SMART CONTRACT AUDIT REPORT

for

Vortex

Prepared By: Yiqun Chen

PeckShield

November 20, 2021

Document Properties

Client	Akropolis
Title	Smart Contract Audit Report
Target	Vortex
Version	1.0-rc
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	November 20, 2021	Xiaotao Wu	Release Candidate #1

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Vortex` 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 Vortex

`Vortex` (v1) is an on-chain basis trading strategy that aims to generate long-term, sustainable and rewarding yields while remaining market-neutral. `Vortex` allows users to generate yield without being exposed to directional price risk. It only requires users to deposit a single asset, i.e., `USDC`, which makes `Vortex` an effective alternative to lending or farming with stablecoins. The maintenance for `Vortex` is low and the returns generated by `Vortex` are periodically compounded, further enhancing yield.

The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Vortex

Item	Description
Name	Akropolis
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 20, 2021

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

- https://github.com/akropolisio/basis/tree/audit/peckshield (3387910)

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

- https://github.com/akropolisio/basis/tree/audit/peckshield (TBD)

1.2 About PeckShield

PeckShield Inc. [14] 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 [13]:

- 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

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

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) [12], 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 `Vortex` 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	3	■ ■ ■
Informational	1	■
Undetermined	0	
Total	6	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Costly akBasisVault From Improper Vault Initialization	Time and State	Confirmed
PVE-002	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-003	Informational	Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-004	Medium	Timely Re-margin During Strategy Margin Buffer Changes	Business Logic	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-006	Low	Potential Sandwich-Based MEV With Imbalanced Positions	Time and State	

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 Possible Costly akBasisVault From Improper Vault Initialization

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `BasisVault`
- Category: Time and State [8]
- CWE subcategory: CWE-362 [3]

Description

The `BasisVault` contract of `Vortex` protocol provides an external `deposit()` function for users to deposit the `want` token to the `Vault` and mint the corresponding shares of `akBasisVault` tokens to the users. While examining the `akBasisVault` token share calculation with the given `want` token amount, we notice an issue that may unnecessarily make the `akBasisVault` token extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `deposit()` routine. The issue occurs when the `Vault` is being initialized under the assumption that the current `Vault` is empty.

```
    /**
     * @notice  deposit function - where users can join the vault and
     *          receive shares in the vault proportional to their ownership
     *          of the funds.
     * @param  _amount    amount of want to be deposited
     * @param  _recipient recipient of the shares as the recipient may not
     *                    be the sender
     * @return shares the amount of shares being minted to the recipient
     *                for their deposit
     */
    function deposit(uint256 _amount, address _recipient)
        external
        nonReentrant
        whenNotPaused
```

```
        returns (uint256 shares)
    {
        require(_amount > 0, "!_amount");
        require(_recipient != address(0), "!_recipient");
        require(totalAssets() + _amount <= depositLimit, "!depositLimit");

        shares = _issueShares(_amount, _recipient);
        // transfer want to the vault
        want.safeTransferFrom(msg.sender, address(this), _amount);

        emit Deposit(_recipient, _amount, shares);
    }
```

Listing 3.1: `BasisVault::deposit()`

```
279     /**
280      * @dev      function for handling share issuance during a deposit
281      * @param  _amount     amount of want to be deposited
282      * @param  _recipient recipient of the shares as the recipient may not
283      *                     be the sender
284      * @return shares the amount of shares being minted to the recipient
285      *                 for their deposit
286      */
287     function _issueShares(uint256 _amount, address _recipient)
288         internal
289         returns (uint256 shares)
290     {
291         if (totalSupply() > 0) {
292             // if there is supply then mint according to the proportion of the pool
293             require(totalAssets() > 0, "totalAssets == 0");
294             shares = (_amount * totalSupply()) / totalAssets();
295         } else {
296             // if there is no supply mint 1 for 1
297             shares = _amount;
298         }
299         _mint(_recipient, shares);
300     }
```

Listing 3.2: `BasisVault::_issueShares()`

Specifically, when the `Vault` is being initialized, the `shares` value directly takes the value of `_amount` (line 297), which is manipulatable by the malicious actor. As this is the first time to deposit, the `totalSupply()` equals the given input amount, i.e., `_amount = 1 WEI`. With that, the actor can further donate a huge amount of `want` to `BasisVault` contract with the goal of making the `akBasisVault` extremely expensive (line 332).

```
323     /**
324      * @dev      function for determining the value of a share of the vault
325      * @param  _shares     amount of shares to convert
326      * @return the value of the inputted amount of shares in want
327      */
```

```
    function _calcShareValue(uint256 _shares) internal view returns (uint256) {
        if (totalSupply() == 0) {
            return _shares;
        }
        return (_shares * totalAssets()) / totalSupply();
    }
```

Listing 3.3: `BasisVault::_calcShareValue()`

An extremely expensive `akBasisVault` can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed `shares` for deposited assets (line 294). If truncated to be zero, the deposited assets are essentially considered dust and kept by the contract without returning any `akBasisVault` tokens.

Recommendation Revise current execution logic of `deposit()` to defensively calculate the mint amount when the `Vault` is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status This issue has been confirmed.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BasisStrategy`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
```

```
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.4: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()` , In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

```
    /**
     * @dev Deprecated. This function has issues similar to the ones found in
     * {IERC20-approve}, and its usage is discouraged.
     *
     * Whenever possible, use {safeIncreaseAllowance} and
     * {safeDecreaseAllowance} instead.
     */
    function safeApprove(
        IERC20 token,
        address spender,
        uint256 value
    ) internal {
        // safeApprove should only be called when setting an initial allowance,
        // or when resetting it to zero. To increase and decrease it, use
        // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
        require(
            (value == 0)  (token.allowance(address(this), spender) == 0),
            "SafeERC20: approve from non-zero to non-zero allowance"
        );
        _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
            spender, value));
    }
```

Listing 3.5: `SafeERC20::safeApprove()`

In current implementation, if we examine the `BasisStrategy::_depositToMarginAccount()` routine that is designed to deposit to the margin account without opening a perpetual position. To accommodate the specific idiosyncrasy, there is a need to use `safeApprover()`, instead of `approve()` (line

762).

```
757     /**
758      * @notice  deposit to the margin account without opening a perpetual position
759      * @param   _amount the amount to deposit into the margin account
760      */
761     function _depositToMarginAccount(uint256 _amount) internal {
762         IERC20(want).approve(address(mcLiquidityPool), _amount);
763         mcLiquidityPool.deposit(
764             perpetualIndex,
765             address(this),
766             int256(_amount) * DECIMAL_SHIFT
767         );
768         emit DepositToMarginAccount(_amount, perpetualIndex);
769     }
```

Listing 3.6: `BasisStrategy::_depositToMarginAccount()`

Note the `_swapTokenOut()` routine in the same contract can be similarly improved.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status This issue has been fixed in the following commit: `e782e0e`.

3.3 Meaningful Events For Important State Changes

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BasisStrategy`
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [4]

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 `BasisStrategy` contract as an example. While examining the events that reflect the `BasisStrategy` dynamics, we notice there is a lack of emitting related events to reflect important state changes. Specifically, when the `setters` are being called, there are no corresponding events being emitted to reflect the occurrence of `setLiquidityPool()/setUniswapPool()/setBasisVault`

`()/setBuffer()/setPerpetualIndex()/setReferrer()/setSlippageTolerance()/setDust()/setTradeMode()/setGovernance()/setVersion()/setLmClaimerAndMcb()/setWeth()/setLong()`.

```
    /**
     * @notice  setter for the mcdex liquidity pool
     * @param   _mcLiquidityPool MCDEX Liquidity and Perpetual Pool address
     * @dev     only callable by owner
     */
    function setLiquidityPool(address _mcLiquidityPool) external onlyOwner {
        mcLiquidityPool = IMCLP(_mcLiquidityPool);
    }

    /**
     * @notice  setter for the uniswap pair pool
     * @param   _pool Uniswap v3 pair pool address
     * @dev     only callable by owner
     */
    function setUniswapPool(address _pool) external onlyOwner {
        pool = _pool;
    }

    /**
     * @notice  setter for the basis vault
     * @param   _vault Basis Vault address
     * @dev     only callable by owner
     */
    function setBasisVault(address _vault) external onlyOwner {
        vault = IBasisVault(_vault);
    }

    /**
     * @notice  setter for buffer
     * @param   _buffer Basis strategy margin buffer
     * @dev     only callable by owner
     */
    function setBuffer(uint256 _buffer) public onlyOwner {
        require(_buffer < 1_000_000, "!_buffer");
        buffer = _buffer;                     //Evan: timely call remargin()
    }

    /**
     * @notice  setter for perpetualIndex value
     * @param   _perpetualIndex MCDEX perpetual index
     * @dev     only callable by owner
     */
    function setPerpetualIndex(uint256 _perpetualIndex) external onlyOwner {
        perpetualIndex = _perpetualIndex;
    }

    /**
     * @notice  setter for referrer for MCDEX rebates
     * @param   _referrer address of the MCDEX referral recipient
```

```
     * @dev      only callable by owner
     */
    function setReferrer(address _referrer) external onlyOwner {
        referrer = _referrer;
    }

    /**
     * @notice  setter for perpetual trade slippage tolerance
     * @param   _slippageTolerance amount of slippage tolerance to accept on perp trade
     * @dev      only callable by owner
     */
    function setSlippageTolerance(int256 _slippageTolerance)
        external
        onlyOwner
    {
        slippageTolerance = _slippageTolerance;
    }

    /**
     * @notice  setter for dust for closing margin positions
     * @param   _dust amount of dust in wei that is acceptable
     * @dev      only callable by owner
     */
    function setDust(int256 _dust) external onlyOwner {
        dust = _dust;
    }

    /**
     * @notice  setter for the tradeMode of the perp
     * @param   _tradeMode uint32 for the perp trade mode
     * @dev      only callable by owner
     */
    function setTradeMode(uint32 _tradeMode) external onlyOwner {
        tradeMode = _tradeMode;
    }

    /**
     * @notice  setter for the governance address
     * @param   _governance address of governance
     * @dev      only callable by governance
     */
    function setGovernance(address _governance) external onlyGovernance {
        governance = _governance;
    }

    /**

     * @notice set router version for network
     * @param _isV2 bool to set the version of rooter
     * @dev only callable by owner
     */
    function setVersion(bool _isV2) external onlyOwner {
```

```
        isV2 = _isV2;
    }

    /**
     * @notice  setter for liquidity mining claim contract
     * @param   _lmClaimer the claim contract
     * @param   _mcb the mcb token address
     * @dev     only callable by owner
     */
    function setLmClaimerAndMcb(address _lmClaimer, address _mcb)
        external
        onlyOwner
    {
        lmClaimer = ILmClaimer(_lmClaimer);
        mcb = _mcb;
    }

    /**
     * @notice  setter for weth depending on the network
     * @param   _weth for weth
     * @dev     only callable by owner
     */
    function setWeth(address _weth) external onlyOwner {
        require(_weth != address(0), "!_weth");
        weth = _weth;
    }

    /**
     * @notice  setter for long asset
     * @param   _long for long
     * @dev     only callable by owner
     */
    function setLong(address _long) external onlyOwner {
        require(_long != address(0), "!_long");
        long = _long;
    }
```

Listing 3.7: `BasisStrategy::setters`

Recommendation Properly emit the related events when the above-mentioned functions are being invoked.

Status This issue has been fixed in the following commit: `f91d28c`.

3.4 Timely Re-margin During Strategy Margin Buffer Changes

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `BasisStrategy`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [6]

Description

The `BasisStrategy` contract provides a privileged `setBuffer()` function for the `owner` to change the basis strategy margin buffer. This `buffer` is a set size and will be maintained so long as active positions are open. When analyzing the `buffer` update routine `setBuffer()`, we notice the need of timely invoking `remargin()` to remargin the strategy such that margin call risk is reduced.

```
    /**
     * @notice  setter for buffer
     * @param   _buffer Basis strategy margin buffer
     * @dev     only callable by owner
     */
    function setBuffer(uint256 _buffer) public onlyOwner {
        require(_buffer < 1_000_000, "!_buffer");
        buffer = _buffer;
    }
```

Listing 3.8: `BasisStrategy::setBuffer()`

If the call to `remargin()` is not immediately invoked after updating the `buffer`, then it is possible for the `Vault` to fall below the margin ratio and for the margin account to get liquidated, which would result in a substantial loss.

Recommendation Timely invoke `remargin()` when `buffer` has been updated.

```
    /**
     * @notice  setter for buffer
     * @param   _buffer Basis strategy margin buffer
     * @dev     only callable by owner
     */
    function setBuffer(uint256 _buffer) public onlyOwner {
        require(_buffer < 1_000_000, "!_buffer");
        buffer = _buffer;
        remargin();
    }
```

Listing 3.9: `BasisStrategy::setBuffer()`

Status This issue has been fixed in the following commit: `e93e50e`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

In the `Vortex` protocol, there are certain privileged accounts, i.e., `owner/governance`. When examining the related contracts, we notice inherent trust on these privileged accounts. To elaborate, we show below the related functions.

Firstly, a number of `setters`, e.g., `setProtocolFeeRecipient()`, `setStrategy()`, `setProtocolFees()` and `setDepositLimit()`, allow for the `owner` to set various protocol-wide risk parameters, including `protocolFeeRecipient`, `strategy`, `performanceFee`, `managementFee`, and `depositLimit`.

```
/**
 * @notice  set the maximum amount that can be deposited in the vault
 * @param   _depositLimit amount of want allowed to be deposited
 * @dev     only callable by owner
 */
function setDepositLimit(uint256 _depositLimit) external onlyOwner {
    depositLimit = _depositLimit;
    emit DepositLimitUpdated(_depositLimit);
}

/**
 * @notice  set the strategy associated with the vault
 * @param   _strategy address of the strategy
 * @dev     only callable by owner
 */
function setStrategy(address _strategy) external onlyOwner {
    require(_strategy != address(0), "!_strategy");
    strategy = _strategy;
    emit StrategyUpdated(_strategy);
}

/**
 * @notice function to set the protocol management and performance fees
 * @param  _performanceFee the fee applied for the strategies performance
 * @param  _managementFee the fee applied for the strategies management
 * @dev    only callable by the owner
 */
function setProtocolFees(uint256 _performanceFee, uint256 _managementFee)
    external
    onlyOwner
{
```

```
        emit ProtocolFeesUpdated(
            managementFee,
            _managementFee,
            performanceFee,
            _performanceFee
        );
        performanceFee = _performanceFee;
        managementFee = _managementFee;
    }

    /**
     * @notice function to set the protocol fee recipient
     * @param  _newRecipient the recipient of protocol fees
     * @dev    only callable by the owner
     */
    function setProtocolFeeRecipient(address _newRecipient) external onlyOwner {
        emit ProtocolFeeRecipientUpdated(protocolFeeRecipient, _newRecipient);
        protocolFeeRecipient = _newRecipient;
    }
```

Listing 3.10: `BasisVault::setters`

Note only the `strategy` is allowed to call the `update()` function of the `BasisVault` contract.

```
    /**
     * @notice function to update the state of the strategy in the vault and pull any funds to be redeposited
     * @param  _amount change in the vault amount sent by the strategy
     * @param  _loss   whether the change is negative or not
     *                 be the sender
     * @return toDeposit the amount to be deposited in to the strategy on this update
     */
    function update(uint256 _amount, bool _loss)
        external
        onlyStrategy
        returns (uint256 toDeposit)
    {
        // if a loss was recorded then decrease the totalLent by the amount, otherwise increase the totalLent
        if (_loss) {
            totalLent -= _amount;
        } else {
            _determineProtocolFees(_amount);
            totalLent += _amount;
        }
        // increase the totalLent by the amount of deposits that havent yet been sent to the vault
        toDeposit = want.balanceOf(address(this));
        totalLent += toDeposit;
        lastUpdate = block.timestamp;
        emit StrategyUpdate(_amount, _loss, toDeposit);
        if (toDeposit > 0) {
            want.approve(strategy, toDeposit);
```

```
            want.safeTransfer(msg.sender, toDeposit);
        }
    }
```

Listing 3.11: `BasisVault::update()`

Secondly, another set of setters, i.e., `setLiquidityPool()`, `setUniswapPool()`, `setBasisVault()`, `setBuffer()`, `setPerpetualIndex()`, `setReferrer()`, `setSlippageTolerance()`, `setDust()`, `setTradeMode()`, `setVersion()`, `setGovernance()`, `setLmClaimerAndMcb()`, `setWeth()`, and `setLong()`, allow for the `owner /governance` to set other risk parameters in `BasisStrategy`, including `mcLiquidityPool`, `pool`, `vault`, `buffer`, `perpetualIndex`, `referrer`, `slippageTolerance`, `dust`, `tradeMode`, `isV2`, `governance`, `lmClaimer`, `mcb`, `weth`, and `long`.

```
    /**
     * @notice  setter for the mcdex liquidity pool
     * @param   _mcLiquidityPool MCDEX Liquidity and Perpetual Pool address
     * @dev     only callable by owner
     */
    function setLiquidityPool(address _mcLiquidityPool) external onlyOwner {
        mcLiquidityPool = IMCLP(_mcLiquidityPool);
    }

    /**
     * @notice  setter for the uniswap pair pool
     * @param   _pool Uniswap v3 pair pool address
     * @dev     only callable by owner
     */
    function setUniswapPool(address _pool) external onlyOwner {
        pool = _pool;
    }

    /**
     * @notice  setter for the basis vault
     * @param   _vault Basis Vault address
     * @dev     only callable by owner
     */
    function setBasisVault(address _vault) external onlyOwner {
        vault = IBasisVault(_vault);
    }

    /**
     * @notice  setter for buffer
     * @param   _buffer Basis strategy margin buffer
     * @dev     only callable by owner
     */
    function setBuffer(uint256 _buffer) public onlyOwner {
        require(_buffer < 1_000_000, "!_buffer");
        buffer = _buffer;                   //Evan: timely call remargin()
    }

    /**
```

```
     * @notice  setter for perpetualIndex value
     * @param   _perpetualIndex MCDEX perpetual index
     * @dev     only callable by owner
     */
    function setPerpetualIndex(uint256 _perpetualIndex) external onlyOwner {
        perpetualIndex = _perpetualIndex;
    }

    /**
     * @notice  setter for referrer for MCDEX rebates
     * @param   _referrer address of the MCDEX referral recipient
     * @dev     only callable by owner
     */
    function setReferrer(address _referrer) external onlyOwner {
        referrer = _referrer;
    }

    /**
     * @notice  setter for perpetual trade slippage tolerance
     * @param   _slippageTolerance amount of slippage tolerance to accept on perp trade
     * @dev     only callable by owner
     */
    function setSlippageTolerance(int256 _slippageTolerance)
        external
        onlyOwner
    {
        slippageTolerance = _slippageTolerance;
    }

    /**
     * @notice  setter for dust for closing margin positions
     * @param   _dust amount of dust in wei that is acceptable
     * @dev     only callable by owner
     */
    function setDust(int256 _dust) external onlyOwner {
        dust = _dust;
    }

    /**
     * @notice  setter for the tradeMode of the perp
     * @param   _tradeMode uint32 for the perp trade mode
     * @dev     only callable by owner
     */
    function setTradeMode(uint32 _tradeMode) external onlyOwner {
        tradeMode = _tradeMode;
    }

    /**
     * @notice  setter for the governance address
     * @param   _governance address of governance
     * @dev     only callable by governance
     */
```

```
    function setGovernance(address _governance) external onlyGovernance {
        governance = _governance;
    }

    /**

     * @notice set router version for network
     * @param _isV2 bool to set the version of rooter
     * @dev only callable by owner
     */
    function setVersion(bool _isV2) external onlyOwner {
        isV2 = _isV2;
    }

    /**
     * @notice  setter for liquidity mining claim contract
     * @param   _lmClaimer the claim contract
     * @param   _mcb the mcb token address
     * @dev     only callable by owner
     */
    function setLmClaimerAndMcb(address _lmClaimer, address _mcb)
        external
        onlyOwner
    {
        lmClaimer = ILmClaimer(_lmClaimer);
        mcb = _mcb;
    }

    /**
     * @notice  setter for weth depending on the network
     * @param   _weth for weth
     * @dev     only callable by owner
     */
    function setWeth(address _weth) external onlyOwner {
        require(_weth != address(0), "!_weth");
        weth = _weth;
    }

    /**
     * @notice  setter for long asset
     * @param   _long for long
     * @dev     only callable by owner
     */
    function setLong(address _long) external onlyOwner {
        require(_long != address(0), "!_long");
        long = _long;
    }
```

Listing 3.12: `BasisStrategy::setters`

Note only the `vault` is allowed to call the `withdraw()` function of the `BasisStrategy` contract.

```
    /**
```

```
     * @notice  withdraw funds from the strategy
     * @param   _amount the amount to be withdrawn
     * @return  loss loss recorded
     * @return  withdrawn amount withdrawn
     * @dev     only callable by the vault
     */
    function withdraw(uint256 _amount)
        external
        onlyVault
        returns (uint256 loss, uint256 withdrawn)
    {
        require(_amount > 0, "withdraw: _amount is 0");
        uint256 longPositionWant;
        if (!isUnwind) {
            mcLiquidityPool.forceToSyncState();
            // remove the buffer from the amount
            uint256 bufferPosition = (_amount * buffer) / MAX_BPS;
            // decrement the amount by buffer position
            uint256 _remAmount = _amount - bufferPosition;
            // determine the shortPosition
            uint256 shortPosition = _remAmount / 2;
            // close the short position
            int256 positionsClosed = _closePerpPosition(shortPosition);
            // determine the long position
            uint256 longPosition = uint256(positionsClosed);
            // check that there are enough long positions, if there is not then close
                all longs
            if (longPosition < IERC20(long).balanceOf(address(this))) {
                // if for whatever reason there are funds left in long when there
                    shouldnt be then liquidate them
                if (getMarginPositions() == 0) {
                    longPosition = IERC20(long).balanceOf(address(this));
                }
                // convert the long to want
                longPositionWant = _swap(longPosition, long, want);
            } else {
                // convert the long to want
                longPositionWant = _swap(
                    IERC20(long).balanceOf(address(this)),
                    long,
                    want
                );
            }
            // check if there is enough margin to cover the buffer and short withdrawal
            // also make sure there are margin positions, as if there are none you can
            // withdraw most of the position
            if (
                getMargin() >
                int256(bufferPosition + shortPosition) * DECIMAL_SHIFT &&
                getMarginPositions() < 0
            ) {
                // withdraw the short and buffer from the margin account
```

```
                mcLiquidityPool.withdraw(
                    perpetualIndex,
                    address(this),
                    int256(bufferPosition + shortPosition) * DECIMAL_SHIFT
                );
            } else {
                if (getMarginPositions() < 0) {
                    _closeAllPerpPositions();
                }
                mcLiquidityPool.withdraw(
                    perpetualIndex,
                    address(this),
                    getMargin()
                );
            }
            withdrawn = longPositionWant + shortPosition + bufferPosition;
        } else {
            withdrawn = _amount;
        }

        uint256 wantBalance = IERC20(want).balanceOf(address(this));
        // transfer the funds back to the vault, if at this point needed isnt covered then
        // record a loss
        if (_amount > wantBalance) {
            IERC20(want).safeTransfer(address(vault), wantBalance);
            loss = _amount - wantBalance;
            withdrawn = wantBalance;
        } else {
            IERC20(want).safeTransfer(address(vault), _amount);
            loss = 0;
            withdrawn = _amount;
        }

        positions.perpContracts = getMarginPositions();
        positions.margin = getMargin();
        emit WithdrawStrategy(withdrawn, loss);
    }
```

Listing 3.13: `BasisStrategy::withdraw()`

Thirdly, the privileged functions in the `BasisStrategy` contract allow the `owner` to harvest the strategy and remargin the strategy.

```
    /**
     * @notice  harvest the strategy. This involves accruing profits from the strategy and depositing
     *          user funds to the strategy. The funds are split into their constituents and then distributed
     *          to their appropriate location.
     *          For the shortPosition a perpetual position is opened, for the long position funds are swapped
```

```
     *          to the long asset. For the buffer position the funds are deposited to
         the margin account idle.
     * @dev      only callable by the owner
     */
    function harvest() public onlyOwner {
        uint256 shortPosition;
        uint256 longPosition;
        uint256 bufferPosition;
        isUnwind = false;

        mcLiquidityPool.forceToSyncState();
        // determine the profit since the last harvest and remove profits from the
            margin
        // account to be redistributed
        uint256 amount;
        bool loss;
        if (positions.unitAccumulativeFunding != 0) {
            (amount, loss) = _determineFee();
        }
        // update the vault with profits/losses accrued and receive deposits
        uint256 newFunds = vault.update(amount, loss);
        // combine the funds and check that they are larger than 0
        uint256 toActivate = IERC20(want).balanceOf(address(this));

        if (toActivate > 0) {
            // determine the split of the funds and trade for the spot position of long
            (shortPosition, longPosition, bufferPosition) = _calculateSplit(
                toActivate
            );
            // deposit the bufferPosition to the margin account
            _depositToMarginAccount(bufferPosition);
            // open a short perpetual position and store the number of perp contracts
            positions.perpContracts += _openPerpPosition(shortPosition, true);
        }
        // record incremented positions
        positions.margin = getMargin();
        positions.unitAccumulativeFunding = getUnitAccumulativeFunding();
        emit Harvest(
            positions.perpContracts,
            IERC20(long).balanceOf(address(this)),
            positions.margin
        );
    }
```

Listing 3.14: `BasisStrategy::harvest()`

```
    /**
     * @notice  remargin the strategy such that margin call risk is reduced
     * @dev     only callable by owner
     */
    function remargin() external onlyOwner {
        // harvest the funds so the positions are up to date
        harvest();
```

```
        // ratio of the short in the short and buffer
        int256 K = (((int256(MAX_BPS) - int256(buffer)) / 2) * 1e18) /
            (((int256(MAX_BPS) - int256(buffer)) / 2) + int256(buffer));
        // get the price of ETH
        (, address oracleAddress, ) = mcLiquidityPool.getPerpetualInfo(
            perpetualIndex
        );
        IOracle oracle = IOracle(oracleAddress);
        (int256 price, ) = oracle.priceTWAPLong();
        // calculate amount to unwind
        int256 unwindAmount = (((price * -getMarginPositions()) -
            K *
            getMargin()) * 1e18) / ((1e18 + K) * price);
        require(unwindAmount != 0, "no changes to margin necessary");
        // check if leverage is to be reduced or increased then act accordingly
        if (unwindAmount > 0) {
            // swap unwindAmount long to want
            uint256 wantAmount = _swap(uint256(unwindAmount), long, want);
            // close unwindAmount short to margin account
            mcLiquidityPool.trade(
                perpetualIndex,
                address(this),
                unwindAmount,
                price + slippageTolerance,
                block.timestamp,
                referrer,
                tradeMode
            );
            // deposit long swapped collateral to margin account
            _depositToMarginAccount(wantAmount);
        } else if (unwindAmount < 0) {
            // the buffer is too high so reduce it to the correct size
            // open a perpetual short position using the unwindAmount
            mcLiquidityPool.trade(
                perpetualIndex,
                address(this),
                unwindAmount,
                price - slippageTolerance,
                block.timestamp,
                referrer,
                tradeMode
            );
            // withdraw funds from the margin account
            int256 withdrawAmount = (price * -unwindAmount) / 1e18;
            mcLiquidityPool.withdraw(
                perpetualIndex,
                address(this),
                withdrawAmount
            );
            // open a long position with the withdrawn funds
            _swap(uint256(withdrawAmount / DECIMAL_SHIFT), want, long);
        }
```

```
        positions.margin = getMargin();
        positions.unitAccumulativeFunding = getUnitAccumulativeFunding();
        positions.perpContracts = getMarginPositions();
        emit Remargined(unwindAmount);
    }
```

Listing 3.15: `BasisStrategy::remargin()`

Fourthly, the privileged functions in the `BasisStrategy` contract allow the `governance` to emergency exit the entire strategy and gather any liquidity mining rewards of `mcb` and transfer them to `governance`.

```
    /**
     * @notice  emergency exit the entire strategy in extreme circumstances
     *          unwind the strategy and send the funds to governance
     * @dev     only callable by governance
     */
    function emergencyExit() external onlyGovernance {
        // unwind strategy unless it is already unwound
        if (!isUnwind) {
            unwind();
        }
        uint256 wantBalance = IERC20(want).balanceOf(address(this));
        // send funds to governance
        IERC20(want).safeTransfer(governance, wantBalance);
        emit EmergencyExit(governance, wantBalance);
    }
```

Listing 3.16: `BasisStrategy::emergencyExit()`

```
    /**
     * @notice  gather any liquidity mining rewards of mcb and transfer them to
         governance
     *          further distribution
     * @param   epoch the epoch to claim rewards for
     * @param   amount the amount to redeem
     * @param   merkleProof the proof to use on the claim
     * @dev     only callable by governance
     */
    function gatherLMrewards(
        uint256 epoch,
        uint256 amount,
        bytes32[] memory merkleProof
    ) external onlyGovernance {
        lmClaimer.claimEpoch(epoch, amount, merkleProof);
        IERC20(mcb).safeTransfer(
            governance,
            IERC20(mcb).balanceOf(address(this))
        );
    }
```

Listing 3.17: `BasisStrategy::gatherLMrewards()`

Fifthly, the privileged function in the `BasisStrategy` contract allows the `owner/governance` to unwind the position in adverse funding rate scenarios.

```
    /**
     * @notice  unwind the position in adverse funding rate scenarios, settle short
         position
     *          and pull funds from the margin account. Then converts the long position
         back
     *          to want.
     * @dev     only callable by the owner
     */
    function unwind() public onlyAuthorised {
        require(!isUnwind, "unwound");
        isUnwind = true;
        mcLiquidityPool.forceToSyncState();
        // swap long asset back to want
        _swap(IERC20(long).balanceOf(address(this)), long, want);
        // check if the perpetual is in settlement, if it is then settle it
        // otherwise unwind the fund as normal.
        if (!_settle()) {
            // close the short position
            _closeAllPerpPositions();
            // withdraw all cash in the margin account
            mcLiquidityPool.withdraw(
                perpetualIndex,
                address(this),
                getMargin()
            );
        }
        // reset positions
        positions.perpContracts = 0;
        positions.margin = getMargin();
        positions.unitAccumulativeFunding = getUnitAccumulativeFunding();
        emit StrategyUnwind(IERC20(want).balanceOf(address(this)));
    }
```

Listing 3.18: `BasisStrategy::unwind()`

Sixthly, the `pause()` and `unpause()` functions allow for the `owner` to set the `BasisVault` contract state to be paused or unpaused. The `Vortex` users can only deposit or withdraw their `want` tokens when `BasisVault` contract is in unpaused state.

```
    /**
     * @notice pause the vault
     * @dev    only callable by the owner
     */
    function pause() external onlyOwner {
        _pause();
    }

    /**
     * @notice unpause the vault
     * @dev    only callable by the owner
```

```
    */
    function unpause() external onlyOwner {
        _unpause();
    }
```

Listing 3.19: `BasisVault::pause()/unpause()`

Lastly, the `updateLock()` function allows for the `owner` to register a `Vault` and the `deactivateVault()` function allows for the `owner` to deactivate a `Vault`.

```
    function registerVault(address _vault) external onlyOwner {
        require(_vault != address(0), "!_zeroAddress");
        isVault[_vault] = true;
        emit VaultRegistered(_vault);
    }

    function deactivateVault(address _vault) external onlyOwner {
        require(isVault[_vault], "!registered");
        isVault[_vault] = false;
        emit VaultDeactivated(_vault);
    }
```

Listing 3.20: `VaultRegistry::updateLock()`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the `owner/governance` 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 `owner/governance` explicit to `Vortex` users.

Status This issue has been confirmed.

3.6 Potential Sandwich-Based MEV With Imbalanced Positions

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `BasisStrategy`
- Category: Time and State [11]
- CWE subcategory: CWE-682 [5]

Description

As mentioned earlier, the protocol requires the timely invocation to rebalance current positions. Because of this rebalance need, there is a constant need of swapping one asset to another. With that, the protocol has provided two helper routines to facilitate the asset conversion: `_swap()` and `_swapTokenOut()`.

```
    function _swap(
        uint256 _amount,
        address _tokenIn,
        address _tokenOut
    ) internal returns (uint256 amountOut) {
        // set up swap params
        if (!isV2) {
            uint256 deadline = block.timestamp;
            address tokenIn = _tokenIn;
            address tokenOut = _tokenOut;
            uint24 fee = IUniswapV3Pool(pool).fee();
            address recipient = address(this);
            uint256 amountIn = _amount;
            uint256 amountOutMinimum = 0;
            uint160 sqrtPriceLimitX96 = 0;
            ISwapRouter.ExactInputSingleParams memory params = ISwapRouter
                .ExactInputSingleParams(
                    tokenIn,
                    tokenOut,
                    fee,
                    recipient,
                    deadline,
                    amountIn,
                    amountOutMinimum,
                    sqrtPriceLimitX96
                );
            // approve the router to spend the tokens
            IERC20(_tokenIn).safeApprove(router, _amount);
            // swap optimistically via the uniswap v3 router
            amountOut = ISwapRouter(router).exactInputSingle(params);
        } else {
            //get balance of tokenOut
            uint256 amountTokenOut = IERC20(_tokenOut).balanceOf(address(this));
            // set the swap params
            uint256 deadline = block.timestamp;
            address[] memory path;
            if (_tokenIn == weth   _tokenOut == weth) {
                path = new address[](2);
                path[0] = _tokenIn;
                path[1] = _tokenOut;
            } else {
                path = new address[](3);
                path[0] = _tokenIn;
                path[1] = weth;
                path[2] = _tokenOut;
            }
            // approve the router to spend the token
            IERC20(_tokenIn).safeApprove(router, _amount);
            IRouterV2(router).swapExactTokensForTokens(
```

```
                _amount,
                0,
                path,
                address(this),
                deadline
            );

            amountOut =
                IERC20(_tokenOut).balanceOf(address(this)) -
                amountTokenOut;
        }
    }
```

Listing 3.21: BasisStrategy :: _swap()

To elaborate, we show above the `_swap()` helper routine. We notice the conversion is routed to `UniswapV2/UniswapV3` in order to swap one asset to another. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of conversion.

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

Recommendation Develop an effective mitigation (e.g., slippage control) to the above sandwich attacks to better protect the interests of protocol users.

Status

4 | Conclusion

In this audit, we have analyzed the `Vortex` design and implementation. The `Vortex` protocol is an on-chain basis trading strategy that aims to generate long-term, sustainable and rewarding yields while remaining market-neutral. 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-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

[5] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

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

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

[8] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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