PRIVATE

Code Assessment

of the Governance and Exchange Smart Contracts

27 September, 2022

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 Governance and 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.

Critical-Severity Findings	2
• Code Corrected	2
High-Severity Findings	2
• Code Corrected	2
Medium-Severity Findings	3
• Code Corrected	3
Low-Severity Findings	16
• Code Corrected	6
• Specification Changed	2
• Code Partially Corrected	1
• Risk Accepted	1
• No Response	6

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 Governance and 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

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

2.1.1 Included in scope

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

```
src/gov/governor/GovernorBravoDelegate.sol
src/gov/governor/GovernorBravoDelegator.sol
src/gov/governor/Timelock.sol
src/gov/governor/GovernorBravoInterfaces.sol
src/gov/governor/GovernorBravoInterfaces.sol
src/gov/governor/SafeMath.sol
src/gov/poly/Poly.sol

src/common/auth/Owned.sol

src/gov/bridge/Broadcaster.sol
src/gov/bridge/Receiver.sol
src/gov/bridge/interfaces/IBroadcaster.sol
src/gov/bridge/interfaces/IReceiver.sol

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 (**Version 1**) 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 two of the subsystems that will implement the next iteration of the Polymarket protocol:

1. Governance
2. Trading Exchange

2.2.1 Governance overview

The Governance system allows holders of the POLY token to vote on proposals pertaining to the improvement of the Polymarket protocol. It runs on Ethereum Mainnet, where the POLY token is also minted and distributed. It is implemented as a fork of Compound's GovernorBravo governance mechanism, with minimal changes related to governance token parameters, token name, and ease of use. Votes on governance proposals are held on the Ethereum Mainnet by holders of the POLY token and succeeding proposals can be executed on the Polygon network, where most of the Polymarket protocol runs, through an Ethereum-to-Polygon unidirectional bridge built on Polygon's Fx-Portal.

The governance is divided in three subsystems: the Governor, the Governance Token, and the Bridge, for a total of six smart contracts:

1. Governor
    1. GovernorBravoDelegate
    2. GovernorBravoDelegator
    3. Timelock
2. Governance Token: POLY
    1. Poly
3. Bridge
    1. Broadcaster
    2. Receiver

2.2.2 Contract GovernorBravoDelegate

Contract GovernorBravoDelegate is where the logic of the governance mechanism is implemented. It allows holders of the governance token POLY to submit new proposals. A proposal is implemented as a list of external calls to arbitrary contracts that will be executed by the Timelock in the case of a successful proposal. After an initial votingDelay, holders of POLY can cast votes on the proposal and the votes last for votingPeriod blocks. Votes are weighted by the amount of tokens held by each voter or delegated by other POLY owners to the voter at the time of the voting start. After votingPeriod blocks have elapsed since the voting has started, voting is suspended, and the proposal is considered defeated if the for votes are less than the against votes or less than quorumVotes. Otherwise, the proposal has succeeded. A successful proposal can be be sent to the Timelock, where it finally can be executed after the Timelock 's delay has elapsed.

GovernorBravoDelegate is a fork of Compound's GovernorBravoDelegate.sol (https://github.com/compound-finance/compound-protocol/blob/master/contracts/Governance/GovernorBravoDelegate.sol), with minimal modifications. The differences with the original version are restricted to the token name, constants related to the token supply, and code linting.

2.2.3 Contract GovernorBravoDelegator

Contract GovernorBravoDelegator is the entry point for calls to the Governor. It acts as a delegatecall proxy with upgradeable implementation. It has a privileged role admin which is allowed to upgrade the implementation contract. A proper deployment of this contract will set admin to the Timelock address so that the governor's implementation can only be upgraded through a succesfull proposal. Calls to this contract other than _setImplementation and public accessors are forwarded with delegatecall to GovernorBravoDelegate.

2.2.4 Contract `Timelock`

The role of `Timelock` is to hold the administrative rights for every system over which the Governance has authority and to execute Governance proposals. Proposals are queued by the Governor after a successful vote and can be executed after a set delay. This delay allows the users to adapt for the proposal outcome and allows the governance to use another proposal to cancel a queued but not executed proposal.

2.2.5 Contract `Poly`

`Poly` implements the governance token for the Polymarket Governance mechanism. It is an ERC20 token with constant total supply, minted on deployment to an account chosen by the deployer. It holds checkpoints for every transfer to allow querying the balance of an account at any point in history for correct vote counting. Supplemental methods in addition to the ERC20 standard are:

1. `delegate()` to delegate one's voting power to another user without transferring the tokens.
2. `delegateBySig()` as before but it can be called on another's user behalf with a signature.
3. `getPriorVotes()` to query an account's voting power at a specific block number.
4. `getCurrentVotes()` to query an account's voting power.

`Poly` is a fork of Compound's `COMP` token, but differently to the original version, it defaults to self-delegation. This provides an easier experience for users, who have to delegate to themselves in order to vote in the original `COMP` implementation, at the cost of more storage writes for every transfer as checkpointing happens even for users not interested in voting.

2.2.6 Contract `Broadcaster`

`Broadcaster` is the Ethereum facing side of the Polymarket bridge between Ethereum Mainnet and Polygon. `Broadcaster` is owned by `Timelock` which can call its `sendMessageToChild()` method. This method allows messages from the governance to be delivered to the `Receiver` contract on Polygon.

2.2.7 Contract `Receiver`

`Receiver` is the Polygon facing side of the Polymarket bridge between Ethereum Mainnet and Polygon. `Receiver` receives messages by governance relayed by the Polygon Fx-Portal mechanism. Message hashes are marked as executable when received. A non-permissioned method `execute` allows to execute the messages in the order they were sent from governance.

2.2.8 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. Governance and 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.

2.2.9 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.

2.2.10 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.

2.2.11 Trust Model

Deployment is assumed to happen correctly. On Ethereum Mainnet, admin rights for the `Timelock` are set to `GovernorBravoDelegator`, and administrative rights on `GovernorBravoDelegator` are set to the `Timelock`. After initial distribution of tokens of `Poly`, owners are assumed to be untrusted, but we assume that the majority of the voters is not malicious. `Broadcaster` is assumed to be owned by `Timelock`.

On Polygon network, `Receiver` is assumed to be the administrator of Polymarket systems. Users with the role of `operator` of `CTFExchange` are untrusted, but it is assumed that they will not censor orders.

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	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	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:

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

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	8

5.1 FeeCharged Event Not Emitted in fillOrder

Design Low Version 2

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.

5.2 Gas Savings Part 2

Design Low Version 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.

5.3 Non-zero Value Transactions From Receiver Need Funds in the Same Call

Design Low Version 2

Calls executed in the `Receiver` with non-zero values require the caller of the `execute` function to provide the needed native token amount. `execute` is supposed to be unpermissioned and callable by anybody, but requiring the caller to provide the value defeats this design point. Adding a `receive() payable` method to `Receiver` will allow keep a pool of native tokens on the contract so that calls with values can be performed without the need of a centralized entity as long as the pool remains liquid.

5.4 OrderStruct.taker Specification Inconsistent

Correctness Low Version 2

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.

5.5 Unchecked Message Array Lengths

Design Low Version 2

Neither `Broadcaster` nor `Receiver` check that the different arrays in a message are of the same length. If, for example, the message is constructed with one `target` missing, the message can be executed successfully although this might have undesired consequences.

5.6 isCrossing Incorrect When takerAmount Is 0

Correctness Low Version 1

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.

5.7 Accidental Token Transfers

Design Low Version 1 Risk Accepted

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.
```

5.8 Gas Savings

Design | Low | Version 1 | Code Partially Corrected

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` perfroms 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.

Critical-Severity Findings	2

- Signatures Are Valid for Any Address Code Corrected
- ORDER_TYPEHASH Is Incorrect Code Corrected

High-Severity Findings	2

- Bridge Nonce Validity Is Dependent on Proposals Success and Ordering Code Corrected
- Fee Rate Not Hashed Code Corrected

Medium-Severity Findings	3

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

Low-Severity Findings	8

- Anybody Can Initialize Receiver Code Corrected
- Code Replication Code Corrected
- Floating Pragma Code Corrected
- Non-optimized Libraries Used Code Corrected
- Order Status Possibly Incorrect Code Corrected
- Struct Order Has Redundant Fields Code Corrected
- Wrong Comments Specification Changed
- Wrong Notice on Order.feeRateBps Specification Changed

6.1 Signatures Are Valid for Any Address

Security Critical Version 1 Code Corrected

`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

Correctness Critical Version 1 Code Corrected

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 Bridge Nonce Validity Is Dependent on Proposals Success and Ordering

Design High Version 1 Code Corrected

Messages sent from governance to Broadcaster must have increasing sequential nonces, which implies that different proposals that will be executed on Polygon cannot be voted concurrently.

If proposal `A` with nonce `1` fails and proposal `B` with nonce `2` succeeds, then `B` will have the wrong nonce on Receiver because `A` was not sent.

Code corrected

Bridge `Receiver.sol` has been rearchitected to receive messages composed of multiple transactions. Transactions in a message are executed atomically in the order specified in the message. There is no global ordering of messages anymore, just of transactions within a message. The nonce mechanism has therefore been removed completely.

6.4 Fee Rate Not Hashed

Design High Version 1 Code Corrected

`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`.

6.5 Fee Approval Required

Security Medium Version 1 Code Corrected

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.

6.6 Unintended Order Types Possible

Design Medium Version 1 Code Corrected

`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`.

6.7 Zero Address EOA Signer Considered Valid

Correctness Medium Version 1 Code Corrected

`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.

6.8 Anybody Can Initialize `Receiver`

Security Low Version 1 Code Corrected

The `Receiver` contract inherits from `FxBaseChildTunnel`, which exposes `setFxRootTunnel(...)` as a unpermissioned public method. Contract deployment and `setFxRootTunnel(...)` initialization happen in separate transactions. A malicious actor could call `setFxRootTunnel()` with an arbitrary argument after contract creation and before legitimate initialization is completed.

Code correct

`Receiver` contract is now `Ownable`. The deployer is the owner of `Receiver`. `setFXRootTunnel` is guarded by the `onlyOwner` modifier. The owner has no further privileges except calling `setFXRootTunnel` once.

6.9 Code Replication

Design Low Version 1 Code Corrected

`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 Floating Pragma

Design Low Version 1 Code Corrected

Governance and 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.

6.11 Non-optimized Libraries Used

Design Low Version 1 Code Corrected

- `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.

6.12 Order Status Possibly Incorrect

Design Low Version 1 Code Corrected

`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 an an adequate way.

6.13 Struct Order Has Redundant Fields

Design Low Version 1 Code Corrected

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`.

6.14 Wrong Comments

Design Low Version 1 Specification Changed

- `GovernorBravoDelegate.initialize` describes the `poly_` argument with "The address of the COMP token".

- `GovernorBravoDelegate` contains comments for some constants that highlight the amount of governance tokens . These amounts are denominated in "PM" instead of "POLY".

Specification changed:

All mentioned comments have been changed to the correct symbols.

6.15 Wrong Notice on `Order.feeRateBps`

Correctness Low Version 1 Specification Changed

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.`

7 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

7.1 Domain Separator Cached

Note Version 1

`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.

7.2 Redundant Field in Receiver

Note Version 2

The struct `Message` in `Receiver` contains the fields `executable` and `executed`. Both fields are always set in conjunction with each other (to opposing values). Therefore, one of the fields is redundant.