PUBLIC

Code Assessment

of the UMA Sports Oracle Smart Contracts

5 June, 2025

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 UMA Sports Oracle according to Scope to support you in forming an opinion on their security risks.

Polymarket implements UMA Sports Oracle, a smart contract that allows the creation of conditional tokens for bets on sport games. Several types of bets are possible for a given game. UMA Sports Oracle allows querying the game result only once in the UMA Optimistic Oracle to resolve the outcome of several bets.

The most critical subjects covered in our audit are external integrations with UMA and the Conditional Token Framework, solvency such that refunds from a game are not spent by another game, and functional correctness of the state transitions, also with respect to asynchronous callbacks and admin actions. Security regarding all the aforementioned subjects is high.

The general subjects covered are documentation, testing, missing refunds, Denial of Service attacks and front-running. Security regarding all the aforementioned subjects is high. Documentation regarding the state transitions is expanded out in the following System Overview. During the review, an issue in the UMA protocol has been uncovered that could have affected the Polymarket contracts. The issue has been resolved with the collaboration of UMA.

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	1
• Code Corrected	1
Low-Severity Findings	3
• Code Corrected	3

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 UMA Sports Oracle 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	18 January 2025	cfd11de0f83f247817420a9facd946022f6e1b24	Initial Version
2	10 Februrary 2025	72c081a446d14b0f1f72522870502a76e69c0ecd	Second Version
3	11 March 2025	bcdad43734cf94a3a8c0a5020b10edc7df4f2450	Minor fixes

For the solidity smart contracts, the compiler version `0.8.27` was chosen.

The following files are in scope:

```
src/UmaSportsOracle.sol
src/modules/Auth.sol
src/libraries/AncillaryDataLib.sol
src/libraries/LineLib.sol
src/libraries/PayoutLib.sol
src/libraries/ScoreDecoderLib.sol
src/libraries/Structs.sol
```

2.1.1 Excluded from scope

Third-party dependencies, testing files, and any other files not listed above are outside the scope of this review.

2.2 System Overview

This system overview describes the latest received version (Version 2) of the contracts as defined in the Assessment Overview.

Polymarket implements an Oracle Contract resolving multiple Sports Markets from a single UMA Oracle request.

The `UmaSportsOracle` allows the permissionless deployment of `Game` - describing a sports event - and wraps oracle requests to UMA's Optimistic Oracle to handle their resolution. For each game, multiple `Markets` can be permissionlessly created, each encoding their own winning conditions that resolve based on the game's outcome. Each *market* is connected to a question in Polymarket's Condition Token Framework with the `UmaSportsOracle` functioning as its resolution source. The question is prepared when a market is created and payouts are reported when the market is resolved.

2.2.1 Game

Games are permissionlessly deployed with function `createGame` that takes as argument `ancillaryData`, which specifies an oracle request describing the game to the UMA Optimistic Oracle. The data must be encoded according to UMIP-183 and formatted as JSON, with the `UmaSportsOracle` appending the address of the message sender.

The ancillaryData (with appended sender address) is the unique identifier of any *game*, and hashes into the game ID under which the `UmaSportsOracle` stores game properties in a local mapping. Furthermore, the `UmaSportsOracle` keeps track of the game's state, the liveness of oracle request for the game, the bond required to propose a result in the Uma Optimistic Oracle, the reward for proposing a result and the timestamp of the current UMA oracle request. For any game, multiple oracle requests can be made to UMA. A new oracle request uses the same parameters as the previous request except for the timestamp of the oracle request. So in case a new request is made only the timestamp is updated in the local mapping.

2.2.2 Multiple Values

The game contract makes a special request to the UMA Optimistic Oracle of type `MULTIPLE_VALUES` that allows the oracle to return multiple values in a single request as specified by UMIP-183. A request consists of three parts: the title of the request, a detailed description of how the oracle request must be resolved, and a list of labels that describe each of the values returned.

```
{
title: string;
description: string;
labels: string[];
}
```

The sender address appended by the `UmaSportsOracle` is ignored by the proposer of the result, or UMA voters in case of dispute, since it is not part of the JSON object. The UMA Oracle returns the result of the request as a single int256 value, so the UMA oracle consumer must decode the value into up to seven return values with each being 32 bits. The last 32 bits are unused:

```
| 32 bits  | 32 bits | 32 bits |... | 32 bits |
| unused   | value6  | value5  |... | value0  |
| 224-255  | 192-223 | 160-191 |... |  0-31   |
```

In case of Polymarket, the sports oracle returns just two values in one request with each value representing the points scored by one of the competing teams in a game; all other values are expected to be 0.

In two cases the UMA Oracle returns special flags: If no answer is possible, for example if the game has been Canceled, the oracle returns `type(int256).max` and if the result proposal is too early the oracle returns `type(int256).min`.

2.2.3 UMA

Oracle requests to UMA go through the following lifecycle: First, it is requested from the oracle. Second, anyone can propose a result by providing a sufficient bond. Third, two things can happen:

- the proposal gets accepted after the liveness period has elapsed, in case no dispute happened and anyone can call `settle` to pay out the reward and bond to the proposer.
- anyone can dispute the proposal by providing a bond and delegate to UMA voters the settlement of this dispute.

The UmaSportsOracle makes "event based" requests to the UMA Oracle. This means that the request is not evaluated at a fixed time which is set in advance, but at the time the first proposal is made. This prohibits users from proposing the special answer "too early" (however, "too early" as answer is still reachable as result of UMA's dispute process). Further, event based requests refund the reward set by the requester immediately after a dispute is raised. That allows the UmaSportsOracle to use funds returned from a dispute to create a new oracle request.

The UmaSportsOracle is deployed on Polygon Mainnet and UMA Voters vote on disputes on Ethereum Mainnet. Hence, the UMA Optimistic Oracle on Polygon bridges the disputes to Ethereum Mainnet via the Polygon Bridge and dispute resolutions are resolved back via the bridge. After a dispute is resolved, anyone can call the function settle in UMA's OptimisticOracleV2 to finalize the oracle request. This completes the dispute and allows the consumer of the request, the UmaSportsOracle, to decode the oracle request and store its result.

The UmaSportOracle enables callbacks from UMA's OptimisticOracleV2 on disputes and settlement via function priceDisputed and priceSettled. The hooks are called with the parameters of the disputed or settled game and allow the UmaSportsOracle to selectively resolve only the latest oracle request in case multiple requests are pending. Callbacks allow the UmaSportsOracle to handle disputes and settlements right when they take place and update the internal state of the game accordingly.

2.2.4 Admin game actions

The UmaSportsOracle inherits the Auth contract, allowing admin actions restricted by the onlyAdmin modifier. Admins can pause a game via pauseGame that pauses game settlement from UMA via the priceSettled hook. Any settlement call during a pause has no effect. The admin can reinstate a game with unpauseGame or settle it with emergencySettleGame by specifying an outcome. Further, admins can change the liveness - the time until expiry of a request - via setCustomLiveness, and the bond needed to propose or dispute a proposal by calling setBond on any oracle request.

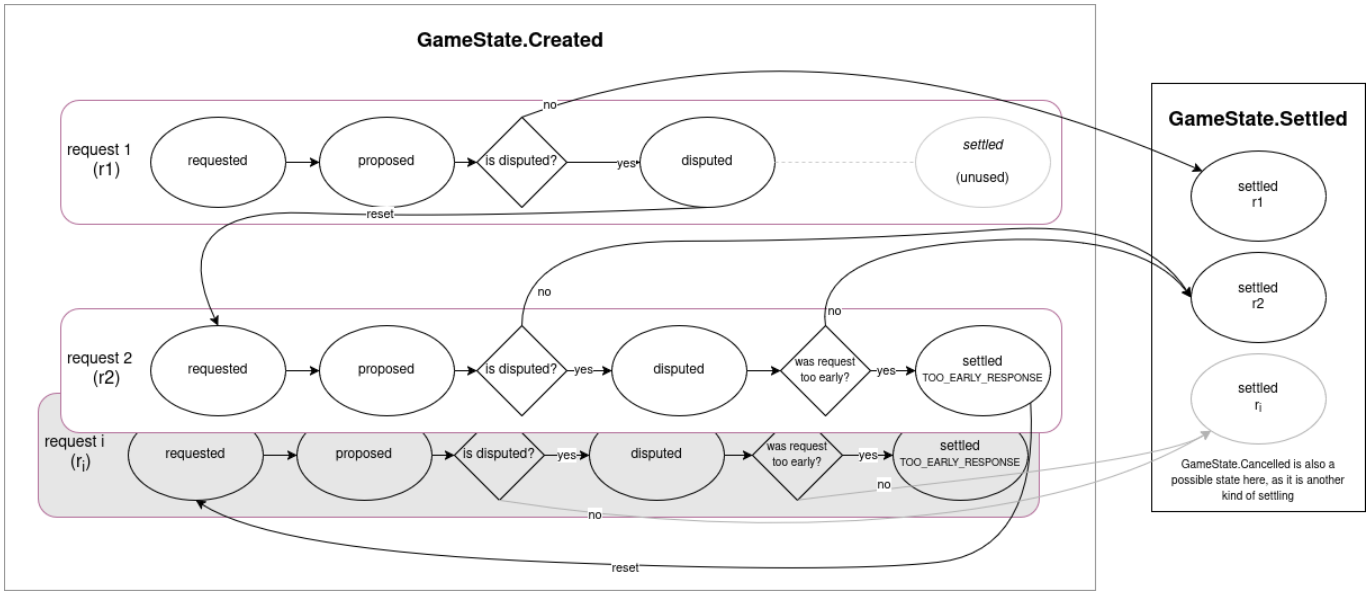
2.2.5 Game lifecycle

The lifecycle of a game starts with the creation of a game with function createGame with the initial state being *Created*. Over time the game is expected to transition into *Settled* in case of a successful settlement via UMA. Final state *EmergencySettled* or intermediate State *Paused* are only reachable via admin actions. The final state *Canceled* can be reached in case the oracle request returns status *Canceled*.

On dispute, the UMA Oracle calls the hook priceDisputed with the parameters of the disputed game. Note that the UMA Oracle refunds the requester since event based requests are used. That means the UmaSportsOracle can re-use the reward to create a new oracle request after a dispute. The implementation always creates a new oracle request after the first request is disputed with the new request replacing the old one. When the old request gets settled the result is ignored. In case the second oracle request is also disputed, the UmaSportsOracle awaits the settlement of the second request.

On settlement the UMA protocol calls the hook priceSettled with the parameters of the settled game. The UMASportsOracle decodes the oracle request according to UMIP-183 and resolves the game with the values returned. The two special flags "too early" or "Canceled" are handled by creating a new oracle request in case of "too early" and canceling the game in case of "Canceled". Canceling splits the payout evenly among all outcomes.

As noted, a game can be reset under certain conditions. Below is a graphic showcasing the lifecycle of a game:



2.2.6 Market

A market is a bet on a game result. The `UmaSportsOracle` allows the permissionless creation of markets of three different types:

- Winner: Bet on the winner of a sports event. Created with `createWinnerMarket`.
- Spreads: Bet on the point spread of a sports event. Created with `createSpreadsMarket`. Spreads must be specified in half points (i.e. 1.5, 2.5, etc.) to avoid ambiguity.
- Totals: Bet on the total points scored in a sports event. Created with `createTotalsMarket`. Totals must be specified in half points (i.e. 1.5, 2.5, etc.) to avoid ambiguity.

Creating a market prepares a new condition in UMA's *Conditional Token Framework*. A conditional token represents a specific question with multiple possible outcomes. Anyone can provide collateral to the conditional token and mint tokens for every outcome. Traders can take a long position on a single outcome by purchasing one of the outcomes from a minter. When the outcome is resolved, the oracle reports the payout for each outcome. Typically, only one outcome will be correct, making outcomes mutually exclusive. However, multiple (or even all) outcomes can be correct if, for example, a game is Canceled. Holders of one or more outcomes can then redeem their tokens for the collateral after an oracle has reported the payout of the outcomes. The `UmaSportsOracle` functions as the oracle for all markets prepared by itself and reports the payout for each outcome to the conditional token. In case of winner market the payout is 1 for the winner and 0 for the loser, whereas it is 1 for both in case of a tie or cancellation. The relative payout of each outcome can then be calculated by dividing the payout by the sum of all payouts in each market, i.e. in case of a cancellation the payout is 1 / 2 for each outcome.

The `UmaSportsOracle` resolves markets using the `resolveMarket` function, which requires the underlying game to be settled, emergency settled, or Canceled. The payout values are calculated using the `PayoutLib` based on the game results:

- Winner Market: The team with the higher score wins the game. The winning team has a payout of 1 and the looser has 0. In case of a tie, the payout is 1 for both teams (i.e. both teams receive half of the reward). The position of the home and away teams is determined by the `ordering` parameter of the game. For `HomeVsAway`, the home team is in the first position and the away team in the second. For `AwayVsHome`, the away team is in the first position and the home team in the second.
- Spread Market: The underdog wins if they win the game or lose by less than the line against the favorite. The favorite wins if the score difference is higher than the line. The payout is 1 for the winner and 0 for the loser. The favorite is always in the first place and the underdog in the second place.

- Totals Market: The outcome "Above the line" wins if the sum of the scores is greater than the line and otherwise outcome "Below the line" wins. The winning outcome has a payout of 1 and the losing outcome of 0. "Above the line" is always in first place and "Below the line" in the second place.

2.2.7 Admin market actions

Admins can pause an individual market via `pauseMarket` and `unpauseMarket`. This block calls to `resolveMarket` and allows admin to resolve a market via `emergencyResolveMarket` reporting the payouts directly to the conditional token.

2.2.8 Trust Model and Roles

- `admin`: Fully trusted. Can emergency settle game and market to any outcome and pause game and markets indefinitely. Can change the default liveness and bond of any oracle request.
- `addressWhitelist`: Fully trusted. Only non-rebasing, non-reentrant, non-blacklistable tokens are assumed to be used.
- `UMA`: Fully trusted to resolve the oracle requests truthfully.

Other roles and expectations:

- Conditional Token: Expected 0x4D97DCd97eC945f40cF65F87097ACe5EA0476045.
- Optimistic Oracle: Expected 0xee3afe347d5c74317041e2618c49534daf887c24.
- Game Creators: Untrusted. Can set initial parameters of the game, but have no control over the game afterward.
- Market Creators: Untrusted. Can create arbitrary markets, but markets are immutable after creation.

Traders are expected to verify that market parameters are set fairly or trust another party to verify them. If for example the resolution time ("liveness") is set to a very small number (i.e. 1 second), there might not be enough time to dispute an oracle request. Further, if the bond or reward is not large enough, then UMA Voters might not be incentivized to vote on the request. See Security Considerations for Users.

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 Open 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:

- 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	0

6 Resolved Findings

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

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	1

- UMA Oracle Allows Creating Undisputable Games Code Corrected

Low-Severity Findings	3

- Cancel Does Not Refund Game Creator Code Corrected
- Missing Arguments in Events Code Corrected
- ready() Can Return True When Request Can Not Be Settled Code Corrected

Informational Findings	3

- Admin Can Irrecoverably Break Market Code Corrected
- Ambiguous Naming of Internal Functions Code Corrected
- Inaccurate Comments Code Corrected

6.1 UMA Oracle Allows Creating Undisputable Games

Design Medium Version 1 Code Corrected

CS-POLY-SPORTS-009

`UmaSportsOracle` integrates with UMA's `OptimisticOracleV2`, and allows creating games with up to `8139` bytes of ancillary data. Because of a pre-existing issue in UMA's `OptimisticOracleV2`, oracle requests with more than `8066` bytes of ancillary data can be created but not disputed. The resulting game can receive an incorrect result proposal that cannot be disputed, and will therefore resolve incorrectly.

UMA's `OptimisticOracleV2` allows creating requests with length of ancillaryData between 0 and 8139. If the ancillary data is in the range `[8067, 8139]`, the request can be answered incorrectly with no possibility of dispute: Both `requestPrice()` and `proposePrice()` succeed for requests of length in the range `[8067, 8139]`, but `disputePrice()` reverts when the request is forwarded to the DVM (UMA's voting mechanism to settle disputes).

This can lead to wrong results being proposed, and then accepted without dispute since the dispute function reverts.

When `disputePrice()` is called, the OptimisticOracleV2 (https://polygonscan.com/address/0xee3afe347d5c74317041e2618c49534daf887c24) forwards the request to the DVM (Data Verification Mechanism), after appending metadata of size 53 (`',ooRequester:0000000000000000000000000000000000000000'`) to the `ancillaryData` (see on github). We are starting with `ancillaryData` in a range of length of `[8067, 8139]` which now becomes `[8120, 8192]` (+53) after appending metadata.

In `disputePrice()`, `_getOracle()` is configured to return the address of the OracleChildTunnel (https://polygonscan.com/address/0xBEd4c1FC0FD95A2020EC351379b22d8582B904e3#code). The `requestPrice()` function of this contract is called to bridge the disputed request to the voting mechanism of UMA token stakers on Ethereum mainnet. `OracleChildTunnel` appends further metadata to the request's `ancillaryData` (code on github). These metadata are of length 73 on Polygon (`',childRequester:ee3afe347d5c74317041e2618c49534daf887c24,childChainId:137'`). Appending these metadata brings the length of `ancillaryData` from `[8120, 8192]` to `[8193, 8265]` . `OracleChildTunnel` then calls `OracleBaseTunnel._requestPrice()` which asserts that the length is below 8192 (link to code). This will cause a revert of the whole `disputePrice()` call.

This means that the limit of 8139 bytes enforced by UMA's `OptimisticOracleV2` for the length of the `ancillaryData` is not strict enough, and it should instead be reduced to `8139 - 73 == 8066` to prevent the creation of requests that cannot be disputed.

Code corrected:

In Version 2, the issue is mitigated by setting the `MAX_ANCILLARY_DATA` constant to `8066` instead of `8139`.

Following communciation with UMA, the issue has also been resolved in the UMA protocol: UMA's `OracleChildTunnel` has been upgraded from 0xBEd4c1FC0FD95A2020EC351379b22d8582B904e3 to 0xac60353a54873c446101216829a6A98cDbbC3f3D. In the new version of `OracleChildTunnel`, the `ancillaryData` is hashed before being sent to Mainnet, reducing the ancillary data to constant size and therefore preventing the issue.

6.2 Cancel Does Not Refund Game Creator

Design Low Version 1 Code Corrected

CS-POLY-SPORTS-001

If the result for a game is the `type(uint256).max` value, meaning the game is cancelled, a potential refund held in the contract is not sent to the game creator. Considering cancelled games and settled games are very similar, finalizing the state to `Canceled` should pay the refund the same way as finalizing to `Settled` does.

Code corrected:

In Version 2, the function `_cancelGame` has been updated to refund the game creator in the event of a cancelled game.

6.3 Missing Arguments in Events

Design Low Version 1 Code Corrected

CS-POLY-SPORTS-002

1. The event `GameCreated` does not log the `ordering` of a game (i.e. home vs away or away vs home). This can make it difficult for external observers to verify that the `ordering` is correct.

2. The `MarketCreated` event emitted at the end of `_createMarket` does not include `underdog` as an argument, which is necessary to characterize Spread markets.

Code corrected:

In Version 2, the events have been updated to include the missing arguments.

6.4 `ready()` Can Return True When Request Can Not Be Settled

Correctness | Low | Version 1 | Code Corrected

CS-POLY-SPORTS-003

The view function `ready()` is supposed to return `true` when the game is ready to be settled. To do so, it queries the optimistic oracle `hasPrice()` function, which returns true in case the request is `Expired` (the first proposal is successful), `Resolved` (there was a dispute which has been resolved, but the request is not settled yet), and `Settled` (the request was settled). However, that means it can return true even when the game is not ready to be settled:

1. If the request is currently in the `Resolved` state, meaning the current dispute is resolved but `settle()` has not been called yet, it is possible that the resolution value is `TOO_EARLY_RESPONSE`. In that case, the game is not ready, since a new request will be created once the current one settles, but the `ready()` function returns true.
2. If the game is currently in the `Paused` state then a call to `settle` would skip the settlement.
3. If the game is already `Settled`, `Canceled`, or `EmergencySettled` then the game is not ready to be settled.

Code corrected:

In Version 2, function `ready` has been removed from the codebase.

6.5 Admin Can Irrecoverably Break Market

Informational | Version 1 | Code Corrected

CS-POLY-SPORTS-004

The `emergencyResolveMarket()` performs insufficient validation of the `payouts` argument, and the admin can put the market in an irrecoverable state by passing an array of length bigger than 2.

The *UmaSportsOracle* admin can call `emergencyResolveMarket()` on an existing market, which they have previously paused. This method allows to manually set the payouts of the market, by calling the `reportPayouts()` function of the `ctf` contract. If `emergencyResolveMarket()` is called with a `payouts` argument of length bigger than 2, for example 3, the call to `ctf.reportPayouts()` can succeed, as long as a condition exists in the `ctf` contract which has the `marketId` as `questionId`, the UmaSportsOracle address as `oracle`, and 3 payouts. The market state can therefore transition to `EmergencyResolved`, but the payouts for the actual `conditionId` is not be set, and it will not be possible to ever set them.

Another way the admin can irrecoverably break a market is by setting very high payout values, for example [type(uint256).max, 0]. The `ctf` contract allows setting such a payout to a condition. However, when the ctf tokens are redeemed, the payout value is multiplied by the token amount resulting in a revert because of overflow. In ConditionalTokens.sol, line 242, `payoutStake.mul(payoutNumerator)` can overflow

```
totalPayout = totalPayout.add(payoutStake.mul(payoutNumerator).div(den));
```

Code corrected:

In Version 2, the function `emergencyResolveMarket` has been updated to validate that the length of the payout array is exactly 2 and to ensure that payout values are either 0 or 1. Thus, the `ctf` contract can not revert on overflow when ctf tokens are redeemed.

6.6 Ambiguous Naming of Internal Functions

Informational | Version 1 | Code Corrected

CS-POLY-SPORTS-005

Internal functions `_isGameCreated()` and `_isMarketCreated()` are ambiguously named, since they check if a game or a market exists, and not if a game or market is in the `Created` state.

Code corrected:

In Version 2 the functions have been renamed to `_doesGameExist()` and `_doesMarketExist()` respectively.

6.7 Inaccurate Comments

Informational | Version 1 | Code Corrected

CS-POLY-SPORTS-007

1. The `createGame()` function has this comment in its natspec regarding the `reward` argument:

```
/// @param reward           - The reward paid to successful proposers and disputers
```

The comment is incorrect as the reward is only paid to successful proposers. Since the oracle requests are event based, the reward is refunded on dispute.

2. The natspecs of `priceSettled()` and `priceDisputed()` are over-simplified and do not fully represent the functions' behavior. For `priceSettled()`, it says:

```
/// Settles the Game by setting the home and away scores
```

which is incomplete, as `priceSettled()` could also ignore the settlement, or create a new request in case the result is `type(int256).min`. The natspec of internal function `_settle()` is also over-simplified, stating:

```
GameState transition: Created -> Settled
```

while the GameState could also stay the same (in case of a reset).

For `priceDisputed()` it says:

```
/// Resets the Game by sending out a new price Request to the OO
```

which is incomplete, as that only happens on the first dispute. On further disputes, `priceDisputed()` is a no-op beside setting the `refund` flag.

3. The natspec of `_createMarket()` states:

```
/// @param underdog     - The Underdog of the Market. Unused for Winner Markets.
```

which while correct, is restrictive. `Underdog` is unused for `Winner` markets, but it is also unused also for `Total` markets.

4. The natspec of `_resetGame()` states:

```
/// @dev We pay for this new request using the refunded reward that is transfered on dispute
```

That is not necessarily correct, as the request could be paid by the admin in case `_resetGame()` is called in `resetGame()`

5. Comment on `GameData` in `Structs.sol` mentions, for the `bond` field of the struct:

```
// The bond which OO proposers must put up
```

However, `bond` only represents part of the bond. `finalFee` specified in the Optimistic Oracle is added to the `bond` parameter to compute the `totalBond`.

Code corrected:

All the code comments listed above have been corrected in Version 2.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Blacklisted Users Block Refunds

Informational | Version 1 | Acknowledged

CS-POLY-SPORTS-006

In a refund, tokens are pushed to the user. If the token has a blacklist and one the recipient of the refund is blacklisted, the transfer reverts. This can block any call that attempts to refund (i.e. `resetGame()`, `priceDisputed()` and `priceSettled()`).

Note that Polymarket uses USDC.e which has blacklisting capabilities, but which are unused and have been renounced. The token however is upgradeable and the owner is a 5 out of 8 multisig controlled by a committee.

Acknowledged:

Client is aware of this behavior, but has decided to keep the code unchanged.

7.2 Refunds Are Lost Due to Admin Actions

Informational | Version 1 | Acknowledged

CS-POLY-SPORTS-008

1. The admin calling `emergencySettleGame()` changes a game's `state` to `EmergencySettled`. If a refund was available for that game, it is not refunded in `emergencySettleGame()`. Those funds are therefore lost.
2. The admin can call `resetGame()` before an existing oracle request is disputed. If the pre-existing oracle request is disputed, the refund is ignored and lost.
3. Similarly, the function `resetGame()` does not reset the requestor's address, so the original requestor can be refunded the admin's request if disputed.

Acknowledged:

Polymarket is aware of this behavior but has decided to keep the code unchanged.

8 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.

8.1 Admin Operational Risks and Considerations

Note Version 1

Admins have special privileges in Polymarket's protocol and should be aware of the following considerations when interacting with the protocol:

1. If a *game* is paused, any call to `priceSettled()` is ignored, and as UMA can call `priceSettled()` only once, the admin would have to call `emergencySettle` to settle the contract. Unpausing the contract after UMA has already called `priceSettled()` results in a *game* stuck in the `Created` state.

2. The function `requestPrice` in `OptimisticOracleV2` reverts when a oracle request with the same ID has been made previously. `OptimisticOracleV2` calculate the ID from the sender, identifier, timestamp and ancillaryData. In case of the `UmaSportsOracle` all request share the same sender and identifier and ancillaryData is unique for each game, so if a game makes two oracle requests in the same block, the second request will revert. That can happen when the `UMASportsOracle` creates a request and the request is proposed, and disputed in the same block, since disputing an oracle request can create a new oracle request. Similarly, a user can theoretically make the admin function `resetGame` revert by front-running the call with a reset request in the same block via an oracle dispute or settlement with an invalid price. However, the request has the same functionality as the request the admin would have created through `resetGame()`.

8.2 Security Considerations for Users

Note Version 1

The deployment of Games and Markets is unpermissioned, with their parameters set by the deployer and later configurable by the admin. Users of the protocol should verify the parameters or trust other parties to do so:

1. The order of the two teams in the ancillary data must match the order specified by the `ordering` parameter.
2. The token and reward used must be sufficient to incentivize proposers requesting prices.
3. The token and bond must be sufficient to incentivize user disputing incorrect price.
4. The liveness period must be long enough to allow participants to dispute a false price.