SMART CONTRACT AUDIT REPORT

for

Horizon Protocol

Prepared By: Xiaomi Huang

PeckShield

July 23, 2023

Document Properties

Client	Horizon Protocol
Title	Smart Contract Audit Report
Target	Horizon
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 23, 2023	Xuxian Jiang	Final Release
1.0-rc	July 18, 2023	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Horizon protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Horizon

Horizon Protocol is a DeFi platform that facilitates the on-chain creation and derivatives trading of synthetic assets that represent the real economy. Horizon Protocol seeks to provide exposure to real-world asset risk/return profiles via smart contracts on the blockchain. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Horizon

Item	Description
Target	Horizon
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	July 23, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- https://github.com/Horizon-Protocol/testnet-contracts.git (299917f)

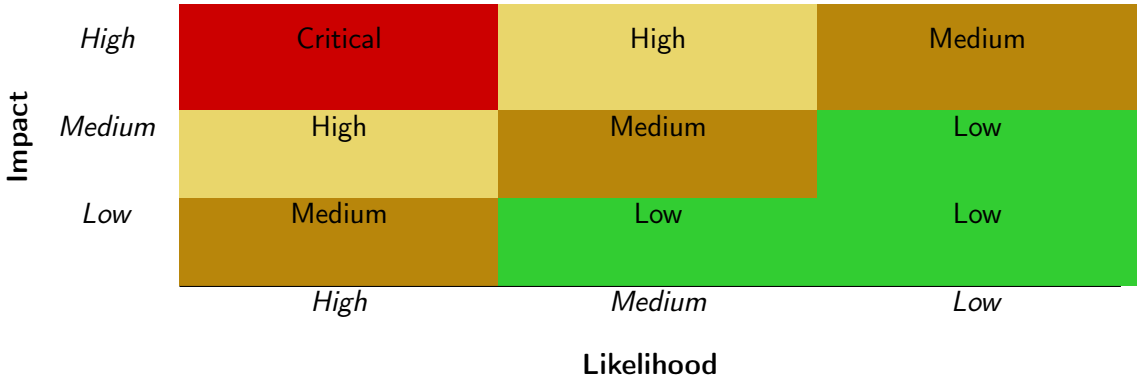
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/Horizon-Protocol/testnet-contracts.git (c345eea)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the Horizon protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■■
Low	4	■■■■
Informational	1	■
Total	7	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 4 low-severity vulnerabilities, and and 1 informational recommendation.

Table 2.1: Key Horizon Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Inconsistent Transfer Logic in Synth::transfer()/transferFrom()	Business Logic	Resolved
PVE-002	Low	Improved SignedSafeMath::mul() Logic	Coding Practices	Resolved
PVE-003	Medium	Improper Logic of Exchanger::calculateAmountAfterSettlement()	Business Logic	Resolved
PVE-004	Low	Explicit collateralKey Enforcement in CollateralShort	Business Logic	Confirmed
PVE-005	Informational	Redundant Data/Code Removal	Coding Practices	Confirmed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Low	Improved Logic of BaseRewardEscrowV2::accountMergingIsOpen()	Business Logic	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Inconsistent Transfer Logic in Synth::transfer()/transferFrom()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Synth
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Horizon protocol allows for the creation of synthetic assets, which are collateralized by stakers with HZN. Specifically, the locking of HZN in a staking contract enables the issuance of synthetic assets (Synths). This pooled collateral model allows users to perform conversions between Synths directly with the smart contract, avoiding the need for counterparties. In the process of analyzing the logic behind each synthetic asset, we notice the current ERC20-compliant token-transferring feature has an inconsistent implementation between transfer() and transferFrom(). To elaborate the inconsistency, we show their implementation below.

By design, these two routines simply transfers the intended amount of the Synths to the recipient. We notice the transfer() routine adds customized logic when the recipient is FEE_ADDRESS or address(0). However, the same customization does not exist in the transferFrom() routine. The same inconsistency is also present in other token-transferring variants, including transferAndSettle() and transferFromAndSettle().

```
63    function transfer(address to, uint value) public onlyProxyOrInternal returns (bool)
          {
64        _ensureCanTransfer(messageSender, value);
65
66        // transfers to FEE_ADDRESS will be exchanged into zUSD and recorded as fee
67        if (to == FEE_ADDRESS) {
68            return _transferToFeeAddress(to, value);
```

```
69          }
70
71          // transfers to 0x address will be burned
72          if (to == address(0)) {
73              return _internalBurn(messageSender, value);
74          }
75
76          return super._internalTransfer(messageSender, to, value);
77      }
78
79      function transferFrom(
80          address from,
81          address to,
82          uint value
83      ) public onlyProxyOrInternal returns (bool) {
84          _ensureCanTransfer(from, value);
85
86          return _internalTransferFrom(from, to, value);
87      }
```

Listing 3.1: `Synth::transfer()/transferFrom()`

Recommendation Revisit the above token-transferring logic for consistency.

Status The issue has been addressed by the following commit: `c345eea`.

3.2 Improved SignedSafeMath::mul() Logic

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SignedSafeMath`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, we notice the variant for the signed integer operation can be improved.

To elaborate, we show below the `mul()` function from the `SignedSafeMath` contract. This function is designed to return the multiplication of two signed integers, reverting on overflow. Since it operates on signed integers, it reverts the cases when the given signed integers are `-1` and `_INT256_MIN`. The reason is that the result needs to be smaller than `2**255`. However, the current implementation only

checks one occasion when `a == -1 && b == _INT256_MIN`. It might encounter an input with `b == -1 && a == _INT256_MIN` as well.

```
49      function mul(int256 a, int256 b) internal pure returns (int256) {
50          // Gas optimization: this is cheaper than requiring 'a' not being zero, but the
51          // benefit is lost if 'b' is also tested.
52          // See: https://github.com/OpenZeppelin/openzeppelin-contracts/pull/522
53          if (a == 0) {
54              return 0;
55          }
57          require(!(a == -1 && b == _INT256_MIN), "SignedSafeMath: multiplication overflow
                ");
59          int256 c = a * b;
60          require(c / a == b, "SignedSafeMath: multiplication overflow");
62          return c;
63      }
```

Listing 3.2: `SignedSafeMath::mul()`

Recommendation Revise the above routine to ensure it reverts all overflowing cases.

Status The issue has been resolved as the incorrect input will still be caught in the second requirement.

3.3 Improper Logic of Exchanger::calculateAmountAfterSettlement()

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Exchanger`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Horizon` protocol has a key `Exchanger` contract that makes use of the pooled collateral model and allows users to perform conversions between synths directly, avoiding the need for counterparties. As a result, this mechanism solves the liquidity and slippage issues experienced by DEXs. While examining one helper routine `calculateAmountAfterSettlement()`, we notice its logic can be improved.

To elaborate, we show below the related `calculateAmountAfterSettlement()` implementation. As the name indicates, this routine calculates the balance of a synth after the settlement. Note the

settlement may result in extra funds being reclaimed or refunded. And the reclaimed or refunded amount will be reflected in the token balance of the give `from` account. With that, the addition of possible `refunded` needs to be added to `amountAfterSettlement` before the resulting sum is compared with the current balance in `balanceOfSourceAfterSettlement`.

```
277     function calculateAmountAfterSettlement(
278         address from,
279         bytes32 currencyKey,
280         uint amount,
281         uint refunded
282     ) public view returns (uint amountAfterSettlement) {
283         amountAfterSettlement = amount;
285         // balance of a synth will show an amount after settlement
286         uint balanceOfSourceAfterSettlement = IERC20(address(issuer().synths(currencyKey
                ))).balanceOf(from);
288         // when there isn't enough supply (either due to reclamation settlement or
                because the number is too high)
289         if (amountAfterSettlement > balanceOfSourceAfterSettlement) {
290             // then the amount to exchange is reduced to their remaining supply
291             amountAfterSettlement = balanceOfSourceAfterSettlement;
292         }
294         if (refunded > 0) {
295             amountAfterSettlement = amountAfterSettlement.add(refunded);
296         }
297     }
```

Listing 3.3: `Exchanger::calculateAmountAfterSettlement()`

Recommendation Revisit the above logic to properly calculate the synth mount after the settlement to properly account for possible synth reclamation and refund.

Status The issue has been confirmed. Meanwhile, the team clarifies that the change is relatively inconsequential as this would prevent a function revert in a niche case where the user tries to burn his entire `zUSD` balance in addition to the exact settlement refund amount (if refunded >0) and if this burn value is less than his debt balance.

3.4 Explicit collateralKey Enforcement in CollateralShort

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CollateralShort`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

The `Horizon` protocol support shorts, which are fixed debt, over-collateralized loans. To open a short, the user must deposit `sUSD` collateral and then choose the synth they wish to short and the size of the position, subject to collateralization requirements. However, instead of issuing the shorted currency, the contract converts the value to sUSD and issues that to the user, which represents the act of selling the asset short.

The short support is mainly implemented in the `CollateralShort` inherited from the common `Collateral` contract. However, our analysis shows the use of `sUSD` as collateral is not explicitly enforced and this enforcement is strongly suggested.

```
8  contract CollateralShort is Collateral {
9      constructor(
10         address _owner,
11         ICollateralManager _manager,
12         address _resolver,
13         bytes32 _collateralKey,
14         uint _minCratio,
15         uint _minCollateral
16     ) public Collateral(_owner, _manager, _resolver, _collateralKey, _minCratio,
           _minCollateral) {}

18     function open(
19         uint collateral,
20         uint amount,
21         bytes32 currency
22     ) external returns (uint id) {
23         // Transfer from will throw if they didn't set the allowance
24         IERC20(address(_synthzUSD())).transferFrom(msg.sender, address(this), collateral
               );

26         id = _open(collateral, amount, currency, true);
27     }
28     ...
29 }
```

Listing 3.4: The `CollateralShort` Contract

Recommendation Enforce the use of `sUSD` as collateral when the `CollateralShort` contract is instantiated.

Status The issue has been confirmed.

3.5 Redundant State/Code Removal

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The `Horizon` protocol makes good use of a number of reference contracts, such as `ERC20`, `Owned`, `SafeDecimalMath`, and `Pausable`, to facilitate its code implementation and organization. For example, the `Exchanger` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `Exchanger` contract, there is a routine to calculate the dynamic fee for the given currency. It comes to our attention that it contains the early exit condition for the special case of no dynamic fee. However, the same early exit condition is checked twice, one at lines 774-776 and another at lines 792-794. With that, we can safely drop the first one as the second one is validated in a common helper routine.

```
769     function _dynamicFeeRateForCurrency(
770         bytes32 currencyKey,
771         DynamicFeeConfig memory config
772     ) internal view returns (uint dynamicFee, bool tooVolatile) {
773         // no dynamic dynamicFee for zUSD or too few rounds
774         if (currencyKey == zUSD  config.rounds <= 1) {
775             return (0, false);
776         }
777         uint roundId = exchangeRates().getCurrentRoundId(currencyKey);
778         return _dynamicFeeRateForCurrencyRound(currencyKey, roundId, config);
779     }
780
781     /// @notice Get dynamicFee for a given currency key (SIP-184)
782     /// @param currencyKey The given currency key
783     /// @param roundId The round id
784     /// @param config dynamic fee calculation configuration params
785     /// @return The dynamic fee and if it exceeds max dynamic fee set in config
786     function _dynamicFeeRateForCurrencyRound(
```

```
787         bytes32 currencyKey,
788         uint roundId,
789         DynamicFeeConfig memory config
790     ) internal view returns (uint dynamicFee, bool tooVolatile) {
791         // no dynamic dynamicFee for zUSD or too few rounds
792         if (currencyKey == zUSD  config.rounds <= 1) {
793             return (0, false);
794         }
795         uint[] memory prices;
796         (prices, ) = exchangeRates().ratesAndUpdatedTimeForCurrencyLastNRounds(
                currencyKey, config.rounds, roundId);
797         dynamicFee = _dynamicFeeCalculation(prices, config.threshold, config.weightDecay
                );
798         // cap to maxFee
799         bool overMax = dynamicFee > config.maxFee;
800         dynamicFee = overMax ? config.maxFee : dynamicFee;
801         return (dynamicFee, overMax);
802     }
```

Listing 3.5: `Exchanger::_dynamicFeeRateForCurrency()`

In addition, there is another redundancy in the following `removeFromArray` routine. Specifically, once the to-be-removed entry is located, the current approach deletes the located array element and next overwrites with the last element. Note the first deletion is redundant and can be safely skipped.

```
389     function removeFromArray(bytes32 entry, bytes32[] storage array) internal returns (
            bool) {
390         for (uint i = 0; i < array.length; i++) {
391             if (array[i] == entry) {
392                 delete array[i];
393
394                 // Copy the last key into the place of the one we just deleted
395                 // If there's only one key, this is array[0] = array[0].
396                 // If we're deleting the last one, it's also a NOOP in the same way.
397                 array[i] = array[array.length - 1];
398
399                 // Decrease the size of the array by one.
400                 array.length--;
401
402                 return true;
403             }
404         }
405         return false;
406     }
```

Listing 3.6: `ExchangeRates::removeFromArray()`

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been confirmed.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Horizon` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure various parameters, set protocol rates, and pause/unpause protocol). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
276     function setIssuanceRatio(uint ratio) external onlyOwner {
277         flexibleStorage().setIssuanceRatio(SETTING_ISSUANCE_RATIO, ratio);
278         emit IssuanceRatioUpdated(ratio);
279     }
280
281     function setTradingRewardsEnabled(bool _tradingRewardsEnabled) external onlyOwner {
282         flexibleStorage().setTradingRewardsEnabled(SETTING_TRADING_REWARDS_ENABLED,
                _tradingRewardsEnabled);
283         emit TradingRewardsEnabled(_tradingRewardsEnabled);
284     }
285
286     function setWaitingPeriodSecs(uint _waitingPeriodSecs) external onlyOwner {
287         flexibleStorage().setWaitingPeriodSecs(SETTING_WAITING_PERIOD_SECS,
                _waitingPeriodSecs);
288         emit WaitingPeriodSecsUpdated(_waitingPeriodSecs);
289     }
290
291     function setPriceDeviationThresholdFactor(uint _priceDeviationThresholdFactor)
            external onlyOwner {
292         flexibleStorage().setPriceDeviationThresholdFactor(
293             SETTING_PRICE_DEVIATION_THRESHOLD_FACTOR,
294             _priceDeviationThresholdFactor
295         );
296         emit PriceDeviationThresholdUpdated(_priceDeviationThresholdFactor);
297     }
```

Listing 3.7: Example Privileged Operations in `SystemSettings`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team intends to introduce multi-sig to mitigate this issue.

3.7 Improved Logic of BaseRewardEscrowV2::accountMergingIsOpen()

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: BaseRewardEscrowV2
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The Horizon protocol has a RewardEscrowV2 contract to keep track of user rewards. To facilitate the user account management, this contract allows the merge of two different user accounts. Our analysis on the account merging logic shows the helper routine of accountMergingIsOpen() can be improved.

In the following, we show the implementation of the accountMergingIsOpen() routine, which has a simple logic in checking whether accountMergingStartTime.add(accountMergingDuration)> block.timestamp. However, it comes with one implicit assumption, i.e., block.timestamp>=accountMergingStartTime. With that, we also suggest to enforce this implicit assumption as well.

```
378     function accountMergingIsOpen() public view returns (bool) {
379         return accountMergingStartTime.add(accountMergingDuration) > block.timestamp;
380     }
```

Listing 3.8: BaseRewardEscrowV2::accountMergingIsOpen()

Recommendation Revisit the above accountMergingIsOpen() routine to make the implicit assumption explicit.

Status The issue has been addressed by the following commit: c345eea.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the Horizon protocol, which is a DeFi platform that facilitates the on-chain creation and derivatives trading of synthetic assets that represent the real economy. Horizon Protocol seeks to provide exposure to real-world asset risk/return profiles via smart contracts on the blockchain. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[7] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.