SMART CONTRACT AUDIT REPORT

for

ONDO

Prepared By: Shuxiao Wang

PeckShield
May 16, 2021

Document Properties

Client	Ondo
Title	Smart Contract Audit Report
Target	Ondo
Version	1.0
Author	Xuxian Jiang
Auditors	Yiqun Chen, Xuxian Jiang, Huaguo Shi
Reviewed by	Shuxiao Wang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 16, 2021	Xuxian Jiang	Final Release
1.0-rc1	May 10, 2021	Xuxian Jiang	Release Candidate #1
0.3	May 3, 2021	Xuxian Jiang	Add More Findings #2
0.2	April 30, 2021	Xuxian Jiang	Add More Findings #1
0.1	April 26, 2021	Xuxian Jiang	Initial Draft

Contact

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

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

Contents

1 | Introduction

Given the opportunity to review the **Ondo** design document and related smart contract source code, 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 Ondo

The goal of Ondo is to allow investors to shift the risk and reward balance between each other. In particular, Ondo classifies investors into two groups: the `senior` and `junior tranche`. The `senior tranche` will receive a fixed percentage over their initial investments. The `junior tranche` will receive any excess returns over the senior tranche. This fixed percentage, called the `hurdle rate`, is determined when the product is created. The underlying investment is liquidity provider tokens on decentralized exchanges, e.g. `Uniswap`, `Sushiswap`, `Balancer`, `Curve`, etc. Liquidity providers inject an equal value of a pair of assets into a liquidity pool. In return they earn fees and incentives for providing liquidity, which is collected by withdrawing liquidity from the pool.

The basic information of Ondo is as follows:

Table 1.1: Basic Information of Ondo

Item	Description
Issuer	Ondo
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 16, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that Ondo assumes a trusted price oracle with timely market price feeds for supported assets and the oracle itself is not part of this audit.

- https://github.com/ondoprotocol/protocol-audit.git (29b2c9a)

1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on the OWASP Risk Rating Methodology [9]:

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

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

Table 1.3: The Full Audit Checklist

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

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

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

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

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered 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 Logic	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the Ondo protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	5	■■■■■
Low	5	■■■■■
Informational	0	
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 need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 5 medium-severity vulnerabilities, and 5 low-severity vulnerabilities.

Table 2.1: Key Ondo Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Improved Logic in Excess Withdrawal	Security Features	Fixed
PVE-002	Low	Duplicate Removal in Registry::tokensDeclaredDead()	Business Logic	Fixed
PVE-003	Low	Explicit Input Validation in AllPairCCO::transition()	Business Logic	Fixed
PVE-004	Low	Improved Sanity Checks Of System/Function Parameters	Coding Practices	Fixed
PVE-005	Medium	Incorrect Performance Fee Calculation	Business Logic	Confirmed
PVE-006	Low	Redundant Code Removal	Coding Practices	Fixed
PVE-007	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Fixed
PVE-008	Medium	Trust Issue of Admin Keys	Security Features	Confirmed
PVE-009	Medium	Possible Front-Running For Reduced Return	Business Logic	Fixed
PVE-010	High	Improved Business Logic in RolloverCCO::deposit()	Business Logic	Fixed
PVE-011	Medium	Potentially Repeated Excess Returns Of RolloverCCO	Business Logic	Resolved

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Improved Logic in Excess Withdrawal

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: BasePairLPStrategy
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The Ondo protocol has a number of essential contracts for different functionalities and duties: strategies, AllPairCCO, RolloverCCO, Registry, and TrancheToken. While examining the BasePairLPStrategy contract, we notice one public function withdrawExcess() is not properly guarded.

To elaborate, we show below the withdrawExcess() routine that is designed to transfer any unused deposits back to the investor. We notice that the current logic employs the isAuthorized(OLib.CCO_ROLE) modifier to require CCO_ROLE authorization. However, the proper authorization should be more explicit, i.e., onlyOrigin(_ccoId). By doing so, only the origin of the intended _ccoID may able to request the return of excess tokens, not any account with the CCO_ROLE authorization.

```
/**
 * @notice Send excess tokens to investor
 */
function withdrawExcess(
  uint256 _ccoId,
  OLib.Tranche tranche,
  address to,
  uint256 amount
) external override isAuthorized(OLib.CCO_ROLE) {
  CCO storage _cco = ccos[_ccoId];
  if (tranche == OLib.Tranche.Senior) {
    uint256 excess = _cco.seniorExcess;
    require(amount <= excess, "Withdrawing too much");
    _cco.seniorExcess -= amount;
    _cco.senior.safeTransfer(to, amount);
```

```
    } else {
      uint256 excess = _cco.juniorExcess;
      require(amount <= excess, "Withdrawing too much");
      _cco.juniorExcess -= amount;
      _cco.junior.safeTransfer(to, amount);
    }
  }
```

Listing 3.1: BasePairLPStrategy::withdrawExcess()

Recommendation Authenticate the caller of `withdrawExcess()` to be `onlyOrigin(_ccoId)`, not the current `isAuthorized(OLib.CCO_ROLE)`.

Status This issue has been fixed in this commit: `1e9db80`.

3.2 Duplicate Removal in Registry::tokensDeclaredDead()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Registry`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Ondo` protocol has a protocol-wide registry `Registry` with a number of access control-related configurations. Our analysis shows that the contract also maintains an internal list of `TrancheToken` instances that can be recycled for gas efficiency.

To elaborate, we show below the responsible `tokensDeclaredDead()` routine. As the name indicates, this routine adds the given list of tokens into an internal array `deadTokens` where the governance can delete to save gas. However, it comes to our attention that the given list is not validated for any duplicate.

```
  /**
   * @notice Manually determine which TrancheToken instances can be recycled
   * @dev Move into another list where createCCO can delete to save gas. Done manually
       for safety.
   * @param _tokens List of tokens
   */
  function tokensDeclaredDead(address[] calldata _tokens)
    external
    onlyGovernance
  {
    for (uint256 i = 0; i < _tokens.length; i++) {
      deadTokens.push(ITrancheToken(_tokens[i]));
    }
```

```
}
```

Listing 3.2: Registry :: tokensDeclaredDead()

Recommendation Revise the `tokensDeclaredDead()` logic to not save into the internal array `deadTokens` with duplicates

Status This issue has been fixed in this commit: `2156b55`.

3.3 Explicit Input Validation in AllPairCCO::transition()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AllPairCCO`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `Ondo` protocol, there is a core `AllPairCCO` contract that contains most of the implementation for `Collateralized Crypto Obligations (CCO)s`. Specifically, rather than creating unique contract instances for each `CCO`, the state for all `CCOs` is stored in a mapping. Each `CCO` has a unique id number created by hashing the metadata on the `CCO`: `asset pair`, `strategy contract`, `start time`, `investment time`, and `duration`. Each `CCO` has to be one of four self-evident states: `Inactive`, `Deposit`, `Live`, and `Withdraw`. In order to facilitate the (linear) state transition among these four states, a helper routine `transition()` is provided.

To elaborate, we show below the `transition()` helper routine. Our analysis shows one specific transition `Live -> Withdraw` can be improved by explicitly enforcing the required states. In particular, the current logic only enforces `require(curState == OLib.State.Live)` (line 120), which can be improved as `require(curState == OLib.State.Live && _nextState == OLib.State.Withdraw )`.

```
106   // Determine if one can move to a new state. For now the transitions
107   // are strictly linear. No state machines, really.
108   modifier transition(uint256 _ccoId, OLib.State _nextState) {
109     CCO storage cco_ = CCOs[_ccoId];
110     OLib.State curState = cco_.state;
111     if (_nextState == OLib.State.Live) {
112       require(
113         curState == OLib.State.Deposit,
114         //        "Cannot transition to Live from current state"
115         invalidTransitionMsg
116       );
117       require(cco_.investAt <= block.timestamp, "Not yet time to invest");
118     } else {
```

```
      require(
        curState == OLib.State.Live,
        //         "Cannot transition to Withdraw from current state"
        invalidTransitionMsg
      );
      require(cco_.redeemAt <= block.timestamp, "Not yet time to redeem");
    }
    cco_.state = _nextState;
    CCOsByState[curState].remove(_ccoId);
    CCOsByState[_nextState].add(_ccoId);
    _;
  }
```

Listing 3.3: AllPairCCO:: transition ()

Recommendation Properly strengthen the `transition()` logic by making state transition explicit.

Status This issue has been fixed in this commit: `7e82cf9`.

3.4 Improved Sanity Checks For System Parameters

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Ondo` protocol is no exception. In the following, we examine a number of routines that can be improved to better validate the given input.

The first routine `AllPairCCO::maybeOpenDeposit()` determines whether a given `CCO` can shift to an open state. Besides the current validation, it is better to add the following requirement, i.e., `require(cco_.startAt > 0 && cco_.startAt <= block.timestamp)`, to ensure the `startAt` field is indeed valid.

```
  // Determine if a CCO can shift to an open state. A CCO is started
  // in an inactive state. It can only move forward when time has
  // moved past the starttime.
  modifier maybeOpenDeposit(uint256 _ccoId) {
    CCO storage cco_ = CCOs[_ccoId];
    if (cco_.state == OLib.State.Inactive) {
      require(cco_.startAt <= block.timestamp, "Not yet time to enroll");
```

```
        cco_.state = OLib.State.Deposit;
        CCOsByState[OLib.State.Inactive].remove(_ccoId);
        CCOsByState[OLib.State.Deposit].add(_ccoId);
      } else if (cco_.state != OLib.State.Deposit) {
        revert("Invalid operation at current state");
      }
      _;
    }
```

Listing 3.4: AllPairCCO::maybeOpenDeposit()

The second routine `canDeposit()` determines whether the given CCO can accept investor deposits. This routine can be similarly improved by verifying `cco_.startAt > 0 && cco_.startAt <= block.timestamp`.

```
    function canDeposit(uint256 _ccoId) external view override returns (bool) {
      CCO storage cco_ = CCOs[_ccoId];
      if (cco_.state == OLib.State.Inactive) {
        return cco_.startAt <= block.timestamp;
      }
      return cco_.state == OLib.State.Deposit;
    }
```

Listing 3.5: AllPairCCO::canDeposit()

The third routine `createAndAddNextCco()` defines the next CCO for this rollover to invest. It is suggested to perform a more through validation on the given `_ccoParams` to define a CCO.

```
    function createAndAddNextCco(
      uint256 _rolloverId,
      OLib.CCOParams memory _ccoParams
    )
      external
      noPanic
      notDead(_rolloverId)
      onlyCreator(_rolloverId)
      nonReentrant
    {
      Rollover storage rollover_ = rollovers[_rolloverId];
      _ccoParams.startTime =
        ccoManager.redeemAt(rollover_.rounds[rollover_.thisRound + 1].ccoId) -
        _ccoParams.enrollment;
      uint256 newCcoId = ccoManager.createCCO(_ccoParams);
      _addNextCco(_rolloverId, newCcoId);
    }
```

Listing 3.6: RolloverCCO::createAndAddNextCco()

Recommendation Validate any untrusted input before it can be accepted for normal processing. Also, guard any changes on the system-wide parameters to ensure they fall in an appropriate range. Also, consider emitting related events for external monitoring and analytics tools.

Status This issue has been fixed in the following commits: `d1d14e5`, `8477b11`, `e802dea`, and `8f96dbb`.

3.5 Incorrect Performance Fee Calculation

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

Description

As mentioned earlier, the `Ondo` protocol allows investors to shift the risk and reward balance between each other. Currently, the protocol supports two types of investors: the `senior tranche` and `junior tranche`. The `senior tranche` will receive a fixed percentage over their initial investments. The `junior tranche` will receive any excess returns over the `senior tranche`. This fixed percentage, called the `hurdle rate`, is determined when the product is created.

In particular, we show below the `AllPairCCO::takePerformanceFee()` routine that is used to calculate the performance fee for the strategist. It comes to our attention that the calculated fee is currently based on received amount by `junior tranche`, instead of the earned amount after the `hurdle rate` reduction. As a result, the `senior tranche` may not get the expected `hurdle rate`.

```
function takePerformanceFee(CCO storage cco) internal returns (uint256 fee) {
  fee = 0;
  if (performanceFeeCollector != address(0)) {
    Asset storage junior = cco.assets[OLib.Tranche.Junior];
    uint256 juniorHurdle =
      junior
        .totalInvested
        .fromUInt()
        .mul((denominator + cco.hurdleRate).fromUInt())
        .div(denominator.fromUInt())
        .toUInt();

    if (junior.received > juniorHurdle) {
      fee = cco
        .performanceFee
        .fromUInt()
        .mul(junior.received.fromUInt())
        .div(denominator.fromUInt())
        .toUInt();
      IERC20(junior.token).safeTransferFrom(
        address(cco.strategy),
        performanceFeeCollector,
```

```
            fee
        );
      }
    }
  }
```

Listing 3.7: AllPairCCO::takePerformanceFee()

Recommendation Revise the above calculations to ensure the `senior tranche` can get the expected `hurdle rate`.

Status This issue has been confirmed.

3.6 Redundant Code Removal

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The `Ondo` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `AccessControl`, to facilitate its code implementation and organization. For example, the `AllPairCCO` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `UniswapStrategy::constructor()` routine, it contains the repeated initialization of `registry` (line 35).

```
30  constructor(
31    address _registry,
32    address _router,
33    address _factory
34  ) BasePairLPStrategy(_registry) {
35    registry = Registry(_registry);
36    uniRouter02 = IUniswapV2Router02(_router);
37    uniFactory = _factory;
38  }
```

Listing 3.8: UniswapStrategy::**constructor**()

In addition, the `updatePool()` routine can be revised as the requirement on `require(token0 != address(sushiToken)|| token1 != address(sushiToken))` is always true – as it is impossible to have a pool with the same `token0` and `token1`.

```
function updatePool(address _pool, address[] calldata pathFromSushi)
  external
  nonReentrant
  noPanic
  isAuthorized(OLib.STRATEGIST_ROLE)
{
  require(pools[_pool]._isSet, "Pool ID not yet registered");
  address token0 = IUniswapV2Pair(_pool).token0();
  address token1 = IUniswapV2Pair(_pool).token1();
  require(
    token0 != address(sushiToken)  token1 != address(sushiToken),
    "Should never need to update pool with sushi token"
  );
  address endToken = pathFromSushi[pathFromSushi.length - 1];
  require(
    IUniswapV2Pair(_pool).token0() == endToken
      IUniswapV2Pair(_pool).token1() == endToken,
    "Not a valid path for pool"
  );
  PoolData storage poolData = pools[_pool];
  delete poolData.pathFromSushi;
  pools[_pool].pathFromSushi = pathFromSushi;
}
```

Listing 3.9: SushiStrategyLP::updatePool()

Similarly, if we examine the `AllPairCCO::depositFromRollover()` routine, the internal requirement `address(cco_.rollover)== msg.sender` (line 441) is redundant as it is already guaranteed by the `onlyRollover(_ccoId, _rolloverId)` modifier.

Recommendation Consider the removal of the redundant code in above routines.

Status This issue has been fixed in the following commits: `1d53499`, `fcc8a89`, and `055bb88`.

3.7 Accommodation of approve() Idiosyncrasies

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

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 this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.10: USDT Token Contract

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. In the following, we use the `SushiStrategyLP::addLiquidity()` routine as an example. This routine is designed to approve a specific token for swap contract. To accommodate the specific idiosyncrasy, for each `safeIncreaseAllowance()`, there is a need to `approve()` twice (lines 445 – 446): the first one reduces the allowance to 0; and the second one sets the new allowance.

```
440   function addLiquidity(
441     address token0,
442     address token1,
443     uint256 amt0,
444     uint256 amt1,
445     uint256 minOut0,
446     uint256 minOut1
447   )
448     internal
449     returns (
450       uint256 out0,
451       uint256 out1,
452       uint256 lp
453     )
454   {
455     IERC20(token0).safeIncreaseAllowance(address(sushiRouter), amt0);
456     IERC20(token1).safeIncreaseAllowance(address(sushiRouter), amt1);
```

```
    (out0, out1, lp) = sushiRouter.addLiquidity(
      token0,
      token1,
      amt0,
      amt1,
      minOut0,
      minOut1,
      address(this),
      block.timestamp
    );
  }
```

Listing 3.11: SushiStrategyLP::addLiquidity()

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `transfer()` function does not have a return value. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

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. To use this library you can add a using SafeERC20 for IERC20. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

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

Status This issue has been fixed in this commit: `074dd9b`.

3.8 Trust Issue of Admin Keys

- ID: PVE-008
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `Ondo` protocol, there is a special administrative account `gov` with `GOVERNANCE_ROLE`. This `gov` account plays a critical role in governing and regulating the system-wide operations (e.g., assign various roles, specify the swap path, and set rollovers). It also has the privilege to regulate or govern the flow of assets among the involved components in the protocol. And the presence of an administrative account can allow for emergency operations.

We emphasize that current privilege assignment is necessary and required for proper protocol operation. However, it is worrisome if the `gov` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the `gov` will be managed by a multi-sig account. Note that a compromised `gov` account is capable of modifying current protocol configuaration with adverse consequences on user funds.

Recommendation Promptly transfer the `gov` privilege to the intended `DAO`-like governance contract.

Status This issue has been confirmed.

3.9 Possible Front-Running For Reduced Return

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `Ondo` protocol, there is a `SushiStrategyLP` strategy that has an additional method `harvest()` to occasionally convert earned `Sushi` into a balance of `senior` and `junior` assets to reinvest into LP tokens, which are then placed into `Masterchef` to earn `Sushi`

To elaborate, we show below the `SushiStrategyLP::harvest()` routine. This routine delegates the call to an internal `_compound()` handler to reinvest `sushi/xsushi` into LP tokens.

```
/**
 * @notice Periodically reinvest sushi/xsushi into LP tokens
 * @param pool Sushiswap pool to reinvest
 */
function harvest(address pool) external isAuthorized(OLib.STRATEGIST_ROLE) {
  PoolData storage poolData = pools[pool];
  _compound(IERC20(pool), poolData);
}
```

Listing 3.12: SushiStrategyLP::harvest()

```
function _compound(IERC20 pool, PoolData storage poolData) internal {
  uint256 sushiAmt = sushiToken.balanceOf(address(this));
  masterChef.deposit(poolData.pid, 0); // Called to trigger update in amount of sushi
      truly available now
```

```
    xSushi.leave(poolData.pendingXSushi);
    poolData.pendingXSushi = 0;
    sushiAmt = sushiToken.balanceOf(address(this)) - sushiAmt;
    if (sushiAmt > 0) {
      address[] memory pathFromSushi = getSushiPath(poolData.pathFromSushi);
      address tokenA = pathFromSushi[pathFromSushi.length - 1];
      address tokenB = IUniswapV2Pair(address(pool)).token0();
      if (tokenB == tokenA) tokenB = IUniswapV2Pair(address(pool)).token1();
      uint256 amt0;
      if (tokenA == address(sushiToken)) {
        amt0 = sushiAmt;
      } else {
        amt0 = swapExactIn(sushiAmt, 0, pathFromSushi);
      }
      uint256 amt0ToSwap;
      (uint256 reserves0, ) =
        SushiSwapLibrary.getReserves(sushiFactory, tokenA, tokenB);
      amt0 -= (amt0ToSwap = calculateSwapInAmount(reserves0, amt0));
      uint256 amt1 = swapExactIn(amt0ToSwap, 0, getPath(tokenA, tokenB));
      (, , uint256 lpAmt) = addLiquidity(tokenA, tokenB, amt0, amt1, 0, 0);
      // TODO: do something with excess - will be extremely minimal though (<2)
      poolData.totalLp += lpAmt;
    }
    pool.safeIncreaseAllowance(
      address(masterChef),
      pool.balanceOf(address(this))
    );
    masterChef.deposit(poolData.pid, pool.balanceOf(address(this)));
  }
```

Listing 3.13: SushiStrategyLP::_compound()

We notice the conversion is routed to `Sushiswap` in order to swap one token to another for liquidity addition. And the swap operation does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user or the `strategy` contract in our case because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status This issue has been fixed in this commit: `6b7f856`.

3.10 Improved Business Logic in RolloverCCO::deposit()

- ID: PVE-010
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `RolloverCCO`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `Ondo` protocol also supports `RolloverCCO` that can automate the investment process by investing in a series of similar CCOs. Investors can deposit money into the `RolloverCCO`, get a token to represent their stake, and redeem it later to collect their investment plus profits (or losses). In the following, we examine the deposit logic.

To elaborate, we use `RolloverCCO::deposit()` routine. The logic is rather straightforward in allowing investors to deposit tokens into a queue to get invested in the next `CCO`. However, it comes to our attention that when there is an excess and the excess is more than the deposited amount for investment, there is a need to reset `_amount = 0`. The reset operation is missing in current logic, resulting in possible loss for the depositing user.

```
function deposit(
  uint256 _rolloverId,
  OLib.Tranche _tranche,
  uint256 _amount
) external noPanic notDead(_rolloverId) nonReentrant {
  Rollover storage rollover_ = rollovers[_rolloverId];
  if (rollover_.investorLastUpdates[_tranche][msg.sender] == 0) {
    rollover_.investorLastUpdates[_tranche][msg.sender] = 1;
  }
  {
    Round storage round_ = rollover_.rounds[rollover_.thisRound + 1];
    uint256 ccoId = round_.ccoId;
    require(ccoId != 0, "No CCO to deposit in yet");
    require(ccoManager.canDeposit(ccoId), "CCO not in deposit state");
    TrancheRound storage trancheRound_ = round_.tranches[_tranche];
    OLib.Investor storage investor_ = trancheRound_.investors[msg.sender];
    uint256 total = trancheRound_.newDeposited += _amount;
    uint256 userSum =
      investor_.userSums.length > 0
        ? investor_.userSums[investor_.userSums.length - 1] + _amount
        : _amount;
    investor_.prefixSums.push(total);
    investor_.userSums.push(userSum);
```

```
460     }
461     if (
462       rollover_.investorLastUpdates[_tranche][msg.sender] <
463       rollover_.thisRound + 1
464     ) {
465       (uint256 shares, uint256 excess) =
466         _updateInvestor(msg.sender, _rolloverId, _tranche);
467       rollover_.investorLastUpdates[_tranche][msg.sender] =
468         rollover_.thisRound +
469         1;
470       if (excess > _amount) {
471         rollover_.assets[_tranche].safeTransfer(msg.sender, excess - _amount);
472       } else {
473         _amount -= excess;
474       }
475       if (shares > 0) {
476         rollover_.rolloverTokens[_tranche].mint(msg.sender, shares);
477       }
478     }
479     rollover_.assets[_tranche].safeTransferFrom(
480       msg.sender,
481       address(this),
482       _amount
483     );
484     emit Deposited(msg.sender, _rolloverId, _amount);
485   }
```

Listing 3.14: RolloverCCO::deposit()

Recommendation Revise the `deposit()` logic in `RolloverCCO` to transfer user funds (via `transferFrom()` at line 479) when the excess is already more than the intended amount for deposit.

Status This issue has been fixed in this commit: `5162589`.

3.11 Potentially Repeated Excess Returns Of RolloverCCO

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `RolloverCCO`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.10, the `Ondo` protocol supports `RolloverCCO` that can automate the investment process by investing in a series of similar CCOs. Our analysis shows an issue that may result in multiple returns of the same excess amount.

To elaborate, the protocol starts with the round number 0 and participating users can call `deposit()` to invest funds into the `RolloverCCO` contract. When the `migrate()` function is called to initiate the investment into the next `CCO`, the first-time investment executes the `_firstInvest()` routine (line 618), which properly advances the `rollover_.thisRound` to 1 (line 771).

```
611   function migrate(uint256 _rolloverId, SlippageSettings memory _slippage)
612     external
613     noPanic
614     notDead(_rolloverId)
615     onlyCreator(_rolloverId)
616   {
617     if (rollovers[_rolloverId].thisRound == 0) {
618       _firstInvest(
619         _rolloverId,
620         _slippage.seniorMinInvest,
621         _slippage.juniorMinInvest
622       );
623     } else {
624       _migrate(_rolloverId, _slippage);
625     }
626   }
```

Listing 3.15: RolloverCCO::migrate()

```
735   function _firstInvest(
736     uint256 _rolloverId,
737     uint256 _seniorMinInvest,
738     uint256 _juniorMinInvest
739   ) internal {
740     Rollover storage rollover_ = rollovers[_rolloverId];
741     Round storage round_ = rollover_.rounds[1];
742     uint256 ccoId = round_.ccoId;
743     require(round_.ccoId != 0, "CCO not set");
744     TrancheRound storage srRound = round_.tranches[OLib.Tranche.Senior];
745     TrancheRound storage jrRound = round_.tranches[OLib.Tranche.Junior];
746     rollover_.assets[OLib.Tranche.Senior].safeIncreaseAllowance(
747       address(ccoManager),
748       srRound.newDeposited
749     );
750     rollover_.assets[OLib.Tranche.Junior].safeIncreaseAllowance(
751       address(ccoManager),
752       jrRound.newDeposited
753     );
754     ccoManager.depositFromRollover(
755       ccoId,
756       _rolloverId,
757       srRound.newDeposited,
758       jrRound.newDeposited
759     );
760     ccoManager.invest(ccoId, _seniorMinInvest, _juniorMinInvest);
761     (srRound.invested, jrRound.invested) = ccoManager.rolloverClaim(
762       ccoId,
```

```
763         _rolloverId
764       );
765       srRound.deposited = srRound.invested;
766       jrRound.deposited = jrRound.invested;
767       srRound.newInvested = srRound.invested;
768       jrRound.newInvested = jrRound.invested;
769       srRound.shares = srRound.invested;
770       jrRound.shares = jrRound.invested;
771       rollover_.thisRound = 1;
772     }
```

Listing 3.16: RolloverCCO:: _firstInvest ()

After that, a user may call `claim()`, which cascadingly calls `updateInvestorDistribute()` to claim the excess amount (line 541). Moreover, if the user calls `deposit()` one more time, the `if` condition (lines 461 – 463) is satisfied to execute the `then`-branch, which includes the `_updateInvestor()` execution. This execution allows the current depositing user to claim the excess amount a second time.

```
492     function claim(uint256 _rolloverId, OLib.Tranche _tranche)
493       external
494       noPanic
495       notDead(_rolloverId)
496       nonReentrant
497     {
498       Rollover storage rollover_ = rollovers[_rolloverId];
499       if (
500         rollover_.investorLastUpdates[_tranche][msg.sender] !=
501         rollover_.thisRound + 1
502       ) {
503         _updateInvestorDistribute(msg.sender, _rolloverId, _tranche);
504       }
505     }
```

Listing 3.17: RolloverCCO::claim()

```
437     function deposit(
438       uint256 _rolloverId,
439       OLib.Tranche _tranche,
440       uint256 _amount
441     ) external noPanic notDead(_rolloverId) nonReentrant {
442       Rollover storage rollover_ = rollovers[_rolloverId];
443       if (rollover_.investorLastUpdates[_tranche][msg.sender] == 0) {
444         rollover_.investorLastUpdates[_tranche][msg.sender] = 1;
445       }
446       {
447         Round storage round_ = rollover_.rounds[rollover_.thisRound + 1];
448         uint256 ccoId = round_.ccoId;
449         require(ccoId != 0, "No CCO to deposit in yet");
450         require(ccoManager.canDeposit(ccoId), "CCO not in deposit state");
451         TrancheRound storage trancheRound_ = round_.tranches[_tranche];
```

```
      OLib.Investor storage investor_ = trancheRound_.investors[msg.sender];
      uint256 total = trancheRound_.newDeposited += _amount;
      uint256 userSum =
        investor_.userSums.length > 0
          ? investor_.userSums[investor_.userSums.length - 1] + _amount
          : _amount;
      investor_.prefixSums.push(total);
      investor_.userSums.push(userSum);
    }
    if (
      rollover_.investorLastUpdates[_tranche][msg.sender] <
      rollover_.thisRound + 1
    ) {
      (uint256 shares, uint256 excess) =
        _updateInvestor(msg.sender, _rolloverId, _tranche);
      rollover_.investorLastUpdates[_tranche][msg.sender] =
        rollover_.thisRound +
        1;
      if (excess > _amount) {
        rollover_.assets[_tranche].safeTransfer(msg.sender, excess - _amount);
      } else {
        _amount -= excess;
      }
      if (shares > 0) {
        rollover_.rolloverTokens[_tranche].mint(msg.sender, shares);
      }
    }
    rollover_.assets[_tranche].safeTransferFrom(
      msg.sender,
      address(this),
      _amount
    );
    emit Deposited(msg.sender, _rolloverId, _amount);
  }
```

Listing 3.18: RolloverCCO::deposit()

Recommendation Revise the `deposit()` to avoid repeated claims of any excess amount after investment.

Status This issue has been resolved.

4 Conclusion

In this audit, we have analyzed the Ondo design and implementation. The system presents a unique, robust offering as a decentralized protocol to allow investors to shift the risk and reward balance between each other. The current code base is well structured and neatly organized. 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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

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

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