SMART CONTRACT AUDIT REPORT

for

Umee Protocol

Prepared By: Yiqun Chen

PeckShield

January 15, 2022

Document Properties

Client	Umee Protocol
Title	Smart Contract Audit Report
Target	Umee
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 15, 2022	Xuxian Jiang	Final Release
1.0-rc1	December 31, 2021	Xuxian Jiang	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 smart contract source code of the Umee 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 Umee

Umee is a decentralized infrastructure for cross chain interactions between networks. It connects users to create lending and borrowing positions, move capital across chains, discover new yield opportunities and explore DeFi applications intersecting networks in a seamless and trustless manner. The audited lending protocol is developed based on AaveV2 and acts as a decentralized non-custodial money market protocol where users can participate as depositors or borrowers. Depositors provide liquidity to the market to earn a passive income, while borrowers are able to borrow in an over-collateralized (perpetually) or under-collateralized (one-block liquidity) fashion. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Umee

Item	Description
Name	Umee Protocol
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 15, 2022

In the following, we show the Git repositories of reviewed files and the commit hash values used in this audit.

- https://github.com/umee-network/umee-v1-contracts.git (cf35c19)
- https://github.com/umee-network/umee-incentives.git (ff5208b)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/umee-network/umee-v1-contracts.git (84c69cd)
- https://github.com/umee-network/umee-incentives.git (8d5b168)

1.2 About PeckShield

PeckShield Inc. [10] 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 the OWASP Risk Rating Methodology [9]:

- 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 checklist of items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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 Logic	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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Umee` protocol. 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	1	■
Low	4	■ ■ ■ ■
Informational	1	■
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 1 medium-severity vulnerability, 4 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Umee Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Revisited Asset Lockup Of Vesting in UmeeVesting::vest()	Business Logic	Resolved
PVE-002	Low	Timely massUpdatePools During Pool Weight Changes	Business Logic	Resolved
PVE-003	Low	Proper Logic Of BaseUniswapAdapter::_getAmountsOutData()	Business Logic	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-005	Low	Possible Double Initialization From Initializer Reentrancy	Time and State	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides 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 Revisited Asset Lockup Of Vesting in UmeeVesting::vest()

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The `Umee` protocol has developed its own incentive mechanism, which provides a vesting schedule for locked assets. The vesting schedule is enforced in a contract named `UmeeVesting`. While reviewing the vesting logic, we notice the current implementation needs to be improved.

To elaborate, we show below the related `vest()` function. As the name indicates, it provides the vesting entrance for staking users. However, it comes to our attention the current implementation does not actually transfer assets from users into the vesting contract for lockup. In other words, the authorized users may repeatedly invoke it with a large `amount` argument to arbitrarily increase the vesting position!

```
    function vest(address account, uint256 amount) external {
        require(vesters[msg.sender], "vest: !vester");

        AccountInfo memory ai = accountInfos[account];
        uint256 _maxLock = maxLockPeriod();

        if (ai.startTime == 0) {
            // If no position exists, create a new one
            ai.startTime = block.timestamp;
            updateGlobalTime(amount, ai.startTime, 0, 0, CREATE);
            // update user position
            ai.total += amount;
            accountInfos[account] = ai;
            totalLockedAmount += amount;
        } else if(umeeBonus == msg.sender) {
```

```
            uint256 newStartTime = (ai.startTime * ai.total + block.timestamp * amount)
                / (ai.total + amount);
            if (newStartTime + _maxLock <= block.timestamp) {
                newStartTime = block.timestamp - (_maxLock) + THREE_WEEKS;
            }
            updateGlobalTime(0, ai.startTime, ai.total, newStartTime, EXTEND);
            ai.startTime = newStartTime;
            // update user position
            accountInfos[account] = ai;
        } else {
            // If a position exists, update user's startdate by weighting current time
                based on UMEE being added
            uint256 newStartTime = (ai.startTime * ai.total + block.timestamp * amount)
                / (ai.total + amount);
            if (newStartTime + _maxLock <= block.timestamp) {
                newStartTime = block.timestamp - (_maxLock) + THREE_WEEKS;
            }
            updateGlobalTime(amount, ai.startTime, ai.total, newStartTime, ADD);
            ai.startTime = newStartTime;
            // update user position
            ai.total += amount;
            accountInfos[account] = ai;
            totalLockedAmount += amount;
        }

        emit LogVest(account, totalLockedAmount, amount, ai);
    }
```

Listing 3.1: `UmeeVesting::vest()`

This issue is also applicable to other routines, including `UmeeLM::claim()` and `UmeeBonus::claim()`.

Recommendation Properly revise the above `vest()` routine to actually transfer the assets for lockup.

Status The issue has been resolved as the team clarifies the intended purpose of `vest()` to account for the vesting positions or schedules without the need of lockup.

3.2 Timely massUpdatePools During Pool Weight Changes

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `UmeeLiquidityMining`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The `Umee` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    /// @notice Update the given pool's weight. Can only be called by the owner.
    /// @param _pid The index of the pool. See `poolInfo`.
    /// @param _weight New weight.
    function set(uint256 _pid, uint256 _weight) public onlyOwner {
        totalWeight = totalWeight - poolInfo[_pid].weight + _weight;
        poolInfo[_pid].weight = _weight.toUint64();

        emit LogSetPool(_pid, _weight);
    }
```

Listing 3.2: `UmeeLiquidityMining::set()`

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated.

Status The issue has been fixed by this commit: `8d5b168`.

3.3 Proper Logic Of BaseUniswapAdapter::_getAmountsOutData()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `BaseUniswapAdapter`
- Category: Business Logic [6]
- CWE subcategory: CWE-837 [3]

Description

The audited `Umee` money-market protocol inherits from `AaveV2` with the `BaseUniswapAdapter` contract, which is designed to perform assets swaps in `UniswapV2`. While reviewing this adapter, we notice the internal helper routines `_getAmountsInData()` and `getAmountsOutData()` need to be revised.

In particular, we show below the related `_getAmountsOutData()` function. Given an input asset amount, this function computes and returns the maximum output amount of the other asset. Specifically, as part of the returned structure, it returns `outPerInPrice` – the price of out amount denominated in the `reserveIn` currency (18 decimals). It comes to our attention the `outPerInPrice` value is computed as `finalAmountIn.mul(10**18).mul(10**reserveOutDecimals).div(bestAmountOut.mul(10**reserveInDecimals))` (lines 406-408), which is the inverse of the intended amount. The correct price should be computed as follows: `bestAmountOut.mul(10**18).mul(10**reserveInDecimals).div(finalAmountIn.mul(10**reserveOutDecimals))`.

```
341   function _getAmountsOutData(
342     address reserveIn,
343     address reserveOut,
344     uint256 amountIn
345   ) internal view returns (AmountCalc memory) {
346     // Subtract flash loan fee
347     uint256 finalAmountIn = amountIn.sub(amountIn.mul(FLASHLOAN_PREMIUM_TOTAL).div
        (10000));
348
349     if (reserveIn == reserveOut) {
350       uint256 reserveDecimals = _getDecimals(reserveIn);
351       address[] memory path = new address[](1);
352       path[0] = reserveIn;
353
354       return
355         AmountCalc(
356           finalAmountIn,
357           finalAmountIn.mul(10**18).div(amountIn),
358           _calcUsdValue(reserveIn, amountIn, reserveDecimals),
359           _calcUsdValue(reserveIn, finalAmountIn, reserveDecimals),
360           path
361         );
```

```
    }

    address[] memory simplePath = new address[](2);
    simplePath[0] = reserveIn;
    simplePath[1] = reserveOut;

    uint256[] memory amountsWithoutWeth;
    uint256[] memory amountsWithWeth;

    address[] memory pathWithWeth = new address[](3);
    if (reserveIn != WETH_ADDRESS && reserveOut != WETH_ADDRESS) {
      pathWithWeth[0] = reserveIn;
      pathWithWeth[1] = WETH_ADDRESS;
      pathWithWeth[2] = reserveOut;

      try UNISWAP_ROUTER.getAmountsOut(finalAmountIn, pathWithWeth) returns (
        uint256[] memory resultsWithWeth
      ) {
        amountsWithWeth = resultsWithWeth;
      } catch {
        amountsWithWeth = new uint256[](3);
      }
    } else {
      amountsWithWeth = new uint256[](3);
    }

    uint256 bestAmountOut;
    try UNISWAP_ROUTER.getAmountsOut(finalAmountIn, simplePath) returns (
      uint256[] memory resultAmounts
    ) {
      amountsWithoutWeth = resultAmounts;

      bestAmountOut = (amountsWithWeth[2] > amountsWithoutWeth[1])
        ? amountsWithWeth[2]
        : amountsWithoutWeth[1];
    } catch {
      amountsWithoutWeth = new uint256[](2);
      bestAmountOut = amountsWithWeth[2];
    }

    uint256 reserveInDecimals = _getDecimals(reserveIn);
    uint256 reserveOutDecimals = _getDecimals(reserveOut);

    uint256 outPerInPrice =
      finalAmountIn.mul(10**18).mul(10**reserveOutDecimals).div(
        bestAmountOut.mul(10**reserveInDecimals)
      );

    return
      AmountCalc(
        bestAmountOut,
        outPerInPrice,
```

```
            _calcUsdValue(reserveIn, amountIn, reserveInDecimals),
            _calcUsdValue(reserveOut, bestAmountOut, reserveOutDecimals),
            (bestAmountOut == 0) ? new address[](2) : (bestAmountOut == amountsWithoutWeth
                [1])
              ? simplePath
              : pathWithWeth
        );
    }
```

Listing 3.3: `BaseUniswapAdapter::_getAmountsOutData()`

The `_getAmountsInData()` routine shares the same issue. Note when the given `reserveIn` is the same as the `reserveOut`, the `_getAmountsInData()` routine also returns the wrong price.

Recommendation Revise the above two routines to compute the intended price.

Status This issue has been resolved as the code is not used in the current protocol.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
```

```
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.4: ZRX.sol

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, 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 `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In the following, we show the `_liquidateAndSwap()` routine in the `FlashLiquidationAdapter` contract. If the `USDT` token is supported as `collateralAsset`, the unsafe version of `IERC20(collateralAsset).transfer(initiator, vars.remainingTokens)` (line 159) may revert as there is no return value in the `USDT` token contract's `transfer()/transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
102  function _liquidateAndSwap(
103    address collateralAsset,
104    address borrowedAsset,
105    address user,
106    uint256 debtToCover,
107    bool useEthPath,
108    uint256 flashBorrowedAmount,
109    uint256 premium,
110    address initiator
111  ) internal {
112    LiquidationCallLocalVars memory vars;
113    vars.initCollateralBalance = IERC20(collateralAsset).balanceOf(address(this));
114    if (collateralAsset != borrowedAsset) {
115      vars.initFlashBorrowedBalance = IERC20(borrowedAsset).balanceOf(address(this));
116
117      // Track leftover balance to rescue funds in case of external transfers into this
         contract
118      vars.borrowedAssetLeftovers = vars.initFlashBorrowedBalance.sub(
         flashBorrowedAmount);
```

```
    }
    vars.flashLoanDebt = flashBorrowedAmount.add(premium);

    // Approve LendingPool to use debt token for liquidation
    IERC20(borrowedAsset).approve(address(LENDING_POOL), debtToCover);

    // Liquidate the user position and release the underlying collateral
    LENDING_POOL.liquidationCall(collateralAsset, borrowedAsset, user, debtToCover,
        false);

    // Discover the liquidated tokens
    uint256 collateralBalanceAfter = IERC20(collateralAsset).balanceOf(address(this));

    // Track only collateral released, not current asset balance of the contract
    vars.diffCollateralBalance = collateralBalanceAfter.sub(vars.initCollateralBalance);

    if (collateralAsset != borrowedAsset) {
      // Discover flash loan balance after the liquidation
      uint256 flashBorrowedAssetAfter = IERC20(borrowedAsset).balanceOf(address(this));

      // Use only flash loan borrowed assets, not current asset balance of the contract
      vars.diffFlashBorrowedBalance = flashBorrowedAssetAfter.sub(vars.
          borrowedAssetLeftovers);

      // Swap released collateral into the debt asset, to repay the flash loan
      vars.soldAmount = _swapTokensForExactTokens(
        collateralAsset,
        borrowedAsset,
        vars.diffCollateralBalance,
        vars.flashLoanDebt.sub(vars.diffFlashBorrowedBalance),
        useEthPath
      );
      vars.remainingTokens = vars.diffCollateralBalance.sub(vars.soldAmount);
    } else {
      vars.remainingTokens = vars.diffCollateralBalance.sub(premium);
    }

    // Allow repay of flash loan
    IERC20(borrowedAsset).approve(address(LENDING_POOL), vars.flashLoanDebt);

    // Transfer remaining tokens to initiator
    if (vars.remainingTokens > 0) {
      IERC20(collateralAsset).transfer(initiator, vars.remainingTokens);
    }
  }
```

Listing 3.5: `FlashLiquidationAdapter::_liquidateAndSwap()`

Note this issue is also applicable to another routine, including `BaseUniswapAdapter::rescueTokens()`. For the `safeApprove()` support, there is a need to approve twice: the first time resets the allowance to zero and the second time approves the intended amount.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related approve()/transfer()/transferFrom().

Status The issue has been fixed by this commit: 6d0e969.

3.5 Possible Double Initialization From Initializer Reentrancy

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: Multiple Contracts
- Category: Time and State [7]
- CWE subcategory: CWE-682 [2]

Description

The Umee protocol supports flexible contract initialization, so that the initialization task does not need to be performed inside the constructor at deployment. This feature is enabled by introducing the initializer() modifier that protects an initializer function from being invoked twice. It becomes known that the popular OpenZepplin reference implementation has an issue that makes it possible to re-enter initializer()-protected functions. In particular, for this to happen, one call may need to be a nested-call of the other, or both calls have to be subcalls of a common initializer()-protected function.

The reentrancy can be dangerous as the initialization is not part of the proxy construction, and it becomes possible by executing an external call to an untrusted address. As part of the fix, there is a need to forbid initializer()-protected functions to be nested when the contract is already constructed.

To elaborate, we show below the current initializer() implementation as well as the fixed implementation.

```
    modifier initializer() {
        require(_initializing  _isConstructor()  !_initialized, "Initializable: contract
             is already initialized");

        bool isTopLevelCall = !_initializing;
        if (isTopLevelCall) {
            _initializing = true;
            _initialized = true;
        }

        _;

        if (isTopLevelCall) {
            _initializing = false;
```

```
        }
    }
```

Listing 3.6: `Initializable::initializer()`

```
    modifier initializer() {
        require(_initializing? _isConstructor() : !_initialized, "Initializable:
            contract is already initialized");

        bool isTopLevelCall = !_initializing;
        if (isTopLevelCall) {
            _initializing = true;
            _initialized = true;
        }

        _;

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

Listing 3.7: Revised `Initializable::initializer()`

Recommendation Enforce the `initializer()` modifier to prevent it from being re-entered.

Status The issue has been fixed by this commit: `25ca998`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [1]

Description

In the `Umee` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting, price oracle configuration, and contract adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function setStatus(bool pause) external onlyOwner {
        paused = pause;
```

```
        emit LogNewStatus(pause);
    }

    /// @notice owner can correct total amount of vested UMEE to adjust for drift of
        central curve vs user curves
    /// @param newCorrection a positive number to deduct from the unvested UMEE to
        correct for central drift
    function setCorrectionVariable(uint256 newCorrection) external onlyOwner {
        require(newCorrection <= IUmeeVesting(vester).totalGroove(), '
            setCorrectionVariable: correctionAmount to large');
        correctionAmount = newCorrection;
        emit LogNewCorrectionVariable(newCorrection);
    }

    /// @notice after every bonus claim, a user has to wait some time before they can
        claim again
    /// @param delay time delay until next claim is possible
    function setClaimDelay(uint256 delay) external onlyOwner {
        claimDelay = delay;
        emit LogNewClaimDelay(delay);
    }
```

Listing 3.8: Example `Setters` in the `UmeeBonus` Contract

In addition, we notice the `owner` account that is able to adjust various protocol-wide risk parameters. Apparently, if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

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

Status This issue has been confirmed with the team. For the time being, the team has confirmed that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account. After the protocol becomes mature, the related `owner` account will be migrated to a `DAO`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Umee` protocol, which is a decentralized infrastructure for cross chain interactions between networks. The audited lending protocol is developed based on `AaveV2` and acts as a decentralized non-custodial money market protocol with its own incentive mechanisms. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

[7] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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