Produced for

Polymarket

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Polymarket Team,

Thank you for trusting us to help Polymarket with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Exchange according to Scope to support you in forming an opinion on their security risks.

Polymarket implements a prediction market for real-life events. This audit covers the governance and exchange part of the protocol.

The most critical subjects covered in our audit are functional correctness, access control, and signature handling.

The contracts show a high level of functional correctness and handle signatures correctly.

The general subjects covered are code complexity and gas efficiency.

The code maintains an adequate level of complexity. Gas efficiency is good but could be improved in some cases.

In summary, we find that the current codebase provides a high level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

-Severity Findings	2
•	2
-Severity Findings	1
•	1
-Severity Findings	3
•	3
-Severity Findings	13
•	8
•	2
•	1
•	1
•	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Exchange repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	15 August 2022	a64208031ca71f65d0a93f5c864b5fe5acbb1db0	Initial Version
2	21 September 2022	5a51bcf9744579d7ac060e1c437522075649c12a	Second Version
3	06 October 2022	2fa43ef23b7ad04345e21e9638de56ec9e164a4c	Third Version
4	28 October 2022	c68f93f6d922bd0cd0cd57ac5be278ab0b58ec1c	Fourth Version

For the solidity smart contracts, the compiler version `0.8.15` was chosen.

2.1.1 Included in scope

This report covers `Phase 1-1` of the smart contract audit for Exchange. The following files are in scope for the firstpart of the first phase:

- src/exchange/BaseExchange.sol
- src/exchange/CTFExchange.sol
- src/exchange/interfaces/IAssetOperations.sol
- src/exchange/interfaces/IAssets.sol
- src/exchange/interfaces/IAuth.sol
- src/exchange/interfaces/IConditionalTokens.sol
- src/exchange/interfaces/IFees.sol
- src/exchange/interfaces/IHashing.sol
- src/exchange/interfaces/INonceManager.sol
- src/exchange/interfaces/IPausable.sol
- src/exchange/interfaces/IRegistry.sol
- src/exchange/interfaces/ISignatures.sol
- src/exchange/interfaces/ITrading.sol
- src/exchange/mixins/AssetOperations.sol
- src/exchange/mixins/Assets.sol
- src/exchange/mixins/Auth.sol
- src/exchange/mixins/Fees.sol
- src/exchange/mixins/Hashing.sol

- src/exchange/mixins/NonceManager.sol
- src/exchange/mixins/Pausable.sol
- src/exchange/mixins/PolyFactoryHelper.sol
- src/exchange/mixins/Registry.sol
- src/exchange/mixins/Signatures.sol
- src/exchange/mixins/Trading.sol
- src/exchange/libraries/Calculator.sol
- src/exchange/libraries/OrderStructs.sol
- src/exchange/libraries/PolyProxyLib.sol
- src/exchange/libraries/PolySafeLib.sol
- src/exchange/libraries/SilentECDSA.sol
- src/exchange/libraries/TransferHelper.sol

2.1.2 Excluded from scope

Any contract inside the repository that are not mentioned in `Scope` are not part of this assessment. All external libraries and imports are assumed to behave correctly according to their high-level specification, without unexpected side effects.

Tests and deployment scripts are excluded from the scope.

2.2 System Overview

This system overview describes the initially received version ( ) of the contracts as defined in the Assessment Overview.

Polymarket uses blockchain technology to implement decentralized prediction markets, where users can bet on future outcomes by trading conditional tokens which are redeemable at a future date at a value that depends on an outcome communicated by a decentralized oracle.

Assessment was performed on the `Exchange` subsystem.

2.2.1 Conditional Tokens

The object of trading in the Polymarket protocol are conditional tokens. Conditional tokens represent outcomes of real-life events and can be arbitrarily complex. Exchange utilizes these tokens to create binary markets for specific events: For each event, a pair of tokens representing a mutually exclusive condition such as `YES` and `NO` answers to a question is created. They depend on an external oracle for the settlement of their value. `YES` and `NO` tokens can be minted in equal amounts in exchange for a collateral token, namely USDC. Conversely, equal amounts of complementary tokens (`YES` and `NO` for the same question) can be merged back together, releasing the deposited collateral tokens. When the respective oracle provides an answer to a question, it settles the value at which `YES` and `NO` tokens can be redeemed individually. For binary outcomes, one side of the bet will gain the whole collateral.

2.2.2 Exchange overview

Polymarket implements a hybrid non-custodial exchange for the trading of conditional tokens representing bets in prediction markets. It consists of a centralized marketplace where trading orders are submitted by the users in the form of EIP712 signed messages, and of a smart contract running on the Polygon network handling the filling of matched orders in an open and verifiable way.

The centralized exchange keeps a pool of not filled or partially filled open orders. When an order matching opportunity arises, the order matching is relayed to the `CTFExchange` contract for execution. A matching is a set of orders that can be executed atomically without the exchange incurring deficits. Besides `BUY` and `SELL` order matching, the exchange also enables the matching of equal-type orders. `BUY` orders of complementary conditional tokens (e.g., `YES` and `NO` tokens) can be matched to create a `mint` operation that will mint an equal amount of tokens for both sides of a binary market. Similarly, `SELL` orders of complementary conditional tokens can be matched to create a `merge` operation that burns the tokens and releases the deposited collateral. As orders are cryptographically signed by users, they can only be filled as the user has specified, minimizing the trust requirements on the centralized component of the exchange.

2.2.3 Contract CTFExchange

`CTFExchange` is the on-chain side of the hybrid trading exchange implemented by Polymarket for Conditional Tokens.

To exchange tokens on the exchange, users set an allowance to the `CTFExchange` contract. Users don't interact directly with `CTFExchange`. They instead submit signed EIP-712 compliant orders to a centralized counterparty which has `operator` rights on the exchange. Addresses with the `operator` role can fill signed orders from any signer through the external functions `fillOrder`, `fillOrders`, and `matchOrders`.

`fillOrder` allows an operator to be the counterparty of a single order, and to (partially) fill it. The token requested by the order maker is provided by the operator, and the token offered by the order maker is withdrawn from the order maker and transferred to the operator. This happens at the price requested by the order maker, in the amount specified by the operator. The operator has full discretion over which orders to fill and which to ignore. `fillOrders` is a vectorized version of `fillOrder`.

`matchOrders` allows the operator to fill orders against each other, without the need for funding from the operator. One `taker` order is matched against an arbitrary amount of `maker` orders. Any price improvement is captured by the taker, and orders are never filled at a worse price than what the order maker specified.

2.2.4 Trust Model

Operators in the Exchange have to be fully trusted. All orders are sent off-chain to the operators and they have the power to match different orders with each other in any order or to discard particular orders altogether.

The `Exchange` contract also has one or multiple admins that are the only addresses allowed to register tokens. Additionally, admins are allowed to pause the contract which halts all order processing from operators.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High			
Medium			
Low			

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- : Related to vulnerabilities that could be exploited by malicious actors
- : Architectural shortcomings and design inefficiencies
- : Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	0
-Severity Findings	0
-Severity Findings	0
-Severity Findings	3

- Gas Savings Part 2
- Accidental Token Transfers
- Gas Savings

5.1 Gas Savings Part 2

`Trading._fillFacingExchange` now transfers the fee from the contract to the operator on every call. When multiple maker orders are processed, there is a fee transfer for every one of them. The fee could be sent after all orders have been processed instead.

Acknowledged:

The client acknowledges the possible gas savings and chooses to keep the code as-is.

5.2 Accidental Token Transfers

Tokens that have been accidentally sent to the contract can not be recovered.

Furthermore, if either the collateral token or one of the outcome tokens have been accidentally sent to the contract, the next executed taker order will receive these tokens due to the implementation of `Trading._updateTakingWithSurplus`.

Risk accepted

Polymarket states:

```
Recovering tokens sent to the contract will require adding a
permissioned ``withdrawTokens`` function, which introduces an unacceptably
large trust assumption.
```

5.3 Gas Savings

The following parts can be optimized for gas efficiency:

- The `OrderStructs.OrderStatus` struct occupies 2 words in storage. Decreasing the size of the `remaining` field by 1 byte could reduce the space requirement to 1 word. This fix has to be applied with caution using safe casts where appropriate.
- The field `token` in the `Registry.OutcomeToken` struct is redundant as a specific struct can only be accessed with that value.
- The call to `validateTokenId(token)` in `Registry.validateComplement` is redundant as the very same call is performed in the following call to `getComplement`.
- `Trading._matchOrders` and `_fillMakerOrder` redundantly compute the order hash again, after it has already been computed by `_validateOrderAndCalcTaking`.
- `Trading._updateOrderStatus` performs multiple redundant storage loads of `status.remaining`.
- `Trading._updateTakingWithSurplus` performs a redundant calculation in the return statement. Returning `actualAmount` yields the same result at this point.
- `Assets.getCollateral()`, `Assets.getCtf()`, `Fees.getFeeReceiver()`, `Fees.getMaxFeeRate()` are redundant since the variables they expose are public and already define equivalent accessors.

Code partially corrected:

`OrderStructs.OrderStatus` still occupies 2 storage slots. All other gas savings have been implemented sufficiently.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

-Severity Findings	2

- Signatures Are Valid for Any Address
- ORDER_TYPEHASH Is Incorrect

-Severity Findings	1

- Fee Rate Not Hashed

-Severity Findings	3

- Fee Approval Required
- Unintended Order Types Possible
- Zero Address EOA Signer Considered Valid

-Severity Findings	10

- FeeCharged Event Not Emitted in fillOrder
- OrderStruct.taker Specification Inconsistent
- Code Replication
- Domain Separator Cached
- Floating Pragma
- Non-optimized Libraries Used
- Order Status Possibly Incorrect
- Struct Order Has Redundant Fields
- Wrong Notice on Order.feeRateBps
- isCrossing Incorrect When takerAmount Is 0

6.1 Signatures Are Valid for Any Address

`Signatures.isValidSignature` checks the validity of a given order's signature. For signature types `POLY_GNOSIS_SAFE` and `POLY_PROXY`, the code makes sure that an order's `maker` address belongs to the same account that signed the order.

This is not true for the signature type `EOA`. Any account can create a signature for an order that contains an arbitrary `maker` address. Since users give token approval to the protocol on order creation, malicious actors can generate orders for an account that already generated an order, but, for example, with a more favorable price. This order will then be executable although the account in question did not authorize it.

Code corrected

`Signatures.verifyEOASignature` has been added, which additionally ensures that the `Order.maker == Order.signer` for EOAs.

6.2 ORDER_TYPEHASH Is Incorrect

The `ORDER_TYPEHASH` in `OrderStructs` does not equal the actual encoded data in `Hashing.hashOrder`. It is used to calculate an EIP-712 compliant hash for an order which is then used to recover the signer of the given order. Since the typehash is incorrect, this mechanism will not work for correctly signed orders.

Code correct

`OrderStructs.ORDER_TYPEHASH` is now computed at compile time on the correct structure signature.

6.3 Fee Rate Not Hashed

`Hashing.hashOrder` does not include the fee rate of an order into the hash. If the signatures are also generated this way and users do not recognize this, operators can always specify `MAX_FEE_RATE_BIPS` fees.

Code correct

Order hash computation now includes `feeRateBps`.

6.4 Fee Approval Required

Fees are charged by transferring the respective amount of tokens from the *receiving user's account* to the fee receiver.

The user has to give additional approval for the token they actually want to receive, which is counter-intuitive and also opens up additional security risks. Since the fee is always smaller than the amount of tokens sent to the user, this special behavior is not necessary as the fees could also be deducted from the amount sent to the user.

Code correct

Fees are deducted directly on the exchange, instead of being pulled from the order maker. Additionally, `_fillOrder` implicitly collects fees by transferring the taking amount minus the fee from the operator.

6.5 Unintended Order Types Possible

`Trading._matchOrders` and `_fillOrder` miss sanity checks for combinations of `makerAssetId`, `takerAssetId` and `side` in the passed order structs.

Eight combinations of `side` in `[BUY, SELL]`, `makerAssetId` in `[ConditionalToken, Collateral]`, and `takerAssetId` in `[ConditionalToken, Collateral]` are possible, but only two of them should be allowed. This seems possible as the side seems redundant or colliding with the combinations (struct Order has redundant fields).

This allows for matching of orders that are not intended. For example, matching of a BUY order with maker asset YES and taker asset USDC to a SELL order with maker asset USDC and taker asset YES is perfectly possible as long as the YES price in these orders is over 1 USDC (otherwise, the fee calculation reverts).

Code correct

Unintended order types are no longer possible as fields `makerAssetId` and `takerAssetId` have been replaced by a single field `tokenId`.

6.6 Zero Address EOA Signer Considered Valid

`isValidSignature()` returns `true` for signer equal to zero address, signatureType `EOA` and invalid signature.

The check is performed at line `70` of `Signature.sol`, `SilentECDSA.recover` returns `0` on error. Setting the `signer` to zero address will incorrectly validate the signature.

Code correct

Signature verification now uses Openzeppelin's `ECDSA.recover` instead of `SilentECDSA`. Invalid signatures now revert instead of returning the 0-address.

6.7 FeeCharged Event Not Emitted in fillOrder

in `_fillOrder`, the fee is charged implicitly by deducting it from the amount that is transferred to the order maker but a `FeeCharged` event is not emitted. For consistency and to allow proper accounting based on events, `FeeCharged` should be emitted.

Code corrected:

The event is now emitted.

6.8 OrderStruct.taker Specification Inconsistent

The `taker` field of the `Order` struct actually identifies the operator which can fill the order, not the taker that can be matched with the order as it seems to be intended from the natspec notice.

Specification changed:

The natspec of `taker` has been modified to reflect its actual usage.

6.9 Code Replication

`Trading._fillOrder` contains the same code that is already present in `_validateOrderAndCalcTaking`. For maintainability reasons, code replications should be avoided.

Code correct

Duplicated code in `Trading._fillOrder` has been refactored into the function `_performOrderChecks` (renamed from `_validateOrderAndCalcTaking`).

6.10 Domain Separator Cached

`Hashing` exposes the `domainSeparator` with an implicit public getter for an immutable variable. When the chain id changes, for example due to a hardfork, the `domainSeparator` will not be correct on the new chain.

Code corrected:

The `domainSeparator` is now re-calculated in case of a change of the chain id.

6.11 Floating Pragma

Exchange uses the floating pragma `<0.9.0`. Contracts should be deployed with the compiler version and flags that were used during testing and auditing. Locking the pragma helps to ensure that contracts are not accidentally deployed using a different compiler version and help ensure a reproducible deployment.

Code correct

Solidity version has been fixed to `0.8.15` in all instantiated contracts. Interfaces, libraries, and abstract contracts are left floating.

6.12 Non-optimized Libraries Used

- `TransferHelper` re-implements transfer functions while there already exists an optimized library (`SafeTransferLib`) implementing these functions in the dependencies of the project.
- `Signatures` uses `SilentECDSA`, a modified version of an outdated OpenZeppelin `ECDSA` version. The current version of this library could be used instead since it provides all required functionalities.

Code corrected:

- `TransferHelper` now utilizes the optimized library for transfer functions.
- `Signatures` utilizes the OpenZepplin library.

6.13 Order Status Possibly Incorrect

`Trading.getOrderStatus` returns an `OrderStatus` struct containing a variable `isCompleted` for any order hash. As the protocol has two distinct mechanisms of invalidating orders, this function might return that an order is still not completed, while in fact it has been invalidated by a nonce increase.

Code corrected:

The field `isCompleted` has been renamed to `isFilledOrCancelled` which describes the behavior in an adequate way.

6.14 Struct Order Has Redundant Fields

In `struct Order`, the fields `side`, `makerAssetId`, and `takerAssetId` coexist redundantly.

If `side` is `BUY`, `makerAssetId` is implied to be `0`. If `side` is `SELL`, `takerAssetId` is implied to be `0`. a single `AssetId` field would therefore be sufficient to fully specify the order, or similarly `side` can be removed from the struct and be derived from `makerAssetId` and `takerAssetId`.

Redundant input arguments increase code complexity and facilitate potential bugs.

Code corrected:

The fields `makerAssetId` and `takerAssetId` have been removed in favor of a new field `tokenId`.

6.15 Wrong Notice on `Order.feeRateBps`

The notice of `feeRateBps` says:

```
If BUY, the fee is levied on the incoming Collateral
```

However, the fee is always charged in the `takerAssetId`, which is not necessarily the collateral.

Specification changed:

The notice for `feeRateBps` now reads: `Fee rate, in basis points, charged to the order maker, charged on proceeds.`

6.16 isCrossing Incorrect When takerAmount Is 0

In the event of a SELL-SELL matching, where tokens should be merged to provide collateral back to the sellers, `CalculatorHelper.isCrossing` returns `true` when at least one side's order has `takerAmount == 0`. In the case that the other side's order has a price greater than `ONE`, the matching is not crossing since no sufficient amount of collateral can ever be redeemed to cover `price > ONE`, however the `isCrossing` returns true.

Code corrected:

`isCrossing` now returns `false` in the mentioned cases.