SMART CONTRACT AUDIT REPORT

for

AMY FINANCE

Prepared By: Yiqun Chen

PeckShield
September 24, 2021

Document Properties

Client	Amy Finance
Title	Smart Contract Audit Report
Target	Amy Finance
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Shulin Bie, Xiaotao Wu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 24, 2021	Xuxian Jiang	Final Release
1.0-rc	September 22, 2021	Xuxian Jiang	Release Candidate

Contact

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

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

Contents

1 Introduction

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

`Amy Finance` is a layer 2 based, liquidity provider (LP)-friendly lending and leveraged-yield farming protocol. By staking assets into the assembly pool, each LP and staker will receive the interests generated from lending and leverage trading. Moreover, users can borrow assets from the pool and conduct margin trading in one app. The transaction fee will be extremely low owing to the L2 native design. By design, the `Amy` protocol aim to lift the revenue and maintain the same level of fund security as the classic lending protocols do.

The basic information of Amy Finance is as follows:

Table 1.1: Basic Information of Amy Finance

Item	Description
Issuer	Amy Finance
Website	https://amy.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 24, 2021

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

- https://github.com/amyfinance/main-contracts-audited.git (94969d6)
- https://github.com/amyfinance/cat-contracts-audited.git (8e3bbfa)

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

- https://github.com/amyfinance/main-contracts-audited.git (3d46b24)
- https://github.com/amyfinance/cat-contracts-audited.git (5624fa1)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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 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) [11], 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 1.3: 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

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 implementation of the Amy 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 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	4	■■■■
Low	6	■■■■■■
Informational	1	■
Total	11	

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 4 medium-severity vulnerabilities, 6 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings of Amy Finance Protocol

ID	Severity	Title	Category	Status
PVE-001	Low	Unknown Entering of Markets For Unknowing Users	Business Logic	Confirmed
PVE-002	Medium	Possible Unintended Uses Of Exchange Unit	Numeric Errors	Confirmed
PVE-003	Low	Strengthened Validity Check of isFTokenValid()	Business Logic	Confirmed
PVE-004	Low	Proper Interest Rate Model Initialization	Business Logic	Fixed
PVE-005	Medium	Proper Utilization Rate Calculation	Business Logic	Fixed
PVE-006	Medium	Proper Interest Attribution in Vault::accrue()	Business Logic	Fixed
PVE-007	Informational	Generation Of Events For Important States Changes	Coding Practices	Fixed
PVE-008	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-009	Low	Improved Logic of createPoolLootBoxConfiguration()	Business Logic	Fixed
PVE-010	Low	Improved Logic of stake()/_withdraw()	Business Logic	Fixed
PVE-011	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed

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 Unknown Entering of Markets For Unknowing Users

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Targets: `BankController`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `FToken` contract in `Amy` follows a similar architectural design of `Compound` with a central `BankController` contract. This contract enforces various invariants and essentially guards the gate to access the normal functionality of `Amy`. In the following, we analyze one specific validation `borrowCheckForLeverage()`. This validation is applied when a supplying user wants to borrow certain assets from the liquidity pool. Our analysis with this routine shows that the validation needs to be improved to block unintended entering of markets for a victim user.

Specifically, this routine assumes the calling user is a supported `FToken`, which unfortunately is not always the case. A calling user can be a crafted contract that has a public `underlying()` function that returns a legitimate asset being supported in the protocol. With that, a malicious actor can successfully violate the assumption and bypass the check (line 392). However, once the check is bypassed, the actor can add the given victim user, i.e., `account`, to enter any market even the user has no intention of entering.

```
386     function borrowCheckForLeverage(
387         address account,
388         address underlying,
389         address fToken,
390         uint256 borrowAmount
391     ) external {
392         require(underlying == IFToken(msg.sender).underlying(), "invalid underlying
                token");
393         require(
```

```
            markets[underlying].isValid,
            "BorrowCheck fails"
        );
        require(!tokenConfigs[underlying].borrowDisabled, "borrow disabled");

        uint borrowCap = borrowCaps[underlying];
        // Borrow cap of 0 corresponds to unlimited borrowing
        if (borrowCap != 0) {
            uint totalBorrows = IFToken(msg.sender).totalBorrows();
            uint nextTotalBorrows = totalBorrows.add(borrowAmount);
            require(nextTotalBorrows < borrowCap, "market borrow cap reached");
        }
        ...
    }
```

Listing 3.1: `BankController::borrowCheckForLeverage()`

Recommendation Properly validate the caller is indeed a supported `FToken` so that the underlying assumption is not violated.

Status This issue has been confirmed.

3.2 Possible Unintended Uses Of Exchange Unit

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `FToken`
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [2]

Description

In the `Amy` protocol, supplying users may deposit their assets into the pool and get corresponding `FToken` in return. As an interest-bearing token, the `FToken` here plays exactly the same role as `cToken` in `Compound` or `aToken` in `Aave`. When examining the `FToken` logic, we notice the implicit requirement of `FToken` decimal that needs to be fixed (otherwise the exchange unit uses for the account health check may be abused for unintended purposes). And the decimal plays a critical role to normalize the `FToken` price and the associated value.

```
    function initFtoken(
        uint256 _initialExchangeRate,
        address _controller,
        address _initialInterestRateModel,
        address _underlying,
        uint256 _borrowSafeRatio,
        string memory _name,
```

```
        string memory _symbol,
        uint8 _decimals,
        address _arbSys
    ) internal {
        initialExchangeRate = _initialExchangeRate;
        controller = IBankController(_controller);
        interestRateModel = IInterestRateModel(_initialInterestRateModel);
        admin = msg.sender;
        underlying = _underlying;
        borrowSafeRatio = _borrowSafeRatio;
        arbSys = _arbSys;
        accrualBlockNumber = getBlockNumber();
        borrowIndex = ONE;
        name = _name;
        symbol = _symbol;
        decimals = _decimals;
        _notEntered = true;
        securityFactor = 3000;
    }
```

Listing 3.2: FToken::initFtoken()

Specifically, the implicit assumption is that all `FTokens` should have the same 18 decimals. However, current initialization routine (as shown above) indicates that this decimal can be passed in as an argument. In other words, it depends on the external off-chain procedure to properly enforce this assumption. Note that a non-18 `FToken` decimal may lead to unexpected results when there is a need to compute or normalize the asset value.

Recommendation Enforce the implicit assumption by ensuring the given decimal is always 18.

Status The issue has been confirmed and the team decides to exercise extra caution in deploying new `FToken` contracts.

3.3 Strengthened Validity Check of isFTokenValid()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Targets: `BankController`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.1, the `BankController` contract enforces various invariants and essentially guards the gate to access the normal functionality of `Amy`. In this contract, there is a specific function `isFTokenValid()` that is designed to determine whether a given `fToken` is valid or not. Our analysis shows that this check may be bypassed.

To elaborate, we show below this `isFTokenValid()` routine. It has a basic logic in querying the given address for the `underlying()` asset and then checking whether it is valid in current market setup. It comes to our attention that the `underlying()` call may get a crafted input that returns a legitimate asset with a registered market. With that, we suggest to strengthen the validity check by further requiring `marketsContains(fToken)`.

```
    function isFTokenValid(address fToken) external view returns (bool) {
        return markets[IFToken(fToken).underlying()].isValid;
    }
```

Listing 3.3: `BankController::isFTokenValid()`

Recommendation Strengthen the validity check in the above `isFTokenValid()`. An example revision is shown in the following:

```
    function isFTokenValid(address fToken) external view returns (bool) {
        return markets[IFToken(fToken).underlying()].isValid && marketsContains(fToken);
    }
```

Listing 3.4: Revised `BankController::isFTokenValid()`

Status This issue has been confirmed.

3.4 Proper Interest Rate Model Initialization

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Targets: `InterestRateModel`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Amy` protocol has an `InterestRateModel` contract which, as the name indicates, is used to provide the required interest rate model for the built-in lending functionality. While reviewing its initialization logic, we notice the current implementation can be improved.

To elaborate, we show below its full implementation. While it properly sets up a number of states (e.g., `interestSlope1` and `interestSlope2`), the current implementation ignores the given argument `_secondsPerBlock` and assigns hardcoded values to `secondsPerBlock` (line 32) and `blocksPerYear` (line 33).

```
    function initialize(
        uint256 _secondsPerBlock,
        uint256 _interestSlope1,
        uint256 _interestSlope2
    ) public initializer {
        OwnableUpgradeSafe.__Ownable_init();

        secondsPerBlock = _secondsPerBlock;
        blocksPerYear = SECONDS_PER_YEAR.div(_secondsPerBlock);
        interestSlope1 = _interestSlope1;
        interestSlope2 = _interestSlope2;
        blocksPerYear = 2102400;
        secondsPerBlock = 15 seconds;
        SECONDS_PER_YEAR = 365 days;
        OPTICAL_USAGE_RATE = 85e18;
        MAX_USAGE_RATE = 100e18;
        BASIC_INTEREST = 10e16;
    }
```

Listing 3.5: `InterestRateModel::initialize()`

Recommendation Properly update the above `initialize()` function to utilize the given arguments.

Status The issue has been fixed by this commit: `faafb69`.

3.5 Proper Utilization Rate Calculation

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: `InterestRateModel`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the same `InterestRateModel` contract, there is a pure function `utilizationRate()` that is used to compute the current utilization rate. Our analysis shows that it ignores the passed-in `reserves` state and computes the utilization rate based on the `cash` and `borrows` only.

```
    function utilizationRate(
        uint256 cash,
        uint256 borrows,
        uint256 reserves
    ) public pure returns (uint256) {
        if (borrows == 0) {
            return 0;
```

```
47         }
48 
49         // borrows / (cash + borrows)
50         return borrows.mul(100e18).div(cash.add(borrows));
51     }
```

Listing 3.6: `InterestRateModel::utilizationRate()`

To elaborate, we show above the related `utilizationRate()` function. It is our understanding that the utilization rate needs to be computed as `borrows / (cash + borrows - reserves)`, instead of the current implementation of `borrows / (cash + borrows)` (line 50).

Recommendation Revise the utilization rate computation as suggested.

Status The issue has been fixed by this commit: `a87a386`.

3.6 Proper Interest Attribution in Vault::accrue()

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Vault`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the same vein as the utilization rate computation in Section 3.5, the `Vault` in the `Amy` protocol allows for the accrual of interests from the borrowed assets for leverage. And a portion of the accrued interest will be reserved for protocol development, insurance fund, or team incentivization purposes. However, our analysis shows that there is not fund reserved for this purpose.

To elaborate, we show below the `accrue()` routine from the `Vault` contract. This routine calls an internal helper function `pendingInterest()` to collect pending interest form the borrowed funds and records the latest accrual timestamp in `lastAccrueTime`. To properly allocate certain portion of collected interests, there is a need to expand the current functionality for reserve purposes.

```
  /// Add more debt to the bank debt pool.
  modifier accrue(uint256 value) {
    if (now > lastAccrueTime) {
      uint256 interest = pendingInterest(value);
      vaultDebtVal = vaultDebtVal.add(interest);
      lastAccrueTime = now;
    }
    _;
  }
```

Listing 3.7: Vault::accrue()

Recommendation Support the reserve funds by revising the above logic in `accrue()`.

Status The issue has been fixed by this commit: `fdad49d`.

3.7 Generation Of Events For Important States Changes

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `BankController`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `BankController` contract as an example. This contract has public functions that are used to transfer the `admin`. While examining the events that reflect the `admin` changes, we notice there is a lack of emitting important events that reflect important state changes. Specifically, when the `admin` is being updated in `claimAdministration()`, there is no respective event being emitted to reflect the update of `admin` (line 229).

```
223     function proposeNewAdmin(address admin_) external onlyMulSig {
224         proposedAdmin = admin_;
225     }

227     function claimAdministration() external {
228         require(msg.sender == proposedAdmin, "Not proposed admin.");
229         admin = proposedAdmin;
230         proposedAdmin = address(0);
231     }
```

Listing 3.8: `BankController::proposeNewAdmin()/claimAdministration()`

Recommendation Properly emit respective events when a new `admin` becomes effective.

Status The issue has been fixed by this commit: `c442da9`.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the Amy protocol, all debt positions are managed by the Vault contract. And there is a privileged account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and strategy 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.

To elaborate, we show below the kill() routine in the Vault contract. This routine allows anyone to liquidate the given position assuming it is underwater and available for liquidation. There is a key factor, i.e., killFactor, that greatly affects the decision on whether the position can be liquidated (line 273). Note that killFactor is a risk parameter that can be dynamically configured by the privileged owner.

```
265   function kill(uint256 id, bytes calldata swapData) external onlyEOA accrue(0)
          nonReentrant {
266     //
267     Position storage pos = positions[id];
268     require(pos.debtShare > 0, "kill:: no debt");
269
270     uint256 debt = _removeDebt(id);
271     uint256 health = IWorker(pos.worker).health(id);
272     uint256 killFactor = config.killFactor(pos.worker, debt);
273     require(health.mul(killFactor) < debt.mul(10000), "kill:: can't liquidate");
274
275     //
276     uint256 beforeToken = SafeToken.myBalance(token);
277     IWorker(pos.worker).liquidateWithData(id, swapData);
278     uint256 back = SafeToken.myBalance(token).sub(beforeToken);
279     // 5% of the liquidation value will become a Clearance Fees
280     uint256 clearanceFees = back.mul(config.getKillBps()).div(10000);
281     // 30% for liquidator reward
282     uint256 prize = clearanceFees.mul(securityFactor).div(10000);
283     // 30% for $AMY token stakers reward
284     // 30% to be converted to $AMY/USDT LP Pair on DoDo
285     // 10% to security fund
286     uint256 securityFund = clearanceFees.sub(prize);
287
```

```
    uint256 rest = back.sub(clearanceFees);

    // Clear position debt and return funds to liquidator and position owner.
    if (prize > 0) {
      if (token == config.getWrappedNativeAddr()) {
        SafeToken.safeTransfer(token, config.getWNativeRelayer(), prize);
        WNativeRelayer(uint160(config.getWNativeRelayer())).withdraw(prize);
        SafeToken.safeTransferETH(msg.sender, prize);
      } else {
        SafeToken.safeTransfer(token, msg.sender, prize);
      }
    }

    if (securityFund > 0) {
      address mulsig = controller.mulsig();
      require(mulsig != address(0), "Vault::kill mulsig is address(0)");
      SafeToken.safeTransfer(token, mulsig, securityFund);

      emit SecurityFund(id, mulsig, securityFund);
    }

    uint256 left = rest > debt ? rest - debt : 0;
    if (left > 0) {
      if (token == config.getWrappedNativeAddr()) {
        SafeToken.safeTransfer(token, config.getWNativeRelayer(), left);
        WNativeRelayer(uint160(config.getWNativeRelayer())).withdraw(left);
        SafeToken.safeTransferETH(pos.owner, left);
      } else {
        SafeToken.safeTransfer(token, pos.owner, left);
      }
    }
    emit Kill(id, msg.sender, pos.owner, health, debt, prize, left);
  }
```

Listing 3.9: `Vault::kill()`

Also, if we examine the privileged function on available `DodoswapWorker`, i.e., `setCriticalStrategies()`, this routine allows the update of new strategies to work on a user's position. It has been highlighted that bad strategies can steal user funds. Note that this privileged function is guarded with `onlyOwner`.

```
  function setCriticalStrategies(IStrategy _addStrat, IStrategy _liqStrat) external
      onlyOwner {
    addStrat = _addStrat;
    liqStrat = _liqStrat;
  }
```

Listing 3.10: `DodoswapWorker::setCriticalStrategies()`

It is worrisome if the privileged `owner` account is a plain EOA account. The discussion with the team confirms that the `owner` account is currently managed by a timelock. A plan needs to be in place to migrate it under community governance. 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 if current contracts will be deployed behind a proxy, 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. The team further clarifies the plan to migrate the admin key to a timelock under the multi-sig governance.

3.9 Improved Logic of createPoolLootBoxConfiguration()

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TokenRewardPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

In the `Amy` protocol, there are two rewarding pools `GoldenCatRewardPool` and `TokenRewardPool`. The second reward pool `TokenRewardPool` has the support of creating a specific configuration for the so-called `LootBox`. Different configurations may be created to allow for customized requirements of the staking amount `stakingAmountRequired` and the staking block count `stakingBlockCountRequired`.

To elaborate, we show below the related `createPoolLootBoxConfiguration()` function. It comes to our attention that the validity checks (lines 148-165) are mainly evaluated in the `while`-loop. However, the `while`-condition of `i>=0` is always evaluated to be true.

```
133     function createPoolLootBoxConfiguration(
134         uint256 _poolId,
135         bool _enabled,
136         uint256 _stakingAmountRequired,
137         uint256 _stakingBlockCountRequired
138     ) external onlyOwner poolExists(_poolId) {
139         require(_stakingAmountRequired != 0, "Invalid Stake Amount Required.");
140         require(
141             _stakingBlockCountRequired != 0,
```

```
            "Invalid Stake Block Count Required."
        );
        Pool storage pool = pools[_poolId];
        // make sure the current config staking amount is larger than all existing ones
        if (pool.lootBoxConfigurationIdTracker.current() > 0) {
            uint256 i = pool.lootBoxConfigurationIdTracker.current().sub(1);
            while (i >= 0) {
                PoolLootBoxConfiguration
                    storage existingPoolLootBoxConfiguration = pool
                        .lootBoxConfigurations[i];
                if (existingPoolLootBoxConfiguration.enabled) {
                    require(
                        _stakingAmountRequired >
                            existingPoolLootBoxConfiguration
                                .stakingAmountRequired,
                        "Stake Amount Less Than Existing Config."
                    );
                }
                if (i == 0) {
                    break;
                } else {
                    i--;
                }
            }
        }
        // update value
        PoolLootBoxConfiguration storage poolLootBoxConfiguration = pool
            .lootBoxConfigurations[
                pool.lootBoxConfigurationIdTracker.current()
            ];
        poolLootBoxConfiguration.enabled = _enabled;
        poolLootBoxConfiguration.stakingAmountRequired = _stakingAmountRequired;
        poolLootBoxConfiguration
            .stakingBlockCountRequired = _stakingBlockCountRequired;
        pool.lootBoxConfigurationIdTracker.increment();
        emit PoolLootBoxConfigurationCreated(
            _poolId,
            pool.lootBoxConfigurationIdTracker.current().sub(1),
            _enabled,
            _stakingAmountRequired,
            _stakingBlockCountRequired
        );
    }
```

Listing 3.11: `TokenRewardPool::createPoolLootBoxConfiguration()`

Moreover, for each configuration, there are two inherent requirements. The first one requires the strict increasing order in terms of `stakingAmountRequired` when compared with earlier ones. The second one requires the strict decreasing order in terms of `stakingBlockCountRequired`. However, the second one is not enforced in the above `createPoolLootBoxConfiguration()` routine.

Recommendation Revise the above `createPoolLootBoxConfiguration()` routine as suggested.

Status The issue has been fixed by this commit: `6822491`.

3.10 Improved Logic of stake()/_withdraw()

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GoldenCatRewardPool`, `TokenRewardPool`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.9, there are two rewarding pools `GoldenCatRewardPool` and `TokenRewardPool` in `Amy`. Each rewarding pool allows for the staking users to claim pending rewards. While reviewing the staking and unstaking logic, we notice the reward-related logic can be improved.

To elaborate, we show below the implementation of the `stake()` function. This routine implements a rather standard staking logic. However, the reward is claimed (line 275) before it is updated or refreshed to the very moment when the staking occurs. Therefore, it will be helpful to claim the reward after it is updated. After that, the main staking logic can be performed.

```
270     function stake(uint256 _poolId, uint256[] calldata _catIds)
271         external
272         nonReentrant
273         poolExists(_poolId)
274     {
275         _claimReward(_poolId);
276 
277         updatePoolRewardInfo(_poolId, msg.sender);
278         require(
279             _catIds.length > 0 && _catIds.length <= 3,
280             "Invalid cat ids length."
281         );
282         Pool storage pool = pools[_poolId];
283         PoolUser storage poolUser = pool.users[msg.sender];
284         // validation
285         require(
286             poolUser.stakingCatCount.add(_catIds.length) <= 3,
287             "Invalid cat ids length."
288         );
289         // transfer cat
290         for (uint256 i = 0; i < _catIds.length; i++) {
291             uint256 catId = _catIds[i];
292             require(catId != 0, "Invalid cat id.");
```

```
            require(
                goldenCat.ownerOf(catId) == msg.sender,
                "Unauthorized cat."
            );
            goldenCat.safeTransferFrom(msg.sender, address(this), catId);
            poolUser.catIds[poolUser.stakingCatCount.add(i)] = catId;
        }
        // update pool stakingCatCount
        poolUser.stakingCatCount = poolUser.stakingCatCount.add(_catIds.length);
        pool.stakingCatCount = pool.stakingCatCount.add(_catIds.length);

        // calculate IQ
        resetPoolUserStakingIQAmount(_poolId, msg.sender);
        emit PoolUserStaked(
            _poolId,
            msg.sender,
            _catIds,
            poolUser.stakingIQAmount
        );
    }
```

Listing 3.12: GoldenCatRewardPool::stake()

Note that the same issue is also applicable to the unstaking logic with the main functionality implemented in `_withdraw()`.

Recommendation Improved the afore-mentioned routines to claim the reward after it is updated.

Status The issue has been fixed by this commit: `377c759`.

3.11 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-011
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `GoldenCatRewardPool`, `TokenRewardPool`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `GoldenCatRewardPool` as an example, the `_claimReward()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation of an external contract requires extra care in avoiding the above `re-entrancy`. For example, the interaction with the external contract (lines 403 – 406) start before effecting the update on internal states (lines 407 – 415), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching `re-entrancy` via the same entry function.

```
398    function _claimReward(uint256 _poolId) private poolExists(_poolId) {
399        Pool storage pool = pools[_poolId];
400        PoolUser storage poolUser = pool.users[msg.sender];
401        updatePoolRewardInfo(_poolId, msg.sender);
402        if (poolUser.rewardsAmountWithdrawable > 0) {
403            pool.rewardsToken.safeTransfer(
404                msg.sender,
405                poolUser.rewardsAmountWithdrawable
406            );
407            pool.rewardsAmountAvailable = pool.rewardsAmountAvailable.sub(
408                poolUser.rewardsAmountWithdrawable
409            );
410            emit PoolUserRewardClaimed(
411                _poolId,
412                msg.sender,
413                poolUser.rewardsAmountWithdrawable
414            );
415            poolUser.rewardsAmountWithdrawable = 0;
416        }
417    }
```

Listing 3.13: `GoldenCatRewardPool::_claimReward()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`. The similar issue is also present in another function, i.e., the same function from the `TokenRewardPool` contract, and the adherence of the `checks-effects-interactions` best practice is strongly recommended.

From another perspective, the current mitigation in applying money-market-level reentrancy protection can be strengthened by elevating the reentrancy protection at the `BankController`-level. In addition, each individual function can be self-strengthened by following the `checks-effects-interactions` principle

Recommendation Apply necessary reentrancy prevention by following the `checks-effects-interactions` principle and utilizing the necessary `nonReentrant` modifier to block possible `re-entrancy`. Also consider strengthening the reentrancy protection at the protocol-level instead of at the current money-market granularity.

Status The issue has been fixed by this commit: `e4d559a`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Amy protocol, which is a layer 2 based, liquidity provider (LP)-friendly lending and leveraged-yield farming protocol. The system continues the innovative design and makes it distinctive and valuable when compared with current lending/yield farming offerings. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-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-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

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

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

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

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

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

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

[14] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[15] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.