SMART CONTRACT AUDIT REPORT

for

Swell Network

PeckShield

Prepared By: Patrick Liu

Hangzhou, China
March 11, 2022

Document Properties

Client	Swell Network
Title	Smart Contract Audit Report
Target	Swell Network
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 11, 2022	Xiaotao Wu	Final Release
1.0-rc	March 3, 2022	Xiaotao Wu	Release Candidate

PeckShield

Contact

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

Name	Patrick Liu
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 `Swell Network` 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 is well designed and engineered, though it can be further improved by addressing our suggestions. This document outlines our audit results.

1.1 About Swell Network

`Swell Network` aims to build a decentralized, open, and liquid `Ethereum` 2 (`ETH2`) staking as service (SAS), that will facilitate rewards in a transparent manner via micro pool staking, unlock staked `ether` by providing a liquid derivative token, and enable powerful flexibility for users to compound yield across the `DeFi` landscape. The basic information of the audited protocol is as follows:

PeckShield

Table 1.1: Basic Information of Swell Network

Item	Description
Name	Swell Network
Website	https://www.swellnetwork.io/
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 11, 2022

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

- https://github.com/SwellNetwork/v2-core.git (6bb73be)

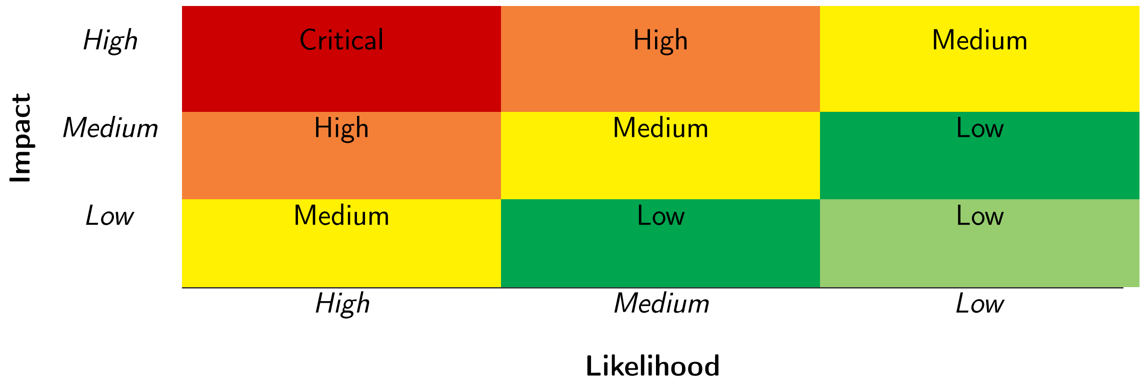
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/SwellNetwork/v2-core.git (7cf206f)

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

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 is considered safe regarding the check item. For any discovered issue, we might further deploy

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

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 design and implementation of the `Swell Network` protocol smart contracts. 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	1	🟦
Informational	1	🟦
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 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key Swell Network Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Reentrancy Protection In SWNFTUpgrade	Time and State	Resolved
PVE-002	Informational	Meaningful Events For Important State Changes	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

PeckShield

3 Detailed Results

3.1 Improved Reentrancy Protection In SWNFTUpgrade

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SWNFTUpgrade`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

By design, the `SWNFTUpgrade` contract is the main entry for interaction with users. In particular, one routine, i.e., `stake()`, is designed to deposit `ETH` into the contract and mint `NFT` token to represent the deposit share.

To elaborate, we show below the code snippet of the `stake()` function. In the function, the `_safeMint()` function is called (line 136) to mint an `ERC721` token for the user. A further examination of `_safeMint()` of `ERC721` shows the `_checkOnERC721Received()` function will be called to ensure the recipient confirms the receipt. If the recipient is an evil attacker, he may launch a re-entrancy attack in the callback function. So far, we also do not know how an attacker can exploit this issue to earn profit. After internal discussion, we consider it is necessary to bring this issue up to the team. Though the implementation of the `stake()` function is well designed and meets the `Checks-Effects-Interactions` pattern, we may intend to use the `ReentrancyGuard::nonReentrant` modifier to protect the `stake()` function at the whole protocol level.

PeckShield

```
122     /// @notice Deposit ETH into official contract
123     /// @param pubKey The public key of the validatator
124     /// @param signature The signature of the withdrawal
125     /// @param depositDataRoot The root of the deposit data
126     /// @return newItemId The token ID of the new token
127     function stake(
128         bytes calldata pubKey,
129         bytes calldata signature,
130         bytes32 depositDataRoot
```

```
    ) external payable returns (uint256 newItemId) {
        require(msg.value >= 1 ether, "Must send at least 1 ETH");
        require(msg.value % ETHER == 0, "stake value not multiple of Ether");
        ...

        _safeMint(msg.sender, newItemId);
        ISWETH(swETHAddress).mint(msg.value);

        positions[newItemId] = Position(
            pubKey,
            msg.value,
            msg.value
        );

        emit LogStake(msg.sender, newItemId, pubKey, msg.value);
    }
```

Listing 3.1: `SWNFTUpgrade::stake()`

Recommendation Apply the non-reentrancy protection in above-mentioned routine.

Status This issue has been fixed by applying the `Check-Effects-Interactions` design pattern: `7cf206f`.

3.2 Meaningful Events For Important State Changes

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A

- Target: `SWNFTUpgrade`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

PeckShield

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `SWNFTUpgrade` contract as an example. While examining the event that reflect the `SWNFTUpgrade` dynamics, we notice there is a lack of emitting related event to reflect important state change. Specifically, when the `setswETHAddress()` is being called, there is no corresponding event being emitted to reflect the occurrence of `setswETHAddress()`.

```
    /// @notice set base token address
    /// @param _swETHAddress The address of the base token
    function setswETHAddress(address _swETHAddress) onlyOwner external {
        require(_swETHAddress != address(0), "Address cannot be 0");
        swETHAddress = _swETHAddress;
    }
```

Listing 3.2: `SWNFTUpgrade::setswETHAddress()`

Recommendation Properly emit the related event when the above-mentioned function is being invoked.

Status This issue has been fixed in the following commit: `51dea8f`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `SWNFTUpgrade/SWDAO`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `Swell Network` protocol, there are two privileged accounts, i.e., `owner`, and `minter`. These accounts play a critical role in governing and regulating the protocol-wide operations (e.g., mint more `swDAO` tokens into circulation or burn `swDAO` tokens from circulation, set base token address, add a new strategy, remove a strategy, and add a new validator into `whiteList`, etc.).

In the following, we use the `SWNFTUpgrade` contract as an example and show the representative functions potentially affected by the privileges of the `owner` accounts.

```
    /// @notice set base token address
    /// @param _swETHAddress The address of the base token
    function setswETHAddress(address _swETHAddress) onlyOwner external {
        require(_swETHAddress != address(0), "Address cannot be 0");
        swETHAddress = _swETHAddress;
    }

    /// @notice Add a new strategy
    /// @param strategy The strategy address to add
    function addStrategy(address strategy) onlyOwner external{
        require(strategy != address(0), "address cannot be 0");
        strategies.push(strategy);
        emit LogAddStrategy(strategy);
    }

```

```
    /// @notice Remove a strategy
    /// @param strategy The strategy index to remove
    function removeStrategy(uint strategy) onlyOwner external{
        require(strategies[strategy] != address(0), "strategy does not exist");
        uint length = strategies.length;
        address last = strategies[length-1];
        emit LogRemoveStrategy(strategy, strategies[strategy]);
        strategies[strategy] = last;
        strategies.pop();
    }

    /// @notice Add a new validator into whiteList
    /// @param pubKey The public key of the validator
    function addWhiteList(bytes calldata pubKey) onlyOwner external{
        whiteList[pubKey] = true;
        emit LogAddWhiteList(msg.sender, pubKey);
    }
```

Listing 3.3: `SWNFTUpgrade::setswETHAddress()/addStrategy()/removeStrategy()/addWhiteList()`

The first function `setswETHAddress()` allows for the `owner` to set the base token address (i.e., `swETHAddress`) for the `SWNFTUpgrade` contract. The second and third functions `addStrategy()/removeStrategy()` allow for the `owner` to add or remove a strategy for the `SWNFTUpgrade` contract. And the fourth function `addWhiteList()` allows for the `owner` to add a new validator into `whiteList`. Note if an existing strategy is deleted by the `owner`, users who have entered this strategy will unable to withdraw their base token from the strategy contract, thus suffer asset loss.

```
    /// @notice Exit strategy for a token
    /// @param tokenId The token ID
    /// @param strategy The strategy index to enter
    /// @return amount The amount of swETH withdrawn
    function exitStrategy(uint tokenId, uint strategy) public returns (uint amount){
        require(_exists(tokenId), "Query for nonexistent token");
        require(strategies[strategy] != address(0), "strategy does not exist");
        require(ownerOf(tokenId) == msg.sender, "Only owner can exit strategy");
        amount = IStrategy(strategies[strategy]).exit(tokenId);
        positions[tokenId].baseTokenBalance += amount;
        emit LogExitStrategy(
        tokenId,
        strategy,
        strategies[strategy],
        msg.sender,
        amount
        );
    }
```

Listing 3.4: `SWNFTUpgrade::exitStrategy()`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged accounts may also be a counter-party risk to the contract 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 Make the list of extra privileges granted to privileged accounts explicit to `Swell Network` protocol users.

Status This issue has been confirmed. The `Swell Network` team confirms that they will use protocol `DAO` multisig for the `owner` on deployment.

PeckShield

4 | Conclusion

In this audit, we have analyzed the `Swell Network` design and implementation. The system presents a unique, robust offering as a decentralized money market protocol with both secure lending and synthetic stablecoins. The audited `Swell Network` aims to build a decentralized, open, and liquid `Ethereum` staking solution. 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.

PeckShield

References

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

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

[5] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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