SMART CONTRACT AUDIT REPORT

for

Alpaca USD (AUSD)

Prepared By: Yiqun Chen

PeckShield

November 22, 2021

Document Properties

Client	Alpaca Finance
Title	Smart Contract Audit Report
Target	Alpaca USD
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang, Patrick Liu
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 22, 2021	Xuxian Jiang	Final Release
1.0-rc	November 16, 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 4
- 1.1 About Alpaca . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

2 Findings 9
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

3 Detailed Results 11
- 3.1 Flawed Authentication in BookKeeper::setAccessControlConfig() . . . 11
- 3.2 ERC20 Compliance Of AlpacaStableCoin . . . 12
- 3.3 Proper Decimal Enforcement in TokenAdapter . . . 15
- 3.4 Improved Precision By Multiplication And Division Reordering . . . 16
- 3.5 Improved Overflow Validation in Multiple Contracts . . . 17
- 3.6 Suggested Adherence Of Checks-Effects-Interactions Pattern . . . 18
- 3.7 Trust Issue of Admin Keys . . . 20
- 3.8 Suggested safeTransfer()/safeTransferFrom() Replacement . . . 21
- 3.9 Fork-Compliant Domain Separator in AlpacaStablecoin . . . 23
- 3.10 Excess Stake Stealing From Permissionless moveStake() . . . 25
- 3.11 Possible DoS Against SystemDebtEngine And LiquidationEngine . . . 27

4 Conclusion 29

References 30

1 Introduction

Given the opportunity to review the design document and related source code of the `Alpaca USD` 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 Alpaca

`Alpaca Finance` is the largest lending protocol allowing leveraged yield farming on `Binance Smart Chain (BSC)`. The audited implementation is the latest product of the platform which is `Alpaca USD (AUSD)`. `AUSD` is a stablecoin pegged to 1 `US Dollar` collateralized by farmable and yield generating assets to maximize capital efficiency. The implementation contains various contracts that support the minting of `AUSD`, `AUSD` debt repayment, debt liquidation, position management, peg stability mechanism, borrowing interest collection and farmable collateral assets. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	Alpaca Finance
Website	https://alpacafinance.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 22, 2021

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

- https://github.com/alpaca-finance/alpaca-stablecoin.git (d51953e)

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

- https://github.com/alpaca-finance/alpaca-stablecoin.git (63d240b)

1.2 About PeckShield

PeckShield Inc. [15] 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 [14]:

- 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) [13], 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 implementation of the `Alpaca USD (AUSD)` 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	2	■■
Medium	2	■■
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 2 high-severity vulnerabilities, 2 medium-severity vulnerabilities, 6 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Audit Findings of Alpaca USD Protocol

ID	Severity	Title	Category	Status
PVE-001	High	Flawed Authentication in BookKeeper::setAccessControlConfig()	Security Features	Fixed
PVE-002	Informational	ERC20 Compliance Of AlpacaStableCoin	Coding Practices	Fixed
PVE-003	Low	Proper Decimal Enforcement in TokenAdapter	Numeric Errors	Fixed
PVE-004	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed
PVE-005	Low	Improved Overflow Validation in Multiple Contracts	Coding Practices	Fixed
PVE-006	Low	Suggested Adherence Of Checks-Effects-Interactions Pattern	Time and State	Fixed
PVE-007	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-008	Low	Suggested safeTransfer()/safeTransferFrom() Replacement	Business Logic	Fixed
PVE-009	Low	Fork-Compliant Domain Separator in AlpacaStablecoin	Business Logic	Fixed
PVE-010	Medium	Excess Stake Stealing From Permissionless moveStake()	Security Features	Fixed
PVE-011	High	Possible DoS Against SystemDebtEngine And LiquidationEngine	Business Logic	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 Flawed Authentication in BookKeeper::setAccessControlConfig()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `BookKeeper`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [4]

Description

The `Alpaca USD` protocol is inspired from the `Maker` protocol, also known as the `Multi-Collateral Dai (MCD)` system, and allows users to generate `AUSD` by leveraging collateral assets approved by the governance. To facilitate the governance of the entire protocol, there is an essential `accessControlConfig` contract that contains the protocol-wide settings for access control management. While reviewing this contract, we notice the key `setter` function on the `accessControlConfig` state may be overwritten for malicious purposes.

To elaborate, we show below the related `setAccessControlConfig()` function. As the name indicates, this function supports the on-chain re-configuration of current `accessControlConfig`. The logic is rather straightforward in authenticating the caller and then validating (and applying) the given `_accessControlConfig`. Our analysis shows that the caller-authentication logic is flawed as it misuses the given `_accessControlConfig` input for authentication, instead of the saved state of `accessControlConfig`.

```
function setAccessControlConfig(address _accessControlConfig) external {
  require(
    IAccessControlConfig(_accessControlConfig).hasRole(
      IAccessControlConfig(_accessControlConfig).OWNER_ROLE(),
      msg.sender
    ),
    "!ownerRole"
```

```
    );

    IAccessControlConfig(_accessControlConfig).hasRole(
      IAccessControlConfig(_accessControlConfig).OWNER_ROLE(),
      msg.sender
    ); // Sanity Check Call
    accessControlConfig = _accessControlConfig;

    emit LogSetAccessControlConfig(msg.sender, _accessControlConfig);
  }
```

Listing 3.1: `BookKeeper::setAccessControlConfig()`

Recommendation Properly authenticate the caller before the current `accessControlConfig` state may be updated.

Status The issue has been fixed in the following commit: 7c8772c.

3.2 ERC20 Compliance Of AlpacaStableCoin

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `AlpacaStableCoin`
- Category: Coding Practices [9]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, the `Alpaca USD` protocol is designed to mint `AUSD`, a stablecoin that is pegged to 1 `US Dollar` with necessary collateralization of farmable and yield-generating assets for improved capital efficiency. In the following, we examine the ERC20 compliance of the `AUSD` token contract.

Specifically, the ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Our analysis shows that there is a minor ERC20 inconsistency or incompatibility issue. Specifically, the current implementation has defined the `decimals` state with the `uint256` type. The ERC20 specification indicates the type of `uint8` for the `decimals` state. Note that this incompatibility issue does not necessarily affect the functionality of `AUSD` in any negative way.

Table 3.1: Basic `View-Only` Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Recommendation Revise the `AlpacaStableCoin` implementation to ensure its ERC20-compliance.

Status The issue has been fixed in the following commit: 796f76f.

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	✓
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	✓
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	—
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	—
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

3.3 Proper Decimal Enforcement in TokenAdapter

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TokenAdapter`
- Category: Numeric Errors [12]
- CWE subcategory: CWE-190 [2]

Description

The `Alpaca USD` protocol provides `GenericTokenAdapter` contracts that allow existing assets to be used as collateral to interact with the core `BookKeeper` contract for `AUSD` collateralization and issuance. Among supported `GenericTokenAdapter` contracts, the `TokenAdapter` contract is designed for well-behaved ERC20 tokens with simple transfer semantics. When examining its internal logic, we notice the implicit requirement of the `TokenAdapter`-supported `collateralToken` to have a fixed decimal. Note the decimal plays a critical role to normalize the token amount for deposits and withdraws.

```
function initialize(
  address _bookKeeper,
  bytes32 collateralPoolId_,
  address collateralToken_
) external initializer {
  PausableUpgradeable.__Pausable_init();
  ReentrancyGuardUpgradeable.__ReentrancyGuard_init();

  live = 1;
  bookKeeper = IBookKeeper(_bookKeeper);
```

```
    collateralPoolId = collateralPoolId_;
    collateralToken = collateralToken_;
    decimals = IToken(collateralToken).decimals();
  }
```

Listing 3.2: `TokenAdapter::initialize()`

Specifically, the implicit assumption is that the `collateralToken` should have the fixed 18 decimals. However, current initialization routine (as shown above) indicates that this decimal is obtained from the given `collateralToken`. In other words, it depends on the given input without the proper enforcement on the smart contract. Note that a non-18 `collateralToken` decimal may lead to unexpected results for the token deposit and withdraw operations.

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

Status The issue has been fixed in the following commit: 501af65.

3.4 Improved Precision By Multiplication And Division Reordering

- ID: PVE-004
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Multiple Contracts`
- Category: Numeric Errors [12]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `FixedSpreadLiquidationStrategy::_calculateLiquidationInfo()` as an example. This routine is used to calculate various metrics for a specific liquidation.

```
{
      // If the remaining debt after liquidation is smaller than `debtFloor`
      if (
        _positionDebtValue > info.actualDebtValueToBeLiquidated &&
        _positionDebtValue.sub(info.actualDebtValueToBeLiquidated) < _vars.debtFloor
```

```
217       ) {
218         // Full Debt Liquidation
219         info.actualDebtValueToBeLiquidated = _positionDebtValue; // [rad]
220         // actualDebtValueToBeLiquidated [rad] * liquidatorIncentiveBps [bps] / 10000 /
              _currentCollateralPrice [ray] /

222         info.collateralAmountToBeLiquidated = info
223           .actualDebtValueToBeLiquidated
224           .div(10000)
225           .mul(_vars.liquidatorIncentiveBps)
226           .div(_currentCollateralPrice); // [wad]
227       } else {
228         // Partial Liquidation
229         info.collateralAmountToBeLiquidated = _maxCollateralAmountToBeLiquidated; // [
              wad]
230       }
```

Listing 3.3: FixedSpreadLiquidationStrategy :: _calculateLiquidationInfo ()

We notice the calculation of the resulting `collateralAmountToBeLiquidated` (lines 222 – 226) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `info.actualDebtValueToBeLiquidated.mul(_vars.liquidatorIncentiveBps).div(_currentCollateralPrice).mul(10000)`. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible. Note the `ibTokenAdapter::_pendingRewards()` routine can be similarly improved.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been fixed in the following commit: 5151c6f.

3.5 Improved Overflow Validation in Multiple Contracts

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [9]
- CWE subcategory: CWE-563 [5]

Description

As mentioned in Section 3.4, `SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. With the extended support of signed integer `int256`, the overflow prevention needs to be properly enforced. In particular, the maximum `int256` is `2**255-1`, instead of `2**255`.

While reviewing the arithmetic operations with the enhancement to block `int256`-related overflows, we notice the enforcement of the maximum `int256` can be improved. In particular, if we examine the `withdraw()` function from the `TokenAdapter` contract, the enforcement of `require(wad <= 2**255)` accidentally uses `2**255` as the maximum `int256`. As a result, the current enforcement needs to be revised as `require(wad < 2**255)`. Note this issue is applicable to a number of contracts, including `ShowStopper`, `FixedSpreadLiquidationStrategy`, `TokenAdapter`, `IbTokenAdapter`, and `LiquidationEngine`.

```
/// @dev Withdraw token from the system to the caller
/// @param usr The destination address to receive collateral token
/// @param wad The amount of collateral to be withdrawn [wad]
function withdraw(
  address usr,
  uint256 wad,
  bytes calldata /* data */
) external override nonReentrant whenNotPaused {
  require(wad <= 2**255, "TokenAdapter/overflow");
  bookKeeper.addCollateral(collateralPoolId, msg.sender, -int256(wad));

  // Move the actual token
  address(collateralToken).safeTransfer(usr, wad);
}
```

Listing 3.4: TokenAdapter::withdraw()

Recommendation Use the right (maximum) number of `int256` for the overflow prevention.

Status The issue has been fixed in the following commit: 3264f09.

3.6 Suggested Adherence Of Checks-Effects-Interactions Pattern

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `IbTokenAdapter`
- Category: Time and State [11]
- CWE subcategory: CWE-663 [6]

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` [17] exploit, and the recent `Uniswap/Lendf.Me` hack [16].

We notice there is an occasion where the `checks-effects-interactions` principle is violated. Using the `IbTokenAdapter` as an example, the `_emergencyWithdraw()` 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`.

Apparently, the interaction with the external contract (line 376) starts before effecting the update on internal states (lines 377−379), 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.

```
369 /// @dev EMERGENCY ONLY. Withdraw collateralTokens from staking contract without
        invoking _harvest
370 /// @param _to The address to received collateralTokens
371 function _emergencyWithdraw(address _to) private {
372   uint256 _share = bookKeeper.collateralToken(collateralPoolId, msg.sender); //[wad]
373   require(_share <= 2**255, "IbTokenAdapter/share-overflow");
374   uint256 _amount = wmul(wmul(_share, netAssetPerShare()), toTokenConversionFactor);
375   address(collateralToken).safeTransfer(_to, _amount);
376   bookKeeper.addCollateral(collateralPoolId, msg.sender, -int256(_share));
377   totalShare = sub(totalShare, _share);
378   stake[msg.sender] = sub(stake[msg.sender], _share);
379   rewardDebts[msg.sender] = rmulup(stake[msg.sender], accRewardPerShare);
380   emit LogEmergencyWithdaraw();
381 }
```

Listing 3.5: IbTokenAdapter::_emergencyWithdraw()

In the meantime, we should mention that 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`. However, it is important to take precautions in making use of `nonReentrant` to block possible `re-entrancy` and the adherence of `checks-effects-interactions` best practice is highly recommended.

Recommendation Apply necessary reentrancy prevention by following the known checks-effects-interactions pattern in addition to the utilization of the `nonReentrant` modifier to block possible `re-entrancy`.

Status The issue has been fixed in the following commit: 4caeb32.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [8]
- CWE subcategory: CWE-287 [4]

Description

In the `Alpaca USD` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and price oracle 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 the related privileged accesses in current contracts.

To elaborate, we show below the `mint()` routine in the `AlpacaStablecoin` contract. This routine allows the privileged account with `MINTER_ROLE` to mint additional stablecoins into circulation. And this `MINTER_ROLE` is managed by the current owner with `OWNER_ROLE`.

```
function mint(address _usr, uint256 _wad) external override {
  require(hasRole(MINTER_ROLE, msg.sender), "!minterRole");

  balanceOf[_usr] = add(balanceOf[_usr], _wad);
  totalSupply = add(totalSupply, _wad);
  emit Transfer(address(0), _usr, _wad);
}
```

Listing 3.6: AlpacaStablecoin::mint()

Moreover, the `AccessControlConfig` contract allows the privileged `owner` with `OWNER_ROLE` to assign other roles, including `GOV_ROLE`, `ADAPTER_ROLE`, `STABILITY_FEE_COLLECTOR_ROLE`, `SHOW_STOPPER_ROLE`, `BOOK_KEEPER_ROLE`. These roles play a variety of duties and are also considered privileged.

```
contract AccessControlConfig is AccessControlUpgradeable {
  bytes32 public constant OWNER_ROLE = DEFAULT_ADMIN_ROLE;
  bytes32 public constant GOV_ROLE = keccak256("GOV_ROLE");
  bytes32 public constant PRICE_ORACLE_ROLE = keccak256("PRICE_ORACLE_ROLE");
  bytes32 public constant ADAPTER_ROLE = keccak256("ADAPTER_ROLE");
  bytes32 public constant LIQUIDATION_ENGINE_ROLE = keccak256("LIQUIDATION_ENGINE_ROLE");
  bytes32 public constant STABILITY_FEE_COLLECTOR_ROLE = keccak256("STABILITY_FEE_COLLECTOR_ROLE");
  bytes32 public constant SHOW_STOPPER_ROLE = keccak256("SHOW_STOPPER_ROLE");
  bytes32 public constant POSITION_MANAGER_ROLE = keccak256("POSITION_MANAGER_ROLE");
  bytes32 public constant MINTABLE_ROLE = keccak256("MINTABLE_ROLE");
  bytes32 public constant BOOK_KEEPER_ROLE = keccak256("BOOK_KEEPER_ROLE");
  ...
```

```
}
```

Listing 3.7: The AccessControlConfig Contract

It is worrisome if the privileged `owner` account is a plain EOA account. 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 if current contracts are to 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.

3.8 Suggested safeTransfer()/safeTransferFrom() Replacement

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `AuthTokenAdapter`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

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 3.8: 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 `deposit()` routine in the `AuthTokenAdapter` contract. If the `USDT` token is supported as `token`, the unsafe version of `token.transferFrom(_msgSender, address(this), _wad)` (line 111) 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)! Note the same issue is also applicable in the `withdraw()` counterpart.

```
101   function deposit(
102     address _urn,
103     uint256 _wad,
104     address _msgSender
105   ) external override nonReentrant whenNotPaused {
106     require(hasRole(WHITELISTED, msg.sender), "AuthTokenAdapter/not-whitelisted");
107     require(live == 1, "AuthTokenAdapter/not-live");
108     uint256 _wad18 = mul(_wad, 10**(18 - decimals));
109     require(int256(_wad18) >= 0, "AuthTokenAdapter/overflow");
110     bookKeeper.addCollateral(collateralPoolId, _urn, int256(_wad18));
111     require(token.transferFrom(_msgSender, address(this), _wad), "AuthTokenAdapter/
          failed-transfer");
112     emit LogDeposit(_urn, _wad, _msgSender);
113   }
```

Listing 3.9: AuthTokenAdapter::deposit()

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

Status The issue has been fixed in the following commit: 1fbca35.

3.9 Fork-Compliant Domain Separator in AlpacaStablecoin

- ID: PVE-009
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `AlpacaStablecoin`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

The `AlpacaStablecoin` token contract strictly follows the widely-accepted ERC20 specification (Section 3.2. In the meantime, we notice the support of EIP-2612 with the `permit()` function that allows for approvals to be made via `secp256k1` signatures. Interestingly, we notice the state variable `DOMAIN_SEPARATOR` is initialized once inside the `initialize()` function (lines 67-75).

```
56   // --- Init ---
57   function initialize(
58     string memory _name,
59     string memory _symbol,
60     uint256 _chainId
61   ) external initializer {
62     AccessControlUpgradeable.__AccessControl_init();
63 
64     name = _name;
65     symbol = _symbol;
66 
67     DOMAIN_SEPARATOR = keccak256(
68       abi.encode(
69         keccak256("EIP712Domain(string name,string version,uint256 chainId,address
             verifyingContract)"),
70         keccak256(bytes(name)),
71         keccak256(bytes(version)),
72         _chainId,
73         address(this)
74       )
75     );
76 
77     // Grant the contract deployer the default admin role: it will be able
78     // to grant and revoke any roles
79     _setupRole(OWNER_ROLE, msg.sender);
80   }
```

Listing 3.10: `AlpacaStablecoin::initialize()`

The DOMAIN_SEPARATOR is used in the permit() function and should be unique to the contract and chain in order to prevent replay attacks from other domains. However, when analyzing this permit() routine, we realize the current implementation needs to be improved by recalculating the value of DOMAIN_SEPARATOR inside the permit() function, for the very purpose of preventing cross-chain replay attacks. Specifically, when there is a chain-level hard-fork, because of the pre-computed DOMAIN_SEPARATOR, a valid signature for one chain could be replayed on the other.

```
    // --- Approve by signature ---
    function permit(address holder, address spender, uint256 nonce, uint256 expiry,
                    bool allowed, uint8 v, bytes32 r, bytes32 s) external
    {
        bytes32 digest = keccak256(abi.encodePacked(
                "\x19\x01",
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(PERMIT_TYPEHASH,
                                     holder,
                                     spender,
                                     nonce,
                                     expiry,
                                     allowed))
        ));

        require(holder != address(0), "VAI/invalid-address-0");
        require(holder == ecrecover(digest, v, r, s), "VAI/invalid-permit");
        require(expiry == 0  now <= expiry, "VAI/permit-expired");
        require(nonce == nonces[holder]++, "VAI/invalid-nonce");
        uint wad = allowed ? uint(-1) : 0;
        allowance[holder][spender] = wad;
        emit Approval(holder, spender, wad);
    }
```

Listing 3.11: VAI::permit()

Recommendation Recalculate the value of DOMAIN_SEPARATOR inside the permit() function.

Status The issue has been fixed in the following commit: 933de39.

3.10 Excess Stake Stealing From Permissionless moveStake()

- ID: PVE-010
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: IbTokenAdapter
- Category: Security Features [8]
- CWE subcategory: CWE-269 [3]

Description

To maximize capital efficiency, the Alpaca USD protocol is unique in being collateralized by farmable and yield-generating assets. To support these yield-generating assets, the contract IbTokenAdapter needs to keep track of the stake from protocol users. While reviewing the accounting logic of staked balance, we notice a number of functions are permission-less and they may be used to grab excess stake from an unknowing user.

To elaborate, we show below the related IbTokenAdapter::moveStake() function. It is designed to move certain amount of staked balance from source to destination. A number of sanity checks are in place to ensure the _source has sufficient collateral and the _destination is prevented from claiming more stake than the actual collateral. However, the current logic allows for excess stake from the _source to be stolen.

```
function moveStake(
  address _source,
  address _destination,
  uint256 _share,
  bytes calldata _data
) external override nonReentrant whenNotPaused {
  _moveStake(_source, _destination, _share, _data);
}

/// @dev Move wad amount of staked balance from source to destination.
/// Can only be moved if underlying assets make sense.
/// @param _source The address to be moved staked balance from
/// @param _destination The address to be moved staked balance to
/// @param _share The amount of staked balance to be moved
function _moveStake(
  address _source,
  address _destination,
  uint256 _share,
  bytes calldata /* data */
) private {
  // 1. Update collateral tokens for source and destination
  uint256 _stakedAmount = stake[_source];
  stake[_source] = sub(_stakedAmount, _share);
  stake[_destination] = add(stake[_destination], _share);
  // 2. Update source's rewardDebt due to collateral tokens have
```

```
    // moved from source to destination. Hence, rewardDebt should be updated.
    // rewardDebtDiff is how many rewards has been paid for that share.
    uint256 _rewardDebt = rewardDebts[_source];
    uint256 _rewardDebtDiff = mul(_rewardDebt, _share) / _stakedAmount;
    // 3. Update rewardDebts for both source and destination
    // Safe since rewardDebtDiff <= rewardDebts[source]
    rewardDebts[_source] = _rewardDebt - _rewardDebtDiff;
    rewardDebts[_destination] = add(rewardDebts[_destination], _rewardDebtDiff);
    // 4. Sanity check.
    // - stake[source] must more than or equal to collateral + lockedCollateral that
        source has
    // to prevent a case where someone try to steal stake from source
    // - stake[destination] must less than or equal to collateral + lockedCollateral
        that destination has
    // to prevent destination from claim stake > actual collateral that he has
    (uint256 _lockedCollateral, ) = bookKeeper.positions(collateralPoolId, _source);
    require(
      stake[_source] >= add(bookKeeper.collateralToken(collateralPoolId, _source),
          _lockedCollateral),
      "IbTokenAdapter/stake[source] < collateralTokens + lockedCollateral"
    );
    (_lockedCollateral, ) = bookKeeper.positions(collateralPoolId, _destination);
    require(
      stake[_destination] <= add(bookKeeper.collateralToken(collateralPoolId,
          _destination), _lockedCollateral),
      "IbTokenAdapter/stake[destination] > collateralTokens + lockedCollateral"
    );
    emit LogMoveStake(_source, _destination, _share);
  }
```

Listing 3.12: IbTokenAdapter::moveStake()

The same issue is also applicable to two other functions, i.e., `onAdjustPosition()` and `onMoveCollateral()`.

Recommendation Revise the permission-less design of the above three affected functions.

Status The issue has been fixed in the following commits: 644a845 and bb4fc3e.

3.11 Possible DoS Against SystemDebtEngine And LiquidationEngine

- ID: PVE-011
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [10]
- CWE subcategory: CWE-841 [7]

Description

As mentioned in Section 3.10, the `Alpaca USD` protocol is unique in being collateralized by farmable and yield-generating assets. The support is mainly implemented in the `IbTokenAdapter` contract. In last section, we have exposed potential side-effect from the permission-less `moveStake()` function. In this section, we examine a possible denial-of-service issue that affects both `SystemDebtEngine` and `LiquidationEngine` contracts.

The issue stems from the following requirement, i.e., `require(stake[_source] >= add(bookKeeper.collateralToken(collateralPoolId, _source), _lockedCollateral))` (line 423), which essentially ensures the `_source` must have sufficient stake to cover the deposited collateral and the position. However, this requirement may be misused if a malicious actor may donate additional collateral to the `_source`. (The donation may be performed in a trustless manner via the `BookKeeper` contract.) As a result, the `_moveStake()` helper may be unfortunately reverted. And the reversion may affect a number of calling functions from `SystemDebtEngine` and `LiquidationEngine` contracts.

```
392   /// @dev Move wad amount of staked balance from source to destination.
393   /// Can only be moved if underlying assets make sense.
394   /// @param _source The address to be moved staked balance from
395   /// @param _destination The address to be moved staked balance to
396   /// @param _share The amount of staked balance to be moved
397   function _moveStake(
398     address _source,
399     address _destination,
400     uint256 _share,
401     bytes calldata /* data */
402   ) private {
403     // 1. Update collateral tokens for source and destination
404     uint256 _stakedAmount = stake[_source];
405     stake[_source] = sub(_stakedAmount, _share);
406     stake[_destination] = add(stake[_destination], _share);
407     // 2. Update source's rewardDebt due to collateral tokens have
408     // moved from source to destination. Hence, rewardDebt should be updated.
409     // rewardDebtDiff is how many rewards has been paid for that share.
410     uint256 _rewardDebt = rewardDebts[_source];
411     uint256 _rewardDebtDiff = mul(_rewardDebt, _share) / _stakedAmount;
```

```
    // 3. Update rewardDebts for both source and destination
    // Safe since rewardDebtDiff <= rewardDebts[source]
    rewardDebts[_source] = _rewardDebt - _rewardDebtDiff;
    rewardDebts[_destination] = add(rewardDebts[_destination], _rewardDebtDiff);
    // 4. Sanity check.
    // - stake[source] must more than or equal to collateral + lockedCollateral that source has
    // to prevent a case where someone try to steal stake from source
    // - stake[destination] must less than or equal to collateral + lockedCollateral that destination has
    // to prevent destination from claim stake > actual collateral that he has
    (uint256 _lockedCollateral, ) = bookKeeper.positions(collateralPoolId, _source);
    require(
      stake[_source] >= add(bookKeeper.collateralToken(collateralPoolId, _source), _lockedCollateral),
      "IbTokenAdapter/stake[source] < collateralTokens + lockedCollateral"
    );
    (_lockedCollateral, ) = bookKeeper.positions(collateralPoolId, _destination);
    require(
      stake[_destination] <= add(bookKeeper.collateralToken(collateralPoolId, _destination), _lockedCollateral),
      "IbTokenAdapter/stake[destination] > collateralTokens + lockedCollateral"
    );
    emit LogMoveStake(_source, _destination, _share);
  }
```

Listing 3.13: IbTokenAdapter::_moveStake()

Recommendation With the support of yield-generating assets, there is a need to make consistency between the stake balance and the actual collateral deposited as well as the debt maintained in `BookKeeper`.

Status The issue has been fixed in the following commits: 644a845 and bb4fc3e.

4 Conclusion

In this audit, we have analyzed the design and implementation of `Alpaca USD (AUSD)`, which is a stablecoin pegged to 1 `US Dollar` collateralized by farmable and yield generating assets to maximize capital efficiency. The current code base is well organized and those identified issues are promptly confirmed and addressed.

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-269: Improper Privilege Management. https://cwe.mitre.org/data/definitions/269.html.

[4] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[5] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

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

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

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

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

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

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

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

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

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

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