Conditional Token Market

Final Audit Report

January 9, 2025

Team Omega

Teamomega.eth.limo

Summary

Buttery Good Games has asked Team Omega to audit their smart contract system that implements a conditional token framework.

We found a number of new issues, which are described in the current report. Specifically, we found **one high severity issue** - these are issues that can lead to a loss of funds, and are essential to fix. We classified **one** issue as “high” - these are issues we believe you should definitely address, even if they do not lead to loss of funds. In addition, **3** issues were classified as “low”, and **10** issues as “info” - we believe the code would improve if these issues were addressed as well.

Of these issues, all issues, except one issue classified as informational, were resolved.

Severity	**Number of issues**	**Number of resolved issues**
High	1	1
Medium	0	0
Low	1	1
Info	5	4

Scope of the Audit

We audited the code from the following Github repository and commit:

```
    https://github.com/butterygg/cfm-v1/commit/5f6b0f5cedbe21814dbe
0ef1741af64b72145b38
```

We excluded the `vendor` folder and all files under it from consideration.

Resolution

After we sent Buttery Good Games a preliminary report, they addressed the issues they chose to fix. We audited the changes, and marked the results in this report.

The issues were addressed in the following commit:

```
    fe7d441f27d2a0e63b7e40ce3d95b152e4631b41
```

Methods Used

The code was audited by two experienced auditors, both of which independently inspected the source code to identify potential security flaws.

The contracts were compiled, deployed, and tested in a test environment.

Disclaimer

The audit makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

Severity definitions

High	Vulnerabilities that can lead to loss of assets or data manipulations.
Medium	Vulnerabilities that are essential to fix, but that do not lead to assets loss or data manipulations
Low	Issues that do not represent direct exploit, such as poor implementations, deviations from best practice, high gas costs, etc
Info	Matters of opinion

Findings

General

G1. Use a deterministic solidity compiler version [info] [resolved]

The `pragma` statements in the solidity files specify a range of versions (anything from 0.8.20 upwards is good)

```
    pragma solidity ^0.8.20;
```

Because the `foundry.toml` does not explicitly specify a Solidity version, this means that the output of the compilation process is nondeterministic: different developers may get different Bytecode as an output, depending on the current release. This can be a problem, as it makes verifying the contracts after deployment difficult, and can in theory lead to different behavior.

Version 0.8.28, which is the latest version as of writing, contains numerous improvements and optimizations, and three bug fixes https://docs.soliditylang.org/en/latest/bugs.html. Although the changes do not directly affect the current contracts, it is good practice to upgrade to a newer, if not the latest version.

Recommendation: Use a specific, recent, solidity version to compile the files. This can be done by either updating the `pragma` statements in the solidity files, but a better way is to choose a specific version by setting `solc_version` in the `foundry.toml`file

Severity: Info
Resolution: The issue was resolved - all solidity files now have a pragma header with a specific version (0.8.20)

G2. Use of require could be replaced with revert and custom errors [info] [resolved]

The contracts in the system are currently using require to enforce conditions, but the use of require could be replaced with the more gas efficient alternative of revert with custom errors.
Recommendation: Replace the use of require with revert and custom errors.
Severity: Info
Resolution: The issue was resolved as recommended.

G3. Must claim copyright for the licensing to be valid [low] [resolved]

The repository contains a copy of the GPL license, but it does not contain a statement that asserts that the copyright belongs to the developers. Such a statement is a precondition for applying the license - the copyright holder must claim ownership to be able to set the conditions of the license, as the license itself states:

```
Developers that use the GNU GPL protect your rights with two
steps: (1) assert copyright on the software, and (2) offer you
this License giving you legal permission to copy, distribute
and/or modify it.
```

Recommendation: Include a copyright claim such as the following, either as a header in all relevant files, in the README, or in any other place you deem fit.

```
(c) Copyright 2024 Buttery Good Games
```

Severity: Low

Resolution: The issue was resolved as recommended.

FlatCFM.sol

FC1. Invalid answer makes resolve always revert and funds permanently stuck [high] [resolved]

The `resolve` function checks the answer from the oracle, and reports the payouts to the conditional tokens according to the answer. However, when the answer returned from the oracle is invalid, the payouts array remains an all 0 array, which will cause the `reportPayouts` call to the conditional tokens revert (`ConditionalTokens.sol` line `101`). This will make it impossible to resolve the market, and all funds in it will be stuck permanently.
Recommendation: In case the answer is invalid, you could either evenly split the payouts by setting all payouts to 1, or always have an invalid option in the market, which will win the funds in case the answer is invalid.
Severity: High
Resolution: The issue was resolved, but see CSM3.

FC2. conditionId state variable is unused and can be removed [info] [resolved]

The `conditionId` state variable is unused in the contract, and can instead be queried by the frontend as needed, or emitted in an event, instead of saving it in the contract unnecessarily.
Recommendation: Remove the `conditionId` from the contract.
Severity: Info
Resolution: The issue was resolved as recommended

FlatCFMFactory.sol

FCF1. Factory should use EIP1167 proxy deployments [low] [resolved]

The `FlatFCMFactory` is currently deploying 1 `FlatCFM` and up to 50 `ConditionalScalarMarket` contracts on each call to `create`. This results in very high gas costs for each market deployment, which could be significantly reduced using the common pattern of proxy deployments, using for example the highly optimized `EIP1167` standard.
Recommendation: Instead of deploying the full contracts each time, use `EIP1167` proxy deployments.
Severity: Low

Resolution: The issue was resolved as recommended.

ConditionalScalarMarket.sol

CSM1. Duplicated logic between merge and redeem [info] [not resolved]

Lines 112 to 125 and 150 to 166 are essentially duplicated logic, and could be moved to a helper function to avoid unnecessary code duplication.
Recommendation: Move the logic of the lines mentioned to a common helper function to avoid repetition.
Severity: Info
Resolution: The developer acknowledged the issue and chose to keep the code as is.

CSM2. Wrong comment on invalid answers [info] [resolved]

The comment in line 51 specifies that if the answer is invalid, no payouts are returned, but the comment at the top of the function correctly states that "if the oracle value is invalid, report 50/50".
Recommendation: Remove the misleading comment in line 51.
Severity: Info
Resolution: The issue was resolved as recommended.

CSM3. Unfair payouts if the market resolved to invalid [high] [resolved]

When calling `ConditionalScalarMarket.split()`, the position is split into 3 partitions (`lo`, `hi`, `invalid`). The long and short tokens are wrapped as ERC20 tokens and sent to the caller, while the `invalid` token remains unwrapped and owned by the `ConditionalScalarMarket` contract.

When calling `redeem`, the caller's wrapped `lo` and `hi` tokens are sent back to the `ConditionalScalarMarket` contract, where they are then redeemed on the conditionalToken contract using the `redeemPositions` call. This will redeem *all* tokens held by the `ConditionalScalarMarket` contract.

This goes wrong is the market resolves to invalid - in that case, the first account that calls `redeem` will redeem all `invalid` tokens in existence - regardless of which tokens they hold themselves.

Recommendation: There are several ways of resolving this: one solution is to wrap `invalid` tokens as ERC20 tokens, on a par with the `lo` and `hi` outcomes.
Severity: High
Resolution: The issue was resolved as recommended.