SMART CONTRACT AUDIT REPORT

for

Pell Network

Prepared By: Xiaomi Huang

PeckShield

May 22, 2024

Document Properties

Client	Pell Network
Title	Smart Contract Audit Report
Target	Pell Network
Version	1.0
Author	Xuxian Jiang
Auditors	Jason Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 22, 2024	Xuxian Jiang	Final Release
1.0-rc	May 22, 2024	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 Pell Network 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 Pell Network

Pell is a restaking protocol that fills in the void in trust and security in the fledgling BTC ecosystem, especially BTC L2 networks. Simultaneously, it expands the single yield profile for BTC and its LSD via a decentralized trust marketplace. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of The Pell Network Protocol

Item	Description
Name	Pell Network
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 22, 2024

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

- https://github.com/0xPellNetwork/restaking-contracts.git (59216a2)

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

- https://github.com/0xPellNetwork/restaking-contracts.git (274dd2d)

1.2 About PeckShield

PeckShield Inc. [9] 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

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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 deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

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

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) [7], 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 `Pell Network` protocol implementation. 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 logics, 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	2	■■
Informational	0	
Total	4	

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 and 2 low-severity vulnerabilities.

Table 2.1: Key Pell Network Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Initialization Logic in Upgradeable Contracts	Coding Practices	Resolved
PVE-002	Medium	Revisited Third-Party Transfer Enforcement in StrategyManagerV2	Business Logic	Resolved
PVE-003	Low	Improved Withdrawal Logic in StrategyBase	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Initialization Logic in Upgradeable Contracts

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

To facilitate possible future upgrade, the key contracts (e.g., `StrategyManagerV2`) are instantiated as a proxy with actual logic contract in the backend. While examining the related contract construction and initialization logic, we notice current initialization logic can be improved.

In the following, we show an example initialization routine from `StrategyManagerV2`. We notice its constructor has a minimal payload by invoking `_disableInitializers()` to block the logic contract from being initialized. However, it comes to our attention that this `StrategyManagerV2` contract also inherits from other contracts, including `ReentrancyGuardUpgradeable`. These inherited contracts also need to be properly initialized within the below `initialize()` routine. Specifically, there is a need to call `__ReentrancyGuard_init()` to initialize the inherited `ReentrancyGuardUpgradeable` contract.

```
function initialize(
  address initialOwner,
  address initialStrategyWhitelister,
  IPauserRegistry _pauserRegistry,
  uint256 initialPausedStatus
) external initializer {
  _DOMAIN_SEPARATOR = _calculateDomainSeparator();
  _initializePauser(_pauserRegistry, initialPausedStatus);
  _transferOwnership(initialOwner);
  _setStrategyWhitelister(initialStrategyWhitelister);
}
```

Listing 3.1: `StrategyManagerV2::initialize()`

Recommendation Improve the above-mentioned `initialize()` routine in `StrategyManagerV2`. Note another contract `DelegationManagerV2` shares the same issue.

Status This issue has been fixed by the following commit: `434e85e`.

3.2 Revisited Third-Party Transfer Enforcement in StrategyManagerV2

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StrategyManagerV2`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The restaking support allows the users to deposit the intended tokens into the staking contract. In the meantime, it also allows the user to redeem the staked funds. While examining the related deposit logic, we notice the current implementation needs to be revisited.

To elaborate, we show below the implementation of two related deposit routines, i.e., `depositIntoStrategyWithStaker()` and `depositIntoStrategyWithSignature()`. Note the latter is explicitly designed to help one address deposit for another. And a signature is required for this routine to eliminate the possibility of griefing attacks, "specifically those targeting stakers who may be attempting to undelegate". However, we notice the first routine does not have the `thirdPartyTransfersForbidden` check in place, which still allows for arbitrary staker being credited and therefore permit the above to-be-blocked grief attack. In addition, the `NatSpec` comment of the first routine contains incorrect information, which should be fixed. Specifically, it currently states the shares are credited to `msg.sender` (line 103), which should be the given staker.

```
102     /**
103      * @notice Deposits `amount` of `token` into the specified `strategy`, with the
            resultant shares credited to `msg.sender`
104      * @param staker Staker address
105      * @param strategy is the specified strategy where deposit is to be made,
106      * @param token is the denomination in which the deposit is to be made,
107      * @param amount is the amount of token to be deposited in the strategy by the staker
108      * @return shares The amount of new shares in the `strategy` created as part of the
            action.
109      * @dev The `msg.sender` must have previously approved this contract to transfer at
            least `amount` of `token` on their behalf.
110      *
111      * WARNING: Depositing tokens that allow reentrancy (eg. ERC-777) into a strategy is
            not recommended.  This can lead to attack vectors
```

```
 *          where the token balance and corresponding strategy shares are not in sync
     upon reentrancy.
 */
function depositIntoStrategyWithStaker(
  address staker,
  IStrategy strategy,
  IERC20 token,
  uint256 amount
) external onlyWhenNotPaused(PAUSED_DEPOSITS) nonReentrant returns (uint256 shares) {
  shares = _depositIntoStrategy(staker, strategy, token, amount);
}

/**
 * @notice Used for depositing an asset into the specified strategy with the resultant
      shares credited to `staker`,
 * who must sign off on the action.
 * Note that the assets are transferred out/from the `msg.sender`, not from the `
     staker`; this function is explicitly designed
 * purely to help one address deposit 'for' another.
 * @param strategy is the specified strategy where deposit is to be made,
 * @param token is the denomination in which the deposit is to be made,
 * @param amount is the amount of token to be deposited in the strategy by the staker
 * @param staker the staker that the deposited assets will be credited to
 * @param expiry the timestamp at which the signature expires
 * @param signature is a valid signature from the `staker`. either an ECDSA signature
     if the `staker` is an EOA, or data to forward
 * following EIP-1271 if the `staker` is a contract
 * @return shares The amount of new shares in the `strategy` created as part of the
     action.
 * @dev The `msg.sender` must have previously approved this contract to transfer at
     least `amount` of `token` on their behalf.
 * @dev A signature is required for this function to eliminate the possibility of
     griefing attacks, specifically those
 * targeting stakers who may be attempting to undelegate.
 * @dev Cannot be called if thirdPartyTransfersForbidden is set to true for this
     strategy
 *
 *  WARNING: Depositing tokens that allow reentrancy (eg. ERC-777) into a strategy is
     not recommended.  This can lead to attack vectors
 *          where the token balance and corresponding strategy shares are not in sync
     upon reentrancy
 */
function depositIntoStrategyWithSignature(
  IStrategy strategy,
  IERC20 token,
  uint256 amount,
  address staker,
  uint256 expiry,
  bytes memory signature
) external onlyWhenNotPaused(PAUSED_DEPOSITS) nonReentrant returns (uint256 shares) {
  require(!thirdPartyTransfersForbidden[strategy], 'StrategyManager.
      depositIntoStrategyWithSignature: third transfers disabled');
```

```
    require(expiry >= block.timestamp, 'StrategyManager.depositIntoStrategyWithSignature
        : signature expired');
    ...
  }
```

Listing 3.2: `StrategyManagerV2::depositIntoStrategyWithStaker()`

Recommendation Revise the above deposit logic to avoid grief attacks.

Status This issue has been fixed by the following commit: `669527f`.

3.3 Improved Withdrawal Logic in StrategyBase

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, the restaking support allows the users to deposit the intended tokens into the staking contract. In the meantime, it also allows the user to redeem the staked funds. While examining the current unstaking logic, we notice the related logic also can be improved.

In the following, we show the implementation of the related withdrawal logic in `StrategyBase`. It has a rather straightforward logic designed to withdraw tokens from the strategy. Note this function is only callable by the `StrategyManager` contract. We notice the internal calculation of withdrawan token amount mirrors the `sharesToUnderlying(amountShares)` call, but claims to be different since the `totalShares` has already been decremented. Our analysis shows that it is incorrect as the `totalShares` has not been decremented yet. With that, we can simply call `sharesToUnderlying(amountShares)` for improved clarity and code maintenance.

```
  function withdraw(
    address recipient,
    IERC20 token,
    uint256 amountShares
  ) external virtual override onlyWhenNotPaused(PAUSED_WITHDRAWALS) onlyStrategyManager
      {
    // call hook to allow for any pre-withdrawal logic
    _beforeWithdrawal(recipient, token, amountShares);

    // copy `totalShares` value to memory, prior to any change
    uint256 priorTotalShares = totalShares;

```

```
    require(amountShares <= priorTotalShares, 'StrategyBase.withdraw: amountShares must be less than or equal to totalShares');

    /**
     * @notice calculation of amountToSend *mirrors* `sharesToUnderlying(amountShares)`, but is different since the `totalShares` has already
     * been decremented. Specifically, notice how we use `priorTotalShares` here instead of `totalShares`.
     */
    // account for virtual shares and balance
    uint256 virtualPriorTotalShares = priorTotalShares + SHARES_OFFSET;
    uint256 virtualTokenBalance = _tokenBalance() + BALANCE_OFFSET;
    // calculate ratio based on virtual shares and balance, being careful to multiply before dividing
    uint256 amountToSend = (virtualTokenBalance * amountShares) / virtualPriorTotalShares;

    // Decrease the `totalShares` value to reflect the withdrawal
    totalShares = priorTotalShares - amountShares;

    _afterWithdrawal(recipient, token, amountToSend);
}
```

Listing 3.3: `StrategyBase::withdraw()`

Recommendation Improve the above routine with the simplified logic and clarity.

Status This issue has been fixed by the following commit: `274dd2d`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

The restaking support in `Pell Network` is designed with a privileged account, i.e., `owner`, that play a critical role in governing and regulating the system-wide operations (e.g., configure parameters, assign roles, and upgrade contract). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
function setMinWithdrawalDelay(uint256 newMinWithdrawalDelay) external onlyOwner {
  _setMinWithdrawalDelay(newMinWithdrawalDelay);
}

...
function setStrategyWithdrawalDelay(IStrategy[] calldata strategies, uint256[]
    calldata withdrawalDelay) external onlyOwner {
  _setStrategyWithdrawalDelay(strategies, withdrawalDelay);
}

...
function updateWrappedTokenGateway(address _newWrappedTokenGateway) external onlyOwner
    {
  emit UpdateWrappedTokenGateway(wrappedTokenGateway, _newWrappedTokenGateway);
  wrappedTokenGateway = _newWrappedTokenGateway;
}
```

Listing 3.4: Example Privileged Operations in DelegationManagerV2

We understand the need of the privileged function for contract maintenance, but at the same time the extra power to the `owner` may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

In the meantime, the key contracts make use of the proxy suppport to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

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 This issue has been mitigated with the use of a multi-sig for the `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Pell Network` protocol, which is a restaking protocol that fills in the void in trust and security in the fledgling `BTC` ecosystem, especially `BTC` L2 networks. Simultaneously, it expands the single yield profile for `BTC` and its `LSD` via a decentralized trust marketplace. During the audit, we notice that the current code base is well organized and those identified issues are promptly mitigated and fixed.

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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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