SMART CONTRACT AUDIT REPORT

for

Velvet Capital V2

Prepared By: Xiaomi Huang

PeckShield

October 9, 2023

Document Properties

Client	Velvet
Title	Smart Contract Audit Report
Target	Velvet Capital V2
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Jinzhuo Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 9, 2023	Xuxian Jiang	Final Release
1.0-rc	October 6, 2023	Xuxian Jiang	Release Candidate

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Velvet Capital V2` 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 Velvet Capital V2

`Velvet Capital V2` is a DeFi protocol that helps users and institutions create tokenized index funds, portfolios and other financial products with additional yield. The protocol provides all the necessary infrastructure for financial product development being integrated with `AMMs`, lending protocols and other `DeFi` primitives to give users a diverse asset management toolkit. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Velvet Capital V2

Item	Description
Target	Velvet Capital V2
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 9, 2023

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the `Velvet Capital V2` protocol assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/Velvet-Capital/protocol-v2-public.git (32452f2)

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

- https://github.com/Velvet-Capital/protocol-v2-public.git (1d33538)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, 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) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

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 `Velvet Capital V2` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	6	■■■■■■
Low	1	■
Informational	0	
Total	8	

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 that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 6 medium-severity vulnerabilities, and 1 low-severity vulnerability.

Table 2.1: Key Velvet Capital V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improper Cooldown Calculation in IndexSwap	Business Logic	Resolved
PVE-002	Medium	Improper Slippage Control Enforcement in SlippageControl	Business Logic	Resolved
PVE-003	Low	Improved Initialization in IndexSwap	Coding Practices	Resolved
PVE-004	Medium	Potential Read-Only Reentrancy Risk in AlpacaHandler	Time and State	Resolved
PVE-005	High	Incorrect Mint Share Calculation in BeefyHandler	Business Logic	Resolved
PVE-006	Medium	Incorrect getTokenBalanceUSD() Logic in BeefyLPHandler	Business Logic	Resolved
PVE-007	Medium	Improper Public Exposure of Token-Approving Function	Security Features	Resolved
PVE-008	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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

3 | Detailed Results

3.1 Improper Cooldown Calculation in IndexSwap

- ID: PVE-001
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: IndexSwapLibrary
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the Velvet Capital V2 protocol, the IndexSwap contract is designed to accept the investment of the supported assets. Each user investment will effectively update the user's lastInvestmentTime and lastWithdrawCooldown. While examining the lastWithdrawCooldown calculation, we observe the computed timestamp needs to be revised.

To elaborate, we show below the related code snippet in calculateCooldown(). The lockup cooldown is computed and then applied to the investor after pool deposit. The computation considers existing cooldown for invested amount and new cooldown for the new investment and calculates the weighted average cooldown. In other words, the new lockup cooldown should be computed as (_mintedLiquidity * _currentCooldownTime + balanceBeforeMint * prevCooldownRemaining)/ _currentUserBalance, instead of current computation (lines 484 – 486).

```
460   function calculateCooldown(
461     uint256 currentBalance,
462     uint256 liquidityMinted,
463     uint256 newCooldown,
464     uint256 lastCooldown,
465     uint256 lastDepositTime
466   ) external view returns (uint256 cooldown) {
467     // Get timestamp when current cooldown ends
468     uint256 cooldownEndsAt = lastDepositTime + lastCooldown;
469     // Current exit remaining cooldown
470     uint256 remainingCooldown = cooldownEndsAt < block.timestamp ? 0 : cooldownEndsAt -
          block.timestamp;
```

```
    // If it's first deposit with zero liquidity, no cooldown should be applied
    if (currentBalance == 0 && liquidityMinted == 0) {
      cooldown = 0;
      // If it's first deposit, new cooldown should be applied
    } else if (currentBalance == 0) {
      cooldown = newCooldown;
      // If zero liquidity or new cooldown reduces remaining cooldown, apply remaining
    } else if (liquidityMinted == 0  newCooldown < remainingCooldown) {
      cooldown = remainingCooldown;
      // For the rest cases calculate cooldown based on current balance and liquidity
          minted
    } else {
      // If the user already owns liquidity, the additional lockup should be in
          proportion to their existing liquidity.
      // Calculated as newCooldown * liquidityMinted / currentBalance
      uint256 additionalCooldown = (newCooldown * liquidityMinted) / currentBalance;
      // Aggregate additional and remaining cooldowns
      uint256 aggregatedCooldown = additionalCooldown + remainingCooldown;
      // Resulting value is capped at new cooldown time (shouldn't be bigger) and falls
          back to one second in case of zero
      cooldown = aggregatedCooldown > newCooldown ? newCooldown : aggregatedCooldown !=
          0 ? aggregatedCooldown : 1;
    }
  }
```

Listing 3.1: `IndexSwapLibrary::calculateCooldown()`

Recommendation Revise the lockup cooldown calculation in the above `calculateCooldown`.

Status The issue has been addressed in the following commit: `e7232f8`.

3.2 Improper Slippage Control Enforcement in SlippageControl

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `SlippageControl`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

To effectively facilitate the token swaps, `Velvet Capital V2` has developed a key `SlippageControl` contract to validate the resulting slippage within the permitted range. However, our analysis shows the slippage enforcement may have a flawed implementation.

To elaborate, we show below the related code snippet of the `_validateLPSlippage()` routine. It has a basic logic to ensure current slippage is no larger than the given `_lpSlippage`. We notice the

the deviation on the computed amount is already normalized to have 18 decimals and the price deviation is also normalized to have 18 decimals. However, the deviation needs to be computed as a percentage, instead of the absolute value from the amount (or price) difference. Moreover, the given `_lpSlippage` has the 4 decimals, which justifies the need to revise the check (line 73) to be the following: `if (absoluteValue * (10 ** 4)> (_lpSlippage * (10 ** 18)))`.

```
47   function _validateLPSlippage(
48     uint _amountA,
49     uint _amountB,
50     uint _priceA,
51     uint _priceB,
52     uint _lpSlippage
53   ) internal view {
54     if (maxSlippage < _lpSlippage) {
55       revert ErrorLibrary.InvalidLPSlippage();
56     }
57
58     /**
59      *  amountA * priceA = amountB * priceB ( in ideal scenario )
60      *  amountA/amountB - priceB/priceA = 0
61      *  When the amount of either token is not fully accepted then the
62      *  amountA and amountB wont be equal to 0 and that becomes our lpSlippage
63      */
64
65     uint amountDivision = (_amountA * (10 ** 18)) / (_amountB);
66     uint priceDivision = (_priceB * (10 ** 18)) / (_priceA);
67     uint absoluteValue = 0;
68     if (amountDivision > priceDivision) {
69       absoluteValue = amountDivision - priceDivision;
70     } else {
71       absoluteValue = priceDivision - amountDivision;
72     }
73     if (absoluteValue * (10 ** 2) > (_lpSlippage * (10 ** 18))) {
74       revert ErrorLibrary.InvalidAmount();
75     }
76   }
```

Listing 3.2: `SlippageControl:_validateLPSlippage()`

Note another routine, i.e., `WombatHandler::_getInternalSlippage()`, can also benefit from the improved slippage control.

Recommendation Improve the above-mentioned routines by adding effective slippage control.

Status The issue has been addressed in the following commit: `1d33538`.

3.3 Improved Initialization in IndexSwap

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `IndexSwap`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

In the `Velvet Capital V2` protocol, the `IndexSwap` contract is designed to accept the investment of the supported assets. In particular, it inherits from a few parent contracts, including `Initializable`, `ERC20Upgradeable`, `UUPSUpgradeable`, `OwnableUpgradeable`, and `CommonReentrancyGuard`. Our analysis shows its initialization logic can be improved.

To illustrate, we show below the initialization routine `init()` of `IndexSwap`. It does properly initialize most inherited parent contracts. However, it forgot to invoke `__ReentrancyGuard_init()` to initialize the reentrancy `_status` as `_NOT_ENTERED`. Though it does not lock up the contract execution, there is still a need to properly initialize the reentrancy status.

```
function init(FunctionParameters.IndexSwapInitData calldata initData) external
    initializer {
  __ERC20_init(initData._name, initData._symbol);
  __UUPSUpgradeable_init();
  __Ownable_init();
  _vault = initData._vault;
  _module = initData._module;
  _accessController = IAccessController(initData._accessController);
  _tokenRegistry = ITokenRegistry(initData._tokenRegistry);
  _oracle = IPriceOracle(initData._oracle);
  _exchange = IExchange(initData._exchange);
  _iAssetManagerConfig = IAssetManagerConfig(initData._iAssetManagerConfig);
  _feeModule = IFeeModule(initData._feeModule);
  WETH = _tokenRegistry.getETH();
}
```

Listing 3.3: `IndexSwap::init()`

Recommendation Revise the above `init()` routine by adding the call to `__ReentrancyGuard_init()`.

Status The issue has been addressed in the following commit: `e7232f8`.

3.4 Potential Read-Only Reentrancy Risk in AlpacaHandler

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `AlpacaHandler`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The `Velvet Capital V2` protocol supports `Alpaca` via the specific `AlpacaHandler` contract. While examining the support, we notice it may lead to a read-only reentrancy issue.

To elaborate, we show below the related code snippet of the `AlpacaHandler` contract. The affected function is `getUnderlyingBalance()`, which aims to return the underlying asset balance of the passed address. However, the `IVaultAlpaca(t).totalToken()` may be affected if the `Alpaca` protocol is in the middle of its strategy execution and the affected `IVaultAlpaca(t).totalToken()` amount cascadingly affects the `AlpacaHandler::getUnderlyingBalance()` result, which impacts a number of other protocol-wide routines in the investment/redemption/rebalance operations.

```
function getUnderlyingBalance(address _tokenHolder, address t) public view override
    returns (uint256[] memory) {
  if (t == address(0)  _tokenHolder == address(0)) {
    revert ErrorLibrary.InvalidAddress();
  }
  uint256[] memory tokenBalance = new uint256[](1);
  uint256 yieldTokenBalance = getTokenBalance(_tokenHolder, t);
  tokenBalance[0] = (yieldTokenBalance * IVaultAlpaca(t).totalToken()) / IVaultAlpaca(
      t).totalSupply();
  return tokenBalance;
}
```

Listing 3.4: `AlpacaHandler::getUnderlyingBalance()`

Recommendation Ensure the above `getUnderlyingBalance()` routine will not be executed in the middle of `Alpaca` strategy execution.

Status The issue has been addressed by removing its support in the following commit: `e7232f8`.

3.5 Incorrect Mint Share Calculation in BeefyHandler

- ID: PVE-005
- Severity: High
- Likelihood: High
- Impact: High
- Target: `BeefyHandler`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

The `Velvet Capital V2` protocol integrates the support of a number of external protocols via protocol-specific `handler`. In the process of examining the `BeefyLP` support, we notice the related `BeefyHandler` has an incorrect deposit logic that may lead to incorrect share calculation.

To elaborate, we show below the related `deposit()` routine. This routine has a rather straight-forward logic in transferring the funds into `VaultBeefy` and minting the pro-rata share based on the deposited amount. However, it comes to our attention that when the funds are in `MOO_VENUS_BNB`, the returned `_mintedAmount` amount is computed based on the `_amount[0]` (line 82), instead of the actual `msg.value`. As a result, a malicious actor may abuse this issue to obtain unreasonably large share with a smaller deposit. And the large share may then be redeemed to steal the vault funds.

```
50   function deposit(
51     address _mooAsset,
52     uint256[] calldata _amount,
53     uint256 _lpSlippage,
54     address _to,
55     address user
56   ) public payable override returns (uint256 _mintedAmount) {
57     if (_mooAsset == address(0)  _to == address(0)) {
58       revert ErrorLibrary.InvalidAddress();
59     }
60     IVaultBeefy asset = IVaultBeefy(_mooAsset);
61     IERC20Upgradeable underlyingToken = IERC20Upgradeable(getUnderlying(_mooAsset)[0]);
62     if (msg.value == 0) {
63       TransferHelper.safeApprove(address(underlyingToken), address(_mooAsset), 0);
64       TransferHelper.safeApprove(address(underlyingToken), address(_mooAsset), _amount
         [0]);
65       asset.deposit(_amount[0]);
66       if (_to != address(this)) {
67         uint256 assetBalance = IERC20Upgradeable(_mooAsset).balanceOf(address(this));
68         TransferHelper.safeTransfer(_mooAsset, _to, assetBalance);
69       }
70     } else {
71       if (_mooAsset != MOO_VENUS_BNB) {
72         revert ErrorLibrary.PleaseDepositUnderlyingToken();
73       }
74
```

```
      asset.depositBNB{value: msg.value}();
      if (_to != address(this)) {
        uint256 assetBalance = IERC20Upgradeable(_mooAsset).balanceOf(address(this));
        TransferHelper.safeTransfer(_mooAsset, _to, assetBalance);
      }
    }
    emit Deposit(msg.sender, _mooAsset, _amount, _to);
    _mintedAmount = _oracle.getPriceTokenUSD18Decimals(address(underlyingToken), _amount
        [0]);
  }
```

Listing 3.5: `BeefyHandler::deposit()`

Recommendation Correct the above issue by enforcing `_amount[0]` is equal to `msg.value` when `_mooAsset == MOO_VENUS_BNB`.

Status The issue has been addressed in the following commit: `e7232f8`.

3.6 Incorrect getTokenBalanceUSD() Logic in BeefyLPHandler

- ID: PVE-006
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `BeefyLPHandler`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

The `Velvet Capital V2` protocol integrates the support of a number of external protocols via protocol-specific `handler`. In the process of examining the `BeefyLP` support, we notice the related `BeefyLPHandler` needs to be revised.

To elaborate, we show below the implementation from the affected `getTokenBalanceUSD()` routine. As the name indicates, this routine returns the USD value of the asset balance. However, the asset balance is currently computed as `_getTokenBalance(_tokenHolder, t)`, which does not take into account `t.getPricePerFullShare()`. As a result, the computed token balance in USD is smaller than the actual amount.

```
  function getTokenBalanceUSD(address _tokenHolder, address t) public view override
      returns (uint256) {
    if (t == address(0)  _tokenHolder == address(0)) {
      revert ErrorLibrary.InvalidAddress();
    }
    address underlyingLpToken = address(IStrategy(address(IVaultBeefy(t).strategy())).
        want());
```

```
    return _calculatePriceForBalance(underlyingLpToken, address(_oracle),
        _getTokenBalance(_tokenHolder, t));
  }
```

Listing 3.6: `BeefyLPHandler::getTokenBalanceUSD()`

Recommendation Revise the above `BeefyLPHandler` to properly compute the right asset balance in USD.

Status The issue has been addressed in the following commit: `e7232f8`.

3.7 Improper Public Exposure of Token-Approving Function

- ID: PVE-007
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `ApproveControl`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Velvet Capital V2` protocol, there is a `ApproveControl` contract that is designed to manage the token spending approvals. However, we notice the key function `setAllowance()` is publicly exposed and allows any unauthorized users to change the allowance.

To elaborate, we show below the related code snippet of the `ApproveControl` contract. By design, the `setAllowance()` routine can be only called internally within the `ExternalSwapHandler` contracts, including `OneInchHandler`, `ZeroExHandler`, and `ParaswapHandler`. In other words, the current definition of being public should be revised to be internal.

```
  function setAllowance(address _token, address _spender, uint256 _sellAmount) public {
    uint256 _currentAllowance = IERC20Upgradeable(_token).allowance(address(this),
        _spender);
    if (_currentAllowance != _sellAmount) {
      IERC20Upgradeable(_token).safeDecreaseAllowance(_spender, _currentAllowance);
      IERC20Upgradeable(_token).safeIncreaseAllowance(_spender, _sellAmount);
    }
  }
```

Listing 3.7: `ApproveControl::setAllowance()`

Recommendation Revise the `setAllowance()` definition to be internal.

Status The issue has been addressed in the following commit: `e7232f8`.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Velvet Capital V2` protocol, there are a series of privileged accounts that play a critical role in governing and regulating the protocol-wide operations (e.g., configure various system parameters and update price oracle). In the following, we show the representative functions potentially affected by the privilege of the accounts.

```
function _addFeed(
  address[] memory base,
  address[] memory quote,
  AggregatorV2V3Interface[] memory aggregator
) public onlyOwner {
  if (!((base.length == quote.length) && (quote.length == aggregator.length)))
    revert ErrorLibrary.IncorrectArrayLength();

  for (uint256 i = 0; i < base.length; i++) {
    if(base[i] == address(0))
      revert ErrorLibrary.InvalidAddress();
    if(quote[i] == address(0))
      revert ErrorLibrary.InvalidAddress();
    if((address(aggregator[i])) == address(0))
      revert ErrorLibrary.InvalidAddress();

    if (aggregatorAddresses[base[i]].aggregatorInterfaces[quote[i]] !=
        AggregatorInterface(address(0))) {
      revert AggregatorAlreadyExists();
    }
    aggregatorAddresses[base[i]].aggregatorInterfaces[quote[i]] = aggregator[i];
  }
  emit addFeed(base, quote, aggregator);
}

/**
 * @notice Update an existing feed
 * @param base base asset address
 * @param quote quote asset address
 * @param aggregator aggregator
 */
function _updateFeed(address base, address quote, AggregatorV2V3Interface aggregator)
    public onlyOwner {
  if(base == address(0))
```

```
      revert ErrorLibrary.InvalidAddress();
    if(quote == address(0))
      revert ErrorLibrary.InvalidAddress();
    if((address(aggregator)) == address(0))
      revert ErrorLibrary.InvalidAddress();

    aggregatorAddresses[base].aggregatorInterfaces[quote] = aggregator;
    emit updateFeed(base, quote, address(aggregator));
  }
```

Listing 3.8: `PriceOracle::_addFeed()&&_updateFeed()`

```
  function upgradeIndexSwap(address[] calldata _proxy, address _newImpl) external
      virtual onlyOwner {
    _setBaseIndexSwapAddress(_newImpl);
    _upgrade(_proxy, _newImpl);
    emit UpgradeIndexSwap(_newImpl);
  }

  /**
   * @notice This function is used to upgrade the Exchange contract
   * @param _proxy Proxy address
   * @param _newImpl New implementation address
   */
  function upgradeExchange(address[] calldata _proxy, address _newImpl) external virtual
       onlyOwner {
    _setBaseExchangeHandlerAddress(_newImpl);
    _upgrade(_proxy, _newImpl);
    emit UpgradeExchange(_newImpl);
  }

  /**
   * @notice This function is used to upgrade the AssetManagerConfig contract
   * @param _proxy Proxy address
   * @param _newImpl New implementation address
   */
  function upgradeAssetManagerConfig(address[] calldata _proxy, address _newImpl)
      external virtual onlyOwner {
    _setBaseAssetManagerConfigAddress(_newImpl);
    _upgrade(_proxy, _newImpl);
    emit UpgradeAssetManagerConfig(_newImpl);
  }

  /**
   * @notice This function is used to upgrade the OffChainRebalance contract
   * @param _proxy Proxy address
   * @param _newImpl New implementation address
   */
  function upgradeOffchainRebalance(address[] calldata _proxy, address _newImpl)
      external virtual onlyOwner {
    _setBaseOffChainRebalancingAddress(_newImpl);
    _upgrade(_proxy, _newImpl);
    emit UpgradeOffchainRebalance(_newImpl);
```

```
}
```

Listing 3.9: `IndexFactory::upgradeIndexSwap()&&upgradeExchange()`

```
function updateWeights(
  uint96[] calldata denorms,
  uint256[] calldata _slippage,
  uint256[] calldata _lpSlippage,
  address _swapHandler
) external virtual nonReentrant onlyAssetManager {
  address[] memory tokens = getTokens();
  validateUpdate(_swapHandler);
  if (denorms.length != tokens.length) {
    revert ErrorLibrary.LengthsDontMatch();
  }
  if (tokens.length != _slippage.length  tokens.length != _lpSlippage.length) {
    revert ErrorLibrary.InvalidSlippageLength();
  }
  index.updateRecords(tokens, denorms);
  rebalance(_slippage, _lpSlippage, _swapHandler);
  emit UpdatedWeights(denorms);
}

/**
 * @notice The function rebalances the portfolio to the updated tokens with the
     updated weights
 * @param inputData The input calldata passed to the function
 */
function updateTokens(
  FunctionParameters.UpdateTokens calldata inputData
) external virtual nonReentrant onlyAssetManager {
  address[] memory _tokens = getTokens();
  validateUpdate(inputData._swapHandler);
  ...
  }
```

Listing 3.10: `Rebalancing::updateWeights()&&updateTokens()`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. The `multi-sig` mechanism could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest to introduce the `multi-sig` mechanism to manage all the privileged accounts to mitigate this issue. Additionally, all changes to privileged operations may need to be mediated with necessary timelocks.

Status The issue has been confirmed by the team. The teams intends to make use of `multi-sig` to mitigate this issue.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Velvet Capital V2` protocol, which is a DeFi protocol that helps users and institutions create tokenized index funds, portfolios and other financial products with additional yield. The protocol provides all the necessary infrastructure for financial product development being integrated with `AMMs`, lending protocols and other `DeFi` primitives to give users a diverse asset management toolkit. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

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

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

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

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

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

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

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