# 0x Protocol

## Security Assessment

**October 4th, 2019**

Prepared For:
Amir Bandeali | *0x Protocol*
amir@0x.org

Prepared By:
Gustavo Grieco | *Trail of Bits*
gustavo.grieco@trailofbits.com

Michael Colburn | *Trail of Bits*
michael.colburn@trailofbits.com

Robert Tonic | *Trail of Bits*
robert.tonic@trailofbits.com

Rajeev Gopalakrishna | *Trail of Bits*
rajeev@trailofbits.com

Changelog:

| | |
|---|---|
| October 4, 2019: | Initial draft delivery |
| October 11, 2019: | Final report for publication |

## Executive Summary

From August 5th through August 9th, 2019, and September 9th through October 4th, 2019, 0x hired Trail of Bits to review the security of their smart contracts focused on their upcoming 3.0 version. This new version includes staking features, aligning market participants with the long-term mission, and objectives of 0x.

The project began with a week-long architecture review focused on the proposed 3.0 functionality and its potential impact on the protocol. The architecture review identified high-risk areas of the 3.0 protocol and areas where the codebase could be modified to better facilitate automated analysis. Several areas of the protocol—such as the extensive use of assembly in the Exchange contract, arithmetic in the `AssetProxyOwner` and `AssetProxy` contracts, and state transitions throughout the entire system—were identified for follow-up review due to either weak controls or insufficient time to fully investigate them. 0x suggested that the future review should focus on automated testing of the 0x protocol's core properties.

This security assessment was conducted over the course of eight person-weeks, with four engineers working from commit hash `abd479dc68fa75719647db261130418725fd40d5` and `d21f978deff1be9321837c0d202ff188d94cb28c` from the 0x repository.

During the first week of the security audit, we familiarized ourselves with the smart contracts used by the 0x system and performed an initial review of the Exchange contract. In the second week, we investigated the properties of orders and transactions and their signature verification. In the third week, we focused on investigating complex interactions and corner cases in the Exchange contract and also started to review the finalized version of the staking contracts. For the final week of the assessment, Trail of Bits concluded the automatic and manual reviews of the 0x contracts, which focused on the staking contracts.

Trail of Bits identified 23 issues, ranging from informational- to high-severity:

- A large number of the issues related to the order operations in the Exchange contract and how users are supposed to interact with these.
- Using the gas price to compute protocol fees enabled zero fee orders and gave makers a discounted price for front-running transactions.
- Issues arose from the MultiSig wallet implementation, including a lack of checks before calls and potential integer overflow in the confirmation counter.
- Issues related to the role of makers in a staking pool and how malicious makers could abuse their authority to add/remove other makers or decrease the operator's share of rewards.
- The remaining Issues were related to the ERC20 standard, missing events in critical operations, and potential race conditions in signature validations.

Throughout the audit, we developed automated analyses to help discover potential bugs in the code. A high-level description of our approach is available below, and tool-specific information is available in the appendices. We delivered all code used in the analyses along with this report to enable continuous analysis as development proceeds.

- Appendix C documents changes in Echidna and Manticore to support the 0x contracts and maximize the code they can review.
- Appendix D documents use of the Manticore symbolic executor to verify specific code performing complex arithmetic computations.
- Appendix E documents our suggested use of Echidna to fuzz during CI testing.

The overall quality of the codebase is good. The architecture is modular and avoids unnecessary complexity. Component interactions are well defined and properly documented. The functions are small and easy to understand.

However, this codebase frequently contains corner cases that are not properly defined in the specification and therefore could lead to security or correctness issues. These cases are difficult to test with traditional unit tests or state-of-the-art fuzzing and symbolic execution tools. For instance, the `Exchange` contract relies on orders that should be filled or canceled, but attempts to fill certain orders that appear valid may fail due to the violation of internal constraints that are unclear in the protocol specification and are not adequately exposed to the end user. Additionally, the MultiSig wallet contracts had correctness and security issues, and unclear specifications.

Trail of Bits recommends that 0x address the identified issues before deployment of the version 3.0 codebase to production. The complexity of the new staking features is significant, and they may contain additional issues. The staking contracts were not under review until the final week of the assessment due to their active development during the assessment and, therefore, we recommend further review of them.

## Engagement Goals & Scope

The goal of the engagement was to evaluate the security of the Exchange and the staking contracts. Specifically, we sought to answer the following questions:

- Can participants abuse the Exchange trust model?
- Can participants delay, block, or alter orders or transactions from other users?
- Can participants bypass signature verification of orders or transaction orders from other users?
- Is it possible to manipulate the Exchange by using specially crafted orders or transactions, or front-running transactions?
- Is it possible for participants to steal or lose tokens?
- Can participants perform Denial of Service or phishing attacks against any of the contracts?
- Can unauthorized users interact with or block staking pools?
- Can operators, makers, or delegates abuse staking or staking pools?
- Can participants manipulate protocol fees or rewards in an unauthorized manner?
- Can a potentially malicious owner remove or block other owners in the MultiSig wallet?
- Can non-owner users confirm, unconfirm, execute, or block transactions in the MultiSig wallet?

## Coverage

The engagement was focused on the following components:

- **Exchange and its libraries:** Exchange contains the main business logic within the 0x Protocol. It is the entry point for essential operations, such as filling and canceling orders, executing transactions, validating signatures, and a number of administrative operations regarding management of asset proxies. Exchange libraries contain the implementations of various libraries and utilities used within the Exchange contract.
  - `0x-monorepo/contracts/exchange`
  - `0x-monorepo/contracts/exchange-libs/`
- **Utils:** `Utils` contains smart contract utilities and libraries used throughout the entire codebase of the 0x smart contracts.
  - `0x-monorepo/contracts/utils`
- **`MultiSig`:** The `MultiSig` wallet contracts are used to control various contracts within the 0x protocol. There is one base contract, the `MultiSigWallet`, that is expanded upon for various uses. Most notably, the `AssetProxyOwner` extends the `MultiSigWalletWithTimeLock`, which extends the `MultiSigWallet`.
  - `0x-monorepo/contracts/multisig`
- **Staking:** The staking package implements the stake-based liquidity incentives. It is subdivided into several directories that implement a mix of features to perform various arithmetic operations, including vault management, fee computation, and several utils libraries. We did not receive the staking code until late in the engagement and therefore only had a limited amount of time to review it.
  - `0x-monorepo/contracts/staking`
- **Access controls.** Many parts of the system expose privileged functionality, such as setting protocol parameters or managing staking pools. We reviewed these functions to ensure they can only be triggered by the intended actors and that they do not contain unnecessary privileges that may be abused.
- **Arithmetic.** We reviewed calculations for logical consistency, as well as rounding issues and scenarios where reverts due to overflow may negatively impact use of the protocol.

Components outside the scope of this assessment were:

- `coordinator` contracts.
- `asset-proxy` contracts.
- `exchange-forwarder` contracts.
- `extensions`.
- `dev-utils` and `test-utils`.
- Off-chain code.

## Automated Testing and Verification

Trail of Bits used automated testing techniques to enhance coverage of certain areas of the protocol. We have developed three unique capabilities for testing smart contracts:

- Slither, a Solidity static analysis framework. Slither can statically verify algebraic relationships between Solidity variables. We used Slither to detect invalid or inconsistent usage of the contracts' APIs across the entire codebase.
- Echidna, a smart contract fuzzer. Echidna can rapidly test security properties via malicious, coverage-guided test case generation. We used Echidna to test the expected system properties of the Exchange contract and its libraries.
- Manticore, a symbolic execution framework. Manticore can exhaustively test security properties via symbolic execution. We used Manticore to verify that rounding errors cannot be used to avoid paying the taker or the corresponding fees (Appendix D).

Automated testing techniques augment our manual security review but do not replace it. Each technique has limitations: Slither may identify security properties that fail to hold when Solidity is compiled to EVM bytecode; Echidna may not randomly generate an edge case that violates a property; and Manticore may fail to complete its analysis. To mitigate these risks we generate 20,000 test cases per property with Echidna, run Manticore for a minimum of one hour, and then manually review all results.

We evaluated 135 security properties across 21 contracts. In the process, we formalized and tested a variety of properties, from high-level ones regarding orders and transactions in the core of the 0x protocol to very specific and low-level ones in basic libraries like `SafeMath` and `LibBytes`. As we detailed in issues TOB-0x-003, TOB-0x-015, and TOB-x0-017, some general properties of the Exchange orders were not simple to formalize. Defining low-level properties was generally easier, since the properties of integers or strings are universal.

Regarding property coverage, the core of the protocol, consisting of the Exchange and its libraries, received substantial coverage. The MultiSig wallet received a moderate amount of coverage, and the staking code has a minimal number of properties.

### Exchange Contract

Exchange contains the main business logic within the 0x Protocol. It is the entry point for essential operations such as filling and canceling orders, executing transactions, validating signatures, and a number of administrative operations regarding management of asset proxies.

We identified security properties for each contract used to implement Exchange. Each property listed is valid regardless of the state, initialized or not, of the Exchange contract.

**utils/contracts/src/Ownable.sol**

| Property | Approach | Result |
|---|---|---|
| Only the owner can transfer the ownership. | Echidna | **Passed** |
| If the owner calls `transferOwnership`, the owner must change. | Echidna | **Passed** |
| The owner cannot be transferred to the 0x0 address. | Echidna | **Passed** |

**exchange/contracts/src/MixinTransactions.sol**

| Property | Approach | Result |
|---|---|---|
| `ExecuteTransaction` will always revert when using random inputs. | Echidna | **Passed** |
| `batchExecuteTransactions` will always revert when using random inputs. | Echidna | **Passed** |
| `executeTransaction` and `batchExecuteTransactions` cannot be used to call any "administrative function." | Unit tests | **Passed** |
| The same transaction cannot be executed twice. | Unit tests | **Passed** |

**exchange/contracts/src/MixinTransferSimulator.sol**

| Property | Approach | Result |
|---|---|---|
| `simulateDispatchTransferFromCalls` will always revert with the error code “`TRANSFERS_SUCCESSFUL.`” | Echidna | **Passed** |

**exchange/contracts/src/MixinAssetProxyDispatcher.sol**

| Property | Approach | Result |
|---|---|---|
| Only the owner can register asset proxies using `registerAssetProxy`. | Echidna | **Passed** |
| If the owner registers a new asset proxy using `registerAssetProxy`, the call to `getAssetProxy` returns the proper value. | Echidna | **Passed** |
| The `0x0` address cannot be registered as an asset proxy. | Echidna | **Passed** |

|  |  |  |
|---|---|---|
| It is not possible to unregister or change asset proxies. | Echidna | **Passed** |

**exchange/contracts/src/MixinProtocolFees.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| Only the owner can update `protocolFeeMultiplier`. | Echidna | **Passed** |
| After the owner calls `setProtocolFeeMultiplier`, `protocolFeeMultiplier` must be updated. | Echidna | **Passed** |
| Only the owner can update `protocolFeeCollector`. | Echidna | **Passed** |
| After the owner calls `setProtocolFeeCollectorAddress`, `protocolFeeCollector` must be updated. | Echidna | **Passed** |

**exchange/contracts/src/MixinSignatureValidator.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| `preSign` is idempotent. | Unit Test | **Not covered** |
| `isValidHashSignature` will return false or revert when using random inputs. | Echidna | **Passed** |
| `isValidOrderSignature` will return false or revert when using random inputs. | Echidna | **Passed** |
| `isValidTransactionSignature` will return false or revert when using random inputs. | Echidna | **Passed** |

**exchange/contracts/src/MixinExchangeCore.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| Orders can only run using data that was signed. | Manual Code Review | **TOB-0x-009** |
| If an order can be filled, then it can be canceled. | Echidna | **Passed** |
| If an order cannot be filled, then it cannot be canceled. | Echidna | **Code Quality** |
| A valid order cannot be fully filled twice. | Echidna | **Passed** |

| | | |
|---|---|---|
| A filled order can be canceled immediately twice. | Echidna | **Passed** |
| A valid order can be partially filled with zero twice. | Echidna | **Passed** |
| If an order can be partially filled, then it should transfer the corresponding maker/taker/fees amounts. | Manual Code Review | **TOB-0x-015** |
| If an order can be partially filled with zero, then it can be partially filled with one token. | Echidna | **TOB-0x-017** |

**utils/contracts/src/MixinMatchOrders.sol**

| Property | Approach | Result |
|---|---|---|
| `matchOrders` should not revert if and only if<br>● `leftOrder` and `rightOrder` orders are valid and fillable.<br>● `leftOrder.makerAssetData == rightOrder.takerAssetData`<br>● `leftOrder.takerAssetData == rightOrder.makerAssetData`<br>● `(leftOrder.makerAssetAmount * rightOrder.makerAssetAmount) >= (leftOrder.takerAssetAmount * rightOrder.takerAssetAmount)` | Echidna | **Passed** |

## Exchange Libraries

Exchange libraries contain the implementations of various libraries and utilities used within the Exchange contract. They include two important libraries that define the data structures for orders and transactions, with a small set of basic operations.

**exchange-libs/contracts/src/LibOrder.sol**

| Property | Approach | Result |
|---|---|---|
| `getTypedDataHash` should never revert. | Echidna | **Passed** |
| `getStructHash` should never revert. | Echidna | **Passed** |
| If x1 and x2 are Orders then `getStructHash(x1) == getStructHash(x1) ⇔ x1 == x2`. | Echidna | **Passed** |

| | | |
|---|---|---|
| If `x1` and `x2` are `Orders` then `getTypedDataHash(x1,b1) == getTypedDataHash(x2,b2) ⇔ x1 == x2.` | Echidna | **Passed** |

**exchange-libs/contracts/src/LibZeroExTransaction.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| `getTypedDataHash` should never revert. | Echidna | **Passed** |
| `getStructHash` should never revert. | Echidna | **Passed** |
| If x and y are `ZeroExTransaction` then `getStructHash(x) == getStructHash(y) ⇔ x == y.` | Echidna | **Passed** |
| If x and y are `ZeroExTransaction` then `getTypedDataHash(x) == getTypedDataHash(y) ⇔ x == y` | Echidna | **Passed** |

**exchange-libs/contracts/src/LibMath.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| `safeGetPartialAmountFloor` never returns 0 when the parameters are non-zero and the numerator is less or equal than denominator. | Manticore | **Verified** |

## Utils Contracts

`Utils` contains smart contract utilities and libraries used throughout the entire codebase of the 0x smart contracts. We identified security properties for each contract or library regardless of how they are used in the codebase.

**utils/contracts/src/LibSafeMath.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| The correct sum is calculated when adding. | Echidna | **Passed** |
| Integer overflows are detected for addition. | Echidna | **Passed** |
| The correct difference is calculated for subtraction. | Echidna | **Passed** |
| Integer overflows are detected when subtracting. | Echidna | **Passed** |
| The correct product is calculated when multiplying. | Echidna | **Passed** |

| | | |
|---|---|---|
| Integer overflows are detected for multiplication. | Echidna | **Passed** |
| The correct quotient is calculated when dividing. | Echidna | **Passed** |
| Division by zero is detected. | Echidna | **Passed** |
| The result of `safeDiv` is less than or equal to its first argument. | Echidna | **Passed** |
| `max256` returns the greater argument. | Echidna | **Passed** |
| `min256` returns the lesser argument. | Echidna | **Passed** |
| `LibSafeMath` and `SafeMath` produce the same results. | Echidna | **Passed** |

**utils/contracts/src/SafeMath.sol**

| Property | Approach | Result |
|---|---|---|
| The correct sum is calculated when adding. | Echidna | **Passed** |
| Integer overflows are detected for addition. | Echidna | **Passed** |
| The correct difference is calculated for subtraction. | Echidna | **Passed** |
| Integer overflows are detected when subtracting. | Echidna | **Passed** |
| The correct product is calculated when multiplying. | Echidna | **Passed** |
| Integer overflows are detected for multiplication. | Echidna | **Passed** |
| The correct quotient is calculated when dividing. | Echidna | **Passed** |
| Division by zero is detected. | Echidna | **Passed** |
| The result of `safeDiv` is less than or equal to its first argument. | Echidna | **Passed** |
| `max256` returns the greater argument. | Echidna | **Passed** |
| `min256` returns the lesser argument. | Echidna | **Passed** |

**utils/contracts/src/LibFractions.sol**

| Property | Approach | Result |
|---|---|---|
| Addition of fractions is commutative. | Echidna | **Passed** |
| Adding zero to a fraction produces the same result. | Echidna | **Passed** |

**utils/contracts/src/LibAddressArray.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| `append` increases the length of the array by 1. | Echidna | **Passed** |
| The array returned by `append` contains the appended address in the last index. | Echidna | **Passed** |
| The index returned by `indexOf` corresponds to the appended address. | Echidna | **Passed** |
| The boolean value returned by `indexOf` is always true when you call it with the array returned by `append` and the appended address. | Echidna | **Passed** |
| The boolean value returned by `contains` is always true when you call it with the array returned by `append` and the appended address. | Echidna | **Passed** |

**utils/contracts/src/LibBytes.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| No sequence of operations over a list of bytes will corrupt its state. | Echidna | **Passed** |
| `equals(bs,bs)` returns true for all lists of bytes. | Echidna | **Passed** |
| `equals` never reverts. | Echidna | **Passed** |
| Calling `writeLength` with a larger length than the input list of bytes should increase the length of it. | Echidna | **Passed** |
| Calling `writeLength` with a smaller length than the input list of bytes should decrease the length of it. | Echidna | **Passed** |
| Decrementing and re-incrementing the length with `writeLength` should not change the original content of the list of bytes. | Echidna | **Passed** |
| Calling bs.`slice(0,bs.length)` returns a new list of bytes with the same content as bs, for all possible lists of bytes. | Echidna | **Passed** |
| Calling `bs.slice(0,0)` returns a new empty list of bytes for all possible lists of bytes. | Echidna | **Passed** |

| | | |
|---|---|---|
| Calling `bs.slice(index+1,index)` reverts for all possible lists of bytes. | Echidna | **Passed** |
| Calling `bs.slice` never changes the content of bs for all possible lists of bytes. | Echidna | **Passed** |
| Calling bs.`sliceDestructive(0,bs.length)` returns a list of bytes with the same content as bs, for all possible lists of bytes. | Echidna | **Passed** |
| Calling `bs.sliceDestructive(0,0)` returns an empty list of bytes for all possible lists of bytes. | Echidna | **Passed** |
| Calling `bs.sliceDestructive(index+1,index)` reverts for all possible lists of bytes. | Echidna | **Passed** |
| If the length of a list of bytes is greater or equal than 0, `PopLastByte` should return the last byte, reduce the length in 1, and preserve the content of the rest of the list. Otherwise, it should revert. | Echidna | **Passed** |
| If the length of a list of bytes is greater or equal than 20, `PopLast20Bytes` should return a list of the last 20 bytes, reduce the length in 20, and preserve the content of the rest of the list. Otherwise, it should revert. | Echidna | **Passed** |
| `readAddress` reads an `address` from a position in a list of bytes. It reverts if the list of bytes' length is less than 20 or if the position is greater than length - 20. | Echidna | **Passed** |
| `readAddress` always returns a valid address or reverts. | Echidna | **Passed** |
| `writeAddress` writes an address into a specific position in a list of bytes. It reverts if the list of bytes' length is less than 20. | Echidna | **Passed** |
| `writeAddress` always stores a valid address or reverts. | Echidna | **Passed** |
| `ReadUint256` reads an `uint256` from a position in a list of bytes. It reverts if the list of bytes' length is less than 32 or if the position is greater than length - 32. | Echidna | **Passed** |
| `writeUInt256` writes an `uint256` into a specific position in a list of bytes. It reverts if the list of bytes' length is less than 32. | Echidna | **Passed** |

| | | |
|---|---|---|
| `ReadBytes32` reads a `bytes32` from a position in a list of bytes. It reverts if the list of bytes' length is less than 32 or if the position is greater than length - 32. | Echidna | **Passed** |
| `writeBytes32` writes a `bytes32` into a specific position in a list of bytes. It reverts if the list of bytes' length is less than 32. | Echidna | **Passed** |
| `ReadBytes4` reads a `bytes4` from a position in a list of bytes. It reverts if the list of bytes' length is less than 4 or if the position is greater than length - 4. | Echidna | **Passed** |
| `readAddress` does not change the input list of bytes. | Echidna | **Passed** |
| `readUint256` does not change the input list of bytes. | Echidna | **Passed** |
| `readBytes32` does not change the input list of bytes. | Echidna | **Passed** |
| `readBytes4` does not change the input list of bytes. | Echidna | **Passed** |
| `writeAddress` does not change the length or the content of the rest of the input list of bytes. | Echidna | **Passed** |
| `writeUint256` does not change the length or the content of the rest of the input list of bytes. | Echidna | **Passed** |
| `writeBytes32` does not change the length or the content of the rest of the input list of bytes. | Echidna | **Passed** |
| `writeBytes4` does not change the length or the content of the rest of the input list of bytes. | Echidna | **Passed** |

**utils/contracts/src/Refundable.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| Refunds are disabled during function calls with the `disableRefundUntilEnd` modifier. | Echidna | **Passed** |
| Refunds retain their enabled or disabled status after calls to functions with the `disableRefundUntilEnd` modifier. | Echidna | **Passed** |

**utils/contracts/src/Ownable.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| Only the owner can transfer the ownership. | Echidna | **Passed** |

| | | |
|---|---|---|
| If the owner calls `transferOwnership`, the owner must change. | Echidna | **Passed** |
| The owner cannot be transferred to the `0x0` address. | Echidna | **Passed** |

## Staking Contracts

The staking package implements the stake-based liquidity incentives. It is subdivided into several directories that implement a mix of features to perform various arithmetic operations, including vault management, fee computation, and several utils libraries.

**staking/contracts/src/libs/LibCobbDouglas.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| The `cobbdouglas` function does not revert when valid input parameters are used. | Echidna | **TOB-0x-016** |
| The `cobbdouglas` function returns expected values when valid input parameters are used. | Custom Script | **Passed** |

## MultiSig Wallet Contracts

The `MultiSig` wallet contracts are used to control various contracts within the 0x protocol. There is one base contract, the `MultiSigWallet`, that is expanded upon for various uses. Most notably, the `AssetProxyOwner` extends the `MultiSigWalletWithTimeLock`, which extends the `MultiSigWallet`. Because of this, many of the properties are shared.

**multisig/contracts/src/MultiSigWallet.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| The required number of confirmations is always less or equal than the length of the list of owners. | Echidna | **Passed** |
| The required number of confirmations is always a positive number. | Echidna | **Passed** |
| The length of the list of owners is always a positive number. | Echidna | **Passed** |
| Changing the required number of confirmations to an invalid value reverts. | Echidna | **Passed** |
| An owner cannot remove another owner without consensus from the `required` number of owners. | Echidna | **Passed** |

| | | |
|---|---|---|
| An executed transaction cannot be executed again. | Unit Test | **Passed** |
| An unexecuted transaction can be executed. | Unit Test | **Passed** |
| Only the wallet contract can add an owner. | Echidna | **Passed** |
| Only the wallet contract can replace an owner. | Echidna | **Passed** |
| Adding the `0x0` address as owner will revert. | Echidna | **Passed** |
| Replacing any owner by the `0x0` address will revert. | Echidna | **Passed** |
| Only the wallet contract can remove an owner. | Echidna | **Passed** |
| Only the wallet contract can change the `required` number of confirmations. | Echidna | **Passed** |
| An owner can confirm a transaction. | Echidna | **Passed** |
| Non-owners cannot confirm transactions. | Echidna | **Passed** |
| An owner can revoke their confirmation of a transaction. | Echidna | **Passed** |
| Non-owners cannot revoke transactions. | Echidna | **Passed** |

**multisig/contracts/src/MultiSigWalletWithTimeLock.sol**

| **Property** | **Approach** | **Result** |
|---|---|---|
| All the properties from `MultiSig` Wallet should hold. | Echidna | **Passed** |
| Only the wallet can change the time lock. | Echidna | **Passed** |
| A transaction can be confirmed before the time lock. | Echidna | **Passed** |

# Project Dashboard

**Application Summary**

| | |
|---|---|
| Name | 0x Protocol |
| Type | Protocol |
| Platform | Solidity |

**Engagement Summary**

| | |
|---|---|
| Dates | September 9th to October 4th, 2019 |
| Method | Whitebox |
| Consultants Engaged | 4 |
| Level of Effort | 8 person-weeks |

**Vulnerability Summary**

| | | |
|---|---|---|
| Total High Severity Issues | 3 | ■ ■ ■ |
| Total Medium Severity Issues | 7 | ■ ■ ■ ■ ■ ■ ■ ■ |
| Total Low Severity Issues | 2 | ■ ■ |
| Total Informational Severity Issues | 11 | ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ |
| Total | 23 | |

**Category Breakdown**

| | | |
|---|---|---|
| Access Controls | 4 | ■ ■ ■ ■ |
| Data Validation | 11 | ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ■ |
| Denial of Service | 1 | ■ |
| Documentation | 1 | ■ |
| Auditing and Logging | 1 | ■ |
| Numerics | 1 | ■ |
| Timing | 2 | ■ ■ |
| Undefined Behavior | 2 | ■ ■ |
| Total | 23 | |

## Recommendations Summary

### Short Term

❑ **Evaluate the impact of protocol fees on nested operations and filter contracts** (TOB-0x-001).

❑ **Properly document the potential reduced cost of front-running for market makers to make sure users are aware of this risk** (TOB-0x-002). Establish a reasonable cap for the `protocolFeeMultiplier` to mitigate this issue.

❑ **Properly document the permanent effects of `cancelorderUpTo` on future orders to warn users of this behavior** (TOB-0x-003). Alternatively, disallow the cancellation of future orders. This will prevent users from locking themselves out of participating with the protocol.

❑ **Document the inherent risks of using validators** (TOB-0x-004) to ensure users are aware of them in case of a validator compromise.

❑ **Document the WETH9 contract's non-standard behavior** (TOB-0x-005) and verify that all code interfacing with it will not break due to this behavior.

❑ **Implement `NoThrow` variants for batch processing of transaction execution and order matching** (TOB-0x-006). This will mitigate the possibility of exchange griefing by a malicious order included in a batch of transactions.

❑ **Enforce a reasonable minimum value for the protocol fee for each order or transaction** (TOB-0x-007). This will prevent orders from bypassing the protocol fee when a transaction has a zero gas price (e.g., in the case of a miner taking orders).

❑ **Add events where appropriate for critical operations identified in** TOB-0x-008. This will make off-chain monitoring and auditing much simpler.

❑ **Properly validate the content and size of the `makerAssetData` and `takerAssetData` fields in orders** (TOB-0x-009, TOB-0x-010). This will prevent reaching potential edge cases and triggering undefined behavior.

❑ **Ensure all uses of `call` check the existence of a contract at the destination address** (TOB-0x-011). This will ensure that the return value of `call` correctly expresses whether the call ran to completion or not.

❏ **Ensure all wallets deriving from `MultiSigWallet` do not redefine `transactionId` to be shorter than a `uint256`** (TOB-0x-012). This will prevent the introduction of a potential overflow scenario.

❏ **Implement the necessary range checks to enforce the timelock described in the specification** (TOB-0x-013). Otherwise correct the specification to match the intended behavior.

❏ **Use `SafeMath` when performing calculations in the wallet contracts** (TOB-0x-014). This will prevent unexpected overflows from occurring.

❏ **Select a proper bound for the accumulated rounding error when calculating fill results** (TOB-0x-015). Add code to keep track of it for each order and disallow a partial fill if it increases beyond the bound. This will limit the magnitude of rounding errors introduced to order fill calculations.

❏ **Reduce the bound value for the Cobb-Douglas function's parameters and properly document the input constraints** (TOB-0x-016). Proper documentation of function constraints allows for more robust testing.

❏ **Define a proper procedure to determine if an order is fillable and document it in the protocol specification** (TOB-0x-017). If necessary, warn the user about potential constraints on the orders. This will improve the user experience in the event of valid but unfillable orders.

❏ **Ensure critical contract (e.g., `StakingProxy.sol` and `ZrxVault.sol`) owners are not EOAs** (TOB-0x-018), but are m-of-n `MultiSig` wallets where m >= 2, so that a single account cannot accidentally/maliciously trigger these extreme scenarios. This will help to mitigate the single point of failure in some staking contracts.

❏ **Correct the `transfer` and `transferFrom` in ERC20.sol to allow self-transfers** (TOB-0x-019) in each possible context if the balance is enough. This will ensure the ERC20 implementation adheres to the standard completely.

❏ **Add a return statement to `MixinStakingPool._assertStakingPoolExists` or remove the return type** (TOB-0x-020). Properly adjust the documentation, if needed. This will prevent potential confusion if another function checks the return value.

❏ **Add proper validation checks on all parameters in `MixinParams.setParams`** (TOB-0x-021). If the validation procedure is unclear or too complex to implement on-chain, document the potential issues that could produce invalid values.

❑ **Document the purpose of the non-operator maker role within a staking pool and caution operators against allowing third-party makers to join**. (TOB-0x-022, TOB-0x-023) This will greatly reduce the likelihood of operators adding makers that they do not control to the pool.

## Long Term

❑ **Consider using an alternative fee that does not depend on the `tx.gasprice`** (TOB-0x-001, TOB-0x-002, TOB-0x-007). This will allow the protocol to better mitigate front-running attacks and avoid giving miners or large market makers an economic advantage within the system.

❑ **Avoid designing user operations that have drastic effects on the post-conditions** (TOB-0x-003) (e.g., they cannot be reversed) without strong pre-conditions to prevent dangerous behavior. This will prevent users from accidentally performing operations they potentially do not want or expect.

❑ **Consider using a blockchain monitoring system to track `SignatureValidatorApproval` events to catch front-running attacks** (TOB-0x-004) and otherwise suspicious behavior in the 0x contracts (TOB-0x-008).

❑ **Use Echidna to review the ERC20 specification and verify your contracts meet the standard** (TOB-0x-005, TOB-0x-019). When interfacing with external ERC20 tokens, be wary of popular tokens that do not properly implement the standard (e.g., many tokens do not include return values for `approve`, `transfer`, `transferFrom`, etc.).

❑ **Take into consideration the effect of malicious inputs when implementing functions that perform a batch of operations** (TOB-0x-006). This will allow the 0x team to design mitigations into the protocol.

❑ **Avoid handling arbitrary encoded data without any proper checks** (TOB-0x-009). This will prevent malicious actors from triggering undefined behavior.

❑ **Review the usage of inline assembly to avoid reading uninitialized data** (TOB-0x-009). This will prevent malicious actors from triggering undefined behavior.

❑ **Review every field that is logged and make sure it is properly validated** (TOB-0x-010). This will prevent logging of malformed input and keep the logs cleaner.

❑ **Ensure the lack of contract existence check in `MultiSigWallet` is well documented and accounted for in any systems depending on this contract** (TOB-0x-011). This will help to prevent confusion around the success of low-level calls to external contracts.

❑ **Use `SafeMath` to avoid potential overflows** (TOB-0x-012). Overflows may trigger undefined behavior or move the system into an inconsistent state.

❑ **Make sure implementation and specification are in sync** (TOB-0x-013).

❑ **Ensure proper testing is applied to the wallet contracts** (TOB-0x-014). Vulnerabilities in these contracts could have far-reaching effects, especially on controlling aspects of the 0x exchange and staking systems.

❑ **Use Echidna or Manticore to:**

- ❑ Test for integer overflows (TOB-0x-012),
- ❑ Verify that code properly implements the specification (TOB-0x-013),
- ❑ Test for properties that could fail after a sequence of transactions (TOB-0x-015),
- ❑ Make sure the arithmetic computations return expected results and do not revert (TOB-0x-016),
- ❑ Test that `fillOrder` never reverts when the order is valid and is used to completely fill an order (TOB-0x-017), and
- ❑ Locate missing parameter validation checks (TOB-0x-021).

❑ **Use Slither to detect when functions are missing appropriate return statements** (TOB-0x-020). This will help to prevent logical errors when a caller expects a return value but only receives the default value.

❑ **Create a new modifier, `onlyStakingPoolOperator`, and use it to restrict calls to `decreaseStakingPoolOperatorShare` and `addMakerToStakingPool`** (TOB-0x-022, TOB-0x-023). This will greatly reduce the attack surface of staking pools and safeguard operators against third-party makers in the event they approve one and are not aware of the role's capabilities.

❑ **Fix the `_assertSenderIsPoolOperatorOrMaker` function to correctly check that `msg.sender` is the `makerAddress` passed to `removeMakerFromStakingPool`** (TOB-0x-023), not simply any maker from that pool. This will greatly reduce the attack surface of staking pools and safeguard operators against third-party makers in the event they approve one and are not aware of the role's capabilities.

## Findings Summary

| # | Title | Type | Severity |
|---|---|---|---|
| 1 | Fee refunds incentivize transaction centralization through market makers | Undefined Behavior | Low |
| 2 | Market makers have a reduced cost for performing front-running attacks | Timing | Medium |
| 3 | cancelOrdersUpTo can be used to permanently block future orders | Data Validation | High |
| 4 | setSignatureValidatorApproval race condition may be exploitable | Timing | Medium |
| 5. | WETH9 transferFrom often does not follow spec | Access Controls | Informational |
| 6. | Batch processing of transaction execution and order matching may lead to exchange griefing | Denial of Service | Medium |
| 7 | Zero fee orders are possible if a user performs transactions with a zero gas price | Data Validation | Medium |
| 8 | Lack of events for critical operations | Auditing and Logging | Informational |
| 9 | Lack of validation in the makerAssetData and takerAssetData leads to unexpected behavior | Undefined Behavior | Informational |
| 10 | Transfers with zero fee amounts can log arbitrary data in their feeAssetData | Data Validation | Informational |
| 11 | MultiSigWallet does not check contract existence before call | Data Validation | Medium |
| 12 | Potential overflow in transactionId allowing arbitrary execution of transactions by a malicious owner | Data Validation | Informational |

| | | | |
|---|---|---|---|
| 13 | Specification-Code mismatch for AssetProxyOwner timelock period | Documentation | High |
| 14 | Potential overflow in MultiSigWalletWithTimelock when calculating whether the timelock has passed | Data Validation | Low |
| 15 | Rounding division errors can accumulate over partial fills | Numerics | Informational |
| 16 | The Cobb–Douglas function is not properly documented and reverts with valid parameters | Data Validation | Medium |
| 17 | Unclear documentation on how order filling can fail | Data Validation | High |
| 18 | Potential single point of failure for "read-only-mode" and "catastrophic-failure-mode" | Access Controls | Informational |
| 19 | ERC20 reverts during certain self-transfer | Data Validation | Informational |
| 20 | _assertStakingPoolExists never returns true | Data Validation | Informational |
| 21 | Calls to setParams may set invalid values and produce unexpected behavior in the staking contracts | Data Validation | Medium |
| 22 | Malicious non-operator maker can decrease staking pool operator share | Access Controls | Informational |
| 23 | Non-operator makers can add or remove other makers | Access Controls | Informational |

## 1. Fee refunds incentivize transaction centralization through market makers

Severity: Low
Type: Undefined Behavior
Target: `0x Protocol fee structure`

Difficulty: Medium
Finding ID: TOB-0x-001

**Description**
According to the 3.0 specification, `ETH` or `WETH` may be used to pay a protocol fee. This protocol fee is required when executing `fillOrder`- and `matchOrder`-style functions. Given the introduction of fees when performing these operations, the context in which a transaction executes will impact the payment of a protocol fee.

> The protocol fee can be paid in either ETH or its WETH equivalent (denominated in wei). If it is not provided as value included in the message call, the Staking contract will attempt to transfer WETH from the taker's address to cover the fee instead. The Exchange contract assumes that the fee was correctly paid if the Staking contract's payProtocolFee function did not revert.

*Figure 1.1: The protocol fee payment process within the 3.0 spec.*

When invoking `executeTransaction` (Figure 1.2), an encoded transaction is provided as `msg.data`. Part of this encoded transaction includes a signature, and the owner of this signature is the context in which the encoded transaction will execute. This functionality makes the use of "filter contracts" easier, since the filter contracts and relays may execute a transaction on behalf of the user.

```
    /// @dev Executes an Exchange method call in the context of signer.
    /// @param transaction 0x transaction structure.
    /// @param signature Proof that transaction has been signed by signer.
    /// @return ABI encoded return data of the underlying Exchange function call.
    function _executeTransaction(
        LibZeroExTransaction.ZeroExTransaction memory transaction,
        bytes memory signature
    )
        internal
        returns (bytes memory)
    {
        bytes32 transactionHash = transaction.getTypedDataHash(EIP712_EXCHANGE_DOMAIN_HASH);

        _assertExecutableTransaction(
            transaction,
            signature,
            transactionHash
        );

        // Set the current transaction signer
        address signerAddress = transaction.signerAddress;
        _setCurrentContextAddressIfRequired(signerAddress, signerAddress);

        // Execute transaction
        transactionsExecuted[transactionHash] = true;
        (bool didSucceed, bytes memory returnData) =
address(this).delegatecall(transaction.data);
        if (!didSucceed) {
            LibRichErrors.rrevert(LibExchangeRichErrors.TransactionExecutionError(
                transactionHash,
                returnData
            ));
        }

        // Reset current transaction signer if it was previously updated
        _setCurrentContextAddressIfRequired(signerAddress, address(0));

        emit TransactionExecution(transactionHash);

        return returnData;
    }
```

*Figure 1.2: The definition of* `_executeTransaction`*, the only function called by* `executeTransaction`*.*

If we use `ETH` as an example, the `executeTransaction` function will pass execution to the function specified within the encoded `msg.data` through the use of `delegateCall`. This also passes `msg.value` to the specified function context. Because of this, the caller of `executeTransaction` must provide `ETH` during invocation. This fundamentally changes the context in which fees are paid, because now the maker and taker are no longer covering protocol fees; the `executeTransaction` caller is.

This functionality is, by itself, not worrisome. However, when compounded with the way market makers receive their protocol fees, this could encourage the practice of "transaction smuggling" (for lack of a better term). By providing a transaction for a market maker to execute through `executeTransaction`, this functionality allows market makers to receive a portion of the protocol fee, then offer it back to the provider of the transaction once fee pools have been disbursed. This effectively allows market makers to reduce the protocol interaction cost for users.

Overall, this could lead to protocol fee monopolization, since the prospect of paying a lower overall protocol fee incentivizes users to let market makers execute the users' transactions.

**Exploit Scenario**
Alice prepares an order to submit to the 0x exchange. Instead of submitting it herself and paying the full protocol fee, she encodes her transaction as a 0x transaction and gives it to Eve's contract (which records Alice's address and the amount of `ETH` sent) to execute. This approach allows Eve to front the `executeTransaction` cost, including protocol fees. At the end of the disbursement epoch, Eve refunds Alice a portion of the protocol fee received.

**Recommendation**
Short term, evaluate the impact of protocol fees on nested operations and filter contracts.

Long term, consider including protocol fees within an order's definition, not within Ethereum transaction semantics.

## 2. Market makers have a reduced cost for performing front-running attacks

Severity: Medium
Difficulty: High
Type: Timing
Finding ID: TOB-0x-002
Target: `0x Protocol 3.0 specification`

**Description**
The 0x Protocol 3.0 specification defines how protocol fees are calculated.

> The protocol fee can be calculated with `tx.gasprice *` protocolFeeMultiplier, where the protocolFeeMultiplier is an upgradable value meant to target a percentage of the gas used for filling a single order. The suggested initial value for the protocolFeeMultiplier is 150000, which is roughly equal to the average gas cost of filling a single order (thereby doubling the net average cost).

*Figure 2.1: The protocol fee definition as defined in the 3.0 specification.*

Market makers receive a portion of the protocol fee for each order filled, and the protocol fee is based on the transaction gas price. Therefore market makers are able to specify a higher gas price for a reduced overall transaction rate, using the refund they will receive upon disbursement of protocol fee pools.

**Exploit Scenario**
Eve is a market-maker maintaining a distribution pool. Alice submits a profitable transaction to Eve's market. Eve sees the unconfirmed transaction and realizes it will result in a lower overall asset price, and submits a transaction with a higher gas cost and protocol fee, front-running Alice's transaction to sell her asset before the price decreases and increasing her profit from the transaction. Because Eve is a market maker, she receives a portion of the protocol fee she paid to front run Alice's transaction, reducing the overall cost.

**Recommendation**
Short term, properly document this issue to make sure users are aware of this risk. Establish a reasonable cap for the `protocolFeeMultiplier` to mitigate this issue.

Long term, consider using an alternative fee that does not depend on the `tx.gasprice` to avoid reducing the cost of performing front-running attacks.

## 3. cancelOrdersUpTo can be used to permanently block future orders

Severity: High
Difficulty: High
Type: Data Validation
Finding ID: TOB-0x-003
Target: `exchange/contracts/src/MixinExchangeCore.sol`

**Description**

Users can cancel an arbitrary number of future orders, and this operation is not reversible.

The `cancelOrdersUpTo` function (Figure 3.1) can cancel an arbitrary number of orders in a single, fixed-size transaction. This function uses a parameter to discard any order with salt less than the input value. However, `cancelOrdersUpTo` can cancel future orders if it is called with a very large value (e.g., `MAX_UINT256 - 1`). This operation will cancel future orders, except for the one with salt equal to `MAX_UINT256`.

```
    function cancelOrdersUpTo(uint256 targetOrderEpoch)
        external
        payable
        nonReentrant
        refundFinalBalance
    {
        address makerAddress = _getCurrentContextAddress();
        // If this function is called via `executeTransaction`, we only update the orderEpoch
for the makerAddress/msg.sender combination.
        // This allows external filter contracts to add rules to how orders are cancelled via
this function.
        address orderSenderAddress = makerAddress == msg.sender ? address(0) : msg.sender;


        // orderEpoch is initialized to 0, so to cancelUpTo we need salt + 1
        uint256 newOrderEpoch = targetOrderEpoch + 1;
        uint256 oldOrderEpoch = orderEpoch[makerAddress][orderSenderAddress];


        // Ensure orderEpoch is monotonically increasing
        if (newOrderEpoch <= oldOrderEpoch) {
            LibRichErrors.rrevert(LibExchangeRichErrors.OrderEpochError(
                makerAddress,
                orderSenderAddress,
                oldOrderEpoch
            ));
        }
        // Update orderEpoch
        orderEpoch[makerAddress][orderSenderAddress] = newOrderEpoch;
        emit CancelUpTo(
            makerAddress,
            orderSenderAddress,
            newOrderEpoch
        );
    }
```
*Figure 3.1: The* `cancelOrdersUpTo` *function.*

**Exploit Scenario**
Alice implements an automatic approach to fill and cancel orders. However, a mistake in the code causes a call to `cancelOrdersUpTo` with a very large value. As a result, all of Alice's orders are canceled, and there is no way to reverse this operation. Alice is forced to use another address for her orders.

**Recommendation**
Short term, properly document this behavior to warn users about the permanent effects of `cancelOrderUpTo` on future orders. Alternatively, disallow the cancelation of future orders.

Long term, avoid designing user operations that have drastic effects on the post-conditions (e.g., they cannot be reversed) without strong pre-conditions to prevent dangerous behavior. This will prevent users from accidentally performing operations they potentially do not want or expect.

## 4. setSignatureValidatorApproval race condition may be exploitable

Severity: Medium
Difficulty: High
Type: Timing
Finding ID: TOB-0x-004
Target: `exchange/contracts/src/MixinSignatureValidator.sol`

**Description**
If a validator is compromised, a race condition in the signature validator approval logic becomes exploitable.

The `setSignatureValidatorApproval` function (Figure 4.1) allows users to delegate the signature validation to a contract. However, if the validator is compromised, a race condition in this function could allow an attacker to validate any amount of malicious transactions.

```
function setSignatureValidatorApproval(
    address validatorAddress,
    bool approval
)
    external
    payable
    nonReentrant
    refundFinalBalance
{
    address signerAddress = _getCurrentContextAddress();
    allowedValidators[signerAddress][validatorAddress] = approval;
    emit SignatureValidatorApproval(
        signerAddress,
        validatorAddress,
        approval
    );
}
```

*Figure 4.1: The* `setSignatureValidatorApproval` *function.*

**Exploit Scenario**

1. Alice calls `setSignatureValidatorApproval(BobContract, True)`. This allows Bob's contract to validate Alice's signatures.
2. An attacker compromises Bob's contract, so Alice removes the approval calling `setSignatureValidatorApproval(Bob, False).`
3. The attacker sees Alice's unconfirmed approval removal and validates a number of malicious transactions or orders before Alice's transaction is mined.
4. If the attacker's transactions are mined before Alice's, the malicious transactions or orders can be executed.

**Recommendation**
Short term, document this behavior to make sure users are aware of the inherent risks of using validators in case of a compromise.

Long term, consider monitoring the blockchain using the `SignatureValidatorApproval` events to catch front-running attacks.

## 5. WETH9 transferFrom often does not follow spec

Severity: Informational
Difficulty: Low
Type: Access Controls
Finding ID: TOB-0x-005
Target: `erc20/contracts/src/WETH9.sol`

**Description**
If the message sender is the source of a `transferFrom` call, the sender's allowance will not be considered, and the transfer will initiate immediately. This breaks invariants expected of `transferFrom`.

Traditionally, the `transferFrom` method moves tokens from one account to another, provided the source account has approved the sender to send such an amount using the ERC20 method `approve`. However, the `transferFrom` function in WETH9's ERC20 token does not require approval if the sender is the source of the account:

```
    function transferFrom(address src, address dst, uint wad)
        public
        returns (bool)
    {
        require(balanceOf[src] >= wad);


        if (src != msg.sender && allowance[src][msg.sender] != uint(-1)) {
            require(allowance[src][msg.sender] >= wad);
            allowance[src][msg.sender] -= wad;
        }


        balanceOf[src] -= wad;
        balanceOf[dst] += wad;


        Transfer(src, dst, wad);


        return true;
    }
}
```
*Figure 5.1: The `transferFrom` function.*

Although it may seem intuitive to allow the owner of the account balance to transfer funds without approval, external tooling may rely on invariants which are now broken.

**Exploit Scenario**
Alice sends a transaction that invokes `transferFrom` with her own address as the source address, assuming it will fail if no approval was set beforehand. Instead, the transfer succeeds, and Alice's funds are lost.

**Recommendation**
Short term, document this contract's non-standard behavior and verify that all code interfacing with it does not break due to this behavior.

Long term, use Echidna to review the ERC20 specification and verify your contracts meet the standard. When interfacing with external ERC20 tokens, be wary of popular tokens that do not properly implement the standard (e.g., many tokens do not include return values for `approve`, `transfer`, `transferFrom`, etc.).

## 6. Batch processing of transaction execution and order matching may lead to exchange griefing

Severity: Medium	Difficulty: Low
Type: Denial of Service	Finding ID: TOB-0x-006
Target: `exchange/contracts/src/{MixinTransactions, MixinMatchOrders}.sol`

**Description**
Batch processing of transaction execution and order matching will iteratively process every transaction and order, which all involve filling. If the asset being filled does not have enough allowance, the asset's `transferFrom` will fail, causing `AssetProxyDispatcher` to revert.

```
function _dispatchTransferFrom(
    bytes32 orderHash,
    bytes memory assetData,
    address from,
    address to,
    uint256 amount
)
    internal
{
       ...
        // Call the asset proxy's transferFrom function with the constructed calldata.
        (bool didSucceed, bytes memory returnData) = assetProxy.call(proxyCalldata);

        // If the transaction did not succeed, revert with the returned data.
        if (!didSucceed) {
            LibRichErrors.rrevert(LibExchangeRichErrors.AssetProxyTransferError(
                orderHash,
                assetData,
                returnData
            ));
        }
       ...
}
```

*Figure 6.1: The `_dispatchTransferFrom` function.*

`NoThrow` variants of batch processing, which are available for filling orders, are not available for transaction execution and order matching. So if one transaction or order fails this way, the entire batch will revert and will have to be re-submitted after the reverting transaction is removed.

**Exploit Scenario**
An attacker submits a valid order to match which gets bundled into a batch after any validation by relayers, but then front-runs its processing by reducing the allowance below the required value. This causes the malicious order to fail and revert the entire batch of matching orders, resulting in exchange griefing that leads to delays and loss of fees to makers.

**Recommendation**
Short term, implement `NoThrow` variants for batch processing of transaction execution and order matching.

Long term, take into consideration the effect of malicious inputs when implementing functions that perform a batch of operations.

## 7. Zero fee orders are possible if a user performs transactions with a zero gas price

Severity: Medium
Difficulty: High
Type: Data Validation
Finding ID: TOB-0x-007
Target: `exchange-libs/contracts/src/LibFillResults.sol`

**Description**
Users can submit valid orders and avoid paying fees if they use a zero gas price.

The computation of fees for each transaction is performed in the `calculateFillResults` function. It uses the gas price selected by the user and the `protocolFeeMultiplier` coefficient:

```
function calculateFillResults(
    LibOrder.Order memory order,
    uint256 takerAssetFilledAmount,
    uint256 protocolFeeMultiplier,
    uint256 gasPrice
)
    internal
    pure
    returns (FillResults memory fillResults)
{
    // Compute proportional transfer amounts
    fillResults.takerAssetFilledAmount = takerAssetFilledAmount;
    fillResults.makerAssetFilledAmount = LibMath.safeGetPartialAmountFloor(
        takerAssetFilledAmount,
        order.takerAssetAmount,
        order.makerAssetAmount
    );
    fillResults.makerFeePaid = LibMath.safeGetPartialAmountFloor(
        takerAssetFilledAmount,
        order.takerAssetAmount,
        order.makerFee
    );
    fillResults.takerFeePaid = LibMath.safeGetPartialAmountFloor(
        takerAssetFilledAmount,
        order.takerAssetAmount,
        order.takerFee
    );
    // Compute the protocol fee that should be paid for a single fill.
    fillResults.protocolFeePaid = gasPrice.safeMul(protocolFeeMultiplier);


    return fillResults;
}
```

*Figure 7.1: The* `calculateFillResults` *function.*

Since the user completely controls the gas price of their transaction and the price could even be zero, the user could feasibly avoid paying fees.

**Exploit Scenario**

The Exchange governance decides to significantly increase `protocolFeeMultiplier` to force the collection of higher fees. Alice does not want to pay increased fees, so she decides to submit her transactions with a gas price equal to zero and process her own transactions as a miner. As a result, she is able to bypass protocol fee collection.

**Recommendation**

Short term, select a reasonable minimum value for the protocol fee for each order or transaction.

Long term, consider not depending on the gas price for the computation of protocol fees. This will avoid giving miners an economic advantage in the system.

## 8. Lack of events for critical operations

Severity: Informational
Difficulty: Low
Type: Auditing and Logging
Finding ID: TOB-0x-008
Target: `Exchange contracts`

**Description**
Several critical operations do not trigger events, which will make it difficult to review the correct behavior of the contracts once deployed.

Critical operations that would benefit from triggering events include:

- Order matching (e.g., `matchOrders`, `batchMatchOrders`)
- Signature validation (e.g., `preSign`, `isValidOrderSignature`, `isValidTransactionSignature`)
- Wrapper functions (e.g., `fillOrKillOrder`, `marketSellOrdersNoThrow`, `marketBuyOrdersNoThrow`)
- Owner operations (e.g., `transferOwnership`)

Users and blockchain monitoring systems will not be able to easily detect suspicious behaviors without events.

**Exploit Scenario**
An attacker compromises the Exchange owner and transfers the ownership to a different address. Since there are no events associated with this critical operation, nobody notices this attack.

**Recommendation**
Short term, add events where appropriate for all critical operations.

Long term, consider using a blockchain monitoring system to track any suspicious behavior in the contracts.

## 9. Lack of validation in the makerAssetData and takerAssetData leads to unexpected behavior

Severity: Informational Difficulty: Low
Type: Undefined Behavior Finding ID: TOB-0x-009
Target: `exchange/contracts/src/MixinAssetProxyDispatcher.sol`, `asset-proxy/contracts/src/ERC20Proxy.sol`

**Description**
The lack of validation in two order fields may cause unexpected results in certain corner cases, which could confuse users or allow an attacker to bring the contract into an invalid state.

Orders contain two important variable-length size fields called `makerAssetData` and `takerAssetData`. In valid orders, these fields should contain the identifier of the asset proxy and the encoded address of the token contract to call `transferFrom`. However, these two fields have no proper validation until they are used in the call to the asset proxy. `_dispatchTransferFrom` validates that the `assetData` parameter is longer than 3 bytes and uses the first 4 bytes to look for the corresponding asset proxy.

```
    function _dispatchTransferFrom(
        bytes32 orderHash,
        bytes memory assetData,
        address from,
        address to,
        uint256 amount
    )
        internal
    {
        // Do nothing if no amount should be transferred.
        if (amount > 0) {
            // Ensure assetData length is valid
            if (assetData.length <= 3) {
                LibRichErrors.rrevert(LibExchangeRichErrors.AssetProxyDispatchError(

 LibExchangeRichErrors.AssetProxyDispatchErrorCodes.INVALID_ASSET_DATA_LENGTH,
                    orderHash,
                    assetData
                ));
            }
            // Lookup assetProxy.
            bytes4 assetProxyId = assetData.readBytes4(0);
            address assetProxy = _assetProxies[assetProxyId];


            // Ensure that assetProxy exists
            if (assetProxy == address(0)) {
                LibRichErrors.rrevert(LibExchangeRichErrors.AssetProxyDispatchError(
                    LibExchangeRichErrors.AssetProxyDispatchErrorCodes.UNKNOWN_ASSET_PROXY,
                    orderHash,
                    assetData
                ));
            }

            …
        }
    }
```

*Figure 9.1: Header of the* `_dispatchTransferFrom` *function.*

It is worth mentioning that the remainder of the data is not validated in any way in the Exchange contract. Finally, the data for the asset proxy will be encoded and the call will be executed:

```
function _dispatchTransferFrom(
    bytes32 orderHash,
    bytes memory assetData,
    address from,
    address to,
    uint256 amount
)
    internal
{

        …

        // Construct the calldata for the transferFrom call.
        bytes memory proxyCalldata = abi.encodeWithSelector(
            IAssetProxy(address(0)).transferFrom.selector,
            assetData,
            from,
            to,
            amount
        );


        // Call the asset proxy's transferFrom function with the constructed calldata.
        (bool didSucceed, bytes memory returnData) = assetProxy.call(proxyCalldata);


        // If the transaction did not succeed, revert with the returned data.
        if (!didSucceed) {
            LibRichErrors.rrevert(LibExchangeRichErrors.AssetProxyTransferError(
                orderHash,
                assetData,
                returnData
            ));
        }
    }
}
```

*Figure 9.2: Tail of the* `_dispatchTransferFrom` *function.*

It is expected that the asset proxy validates the data, but the inline assembly code reads what is in memory in the position where the `assetData` is assumed to be:

```
/////// Read token address from calldata ///////
// * The token address is stored in `assetData`.
//
// * The "offset to assetData" is stored at offset 4 in the calldata (table 1).
//   [assetDataOffsetFromParams = calldataload(4)]
//
// * Notes that the "offset to assetData" is relative to the "Params" area of calldata;
//   add 4 bytes to account for the length of the "Header" area (table 1).
//   [assetDataOffsetFromHeader = assetDataOffsetFromParams + 4]
//
// * The "token address" is offset 32+4=36 bytes into "assetData" (tables 1 & 2).
//   [tokenOffset = assetDataOffsetFromHeader + 36 = calldataload(4) + 4 + 36]
let token := calldataload(add(calldataload(4), 40))
```

*Figure 9.3: Part of the* `ERC20Proxy` *contract.*

**Exploit Scenario**

An attacker or a user could submit a valid order where the `makerAssetData` and `takerAssetData` fields are not properly encoded and are shorter than expected. The lack of checks will cause the asset proxy to read uninitialized memory. This uninitialized data was not signed, so should not be used by the asset proxy in any way. However, in certain cases, the transaction will unexpectedly succeed when it should certainly fail (e.g., when the token address ends with zeros, such as this high-profile ERC20 token). As a result of that, a `Fill` event with incorrect data will be emitted and some component of the 0x Exchange could transition into an invalid state.

**Recommendation**
Short term, properly validate the content and size of the `makerAssetData` and `takerAssetData` fields.

Long term:
- Avoid handling arbitrary encoded data without any proper checks.
- Review the usage of inline assembly to avoid reading uninitialized data.

## 10. Transfers with zero fee amounts can log arbitrary data in their feeAssetData

Severity: Informational
Difficulty: Low
Type: Data Validation
Finding ID: TOB-0x-010
Target: `exchange/contracts/src/{MixinExchangeCore, MixinAssetProxyDispatcher}.sol`

**Description**

When an order is filled, there is no validation performed on the `makerFeeAssetData` or `takerFeeAssetData` if the `makerFee` or `takerFee`, respectively, is equal to 0. This allows a user to insert arbitrary data into these fields, which will be emitted as part of a `Fill` event.

When `fillOrder` is invoked, the actual asset transfer is carried out by four separate calls to `_dispatchTransferFrom`, one each for the `makerAsset`, `takerAsset`, `makerFee`, and `takerFee` transfers. Inside this function, validation of the `assetData` parameter is performed. However, in the event that the call's respective asset amount is 0, this function's entire body, including validation, is skipped entirely.

```
   function _dispatchTransferFrom(
        bytes32 orderHash,
        bytes memory assetData,
        address from,
        address to,
        uint256 amount
    )
        internal
    {
        // Do nothing if no amount should be transferred.
        if (amount > 0) {
            // Ensure assetData length is valid
            if (assetData.length <= 3) {
                LibRichErrors.rrevert(LibExchangeRichErrors.AssetProxyDispatchError(

LibExchangeRichErrors.AssetProxyDispatchErrorCodes.INVALID_ASSET_DATA_LENGTH,
                    orderHash,
                    assetData
                ));
            }
```

*Figure 10.1:* `_dispatchTransferFrom` *function signature and input validation.*

Even if no `makerFee` (or `takerFee`) transfer occurs, the corresponding event emitted to log this call to `fillOrder` includes the unvalidated `assetData`. Note that this issue only applies to the maker or taker fees, and not the order amounts themselves, as orders with a 0 maker or taker amount are explicitly flagged as invalid.

```
        emit Fill(
            order.makerAddress,
            order.feeRecipientAddress,
            order.makerAssetData,
            order.takerAssetData,
            order.makerFeeAssetData,
            order.takerFeeAssetData,
            orderHash,
            takerAddress,
            msg.sender,
            fillResults.makerAssetFilledAmount,
            fillResults.takerAssetFilledAmount,
            fillResults.makerFeePaid,
            fillResults.takerFeePaid,
            fillResults.protocolFeePaid
        );
```

*Figure 10.2: The `Fill` event emitted as part of a successful call to `fillOrder`.*

**Exploit Scenario**
Eve submits an order to the 0x Exchange with the `makerFee` set to 0 and the `makerAssetData` set to arbitrary data. When her order is filled, an event is emitted recording the erroneous `makerAssetData` value. This may have unintended side effects on systems that expect the `makerAssetData` to adhere to a particular format.

**Recommendation**
Short term, perform basic validation of the asset data regardless of the amount being transferred.

Long term, review every every field that is logged and make sure it is properly validated.

## 11. MultiSigWallet does not check contract existence before call

Severity: Medium
Difficulty: Low
Type: Data Validation
Finding ID: TOB-0x-011
Target: `multisig/contracts/src/MultiSigWallet.sol`

**Description**
Within the `MultiSigWallet` contract, the `_externalCall` function is used to perform calls to an external contract address. However, there is no check to ensure `destination` is a contract. As a result, if the address provided is not a contract address, it will implicitly return `true`.

```
    // call has been separated into its own function in order to take advantage
    // of the Solidity's code generator to produce a loop that copies tx.data into memory.
    function _externalCall(address destination, uint value, uint dataLength, bytes data)
internal returns (bool) {
        bool result;
        assembly {
            let x := mload(0x40)   // "Allocate" memory for output (0x40 is where "free
memory" pointer is stored by convention)
            let d := add(data, 32) // First 32 bytes are the padded length of data, so
exclude that
            result := call(
                sub(gas, 34710),   // 34710 is the value that solidity is currently emitting
                                   // It includes callGas (700) + callVeryLow (3, to pay for
SUB) + callValueTransferGas (9000) +
                                   // callNewAccountGas (25000, in case the destination
address does not exist and needs creating)
                destination,
                value,
                d,
                dataLength,        // Size of the input (in bytes) - this is what fixes the
padding problem
                x,
                0                  // Output is ignored, therefore the output size is zero
            )
        }
        return result;
    }
```

*Figure 11.1: The `_externalCall` function definition.*

Furthermore, the `AssetProxyOwner` derives from the `MultiSigWalletWithTimelock`, but overloads the `executeTransaction` function (Figure 11.2), instead using `address(...).call`. Even in this case, the `destination` is not validated as a contract.

```
   function executeTransaction(uint256 transactionId)
        public
        notExecuted(transactionId)
        fullyConfirmed(transactionId)
    {
...
```

```
        uint256 transactionConfirmationTime = confirmationTimes[transactionId];
        for (uint i = 0; i != length; i++) {
            // Ensure that each function call is past its timelock
            _assertValidFunctionCall(
                transactionConfirmationTime,
                data[i],
                destinations[i]
            );
            // Call each function
            // solhint-disable-next-line avoid-call-value
            (bool didSucceed,) = destinations[i].call.value(values[i])(data[i]);
            // Ensure that function call was successful
            require(
                didSucceed,
                "FAILED_EXECUTION"
            );
...
    }
```

*Figure 11.2: A snippet of the* `executeTransaction` *function definition.*

**Exploit Scenario**
Ailce uses the `MultiSigWallet` wallet to submit a call to an address believed to be a contract. Unbeknownst to Alice, the contract has been destroyed. Due to a lack of contract existence checks in the `MultiSigWallet`, Alice's call returns a success even though it did not successfully execute.

**Recommendation**
Short term, ensure all uses of `call` check the existence of a contract at the destination address.

Long term, ensure this limitation is well documented and accounted for in any systems depending on the `MultiSigWallet`.

## 12. Potential overflow in transactionId allowing arbitrary execution of transactions by a malicious owner

Severity: Informational
Difficulty: High
Type: Data Validation
Finding ID: TOB-0x-012
Target: `multisig/contracts/src/MultiSigWallet.sol`

**Description**

The `MultiSigWallet` does not use `SafeMath`, resulting in the potential for overflow and underflow of numeric values. This allows a malicious owner to overflow `transactionId` through the use of `submitTransaction`, rewrite an existing transaction entry, increase the now-overwritten transaction's confirmation, and potentially execute the overwritten transaction.

When `submitTransaction` is executed (Figure 12.1), a `transactionId` is generated through the invocation of `_addTransaction`.

```
   /// @dev Allows an owner to submit and confirm a transaction.
    /// @param destination Transaction target address.
    /// @param value Transaction ether value.
    /// @param data Transaction data payload.
    /// @return Returns transaction ID.
    function submitTransaction(address destination, uint value, bytes data)
        public
        returns (uint transactionId)
    {
        transactionId = _addTransaction(destination, value, data);
        confirmTransaction(transactionId);
    }
```

*Figure 12.1: The* `submitTransaction` *function definition.*

The `_addTransaction` function (Figure 12.2) creates a new entry within the `transactions` mapping with the `transactionCount` as the ID. Subsequently, the `transactionCount` is incremented. Since `SafeMath` is not used, the `transactionCount` can be overflowed into an existing `transactionId` through repeated calling of `submitTransaction`.

```
   /// @dev Adds a new transaction to the transaction mapping, if transaction does not exist
yet.
    /// @param destination Transaction target address.
    /// @param value Transaction ether value.
    /// @param data Transaction data payload.
    /// @return Returns transaction ID.
    function _addTransaction(address destination, uint value, bytes data)
        internal
        notNull(destination)
        returns (uint transactionId)
    {
        transactionId = transactionCount;
        transactions[transactionId] = Transaction({
            destination: destination,
            value: value,
            data: data,
            executed: false
        });
        transactionCount += 1;
        Submission(transactionId);
    }
```
*Figure 12.2: The* `_addTransaction` *function definition, highlighting the overwriting of a transaction with a particular* `transactionId`*.*

After the `_addTransaction` function returns the `transactionId`, the `transactionId` is then passed to the `confirmTransaction` function (Figure 12.3), automatically adding the `msg.sender` to the mapping of confirmations, and subsequently attempting to execute the transaction.

```
   /// @dev Allows an owner to confirm a transaction.
    /// @param transactionId Transaction ID.
    function confirmTransaction(uint transactionId)
        public
        ownerExists(msg.sender)
        transactionExists(transactionId)
        notConfirmed(transactionId, msg.sender)
    {
        confirmations[transactionId][msg.sender] = true;
        Confirmation(msg.sender, transactionId);
        executeTransaction(transactionId);
    }
```
*Figure 12.3: The* `confirmTransaction` *function definition.*

If the transaction is now appropriately confirmed (which, assuming the original transaction was, the newly replaced transaction is), the `executeTransaction` function (Figure 12.4) will execute the transaction.

```
   /// @dev Allows anyone to execute a confirmed transaction.
    /// @param transactionId Transaction ID.
    function executeTransaction(uint transactionId)
        public
        ownerExists(msg.sender)
        confirmed(transactionId, msg.sender)
        notExecuted(transactionId)
    {
        if (isConfirmed(transactionId)) {
            Transaction storage txn = transactions[transactionId];
            txn.executed = true;
            if (_externalCall(txn.destination, txn.value, txn.data.length, txn.data))
                Execution(transactionId);
            else {
                ExecutionFailure(transactionId);
                txn.executed = false;
            }
        }
    }
```

*Figure 12.4: The `executeTransaction` function definition, highlighting the `isConfirmed` check, which dynamically calculates confirmations relative to the current contract owners.*

```
    /// @dev Returns the confirmation status of a transaction.
     /// @param transactionId Transaction ID.
     /// @return Confirmation status.
     function isConfirmed(uint transactionId)
         public
         constant
         returns (bool)
     {
         uint count = 0;
         for (uint i=0; i<owners.length; i++) {
             if (confirmations[transactionId][owners[i]])
                 count += 1;
             if (count == required)
                 return true;
         }
     }
```

*Figure 12.5: The `isConfirmed` function definition, highlighting dynamic calculation of confirmation counts.*

**Exploit Scenario**
Alice, Bob, and Eve are owners of the `MultiSigWallet`, and two confirmations are required, meaning two out of three owners must agree to submit any given transaction. Eve is a malicious owner.

1. Alice submits a transaction with a `transactionId` of `1`.
2. Eve performs `UINT256_MAX + 2` invocations of submitTransaction, which allows Eve's submitted transaction to overwrite Alice's transactionId of `1`.
3. Eve's confirmation of the transaction is applied (happens implicitly upon submission), and subsequently the transaction is executed.

If Eve's submitted transaction executes `changeRequirement` with an argument of `1`, the other owners are no longer required to confirm a transaction. Eve can then submit arbitrary transactions for execution without waiting for transaction confirmation by Alice and Bob, such as those protected by the `onlyWallet` modifier. This can potentially be performed in a single transaction if Eve's address is a contract address.

While newly submitted transactions are the easiest exploitation of this problem, under certain circumstances previously executed transactions can be overwritten and re-executed. To expand on the previous example: If a transaction was previously executed and the transaction was authorized by Alice and Charlie, but Charlie has since been removed from the owner array, the exploit can still be performed by Eve.

Two aspects of the contract's execution allow this:

1. The dynamic calculation of `isConfirmed` no longer considers Charlie an owner and disregards his confirmation, so the transaction is no longer confirmed until one more owner confirms the transaction. Upon successful overflow of the previously confirmed and executed transaction, Eve applies her confirmation, allowing confirmation to meet the requirement again (2).
2. The previous transaction is overwritten with one where executed field is false. Thus, the `executeTransaction` modifier notExecuted allows execution.

Due to the use of the `uint256` type for the `transactionId`, this overflow is not realistically exploitable with current execution constraints (Figure 12.6). However, if the `MultiSigWallet` is modified to change the type of `transactionId` to a shorter-width integer, exploitability may become easier.

```
>>>(2**256+1)/1000000/60/60/24/365
3671743063080802746815416825491118336290905145409708398004109081L
```

*Figure 12.6: Calculations for the number of years required to exploit this functionality with a* `uint256 transactionId`*, performing 1,000,000 transactions per second.*

**Recommendation**
Short term, ensure all wallets deriving from the `MultiSigWallet` do not redefine `transactionId` to be shorter than a `uint256`.

Long term, use `SafeMath` to avoid potential overflows. Properly test for integer overflows using Echidna or Manticore.

## 13. Specification-Code mismatch for AssetProxyOwner timelock period

Severity: High
Difficulty: Low
Type: Documentation
Finding ID: TOB-0x-013
Target: `multisig/contracts/src/{AssetProxyOwner, MultiSigWalletWithTimeLock}.sol`

**Description**
The specification for `AssetProxyOwner` says: "*The AssetProxyOwner is a time-locked multi-signature wallet that has permission to perform administrative functions within the protocol. Submitted transactions must pass a 2 week timelock before they are executed.*"

The `MultiSigWalletWithTimeLock.sol` and `AssetProxyOwner.sol` contracts' timelock-period implementation/usage does not enforce the two-week period, but is instead configurable by the wallet owner without any range checks. Either the specification is outdated (most likely), or this is a serious flaw.

**Exploit Scenario**
Assuming the specification is correct and indeed expects a two-week timelock: Alice, Bob and Eve are the owners of `AssetProxyOwner`, which has been configured with a timelock period of one day. One of them submits a transaction assuming a timelock period of two weeks, but it can be executed after one day, which is not what they expect according to the specification.

**Recommendation**
Short term, implement the necessary range checks to enforce the timelock described in the specification. Otherwise correct the specification to match the intended behavior.

Long term, make sure implementation and specification are in sync. Use Echidna or Manticore to test that your code properly implements the specification.

## 14. Potential overflow in MultiSigWalletWithTimelock when calculating whether the timelock has passed

Severity: Low
Difficulty: High
Type: Data Validation
Finding ID: TOB-0x-014
Target: `multisig/contracts/src/MultiSigWalletWithTimeLock.sol`

**Description**
Within the `pastTimeLock` modifier, a `require` statement validates that a transaction's timelock has passed. However, an overflow is possible when calculating the amount of time that must pass for a given lock to be unlocked due to the lack of `SafeMath` use.

```
    modifier pastTimeLock(uint256 transactionId) {
        require(
            block.timestamp >= confirmationTimes[transactionId] + secondsTimeLocked,
            "TIME_LOCK_INCOMPLETE"
        );
        _;
    }
```

*Figure 14.1: The* `pastTimeLock` *modifier definition, highlighting the addition of the* `confirmationTime` *for a given transaction with the* `secondsTimeLocked` *without the use of* `SafeMath`.

**Exploit Scenario**
Alice and Eve are owners of the `MultiSigWalletWithTimelock`. Alice wishes to set the timelock to be unlocked at a date which will virtually never be encountered, allowing the wallet to be abandoned. Accordingly, Alice submits a transaction to execute `changeTimeLock` with a large number. Eve knows this submitted transaction will cause an overflow, allowing immediate execution of previously locked transactions. Eve confirms and executes Alice's transaction, and is now able to execute previously locked transactions.

**Recommendation**
Short term, use `SafeMath` when performing calculations in the wallet contracts.

Long term, ensure proper testing is applied to the wallet contracts. Vulnerabilities in these contracts could have far-reaching effects, especially on controlling aspects of the 0x exchange and staking systems.

## 15. Rounding division errors can accumulate over partial fills

Severity: Informational
Difficulty: Medium
Type: Numerics
Finding ID: TOB-0x-015
Target: `exchange-libs/contracts/src/LibFillResults.sol`

**Description**

The accumulation of rounding errors can produce unexpected results over a number of partial fills. In certain situations where the taker asset amount is large and the partial fills are made with very small values, it is possible to accumulate the rounding errors to pay less to the taker and fees address.

The computation of the amounts to transfer for the taker, maker, and fees is performed in the `calculateFillResults` function:

```
function calculateFillResults(
     LibOrder.Order memory order,
     uint256 takerAssetFilledAmount,
     uint256 protocolFeeMultiplier,
     uint256 gasPrice
 )
     internal
     pure
     returns (FillResults memory fillResults)
 {
     // Compute proportional transfer amounts
     fillResults.takerAssetFilledAmount = takerAssetFilledAmount;
     fillResults.makerAssetFilledAmount = LibMath.safeGetPartialAmountFloor(
         takerAssetFilledAmount,
         order.takerAssetAmount,
         order.makerAssetAmount
     );
     fillResults.makerFeePaid = LibMath.safeGetPartialAmountFloor(
         takerAssetFilledAmount,
         order.takerAssetAmount,
         order.makerFee
     );
     fillResults.takerFeePaid = LibMath.safeGetPartialAmountFloor(
         takerAssetFilledAmount,
         order.takerAssetAmount,
         order.takerFee
     );
```

```


        // Compute the protocol fee that should be paid for a single fill.
        fillResults.protocolFeePaid = gasPrice.safeMul(protocolFeeMultiplier);


        return fillResults;
    }
```
*Figure 15.1: The* `calculateFillResults` *function.*

It is important to note that the `takerAssetFilledAmount` is completely controlled by the sender, and remaining values are computed using `safeGetPartialAmountFloor`. This function calculates the corresponding amount to transfer, ensuring that the rounding error is less than 0.1%:

```
    function safeGetPartialAmountFloor(
        uint256 numerator,
        uint256 denominator,
        uint256 target
    )
        internal
        pure
        returns (uint256 partialAmount)
    {
        if (isRoundingErrorFloor(
                numerator,
                denominator,
                target
        )) {
            LibRichErrors.rrevert(LibMathRichErrors.RoundingError(
                numerator,
                denominator,
                target
            ));
        }


        partialAmount = numerator.safeMul(target).safeDiv(denominator);
        return partialAmount;
    }
```
*Figure 15.2: The* `safeGetPartialAmountFloor` *function.*

However, although the rounding error is bounded, it can be accumulated over partial fills. In particular, the `partialAmount` can be zero, so maker and fee amounts can be zero if the `takerAssetAmount` is very large.

**Exploit Scenario**
A malicious user can perform several small partial fills of an order to avoid paying the corresponding amount to the taker and/or the fees. For instance, she can create an order with `takerAssetAmount = 999,910,000,000,000` and `makerAssetAmount = 1,000,000,000.` To exploit the rounding issue, she will make partial fills with `10,000,000,000` tokens each one. This should work since the relative rounding error is less than 0.01%:

```
>>> round(((10000000000*1000000000./999910000000000) -
floor(10000000000*1000000000./999910000000000)) /
(10000000000*1000000000./999910000000000),5)*100
0.009000000000000001
```

This will require a large amount of transactions to fill this order: `99,991` to be exact. At the end, the taker should receive 1,000,000,000, however, it will receive:

```
>>> floor(999910000000000/10000000000) * floor(10000000000*1000000000/999910000000000)
999910000
```

As a result of this, the taker will receive `90,000` tokens less than expected.

**Recommendation**
Short term, select a proper bound for the accumulated rounding error, add code to keep track of it for each order and disallow a partial fill if it increases beyond the bound.

Long term, use Echidna or Manticore to test for properties that could fail after a sequence of transactions.

## 16. The Cobb-Douglas function is not properly documented and reverts with valid parameters

Severity: Medium
Difficulty: High
Type: Data Validation
Finding ID: TOB-0x-016
Target: `contracts/staking/contracts/src/libs/LibCobbDouglas.sol`

**Description**
Documentation indicates that the Cobb–Douglas function (Figure 16.2) should not revert for inputs within bounds as described in Figure 16.1. However, it appears that there are inputs which lead the function to revert, as described in Figure 16.3. This issue was directly identified using Echidna, our property based testing tool for smart contracts.

```
totalRewards < bound
fees <= totalFees < bound && totalFees > 0
stake <= totalStake < bound && totalStake > 0
alphaNumerator <= alphaDenominator < bound && alphaDenominator > 0
where bound = 0x200000000000000000000000000000000
```

*Figure 16.1: The expected bounds for the Cobb–Douglas function.*

```
    function cobbDouglas(
        uint256 totalRewards,
        uint256 ownerFees,
        uint256 totalFees,
        uint256 ownerStake,
        uint256 totalStake,
        uint256 alphaNumerator,
        uint256 alphaDenominator
    )
        internal
        pure
        returns (uint256 ownerRewards)
    {
        int256 feeRatio = LibFixedMath._toFixed(ownerFees, totalFees);
        int256 stakeRatio = LibFixedMath._toFixed(ownerStake, totalStake);
        if (feeRatio == 0 || stakeRatio == 0) {
            return ownerRewards = 0;
        }
...
        int256 n = feeRatio <= stakeRatio ?
            LibFixedMath._div(feeRatio, stakeRatio) :
            LibFixedMath._div(stakeRatio, feeRatio);
        n = LibFixedMath._exp(
            LibFixedMath._mulDiv(
                LibFixedMath._ln(n),
                int256(alphaNumerator),
                int256(alphaDenominator)
            )
        );
...
        n = feeRatio <= stakeRatio ?
            LibFixedMath._mul(stakeRatio, n) :
```

```
            LibFixedMath._div(stakeRatio, n);
        // Multiply the above with totalRewards.
        ownerRewards = LibFixedMath._uintMul(n, totalRewards);
    }
}
```

*Figure 16.2: A snippet of the* `cobbDouglas` *function, highlighting actual value computations.*

```
cobbdouglas(0,0,519229685853482762853049632922009 6,340282366920938463463374
607431768211456,340282366920938463463374607431768211456,0,134217728)
```

*Figure 16.3: An input leading to revert in the* `cobbDouglas` *function found by Echidna.*

**Exploit Scenario**
Any contract using the `cobbDouglas` function to compute fee-based rewards for staking pools can unexpectedly revert even if the input parameters are valid, potentially blocking essential operations and leaving the contract in an invalid state.

**Recommendation**
Short term, reduce the `bound` value for the parameters and properly document the input constraints for this function. We suggest  the use of `2**127-1` as bound, but the `LibFixedMath` library should be reviewed for potential issues before confirming this value.

Long term, use Echidna and Manticore to make sure the arithmetic computations return expected results and do not revert.

## 17. Unclear documentation on how order filling can fail

Severity: High
Difficulty: Medium
Type: Data Validation
Finding ID: TOB-0x-017
Target: `0x Protocol 3.0 specification, exchange/contracts/src/MixinExchangeCore.sol`

**Description**

The 0x documentation is unclear about how to determine whether orders are fillable or not. Even some fillable orders cannot be completely filled.

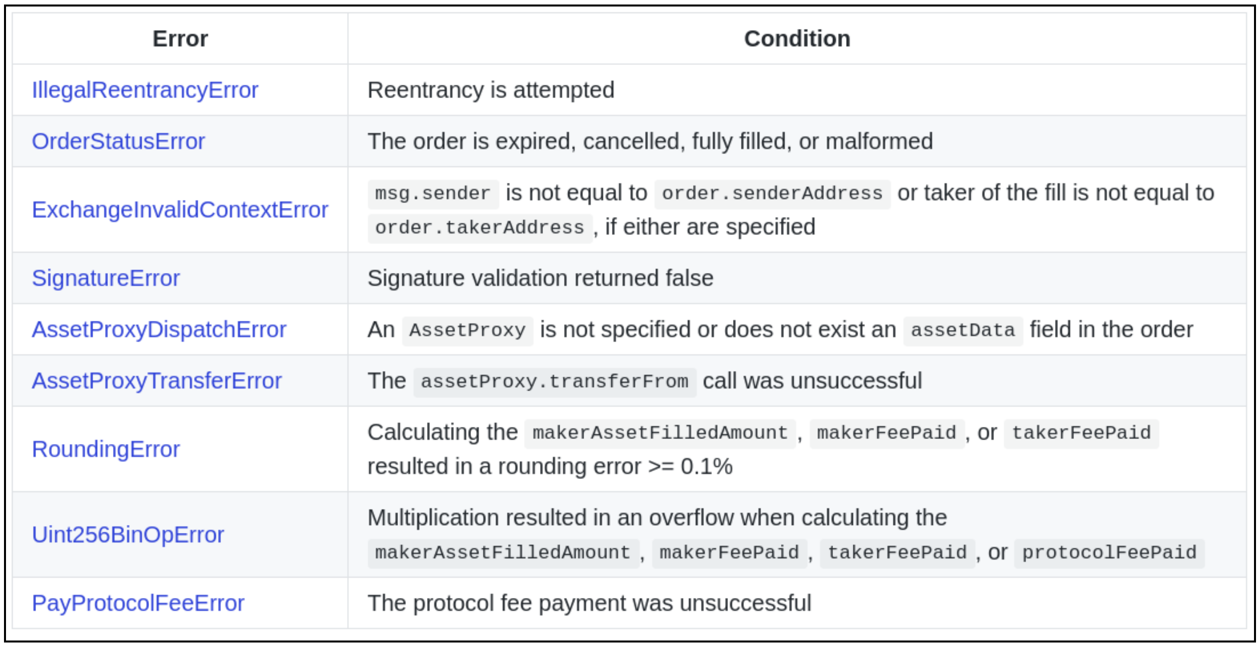
The 0x specification does not state clearly enough how fillable orders are determined. The `getOrderInfo` function can be used to learn whether an order is fillable or not:

```
function getOrderInfo(LibOrder.Order memory order)
       public
       view
       returns (LibOrder.OrderInfo memory orderInfo)
{
       // Compute the order hash and fetch the amount of takerAsset that has already been
filled
       (orderInfo.orderHash, orderInfo.orderTakerAssetFilledAmount) =
_getOrderHashAndFilledAmount(order);


       // If order.makerAssetAmount is zero, we also reject the order.
       // While the Exchange contract handles them correctly, they create
       // edge cases in the supporting infrastructure because they have
       // an 'infinite' price when computed by a simple division.
       if (order.makerAssetAmount == 0) {
           orderInfo.orderStatus = uint8(LibOrder.OrderStatus.INVALID_MAKER_ASSET_AMOUNT);
           return orderInfo;
       }


       // If order.takerAssetAmount is zero, then the order will always
       // be considered filled because 0 == takerAssetAmount == orderTakerAssetFilledAmount
       // Instead of distinguishing between unfilled and filled zero taker
       // amount orders, we choose not to support them.
       if (order.takerAssetAmount == 0) {
           orderInfo.orderStatus = uint8(LibOrder.OrderStatus.INVALID_TAKER_ASSET_AMOUNT);
           return orderInfo;
```

```
        }


        // Validate order availability
        if (orderInfo.orderTakerAssetFilledAmount >= order.takerAssetAmount) {
            orderInfo.orderStatus = uint8(LibOrder.OrderStatus.FULLY_FILLED);
            return orderInfo;
        }


        // Validate order expiration
        // solhint-disable-next-line not-rely-on-time
        if (block.timestamp >= order.expirationTimeSeconds) {
            orderInfo.orderStatus = uint8(LibOrder.OrderStatus.EXPIRED);
            return orderInfo;
        }


        // Check if order has been cancelled
        if (cancelled[orderInfo.orderHash]) {
            orderInfo.orderStatus = uint8(LibOrder.OrderStatus.CANCELLED);
            return orderInfo;
        }
        if (orderEpoch[order.makerAddress][order.senderAddress] > order.salt) {
            orderInfo.orderStatus = uint8(LibOrder.OrderStatus.CANCELLED);
            return orderInfo;
        }


        // All other statuses are ruled out: order is Fillable
        orderInfo.orderStatus = uint8(LibOrder.OrderStatus.FILLABLE);
        return orderInfo;
    }
```

*Figure 17.1: The* `getOrderInfo` *function.*

However, even if the order appears to be fillable, the `fillOrder` code can revert for a variety of reasons:



| Error | Condition |
|---|---|
| IllegalReentrancyError | Reentrancy is attempted |
| OrderStatusError | The order is expired, cancelled, fully filled, or malformed |
| ExchangeInvalidContextError | `msg.sender` is not equal to `order.senderAddress` or taker of the fill is not equal to `order.takerAddress`, if either are specified |
| SignatureError | Signature validation returned false |
| AssetProxyDispatchError | An `AssetProxy` is not specified or does not exist an `assetData` field in the order |
| AssetProxyTransferError | The `assetProxy.transferFrom` call was unsuccessful |
| RoundingError | Calculating the `makerAssetFilledAmount`, `makerFeePaid`, or `takerFeePaid` resulted in a rounding error >= 0.1% |
| Uint256BinOpError | Multiplication resulted in an overflow when calculating the `makerAssetFilledAmount`, `makerFeePaid`, `takerFeePaid`, or `protocolFeePaid` |
| PayProtocolFeeError | The protocol fee payment was unsuccessful |

*Figure 17.2: List of possible revert causes in the `fillOrder` function.*

While this list seems complete, it does not mention exactly where the errors are triggered or how to avoid some of them. For instance, if the `RoundingError` is triggered, there is no easy way to know exactly which computation caused it, and it's not specified how a user should overcome it. Additionally, a revert caused by overflow in the calculation of the maker/taker/fee amounts can even block valid orders from being completely filled. This error is caused by the `safeGetPartialAmountFloor` code when the `partialAmount` is calculated multiplying two amounts that can overflow (e.g., `takerAssetFilledAmount` and `order.makerAssetAmount`).

```
    function safeGetPartialAmountFloor(
        uint256 numerator,
        uint256 denominator,
        uint256 target
    )
        internal
        pure
        returns (uint256 partialAmount)
    {
        if (isRoundingErrorFloor(
                numerator,
                denominator,
                target
        )) {
            LibRichErrors.rrevert(LibMathRichErrors.RoundingError(
                numerator,
                denominator,
                target
            ));
        }


        partialAmount = numerator.safeMul(target).safeDiv(denominator);
        return partialAmount;
    }
```

*Figure 17.3: The* `safeGetPartialAmountFloor` *function.*

**Exploit Scenario**
Alice wants to fill an order, but she is unable to determine if an order is fillable without actually filling it. Her order has `takerAssetAmount = 99991` and `makerAssetAmount = 5000`. She will make partial fills with 20 tokens each one. This should work since the relative rounding error is less than 0.01%:

```
>>> round(((20*5000./99991) - floor(20*5000./99991)) / (20*5000./99991),5)*100
0.009000000000000001
```

However, the order cannot be exactly completed with these partial fills, since:

```
>>> floor(99991 / 20) * 20
99980.0
```

Alice needs to fill that additional 11 tokens. However `fillOrder` will fail, since the relative error will be too large. As a result of that, Alice will have no way to complete the order and she will have no other alternative than to cancel it. Moreover, there will be no additional information or documentation on why her order fails or how to overcome the error.

**Recommendation**
Short term, define a proper procedure to determine if an order is fillable and document it in the protocol specification. If necessary, warn the user about potential constraints on the orders.

Long term, use Echidna or Manticore to test that `fillOrder` never reverts when the order is valid and is used to fully fill an order.

## 18. Potential single point of failure for "read-only-mode" and "catastrophic-failure-mode"

Severity: Informational  Difficulty: High
Type: Access Controls  Finding ID: TOB-0x-018
Target: `staking/contracts/src/{StakingProxy, ZrxVault}.sol`

**Description**
The critical read-only-mode and catastrophic-failure-mode can be activated by users who are authorized by the owners of `StakingProxy.sol` and `ZrxVault.sol` respectively.

There could be a single point of failure (insider threat or a centralisation risk) if these owners or their authorized users are controlled by EOAs and not a m-of-n `MultiSig` wallet, where such accounts accidentally/maliciously trigger the read-only-mode and/or catastrophic-failure-mode.

**Exploit Scenario**
`StakingProxy.sol` and `ZrxVault.sol` owner accounts are EOAs controlled by private keys. Alice gets hold of these keys and triggers the read-only-mode and catastrophic-failure-mode causing the exchange to stop charging protocol fees, staking contract set to read-only mode, ZRX vault detached from the staking contract and allowing users to withdraw their funds from the ZRX vault directly.

**Recommendation**
Ensure critical contract (e.g., `StakingProxy.sol` and `ZrxVault.sol`) owners are not EOAs but are m-of-n `MultiSig` wallets where m >= 2, so that a single account cannot accidentally/maliciously trigger these extreme scenarios.

## 19. ERC20 reverts during certain self-transfer

Severity: Informational
Type: Data Validation
Target: `erc20/contracts/src/ERC20.sol`

Difficulty: High
Finding ID: TOB-0x-019

**Description**

If the amount of tokens to do a self-transfer using a `transfer/transferFrom` call is larger than 2**128, the transfer will fail. This breaks invariants expected of transfer functions.

Traditionally, the `transferFrom` method moves tokens from one account to another, provided the source account has approved the sender to send such an amount using the ERC20 method `approve`. However, the `transferFrom` function in WETH9's ERC20 token does not require approval if the sender is the source of the account:

```
function transfer(address _to, uint256 _value)
    external
    returns (bool)
{
    require(
        balances[msg.sender] >= _value,
        "ERC20_INSUFFICIENT_BALANCE"
    );
    require(
        balances[_to] + _value >= balances[_to],
        "UINT256_OVERFLOW"
    );


    balances[msg.sender] -= _value;
    balances[_to] += _value;


    emit Transfer(
        msg.sender,
        _to,
        _value
    );


    return true;
    }
```

*Figure 19.1: The* `transfer` *function.*

Although it may seem intuitive to allow the owner of the account balance to transfer funds without approval, external tooling may rely on invariants which are now broken.

**Exploit Scenario**

The 0x teams extensively uses the ERC20 contract for testing. If a token transfer unexpectedly reverts, it can hide a severe bug that can be triggered by a correctly implemented ERC20 token. While self-transfers are not commonly utilized by users, they are useful for testing, so it's normal to assume that they will not revert in a testing environment.

**Recommendation**

Short term, re-implement the `transfer` and `transferFrom` to allow self-transfers in each possible context if the balance is enough.

Long term, use Echidna to review the ERC20 specification and verify your contracts meet the standard. When interfacing with external ERC20 tokens, be wary of popular tokens that do not properly implement the standard (e.g., many tokens do not include return values for `approve`, `transfer`, `transferFrom`, etc.).

## 20. _assertStakingPoolExists never returns true

Severity: Informational
Difficulty: Low
Type: Data Validation
Finding ID: TOB-0x-020
Target: `staking/contracts/src/staking_pools/MixinStakingPool.sol`

**Description**
The _assertStakingPoolExists should return a bool to determine if the staking pool exists or not; however, it only returns false or reverts.

The _assertStakingPoolExists function checks that a staking pool exists given a pool id parameter:

```
/// @dev Reverts iff a staking pool does not exist.
/// @param poolId Unique id of pool.
function _assertStakingPoolExists(bytes32 poolId)
    internal
    view
    returns (bool)
{
    if (_poolById[poolId].operator == NIL_ADDRESS) {
        // we use the pool's operator as a proxy for its existence
        LibRichErrors.rrevert(
            LibStakingRichErrors.PoolExistenceError(
                poolId,
                false
            )
        );
    }
}
```

*Figure 20.1: The `_assertStakingPoolExists` function.*

However, this function does not use a `return` statement and therefore, it will always return false or revert.

**Exploit Scenario**
The 0x teams uses _assertStakingPoolExists to check if a staking pool exists in order to verify the return value of this function. Since this function always returns false, the deployed contract will not work as expected and the contract will have to be upgraded or redeployed.

**Recommendation**

Short term, add a return statement or remove the return type. Properly adjust the documentation, if needed.

Long term, use Slither to detect when functions are missing appropriate return statements.

## 21. Calls to setParams may set invalid values and produce unexpected behavior in the staking contracts

Severity: Medium
Difficulty: High
Type: Data Validation
Finding ID: TOB-0x-021
Target: `staking/contracts/src/sys/MixinParams.sol`

**Description**
Certain parameters of the contracts can be configured to invalid values, causing a variety of issues and breaking expected interactions between contracts.

`setParams` allows the owner of the staking contracts to reparameterize critical parameters. However, reparameterization lacks sanity/threshold/limit checks on all parameters. Once a parameter change is performed, the `_setParams` function will set up the new values as shown in Figure 21.1.

```
function _setParams(
   uint256 _epochDurationInSeconds,
   uint32 _rewardDelegatedStakeWeight,
   uint256 _minimumPoolStake,
   uint256 _maximumMakersInPool,
   uint32 _cobbDouglasAlphaNumerator,
   uint32 _cobbDouglasAlphaDenominator
)
   private
{
   epochDurationInSeconds = _epochDurationInSeconds;
   rewardDelegatedStakeWeight = _rewardDelegatedStakeWeight;
   minimumPoolStake = _minimumPoolStake;
   maximumMakersInPool = _maximumMakersInPool;
   cobbDouglasAlphaNumerator = _cobbDouglasAlphaNumerator;
   cobbDouglasAlphaDenominator = _cobbDouglasAlphaDenominator;


   emit ParamsSet(
       _epochDurationInSeconds,
       _rewardDelegatedStakeWeight,
       _minimumPoolStake,
       _maximumMakersInPool,
       _cobbDouglasAlphaNumerator,
       _cobbDouglasAlphaDenominator
   );
}
```
*Figure 21.1: The `_setParams` function.*

Critical staking parameters are reparameterized without any sanity/threshold/limit checks.

**Exploit Scenario**
This issue has two exploit scenarios:

- **Scenario 1**. Alice is the owner of the staking contracts and decides to update the parameters. However, she accidentally includes invalid values for `cobbDouglasAlphaNumerator` or `cobbDouglasAlphaDenominator` parameters. After the update, any contract depending on the `cobbDouglas` function will return invalid values or revert, breaking several important interactions between them.
- **Scenario 2**. Alice wants to either start a new pool or join one. She makes estimations based on the current parameters to determine if it is economically viable to invest. At the same time, the owner of the staking contracts, Bob, is deciding to change some parameters. Alice decides to interact with the contracts at the same time that the parameters are changed. As a result, Alice's decision could lead to an economic loss for her.

**Recommendation**
Short term, add proper validation checks on all parameters in `setParams`. If the validation procedure is unclear or too complex to implement on-chain, document the potential issues that could produce invalid values.

Long term, use Echidna and Manticore to locate missing parameter checks.

## 22. Malicious non-operator maker can decrease staking pool operator share

Severity: Informational                                Difficulty: Medium
Type: Access Controls                                  Finding ID: TOB-0x-022
Target: `staking/contracts/src/staking_pools/MixinStakingPool.sol`

**Description**
Every staking pool has one operator who manages the pool of delegates and market makers. These makers credit the protocol fees their orders generate toward the pool. Note that according to discussion with the 0x Protocol team, all non-operator maker addresses are intended to be controlled by the operator. However, this is not documented anywhere and may lead to confusion for operators as to the purpose of the role and they may approve requests by third-party makers to join the pool.

Staking rewards are calculated based on the amount of `ZRX` staked by the pool as well as the amount of protocol fees the pool brings in to the ecosystem. When the rewards are distributed, a set portion goes to the operator while the rest is split among the delegates according to their stake. The operator sets their share when they create the pool and although the operator's share can be modified afterwards, it can only be reduced.

```
   /// @dev Decreases the operator share for the given pool (i.e. increases pool rewards for
members).
    /// @param poolId Unique Id of pool.
    /// @param newOperatorShare The newly decreased percentage of any rewards owned by the
operator.
    function decreaseStakingPoolOperatorShare(bytes32 poolId, uint32 newOperatorShare)
        external
        onlyStakingPoolOperatorOrMaker(poolId)
    {
        // load pool and assert that we can decrease
        uint32 currentOperatorShare = _poolById[poolId].operatorShare;
        _assertNewOperatorShare(
            poolId,
            currentOperatorShare,
            newOperatorShare
        );

        // decrease operator share
        _poolById[poolId].operatorShare = newOperatorShare;
```

*Figure 22.1: The* `decreaseStakingPoolOperatorShare` *function.*

The function that permits the operator share to be reduced, `decreaseStakingPoolOperatorShare`, can currently be called by the pool operator or any of the makers linked to that pool. The ability for the makers to also call this function seems unintended, as the makers would then be incentivized to do so in order to increase their share of the rewards.

```
    /// @dev Asserts that the sender is the operator of the input pool or the input maker.
    /// @param poolId Pool sender must be operator of.
    modifier onlyStakingPoolOperatorOrMaker(bytes32 poolId) {
        _assertSenderIsPoolOperatorOrMaker(poolId);
        _;
    }
```
*Figure 22.2: The* `onlyStakingPoolOperatorOrMaker` *modifier.*

**Exploit Scenario**
Alice is the operator of a successful pool. Bob wishes to join her pool as a maker. Alice approves his request to join and Bob proceeds to reduce Alice's share of the staking rewards to 0. Alice notices this and is forced to recreate her pool, and users will have to move their stake to the new pool.

**Recommendations**
Short term, document the purpose of the non-operator maker role within a staking pool and caution operators against allowing third-party makers to join.

Long term, remove the ability for non-operator makers to perform administrative functions within a pool by creating a new modifier `onlyStakingPoolOperator` and use it to restrict calls to `decreaseStakingPoolOperatorShare`.

## 23. Non-operator makers can add or remove other makers

Severity: Informational
Difficulty: Medium
Type: Access Controls
Finding ID: TOB-0x-023
Target: `staking/contracts/src/staking_pools/MixinStakingPool.sol`

**Description**
Every staking pool has one operator who manages the pool of delegates and market makers. These makers credit the protocol fees their orders generate toward the pool. Provided there is room in the pool, a maker can request to join it. Note that according to discussion with the 0x Protocol team, all non-operator maker addresses are intended to be controlled by the operator. However, this is not documented anywhere and may lead to confusion for operators as to the purpose of the role and they may approve requests by third-party makers to join the pool.

```
    /// @dev Allows caller to join a staking pool if already assigned.
    /// @param poolId Unique id of pool.
    function joinStakingPoolAsMaker(bytes32 poolId)
        external
    {
        // Is the maker already in a pool?
        address makerAddress = msg.sender;
        IStructs.MakerPoolJoinStatus memory poolJoinStatus =
_poolJoinedByMakerAddress[makerAddress];
        if (poolJoinStatus.confirmed) {
            LibRichErrors.rrevert(LibStakingRichErrors.MakerPoolAssignmentError(

LibStakingRichErrors.MakerPoolAssignmentErrorCodes.MakerAddressAlreadyRegistered,
                makerAddress,
                poolJoinStatus.poolId
            ));
        }

        poolJoinStatus.poolId = poolId;
        _poolJoinedByMakerAddress[makerAddress] = poolJoinStatus;
    }
```
*Figure 23.1: The `joinStakingPoolAsMaker` function.*

This request, according to the code comments in Figure 23.2 below, must be approved only by the pool operator. Additionally, it should only be possible for makers to be removed from the pool by the pool operator or by their own choice (e.g., if they wish to join a different pool as a maker instead).

```
   /// @dev Adds a maker to a staking pool. Note that this is only callable by the pool
operator.
    /// Note also that the maker must have previously called joinStakingPoolAsMaker.
    /// @param poolId Unique id of pool.
    /// @param makerAddress Address of maker.
    function addMakerToStakingPool(
        bytes32 poolId,
        address makerAddress
    )
```

```
        external
        onlyStakingPoolOperatorOrMaker(poolId)
```

*Figure 23.2: The* `addMakerToStakingPool` *function declaration.*

```
   /// @dev Removes a maker from a staking pool. Note that this is only callable by the pool
operator or maker.
    /// Note also that the maker does not have to *agree* to leave the pool; this action is
    /// at the sole discretion of the pool operator.
    /// @param poolId Unique id of pool.
    /// @param makerAddress Address of maker.
    function removeMakerFromStakingPool(
        bytes32 poolId,
        address makerAddress
    )
        external
        onlyStakingPoolOperatorOrMaker(poolId)
```

*Figure 23.3: The* `removeMakerFromStakingPool` *function declaration.*

However, currently the functions to add or remove a maker from a pool can be called by any maker in the pool as permitted by the `onlyStakingPoolOperatorOrMaker` modifier. This would allow a malicious maker to remove other makers from a pool. To recover from this, because operators cannot unilaterally add makers to a pool, each of the individual makers would have to request to join the pool again. This would potentially reduce the amount of staking rewards a pool earns depending on how quickly it was noticed and remedied. Malicious makers could also add other malicious makers to the pool without the operator's consent.

```
   /// @dev Asserts that the sender is the operator of the input pool or the input maker.
    /// @param poolId Pool sender must be operator of.
    modifier onlyStakingPoolOperatorOrMaker(bytes32 poolId) {
        _assertSenderIsPoolOperatorOrMaker(poolId);
        _;
    }
```

*Figure 23.4: The* `onlyStakingPoolOperatorOrMaker` *modifier.*

```
    /// @dev Asserts that the sender is the operator of the input pool or the input maker.
    /// @param poolId Pool sender must be operator of.
    function _assertSenderIsPoolOperatorOrMaker(bytes32 poolId)
        private
        view
    {
        address operator = _poolById[poolId].operator;
        if (
            msg.sender != operator &&
            getStakingPoolIdOfMaker(msg.sender) != poolId
        ) {
            LibRichErrors.rrevert(
                LibStakingRichErrors.OnlyCallableByPoolOperatorOrMakerError(
                    msg.sender,
                    poolId
                )
            );
        }
    }
```
*Figure 23.4: The* `_assertSenderIsPoolOperatorOrMaker` *modifier.*

**Exploit Scenario**
Alice and Bob each operate successful staking pools. Carol, a large staker (but not a maker) in Bob’s pool, requests to join Alice's staking pool as a maker. Alice approves Carol's request. Carol then removes all other makers from Alice's pool. Any orders that are filled before those makers are able to rejoin Alice's pool do not contribute to Alice's pool's staking rewards, but they indirectly increase Bob's pool's share. Since Carol is a large staker in Bob’s pool, she also benefits from this.

**Recommendations**
Short term, document the purpose of the non-operator maker role within a staking pool and caution operators against allowing third-party makers to join.

Long term, create a new modifier—`onlyStakingPoolOperator`—and use it to restrict calls to `addMakerToStakingPool`. For `removeMakerFromStakingPool`, fix the `_assertSenderIsPoolOperatorOrMaker` function to correctly check that `msg.sender` is the `makerAddress` passed to `removeMakerFromStakingPool`, not simply any maker from that pool. This will allow makers to only remove themselves while still allowing the operator to remove any maker.

## A. Vulnerability Classifications

| **Vulnerability Classes** | |
|---|---|
| **Class** | **Description** |
| Access Controls | Related to authorization of users and assessment of rights |
| Auditing and Logging | Related to auditing of actions or logging of problems |
| Authentication | Related to the identification of users |
| Configuration | Related to security configurations of servers, devices or software |
| Cryptography | Related to protecting the privacy or integrity of data |
| Data Exposure | Related to unintended exposure of sensitive information |
| Data Validation | Related to improper reliance on the structure or values of data |
| Denial of Service | Related to causing system failure |
| Documentation | Related to documentation errors, omissions, or inaccuracies |
| Error Reporting | Related to the reporting of error conditions in a secure fashion |
| Patching | Related to keeping software up to date |
| Session Management | Related to the identification of authenticated users |
| Timing | Related to race conditions, locking, or order of operations |
| Undefined Behavior | Related to undefined behavior triggered by the program |

| **Severity Categories** | |
|---|---|
| **Severity** | **Description** |
| Informational | The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth |
| Undetermined | The extent of the risk was not determined during this engagement |
| Low | The risk is relatively small or is not a risk the customer has indicated is important |
| Medium | Individual user's information is at risk, exploitation would be bad for |

| | |
|---|---|
| | client’s reputation, moderate financial impact, possible legal implications for client |
| High | Large numbers of users, very bad for client’s reputation, or serious legal or financial implications |

| **Difficulty Levels** | |
|---|---|
| **Difficulty** | **Description** |
| Undetermined | The difficulty of exploit was not determined during this engagement |
| Low | Commonly exploited, public tools exist or can be scripted that exploit this flaw |
| Medium | Attackers must write an exploit, or need an in-depth knowledge of a complex system |
| High | The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue |

## B. 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.

**General Recommendation**

- **Use updated versions of contracts instead of older, archived versions.** There are three contracts in `asset-proxy/contracts/archive`, all of which have updated versions:
    - `MixinAuthorizable.sol` and `Ownable.sol` have updated versions in `utils/contracts/src/`. These have exception handling using `LibRichErrors`, which could be used instead.
    - In the case of `MixinAssetProxyDispatcher.sol`, the archived version implements `transferFrom` in `_dispatchTransferFrom()` using assembly for gas efficiency reasons, while the updated version uses Solidity. Unless there is a good reason (such as gas efficiency) to have these two separate versions, it is better to upgrade to the Solidity version, which is more readable and auditable. If there are good reasons to use both assembly and Solidity, make sure those reasons are documented.

**exchange/contracts/src/MixinSignatureValidator.sol:**

- **Consider disallowing the approval of `0x0` in `setSignatureValidatorApproval`.** The zero address is an uninitialized value in EVM. Disallowing unexpected uses of this value can help debug issues in smart contracts.

**exchange/contracts/src/MixinWrapperFunctions.sol:**

- The documentation on the `executeTransaction` and `batchExecuteTransactions` function notes important functionality related to unused `ETH`, which `batchFillOrdersNoThrow`, `marketSellOrdersNoThrow`, and `marketBuyOrdersNoThrow` from `exchange/contracts/src/MixinTransactions.sol` do not have. The documentation specifically states: *"Refund any unused value (ETH) that was sent with the message call (note: all intermediate refunds will be disabled until this step)."* This clarifies that intermediate refunds will be disabled until the very end. The documentation should be reviewed for all of these functions and updated where necessary.

**exchange/contracts/src/MixinExchangeCore.sol:**

- **Consider reverting when orders are canceled.** The use of `fillOrder` and `cancelOrder` in invalid orders should be consistent. If this is not the case, it should be clear in the user documentation why invalid orders can be canceled.

**exchange/contracts/utils/contracts/src/LibBytes.sol:**

- **Consider using SafeMath.** There are multiple places where addition is performed on `index`. There are cases (e.g., `readAddress`) where the index value may be tainted by public functions.

**staking/contracts/src/StakingProxy.sol:**

- **Consider checking for `0x0` in `_attachStakingContract()`.** The zero address is an uninitialized value in EVM. While `batchExecute()` has this check before invoking `delegatecall`, it is safer to also add this check in `_attachStakingContract()` to prevent the creation of an invalid `stakingContract`.

**staking/contracts/src/StakingProxy.sol:**

- **Consider checking for contract existence before `delegatecall`.** The two instances of `delegatecall` in `StakingProxy.sol` do not perform a contract existence check on the staking contract.

**staking/contracts/src/sys/MixinFinalizer.sol:**

- **Consider using SafeMath.** There are two places in `MixinFinalizer.sol` where `safeSub` is not used for performing `epoch - 1`. While neither instance leads to an overflow currently because of a previous check for `epoch == 0`, it is safer to use `safeSub` even here just in case the logic changes in future.

**exchange/contracts/src/MixinMatching.sol:**

- **Consider reverting when an order is matched with itself.** The `matchOrder` function have a corner case that allows to match certain orders with themselves. It is unclear what is the expected behaviour in that situation, so unless there is a defined use case, we recommend to revert in that matching.

**contracts/exchange/test/signature_validator.ts**

- **Implement a unit test to check for the `preSign` idempotency.** The `preSign` function should be idempotent if it is called more than once, but currently there is no unit test to verify this property.

**staking/contracts/src/immutable/MixinDeploymentConstants.sol**

- **Consider removing commented constants in MixinDeploymentConstants**. The constants such WETH_ADDRESS, WETH_ASSET_DATA and WETH_ASSET_PROXY_ADDRESS in this contract should be carefully verified before deploying it since they are quite easy to confuse with each other.

# C. Tool Improvements

This section details enhancements and fixes in our tools produced during the engagement. These enhancements improved our ability to test the 0x contracts and increased the depth of testing possible by using them.

**Echidna**

- Support for ABIEncoderV2 (PR) and fixes (PR).
- Improved initialization support when creating multiple contracts (PR).
- User contract addresses created by function calls in the generation of inputs (PR).
- Improved support for saving and loading single-transaction coverage (PR).
- Improved coverage detection using complete list of transactions (PR).
    - Improved support for saving and loading multi-transaction coverage.
    - New mutation modes for multiple transactions.
    - Optimized single-transaction mode for mutation and generation.
- Issues with fixes in development:
    - Echidna generates only transactions with `tx.gasprice == 0` (Issue).
    - Echidna crashes when trying to call a nonexistent contract in the constructor (Issue).

**Manticore**

- Fix for correct `mulmod` and `addmod` symbolic inputs (PR).
- A general approach to handle symbolic imprecisions (PR).

**Slither**

- Fix for parsing of infinite loops with break (PR)

## D. Formal verification using Manticore

Trail of Bits used Manticore, our open-source dynamic EVM analysis tool that takes advantage of symbolic execution, to find issues in the Solidity components of 0x Protocol. Symbolic execution allows us to explore program behavior in a broader way than classical testing methods, such as fuzzing.

Trail of Bits used Manticore to determine if certain invalid contract states were feasible. In particular, we verified that the `safeGetPartialAmountFloor` function cannot be used to completely avoid paying the taker or the corresponding fees. This function is used to compute the amount of partial amount of `t` to pay during a partial order fill. It has checks to avoid a rounding error greater or equal than 0.1% during the computation of `t*n/d`.

We used Manticore to symbolically explore the `safeGetPartialAmountFloor` function from the `LibMath` library using the following test harness:

```
import "../src/LibMath.sol";

contract CryticTestLibMath {

   function crytic_safeGetPartialAmountFloor(uint256 n, uint256 d, uint256 t) public returns
(bool) {
       if (n == 0 || d == 0 || t == 0 || n > d)
         return true;

       uint p = LibMath.safeGetPartialAmountFloor(n,d,t);
       if (p > 0)
         return true;
       return false;
   }
}
```

*Figure D.1: The* `crytic_safeGetPartialAmountFloor` *property.*

Manticore was able to prove that there is no input that will falsify this property by evaluating all the possible traces during the symbolic exploration.

The focus and the timeframe alloted for the engagement did not allow for further development of Manticore verifications. We encourage 0x to continue using Manticore to verify additional core properties.

## E. Integrating fuzzing into the development and testing cycle

During the audit, we made several improvements to Echidna, including one to save the state of the fuzzing campaign when the campaign ends. Echidna can save the lists of collected transactions with the `corpusDir` configuration keyword. For instance, after a campaign to fuzz the LibBytes test finishes, the transactions are available for inspection:

```
[
Tx{_call =
      Left
        ("set",
         [AbiBytesDynamic

"\t\245W\140@\ESC\DC2\207\138\202\217\176\242\250.\SI\171\222\\\129H1"]),
    _src = 42424242, _dst = a329c0648769a73afac7f9381e08fb43dbea72,
    _gas' = 4294967295, _value = 0, _delay = (389523, 35832)},
 Tx{_call =
      Left
        ("publicWriteAddress",
         [AbiUInt 256 0,
          AbiAddress 1330211579262674796381903807458157140913234527569]),
    _src = 43434343, _dst = a329c0648769a73afac7f9381e08fb43dbea72,
    _gas' = 4294967295, _value = 0, _delay = (131536, 39671)}
]
```

*Figure E.1: An example transaction output from Echidna.*

This corpus of transactions can be used as input to future fuzzing campaigns, allowing Echidna to reproduce the same coverage. During development, this increases confidence in bug fixes and quickly detects regressions. This feature is also useful for continuous integration testing systems. Once the fuzzing procedure has been tuned for speed, integrate it into your CI as follows:

1. After the initial fuzzing campaign, save the corpus generated by every test.
2. For every internal milestone, new feature, or public release, re-run the fuzzing campaign starting with the current corpora for each test for at least 24 hours.
3. Update the corpora with the new inputs generated.

Over time, the corpora will represent thousands of CPU hours of refinement, and will be valuable for efficiently covering code with fuzz tests. An attacker could also use them to quickly identify vulnerable code. To avoid this additional risk, keep the fuzzing corpora in an access-controlled storage location rather than a public repository. For example, some CI systems allow maintainers to keep a cache to accelerate building and testing and the fuzz test corpora could be stored there.