SMART CONTRACT AUDIT REPORT

for

XCarnival

Prepared By: Xiaomi Huang

PeckShield

July 25, 2022

Document Properties

Client	XCarnival Finance
Title	Smart Contract Audit Report
Target	XCarnival Protocol
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 25, 2022	Luck Hu	Final Release
1.0-rc	July 18, 2022	Luck Hu	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

Given the opportunity to review the design document and related smart contract source code of the XCarnival 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 ERC20-compliance, security or performance. This document outlines our audit results.

1.1 About XCarnival Protocol

The XCarnival is an NFT lending protocol that lets users borrow tokens quickly without selling their NFTs. By supplying tokens into the pools, depositors can earn interests and rewards. And the protocol essentially offers yields on NFT assets, with which users can receive airdrops by pledging their NFTs. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of the XCarnival

Item	Description
Name	XCarnival Finance
Website	https://xcarnival.fi/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 25, 2022

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

- https://github.com/xcarnival/pawn.git (b12b440)

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

- https://github.com/xcarnival/pawn.git (6c5cbdf)

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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [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.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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 `XCarnival` protocol implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	3	■ ■ ■
Low	2	■ ■
Informational	0	
Total	6	

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 high-severity vulnerability, 3 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key XCarnival Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-002	Medium	Potential DoS Against auction()	Business Logic	Mitigated
PVE-003	Low	Proper whenNotPaused(2) Check Before NFT Withdrawal	Coding Practices	Fixed
PVE-004	Low	Tightened Access Control In borrowAllowed()	Coding Practices	Fixed
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-006	Medium	Timely massUpdatePools() During Pool Updates	Business Logic	Confirmed

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 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: Medium
- Target: `XToken`, `XNFT`
- 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 recent `Uniswap/Lendf.Me` hack [12].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `XToken` as an example, the `borrowInternal()` 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`. For example, the interaction with the external contract (line 212) starts before effecting the update on internal states (lines 214-217), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` to the `XNFT::withdrawNFT()`, the user can borrow assets after the `NFT` is withdrawn.

```
198 function borrowInternal(uint256 orderId, address payable borrower, uint256 borrowAmount
    ) internal nonReentrant{
199
```

```
    controller.borrowAllowed(address(this), orderId, borrower, borrowAmount);

    require(accrualBlockNumber == getBlockNumber(),"block number check fails");

    require(getCashPrior() >= borrowAmount, "insufficient balance of underlying asset");

    BorrowLocalVars memory vars;

    vars.orderBorrows = borrowBalanceStoredInternal(orderId);
    vars.orderBorrowsNew = addExp(vars.orderBorrows, borrowAmount);
    vars.totalBorrowsNew = addExp(totalBorrows, borrowAmount);

    doTransferOut(borrower, borrowAmount);

    orderBorrows[orderId].principal = vars.orderBorrowsNew;
    orderBorrows[orderId].interestIndex = borrowIndex;

    totalBorrows = vars.totalBorrowsNew;

    controller.borrowVerify(orderId, address(this), borrower);

    emit Borrow(orderId, borrower, borrowAmount, vars.orderBorrowsNew, vars.
        totalBorrowsNew);
}
```

Listing 3.1: `XToken::borrowInternal()`

While 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`, it is important to take precautions to thwart possible `re-entrancy`. The similar issue is also present in the `redeemInternal()` function in `XToken`.

Another example of the `checks-effects-interactions` principle violation is the `withdrawNFT()` function (see the code snippet below). It externally calls the `NFT` contract to transfer the `NFT` asset back to the pledger. However, the interaction with the external contract (lines 246 and 262) starts before effecting the update on internal states (lines 264), hence violating the principle. In this particular case, the external contract (the `NFT` contract) is capable of launching `re-entrancy` to the `XToken::borrow()` function to borrow tokens after the `NFT` is withdrawn. A similar issue is also present in the `XNFT::notifyRepayBorrow()` function in the `XNFT` contract.

```
231  function withdrawNFT(uint256 orderId) external nonReentrant whenNotPaused(2){
232      LiquidatedOrder storage liquidatedOrder = allLiquidatedOrder[orderId];
233      Order storage _order = allOrders[orderId];
234      if(isOrderLiquidated(orderId)){
235          require(liquidatedOrder.auctionWinner == address(0), "the order has been
             withdrawn");
236          require(!allLiquidatedOrder[orderId].isPledgeRedeem, "redeemed by the pledgor");
237          CollectionNFT memory collectionNFT = collectionWhiteList[_order.collection];
238          uint256 auctionDuration;
239          if(collectionNFT.auctionDuration != 0){
```

```
            auctionDuration = collectionNFT.auctionDuration;
        }else{
            auctionDuration = auctionDurationOverAll;
        }
        require(block.timestamp > liquidatedOrder.liquidatedStartTime.add(auctionDuration), "the auction is not yet closed");
        require(msg.sender == liquidatedOrder.auctionAccount  (liquidatedOrder.auctionAccount == address(0) && msg.sender == liquidatedOrder.liquidator), "you can't extract NFT");
        transferNftInternal(address(this), msg.sender, _order.collection, _order.tokenId, _order.nftType);
        if(msg.sender == liquidatedOrder.auctionAccount && liquidatedOrder.auctionPrice != 0){
            uint256 profit = liquidatedOrder.auctionPrice.sub(liquidatedOrder.liquidatedPrice);
            uint256 compensatePledgerAmount = profit.mul(compensatePledgerRate).div(1e18);
            doTransferOut(liquidatedOrder.xToken, payable(_order.pledger), compensatePledgerAmount);
            uint256 liquidatorAmount = profit.mul(rewardFirstRate).div(1e18);
            doTransferOut(liquidatedOrder.xToken, payable(liquidatedOrder.liquidator), liquidatorAmount);

            addUpIncomeMap[liquidatedOrder.xToken] = addUpIncomeMap[liquidatedOrder.xToken] + (profit - compensatePledgerAmount - liquidatorAmount);
        }
        liquidatedOrder.auctionWinner = msg.sender;
    }else{
        require(!_order.isWithdraw, "the order has been drawn");
        require(_order.pledger != address(0) && msg.sender == _order.pledger, "withdraw auth failed");
        uint256 borrowBalance = controller.getOrderBorrowBalanceCurrent(orderId);
        require(borrowBalance == 0, "order has debt");
        transferNftInternal(address(this), _order.pledger, _order.collection, _order.tokenId, _order.nftType);
    }
    _order.isWithdraw = true;
    emit WithDraw(_order.collection, _order.tokenId, orderId, _order.pledger, msg.sender);
}
```

Listing 3.2: `XNFT::withdrawNFT()`

From another perspective, the current mitigation in applying money-market-level `re-entrancy` protection in `XToken/XNFT` can be strengthened by elevating the `re-entrancy` protection at the `P2Controller`-level. In addition, each individual function can be self-strengthened by following the `checks-effects-interactions` principle.

Recommendation Apply necessary `re-entrancy` prevention by following the `checks-effects-interactions` principle and consider strengthening the `re-entrancy` protection at the protocol-level instead of at the current money-market granularity.

Status This issue has been fixed in the following commits: `ad307fe` and `ba08254`

3.2 Potential DoS Against auction()

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

Description

The XNFT contract provides an `auction()` routine for users to bid for the auction of the liquidated order. While examining the current auction logic, we notice the existence of potential DoS (denial-of-service) that needs to be avoided in the implementation.

To elaborate, we show below the implementation of the `auction()` routine. It will repay the liquidation price back to the liquidator or repay the auction price back to the previous bidder. However, it comes to our attention that the `auction()` routine may always revert if the underlying token is ETH and the liquidator or the previous bidder refuses to receive ETH. As a result, the liquidator or the previous bidder will finally win the auction and withdraw the NFT after the auction.

```
    function auction(uint256 orderId, uint256 amount) payable external nonReentrant whenNotPaused(3){
        require(isOrderLiquidated(orderId), "this order is not a liquidation order");
        LiquidatedOrder storage liquidatedOrder = allLiquidatedOrder[orderId];
        require(liquidatedOrder.auctionWinner == address(0), "the order has been withdrawn");
        require(!liquidatedOrder.isPledgeRedeem, "redeemed by the pledgor");
        Order storage _order = allOrders[orderId];
        if(IXToken(liquidatedOrder.xToken).underlying() == ADDRESS_ETH){
            amount = msg.value;
        }
        uint256 price;
        if(liquidatedOrder.auctionAccount == address(0)){
            price = liquidatedOrder.liquidatedPrice;
        }else{
            price = liquidatedOrder.auctionPrice;
        }

        bool isPledger = auctionAllowed(_order.pledger, msg.sender, _order.collection, liquidatedOrder.liquidatedStartTime, price, amount);

        if(isPledger){
            uint256 fine = price.mul(pledgerFineRate).div(1e18);
```

```
            uint256 _amount = liquidatedOrder.liquidatedPrice.add(fine); // Luck: price.add(fine) or possible _amount < price?
            doTransferIn(liquidatedOrder.xToken, payable(msg.sender), _amount);
            uint256 rewardFirst = fine.mul(rewardFirstRate).div(1e18);
            if(liquidatedOrder.auctionAccount != address(0)){
                doTransferOut(liquidatedOrder.xToken, payable(liquidatedOrder.liquidator), rewardFirst);
                uint256 rewardLast = fine.mul(rewardLastRate).div(1e18);
                doTransferOut(liquidatedOrder.xToken, payable(liquidatedOrder.auctionAccount), (rewardLast + liquidatedOrder.auctionPrice));

                addUpIncomeMap[liquidatedOrder.xToken] = addUpIncomeMap[liquidatedOrder.xToken] + (fine - rewardFirst - rewardLast);
            }else{
                doTransferOut(liquidatedOrder.xToken, payable(liquidatedOrder.liquidator), (liquidatedOrder.liquidatedPrice + rewardFirst));

                addUpIncomeMap[liquidatedOrder.xToken] = addUpIncomeMap[liquidatedOrder.xToken] + (fine - rewardFirst);
            }
            transferNftInternal(address(this), msg.sender, _order.collection, _order.tokenId, _order.nftType);
            _order.isWithdraw = true;
            liquidatedOrder.isPledgeRedeem = true;
            liquidatedOrder.auctionWinner = msg.sender;
            liquidatedOrder.auctionAccount = msg.sender;
            liquidatedOrder.auctionPrice = _amount;

            emit AuctionNFT(orderId, liquidatedOrder.xToken, msg.sender, amount, true);
            emit WithDraw(_order.collection, _order.tokenId, orderId, _order.pledger, msg.sender);
        }else{
            doTransferIn(liquidatedOrder.xToken, payable(msg.sender), amount);
            if(liquidatedOrder.auctionAccount == address(0)){
                doTransferOut(liquidatedOrder.xToken, payable(liquidatedOrder.liquidator), liquidatedOrder.liquidatedPrice); // Luck: if the XToken is ETH market, a malicious user may block new auction by refusing to accept ETH
            }else{
                doTransferOut(liquidatedOrder.xToken, payable(liquidatedOrder.auctionAccount), liquidatedOrder.auctionPrice);
            }

            liquidatedOrder.auctionAccount = msg.sender;
            liquidatedOrder.auctionPrice = amount;

            emit AuctionNFT(orderId, liquidatedOrder.xToken, msg.sender, amount, false);
        }
    }
```

Listing 3.3: `XNFT::auction()`

```
    function doTransferOut(address xToken, address payable account, uint256 amount)
        internal{
        if(amount == 0) return;
        if (IXToken(xToken).underlying() != ADDRESS_ETH) {
            IERC20(IXToken(xToken).underlying()).safeTransfer(account, amount);
        } else {
            account.transfer(amount);
        }
    }
```

Listing 3.4: `XNFT::doTransferOut()`

Note the same issue exists in the `XNFT::withdrawNFT()` routine.

Recommendation Avoid the above denial-of-service risk in the above `auction()/withdrawNFT()` routines.

Status This issue has been mitigated in the following commit by introducing an admin controlled configuration that indicates whether the liquited order supports the auctioneer as a contract or not: `6c5cbdf`.

3.3 Proper whenNotPaused(2) Check Before NFT Withdrawal

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `XNFT`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In the `XNFT` contract, users can pledge and withdraw their NFT assets. The pledging and withdrawal can be paused by the admin via the `setPause()` routine. While examining the pause control of the withdrawal, we notice the occasions where user can withdraw their NFT assets even when the withdrawal is paused.

To elaborate, we show below code snippets from the `XNFT` contract. As the name indicates, the `notifyOrderLiquidated()` is invoked when the order is liquidated from the `XToken`. However, if the liquidator is the order pledger (line 288-295), the `NFT` will be withdrawn but the withdrawal pause control is not considered. And the `notifyRepayBorrow()` is invoked from the `XToken::repayBorrowAndClaim()` routine where the pledger repays the borrow and claims the pledged `NFT`. This is also a special case of withdrawal, however it does not take the withdrawal pause control into consideration. Both of the two occasions will lead to the `NFT` withdrawal even when the withdrawal is paused.

```
// 1 pledge, 2 withdraw, 3 auction
function setPause(uint256 index, bool isPause) external onlyAdmin{
    pausedMap[index] = isPause;
}
```

Listing 3.5: `XNFT::setPause`

```
    function notifyOrderLiquidated(address xToken, uint256 orderId, address liquidator,
        uint256 liquidatedPrice) external{
        require(msg.sender == address(controller), "auth failed");
        require(liquidatedPrice > 0, "invalid liquidate price");
        LiquidatedOrder storage liquidatedOrder = allLiquidatedOrder[orderId];
        require(liquidatedOrder.liquidator == address(0), "order has been liquidated");

        liquidatedOrder.liquidatedPrice = liquidatedPrice;
        liquidatedOrder.liquidator = liquidator;
        liquidatedOrder.xToken = xToken;
        liquidatedOrder.liquidatedStartTime = block.timestamp;

        Order storage order = allOrders[orderId];
        if(liquidator == order.pledger){
            liquidatedOrder.auctionWinner = liquidator;
            liquidatedOrder.isPledgeRedeem = true;
            order.isWithdraw = true;
            transferNftInternal(address(this), order.pledger, order.collection, order.
                tokenId, order.nftType);

            emit WithDraw(order.collection, order.tokenId, orderId, order.pledger,
                liquidatedOrder.auctionWinner);
        }
    }

    function notifyRepayBorrow(uint256 orderId) external{
        require(msg.sender == address(controller), "auth failed");
        require(!isOrderLiquidated(orderId), "withdrawal is not allowed for this order")
            ;
        Order storage _order = allOrders[orderId];
        require(tx.origin == _order.pledger, "you are not pledgor"); // Luck: pledger
            can't be contract? EOA->contract A->pledge/borrow/repayBorrow()
        require(!_order.isWithdraw, "the order has been drawn");
        transferNftInternal(address(this), _order.pledger, _order.collection, _order.
            tokenId, _order.nftType);
        _order.isWithdraw = true;

        emit WithDraw(_order.collection, _order.tokenId, orderId, _order.pledger, _order
            .pledger);
    }
```

Listing 3.6: `XNFT.sol`

Recommendation Properly apply the `whenNotPaused(2)` check for all occasions of `NFT` withdrawal.

Status This issue has been fixed in the following commit: `9e2534d`.

3.4 Tightened Access Control In borrowAllowed()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `P2Controller`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In the `XCarnival` protocol, the `P2Controller` contract is the protocol controller of all the protocol services. For example, it controls whether a user borrow operation is allowed or not. While examining the logic to validate the borrow operation, we notice the need of tightening the access control to the `borrowAllowed()` routine.

To elaborate, we show below the code implementation of the `borrowAllowed()` routine. It validates the input parameters with current pool states, and return if the borrow is allowed or revert if it is not allowed. Specially, there is a state variable `orderDebtStates[orderId]` that records the allowed borrow pool for the given order. If the input `xToken` doesn't equal to the `orderDebtStates[orderId]`, the borrow operation is refused, meaning the pledger can borrow from no more than one pool for the given order. The `orderDebtStates[orderId]` is updated (line 90-92) at the end of the `borrowAllowed()` during the first time the pledger borrows for the order. However, it comes to our attention that the `borrowAllowed()` is public accessible, which may lead to the `orderDebtStates[orderId]` being faked by malicious user and the pledger can not borrow from other pool for its order any more.

```
61    function borrowAllowed(address xToken, uint256 orderId, address borrower, uint256
          borrowAmount) external whenNotPaused(xToken, 3){
62        require(poolStates[xToken].isListed, "xToken not listed");
63
64        address _collection = orderAllowed(orderId, borrower);
65
66        // (address _collection , , ,) = xNFT.getOrderDetail(orderId);
67
68        CollateralState storage _collateralState = collateralStates[_collection];
69        require(_collateralState.isListed, "collection not exist");
70        require(_collateralState.supportPools[xToken]  _collateralState.
              isSupportAllPools, "collection don't support this pool");
71
72        address _lastXToken = orderDebtStates[orderId];
73        require(_lastXToken == address(0)  _lastXToken == xToken, "only support
              borrowing of one xToken");
74
```

```
        (uint256 _price, bool valid) = oracle.getPrice(_collection, IXToken(xToken).underlying());
        require(_price > 0 && valid, "price is not valid");

        // Borrow cap of 0 corresponds to unlimited borrowing
        if (poolStates[xToken].borrowCap != 0) {
            require(IXToken(xToken).totalBorrows().add(borrowAmount) < poolStates[xToken].borrowCap, "pool borrow cap reached");
        }

        uint256 _maxBorrow = mulScalarTruncate(_price, _collateralState.collateralFactor);
        uint256 _mayBorrowed = borrowAmount;
        if (_lastXToken != address(0)){
            _mayBorrowed = IXToken(_lastXToken).borrowBalanceStored(orderId).add(borrowAmount);
        }
        require(_mayBorrowed <= _maxBorrow, "borrow amount exceed");

        if (_lastXToken == address(0)){
            orderDebtStates[orderId] = xToken;
        }
    }
```

Listing 3.7: P2Controller::borrowAllowed()

Our analysis shows that the `borrowAllowed()` shall be restricted to be called only from the `XToken` contract.

Recommendation Tighten the access control policy on the above-mentioned `borrowAllowed()` routine, so that it can only be accessed from the `XToken`.

Status This issue has been fixed in the following commit: `4c6b31d`.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `XCarnival` protocol, there is a privileged account (i.e. `admin`) that plays a critical role in governing and regulating the system-wide operations (e.g., withdraw auction income, update user borrow/supply amount for liquidity mining, etc.). Our analysis shows that the privileged account needs to be scrutinized. In the following, we use the `XNFT/LiquidityMining` contracts as examples to show the representative functions potentially affected by the privileges of the privileged account.

Specifically, the privileged functions in the `XNFT` contract allow for the `admin` to set new `controller`, set new `xAirDrop` and withdraw assets (may include user assets) from the contract, etc.

```
function setController(address _controller) external onlyAdmin{
  controller = IP2Controller(_controller);
}

function withdraw(address xToken, uint256 amount) external onlyAdmin{
  doTransferOut(xToken, payable(admin), amount);
}

function setXAirDrop(IXAirDrop _xAirDrop) external onlyAdmin{
  xAirDrop = _xAirDrop;
}
```

Listing 3.8: Example Privileged Operations in the `XNFT` Contract

And the privileged functions in the `LiquidityMining` contract allow for the `admin` to update users borrow/supply amount in the `XToken`, which impacts the shares of the rewards distribution for liquidity mining.

```
modifier onlyController() {
  require(msg.sender == controller  msg.sender == admin, "require controller auth");
  _;
}

function updateBorrow(address xToken, address collection, uint256 amount, address account, uint256 orderId, bool isDeposit) external onlyController nonReentrant{
  if(wAddressToBaseAddressMap[collection] != address(0x0)){
      collection = wAddressToBaseAddressMap[collection];
  }
  PoolInfo storage poolInfo = borrowPoolInfoMap[xToken][collection];
  if(poolInfo.xToken == address(0)) return;
  UserInfo storage user = borrowUserInfoMap[xToken][collection][account];
  if(!isDeposit && user.amount == 0) return;
  updatePool(xToken, collection, true);
  if((isDeposit && user.amount > 0)  !isDeposit){
      uint256 pending = user.amount.mul(poolInfo.accPerShare).div(1e18).sub(user.rewardDebt);
      user.rewardToClaim = user.rewardToClaim.add(pending);
  }
  poolInfo.amount = poolInfo.amount.sub(user.orders[orderId]).add(amount);
  user.amount = user.amount.sub(user.orders[orderId]).add(amount);
  user.rewardDebt = user.amount.mul(poolInfo.accPerShare).div(1e18);
  user.orders[orderId] = amount;
}

function updateSupply(address xToken, uint256 amount, address account, bool isDeposit) external onlyController nonReentrant{
  PoolInfo storage poolInfo = supplyPoolInfoMap[xToken];
```

```
    if(poolInfo.xToken == address(0)) return;
    UserInfo storage user = supplyUserInfoMap[xToken][account];
    if(!isDeposit && user.amount == 0) return;
    updatePool(xToken, address(0), false);
    if((isDeposit && user.amount > 0)  !isDeposit){
        uint256 pending = user.amount.mul(poolInfo.accPerShare).div(1e18).sub(user.
            rewardDebt);
        user.rewardToClaim = user.rewardToClaim.add(pending);
    }
    poolInfo.amount = poolInfo.amount.sub(user.amount).add(amount);
    user.amount = amount;
    user.rewardDebt = user.amount.mul(poolInfo.accPerShare).div(1e18);
  }
```

Listing 3.9: Example Privileged Operations in the `LiquidityMining` Contract

There are still other privileged functions not listed here. Notice that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the `admin` 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.

Recommendation Making the above privileges explicit among protocol users.

Status This issue has been confirmed by the team.

3.6 Timely massUpdatePools() During Pool Updates

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `LiquidityMining`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `LiquidityMining` contract provides an incentive mechanism that rewards the borrowing and supplying in the `XToken` with the configured `erc20Token`. The rewards are carried out by designating a borrow pool and a supply pool for each `XToken`. And the users are rewarded in proportional to their shares of liquidity in each reward pool.

The reward pools can be dynamically added via `addPool()` and the weights of borrow pools can be adjusted via `setPool()`. When analyzing the pool weight update routine `setPool()`, we notice the need of timely invoking the `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
function setPool(address xToken, address collection, uint256 allocPoint, bool isBorrow
    ) external onlyAdmin{
  if(isBorrow){
      PoolInfo storage poolInfo = borrowPoolInfoMap[xToken][collection];
      require(poolInfo.xToken != address(0), "pool not exists!");

      borrowTotalAllocPoint = borrowTotalAllocPoint.sub(poolInfo.allocPoint).add(
          allocPoint);
      poolInfo.xToken = xToken;
      poolInfo.collection = collection;
      poolInfo.allocPoint = allocPoint;
  }else{
      PoolInfo storage poolInfo = supplyPoolInfoMap[xToken];
      require(poolInfo.xToken != address(0), "pool not exists!");
      poolInfo.xToken = xToken;
  }
}
```

Listing 3.10: LiquidityMining::setPool()

Similarly, the reward rates for the borrow pools and supply pools can be updated via the `setBorrowPerBlockReward()/setSupplyPerBlockRewardMap()` routines. There is also the need to timely invoking the `massUpdatePools()` to update the reward distribution before the new reward rates become effective.

```
function setBorrowPerBlockReward(uint256 _borrowPerBlockReward) external onlyAdmin{
    borrowPerBlockReward = _borrowPerBlockReward;
}

function setSupplyPerBlockRewardMap(address xToken, uint256 perBlockReward) external
    onlyAdmin{
    supplyPerBlockRewardMap[xToken] = perBlockReward;
}
```

Listing 3.11: LiquidityMining.sol

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, these interfaces are restricted to the admin (via the `onlyAdmin` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` before any pool's weight or reward rate is updated.

Status This issue has been confirmed. And the team clarified that: We will call function (`massUpdatePools()`) manually for the time being.

4 Conclusion

In this audit, we have analyzed the `XCarnival` protocol design and implementation. The `XCarnival` is an `NFT` lending protocol that lets users borrow tokens quickly without selling their `NFT`s. By supplying tokens into the `XToken`, depositors can earn interests and rewards. And the protocol essentially offers yields on `NFT` assets, with which users can receive airdrops by pledging their `NFT`s.

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-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.