SMART CONTRACT AUDIT REPORT

for

YAM FINANCE

Prepared By: Shuxiao Wang

PeckShield

February 2, 2021

Document Properties

Client	Yam Finance
Title	Smart Contract Audit Report
Target	Umbrella
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Huaguo Shi
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	February 2, 2021	Xuxian Jiang	Release Candidate #1
0.2	January 29, 2021	Xuxian Jiang	Add More Findings
0.1	January 25, 2021	Xuxian Jiang	Initial Draft

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `YAMv3`'s `Umbrella Protection Protocol` (abbreviated as `Umbrella`), 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 YAMv3/Umbrella

`YAM` is an innovative protocol of elastic supply cryptocurrency and community-based governance. The audited `Umbrella Protection Protocol` aims to provide a much-needed risk management solution by mitigating damages caused by unexpected exploits and providing corresponding coverage. It is designed to enable `Protection Providers` to earn premium fees in return for staking funds to be paid out in the event of an exploit to `Protection Seekers`, who purchase coverage at a specific rate for a custom duration. There are two pool types in the protocol: The first type (i.e., the `MetaPools`) is funded by `Protection Providers` and provides coverage on the second pool type (i.e., the `Coverage Pools`), which is accessed individually by the `Protection Seekers`.

The basic information of the `Umbrella Protection Protocol` is as follows:

Table 1.1: Basic Information of `Umbrella Protection Protocol`

Item	Description
Issuer	Yam Finance
Website	https://yam.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 2, 2021

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

- https://github.com/yam-finance/yamV3 (24a9853)

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

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.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

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

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

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.

- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.

- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [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 Umbrella implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	3	■ ■ ■
Low	2	■ ■
Informational	2	■ ■
Total	7	

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 3 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key Umbrella Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Miscalculation Of claimablePremiums()	Business Logic	
PVE-002	Medium	Out-Of-Bound Access For First _setSettling()	Coding Practices	
PVE-003	Low	Sanity Check Of rollover Parameter In initialize()	Numeric Errors	
PVE-004	Medium	Proper Adjustment Of totalProtectionSeconds In _withdraw()	Business Logic	
PVE-005	Medium	Possible Miscalculation Of totalProtectionSeconds	Business Logic	
PVE-006	Informational	Suggested payable Support in provideCoverage()	Business Logic	
PVE-007	Informational	Trust Issue of Arbiters	Security Features	

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Possible Miscalculation Of claimablePremiums()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UmbrellaMetaPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

There are two types of pools in the Umbrella protocol: `MetaPools` and `Coverage Pools`. The `Protection Providers` deposit assets in terms of coverage funds into the `MetaPools` and the `Protection Seekers` access the `Coverage Pools` by either paying coverage cost or claiming the coverage amount in the event of an exploit. In this section, we examine the logic of claimable premiums that can be collected by a `protection provider`.

To elaborate, we show below the `claimablePremiums()` routine that computes the claimable premiums of a given provider. The computation is based on the following logic: It firstly computes the contributions from the provider's coverage deposit (in terms of total protection seconds, i.e., `whoTPS` - line 846) as well as the overall protection seconds (i.e., `globalTPS` - line 848), and then delegates the computation to an internal helper `_claimablePremiums()`.

```
838     /// @notice Calculate claimable premiums for a provider
839     function claimablePremiums(address who)
840         public
841         view
842         returns (uint256)
843     {
844         uint256 timestamp = block.timestamp;
845         uint256 newTokenSecondsProvided = (timestamp - providers[who].lastUpdate).mul(
            providers[who].shares);
846         uint256 whoTPS = providers[who].totalTokenSecondsProvided.add(
            newTokenSecondsProvided);
847         uint256 newTTPS = (timestamp - lastUpdatedTPS).mul(reserves);
```

```
        uint256 globalTPS = totalProtectionSeconds.add(newTTPS);
        return _claimablePremiums(providers[who].premiumIndex, whoTPS, globalTPS);
    }

    function _claimablePremiums(uint256 index, uint256 providerTPS, uint256 globalTPS)
        internal
        view
        returns (uint256)
    {
        return premiumsAccum
                    .sub(index)
                    .mul(providerTPS)
                    .div(totalProtectionSeconds);
    }
```

Listing 3.1: UmbrellaMetaPool::claimablePremiums()

However, we notice in the internal helper the computation directly makes use of the global state of `totalProtectionSeconds`, instead of the computed `globalTPS`. As a result, it may use the old state, instead of the latest one, to calculate the claimable premiums. Fortunately, it so far only affects the viewer function, i.e., `claimablePremiums()`, as other calls to the internal helper ensure `totalProtectionSeconds` is always identical to the given `globalTPS`.

Recommendation Compute the claimable premiums with the given `globalTPS`, not the global state of `totalProtectionSeconds`.

Status

3.2 Out-Of-Bound Access For First _setSettling()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact:Medium
- Target: `UmbrellaMetaPool`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1117 [1]

Description

In the event of an exploit, the coverage funds from the `Protection Providers` can be claimed by `Protection Seekers`. In order for a `protection seeker` to claim the coverage fund, the `arbiter` needs to settle the claim so that the protocol can start to honor the claim.

In the following, we show the `_setSettling()` function that is called by the `arbiter` to set a so-called concept to be claimable. In the implementation, it provides an argument, i.e., `needs_sort`, that indicates the need of sorting existing claims based on the `settle time`.

```
934     ///@notice Sets a concept as settling (allowing claims)
935     function _setSettling(uint8 conceptIndex, uint32 settleTime, bool needs_sort)
936         public
937         onlyArbiter
938     {
939         require(conceptIndex < coveredConcepts.length, "ProtectionPool::_setSettling: !
            index");
940         require( settleTime < block.timestamp,        "ProtectionPool::_setSettling: !
            settleTime");
941         if (!needs_sort) {
942             // allow out of order if we sort, otherwise revert
943             uint32 last = claimTimes[conceptIndex][claimTimes[conceptIndex].length - 1];
944             require(settleTime > last, "ProtectionPool::_setSettling: !settleTime");
945         }
946         // add a claim time
947         claimTimes[conceptIndex].push(settleTime);
948         if (needs_sort) {
949             uint256 lastIndex = claimTimes[conceptIndex].length - 1;
950             quickSort(claimTimes[conceptIndex], int(0), int(lastIndex));
951         }
952     }
```

Listing 3.2: UmbrellaMetaPool::_setSettling()

It comes to our attention the internal array `claimTimes[]` is accessed by by an index that may lead to out-of-bound violation. In particular, if we pay attention to the code at lines 943 and 949, it computes the index as `claimTimes[conceptIndex].length - 1`. For the very first call to `_setSettling()`, there is no element in the array. Therefore, the length is in essence 0, the access of -1 index leads to an out-of-bound access violation.

Recommendation Revised the `_setSettling()` logic to accommodate the scenario when the initial array of `claimTimes[]` is empty.

Status

3.3 Sanity Check Of rollover Parameter In initialize()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UmbrellaMetaPool`
- Category: Numeric Errors [8]
- CWE subcategory: CWE-190 [2]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Umbrella protocol is no exception. Specifically, if we examine the `UmbrellaMetaPool` contract, it has defined a number of system-wide risk parameters, e.g., `rollover`, `creatorFee`, and `arbiterFee`. In the following, we show the `initialize()` routine that sets up these parameters.

```
    function initialize(
        address payToken_,
        uint64 coefficients,
        uint128 creatorFee_,
        uint128 arbiterFee_,
        uint128 rollover_,
        uint128 minPay_,
        string[] memory coveredConcepts_,
        string memory description_,
        address creator_,
        address arbiter_
    )
        public
    {
        require(!initialized, "initialized");
        initialized = true;
        require(coveredConcepts_.length < 16, "too many concepts");

        // TODO: Move to factory
        require(arbiterFee_ <= MAX_ARB_FEE, "!arb fee");
        require(creatorFee_ <= MAX_CREATE_FEE, "!create fee");
        // :TODO

        intialize_rate(coefficients);

        payToken         = payToken_;
        arbiterFee       = arbiterFee_;
        creatorFee       = creatorFee_;
        rollover         = rollover_;
        coveredConcepts  = coveredConcepts_;
        description      = description_;
        creator          = creator_;
        arbiter          = arbiter_;
        minPay           = minPay_;
```

```
        claimTimes        = new uint32[][](coveredConcepts_.length);

        if (creator_ == arbiter_) {
            // auto accept if creator is arbiter
            arbSet = true;
            accepted = true;
        }
    }
```

Listing 3.3: UmbrellaMetaPool:: initialize ()

This parameter defines an important aspect of the protocol operation and needs to exercise extra care when configuring or updating it. Our analysis shows the configuration logic on it can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `rollover` may charge unreasonable share of premiums into reserves, hence undermining the protocol integrity (line 929).

```
906     /// @dev updates various vars relating to premiums and fees
907     function _update(uint128 coverageRemoved, uint128 premiumsPaid)
908         internal
909     {
910         utilized = utilized.sub(coverageRemoved);
911         uint128 arbFees;
912         uint128 createFees;
913         uint128 rollovers;
914         if (arbiterFee > 0) {
915             arbFees = premiumsPaid.mul(arbiterFee).div(BASE);
916             arbiterFees = arbiterFees.add(arbFees); // pay arbiter
917         }
918         if (creatorFee > 0) {
919             createFees = premiumsPaid.mul(creatorFee).div(BASE);
920             creatorFees = creatorFees.add(createFees); // pay creator
921         }
922         if (rollover > 0) {
923             rollovers = premiumsPaid.mul(rollover).div(BASE);
924             reserves = reserves.add(rollovers); // rollover some % of premiums into
                    reserves
925         }
926
927         // push remaining premiums to premium pool
928         // SAFETY: BASE is 10**18, all others are bounded such that sum(r, c, a) < BASE.
929         premiumsAccum = premiumsAccum.add(premiumsPaid - arbFees - createFees -
                rollovers);
930     }
```

Listing 3.4: UmbrellaMetaPool::_update()

Recommendation Validate any changes regarding the system-wide parameters to ensure the changes fall in an appropriate range.

Status

3.4 Proper Adjustment Of totalProtectionSeconds In _withdraw()

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: UmbrellaMetaPool
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, there are two types of pools in the Umbrella protocol: MetaPools and Coverage Pools. The Protection Providers deposit assets in terms of coverage funds into the MetaPools and the Protection Seekers access the Coverage Pools by either paying coverage cost or claiming the coverage amount in the event of an exploit. In the following, we examine the logic when a protection provider intends to withdraw previously deposited coverage funds from the MetaPools.

In particular, we show below the internal _withdraw() routine that handles the withdrawal request. It comes to our attention that when the particular protection provider withdraws all of his/her share, the current logic resets the providers[msg.sender].totalTokenSecondsProvided. However, it does not accordingly adjust the totalProtectionSeconds. Without proper adjustment, this portion of share from late accumulation of premiums may never be credited to staying protection providers.

```
function _withdraw(uint128 asShares)
    internal
{
    require(         providers[msg.sender].withdrawInitiated + LOCKUP_PERIOD <  block.timestamp, "ProtectionPool::withdraw: locked");
    require(                 providers[msg.sender].lastProvide + LOCKUP_PERIOD <  block.timestamp, "ProtectionPool::withdraw: locked2");
    require(providers[msg.sender].withdrawInitiated + WITHDRAW_GRACE_PERIOD >= block.timestamp, "ProtectionPool::withdraw: expired");

    // get premiums
    _claimPremiums();

    // update reserves & balance
    uint128 underlying = exit(asShares);
    require(reserves >= utilized, "ProtectionPool::withdraw: !liquidity");
    if (providers[msg.sender].shares == 0) {
        providers[msg.sender].totalTokenSecondsProvided = 0;
    }
    // payout
```

```
        IERC20(payToken).safeTransfer(msg.sender, underlying);
        emit Withdraw(msg.sender, underlying);
    }
```

Listing 3.5: UmbrellaMetaPool::_withdraw()

Recommendation Revised the withdraw logic to properly adjust `totalProtectionSeconds`. An example revision is shown below:

```
    function _withdraw(uint128 asShares)
        internal
    {
        require(        providers[msg.sender].withdrawInitiated + LOCKUP_PERIOD <  block.timestamp, "ProtectionPool::withdraw: locked");
        require(                providers[msg.sender].lastProvide + LOCKUP_PERIOD <  block.timestamp, "ProtectionPool::withdraw: locked2");
        require(providers[msg.sender].withdrawInitiated + WITHDRAW_GRACE_PERIOD >= block.timestamp, "ProtectionPool::withdraw: expired");

        // get premiums
        _claimPremiums();

        // update reserves & balance
        uint128 underlying = exit(asShares);
        require(reserves >= utilized, "ProtectionPool::withdraw: !liquidity");
        if (providers[msg.sender].shares == 0) {
            totalProtectionSeconds -= providers[msg.sender].totalTokenSecondsProvided;
            providers[msg.sender].totalTokenSecondsProvided = 0;
        }
        // payout
        IERC20(payToken).safeTransfer(msg.sender, underlying);
        emit Withdraw(msg.sender, underlying);
    }
```

Listing 3.6: Revised UmbrellaMetaPool::_withdraw()

Status

3.5 Possible Miscalculation Of totalProtectionSeconds

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `UmbrellaMetaPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As shown in previous sections, the global state of `totalProtectionSeconds` plays an important role in keeping track of the overall contribution from deposited coverage funds. This global state is used to compute the share of accumulated premiums as payout to current `protection providers`.

To elaborate, we show below the code snippet of two modifiers in the `UmbrellaMetaPool` contract. These two modifiers compute or update the contribution of each `protection provider` as well as the overall contributions from all `protection providers`.

```
  modifier updateTokenSecondsProvided(address account) {
    uint256 timestamp = block.timestamp;
    uint256 newTokenSecondsProvided =
          (timestamp - providers[account].lastUpdate).mul(providers[account].shares);

    // update user protection seconds, and last updated
    providers[account].totalTokenSecondsProvided = providers[account].
        totalTokenSecondsProvided.add(newTokenSecondsProvided);
    providers[account].lastUpdate = safe32(timestamp);

    // increase total protection seconds
    uint256 newGlobalTokenSecondsProvided = (timestamp - lastUpdatedTPS).mul(reserves)
        ;
    totalProtectionSeconds = totalProtectionSeconds.add(newGlobalTokenSecondsProvided)
        ;
    lastUpdatedTPS = safe32(timestamp);
    _;
  }

  modifier updateGlobalTPS() {
    uint256 timestamp = block.timestamp;

    // increase total protection seconds
    uint256 newGlobalTokenSecondsProvided = (timestamp - lastUpdatedTPS).mul(reserves)
        ;
    totalProtectionSeconds = totalProtectionSeconds.add(newGlobalTokenSecondsProvided)
        ;
    lastUpdatedTPS = safe32(timestamp);
    _;
```

```
265     }
```

Listing 3.7: Two Modifiers in UmbrellaMetaPool: updateTokenSecondsProvided() and updateGlobalTPS()

While analyzing these two modifiers, we notice that the current method of computing the global state of `totalProtectionSeconds` needs to be revised. In particular, it calculates the addition as `(timestamp - lastUpdatedTPS).mul(reserves)` (lines 251 and 261), which fails to taking into account possible contribution from accumulated premiums. A better approach is to compute as `(timestamp - lastUpdatedTPS).mul(totalShares)`. By doing so, we can fairly distribute the credits to all current `protection providers`.

Note there are three routes that are affected: `updateTokenSecondsProvided()`, `updateGlobalTPS()`, and `claimablePremiums()`.

Recommendation Adjust the method to properly compute the `totalProtectionSeconds` state.

Status

3.6 Suggested payable Support in provideCoverage()

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `UmbrellaMetaPool`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The coverage funds in `MetaPools` are dynamic. A `protection provider` may increase or decrease the balance by depositing or withdrawing in terms of the configured `payToken`. A `protection seeker` may pay coverage cost or claim coverage, which affects the fund balance as well.

To elaborate, we show below the `buyProtection()` routine that a `protection seeker` pays the coverage cost for the intended coverage. This routine has a nice feature in accepting `ETH` payment if the `payToken` is `WETH`.

```
    /// @notice Purchase protection
    /// @dev accepts ETH payment if payToken is WETH
    function buyProtection(
        uint8 conceptIndex,
        uint128 coverageAmount,
        uint128 duration,
        uint128 maxPay,
        uint256 deadline
    )
        public
```

```
        payable
        hasArbiter
    {
        // check deadline
        require(block.timestamp <= deadline,             "ProtectionPool::
            buyProtection: !deadline");
        require(   conceptIndex <  coveredConcepts.length, "ProtectionPool::
            buyProtection: !conceptIndex");

        // price coverage
        uint128 coverage_price = _price(coverageAmount, duration, utilized, reserves);

        // check payment
        require(utilized.add(coverageAmount) <= reserves, "ProtectionPool::buyProtection
            : overutilized");
        require(              coverage_price >= minPay,   "ProtectionPool::buyProtection
            : price < minPay");
        require(              coverage_price <= maxPay,   "ProtectionPool::buyProtection
            : too expensive");

        // push protection onto array
        // protection buying stops in year 2106 due to safe cast
        protections.push(
          Protection({
              coverageAmount: coverageAmount,
              paid: coverage_price,
              holder: msg.sender,
              start: safe32(block.timestamp),
              expiry: safe32(block.timestamp + duration),
              conceptIndex: conceptIndex,
              status: Status.Active
          })
        );

        // increase utilized
        utilized = utilized.add(coverageAmount);

        if (payToken == address(WETH) && msg.value > 0) {
            // wrap eth => WETH if necessary
            uint256 remainder = msg.value.sub(coverage_price, "ProtectionPool::
                buyProtection: underpayment");
            WETH.deposit.value(coverage_price)();

            // send back excess, 2300 gas
            if (remainder > 0) {
                msg.sender.transfer(remainder);
            }
        } else {
            require(msg.value == 0, "ProtectionPool::buyProtection: payToken !WETH, dont
                 send eth");
            IERC20(payToken).safeTransferFrom(msg.sender, address(this), coverage_price)
                ;
```

```
        }

        // events
        emit NewProtection(coveredConcepts[conceptIndex], coverageAmount, safe32(
            duration), coverage_price);
    }
```

Listing 3.8: buyProtection()

However, if we examine the `provideCoverage()` counterpart that a `protection provider` deposits the coverage funds, the `ETH` payment is not supported. For consistency, it is suggested to add the `ETH` payment support as well.

```
    ///@notice Provide coverage - liquidity is locked for at minimum 1 week
    function provideCoverage(
        uint128 amount
    )
        public
        hasArbiter
        updateTokenSecondsProvided(msg.sender)
    {
        require(amount > 0, "ProtectionPool::provideCoverage: amount 0");
        _claimPremiums();
        enter(amount);
        // TODO delete before mainnet
        /* require(reserves <= MAX_RESERVES, "ProtectionPool::provideCoverage: Max
            reserves met for alpha"); */
        IERC20(payToken).safeTransferFrom(msg.sender, address(this), amount);
        emit ProvideCoverage(msg.sender, amount);
    }
```

Listing 3.9: provideCoverage()

Recommendation Revise the above `provideCoverage()` logic to support the `ETH`-based payment if the `payToken` is `WETH`.

Status

3.7 Trust Issue of Arbiters

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `UmbrellaMetaPool`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In Umbrella, there is a privileged contract, i.e., `arbiter`, that plays a critical role in setting a particular concept to be claimable. If the `arbiter` is unwilling or unable to set a particular concept (via `_setSettling()` in Section 3.2), it is impossible for current `protection seekers` to claim their loss.

In the following, we show the `claim()` routine that is used by `protection seekers` to make a claim. The implementation places an check that requires there is an active settlement (line 612), which is controlled by the `arbiter` account.

```
599     function claim(uint256 pid)
600         public
601         updateGlobalTPS
602     {
603         Protection storage protection = protections[pid];
604         require(
605             protection.holder == msg.sender
606             operators[protection.holder][msg.sender] == true,
607             "ProtectionPool::claim: !operator"
608         );
609
610         // ensure: settling, active, and !expiry
611         require(protection.status == Status.Active, "ProtectionPool::claim: !active");
612         require(_hasSettlement(protection.conceptIndex, protection.start, protection.
                expiry), "ProtectionPool::claim: !start");
613
614         protection.status = Status.Claimed;
615
616         // decrease utilized and reserves
617         utilized = utilized.sub(protection.coverageAmount);
618         reserves = reserves.sub(protection.coverageAmount);
619
620         // transfer coverage + payment back to coverage holder
621         uint256 payout = protection.coverageAmount.add(protection.paid);
622         IERC20(payToken).safeTransfer(protection.holder, payout);
623         emit Claim(protection.holder, pid, payout);
624     }
```

Listing 3.10: UmbrellaMetaPool::claim()

We emphasize that this privilege is necessary and does needs this specific role for settlement. The privileged just needs to be managed or governed by a DAO-like structure.

We point out that a compromised `arbiter` account poses risks to the claim-ability of coverage funds.

Recommendation Promptly design a trustless, decentralized scheme to reduce the concern on the centralized `arbiter` privilege.

Status

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `YAMv3`'s `Umbrella Protection Protocol`, which provides a much-needed risk management solution that aims to mitigate damages caused by possible exploits by providing related coverage. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1117: Callable with Insufficient Behavioral Summary. https://cwe.mitre.org/data/definitions/1117.html.

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

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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