SMART CONTRACT AUDIT REPORT

for

SwopX Protocol

Prepared By: Yiqun Chen

PeckShield

January 20, 2022

Document Properties

Client	SwopX Protocol
Title	Smart Contract Audit Report
Target	SwopX
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 20, 2022	Xuxian Jiang	Final Release
1.0-rc1	December 18, 2021	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **SwopX** design document and related smart contract source code, 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

`SwopX` protocol has an asset token (`SwopX721`) and a utility token (`SwopX20`). Users can mint `NFT` tokens for selling or swapping with other `NFTs`. During the sale, if there is a match, both seller and buyer get rewards. The protocol also has a time-bound library that cycles every 30 days for claiming the rewards. In addition, the protocol has a `SwapPlace` contract that interacts with the `SushiSwap DEX` to obtain the USD price of the `SwopX20` utility token.

The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of SwopX

Item	Description
Name	SwopX Protocol
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 20, 2022

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

- https://github.com/pcanwar/swap.git (fbd544a)

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 (98e6739)

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

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 `SwopX` 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	2	■■
Medium	2	■■
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 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 high-severity vulnerabilities, 2 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key SwopX Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic in SwopX::redeemMint()	Business Logic	Fixed
PVE-002	Medium	Revisited Design in SwopX::createTokenForSwopX()	Business Logic	Fixed
PVE-003	High	Price Manipulation in SwopXUtil	Time and State	Fixed
PVE-004	High	Incorrect Logic in SwopXStaking::calculatePendingStake()	Business Logic	Fixed
PVE-005	Medium	Incorrect Logic in SwopXPlace::buyItemByToken()	Business Logic	Fixed
PVE-006	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.

3 | Detailed Results

3.1 Improved Logic in SwopX::redeemMint()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `SwopX`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [3]

Description

The `SwopX` protocol allows to buy an `NFT` with a so-called `lazy mint` mechanism. With this mechanism, the `NFT` will not be minted until it is purchased. While reviewing its logic, we notice the current implementation can be improved.

To elaborate, we show below the related `redeemMint()` function. It implements a rather straightforward logic in taking the payment to buy a freshly-minted `NFT`. However, it comes to our attention the payment requires the contract to approve itself with the `offered price`: `WETH.safeApprove(address (this), offeredPrice)` (line 133). While the intention here is to approve the payment, the actual approval needs to initiate from the buyer, instead of the contract itself. In other words, the buy still needs to approve `SwopX` contract for the payment amount!

```
117     function redeemMint(IERC20 _erc20Contract, address signer, uint256 price, uint256
            nonce,
118         uint offeredPrice,
119         bytes calldata signature) nonReentrant supportInterface(_erc20Contract)
120          external   {
121         IERC20 WETH = IERC20(_erc20Contract);
122 
123         //require(_hasRole(keccak256("LAZY_MINTER_ROLE"), signer) == true, "N1");
124         // uint _price = ISwopXUti(IswopXUti).getETHPrice( price);
125         uint callItFee = price * 100;
126         uint fee = callItFee / 1e4;
127         uint payment =  price + fee ;
128         require(offerdPrice >= payment, "A1");
```

```
        require(signer != msg.sender, "A2");
        require(WETH.balanceOf(msg.sender) >= offerdPrice, "A3");
        require(_verify(signer, _hash(price, nonce), signature), "N2");
        require(identifiedSignature[signature] == false, "R1");
        WETH.safeApprove(address(this), offerdPrice);
        WETH.safeTransferFrom(msg.sender, address(this), offerdPrice);
        address _receiver = receiverTo;
        uint tokenId = createTokenForSwopX(signer);
        uint ethPrice = ISwopXUti(IswopXUti).getETHPrice( offerdPrice);
        storaged(tokenId, price, ethPrice, false);
        _transfer(signer, msg.sender, tokenId);
        WETH.safeTransferFrom(address(this), _receiver, fee);
        WETH.safeTransferFrom(address(this), signer, offerdPrice - fee);

        identifiedSignature[signature] = true;
        emit RedeemMint(tokenId, signer, msg.sender, offerdPrice - fee, fee);

    }
```

Listing 3.1: `SwopX::redeemMint()`

Recommendation Properly revise the above `redeemMint()` routine to arrange the right payment.

Status The issue has been fixed by this commit: `68bdae9`.

3.2 Revisited Design in SwopX::createTokenForSwopX()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SwopX`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [3]

Description

The `lazy mint` mechanism requires the instant mint of a new `NFT`, which is facilitated with a helper routine `createTokenForSwopX()`. Our analysis on this helper routine shows it unnecessarily grants to the current `swopXPlace` contract the privilege to manage all of the caller's `NFTs` created in `SwopX`.

To elaborate, we show below the `createTokenForSwopX()` function. The issue stems from the `setApprovalForAll()` call (line 166), which approves the `swopXPlace` contract to manage the user's `NFTs`. Note that there is a privileged interface in place that allows the owner to change the current `swopXPlace` contract. And this design unnecessarily grants extra privileges from unknowing users.

```
159     function createTokenForSwopX(address account ) public returns (uint) {
160
161         require(account != address(0), "Z1");
```

```
        address _swopXAddress = swopXPlaceAddress;
        _tokenIdCounter.increment();
        uint256 newItemId = _tokenIdCounter.current();
        _safeMint(account, newItemId);
        setApprovalForAll(_swopXAddress, true);
        // _setTokenURI(newItemId, tokenURI_);

        return newItemId;

    }
```

Listing 3.2: `SwopX::createTokenForSwopX()`

Recommendation Revisit the above `createTokenForSwopX()` routine to better protect users in not requiring extra privileges.

Status The issue has been fixed by this commit: `f86b468`.

3.3 Price Manipulation in SwopXUtil

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: High
- Target: `SwopXUtil`
- Category: Time and State [6]
- CWE subcategory: CWE-663 [2]

Description

To facilitate the sale of `NFTs`, there is a constant need of swapping one asset to another in `SwopX`. Accordingly, the protocol has provided helper routines to facilitate the asset conversion: `_toUSD()`, `getUSDPrice()`, `getETHPrice()`, and `getMaticPrice()`.

```
    function toUSD() public view returns(uint){
        uint _amount = amount;
        IERC20Metadata SwopXtoken = IERC20Metadata(pair.token0());
        (uint usd, uint swopx,) = pair.getReserves();
        uint res  = swopx*(10 ** SwopXtoken.decimals());
        return((_amount * res )/usd);
    }


    function getUSDPrice(uint _amount) public view returns(uint)
      {

        IERC20Metadata tokenEthUsdc = IERC20Metadata(ethUsdc.token0());
        (uint usd, uint weth,) = ethUsdc.getReserves();
        uint res  = weth*(10 ** tokenEthUsdc.decimals());
```

```
        return((_amount * res )/usd); // return amount of ethereum needed to buy item.
      }


  // to set an item price to ETH.. WETH

    function getETHPrice( uint _amount) public view returns(uint)
      {

        IERC20Metadata tokenEthUsdc = IERC20Metadata(ethUsdc.token1());
        (uint usd  , uint weth,) = ethUsdc.getReserves();
        uint res  = usd*(10** tokenEthUsdc.decimals());
        return((_amount * res )/weth);
    }


    function getMaticPrice( uint _amount) public view returns(uint)
      {

        IERC20Metadata tokenMatic = IERC20Metadata(maticUsd.token1());
        (uint usd  , uint weth,) = maticUsd.getReserves();
        uint res  = usd*(10** tokenMatic.decimals());
        return((_amount * res )/weth);
    }
```

Listing 3.3: `StakingV2::deposit()`

To elaborate, we show above these helper routines. We notice the conversion is routed to `UniswapV2`-based DEXs and the related spot reserves are used to compute the price! Therefore, they are vulnerable to possible front-running attacks, resulting in possible loss for the token conversion.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of farming users.

Status This issue has been fixed in the following commits: `4f0060d` and `a7691a8`.

3.4 Incorrect Logic in SwopXStaking::calculatePendingStake()

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: `SwopXStaking`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [3]

Description

To engage protocol users, the protocol has a built-in staking contract `SwopXStaking` that provides rewards to staking users. And this staking contract keeps track of the current staking amount for each user and provides corresponding rewards. Our analysis on this staking contract shows the current accounting scheme is flawed.

In the following, we show below the related `setAccount()` routine. This routine will be invoked for every single deposit, including the new user's first deposit. Based on the implementation, it makes user of another routine `calculatePendingStake()` to compute the latest pending rewards. However, the `calculatePendingStake()` routine imposes three requirements: (I) `require(_holder != address(0)` (line 87), (II) `require(!holders.contains(_holder)` (line 88), and (III) `require(balance[_holder] <= 0)` (line 89). While first requirement is reasonable, the second and the third requirement may unnecessarily block legitimate deposits!

```
function setAccount(address account) private {
        require( account != address(0) , "zero address");

        uint pendingBalance = calculatePendingStake(account);
        if (pendingBalance > 0) {
            uint stakedAmount = balance[account];
            stakedAmount += pendingBalance;
            balance[account] = stakedAmount;
            uint _totalRewards = totalRewards;
            _totalRewards += stakedAmount;
            totalRewards = _totalRewards;
            emit Transferred(account, pendingBalance);
        }
        stakingClaimedTime[account] = block.timestamp;
    }
```

Listing 3.4: `SwopXStaking::setAccount()`

```
86      function calculatePendingStake(address _holder) public view returns (uint) {
87          require( _holder != address(0) , "Zero Address");
88          require( !holders.contains(_holder) , "Non_Holders");
89          require( balance[_holder] <= 0 , "No_Balance");
```

```
        uint calculateTimeDiff = block.timestamp - stakingClaimedTime[_holder];
        uint stakedAmount = balance[_holder];
        uint pendingBalanceAfterStaking ;
        uint _amount = stakedAmount * rewardRate * calculateTimeDiff;
        pendingBalanceAfterStaking = (_amount/ rewardInterval) / 1e4;

        return pendingBalanceAfterStaking;
    }
```

Listing 3.5: `SwopXStaking::calculatePendingStake()`

Recommendation Revisit the above deposit/reward logic to prevent legitimate users from being blocked.

Status This issue has been fixed in the following commits: `68bdae9` and `62eac5e`.

3.5 Incorrect Logic in SwopXPlace::buyItemByToken()

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SwopXPlace`
- Category: Business Logic [5]
- CWE subcategory: CWE-837 [3]

Description

The `SwopX` protocol has another core contract i.e., `SwopXPlace`, that allows for making purchases or swaps. While analyzing this contract, we notice a key function `buyItemByToken()` whose logic can be improved.

To elaborate, we show below the `buyItemByToken()` function. As the name indicates, this function is used to purchase a specific `NFT` with the given payment token and amount. It comes to our attention that the item value is computed as `itemValue = amount - feeCall` (line 369), which should be `itemValue = ethPrice - feeCall`. The `feePaid` is currently calculated as `amount - ethPrice` (line 370), which needs to be revised as `amount - ethPrice + feeCall`.

```
356    function buyItemByToken(IERC20 _erc20Contract, uint _itemId, uint amount) public
           nonReentrant supportInterface(_erc20Contract) {
357        require(amount > 0,"Zero_Amount");
358        // require(ISwopX(swopXAddress).isForSale(_itemId) == true,"Not_For_Sale");
359        IERC20 WETH = IERC20(_erc20Contract);

361        address  oldOwnered = IERC721(swopXAddress).ownerOf(_itemId);

363        uint usdPrice = ISwopX(swopXAddress).usdPrice(_itemId);
364        uint ethPrice = ISwopX(swopXAddress).ethPrice(_itemId);
```

```
        uint usdValue = ISwopXUti(SWOPXUTI).getUSDPrice(amount);

        uint feeCall = calculatedFee(_itemId);
        uint itemValue = amount - feeCall;
        uint feePaid =  amount - ethPrice ;

        require(itemValue >= ethPrice, "Invalid");
        require(usdValue >= usdPrice, "Invalid");

        uint currentAmountAfterFee = itemValue + feePaid;
        require(WETH.balanceOf(msg.sender) > currentAmountAfterFee, "NOT");
        require(currentAmountAfterFee >= amount, "NOT");

        tokenOwner[_itemId] = msg.sender;
        // IERC20(WETH).approve(oldOwnered, itemValue);
        // IERC20(WETH).approve(receiverTo, feePaid);
        WETH.safeApprove(address(this), currentAmountAfterFee);
        WETH.safeTransferFrom(msg.sender, address(this), currentAmountAfterFee);
        WETH.safeTransferFrom(address(this), receiverTo, feePaid);
        WETH.safeTransferFrom(address(this), oldOwnered, itemValue);

        // _safeTxBuyFrom(msg.sender, address(this), itemValue + feePaid);
        // _safeTxBuyFrom(address(this), oldOwnered, itemValue);
        // _safeTxBuyFrom(address(this), receiverTo, feePaid);
        IERC721(swopXAddress).transferFrom(oldOwnered, msg.sender, _itemId);
        emit BuyLog(oldOwnered, msg.sender, itemValue,_itemId, feePaid );
    }
```

Listing 3.6: `SwopXPlace::buyItemByToken()`

Moreover, the sanity checks with the queried `ethPrice` as well as the `currentAmountAfterFee` need to be accordingly adjusted.

Recommendation Revise the above `buyItemByToken()` logic to properly purchase the intended `NFT`.

Status This issue has been fixed in the following commits: `68bdae9` and `62eac5e`.

3.6 Redundant State/Code Removal

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [4]
- CWE subcategory: CWE-563 [1]

Description

The Swopx protocol makes good use of a number of reference contracts, such as ERC20, SafeERC20, SafeMath, and ERC721, to facilitate its code implementation and organization. For example, the SwopXPlace 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 SwopXPlaceStorage contract, there are a number of storage states that are defined, but not used. Examples include the following states, i.e., isAlreadyMacthed, _allTokens, and addressToRewardMile.

```
    mapping(address => mapping(address => bool)) internal _iMatched;
    mapping(uint256 => mapping(uint256 => bool)) internal _isMatched;

    mapping(uint => bool) internal isAlreadyMacthed;
    //mapping(uint => uint) internal served;

    // mapping(address => uint256) internal totalMatch;
    mapping(address => uint256) internal _allTokens;

    mapping(address => mapping (uint => rewardMileStone)) public addressToRewardMile;
```

Listing 3.7: The SwopXPlaceStorage Contract

In addition, the SwopXControl contract defines an unused state achievedHash, which can be safely removed. The library contract SetMileStone makes use of the local variable reTimer, which can removed as well.

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The issue has been fixed by this commit: 68bdae9.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `SwopX` protocol, which has an asset token `SwopX721` and a utility token `SwopX20`. Users can mint `NFT` tokens for selling or swapping with other `NFTs`. 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.

References

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

[2] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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