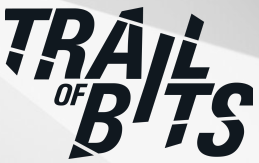

Ondo Finance: Ondo Protocol

Security Assessment

November 18, 2022

Prepared for:
Ondo Team
Ondo Finance

Prepared by: **Anish Naik, Justin Jacob, and Damilola Edwards**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Executive Summary

Engagement Overview

Ondo Finance engaged Trail of Bits to review the security of the Ondo protocol. From October 11 to October 24, 2022, a team of two consultants conducted a security review of the client-provided source code, with four person-weeks of effort. Details of the project's timeline, test targets, and coverage are provided in subsequent sections of this report.

Project Scope

Our testing efforts were focused on the identification of flaws that could result in a compromise of confidentiality, integrity, or availability of the target system. We conducted this audit with full knowledge of the system, including access to the source code and documentation. We performed static and dynamic testing of the target system and its codebase, using both automated and manual processes.

Summary of Findings

The audit uncovered significant flaws that could impact system confidentiality, integrity, or availability. A summary of the findings and details on notable findings are provided below.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	0
Low	2
Informational	4
Undetermined	0

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	5
Denial of Service	1
Timing	1
Undefined Behavior	1

Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

- **TOB-ONDO-1**
  An attacker could grief the protocol by exhausting the minting and redeeming rate limits in a single transaction.

- **TOB-ONDO-2**
  Investing different collateral tokens in a single strategy contract could result in incorrect accounting and the loss of user funds.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineers were associated with this project:

Anish Naik, Consultant
anish.naik@trailofbits.com

Justin Jacob, Consultant
justin.jacob@trailofbits.com

Damilola Edwards, Consultant
damilola.edwards@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
October 5, 2022	Pre-project kickoff call
October 18, 2022	Status update meeting #1
October 25, 2022	Delivery of report draft and report readout meeting
November 18, 2022	Delivery of final report and fix review

Project Goals

The engagement was scoped to provide a security assessment of the Ondo protocol. Specifically, we sought to answer the following non-exhaustive list of questions:

- Could an attacker steal funds from the system?
- Are appropriate access controls in place?
- Are there any flaws in the upgrade and migration mechanisms?
- Are the arithmetic calculations performed during token minting and redeeming operations correct?
- Is the protocol vulnerable to denial-of-service (DoS) attacks?
- Is the arithmetic for handling various types of collateral performed correctly?
- Are user-provided parameters sufficiently validated?
- Are there any economic attack vectors in the system?
- Does the protocol convert tokens to and from shares correctly?
- Is the share price prone to manipulation?
- Could the use of low-level calls in the codebase cause any problems?
- Could a user's funds become stuck in the system?

Project Targets

The engagement involved a review and testing of the following target.

Ondo Protocol

Repository	https://github.com/ondoprotocol/monopoly
Version	`814cfcfa04a7bfa4ae3fa395cafa329767dc67ec`
Type	Solidity
Platforms	Ethereum and Polygon

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches and their results include the following:

`PSM` and `PolyMinter`: The `PSM` (Peg Stability Module) contract is responsible for minting and redeeming MONO tokens against a provided collateral type. Similarly, the `PolyMinter` contract is responsible for minting and redeeming POLY tokens against a provided collateral type. We conducted the following manual and automated reviews of these contracts:

- We reviewed the conversions of token decimals to ensure that they are performed correctly.
- We reviewed the rate-limiting functionality to ensure that the minting and redeeming rate limits are enforced correctly and that the arithmetic is sound.
- We reviewed the use of access control modifiers to ensure that necessary access controls are in place for privileged operations.
- We performed a manual review of the two contracts to look for logical flaws and DoS vectors that could be exploited by external users. Through this review, we found that the rate limits imposed on MONO deposit and withdrawal operations in the `PSM` contract could allow malicious users to launch DoS attacks (TOB-ONDO-1). The `PolyMinter` contract is vulnerable to the same issue.
- We reviewed the bounds on all owner-controlled system parameters and found that minting and redeeming fees lack upper bounds (TOB-ONDO-8).
- We reviewed the `PSM` contract's issuance and redemption mechanism and found that it introduces an arbitrage opportunity (TOB-ONDO-6). The `PolyMinter` contract is vulnerable to the same issue.

`Zapper`: The `Zapper` helper contract streamlines the process of converting USDC to stMONO by condensing user workflows that would otherwise require multiple transactions into a single function call. We conducted a manual review of the interactions between the `Zapper`, `PSM`, and `Rewarder` contracts to ensure that they are done correctly.

`Treasury`: The `Treasury` contract serves as the accounting engine for MONO and POLY. For MONO, the treasury is responsible for minting and burning MONO tokens, enforcing global minting limits, minting protocol-owned MONO tokens, paying out yields to the rewarder, and taking fees. For POLY, the treasury is responsible for minting and burning POLY tokens, setting the price of POLY, and maintaining the equity buffer between a set floor and ceiling. The treasury also handles investments of its collateral by transferring and

withdrawing collateral to whitelisted strategies and accounting for profit-and-loss (PnL). We conducted the following manual and automated reviews of this contract:

- We reviewed each function in the contract to ensure that the necessary access controls are in place.
- We reviewed the depositing and redeeming flow for both POLY/MONO and underlying collateral tokens.
- We checked whether an attacker could bypass any of the access controls on `Treasury` functions to steal funds from the contract.
- We reviewed the arithmetic operations and state changes that occur within the `Treasury` contract during the investment of assets into strategies. In conducting this review, we found that investing multiple tokens with different decimals in the same strategy will result in incorrect PnL reporting, which may result in the loss of user or protocol funds (TOB-ONDO-2).
- We reviewed the migration logic to ensure that external users cannot steal funds, the requisite tokens are transferred, and all the necessary state changes are performed.
- We tested the solvency of the POLY and MONO tokens by ensuring that the treasury and associated strategies have the funds necessary to back each token.

`TimeBasedRateLimiter`: The `TimeBasedRateLimiter` abstract contract implements two rate limiters: one for minting and one for redeeming. This is used by other contracts such as `PSM` and `PolyMinter`. Each limit is associated with a duration, after which the tracked amount is reset. We conducted a manual review and automated review of this contract to ensure that all of its functions work correctly and are in alignment with the documentation.

`Rewarder`: The `Rewarder` contract provides users with a mechanism to stake their MONO tokens to earn yield. We conducted the following manual and automated reviews of this contract:

- We reviewed the interest accrual process to determine whether it is vulnerable to front-running or sandwich attacks.
- We reviewed the arithmetic that is performed and the state changes that occur during deposits, cooldown initiations, and redemptions to identify any edge cases that may result in undefined behavior.
- We reviewed the contract for flaws that would allow users to manipulate share prices.

- We tested the contract to ensure that the share price of a staked MONO token (stMONO/MONO) is monotonically increasing.

- We reviewed the contract's interaction with the `OndoNFT` contract to verify that NFTs minted during staking cannot be redeemed for MONO tokens before the cooldown is complete and that the NFT `Cooldown` struct cannot be reused to redeem tokens more than once.

`registry/`: This folder holds the `Registry`, `RegistryClient`, and `Roles` contracts, which serve as the foundations for the system's role-based access controls (RBACs). Inheritors of these contracts have access to RBAC modifiers designed to restrict the accessibility of certain functions; they also have access to the global references for the MONO and POLY tokens and to the Office of Foreign Assets Control's (OFAC) sanctions list. We conducted a manual review of these contracts to check that the `Registry` contract honors the role-based hierarchy defined in the `Roles` contract and to ensure that all of the contracts use the correct modifier from the `RegistryClient` contract based on specific use cases.

`Poly`, `Mono`, `StakedMono`, and `OndoNFT`: The `Poly`, `Mono`, and `StakedMono` contracts are all extensions of the OpenZeppelin `ERC20PresetMinterPauserUpgradeable` contract with no additional modifications. Similarly, the `OndoNFT` contract is an extension of the OpenZeppelin `ERC721PresetMinterPauserAutoIdUpgradeable` contract with no additional modifications. We manually reviewed these contracts to ensure that the `_disableInitializers` function call is used to lock the implementation contracts.

`strategies/`: This folder holds the `BaseStablecoinStrategy`, `CompoundStrategy`, and `MultisigStrategy` contracts. `BaseStablecoinStrategy` is an abstract contract that implements a template pattern for inheriting strategy contracts. The `CompoundStrategy` contract acts as a vessel for investing stablecoins into the Compound protocol, while the `MultisigStrategy` contract acts as a vessel for transferring stablecoins to wallets, such as multisignature wallets. We conducted the following manual reviews of these contracts:

- We reviewed the flow of funds between the strategies and the treasury for flaws that could result in interruptions. We found that the use of the `safeApprove` function could cause the treasury's redemption transactions from the strategies to revert (TOB-ONDO-7).

- We reviewed the use of the `beforeInvest` and `beforeRedeem` hooks to determine whether users could manipulate prices.

- We verified that the system's interactions with Compound's `CToken` contract are performed in order and correctly.

`factory/` and `TokenProxy`: The `factory/` folder contains the `MonoFactory`, `PolyFactory`, `StakerNFTFactory`, and `StakerTokenFactory` contracts, which serve as factories for the upgradeable `Mono`, `Poly`, `OndoNFT`, and `StakedMono` token contracts, respectively. We reviewed the factories to check that the contracts are initialized with the `TokenProxy` contract (which is a wrapper for the OpenZeppelin `TransparentUpgradeableProxy` contract), that the administrator of the proxy contracts is set to a `ProxyAdmin` contract rather than the guardian, and that the functions have the correct access controls in place.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. The following list outlines the coverage limitations of the engagement and indicates system elements that may warrant further review:

- **`external/`:** This folder holds external dependencies such as OpenZeppelin contracts, Chainanalysis interfaces, and Compound interfaces. We did not review these contracts and interfaces, as they were considered out of scope for this audit.
- **Interest calculations:** The interest calculations for staked MONO holders were taken from MakerDAO's `jug` contract. Due to time limitations, we did not review these calculations.
- **External contract interactions:** The `CompoundStrategy` contract interacts with the Compound system for lending stablecoins in exchange for yield. Although we validated the use of the functions, we did not review the implementation of those functions (e.g., `cToken.mint` and `cToken.accrueInterest`).

Automated Testing

Trail of Bits has developed unique tools for testing smart contracts. In this assessment, we used Echidna, a smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation, to check various system states.

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations; for example, Echidna may not randomly generate an edge case that violates a property. We follow a consistent process to maximize the efficacy of testing security properties. When using Echidna, we generate 100,000 test cases per property.

Our automated testing and verification work focused on the following properties and elements:

Ondo protocol end-to-end global system checks. We verified that the following critical system properties are maintained to ensure correct protocol behavior:

Property	Tool	Result
The `Treasury` contract's `equityBuffer` is less than or equal to `equityCap`.	Echidna	**Passed**
The `Treasury` contract's `equityBuffer` is greater than or equal to `equityFloor`.	Echidna	**Passed**
The `Treasury` contract's `userMinted` is less than or equal to `maxMintLimit`.	Echidna	**Passed**
The `totalSupply` of MONO minus the supply of protocol-minted MONO is less than or equal to the USDC balance of the `Treasury` contract plus the USDC balance of the `MockStrategy` contract, adjusted for token decimals.	Echidna	**Passed**
The `totalSupply` of POLY is less than or equal to the USDC balance of the `Treasury` contract plus the USDC balance of the `MockStrategy` contract, adjusted for token decimals and	Echidna	**Passed**

the POLY/USDC exchange rate.		
The exchange rate of one staked MONO token for one MONO token (stMONO for MONO) is monotonically increasing.	Echidna	**Passed**

`PolyMinter` and `PSM`. We verified that the minting and redeeming limits are maintained for both the `PolyMinter` and `PSM` contracts.

Property	Tool	Result
The `PSM` contract's `currentMintAmount` is less than or equal to `mintLimit`.	Echidna	**Passed**
The `PSM` contract's `currentRedeemAmount` is less than or equal to `redeemLimit`.	Echidna	**Passed**
The `PolyMinter` contract's `currentMintAmount` is less than or equal to `mintLimit`.	Echidna	**Passed**
The `PolyMinter` contract's `currentRedeemAmount` is less than or equal to `redeemLimit`.	Echidna	**Passed**

`Rewarder`. We verified that a staking pool's balance increases when a user stakes MONO, that the requisite NFT representing the user's staked MONO is created when the cooldown period is initiated, and that the user can redeem the NFT only if the user owns the NFT and the cooldown period is complete.

Property	Tool	Script
During deposit operations, `staker.balance` increases by the `amount` deposited and the interest accrued on the `staker.balance`.	Echidna	**Passed**

The `staker.balance` of the liquid staking pool increases when the cooldown period is initiated.	Echidna	**Passed**
An `ondoNFT` for the requesting user is created when the cooldown period is initiated.	Echidna	**Passed**
An `ondoNFT` can be successfully redeemed only if the cooldown has completed.	Echidna	**Passed**
A withdrawal of `amount` MONO from the liquid staking pool decreases the pool's `staker.balance` by `amount` minus the accrued interest.	Echidna	**Passed**

`Treasury.` We verified that the internal bookkeeping records are updated when strategy investments and withdrawals are made by the treasury.

Property	Tool	Result
Treasury investments for `amount` increase `investedAmount` by `amount`, scaled by the necessary token decimals.	Echidna	**Passed**
Treasury divestitures for `amount` decrease `investedAmount` *approximately* by `amount`, scaled by the necessary token decimals.	Echidna	**Further Investigation Required***

*Due to time constraints, we were unable to fully test this property. We recommend that the Ondo Finance team test this property to ensure that it is validated.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The protocol uses Solidity 0.8.16 for arithmetic operations, and most of the operations are documented with inline documentation. The Ondo Finance team has performed extensive unit testing of the arithmetic operations and has integrated some fuzz testing for additional validation of the calculations. We recommend that the team continue maturing its fuzzing test suite as the protocol develops and document all mathematical operations.	**Satisfactory**
Auditing	All functions involved in critical state-changing operations emit events. Additionally, the Ondo Finance team provided documentation on the deployment, post-deployment, and off-chain monitoring solution. However, it is unclear whether the Ondo Finance team has developed an incident response plan. Recommendations on creating an incident response plan are provided in appendix E.	**Satisfactory**
Authentication / Access Controls	The system uses hierarchical RBACs. Of the 20 total roles in the system, some are assigned to various contracts, and the rest are assigned to various multisignature wallets. The roles are used to gate and partition privileged actions across a large number of actors. However, this level of complexity could cause problems as development continues. We recommend that the Ondo Finance team reevaluate the protocol's roles to identify any that could be consolidated or removed to simplify the hierarchy. Finally, the team should document each role's privileges and keep the documentation	**Satisfactory**

	updated as development continues.	
Complexity Management	Each contract in the protocol has a clear purpose, and there are no signs of excessive inheritance or high cyclomatic complexity. All functions are concise, are well documented, have a clear purpose, and are appropriately tested.	**Satisfactory**
Decentralization	The efficacy of the system relies on privileged actors to manually perform necessary operations; for example, operations related to PnL distribution, user withdrawals, updates to state variables that directly impact users' solvency and funds, investments into and divestments from strategies, transfers of rewards to users for staking MONO, and transfers of rewards to POLY users all require manual intervention by privileged actors. Additionally, users' core yield-earning strategy is to transfer funds to a team-owned multisignature address that will perform *off-chain* investments; therefore, these investments are not traceable on-chain. Effectively, the system is highly dependent on guardian intervention and trust in external actors. Without manual intervention, the protocol is not usable. We recommend that the Ondo Finance team redesign centralized aspects of the protocol to reduce the element of trust, and automate the necessary functions that determine the security of a user's investment.	**Weak**
Documentation	The code contains a large amount of inline documentation that clearly explains the respective functionality of each function. Additionally, we received a deployment checklist from the Ondo Finance team that clearly outlines the necessary steps during contract deployment. However, users could benefit from additional user-facing documentation and diagrams highlighting the end-to-end use of the system.	**Moderate**
Front-Running Resistance	The system is resistant to the front-running of contract initializations and of PnL reports. Note that front-running PnL reports will become a more viable exploit if PnL reporting becomes automated. We recommend that the	**Further Investigation Required**

	Ondo Finance team develop user-facing documentation on the arbitrage opportunity that exists because of MONO's peg and any other arbitrage opportunities that may exist within the system (see TOB-ONDO-6). Additionally, the team should re-investigate the protocol's front-running resistance as the codebase matures.	
Low-Level Manipulation	The protocol relies on low-level assembly for interest calculations. As explained in the Project Coverage section, this logic was taken from MakerDAO and was not reviewed due to time constraints. Given the criticality of the logic, we recommend that the Ondo Finance team perform additional isolated testing of the logic and document its use thoroughly.	**Further Investigation Required**
Testing and Verification	The system has an extensive unit test suite in addition to a few fuzz tests for more complex scenarios. However, the discovery of TOB-ONDO-2 indicates that additional unit testing would be beneficial. Additionally, the supplementary Echidna end-to-end testing code provided by Trail of Bits should be implemented in Foundry (or used "as-is") and be used to validate critical system properties.	**Moderate**

Summary of Recommendations

The Ondo protocol is a work in progress with multiple planned iterations. Trail of Bits recommends that Ondo Finance address the findings detailed in this report and take the following additional steps prior to deployment:

- **Reduce the centralization of the system by taking the following steps**:
    - **Automate the PnL reporting for all strategies** by implementing either periodic “harvests” or automated updates to PnL reports on each user interaction.
    - **Automate the distribution of PnL to the rewarder and POLY token holders.** Develop arithmetic properties that will determine when funds should be transferred to stakers and POLY token holders.
    - **Reduce the number of system parameters that can be manipulated by privileged actors.**
    - **Guarantee that user withdrawals do not require manual intervention.** Currently, user withdrawals may require manual divestitures and token transfers.
- **Review the codebase for any front-running attack vectors that may result from implementing the automations and architectural updates suggested above.** Specifically, automating PnL reporting could introduce opportunities to front-run both profits and losses.
- **Implement limits on the number of assets that can be transferred to the multisignature strategy** for off-chain investments to ensure that there is always sufficient capital on-chain to meet on-demand withdrawals.
- **Provide transparency on the system’s reliance on off-chain investments, or remove this reliance,** to reduce the trust needed by users to participate in the system.
- **Identify all system properties that are expected to hold and use dynamic end-to-end fuzz testing to validate those system properties.**
- **Document the risks associated with DoS attacks and improve the user-facing documentation regarding the use of the protocol.**

- **Simplify the RBAC hierarchy** to reduce the attack surface and mitigate any future risks. Additionally, review the best practices for using multisignature wallets provided in appendix F.
- **Develop a detailed incident response plan** to ensure that any issues that arise can be addressed promptly and without confusion. (See appendix E for related recommendations.)
- **Use automated dependency monitoring** (e.g., Dependabot) to identify updates or bug fixes for external dependencies.
- **Perform another audit** after completing the above steps.

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Risk of DoS attacks due to rate limits	Denial of Service	High
2	Risk of accounting errors due to missing check in the invest function	Data Validation	High
3	Missing functionality in the _rescueTokens function	Data Validation	Low
4	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
5	Lack of contract existence check on call	Data Validation	Informational
6	Arbitrage opportunity in the PSM contract	Timing	Informational
7	Problematic use of safeApprove	Data Validation	Low
8	Lack of upper bound for fees and system parameters	Data Validation	Informational

Detailed Findings

1. Risk of DoS attacks due to rate limits	
Severity: **High**	Difficulty: **Medium**
Type: Denial of Service	Finding ID: TOB-ONDO-1
Target: `contracts/PSM.sol`	

Description

Due to the rate limits imposed on MONO deposit and withdrawal operations, malicious users could launch denial-of-service (DoS) attacks.

The PSM contract extends the `TimeBasedRateLimiter` contract, which defines the maximum number of MONO tokens that can be minted and redeemed within a preset duration window. This allows the protocol to manage the inflow and outflow of assets (figures 1.1 and 1.2).

```
 93    function _checkAndUpdateMintLimit(uint256 amount) internal {
 94     require(amount > 0, "RateLimit: mint amount can't be zero");
 95
 96     if (block.timestamp >= lastResetMintTime + resetMintDuration) {
 97       // time has passed, reset
 98       currentMintAmount = 0;
 99       lastResetMintTime = block.timestamp;
100     }
101     require(
102       amount <= mintLimit - currentMintAmount,
103       "RateLimit: Mint exceeds rate limit"
104     );
105
106     currentMintAmount += amount;
107    }
```

Figure 1.1: The `_checkAndUpdateMintLimit` function in TimeBasedRateLimiter.sol#L93–107

```
117    function _checkAndUpdateRedeemLimit(uint256 amount) internal {
118     require(amount > 0, "RateLimit: redeem amount can't be zero");
119
120     if (block.timestamp >= lastResetRedeemTime + resetRedeemDuration) {
121       // time has passed, reset
122       currentRedeemAmount = 0;
123       lastResetRedeemTime = block.timestamp;
```

```
124     }
125     require(
126       amount <= redeemLimit - currentRedeemAmount,
127       "RateLimit: Redeem exceeds rate limit"
128     );
129     currentRedeemAmount += amount;
130   }
```

Figure 1.2: The `_checkAndUpdateRedeemLimit` function in TimeBasedRateLimiter.sol#L117–130

However, a dedicated adversary could deposit enough collateral assets to reach the minting limit and immediately withdraw enough MONO to reach the redeeming limit in the same transaction; this would exhaust the limits of a given duration window, preventing all subsequent users from entering and exiting the system. Additionally, if the duration is long (e.g., a few hours or a day), such attacks would cause the PSM contract to become unusable for extended periods of time. It is important to note that adversaries conducting such attacks would incur the fee imposed on redemption operations, making the attack less appealing.

Exploit Scenario

Alice, the guardian, sets `resetMintDuration` and `resetRedeemDuration` to one day. As soon as the minting and redeeming limits reset, Eve deposits a large amount of USDC for MONO and immediately burns the MONO for USDC. This transaction maxes out the minting and redeeming limits, preventing users from performing deposits and withdrawals for an entire day.

Recommendations

Short term, set the `resetMintDuration` and `resetRedeemDuration` variables to reasonably small values to mitigate the impact of DoS attacks. Note that setting them to zero would defeat the purpose of rate limiting because the limits will be reset on every minting or redeeming operation.

Long term, implement off-chain monitoring to detect unusual user interactions, and develop an incident response plan to ensure that any issues that arise can be addressed promptly and without confusion.

2. Risk of accounting errors due to missing check in the invest function

Severity: **High**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-ONDO-2
Target: `contracts/Treasury.sol`	

Description

Because of a missing check in the `invest` function, investing multiple tokens with different decimals in the same strategy will result in incorrect profit-and-loss (PnL) reporting, which could result in the loss of user or protocol funds.

The `invest` function is responsible for transferring funds from the treasury to a strategy and for updating the strategy's investment balance (i.e., `strategyInvestedAmount`). However, the `invest` function accepts any token in the `collateral` array alongside the token amounts to be transferred. Therefore, if multiple tokens with different decimals are used to invest in the same strategy, the treasury's investment records would not accurately reflect the true balance of the strategy, resulting in accounting errors within the protocol.

```
694    function invest(
695     address strategy,
696     uint256 collateralAmount,
697     uint256 collateralIndex
698    ) external whenNotPaused whenTreasuryActive onlyFundManager {
699     IERC20 collateralToken = collateral[collateralIndex].collateralToken;
700     require(
701       address(collateralToken) != address(0),
702       "Treasury: Cannot used a removed collateral token"
703     );
704     // Require that the strategy address is approved
705     require(
706       hasRole(strategy, Roles.STRATEGY_CONTRACT),
707       "Treasury: Must send funds to approved strategy contract"
708     );
709
710     // Scale up invested amount
711     investedAmount += _scaleUp(collateralAmount, collateralIndex);
712
713     // Account for investment in strategyInvestedAmounts
714     strategyInvestedAmounts[strategy] += collateralAmount;
715
716     // Transfer collateral to strategy
717     collateralToken.safeTransfer(strategy, collateralAmount);
```

Figure 2.1: The `invest` function in `Treasury.sol#L694–719`

Exploit Scenario

The `CompoundStrategy` contract is supposed to accept only USDC as its investment token. However, the fund manager, Bob, triggers investments into the Compound strategy using FRAX tokens instead. Due to insufficient validation of the `collateralToken` value, the transaction succeeds, causing a mismatch between the treasury's account and the total value of assets in the strategy.

Recommendations

Short term, implement a check within the `invest` function to ensure that the `collateralToken` address of the supplied `collateralIndex` value matches the token accepted by the `strategy`.

Long term, carefully review token usage across all contracts to ensure that each token's decimal places are taken into consideration.

3. Missing functionality in the _rescueTokens function

Severity: **Low**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-ONDO-3
Target: `contracts/RegistryClient.sol`	

Description

The `RegistryClient` contract is a helper contract designed to aid in the protocol's role-based access control (RBAC) mechanism. It has various helper functions that serve as safety mechanisms to rescue funds trapped inside a contract. The inline documentation for the `_rescueTokens` function states that if the `_amounts` array contains a zero-value entry, then the entire token's balance should be transferred to the caller. However, this functionality is not present in the code; instead, the function sends zero tokens to the caller on a zero-value entry.

```
 * @dev If the _amount[i] is 0, then transfer all the tokens
 *
 * @param _tokens List of tokens
 * @param _amounts Amount of each token to send
 */
function _rescueTokens(address[] calldata _tokens, uint256[] memory _amounts)
  internal
  virtual
{
  for (uint256 i = 0; i < _tokens.length; i++) {
    uint256 amount = _amounts[i];
    IERC20(_tokens[i]).safeTransfer(msg.sender, amount);
  }
}
```

Figure 3.1: The `_rescueTokens` function in `RegistryClient.sol:#L192–L205`

Exploit Scenario

A user accidentally sends ERC20 tokens to the PSM contract. To rescue the tokens, Bob, the guardian of the Ondo protocol contracts, calls `_rescueTokens` and specifies zero as the amount for the ERC20 tokens in question. However, the contract incorrectly transfers zero tokens to him and, therefore, the funds are not rescued.

Recommendations

Short term, adjust the `_rescueTokens` function so that it transfers the entire token balance when it reaches a zero-value entry in the `_amounts` array.

Long term, improve the unit test coverage to cover additional edge cases and to ensure that the system behaves as expected.

4. Solidity compiler optimizations can be problematic	
Severity: **Informational**	Difficulty: **High**
Type: Undefined Behavior	Finding ID: TOB-ONDO-4
Target: Ondo protocol	

Description
The Ondo protocol contracts have enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised.

Security issues due to optimization bugs have occurred in the past. A medium- to high-severity bug in the Yul optimizer was introduced in Solidity version 0.8.13 and was fixed only recently, in Solidity version 0.8.17. Another medium-severity optimization bug—one that caused memory writes in inline assembly blocks to be removed under certain conditions— was patched in Solidity 0.8.15.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario
A latent or future bug in Solidity compiler optimizations causes a security vulnerability in the Ondo protocol contracts.

Recommendations
Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

5. Lack of contract existence check on call

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ONDO-5
Target: `contracts/factory/MonoFactory.sol`, `contracts/factory/PolyFactory.sol`, `contracts/RegistryClient.sol`	

Description

The factories and the registry client all have a `multiexcall` function, which is designed to create batched calls to various target addresses. To do this, it uses the `call` opcode to execute arbitrary calldata. If the target address is set to an incorrect address, the address of an externally owned account (EOA), or the address of a contract that is subsequently destroyed, a `call` to the target will still return `true`. However, the `multiexcall` functions in the factories and the registry client do not include contract existence checks to account for this behavior.

```
 function multiexcall(ExCallData[] calldata exCallData)
  external
  payable
  override
  onlyGuardian
  returns (bytes[] memory results)
{
  results = new bytes[](exCallData.length);
  for (uint256 i = 0; i < exCallData.length; ++i) {
    (bool success, bytes memory ret) =
      address(exCallData[i].target).call{value: exCallData[i].value}(
        exCallData[i].data
      );
    require(success, "Call Failed");
    results[i] = ret;
  }
}
```

Figure 5.1: The `multiexcall` function in MonoFactory.sol:#L120–L136

The Solidity documentation includes the following warning:

```
The low-level functions call, delegatecall and staticcall return true as their first
return value if the account called is non-existent, as part of the design of the
EVM. Account existence must be checked prior to calling if needed.
```

Figure 5.2: A snippet of the Solidity documentation detailing unexpected behavior related to `call`

Exploit Scenario

Bob, the guardian of the Ondo protocol contracts, uses the `multiexcall` function but accidentally passes in an incorrect address that points to an EOA. As a result, the call succeeds without executing any code.

Recommendations

Short term, implement a contract existence check before each `call`. Document the fact that calling an EOA or self-destructed contract will still return `true`.

Long term, carefully review the Solidity documentation, especially the "Warnings" section, and the pitfalls of using low-level patterns.

6. Arbitrage opportunity in the PSM contract	
Severity: **Informational**	Difficulty: **High**
Type: Timing	Finding ID: TOB-ONDO-6
Target: `contracts/PSM.sol`	

Description
Given two `PSM` contracts for two different stablecoins, users could take advantage of the difference in price between the two stablecoins to engage in arbitrage.

This arbitrage opportunity exists because each `PSM` contract, regardless of the underlying stablecoin, holds that 1 MONO is worth $1. Therefore, if 100 stablecoin tokens are deposited into a `PSM` contract, the contract would mint 100 MONO tokens regardless of the price of the collateral token backing MONO.

The `PolyMinter` contract is vulnerable to the same arbitrage opportunity.

Exploit Scenario
USDC is trading at $0.98, and USDT is trading at $1. A user deposits 10,000 USDC tokens into the `PSM` contract and receives 10,000 MONO, which is worth $10,000, assuming there are no minting fees. The user then burns the 10,000 MONO he received from depositing the USDC, and he receives $9,990 worth of USDT in exchange, assuming there is a 0.1% redemption fee. Therefore, the arbitrageur was able to make a risk-free profit of $190.

Recommendations
Short term, document all front-running and arbitrage opportunities throughout the protocol, and ensure that users are aware of them. As development continues, reassess the risks associated with these opportunities and the adverse effects they could have on the protocol.

Long term, use an off-chain monitoring solution to detect any anomalous price fluctuations for the supported collateral tokens. Additionally, develop an incident response plan to ensure that any issues that arise can be addressed promptly and without confusion (see appendix E).

7. Problematic use of safeApprove

Severity: **Low**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ONDO-7
Target: `contracts/BaseStablecoinStrategy.sol`	

Description

In order for users to earn yield on the collateral tokens they deposit, the `Treasury` contract sends the collateral tokens to a yield-bearing strategy contract that inherits from the `BaseStablecoinStrategy` contract. When a privileged actor calls the `redeem` function, the function approves the `Treasury` contract to pull the necessary funds from the strategy.

However, the function approves the `Treasury` contract by calling the `safeApprove` function.

```
  [...]
  _redeem(amount);

  stablecoin.safeApprove(getCurrentTreasury(), amount);
  emit Redeem(address(stablecoin), amount, currentPosition, profit, loss);
}
```

Figure 7.1: A snippet of the `redeem` function in `BaseStablecoinStrategy.sol:#L106–L110`

As explained in the OpenZeppelin documentation, `safeApprove` should be called only if the currently approved amount is zero—when setting an initial allowance or when resetting the allowance to zero. Therefore, if the entire approved amount is not pulled by calling `Treasury.withdraw`, subsequent redemption operations will revert.

Additionally, OpenZeppelin's documentation indicates that the `safeApprove` function is officially deprecated.

```
/**
 * @dev Deprecated. This function has issues similar to the ones found in
 * {IERC20-approve}, and its usage is discouraged.
 *
 * Whenever possible, use {safeIncreaseAllowance} and
 * {safeDecreaseAllowance} instead.
 */
function safeApprove(
    IERC20 token,
```

```
    address spender,
    uint256 value
) internal {
    // safeApprove should only be called when setting an initial allowance,
    // or when resetting it to zero. To increase and decrease it, use
    // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
    require(
        (value == 0) || (token.allowance(address(this), spender) == 0),
        "SafeERC20: approve from non-zero to non-zero allowance"
    );
    _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
spender, value));
}
```

Figure 7.2: The `safeApprove` function in SafeERC20.sol#L39–L59

Exploit Scenario

Bob, the fund manager of the Ondo protocol, deposits 200 tokens into the `CompoundStrategy` contract and calls `redeem` to redeem 100 of them. However, he calls `Treasury.withdraw` for only 50 tokens. Because the allowance allocated for `safeApprove` has not been completely used, he is blocked from performing subsequent redemption operations for the remaining 100 tokens.

Recommendations

Short term, use `safeIncreaseAllowance` and `safeDecreaseAllowance` instead of `safeApprove`. Alternatively, document the fact that the `Treasury` should use up the entire approval amount before the privileged actor can call a strategy's `redeem` function.

Long term, expand the dynamic fuzz testing suite to identify edge cases like this.

8. Lack of upper bound for fees and system parameters

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-ONDO-8
Target: `contracts/PSM.sol`	

Description

The PSM contract's `setMintFee` and `setRedeemFee` functions, used by privileged actors to set optional minting and redeeming fees, do not have an upper bound on the fee amount that can be set; therefore, a privileged actor could set minting and redeeming fees to any value. Excessively high fees resulting from typos would likely not be noticed until they cause disruptions in the system.

```
289    * @notice Sets PSM's mint fee
290    *
291    * @param _mintFee new mint fee specified in basis points
292    */
293    function setMintFee(uint256 _mintFee) external onlyMono {
294     mintFee = _mintFee;
295     emit MintFeeSet(_mintFee);
296    }
297
298    /**
299    * @notice Sets PSM's redeem fee.
300    *
301    * @param _redeemFee new redem fee specified in basis points
302    */
303    function setRedeemFee(uint256 _redeemFee) external onlyMono {
304     redeemFee = _redeemFee;
305     emit RedeemFeeSet(_redeemFee);
306    }
```

Figure 8.1: The `setMintFee` and `setRedeemFee` functions in PSM.sol#L289–306

Additionally, a large number of system parameters throughout the `Rewarder`, `Treasury`, PSM, and `PolyMinter` contracts are unbounded.

Recommendations

Short term, set upper limits for the minting and redeeming fees and for all the system parameters that do not currently have upper bounds. The upper bounds for the fees should be high enough to encompass any reasonable value but low enough to catch mistakenly or maliciously entered values that would result in unsustainably high fees.

Long term, carefully document the owner-specified values that dictate the financial properties of the contracts and ensure that they are properly constrained.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Front-Running Resistance	The system's resistance to front-running attacks
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, they enhance code readability and may prevent the introduction of vulnerabilities in the future.

Rewarder

- **Modifier side effects:** Modifiers should only implement checks, not make state changes or external calls. Having modifiers make such changes or calls violates the checks-effects-interactions pattern. These side effects may go unnoticed by developers.

```
154   modifier accrueAllInterest() {
155    accrueInterestOnAllStakers();
156    _;
157   }
158
159   /**
160   * @notice Update the balance of every staker to reflect interest accrued
161   */
162   function accrueInterestOnAllStakers() public {
163    uint256 timeElapsed = block.timestamp - lastInterestAccrualTimestamp;
164    uint256 size = stakers.length;
165    for (uint256 i = 0; i < size; ++i) {
166      stakers[i].balance = calculateInterest(
167        stakers[i].balance,
168        stakers[i].perSecondYield,
169        timeElapsed
170      );
171    }
172    lastInterestAccrualTimestamp = block.timestamp;
173   }
```

Figure C.1: The `accrueAllInterest` modifier and the `accrueInterestOnAllStakers` function in `Rewarder.sol#L154–173`

D. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified, and its associated risks, understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the below output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.
- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.
- ❏ **They have a security mailing list for critical announcements.** Their team should advise users (like you!) when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.
- ❏ **The contract uses `SafeMath`.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath` usage.
- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.
- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can abuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.
- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.
- ❏ **The owner cannot blacklist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a blacklist. Identify blacklisting features by hand.
- ❏ **The team behind the token is known and can be held responsible for abuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC20 Tokens

ERC20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a boolean.** Several tokens do not return a boolean on these functions. As a result, their calls in the contract might fail.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC20 standard and may not be present.
- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is below 255.
- ❏ **The token mitigates the known ERC20 race condition.** The ERC20 standard has a known ERC20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.
- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.
- ❏ **Potential interest earned from the token is taken into account.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not taken into account.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.
- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.
- ❏ **The tokens are located in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.
- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.
- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC721 Tokens

ERC721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC721 standard requires that such transfers revert.
- ❏ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of those contracts can result in a loss of assets.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC721 standard and may not be present.
- ❏ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❏ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❏ **The `ownerOf` function reverts if the `tokenId` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❏ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❏ **The token ID of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of the ERC721 Standard

To mitigate the risks associated with ERC721 contracts, conduct a manual review of the following conditions:

- ❏ **The `onERC721Received` callback is taken into account.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).

- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave similarly to `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- ❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.

E. Incident Response Plan Recommendations

This section provides recommendations on formulating an incident response plan.

- **Identify the parties (either specific people or roles) responsible for implementing the mitigations when an issue occurs (e.g., deploying smart contracts, pausing contracts, upgrading the front end, etc.).**
- **Document internal processes for addressing situations in which a deployed remedy does not work or introduces a new bug.**
    - Consider documenting a plan of action for handling failed remediations.
- **Clearly describe the intended contract deployment process.**
- **Outline the circumstances under which Ondo will compensate users affected by an issue (if any).**
    - Issues that warrant compensation could include an individual or aggregate loss or a loss resulting from user error, a contract flaw, or a third-party contract flaw.
- **Document how the team plans to stay up to date on new issues that could affect the system; awareness of such issues will inform future development work and help the team secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.**
    - Identify sources of vulnerability news for each language and component used in the system, and subscribe to updates from each source. Consider creating a private Discord channel in which a bot will post the latest vulnerability news; this will provide the team with a way to track all updates in one place. Lastly, consider assigning certain team members to track news about vulnerabilities in specific components of the system.
- **Determine when the team will seek assistance from external parties (e.g., auditors, affected users, other protocol developers, etc.) and how it will onboard them.**
    - Effective remediation of certain issues may require collaboration with external parties.
- **Define contract behavior that would be considered abnormal by off-chain monitoring solutions.**

It is best practice to perform periodic dry runs of scenarios outlined in the incident response plan to find omissions and opportunities for improvement and to develop "muscle memory." Additionally, document the frequency with which the team should perform dry runs of various scenarios, and perform dry runs of more likely scenarios more regularly. Create a template to be filled out with descriptions of any necessary improvements after each dry run.

F. Security Best Practices for Using Multisignature Wallets

Consensus requirements for sensitive actions, such as spending the funds in a wallet, are meant to mitigate risks, such as the following:

- The overruling of others' judgment by any one person
- Failures caused by any one person's mistake
- Failures caused by the compromise of any one person's credentials

In a 2-of-3 multisignature Ethereum wallet, for example, the execution of a "spend" transaction requires the consensus of two individuals in possession of two of the wallet's three private keys. For this model to be useful, it must fulfill the following requirements:

1. The private keys must be stored or held separately, and access to each one must be limited to a different individual.
2. If the keys are physically held by third-party custodians (e.g., a bank), multiple keys should not be stored with the same custodian. (Doing so would violate requirement #1.)
3. The person asked to provide the second and final signature on a transaction (i.e., the co-signer) ought to refer to a pre-established policy specifying the conditions for approving the transaction by signing it with his or her key.
4. The co-signer also ought to verify that the half-signed transaction was generated willfully by the intended holder of the first signature's key.

Requirement #3 prevents the co-signer from becoming merely a "deputy" acting on behalf of the first signer (forfeiting the decision-making responsibility to the first signer and defeating the security model). If the co-signer can refuse to approve the transaction for any reason, the due-diligence conditions for approval may be unclear. That is why a policy for validating transactions is needed. A verification policy could include the following:

- A protocol for handling a request to co-sign a transaction (e.g., a half-signed transaction will be accepted only via an approved channel)
- A whitelist of specific Ethereum addresses that are allowed to be the payee of a transaction
- A limit on the amount of funds spent in a single transaction or in a single day

Requirement #4 mitigates the risks associated with a single stolen key. For example, say that an attacker somehow acquired the unlocked Ledger Nano S of one of the signatories. A voice call from the co-signer to the initiating signatory to confirm the transaction would reveal that the key had been stolen and that the transaction should not be co-signed. If the signatory were under an active threat of violence, he or she could use a "duress code" (a code word, a phrase, or another signal agreed upon in advance) to covertly alert the others that the transaction had not been initiated willfully, without alerting the attacker.