



SMART CONTRACT AUDIT REPORT

for

Xterio Protocol



Prepared By: Xiaomi Huang

PeckShield
March 19, 2024

Document Properties

Client	Xterio
Title	Smart Contract Audit Report
Target	Xterio
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	March 19, 2024	Xuxian Jiang	Final Release
1.0-rc1	March 17, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 156 0639 2692
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About Xterio Protocol	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	6
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Strengthened Validation on Function Arguments	12
3.2	Suggested Adherence of Checks-Effects-Interactions	13
3.3	Trust Issue of Admin Keys	14
3.4	Improved Constructor Logic in TokenGateway	15
4	Conclusion	17
	References	18

1 | Introduction

Given the opportunity to review the design document and related smart contract source code of the `Xterio` 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.

Table 1.1: Basic Information of Xterio

Item	Description
Name	Xterio
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 19, 2024

1.1 About Xterio Protocol

`Xterio` is a Web3 gaming ecosystem & infrastructure, distinguishing itself as a gaming publisher with top-notch development skills and unparalleled distribution expertise. `XTER` is the ecosystem-wide ERC20 token that is set to play a central role in all first-party games, which include digital asset purchases like in-game characters, weapons, outfits, and many more. Its integration can also take on a more flexible approach, such as a payment option or a reward system. `Xterio` is dedicated to the distribution, operation, and marketing of mobile and PC Web3 games to increase `XTER` utilization.

The early publishing success already sets the protocol apart in the current market, and the `NFT launchpad` stands as one of the most, if not the most, successful in recent performance. Notably, it has successfully launched `Overworld`, `Age of Dino`, `Persona Journey`, `Army of Fortune`, and other game

NFTs, contributing to the platform's TVL reaching tens of millions. In the following, we show the Git repository of reviewed files and the commit hash value used in this audit.

- <https://github.com/XterioTech/xt-contracts.git> (3665d15)

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

- <https://github.com/XterioTech/xt-contracts.git> (0fae162)

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	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 the 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 checklist of 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) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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 Logic	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.



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.



2 | Findings

2.1 Summary

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

Severity	# of Findings	
Critical	0	
High	0	
Medium	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 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 Xterio Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Strengthened Validation of Function Parameters	Coding Practices	Resolved
PVE-002	Medium	Suggested Adherence of Checks-Effects-Interactions	Timing And State	Resolved
PVE-003	Low	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-002	Low	Improved Constructor Logic in Token-Gateway	Coding Practices	Resolved

Besides 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 Strengthened Validation on Function Arguments

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MarketplaceV2, DepositMinter
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Xterio protocol is no exception. Specifically, if we examine the `DepositMinter` contract, it has defined a number of protocol-wide states and parameters, such as `auctionStartTime` / `auctionEndTime` and `unitPrice`. In the following, we show the corresponding routines that initialize the related storage states.

```

115     function setAuctionEndTime(uint256 _t) external onlyRole(MANAGER_ROLE) {
116         require(_t > block.timestamp, "DepositMinter: invalid timestamp");
117         auctionEndTime = _t;
118     }
119
120     function setAuctionStartTime(uint256 _t) external onlyRole(MANAGER_ROLE) {
121         auctionStartTime = _t;
122     }

```

Listing 3.1: `DepositMinter::setAuctionEndTime()/setAuctionStartTime()`

These states and 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. For example, the above routines can be improved by enforcing the following requirements respectively: `require(auctionStartTime_ < _t)` and `require(auctionEndTime_ > _t)`.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. Note the same issue is also applicable to the MarketplaceV2 contract.

Status This issue has been fixed in the following commits: 24f356d. and 0fae162.

3.2 Suggested Adherence of Checks-Effects-Interactions

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Time and State [6]
- 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 [11] exploit, and the Uniswap/Lendf.Me hack [10].

We notice there are occasions where the checks-effects-interactions principle is violated. Using the AuctionMinter as an example, the `sendPayment()` function (see the code snippet below) is provided to externally call to transfer native coins. However, the invocation of an external contract requires extra care in avoiding the above re-entrancy. For example, the interaction with the low-level call (line 69) start before effecting the update on internal state (line 71), 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 same entry function. Fortunately, there is a `nonReentrant` modifier that blocks the reentrancy. However, the adherence of the checks-effects-interactions best practice is still strongly recommended.

```

62     function sendPayment() external {
63         require(
64             block.timestamp > auctionEndTime,
65             "AuctionMinter: payment can only be made after the auction has ended"
66         );
67         require(!paymentSent, "AuctionMinter: payment already sent");
68         uint256 value = _heap.size() * _heap.minBid().price;
69         (bool success, ) = paymentRecipient.call{value: value}("");
70         require(success, "AuctionMinter: failed to send payment");
71         paymentSent = true;

```

72

}

Listing 3.2: AuctionMinter::sendPayment()

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions principle. Note this issue affects other similar routines, including MarketplaceV2::_atomicMatch() and WhitelistMinter::mintWithSig().

Status This issue has been fixed in the following commit: 24f356d.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Xterio protocol, there is a privileged account (with the) that plays a critical role in governing and regulating the system-wide operations (e.g., assign roles and configure payment tokens). 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 the related privileged accesses in current contracts.

```

85     function setGateway(address _g) external onlyRole(DEFAULT_ADMIN_ROLE) {
86         gateway = _g;
87     }
88
89     function setRecipient(address _r) external onlyRole(DEFAULT_ADMIN_ROLE) {
90         paymentRecipient = _r;
91     }
92
93     function setNftAddress(address _addr) external onlyRole(MANAGER_ROLE) {
94         nftAddress = _addr;
95     }
96
97     function setNftAmount(uint256 _amt) external onlyRole(MANAGER_ROLE) {
98         require(
99             nftAmount == 0 & block.timestamp < auctionEndTime,
100             "DepositMinter: nftAmount can not be changed after the auction ended"
101         );
102         nftAmount = _amt;
103     }
104

```

```

105     function setLimitForBuyerAmount(
106         uint256 _amt
107     ) external onlyRole(MANAGER_ROLE) {
108         require(
109             block.timestamp < auctionEndTime,
110             "DepositMinter: limitForBuyerAmount can only be set before the auction end"
111         );
112         limitForBuyerAmount = _amt;
113     }

```

Listing 3.3: Example Privileged Operations in `DepositMinter`

It is worrisome if the privileged account is a 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, a timelock-based mechanism can also be considered as mitigation.

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 conformed and mitigated. The team has used multi-sig contract for the privileged account.

3.4 Improved Constructor Logic in `TokenGateway`

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

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

In the following, we shows its initialization routine. We notice its constructor does not have any payload. With that, it can be improved by adding the following statement, i.e., `_disableInitializers()`. Note this statement is called in the logic contract where the initializer is locked. Therefore any user will not able to call the `initialize()` function in the state of the logic contract and perform any

malicious activity. Note that the proxy contract state will still be able to call this function since the constructor does not effect the state of the proxy contract.

```
95     /**
96      * NFTGateway is an upgradeable function.
97      * When initializing the gateway, a gateway admin address
98      * should be designated.
99      */
100     function initialize(address _gatewayAdmin) public initializer {
101         _grantRole(DEFAULT_ADMIN_ROLE, _gatewayAdmin);
102         _grantRole(GATEWAY_MANAGER_ROLE, _gatewayAdmin);
103     }
```

Listing 3.4: TokenGateway::initialize()

Recommendation Improve the above-mentioned constructor routines in all existing upgradeable contracts, including TokenGateway, DepositMinter, and MarketplaceV2.

Status This issue has been fixed in the following commit: 24f356d.



4 | Conclusion

In this audit, we have analyzed the design and implementation of the `xterio` protocol, which is a web3 gaming ecosystem & infrastructure, distinguishing itself as a gaming publisher with top-notch development skills and unparalleled distribution expertise. `xTER` is the ecosystem-wide ERC20 token that is set to play a central role in all first-party games, which include digital asset purchases like in-game characters, weapons, outfits, and many more. Its integration can also take on a more flexible approach, such as a payment option or a reward system. `xterio` is dedicated to the distribution, operation, and marketing of mobile and PC Web3 games to increase `xTER` utilization. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity`-based 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 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: Concurrency. <https://cwe.mitre.org/data/definitions/557.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>.
- [10] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. <https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09>.

- [11] David Siegel. Understanding The DAO Attack. <https://www.coindesk.com/understanding-dao-hack-journalists>.

