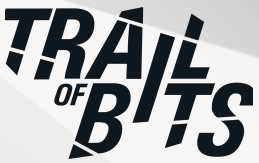

Seaport Protocol

Security Assessment

May 20, 2022

Prepared for:
0age
OpenSea

Prepared by: **Nat Chin, Troy Sargent, Bo Henderson, and Robert Schneider**

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 80+ 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

OpenSea engaged Trail of Bits to review the security of its Seaport system. From April 18 to May 12, 2022, a team of two consultants conducted a security review of the client-provided source code, with seven 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 a smart contract, a loss of funds, or unexpected behavior in the target system. We conducted this audit with full knowledge of the target 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 did not uncover significant flaws that could result in the compromise of a smart contract, a loss of funds, or unexpected behavior in the target system.

EXPOSURE ANALYSIS

Severity	*Count*
High	0
Medium	0
Low	2
Informational	7
Undetermined	2

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	5
Patching	1
Testing	1
Timing	2
Undefined Behavior	2

Notable Findings

An overview of several notable findings is provided below.

- **Error-prone approach to data validation (TOB-OSC-4, TOB-OSC-10)**
  The Seaport system relies on assert functions to validate the results of function calls to perform reentrancy checks. Instead of using modifiers to handle this validation, the contracts invoke these assert functions through a complex nested flow. This practice is error-prone, as it can result in the omission of data validation.

- **Unexpected behavior due to the use of risky Solidity components (TOB-OSC-8, TOB-OSC-3)**
  The contracts use Solidity compiler optimizations such as the new Yul optimization pipeline. These optimizations introduce risks, as bugs in those components could create exploitable issues in the Seaport codebase. They also make testing the code with different compiler versions and settings impossible.

Summary of Recommendations

The OpenSea Seaport contracts are a work in progress with multiple planned iterations. Trail of Bits recommends that OpenSea address the findings detailed in this report and take the following additional steps prior to deployment:

- Extend the Echidna tests to cover additional properties related to the creation of complex orders with conduits and zones, the fulfillment of orders, and the order-matching process. Most importantly, identify and check the system invariants, including those in appendix G.
- Write additional user documentation on the expected behavior of the system. The documentation should cover common errors users may encounter and explain what causes them and how to handle them.
- Create a flowchart diagram outlining the entire system architecture, including the off-chain and on-chain components. A flowchart can help optimize the user flow and clarify the interactions between different systems.
- Test the reference implementation against the optimized implementation. To do this, perform differential fuzzing of the values returned by the pure functions and verify that after each transaction, the contracts have the same state in both implementations.
- Document the security risks of the system. Ensure that exchanges that contain vulnerabilities are not added to the protocol as channels, especially if those exchanges make calls with user-provided addresses and data.

Project Summary

Contact Information

The following managers were associated with this project:

Dan Guido, Account Manager
dan@trailofbits.com

Cara Pearson, Project Manager
cara.pearson@trailofbits.com

The following engineers were associated with this project:

Nat Chin, Consultant
natalie.chin@trailofbits.com

Troy Sargent, Consultant
troy.sargent@trailofbits.com

Bo Henderson, Consultant
bo.henderson@trailofbits.com

Robert Schneider, Consultant
robert.schneider@trailofbits.com

Project Timeline

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

Date	Event
April 7, 2022	Pre-project kickoff call
April 25, 2022	Status update meeting #1
May 2, 2022	Status update meeting #2
May 9, 2022	Status update meeting #3
May 17, 2022	Delivery of report draft
May 17, 2022	Report readout meeting
May 20, 2022	Delivery of final report

Project Goals

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

- Could an attacker steal funds from the system?
- Are there appropriate access controls in place for user and admin operations?
- Could an attacker trap the system?
- Are there any denial-of-service attack vectors?
- Could users lose access to their funds?
- Does the system validate and limit fee amounts?
- Does the system validate function inputs correctly?
- What are the risks associated with token and contract callbacks?
- What are the risks associated with the use of conduits in orders and the use of channels?

Project Targets

The engagement involved a review and testing of the following target. We worked from the first commit in our manual review and from the second in our fuzz testing. The second commit added the `Conduit` and `ConduitController` contracts.

Seaport

Repository	https://github.com/ProjectOpenSea/seaport
Versions	`f17082fca3e99b409f53040d8858e84b0246aa22`, `00bd847df9971e6c1e61c7c4b58e6db6ce95a93f`
Type	Solidity
Platform	Ethereum

Trail of Bits reviewed the `Consideration` contract suite, which has since been renamed `Seaport`.

Project Coverage

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

`Consideration` and related contracts. The in-scope contracts include a main contract, `Consideration`, which is the entry point through which users validate and fulfill orders. This contract inherits from contracts including `ConsiderationInternalView`, `ConsiderationPure`, `ConsiderationInternal`, and `ConsiderationBase`, which all define helper functions to facilitate order matching. We performed a manual review of these contracts and their assembly code and used Echidna to test them.

`ConsiderationStructs` and `ConsiderationEnums`. These two contracts contain the structs and enums used to represent orders in the validation and fulfillment of orders. We performed a manual review of these contracts, which are also used to generate expected calldata for Echidna testing.

`ConsiderationConstants`. This contract defines the pointers, offsets, and memory layouts of the structs used extensively throughout the system. We performed a manual review of this contract.

During the audit, OpenSea discovered and patched a bug affecting the fulfillment of orders. Specifically, attempts to fulfill orders involving batched transfers of ERC1155 tokens with different IDs would revert. OpenSea removed the ERC1155 batched transfer functionality from the marketplace; the change was introduced in the second commit listed in the "Project Targets" section.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- The off-chain orderbook
- The user interface
- The external off-chain components that interact with the `Consideration` contracts

Automated Testing

Trail of Bits uses automated techniques to extensively test the security properties of software. We use both open-source static analysis and fuzzing utilities, along with tools developed in house, to perform automated testing of source code and compiled software.

Test Harness Configuration

We used the following tools in the automated testing phase of this project:

Tool	Description	Policy
Slither	A static analysis framework that can statically verify algebraic relationships between Solidity variables	Appendix E
Echidna	A smart contract fuzzer that can rapidly test security properties via malicious, coverage-guided test case generation	Appendix D

Test Results

The results of this focused testing are detailed below.

Single-line assembly equivalence. The codebase contains a significant amount of assembly. We used Echidna to check the equivalence of complex assembly operations.

Property	Tool	Result
The assembly code checks whether there is code at the provided `tokenAddress` and, if there is, whether the call to that address fails.	Echidna	**Passed**
The assembly code checks the result of each external call, verifying that the call either failed or did not return any data.	Echidna	**Passed**

`Consideration` contracts. These contracts allow offerers to validate orders and callers to fulfill and match orders. We used Echidna to check the assumptions made throughout these contracts.

ID	Property	Tool	Result
1	Once an order has been validated, `getOrderStatus` will return true for `isValidated`.	Echidna	**Passed**
2	Once an order has been validated, `getOrderStatus` will return false for `isCanceled` until it is canceled.	Echidna	**Passed**
3	With the correct preconditions and arguments, a call to `validate()` will always succeed.	Echidna	**Passed**
4	Once the entire order has been filled, the size of the filled order is equal to the order's size and is nonzero.	Echidna	**Passed**
5	With the correct preconditions, a call to `cancel` will always succeed; additionally, `getOrderStatus` will return true for `isCanceled`.	Echidna	**Passed**
6	With the correct preconditions, a call to `incrementNonce` will always succeed.	Echidna	**Passed**
7	With the correct preconditions, a call to `fulfillBasicOrder` will always succeed.	Echidna	**Passed**
8	With the correct preconditions, a call to `fulfillAdvancedOrder` will always succeed.	Echidna	**Passed**
9	The remaining portion of a partially filled order can be filled via a call to `fulfillAdvancedOrder`.	Echidna	**Passed**

10	As long as they have received the correct approvals, unrestricted (open) orders can be fulfilled regardless of whether a conduit is being used.	Echidna	**Passed**
11	If a basic or advanced order is successful, the seller will receive all of the consideration items, and the buyer, all of the offer items.	Echidna	**Passed**

Slither script. We wrote Slither scripts to detect improper uses of the `_reentrancyGuard`.

Property	Tool	Script
The same value is not written to `_reentrancyGuard` multiple times.	Slither	Appendix E
All paths (e.g., conditional statements) result in the same `_reentrancyGuard` value.	Slither	Appendix E
All internal calls result in the same `_reentrancyGuard` value.	Slither	Appendix E
The value of `_reentrancyGuard` is not set to `_ENTERED` at the end of a path; if it is, the contract will be trapped.	Slither	Appendix E
The value of `_reentrancyGuard` is set to `_ENTERED` before an external call is executed.	Slither	Appendix E

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 system uses Solidity v0.8.0 arithmetic operations, most of which are tested through unit tests. Moreover, many of the arithmetic and parameter-tuning operations are documented. However, automated fuzz testing of the system to detect complex arithmetic bugs (like that described in TOB-OSC-11) would be beneficial. See appendix G for a list of system invariants that could be tested through fuzzing.	**Moderate**
Auditing	The critical state-changing operations emit sufficient events. The OpenSea team provided a detailed incident response plan that includes points of contact and outlines the steps to be taken when a vulnerability is raised. Appendix H details additional recommendations on developing and maintaining an incident response plan.	**Satisfactory**
Authentication / Access Controls	The system generally adheres to the principle of least privilege; the level of access granted to privileged users is limited, and users can enter and exit the system at will. Moreover, users can adjust their security settings to reflect their use of the system (e.g., the use of a conduit). They can also use nonces to cancel orders and zones to validate orders (with some limitations).	**Satisfactory**
Complexity Management	The system includes a contract that inherits from pure, view, struct, and enum contracts, which helps modularize the architecture. Most functions are documented and concise; however, the complicated use of assembly reduces the codebase's readability and increases the	**Weak**

	likelihood of bugs.	
Cryptography and Key Management	The system performs EIP-712 struct hashing and signature verification correctly. Additionally, the use of the EIP-712 standard decreases the risk of a phishing attack against OpenSea users. We did not evaluate the management of hot wallet keys.	**Satisfactory**
Decentralization	The off-chain orderbook may constitute a point of failure in the system, as the compromise of the off-chain system could lead to a denial-of-service condition. While the Seaport exchange is immutable, the deployment risks for users should be thoroughly documented. Additionally, as users can opt in to the use of conduits, the risks associated with the use of third-party conduits should be explicitly documented.	**Moderate**
Documentation	We were provided with documentation sufficient for analysis of the protocol's process flows, data structures, and assembly code. However, we recommend developing additional documentation regarding the ramifications of low-level calls.	**Satisfactory**
Front-Running Resistance	Using an off-chain orderbook carries an inherent front-running risk; specifically, because callers submit offerers' signatures to the blockchain, an offerer's order could be front-run. Channel updates are also vulnerable to front-running, through which an attacker could transfer funds prior to the removal of a channel (TOB-OSC-9).	**Moderate**
Low-Level Manipulation	The system uses numerous low-level calls to reduce storage-related gas costs. The system also checks the size of the data returned in external calls, and those calls will not result in silent failures.	**Moderate**

Testing and Verification	The system has almost complete unit test coverage, and the few coverage gaps are caused by unfinished tests. In addition to finishing these tests, we recommend using Echidna to perform property testing and differential fuzzing against the reference implementation (TOB-OSC-8).	**Moderate**

Summary of Findings

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

ID	Title	Type	Severity
1	Project dependencies contain vulnerabilities	Patching	Low
2	Lack of zero-value checks on functions	Data Validation	Informational
3	Solidity compiler optimizations can be problematic	Undefined Behavior	Informational
4	Error-prone approach to data validation	Undefined Behavior	Undetermined
5	User-controlled return data can trigger an out-of-gas error	Data Validation	Informational
6	Failure to check existence of orders before cancellation	Data Validation	Informational
7	Callbacks can be used to alter token state	Data Validation	Informational
8	Use of Yul optimization pipeline and solc 0.8.13	Testing	Informational
9	Potential front-running of channel-removal transactions	Timing	Informational
10	Lack of a zero-value check in the validate function	Timing	Low
11	fulfillAdvancedOrder may revert and prevent order fulfillment	Data Validation	Undetermined

Detailed Findings

1. Project dependencies contain vulnerabilities	
Severity: **Low**	Difficulty: **High**
Type: Patching	Finding ID: TOB-OSC-1
Target: `consideration/`	

Description
Although dependency scans did not yield a direct threat to the Seaport codebase, `yarn audit` identified a dependency with a known vulnerability. Due to the sensitivity of the deployment code and its environment, it is important to ensure dependencies are not malicious. Problems with dependencies in the development pipeline could have a significant effect on the Seaport system as a whole. The `yarn audit` output detailing the vulnerability is provided below:

GHSA ID	Description	Dependency	Severity
GHSA-27v7-qhfv-rqq8	Insecure Credential Storage in web3	`web3`	Low

Figure 1.1: An advisory affecting the Seaport codebase's web3 dependency

Exploit Scenario
Alice installs the Consideration dependencies on a clean machine. Unbeknownst to Alice, a dependency of the project contains an exploitable high-severity bug that could lead to the disclosure of sensitive information. Alice subsequently uses the dependency, disclosing sensitive information to an unknown actor.

Recommendations
Short term, use `yarn audit` to ensure dependencies are up to date. Several node modules have been documented as malicious because they execute malicious code when installing dependencies to projects. Keep modules current and verify their integrity after installation.

Long term, consider integrating automated dependency auditing into the development workflow. If a dependency cannot be updated when a vulnerability is disclosed, ensure that the Seaport codebase does not use and is not affected by the vulnerable functionality of the dependency.

2. Lack of zero-value checks on functions

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-OSC-2
Target: `consideration/`	

Description

Certain setter functions fail to validate incoming arguments, so callers can accidentally set important state variables to the zero address.

For example, in the constructor of the `Consideration` contract, developers can define the legacy proxy registry, legacy token transfer proxy, and expected proxy implementation parameters and set their addresses to the zero address.

```
/**
 * @dev Derive and set hashes, reference chainId, and associated domain
 *      separator during deployment.
 *
 * @param legacyProxyRegistry         A proxy registry that stores per-user
 *                                    proxies that may optionally be used to
 *                                    transfer approved ERC721+1155 tokens.
 * @param legacyTokenTransferProxy    A shared proxy contract that may
 *                                    optionally be used to transfer
 *                                    approved ERC20 tokens.
 * @param requiredProxyImplementation The implementation that must be set on
 *                                    each proxy in order to utilize it.
 */
constructor(
    address legacyProxyRegistry,
    address legacyTokenTransferProxy,
    address requiredProxyImplementation
) {
    // Derive hashes, reference chainId, and associated domain separator.
    _NAME_HASH = keccak256(bytes(_NAME));
    _VERSION_HASH = keccak256(bytes(_VERSION));
```

Figure 2.1: The constructor of `ConsiderationBase.sol`

A failure to immediately reset an address that has been set to the zero address could result in unexpected behavior.

Exploit Scenario

Alice accidentally sets a proxy implementation to the zero address when deploying a new version of the `Consideration` contract. The misconfiguration causes the system to behave unexpectedly.

Recommendations

Short term, add zero-value checks to all constructor functions and for all setter arguments to ensure that users cannot accidentally set incorrect values, misconfiguring the system. Document any arguments that are intended to be set to the zero address, highlighting the expected values of those arguments on each chain.

Long term, use the Slither static analyzer to catch common issues such as this one. Consider integrating a Slither scan into the project's continuous integration pipeline, pre-commit hooks, or build scripts.

3. Solidity compiler optimizations can be problematic

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-OSC-3
Target: `consideration/`	

Description
OpenSea has 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.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the `emscripten`-generated `solc-js` compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of `keccak256` was reported.

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—or in the Emscripten transpilation to `solc-js`—causes a security vulnerability in the Consideration 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.

4. Error-prone approach to data validation

Severity: **Undetermined**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-OSC-4
Target: `consideration/`	

Description

The system lacks robust data validation checks. The contracts call assertion functions to validate the results of external calls and assume that reentrancy checks will be performed in nested function calls rather than executing per-function reentrancy checks.

The contracts rely on a nested function flow in which functions perform assertions to validate the results of function calls. As a result, determining the data validation expected to occur is challenging. For example, after executing a token transfer, the `_transferERC20`, `_transferERC721`, and `_transferERC1155` functions need to call `_assertValidTokenTransfer` to check that there is code behind the address of the callee contract.

Similarly, the system's reentrancy guards implicitly assume that state-modifying functions will call `_setReentrancyGuard` and view functions will call `_assertNotReentrant`. This pattern is error-prone because those calls often occur in nested internal calls and may be skipped. Typically, function modifiers are used to clearly indicate that a function will be locked prior to its execution and unlocked upon the completion of its execution. We have provided a lint that can statically detect functions that fail to follow this pattern (appendix E). These lints do not currently raise any warnings, but future iterations of the codebase must also pass these checks.

This diffuse system of data validation requires developers and auditors to increase their focus on the context of a call, which is made more difficult by the use of low-level assembly. More importantly, it makes the code less robust. Developers cannot modify a function in isolation; instead, they have to look at all transactions and stack traces to ensure that the required validation is performed correctly. This process is error-prone and increases the likelihood that high-severity issues will be introduced into the system.

Exploit Scenario

Alice, a Configuration protocol developer, adds a new function that calls an existing function. This existing function makes implicit assumptions about the data validation that occurs before it is called. However, Alice is not fully aware of those assumptions and fails to

implement the required data validation, creating an attack vector that can be used to steal funds from the protocol.

Recommendations

Short term, integrate the lint provided in appendix E into the repository's continuous integration pipeline and add tests for publicly callable functions to ensure that they adequately handle reverts. Additionally, consider creating a flowchart to map out the expected use of reentrancy guards and to ensure that a mutex is set and checked in all functions that require one.

Long term, ensure that the protocol's functions perform exhaustive validation of their inputs and of the system's state and that they do not assume that validation has been performed further up in the call stack (or will be performed further down). Such assumptions make the code brittle and increase the likelihood that vulnerabilities will be introduced when the code is modified. Any implicit assumptions regarding data validation or access controls should be explicitly documented; otherwise, modifications to the code could break those important assumptions.

5. User-controlled return data can trigger an out-of-gas error

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-OSC-5
Target: `ConsiderationInternal.sol`, `ConsiderationPure.sol`	

Description

When an external call fails, `_revertWithReasonIfOneIsReturned` copies the return data into memory. However, there is no limit on the size of the return data it copies. An attacker could exploit this to force `_revertWithReasonIfOneIsReturned` to raise an out-of-gas error instead of an error indicating that the external call failed.

The `_revertWithReasonIfOneIsReturned` function is meant to bubble up the reasons for the revert of an external call:

```
function _revertWithReasonIfOneIsReturned() internal pure {
    assembly {
        // If data was returned...
        if returndatasize() {
            // Copy returndata to memory, overwriting existing memory.
            returndatacopy(0, 0, returndatasize())

            // Revert, specifying memory region with copied returndata.
            revert(0, returndatasize())
        }
    }
}
```

Figure 5.1: `ConsiderationPure.sol#L1254-1265`

To do this, it copies the external call's return data into memory and returns that data.

After an external call, the caller will retain at least 1/64th of the gas available before the call (see EIP-150). One might assume that this amount of gas would be sufficient for `_revertWithReasonIfOneIsReturned` to bubble up the reasons for the failure. However, a malicious actor could craft an external call that would trigger the expansion of a large amount of memory, causing the transaction to consume all of the gas. The user might then believe that the transaction failed because too little gas was provided, when in reality, the external call was the source of the failure.

An attacker could thereby trick a user into performing the same transaction multiple times (and adding more gas each time), causing the user to incur a loss.

Exploit Scenario

Eve creates a consideration item to sell her NFT for ETH on the Seaport marketplace. She provides the address of her smart contract wallet, which performs gas-intensive operations, in the `receive` method. Bob calls `fulfillBasicOrder` on Eve's consideration item. The `_transferEth` function sends a low-level call to Eve's smart contract wallet, which consumes nearly all of the forwarded gas, calls `revert`, and returns a large array of bytes to the calling contract. The `Consideration` contract's `_revertWithReasonIfOneIsReturned` function then calls `returndatacopy`, which throws an out-of-gas exception rather than reverting as intended. Bob tries resending the transaction with much more gas, but the transaction fails again. As a result, Bob loses the gas he sent with the transactions and does not receive the NFT.

Recommendations

Short term, have the contract check the size of the return data before loading it into memory and return a generic error message if it is too large.

Note that the changes made to `_revertWithReasonIfOneIsReturned` (those introduced in commit `00bd847df9971e6c1e61c7c4b58e6db6ce95a93f`) are not in line with our recommendations and introduce needless complexity without addressing this issue.

Long term, document this issue and any mitigations that have been implemented for it to inform users, developers, and third-party integrations that an out-of-gas error may be raised when a call to an external contract fails. Consider integrating Echidna into the development process to thoroughly test all user-controlled inputs and to check for any inconsistent or unexpected behavior caused by those inputs.

6. Failure to check existence of orders before cancellation

Severity: **Informational**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-OSC-6
Target: `consideration/contracts/Consideration.sol`	

Description
When a user calls `cancel` on an order, the `isCanceled` property is set to `true`, and the `cancel` function will return successfully regardless of whether the order has been validated or signed by the user. As a result, a user could mistakenly cancel a nonexistent order, leaving the order he or she meant to cancel available for fulfillment.

Exploit Scenario
Bob signs an order to sell an NFT and publishes the order. The floor price of the NFT rises, so Bob signs and publishes a new order with a higher price. He then tries to cancel his old order but accidentally calls `cancel` on a nonexistent order. Despite the mistake, the call is successful, leading Bob to think that he canceled the correct order. When his original order is fulfilled at the lower price, he is caught by surprise.

Recommendations
Short term, provide clear user documentation informing users to check the status of an order after attempting to cancel the order; that way, users will be sure that they have canceled the order(s) they intended to cancel. Additionally, consider redesigning the UI such that it provides users with a list of valid orders when they attempt to make a cancellation; this will enable users to select a valid order from the list instead of providing the order parameters themselves.

Long term, review all opportunities for user error and ensure that the documentation clearly describes the actions users can take to minimize risk.

7. Callbacks can be used to alter token state

Severity: **Informational**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-OSC-7
Target: `ConsiderationInternal.sol`	

Description

The callback function executed when transferring an NFT can be used to alter the state of another NFT contract (changing its ether balance or number of experience points or equipped items, for example). A state change could cause an unexpected decrease in the value of a purchase.

Most standard NFT implementations use an `onERC{721|1155}Received` hook to execute a callback when a token is transferred. The hook is called on the recipient contract:

```
if (to.isContract()) {
    try IERC721Receiver(to).onERC721Received(_msgSender(), from, tokenId, _data)
returns (bytes4 retval) {
        return retval == IERC721Receiver.onERC721Received.selector;
        …
```

Figure 7.1: The `_checkOnERC721Received` function in `ERC721.sol#L394-L396`

During the order fulfillment process, the callback is called each time `_transfer` is invoked:

```
// Transfer the item specified by the execution.
_transfer(item, execution.offerer, execution.conduitKey);
```

Figure 7.2: The `_performFinalChecksAndExecuteOrders` function in `ConsiderationInternal.sol#L1897-L1898`

When the callback executes, a recipient can alter the state of any other NFT he or she owns by transferring its assets, removing its experience points, or otherwise changing its attributes. In this way, the user can lower the value of another NFT yet to be transferred.

Exploit Scenario

Eve, a Seaport user, creates an offer for an NFT that has 10 ETH and 10 experience points. The NFT is priced at 1 ETH. Bob fulfills the order by providing his NFT as the consideration item. When the transfer is being processed, `onERC721Received` is called on Eve's recipient contract. Eve, through that method, calls her NFT's contract and withdraws its ETH, resetting its experience point balance to zero. When the fulfillment process is complete,

Bob receives Eve's NFT; however, it lacks the ETH and experience points he expected to receive when he sent the transaction.

Recommendations

Short term, implement on-chain validation of the most common collections of NFTs with attributes that can be altered by their owners, or freeze NFTs prior to executing state changes. Additionally, expand the user documentation to explain the risk inherent in purchasing NFTs with changeable attributes. Lastly, identify the riskiest NFT collections and ensure that the UI highlights the underlying risks.

Long term, evaluate mechanisms for enabling users to check the state of an NFT after a transfer but before the fulfillment transaction is complete. One solution would be to implement an optional Oracle call that is able to check whether a particular order has the same state it had when it was listed once the order has been fulfilled.

8. Use of Yul optimization pipeline and solc 0.8.13

Severity: **Informational**	Difficulty: **Low**
Type: Testing	Finding ID: TOB-OSC-8
Target: `Consideration.sol`	

Description

The Yul intermediate language pipeline is used to compile the `Consideration` contract. This pipeline was considered experimental until March 16, 2022, the day Solidity version 0.8.13 was released.

Presumably, the Solidity code compiled through this pipeline could not be compiled through the previous version of the pipeline, as compilation would result in "stack-too-deep" errors. Thus, the codebase cannot be compiled and tested without the new optimization pipeline. Ideally, it would be possible to compile the code with and without optimizations, and differential fuzzing of those two versions would not identify any discrepancies.

Additionally, solc 0.8.13 features a compiler directive for optimizing in-line Yul blocks that are marked as memory-safe. This feature should not be used when the codebase is compiled, as `Consideration`'s in-line Yul accesses memory and directly manipulates the free memory pointer.

Recommendations

Short term, write differential fuzzing tests to ensure equivalence between the functions containing Yul and the functions in the reference implementation. Additionally, ensure that both implementations pass these tests.

Long term, compare the gas costs of the optimized and reference implementations and identify any functions that reduce the code's readability in exchange for insignificant gas savings. If the gas cost of the reference version is not prohibitively expensive, use it instead of the optimized version.

Additionally, continue to expand the test suite, prioritizing property testing, and monitor the Solidity GitHub repository for issues related to the Yul pipeline.

9. Potential front-running of channel-removal transactions

Severity: **Informational**	Difficulty: **High**
Type: Timing	Finding ID: TOB-OSC-9
Target: `Consideration.sol`	

Description

If a compromised channel is added to a conduit or a vulnerability is discovered in an existing channel, the conduit's owner may wish to revoke the channel's access to the conduit (i.e., to remove the channel). However, transactions sent by conduit owners to remove channels are vulnerable to front-running. Because users approve conduits to spend tokens on their behalf, an attacker who has front-run a transaction could then use a channel that allows arbitrary calls to steal users' tokens.

When a user sends a transaction to remove a vulnerable channel, the user inadvertently reveals that he or she has authorized a vulnerable channel. An attacker could then target the user programmatically, by decoding the transaction's calldata to identify the transaction in the mempool. This vulnerability may have a particularly strong impact on users who elect to use a conduit not controlled by OpenSea's multisignature contract or governance.

However, the majority of users will likely use OpenSea's first-party conduit and will thus be able to remove a vulnerable channel by executing only one transaction. Users can send these single transactions via private relayers such as flash bots to prevent them from being detected in public mempools.

Exploit Scenario

A channel contract is found to contain a vulnerability that enables attackers to call token contracts directly. When the many users who have activated the channel contract in their conduits are informed of the vulnerability, they send transactions to remove the channel. In doing so, they reveal that their addresses are vulnerable to attack. An attacker then uses the conduits they have approved to transfer their tokens to his own account.

Recommendations

Short term, set up an infrastructure for sending channel updates through trusted private relay networks.

Long term, educate users on the risks associated with using third-party conduits (appendix C) and channels, and investigate designs that can help prevent the abuse of users' token approvals.

10. Lack of a zero-value check in the validate function	
Severity: **Low**	Difficulty: **Medium**
Type: Timing	Finding ID: TOB-OSC-10
Target: `Consideration.sol`	

Description

Certain token `transfer` functions have an assertion, `_assertNonZeroAmount`, that reverts if the token amount is set to zero. However, the `validate` function does not check whether an order's token amount is set to zero and will validate such an order, returning a boolean value of `true`.

This lack of validation is particularly problematic for users whose orders include ether, ERC20 tokens, or ERC1155 tokens; this is because an order with an amount of zero would appear to be valid but would cause those tokens' respective `transfer` functions to revert. The "valid" order would also be stored on-chain, misleading users who rely on the on-chain data to identify transactions that no longer require signature verification (and causing them to waste gas).

```
function _assertNonZeroAmount(uint256 amount) internal pure {
    if (amount == 0) {
        revert MissingItemAmount();
    }
}
```

Figure 10.1: The `_assertNonZeroAmount` function

Recommendations

Short term, add a check for zero-value token amounts to the `validate` function. Alternatively, if the `validate` function is currently behaving as intended, clearly document its expected behavior for users and third-party integrations.

Long term, review the system's functions to ensure that their data validation behavior is consistent.

11. fulfillAdvancedOrder may revert and prevent order fulfillment

Severity: **Undetermined**	Difficulty: **Medium**
Type: Data Validation	Finding ID: TOB-OSC-11
Target: `Consideration.sol`	

Description

The functions `_getFraction` and `_locateCurrentAmount` are used in the execution of advanced orders to facilitate partial orders and the use of floating prices. The use of fractions or floating prices may cause arithmetic overflows in checked blocks, leading to runtime panics and causing `fulfillAdvancedOrder` to revert.

An overflow could cause a fulfillable order to suddenly become unfulfillable; every attempt to fill the order would then cause a revert, violating user expectations. Notably, an attacker could cause an overflow by using a fractional amount scaled to nearly the maximum value of a 256-bit unsigned integer; alternatively, he or she could set an end price that would cause an overflow as the order approached its expiration.

```
function _getFraction(
    uint256 numerator,
    uint256 denominator,
    uint256 value
) internal pure returns (uint256 newValue) {
    [...]
    uint256 valueTimesNumerator = value * numerator;
    [...]
```

Figure 11.1: The `_getFraction` function

```
function _locateCurrentAmount(
    uint256 startAmount,
    uint256 endAmount,
    uint256 elapsed,
    uint256 remaining,
    uint256 duration,
    bool roundUp
) internal pure returns (uint256) {
     [...]
       uint256 totalBeforeDivision = ((startAmount * remaining) +
           (endAmount * elapsed) +
           extraCeiling);
     [...]
```

Figure 11.2: The `_locateCurrentAmount` function

Recommendations

Short term, investigate the impact of reverts caused by overflows in checked blocks and determine whether more input validation is required. Document this overflow behavior for external parties.

Long term, write unit and fuzz tests that trigger this behavior. Then, either develop a patch for the issue or update the documentation to clarify that reverts are expected in certain edge cases.

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. Risks Associated with Third-Party Conduits

Third parties can deploy conduits, which control asset transfers. Third-party conduits increase the risk of a compromise of user funds, as their owners could add malicious or vulnerable channels. Users should not interact with conduits without thorough investigation to ensure that the conduits' channels do not contain security vulnerabilities and that the conduits' owners are properly managed by a multisignature wallet or a decentralized autonomous organization (DAO). Users should verify that conduit owners do the following:

- **Conduct third-party security reviews of channels.** Before a channel is added to a conduit, it should undergo a comprehensive security review by a third-party auditor.
- **Document the channel update process.** Conduit owners should justify why channels are added and removed and should create a robust pipeline to prevent malicious activity. For each conduit, a list of channels, their contract addresses, and background documents (e.g., audit reports and governance form discussions) should be readily available.
- **Address Slither's findings.** Slither, a Solidity static analysis tool, will catch many common security findings and should be integrated in the channel development process.

D. Echidna Integration

During the audit, we integrated Echidna with the codebase to implement invariant checks. This practice allowed us to identify system properties and implement fuzz tests, which automatically generate random inputs to call smart contracts.

Differential Testing

Differential Echidna tests are used to check equivalence in assembly blocks. In running the `ExtcodeSize` testing contract (figure E.1), we compared the result of the original line of assembly with a simplified implementation. In the `test_equivalence` function (the only publicly callable function in these contracts), there is a 50% chance that `tokenAddress` is equivalent to zero or a newly deployed `TestToken`. With this test, Echidna will explore two states—contracts that have a nonzero size and contracts that have a size of zero—to try to find an input that breaks equivalence:

```
pragma solidity 0.8.13;
import "../../test/TestERC20.sol";
// echidna-test-2.0 . --contract ExtcodeSize  --test-mode assertion
contract ExtcodeSize {
      function original_extcodesize(address tokenAddress, bool success) private view
returns (bool result){
            assembly {
                  result := iszero(and(iszero(iszero(extcodesize(tokenAddress))),
success))
            }
      }
      function simplified_extcodesize(address tokenAddress, bool success) private
view returns(bool result) {
            assembly {
                  result := or(iszero(extcodesize(tokenAddress)), iszero(success))
            }
      }
      function test_equivalence(uint128 num, bool success) public {
            address tokenAddress = address(0);
            if (num%2 == 0) {
                  tokenAddress = address(new TestERC20());
            }
            assert(original_extcodesize(tokenAddress, success) ==
simplified_extcodesize(tokenAddress, success));
      }
}
```

Figure D.1: The `ExtcodeSize.sol` Echidna test

The highlighted assertion calls the two functions with the same parameters and asserts that the returned values are equivalent. This pattern can be extended to tests that compare the results of a series of functions outside of single-line assembly.

Stateful End-to-End Tests

As opposed to stateless tests, which allow Echidna to explore a system's state only over a single transaction, stateful end-to-end tests allow Echidna to explore a much wider range of contract behavior and to detect violations that require the state to change across multiple interactions. We used these tests to test properties against a deployed version of the `Consideration` contracts by calling various permutations of functions to reach different code paths.

To guide Echidna to produce valid orders, we derived order information from a seed to produce quasi-random orders. Echidna explored six routes, consisting of transfers between ether, ERC721, ERC1155, and ERC20 tokens:

1. Ether to ERC1155
2. Ether to ERC721
3. ERC20 to ERC721
4. ERC20 to ERC1155
5. ERC721 to ERC20
6. ERC115 to ERC20

There are multiple consideration and offer items for advanced orders, but only one item for basic orders. For partial orders, the fuzzer provides a fractional amount, filling a portion of the available number of consideration and offer items. To simplify testing, each order is formed and the corresponding tokens are minted to the buyer and seller. The order is then processed by `Consideration`, and, finally, the invariants that represent how much each account should receive of each token are validated.

```
function testFulfillAdvancedOrder(bytes32 seed, uint120 numerator, uint120
denominator) public payable {
    // FULL_OPEN: 0, PARTIAL_OPEN: 1
    uint orderType = uint(seed) % 2;

    // For partial orders we validate the fraction
    // to avoid BadFraction and InexactFraction reverts
    if (orderType == 1) {
        uint amount = uint256(uint112(uint256(seed)));
        require(numerator < denominator && numerator != 0);
        uint256 valueTimesNumerator = amount * numerator;
        bool exact;
        uint newValue;
```

```
        assembly {
            newValue := div(valueTimesNumerator, denominator)
            exact := iszero(mulmod(amount, numerator, denominator))
        }
        require(exact);
    }
    // Evenly distribute route between 0 and 5
    uint256 route = uint(seed) % (6);
    (OrderParameters memory orderParams, uint totalTokens, uint totalItems, uint
uniqueId) = createOrderParameters(_seller, _buyer, seed, route, false);
    orderParams.conduitKey = _conduitKeyActive;

    // Sign order on behalf of seller
    uint256 nonce = _opensea.getNonce(_seller);
    bytes32 orderHash =
_opensea.getOrderHash(convertOrderParametersToOrderComponents(orderParams, nonce));
    (, bytes32 domainSeparator, ) = _opensea.information();
    bytes memory sig = signOrder(orderHash, domainSeparator);

    // Send entire balance for ether orders (should refund)
    uint offerItemType = uint(orderParams.offer[0].itemType);
    uint value = offerItemType < 2 ? address(this).balance : 0;
    AdvancedOrder memory order;
    if (orderType == 0) /*FULL_OPEN*/ {
        order = AdvancedOrder({
            parameters: orderParams,
            signature: sig,
            numerator: uint120(1),
            denominator: uint120(1),
            extraData: abi.encode(bytes32(0))
        });
    } else /*PARTIAL_OPEN*/ {
        order = AdvancedOrder({
            parameters: orderParams,
            signature: sig,
            numerator: numerator,
            denominator: denominator,
            extraData: abi.encode(bytes32(0))
        });
        // Scale order to fill fractional amount
        uint remaining = totalTokens - ((totalTokens * numerator) / denominator);
        totalTokens -= remaining;
```

```
        pendingPartialOrders[pendingPartialOrderIndex] = order;
        pendingPartialOrdersAmount[pendingPartialOrderIndex++] = remaining;
    }
    // This has no effect without providing a merkle root
    CriteriaResolver[] memory resolvers = new CriteriaResolver[](0);
    try _opensea.fulfillAdvancedOrder{value: value}(order, resolvers,
_conduitKeyActive) returns(bool res) {
        assert(res);
    }
    catch Panic(uint reason) {
        emitAndFail("_opensea.fulfillAdvancedOrder FAILED", route, reason);
    }

    // Check that buyers and sellers received expected amounts
    _assertFundsReceived(_seller, _buyer, route, totalTokens, totalItems, uniqueId);
}
```

Figure D.2: Advanced order fuzzing

After fulfilling orders, the anticipated quantity and identifier (the ID for ERC1155 and ERC721) of tokens for the buyer and seller are checked:

```
function _assertFundsReceived(address seller, address buyer, uint256 route, uint256
totalTokens, uint256 totalItems, uint uid) internal {
    if (route == 0) /*NATIVE TO ERC721*/ {
        if(seller.balance < totalTokens) {
            emitAndFail("/*NATIVE TO ERC721*/ seller", seller.balance, totalTokens);
        }
        if (_erc721.balanceOf(buyer) < totalItems) {
            emitAndFail("/*NATIVE TO ERC721*/ buyer", _erc721.balanceOf(buyer),
totalItems);
        }
    } else if (route == 1) /*NATIVE TO ERC1155*/ {
        if(seller.balance < totalTokens) {
            emitAndFail("/*NATIVE TO ERC1155*/ seller", seller.balance,
totalTokens);
        }
        if (_erc1155.balanceOf(buyer, uid) < totalTokens) {
            emitAndFail("/*NATIVE TO ERC1155*/ buyer", _erc1155.balanceOf(buyer,
uid), totalTokens);
        }
    } else if (route == 2) /*ERC20 TO ERC721*/ {
        if (_erc20.balanceOf(seller) < totalTokens) {
            emitAndFail("/*ERC721 TO ERC20 */ FAILED", _erc20.balanceOf(seller),
totalTokens);
        }
        if (_erc721.balanceOf(buyer) < totalItems) {
            emitAndFail("/*ERC721 TO ERC20 */ FAILED", _erc721.balanceOf(buyer),
```

```
totalItems);
        }
    } else if (route == 3) /*ERC20 TO ERC1155*/ {
        if (_erc1155.balanceOf(buyer, uid) < totalTokens) {
            emitAndFail(" /*ERC115 TO ERC20 */ buyer", _erc1155.balanceOf(buyer,
uid), totalTokens);
        }
        if (_erc20.balanceOf(seller) < totalTokens) {
            emitAndFail(" /*ERC115 TO ERC20 */ seller", _erc20.balanceOf(seller),
totalTokens);
        }
    } else if (route == 4) /*ERC721 TO ERC20 */ {
        if (_erc20.balanceOf(buyer) < totalTokens) {
            emitAndFail("/*ERC20 TO ERC721*/ FAILED", _erc20.balanceOf(buyer),
totalTokens);
        }
        if (_erc721.balanceOf(seller) < totalItems) {
            emitAndFail("/*ERC20 TO ERC721*/ FAILED", _erc721.balanceOf(seller),
totalItems);
        }
    } else if (route == 5) /*ERC115 TO ERC20 */ {
        if (_erc1155.balanceOf(seller, uid) < totalTokens) {
            emitAndFail(" /*ERC20 TO ERC1155*/ seller", _erc1155.balanceOf(seller,
uid), totalTokens);
        }
        if (_erc20.balanceOf(buyer) < totalTokens) {
            emitAndFail(" /*ERC20 TO ERC1155*/ buyer", _erc20.balanceOf(buyer),
totalTokens);
        }
    }
}
```

Figure D.3: Validating receipt of funds

The setup in figure E.4 allows Echidna to target the order validation flow, which creates an order with the adequate parameters, validates the order, and asserts that the order status was validated and not canceled:

```
function testValidate(bytes32 seed) public override {
    Order[] memory orders = new Order[](1);
    orders[0] = Order({
        parameters: createOrderParameters(seed, uint256(seed)),
        signature: DEFAULT_SIG
    });

    bool res = _opensea.validate(orders);
    assert(res);

    bytes32 orderHash =
_opensea.getOrderHash(convertOrderParametersToOrderComponents(orders[0].parameters))
;
    assert(orderHash != bytes32(0));
```

```
    (bool valid, bool cancelled, uint256 filled, uint256 size) =
_opensea.getOrderStatus(orderHash);
    assert(valid);
    assert(!cancelled);
}
```

Figure D.4: Example of `testValidate` preconditions and postconditions

Further Development

We recommend continuing to add invariants and increasing the code coverage of property testing by doing the following:

- Generate random recipients and check that they receive consideration items.
- Create floating orders, force the block time forward, and assert that they are correctly filled.
- Add property testing for `matchOrders`, `matchAdvancedOrders`, and `matchAvailableAdvancedOrders`.
- Write helper functions to generate Merkle roots to conduct property testing of the criteria resolver functionality.
- Incorporate zone order validation into the property tests.

E. Slither Script

Tokens transferred by the Seaport exchange may have hooks, and the contracts with which it interacts may have callbacks. Because it is important that reentrant calls do not unexpectedly update the state of the exchange during inner calls, changing the execution result, functions that make external calls use `_setReentrancyGuard`. This guard creates a global lock on a contract, preventing multiple interactions with state-modifying external functions in the same transaction.

The following script can be used to explore the paths of all Seaport entry points and to verify that the reentrancy guard is set correctly by tracking its value on every node.

The script checks for the following issues:

- A value (e.g., `_NOT_ENTERED`) is written to `_reentrancyGuard` multiple times in the same path.
- A control flow structure (e.g., an `if`, `then`, or `else` statement) results in different `_reentrancyGuard` values.
- `_reentrancyGuard` has different values in the return statements of an internal call.
- An entry point causes `_reentrancyGuard` to be set to `_ENTERED`, trapping the contract.

```
import sys
from enum import Enum
from typing import Optional, Dict, Set, Tuple

from slither import Slither
from slither.core.cfg.node import Node
from slither.core.declarations import Function
from slither.slithir.operations import Assignment, InternalCall, HighLevelCall,
LowLevelCall

# Known limitations
# - Constructor and modifiers are not handled
# - Recursion, or function that do not return (or always revert) are not supported
# - Function pointers are not supported
# - Writing _reentrancyGuard in assembly is not supported

class Entered(Enum):
    NOT_SET = 0
    NOT_ENTERED = 1
    ENTERED = 2

# pylint: disable=too-many-branches
```

```
def _transfer_function(
    node: Node, entered: Optional[Entered], callstack: Set[Function], results:
Set[str]
) -> Entered:
    """
    Iterate over the IRs of a given block

    :param node:
    :type node:
    :param entered:
    :type entered:
    :param callstack:
    :type callstack:
    :return:
    :rtype:
    """
    for ir in node.irs:
        if isinstance(ir, Assignment):
            if ir.lvalue.name == "_reentrancyGuard" and ir.rvalue.name ==
"_NOT_ENTERED":
                if entered == Entered.NOT_ENTERED:
                    results.add(
                        f"_NOT_ENTERED is written two times in {node}
({node.source_mapping_str})"
                    )
                entered = Entered.NOT_ENTERED
            if ir.lvalue.name == "_reentrancyGuard" and ir.rvalue.name ==
"_ENTERED":
                if entered == Entered.ENTERED:
                    results.add(
                        f"_ENTERED is written two times in {node}
({node.source_mapping_str})"
                    )
                entered = Entered.ENTERED

        if isinstance(ir, InternalCall):
            call_state: Dict[Node, Entered] = {}

            if ir.function in callstack:
                print(
                    f"The script does not handle codebases with recursive calls
({ir.function} in {node} ({node.source_mapping_str})")
                sys.exit(-1)
            _explore(
                ir.function.entry_point,
                call_state,
                callstack | {ir.function},
                results,
                init_value=entered,
            )

            state_after_internal_call: Optional[Entered] = None
            for node_function in ir.function.nodes:
```

```
                if node_function.will_return:
                    candidate = call_state[node_function]
                    if (
                        state_after_internal_call is not None
                        and candidate != state_after_internal_call
                    ):
                        results.add(
                            f"The function {ir.function}
({ir.function.source_mapping_str}) return different state "
                        )
                    state_after_internal_call = candidate
            if state_after_internal_call is None:
                print(f"Can't propagage info because {ir.function} always reverts")
                sys.exit(-1)
            entered = state_after_internal_call

        if (
            isinstance(ir, (HighLevelCall, LowLevelCall))
            and entered != Entered.ENTERED
            and ir.can_reenter()
        ):
            results.add(
                f"{node} ({node.source_mapping_str}) is not protected by the
reentrancy guard"
            )

    assert entered
    return entered


def _merge_fathers(
    node: Node, state: Dict[Node, Entered], results: Set[str]
) -> Tuple[bool, Optional[Entered]]:
    """
    Merge the value from the fathers

    :param node: Given node
    :type node:
    :param state: Curretn state
    :type state:
    :return: (bool, Entered): if not all the fathers were explored, merged state
    :rtype:
    """
    state_from_fathers: Optional[Entered] = None
    no_fix_point = False
    for father in node.fathers:
        if father not in state:
            no_fix_point = True
        else:
            candidate = state[father]

            if state_from_fathers is not None and candidate != state_from_fathers:
                results.add(
```

```
                    f"Not all fathers have the same state {node}
({node.source_mapping_str})"
                )

            state_from_fathers = candidate
    return no_fix_point, state_from_fathers


def _explore(
    node: Node,
    state: Dict[Node, Entered],
    callstack: Set[Function],
    results: Set[str],
    init_value: Optional[Entered] = None,
) -> None:
    """
    Explore iterate over all the nodes and propagate the value assigned to
_NOT_ENTERED
    The fix point is reached on a node if
    - It was already explored
    - All fathers were explored
    - The propagation on the IR did not lead to new info

    Because writing to _reentrancyGuard are simple assignement, outside of loop, the
convergence is fast

    During the exploration, the function look for:
    - If _reentrancyGuard is written to  _NOT_ENTERED in a path where it already has
this value (same for _ENTERED)
    - If there is a control flow structure (if/then/else, ..), _reentrancyGuard must
have only 1 possible value
    - Similarly, on the internal call, all the return statement must leave
_reentrancyGuard with the same value

    :param node: Entry point
    :type node:
    :param state: Current value of _reentrancyGuard
    :type state:
    :param init_value: Initial _reentrancyGuard value (only for internal call)
    :type init_value:
    :return:
    :rtype:
    """

    original_entered_end_value: Optional[Entered] = None
    no_fix_point = False

    if node in state:
        original_entered_end_value = state[node]
    else:
        no_fix_point = True

    (no_fix_point_father, entered) = _merge_fathers(node, state, results)
```

```
    no_fix_point |= no_fix_point_father

    if init_value:
        entered = init_value

    entered = _transfer_function(node, entered, callstack, results)
    state[node] = entered

    if original_entered_end_value is not None and original_entered_end_value !=
entered:
        no_fix_point = True

    if no_fix_point:
        for son in node.sons:
            _explore(son, state, callstack, results)


def run_analysis(function: Function) -> None:
    if not function.is_implemented:
        return
    state: Dict[Node, Entered] = {}

    results: Set[str] = set()

    _explore(function.entry_point, state, {function}, results,
init_value=Entered.NOT_SET)

    # Check that all the return statement ends with NOT_SET or _NOT_ENTERED
    for node in function.nodes:
        if node.will_return:
            entered = state[node]

            if entered == Entered.ENTERED:
                results.add(f"Function {function} ends in the entered state")

    if results:
        print(f"# In {function.canonical_name}:")
        for r in results:
            print(r)


def main() -> None:
    sl = Slither(".", ignore_compile=True)

    contracts = sl.get_contract_from_name("Consideration")
    if not contracts:
        print("Consideration not found")
    for contract in contracts:
        for function in contract.functions_entry_points:
            run_analysis(function)
```

```
if __name__ == "__main__":
    main()
```

Figure E.1: The reentrancy guard verification script

Because this script currently does not raise any alarms, we created a test case (figure C.2) to demonstrate its utility:

```
contract Consideration{

    uint256 internal constant _NOT_ENTERED = 1;
    uint256 internal constant _ENTERED = 2;

    uint256 internal _reentrancyGuard;


    function set_not() internal{
        _reentrancyGuard = _NOT_ENTERED;
    }

    function can_return_entered(bool b) public{
        _reentrancyGuard = _ENTERED;
        if(b){
            set_not();
        }
    }

    function can_set_two_times(bool b) public{
        set_not();
        set_not();
    }

    function different_return(bool b) internal{
        if(b){
            set_not();
            return;
        }
    }

    function f() public{
       different_return(true);
    }

    function let_variable_set_to_entered_state() public{
        _reentrancyGuard = _ENTERED;
    }


    function protected_call() public{
        _reentrancyGuard = _ENTERED;
        (msg.sender).call("");
        _reentrancyGuard = _NOT_ENTERED;
```

```
    }


    function unprotected_call() public{
        (msg.sender).call("");
    }
}
```

Figure E.2: The failing test contract used to run the reentrancy verification script

```
# In Consideration.can_return_entered(bool):
Not all fathers have the same state END_IF (test.sol#15-17)
Function can_return_entered ends in the entered state
# In Consideration.can_set_two_times(bool):
_NOT_ENTERED is written two times in EXPRESSION _reentrancyGuard =
_NOT_ENTERED (test.sol#10)
# In Consideration.f():
The function different_return (test.sol#25-30) return different state
# In Consideration.let_variable_set_to_entered_state():
Function let_variable_set_to_entered_state ends in the entered state
# In Consideration.unprotected_call():
EXPRESSION (msg.sender).call() (test.sol#49) is not protected by the
reentrancy guard
```

Figure E.3: The script's output for this failing test contract

F. Transaction Traces

We set up an interactive JavaScript console to aid in determining the system's memory and stack inputs. We ran a local hardhat node in the background and deployed all the required contracts before launching the console. We wrote helper functions that provided sensible default values for orders and used the `debug_traceTransaction` JSON RPC method to generate traces of transactions that attempted to validate or fulfill orders. These transaction traces provided snapshots of the stack and memory after every opcode, which aided our investigation of the assembly code. We identified rare opcodes (SSTORE, SLOAD, CALLER, etc.) first and mapped their locations in the source code to provide landmarks that aided us in identifying the most high-risk assembly in the trace.

By mapping the source code to memory and stack snapshots, we were able to analyze the validity of the system's handling of memory. Assembly code that operated in reserved slots of memory or that overwrote and later restored memory slots received special attention.

For example, in `_getOrderHash`, some memory slots were overwritten to generate type hashes, avoiding unnecessary memory allocation to preserve gas. The trace of one of these hashes, shown below, helped us verify that memory was being correctly overwritten and then restored.


```
{
      "index": 1147,
      "pc": 16611,
      "op": "SHA3",
      "gas": 26039,
      "gasCost": 36,
      "depth": 1,
      "stack": [
        "0000000000000000000000000000000000000000000000000000000000000044",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "0000000000000000000000000000000000000000000000000000000000000000",
        "00000000000000000000627306090abab3a6e1400e9345bc60c78a8bef57",
        "0000000000000000000000000000000000000000000000000000000000000064",
        "00000000000000000000000000000000000000000000000000000000000000a4",
        "0000000000000000000000000000000000000000000000000000000000000f3a",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "0000000000000000000000000000000000000000000000000000000000000630",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "00000000000000000000000000000000000000000000000000000000000001c0",
        "0000000000000000000000000000000000000000000000000000000000000080",
        "0000000000000000000000000000000000000000000000000000000000000020",
        "00000000000000000000000000000000000000000000000000000000000003c0",
        "00000000000000000000000000000000000000000000000000000000000000c0",
        "0000000000000000000000000000000000000000000000000000000000000020",
        "00000000000000000000000000000000000000000000000000000000000003c0"
```

```
      ],
      "memory": [
        "0000000000000000000000627306090abab3a6e1400e9345bc60c78a8bef57",
        "0000000000000000000000000000000000000000000000000000000000000002",
        "00000000000000000000000000000000000000000000000000000000000003c0",
        "0000000000000000000000000000000000000000000000000000000000000000",
        "0000000000000000000000627306090abab3a6e1400e9345bc60c78a8bef57",
        "0000000000000000000000000000000000000000000000000000000000000000",
        "00000000000000000000000000000000000000000000000000000000000001e0",
        "00000000000000000000000000000000000000000000000000000000000002c0",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "000000000000000000000000000000000000000000000000000626aa1ad",
        "000000000000000000000000000000000000000000000000000626aafbd",
        "0000000000000000000000000000000000000000000000000000000000000000",
        "0000000000000000000000000000000000000000000000000000000000000000",
        "0000000000000000000000000000000000000000000000000000000000000000",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "0000000000000000000000000000000000000000000000000000000000000220",
        "0000000000000000000000000000000000000000000000000000000000000002",
        "00000000000000000000000f12b5dd4ead5f743c6baa640b0216200e89b60da",
        "0000000000000000000000000000000000000000000000000000000000000000",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "0000000000000000000000000000000000000000000000000000000000000300",
        "0000000000000000000000000000000000000000000000000000000000000001",
        "00000000000000000000008cdaf0cd259887258bc13a92c0a6da92698644c0",
        "0000000000000000000000000000000000000000000000000000000000000000",
        "000000000000000000000000000000000000000001bc16d674ec80000",
        "0000000000000000000000000000000000000000de0b6b3a7640000",
        "0000000000000000000000627306090abab3a6e1400e9345bc60c78a8bef57",
        "8c4fcd46c528a1a5c16be866d612801c8fd7aad1a210ab7f46cc8a3196c7e107"
      ],
      "storage": {
        "0000000000000000000000000000000000000000000000000000000000000000":
"0000000000000000000000000000000000000000000000000000000000000001",
        "ded101565a23504cd4339827add3a8b16f30ceff2912c1b6ef015848eade7942":
"0000000000000000000000000000000000000000000000000000000000000001"
      }
    }
```


Figure F.1: A snapshot of the machine state before the keccak256 hash on line 398 of `ConsiderationInternalView.sol`

Due to time constraints, this tooling is incomplete and was used only to investigate certain transaction types. Given more time, we would have taken the following steps to make transaction traces easier to generate:

- Refactor useful utilities such as `getAndVerifyOrderHash` out of `test/index.js` so that they could be available for reuse by other tools, including an interactive console.

- Use the test utilities to create a wrapper for each external method of `Consideration`. The wrapper, which would be called without arguments, would set up accounts and the contract's state (e.g., by minting and approving required tokens) and would set default arguments for a successful call. It would then generate and save the transaction trace to a file for further review.

- Accept parameters allowing developers to selectively override certain parameters to easily explore the transaction traces of edge cases and failure modes.

The core function that generates traces is `debug_traceTransaction`, which provides the target transaction hash as the first and only parameter. An index was added to each EVM snapshot to make it easier to map opcode executions from the trace to locations in the source code. This function can be incorporated elsewhere in the codebase to help investigate EVM internals:

```
const traceTx = async (txHash, filename) => {
  await provider.send("debug_traceTransaction", [txHash]).then((res) => {
    if (filename) {
      const indexedRes = {
        ...res,
        structLogs: res.structLogs.map((structLog, index) => ({
          index,
          ...structLog,
        })),
      };
      fs.writeFileSync(filename, JSON.stringify(indexedRes, null, 2));
    } else {
      log(res);
    }
  });
};
```

Figure F.2: The utility for generating a transaction trace

G. System Invariants

Seaport

The Seaport system relies on various invariants regarding the fulfillment and validation of orders.

Validating Orders

- The status of a canceled order should be invalid and canceled.
- A call to the `validate` function on a restricted order should revert if the caller is not authorized to match the order.

Fulfilling Orders

- If the offerer (seller) of an order does not own all of the offer items, the fulfillment transaction should revert.
- If the buyer of an order does not own the consideration item(s), the fulfillment transaction should revert.
- For each order route type, the corresponding consideration item(s) should be sent to the buyer, and the offer item(s), to the seller.
- When a full order is fulfilled, all of its consideration items should be transferred.
- When a partial order is fulfilled, a portion of the consideration and offer items should be transferred, and the remaining items should be transferred in a separate transaction (or separate transactions).
- Items that have not been offered should not be transferred in a fulfillment transaction.

Conduit – OpenSea

- Unless the `execute` function reverts, it should return the correct function selector.
- A successful call to the `execute` function should result in a transfer of ERC20, ERC721, or ERC1155 tokens.
- An attempt to transfer more than one ERC721 item through a single conduit should cause the `execute` function to revert.
- The `execute` function should revert if it is called on a conduit that does not exist.

ConduitController – OpenSea

- The following functions should not revert when called on an existing conduit:

    - `getPotentialOwner`
    - `getChannelStatus`
    - `getTotalChannels`
    - `getChannel`
    - `getChannels`
    - `acceptOwnership`
    - `ownerOf`
    - `getKey`
    - `updateChannel`

- Only conduit owners should be able to update channels.
- A transfer of a conduit's ownership to `address(0)` should always revert.
- A transfer of a conduit's ownership to a valid address should always result in an update to `_conduits[conduit].potentialOwner`.
- The cancellation of a conduit-ownership transfer should cause`_conduits[conduit].potentialOwner` to be zeroed out.
- Only the prospective new owner of a conduit can call `acceptOwnership`.
- When a conduit's new owner accepts the ownership transfer, `_conduits[conduit].potentialOwner` should be zeroed out, and `_conduits[conduit].owner` should be set to the new owner's address.
- A call to the `getChannel` function to retrieve the number of a channel in a conduit should revert if the function is called with a channel index that exceeds the total number of channels in the `conduit`.
- The size of the `getTotalChannels` function's return value should always be equal to the length of the list returned by `getChannels`.

H. Incident Response Recommendations

In this section, we provide recommendations around the formulation of an incident response plan.

Identify who (either specific people or roles) is responsible for carrying out the mitigations (deploying smart contracts, pausing contracts, upgrading the front end, etc.).

- Specifying these roles will strengthen the incident response plan and ease the execution of mitigating actions when necessary.

Document internal processes for situations in which a deployed remediation does not work or introduces a new bug.

- Consider adding a fallback scenario that describes an action plan in the event of a failed remediation.

Clearly describe the intended process of contract deployment.

Consider whether and under what circumstances OpenSea will make affected users whole after certain issues occur.

- Some scenarios to consider include an individual or aggregate loss, a loss resulting from user error, a contract flaw, and a third-party contract flaw.

Document how OpenSea plans keep up to date on new issues, both to inform future development and to secure the deployment toolchain and the external on-chain and off-chain services that the system relies on.

- For each language and component, describe noteworthy sources for vulnerability news. Subscribe to updates for each source. Consider creating a special private Discord channel with a bot that will post the latest vulnerability news; this will help the team keep track of updates all in one place. Also consider assigning specific team members to keep track of the vulnerability news of a specific component of the system.

Consider scenarios involving issues that would indirectly affect the system.

Determine when and how the team would reach out to and onboard external parties (auditors, affected users, other protocol developers, etc.) during an incident.

- Some issues may require collaboration with external parties to efficiently remediate them.

Define contract behavior that is considered abnormal for off-chain monitoring.

- Consider adding more resilient solutions for detection and mitigation, especially in terms of specific alternate endpoints and queries for different data as well as status pages and support contacts for affected services.

Combine issues and determine whether new detection and mitigation scenarios are needed.

Perform periodic dry runs of specific scenarios in the incident response plan to find gaps and opportunities for improvement and to develop muscle memory.

- Document the intervals at which the team should perform dry runs of the various scenarios. For scenarios that are more likely to happen, perform dry runs more regularly. Create a template to be filled in after a dry run to describe the improvements that need to be made to the incident response.

I. 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.