



SMART CONTRACT AUDIT REPORT

for

Nexon Pooled Protocol



Prepared By: Xiaomi Huang

PeckShield
March 23, 2023

Document Properties

Client	Nexon Finance
Title	Smart Contract Audit Report
Target	Nexon Pooled
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 23, 2023	Xuxian Jiang	Final Release
1.0-rc	March 22, 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	4
1.1	About Nexon Pooled	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	9
2.1	Summary	9
2.2	Key Findings	10
3	Detailed Results	11
3.1	Non ERC20-Compliance Of CToken	11
3.2	Interface Inconsistency Between CErc20 And CEther	14
3.3	Possible Front-Running For Unintended Payment In repayBorrowBehalf()	15
3.4	Trust Issue of Admin Keys	18
4	Conclusion	20
	References	21

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Nexon Pooled` 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 Nexon Pooled

The `Nexon Pooled` protocol is a pioneering decentralized lending protocol on `zkSync Era`, offering users the ability to lend their assets or obtain leverage through borrowing. The `Pooled v1` smart contract is originally based on `Compound Finance v2`. The platform emphasizes UX, algorithmic risk optimization, and composability. By building on `zkSync Era`, `Pooled v1` provides ultra-low transaction fees, superior UX, speedy transactions, and enhanced capital efficiency. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The `Nexon Pooled` Protocol

Item	Description
Name	Nexon Finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 23, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the lending protocol assumes a trusted price oracle and this audit only covers the following contracts: `CToken.sol`, `CErc20.sol`, `CEther.sol`, `Comptroller.sol`, and `PythOracle.sol`.

- <https://github.com/nexon-finance/nexon-contracts.git> (e8aed75)

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

- <https://github.com/nexon-finance/nexon-contracts.git> (39e11a5)

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	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
		Likelihood		

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.

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

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 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 `Nexon Pooled` 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 Nexon Pooled Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Non ERC20-Compliance Of CToken	Coding Practices	Resolved
PVE-002	Low	Interface Inconsistency Between CErc20 And CEther	Coding Practice	Resolved
PVE-003	Low	Possible Front-Running For Unintended Payment In repayBorrowBehalf()	Time And State	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 Non ERC20-Compliance Of CToken

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: CToken
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

Each asset supported by the Nexon Pooled protocol is integrated through a so-called `CToken` contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting `CTokens`, users can earn interest through the `CToken`'s exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use `CTokens` as collateral. There are currently two types of `CTokens`: `CErc20` and `CEther`. In the following, we examine the ERC20 compliance of these `CTokens`.

Table 3.2: Key State-Changing Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits <code>Transfer()</code> event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	×
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits <code>Transfer()</code> event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits <code>Approval()</code> event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to <code>address(0x0)</code> when new tokens are generated	✓
Approval() event	Is emitted on any successful call to <code>approve()</code>	✓

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we

examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the `CToken` contract. Specifically, the current `transfer()` function simply returns the related error code if the sender does not have sufficient balance to spend. A similar issue is also present in the `transferFrom()` function that does not revert when the sender does not have the sufficient balance or the message sender does not have the enough allowance.

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on <code>transfer()/transferFrom()</code> calls	—
Rebasing	The <code>balanceOf()</code> function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

Recommendation Revise the `CToken` implementation to ensure its ERC20-compliance.

Status This issue has been fixed in the following commit `070f7a8`.

3.2 Interface Inconsistency Between CErc20 And CEther

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.1, each asset supported by the Nexon Pooled protocol is integrated through a so-called cToken contract, which is an ERC20 compliant representation of balances supplied to the protocol. And cTokens are the primary means of interacting with the Nexon Pooled protocol when a user wants to mint(), redeem(), borrow(), repay(), liquidate(), or transfer(). Moreover, there are currently two types of cTokens: CErc20 and CEther. Both types expose the ERC20 interface and they wrap an underlying ERC20 asset and Ether, respectively.

While examining these two types, we notice their interfaces are surprisingly different. Using the repayBorrow() function as an example, the CErc20 type returns an error code while the CEther type simply reverts upon any failure. The similar inconsistency is also present in other routines, including repayBorrowBehalf(), mint(), and liquidateBorrow().

```

104     function repayBorrow(uint repayAmount) external override returns (uint) {
105         (uint err, ) = repayBorrowInternal(repayAmount);
106         return err;
107     }
108
109     /**
110      * @notice Sender repays a borrow belonging to borrower
111      * @param borrower the account with the debt being paid off
112      * @param repayAmount The amount to repay
113      * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
114      */
115     function repayBorrowBehalf(
116         address borrower,
117         uint repayAmount
118     ) external override returns (uint) {
119         (uint err, ) = repayBorrowBehalfInternal(borrower, repayAmount);
120         return err;
121     }

```

Listing 3.1: CErc20::repayBorrow()/repayBorrowBehalf()

```

93     function repayBorrow() external payable {
94         (uint err, ) = repayBorrowInternal(msg.value);
95         requireNoError(err, "repayBorrow failed");
96     }

```

```

97
98  /**
99   * @notice Sender repays a borrow belonging to borrower
100   * @dev Reverts upon any failure
101   * @param borrower the account with the debt being payed off
102   */
103  function repayBorrowBehalf(address borrower) external payable {
104      (uint err, ) = repayBorrowBehalfInternal(borrower, msg.value);
105      requireNoError(err, "repayBorrowBehalf failed");
106  }

```

Listing 3.2: CEther::repayBorrow()/repayBorrowBehalf()

Recommendation Ensure the consistency between these two types: CErc20 and CEther.

Status This issue has been fixed in the following commit [cb89b23](#).

3.3 Possible Front-Running For Unintended Payment In repayBorrowBehalf()

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: CToken
- Category: Time and State [7]
- CWE subcategory: CWE-663 [4]

Description

As mentioned earlier, the Nexon Pooled protocol is in essence an over-collateralized lending pool that has the lending functionality and supports a number of normal lending functionalities for supplying and borrowing users, i.e., mint()/redeem() and borrow()/repay(). In the following, we examine one specific functionality, i.e., repay().

To elaborate, we show below the core routine repayBorrowFresh() that actually implements the main logic behind the repay() routine. This routine allows for repaying partial or full current borrowing balance. It is interesting to note that the Nexon Pooled protocol supports the payment on behalf of another borrowing user (via repayBorrowBehalf()). And the repayBorrowFresh() routine supports the corner case when the given amount is larger than the current borrowing balance. In this corner case, the protocol assumes the intention for a full repayment.

```

1202  function repayBorrowFresh(
1203      address payer,
1204      address borrower,
1205      uint256 repayAmount
1206  ) internal returns (uint256, uint256) {

```

```

1207     /* Fail if repayBorrow not allowed */
1208     uint256 allowed = comptroller.repayBorrowAllowed(
1209         address(this),
1210         payer,
1211         borrower,
1212         repayAmount
1213     );
1214     if (allowed != 0) {
1215         return (
1216             failOpaque(
1217                 Error.COMPTROLLER_REJECTION,
1218                 FailureInfo.REPAY_BORROW_COMPTROLLER_REJECTION,
1219                 allowed
1220             ),
1221             0
1222         );
1223     }

1225     /* Verify market's block number equals current block number */
1226     if (accrualBlockNumber != getBlockNumber()) {
1227         return (
1228             fail(
1229                 Error.MARKET_NOT_FRESH,
1230                 FailureInfo.REPAY_BORROW_FRESHNESS_CHECK
1231             ),
1232             0
1233         );
1234     }

1236     RepayBorrowLocalVars memory vars;

1238     /* We remember the original borrowerIndex for verification purposes */
1239     vars.borrowerIndex = accountBorrows[borrower].interestIndex;

1241     /* We fetch the amount the borrower owes, with accumulated interest */
1242     (vars.mathErr, vars.accountBorrows) = borrowBalanceStoredInternal(
1243         borrower
1244     );
1245     if (vars.mathErr != MathError.NO_ERROR) {
1246         return (
1247             failOpaque(
1248                 Error.MATH_ERROR,
1249                 FailureInfo
1250                     .REPAY_BORROW_ACCUMULATED_BALANCE_CALCULATION_FAILED,
1251                 uint256(vars.mathErr)
1252             ),
1253             0
1254         );
1255     }

1257     /* If repayAmount == -1, repayAmount = accountBorrows */
1258     if (repayAmount == type(uint256).max) {

```



```

1259         vars.repayAmount = vars.accountBorrows;
1260     } else {
1261         vars.repayAmount = repayAmount;
1262     }

1264     //////////////////////////////////////
1265     // EFFECTS & INTERACTIONS
1266     // (No safe failures beyond this point)

1268     /*
1269     * We call doTransferIn for the payer and the repayAmount
1270     * Note: The cToken must handle variations between ERC-20 and ETH underlying.
1271     * On success, the cToken holds an additional repayAmount of cash.
1272     * doTransferIn reverts if anything goes wrong, since we can't be sure if side
1273     * effects occurred.
1274     * it returns the amount actually transferred, in case of a fee.
1275     */
1276     vars.actualRepayAmount = doTransferIn(payer, vars.repayAmount);

1277     /*
1278     * We calculate the new borrower and total borrow balances, failing on underflow
1279     * :
1280     * accountBorrowsNew = accountBorrows - actualRepayAmount
1281     * totalBorrowsNew = totalBorrows - actualRepayAmount
1282     */
1283     (vars.mathErr, vars.accountBorrowsNew) = subUInt(
1284         vars.accountBorrows,
1285         vars.actualRepayAmount
1286     );
1287     require(
1288         vars.mathErr == MathError.NO_ERROR,
1289         "REPAY_BORROW_NEW_ACCOUNT_BORROW_BALANCE_CALCULATION_FAILED"
1290     );

1291     (vars.mathErr, vars.totalBorrowsNew) = subUInt(
1292         totalBorrows,
1293         vars.actualRepayAmount
1294     );
1295     require(
1296         vars.mathErr == MathError.NO_ERROR,
1297         "REPAY_BORROW_NEW_TOTAL_BALANCE_CALCULATION_FAILED"
1298     );

1299     /* We write the previously calculated values into storage */
1300     accountBorrows[borrower].principal = vars.accountBorrowsNew;
1301     accountBorrows[borrower].interestIndex = borrowIndex;
1302     totalBorrows = vars.totalBorrowsNew;

1303

1304     /* We emit a RepayBorrow event */
1305     emit RepayBorrow(
1306         payer,
1307         borrower,

```

```

1309         vars.actualRepayAmount,
1310         vars.accountBorrowsNew,
1311         vars.totalBorrowsNew
1312     );
1314     return (uint256(Error.NO_ERROR), vars.actualRepayAmount);
1315 }

```

Listing 3.3: CToken::repayBorrowFresh()

This is a reasonable assumption, but our analysis shows this assumption may be taken advantage of to launch a front-running `borrow()` operation, resulting in a higher borrowing balance for repayment. To avoid this situation, it is suggested to disallow the repayment amount of `-1` to imply the full repayment. In fact, it is always suggested to use the exact payment amount in the `repayBorrowBehalf()` case.

Recommendation Revisit the generous assumption of using repayment amount of `-1` as the indication of full repayment.

Status This issue has been fixed in the following commit `070f7a8`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

As mentioned before, the Nexon Pooled protocol is a decentralized lending protocol. While reviewing the contract, we notice there is an administrative account, `admin`, which plays a critical role in governing and regulating operations. It also has the privilege to control or govern the flow of assets managed by various protocol contract. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged `admin` account and its related privileged accesses in current contract.

```

94     function setDirectPrice(address token, uint256 price) external onlyAdmin {
95         directPrices[token] = price;
96     }
98     function updateAdmin(address newAdmin) external onlyAdmin {
99         admin = newAdmin;
100     }

```

```
102     function updatePythPriceIds(  
103         address[] memory _tokens,  
104         bytes[] memory _ids  
105     ) external onlyAdmin {  
106         for (uint256 i = 0; i < _tokens.length; i++) {  
107             pythPriceIds[_tokens[i]] = bytes32(_ids[i]);  
108         }  
109     }
```

Listing 3.4: Multiple Setters in PythOracle

We understand the need of the privileged functions for contract maintenance, but it is worrisome if the privileged `admin` account is a plain EOA account as this may pose counter-party risk to the exchange users. 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, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts may have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 resolved as the team plans to use the `multisig` to act as the privileged owner.

4 | Conclusion

In this audit, we have analyzed the `Nexon Pooled` protocol design and implementation. The protocol is a pioneering decentralized lending protocol on `zkSync Era`, offering users the ability to lend their assets or obtain leverage through borrowing. The `Pooled v1` smart contract is originally based on `Compound Finance v2`. The platform emphasizes UX, algorithmic risk optimization, and composability. By building on `zkSync Era`, `Pooled v1` provides ultra-low transaction fees, superior UX, speedy transactions, and enhanced capital efficiency. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed 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-1041: Use of Redundant Code. <https://cwe.mitre.org/data/definitions/1041.html>.
- [2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [3] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.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: Concurrency. <https://cwe.mitre.org/data/definitions/557.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>.