PUBLIC

Code Assessment

of the Conditional Tokens Smart Contracts

11 April, 2024

Produced for

Polymarket

by

CHAINSECURITY

Contents

1 Executive Summary

Dear all,

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 Conditional Tokens according to Scope to support you in forming an opinion on their security risks.

Polymarket uses gnosis conditional tokens to represent positions in prediction markets with binary outcomes.

The most critical subjects covered in our audit are functional correctness and the resilience of elliptic curve calculations used in ID computation.

Security regarding functional correctness is high. Furthermore, the possibility of negating IDs on the used elliptic curve (and the subsequent possibility if creating "all-purpose" tokens) does not pose a security risk within the conditional token framework but adds additional complexity that should be taken into consideration when using conditional tokens (see Infinite minting of position tokens with no value).

In summary, we find that the 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	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

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 Conditional Tokens 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	17 September 2020	eeefca66eb46c800a9aaab88db2064a99026fde5	Initial Version

For the solidity smart contracts, the compiler version `^0.5.1` was chosen.

2.1.1 Included in scope

This report covers the gnosis conditional tokens contracts.

- contracts/ERC1155/ERC1155.sol
- contracts/ERC1155/ERC1155TokenReceiver.sol
- contracts/ERC1155/IERC1155.sol
- contracts/ERC1155/IERCTokenReceiver.sol
- contracts/ConditionalTokens.sol
- contracts/CTHelpers.sol
- contracts/Migrations.sol

2.1.2 Excluded from scope

Any contracts 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.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Conditional tokens are an ERC-1155 representation of prediction markets with arbitrary complexity. Polymarket uses these conditional tokens to represent such prediction markets with binary outcomes.

2.2.1 ConditionalTokens

`ConditionalTokens` is an ERC-1155 contract that allows users to create tokens tied to questions with up to 256 outcomes (conditions). Each position a user can hold is identified by a unique `positionId` that corresponds to an ERC-1155 token id. A position is tied to a collateral token and a `collectionId`. A collection is a condition tied to a subset of all possible outcomes.

A user can call `prepareCondition()` by defining an oracle address, a unique identifier `questionId`, and the number of outcome slots the condition has. This condition can then only be resolved by the oracle.

Once the condition is prepared, the user can call `splitPosition()` by defining a collateral token and a valid outcome partition (at least two different disjoint subsets of the total outcome set covering all available outcome slots) for the given condition. The function then mints one outcome token of each partition per supplied collateral.

For example, a user splitting on a condition with 2 outcomes and providing one collateral token will receive one token for each outcome. For conditions with more than two outcomes, the user can also split a position into at least two distinct subsets of outcomes. For example, for a condition with 3 outcomes ($A|B|C$), a user can split one collateral token into one A token and one $B|C$ token. Furthermore, the $B|C$ token can then also be split into one B and one C token.

If a user holds a position for each outcome of a condition, they can call `mergePositions()` to redeem the underlying collateral token. For conditions with more than two outcomes, it is possible to merge a subset of outcomes. For example, for a condition with 3 outcomes ($A|B|C$), a user can merge one A and one B token to receive one $A|B$ token.

A condition can only be resolved by the oracle of the condition using `reportPayouts()`. This function will set the outcome of the condition and set the payout vector for each outcome of the condition (`payoutNumerators` and `payoutDenominator`). The payout for each outcome is a fraction of the total collateral tokens. The total payout ratio for all outcomes must be 1.

Once the condition is resolved, the positions can be redeemed using `redeemPositions()`. For each position, the user will receive the payout set by the oracle for that specific outcome.

A position in the conditional token contract is represented by a `positionId`. A position is tied to a collateral token and a `collectionId`, which is computed from a `conditionId` and an outcome. A collection is therefore a set of outcomes for a condition.

The contract also offers the possibility to combine multiple conditions into a single position allowing for positions that have to correctly predict the outcome of multiple questions in order to receive a payout.

For this purpose, collection IDs are computed by adding the hash of a condition and an outcome set to the collection ID of another condition on an elliptic curve. This approach ends up with the same position ID no matter from which condition the first split is performed. For such composite conditions, the contract's functions work slightly differently:

- `splitPosition()` burns the tokens of the "parent" condition instead of transferring collateral from the user.
- `mergePositions()` and `redeemPositions()` mint "parent" condition tokens instead of transferring collateral to the user.

2.2.2 Contract `CTHelpers`

`CTHelpers` offers helper functions to compute condition IDs, collection IDs and position IDs.

A `conditionId` is computed as the keccak256 hash of the oracle, a question ID and the number of outcomes.

A `collectionId` is computed as the keccak256 hash of a `conditionId` and an index set of outcomes. The index set is a bit array where each bit represents an outcome. For example, a condition with 3 outcomes ($A|B|C$) will have a 3-bit index set.

For a combination of collections, the `collectionId` is computed as a multi-hash set of `collectionId` points on the `bn_128` elliptic curve. This is done to leverage the commutative property of point addition on elliptic curves over finite fields, which allows the combination of collections in any order. To compute the corresponding elliptic curve point of the `collectionId`, the first 254 bits of the ID are used as the x coordinate, the 255th bit is used as a parity bit and the 256th bit is not used. The y coordinate is then computed using the elliptic curve equation.

A `positionId` is a `collectionId` tied to a collateral token. It is computed as the keccak256 hash of the `collectionId` and the collateral token address.

2.2.3 Polymarket Conditional Token Usage

Polymarket uses conditional tokens to represent conditions with binary outcomes. Furthermore, Polymarket does not create combinatorial markets over multiple conditions. This means that only conditions with 2 outcomes are created with `parentCollectionId` set to 0.

2.2.4 Trust Model

Anyone can create a condition with `prepareCondition()`. However, only the `oracle` provided during condition preparation can resolve it by calling `reportPayouts()`. This means that the oracle has the power to set the outcome of a condition and the payout for each position. The oracle is a trusted party and the system relies on the oracle to resolve conditions correctly.

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 our findings. The findings are split into these different categories:

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	0

6 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.

6.1 Infinite Minting of Position Tokens With No Value

Note Version 1

For any resolved condition it is possible to mint an infinite amount of tokens for the losing outcomes from a single 0 token (i.e., a token with a collection ID of `0x0`). For this to work, one outcome must receive all the payout and the other outcomes must receive none.

For example, for a condition with binary outcomes (*True*|*False*), if the condition resolves to *True*, it is possible to mint an infinite amount of *False* tokens from a single 0 token. Because the *False* tokens have no value at this point, it is not an issue by itself but it should always be taken into account when designing a market around conditional tokens.

We will now explain how to mint an infinite amount of losing tokens from a single 0 token.

From a previous audit on the conditional tokens contracts, we learn how to obtain one 0 token from one collateral token (Chapter 6). Furthermore, it is stated that a 0 token, tied to collateral token CT, can be used with any position tied to CT.

In case a particular market is created using Polymarket's `NegRiskAdapter` (instead of directly working with the `ConditionalTokens` contract), the conversion of a 0 token to a token representing a Polymarket condition must be tied to wrapped collateral. Polymarket uses wrapped collateral tokens as collateral for conditional tokens. Users can obtain wrapped collateral tokens by redeeming conditional tokens directly on the used `ConditionalTokens` instead of the adapter.

We assume that we have a condition with a `conditionId` C with two possible outcomes (*True*|*False*) and that the condition has resolved to *True*. Furthermore, the notation `H(T)` represents the elliptic curve multi-hash set of `conditionId` C with outcome *True* while `H(F)` represents the same for outcome *False*.

The first step of converting a 0 token to a *False* token is a call to `redeemPositions()` with the following arguments:

- `collateralToken`: The collateral token CT used when creating the 0 token.
- `parentCollectionId`: $-H(T)$, the additive inverse of $H(T)$ on the given elliptic curve.
- `conditionId`: C.
- `indexSets`: `[0b01]`.

Because the condition has resolved to *True*, the function will convert one 0 token (the result of the calculation $-H(T) + H(T)$ that is performed in `redeemPositions()` to calculate the `collectionId` of the tokens that should be redeemed) to one $-H(T)$ token. We note that redeeming to $-H(F)$ tokens is not possible as it would require redeeming an index set of $0b10$ which does not have a payout vector.

The next step is a call to `splitPosition()` with the following arguments:

- `collateralToken`: CT.
- `parentCollectionId`: $-H(T)$.
- `conditionId`: C.
- `partition`: `[0b01, 0b10]`.

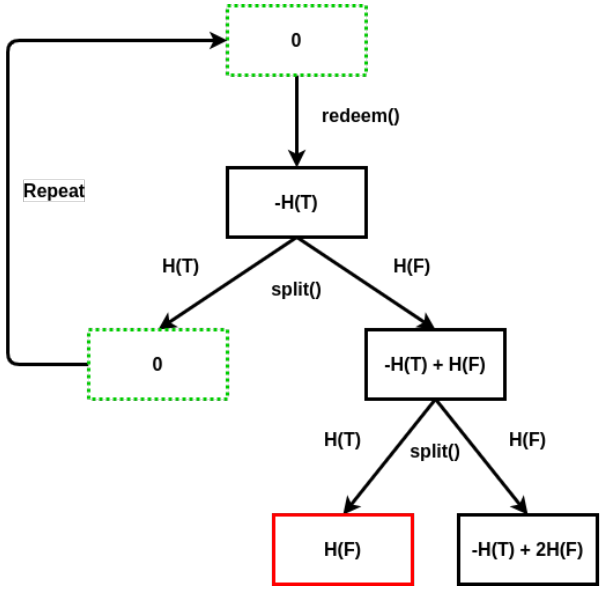
- `amount`: The amount of obtained $-H(T)$ tokens.

Equal amounts of 0 tokens and one $-H(T) + H(F)$ tokens are created. The new 0 tokens can be re-used with the first step of this conversion.

$-H(T) + H(F)$ tokens can be split in another step to equal amounts of $H(F)$ and $-H(T) + 2H(F)$ tokens using `splitPosition()`:

- `collateralToken`: CT.
- `parentCollectionId`: $-H(T) + H(F)$.
- `conditionId`: C.
- `partition`: `[0b01, 0b10]`.
- `amount`: The amount of obtained $-H(T) + H(F)$ tokens.

The following graphic gives an overview of the conversion steps:



We notice that from a single 0 token we can obtain an infinite amount of $H(F)$ tokens, since a single 0 token can be transformed to a 0 and a $H(F)$ token. The same issue arises for conditions with more than two outcomes where all losing outcome tokens can be minted infinitely from a single 0 token.

It is therefore important that, once a condition is resolved in the `ConditionalTokens` contract, there should no longer exist any arbitrage opportunities for the respective tokens and no external contract's accounting should rely on the token amounts to be correct.

6.2 No Irregular ERC-20 Tokens Supported

Note **Version 1**

The `ConditionalTokens` contract does not support ERC-20 tokens that do not return values in the transfer/approval functions (e.g., USDT).