendation	Fixed
5	-	Non zero address checks	Recommendation	Fixed
6	-	User receives fewer tokens than expected via the function `withdraw()`	Note	-
7	-	Potential centralization risks	Note	-
8	-	Wrapped Rain Vault should not support flash-mint functionality	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Share value dilution due to incorrect ratio in function `deposit()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The current design of the function `deposit()` mints shares strictly at a 1:1 ratio based on the after-tax amount received as the deposited `Rain Coin` uses a reflection-tax mechanism. However, this approach causes dilution of the share value held by earlier depositors.

For example, when the contract starts with an empty vault and the reflection tax is 10%, User A deposits 10 assets and receives 9 shares as the actual amount received by the contract after tax is 9 assets. At this point, the `totalSupply` is 9 shares and `totalAssets` is 9 assets. After some time, reflections increase the contract's `Rain Coin` balance to 9.2 assets and User A's 9 shares correspond to 9.2 assets.

If User B later deposits 10 assets, the contract again receives 9 assets after tax and mints 9 new shares. The `totalSupply` becomes 18 shares and `totalAssets` becomes 18.2 assets. As

a result, the 9 shares held by User A now correspond to only 9.1 assets, meaning their claim on the underlying has been diluted.

```
157   function deposit(uint256 assets, address receiver)
158       public
159       override
160       nonReentrant
161       returns (uint256 shares)
162   {
163       require(assets > 0, "wRAIN: zero assets");
164       require(receiver != address(0), "wRAIN: receiver zero");
165
166       IERC20 _asset = IERC20(asset());
167
168       uint256 balanceBefore = totalAssets();
169       _asset.safeTransferFrom(msg.sender, address(this), assets);
170       uint256 balanceAfter = totalAssets();
171
172       uint256 received = balanceAfter - balanceBefore;
173       require(received > 0, "wRAIN: no tokens received");
174
175       shares = received;
176       _mint(receiver, shares);
177
178       emit Deposit(msg.sender, receiver, received, shares);
179   }
```

Listing 2.1: contracts/WrappedRain.sol

```
36   function convertToShares(uint256 assets)
37       public
38       pure
39       override
40       returns (uint256)
41   {
42       return assets;
43   }
```

Listing 2.2: contracts/WrappedRain.sol

Impact A later depositor can dilute earlier shareholders' share value.

Suggestion Revise the logic accordingly.

2.1.2 Inconsistent logic between function `previewWithdraw()` and `withdraw()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `previewWithdraw()` is designed to estimate the amount of shares that need to be burned for a specified amount of assets while function `withdraw()` aims to

burn the amount of shares for a specified amount of assets. However, the two functions have inconsistent logic when either `totalSupply` equals 0 or `totalAssets` equals 0.

Specifically, function `previewWithdraw()`returns a 1:1 amount of shares equal to the requested assets. Meanwhile, function `withdraw()` will revert as the vault does not have sufficient underlying assets. This inconsistency can result in incorrect withdrawal expectations and break the valuation logic relied upon by front-end interfaces and integration tools.

```
94   function previewWithdraw(uint256 assets)
95       public
96       view
97       override
98       returns (uint256)
99   {
100      uint256 supply = totalSupply();
101      uint256 total = totalAssets();
102
103      if (supply == 0 || total == 0) {
104          return assets;
105      }
106
107      uint256 numerator = assets * supply;
108      return (numerator + total - 1) / total;
109  }
```

Listing 2.3: contracts/WrappedRain.sol

Impact The inconsistency can disrupts the valuation logic of front-end applications and integrators.

Suggestion Revise the logic accordingly.

2.1.3 Inconsistency in asset conversion logic at empty vault state

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description According to the standard `ERC4626`, when `totalAssets` is 0, the function `convertToAssets()`should return `shares.mulDiv(1, totalSupply() + 10 ** _decimalsOffset(), rounding.DOWN)`, which is 0. However, in the `WrappedRain` contract, it returns shares. This inconsistency may cause errors in external calls.

```
45   /// @notice Convert shares -> assets (pro-rata claim).
46   function convertToAssets(uint256 shares)
47       public
48       view
49       override
50       returns (uint256)
51   {
52       uint256 supply = totalSupply();
53       uint256 assets = totalAssets();
54
```

```
55        if (supply == 0 || assets == 0) {
56            return shares;
57        }
58
59        return (shares * assets) / supply;
60    }
```

Listing 2.4: contracts/WrappedRain.sol

```
125   /// @inheritdoc IERC4626
126   function convertToAssets(uint256 shares) public view virtual returns (uint256) {
127       return _convertToAssets(shares, Math.Rounding.Floor);
```

Listing 2.5: @openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol

```
229   /**
230    * @dev Internal conversion function (from shares to assets) with support for rounding
           direction.
231    */
232   function _convertToAssets(uint256 shares, Math.Rounding rounding) internal view virtual
          returns (uint256) {
233       return shares.mulDiv(totalAssets() + 1, totalSupply() + 10 ** _decimalsOffset(), rounding);
234   }
```

Listing 2.6: @openzeppelin/contracts/token/ERC20/extensions/ERC4626.sol

Impact This inconsistency may cause errors in external calls.

Suggestion Revise the logic accordingly.

2.2 Recommendation

2.2.1 Use consistent token symbol naming

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `constructor()`, the symbol of the ERC20 token is configured as `"WRAIN"`. However, other parts of the contract `WrappedRain` (e.g., error messages and comments), reference the symbol as `"wRAIN"`. It is recommended to standardize the symbol to avoid confusing users.

```
23    constructor(IERC20 _underlying)
24        ERC20("Wrapped Rain Coin", "WRAIN")
25        ERC4626(_underlying)
26        Ownable(msg.sender)
27    {}
```

Listing 2.7: contracts/WrappedRain.sol

Suggestion It is recommended to use consistent naming for the symbol to avoid confusing users.

2.2.2 Non zero address checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In function `constructor()`, the address variables `_underlying` is not checked to ensure they are not zero. It is recommended to add such checks to prevent potential misoperations.

```
22   /// @param _underlying The reflective RAIN token.
23   constructor(IERC20 _underlying)
24       ERC20("Wrapped Rain Coin", "WRAIN")
25       ERC4626(_underlying)
26       Ownable(msg.sender)
27   {}
```

Listing 2.8: contracts/WrappedRain.sol

Suggestion Add non-zero address checks accordingly.

2.3 Note

2.3.1 User receives fewer tokens than expected via the function `withdraw()`

Introduced by `Version 1`

Description When a user invokes the function `withdraw()` with an intended amount of underlying token assets, they will actually receive fewer tokens than expected due to the tax mechanism of the underlying `Rain Coin`.

2.3.2 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., `initialOwner`) can conduct sensitive operations, which introduces potential centralization risks. For example, `msg.sender` acts as the `initialOwner` and it can transfer the owner's rights to a malicious address based on the protocol. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.3 Wrapped Rain Vault should not support flash-mint functionality

Introduced by `Version 1`

Description `Wrapped Rain Vault` should not provide `flashMint` functionality, because such features would allow an attacker to artificially borrow a large amount of shares and force the vault into an empty state. For example, if the vault initially has a shares-to-assets ratio of 1:2, a `flashMint` mechanism would allow an attacker to mint a large number of temporary shares and reduce totalSupply to zero under the protocol's reset conditions. Once the vault is reset, the attacker can mint new shares at a 1:1 rate, repay the borrowed assets immediately, and capture a profit from the manipulated shares-to-assets ratio.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS