Read the Docs Template Documentation

Release 1.0

Read the Docs

Nov 24, 2020

BASICS

CHAPTER

ONE

CONTENTS

Keyword Index, Search Page

1.1 Orders

An order is a message passed into the 0x Protocol to facilitate an ERC20->ERC20 trade. There are currently two types of orders in 0x V4: **Limit** and **RFQ**.

Note: 0x Orders currently support the exchange of ERC20 Tokens. Other asset classes, like ERC721, will be added in the future based on community demand.

1.1.1 Limit Orders

Limit orders are the standard 0x Order, which encodes a possible trade between a maker and taker at a fixed price. These orders are typically distributed via Mesh/SRA (open orderbook) or OTC, and can be filled through the functions on the Exchange Proxy.

The `LimitOrder` struct has the following fields:

Field	Type	Description
`makerToken`	`address`	The ERC20 token the maker is selling and the maker is selling to the taker.
`takerToken`	`address`	The ERC20 token the taker is selling and the taker is selling to the maker.
`makerAmount`	`uint128`	The amount of makerToken being sold by the maker.
`takerAmount`	`uint128`	The amount of takerToken being sold by the taker.
`takerTokenFeeAmount`	`uint128`	Amount of takerToken paid by the taker to the feeRecipient.
`maker`	`address`	The address of the maker, and signer, of this order.
`taker`	`address`	Allowed taker address. Set to zero to allow any taker.
`sender`	`address`	Allowed address to directly call `fillLimitOrder()` (`msg.sender`). This is distinct from `taker` in meta-transactions. Set to zero to allow any caller.
`feeRecipient`	`address`	Recipient of maker token or taker token fees (if non-zero).
`pool`	`bytes32`	The staking pool to attribute the 0x protocol fee from this order. Set to zero to attribute to the default pool, not owned by anyone.
`expiry`	`uint64`	The Unix timestamp in seconds when this order expires.
`salt`	`uint256`	Arbitrary number to enforce uniqueness of the order's hash.

1.1.2 RFQ Orders

RFQ orders are a stripped down version of standard limit orders, supporting fewer fields and a leaner settlement process. These orders are fielded just-in-time, directly from market makers, during the construction of a swap quote on 0x API, and can be filled through the `fillRfqOrder()` function on the Exchange Proxy.

Some notable differences from regular limit orders are:

- RFQ orders can only be filled once. Even a partial fill will mark the order as `FILLED`.
- The only fill restrictions that can be placed on an RFQ order is on the `tx.origin` of the transaction.
- There are no taker token fees.

The `RFQOrder` struct has the following fields:

Field	Type	Description
`makerToken`	`address`	The ERC20 token the maker is selling and the maker is selling to the taker.
`takerToken`	`address`	The ERC20 token the taker is selling and the taker is selling to the maker.
`makerAmount`	`uint128`	The amount of makerToken being sold by the maker.
`takerAmount`	`uint128`	The amount of takerToken being sold by the taker.
`maker`	`address`	The address of the maker, and signer, of this order.
`txOrigin`	`address`	The allowed address of the EOA that submitted the Ethereum transaction.
`pool`	`bytes32`	The staking pool to attribute the 0x protocol fee from this order. Set to zero to attribute to the default pool, not owned by anyone.
`expiry`	`uint64`	The Unix timestamp in seconds when this order expires.
`salt`	`uint256`	Arbitrary number to enforce uniqueness of the order's hash.

1.1.3 How To Sign

Both Limit & RFQ orders must be signed by the *maker*. This signature is needed to fill an order, see Basic Functionality.

The protocol accepts signatures defined by the following struct:

```
struct {
    uint8 signatureType; // Either 2 or 3
    uint8 v; // Signature data.
    bytes32 r; // Signature data.
    bytes32 s; // Signature data.
}
```

There are two types of signatures supported: `EIP712` and `EthSign`.

- The `EIP712` signature type is best for web frontends that present an order to be signed through Metamask in a human-readable format. It relies on the eth_signTypedData JSON-RPC method exposed by MetaMask. This signature has the `signatureType` of `2`.
- The `EthSign` signature is best for use with headless providers, such as when using a geth node. This relies on the `eth_sign` JSON-RPC method common to all nodes. This signature has the `signatureType` of `3`.

In both cases, the `@0x/protocol-utils` package simplifies generating these signatures.

Note: The Protocol Utils package is still under development. This message will be removed once the package is published. - 11/24/2020.

```
const utils = require('@0x/protocol-utils');
const order = new utils.LimitOrder({ // or utils.RfqOrder
    makerToken: '0x6B175474E89094C44Da98b954EedeAC495271d0F', // DAI
    takerToken: '0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2', // WETH
    ... // Other fields
});
// Generate an EIP712 signature
const signature = await order.eip712SignTypedDataWithProviderAsync(
    web3.currentProvider,
    makerAddress,
);
// Generate an EthSign signature
const signature = await order.ethSignHashWithProviderAsync(
    web3.currentProvider,
    makerAddress,
);
```

1.2 Basic Functionality

Below is a catalog of basic Exchange functionality. For more advanced usage, like meta-transactions and dex aggregation, see the Advanced section.

Limit Orders	**Overview**
fillLimitOrder	Fills a Limit Order up to the amount requested.
fillOrKillLimitOrder	Fills exactly the amount requested or reverts.
cancelLimitOrder	Cancels an order so that it can no longer be filled.
batchCancelLimitOrders	A batch call to *cancelLimitOrder*.
cancelLimitPairOrders	Cancels orders in a specific market pair. Ex: Cancel all orders selling WETH for USDC.
batchCancelLimitPairOrders	A batch call to *cancelLimitPairOrders*.
getLimitOrderInfo	Returns the state of a given order.
getLimitOrderHash	Returns the EIP-712 hash for an order.
RFQ Orders	**Overview**
fillRfqOrder	These operate the same as the above functions, only for RFQ Orders.
fillOrKillRfqOrder	
cancelRfqOrder	
batchCancelRfqOrders	
cancelPairRfqOrders	
batchCancelPairRfqOrders	
getRfqOrderInfo	
getRfqOrderHash	
Protocol Fees	**Overview**
getProtocolFeeMultiplier	Takers of limit orders pay a protocol fee of *Multiplier * tx.gasprice*. This returns the *Multiplier*.
transferProtocolFeesForPools	Transfers protocol fees from escrow to the 0x Staking System. This should be called near the end of each epoch.

1.2.1 Limit Orders

These are the basic functions for using a Limit Order.

fillLimitOrder

Limit orders can be filled with the `fillLimitOrder()` or `fillOrKillLimitOrder()` functions on the Exchange Proxy. The address calling these function will be considered the "taker" of the order.

`fillLimitOrder()` fills a single limit order for **up to** `takerTokenFillAmount`:

```
function fillLimitOrder(
    // The order
    LimitOrder calldata order,
    // The signature
    Signature calldata signature,
    // How much taker token to fill the order with
    uint128 takerTokenFillAmount
)
    external
    payable
    // How much maker token from the order the taker received.
    returns (uint128 takerTokenFillAmount, uint128 makerTokenFillAmount);
```

fillOrKillLimitOrder

`fillOrKillLimitOrder()` fills a single limit order for **exactly** `takerTokenFillAmount`:

```
function fillOrKillLimitOrder(
    // The order
    LimitOrder calldata order,
    // The signature
    Signature calldata signature,
    // How much taker token to fill the order with
    uint128 takerTokenFillAmount
)
    external
    payable
    // How much maker token from the order the taker received.
    returns (uint128 makerTokenFillAmount);
```

cancelLimitOrder

Because there is no way to un-sign an order that has been distributed, limit orders must be cancelled on-chain through one of several functions. They can only be called by the order's maker.

`cancelLimitOrder()` cancels a single limit order created by the caller:

```
function cancelLimitOrder(
    // The order
    LimitOrder calldata order
)
    external;
```

batchCancelLimitOrders

`batchCancelLimitOrders()` cancels multiple limit orders created by the caller:

```
function batchCancelLimitOrders(
    // The orders
    LimitOrder[] calldata orders
)
    external;
```

cancelLimitPairOrders

`cancelLimitPairOrders()` will cancel all limit orders created by the caller with with a maker and taker token pair and a `salt` field < the `salt` provided. Subsequent calls to this function with the same tokens must provide a `salt` >= the last call to succeed.

```
function cancelLimitPairLimitOrders(
    address makerToken,
    address takerToken,
    uint256 salt;
)
    external;
```

batchCancelLimitPairOrders

`batchCancelLimitPairOrders()` performs multiple `cancelLimitPairOrders()` at once. Each respective index across arrays is equivalent to a single call.

```
function batchCancelLimitPairOrders(
    address[] makerTokens,
    address[] takerTokens,
    uint256[] salts;
)
    external;
```

getLimitOrderInfo

The Exchange Proxy exposes a function `getLimitOrderInfo()` to query information about a limit order, such as its fillable state and how much it has been filled by.

```
enum OrderStatus {
    INVALID,
    FILLABLE,
    FILLED,
    CANCELLED,
    EXPIRED
}

struct OrderInfo {
    // The order hash.
    bytes32 orderHash;
    // Current state of the order.
    OrderStatus status;
```

(continues on next page)

(continued from previous page)

```
    // How much taker token has been filled in the order.
    uint128 takerTokenFilledAmount;
}

function getLimitOrderInfo(
    // The order
    LimitOrder calldata order
)
    external
    view
    returns (OrderInfo memory orderInfo);
```

getLimitOrderHash

The hash of the order is used to uniquely identify an order inside the protocol. It is computed following the EIP712 spec standard. In solidity, the hash is computed as:

```
/// @dev Get the canonical hash of a limit order.
/// @param order The limit order.
/// @return orderHash The order hash.
function getLimitOrderHash(LibNativeOrder.LimitOrder calldata order)
    external
    view
    returns (bytes32 orderHash);
```

The simplest way to generate an order hash is by calling this function, ex:

```
bytes32 orderHash = IZeroEx(0xDef1C0ded9bec7F1a1670819833240f027b25EfF).
↪getLimitOrderHash(order);
```

The hash can be manually generated using the following code:

```
bytes32 orderHash = keccak256(abi.encodePacked(
    '\x19\x01',
    // The domain separator.
    keccak256(abi.encode(
        // The EIP712 domain separator type hash.
        keccak256(abi.encodePacked(
            'EIP712Domain(',
            'string name,',
            'string version,',
            'uint256 chainId,',
            'address verifyingContract)'
        )),
        // The EIP712 domain separator values.
        'ZeroEx',
        '1.0.0',
        1, // For mainnet
        0xDef1C0ded9bec7F1a1670819833240f027b25EfF, // Address of the Exchange Proxy
    )),
    // The struct hash.
    keccak256(abi.encode(
        // The EIP712 type hash.
        keccak256(abi.encodePacked(
            'LimitOrder(',
```

(continues on next page)

(continued from previous page)

```
            'address makerToken,',
            'address takerToken,',
            'uint128 makerAmount,',
            'uint128 takerAmount,',
            'uint128 takerTokenFeeAmount,',
            'address taker,',
            'address maker,',
            'address sender,',
            'address feeRecipient,',
            'bytes32 pool,',
            'uint64 expiry,',
            'uint256 salt)'
        )),
        // The struct values.
        order.makerToken,
        order.takerToken,
        order.makerAmount,
        order.takerAmount,
        order.takerTokenFeeAmount,
        order.maker,
        order.taker,
        order.sender,
        order.feeRecipient,
        order.pool,
        order.expiry,
        order.salt
    ))
));
```

1.2.2 RFQ Orders

These are the basic functions for using an RFQ Order.

fillRfqOrder

RFQ orders can be filled with the `fillRfqOrder()` or `fillOrKillRfqOrder()` functions on the Exchange Proxy. The address calling this function will be considered the "taker" of the order.

`fillRfqOrder()` fills a single RFQ order for **up to** `takerTokenFillAmount`:

```
function fillRfqOrder(
    // The order
    RfqOrder calldata order,
    // The signature
    Signature calldata signature,
    // How much taker token to fill the order with
    uint128 takerTokenFillAmount
)
    external
    payable
    // How much maker token from the order the taker received.
    returns (uint128 takerTokenFillAmount, uint128 makerTokenFillAmount);
```

fillOrKillRfqOrder

`fillOrKillRfqOrder()` fills a single RFQ order for **exactly** `takerTokenFillAmount`:

```
function fillOrKillRfqOrder(
    // The order
    RfqOrder calldata order,
    // The signature
    Signature calldata signature,
    // How much taker token to fill the order with
    uint128 takerTokenFillAmount
)
    external
    payable
    // How much maker token from the order the taker received.
    returns (uint128 makerTokenFillAmount);
```

cancelRfqOrder

Similar to limit orders, RFQ orders can be cancelled on-chain through a variety of functions, which can only be called by the order's maker.

`cancelRfqOrder()` cancels a single RFQ order created by the caller:

```
function cancelRfqOrder(
    // The order
    RfqOrder calldata order
)
    external;
```

batchCancelRfqOrders

`batchCancelRfqOrders()` cancels multiple RFQ orders created by the caller:

```
function batchCancelRfqOrders(
    // The orders
    RfqOrder[] calldata orders
)
    external;
```

cancelPairRfqOrders

`cancelPairRfqOrders()` will cancel all RFQ orders created by the caller with with a maker and taker token pair and a `salt` field < the `salt` provided. Subsequent calls to this function with the same tokens must provide a `salt` >= the last call to succeed.

```
function cancelPairRfqOrders(
    address makerToken,
    address takerToken,
    uint256 salt;
)
    external;
```

batchCancelPairRfqOrders

`batchCancelPairRfqOrders()` performs multiple `cancelPairRfqOrders()` at once. Each respective index across arrays is equivalent to a single call.

```
function batchCancelPairRfqOrders(
    address[] makerTokens,
    address[] takerTokens,
    uint256[] salts;
)
    external;
```

getRfqOrderInfo

The Exchange Proxy exposes a function `getRfqOrderInfo()` to query information about an RFQ order, such as its fillable state and how much it has been filled by.

```
enum OrderStatus {
    INVALID,
    FILLABLE,
    FILLED,
    CANCELLED,
    EXPIRED
}

struct OrderInfo {
    // The order hash.
    bytes32 orderHash;
    // Current state of the order.
    OrderStatus status;
    // How much taker token has been filled in the order.
    uint128 takerTokenFilledAmount;
}

function getRfqOrderInfo(
    // The order
    RfqOrder calldata order
)
    external
    view
    returns (OrderInfo memory orderInfo);
```

getRfqOrderHash

The hash of the order is used to uniquely identify an order inside the protocol. It is computed following the EIP712 spec standard. In solidity, the hash is computed as:

The simplest way to generate an order hash is by calling this function, ex:

```
bytes32 orderHash = IZeroEx(0xDef1C0ded9bec7F1a1670819833240f027b25EfF).
↪getRfqOrderHash(order);
```

The hash can be manually generated using the following code:

```
bytes32 orderHash = keccak256(abi.encodePacked(
    '\x19\x01',
    // The domain separator.
    keccak256(abi.encode(
        // The EIP712 domain separator type hash.
        keccak256(abi.encodePacked(
            'EIP712Domain(',
            'string name,',
            'string version,',
            'uint256 chainId,',
            'address verifyingContract)'
        )),
        // The EIP712 domain separator values.
        'ZeroEx',
        '1.0.0',
        1, // For mainnet
        0xDef1C0ded9bec7F1a1670819833240f027b25EfF, // Address of the Exchange Proxy
    )),
    // The struct hash.
    keccak256(abi.encode(
        // The EIP712 type hash.
        keccak256(abi.encodePacked(
            'RfqOrder(',
            'address makerToken,',
            'address takerToken,',
            'uint128 makerAmount,',
            'uint128 takerAmount,',
            'address maker,'
            'address txOrigin,'
            'bytes32 pool,',
            'uint64 expiry,',
            'uint256 salt)'
        )),
        // The struct values.
        order.makerToken,
        order.takerToken,
        order.makerAmount,
        order.takerAmount,
        order.maker,
        order.txOrigin,
        order.pool,
        order.expiry,
        order.salt
    ))
));
```

1.2.3 Protocol Fees

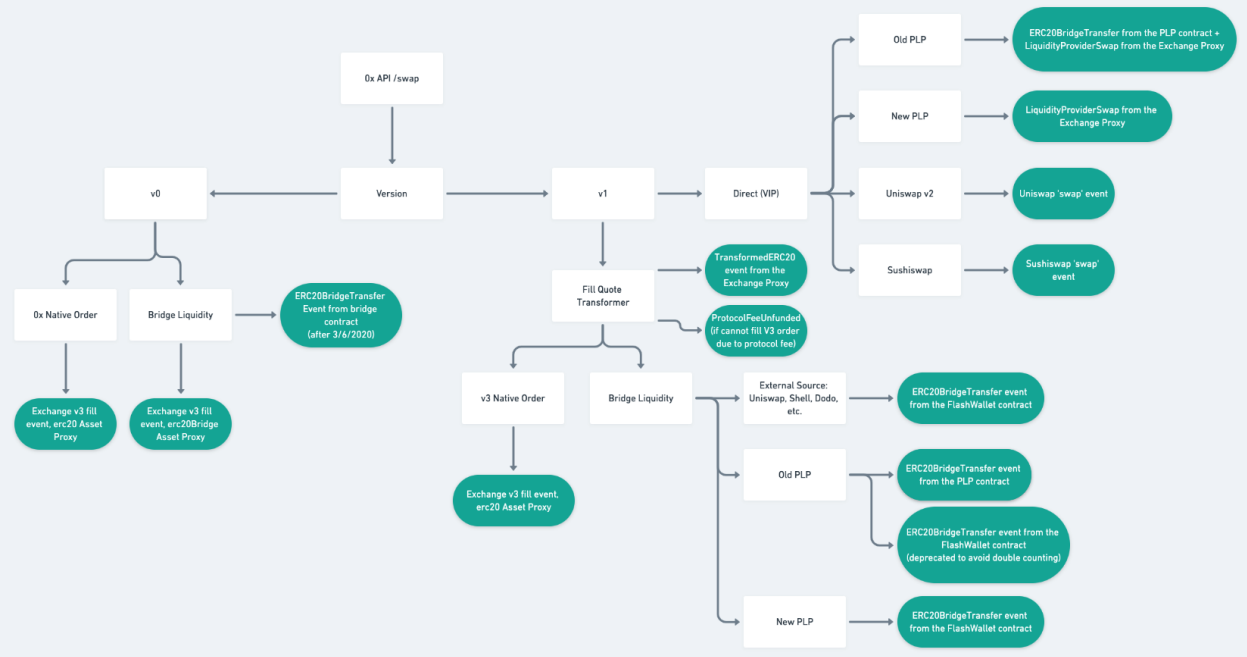
getProtocolFeeMultiplier

transferProtocolFeesForPools

1.3 Events

This page defines 0x events emitted when interacting with the Exchange Proxy. The diagram below illustrates how events are emitted when trading through the Exchange Proxy.

Warning: There are pending upgrades that impact these events. Please see the Releases page for more details.



1.3.1 Event Catalogue

This is a complete catalogue of 0x events emitted when interacting with the Exchange Proxy system of contracts.

Note: This catalogue does not include events emitted by tokens or other exchanges, like Uniswap. It also only lists 0x V3 events that are emitted during a Fill; for an extensive list of V3 events, see the V3 Spec.

Event	Description	Emitted By
Deployed	Emitted when a contract is deployed for use by the Exchange Proxy.	Exchange-Proxy
ERC20BridgeTransfer	Emitted when a trade occurs.	Flash-Wallet
Fill	Emitted by Exchange V3 when an order is filled.	Exchange V3
MetaTransactionExecuted	Emitted when a meta-transaction is executed on the Exchange Proxy. Note that this differs from meta-transactions that are executed on Exchange V3.	Exchange-Proxy
Migrated	Emitted when *ExchangeProxy.migrate()* is called.	Exchange-Proxy
ProtocolFeeUnfunded	Emitted when an order is skipped due to a lack of funds to pay the 0x Protocol fee.	Flash-Wallet
ProxyFunctionUpdated	Emitted when a function is upgraded via *extend()* or *rollback()*	Exchange-Proxy
QuoteSignerUpdated	Emitted when *ExchangeProxy.setQuoteSigner()* is called.	Exchange-Proxy
TransformedERC20	Emitted when the Transformer Deployer is upgraded.	Flash-Wallet
TransformerMetadata	A general, customizable event emitted that can be emitted by transformers as-needed.	Flash-Wallet

Deployed

```
/// @dev Emitted when a contract is deployed via `deploy()`.
/// @param deployedAddress The address of the deployed contract.
/// @param nonce The deployment nonce.
/// @param sender The caller of `deploy()`.
event Deployed(address deployedAddress, uint256 nonce, address sender);
```

ERC20BridgeTransfer

```
/// @dev Emitted when a trade occurs.
/// @param inputToken The token the bridge is converting from.
/// @param outputToken The token the bridge is converting to.
/// @param inputTokenAmount Amount of input token.
/// @param outputTokenAmount Amount of output token.
/// @param from The bridge address, indicating the underlying source of the fill.
/// @param to The `to` address, currrently `address(this)`
event ERC20BridgeTransfer(
    IERC20TokenV06 inputToken,
    IERC20TokenV06 outputToken,
    uint256 inputTokenAmount,
    uint256 outputTokenAmount,
```

(continues on next page)

(continued from previous page)

```
    address from,
    address to
);
```

Fill

```
event Fill(
    address indexed makerAddress,          // Address that created the order.
    address indexed feeRecipientAddress,   // Address that received fees.
    bytes makerAssetData,                  // Encoded data specific to makerAsset.
    bytes takerAssetData,                  // Encoded data specific to takerAsset.
    bytes makerFeeAssetData,               // Encoded data specific to makerFeeAsset.
    bytes takerFeeAssetData,               // Encoded data specific to takerFeeAsset.
    bytes32 indexed orderHash,             // EIP712 hash of order (see LibOrder.
↪getTypedDataHash).
    address takerAddress,                  // Address that filled the order.
    address senderAddress,                 // Address that called the Exchange␣
↪contract (msg.sender).
    uint256 makerAssetFilledAmount,        // Amount of makerAsset sold by maker and␣
↪bought by taker.
    uint256 takerAssetFilledAmount,        // Amount of takerAsset sold by taker and␣
↪bought by maker.
    uint256 makerFeePaid,                  // Amount of makerFeeAssetData paid to␣
↪feeRecipient by maker.
    uint256 takerFeePaid,                  // Amount of takerFeeAssetData paid to␣
↪feeRecipient by taker.
    uint256 protocolFeePaid                // Amount of eth or weth paid to the␣
↪staking contract.
);
```

Killed

```
/// @dev Emitted when a contract is killed via `kill()`.
/// @param target The address of the contract being killed..
/// @param sender The caller of `kill()`.
event Killed(address target, address sender);
```

LimitOrderFilled

```
/// @dev Emitted whenever a `LimitOrder` is filled.
/// @param orderHash The canonical hash of the order.
/// @param maker The maker of the order.
/// @param taker The taker of the order.
/// @param feeRecipient Fee recipient of the order.
/// @param takerTokenFilledAmount How much taker token was filled.
/// @param makerTokenFilledAmount How much maker token was filled.
/// @param pool The fee pool associated with this order.
event LimitOrderFilled(
    bytes32 orderHash,
    address maker,
    address taker,
```

(continues on next page)

(continued from previous page)

```
    address feeRecipient,
    address makerToken,
    address takerToken,
    uint128 takerTokenFilledAmount,
    uint128 makerTokenFilledAmount,
    bytes32 pool
);
```

MetaTransactionExecuted

```
/// @dev Emitted whenever a meta-transaction is executed via
///      `executeMetaTransaction()` or `executeMetaTransactions()`.
/// @param hash The meta-transaction hash.
/// @param selector The selector of the function being executed.
/// @param signer Who to execute the meta-transaction on behalf of.
/// @param sender Who executed the meta-transaction.
event MetaTransactionExecuted(
    bytes32 hash,
    bytes4 indexed selector,
    address signer,
    address sender
);
```

Migrated

```
/// @dev Emitted when `migrate()` is called.
/// @param caller The caller of `migrate()`.
/// @param migrator The migration contract.
/// @param newOwner The address of the new owner.
event Migrated(address caller, address migrator, address newOwner);
```

ProtocolFeeUnfunded

```
/// @dev Emitted when a trade is skipped due to a lack of funds
///      to pay the 0x Protocol fee.
/// @param orderHash The hash of the order that was skipped.
event ProtocolFeeUnfunded(bytes32 orderHash);
```

ProxyFunctionUpdated

```
/// @dev A function implementation was updated via `extend()` or `rollback()`.
/// @param selector The function selector.
/// @param oldImpl The implementation contract address being replaced.
/// @param newImpl The replacement implementation contract address.
event ProxyFunctionUpdated(bytes4 indexed selector, address oldImpl, address newImpl);
```

QuoteSignerUpdated

```
/// @dev Raised when `setQuoteSigner()` is called.
/// @param quoteSigner The new quote signer.
event QuoteSignerUpdated(address quoteSigner);
```

RfqOrderFilled

```
/// @dev Emitted whenever an `RfqOrder` is filled.
/// @param orderHash The canonical hash of the order.
/// @param maker The maker of the order.
/// @param taker The taker of the order.
/// @param takerTokenFilledAmount How much taker token was filled.
/// @param makerTokenFilledAmount How much maker token was filled.
/// @param pool The fee pool associated with this order.
event RfqOrderFilled(
    bytes32 orderHash,
    address maker,
    address taker,
    address makerToken,
    address takerToken,
    uint128 takerTokenFilledAmount,
    uint128 makerTokenFilledAmount,
    bytes32 pool
);
```

TransformedERC20

```
/// @dev Raised upon a successful `transformERC20`.
/// @param taker The taker (caller) address.
/// @param inputToken The token being provided by the taker.
///        If `0xeee...`, ETH is implied and should be provided with the call.`
/// @param outputToken The token to be acquired by the taker.
///        `0xeee...` implies ETH.
/// @param inputTokenAmount The amount of `inputToken` to take from the taker.
/// @param outputTokenAmount The amount of `outputToken` received by the taker.
event TransformedERC20(
    address indexed taker,
    address inputToken,
    address outputToken,
    uint256 inputTokenAmount,
    uint256 outputTokenAmount
);
```

TransformerDeployerUpdated

```
/// @dev Raised when `setTransformerDeployer()` is called.
/// @param transformerDeployer The new deployer address.
event TransformerDeployerUpdated(address transformerDeployer);
```

TransformerMetadata

```
/// @dev A transformer that just emits an event with an arbitrary byte payload.
event TransformerMetadata(
    bytes32 callDataHash,
    address sender,
    address taker,
    bytes data
);
```

1.4 Allowances

After the official release, allowances will be set directly on the Exchange V4 Proxy contract. Presently, while we are in beta, allowances should be set on the Allowance Target Address.

The motivation for eliminating the separate Allowance Target in the official release is to reduce transaction costs. Depending on the operational overhead for our integrators, we may support allowances on both the Exchange V4 & the Allowance Target after the offical relase.

1.5 Protocol Fees

1.6 Addresses

Note: This page is auto-generated. See the contract-addresses package for an exhaustive list of contracts across all networks.

1.6.1 Exchange V4

exchangeProxy	0xdef1c0ded9bec7f1a1670819833240f027b25eff
exchangeProxyFlashWallet	0x22f9dcf4647084d6c31b2765f6910cd85c178c18
exchangeProxyGovernor	0x618f9c67ce7bf1a50afa1e7e0238422601b0ff6e
exchangeProxyLiquidityProviderSandbox	0xdb971b18ea5075734cec1241732cc1b41031dfc9
exchangeProxyTransformerDeployer	0x39dce47a67ad34344eab877eae3ef1fa2a1d50bb

1.6.2 Transformers

wethTransformer	0x68c0bb685099dc7cb5c5ce2b26185945b357383e
payTakerTransformer	0x49b9df2c58491764cf40cb052dd4243df63622c7
fillQuoteTransformer	0xfbfb26935f15db6a319a43db5085245a6df1e408
affiliateFeeTransformer	0x4581b59a05ba373b9f67676f66bdb5fcd67e7567

1.6.3 ZRX / Staking

staking	0x2a17c35ff147b32f13f19f2e311446eeb02503f3
stakingProxy	0xa26e80e7dea86279c6d778d702cc413e6cffa777
zrxToken	0xe41d2489571d322189246dafa5ebde1f4699f498
zrxVault	0xba7f8b5fb1b19c1211c5d49550fcd149177a5eaf

1.6.4 Miscellaneous

devUtils	0x74134cf88b21383713e096a5ecf59e297dc7f547
etherToken	0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2
erc20BridgeSampler	0xd8c38704c9937ea3312de29f824b4ad3450a5e61

1.7 ERC20 Transformations

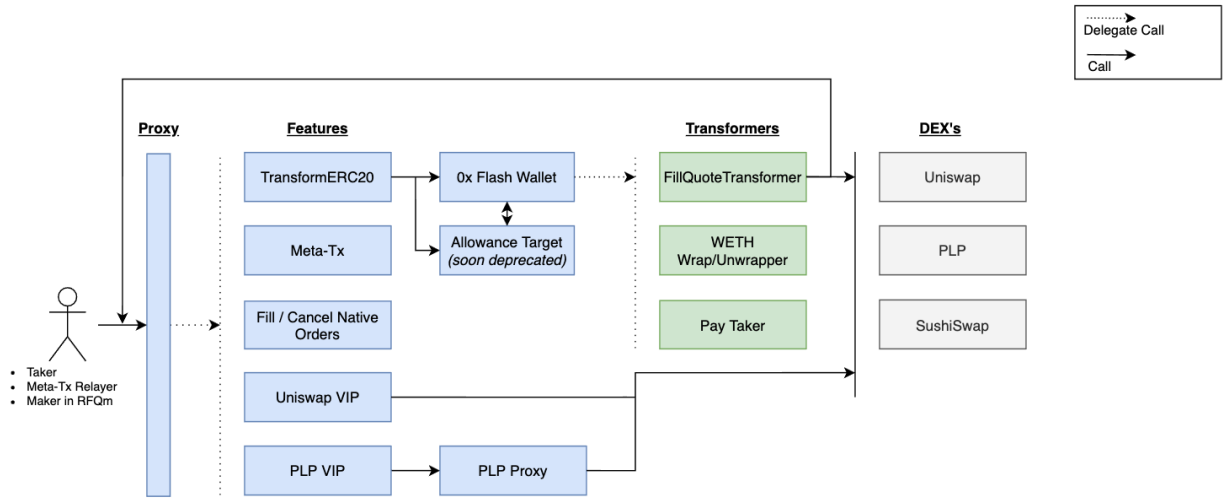
1.8 Aggregation

1.9 WETH Wrapping

1.10 Pluggable Liquidity (PLP)

1.11 Meta-Transactions

1.12 Overview



1.13 Proxy

1.14 Features

1.15 Flash Wallet

1.16 Transformers

1.17 Protocol Fees

1.18 Staking

1.19 Governance

1.20 Voting

1.21 Audits

Below are links to our third-party audit reports.

Release	Reports
Exchange V4	We have an external audit scheduld with Consensys Diligence that will run from November 30th - December 14th, 2020.
Exchange V3	• Trail of Bits • Consensys Diligence (Exchange) • Consensys Diligence (Staking)
Exchange V2.1	• First • Consensys Diligence
MultiAssetProxy	• Consensys Diligence
ERC1155Proxy	• Consensys Diligence
StaticCallProxy	• No third-party audit.
ERC20BridgeProxy	• No third-party audit.

1.21.1 Exchange V4 Audit Scope

Hash:

Areas of Interest:

-

1.22 Bounties

We run an ongoing bug bounty for the 0x Protocol smart contracts! The program is open to anyone and rewards up to **$100,000 for critical exploits**. The scope and disclosure instructions are below.

1.22.1 Rewards

The severity of reported vulnerabilities will be graded according to the CVSS (Common Vulnerability Scoring Standard). The following table will serve as a guideline for reward decisions:

Exploit Score	**Reward**
Critical (CVSS 9.0 - 10.0)	$10,000 - $100,000
High (CVSS 7.0 - 8.9)	$2,500 - $10,000
Medium (CVSS 4.0 - 6.9)	$1,000 - $2,500
Low (CVSS 0.0 - 3.9)	$0 - $1,000

Please note that any rewards will ultimately be awarded at the discretion of ZeroEx Intl. All rewards will be paid out in ZRX.

1.22.2 Areas of Interest

Area	Examples
Loss of funds	• A user loses funds in a way that they did not explicitly authorize (e.g an account is able to gain access to an ``AssetProxy``and drain user funds). • A user authorized a transaction or trade but spends more assets than normally expected (e.g an order is allowed to be over-filled).
Unintended contract state	• A user is able to update the state of a contract such that it is no longer useable (e.g permanently lock a mutex). • Any assets get unexpectedly "stuck" in a contract with regular use of the contract's public methods. • An action taken in the staking contracts is applied to an incorrect epoch.
Bypassing time locks	• The `ZeroExGovernor` is allowed to bypass the timelock for transactions where it is not explicitly allowed to do so. • A user is allowed to bypass the `ZeroExGovernor`.
Incorrect math	• Overflows or underflow result in unexpected behavior. • The staking reward payouts are incorrect.

1.22.3 Scope

The following contracts are in scope of the bug bounty. Please note that any bugs already reported are considered out of scope. See the Audits page for 3rd party security reports.

Release	Contracts	Commit Hash
Exchange V3	• ERC20BridgeProxy.sol (spec) • Exchange.sol (spec) • ZeroExGovernor.sol (spec) • Staking.sol (spec) • StakingProxy.sol (spec) • ZrxVault.sol (spec)	fb8360edfd
Exchange V2.1	• src/2.0.0/protocol • src/2.0.0/utils	ff70c5ecfe
MultiAssetProxy	• MultiAssetProxy.sol	c4d9ef9f83
ERC1155Proxy	• ERC1155Proxy.sol	77484dc69e
StaticCallProxy	• StaticCallProxy.sol	54f4727adc
ERC20BridgeProxy	• ERC20BridgeProxy.sol	281658ba34
ExchangeProxy	• contracts/src	7967a8416c

1.22.4 Disclosures

Please e-mail all submissions to security@0x.org with the subject "BUG BOUNTY". Your submission should include any steps required to reproduce or exploit the vulnerability. Please allow time for the vulnerability to be fixed before discussing any findings publicly. After receiving a submission, we will contact you with expected timelines for a fix to be implemented.

1.23 Contributing

We are an open source project and welcome contributions! Please see the resources below.

Monthly dev call.

1.24 Exceptional ERC20s

Some ERC20s have unique behavior that may require extra handling. We document these here as they are discovered.

1.24.1 Assert vs Require

These ERC20's use *assert* instead of *require*, which means that if the token reverts then (nearly) all of the gas from your transaction will be consumed. Specifically, you are left with 1/64 of the gas limit. Be mindful of this when implementing fallback logic; for example, if a call to *transferFrom* reverts then note you will only have 1/64 of the gas limit to handle the exception.

Known tokens:

- KNC
- LINK
- sUSD
- USDT

1.25 Releases

This page outlines upcoming releases and expected changes.

Release	Est Release Date	Status
Tinker	11/12/20	Deployed on 11/13/20.
Tailor	11/18/20	In audits.
Soldier	12/07/20	Code review.
Sailor	12/07/20	Under development.

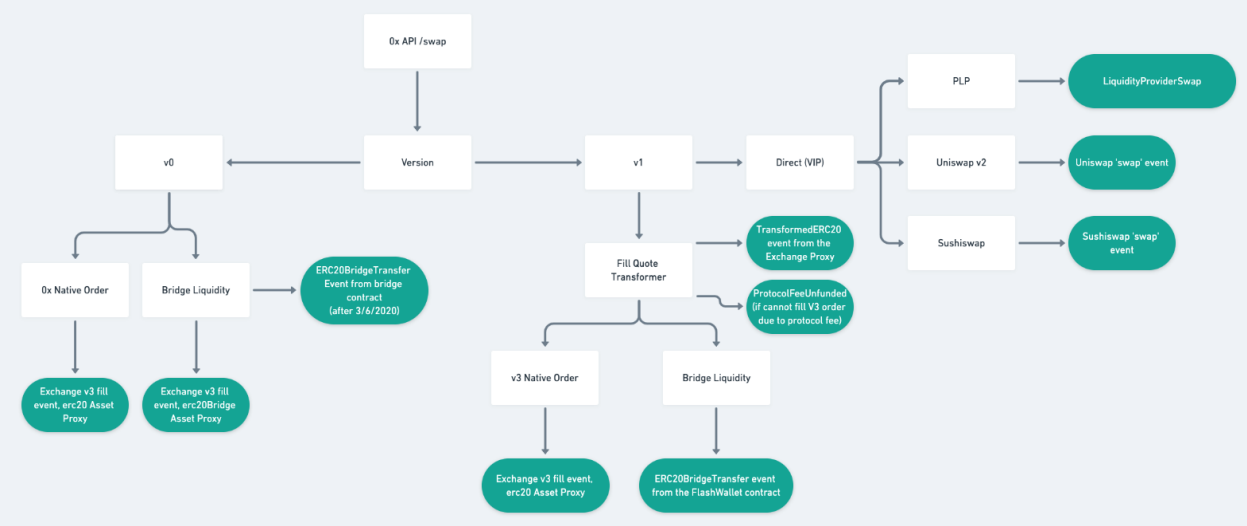
1.25.1 Tinker

- Upgrade features to allow allowances set on the ExhcangeProxy contract. This will fallback to the Allowance Target. (Moved to Sailor)
- Deploy VIP PLP feature. This introduces a new event that is emitted by the Exchange Proxy when filling through the VIP PLP.

```
event LiquidityProviderSwap(
    address inputToken,
    address outputToken,
    uint256 inputTokenAmount,
    uint256 outputTokenAmount,
    address provider,
    address recipient
);
```

1.25.2 Tailor

- PLP instances will no longer emit *ERC20BridgeTransfer* events.



1.25.3 Soldier

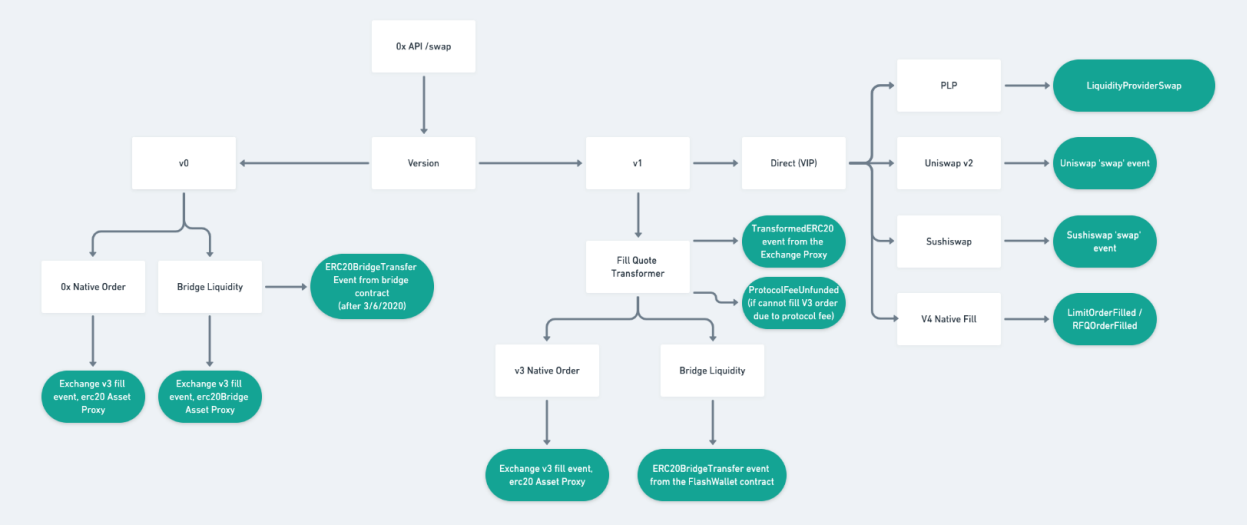
- Deploy feature that implements V4 Limit and RFQ orders (see the Orders Page). This enables us to fill V4 limit orders through the Exchange Proxy, but does not yet allow aggregation.
- New events will be introduced. The proposed events are below.

```
/// @dev Emitted whenever a `LimitOrder` is filled.
/// @param orderHash The canonical hash of the order.
/// @param maker The maker of the order.
/// @param taker The taker of the order.
/// @param feeRecipient Fee recipient of the order.
/// @param takerTokenFilledAmount How much taker token was filled.
/// @param makerTokenFilledAmount How much maker token was filled.
/// @param pool The fee pool associated with this order.
event LimitOrderFilled(
    bytes32 orderHash,
    address maker,
    address taker,
    address feeRecipient,
    address makerToken,
    address takerToken,
    uint128 takerTokenFilledAmount,
    uint128 makerTokenFilledAmount,
    uint128 takerFeeAmount,
    uint128 protocolFeePaid,
    bytes32 pool
);

/// @dev Emitted whenever an `RfqOrder` is filled.
/// @param orderHash The canonical hash of the order.
/// @param maker The maker of the order.
/// @param taker The taker of the order.
/// @param takerTokenFilledAmount How much taker token was filled.
```

(continues on next page)

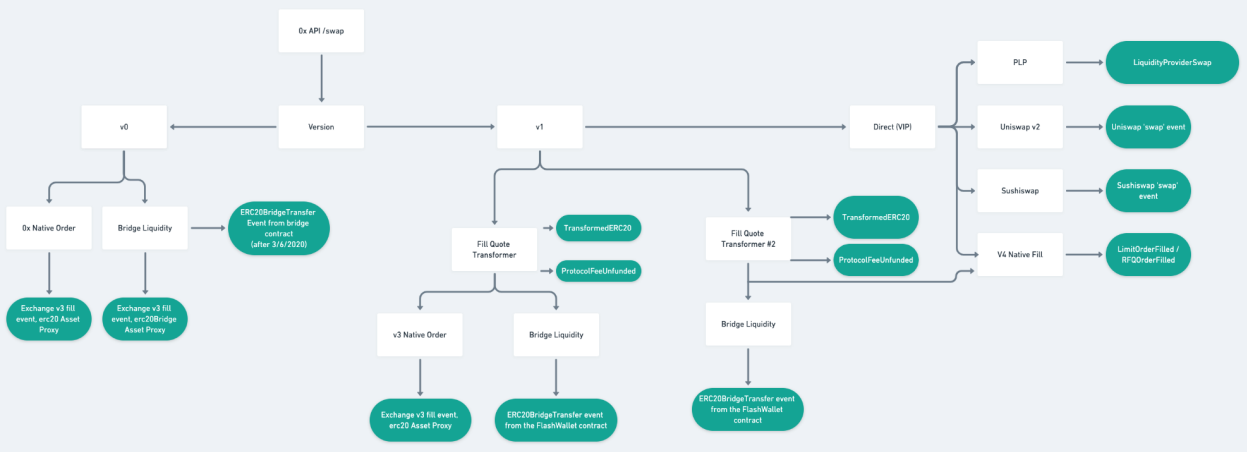
(continued from previous page)

```
/// @param makerTokenFilledAmount How much maker token was filled.
/// @param pool The fee pool associated with this order.
event RfqOrderFilled(
    bytes32 orderHash,
    address maker,
    address taker,
    address makerToken,
    address takerToken,
    uint128 takerTokenFilledAmount,
    uint128 makerTokenFilledAmount,
    uint128 takerFeeAmount,
    uint128 protocolFeePaid,
    bytes32 pool
);

/// @dev Emitted whenever a limt or RFQ order is cancelled.
/// @param orderHash The canonical hash of the order.
event OrderCancelled(
    bytes32 orderHash
);

/// @dev Emitted whenever limit or RFQ orders are cancelled by pair by a maker.
/// @param maker The maker of the order.
/// @param makerToken The maker token in a pair for the orders cancelled.
/// @param takerToken The taker token in a pair for the orders cancelled.
/// @param minValidSalt The new minimum valid salt an order with this pair must
///        have.
event PairOrdersUpToCancelled(
    address maker,
    address makerToken,
    address takerToken,
    uint256 minValidSalt
);
```



1.25.4 Sailor

- A new transformer (like FillQuoteTransformer) that aggregates V4 orders instead of forwarding to Exchange V3.
- This enables us to run simbot trials against V4 before the external audit begins.
- WE DO NOT expect teams to be upgraded to V4 at this point; they can continue using the existing FillQuoteTransformer. At this point teams can begin testing their V4 tooling.
- Upgrade features to allow allowances set on the ExhcangeProxy contract. This will fallback to the Allowance Target.



CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search