BLOCKSEC

Security Audit
Report for PSM

Date: November 22, 2024 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Lista
Target	PSM

Version History

Version	Date	Description
1.0	November 22, 2024	First release

Signature

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 code repositories of the PSM [1] of Lista.

Please note that this audit covers only the following contracts:

- contracts/psm/EarnPool.sol
- contracts/psm/LisUSDPoolSet.sol
- contracts/psm/PSM.sol
- contracts/psm/VaultManager.sol
- contracts/psm/VenusAdapter.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
PSM	`Version 1`	`34d738a9cd6ad67eaf7e43ca4b153e658d6a48d8`
	`Version 2`	`fca5f0c9acc57141ef53c793cfba66cfc4f24c6a`

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/lista-dao/lista-dao-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 and Common Weakness Enumeration. 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.

Chapter 2 Findings

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

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

ID	Severity	Description	Category	Status
1	Low	Timely update of snapshot during distributor update	DeFi Security	Confirmed
2	Medium	Potential DoS in withdrawal process due to improper checks in function `setAdapter()`	DeFi Security	Fixed
3	-	Lack of check in function `removePSM()`	Recommendation	Fixed
4	-	Redundant code	Recommendation	Confirmed
5	-	Potential centralization risk	Note	-
6	-	Supported stablecoin decimal consistency with lisUSD	Note	-
7	-	Using function `emergencyWithdraw()` to retrieve funds from inactive adapters	Note	-

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Timely update of snapshot during distributor update

Severity Low

Status Confirmed

Introduced by `Version 1`

Description When the privileged `MANAGER` role invokes the function `removeDistributor()`, but has not yet updated a pool's `distributor` by invoking the function `setDistributor()`, any deposits made by users will encounter an issue. Specifically, though the new `distributor` inherits its data from the old one, the newly deposited tokens will not immediately start accumulating rewards. Instead, users need to manually invoke the `takeSnapshot()` function to trigger this process.

```
345 function takeSnapshot(address user, address pool) public {
346   address distributor = pools[pool].distributor;
347   // ensure the distributor address is set
348   if (distributor != address(0)) {
349     IStakeLisUSDListaDistributor(distributor).takeSnapshot(user, poolEmissionWeights[pool][user
          ]);
350   }
```

```
351 }
352
353
354 function setDistributor(address pool, address _distributor) external onlyRole(MANAGER) {
355   require(_distributor != address(0), "distributor cannot be zero address");
356   require(pools[pool].distributor == address(0), "distributor already exists");
357
358
359   pools[pool].distributor = _distributor;
360
361
362   emit SetDistributor(pool, _distributor);
363 }
364
365
366 /**
367  * @dev remove distributor address
368  * @param pool pool address
369  */
370 function removeDistributor(address pool) external onlyRole(MANAGER) {
371   address distributor = pools[pool].distributor;
372   pools[pool].distributor = address(0);
373
374
375   emit RemoveDistributor(pool, distributor);
376 }
```

Listing 2.1: contracts/psm/LisUSDPoolSet.sol

Impact Users may lose rewards due to not timely invoking the `takeSnapshot()` function after the `distributor` is updated.

Suggestion Revise the logic to ensure fair rewards distribution.

Feedback from the project When we deploy a new `distributor`, we will manually update snapshots for all users.

2.1.2 Potential DoS in withdrawal process due to improper checks in function `setAdapter()`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the `VaultManager` contract, the protocol's manager can use the function `setAdapter()` to set a new `adapter` or disable an `adapter` only when its `netDepositAmount` is less than or equal to 10. However, when an issue occurs with an external DeFi protocol connected to an `adapter`, its `netDepositAmount` may still be greater than 10. This prevents the `manager` from timely disabling the corresponding `adapter` as an emergency measure.

Even worse, both the `withdraw()` and `withdrawAll()` functions iterate through all active `adapters` to perform withdrawal operations. As a result, if one `adapter` encounters a prob-

lem, the manager cannot disable the problematic `adapter`, which could also affect the normal operation of other `adapters`.

```
187 function setAdapter(uint256 index, bool active, uint256 point) external onlyRole(MANAGER) {
188   require(index < adapters.length, "index out of range");
189   if (!active) {
190     require(IAdapter(adapters[index].adapter).netDepositAmount() <= 10, "adapter has net deposit
          amount");
191   }
192   adapters[index].active = active;
193   adapters[index].point = point;
194
195
196   emit SetAdapter(adapters[index].adapter, active, point);
197 }
```

Listing 2.2: contracts/psm/VaultManager.sol

```
226 function rebalance() external onlyRole(BOT) {
227   require(adapters.length > 0, "no adapter");
228
229
230   for (uint256 i = 0; i < adapters.length; i++) {
231     if (adapters[i].active) {
232       IAdapter(adapters[i].adapter).withdrawAll();
233     }
234   }
235   uint256 amount = IERC20(token).balanceOf(address(this));
236
237
238   if (amount > 0) {
239     _distribute(amount);
240   }
241
242
243   emit ReBalance(amount);
244 }
```

Listing 2.3: contracts/psm/VaultManager.sol

```
118 function withdraw(address receiver, uint256 amount) external nonReentrant onlyPSMOrManager {
119   require(amount > 0, "withdraw amount cannot be zero");
120   uint256 remain = amount;
121   uint256 vaultBalance = IERC20(token).balanceOf(address(this));
122   if (vaultBalance >= amount) {
123     // withdraw token from vault manager
124     IERC20(token).safeTransfer(receiver, amount);
125     remain = 0;
126   } else {
127     if (vaultBalance > 0) {
128       IERC20(token).safeTransfer(receiver, vaultBalance);
129       remain -= vaultBalance;
130     }
131   }
```

```
132
133
134   if (remain > 0) {
135     require(adapters.length > 0, "no adapter");
136     // withdraw token from adapters
137     uint256 startIdx = block.number % adapters.length;
138
139
140     for (uint256 i = 0; i < adapters.length; i++) {
141       uint256 idx = (startIdx + i) % adapters.length;
142       // only active adapter can be used
143       if (adapters[idx].active) {
144         uint256 netDeposit = IAdapter(adapters[idx].adapter).netDepositAmount();
145         if (netDeposit == 0) {
146           continue;
147         }
148         if (netDeposit >= remain) {
149           IAdapter(adapters[idx].adapter).withdraw(receiver, remain);
150           remain = 0;
151           break;
152         } else {
153           remain -= netDeposit;
154           IAdapter(adapters[idx].adapter).withdraw(receiver, netDeposit);
155         }
156       }
157     }
158   }
159
160
161   require(remain == 0, "not enough available balance");
162
163
164   emit Withdraw(receiver, amount);
165 }
```

Listing 2.4: contracts/psm/VaultManager.sol

Impact Functions `withdrawAll()` and `withdraw()` may fail to execute properly.

Suggestion Revise the logic to ensure that the protocol's `manager` can timely disable an `adapter` when it becomes unavailable.

2.2 Additional Recommendation

2.2.1 Lack of check in function `removePSM()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The `removePSM()` function does not verify whether the passed `_token` address has a corresponding `PSM` before deletion. This means that even if the token has not set a `PSM`, the privileged `MANAGER` role can still successfully invoke the function, which will emit the `RemovePSM`

event, even though no PSM is actually removed. This could lead to misleading information in the logs.

```
124 function removePSM(address _token) external onlyRole(MANAGER) {
125   delete psm[_token];
126
127
128   emit RemovePSM(_token);
129 }
```

Listing 2.5: contracts/psm/EarnPool.sol

Suggestion Add a check to ensure the _token address is valid.

2.2.2 Redundant code

Status Confirmed

Introduced by Version 1

Description There are multiple instances of redundant logic in the protocol:

In the PSM contract, the function emergencyWithdraw() is designed for the protocol's admin to either perform emergency withdrawals under special circumstances or retrieve mistakenly transferred tokens. However, as an implementation contract, the contract does not implement a payable fallback() function, which means it doesn't support receiving native tokens. In this case, the logic for withdrawing native tokens is redundant. The same issue exists in the contract LisUSDPoolSet.

In the VenusAdapter contract, the EmergencyWithdraw event is redundant, as the contract does not emit the corresponding log.

In the VenusAdapter contract, the function redeem() of Venus is invoked in the private function _withdrawFromVenus() to withdraw assets deposited in Venus. However, it does not involve any token transfers from VenusAdapter to Venus. Therefore, the invocation of safeIncreaseAllowance() in this context is redundant.

```
334 function emergencyWithdraw(address _token, uint256 _amount) external onlyRole(DEFAULT_ADMIN_ROLE
        ) {
335   if (_token == address(0)) {
336     (bool success, ) = payable(msg.sender).call{ value: _amount }("");
337     require(success, "Withdraw failed");
338   } else {
339     IERC20(_token).safeTransfer(msg.sender, _amount);
340   }
341   emit EmergencyWithdraw(_token, _amount);
342 }
```

Listing 2.6: contracts/psm/PSM.sol

```
306 function emergencyWithdraw(address _token, uint256 _amount) external onlyRole(DEFAULT_ADMIN_ROLE
        ) {
307   if (_token == address(0)) {
308     (bool success, ) = payable(msg.sender).call{ value: _amount }("");
309     require(success, "Withdraw failed");
```

```
310   } else {
311     IERC20(_token).safeTransfer(msg.sender, _amount);
312   }
313   emit EmergencyWithdraw(_token, _amount);
314 }
```

Listing 2.7: contracts/psm/LisUSDPoolSet.sol

```
24 event EmergencyWithdraw(address token, uint256 amount);
```

Listing 2.8: contracts/psm/VenusAdapter.sol

```
154 function _withdrawFromVenus(uint256 vTokenAmount) private returns (uint256) {
155   uint256 before = IERC20(token).balanceOf(address(this));
156   IERC20(vToken).safeIncreaseAllowance(vToken, vTokenAmount);
157   IVBep20Delegate(vToken).redeem(vTokenAmount);
158   return IERC20(token).balanceOf(address(this)) - before;
159 }
```

Listing 2.9: contracts/psm/VenusAdapter.sol

Suggestion Remove the redundant code mentioned above.

2.3 Note

2.3.1 Potential centralization risk

Introduced by Version 1

Description In the current implementation, several privileged roles are set to govern and regulate the system-wide operation (e.g., parameter setting, pause/unpause and grant roles). Additionally, the function `emergencyWithdraw()` allows the privileged `admin` to withdraw all assets in the `LisUSDPoolSet` contract including the users's staking `lisUSD`. The `admin` also has the ability to upgrade all the implementation contracts. If the private keys of them are lost or maliciously exploited, it could potentially lead to losses for users.

Feedback from the project Admin role will be TimeLock contract

2.3.2 Supported stablecoin decimal consistency with lisUSD

Introduced by Version 1

Description In the `PSM` contract, a fee is charged with `lisUSD` during the buying and selling process. However, during the fee calculation and deduction process, the contract assumes that the amount and value of the `stablecoin` used for buying and selling `lisUSD` are on a 1:1 ratio with `lisUSD`. This requires the contract to ensure that the decimals of the `stablecoin` and `lisUSD` are the same.

2.3.3 Using function `emergencyWithdraw()` to retrieve funds from inactive adapters

Introduced by `Version 1`

Description In the `VaultManager` contract, the `rebalance()` function first withdraws all assets from each active `adapter` and then redistributes them based on the points of each `adapter`. However, if an `adapter` is in an inactive state, it may still contain dust, which need to be manually retrieved through function `emergencyWithdraw()`.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS