Security Audit Report for PRT Contracts

Date: Oct 20, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.10.26 11:46:04 +08'00'

Contents

Report Manifest

Item	Description
Client	Cruise
Target	PRT Contracts

Version History

Version	Date	Description
1.0	Oct 20, 2023	First Release

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 5 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 of the PRT Contracts [1] of the Cruise. The PRT Contracts are used to separate the staking yield of the `stETH` into two tokens: `yToken` and `hodlToken`. Users transfer `Ether` to the PRT Contracts, and get corresponding amount of `yToken` and `hodlToken`. Initially, only the `yToken` accumulates the yield of the staked `stETH`. If the price returned by oracle hits a "strike price" (called "trigger" in the context of the PRT Contracts), the yield on the `yToken` stops, and `hodlToken` starts to accumulate the yield. The `hodlToken` can be used to redeem the original deposits for the users.

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.

Project	Version	Commit Hash
PRT Contracts	`Version 1`	`c6b8c92fa39e3de55823e11e413e7ce0ed5ca525`
	`Version 2`	`27aebf237d7b49ef040652f18f4967132bdff672`
	`Version 3`	`9184e726b3f7137b808e855f532fd5891ba3183a`

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, 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] https://github.com/cruise-fi/prt-contracts

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 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 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 four 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.
- **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 find **two** potential issues. Besides, we also have **one** recommendation and **one** note.

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

ID	Severity	Description	Category	Status
1	Low	Unlimited deposit after the trigger	DeFi Security	Fixed
2	Low	Incorrect check of the redeem process	DeFi Security	Fixed
3	-	Remove unused fields	Recommendation	Fixed
4	-	Potential incorrect initialization parameters	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Unlimited deposit after the trigger

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 1`

Description As introduced in Section 1.1, the `yToken` stops accumulating yields after the trigger (previously accumulated can be claimed). Therefore, minting should not be allowed after the trigger.

```
60    function mint() external payable {
61        // TODO: rework this so that we don't need the '- 1' part
62        uint256 before = stEth.balanceOf(address(this));
63        stEth.submit{value: msg.value}(address(0));
64        uint256 delta = stEth.balanceOf(address(this)) - before;
65        deposits += delta;
66
67        // subtract 1 to account for stETH behavior
68        hodlToken.mint(msg.sender, delta);
69        // mint yToken second for proper accounting
70        yToken.mint(msg.sender, delta);
71    }
```

Listing 2.1: Vault.sol

Impact The `yToken` minted after the trigger is of no value.

Suggestion Prohibit the minting after the trigger.

2.1.2 Incorrect check of the redeem process

Severity Low

Status Fixed in `Version 2`

Introduced by Version 1

Description Like the issue described in Section 2.1.2, the yToken is not accumulating yields after the trigger in the vault. Therefore, yToken should not be used for the redeem process after the trigger. Users holding only the hodlToken should be able to redeem after the trigger in the vault.

```
73  function redeem(uint256 amount) external {
74    require(IERC20(address(yToken)).balanceOf(msg.sender) >= amount);
75    require(IERC20(address(hodlToken)).balanceOf(msg.sender) >= amount);
```

Listing 2.2: Vault.sol

Impact After the trigger, the yToken is of no value, so the users with only the hodlToken should be able to redeem.

Suggestion Remove the amount check of the yToken if the vault is triggered.

2.2 Additional Recommendation

2.2.1 Remove unused fields

Status Fixed in Version 3

Introduced by Version 1

Description The following fields and variables are not used and should be removed for gas optimization.

- The claimed field in the UserInfo struct of the YToken contract.
- The deployedRoundId field of the Vault contract.

Suggestion Remove the unused fields.

2.3 Note

2.3.1 Potential incorrect initialization parameters

Description In the Vault contract, there are several parameters that can be set upon vault creation. The correctness of these parameters are up to the maintainers of the project.

```
32    constructor(string memory name_,
33          string memory symbol_,
34          uint256 strike_,
35          address stEth_,
36          address oracle_) {
37
38      // Strike price with 8 decimals
39      strike = strike_;
40
41      yToken = new YToken(address(this),
42                    string.concat("Yield ", name_),
43                    string.concat("y", symbol_));
44
45      hodlToken = new HodlToken(address(this),
46                        string.concat("PRT ", name_),
```

```
47                          string.concat("prt", symbol_));
48
49          stEth = IStEth(stEth_);
50          oracle = IOracle(oracle_);
51
52          deployedAt = block.timestamp;
53          deployedRoundId = oracle.roundId();
54      }
```

Listing 2.3: Vault.sol