



SMART CONTRACT AUDIT REPORT

for

Onyx



Prepared By: Xiaomi Huang

PeckShield
October 22, 2024

Document Properties

Client	Onyx
Title	Smart Contract Audit Report
Target	Onyx
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 22, 2024	Xuxian Jiang	Final Release
1.0-rc	October 10, 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	4
1.1	About Onyx	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	Improved Validation of Function Parameters	11
3.2	Timely Rewards Update Upon rewardPerBlock Change	12
3.3	Suggested Adherence of Checks-Effects-Interactions	13
3.4	Trust Issue Of Admin Keys	14
4	Conclusion	16
	References	17

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the staking support in `Onyx`, 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 Onyx

`Onyx Protocol` is a blockchain that enables the streamlined issuance and management of digital assets via smart contracts. It is maintained by a federation of block signers and features a scalable `UTXO` model for efficient transaction validation. The protocol is governed by `Onyx DAO` through `Onyxcoin (XCN)`, which allows token holders to vote on protocol updates. This system is designed to minimize intermediary involvement and increase transparency in financial transactions. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Audited Contracts

Item	Description
Target	Onyx
Website	https://onyx.org/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 22, 2024

In the following, we show the audited contract code deployed at the following addresses:

- CHNStaking: <https://etherscan.io/address/0x23445c63feef8d85956dc0f19ade87606d0e19a9>

- CHNGovernance: <https://etherscan.io/address/0xdec2f31c3984f3440540dc78ef21b1369d4ef767>
- CHNReward: <https://etherscan.io/address/0x28ca9caae31602d0312ebf6466c9dd57fca5da93>
- MasterChef: <https://etherscan.io/address/0x3fa642c0bbad64569eb8424af35f518347249216>

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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) [9], 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 implementation of the staking support in `Onyx`. 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	1	
Low	3	
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 1 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key Onyx Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation of Function Parameters	Coding Practices	Confirmed
PVE-002	Low	Timely Rewards Update Upon rewardPerBlock Change	Business Logic	Confirmed
PVE-003	Low	Suggested Adherence of Checks-Effects-Interactions	Time And State	Confirmed
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Improved Validation of Function Parameters

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: CHNStaking
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Onyx` protocol is no exception. Specifically, if we examine the `CHNStaking` contract, it has defined a number of protocol-wide risk parameters, such as `rewardPerBlock` and `MULTIPLIER`. In the following, we show the corresponding routine that allows for their changes.

```

1126     function initialize(
1127         IERC20 _rewardToken,
1128         uint256 _rewardPerBlock,
1129         uint256 _startBlock,
1130         uint256 _bonusEndBlock,
1131         uint256 _multiplier,
1132         address _rewardVault
1133     ) public initializer {
1134         require(_rewardVault != address(0) && address(_rewardToken) != address(0), "Zero
            address validation");
1135         require(_startBlock < _bonusEndBlock, "Start block lower than bonus end block");
1136         require(_rewardPerBlock < _rewardToken.totalSupply(), "Reward per block bigger
            than reward token total supply");
1137         require(BONUS_MULTIPLIER < 100, "Bonus multiplier bigger than 100x reward bonus")
            ;
1138         __Ownable_init();
1139         rewardToken = _rewardToken;
1140         rewardPerBlock = _rewardPerBlock;
1141         startBlock = _startBlock;
1142         bonusEndBlock = _bonusEndBlock;
1143         BONUS_MULTIPLIER = _multiplier;

```

```

1144     rewardVault = _rewardVault;
1145 }

```

Listing 3.1: CHNStaking:: initialize ()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, the above routine can be improved by enforcing the following requirement, i.e., `require(_multiplier < 100)` (line 1137).

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been confirmed.

3.2 Timely Rewards Update Upon rewardPerBlock Change

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: CHNStaking
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The CHNStaking contract provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the reward per block can be adjusted via `setRewardPerblock()`. When analyzing the `setRewardPerblock()` routine, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new `rewardPerBlock` becomes effective.

```

1196     function setRewardPerblock(uint256 speed)
1197     public
1198     onlyOwner {
1199         rewardPerBlock = speed;
1200     }

```

Listing 3.2: CHNStaking::setRewardPerblock()

If the call to `massUpdatePools()` is not immediately invoked before updating `rewardPerBlock`, certain situations may be crafted to create an unfair reward distribution. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` before the `rewardPerBlock` update.

Status The issue has been confirmed.

3.3 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: LiquidityMining
- Category: Time and State [8]
- CWE subcategory: CWE-663 [3]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the DAO [13] exploit, and the Uniswap/Lendf.Me hack [12].

We notice an occasion where the `checks-effects-interactions` principle is violated. Using the `MasterChef` as an example, the `deposit()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`.

Apparently, the interaction with the external contract (line 871) starts before effecting the update on internal state (line 876–878), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the very same `deposit()` function.

```

856     function deposit(uint256 _pid, uint256 _amount) public {
857
858         PoolInfo storage pool = poolInfo[_pid];
859         UserInfo storage user = userInfo[_pid][msg.sender];
860         updatePool(_pid);
861         if (user.amount > 0) {
862             uint256 pending =
863                 user.amount.mul(pool.accXcnPerShare).div(1e12).sub(

```

```

864         user.rewardDebt
865     );
866     if (pending > 0) {
867         safeXcnTransfer(msg.sender, pending);
868     }
869 }
870 if (_amount > 0) {
871     pool.lpToken.safeTransferFrom(
872         address(msg.sender),
873         address(this),
874         _amount
875     );
876     user.amount = user.amount.add(_amount);
877 }
878 user.rewardDebt = user.amount.mul(pool.accXcnPerShare).div(1e12);
879 emit Deposit(msg.sender, _pid, _amount);
880 }

```

Listing 3.3: MasterChef::deposit()

In the meantime, we should mention that the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for re-entrancy.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions best practice. Note this issue also affects other routines, including MasterChef::withdraw() and CHNStaking::stake().

Status The issue has been confirmed.

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 [2]

Description

In the onyx protocol, there is a privileged account (owner) that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure parameters, manage reward pools, and recover stuck funds). In the following, we show the representative functions potentially affected by the privilege of this account.

```
1159     function add(
```

```

1160     uint256 _allocPoint,
1161     IERC20 _stakeToken
1162 ) public onlyOwner {
1163     ...
1164 }
1165
1166 // Update the given pool's XCN allocation point. Can only be called by the Timelock
    and DAO.
1167 // This function can be only called by Timelock and DAO with voting power
1168 function set(
1169     uint256 _pid,
1170     uint256 _allocPoint
1171 ) public onlyOwner validatePoolByPid(_pid) {
1172     ...
1173 }
1174
1175 // Update reward per block by the Timelock and DAO
1176 function setRewardPerblock(uint256 speed)
1177     public
1178     onlyOwner {
1179     ...
1180 }
1181

```

Listing 3.4: Example Privileged Operations in CHNStaking

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

4 | Conclusion

In this audit, we have analyzed the design and implementation of the staking support in `Onyx`, which is a blockchain that enables the streamlined issuance and management of digital assets via smart contracts. It is maintained by a federation of block signers and features a scalable `UTXO` model for efficient transaction validation. The protocol is governed by `Onyx DAO` through `Onyxcoin (XCN)`, which allows token holders to vote on protocol updates. This system is designed to minimize intermediary involvement and increase transparency in financial transactions. 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-663: Use of a Non-reentrant Function in a Concurrent Context. <https://cwe.mitre.org/data/definitions/663.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 CATEGORY: Concurrency. <https://cwe.mitre.org/data/definitions/557.html>.
- [9] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.

- [10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [11] PeckShield. PeckShield Inc. <https://www.peckshield.com>.
- [12] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.
- [13] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

