BLOCKSEC

Security Audit
Report for Penpie Contracts

Date: January 27, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Magpiexyz
Target	Penpie Contracts

Version History

Version	Date	Description
1.0	January 27, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-02-27 02:42:47+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

This audit focuses on the Penpie Contracts contract [1] for Magpiexyz. Penpie is a next-generation DeFi platform designed to provide Pendle Finance users with yield and veTokenomics boosting services. Integrated with Pendle Finance, Penpie focuses on locking PENDLE tokens to obtain governance rights and enhanced yield benefits within Pendle Finance.

Specifically, for the version 1 and 2, only the following contracts in the repository are included in the scope of this audit. Other files are not within the scope of this audit.

- contracts/rewards/PRTStaking.sol

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
Penpie Contracts	`Version 1`	`c4846a4e9118f01637e3148c25c3b1a3a941dcff`
	`Version 2`	`bd5560892f837fc0f7bb28a7101f736a33bc2552`

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.

[1] https://github.com/magpiexyz/penpie-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 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.

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

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

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **two** potential security issues. Besides, we have **one** recommendations and **one** note.

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

ID	Severity	Description	Category	Status
1	Low	Potential front running of `queueUSDT`	DeFi Security	Confirmed
2	Low	Unfairness in the USDT claim mechanism	DeFi Security	Confirmed
3	-	Remove the redundant code	Recommendation	Fixed
4	-	Potential centralization risk	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Potential front running of function `queueUSDT()`

Severity Low

Status Confirmed

Introduced by `Version 1`

Description Currently, the contract `PRTStaking` does not take the staking duration time into consideration. Thus, a malicious user could front-run the function `queueUSDT()` to get the USDT back immediately by staking the PRT token. This could result in unfairness to other users.

```
184   function queueUSDT(uint256 _amount) external whenNotPaused nonReentrant {
185       if (_amount == 0) revert NotAllowZeroAmount();
186
187       IERC20(USDT).safeTransferFrom(msg.sender, address(this), _amount);
188
189       if (totalPRTStaked == 0) {
190           queuedUSDT += _amount;
191       } else {
192           if (queuedUSDT > 0) {
193               _amount += queuedUSDT;
194               queuedUSDT = 0;
195           }
196
197           uint256 prtLeft = PRTBalance();
198           if (prtLeft >= _amount) {
199               totalPRTBurned += _amount;
200               usdtPerPRTStored += (_amount * 10 ** PRTDecimal) / totalPRTStaked;
201               IERC20Burnable(PRT).burn(address(this), _amount);
202           } else {
203               totalPRTBurned += prtLeft;
204               queuedUSDT += _amount - prtLeft;
```

```
205             usdtPerPRTStored += (prtLeft * 10 ** PRTDecimal) / totalPRTStaked;
206             IERC20Burnable(PRT).burn(address(this), prtLeft);
207         }
208
209         emit USDTQueued(msg.sender, _amount);
210     }
211 }
```

Listing 2.1: contracts/rewards/PRTStaking.sol

Impact A malicious user could front run the function `queueUSDT()` to get the USDT back immediately.

Suggestion Revise the logic accordingly.

Feedback from the Project This is by design. Admin will always queue USDT after completion of staking period and USDT queue number will be equals to all users staked PRT. By doing so, user can only able to claim USDT equals to or less than his/her staked numbers of PRT.

2.1.2 Unfairness in the USDT claim mechanism

Severity Low

Status Confirmed

Introduced by `Version 1`

Description In the `PRTStaking` contract, a user's claimable amount of USDT is limited to the amount of PRT they have staked (i.e., `PRTStaked`) if their `entitledUSDT` (the total USDT amount they are entitled to claim) exceeds their `PRTStaked`. However, a malicious user can exploit this by staking an additional amount equal to `(entitledUSDT - user.PRTStaked)` when their entitledUSDT is greater than their `PRTStaked`. After staking the additional amount, if they immediately withdraw, they can claim more USDT equivalent to this additional stake. This exploit allows the malicious user to bypass the intended limit and potentially claim more USDT than they should, leading to unfairness among other users in the system.

```
 94   function claimableUSDT(
 95       address _account
 96   ) public view returns (uint256 claimable, uint256 entitledUSDT, bool debtCleaned) {
 97       entitledUSDT = _entitledUSDT(_account);
 98
 99       UserInfo storage user = userInfos[_account];
100
101       // a user can not claim more than what his PRT staked
102
103       if (entitledUSDT >= user.PRTStaked) {
104           claimable = user.PRTStaked;
105           debtCleaned = true;
106       } else {
107           claimable = entitledUSDT;
108           debtCleaned = false;
109       }
110   }
```

Listing 2.2: contracts/rewards/PRTStaking.sol

Impact The malicious user could immediately get the corresponding USDT of his second stake with the amount of `(entitledUSDT-user.PRTStaked)`.

Suggestion Revise the logic accordingly.

Feedback from the Project This is by design. User will get more USDT after stakes PRT equals to `(entitledUSDT-user.PRTStaked)` but can not further withdraw PRT.

2.2 Additional Recommendation

2.2.1 Remove the redundant code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `PRTStaking`, the event `DebtCleaned` and the error `QueueAmount-IsLargerThanStakedPRT` are unused. It is recommended to remove the code for gas optimization.

```
57    event DebtCleaned(address indexed _queuer, uint256 _amount, uint256 _refund);
58    event USDTReQueued(uint256 _amount);
59
60    /* ============ Errors ============ */
61
62    error NotAllowZeroAddress();
63    error PRTBalanceNotEnough();
64    error InconsistentDecimals();
65    error MustForCleanDebt();
66    error NotAllowZeroAmount();
67    error QueueAmountIsLargerThanStakedPRT();
```

Listing 2.3: contracts/rewards/PRTStaking.sol

Suggestion Remove the redundant code.

2.3 Note

2.3.1 Potential centralization risk

Introduced by `Version 1`

Description The contract `PRTStaking` is pausable and upgradeable which can be operated by the owner. If the owner's private key is lost or compromised, it could lead to losses for the protocol and users.

Feedback from the Project Our admin is multisig.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS