Beamer Smart Contracts Review

By: ChainSafe Systems

October 2022

Beamer Smart Contracts Review

Auditors: Anderson Lee, Oleksii Matiiasevych

WARRANTY

This Code Review is provided on an “as is” basis, without warranty of any kind, express or implied. It is not intended to provide legal advice, and any information, assessments, summaries, or recommendations are provided only for convenience (each, and collectively a “recommendation”). Recommendations are not intended to be comprehensive or applicable in all situations. ChainSafe Systems does not guarantee that the Code Review will identify all instances of security vulnerabilities or other related issues.

Introduction

Brainbot Technologies requested ChainSafe Systems to perform a review of the Beamer smart contracts. The contracts can be identified by the following git commit hash:

`6d53b8f6c69e2c51bea23c0974c525123eba3019`

There are 12 contracts in scope including their parent contracts and interfaces.
After the initial review, Beamer team applied a number of updates which can be identified by the following git commit hash:

`53610b9b890e75e724d1996033e05ea5e0823984`

Additional verification was performed after that.

Disclaimer

The review makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about the fitness of the contracts for any specific purpose, or their bug free status.

Executive Summary

There are no known compiler bugs for the specified compiler version (0.8.13), that might affect the contracts' logic.

There were 0 critical, 1 major, 1 minor, 32 informational/optimizational issues identified in the initial version of the contracts. The minor and major issues found in the contracts were not present in the final version of the contracts. They are described below for historical purposes. We like the utility of the project, especially being implemented in such a concise way.

Critical Bugs and Vulnerabilities

No critical issues were identified.

Line by Line Review. Fixed Issues

1. ArbitrumL1Messenger, line 12: Note, the `Resolver` import could be removed.

2. ArbitrumL1Messenger, line 76: Optimization, in the `sendMessage()` function the `deposits[tx.origin]` value is read from storage twice.

3. FillManager, line 106: Optimization, in the `fillRequest()` function the `l1Resolver` variable is read from storage twice.

4. FillManager, line 113. Note, the `fillRequest()` function updates `fills[requestId]` in storage after doing external calls. It is a general recommendation to apply effects before external interactions.

5. FillManager, line 144. ***Major***, the `invalidateFill()` function could be called before the `fillRequest()` but in the same block, so that the `fillId` is known. This could lead to a malicious challenger proving that the valid claim request is invalid.

6. FillManager, line 146: Optimization, in the `invalidateFill()` function the `l1Resolver` variable is read from storage twice.

7. RequestManager, line 37. Optimization, the `Request.validUntil` field could fit into `uint32`.

8. RequestManager, line 38. Optimization, the `Request.lpFee` field could fit into `uint88`.

9. RequestManager, line 39. Optimization, the `Request.protocolFee` field could fit into `uint88`.

10. RequestManager, line 40. Optimization, the `Request.activeClaims` field could fit into a smaller type.

11. RequestManager, line 288. Optimization, the `createRequest()` function reads `currentNonce` from storage thrice. Consider introducing a local variable.

12. RequestManager, line 315. Optimization, the `createRequest()` function excessively reads `newRequest.validUntil` from storage instead of having it in a local variable or calculating in place.

13. RequestManager, line 376. Optimization, the `claimRequest()` function reads `currentNonce` from storage four times. Consider introducing a local variable.

14. RequestManager, line 381. Optimization, the `claimRequest()` function reads `claimStake` from storage twice. Consider introducing a local variable or use `msg.value`.

15. RequestManager, line 382. Optimization, the `claimRequest()` function excessively assigns `address(0)` to `claim.lastChallenger`. All values in storage are 0 by default.

16. RequestManager, line 383. Optimization, the `claimRequest()` function excessively assigns 0 to `claim.challengerStakeTotal`. All values in storage are 0 by default.

17. RequestManager, line 384. Optimization, the `claimRequest()` function excessively assigns 0 to `claim.withdrawnAmount`. All values in storage are 0 by default.

18. RequestManager, line 434. Optimization, the `challengeClaim()` function reads `claims[claimId]` properties from storage multiple times.

19. RequestManager, line 436. ***Minor***, the `challengeClaim()` function allows a challenge, after the `L1` claim resolution, which will result in a locked stake.

20. RequestManager, line 456. Optimization, in the `challengeClaim()` function the `msg.sender == nextActor` requirement could be put in the above else clause and replaced with `msg.sender == claim.claimer`. After this change you do not need the `nextActor` variable anymore and should use `msg.sender` instead.

21. RequestManager, line 460: Optimization, the `challengeClaim()` function reads `claim.claimerStake` from storage twice.

22. RequestManager, line 464. Optimization, the `challengeClaim()` function reads the claim. `challengerStakeTotal` from storage twice.

23. RequestManager, line 467. Optimization, the `challengeClaim()` could pointlessly update `claim.termination` with its own value.

24. RequestManager, line 472. Optimization, in the `challengeClaim()` function the requirement `claim.termination >= minimumTermination` is always true.

25. RequestManager, line 473. Optimization, the `challengeClaim()` reads `claim.claimTermination` from storage multiple times.

26. RequestManager, line 504: Optimization, the `withdraw()` function reads `claims[claimId]` properties from storage multiple times.

27. RequestManager, line 515. Optimization, in the `withdraw()` function the `claim.withdrawnAmount` variable is read from storage thrice.

28. Resolver, line 60. Optimization, in the `resolve()` function the `sourceChainInfo` fields read twice from storage. Read it into memory to save gas.

Line by Line Review. Acknowledged Findings

1. RequestManager, line 41. Optimization, the `Request.withdrawClaimId` field could fit into a smaller type.

2. RequestManager, line 187. Note, consider setting `transferLimit` on a per token basis.

3. RequestManager, line 462. Optimization, the `challengeClaim()` could pointlessly update `claim.lastChallenger` with its own value.

Anderson Lee

Oleksii Matiiasevych