SMART CONTRACT AUDIT REPORT

for

Megapool

PeckShield

Prepared By: Xiaomi Huang

PeckShield

May 2, 2025

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 2, 2025	Xuxian Jiang	Final Release
1.0-rc	May 1, 2025	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Megapool 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 Megapool

Megapool is a comprehensive smart contract system for managing token pools, staking, and yield distribution onchain. It implements a flexible and secure system with a number of unique features, including token pooling with TVL limits, fixed yield staking with lock periods, stPELL token system with cooldown periods, reward distribution through authorized rewarders, and integration with Pell's strategy system. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of The Pell Protocol

Item	Description
Name	Pell Network
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 2, 2025

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

- https://github.com/0xPellNetwork/megapool-contracts.git (600bbca)

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

- https://github.com/0xPellNetwork/megapool-contracts.git (d625843)

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

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 [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.

PeckShield

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 `Megapool` 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 Megapool Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Restricted Staker Enforcement in StakedPell	Security Features	Resolved
PVE-002	Low	Possible Denial-of-Service in User Withdrawal	Time And State	Resolved
PVE-003	Low	Improved Initialization Logic in Upgradeable Contracts	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 Incorrect Restricted Staker Enforcement in StakedPell

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `StakedPell`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The `Megapool` protocol has a core `StakedPell` contract that allows users to stake for rewards. While examining the logic to withdraw user stakes, we notice an issue that somehow allows a restricted staker to withdraw.

In the following, we show the implementation of the related `_withdraw()` routine. As the name indicates, this routine is designed to handle a withdrawal request. By design, a restricted staker should be blocked. However, it comes to our attention that current implementation fails to achieve that. In particular, it only checks the restriction status of `caller` and `receiver`, but not the actual `_owner`. As a result, a restricted staker can simply bypass the restriction and perform the withdrawal without being blocked.

```
function _withdraw(
  address caller,
  address receiver,
  address _owner,
  uint256 assets,
  uint256 shares
) internal override nonReentrant notZero(assets) notZero(shares) {
  if (hasRole(FULL_RESTRICTED_STAKER_ROLE, caller)  hasRole(
      FULL_RESTRICTED_STAKER_ROLE, receiver)) {
    revert('StakedPell: operation not allowed');
  }
  super._withdraw(caller, receiver, _owner, assets, shares);
  _checkMinShares();
```

```
}
```

Listing 3.1: `StakedPell::_withdraw()`

Recommendation Improve the above-mentioned routine to properly block a restricted staker from withdrawing.

Status This issue has been fixed by the following commit: `13f765d`.

3.2 Possible Denial-of-Service in User Withdrawal

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `StakedPell`
- Category: Time and State [7]
- CWE subcategory: CWE-682 [2]

Description

The `Megapool` protocol has a core `StakedPell` contract that allows users to stake tokens for rewards. By design, the user stake may be withdrawn after a necessary cooldown duration. While examining the actual unstake logic, we notice the cooldown duration enforcement may be abused to block a legitimate user from withdrawing.

```
function cooldownSharesFor(address receiver, uint256 shares) external override
    whenNotWithdrawPaused ensureCooldownOn returns (uint256) {
  require(shares <= maxRedeem(_msgSender()), 'StakedPell: excessive redeem amount');
  uint256 assets = previewRedeem(shares);

  cooldowns[receiver].cooldownEnd = uint104(block.timestamp) + cooldownDuration;
  cooldowns[receiver].underlyingAmount += assets;

  _withdraw(_msgSender(), address(SILO), _msgSender(), assets, shares);

  return assets;
}
```

Listing 3.2: `StakedPell::cooldownSharesFor()`

To elaborate, we show above a related `cooldownSharesFor()` routine. This routine allows any user to manipulate the cooldown expiry timestamp at the cost of losing `1 wei` of share. In other words, before a user's withdrawal request is ready to be served, it is always possible to extend the expiry timestamp by another cool down duration. To fix, we suggest to validate the caller to be a trusted entity or require the user's authorization.

Recommendation Improve the above cool down enforcement mechanism to ensure it will not block legitimate user withdrawal.

Status This issue has been fixed by the following commit: `a2ea2e5`.

3.3 Improved Initialization Logic in Upgradeable Contracts

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MegapoolManager`, `FixedYieldStaking`
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

To facilitate possible future upgrade, a number of core contracts in `Megapool` is instantiated as a proxy with actual logic contract in the backend. While examining the related contract construction and initialization logic, we notice current initialization implementation can be improved.

In the following, we show the initialization routine from an example `MegapoolManager` contract. We notice its constructor is missing and can be added with the following statement, i.e., `_disableInitializers();`, to prevent the logic contract from being initialized by unauthorized party. In the meantime, we notice the `initialize()` function can be improved by also initializing one parent contract, i.e., `Ownable2StepUpgradeable`. In other words, there is a need to call `__Ownable2Step_init()` as well inside the `initialize()` function.

```
function initialize(address initialOwner, address initialMegapoolWhitelister,
    IPauserRegistry _pauserRegistry) external initializer {
  __ReentrancyGuard_init();
  _initializePauser(_pauserRegistry, UNPAUSE_ALL);
  _setMegapoolWhitelister(initialMegapoolWhitelister, true);
  _transferOwnership(initialOwner);
}
```

Listing 3.3: `MegapoolManager::initialize()`

Recommendation Improve the above-mentioned `initialize()` function, Note this issue also affects another contracts, i.e., `FixedYieldStaking`.

Status This issue has been fixed by the following commit: `a3825f5`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

The core contracts in Megapool are designed with a privileged account, i.e., owner, that plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, manage roles/blacklists, 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 setCooldownDuration(uint24 duration) external override onlyOwner {
  require(duration <= MAX_COOLDOWN_DURATION, 'StakedPell: invalid cooldown');

  uint24 previousDuration = cooldownDuration;
  cooldownDuration = duration;
  emit CooldownDurationUpdated(previousDuration, cooldownDuration);
}

/// @inheritdoc IStakedPell
function configPoolLimit(uint256 _poolDepositLimit, uint256 _perAddressLimit) external
    override onlyOwner {
  emit ConfigPoolLimit(maxPoolDeposit, _poolDepositLimit);
  emit ConfigPerAddressLimit(maxPerAddressDeposit, _perAddressLimit);
  maxPoolDeposit = _poolDepositLimit;
  maxPerAddressDeposit = _perAddressLimit;
}

/// @inheritdoc IStakedPell
function updateDepositPause(bool _isPaused) external onlyOwner {
  _pausedDeposit = _isPaused;
  emit DepositPauseUpdated(_isPaused);
}

/// @inheritdoc IStakedPell
function updateWithdrawPause(bool _isPaused) external onlyOwner {
  _pausedWithdraw = _isPaused;
  emit WithdrawPauseUpdated(_isPaused);
}
```

Listing 3.4: Example Privileged Operations in StakedPell

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.

PeckShield

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Megapool` protocol, which is a comprehensive smart contract system for managing token pools, staking, and yield distribution on-chain. It implements a flexible and secure system with a number of unique features, including token pooling with `TVL` limits, fixed yield staking with lock periods, `stPELL` token system with cooldown periods, reward distribution through authorized rewarders, and integration with `Pell`'s strategy system. 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.

PeckShield

References

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

[2] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[3] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.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: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[7] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.