SMART CONTRACT AUDIT REPORT

for

Velvet Capital (V4)

Prepared By: Xiaomi Huang

PeckShield

December 17, 2024

Document Properties

Client	Velvet
Title	Smart Contract Audit Report
Target	Velvet Capital V4
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 17, 2024	Xuxian Jiang	Final Release
1.0-rc	December 16, 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 V4` 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 V4

`Velvet Capital V4` 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 V4

Item	Description
Target	Velvet Capital V4
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	December 17, 2024

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 V4` 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/Velvet-v4.git (139d411)

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/Velvet-v4.git (cc24e50f7)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

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

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a 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) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

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 V4` 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	0	
Medium	5	■■■■■
Low	4	■■■■
Informational	0	
Total	9	

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 5 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key Velvet Capital V4 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incorrect Fund Source Validation in VaultManager	Business Logic	Resolved
PVE-002	Medium	Improper Dust Funds Return in PositionMangers	Business Logic	Resolved
PVE-003	Low	Incorrect updateGnosisAddresses() Logic in PortfolioFactory	Business Logic	Resolved
PVE-004	Low	Revisited Fee Update Logic in FeeManagement	Business Logic	Resolved
PVE-005	Low	Revisited UniswapV3 Support in Position Manager Wrapper	Business Logic	Resolved
PVE-006	Medium	Incorrect Performance Fee Calculation in FeeCalculations	Business Logic	Resolved
PVE-007	Medium	Incorrect Parameters in BorrowManager::repayBorrow()	Coding Practices	Resolved
PVE-008	Medium	Incorrect Repayment Tx Preparation in VenusAssetHandler	Business Logic	Resolved
PVE-009	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 Incorrect Fund Source Validation in VaultManager

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `VaultManager`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

In the `Velvet Capital V4` protocol, there is a key contract `VaultManager` contract that extends functionality for managing deposits and withdrawals in the vault. While examining the deposit logic, we observe current implementation has an issue that needs to be fixed.

To elaborate, we show below the code snippet from the related `_multiTokenTransferWithPermit()`. It is an internal helper routine that processes multi-token deposits and calculates the minimum deposit ratio. Our analysis shows two internal calls, i.e., `permit2.permit()` (line 621) and `portfolioTokens[i].allowance()` (line 628), should be based on the given parameter of `_depositFor`, not the calling user `msg.sender`.

```
607   function _multiTokenTransferWithPermit(
608     uint256[] calldata depositAmounts,
609     IAllowanceTransfer.PermitBatch calldata _permit,
610     bytes calldata _signature,
611     address _depositFor
612   ) internal returns (uint256) {
613     // Validate deposit amounts and get initial token balances
614     (
615       uint256 amountLength,
616       address[] memory portfolioTokens,
617       uint256[] memory tokenBalancesBefore,
618       TokenBalanceLibrary.ControllerData[] memory controllersData
619     ) = _validateAndGetBalances(depositAmounts);
620
621     try permit2.permit(msg.sender, _permit, _signature) {
```

```
      // No further implementation needed if permit succeeds
    } catch {
      // Check allowance for each token in depositAmounts array
      uint256 depositAmountsLength = depositAmounts.length;
      for (uint256 i; i < depositAmountsLength; i++) {
        if (
          IERC20Upgradeable(portfolioTokens[i]).allowance(
            msg.sender,
            address(this)
          ) < depositAmounts[i]
        ) revert ErrorLibrary.InsufficientAllowance();
      }
    }

    // Handles the token transfer and minRatio calculations
    return
      _handleTokenTransfer(
        _depositFor,
        amountLength,
        depositAmounts,
        portfolioTokens,
        tokenBalancesBefore,
        true,
        controllersData
      );
  }
```

Listing 3.1: `VaultManager::_multiTokenTransferWithPermit()`

Recommendation Revise the above routine to ensure the proper parameters are given for the two internal calls.

Status The issue has been resolved as the team confirms the `_depositFor` variable is identical as `msg.sender`.

3.2 Improper Dust Funds Return in PositionMangers

- ID: PVE-002
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `PositionMangers`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

To effectively facilitate the interaction with external DEX engines, `Velvet Capital V4` has developed a wrapper contract `PositionManagerAbstract`. The wrapper contract streamlines the process of asset deposits and withdrawals. In the process of examining the asset flow, we notice an issue that occurs when refunding dust funds back to the user.

To elaborate, we show below the related code snippet of the `increaseLiquidity()` routine. It has a basic logic to increase liquidity in an existing `Uniswap V3` position and mints corresponding wrapper tokens. We notice the final dust tokens are refunded back to the calling user. However, the return amount should be `balance0After` and `balance1After`, not current `balance0After - balance0Before` (line 198) and `balance1After - balance1Before` (line 199).

```
125  function increaseLiquidity(
126    WrapperFunctionParameters.WrapperDepositParams memory _params
127  ) external notPaused nonReentrant {
128    if (
129      address(_params._positionWrapper) == address(0)
130      _params._dustReceiver == address(0)
131    ) revert ErrorLibrary.InvalidAddress();
132
133    uint256 tokenId = _params._positionWrapper.tokenId();
134    address token0 = _params._positionWrapper.token0();
135    address token1 = _params._positionWrapper.token1();
136
137    // Reinvest any collected fees back into the pool before adding new liquidity.
138    _collectFeesAndReinvest(
139      _params._positionWrapper,
140      tokenId,
141      token0,
142      token1,
143      _params._tokenIn,
144      _params._tokenOut,
145      _params._amountIn
146    );
147
148    // Track token balances before the operation to calculate dust later.
149    uint256 balance0Before = IERC20Upgradeable(token0).balanceOf(address(this));
150    uint256 balance1Before = IERC20Upgradeable(token1).balanceOf(address(this));
151
152    // Transfer the desired liquidity tokens from the caller to this contract.
153    _transferTokensFromSender(
154      token0,
155      token1,
156      _params._amount0Desired,
157      _params._amount1Desired
158    );
159
160    uint256 balance0After = IERC20Upgradeable(token0).balanceOf(address(this));
161    uint256 balance1After = IERC20Upgradeable(token1).balanceOf(address(this));
162
163    _params._amount0Desired = balance0After - balance0Before;
164    _params._amount1Desired = balance1After - balance1Before;
165
166    // Approve the Uniswap manager to use the tokens for liquidity.
```

```
    _approveNonFungiblePositionManager(
      token0,
      token1,
      _params._amount0Desired,
      _params._amount1Desired
    );

    // Increase liquidity at the position.
    (uint128 liquidity, , ) = uniswapV3PositionManager.increaseLiquidity(
      INonfungiblePositionManager.IncreaseLiquidityParams({
        tokenId: tokenId,
        amount0Desired: _params._amount0Desired,
        amount1Desired: _params._amount1Desired,
        amount0Min: _params._amount0Min,
        amount1Min: _params._amount1Min,
        deadline: block.timestamp
      })
    );

    // Mint wrapper tokens corresponding to the liquidity added.
    _mintTokens(_params._positionWrapper, tokenId, liquidity);

    // Calculate token balances after the operation to determine any remaining dust.
    balance0After = IERC20Upgradeable(token0).balanceOf(address(this));
    balance1After = IERC20Upgradeable(token1).balanceOf(address(this));

    // Return any dust to the caller.
    _returnDust(
      _params._dustReceiver,
      token0,
      token1,
      balance0After - balance0Before,
      balance1After - balance1Before
    );
    ...
  }
```

Listing 3.2: `PositionManagerAbstract:increaseLiquidity()`

Note other contracts are also affected, namely `PositionMangerAbstractUniswap` and `PositionManagerAbstractAlgebra`
.

Recommendation Improve the above-mentioned routines by properly refunding users the dust tokens.

Status The issue has been addressed in the following PR: `43`.

3.3 Incorrect updateGnosisAddresses() Logic in PortfolioFactory

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PortfolioFactory`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In the `Velvet Capital V4` protocol, the `PortfolioFactory` contract is designed to instantiate the `Gnosis`-based portfolio contracts. Our analysis shows it has a flawed setter function `updateGnosisAddresses()`.

To illustrate, we show below the implementation of this setter function `updateGnosisAddresses()`. As the name indicates, it is used to update `Gnosis` deployment addresses. However, the validation should be performed to ensure they are not `address(0)`. However, current implementation ensures all input addresses must be `address(0)`.

```
function updateGnosisAddresses(
  address _newGnosisSingleton,
  address _newGnosisFallbackLibrary,
  address _newGnosisMultisendLibrary,
  address _newGnosisSafeProxyFactory
) external virtual onlyOwner {
  if (
    _newGnosisSingleton != address(0) ||
    _newGnosisFallbackLibrary != address(0) ||
    _newGnosisMultisendLibrary != address(0) ||
    _newGnosisSafeProxyFactory != address(0)
  ) revert ErrorLibrary.InvalidAddress();
  gnosisSingleton = _newGnosisSingleton;
  gnosisFallbackLibrary = _newGnosisFallbackLibrary;
  gnosisMultisendLibrary = _newGnosisMultisendLibrary;
  gnosisSafeProxyFactory = _newGnosisSafeProxyFactory;

  emit UpdateGnosisAddresses(
    _newGnosisSingleton,
    _newGnosisFallbackLibrary,
    _newGnosisMultisendLibrary,
    _newGnosisSafeProxyFactory
  );
}
```

Listing 3.3: `PortfolioFactory::updateGnosisAddresses()`

Recommendation Revise the above `updateGnosisAddresses()` routine by properly update `Gnosis` deployment addresses.

Status The issue has been addressed in the following PR: 43.

3.4 Revisited Fee Update Logic in FeeManagement

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: FeeManagement
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The Velvet Capital V4 protocol allows the governance to dynamically configure a number of risk parameters and fee settings. While reviewing the update logic to related parameters and fees, we notice the update of the fee settings warrants the need of refreshing the latest fee rate.

To elaborate, we show below the updateManagementFee() function. It implements a rather straightforward logic in validating and applying the new management fee. It comes to our attention that the internal accounting for protocol and management fees needs to be timely refreshed before applying the new fees.

```
function updateManagementFee() external onlyAssetManager {
  if (proposedManagementFeeTime == 0) revert ErrorLibrary.NoNewFeeSet();

  if (block.timestamp < (proposedManagementFeeTime + 28 days))
    revert ErrorLibrary.TimePeriodNotOver();

  managementFee = newManagementFee;
  proposedManagementFeeTime = 0;

  feeModule.chargeProtocolAndManagementFees();

  emit UpdateManagementFee(newManagementFee);
}
```

Listing 3.4: FeeManagement::updateManagementFee()

Recommendation Ensure the above updateManagementFee() routine will timely collect protocol and management fee before new fees are applied.

Status The issue has been addressed in the following PR: 43.

3.5 Revisited UniswapV3 Support in Position Manager Wrapper

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SwapVerificationLibrary
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

As mentioned earlier (Section 3.2), Velvet Capital V4 has developed a wrapper contract PositionManagerAbstract to streamline the process of asset deposits and withdrawals. In the process of examining the abstract logic in PositionManagerAbstract, we notice that the wrapper contract only supports Algebra, not UniswapV3.

To elaborate, we show below the related _getUnderlyingAmounts() routine. This routine has a rather straightforward logic in calculating the underlying tokens from the wrapped position. It comes to our attention that the pool is computed with two parameters token0 and token1. While it perfectly supports the pool calculation in Algebra, it does not compute the pool address in UniswapV3 without the extra third parameter of fee.

```
function _getUnderlyingAmounts(
  IPositionWrapper _positionWrapper,
  address _factory,
  uint160 sqrtRatioAX96,
  uint160 sqrtRatioBX96,
  uint128 _existingLiquidity
) internal returns (uint256 amount0, uint256 amount1) {
  IFactory factory = IFactory(_factory);
  IPool pool = IPool(
    factory.poolByPair(_positionWrapper.token0(), _positionWrapper.token1())
  );

  int24 tick = pool.globalState().tick;
  uint160 sqrtRatioX96 = TickMath.getSqrtRatioAtTick(tick);

  (amount0, amount1) = LiquidityAmounts.getAmountsForLiquidity(
    sqrtRatioX96,
    sqrtRatioAX96,
    sqrtRatioBX96,
    _existingLiquidity
  );
}
```

Listing 3.5: LiquidityAmountsCalculations::_getUnderlyingAmounts()

Recommendation Correct the above issue by properly abstracting the pool calculation in both `Algebra` and `UniswapV3`.

Status The issue has been addressed in the following PR: `52`.

3.6 Incorrect Performance Fee Calculation in FeeCalculations

- ID: PVE-006
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `FeeCalculations`
- Category: Business LogicPortfolioCalculationsciteCWE-840
- CWE subcategory: CWE-837 [3]

Description

The `Velvet Capital V4` protocol has the need of calculating the portfolio value for each vault. In the process of examining the share calculation when minting respective performance fee, we notice the related fee calculation should be improved.

To elaborate, we show below the implementation from the affected `_calculatePerformanceFeeToMint()` routine. As the name indicates, this routine computes the performance fee to mint based on the high watermark principle. According to current implementation, the performance fee is computed as `((performanceIncrease * _totalSupply * _feePercentage)* ONE_ETH_IN_WEI)/ TOTAL_WEIGHT` (lines 191–193), which should be revised as `((performanceIncrease * _totalSupply * _feePercentage)/ ONE_ETH_IN_WEI)/ TOTAL_WEIGHT`.

```
186  function _calculatePerformanceFeeToMint(
187    uint256 _currentPrice,
188    uint256 _highWaterMark,
189    uint256 _totalSupply,
190    uint256 _vaultBalance,
191    uint256 _feePercentage
192  ) internal pure returns (uint256 tokensToMint) {
193    if (_currentPrice <= _highWaterMark) {
194      return 0; // No fee if current price is below or equal to high watermark
195    }
196
197    uint256 performanceIncrease = _currentPrice - _highWaterMark;
198    uint256 performanceFee = ((performanceIncrease *
199      _totalSupply *
200      _feePercentage) * ONE_ETH_IN_WEI) / TOTAL_WEIGHT;
201
202    tokensToMint =
203      (performanceFee * _totalSupply) /
204      (_vaultBalance - performanceFee);
```

```
}
```

Listing 3.6: `FeeCalculations::_calculatePerformanceFeeToMint()`

Recommendation Revise the above routine to properly compute the performance fee.

Status The issue has been addressed in the following PR: `52`.

3.7 Incorrect Parameters in BorrowManager::repayBorrow()

- ID: PVE-007
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `BorrowManager`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

In `Velvet Capital V4`, the portfolio may borrow funds from external lending protocol. While reviewing the related loan repayment logic, we notice an issue that does not properly pass parameters for flashloan-based repay.

To elaborate, we show below the related code snippet of the `repayBorrow()` routine. By design, this routine handles the repayment of borrowed tokens during withdrawal. We notice the repayment is performed via a flashloan, which is passed with the incorrect parameter orders. In particular, the last two parameters are currently passed in the order of `repayData` and `borrowedTokens`. Their orders need to be reversed.

```
    function repayBorrow(
        uint256 _portfolioTokenAmount,
        uint256 _totalSupply,
        FunctionParameters.withdrawRepayParams calldata repayData
    ) external onlyPortfolioManager {
        // Get all supported controllers from the protocol configuration
        // There can be multiple controllers from Venus side, hence the loop
        address[] memory controllers = _protocolConfig
            .getSupportedControllers();

        beforeRepayVerification(
            repayData._factory,
            repayData._solverHandler,
            repayData._bufferUnit
        );

        // Iterate through all controllers to repay borrows for each
        for (uint j; j < controllers.length; j++) {
```

```
            address _controller = controllers[j];

            // Get the asset handler for the current controller
            IAssetHandler assetHandler = IAssetHandler(
                _protocolConfig.assetHandlers(_controller)
            );

            (, address[] memory borrowedTokens) = assetHandler.getAllProtocolAssets(
                _vault,
                _controller
            ); // Get all borrowed tokens for the vault under the controller

            // Check if there are any borrowed tokens
            if(borrowedTokens.length == 0) continue; // If no borrowed tokens, skip to the next controller


            // Prepare the data for the flash loan execution
            bytes memory data = abi.encodeWithSelector(
                IAssetHandler.executeUserFlashLoan.selector,
                _vault,
                address(this),
                _portfolioTokenAmount,
                _totalSupply,
                repayData,
                borrowedTokens
            );

            // Perform the delegatecall to the asset handler
            // This allows the asset handler to execute the flash loan in the context of this contract
            (bool success, ) = address(assetHandler).delegatecall(data);

            // Check if the delegatecall was successful
            // If not, revert the transaction with a custom error
            if (!success) revert ErrorLibrary.CallFailed();
        }
    }
```

Listing 3.7: `BorrowManager::repayBorrow()`

Recommendation Revise the above routine to pass the correct parameters.

Status The issue has been addressed in the following PR: 57.

3.8 Incorrect Repayment Tx Preparation in VenusAssetHandler

- ID: PVE-008
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `VenusAssetHandler`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

As mentioned in Section 3.7, `Velvet Capital V4` may borrow funds from external lending protocol. While reviewing the related loan repayment logic, we notice another issue that stems from the preparation of repayment transactions.

To elaborate, we show below the related code snippet of the `repayTransactions()` routine. This routine needs to compute the correct token amount for repayment. It comes to our attention that the local variable of `amountToRepay` should be computed for each token, instead of the same `amountToRepay` for all tokens (line 791).

```
774     function repayTransactions(
775         address executor,
776         FunctionParameters.FlashLoanData memory flashData
777     ) internal pure returns (MultiTransaction[] memory transactions) {
778         uint256 tokenLength = flashData.debtToken.length; // Get the number of debt tokens
779         transactions = new MultiTransaction[](tokenLength * 2); // Initialize the transactions array
780         uint256 count;
781         uint256 amountToRepay = flashData.isMaxRepayment
782             ? type(uint256).max // If it's a max repayment, repay the max amount
783             : flashData.debtRepayAmount[0]; // Otherwise, repay the debt amount
784         // Loop through the debt tokens to handle repayments
785         for (uint i = 0; i < tokenLength; i++) {
786             // Approve the debt token for the protocol
787             transactions[count].to = executor;
788             transactions[count].txData = abi.encodeWithSelector(
789                 bytes4(keccak256("vaultInteraction(address,bytes)")),
790                 flashData.debtToken[i],
791                 approve(flashData.protocolTokens[i], amountToRepay)
792             );
793             count++;
794 
795             // Repay the debt using the protocol token
796             transactions[count].to = executor;
797             transactions[count].txData = abi.encodeWithSelector(
798                 bytes4(keccak256("vaultInteraction(address,bytes)")),
799                 flashData.protocolTokens[i],
800                 repay(amountToRepay)
```

```
            );
            count++;
        }
    }
```

Listing 3.8: `VenusAssetHandler::repayTransactions()`

Recommendation Revise the above routine to compute the correct repay amount for each borrowed token.

Status The issue has been addressed in the following PR: `57`.

3.9 Trust Issue of Admin Keys

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Velvet Capital V4` 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, pull funds, and upgrade proxies). In the following, we show the representative functions potentially affected by the privilege of the accounts.

```
  function upgradeBorrowManager(
    address[] calldata _proxy,
    address _newImpl
  ) external virtual onlyOwner {
    _setBaseBorrowManager(_newImpl);
    _upgrade(_proxy, _newImpl);
    emit UpgradeBorrowManager(_newImpl);
  }

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

```
    emit UpgradePortfolio(_newImpl);
  }

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

Listing 3.9: Example Privileged Operations in `PortfolioFactory`

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.

Moreover, it should be noted that current contracts may have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 V4` 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-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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

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