SMART CONTRACT AUDIT REPORT

for

FINNEXUS PROTOCOL

Prepared By: Shuxiao Wang

Hangzhou, China

December 15, 2020

Document Properties

Client	FinNexus Protocol
Title	Smart Contract Audit Report
Target	FinNexus OptionsV1.0
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Jeff Liu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 15, 2020	Xuxian Jiang	Final Release
1.0-rc1	December 12, 2020	Xuxian Jiang	Release Candidate #1
0.4	December 5, 2020	Xuxian Jiang	Additional Findings #3
0.3	December 2, 2020	Xuxian Jiang	Additional Findings #2
0.2	November 28, 2020	Xuxian Jiang	Additional Findings #1
0.1	November 23, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of FinNexus OptionsV1.0, 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 FinNexus OptionsV1.0

The FinNexus Protocol for Options (FPO) is a cross-chain, permissionless protocol for options. FPO is innovative in writing options exposure for multiple assets from within collateral pools. Specifically, the proposed Multi-Asset Single Pool (MASP) methodology for decentralized peer-to-pool options platforms enables anyone anywhere to leverage or hedge their positions in a variety of cryptoassets. Currently live on Ethereum and Wanchain, FinNexus intends to bring its blockchain-agnostic FPO to other chains. The audited protocol provides a valuable instrument to hedge risks and control excessive exposure from market fluctuation and dynamics, therefore presenting a unique contribution to current DeFi ecosystem.

The basic information of FinNexus OptionsV1.0 is as follows:

Table 1.1: Basic Information of FinNexus OptionsV1.0

Item	Description
Issuer	FinNexus Protocol
Website	https://www.finnexus.io
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 15, 2020

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that FinNexus OptionsV1.0 assumes a trusted oracle with timely market price feeds and another oracle for option price feeds. These two oracles as well as the `Black-Scholes Merton (BSM)` economic model, including its applicability and parameter selection, are not part of this audit.

- https://github.com/FinNexus/FinnexusOptionsV1.0.git (01eb502)

1.2 About PeckShield

PeckShield Inc. [15] 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 [14]:

- 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.

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

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 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) [13], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 FinNexus OptionsV1.0 Protocol design and 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	0	
Medium	2	■■
Low	5	■■■■■
Informational	4	■■■■
Total	11	

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, 5 low-severity vulnerabilities, and 4 informational recommendations.

Table 2.1: Key FinNexus OptionsV1.0 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Necessity of Single-Shot Initialization	Init. and Cleanup	Confirmed
PVE-002	Medium	Possible Front-Running DoS Against FPTCoin Redemption	Business Logics	Confirmed
PVE-003	Low	Inaccurate Permission Checking in addCollateral()/_paybackWorth()	Business Logics	Confirmed
PVE-004	Informational	Improved Corner Case Handling in whiteListUint32	Coding Practices	Confirmed
PVE-005	Low	Potential Overflow For Option Rate Calculation	Numeric Errors	Fixed
PVE-006	Informational	Gas Optimization With Saved Transfers	Coding Practices	Fixed
PVE-007	Medium	Trust Issue of Admin Keys Behind CollateralPool	Security Features	Confirmed
PVE-008	Informational	Removal of Redundant Code	Coding Practices	Fixed
PVE-009	Low	Improved Collateral Amount Calculation	Numeric Errors	Confirmed
PVE-010	Low	Improved Sanity Checks For System Parameters	Coding Practices	Fixed
PVE-011	Informational	Improved Extra Hop Unwrapping in Delegated Calls	Coding Practices	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.

3 Detailed Results

3.1 Necessity of Single-Shot Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: SharedCoin
- Category: Initialization and Cleanup [11]
- CWE subcategory: CWE-1188 [3]

Description

Ethereum smart contracts are typically immutable by default. Once they are created, there is no way to alter them, effectively acting as an unbreakable contract among participants. In the meantime, there are several scenarios where there is a need to upgrade the contracts, either to add new functionalities or mitigate potential bugs.

The upgradeability support comes with a few caveats. One important caveat is related to the initialization of new contracts that are just deployed to replace old contracts. Due to the inherent requirement of any proxy-based upgradeability system, no constructors can be used in upgradeable contracts. This means we need to change the constructor of a new contract into a regular function (typically named `initialize()`) that basically executes all the setup logic.

However, a follow-up caveat is that during a contract's lifetime, its constructor is guaranteed to be called exactly once (and it typically happens at the very moment of being deployed). But a regular function may be called multiple times! In order to ensure that a contract will only be initialized once, we need to guarantee that the chosen `initialize()` function can be called only once during the entire lifetime. This guarantee is typically implemented as a modifier named `initializer`.

FinNexus OptionsV1.0 implements the upgradeability logic in `baseProxy`, which unfortunately does not provide the `initializer` modifier support. To facilitate our discussion, we show the code snippet of `SharedCoin` below.

```
pragma solidity =0.5.16;
import "../modules/SafeMath.sol";
```

```
import "./FPTData.sol";
contract SharedCoin is FPTData  {
    using SafeMath for uint256;
    function initialize() onlyOwner public{
        name = "finnexus pool token";
        symbol = "FPT";
        _totalSupply = 0;
    }
    ...
}
```

Listing 3.1: SharedCoin:: initialize ()

Apparently the above logic related to `initialize()` only protects the caller is authenticated and allowed by the system. But it does not provide the guarantee that the `initialize()` function can be called only once. Considering the need of multiple versions arranged for future upgrades, we strongly suggest the adoption of the known `initializer` modifier.

The same issue is also applicable to other `initialize()` routines in `CollateralPool`, `FNXMinePool` , `OptionsManagerV2`, and `OptionsBase`. Note that `FNXMinePool` and `OptionsManagerV2` have empty `initialize()` routines as the placeholders.

We also highlight that due to the inherent requirement of any proxy-based upgradeability system, no constructors of the logic contracts need to be used and the related functionalities can move to `initialize()`. However, there are several `contructors()` routines violate this requirement, including `CollateralPool`, `OptionsManagerV2`, and `OptionsPool`.

Recommendation Adopt the `VersionedInitializable` contract from `OpenZeppelin` for proper initialization with the required guarantee of executing the intended `initialize()` function only once during the entire lifetime.

```
pragma solidity =0.5.16;
import "../modules/SafeMath.sol";

import "./FPTData.sol";
contract SharedCoin is FPTData  {
    using SafeMath for uint256;
    function initialize() onlyOwner public initializer {
        name = "finnexus pool token";
        symbol = "FPT";
        _totalSupply = 0;
    }

    /**
     * @dev Modifier to use in the initializer function of a contract.
     */
    modifier initializer() {
        uint256 revision = getRevision();
        require(
```

```
            initializing
                isConstructor()
                revision > lastInitializedRevision,
            "Contract instance has already been initialized"
        );

        bool isTopLevelCall = !initializing;
        if (isTopLevelCall) {
            initializing = true;
            lastInitializedRevision = revision;
        }

        _;

        if (isTopLevelCall) {
            initializing = false;
        }
    }
```

Listing 3.2: SharedCoin.sol

Status This issue has been confirmed. The team decides to address this issue during the next upgrade.

3.2 Possible Front-Running DoS Against FPTCoin Redemption

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `FPTCoin`
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

FinNexus OptionsV1.0 is an on-chain peer-to-pool options trading protocol built on Ethereum and Wanchain. The pool has well-defined APIs that allow for liquidity providers ("writers") to efficiently add or remove funds. By doing so, funds from liquidity providers can be distributed among many hedge contracts simultaneously. It not only diversifies the liquidity allocation and makes efficient use of funds in the pool, but collectively shares the associated risks from one particular writer to all active liquidity providers.

The defined APIs for pool management mainly include `addCollateral()` and `redeemCollateral()`. The `addCollateral()` routine is used to add funds into the pool while the `redeemCollateral()` routine is used to withdraw funds from the pool. Meanwhile, he pool supports a lockup period for new funds into the pool. Specifically, for each liquidity provider, the associated lockup period is recorded as `[itemTimeMap[account], itemTimeMap[account].add(limitation)]`. Moreover, when any `mint()`, `transfer()` or `transferFrom()` action occurs, there is an accompanying lockup verification modifier, i.e., `OutLimitation`. In the following, we outline the code logic of `burn()` and `OutLimitation`.

```
    /**
     * @dev burn user's FPT when user redeem FPTCoin.
     * @param account user's account.
     * @param amount amount of FPT.
     */
    function burn(address account, uint256 amount) public onlyManager OutLimitation(
        uint256(account)) {
        require(address(_FnxMinePool) != address(0),"FnxMinePool is not set");
        _FnxMinePool.burnMinerCoin(account,amount);
        SharedCoin._burn(account,amount);
    }
```

Listing 3.3: FPTCoin::burn()

```
12      /**
13       * @dev set time limitation, only owner can invoke.
14       * @param _limitation new time limitation.
15       */
16      function setTimeLimitation(uint256 _limitation) public onlyOwner {
17          limitation = _limitation;
18      }
19      function setItemTimeLimitation(uint256 item) internal{
20          itemTimeMap[item] = now;
21      }
22      function getTimeLimitation() public view returns (uint256){
23          return limitation;
24      }
25      /**
26       * @dev Retrieve user's start time for burning.
27       * @param item item key.
28       */
29      function getItemTimeLimitation(uint256 item) public view returns (uint256){
30          return itemTimeMap[item]+limitation;
31      }
32      modifier OutLimitation(uint256 item) {
33          require(itemTimeMap[item]+limitation<now,"Time limitation is not expired!");
34          _;
35      }
```

Listing 3.4: timeLimitation :: OutLimitation

By examining the above routines, we identify a possible front-running attack that may block an ongoing withdrawal attempt. Specifically, when a `transfer()` or `transferFrom()` action occurs, the lockup period of the receiver, i.e., `itemTimeMap[to]`, might be accordingly updated. Therefore, upon the observation of a `burn()` attempt from a victim, a malicious actor could intentionally transfer 1 `WEI` to the victim. By doing so, the `itemTimeMap` of the victim is updated with current timestamp. As a result, the specific `burn()` attempt is blocked as it occurs in the lockup period (line 33).

Recommendation A mitigation to the above front-running attacks need to prevent malicious actors from tampering with legitimate accounts. In the meantime, we acknowledge that front-running attacks are inherent in current DeFi system and there is still a need to search for more effective countermeasures.

Status This issue has been confirmed. The team has considered possible penalty countermeasures that can be applied to the malicious actor who exploits this issue. These penalty countermeasures may cause greater loss and thus can effectively deter the wrongdoing of possible exploitation of this issue. In the meantime, the team will accordingly adjust the `time limitation` after the protocol becomes stable.

3.3 Inaccurate Permission Checking in addCollateral()/_paybackWorth()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CollateralCal`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

FinNexus OptionsV1.0 is innovative in proposing a `Multi-Asset Single Pool (MASP)` methodology for decentralized peer-to-pool options platforms. By design, it is able to accommodate multiple assets as the collateral, e.g., `FNX` and `USDC`. Moreover, it provides a fine-grained control on how the collateral might be used, including `allowBuyOptions`, `allowSellOptions`, `allowExerciseOptions`, `allowAddCollateral`, and `allowRedeemCollateral`.

```
    /**
     * @dev Implementation of a whitelist filters a eligible address.
     */
contract AddressWhiteList is Halt {

    using whiteListAddress for address[];
    uint256 constant internal allPermission = 0xffffffff;
    uint256 constant internal allowBuyOptions = 1;
    uint256 constant internal allowSellOptions = 1<<1;
    uint256 constant internal allowExerciseOptions = 1<<2;
    uint256 constant internal allowAddCollateral = 1<<3;
    uint256 constant internal allowRedeemCollateral = 1<<4;
    ...
}
```

Listing 3.5: AddressWhiteList

Our analysis shows that the fine-grained control on the collateral is not properly enforced. Using the `addCollateral()` routine as an example, it allows for users to deposit collateral into the pool. This routine takes two argument: `collateral` and `amount` and invokes an internal handler routine `getPayableAmount()`.

```
    /**
     * @dev Deposit collateral in this pool from user.
     * @param collateral The collateral coin address which is in whitelist.
     * @param amount the amount of collateral to deposit.
     */
    function addCollateral(address collateral,uint256 amount) nonReentrant notHalted
        public payable {
        amount = getPayableAmount(collateral,amount);
        uint256 fee = _collateralPool.addTransactionFee(collateral,amount,3);
        amount = amount-fee;
        uint256 price = oraclePrice(collateral);
        uint256 userPaying = price*amount;
        require(checkAllowance(msg.sender,(_collateralPool.getUserPayingUsd(msg.sender)+
            userPaying)/1e8),
            "Allowances : user's allowance is unsufficient!");
        uint256 mintAmount = userPaying/getTokenNetworth();
        _collateralPool.addUserPayingUsd(msg.sender,userPaying);
        _collateralPool.addCollateralBalance(collateral,amount);
        _collateralPool.addUserInputCollateral(msg.sender,collateral,amount);
         _collateralPool.addNetWorthBalance(collateral,int256(amount));
        emit AddCollateral(msg.sender,collateral,amount,mintAmount);
        _FPTCoin.mint(msg.sender,mintAmount);
    }
```

Listing 3.6: CollateralCal :: addCollateral ()

In the following, we show the code snippet of `getPayableAmount()`. The line at 310 shows the permission checking, i.e., `checkAddressPermission(settlement,allowBuyOptions)`. However, in the context of adding collateral into the pool, the `allowBuyOptions` is being validated, not the proper `allowAddCollateral`.

```
306     /**
307      * @dev the auxiliary function for getting user's transfer
308      */
309     function getPayableAmount(address settlement,uint256 settlementAmount) internal
            returns (uint256) {
310         require(checkAddressPermission(settlement,allowBuyOptions) , "settlement is
            unsupported token");
311         if (settlement == address(0)){
312             settlementAmount = msg.value;
313             address payable poolAddr = address(uint160(address(_collateralPool)));
314             poolAddr.transfer(settlementAmount);
315         }else if (settlementAmount > 0){
316             IERC20 oToken = IERC20(settlement);
317             uint256 preBalance = oToken.balanceOf(address(this));
318             oToken.transferFrom(msg.sender, address(this), settlementAmount);
```

```
            uint256 afterBalance = oToken.balanceOf(address(this));
            require(afterBalance-preBalance==settlementAmount,"settlement token transfer
                error!");
            oToken.transfer(address(_collateralPool),settlementAmount);
        }
        require(isInputAmountInRange(settlementAmount),"input amount is out of input
            amount range");
        return settlementAmount;
    }
```

Listing 3.7: CollateralCal ::getPayableAmount()

Other inaccurate validations of collateral permissions also occur in `CollateralCal::_getCollateralAndPremiumBalanc` `()` (line 195) and `CollateralCal::_paybackWorth()` (line 290).

Recommendation Revise the logic to properly implement the proper permission validation. An example revision is shown below:

```
    /**
     * @dev the auxiliary function for getting user's transfer
     */
    function getPayableAmount(address settlement,uint256 settlementAmount) internal
        returns (uint256) {
        require(checkAddressPermission(settlement,allowAddCollateral) , "settlement is
            unsupported token");
        if (settlement == address(0)){
            settlementAmount = msg.value;
            address payable poolAddr = address(uint160(address(_collateralPool)));
            poolAddr.transfer(settlementAmount);
        }else if (settlementAmount > 0){
            IERC20 oToken = IERC20(settlement);
            uint256 preBalance = oToken.balanceOf(address(this));
            oToken.transferFrom(msg.sender, address(this), settlementAmount);
            uint256 afterBalance = oToken.balanceOf(address(this));
            require(afterBalance-preBalance==settlementAmount,"settlement token transfer
                error!");
            oToken.transfer(address(_collateralPool),settlementAmount);
        }
        require(isInputAmountInRange(settlementAmount),"input amount is out of input
            amount range");
        return settlementAmount;
    }
```

Listing 3.8: CollateralCal ::getPayableAmount()

Status This issue has been confirmed. The team has confirmed that these fine-grained controls have become unnecessary and have thus been used interchangeably.

3.4 Improved Corner Case Handling in whiteListUint32

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `whiteListUint32`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [1]

Description

FinNexus OptionsV1.0 makes use of a number of well-defined libraries, including `AddressWhiteList`, `Halt`, `Ownable`, and `whiteListUint32`. These libraries greatly facilitate the code organization and maintenance of the protocol implementation.

During our analysis of one specific library, i.e., `whiteListUint32`, we notice certain corner cases can be better handled. To elaborate, we show below the code snippet of `_getEligibleIndexUint32()` in the library.

The FinNexus OptionsV1.0 protocol takes a rather prudent approach in maintaining a threshold of 80% of locked funds above which no new option will be created. This restriction is enforced when a new option always needs to lock certain amount of funds in the pool (in the `lock()` routine as shown below), i.e., `require(lockedAmount.add(amount).mul(10).div(totalBalance())< 8)` (line 115).

```
    function _getEligibleIndexUint32(uint32[] memory whiteList,uint32 temp) internal
        pure returns (uint256){
        uint256 len = whiteList.length;
        uint256 i=0;
        for (;i<len;i++){
            if (whiteList[i] == temp)
                break;
        }
        return i;
    }
```

Listing 3.9: whiteListUint32::_getEligibleIndexUint32()

This particular routine attempts to locate the index of a given `uint32 temp` within the internal `whiteList`. It came to our attention when the given `temp` does not show up in the list. It returns the current length of the list. Though our analysis shows there is no noticeable harm from this return value, we feel the need for a library function to ensure it always proper return value(s).

There are two other routines – `_getEligibleIndexUint256()`, `_getEligibleIndexAddress()` – that share the same issue.

Recommendation Revise the logic to ensure it always returns correct values. An example revision is shown below:

```
    function _getEligibleIndexUint32(uint32[] memory whiteList,uint32 temp) internal
        pure returns (uint256){
        uint256 len = whiteList.length;
        uint256 i=0;
        for (;i<len;i++){
            if (whiteList[i] == temp)
                break;
        }
        require(i<len, "not found!")
        return i;
    }
```

Listing 3.10: whiteListUint32::_getEligibleIndexUint32()

Status The team has confirmed that this is part of design and there is no need to change.

3.5 Potential Overflow For Option Rate Calculation

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: OptionsManagerV2
- Category: Numeric Errors [12]
- CWE subcategory: CWE-190 [4]

Description

`SafeMath` is a Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, we find that it is not widely used in current code base.

For example, while reviewing the `_getOptionsPriceRate()` routine, we notice an internal variable `buyOccupied` is calculated via `buyOccupied = ((optType == 0)== (strikePrice>underlyingPrice))? strikePrice*amount:underlyingPrice*amount` (line 143). Both `strikePrice` and `amount` may be directly taken from user input. As a result, it can directly report possibly wrong option price. Fortunately, the execution path for actual option purchase properly validate the given arguments and block possible attempts of having a corrupted option prices.

```
    function getOptionsPrice(uint256 underlyingPrice, uint256 strikePrice, uint256
        expiration,
                    uint32 underlying,uint256 amount,uint8 optType) public view returns(
                        uint256){
        uint256 ratio = _getOptionsPriceRate(underlyingPrice,strikePrice,amount,optType)
            ;
        uint256 optPrice = _optionsPrice.getOptionsPrice(underlyingPrice,strikePrice,
            expiration,underlying,optType);
        return (optPrice*ratio)>>32;
```

```
    }
    function _getOptionsPriceRate(uint256 underlyingPrice, uint256 strikePrice,uint256
        amount,uint8 optType) internal view returns(uint256){
        (uint256 totalCollateral,uint256 rate) = getCollateralAndRate();
        uint256 lockedWorth = _FPTCoin.getTotalLockedWorth();
        require(totalCollateral>=lockedWorth,"collateral is insufficient!");
        totalCollateral = totalCollateral - lockedWorth;
        uint256 buyOccupied = ((optType == 0) == (strikePrice>underlyingPrice)) ?
            strikePrice*amount:underlyingPrice*amount;
        (uint256 callCollateral,uint256 putCollateral) = _optionsPool.
            getAllTotalOccupiedCollateral();
        uint256 totalOccupied = (callCollateral + putCollateral + buyOccupied)*rate
            /1000;
        buyOccupied = ((optType == 0 ? callCollateral : putCollateral) + buyOccupied)*
            rate/1000;
        require(totalCollateral>=totalOccupied,"collateral is insufficient!");
        return calOptionsPriceRatio(buyOccupied,totalOccupied,totalCollateral);
    }
```

Listing 3.11: OptionsManagerV2::getOptionsPrice()

Recommendation Revise the logic accordingly to ensure the `getOptionsPrice()` routine behaviors consistently with actual option-buying execution path.

Status This issue has been fixed in the following commit: b393346.

3.6 Gas Optimization With Saved Transfers

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CollateralCal`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1041 [1]

Description

As mentioned in Section 3.2, FinNexus OptionsV1.0 is an on-chain peer-to-pool options trading protocol that allows for liquidity providers ("writers") to efficiently add or remove funds. By doing so, funds from liquidity providers can be distributed among many hedge contracts simultaneously. It not only diversifies the liquidity allocation and makes efficient use of funds in the pool, but collectively shares the associated risks from one particular writer to all active liquidity providers.

While reviewing the fund-addition logic into the pool, we notice certain optimization can be applied to avoid unnecessary gas waste. To elaborate, we show below the `getPayableAmount()` routine. This routine is used to transfer funds from the depositing user to the pool. However, we notice that for ERC20 tokens, they are moved in two steps: The first step (line 318) moves the funds from the user to the `OptionMangerV2` contract and the second step (line 321) moves the funds from the `OptionMangerV2` contract to the collateral pool. This is unnecessary as the above two steps can be consolidated into one single step by directly moving the funds from the depositing user to the collateral pool.

```
306     /**
307      * @dev the auxiliary function for getting user's transfer
308      */
309     function getPayableAmount(address settlement,uint256 settlementAmount) internal returns (uint256) {
310         require(checkAddressPermission(settlement,allowBuyOptions) , "settlement is unsupported token");
311         if (settlement == address(0)){
312             settlementAmount = msg.value;
313             address payable poolAddr = address(uint160(address(_collateralPool)));
314             poolAddr.transfer(settlementAmount);
315         }else if (settlementAmount > 0){
316             IERC20 oToken = IERC20(settlement);
317             uint256 preBalance = oToken.balanceOf(address(this));
318             oToken.transferFrom(msg.sender, address(this), settlementAmount);
319             uint256 afterBalance = oToken.balanceOf(address(this));
320             require(afterBalance-preBalance==settlementAmount,"settlement token transfer error!");
321             oToken.transfer(address(_collateralPool),settlementAmount);
322         }
323         require(isInputAmountInRange(settlementAmount),"input amount is out of input amount range");
324         return settlementAmount;
325     }
```

Listing 3.12: CollateralCal :: getPayableAmount()

Recommendation Consolidate the above two steps into one by directly moving the funds from the depositing user to the collateral pool.

Status This issue has been fixed in the following commit: b393346.

3.7 Trust Issue of Admin Keys Behind CollateralPool

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `CollateralPool`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [5]

Description

In FinNexus OptionsV1.0, there is a protocol-wide admin key in `Owner`. This `Owner` plays a critical role in configuring or updating the collateral pools' manager, which has the authority to not only update internal accounting records, but also actually move funds out to an external arbitrary recipient.

If we take a close look at `transferPayback()`, this specific routine takes three arguments: `recieptor`, `settlement`, and `payback`. The first argument is the `recieptor` specifies the destination for the withdrawal, the second argument indicates the collateral asset, and the third parameter shows the actual amount. This is a privileged routine governed by the `onlyManager` modifier.

```
    /**
     * @dev Operation for transfer user's payback. Only manager contract can invoke this
         function.
     * @param recieptor the recieptor account.
     * @param settlement the settlement coin address.
     * @param payback the payback amount
     */
    function transferPayback(address payable recieptor,address settlement,uint256
        payback)public onlyManager{
        _transferPayback(recieptor,settlement,payback);
    }
```

Listing 3.13: CollateralPool :: transferPayback()

As mentioned earlier, the collateral pool's manager can be updated by the `Owner`. Instead of having a single EOA account as the `Owner`, an alternative is to make use of a multi-sig wallet. To further eliminate the administration key concern, it may be required to transfer the role to a community-governed DAO. In the meantime, a timelock-based mechanism might also be applicable for mitigation.

Recommendation Promptly transfer the `Owner` privilege to an appropriate governance contract.

Status This issue has been confirmed. At the current stage, this is necessary for protocol-wide operation.

3.8 Removal of Redundant Code

- ID: PVE-008
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `OptionsManager`
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [6]

Description

FinNexus OptionsV1.0 makes good use of a number of reference contracts, such as `AddressWhiteList`, `Allowances`, `Ownable`, and `Halt` to facilitate its code implementation and organization. For example, the `OptionsData` 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 `exerciseOption()` in the `OptionsManager` contract, there is a need to calculate the option price for purchase (line 124). To elaborate, we show the related code snippet below.

```
113     /**
114     * @dev User exercise option.
115     * @param optionsId option's ID which was wanted to exercise, must owned by user
116     * @param amount user input amount of option user want to exercise.
117     */
118     function exerciseOption(uint256 optionsId,uint256 amount) nonReentrant notHalted
            InRange(amount) public{
119         uint256 allPay = _optionsPool.getExerciseWorth(optionsId,amount);
120         require(allPay > 0,"This option cannot exercise");
121         (,,uint8 optType,uint32 underlying,uint256 expiration,uint256 strikePrice,) =
                _optionsPool.getOptionsById(optionsId);
122         expiration = expiration.sub(now);
123         uint256 currentPrice = oracleUnderlyingPrice(underlying);
124         uint256 optPrice = _optionsPrice.getOptionsPrice(currentPrice,strikePrice,
                expiration,underlying,optType);
125         _optionsPrice.getOptionsPrice(currentPrice,strikePrice,expiration,underlying,
                optType);
126         _optionsPool.burnOptions(msg.sender,optionsId,amount,optPrice);
127         (address settlement,uint256 fullPay) = _optionsPool.getBurnedFullPay(optionsId,
                amount);
128         _collateralPool.addNetWorthBalance(settlement,int256(fullPay));
129         _paybackWorth(allPay,2);
130         emit ExerciseOption(msg.sender,optionsId,amount,allPay);
131     }
```

Listing 3.14: OptionsManager::exerciseOption()

We notice the call to `getOptionsPrice()` has been invoked twice with the same exact arguments and the second call does not assign the return value to any intermediate variable. Therefore, we consider the second call is unnecessary and can be safely removed.

In addition, we also notice the event `DebugEvent` has been defined (in `ManagerData` and `OptionsData`), but is never emitted.

Recommendation Delete unused `DebugEvent` events and remove duplicate code in `exerciseOption()` with the following revision:

```
    /**
    * @dev User exercise option.
    * @param optionsId option's ID which was wanted to exercise, must owned by user
    * @param amount user input amount of option user want to exercise.
    */
    function exerciseOption(uint256 optionsId,uint256 amount) nonReentrant notHalted InRange(amount) public{
        uint256 allPay = _optionsPool.getExerciseWorth(optionsId,amount);
        require(allPay > 0,"This option cannot exercise");
        (,,uint8 optType,uint32 underlying,uint256 expiration,uint256 strikePrice,) = _optionsPool.getOptionsById(optionsId);
        expiration = expiration.sub(now);
        uint256 currentPrice = oracleUnderlyingPrice(underlying);
        uint256 optPrice = _optionsPrice.getOptionsPrice(currentPrice,strikePrice,expiration,underlying,optType);
        _optionsPool.burnOptions(msg.sender,optionsId,amount,optPrice);
        (address settlement,uint256 fullPay) = _optionsPool.getBurnedFullPay(optionsId,amount);
        _collateralPool.addNetWorthBalance(settlement,int256(fullPay));
        _paybackWorth(allPay,2);
        emit ExerciseOption(msg.sender,optionsId,amount,allPay);
    }
```

Listing 3.15: OptionsManager::exerciseOption()

Status This issue has been fixed in the following commit: b393346.

3.9 Improved Collateral Amount Calculation

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: OptionsManager
- Category: Business Logics [10]
- CWE subcategory: CWE-841 [7]

Description

As discussed in Section 3.3, the FinNexus OptionsV1.0 protocol proposes a Multi-Asset Single Pool (MASP) methodology for decentralized peer-to-pool options platforms. For liquidity providers, it takes a rather prudent approach in maintaining a required collateralization ratio of each supported collateral asset. Moreover, each operation will be charged for a specific fee, including buyFee, sellFee, exerciseFee, addColFee, and redeemColFee.

While reviewing the addColFee logic, we notice the fee calculation logic can be better improved. To elaborate, we show the code snippet of addCollateral() that allows depositing users to add collateral into the pool.

```
67     /**
68      * @dev Deposit collateral in this pool from user.
69      * @param collateral The collateral coin address which is in whitelist.
70      * @param amount the amount of collateral to deposit.
71      */
72     function addCollateral(address collateral,uint256 amount) nonReentrant notHalted
           public payable {
73         amount = getPayableAmount(collateral,amount);
74         uint256 fee = _collateralPool.addTransactionFee(collateral,amount,3);
75         amount = amount-fee;
76         uint256 price = oraclePrice(collateral);
77         uint256 userPaying = price*amount;
78         require(checkAllowance(msg.sender,(_collateralPool.getUserPayingUsd(msg.sender)+
               userPaying)/1e8),
79             "Allowances : user's allowance is unsufficient!");
80         uint256 mintAmount = userPaying/getTokenNetworth();
81         _collateralPool.addUserPayingUsd(msg.sender,userPaying);
82         _collateralPool.addCollateralBalance(collateral,amount);
83         _collateralPool.addUserInputCollateral(msg.sender,collateral,amount);
84          _collateralPool.addNetWorthBalance(collateral,int256(amount));
85         emit AddCollateral(msg.sender,collateral,amount,mintAmount);
86         _FPTCoin.mint(msg.sender,mintAmount);
87     }
```

Listing 3.16: OptionsManager::addCollateral()

As shown at line 74, the fee is calculated via addTransactionFee() which in essence returns as FeeRates[feeType]*amount/1000. And the actual deposited amount becomes amount - FeeRates[

`feeType]*amount/1000`. This may not be appropriate as the deposited amount is better computed as `amount.mul(1000).div(1000+FeeRates[feeType])`.

Recommendation Revise the calculation of deposited amount to better reflect the fee design.

```
    /**
     * @dev Deposit collateral in this pool from user.
     * @param collateral The collateral coin address which is in whitelist.
     * @param amount the amount of collateral to deposit.
     */
    function addCollateral(address collateral,uint256 amount) nonReentrant notHalted
        public payable {
        amount = getPayableAmount(collateral,amount);
        ...
        amount = amount.mul(1000).div(1000+FeeRates[addColFee]); // The access to
            FeeRates and addColFee is omitted
        uint256 price = oraclePrice(collateral);
        uint256 userPaying = price*amount;
        require(checkAllowance(msg.sender,(_collateralPool.getUserPayingUsd(msg.sender)+
            userPaying)/1e8),
            "Allowances : user's allowance is unsufficient!");
        uint256 mintAmount = userPaying/getTokenNetworth();
        _collateralPool.addUserPayingUsd(msg.sender,userPaying);
        _collateralPool.addCollateralBalance(collateral,amount);
        _collateralPool.addUserInputCollateral(msg.sender,collateral,amount);
         _collateralPool.addNetWorthBalance(collateral,int256(amount));
        emit AddCollateral(msg.sender,collateral,amount,mintAmount);
        _FPTCoin.mint(msg.sender,mintAmount);
    }
```

Listing 3.17: OptionsManager::addCollateral()

Status This issue has been confirmed.

3.10 Improved Sanity Checks For System Parameters

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OptionsManagerV2`, `InputRange`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The FinNexus OptionsV1.0 protocol is no exception. Specifically, if we examine the `OptionsManagerV2` contract, it has defined the following parameters: `minPriceRate`, `maxPriceRate`, and `collateralRate`. These parameters define the valid range rate for input strike prices, and the collateralization ratio of each collateral asset, respectively.

Our analysis shows the update logic on these parameters 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 `minPriceRate` and `maxPriceRate` will revert every `buyOption()` operation, hence preventing the options from being purchased.

To elaborate, we show below its code snippet of `setPriceRateRange()`. This routine updates the valid range rate for strike prices. However, they can be improved to validate that the given `_minPriceRate` and `_maxPriceRate` fall in an appropriate range.

```
    /**
    * @dev set input price valid range rate, thousandths.
    */
    function setPriceRateRange(uint256 _minPriceRate,uint256 _maxPriceRate) public onlyOwner{
        minPriceRate = _minPriceRate;
        maxPriceRate = _maxPriceRate;
    }
```

Listing 3.18: OptionsManagerV2::setPriceRateRange()

Note that these two parameters `minPriceRate` and `maxPriceRate` are used to validate whether an input strike price should be accepted. And the validation is enforced for every option purchase (line 69 in `buyOption()`).

The same issue is also applicable to `minAmount` and `maxAmount` in `ImputRange::setInputAmountRange()`. In the meantime, we also strongly suggest to validate the given `optType` in `calOptionsOccupied()` can be 0 or 1 only.

```
65    function buyOption(address settlement,uint256 settlementAmount, uint256 strikePrice,uint32 underlying,
66                uint32 expiration,uint256 amount,uint8 optType) nonReentrant notHalted InRange(amount) public payable{
67        uint256 type_ly_expiration = optType+(uint256(underlying)<<64)+(uint256(expiration)<<128);
68        (uint256 settlePrice,uint256 underlyingPrice) = oracleAssetAndUnderlyingPrice(settlement,underlying);
69        checkStrikePrice(strikePrice,underlyingPrice);
70        uint256 optRate = _getOptionsPriceRate(underlyingPrice,strikePrice,amount,optType);
71
72        uint256 optPrice = _optionsPool.createOptions(msg.sender,settlement,type_ly_expiration,
73            uint128(strikePrice),uint128(underlyingPrice),uint128(amount),uint128((settlePrice<<32)/optRate));
74        optPrice = (optPrice*optRate)>>32;
75        buyOption_sub(settlement,settlementAmount,optPrice,settlePrice,amount);
76    }
```

Listing 3.19: OptionsManagerV2::buyOption()

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range. If necessary, also consider emitting relevant events for their changes.

Status This issue has been fixed in the following commit: b393346.

3.11 Improved Extra Hop Unwrapping in Delegated Calls

- ID: PVE-011
- Severity: Informational
- Likelihood: None
- Impact: None
- Target: `YAMDelegate`, `YAMRebaser`
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [6]

Description

FinNexus OptionsV1.0 accommodates the upgradeability support by deploying a proxy contract in front of the actual implementation (or logic contract). In particular, the `baseProxy`-based contract behaves as the proxy by relaying calls to the backend logic contract (e.g., `CollateralPool`, `FNXMinePool`, `OptionsManagerV2`, and `OptionsPool`). The call-relaying is mainly implemented by two helper routines: `delegateAndReturn()` and `delegateToViewAndReturn()`. The first one mainly relay external calls that may inflict state changes while the second one is mainly for `getter`-related calls without causing any state change.

We notice that the `delegateToViewAndReturn()` implementation (as shown below) returns results or forwards reverts to its caller. However, as it relays the call by making a `staticcall` call to itself, hence bringing an extra hop in the call chain. Note that each extra hop will introduce additional two `uint256` integers as the prefix of the wrapper `returndata`. In order to ensure the returned results are intact, we accordingly need to remove the two-`uint256`-integers prefix before returning back to the caller. The current implementation properly adjusts the offset of return bytes, i.e., `return(add(free_mem_ptr, 0x40), returndatasize)` (line 66). However, its length also needs to reduce the two `uint256` integers as follows, i.e., `return(add(free_mem_ptr, 0x40), returndatasize-0x40)`.

```
57     function delegateToViewAndReturn() internal view returns (bytes memory) {
58         (bool success, ) = address(this).staticcall(abi.encodeWithSignature("
               delegateToImplementation(bytes)", msg.data));
59
60         assembly {
61             let free_mem_ptr := mload(0x40)
62             returndatacopy(free_mem_ptr, 0, returndatasize)
63
64             switch success
65             case 0 { revert(free_mem_ptr, returndatasize) }
66             default { return(add(free_mem_ptr, 0x40), returndatasize) }
```

```
        }
    }

    function delegateAndReturn() private returns (bytes memory) {
        (bool success, ) = implementation.delegatecall(msg.data);

        assembly {
            let free_mem_ptr := mload(0x40)
            returndatacopy(free_mem_ptr, 0, returndatasize)

            switch success
            case 0 { revert(free_mem_ptr, returndatasize) }
            default { return(free_mem_ptr, returndatasize) }
        }
    }
```

Listing 3.20: baseProxy::delegateToViewAndReturn()

Recommendation Unwrap the extra call by accordingly reducing the `returndatasize` as well (in addition to adjusting the offset of returned bytes in `free_mem_ptr`).

```
    function delegateToViewAndReturn() internal view returns (bytes memory) {
        (bool success, ) = address(this).staticcall(abi.encodeWithSignature("
            delegateToImplementation(bytes)", msg.data));

        assembly {
            let free_mem_ptr := mload(0x40)
            returndatacopy(free_mem_ptr, 0, returndatasize)

            switch success
            case 0 { revert(free_mem_ptr, returndatasize) }
            default { return(add(free_mem_ptr, 0x40), returndatasize-0x40) }
        }
    }

    function delegateAndReturn() private returns (bytes memory) {
        (bool success, ) = implementation.delegatecall(msg.data);

        assembly {
            let free_mem_ptr := mload(0x40)
            returndatacopy(free_mem_ptr, 0, returndatasize)

            switch success
            case 0 { revert(free_mem_ptr, returndatasize) }
            default { return(free_mem_ptr, returndatasize) }
        }
    }
```

Listing 3.21: baseProxy::delegateToViewAndReturn()

Status This issue has been confirmed. The team decides to address this issue during the next upgrade.

4 Conclusion

In this audit, we have analyzed the FinNexus OptionsV1.0 design and implementation. The system presents a unique offering in current DeFi ecosystem by proposing the Multi-Asset Single Pool (MASP) methodology for decentralized peer-to-pool options platforms and enabling anyone anywhere to leverage or hedge their positions in a variety of cryptoassets. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[3] MITRE. CWE-1188: Insecure Default Initialization of Resource. https://cwe.mitre.org/data/definitions/1188.html.

[4] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

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

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

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

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

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

[11] MITRE. CWE CATEGORY: Initialization and Cleanup Errors. https://cwe.mitre.org/data/definitions/452.html.

[12] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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