SMART CONTRACT AUDIT REPORT

for

SwopXLending

Prepared By: Xiaomi Huang

PeckShield

August 22, 2022

Document Properties

Client	SwopX
Title	Smart Contract Audit Report
Target	SwopXLending
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 22, 2022	Xuxian Jiang	Final Release
1.0-rc1	August 21, 2021	Xuxian Jiang	Release Candidate #1

PeckShield

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

PeckShield

1 Introduction

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

The SwopX Lending allows members to collateralize their NFTs and access short and long term liquidity from lenders via the SwopX Lending protocol smart contract. In particular, when a loan starts, the borrower and lender will have NFT receipts and they can sell it to a new owner. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of SwopXLending

Item	Description
Name	SwopX
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 22, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. The given repository has a number of contract files and this audit only covers the SwopXLending contract.

- https://github.com/pcanwar/swap.git (fc8a4ff)

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

- https://github.com/pcanwar/swap.git (c88bdd3)

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

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

PeckShield

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

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

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 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 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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `SwopXLending` 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	2	🟦🟦
Low	1	🟦
Informational	0	
Total	3	

PeckShield

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 2 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key SwopXLending Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited nonce Management in submit()	Business Logic	Fixed
PVE-002	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-003	Low	Redundant Data/Code Removal	Coding Practices	Fixed

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.

PeckShield

3 | Detailed Results

3.1 Revisited nonce Management in submit()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: SwopXLending
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The SwopX Lending protocol allows members to collateralize their NFTs and access short and long term liquidity from lenders The borrowers start a loan when they submit a deal. While reviewing the loan submission logic, we notice the current nonce management needs to be revisited.

To elaborate, we show below the related submit() function. It implements a rather straightforward logic in validating the given arguments and then starting a new loan. However, it comes to our attention the validation of given lenderSignature and borrowerSignature requires the freshness of lenderNonce and borrowerNonce. However, the current implementation does not mark both nonces used after the validation, which violates the refreshness requirement.

```
    function submit(uint256 [2] calldata nonces , address _paymentAddress, address
        _lender,
                address _nftcontract,
                uint256 _nftTokenId,
                 uint256 [3] calldata _loanAmounLoanCost,
                uint256 _offeredTime, bytes32 _gist, bytes calldata borrowerSignature,
                    bytes calldata lenderSignature)
         external whenNotPaused nonReentrant supportInterface(_paymentAddress)
        {
         LendingAssets memory _m = LendingAssets({
         paymentContract: address(_paymentAddress),
         listingTime: clockTimeStamp(),
         totalPrincipal:_loanAmounLoanCost[0],
         totalInterest: _loanAmounLoanCost[1],
         totalInterestPaid:0,
         totalAmountLoan:_loanAmounLoanCost[0] + _loanAmounLoanCost[1],
         totalAmountPaid:0,
         termId:1,
         isPaid:false,
         borrowerNonce: nonces[0],
         lenderNonce: nonces[1],
         nftcontract:_nftcontract,
         nftTokenId:_nftTokenId,
         gist: _gist
         });
         require(IERC721(_m.nftcontract).ownerOf( _m.nftTokenId) == msg.sender ,"Not NFT
             Owner");
         require(identifiedSignature[_lender][_m.lenderNonce] != true, "Lender is not
             interested");
         require(_offeredTime >= clockTimeStamp(), "offer expired" );
         require(IERC20(_m.paymentContract).allowance(_lender, receiverAddress) >= _m.
             totalPrincipal, "Not enough allowance" );
         require(_loanAmounLoanCost[2] >= calculatedFee(_m.totalPrincipal),"fee");
         require(_verify(_lender, _hashLending (_m.lenderNonce,_m.paymentContract,
             _offeredTime,
             _m.totalPrincipal,_m.totalInterest,_m.nftcontract,
             msg.sender,_m.nftTokenId,_m.gist)
             ,lenderSignature),"Invalid lender signature");
         require(_verify(msg.sender, _hashBorrower (_m.borrowerNonce,_m.nftcontract,_m.
             nftTokenId,_m.gist),borrowerSignature),"Invalid borrower signature");

         uint256 counterId = counter();
         _assets[counterId] = _m;

         _receipt[counterId].lenderToken = nftCounter();
         _receipt[counterId].borrowerToken = nftCounter();
         Receipt memory _nft = _receipt[counterId];
         _mint(_lender, _nft.lenderToken ) ;
         _setTokenURI(_nft.lenderToken,  string(abi.encodePacked(Strings.toString(_nft.
             lenderToken), ".json")));
         _mint(msg.sender, _nft.borrowerToken ) ;
         _setTokenURI(_nft.borrowerToken,  string(abi.encodePacked(Strings.toString(_nft.
             borrowerToken), ".json")));
         ...
     }
```

Listing 3.1: `SwopXLending::submit()`

Recommendation Properly mark the nonces as used to ensure their refreshness.

Status The issue has been fixed by this commit: `c88bdd3`.

3.2 Trust Issue of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SwopXLending`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `SwopX Lending` contract, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., add new payment tokens, withdraw funds, etc). 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.

```
    function resetTxFee(uint256 _fee, uint256 _txInterestfee) external onlyOwner {
        txfee = _fee;
        txInterestfee = _txInterestfee;
        emit FeeLog(msg.sender, txfee, txInterestfee);
    }


    /*
    * @notice: addToken function for adding cryptocurancy contract address
    * @param _contract address is a contract address
    * @param _mode bool true/ false
    */
    function addToken(address _contract, bool _mode) external
    zeroAddress(_contract) onlyOwner {
        erc20Addrs[IERC20(_contract)] = _mode;
        emit CryptoLog(_contract, _mode) ;
    }
    ...

    function withdraw(address _contract, address _to) external
    zeroAddress(_contract) onlyOwner {
        uint _amount = IERC20(_contract).balanceOf(receiverAddress);
        IERC20(_contract).safeTransfer(_to, _amount);
        emit WithdrawLog(_contract, _to, _amount);
    }
```

Listing 3.2: Various Privileged Functions in `SwopXLending`

Notice that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Moreover, it should be noted that if current contracts are planned to deploy behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Making the above privileges explicit among protocol users.

Status This issue has been confirmed and the team clarifies that the admin key is managed by a multi-sig wallet.

3.3 Redundant State/Code Removal

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SwopXLending
- Category: Coding Practices [5]
- CWE subcategory: CWE-563 [2]

Description

The SwopX Lending protocol makes good use of a number of reference contracts, such as ERC20, SafeERC20, ERC721, and Ownable, to facilitate its code implementation and organization. For example, the SwopXLending smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the SwopXLending contract, there is a number of events that are defined and one of them, i.e., PusedTransferLog, is not used anywhere in the current protocol. The unused event can be safely removed.

```
    // PayLog and PaymentLog event are called in the makePayment function
    event PayLog(uint256 indexed counterId, address indexed nftcontract, uint256 tokenId
        , uint256 paidAmount, uint256 currentTerm, uint256 fee,bytes32 [] proof);

    event PaymentLog(uint256 indexed counterId, address indexed nftcontract, uint256
        tokenId, bool isPaid);

    // DefaultLog event is called in the default function
    event DefaultLog(uint256 indexed counterId, address nftcontract, uint256 tokenId,
        address indexed lender, uint256 fee);

    event PusedTransferLog(address indexed nftcontract, address indexed to, uint256
        tokenId);

    // FeeLog event is called in the resetTxFee function
    event FeeLog(address account, uint256 loanFee, uint256 interestFee );

    // CryptoLog FeeLog event is called in the addToken function
    event CryptoLog(address contracts, bool isSupported);
```

Listing 3.3: Various Events Defined in The `SwopXLending` Contract

Recommendation Consider the removal of the unused event with a simplified, consistent implementation.

Status The issue has been fixed by this commit: `fcdbbf4`.

PeckShield

4 Conclusion

In this audit, we have analyzed the design and implementation of the `SwopX Lending` protocol, which allows members to collateralize their `NFTs` and access short and long term liquidity from lenders. In particular, when a loan starts, the borrower and lender will have `NFT` receipts and they can sell it to a new owner. The current code base can be further improved in both design and implementation. Those identified issues are promptly confirmed and fixed.

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.

PeckShield

References

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

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

[3] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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

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

PeckShield