SMART CONTRACT AUDIT REPORT

for

SpiritV2 Protocol

Prepared By: Xiaomi Huang

PeckShield

August 8, 2022

Document Properties

Client	SpiritV2 Protocol
Title	Smart Contract Audit Report
Target	SpiritV2
Version	1.0
Author	Xuxian Jiang
Auditors	Xiaotao Wu, Patrick Liu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 8, 2022	Xuxian Jiang	Final Release
1.0-rc	July 23, 2022	Xuxian Jiang	Release Candidate #1

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 source code of the `SpiritV2` 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 SpiritV2

`SpiritV2` is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. The protocol is forked from the `solidly AMM` with the new curve ($x^3y + xy^3 = k$) for efficient stable swaps. However, it removes the built-in `NFT`-based voting mechanism and modifies the fee structure to meet its own tokenomics. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of SpiritV2

Item	Description
Name	SpiritV2 Protocol
Website	https://www.spiritswap.finance/
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	August 8, 2022

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

- https://github.com/Heesho/SpiritV2.git (66f082d)

1.2 About PeckShield

PeckShield Inc. [12] 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 accordingly classified into four categories, 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 deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

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

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. 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 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 design and implementation of the `SpiritV2` 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 logics, 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	2	■■
Informational	2	■■
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 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Voting Amplification With Sybil Attacks	Business Logic	Resolved
PVE-002	Low	Possible Denial-Of-Service in SpiritMasterChef	Coding Practices	Confirmed
PVE-003	Low	Removal of Unused State in Gauges	Coding Practices	Confirmed
PVE-004	Medium	Improper Funding Source In inSpirit::deposit_for()	Business Logic	Confirmed
PVE-005	Informational	Suggested immutable Use in BaseV1Pair	Coding Practices	Confirmed
PVE-006	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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.

3 | Detailed Results

3.1 Voting Amplification With Sybil Attacks

- ID: PVE-001
- Severity: Informational
- Likelihood: High
- Impact: N/A
- Target: `SpiritToken`
- Category: Business Logics [8]
- CWE subcategory: CWE-841 [5]

Description

In `SpiritV2`, the protocol `SPIRIT` token contract is enhanced with voting support so that it can be used to cast and record the votes. Moreover, the `SPIRIT` contract allows for dynamic delegation of a voter to another, though the delegation is not transitive.

Our analysis on the `SPIRIT` token contract shows that its voting use may be vulnerable to a new type of so-called `Sybil` attacks (credits go to Jong Seok Park[11]). For elaboration, let's assume at the very beginning there is a malicious actor named `Malice`, who owns 100 `SPIRIT` tokens. `Malice` has an accomplice named `Trudy` who currently has 0 balance of `SPIRIT`. This `Sybil` attack can be launched as follows:

```
    function _delegate(address delegator, address delegatee) internal {
        address currentDelegate = _delegates[delegator];
        uint256 delegatorBalance = balanceOf(delegator); // balance of underlying
            SPIRITs (not scaled);
        _delegates[delegator] = delegatee;

        emit DelegateChanged(delegator, currentDelegate, delegatee);

        _moveDelegates(currentDelegate, delegatee, delegatorBalance);
    }

    function _moveDelegates(
        address srcRep,
        address dstRep,
        uint256 amount
```

```
    ) internal {
        if (srcRep != dstRep && amount > 0) {
            if (srcRep != address(0)) {
                // decrease old representative
                uint32 srcRepNum = numCheckpoints[srcRep];
                uint256 srcRepOld = srcRepNum > 0
                    ? checkpoints[srcRep][srcRepNum - 1].votes
                    : 0;
                uint256 srcRepNew = srcRepOld.sub(amount);
                _writeCheckpoint(srcRep, srcRepNum, srcRepOld, srcRepNew);
            }

            if (dstRep != address(0)) {
                // increase new representative
                uint32 dstRepNum = numCheckpoints[dstRep];
                uint256 dstRepOld = dstRepNum > 0
                    ? checkpoints[dstRep][dstRepNum - 1].votes
                    : 0;
                uint256 dstRepNew = dstRepOld.add(amount);
                _writeCheckpoint(dstRep, dstRepNum, dstRepOld, dstRepNew);
            }
        }
    }
```

Listing 3.1: SpiritToken :: _delegate()

1. `Malice` initially delegates the voting to `Trudy`. Right after the initial delegation, `Trudy` can have 100 votes if he chooses to cast the vote.

2. `Malice` transfers the full 100 balance to M_1 who also delegates the voting to `Trudy`. Right after this delegation, `Trudy` can have 200 votes if he chooses to cast the vote. The reason is that the `SPIRIT` contract's `transfer()` does NOT `_moveDelegates()` together. In other words, even now `Malice` has 0 balance, the initial delegation (of `Malice`) to `Trudy` will not be affected, therefore `Trudy` still retains the voting power of 100 `SPIRIT`. When M_1 delegates to `Trudy`, since M_1 now has 100 `SPIRIT`, `Trudy` will get additional 100 votes, totaling 200 votes.

3. We can repeat by transferring M_i's 100 `SPIRIT` balance to M_{i+1} who also delegates the votes to `Trudy`. Every iteration will essentially add 100 voting power to `Trudy`. In other words, we can effectively amplify the voting powers of `Trudy` arbitrarily with new accounts created and iterated!

Recommendation To mitigate, it is necessary to accompany every single `transfer()` and `transferFrom()` with the `_moveDelegates()` so that the voting power of the sender's delegate will be moved to the destination's delegate. By doing so, we can effectively mitigate the above `Sybil` attacks.

Status This issue has been resolved as the team confirms that this feature is not used in the current ecosystem.

3.2 Possible Denial-Of-Service in SpiritMasterChef

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `SpiritMasterChef`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

The `SpiritV2` protocol provides an incentive mechanism that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool. While analyzing the incentive mechanism, we notice a privileged function allows for the update of a protocol-wide parameter, i.e., `spiritPerBlock`, without being checked.

Specifically, we show below the related function `SpiritMasterChef::updateEmissionRate()`. As the name indicates, this function updates the emission rate of the protocol tokens `SPIRIT`. And this function can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `spiritPerBlock` may revert every single operation behind `deposit`, `withdraw`, and `updatePool`, hence hurting the adoption of the protocol.

```
    //Pancake has to add hidden dummy pools inorder to alter the emission, here we make
        it simple and transparent to all.
    function updateEmissionRate(uint256 _spiritPerBlock) public onlyOwner {
        massUpdatePools();
        spiritPerBlock = _spiritPerBlock;
    }
```

Listing 3.2: SpiritMasterChef::updateEmissionRate()

```
    function updatePool(uint256 _pid) public {
        PoolInfo storage pool = poolInfo[_pid];
        if (block.number <= pool.lastRewardBlock) {
            return;
        }
        uint256 lpSupply = pool.lpToken.balanceOf(address(this));
        if (lpSupply == 0  pool.allocPoint == 0) {
            pool.lastRewardBlock = block.number;
            return;
        }
```

```
        uint256 multiplier = getMultiplier(pool.lastRewardBlock, block.number);
        uint256 spiritReward = multiplier.mul(spiritPerBlock).mul(pool.allocPoint).div(
            totalAllocPoint);
        spirit.mint(devaddr, spiritReward.div(10));
        spirit.mint(address(this), spiritReward);
        pool.accSpiritPerShare = pool.accSpiritPerShare.add(spiritReward.mul(1e12).div(
            lpSupply));
        pool.lastRewardBlock = block.number;
    }
```

Listing 3.3: SpiritMasterChef :: updatePool()

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status This issue has been confirmed.

3.3 Removal of Unused State in Gauges

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [4]

Description

The `SpiritV2` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `ReentrancyGuard`, and `Address`, to facilitate its code implementation and organization. For example, the `AdminGaugeProxy` 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 `Gauge`-related contracts, there are a number of local variables that are defined, but not used. Examples include the `_base` state and `minFee` state. These unused states can be safely removed for production deployment.

```
contract StableGaugeProxy is ProtocolGovernance, ReentrancyGuard {
    using SafeERC20 for IERC20;

    MasterChef public MASTER;
    IERC20 public inSPIRIT;
    IERC20 public SPIRIT;
    IERC20 public immutable TOKEN; // mInSpirit

    address public admin; //Admin address to manage gauges like add/deprecate/resurrect
```

```
    uint256 public minFee = 100 ether;

    ...
}
```

Listing 3.4: Various States in StableGaugeProxy

```
    mapping(address => uint256) public userRewardPerTokenPaid;
    mapping(address => uint256) public rewards;

    uint256 private _totalSupply;
    uint256 public derivedSupply;
    mapping(address => uint256) private _balances;
    mapping(address => uint256) public derivedBalances;
    mapping(address => uint256) private _base;
```

Listing 3.5: Various States in Gauge

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

Status This issue has been confirmed.

3.4 Improper Funding Source In inSpirit::deposit_for()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: inSpirit
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The SpiritV2 has a key inSpirit contract that provides the functionality of computing the time-dependent vote weights. By design, the vote weight decays linearly over time and the lock time cannot be more than MAXTIME (4 years). While reviewing the current locking logic, we notice the key helper routine _deposit_for() needs to be revised.

To elaborate, we show below the implementation of this _deposit_for() helper routine. In fact, it is an internal function to perform deposit and lock tokens for a user. This routine has a number of arguments and the first one _addr is the address to receive the balance. It comes to our attention that the _addr address is also the one to actually provide the assets, assert ERC20(self.token).transferFrom(_addr, self, _value) (line 377). In fact, the msg.sender should be the one to provide the assets for locking! Otherwise, this function may be abused to lock tokens from users who have approved the locking contract before without their notice.

```
def _deposit_for(_addr: address, _value: uint256, unlock_time: uint256, locked_balance: LockedBalance, type: int128):
    """
    @notice Deposit and lock tokens for a user
    @param _addr User's wallet address
    @param _value Amount to deposit
    @param unlock_time New time when to unlock the tokens, or 0 if unchanged
    @param locked_balance Previous locked amount / timestamp
    """
    _locked: LockedBalance = locked_balance
    supply_before: uint256 = self.supply

    self.supply = supply_before + _value
    old_locked: LockedBalance = _locked
    # Adding to existing lock, or if a lock is expired - creating a new one
    _locked.amount += convert(_value, int128)
    if unlock_time != 0:
        _locked.end = unlock_time
    self.locked[_addr] = _locked

    # Possibilities:
    # Both old_locked.end could be current or expired (>/< block.timestamp)
    # value == 0 (extend lock) or value > 0 (add to lock or extend lock)
    # _locked.end > block.timestamp (always)
    self._checkpoint(_addr, old_locked, _locked)

    if _value != 0:
        assert ERC20(self.token).transferFrom(_addr, self, _value)

    log Deposit(_addr, _value, _locked.end, type, block.timestamp)
    log Supply(supply_before, supply_before + _value)
```

Listing 3.6: `inSpirit::_deposit_for()`

Recommendation Revise the above helper routine to use the right funding source to transfer the assets for locking.

Status This issue has been confirmed.

3.5 Suggested immutable Use in BaseV1Pair

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: BaseV1Pair
- Category: Coding Practices [7]
- CWE subcategory: CWE-561 [3]

Description

Since version 0.6.5, Solidity introduces the feature of declaring a state as immutable. An immutable state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as immutable is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an immutable state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of immutable states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the SpiritV2 protocol, we observe there are several variables that need not to be updated dynamically. They can be declared as immutable for gas efficiency. Examples include the name and name defined in the BaseV1Pair contract.

```
// The base pair of pools, either stable or volatile
contract BaseV1Pair {

    string public name;
    string public symbol;
    uint8 public constant decimals = 18;
    ...
}
```

Listing 3.7: The States Defined in BaseV1Pair

Recommendation Revisit the state variable definition and make good use of immutable/constant states.

Status The issue has been confirmed.

3.6 Trust Issue Of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `SpiritV2` protocol, there is a privileged account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring the fee rates as well as adding new incentive pools). In the following, we show the representative functions potentially affected by the privilege of the account.

```
    function setStableFee(uint256 _fee) external {
        require(msg.sender == owner);
        require(_fee >= 100 && _fee <= 10000, "!range");
        stableFee = _fee;
    }

    function setVariableFee(uint256 _fee) external {
        require(msg.sender == owner);
        require(_fee >= 100 && _fee <= 10000, "!range");
        variableFee = _fee;
    }
```

Listing 3.8: `BaseV1Factory::setStableFee()/setVariableFee()`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed by the team. The team introduces `multi-sig` mechanism to manage the privileged account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the SpiritV2 protocol, which is designed to allow low-cost, low-slippage trades on uncorrelated or tightly correlated assets. The protocol is forked from the solidly AMM with the new curve $(x^3y + xy^3 = k)$ for efficient stable swaps. However, it removes the built-in NFT-based voting mechanism and modifies the fee structure to meet its own tokenomics. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that Solidity-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[3] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.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] Jong Seok Park. Sushiswap Delegation Double Spending Bug. https://medium.com/bulldax-finance/sushiswap-delegation-double-spending-bug-5adcc7b3830f.

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