A CONSENSYS DILIGENCE AUDIT REPORT

0x v3 Exchange

Date	September 2019
Lead Auditor	Steve Marx
Co-auditors	Sergii Kravchenko, Alex Wade

1 Summary

ConsenSys Diligence conducted a security audit on version 3 of the 0x Exchange contract system.

ConsenSys has previously audited 0x v2. The 0x v2 audit report is good background reading.

2 Audit Scope

The scope of this audit was the following projects within the 0x monorepo:

- `exchange`
- `exchange-libs`
- `multisig`

- `utils`

A separate report will cover the `staking` contracts.

The following files were reviewed:

File Name	SHA-1 Hash
exchange/contracts/src/Exchange.sol	cb6733c32d3306348791b83a9ae76460b75555df
exchange/contracts/src/MixinAssetProxyDispatcher.sol	ee5492092ebea3397d53163cad5cfe8b8050f88e
exchange/contracts/src/MixinExchangeCore.sol	87f9d192c0d75569ee95705baa9c1cdfd129d7a5
exchange/contracts/src/MixinMatchOrders.sol	42868be4aea9327a636766682a8655686af3fc72
exchange/contracts/src/MixinProtocolFees.sol	4982d287aaa206897698039fb34f95f53deda0b5
exchange/contracts/src/MixinSignatureValidator.sol	a69bf0916642b2abaf7e2705d704c00bf2e79150
exchange/contracts/src/MixinTransactions.sol	c3108f751ef627e171ad35c445c8e38cbe0c4d2c
exchange/contracts/src/MixinTransferSimulator.sol	b3ceb9d2e4a8cc1c55648548b950ad1114d29961
exchange/contracts/src/MixinWrapperFunctions.sol	69ea7edd94fc6fd1ede6c6bad139e3e61472c3df
exchange/contracts/src/interfaces/IAssetProxy.sol	21860ce6d0fe6286966dab04b39784f6e2d23857
exchange/contracts/src/interfaces/IAssetProxyDispatcher.sol	f3022084eee2e1a87d4bc023d2aa58d44a3bc3c3
exchange/contracts/src/interfaces/IEIP1271Data.sol	3e98264aa000a238a3f954b17acb6c6606fb3104
exchange/contracts/src/interfaces/IEIP1271Wallet.sol	d99b3b52044cba515a1eebbee67cf5db4f0ae280

File Name	SHA-1 Hash
exchange/contracts/src/interfaces/IExchange.sol	82d342133ab823431dc072558 53f99da8cd49b10
exchange/contracts/src/interfaces/IExchangeCore.sol	48b0562a46653734202a40cc 2ce7fcf0e653327a
exchange/contracts/src/interfaces/IMatchOrders.sol	db34eec2bf4bc41c3b51ec358 03e1c5aaae4a6fb
exchange/contracts/src/interfaces/IProtocolFees.sol	bcc0151ed53fa72a87102f18015 b3bcbf604b4cc
exchange/contracts/src/interfaces/ISignatureValidator.sol	e2304c3b8612ec7b7899d163b 82a1bb1145c191a
exchange/contracts/src/interfaces/ITransactions.sol	a2f67b8a9e047c0dc7c33efda4 223e087a6e90b4
exchange/contracts/src/interfaces/ITransferSimulator.sol	02ea8f864e3277e1f7c30e0ea3 8aac177625177d
exchange/contracts/src/interfaces/IWallet.sol	81fbaee73e754cfbc57882e1cd8 1be5fbf70b9de
exchange/contracts/src/interfaces/IWrapperFunctions.sol	d1b20adfa9b2639aff21e8a0d8f 864a5b9435fa4
exchange/contracts/src/libs/LibExchangeRichErrorDecoder.sol	02c13f0e1c57b12da14b0384be bb38d1039bc7c1
exchange-libs/contracts/src/IWallet.sol	d3c769706e00d8a68175a261d 79c04a8750b6118
exchange-libs/contracts/src/LibEIP712ExchangeDomain.sol	823955e1f1b21a34ad3fda91c7e 691dd68e9a62e
exchange-libs/contracts/src/LibExchangeRichErrors.sol	e58712de5e18edfe951ea694124 859ca1a1c05f5
exchange-libs/contracts/src/LibFillResults.sol	49422e7a81067b52f6acc8fe5d e1acf21134ee7a

File Name	SHA-1 Hash
exchange-libs/contracts/src/LibMath.sol	ca6e24ec1de03bdea83351ce5f96082f8f5a9976
exchange-libs/contracts/src/LibMathRichErrors.sol	7f3b0be62d7a8d6f3026018aad08dcc9cbb41825
exchange-libs/contracts/src/LibOrder.sol	114be366ad7a0a711a0c2e552500a2c9fb1bbddf
exchange-libs/contracts/src/LibZeroExTransaction.sol	95ea4427d1df12aef259e07ac6215f2e2d9bd6d9
multisig/contracts/src/AssetProxyOwner.sol	df9ed7cba84c1362fee9de80d7774592323a86df
multisig/contracts/src/MultiSigWallet.sol	33b84d070486847dcc86a140fd682a1d8c953164
multisig/contracts/src/MultiSigWalletWithTimeLock.sol	c54d8b6631eacb20fe6bfad6ee268ab81c112614
utils/contracts/src/Authorizable.sol	2ae731a21730cfdd30feb5d20da4d4d2fa194e1d
utils/contracts/src/LibAddress.sol	33eef1855488fbbbfd1eed92101f379343a8f0f7
utils/contracts/src/LibAddressArray.sol	b13d0359922c04fadb4b24abd3d5318462c62d8e
utils/contracts/src/LibAddressArrayRichErrors.sol	883bc123ba699ba1efc11a75f806e1150e8af1ba
utils/contracts/src/LibAuthorizableRichErrors.sol	abfba41b1c63ba91803721d4d0ec6a7a1678752b
utils/contracts/src/LibBytes.sol	7a0c37b1577f5a12378fbf529177ca62314a4e62
utils/contracts/src/LibBytesRichErrors.sol	611b4e660351ee4e24140074ee1df49756e496ec

File Name	SHA-1 Hash
utils/contracts/src/LibEIP1271.sol	2fe0c70163677ea228d9bcfecdbba2627a5be77f
utils/contracts/src/LibEIP712.sol	3b486180d6ee3e6d5e1f2fa57c1ca060a1bdca9b
utils/contracts/src/LibFractions.sol	552a637f32edb135942cd1ea25e88d6972b8cf79
utils/contracts/src/LibOwnableRichErrors.sol	dfda0c5639f5fc994712421dc92b284071fc9e56
utils/contracts/src/LibReentrancyGuardRichErrors.sol	8af2504839d0b9a4a7a4694886704bee31fb43ad
utils/contracts/src/LibRichErrors.sol	3be89d9503f6fb6aee08aa515119af83d63f7d29
utils/contracts/src/LibSafeMath.sol	f095f7330b0d2b0d85370b47bd5ac98360ed5b48
utils/contracts/src/LibSafeMathRichErrors.sol	7785c4a4076e3f0be3319ec4bc17aca0090c2ce0
utils/contracts/src/Ownable.sol	8ede7b82d2ee0ed63b2162709d8afa7250efc3cf
utils/contracts/src/ReentrancyGuard.sol	5364694b8a2bba36861bfdd8d5886ece26e301a4
utils/contracts/src/Refundable.sol	0fe9acae963bb683b6c3539de8377ed05240bae0
utils/contracts/src/SafeMath.sol	5b675f9c12bf862a72c7dc71d00839214d970d34
utils/contracts/src/interfaces/IAuthorizable.sol	3a438f74bdb79cf6bff4dbe52a31651928601022
utils/contracts/src/interfaces/IOwnable.sol	5fe3a74b7d5948bba5644db684459d87e84fb5c6

The audit activities can be grouped into the following three broad categories:

1. **Security:** Identifying security related issues within the contract.
2. **Architecture:** Evaluating the system architecture through the lens of established smart contract best practices.
3. **Code quality:** A full review of the contract source code. The primary areas of focus include:
    - Correctness
    - Readability
    - Scalability
    - Code complexity
    - Quality of test coverage

3 System Overview

The 0x Exchange is a decentralized exchange where various on-chain assets can be traded. It uses an approach the 0x team refers to as “off-chain order relay with on-chain settlement”. This means that, in the typical case, traders use signatures to indicate their willingness to perform a certain trade, and anyone can deliver those trades to the on-chain exchange contract, where the trade will be executed.

The 0x protocol 3.0 specification is an excellent explanation of the exchange and its inner workings.

4 Key Observations/Recommendations

- The exchange documentation is excellent. Not only does it explain how the contract is used, but it gives a detailed explanation of what each function does.
- The code is clear and includes helpful comments.
- Code for the exchange is spread across quite a few files. This sometimes makes it difficult to follow various paths through the code.
- There is quite a bit of low-level assembly. This carries a risk, particularly where direct memory access is involved. It would be good to stick to Solidity as where possible.

- Signature checking, as in 0x v2 remains an area of high complexity. If possible, it would be good to reduce the number of signature methods.

5 Security Specification

This section describes, **from a security perspective**, the expected behavior of the system under audit. It is not a substitute for documentation. The purpose of this section is to identify specific security properties that were validated by the audit team.

5.1 Actors

The relevant actors are as follows:

- **0x team** – deploys and initializes the system. In particular, the 0x team is able to update some parameters around protocol fees, as well as updating allowed `AssetProxy` addresses, which are responsible for decoding order settlement information.
- **Traders** – *makers*, who propose trades, and *takers*, who take those trades
- **Relayers** – third parties who send trades to the exchange contract to be executed

5.2 Trust Model

In any smart contract system, it's important to identify what trust is expected/required between various actors. For this audit, we established the following trust model:

- Traders should not have to trust relayers. The only action a malicious relayer should be able to take against the interest of a trader is to fail to relay the trade. If this happens, a trader should be able to publish the trade themselves or through another relayer.
- Traders should not have to trust the 0x team. 0x can pause trading, but this only prevents further use of the contracts. 0x can also upgrade various system components, but such upgrades require a waiting period, giving traders a time to stop using the contract.

6 Issues

Each issue has an assigned severity:

- Minor issues are subjective in nature. They are typically suggestions around best practices or readability. Code maintainers should use their own judgment as to whether to address such issues.
- Medium issues are objective in nature but are not security vulnerabilities. These should be addressed unless there is a clear reason not to.
- Major issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Critical issues are directly exploitable security vulnerabilities that need to be fixed.

6.1 An account that confirms a transaction via `AssetProxyOwner` can indefinitely block that transaction

Major ✓ Fixed

Resolution

This is fixed in 0xProject/0x-monorepo#2297 by allowing transactions to be "over confirmed" without resetting the confirmation time. As long as there are enough honest signers, this prevents a malicious signer from blocking transactions.

Description

When a transaction reaches the required number of confirmations in `confirmTransaction()`, its confirmation time is recorded:

code/contracts/multisig/contracts/src/MultiSigWalletWithTimeLock.sol:L86-L100

```
/// @dev Allows an owner to confirm a transaction.
/// @param transactionId Transaction ID.
function confirmTransaction(uint256 transactionId)
    public
    ownerExists(msg.sender)
    transactionExists(transactionId)
    notConfirmed(transactionId, msg.sender)
    notFullyConfirmed(transactionId)
{
    confirmations[transactionId][msg.sender] = true;
    emit Confirmation(msg.sender, transactionId);
    if (isConfirmed(transactionId)) {
        _setConfirmationTime(transactionId, block.timestamp);
    }
}
```

Before the time lock has elapsed and the transaction is executed, any of the owners that originally confirmed the transaction can revoke their confirmation via `revokeConfirmation()`:

code/contracts/multisig/contracts/src/MultiSigWallet.sol:L249-L259

```
/// @dev Allows an owner to revoke a confirmation for a transaction.
/// @param transactionId Transaction ID.
function revokeConfirmation(uint256 transactionId)
    public
    ownerExists(msg.sender)
    confirmed(transactionId, msg.sender)
    notExecuted(transactionId)
{
    confirmations[transactionId][msg.sender] = false;
    emit Revocation(msg.sender, transactionId);
}
```

Immediately after, that owner can call `confirmTransaction()` again, which will reset the confirmation time and thus the time lock.

This is especially troubling in the case of a single compromised key, but it's also an issue for disagreement among owners, where any *m* of the *n* owners should be able to execute transactions but could be blocked.

Mitigations

Only an owner can do this, and that owner has to be part of the group that originally confirmed the transaction. This means the malicious owner may

have to front run the others to make sure they're in that initial confirmation set.

Even once a malicious owner is in position to execute this perpetual delay, they need to call `revokeConfirmation()` and `confirmTransaction()` again each time. Another owner can attempt to front the attacker and execute their own `confirmTransaction()` immediately after the `revokeConfirmation()` to regain control.

Recommendation

There are several ways to address this, but to best preserve the original `MultiSigWallet` semantics, once a transaction has reached the required number of confirmations, it should be impossible to revoke confirmations. In the original implementation, this is enforced by immediately executing the transaction when the final confirmation is received.

6.2 Orders with signatures that require regular validation can have their validation bypassed if the order is partially filled Major ✓ Fixed

Resolution

This is fixed in 0xProject/0x-monorepo#2246. Signatures are now always validated each time, regardless of type.

Description

The signature types `Wallet`, `Validator`, and `EIP1271Wallet` require explicit validation to authorize each action performed on a given order. This means that if an order was signed using one of these methods, the `Exchange` must perform a validation step on the signature each time the order is submitted for a partial fill. In contrast, the other canonical signature types (`EIP712`, `EthSign`, and `PreSigned`) are only required to be validated by the `Exchange` on the order's first fill; subsequent fills take the order's existing fill amount as implicit validation that the order has a valid, published signature.

This re-validation step for `Wallet`, `Validator`, and `EIP1271Wallet` signatures is intended to facilitate their use with contracts whose validation depends on

some state that may change over time. For example, a validating contract may call into a price feed and determine that some order is invalid if its price deviates from some expected range. In this case, the repeated validation allows 0x users to make orders with custom fill conditions which are evaluated at run-time.

We found that if the sender provides the contract with an invalid signature after the order in question has already been partially filled, the regular validation check required for `Wallet`, `Validator`, and `EIP1271Wallet` signatures can be bypassed entirely.

Examples

Signature validation takes place in `MixinExchangeCore._assertFillableOrder`. A signature is only validated if it passes the following criteria:

code/contracts/exchange/contracts/src/MixinExchangeCore.sol:L372-L381

```
// Validate either on the first fill or if the signature type requires
// regular validation.
address makerAddress = order.makerAddress;
if (orderInfo.orderTakerAssetFilledAmount == 0 ||
    _doesSignatureRequireRegularValidation(
        orderInfo.orderHash,
        makerAddress,
        signature
    )
) {
```

In effect, signature validation only occurs if:

- `orderInfo.orderTakerAssetFilledAmount == 0` OR
- `_doesSignatureRequireRegularValidation(orderHash, makerAddress, signature)`

If an order is partially filled, the first condition will evaluate to false. Then, that order's signature will only be validated if `_doesSignatureRequireRegularValidation` evaluates to true:

code/contracts/exchange/contracts/src/MixinSignatureValidator.sol:L183-L206

```
function _doesSignatureRequireRegularValidation(
    bytes32 hash,
    address signerAddress,
    bytes memory signature
)
    internal
    pure
    returns (bool needsRegularValidation)
{
    // Read the signatureType from the signature
    SignatureType signatureType = _readSignatureType(
        hash,
        signerAddress,
        signature
    );

    // Any signature type that makes an external call needs to be revalidated
    // with every partial fill
    needsRegularValidation =
        signatureType == SignatureType.Wallet ||
        signatureType == SignatureType.Validator ||
        signatureType == SignatureType.EIP1271Wallet;
    return needsRegularValidation;
}
```

The `SignatureType` returned from `_readSignatureType` is directly cast from the final byte of the passed-in signature. Any value that does not cast to `Wallet`, `Validator`, and `EIP1271Wallet` will cause `_doesSignatureRequireRegularValidation` to return false, skipping validation.

The result is that an order whose signature requires regular validation can be forced to skip validation if it has been partially filled, by passing in an invalid signature.

Recommendation

There are a few options for remediation: 1. Have the `Exchange` validate the provided signature every time an order is filled. 2. Record the first seen signature type or signature hash for each order, and check that subsequent actions are submitted with a matching signature.

The first option requires the fewest changes, and does not require storing additional state. While this does mean some additional cost validating subsequent signatures, we feel the increase in flexibility is well worth it, as a

maker could choose to create multiple valid signatures for use across different order books.

6.3 Changing the owners or required confirmations in the `AssetProxyOwner` can unconfirm a previously confirmed transaction Medium ✓ Fixed

Resolution

This issue is somewhat inaccurate: `isConfirmed()` breaks out of the loop once it's found the correct number of confirmations. That means that lowering the number of required confirmations is not a problem.

Further, 0xProject/0x-monorepo#2297 allows signers to confirm transactions that have already been confirmed.

Increasing signing requirements or changing signers can still unconfirm previously confirmed transactions, but the development team is happy with that behavior.

Description

Once a transaction has been confirmed in the `AssetProxyOwner`, it cannot be executed until a lock period has passed. During that time, any change to the number of required confirmations will cause this transaction to no longer be executable.

If the number of required confirmations was *decreased*, then one or more owners will have to revoke their confirmation before the transaction can be executed.

If the number of required confirmations was *increased*, then additional owners will have to confirm the transaction, and when the new required number of confirmations is reached, a new confirmation time will be recorded, and thus the time lock will restart.

Similarly, if an owner that had previously confirmed the transaction is replaced, the number of confirmations will drop for existing transactions, and they will need to be confirmed again.

This is not disastrous, but it's almost certainly unintended behavior and may make it difficult to make changes to the multisig owners and parameters.

Examples

`executeTransaction()` requires that at the time of execution, the transaction is confirmed:

code/contracts/multisig/contracts/src/AssetProxyOwner.sol:L115-L118

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

`isConfirmed()` checks for exact equality with the number of required confirmations. Having too many confirmations is just as bad as too few:

code/contracts/multisig/contracts/src/MultiSigWallet.sol:L318-L335

```
/// @dev Returns the confirmation status of a transaction.
/// @param transactionId Transaction ID.
/// @return Confirmation status.
function isConfirmed(uint256 transactionId)
    public
    view
    returns (bool)
{
    uint256 count = 0;
    for (uint256 i = 0; i < owners.length; i++) {
        if (confirmations[transactionId][owners[i]]) {
            count += 1;
        }
        if (count == required) {
            return true;
        }
    }
}
```

If additional confirmations are required to reconfirm a transaction, that resets the time lock:

code/contracts/multisig/contracts/src/MultiSigWalletWithTimeLock.sol:L86-L100

```
/// @dev Allows an owner to confirm a transaction.
/// @param transactionId Transaction ID.
function confirmTransaction(uint256 transactionId)
    public
    ownerExists(msg.sender)
    transactionExists(transactionId)
    notConfirmed(transactionId, msg.sender)
    notFullyConfirmed(transactionId)
{
    confirmations[transactionId][msg.sender] = true;
    emit Confirmation(msg.sender, transactionId);
    if (isConfirmed(transactionId)) {
        _setConfirmationTime(transactionId, block.timestamp);
    }
}
```

Recommendation

As in issue 6.1, the semantics of the original `MultiSigWallet` were that once a transaction is fully confirmed, it's immediately executed. The time lock means this is no longer possible, but it is possible to *record* that the transaction is confirmed and never allow this to change. In fact, the confirmation time already records this. Once the confirmation time is non-zero, a transaction should always be considered confirmed.

6.4 Reentrancy in `executeTransaction()` Medium Won't Fix

Resolution

From the development team:

- Reentrancy would be dangerous in `executeTransaction` if combined with updating the `currentContextAddress`. However, this is is prevented by checking

`currentContextAddress_ != address(0)` when validating a transaction.

- `executeTransaction` also inherits a lot of the safety from the reentrancy protection on other individual functions in the `Exchange` contract.
- Setting `transactionsExecuted` before making the `delegatecall` also prevents the same transaction from being executed multiple times.

Description

In `MixinTransactions`, `executeTransaction()` and `batchExecuteTransactions()` do not have the `nonReentrant` modifier. Because of that, it is possible to execute nested transactions or call these functions during other reentrancy attacks on the exchange. The reason behind that decision is to be able to call functions with `nonReentrant` modifier as delegated transactions.

Nested transactions are partially prevented with a separate check that does not allow transaction execution if the exchange is currently in somebody else's context:

code/contracts/exchange/contracts/src/MixinTransactions.sol:L155-L162

```
// Prevent `executeTransaction` from being called when context is already set
address currentContextAddress_ = currentContextAddress;
if (currentContextAddress_ != address(0)) {
    LibRichErrors.rrevert(LibExchangeRichErrors.TransactionInvalidContextErr
        transactionHash,
        currentContextAddress_
    ));
}
```

This check still leaves some possibility of reentrancy. Allowing that behavior is dangerous and may create possible attack vectors in the future.

Recommendation

Add a new modifier to `executeTransaction()` and `batchExecuteTransactions()` which is similar to `nonReentrant` but uses different storage slot.

6.5 “Poison” order that consumes gas can block market trades Medium Won't Fix

Resolution

From the development team:

> This can be prevented fairly easily by performing an `eth_call` off-chain before attempting to fill any orders (which is pretty standard practice). Hard coding gas limits reduces flexibility and may ultimately prevent some use cases from developing in the future.

(Note from the audit team: Hardcoding is not necessary. A parameter would do.)

Description

The market buy/sell functions gather a list of orders together for the same asset and try to fill them in order until a target amount has been traded.

These functions use `MixinWrapperFunctions._fillOrderNoThrow()` to attempt to fill each order but ignore failures. This way, if one order is unfillable for some reason, the overall market order can still succeed by filling other orders.

Orders can still force `_fillOrderNoThrow()` to revert by using an external contract for signature validation and having that contract consume all available gas.

This makes it possible to advertise a “poison” order for a low price that will block all market orders from succeeding. It’s reasonable to assume that off-chain order books will automatically include the best prices when constructing market orders, so this attack would likely be quite effective. Note that such an attack costs the attacker nothing because all they need is an on-chain contract that consumers all available gas (maybe via an `assert`). This makes it a very appealing attack vector for, e.g., an order book that wants to temporarily disable a competitor.

Details

`_fillOrderNoThrow()` forwards all available gas when filling the order:

code/contracts/exchange/contracts/src/MixinWrapperFunctions.sol:L340-L348

```
// ABI encode calldata for `fillOrder`
bytes memory fillOrderCalldata = abi.encodeWithSelector(
    IExchangeCore(address(0)).fillOrder.selector,
    order,
    takerAssetFillAmount,
    signature
);

(bool didSucceed, bytes memory returnData) = address(this).delegatecall(fill
```

Similarly, when the `Exchange` attempts to fill an order that requires external signature validation (`Wallet`, `Validator`, or `EIP1271Wallet` signature types), it forwards all available gas:

code/contracts/exchange/contracts/src/MixinSignatureValidator.sol:L642

```
(bool didSucceed, bytes memory returnData) = verifyingContractAddress.static
```

If the verifying contract consumes all available gas, it can force the overall transaction to revert.

Pedantic Note

Technically, it's impossible to consume *all* remaining gas when called by another contract because the EVM holds back a small amount, but even at the block gas limit, the amount held back would be insufficient to complete the transaction.

Recommendation

Constrain the gas that is forwarded during signature validation. This can be constrained either as a part of the signature or as a parameter provided by the taker.

6.6 Front running in `matchOrders()` Medium Won't Fix

Resolution

From the development team:

- Front-running is typically prevented with a combination of external contracts and various off-chain mechanics.
- These functions are primarily intended to be used with “matching relayers”. In this model, orders must set their `takerAddress` or `senderAddress` to the address of the matcher, who is the only party allowed to actually fill the orders. This prevents any other address from participating in a gas auction.
- A commit-reveal scheme would be difficult to take advantage of in practice, since orders could be filled through a number of other functions on the `Exchange` contract. All of these functions would have to adhere to the commit-reveal scheme in order to be effective.

Description

Calls to `matchOrders()` are made to extract profit from the price difference between two opposite orders: left and right.

code/contracts/exchange/contracts/src/MixinMatchOrders.sol:L106-L111

```
function matchOrders(
    LibOrder.Order memory leftOrder,
    LibOrder.Order memory rightOrder,
    bytes memory leftSignature,
    bytes memory rightSignature
)
```

The caller only pays protocol and transaction fees, so it’s almost always profitable to front run every call to `matchOrders()`. That would lead to gas

auctions and would make `matchOrders()` difficult to use.

Recommendation

Consider adding a commit-reveal scheme to `matchOrders()` to stop front running altogether.

6.7 The `Exchange` owner should not be able to call `executeTransaction` or `batchExecuteTransaction`

Medium Won't Fix

Resolution

From the development team:

> While this is a minor inconsistency in the logic of these functions, it is in no way dangerous. `currentContextAddress` is not used when calling any admin functions, so the address of the transaction signer will be completely disregarded.

Description

If the owner calls either of these functions, the resulting `delegatecall` can pass `onlyOwner` modifiers even if the transaction signer is not the owner. This is because, regardless of the `contextAddress` set through `_executeTransaction`, the `onlyOwner` modifier checks `msg.sender`.

Examples

1. `_executeTransaction` sets the context address to the signer address, which is not `msg.sender` in this case:

   code/contracts/exchange/contracts/src/MixinTransactions.sol:L102-L104

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

2. The resulting `delegatecall` could target an admin function like this one:

code/contracts/exchange/contracts/src/MixinAssetProxyDispatcher.sol:L38-L61

```
/// @dev Registers an asset proxy to its asset proxy id.
///      Once an asset proxy is registered, it cannot be unregistered.
/// @param assetProxy Address of new asset proxy to register.
function registerAssetProxy(address assetProxy)
    external
    onlyOwner
{
    // Ensure that no asset proxy exists with current id.
    bytes4 assetProxyId = IAssetProxy(assetProxy).getProxyId();
    address currentAssetProxy = _assetProxies[assetProxyId];
    if (currentAssetProxy != address(0)) {
        LibRichErrors.rrevert(LibExchangeRichErrors.AssetProxyExistsErr
            assetProxyId,
            currentAssetProxy
        ));
    }

    // Add asset proxy and log registration.
    _assetProxies[assetProxyId] = assetProxy;
    emit AssetProxyRegistered(
        assetProxyId,
        assetProxy
    );
}
```

3. The `onlyOwner` modifier does not check the context address, but checks `msg.sender`:

code/contracts/utils/contracts/src/Ownable.sol:L35-L45

```
function _assertSenderIsOwner()
    internal
    view
{
    if (msg.sender != owner) {
        LibRichErrors.rrevert(LibOwnableRichErrors.OnlyOwnerError(
            msg.sender,
            owner
        ));
    }
}
```

Recommendation

Add a check to `_executeTransaction` that prevents the owner from calling this function.

6.8 Anyone can front run `MixinExchangeCore.cancelOrder()` Medium Won't Fix

Resolution

From the development team:

- Front-running is typically prevented with a combination of external contracts and various off-chain mechanics.
- It is not possible to cancel an order by providing less data to the `cancelOrder` function without drastically changing the logic of the fill functions. However, this type of behavior could possibly be enforced by using external contracts that are set to the `senderAddress` of the related orders.

Description

In order to cancel an order, an authorized address (maker or sender) calls `cancelOrder(LibOrder.Order memory order)`. When calling that function, all data for the order becomes visible to everyone on the network, and anyone can fill that order before it's canceled.

Usually, a maker is canceling an order because it's no longer profitable for them, so an attacker is likely to profit from front running the `cancelOrder()` transaction.

Recommendation

Make it impossible to front run order cancelation by providing less data to the `cancelOrder()` function such that this data is insufficient to execute the order.

6.9 By manipulating the gas limit, relayers can affect the outcome of `ZeroExTransaction`s Minor Won't Fix

Resolution

From the development team:

> While this is an annoyance when used in combination with `marketBuyOrdersNoThrow` and `marketSellOrdersNoThrow`, it does not seem worth it to add a `gasLimit` to 0x transactions for this reason alone. Instead, this quirk should be documented along with a recommendation to use the `fillOrKill` variants of each market fill function when used in combination with 0x transactions.

Description

`ZeroExTransaction`s are meta transactions supported by the `Exchange`. They do not require that they are executed with a specific amount of gas, so the transaction relayer can choose how much gas to provide. By choosing a low gas limit, a relayer can affect the outcome of the transaction.

A `ZeroExTransaction` specifies a signer, an expiration, and call data for the transaction:

code/contracts/exchange-libs/contracts/src/LibZeroExTransaction.sol:L41-L47

```
struct ZeroExTransaction {
    uint256 salt;                   // Arbitrary number to ensure uniqueness
    uint256 expirationTimeSeconds;  // Timestamp in seconds at which transact
    uint256 gasPrice;               // gasPrice that transaction is required
    address signerAddress;          // Address of transaction signer.
    bytes data;                     // AbiV2 encoded calldata.
}
```

In `MixinTransactions._executeTransaction()`, all available gas is forwarded in the delegate call, and the transaction is marked as executed:

code/contracts/exchange/contracts/src/MixinTransactions.sol:L107-L108

```
transactionsExecuted[transactionHash] = true;
(bool didSucceed, bytes memory returnData) = address(this).delegatecall(trar
```

Examples

A likely attack vector for this is front running a `ZeroExTransaction` that ultimately invokes `_fillNoThrow()`. In this scenario, an attacker sees the call to `executeTransaction()` and makes their own call with a lower gas limit, causing the order being filled to run out of gas but allowing the transaction as a whole to succeed.

If such an attack is successful, the `ZeroExTransaction` cannot be replayed, so the signer must produce a new signature and try again, ad infinitum.

Recommendation

Add a `gasLimit` field to `ZeroExTransaction` and forward exactly that much gas via `delegatecall`. (Note that you must explicitly check that sufficient gas is available because the EVM allows you to supply a gas parameter that exceeds the actual remaining gas.)

6.10 Front running market orders Minor Won't Fix

Resolution

From the development team:

- Front-running is typically prevented with a combination of external contracts and various off-chain mechanics.
- Users should always understand the risk of using market orders in any market or exchange structure. Although they increase convenience and arguably have a better UX, they almost always carry more risk than other order types.
- Users can always enforce a worst price by padding a market fill with an appropriate number of orders that do not exceed the worst acceptable price.

Description

`MixinWrapperFunctions` defines a number of functions for market buy/sell orders. These functions take a list of orders and a target asset amount to buy or sell. They fill each order in turn until the target has been reached.

These functions provide an appealing opportunity for front running because of the near-guaranteed profit to be had. This is most easily explained with an example:

1. Alice wishes to buy 10 FOO tokens. She creates a market buy order to purchase tokens first from Bob, who is selling 4 FOO tokens at $9 each, and then from Eve, who is selling 20 tokens at $10 each.
2. Eve front runs this market order with a transaction that buys all 4 FOO tokens from Bob for $9 each.
3. Alice's transaction goes through, but because Bob's inventory has been depleted, all 10 FOO tokens are purchased from Eve at a price of $10 each. By front running, Eve gained $4.

In a more traditional front running scheme, Alice would have just been trying to make a simple purchase of FOO tokens at $9 each, and Eve would be taking on non-trivial risk by buying them first and hoping Alice (or another buyer) would be willing to pay a higher price later.

With a market order, however, Eve's front running is nearly risk free because she knows the market order already commits Alice to buying at the higher price.

Recommendation

For the most part, traders will simply have to understand the risks of market orders and take care to only authorize trades they will be happy with.

That said, each order in a market order could specify a maximum quantity, e.g. "I want 10 FOO tokens, and I'm willing to buy up to 10 from Bob but only up to 5 from Eve." This would limit the trader's exposure to increased prices due to front running, but it would retain the convenience and efficiency of market orders.

6.11 Modifier ordering plays a significant role in modifier efficacy Minor ✓ Fixed

Resolution

This is fixed in 0xProject/0x-monorepo#2228 by introducing a new modifier that combines the two: `refundFinalBalance`.

Description

The `nonReentrant` and `refundFinalBalance` modifiers always appear together across the 0x monorepo. When used, they invariably appear with `nonReentrant` listed first, followed by `refundFinalBalance`. This specific order appears inconsequential at first glance but is actually important. The order of execution is as follows:

1. The `nonReentrant` modifier runs (`_lockMutexOrThrowIfAlreadyLocked`).
2. If `refundFinalBalance` had a prefix, it would run now.
3. The function itself runs.
4. The `refundFinalBalance` modifier runs (`_refundNonZeroBalanceIfEnabled`).
5. The `nonReentrant` modifier runs (`_unlockMutex`).

The fact that the `refundFinalBalance` modifier runs before the mutex is unlocked is of particular importance because it potentially invokes an external call, which may reenter. If the order of the two modifiers were flipped, the mutex would unlock *before* the external call, defeating the purpose of the reentrancy guard.

Examples

code/contracts/exchange/contracts/src/MixinExchangeCore.sol:L64-L65

```
nonReentrant
refundFinalBalance
```

Recommendation

Although the order of the modifiers is correct as-is, this pattern introduces cognitive overhead when making or reviewing changes to the 0x codebase. Because the two modifiers always appear together, it may make sense to combine the two into a single modifier where the order of operations is explicit.

6.12 Several overflows in `LibBytes` Minor ✓ Addressed

Resolution

This is addressed in 0xProject/0x-monorepo#2265. Unused functions have been removed. The remaining functions are only used with safe parameters (ones guaranteed not to overflow).

Description

Several functions in `LibBytes` have integer overflows.

Examples

`LibBytes.readBytesWithLength` returns a pointer to a `bytes` array within an existing `bytes` array at some given `index`. The length of the nested array is added to the given `index` and checked against the parent array to ensure the data in

the nested array is within the bounds of the parent. However, because the addition can overflow, the bounds check can be bypassed to return an array that points to data out of bounds of the parent array.

code/contracts/utils/contracts/src/LibBytes.sol:L546-L553

```
if (b.length < index + nestedBytesLength) {
    LibRichErrors.rrevert(LibBytesRichErrors.InvalidByteOperationError(
        LibBytesRichErrors
            .InvalidByteOperationErrorCodes.LengthGreaterThanOrEqualsNestedB
        b.length,
        index + nestedBytesLength
    ));
}
```

The following functions have similar issues:

- `readAddress`
- `writeAddress`
- `readBytes32`
- `writeBytes32`
- `readBytes4`

Recommendation

An overflow check should be added to the function. Alternatively, because `readBytesWithLength` does not appear to be used anywhere in the 0x project, the function should be removed from `LibBytes`. Additionally, the following functions in `LibBytes` are also not used and should be considered for removal:

- `popLast20Bytes`
- `writeAddress`
- `writeBytes32`
- `writeUint256`
- `writeBytesWithLength`
- `deepCopyBytes`

6.13 `NSignatureTypes` enum value bypasses Solidity safety checks Minor Won't Fix

Resolution

From the development team:

> This has been left unchanged in order to provide more context with a revert when an invalid signature type is used.

Description

The `ISignatureValidator` contract defines an enum `SignatureType` to represent the different types of signatures recognized within the exchange. The final enum value, `NSignatureTypes`, is not a valid signature type. Instead, it is used by `MixinSignatureValidator` to check that the value read from the signature is a valid enum value. However, Solidity now includes its own check for enum casting, and casting a value over the maximum enum size to an enum is no longer possible.

Because of the added `NSignatureTypes` value, Solidity's check now recognizes `0x08` as a valid `SignatureType` value.

Examples

The check is made here:

code/contracts/exchange/contracts/src/MixinSignatureValidator.sol:L441-L449

```
// Ensure signature is supported
if (uint8(signatureType) >= uint8(SignatureType.NSignatureTypes)) {
    LibRichErrors.rrevert(LibExchangeRichErrors.SignatureError(
        LibExchangeRichErrors.SignatureErrorCodes.UNSUPPORTED,
        hash,
        signerAddress,
        signature
    ));
}
```

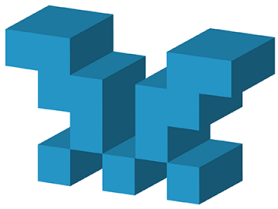
Recommendation

The check should be removed, as should the `SignatureTypes.NSignatureTypes` value.

7 Tool-Based Analysis

Several tools were used to perform automated analysis of the reviewed contracts. These issues were reviewed by the audit team, and relevant issues are listed in the Issues section.

7.1 MythX



MythX is a security analysis API for Ethereum smart contracts. It performs multiple types of analysis, including fuzzing and symbolic execution, to detect many common vulnerability types. The tool was used for automated vulnerability discovery for all audited contracts and libraries. More details on MythX can be found at mythx.io.

The full set of MythX results for both the exchange and staking contracts are available in a separate report.

7.2 Surya

Surya is an utility tool for smart contract systems. It provides a number of visual outputs and information about structure of smart contracts. It also supports querying the function call graph in multiple ways to aid in the manual inspection and control flow analysis of contracts.

Below is a complete list of functions with their visibility and modifiers:

Sūrya's Description Report

Files Description Table

File Name	SHA-1 Hash
exchange/contracts/src/Exchange.sol	cb6733c32d3306348791b83a9ae76460b75555df
exchange/contracts/src/MixinAssetProxyDispatcher.sol	ee5492092ebea3397d53163cad5cfe8b8050f88e

File Name	SHA-1 Hash
exchange/contracts/src/MixinExchangeCore.sol	87f9d192c0d75569ee95705baa9c1cdfd129d7a5
exchange/contracts/src/MixinMatchOrders.sol	42868be4aea9327a636766682a8655686af3fc72
exchange/contracts/src/MixinProtocolFees.sol	4982d287aaa206897698039fb34f95f53deda0b5
exchange/contracts/src/MixinSignatureValidator.sol	a69bf0916642b2abaf7e2705d704c00bf2e79150
exchange/contracts/src/MixinTransactions.sol	c3108f751ef627e171ad35c445c8e38cbe0c4d2c
exchange/contracts/src/MixinTransferSimulator.sol	b3ceb9d2e4a8cc1c55648548b950ad1114d29961
exchange/contracts/src/MixinWrapperFunctions.sol	69ea7edd94fc6fd1ede6c6bad139e3e61472c3df
exchange/contracts/src/interfaces/IAssetProxy.sol	21860ce6d0fe6286966dab04b39784f6e2d23857
exchange/contracts/src/interfaces/IAssetProxyDispatcher.sol	f3022084eee2e1a87d4bc023d2aa58d44a3bc3c3
exchange/contracts/src/interfaces/IEIP1271Data.sol	3e98264aa000a238a3f954b17acb6c6606fb3104
exchange/contracts/src/interfaces/IEIP1271Wallet.sol	d99b3b52044cba515a1eebbee67cf5db4f0ae280
exchange/contracts/src/interfaces/IExchange.sol	82d342133ab823431dc07255853f99da8cd49b10
exchange/contracts/src/interfaces/IExchangeCore.sol	48b0562a46653734202a40cc2ce7fcf0e653327a
exchange/contracts/src/interfaces/IMatchOrders.sol	db34eec2bf4bc41c3b51ec35803e1c5aaae4a6fb
exchange/contracts/src/interfaces/IProtocolFees.sol	bcc0151ed53fa72a87102f18015b3bcbf604b4cc

File Name	SHA-1 Hash
exchange/contracts/src/interfaces/ISignatureValidator.sol	e2304c3b8612ec7b7899d163b82a1bb1145c191a
exchange/contracts/src/interfaces/ITransactions.sol	a2f67b8a9e047c0dc7c33efda4223e087a6e90b4
exchange/contracts/src/interfaces/ITransferSimulator.sol	02ea8f864e3277e1f7c30e0ea38aac177625177d
exchange/contracts/src/interfaces/IWallet.sol	81fbaee73e754cfbc57882e1cd81be5fbf70b9de
exchange/contracts/src/interfaces/IWrapperFunctions.sol	d1b20adfa9b2639aff21e8a0d8f864a5b9435fa4
exchange/contracts/src/libs/LibExchangeRichErrorDecoder.sol	02c13f0e1c57b12da14b0384bebb38d1039bc7c1
exchange-libs/contracts/src/IWallet.sol	d3c769706e00d8a68175a261d79c04a8750b6118
exchange-libs/contracts/src/LibEIP712ExchangeDomain.sol	823955e1f1b21a34ad3fda91c7e691dd68e9a62e
exchange-libs/contracts/src/LibExchangeRichErrors.sol	e58712de5e18edfe951ea694124859ca1a1c05f5
exchange-libs/contracts/src/LibFillResults.sol	49422e7a81067b52f6acc8fe5de1acf21134ee7a
exchange-libs/contracts/src/LibMath.sol	ca6e24ec1de03bdea83351ce5f96082f8f5a9976
exchange-libs/contracts/src/LibMathRichErrors.sol	7f3b0be62d7a8d6f3026018aad08dcc9cbb41825
exchange-libs/contracts/src/LibOrder.sol	114be366ad7a0a711a0c2e552500a2c9fb1bbddf

File Name	SHA-1 Hash
exchange-libs/contracts/src/LibZeroExTransaction.sol	95ea4427d1df12aef259e07ac6215f2e2d9bd6d9
multisig/contracts/src/AssetProxyOwner.sol	df9ed7cba84c1362fee9de80d7774592323a86df
multisig/contracts/src/MultiSigWallet.sol	33b84d070486847dcc86a140fd682a1d8c953164
multisig/contracts/src/MultiSigWalletWithTimeLock.sol	c54d8b6631eacb20fe6bfad6ee268ab81c112614
utils/contracts/src/Authorizable.sol	2ae731a21730cfdd30feb5d20da4d4d2fa194e1d
utils/contracts/src/LibAddress.sol	33eef1855488fbbbfd1eed92101f379343a8f0f7
utils/contracts/src/LibAddressArray.sol	b13d0359922c04fadb4b24abd3d5318462c62d8e
utils/contracts/src/LibAddressArrayRichErrors.sol	883bc123ba699ba1efc11a75f806e1150e8af1ba
utils/contracts/src/LibAuthorizableRichErrors.sol	abfba41b1c63ba91803721d4d0ec6a7a1678752b
utils/contracts/src/LibBytes.sol	7a0c37b1577f5a12378fbf529177ca62314a4e62
utils/contracts/src/LibBytesRichErrors.sol	611b4e660351ee4e24140074ee1df49756e496ec
utils/contracts/src/LibEIP1271.sol	2fe0c70163677ea228d9bcfecdbba2627a5be77f
utils/contracts/src/LibEIP712.sol	3b486180d6ee3e6d5e1f2fa57c1ca060a1bdca9b
utils/contracts/src/LibFractions.sol	552a637f32edb135942cd1ea25e88d6972b8cf79

File Name	SHA-1 Hash
utils/contracts/src/LibOwnableRichErrors.sol	dfda0c5639f5fc994712421dc92b284071fc9e56
utils/contracts/src/LibReentrancyGuardRichErrors.sol	8af2504839d0b9a4a7a4694886704bee31fb43ad
utils/contracts/src/LibRichErrors.sol	3be89d9503f6fb6aee08aa515119af83d63f7d29
utils/contracts/src/LibSafeMath.sol	f095f7330b0d2b0d85370b47bd5ac98360ed5b48
utils/contracts/src/LibSafeMathRichErrors.sol	7785c4a4076e3f0be3319ec4bc17aca0090c2ce0
utils/contracts/src/Ownable.sol	8ede7b82d2ee0ed63b2162709d8afa7250efc3cf
utils/contracts/src/ReentrancyGuard.sol	5364694b8a2bba36861bfdd8d5886ece26e301a4
utils/contracts/src/Refundable.sol	0fe9acae963bb683b6c3539de8377ed05240bae0
utils/contracts/src/SafeMath.sol	5b675f9c12bf862a72c7dc71d00839214d970d34
utils/contracts/src/interfaces/IAuthorizable.sol	3a438f74bdb79cf6bff4dbe52a31651928601022
utils/contracts/src/interfaces/IOwnable.sol	5fe3a74b7d5948bba5644db684459d87e84fb5c6

Contracts Description Table

Contract	Type	Bases		
L	**Function Name**	**Visibility**	**Mutability**	**Modifiers**

Contract	Type	Bases		
Exchange	Implementation	LibEIP712ExchangeDomain, MixinMatchOrders, MixinWrapperFunctions, MixinTransferSimulator		
L	<Constructor>	Public ❗	🛑	LibEIP712ExchangeDomain
MixinAssetProxyDispatcher	Implementation	Ownable, IAssetProxyDispatcher		
L	registerAssetProxy	External ❗	🛑	onlyOwner
L	getAssetProxy	External ❗		NO❗
L	_dispatchTransferFrom	Internal 🔒	🛑	
MixinExchangeCore	Implementation	IExchangeCore, Refundable, LibEIP712ExchangeDomain, MixinAssetProxyDispatcher, MixinProtocolFees, MixinSignatureValidator		

Contract	Type	Bases		
L	cancelOrdersUpTo	External ❗	💵	refundFinalBalanceNoReentry
L	fillOrder	Public ❗	💵	refundFinalBalanceNoReentry
L	cancelOrder	Public ❗	💵	refundFinalBalanceNoReentry
L	getOrderInfo	Public ❗		NO❗
L	_fillOrder	Internal 🔒	🛑	
L	_cancelOrder	Internal 🔒	🛑	
L	_updateFilledState	Internal 🔒	🛑	
L	_updateCancelledState	Internal 🔒	🛑	
L	_assertFillableOrder	Internal 🔒		
L	_assertValidCancel	Internal 🔒		
L	_settleOrder	Internal 🔒	🛑	
L	_getOrderHashAndFilledAmount	Internal 🔒		

Contract	Type	Bases		
MixinMatchOrders	Implementation	MixinExchangeCore, IMatchOrders		
└	batchMatchOrders	Public ❗	💵	refundFinalBalanceNoReentry
└	batchMatchOrdersWithMaximalFill	Public ❗	💵	refundFinalBalanceNoReentry
└	matchOrders	Public ❗	💵	refundFinalBalanceNoReentry
└	matchOrdersWithMaximalFill	Public ❗	💵	refundFinalBalanceNoReentry
└	_assertValidMatch	Internal 🔒		
└	_batchMatchOrders	Internal 🔒	🛑	
└	_matchOrders	Internal 🔒	🛑	
└	_settleMatchedOrders	Internal 🔒	🛑	
MixinProtocolFees	Implementation	IProtocolFees, Ownable		
└	setProtocolFeeMultiplier	External ❗	🛑	onlyOwner

Contract	Type	Bases		
└	setProtocolFeeCollectorAddress	External ❗	🛑	onlyOwner
└	_paySingleProtocolFee	Internal 🔒	🛑	
└	_payTwoProtocolFees	Internal 🔒	🛑	
└	_payProtocolFeeToFeeCollector	Internal 🔒	🛑	
MixinSignatureValidator	Implementation	LibEIP712ExchangeDomain, LibEIP1271, ISignatureValidator, MixinTransactions		
└	preSign	External ❗	💵	refundFinalBalanceNoReentry
└	setSignatureValidatorApproval	External ❗	💵	refundFinalBalanceNoReentry
└	isValidHashSignature	Public ❗		NO❗
└	isValidOrderSignature	Public ❗		NO❗

Contract	Type	Bases		
L	isValidTransactionSignature	Public ❗		NO❗
L	_isValidOrderWithHashSignature	Internal 🔒		
L	_isValidTransactionWithHashSignature	Internal 🔒		
L	_validateHashSignatureTypes	Private 🔐		
L	_readSignatureType	Private 🔐		
L	_readValidSignatureType	Private 🔐		
L	_encodeEIP1271OrderWithHash	Private 🔐		
L	_encodeEIP1271TransactionWithHash	Private 🔐		
L	_validateHashWithWallet	Private 🔐		
L	_validateBytesWithWallet	Private 🔐		

Contract	Type	Bases		
└	_validateBytesWithValidator	Private 🔐		
└	_staticCallEIP1271WalletWithReducedSignatureLength	Private 🔐		
MixinTransactions	Implementation	Refundable, LibEIP712ExchangeDomain, ISignatureValidator, ITransactions		
└	executeTransaction	Public ❗	💵	disableRefundUntilEnd
└	batchExecuteTransactions	Public ❗	💵	disableRefundUntilEnd
└	_executeTransaction	Internal 🔒	🛑	
└	_assertExecutableTransaction	Internal 🔒		
└	_setCurrentContextAddressIfRequired	Internal 🔒	🛑	
└	_getCurrentContextAddress	Internal 🔒		

Contract	Type	Bases		
MixinTransferSimulator	Implementation	MixinAssetProxyDispatcher		
└	simulateDispatchTransferFromCalls	Public ❗	🛑	NO❗
MixinWrapperFunctions	Implementation	IWrapperFunctions, MixinExchangeCore		
└	fillOrKillOrder	Public ❗	💵	refundFinalBalanceNoReentry
└	batchFillOrders	Public ❗	💵	refundFinalBalanceNoReentry
└	batchFillOrKillOrders	Public ❗	💵	refundFinalBalanceNoReentry
└	batchFillOrdersNoThrow	Public ❗	💵	disableRefundUntilEnd
└	marketSellOrdersNoThrow	Public ❗	💵	disableRefundUntilEnd
└	marketBuyOrdersNoThrow	Public ❗	💵	disableRefundUntilEnd

Contract	Type	Bases		
L	marketSellOrdersFillOrKill	Public ❗	💵	NO❗
L	marketBuyOrdersFillOrKill	Public ❗	💵	NO❗
L	batchCancelOrders	Public ❗	💵	refundFinalBalanceNoReentry
L	_fillOrKillOrder	Internal 🔒	🛑	
L	_fillOrderNoThrow	Internal 🔒	🛑	
IAssetProxy	Implementation			
L	transferFrom	External ❗	🛑	NO❗
L	getProxyId	External ❗		NO❗
IAssetProxyDispatcher	Implementation			
L	registerAssetProxy	External ❗	🛑	NO❗
L	getAssetProxy	External ❗		NO❗
IEIP1271Data	Implementation			
L	OrderWithHash	External ❗		NO❗

Contract	Type	Bases		
L	ZeroExTransactionWithHash	External ❗		NO❗
IEIP1271Wallet	Implementation	LibEIP1271		
L	isValidSignature	External ❗		NO❗
IExchange	Implementation	IProtocolFees, IExchangeCore, IMatchOrders, ISignatureValidator, ITransactions, IAssetProxyDispatcher, ITransferSimulator, IWrapperFunctions		
IExchangeCore	Implementation			
L	cancelOrdersUpTo	External ❗	💵	NO❗
L	fillOrder	Public ❗	💵	NO❗
L	cancelOrder	Public ❗	💵	NO❗
L	getOrderInfo	Public ❗		NO❗
IMatchOrders	Implementation			

Contract	Type	Bases		
└	batchMatchOrders	Public ❗	💵	NO❗
└	batchMatchOrdersWithMaximalFill	Public ❗	💵	NO❗
└	matchOrders	Public ❗	💵	NO❗
└	matchOrdersWithMaximalFill	Public ❗	💵	NO❗
IProtocolFees	Implementation			
└	setProtocolFeeMultiplier	External ❗	🛑	NO❗
└	setProtocolFeeCollectorAddress	External ❗	🛑	NO❗
└	protocolFeeMultiplier	External ❗		NO❗
└	protocolFeeCollector	External ❗		NO❗
ISignatureValidator	Implementation			
└	preSign	External ❗	💵	NO❗
└	setSignatureValidatorApproval	External ❗	💵	NO❗

Contract	Type	Bases		
L	isValidHashSignature	Public ❗		NO❗
L	isValidOrderSignature	Public ❗		NO❗
L	isValidTransactionSignature	Public ❗		NO❗
L	_isValidOrderWithHashSignature	Internal 🔒		
L	_isValidTransactionWithHashSignature	Internal 🔒		
ITransactions	Implementation			
L	executeTransaction	Public ❗	💵	NO❗
L	batchExecuteTransactions	Public ❗	💵	NO❗
L	_getCurrentContextAddress	Internal 🔒		
ITransferSimulator	Implementation			

Contract	Type	Bases		
└	simulateDispatchTransferFromCalls	Public ❗	🛑	NO❗
IWallet	Implementation			
└	isValidSignature	External ❗		NO❗
IWrapperFunctions	Implementation			
└	fillOrKillOrder	Public ❗	💵	NO❗
└	batchFillOrders	Public ❗	💵	NO❗
└	batchFillOrKillOrders	Public ❗	💵	NO❗
└	batchFillOrdersNoThrow	Public ❗	💵	NO❗
└	marketSellOrdersNoThrow	Public ❗	💵	NO❗
└	marketBuyOrdersNoThrow	Public ❗	💵	NO❗
└	marketSellOrdersFillOrKill	Public ❗	💵	NO❗

Contract	Type	Bases		
└	marketBuyOrdersFillOrKill	Public ❗	💵	NO❗
└	batchCancelOrders	Public ❗	💵	NO❗
LibExchangeRichErrorDecoder	Implementation			
└	decodeSignatureError	Public ❗		NO❗
└	decodeEIP1271SignatureError	Public ❗		NO❗
└	decodeSignatureValidatorNotApprovedError	Public ❗		NO❗
└	decodeSignatureWalletError	Public ❗		NO❗
└	decodeOrderStatusError	Public ❗		NO❗
└	decodeExchangeInvalidContextError	Public ❗		NO❗
└	decodeFillError	Public ❗		NO❗

Contract	Type	Bases		
└	decodeOrderEpochError	Public ❗		NO❗
└	decodeAssetProxyExistsError	Public ❗		NO❗
└	decodeAssetProxyDispatchError	Public ❗		NO❗
└	decodeAssetProxyTransferError	Public ❗		NO❗
└	decodeNegativeSpreadError	Public ❗		NO❗
└	decodeTransactionError	Public ❗		NO❗
└	decodeTransactionExecutionError	Public ❗		NO❗
└	decodeIncompleteFillError	Public ❗		NO❗
└	_assertSelectorBytes	Private 🔐		
IWallet	Implementation			

Contract	Type	Bases		
L	isValidSignature	External ❗		NO❗
LibEIP712ExchangeDomain	Implementation			
L	<Constructor>	Public ❗	🛑	
LibExchangeRichErrors	Library			
L	SignatureErrorSelector	Internal 🔒		
L	SignatureValidatorNotApprovedErrorSelector	Internal 🔒		
L	EIP1271SignatureErrorSelector	Internal 🔒		
L	SignatureWalletErrorSelector	Internal 🔒		
L	OrderStatusErrorSelector	Internal 🔒		
L	ExchangeInvalidContextErrorSelector	Internal 🔒		

Contract	Type	Bases		
L	FillErrorSelector	Internal 🔒		
L	OrderEpochErrorSelector	Internal 🔒		
L	AssetProxyExistsErrorSelector	Internal 🔒		
L	AssetProxyDispatchErrorSelector	Internal 🔒		
L	AssetProxyTransferErrorSelector	Internal 🔒		
L	NegativeSpreadErrorSelector	Internal 🔒		
L	TransactionErrorSelector	Internal 🔒		
L	TransactionExecutionErrorSelector	Internal 🔒		
L	IncompleteFillErrorSelector	Internal 🔒		
L	BatchMatchOrdersErrorSelector	Internal 🔒		

Contract	Type	Bases		
L	TransactionGasPriceErrorSelector	Internal 🔒		
L	TransactionInvalidContextErrorSelector	Internal 🔒		
L	PayProtocolFeeErrorSelector	Internal 🔒		
L	BatchMatchOrdersError	Internal 🔒		
L	SignatureError	Internal 🔒		
L	SignatureValidatorNotApprovedError	Internal 🔒		
L	EIP1271SignatureError	Internal 🔒		
L	SignatureWalletError	Internal 🔒		
L	OrderStatusError	Internal 🔒		
L	ExchangeInvalidContextError	Internal 🔒		
L	FillError	Internal 🔒		

Contract	Type	Bases		
L	OrderEpochError	Internal 🔒		
L	AssetProxyExistsError	Internal 🔒		
L	AssetProxyDispatchError	Internal 🔒		
L	AssetProxyTransferError	Internal 🔒		
L	NegativeSpreadError	Internal 🔒		
L	TransactionError	Internal 🔒		
L	TransactionExecutionError	Internal 🔒		
L	TransactionGasPriceError	Internal 🔒		
L	TransactionInvalidContextError	Internal 🔒		
L	IncompleteFillError	Internal 🔒		
L	PayProtocolFeeError	Internal 🔒		
LibFillResults	Library			

Contract	Type	Bases		
└	calculateFillResults	Internal 🔒		
└	calculateMatchedFillResults	Internal 🔒		
└	addFillResults	Internal 🔒		
└	_calculateMatchedFillResults	Private 🔐		
└	_calculateMatchedFillResultsWithMaximalFill	Private 🔐		
└	_calculateCompleteFillBoth	Private 🔐		
└	_calculateCompleteRightFill	Private 🔐		
LibMath	Library			
└	safeGetPartialAmountFloor	Internal 🔒		
└	safeGetPartialAmountCeil	Internal 🔒		
└	getPartialAmountFloor	Internal 🔒		

Contract	Type	Bases		
L	getPartialAmountCeil	Internal 🔒		
L	isRoundingErrorFloor	Internal 🔒		
L	isRoundingErrorCeil	Internal 🔒		
LibMathRichErrors	Library			
L	DivisionByZeroError	Internal 🔒		
L	RoundingError	Internal 🔒		
LibOrder	Library			
L	getTypedDataHash	Internal 🔒		
L	getStructHash	Internal 🔒		
LibZeroExTransaction	Library			
L	getTypedDataHash	Internal 🔒		
L	getStructHash	Internal 🔒		
TestLibEIP712ExchangeDomain	Implementation	LibEIP712ExchangeDomain		

Contract	Type	Bases		
L	<Constructor>	Public ❗	🛑	LibEIP712ExchangeDomain
TestLibFillResults	Implementation			
L	calculateFillResults	Public ❗		NO❗
L	calculateMatchedFillResults	Public ❗		NO❗
L	addFillResults	Public ❗		NO❗
TestLibMath	Implementation			
L	safeGetPartialAmountFloor	Public ❗		NO❗
L	safeGetPartialAmountCeil	Public ❗		NO❗
L	getPartialAmountFloor	Public ❗		NO❗
L	getPartialAmountCeil	Public ❗		NO❗
L	isRoundingErrorFloor	Public ❗		NO❗
L	isRoundingErrorCeil	Public ❗		NO❗

Contract	Type	Bases		
TestLibOrder	Implementation			
L	getTypedDataHash	Public ❗		NO❗
L	getStructHash	Public ❗		NO❗
TestLibZeroExTransaction	Implementation			
L	getTypedDataHash	Public ❗		NO❗
L	getStructHash	Public ❗		NO❗
AssetProxyOwner	Implementation	MultiSigWalletWithTimeLock		
L	<Constructor>	Public ❗	🛑	MultiSigWalletWithTimeLock
L	registerFunctionCall	External ❗	🛑	onlyWallet
L	executeTransaction	Public ❗	🛑	notExecuted fullyConfirmed
L	_registerFunctionCall	Internal 🔒	🛑	
L	_assertValidFunctionCall	Internal 🔒		

Contract	Type	Bases		
MultiSigWallet	Implementation			
└	<Fallback>	External ❗	💵	NO❗
└	<Constructor>	Public ❗	🛑	validRequirement
└	addOwner	Public ❗	🛑	onlyWalletownerDoesNotExistnotNullvalidRequirement
└	removeOwner	Public ❗	🛑	onlyWalletownerExists
└	replaceOwner	Public ❗	🛑	onlyWalletownerExistsownerDoesNotExist
└	changeRequirement	Public ❗	🛑	onlyWalletvalidRequirement
└	submitTransaction	Public ❗	🛑	NO❗
└	confirmTransaction	Public ❗	🛑	ownerExiststransactionExistsnotConfirmed

Contract	Type	Bases		
L	revokeConfirmation	Public ❗	🛑	ownerExists confirmed notExecuted
L	executeTransaction	Public ❗	🛑	ownerExists confirmed notExecuted
L	_externalCall	Internal 🔒	🛑	
L	isConfirmed	Public ❗		NO❗
L	_addTransaction	Internal 🔒	🛑	notNull
L	getConfirmationCount	Public ❗		NO❗
L	getTransactionCount	Public ❗		NO❗
L	getOwners	Public ❗		NO❗
L	getConfirmations	Public ❗		NO❗
L	getTransactionIds	Public ❗		NO❗
MultiSigWalletWithTimeLock	Implementation	MultiSigWallet		

Contract	Type	Bases		
L	<Constructor>	Public ❗	🛑	MultiSigWallet
L	changeTimeLock	Public ❗	🛑	onlyWallet
L	confirmTransaction	Public ❗	🛑	ownerExists transactionExists notConfirmed notFullyConfirmed
L	executeTransaction	Public ❗	🛑	notExecuted fullyConfirmed pastTimeLock
L	_setConfirmationTime	Internal 🔒	🛑	
Authorizable	Implementation	Ownable, IAuthorizable		
L	<Constructor>	Public ❗	🛑	Ownable
L	addAuthorizedAddress	External ❗	🛑	onlyOwner
L	removeAuthorizedAddress	External ❗	🛑	onlyOwner

Contract	Type	Bases		
└	removeAuthorizedAddressAtIndex	External ❗	🛑	onlyOwner
└	getAuthorizedAddresses	External ❗		NO❗
└	_assertSenderIsAuthorized	Internal 🔒		
└	_addAuthorizedAddress	Internal 🔒	🛑	
└	_removeAuthorizedAddressAtIndex	Internal 🔒	🛑	
LibAddress	Library			
└	isContract	Internal 🔒		
LibAddressArray	Library			
└	append	Internal 🔒		
└	contains	Internal 🔒		
└	indexOf	Internal 🔒		
LibAddressArrayRichErrors	Library			

Contract	Type	Bases		
└	MismanagedMemoryError	Internal 🔒		
LibAuthorizableRichErrors	Library			
└	AuthorizedAddressMismatchError	Internal 🔒		
└	IndexOutOfBoundsError	Internal 🔒		
└	SenderNotAuthorizedError	Internal 🔒		
└	TargetAlreadyAuthorizedError	Internal 🔒		
└	TargetNotAuthorizedError	Internal 🔒		
└	ZeroCantBeAuthorizedError	Internal 🔒		
LibBytes	Library			
└	rawAddress	Internal 🔒		
└	contentAddress	Internal 🔒		

Contract	Type	Bases		
L	memCopy	Internal 🔒		
L	slice	Internal 🔒		
L	sliceDestructive	Internal 🔒		
L	popLastByte	Internal 🔒		
L	equals	Internal 🔒		
L	readAddress	Internal 🔒		
L	writeAddress	Internal 🔒		
L	readBytes32	Internal 🔒		
L	writeBytes32	Internal 🔒		
L	readUint256	Internal 🔒		
L	writeUint256	Internal 🔒		
L	readBytes4	Internal 🔒		
L	writeLength	Internal 🔒		
LibBytesRichErrors	Library			
L	InvalidByteOperationError	Internal 🔒		

Contract	Type	Bases		
LibEIP1271	Implementation			
LibEIP712	Library			
L	hashEIP712Domain	Internal 🔒		
L	hashEIP712Message	Internal 🔒		
LibFractions	Library			
L	add	Internal 🔒		
L	normalize	Internal 🔒		
L	normalize	Internal 🔒		
L	scaleDifference	Internal 🔒		
LibOwnableRichErrors	Library			
L	OnlyOwnerError	Internal 🔒		
L	TransferOwnerToZeroError	Internal 🔒		
LibReentrancyGuardRichErrors	Library			

Contract	Type	Bases		
└	IllegalReentrancyError	Internal 🔒		
LibRichErrors	Library			
└	StandardError	Internal 🔒		
└	rrevert	Internal 🔒		
LibSafeMath	Library			
└	safeMul	Internal 🔒		
└	safeDiv	Internal 🔒		
└	safeSub	Internal 🔒		
└	safeAdd	Internal 🔒		
└	max256	Internal 🔒		
└	min256	Internal 🔒		
LibSafeMathRichErrors	Library			
└	Uint256BinOpError	Internal 🔒		
└	Uint256DowncastError	Internal 🔒		
Ownable	Implementation	IOwnable		

Contract	Type	Bases		
L	<Constructor>	Public ❗	🛑	
L	transferOwnership	Public ❗	🛑	onlyOwner
L	_assertSenderIsOwner	Internal 🔒		
ReentrancyGuard	Implementation			
L	_lockMutexOrThrowIfAlreadyLocked	Internal 🔒	🛑	
L	_unlockMutex	Internal 🔒	🛑	
Refundable	Implementation	ReentrancyGuard		
L	_refundNonZeroBalanceIfEnabled	Internal 🔒	🛑	
L	_refundNonZeroBalance	Internal 🔒	🛑	
L	_disableRefund	Internal 🔒	🛑	
L	_enableAndRefundNonZeroBalance	Internal 🔒	🛑	

Contract	Type	Bases		
└	_areRefundsDisabled	Internal 🔒		
SafeMath	Implementation			
└	_safeMul	Internal 🔒		
└	_safeDiv	Internal 🔒		
└	_safeSub	Internal 🔒		
└	_safeAdd	Internal 🔒		
└	_max256	Internal 🔒		
└	_min256	Internal 🔒		
IAuthorizable	Implementation	IOwnable		
└	addAuthorizedAddress	External ❗	🛑	NO❗
└	removeAuthorizedAddress	External ❗	🛑	NO❗
└	removeAuthorizedAddressAtIndex	External ❗	🛑	NO❗
└	getAuthorizedAddresses	External ❗		NO❗
IOwnable	Implementation			

Contract	Type	Bases		
L	transferOwnership	Public ❗	🛑	NO❗

Legend

Symbol	Meaning
🛑	Function can modify state
💵	Function is payable

Appendix 1 - Disclosure

ConsenSys Diligence (“CD”) typically receives compensation from one or more clients (the “Clients”) for performing the analysis contained in these reports (the “Reports”). The Reports may be distributed through other means, including via ConsenSys publications and other distributions.

The Reports are not an endorsement or indictment of any particular project or team, and the Reports do not guarantee the security of any particular project. This Report does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. No Report provides any warranty or representation to any Third-Party in any respect, including regarding the bugfree nature of code, the business model or proprietors of any such business model, and the legal compliance of any such business. No third party should rely on the Reports in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset. Specifically, for the avoidance of doubt, this Report does not constitute investment advice, is not intended to be relied upon as investment advice, is not an endorsement of this project or team, and it is not a guarantee as to the absolute security of the project. CD owes no duty to any Third-Party by virtue of publishing these Reports.

PURPOSE OF REPORTS The Reports and the analysis described therein are created solely for Clients and published with their consent. The scope of our review is limited to a review of Solidity code and only the Solidity code we

note as being within the scope of our review within this report. The Solidity language itself remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond Solidity that could present security risks. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty.

CD makes the Reports available to parties other than the Clients (i.e., “third parties”) – on its website. CD hopes that by making these analyses publicly available, it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.

LINKS TO OTHER WEB SITES FROM THIS WEB SITE You may, through hypertext or other computer links, gain access to web sites operated by persons other than ConsenSys and CD. Such hyperlinks are provided for your reference and convenience only, and are the exclusive responsibility of such web sites’ owners. You agree that ConsenSys and CD are not responsible for the content or operation of such Web sites, and that ConsenSys and CD shall have no liability to you or any other person or entity for the use of third party Web sites. Except as described below, a hyperlink from this web Site to another web site does not imply or mean that ConsenSys and CD endorses the content on that Web site or the operator or operations of that site. You are solely responsible for determining the extent to which you may use any content at any other web sites to which you link from the Reports. ConsenSys and CD assumes no responsibility for the use of third party software on the Web Site and shall have no liability whatsoever to any person or entity for the accuracy or completeness of any outcome generated by such software.

TIMELINESS OF CONTENT The content contained in the Reports is current as of the date appearing on the Report and is subject to change without notice. Unless indicated otherwise, by ConsenSys and CD.