BLOCKSEC

Security Audit
Report for Penpie Contracts

Date: September 5, 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	September 5, 2025	First release

Signature

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

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-09-05 06:52:10+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 Penpie Contracts of Magpiexyz.

Penpie is a DeFi platform that enhances Pendle Finance yields through veTokenomics optimization. Users convert PENDLE to mPENDLE at 1:1 ratio for maximized rewards while maintaining flexibility. Penpie locks the converted PENDLE as vePENDLE, accumulating governance power to boost user yields. The platform enables cost-effective voting rights through PNP tokens and offers sustainable, enhanced APR for PENDLE holders via its innovative tokenomics model.

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

- contracts/pendle/PenpieAutoMarket.sol
- contracts/pendle/PenpieAutoMarketManager.sol
- contracts/pendle/SmartPendleConvert.sol
- contracts/pendle/zapInAndOutHelper.sol
- contracts/BuyBackBurnProvider.sol
- contracts/pendle/PendleStakingBaseUpg.sol
- contracts/pendle/PendleStakingSideChain.sol
- contracts/pendle/PendleMarketDepositHelper.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	Commit Hash
Penpie Contracts	`Version 1`	`a5357d1b2c7999eb7c002c8e3eade3128a9282f1`
	`Version 2`	`b205936da8e19d8028e710747714ae3f7c446012`

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

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.

1.3 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.

[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.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

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

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

ID	Severity	Description	Category	Status
1	High	Potential DoS due to lack of `receive()` or `fallback()` function	Security Issue	Fixed
2	Medium	Potential inflation attack in deposit() functions	Security Issue	Confirmed
3	Medium	Potential front running of the function `batchCollectAndCompound()`	Security Issue	Confirmed
4	-	Revise typos	Recommendation	Fixed
5	-	Lack of check on function `addAutoFee()`	Recommendation	Fixed
6	-	Improper `receiptToken` balance calculation in function `_depositLPToAutoMarket()`	Recommendation	Fixed
7	-	Potential centralization risks	Note	-
8	-	Proxy deployment and implementation binding should be atomic	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Potential DoS due to lack of `receive()` or `fallback()` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `harvestReward()` allows users to harvest market rewards and then claim the rewards to the `autoMarket`. The contract refunds the `harvestCallerPendleFee` to the caller as `Ether`. However, when the contract `PenpieAutoMarketManager` interacts through functions like `depositWithPenpieLP()`, `withdrawAsPenpieLP()` and so on, it will invoke the function `harvestReward()` within the execution. The manager contract lacks payable `receive()` or `fallback()` functions to handle `Ether` transfers. Therefore, the transaction will revert and result in the DoS of the manager contract.

```
98        if (block.timestamp - lastHarvestTime > IPenpieAutoMarketManager(manager).harvestCoolDown()
              ||force) {
99            IPendleStaking(pendleStaking).harvestMarketReward(pendleMarket, msg.sender, 0);
100
```

```
101         IMasterPenpie(masterPenpie).multiclaim(stakingTokens);
```

Listing 2.1: contracts/pendle/PenpieAutoMarket.sol

```
231 function depositWithPenpieLP(address autoMarketAddress, uint256 lpAmount) external
        nonReentrant onlyValidAutoMarket(autoMarketAddress) {
232     if (lpAmount == 0) revert InvalidAmount();
233
234     (address receiptToken, , ) = getAutoMarketInfo(autoMarketAddress);
235
236     // Transfer receipt tokens from user to this contract
237     IERC20(receiptToken).safeTransferFrom(msg.sender, address(this), lpAmount);
238
239     // Harvest if enough time has passed since last harvest to maximize user value
240     // This ensures the user gets the most accurate share calculation
241     PenpieAutoMarket(autoMarketAddress).harvestReward(false);
```

Listing 2.2: contracts/pendle/PenpieAutoMarketManager.sol

```
303 function withdrawAsPenpieLP(address autoMarketAddress, uint256 aMaketamount) external
        nonReentrant onlyValidAutoMarket(autoMarketAddress) {
304     if (aMaketamount == 0) revert InvalidAmount();
305
306     (address receiptToken, , ) = getAutoMarketInfo(autoMarketAddress);
307
308     // Harvest if enough time has passed since last harvest to maximize user value
309     // This ensures the user gets the most accurate redemption value
310     PenpieAutoMarket(autoMarketAddress).harvestReward(false);
```

Listing 2.3: contracts/pendle/PenpieAutoMarketManager.sol

```
355     uint256 receiptBalance = IERC20(receiptToken).balanceOf(address(this));
356     if (receiptBalance > 0) {
357         // Harvest if enough time has passed since last harvest to maximize user value
358         // This ensures the user gets the most accurate share calculation
359         PenpieAutoMarket(autoMarketAddress).harvestReward(false);
```

Listing 2.4: contracts/pendle/PenpieAutoMarketManager.sol

```
374 function redeemAndSwapFromMarket(
375     address autoMarketAddress,
376     uint256 autoMarketShares,
377     ZapOutData calldata zapOutData
378 ) external nonReentrant onlyValidAutoMarket(autoMarketAddress) {
379     if (autoMarketShares == 0) revert InvalidAmount();
380     if (zapOutData.tokenZapOutData.tokenForZapOut == address(0)) revert ZeroAddress();
381
382     (address receiptToken, , ) = getAutoMarketInfo(autoMarketAddress);
383
384     // Harvest if enough time has passed since last harvest to maximize user value
385     // This ensures the user gets the most accurate redemption value
386     PenpieAutoMarket(autoMarketAddress).harvestReward(false);
```

Listing 2.5: contracts/pendle/PenpieAutoMarketManager.sol

```
763         if (harvestCallerTotalPendleReward > 0) {
764             IERC20(PENDLE).approve(ETHZapper, harvestCallerTotalPendleReward);
765 
766             IETHZapper(ETHZapper).swapExactTokensToETH(
767                 PENDLE,
768                 harvestCallerTotalPendleReward,
769                 _minEthToRecieve,
770                 _caller
771             );
772         }
```

Listing 2.6: contracts/pendle/PendleStakingBaseUpg.sol

Impact The transaction will revert and result in the DoS of the manager contract.

Suggestion Revise the refund caller fee logic.

2.1.2 Potential inflation attack in deposit() functions

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the contract `PenpieAutoMarketManager`, users can participate in the `autoMarket` with `Pendle LP`s, `Wrapped Pendle LP`s, or `Penpie LP`s. The contracts will help users convert their tokens to the `receiptToken` and deposit into the `autoMarket`. However, the design is problematic due to share calculation rounding down in the inherited `ERC4626` implementation. Specifically, a malicious user could front-run initial deposits and manipulate share prices through donation attacks. As a result, this design enables share price manipulation that may cause legitimate users to receive zero shares.

```
83          __ERC4626_init(IERC20MetadataUpgradeable(_penpieReceiptToken));
84          __ERC20_init(
85              string.concat("Auto", IERC20MetadataUpgradeable(_penpieReceiptToken).name()),
86              string.concat("Auto ", IERC20MetadataUpgradeable(_penpieReceiptToken).symbol())
87          );
88          __ReentrancyGuard_init();
```

Listing 2.7: contracts/pendle/PenpieAutoMarket.sol

This issue exists in multiple functions: `depositWithPendleLP()`, `depositWithWrappedPendleLP()`, and `depositWithPenpieLP()`.

```
187         if (autoMarketShares > 0) {
188             emit AutoMarketDeposit(msg.sender, pendleMarket, lpAmount, autoMarketShares, false);
189         }
```

Listing 2.8: contracts/pendle/PenpieAutoMarketManager.sol

```
223         if (autoMarketShares > 0) {
224             emit AutoMarketWrappedLPDeposit(msg.sender, pendleMarket, wrappedLPToken,
                    wrappedLPAmount, autoMarketShares);
```

```
225       }
```

Listing 2.9: contracts/pendle/PenpieAutoMarketManager.sol

```
244       IERC20(receiptToken).safeApprove(autoMarketAddress, lpAmount);
245       uint256 autoMarketShares = PenpieAutoMarket(autoMarketAddress).deposit(lpAmount, msg.sender
              );
246       IERC20(receiptToken).safeApprove(autoMarketAddress, 0);
```

Listing 2.10: contracts/pendle/PenpieAutoMarketManager.sol

Impact The legitimate user may lose their deposited tokens due to the precision loss.

Suggestion Revise the logic accordingly.

Feedback from the project The project will make an initial deposit after the deployment.

2.1.3 Potential front running of the function `batchCollectAndCompound()`

Severity Medium

Status Confirmed

Introduced by `Version 1`

Description In the function `batchCollectAndCompound()` function, the contract will collect the reward tokens and convert them to receipt tokens. After the conversion, the contract will transfer the receipt tokens to the `autoMarket`, which increases the share price immediately. This function can only be invoked by allowed bots. The malicious users can monitor the invocation and front-run it to get shares at a lower price, which leads to an unfair price mechanism for all participants.

```
550   function batchCollectAndCompound(
551       address[] calldata autoMarkets,
552       address[] calldata rewardTokens,
553       uint256[] calldata amounts,
554       ZapInData[] calldata zapInData
555   ) external nonReentrant {
556       if (!allowedBot[msg.sender]) revert NotAllowed();
557       if (autoMarkets.length == 0) revert InvalidLength();
558       if (autoMarkets.length != rewardTokens.length || autoMarkets.length != amounts.length) {
559           revert LengthMismatch();
560       }
```

Listing 2.11: contracts/pendle/PenpieAutoMarketManager.sol

Impact The user can make profit by front running the function, which leads to an unfair price mechanism for all participants.

Suggestion Revise the logic accordingly.

Feedback from the project The bot deployed by the team will periodically invoke this function every 4 hours to reduce the impact of this issue.

2.2 Recommendation

2.2.1 Revise typos

Status Fixed in Version 2

Introduced by Version 1

Description Several typos exist in event messages and variables across the codebase. These issues reduce code readability and may lead to confusion or errors in usage and maintenance.

For example, the contract will emit an event when the pendleRouter is updated. The event name should be pendleRouterUpdated rather than pendleRoutreUpdated.

Besides, the constant variable ADDREESS_ZERO should be ADDRESS_ZERO.

```
28    address private constant ADDREESS_ZERO = address(0);
```

Listing 2.12: contracts/pendle/zapInAndOutHelper.sol

```
63    event pendleRoutreUpdated(address _pendleRouterV4);
```

Listing 2.13: contracts/pendle/zapInAndOutHelper.sol

Suggestion Revise the typos in the contract.

2.2.2 Lack of check on function addAutoFee()

Status Fixed in Version 2

Introduced by Version 1

Description In the PenpieAutoMarketManager.sol contract, the function addAutoFee() allows the owner to add new fee entries to the feeInfos array and accumulates the fee values into totalAutoFee.

However, the function addAutoFee() lacks a validation check to ensure that totalAutoFee does not exceed the DENOMINATOR after being increased. This means that when multiple fees are added consecutively, the accumulated total fee may surpass the predefined limit.

```
619   function addAutoFee(
620       uint256 _value,
621       address _to
622   ) external onlyOwner {
623       if (_value >= DENOMINATOR) revert InvalidFee();
624
625       feeInfos.push(
626           Fees({
627               value: _value,
628               to: _to
629           })
630       );
631       totalAutoFee += _value;
632   }
```

Listing 2.14: contracts/pendle/PenpieAutoMarketManager.sol

Suggestion Revise the logic accordingly.

2.2.3 Improper `receiptToken` balance calculation in function `_depositLPToAutoMarket()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `_depositLPToAutoMarket()`, the protocol first deposits the user's `Pendle LP`s into the `PendleStaking` contract and receives a corresponding amount of `receiptToken`. It then deposits these `receiptToken` into the `autoMarketAddress` contract to obtain shares.

However, the protocol currently determines the number of `receiptToken` by directly querying the contract's `receiptToken` balance. This approach may yield inaccurate results. The correct method should be to measure the delta in the contract's `receiptToken` balance before and after invoking `depositMarket()`, as only this balance change reliably reflects the actual number of `receiptToken` obtained from the deposit.

The same issue also occurs in the functions `_withdrawFromAutoMarketToPendleLP()`, `redeemAndSwapFromMarket()`, `zapAndStakeToAutoMarket()`, and `withdrawAsPenpieLP()`.

```
455       IERC20(pendleMarket).safeApprove(address(pendleStaking), lpAmount);
456       depositHelper.depositMarket(pendleMarket, lpAmount);
457       IERC20(pendleMarket).safeApprove(address(pendleStaking), 0);
458
459       // Get the receipt tokens from the deposit
460       uint256 receiptBalance = IERC20(receiptToken).balanceOf(address(this));
461       if (receiptBalance > 0) {
462           // Harvest if enough time has passed since last harvest to maximize user value
463           // This ensures the user gets the most accurate share calculation
464           PenpieAutoMarket(autoMarketAddress).harvestReward(false);
```

Listing 2.15: contracts/pendle/PenpieAutoMarketManager.sol

Suggestion Revise the logic accordingly.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., `Manager`) can conduct sensitive operations, which introduces potential centralization risks. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

2.3.2 Proxy deployment and implementation binding should be atomic

Introduced by `Version 1`

Description To prevent potential front-running attacks, it is recommended that proxy deployment and implementation binding be executed atomically within a single transaction. If these operations are performed separately, malicious actors could exploit the time window

between deployment and binding to front-run the implementation binding transaction. An attacker could potentially bind their own malicious implementation to the newly deployed proxy contract, gaining unauthorized control over the protocol's upgrade mechanism and user funds. This race condition poses significant security risks including complete protocol compromise, fund theft, and unauthorized access to privileged functions.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS