Majority Protocol Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

Jorge

January 27, 2026

Contents

1 About Cyfrin **2**

2 Disclaimer **2**

3 Risk Classification **2**

4 Protocol Summary **2**

5 Audit Scope **2**

6 Executive Summary **3**

7 Findings **7**

7.1 Critical Risk . . . 7

7.1.1 `DepositManager::_refundEntryFee` doesn't deduct referral rewards allowing users to join then leave games to drain tokens via inflated referral rewards they aren't entitled to . . . 7

7.1.2 Attacker can drain all tokens from cancelled game since `SessionManager::refundCancelledGame` doesn't validate caller actually joined the game . . . 7

7.1.3 Users can participate in an infinite number of games they haven't joined, bypassing all entry fee requirements while still becoming winners and claiming prizes . . . 7

7.2 High Risk . . . 9

7.2.1 Impossible for user to get refund after re-joining a rescheduled game which is subsequently cancelled . . . 9

7.2.2 `DepositManager::getRewards` always includes `REFERRER_FEE` resulting in 2 percent of every games' rewards not being distributed to winners when there were no referrers . . . 9

7.2.3 Impossible to claim rewards when ranked rewards or number of winners are not set, resulting in permanently locked tokens once game has concluded . . . 9

7.2.4 Impossible to claim rewards when `XPTiers` are not set, resulting in permanently locked tokens once game has concluded . . . 10

7.2.5 If multiple users call `DefaultSession::assertResults` all but the first caller lose their bonds 10

7.3 Medium Risk . . . 12

7.3.1 User can join after the first question is revealed to gain an advantage over other users . . . 12

7.3.2 `MajorityChoicePrompt`, `SPBinaryPrompt` and `TriviaChoicePrompt` will not work correctly when used with different instances of `SessionManager` . . . 12

7.3.3 Incorrect `recordResult` recorded for each question in `recordResults` . . . 12

7.3.4 If zero xp is earned by all users, once game has concluded `SessionManager::claimRewards` panic reverts due to division by zero but game also can't be cancelled resulting in locked tokens 13

7.3.5 No validation on `reactionDeadline` allows multiple griefing scenarios . . . 13

7.3.6 Game creator can call `TriviaChoicePrompt::revealSolutions` before the `reactionDeadline` or end of game, griefing players from submitting answers while still retaining player entry fees . . . 14

7.3.7 `SPBinaryPrompt::getScore` and `getResult` conflict on what score users who didn't participate should receive, `getScore` also rewards users who got the wrong answer . . . 14

7.3.8 `SessionManager::rescheduleGame` advances the start time but not the end time allowing for a griefing attack where the game creator can collect fees while preventing users from participating . . . 14

7.3.9 User can set their answer's probability value to `uint16.max`, manipulating `result.probabilityAverage` in their favor . . . 15

7.4 Low Risk . . . 16

7.4.1 Prevent negative assertion following previous truthful assertion in `DefaultSession::assertionResolvedCallback` . . . 16

7.4.2 Excessive amount `maximumContestants` could make games to revert in `DefaultSession::recordResults` due to out of gas . . . 16

7.4.3 Referral rewards accumulate to `address(0)` when players aren't referred . . . 17

7.4.4 SessionManager::cancelGameIfCreatorMissing, endGame could revert due to out of gas if there are too many question in a game . . . 17
7.4.5 Same user can join the same game multiple times increasing their chance of winning by preventing other players from participating . . . 17
7.4.6 DepositManager::sponsorGame should revert if the game is Cancelled or Concluded . . . 18
7.4.7 SessionManager::revealGameQuestion doesn't validate that input _questionId belongs to input _gameId . . . 18
7.5 Informational . . . 19
7.5.1 Use named mappings to explicitly denote the purpose of keys and values . . . 19
7.5.2 Perform storage updates prior to external calls . . . 19
7.5.3 Fix comment in revealSolution . . . 19
7.5.4 Malicious user can front run the revealSolutions call committing the correct solution . . . 19
7.5.5 Remove obsolete return statements when using named return values . . . 19
7.5.6 Rename all sessionId to gameId or vice versa for consistency . . . 20
7.5.7 Game creator can grief winners by cancelling the game once it has ended, preventing winners from receiving their rewards . . . 20
7.5.8 Array length checks in FixedRanksReward::getRewards, getReward check against the wrong comparator . . . 20
7.5.9 DefaultSession::assertResults should revert if proposedWinners, totalXPs and totalTimes array lengths don't match . . . 20
7.5.10 Anyone should be able to conclude the game once winners have been determined . . . 21
7.5.11 Prompt::finalizedAnswer is never set . . . 21
7.5.12 Prompt::gameId is not validated to belong to the questionId and never used, could be removed . . . 21
7.5.13 DefaultSession::assertResults should verify input sessionId belongs to a game associated with its instance . . . 21
7.5.14 getReactionTime is returning the reactionDeadline even if the user didn't participate in the game . . . 22
7.6 Gas Optimization . . . 23
7.6.1 Use uint128 to pack DepositManager::protocolFee, maxCreatorFee into the same storage slot . . . 23
7.6.2 Cache identical storage reads and only write to storage once . . . 23
7.6.3 Prefer calldata to memory for external read-only function inputs . . . 23
7.6.4 In Solidity don't initialize to default values . . . 23
7.6.5 Perform input-related checks prior to reading storage . . . 24
7.6.6 More efficient implementation of SessionManager::joinGame via better storage packing . . . 24
7.6.7 Use uint32 for timestamps for better storage packing . . . 24
7.6.8 Don't copy entire Assertion struct from storage to memory in DefaultSession::assertionResolvedCallback . . . 24

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Majority Protocol is a decentralized blockchain platform that enables creators to host trivia and opinion-based games where players compete for prizes. The protocol operates on a session-based model where game creators define entry fees, questions, reward distributions, and game parameters. Players join games by paying entry fees in supported tokens (like USDC), which form a prize pool that gets distributed to winners after deducting protocol and creator fees.

The core gameplay follows a commit-reveal pattern to ensure fairness and prevent front-running. Once a game starts, questions are revealed sequentially, and players submit hashed commitments of their answers during a reaction deadline period. After the deadline, players reveal their actual answers, which must match their commitments. The protocol supports multiple question types through different prompt strategies: TriviaChoice for traditional trivia with predetermined correct answers, MajorityChoice where the most popular answer wins, and SPBinary which uses the "Surprisingly Popular" algorithm for binary choices with probability predictions.

Game results are determined through integration with UMA's Optimistic Oracle, which allows for decentralized verification of winners and XP calculations. The protocol implements sophisticated reward distribution mechanisms, supporting both fixed-rank rewards (predetermined percentages for 1st, 2nd, 3rd place, etc.) and proportional rewards based on XP earned. A referral system incentivizes user acquisition by allocating 2% of entry fees to referrers. The entire system is governed by a Registry contract that controls which session strategies, prompt strategies, reward strategies, and payment tokens can be used, with protocol admins maintaining quality control over these components. The protocol also includes comprehensive safety features such as game cancellation with refunds, rescheduling capabilities, contestant limits, and verification requirements for certain games.

5 Audit Scope

The following files were in scope for this audit:

```
src/offchain/uma/SessionResultAsserter.sol
src/prompt/MajorityChoicePrompt.sol
src/prompt/SPBinaryPrompt.sol
```

```
src/prompt/TriviaChoicePrompt.sol
src/reward/FixedRanksReward.sol
src/reward/ProportionalToXPReward.sol
src/session/DefaultSession.sol
src/DepositManager.sol
src/QuestionManager.sol
src/Registry.sol
src/Roles.sol
src/SessionManager.sol
src/SessionManagerBase.sol
```

6 Executive Summary

Over the course of 6 days, the Cyfrin team conducted an audit on the Majority Protocol smart contracts provided by Majority. In this period, a total of 46 issues were found.

The findings consist of 3 Critical, 5 High, 9 Medium and 7 Low with the remainder being information and gas optimizations.

Of the 3 Critical:

- 7.1.1 players could generate inflated referral rewards; once the game ends and referrers claim their inflated rewards, there will not be enough tokens to distribute to winners
- 7.1.2 permissionless attacker could drain all tokens from cancelled games, preventing players and sponsors from being refunded
- 7.1.3 players could completely bypass all game fee requirements, participating in an infinite number of games for free while still being able to win and claim prizes

Of the 5 High, the first 4 resulted in a state where tokens would be permanently locked inside the immutable `SessionManager` contract and the 5th resulted in users losing their bond:

- 7.2.1 impossible for players to get their game fee refunded after rejoining a rescheduled game which is subsequently cancelled, resulting in their tokens being permanently locked
- 7.2.2 referrer fee always subtracted during winner payout calculation even when there were no referrals, resulting in less tokens being distributed to winners with the difference permanently locked
- 7.2.3, 7.2.4 - impossible for winners to claim rewards when ranked rewards, number of winners or xp tiers were not set, resulting in token winnings being permanently locked
- 7.2.5 if multiple users asserted the results, all but the first caller lost their bond even if all their assertions were truthful

The 9 Medium and 7 Low covered a wide range of issues such as game theory advantages (7.3.1, 7.3.9), game creator griefing attacks (7.3.5, 7.3.6, 7.3.8) and other assorted findings that were either less likely or had smaller impact.

Test Suite Analysis

The protocol had a good test suite with >90% line coverage for most contracts.

Post Audit Recommendations

Even though the protocol had a good test suite with strong coverage, we discovered a significant number of Critical & High findings by thinking "outside the box" using an attacker mindset. Many of our best findings exploited a lack of state validation where the code made assumptions about inputs and contract state rather than explicitly validating those assumptions. As part of the audit we provided numerous defensive recommendations concerning state validation to help prevent similar vulnerabilities which were all implemented during mitigation fixes.

Due to the significant number of Critical & High severity findings it is statistically likely that more serious vulnerabilities still remain which could not be discovered during the 6-day audit window. Hence it is recommended that prior

to deploying significant capital on-chain in a production environment, another audit be conducted with a different pair of Cyfrin auditors during which no Critical or High severity findings should be found.

Summary

Project Name	Majority Protocol
Repository	engage-protocol
Commit	cca0cb3cedca...
Fix Commit	9b487e5a1560...
Audit Timeline	July 14th - July 21st, 2025
Methods	Manual Review

Issues Found

Critical Risk	3
High Risk	5
Medium Risk	9
Low Risk	7
Informational	14
Gas Optimizations	8
Total Issues	46

Summary of Findings

[C-1] `DepositManager::_refundEntryFee` doesn't deduct referral rewards allowing users to join then leave games to drain tokens via inflated referral rewards they aren't entitled to	Resolved
[C-2] Attacker can drain all tokens from cancelled game since `SessionManager::refundCancelledGame` doesn't validate caller actually joined the game	Resolved
[C-3] Users can participate in an infinite number of games they haven't joined, bypassing all entry fee requirements while still becoming winners and claiming prizes	Resolved
[H-1] Impossible for user to get refund after re-joining a rescheduled game which is subsequently cancelled	Resolved
[H-2] `DepositManager::getRewards` always includes `REFERRER_FEE` resulting in 2 percent of every games' rewards not being distributed to winners when there were no referrers	Resolved
[H-3] Impossible to claim rewards when ranked rewards or number of winners are not set, resulting in permanently locked tokens once game has concluded	Resolved

[H-4] Impossible to claim rewards when `XPTiers` are not set, resulting in permanently locked tokens once game has concluded	Resolved
[H-5] If multiple users call `DefaultSession::assertResults` all but the first caller lose their bonds	Resolved
[M-1] User can join after the first question is revealed to gain an advantage over other users	Resolved
[M-2] `MajorityChoicePrompt`, `SPBinaryPrompt` and `TriviaChoicePrompt` will not work correctly when used with different instances of `SessionManager`	Resolved
[M-3] Incorrect `recordResult` recorded for each question in `recordResults`	Resolved
[M-4] If zero xp is earned by all users, once game has concluded `SessionManager::claimRewards` panic reverts due to division by zero but game also can't be cancelled resulting in locked tokens	Resolved
[M-5] No validation on `reactionDeadline` allows multiple griefing scenarios	Resolved
[M-6] Game creator can call `TriviaChoicePrompt::revealSolutions` before the `reactionDeadline` or end of game, griefing players from submitting answers while still retaining player entry fees	Resolved
[M-7] `SPBinaryPrompt::getScore` and `getResult` conflict on what score users who didn't participate should receive, `getScore` also rewards users who got the wrong answer	Resolved
[M-8] `SessionManager::rescheduleGame` advances the start time but not the end time allowing for a griefing attack where the game creator can collect fees while preventing users from participating	Resolved
[M-9] User can set their answer's probability value to `uint16.max`, manipulating `result.probabilityAverage` in their favor	Resolved
[L-1] Prevent negative assertion following previous truthful assertion in `DefaultSession::assertionResolvedCallback`	Resolved
[L-2] Excessive amount `maximumContestants` could make games to revert in `DefaultSession::recordResults` due to out of gas	Resolved
[L-3] Referral rewards accumulate to `address(0)` when players aren't referred	Resolved
[L-4] `SessionManager::cancelGameIfCreatorMissing`, `endGame` could revert due to out of gas if there are too many question in a game	Resolved
[L-5] Same user can join the same game multiple times increasing their chance of winning by preventing other players from participating	Resolved
[L-6] `DepositManager::sponsorGame` should revert if the game is `Cancelled` or `Concluded`	Resolved
[L-7] `SessionManager::revealGameQuestion` doesn't validate that input `_questionId` belongs to input `_gameId`	Resolved
[I-01] Use named mappings to explicitly denote the purpose of keys and values	Resolved
[I-02] Perform storage updates prior to external calls	Resolved
[I-03] Fix comment in `revealSolution`	Resolved
[I-04] Malicious user can front run the `revealSolutions` call committing the correct solution	Acknowledged

[I-05] Remove obsolete `return` statements when using named return values	Resolved
[I-06] Rename all `sessionId` to `gameId` or vice versa for consistency	Resolved
[I-07] Game creator can grief winners by cancelling the game once it has ended, preventing winners from receiving their rewards	Acknowledged
[I-08] Array length checks in `FixedRanksReward::getRewards`, `getReward` check against the wrong comparator	Resolved
[I-09] `DefaultSession::assertResults` should revert if `proposedWinners`, `totalXPs` and `totalTimes` array lengths don't match	Resolved
[I-10] Anyone should be able to conclude the game once winners have been determined	Resolved
[I-11] `Prompt::finalizedAnswer` is never set	Resolved
[I-12] `Prompt::gameId` is not validated to belong to the `questionId` and never used, could be removed	Resolved
[I-13] `DefaultSession::assertResults` should verify input `sessionId` belongs to a game associated with its instance	Resolved
[I-14] `getReactionTime` is returning the `reactionDeadline` even if the user didn't participate in the game	Resolved
[G-1] Use `uint128` to pack `DepositManager::protocolFee`, `maxCreatorFee` into the same storage slot	Resolved
[G-2] Cache identical storage reads and only write to storage once	Resolved
[G-3] Prefer `calldata` to `memory` for external read-only function inputs	Resolved
[G-4] In Solidity don't initialize to default values	Resolved
[G-5] Perform input-related checks prior to reading storage	Resolved
[G-6] More efficient implementation of `SessionManager::joinGame` via better storage packing	Resolved
[G-7] Use `uint32` for timestamps for better storage packing	Resolved
[G-8] Don't copy entire `Assertion` struct from `storage` to `memory` in `DefaultSession::assertionResolvedCallback`	Resolved

7 Findings

7.1 Critical Risk

7.1.1 `DepositManager::_refundEntryFee` doesn't deduct referral rewards allowing users to join then leave games to drain tokens via inflated referral rewards they aren't entitled to

Description: `DepositManager::_payEntryFee` increments the referral rewards for the user's referrer:

```
referralRewards[gameId][Registry(registry).referrers(player)] += pool.ticketPrice * REFERRER_FEE;
```

But `DepositManager::_refundEntryFee` doesn't deduct referral rewards when a user leaves the game and their fee is refunded.

Impact: Malicious users can intentionally join then leave rescheduled games to drain tokens from the contract via inflated referral rewards they aren't entitled to.

This bug can also occur naturally without malicious users simply by users joining then leaving, giving referrers more reward allocation than they are entitled to. Once the game ends and referrers claim their inflated rewards, there will not be enough tokens to distribute to winners or for creator / protocol fees.

Recommended Mitigation: `DepositManager::_refundEntryFee` should deduct from the referral rewards when refunding the game fee, opposite to how `_payEntryFee` adds to the referral rewards when receiving the game fee.

Majority Games: Fixed in commit 50a1e6b.

Cyfrin: Verified.

7.1.2 Attacker can drain all tokens from cancelled game since `SessionManager::refundCancelledGame` doesn't validate caller actually joined the game

Description: Attacker can drain all tokens from cancelled game since `SessionManager::refundCancelledGame` doesn't validate caller actually joined the game.

Impact: Any cancelled game can be completely drained of tokens by a permissionless attacker.

Proof of Concept: Add test to `SessionManagerLeaveGame.t.sol`:

```
function test_attackerDrainsCancelledGame() public {
    // create game
    _createGame();

    // contestant joins
    vm.startPrank(contestants[0]);
    TestUSDC(usdc).approve(address(sessionManager), 10 ether);
    sessionManager.joinGame(1);

    // game is cancelled
    vm.stopPrank();
    sessionManager.cancelGame(1);
    vm.warp(block.timestamp + (type(uint256).max - block.timestamp));

    // attacker who never joined gets a refund they don't deserve
    address attacker = address(0x1337);
    vm.startPrank(attacker);
    sessionManager.refundCancelledGame(1);
    vm.stopPrank();

    // attacker can repeat this using different addresses
    // which all get a refund even though they never joined
    // the game, until the tokens have been totally drained

    // attacker has drained all the tokens
    assertEq(TestUSDC(usdc).balanceOf(attacker), 10 ether);
```

```
    assertEq(TestUSDC(usdc).balanceOf(address(sessionManager)), 0 ether);
    (,,,, uint256 totalCollectedAmount, address token, bool feesPaid) = sessionManager.gamePools(1);
    assertEq(totalCollectedAmount, 0 ether);
    assertEq(token, usdc);
    assertEq(feesPaid, false);

    // user who actually joined game can't get refund as
    // tokens have been drained
    vm.expectRevert(); // NotEnoughFunds(0x5615dEB798BB3E4dFa0139dFa1b3D433Cc23b72f, 0)]
    vm.startPrank(contestants[0]);
    sessionManager.refundCancelledGame(1);
    vm.stopPrank();
}
```

Recommended Mitigation: Only allow users who joined a game to claim refunds.

Majority Games: Fixed in commit 7692203.

Cyfrin: Verified.

7.1.3 Users can participate in an infinite number of games they haven't joined, bypassing all entry fee requirements while still becoming winners and claiming prizes

Description: `SessionManager::commitReaction` only checks that the user has joined `_gameId`, but doesn't verify that `_questionId` belongs to `_gameId`:

```
function commitReaction(uint256 _gameId, uint256 _questionId, bytes32 _commit)
        external
        onlyState(_gameId, SessionState.Ongoing)
    {
        require(contestants[_gameId][msg.sender], NotJoined(msg.sender, _gameId));
        IPromptStrategy(questionCommitment[_questionId].promptStrategy).commitReaction(
            _gameId, _questionId, _commit, msg.sender
        );
    }
```

Then in the strategy contracts `commitReaction` also doesn't validate this; it just saves the user's reaction into `reactions[_questionId][_user]` which has no associated to any `gameId`:

```
function commitReaction(uint256 _gameId, uint256 _questionId, bytes32 _commit, address _user) external
↪  {
        require(revealedAt[_questionId] != 0, QuestionNotRevealed(_questionId));
        require(
            revealedQuestions[_questionId].sessionManager == msg.sender,
            OnlySessionManager(revealedQuestions[_questionId].sessionManager, msg.sender)
        );
        require(
            revealedAt[_questionId] + revealedQuestions[_questionId].reactionDeadline >
            ↪  block.timestamp, /
            ReactionDeadlinePassed(_user, _questionId)
        );
        Reaction storage r = reactions[_questionId][_user];
        require(r.baseReaction.timestamp == 0, AnswerAlreadyCommitted(_user, _gameId, _questionId));

        r.baseReaction.commit = _commit; <------
        r.baseReaction.timestamp = block.timestamp;

        emit AnswerCommitted(_gameId, _questionId, _user, _commit);
    }
```

Similarly `SessionManager::revealReactions` also doesn't check that `_questionId` is associated with `_gameId`:

```
function revealReaction(
      uint256 _gameId,
      uint256 _questionId,
      bytes calldata _selection,
      uint256 salt,
      address _user
  ) external {
      uint16 selection = abi.decode(_selection, (uint16));
      require(
          revealedQuestions[_questionId].sessionManager == msg.sender,
          OnlySessionManager(revealedQuestions[_questionId].sessionManager, msg.sender)
      );
      Reaction storage r = reactions[_questionId][_user];
      require(r.baseReaction.timestamp != 0, AnswerNotCommitted(_user, _gameId, _questionId));
      require(!r.baseReaction.revealed, AnswerAlreadyRevealed(_user, _gameId, _questionId));
      require(
          keccak256(abi.encodePacked(_gameId, _questionId, selection, salt)) == r.baseReaction.commit,
          RevealMismatch(_gameId, _questionId, selection, salt, r.baseReaction.commit)
      );

      r.answer = selection;<------
      r.baseReaction.revealed = true;<-------
      address[][] storage choiceCounter = choiceCounters[_questionId];
      uint256 choicesLength = revealedQuestions[_questionId].choices.length;
      if (choiceCounter.length == 0) {
          // set length of choiceCounter to choicesLength
          assembly {
              sstore(choiceCounter.slot, add(sload(choiceCounter.slot), choicesLength))
          }
      }
      choiceCounters[_questionId][selection].push(_user); <-------
      emit AnswerRevealed(_gameId, _questionId, _user, selection);
  }
```

Impact: A malicious user can permanently bypass all game entry fees by joining one game; they can even create their own game with zero `ticketPrice`, join it and then participate in an infinite number of games for free.

Such users gain a huge competitive advantage of other users since they bypass game entry fees but can still become winners and claim winnings.

Proof of Concept: Run this POC in `SessionManagerReactions.t.sol`

```
function test_commitReaction_anotherGame() public {
      //@audit
      _createGame();
      _startGame();

      // create and start second game
      _createGame();
      uint256 startTime = sessionManager.getStartTime(2);
      vm.warp(startTime);

      // mint and join contestants

      for (uint256 i = 1; i < contestants.length; i++) {
          //start from 1; doing joint the contestant 0 to the second game
          // mint
          contestants[i] = makeAddr(string(abi.encodePacked("contestant-", Strings.toString(i))));
          TestUSDC(usdc).mint(contestants[i], 10 ether);

          // join
          vm.startPrank(contestants[i]);
```

```
            TestUSDC(usdc).approve(address(sessionManager), 10 ether);
            sessionManager.joinGame(2);
            vm.stopPrank();
        }

        //start the game
        sessionManager.startAndRevealGameQuestion(2, 1, abi.encode(question), salt); // game2 question 1

        bytes32 commit = keccak256(abi.encodePacked("test commit"));

        vm.prank(contestants[0]);
        sessionManager.commitReaction(1, 1, commit); //contestant 0 enter in the first game not the
        ↪  second // passing the first game but participating in the question 1 that belong to the
        ↪  second game
    }
```

Recommended Mitigation: When users perform *all* game-related actions, validate that:

- they have actually joined the game
- their input `questionId` belongs to their input `gameId`

Majority Games:

Fixed in commits 62cafca, 01d5cc2, f0e77f9.

Cyfrin: Verified.

7.2 High Risk

7.2.1 Impossible for user to get refund after re-joining a rescheduled game which is subsequently cancelled

Description: Impossible for user to get refund after re-joining a rescheduled game which is subsequently cancelled.

Impact: The user's fee for joining the game is permanently locked inside the immutable `SessionManager` contract.

Proof of Concept: Add the PoC to `SessionManagerLeaveGame.t.sol`:

```
function test_gameRescheduled_Leave_JoinAgain_GameCancelled_UserRefundReverts() public {
    // create a game
    uint256 timeAfterRescheduling = sessionManager.minimumRescheduleTime();
    _createGame();
    uint256 rescheduleDelta = type(uint256).max - sessionManager.getStartTime(1);

    // user joins the game
    uint256 gameId = 1;
    uint256 gameFee = 10 ether;
    address user = contestants[0];
    uint256 startTime = sessionManager.getStartTime(gameId);
    vm.startPrank(user);
    TestUSDC(usdc).approve(address(sessionManager), gameFee);
    sessionManager.joinGame(gameId);
    vm.stopPrank();

    // game gets rescheduled
    sessionManager.rescheduleGame(1, startTime + rescheduleDelta);
    vm.warp(block.timestamp + timeAfterRescheduling);

    // user leaves rescheduled game
    vm.prank(user);
    sessionManager.leaveRescheduledGame(gameId);

    // user got refunded the game fee
    assertEq(TestUSDC(usdc).balanceOf(user), gameFee);
    assertEq(TestUSDC(usdc).balanceOf(address(sessionManager)), 0 ether);

    // user decides to re-join the game
    vm.startPrank(user);
    TestUSDC(usdc).approve(address(sessionManager), gameFee);
    sessionManager.joinGame(gameId);
    vm.stopPrank();

    // user decides to leave again; impossible
    vm.expectRevert(); // AlreadyRefunded(0xd52E4d00E363cB91d9051fBFDC80c292a1da630B, 1)]
    vm.prank(user);
    sessionManager.leaveRescheduledGame(gameId);

    // game is cancelled
    sessionManager.cancelGame(gameId);

    // impossible for user to get a refund!
    vm.expectRevert(); // AlreadyRefunded(0xd52E4d00E363cB91d9051fBFDC80c292a1da630B, 1)]
    vm.prank(user);
    sessionManager.refundCancelledGame(gameId);

    // user's game fee is permanently stuck in the session manager contract!
    assertEq(TestUSDC(usdc).balanceOf(user), 0);
    assertEq(TestUSDC(usdc).balanceOf(address(sessionManager)), gameFee);
}
```

Recommended Mitigation: In `DepositManager::_payEntryFee` add this:

```
// reset user refunded status when joining the game; this allows
// users to get refunded if they rejoin a game which later gets cancelled
if(hasRefunded[gameId][player]) hasRefunded[gameId][player] = false;
```

Majestic Games: Fixed in commit 3ac5654 by not allowing users who have been refunded to rejoin the same game.

Cyfrin: Verified.

7.2.2 `DepositManager::getRewards` always includes `REFERRER_FEE` resulting in 2 percent of every games' rewards not being distributed to winners when there were no referrers

Description: `DepositManager::getRewards` always includes `REFERRER_FEE` when calculating the percentage of `totalCollectedAmount` available to distribute to winners:

```
function getRewards(uint256 gameId) public view returns (uint256) {
    return gamePools[gameId].totalCollectedAmount
        * (BASIS_POINTS - (gamePools[gameId].creatorFee + gamePools[gameId].protocolFee +
        ↪  REFERRER_FEE)) / BASIS_POINTS;
}
```

Impact: If no referral rewards were accrued for a game, this calculation results in the game rewards being less than they should since the `REFERRER_FEE` basis points are still used to deduct from the `totalCollectedAmount`.

The missing 2% of rewards are permanently stuck in the contract unable to be paid out to game winners or retrieved by the sponsor.

Recommended Mitigation: Rather than using `REFERRER_FEE`, change `DepositManager::_payEntryFee, _refundEntryFee` to increment/decrement the total amount of referral rewards in a new storage variable.

Then in `DepositManager::getRewards` deduct the total amount of referral rewards from `gamePools[gameId].totalCollectedAmount`.

Majority Games: Fixed in commit e090f2e by introducing a `CLAIMER_ROLE` which can collect referral fees assigned to `address(0)`, such that referral fees are always collected. `Registry::setReferrer` has been modified to prevent an address having `CLAIMER_ROLE` from becoming a referrer since then they couldn't collect fees associated with their address.

Cyfrin: Verified. We note that `AccessControl::grantRole` has not been overridden such that a referrer could be granted `CLAIMER_ROLE` which would prevent them from claiming referrals associated with their address.

7.2.3 Impossible to claim rewards when ranked rewards or number of winners are not set, resulting in permanently locked tokens once game has concluded

Description: `FixedRanksReward::setRankedRewards` enforces that ranked rewards can only be set when the game is in the `Created` state:

```
    function setRankedRewards(uint256 sessionId, uint256[] calldata _rankedRewards) external {
        require(sessionManager.getSessionState(sessionId) == SessionState.Created,
        ↪  NotCreated(sessionId));
```

The same is also true for `ProportionalToXPReward::setNumberOfWinners`.

But `SessionManager::startAndRevealGameQuestion` will happily start the game without ranked rewards / number of winners being set, and the game will progress all the way to the final `Concluded` state, giving the appearance that everything is OK.

Impact: Once the game has concluded, when the winners try to claim their rewards this will revert with `RankedRewardsNotSet` or `NumberOfWinnersMismatch`. There is no way to claim the rewards and because the game is in the `Concluded` state it can't be cancelled - the tokens are permanently locked in the contract.

Proof of Concept: Add the PoC to `SessionManagerEndGame.t.sol`:

```
function test_setRankedRewardsNotCalled_gameStarted_gameConcludes_cantClaimRewards() public {
    _createGame();

    _startGame();
    _revealQuestion();
    _warpToEndTime();
    sessionManager.endGame(1);
    _concludeGame();

    vm.expectRevert(); // RankedRewardsNotSet(1)
    vm.prank(contestants[0]);
    sessionManager.claimRewards(1, 0);
}
```

Recommended Mitigation: Don't allow the game to be started unless ranked rewards / number of winners have been set. Ideally:

- the `IRewardStrategy` interface would have an external function `rewardsConfigured` which returns `true` if its rewards mechanism has been configured and `false` otherwise
- `FixedRanksReward` and `ProportionalToXPReward` would both implement `rewardsConfigured` checking whether their internal reward implementations have been correctly configured
- `SessionManager::startAndRevealGameQuestion` would call `rewardsConfigured` on its reward strategy and revert if it returned `false`

Majority Games: Fixed in commits a2e353e, 96d5fbe.

Cyfrin: Verified.

7.2.4 Impossible to claim rewards when `XPTiers` are not set, resulting in permanently locked tokens once game has concluded

Description: `DefaultSession::setXPTiers` enforces that XP tiers can only be set when the game is in the Created state:

```
function setXPTiers(uint256 gameId, uint256[] memory _xpTiers) external {
        require(
            msg.sender == SessionManager(sessionManager).getCreator(gameId),
            NotGameCreator(SessionManager(sessionManager).getCreator(gameId), msg.sender)
        );
        require(_xpTiers.length >= 2, ArrayLengthMismatch());
        require(SessionManager(sessionManager).getSessionState(gameId) == SessionState.Created,
        ↪  GameNotCreated(gameId)); <--------
        xpTiers[gameId] = _xpTiers;
        emit XpTiersUpdated(gameId, _xpTiers);
    }
```

But `SessionManager::startAndRevealGameQuestion` will start the game without XP Tier being set, and the game will progress all the way to the final Concluded state. This will affect `SPBinaryPrompt.sol` and `TriviaChoicePrompt.sol` which are using the XP tiers to get the results:

```
function getResult(uint256 gameId, uint256 questionId, address user) public view returns (Result
↪  memory) {
        SessionManager sessionManager = SessionManager(revealedQuestions[questionId].sessionManager);
        uint256 score = getScore(questionId, user);
        address sessionStrategy = sessionManager.getSessionStrategy(gameId);
        uint256[] memory xpTiers = ISessionStrategy(sessionStrategy).getXPTiers(gameId); <-------
        if (score > 0) {
            return
                Result({xp: xpTiers[0] / 2 + score * xpTiers[0] / 2 / 10000, time:
                ↪  _getReactionTime(questionId, user)});
```

```
        } else {
            return Result({xp: xpTiers[1], time: _getReactionTime(questionId, user)});
        }
    }
```

These xp results are being used by `assertResults` but since the xp are not set the xp for user will be all zero.

```
function assertResults(
        uint256 sessionId,
        string calldata resultCid,
        address[] calldata proposedWinners,
        uint256[] calldata totalXPs, <-----
        uint256[] calldata totalTimes
```

When user is going to claim his rewards he end up receiving zero amount because the rewards are base in the xp; note this is valid just for `ProportionalToXpReward` strategy.

Impact: Once the game has concluded, when the winners try to claim their rewards they will end up receiving zero amount.

Proof of Concept: Run this proof of concept in `test/SessionManagerEndGameTest`

```
function test_getGameEndTime_notXptiers_set() public {
        _createGame_notSession();

        vm.prank(address(this));
        FixedRanksReward(address(rewardStrategy)).setRankedRewards(1, Solarray.uint256s(10000));

        _startGame();
        _revealQuestion();
        _warpToEndTime();
        sessionManager.endGame(1);
        _concludeGame();
    }

    function _createGame_notSession() internal {
        uint256 _startTime = block.timestamp + 1 days;
        uint256 _endTime = _startTime + sessionManager.maxGameDuration() - 1 seconds;

        uint256 gameId = sessionManager.createGame({
            _startTime: _startTime,
            _endTime: _endTime,
            _ticketPrice: 10 ether,
            _creatorFee: 1000,
            _creatorFeeReceiver: address(this),
            _token: usdc,
            _promptHashes: promptHashes,
            _promptStrategies: promptStrategies,
            _sessionStrategy: sessionStrategy,
            _rewardStrategy: rewardStrategy,
            _verificationRequired: false
        });
    }
```

Recommended Mitigation: Same as `H-3`; don't allow the game to be started unless the xp tiers have been set.

Majority Games: Fixed in commit 65727de.

Cyfrin: Verified.

7.2.5 If multiple users call `DefaultSession::assertResults` all but the first caller lose their bonds

Description: The `assertResults` is a permissionless function that allow anyone to assert a result for a `gameId` (sessionId):

```
function assertResults(
        uint256 sessionId,
        string calldata resultCid,
        address[] calldata proposedWinners,
        uint256[] calldata totalXPs,
        uint256[] calldata totalTimes
    ) external returns (bytes32 assertionId) {
        require(SessionManager(sessionManager).getSessionState(sessionId) == SessionState.Ended,
        ↪  GameNotEnded());
        return assertDataFor(
            sessionId, resultCid, resolutionGitRepoAtCommitHash, proposedWinners, totalXPs, totalTimes,
            ↪  msg.sender
        );
    }
```

Users that call this function have to pay a usdc bond of 250 dollars, see minimum bonds value.

```
 function assertDataFor(
        uint256 sessionId,
        string calldata resultCid,
        string memory resolutionGitRepoAtCommitHash,
        address[] calldata winners,
        uint256[] calldata totalXPs,
        uint256[] calldata totalTimes,
        address asserter
    ) internal returns (bytes32 assertionId) {
        asserter = asserter == address(0) ? msg.sender : asserter;
        uint256 bond = optimisticOracle.getMinimumBond(address(usdc)); /
        usdc.safeTransferFrom(msg.sender, address(this), bond);  <--------
        usdc.forceApprove(address(optimisticOracle), bond); <-------
}
```

The problem is that this function is not restricting users to call `assertResults` more than once for the same `sessionId` with that being say lets explore what would happen if `assertResults` is called twice for the same `sessionId`, `winners`, `totalXPs` and `TotalTimes` ; note that since the `assertResults` function is permissionless it can naturally be called twice by two different users at the same time:

1. A game has ended
2. User A call `assertResults` passing the session ID and the correct values.
3. User B didn't notice that that user A already call `assertResults` and call again `assertResults`
4. User A true is resolver positive and the `recordResults` function is called setting `winners[sessionId] = assertion.winners;`
5. User B assertion is resolved positive, the oracle call revert in `recordResults` making the user loss his funds

```
function recordResults(uint256 sessionId, bytes32 assertionId) public {

        require(SessionManager(sessionManager).getSessionState(sessionId) == SessionState.Ended,
        ↪  GameNotEnded());
        require(
            sessionId == assertions[assertionId].sessionId,
            SessionIdMismatch(sessionId, assertions[assertionId].sessionId)
        );
        require(assertions[assertionId].resolved, AssertionNotInitialized(assertionId));
        require(winners[sessionId].length == 0, WinnersAlreadyRecorded(sessionId));
        ...
```

```
        winners[sessionId] = assertion.winners;
    }
```

Note that the UMA protocol recommends don't revert in the callback:

```
recipient _must_ implement these callbacks and not revert or the assertion resolution will be blocked.
```

Impact: If `assertResults` is called more than once by different users just the first caller will recover their bond; the other users end up losing their money.

Proof of Concept: Run the next proof of concept in `file:test/session/DefaultSession.sol`

```
function test_RecordResults_double() public { //@audit
        // Mock SessionManager to return Ended state
        vm.mockCall(
            sessionManager, abi.encodeCall(SessionManager.getSessionState, gameId),
            ↪  abi.encode(SessionState.Ended)
        );

        string memory resultCid = "QmTestResultCID";
        address[] memory proposedWinners = new address[](2);
        proposedWinners[0] = player1;
        proposedWinners[1] = player2;

        uint256[] memory totalXPs = new uint256[](2);
        totalXPs[0] = 200;
        totalXPs[1] = 150;

        uint256[] memory totalTimes = new uint256[](2);
        totalTimes[0] = 30;
        totalTimes[1] = 45;

        vm.mockCall(
            optimisticOracle,
            abi.encodeWithSelector(OptimisticOracleV3Interface.assertTruth.selector),
            abi.encode(keccak256("assertionId43"))
        );
        bytes32 assertionId =
            defaultSession.assertResults(gameId, resultCid, proposedWinners, Solarray.uint256s(200,
            ↪  150), totalTimes);

        vm.mockCall(
            optimisticOracle,
            abi.encodeWithSelector(OptimisticOracleV3Interface.assertTruth.selector),
            abi.encode(keccak256("assertionId44"))
        );
        bytes32 assertionIdtwo =
            defaultSession.assertResults(gameId, resultCid, proposedWinners, Solarray.uint256s(200,
            ↪  150), totalTimes); //second assertion with the same values

        // 2. Call the callback to mark the assertion as resolved
        vm.prank(address(optimisticOracle));
        defaultSession.assertionResolvedCallback(assertionId, true);

        vm.startPrank(address(optimisticOracle));
        vm.expectRevert();
        defaultSession.assertionResolvedCallback(assertionIdtwo, true);
    }
```

Recommended Mitigation: Either prevent multiple in-process assertions for the same `sessionId`, or just return in the callback without reverting if it was already processed.

Majority Games: Fixed in commit 4c5483f by returning in the callback without reverting if the assertion has already been processed.

Cyfrin: Verified.

7.3 Medium Risk

7.3.1 User can join after the first question is revealed to gain an advantage over other users

Description: Users can join a game while the game is ongoing:

```
function joinGame(uint256 _gameId) external {
       require(
           games[_gameId].state == SessionState.Created || games[_gameId].state ==
           ↪  SessionState.Ongoing,
           InvalidGameState(SessionState.Created, games[_gameId].state)
       );
   }
```

`SessionManager::startAndRevealGameQuestion` both moves the game to the `Ongoing` state and reveals the first question.

Impact: A user can get an unfair advantage over others by always waiting for the first question to be revealed, and only joining a game if they know the answer to that question.

Recommended Mitigation: Consider don't allow user join to the game when the game is already ongoing:

```
  function joinGame(uint256 _gameId) external {
-      require(
-           games[_gameId].state == SessionState.Created || games[_gameId].state ==
↪  SessionState.Ongoing,
-           InvalidGameState(SessionState.Created, games[_gameId].state)
-       );
+      require(
+           games[_gameId].state == SessionState.Created,
+           InvalidGameState(SessionState.Created, games[_gameId].state)
+       );
```

Majority Games: Fixed in commit 6ec205f.

Cyfrin: Verified.

7.3.2 `MajorityChoicePrompt`, `SPBinaryPrompt` and `TriviaChoicePrompt` will not work correctly when used with different instances of `SessionManager`

Description: `MajorityChoicePrompt` is supposed to support multiple instances of `SessionManager`, however every instance of `SessionManager` starts with `QuestionManager::nextQuestionId = 0`.

This is problematic as `MajorityChoicePrompt::revealReaction` does this:

```
Reaction storage r = reactions[_questionId][_user];
require(!r.baseReaction.reactions[_questionId][_user], AnswerAlreadyRevealed(_user, _gameId,
↪  _questionId));
```

When a user plays `questionId = 0` on the first instance of `SessionManager` everything will work ok and `reactions[_questionId][_user].revealed` will be set to `true`.

If that same user plays `questionId = 0` on a second instance of `SessionManager` which uses the same instance of `MajorityChoicePrompt`, then `MajorityChoicePrompt::revealReaction` will revert with `AnswerAlreadyRevealed`.

Another potential issue is that `results[questionId][player]` will have valid results stored for a player from games on the first instance and this mapping doesn't differentiate between the different instances of `SessionManager`.

Recommended Mitigation: The simplest fix is that each `SessionManager` instances gets its own fresh `MajorityChoicePrompt` instance; the same issue likely affects `SPBinaryPrompt` and `TriviaChoicePrompt`.

Another option is that:

- there should only be 1 active instance of `SessionManager` at one time

- when a new instance of `SessionManager` is made active, it should be initialized with `gameId`, `sessionId` and `questionId` that are greater than the previous active instance
- add tests to the test suite which exercise this exact scenario to ensure everything will continue to work as expected

Majority Games: Fixed in commits 4b151db, 46d00d3, 35c63e7.

Cyfrin: Verified.

7.3.3 Incorrect `recordResult` recorded for each question in `recordResults`

Description: When an assertion is resolved the UMA oracle makes a call back to `DefaultSession::assertionResolvedCallback` if the assertion was truthful, it calls `recordResults`:

```
function recordResults(uint256 sessionId, bytes32 assertionId) public {
        ...
        uint256[] memory questionIds = SessionManager(sessionManager).getQuestionsForGame(sessionId);

        for (uint256 i = 0; i < assertion.winners.length; ++i) {
            address winner = assertion.winners[i]; //@audit how many winners could be?
            for (uint256 j = 0; j < questionIds.length; ++j) {
                (, address promptStrategy) =
                ↪  SessionManager(sessionManager).questionCommitment(questionIds[j]);
                IPromptStrategy(promptStrategy).recordResult(
                    questionIds[j], winner, assertion.totalXPs[i], assertion.totalTimes[i]
                ); <------
            }
          ...
        }

        winners[sessionId] = assertion.winners;
    }
```

As you can see the `recordResults` is calling the `recordResult` function for the specific strategies:

```
function recordResult(uint256 questionId, address player, uint256 xp, uint256 time) external {
        address sessionStrategy =
        ↪  SessionManager(revealedQuestions[questionId].sessionManager).getSessionStrategy(
            revealedQuestions[questionId].gameId
        );
        require(sessionStrategy == msg.sender, OnlySessionStrategy(sessionStrategy, msg.sender));

        results[questionId][player] = Result({xp: xp, time: time}); <------
    }
```

The problem is that the `recordResult` function is mean to savee the xp and time of the specific question of a gameId but what the `recordResults` function is passing the average of the user xp and time for all question corresponding to a specific gameId(sessionId).

Impact: Incorrect value passed for `recordResult` in all startgyes this will return incorrect values in `getResult` which is called in `_calculatePlayerSessionResult` and used as a view function.

Majority Games: Fixed in commit a3bcfb6.

Cyfrin: Verified.

7.3.4 If zero xp is earned by all users, once game has concluded `SessionManager::claimRewards` panic reverts due to division by zero but game also can't be cancelled resulting in locked tokens

Description: `ProportionalToXPReward::getReward` divides by `totalXP`, but if none of the users have earned XP, this will panic revert due to division by zero:

```
uint256 userXP;
uint256 totalXP;
for (uint256 i; i < winners.length; ++i) {
    (, uint256 xp,) = sessionStrategy.userResult(sessionId, winners[i]);
    if (i == position) {
        userXP = xp;
    }
    totalXP += xp;
}
reward = userXP * prizePool / totalXP;
```

`DefaultSession::setXPTiers` does not enforce non-zero xp tiers, it only enforces that at least two tiers must exist:

```
function setXPTiers(uint256 gameId, uint256[] calldata _xpTiers) external {
    require(
        msg.sender == SessionManager(sessionManager).getCreator(gameId),
        NotGameCreator(SessionManager(sessionManager).getCreator(gameId), msg.sender)
    );
    require(_xpTiers.length >= 2, ArrayLengthMismatch());
    require(xpTiers[gameId].length == 0, XpTiersAlreadySet(gameId));
    require(SessionManager(sessionManager).getSessionState(gameId) == SessionState.Created,
    ↪  GameNotCreated(gameId));
    xpTiers[gameId] = _xpTiers;
    emit XpTiersSet(gameId, _xpTiers);
}
```

Impact: If zero xp is earned by all users, once game has concluded `SessionManager::claimRewards` panic reverts due to division by zero but game also can't be cancelled because it is in the `Concluded` state, resulting in locked tokens.

Recommended Mitigation: Consider enforcing minimum value of 1 for every xp tier in `DefaultSession::setXPTiers`.

Majority Games: Fixed in commit 951a454 by enforcing non-zero values for every xp tier and also capping xp tiers to max 20.

Cyfrin: Verified.

7.3.5 No validation on `reactionDeadline` allows multiple griefing scenarios

Description: When a creator creates a game they send an array of bytes32 `promptHash` variables associated with the questions:

```
function createGame(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _ticketPrice,
        uint256 _creatorFee,
        address _token,
        address _creatorFeeReceiver,
        bytes32[] memory _promptHashes, <-----
        address[] memory _promptStrategies,
        address _sessionStrategy,
        address _rewardStrategy,
        bool _verificationRequired
    ) external returns (uint256 gameId) {...}
```

These `promptHash` are then reveled in the `_revealPrompt` function when creator call `startAndRevealGameQuestion` or `revealGameQuestion`, the `_revealPrompt` is checking the `keccak256(abi.encodePacked(_prompt, _salt)) == promptInitData.promptHash` and calls `revealQuestion` in the strategies:

```
function revealQuestion(bytes memory question, uint256 questionId) external {
        Prompt memory q = abi.decode(question, (Prompt));  <------
        require(registry.engageProtocols(msg.sender), InvalidSessionManager(msg.sender));
        require(q.sessionManager == msg.sender, OnlySessionManager(q.sessionManager, msg.sender));
        (, address promptStrategy) = SessionManager(q.sessionManager).questionCommitment(questionId);
        require(promptStrategy == address(this), InvalidPromptCall(questionId, promptStrategy));
        revealedQuestions[questionId] = q;
        revealedAt[questionId] = block.timestamp;
    }
```

The input parameter `bytes memory question` is decoded and converted in the `Prompt` struct:

```
struct Prompt {
        address sessionManager;
        uint256 gameId;
        string questionText;
        uint256 reactionDeadline;  <-----
        string finalizedAnswer;
        string[] media; <------
        string[] choices; <------
    }
```

Impact: * the possible range of values for `reactionDeadline` is never validated; if the game creator sets it to 0 or `type(uint256).max` then answering questions will always revert; users will be unable to earn xp but the game can still be concluded by the game creator in order to prevent users from claiming refunds from their fees

- `media` and `choices` should be validated to have the same length; creator owner can make mistakes setting up the questions and don't set properly this values making users harder to respond and probably lead to loss of funds(since users have not the choices and media set up they probably answer wrong).

Both cases can result in loss of funds for users.

Proof of Concept: Run this proof of concept in `SessionManagerEndGame.t.sol`

```
function test_zero_reactionDeadline() public {
        question.reactionDeadline = 0;
        bytes memory qEncoded = abi.encode(question);
        bytes32 questionHash = keccak256(abi.encodePacked(qEncoded, salt));
        promptHashes[0] = questionHash;
        promptStrategies[0] = promptStrategy;

        _createGame();
        _startGame();
        _warpToEndTime();

        sessionManager.endGame(1);
    }
```

Recommended Mitigation: Validate that:

- `string[] media` and `string[] choices` are equal in length
- `reactionDeadline` is within an admin-controlled minimum & maximum range
- `reactionDeadline` does not extend past a game's `endTime`

Majority Games: Fixed in commit 4cb2e42 by restricting `reactionDeadline`. Note that `media` and `choices` are not necessarily related, it's up to the creator & presentation frontend to make it work in a meaningful way.

Cyfrin: Verified.

7.3.6 Game creator can call `TriviaChoicePrompt::revealSolutions` before the `reactionDeadline` or end of game, griefing players from submitting answers while still retaining player entry fees

Description: `TriviaChoicePrompt::revealSolutions` calls `_revealSolutions` which doesn't validate that `reactionDeadline` has expired. This causes `TriviaChoicePrompt::commitReaction` to revert when users try to submit answers:

```
function commitReaction(uint256 _gameId, uint256 _questionId, bytes32 _commit, address _user) external
↪  {
       require(revealedAt[_questionId] != 0, QuestionNotRevealed(_questionId));
       require(
           revealedQuestions[_questionId].sessionManager == msg.sender,
           OnlySessionManager(revealedQuestions[_questionId].sessionManager, msg.sender)
       );
       require(solutionRevealedAt[_questionId] == 0, SolutionAlreadyRevealed(_questionId)); <------
       require(
           revealedAt[_questionId] + revealedQuestions[_questionId].reactionDeadline > block.timestamp,
           ReactionDeadlinePassed(_user, _questionId)
       );
       Reaction storage r = reactions[_questionId][_user];
       require(r.baseReaction.timestamp == 0, AnswerAlreadyCommitted(_user, _gameId, _questionId));

       r.baseReaction.commit = _commit;
       r.baseReaction.timestamp = block.timestamp;

       emit AnswerCommitted(_gameId, _questionId, _user, _commit);
   }
```

Impact: A malicious game creator can immediately reveal solutions, preventing users from submitting answers and earning xp. The game creator can still keep the users' entry fees, griefing users.

Proof of Concept: Run this test in `test/prompt/TriviaChoicePropmt.t.sol`

```
function test_solution_as_soon_as_reveledQuestion() public {
       triviaChoice.setRevealedQuestions();
       triviaChoice.revealSolutions(
           1, Solarray.uint256s(1), Solarray.bytess(abi.encode(uint16(1))), Solarray.uint256s(1234)
       );

       vm.expectRevert(abi.encodeWithSelector(TriviaChoicePrompt.SolutionAlreadyRevealed.selector, 1));
       triviaChoice.commitReaction(1, 1, keccak256(abi.encode(uint16(1))), address(this));
   }
```

Recommended Mitigation: Don't allow the game creator to call `revealSolution` until the game has end; you can use the games mapping in the session manager `require(sessionManager.games(_gameId).state = SessionState.Ended)`.

Majority Games: Fixed in commit 4d3f8b5.

Cyfrin: Verified. A different fix was chosen which is actually quite an elegant solution that removes the incentive for this the griefing attack because creators can't reveal solutions early without blocking their own ability to conclude the game, distribute rewards and claim their fees.

7.3.7 `SPBinaryPrompt::getScore` and `getResult` conflict on what score users who didn't participate should receive, `getScore` also rewards users who got the wrong answer

Description: `SPBinaryPrompt::getScore` returns 0 if a user didn't participate, but `getResult` calls `getScore`, and if the score is 0, returns `xpTIers[1]`.

`SPBinaryPrompt::getScore` also gives users a score based on the probability prediction even if the user chose the wrong answer, since it never checks `answerAIsWinner == reactions[questionId][player].answer`.

Impact: Users who didn't participate can actually get > 0 score if `xpTiers[1] > 0`. Users who didn't get the right answer still get rewarded based on their probability prediction.

Recommended Mitigation: Resolve the inconsistency between `SPBinaryPrompt::getScore` and `getResult`. Don't reward users who got the wrong answer.

Majority Games: Fixed in commits 50657e9, a55eb19.

Cyfrin: Verified.

7.3.8 `SessionManager::rescheduleGame` advances the start time but not the end time allowing for a griefing attack where the game creator can collect fees while preventing users from participating

Description: When a game creator reschedules a game, they set a new start time, but the end time is never updated.

```
function rescheduleGame(uint256 _gameId, uint256 _newStartTime)
       external
       onlyCreator(_gameId)
       onlyState(_gameId, SessionState.Created)
   {
       Game storage game = games[_gameId];
       require(
           _newStartTime > game.startTime + minimumRescheduleTime,
           RescheduleTooSoon(game.startTime, minimumRescheduleTime, _newStartTime)
       );
       require(game.originalStartTime == 0, GameIsAlreadyRescheduled(_gameId));
       game.originalStartTime = game.startTime;
       game.startTime = _newStartTime; <------ just moving start time?
       emit GameRescheduled(_gameId, _newStartTime);
   }
```

The maximum game duration is 10 minutes, and the `minimumRescheduleTime` is 15 minutes, so the `rescheduleGame` function will always set a new start time that is later than the end time.

Impact: The `rescheduleGame` function does not work as expected which can result in the official game "end time" being in the past when the game starts. However the actual gameplay is controlled by:

- When questions are revealed
- The reaction deadlines for each question
- When solutions are revealed

Hence the creator can:

- Reschedule without updating end time
- Rush through revealing all questions
- Use minimum reaction deadlines (5 seconds each)
- Complete the game in minutes instead of the intended duration
- Conclude the game & collect fees while players barely had time to participate and can't get their game fee refunded

This issue allows creators to effectively steal entry fees by conducting "speed-run" games that players can't reasonably participate in.

Proof of Concept: The existing test in the test suite shows that the values passed for the new start time are typically greater than one day:

```
// file: /test/SessionManagerRescheduleTest

 function test_rescheduleGame_VerifyOriginalStartTime() public {
```

```
        uint256 gameId = _createGame();

        uint256 originalStartTime = sessionManager.getStartTime(gameId);
        uint256 newStartTime = originalStartTime + sessionManager.minimumRescheduleTime() + 1 days;

        vm.prank(creator);
        sessionManager.rescheduleGame(gameId, newStartTime);

        // Verify original start time is preserved
        assertEq(sessionManager.getOriginalStartTime(gameId), originalStartTime);
    }
```

Recommended Mitigation: Consider updating the end time as well. The end time could be calculated by adding the original game duration (the difference between start time and the original start time) to the new start time.

Majority Games: Fixed in commit ddb690f.

Cyfrin: Verified.

7.3.9 User can set their answer's probability value to `uint16.max`, manipulating `result.probabilityAverage` in their favor

Description: In `SPBinaryPrompt.sol`, users commit an `answer` and a `probability`. When this value is revealed, the `revealReaction` function does not check if the probability exceeds 10,000 (which should be the maximum value based on how the `getScore` function uses probability).

```
function revealReaction(
        uint256 _gameId,
        uint256 _questionId,
        bytes calldata _selection,
        uint256 salt,
        address _user
    ) external {
        (bool answer, uint16 probability) = abi.decode(_selection, (bool, uint16));
        require(
            revealedQuestions[_questionId].sessionManager == msg.sender,
            OnlySessionManager(revealedQuestions[_questionId].sessionManager, msg.sender)
        );
        Reaction storage r = reactions[_questionId][_user];
        require(r.baseReaction.timestamp != 0, AnswerNotCommitted(_user, _gameId, _questionId));
        require(!r.baseReaction.revealed, AnswerAlreadyRevealed(_user, _gameId, _questionId));
        require(
            keccak256(abi.encodePacked(_gameId, _questionId, answer, probability, salt)) ==
            ↪  r.baseReaction.commit,
            RevealMismatch(_gameId, _questionId, answer, probability, salt, r.baseReaction.commit)
        );

        r.answer = answer;
        r.probability = probability;

        r.baseReaction.revealed = true;
        ResultAggregate storage result = resultAggregates[_questionId];
        result.respondents++;
        result.answerTotal += answer ? PRECISION : 0;
        result.probabilityTotal += probability; <-------
        result.answerAverage = result.answerTotal / result.respondents;
        result.probabilityAverage = result.probabilityTotal / result.respondents; <------
        emit AnswerRevealed(_gameId, _questionId, _user, answer, probability);
    }
```

As shown, the individual probability is used to calculate the `probabilityAverage`, which is a critical value in this strategy because it is used to compute the score that determines a user's results.

Impact: A malicious user can manipulate the `probabilityAverage` to improve their score, potentially securing a higher ranking among winners.

Proof of Concept: Run the next proof of concept in `test/prompt/SPBinaryPromptTest.t.sol`:

```
function test_revealReaction_Success_full_proability() public {
        _createQuestion();

        vm.warp(block.timestamp + 100);

        bytes32 commit = keccak256(abi.encodePacked(gameId, questionId, true, int16(5000),
        ↪  uint256(type(uint16).max)));

        vm.prank(mockSessionManager);
        prompt.commitReaction(gameId, questionId, commit, user0);

        vm.warp(block.timestamp + 100);

        vm.prank(mockSessionManager);
        prompt.revealReaction(gameId, questionId, abi.encode(true, int16(5000)), type(uint16).max,
        ↪  user0);
    }
```

Recommended Mitigation: Consider capping probability at 10,000 if a user commits a higher value:

```
function revealReaction(
        uint256 _gameId,
        uint256 _questionId,
        bytes calldata _selection,
        uint256 salt,
        address _user
    ) external {
        (bool answer, uint16 probability) = abi.decode(_selection, (bool, uint16));
        require(
            revealedQuestions[_questionId].sessionManager == msg.sender,
            OnlySessionManager(revealedQuestions[_questionId].sessionManager, msg.sender)
        );
        Reaction storage r = reactions[_questionId][_user];
        require(r.baseReaction.timestamp != 0, AnswerNotCommitted(_user, _gameId, _questionId));
        require(!r.baseReaction.revealed, AnswerAlreadyRevealed(_user, _gameId, _questionId));
        require(
            keccak256(abi.encodePacked(_gameId, _questionId, answer, probability, salt)) ==
            ↪  r.baseReaction.commit,
            RevealMismatch(_gameId, _questionId, answer, probability, salt, r.baseReaction.commit)
        );
+     if (probability > PRECISION ) { probability = PRECISION; }

        r.answer = answer;
        r.probability = probability;

        r.baseReaction.revealed = true;
        ResultAggregate storage result = resultAggregates[_questionId];
        result.respondents++;
        result.answerTotal += answer ? PRECISION : 0;
        result.probabilityTotal += probability;
        result.answerAverage = result.answerTotal / result.respondents;
        result.probabilityAverage = result.probabilityTotal / result.respondents;
        emit AnswerRevealed(_gameId, _questionId, _user, answer, probability);
    }
```

Majority Games: Fixed in commit 2eaae4d.

Cyfrin: Verified.

7.4 Low Risk

7.4.1 Prevent negative assertion following previous truthful assertion in `DefaultSession::assertionResolvedCallback`

Description: `DefaultSession::assertionResolvedCallback` does not handle the state where:

- it is first called with `assertedTruthfully = true` for given `assertionId`
- it is later called with `assertedTruthfully = false` for the same `assertionId`

Impact: In this state `delete assertions[assertionId];` is executed even though the results have already been recorded based on the first truthful assertion.

Recommended Mitigation: `DefaultSession::assertionResolvedCallback` should revert if `assertions[assertionId].resolved`:

```
    function assertionResolvedCallback(bytes32 assertionId, bool assertedTruthfully) public override {
        require(msg.sender == address(optimisticOracle), NotOptimisticOracle(msg.sender));
+       require(!assertions[assertionId].resolved, AssertionAlreadyResolved(assertionId));
```

Majority Games: Fixed in commit 99ec735.

Cyfrin: Verified.

7.4.2 Excessive amount `maximumContestants` could make games to revert in `DefaultSession::recordResults` due to out of gas

Description: `SessionManager::maximumContestants` is initially set to 1 million so potentially a large number of contestants can join each game:

```
/**
 * @notice Maximum number of contestants allowed in a game
 */
uint256 public maximumContestants = 1_000_000;
```

`DefaultSession::recordResults` iterates over all the winners to record the result for each question in the respective strategies:

```
function recordResults(uint256 sessionId, bytes32 assertionId) public {
    ...
       for (uint256 i = 0; i < assertion.winners.length; ++i) { <------
           address winner = assertion.winners[i]; //@audit how many winners could be?
           for (uint256 j = 0; j < questionIds.length; ++j) {
               (, address promptStrategy) =
               ↪  SessionManager(sessionManager).questionCommitment(questionIds[j]);
               IPromptStrategy(promptStrategy).recordResult(
                   questionIds[j], winner, assertion.totalXPs[i], assertion.totalTimes[i]
               );
           }
        ...
   }
```

If there are many winners this loop iteration could revert due to out-of-gas.

Impact: Games may not finish due to out of gas.

Proof of Concept: Taking in consideration that the block gas limit in base is around 30M and if we call `forge test --mt test_RecordResults_Success --gas-report` we could see that the `assertionResolvedCallback` is costing an avg 280587 for just two winners if we divide 30M / 280587 we get approximately 100 winner maximum.

Assumptions: Block gas limit: 30,000,000 gas Function overhead: ~50,000 gas

Average questions per game: 5-10 questions Conservative Estimate (10 questions per game): Gas per winner: 25,000 + (7,600 × 10) = 101,000 gas Available gas: 30,000,000 - 50,000 = 29,950,000 gas Maximum winners: 29,950,000 ÷ 101,000 296 winners

Optimistic Estimate (5 questions per game): Gas per winner: 25,000 + (7,600 × 5) = 63,000 gas Maximum winners: 29,950,000 ÷ 63,000 475 winners

Realistic Maximum: ~300-400 winners

Recommended Mitigation: Consider set a realistic `maximumContestants` to approx. 1000 participants. Alternatively another approach is just keep the winners in the array and create another function where user can `recordResult` by chunks.

Majority Games: Fixed in commit a2e353e.

Cyfrin: Verified.

7.4.3 Referral rewards accumulate to `address(0)` when players aren't referred

Description: `DepositManager::_payEntryFee` does this:

```
referralRewards[gameId][Registry(registry).referrers(player)] += pool.ticketPrice * REFERRER_FEE;
```

But `Registry(registry).referrers(player)` returns `address(0)` when `player` has not been referred.

Impact: Referral rewards accumulate to `address(0)`. These can't be claimed but it is still incorrect and should be fixed.

Recommended Mitigation: Only allocate referral rewards if the player has actually been referred; eg if `Registry(registry).referrers(player) != address(0)`.

Majority Games: Fixed in commit e090f2e by introducing a `CLAIMER_ROLE` which can collect referral fees assigned to `address(0)`, such that referral fees are always collected. `Registry::setReferrer` has been modified to prevent an address having `CLAIMER_ROLE` from becoming a referrer since then they couldn't collect fees associated with their address.

Cyfrin: Verified. We note that `AccessControl::grantRole` has not been overridden such that a referrer could be granted `CLAIMER_ROLE` which would prevent them from claiming referrals associated with their address.

7.4.4 `SessionManager::cancelGameIfCreatorMissing, endGame` could revert due to out of gas if there are too many question in a game

Description: Since there are no restrictions on the number of questions a game can support, a game could have so many questions that it causes `SessionManager::cancelGameIfCreatorMissing, endGame` to revert due to out-of-gas errors.

- `endGame` iterates over all questions:

```
function endGame(uint256 _gameId) external onlyState(_gameId, SessionState.Ongoing) {
       require(block.timestamp >= games[_gameId].endTime, GameIsNotEnded(games[_gameId].endTime,
       ↪  block.timestamp));
       uint256[] storage questions = gameQuestions[_gameId];
       for (uint256 i = 0; i < questions.length; i++) {
           require(_isRevealed(questions[i]), QuestionNotRevealed(_gameId, questions[i]));
       } <-----------
       games[_gameId].state = SessionState.Ended;
       emit GameEnded(_gameId);
   }
```

- so does `cancelGameIfCreatorMissing`:

```
 function cancelGameIfCreatorMissing(uint256 _gameId) external {
       require(
           games[_gameId].state != SessionState.Cancelled,
```

```
            InvalidGameState(SessionState.Cancelled, games[_gameId].state)
        );
        require(
            games[_gameId].state != SessionState.Concluded,
            InvalidGameState(SessionState.Concluded, games[_gameId].state)
        );
        require(block.timestamp >= games[_gameId].endTime, GameIsNotEnded(games[_gameId].endTime,
        ↪  block.timestamp));
        uint256[] storage questions = gameQuestions[_gameId];
        for (uint256 i = 0; i < questions.length; i++) { <-------

            if (!_isRevealed(questions[i])) {
                games[_gameId].state = SessionState.Cancelled;
                emit GameCancelled(_gameId);
                return;
            }
        }
        revert GameWaitingForConclusion(_gameId);
    }
```

Impact: `SessionManager::cancelGameIfCreatorMissing`, `endGame` could revert if there are too many questions in a game.

- If `endGame` cannot complete, users and the creator lose their funds and fees
- If `cancelGameIfCreatorMissing` reverts, users lose their funds if the creator is missing

Recommended Mitigation: Limit the number of questions in a game.

Majority Games: Fixed in commit cb88233.

Cyfrin: Verified.

7.4.5 Same user can join the same game multiple times increasing their chance of winning by preventing other players from participating

Description: `SessionManager::joinGame` doesn't validate whether the user joining has already joined. As long as the game is still in the `Created` state, the same user can join multiple times each time incrementing `numContestants`.

Impact: The same user can take all or most of the available player positions massively increasing their chances of winning since less players are able to compete against them. When `MajorityChoicePrompt` is used this could be especially powerful.

Games have an optional `verificationRequired` "whitelist" feature to prevent the same player using multiple addresses from taking over a game, but a player can abuse this bug to bypasses the `verificationRequired` option since the same whitelisted address can join the same game multiple times preventing other players from joining.

Recommended Mitigation: `SessionManager::joinGame` should revert if `contestants[_gameId][msg.sender] == true`. Consider wrapping this into a modifier `onlyNotJoinedGame` and putting that modifier onto `joinGame`.

Majority Games: Fixed in commit 2bba52d.

Cyfrin: Verified.

7.4.6 `DepositManager::sponsorGame` should revert if the game is `Cancelled` or `Concluded`

Description: `DepositManager::sponsorGame` doesn't verify the state of the game when accepting sponsorship amounts:

```
function sponsorGame(uint256 gameId, uint256 amount) external {
    GamePool storage pool = gamePools[gameId];
    pool.totalCollectedAmount += amount;
    sponsorAmounts[msg.sender][gameId] += amount;
```

```
    emit GameSponsored(gameId, msg.sender, pool.token, amount);
    SafeERC20.safeTransferFrom(IERC20(pool.token), msg.sender, address(this), amount);
}
```

Impact: Sponsors can sponsor `Cancelled` or `Concluded` games; in the case of `Concluded` games there is no way to retrieve their tokens. Sponsors can also sponsor non-existent games since `gameId` is not validated to belong to an actual game at all.

Recommended Mitigation: `DepositManager::sponsorGame` should revert if the game is `Cancelled` or `Concluded`.

Majority Games: Fixed in commit e01a1df - only allowing sponsorships for existing games in the `Created` or `Ongoing` state.

Cyfrin: Verified.

7.4.7 `SessionManager::revealGameQuestion` doesn't validate that input `_questionId` belongs to input `_gameId`

Description: `SessionManager::revealGameQuestion` doesn't validate that input `_questionId` belongs to input `_gameId`. It calls `QuestionManager::_revealPrompt` which ends up calling the `revealQuestion` function of the relevant prompt contract, but none of these verify that the input `_questionId` belongs to input `_gameId`.

Impact: A game creator can bypass the requirement that a game must be in the `Ongoing` state in order to reveal questions, by calling `SessionManager::revealGameQuestion` with `_gameId` of another game that is in the `Ongoing` state even if their game is not.

Recommended Mitigation: * Verify that the input `_questionId` belongs to input `_gameId` and consider applying the same fix such that it is also enforced for `startAndRevealGameQuestion`. One way to do this is by adding this check inside `QuestionManager::_revealPrompt`:

```
    function _revealPrompt(uint256 _gameId, uint256 _questionId, bytes memory _prompt, uint256 _salt)
    ↪  internal {
        PromptInitData storage promptInitData = questionCommitment[_questionId];
+       require(
+               _gameId == promptInitData.sessionId,
+               InvalidSessionIdForQuestion(_questionId, _gameId, promptInitData.sessionId)
+       );
```

- Consider also restricting functions such as `startAndRevealGameQuestion` and `revealGameQuestion` using the `onlyCreator` modifier - though technically this shouldn't be strictly necessary as only the game creator possesses the necessary salts.

Majority Games: Fixed in commit 15a2459.

Cyfrin: Verified.

7.5 Informational

7.5.1 Use named mappings to explicitly denote the purpose of keys and values

Description: Use named mappings to explicitly denote the purpose of keys and values; the protocol does use named mappings in some places but not others:

```
session/DefaultSession.sol
60:     mapping(uint256 gameId => uint256[]) public xpTiers;

QuestionManager.sol
39:     mapping(uint256 => uint256[]) public gameQuestions;
40:     mapping(uint256 => PromptInitData) public questionCommitment;

Registry.sol
29:     mapping(address => bool) public promptStrategies;
30:     mapping(address => bool) public sessionStrategies;
31:     mapping(address => bool) public rewardStrategies;
32:     mapping(address => bool) public paymentTokens;
33:     mapping(address => bool) public engageProtocols;

DepositManager.sol
72:     mapping(uint256 => mapping(address => bool)) public hasClaimed;
77:     mapping(uint256 => mapping(address => bool)) public hasRefunded;

SessionManager.sol
164:     mapping(uint256 => Game) public games;
174:     mapping(address => bool) public isVerificationApproved;
179:     mapping(address => uint256 timestamp) public liveness;

reward/FixedRanksReward.sol
29:     mapping(uint256 sessionId => uint256[]) public rankedRewards;

offchain/uma/SessionResultAsserter.sol
30:     mapping(bytes32 => Assertion) public assertions;
```

Majority Games: Fixed in commit 130e0a3 where we felt this added value.

Cyfrin: Verified.

7.5.2 Perform storage updates prior to external calls

Description: Most times it is safer to perform storage updates prior to external calls:

- DepositManager.sol

```
// switch these around in `_refundEntryFee`
186:         SafeERC20.safeTransfer(IERC20(pool.token), player, pool.ticketPrice);
187:         pool.totalCollectedAmount -= pool.ticketPrice;
```

- SessionManager.sol

```
// in `joinGame` perform the 2 storage updates prior to calling `_payEntryFee`
311:         _payEntryFee(_gameId, msg.sender);
312:         contestants[_gameId][msg.sender] = true;
313:         games[_gameId].numContestants++;
```

Majestic Games: Fixed in commit 6525ee1.

Cyfrin: Verified.

7.5.3 Fix comment in `revealSolution`

Description: The comment above `revealSolutions` says it is is meant to be called by the session manager but that's not true anymore, anyone can call it.

Majority Games: Fixed in commit acb42cb.

Cyfrin: Verified.

7.5.4 Malicious user can front run the `revealSolutions` call committing the correct solution

Description: A malicious user can front run the `revealSolutions` solution call taking the solution and committing the correct solution before the `revealSolutions` txn get through.

```
function commitReaction(uint256 _gameId, uint256 _questionId, bytes32 _commit, address _user) external
↪  {

       require(solutionRevealedAt[_questionId] == 0, SolutionAlreadyRevealed(_questionId));  <----
       require(
           revealedAt[_questionId] + revealedQuestions[_questionId].reactionDeadline > block.timestamp,
           ReactionDeadlinePassed(_user, _questionId)
       ); <----
      ...

       r.baseReaction.commit = _commit;
       r.baseReaction.timestamp = block.timestamp;

       emit AnswerCommitted(_gameId, _questionId, _user, _commit);
   }
```

A malicious user can just wait until the solution is reveled and commit the correct solution as long as this condition is met: `revealedAt[_questionId] + revealedQuestions[_questionId].reactionDeadline > block.timestamp`.

There is not check in `revealSolutions` that can prevent the reveal the solution too early.

Impact: Malicious can wait until the `revealSolutions` is called to commit the correct solution. Base has no mempool however.

Majority Games: Acknowledged.

7.5.5 Remove obsolete `return` statements when using named return values

Description: Remove obsolete `return` statements when using named return values in:

- `DefaultSession::_calculatePlayerSessionResult`

Majority Games: Fixed in commit cc1c9d1.

Cyfrin: Verified.

7.5.6 Rename all `sessionId` to `gameId` or vice versa for consistency

Description: `sessionId` appears to be used interchangeably with `gameId`; for consistency it would be best to rename all `sessionId` to `gameId` (or vice versa) where the same meaning is intended:

```
session/DefaultSession.sol
44:     error SessionIdMismatch(uint256 sessionId, uint256 assertionSessionId);
137:      * @param sessionId The session ID
144:         uint256 sessionId,
150:         require(SessionManager(sessionManager).getSessionState(sessionId) == SessionState.Ended,
↪  GameNotEnded());
152:             sessionId, resultCid, resolutionGitRepoAtCommitHash, proposedWinners, totalXPs,
↪  totalTimes, msg.sender
```

```
158:     * @param sessionId The session ID
161:    function recordResults(uint256 sessionId, bytes32 assertionId) public {
162:        require(SessionManager(sessionManager).getSessionState(sessionId) == SessionState.Ended,
↪  GameNotEnded());
164:            sessionId == assertions[assertionId].sessionId,
165:            SessionIdMismatch(sessionId, assertions[assertionId].sessionId)
168:        require(winners[sessionId].length == 0, WinnersAlreadyRecorded(sessionId));
170:        uint256[] memory questionIds =
↪  SessionManager(sessionManager).getQuestionsForGame(sessionId);
180:            userResult[assertion.sessionId][winner] =
184:        winners[sessionId] = assertion.winners;
195:                dataAssertion.sessionId,
201:            recordResults(assertions[assertionId].sessionId, assertionId);

session/ISessionStrategy.sol
24:    error WinnersAlreadyRecorded(uint256 sessionId);
57:     * @param sessionId The session ID
64:        uint256 sessionId,
73:     * @param sessionId The session ID
76:    function recordResults(uint256 sessionId, bytes32 assertionId) external;

reward/IRewardStrategy.sol
13:    error NotCreator(uint256 sessionId, address sender);
14:    error AlreadySet(uint256 sessionId);
15:    error NotCreated(uint256 sessionId);

offchain/uma/SessionResultAsserter.sol
20:        uint256 sessionId;
33:        uint256 indexed sessionId,
41:        uint256 indexed sessionId,
62:        uint256 sessionId,
79:                "sessionId asserted: ",
80:                sessionId,
107:            sessionId, resultCid, resolutionGitRepoAtCommitHash, asserter, false, winners,
↪  totalXPs, totalTimes
109:        emit DataAsserted(sessionId, resultCid, resolutionGitRepoAtCommitHash, asserter,
↪  assertionId);

reward/FixedRanksReward.sol
19:    event RankedRewardsUpdated(uint256 indexed sessionId, uint256[] rankedRewards);
21:    error RankedRewardsNotSet(uint256 sessionId);
22:    error InvalidRanks(uint256 sessionId, uint256 numRanks);
23:    error InvalidTotalPoints(uint256 sessionId, uint256 numPoints);
29:    mapping(uint256 sessionId => uint256[]) public rankedRewards;
47:     * @param sessionId The ID of the game
50:    function setRankedRewards(uint256 sessionId, uint256[] calldata _rankedRewards) external {
51:        require(sessionManager.getSessionState(sessionId) == SessionState.Created,
↪  NotCreated(sessionId));
52:        require(sessionManager.getCreator(sessionId) == msg.sender, NotCreator(sessionId,
↪  msg.sender));
53:        require(rankedRewards[sessionId].length == 0, AlreadySet(sessionId));
54:        require(_rankedRewards.length > 0, InvalidRanks(sessionId, _rankedRewards.length));
55:        require(_rankedRewards.length <= 20, InvalidRanks(sessionId, _rankedRewards.length));
62:        require(totalPoints == BASIS_POINTS, InvalidTotalPoints(sessionId, totalPoints));
64:        rankedRewards[sessionId] = _rankedRewards;
65:        emit RankedRewardsUpdated(sessionId, _rankedRewards);
69:    function getRewards(uint256 sessionId, address[] calldata winners, uint256 prizePool)
74:        require(rankedRewards[sessionId].length > 0, RankedRewardsNotSet(sessionId));
77:            rewards[i] = prizePool * rankedRewards[sessionId][i] / BASIS_POINTS;
82:    function getReward(uint256 sessionId, address[] calldata, uint256 position, uint256 prizePool)
87:        require(rankedRewards[sessionId].length > 0, RankedRewardsNotSet(sessionId));
88:        reward = prizePool * rankedRewards[sessionId][position] / BASIS_POINTS;
```

```
reward/ProportionalToXPReward.sol
19:     event NumberOfWinnersUpdated(uint256 indexed sessionId, uint256 numberOfWinners);
21:     error NumberOfWinnersMismatch(uint256 sessionId, uint256 numberOfWinners);
28:     mapping(uint256 sessionId => uint256 numberOfWinners) public numberOfWinners;
35:     function getRewards(uint256 sessionId, address[] calldata winners, uint256 prizePool)
40:         require(numberOfWinners[sessionId] == winners.length, NumberOfWinnersMismatch(sessionId,
↪  winners.length));
41:         ISessionStrategy sessionStrategy =
↪  ISessionStrategy(sessionManager.getSessionStrategy(sessionId));
45:             (, uint256 xp,) = sessionStrategy.userResult(sessionId, winners[i]);
56:     function getReward(uint256 sessionId, address[] calldata winners, uint256 position, uint256
↪  prizePool)
61:         require(numberOfWinners[sessionId] == winners.length, NumberOfWinnersMismatch(sessionId,
↪  winners.length));
62:         ISessionStrategy sessionStrategy =
↪  ISessionStrategy(sessionManager.getSessionStrategy(sessionId));
66:             (, uint256 xp,) = sessionStrategy.userResult(sessionId, winners[i]);
75:     function setNumberOfWinners(uint256 sessionId, uint256 _numberOfWinners) external {
76:         require(sessionManager.getSessionState(sessionId) == SessionState.Created,
↪  NotCreated(sessionId));
77:         require(sessionManager.getCreator(sessionId) == msg.sender, NotCreator(sessionId,
↪  msg.sender));
78:         require(numberOfWinners[sessionId] == 0, AlreadySet(sessionId));
79:         numberOfWinners[sessionId] = _numberOfWinners;
80:         emit NumberOfWinnersUpdated(sessionId, _numberOfWinners);
```

Majority Games: Fixed in commit 75663d1 - everything is now `sessionId`.

Cyfrin: Verified.

7.5.7 Game creator can grief winners by cancelling the game once it has ended, preventing winners from receiving their rewards

Description: `SessionManager::cancelGame` allows the game creator to cancel the game when it is in the `Ended` state:

```
function cancelGame(uint256 _gameId) external onlyCreator(_gameId) {
    require(
        games[_gameId].state != SessionState.Cancelled,
        InvalidGameState(SessionState.Cancelled, games[_gameId].state)
    );
    require(
        games[_gameId].state != SessionState.Concluded,
        InvalidGameState(SessionState.Concluded, games[_gameId].state)
    );
    games[_gameId].state = SessionState.Cancelled;
    emit GameCancelled(_gameId);
}
```

Impact: Once the game has ended but not yet concluded, the game creator can cancel if they don't like who the winners are. This griefs the winners preventing them from collecting their rewards.

Recommended Mitigation: Don't allow the game creator to cancel the game in the `Ended` state:

```
    function cancelGame(uint256 _gameId) external onlyCreator(_gameId) {
        require(
            games[_gameId].state != SessionState.Cancelled,
            InvalidGameState(SessionState.Cancelled, games[_gameId].state)
        );
+       require(
+           games[_gameId].state != SessionState.Ended,
+           InvalidGameState(SessionState.Ended, games[_gameId].state)
+       );
```

```
        require(
            games[_gameId].state != SessionState.Concluded,
            InvalidGameState(SessionState.Concluded, games[_gameId].state)
        );
        games[_gameId].state = SessionState.Cancelled;
        emit GameCancelled(_gameId);
    }
```

Majority Games: Acknowledged due to the Oracle's inability to settle according to the calculation rules (e.g. crash happens).

7.5.8 Array length checks in `FixedRanksReward::getRewards`, `getReward` check against the wrong comparator

Description: The check in `FixedRanksReward::getRewards` should compare using `>=` against input `winners.length`:

```
-         require(rankedRewards[sessionId].length > 0, RankedRewardsNotSet(sessionId));
+         require(rankedRewards[sessionId].length >= winners.length, RankedRewardsNotSet(sessionId));
```

Similarly the check in `FixedRanksReward::getReward` should compare using `>` against input `position`:

```
-        require(rankedRewards[sessionId].length > 0, RankedRewardsNotSet(sessionId));
+        require(rankedRewards[sessionId].length > position, RankedRewardsNotSet(sessionId));
```

The error should likely be changed to `PositionNotInRankedRewards` or something similar.

Majority Games: Fixed in commit 6717163.

Cyfrin: Verified.

7.5.9 `DefaultSession::assertResults` should revert if `proposedWinners`, `totalXPs` and `totalTimes` array lengths don't match

Description: `DefaultSession::assertResults` should revert if `proposedWinners`, `totalXPs` and `totalTimes` array lengths don't match.

Impact: If an asserter makes a mistake passing different length of `proposedWinners`, `totalXPs` and `totalTimes`, the asserter will loss their bond.

Recommended Mitigation: Check that `proposedWinners`, `totalXPs` and `totalTimes` have the same length.

Majestic Games: Fixed in commit aafd672.

Cyfrin: Verified.

7.5.10 Anyone should be able to conclude the game once winners have been determined

Description: Currently only the game creator can call `SessionManager::concludeGame`, even though at this point the winners have been determined.

Impact: If the game creator doesn't like who won, they can not conclude the game. The game could then be cancelled by users via `SessionManager::cancelGameIfCreatorMissing` to get their game fee refunded, but this allows a game creator to not pay out winners if they don't like who won.

Recommended Mitigation: Allow anyone to call `SessionManager::concludeGame`. Since during this time the game creator can also call `SessionManager::cancelGame`, perhaps allow a timeout period before anyone can call `SessionManager::concludeGame` using an offset from when the game entered the `End` state.

Majority Games: Fixed in commit dca8622.

Cyfrin: Verified.

7.5.11 `Prompt::finalizedAnswer` is never set

Description: When a game creator reveal a question the session manager checks the hash previously created and calls `revealQuestion` in the strategies. The strategy decodes the `Prompt` and sets it in `revealedQuestions[questionId]`. Each `Prompt` struct has its own `finalizedAnswer`:

```
struct Prompt {
        address sessionManager;
        uint256 gameId;
        string questionText;
        uint256 reactionDeadline;
        string finalizedAnswer;
        string[] media;
        string[] choices;
    }
```

After all votes are revealed the final answer has to be set in the `revealedQuestions[questionId].finalizedAnswer`. The problem is that no strategies are exposing a function to set this value.

Impact: `Prompt::finalizedAnswer` is never set after the answers are revealed; it appears to not be used at all.

Recommended Mitigation: Either set or remove `Prompt::finalizedAnswer`.

Majority Games: Fixed in commit 581a98d.

Cyfrin: Verified.

7.5.12 `Prompt::gameId` is not validated to belong to the `questionId` and never used, could be removed

Description: When a creator creates a game they send an array of bytes32 `promptHash` variables associate with the questions:

```
function createGame(
        uint256 _startTime,
        uint256 _endTime,
        uint256 _ticketPrice,
        uint256 _creatorFee,
        address _token,
        address _creatorFeeReceiver,
        bytes32[] memory _promptHashes, <-----
        address[] memory _promptStrategies,
        address _sessionStrategy,
        address _rewardStrategy,
        bool _verificationRequired
    ) external returns (uint256 gameId) {...}
```

These `promptHash` are then reveled in the `_revealPrompt` function when creator call `startAndRevealGameQuestion` or `revealGameQuestion`. `_revealPrompt` is checking `keccak256(abi.encodePacked(_prompt, _salt)) == promptInitData.promptHash` and calling `revealQuestion` in the strategies:

```
function revealQuestion(bytes memory question, uint256 questionId) external {
        Prompt memory q = abi.decode(question, (Prompt));  <------
        require(registry.engageProtocols(msg.sender), InvalidSessionManager(msg.sender));
        require(q.sessionManager == msg.sender, OnlySessionManager(q.sessionManager, msg.sender));
        (, address promptStrategy) = SessionManager(q.sessionManager).questionCommitment(questionId);
        require(promptStrategy == address(this), InvalidPromptCall(questionId, promptStrategy));
        revealedQuestions[questionId] = q;
        revealedAt[questionId] = block.timestamp;
    }
```

The bytes Prompt is decode and converted in the `Prompt` struct:

```
struct Prompt {
```

```
        address sessionManager;
        uint256 gameId; <------
        string questionText;
        uint256 reactionDeadline;
        bytes32 solutionCommitment;
        uint16 solution;
        string[] media;
        string[] choices;
    }
```

Hence `gameId` is never validated to belong to that `questionId`. Is is also never read anywhere apart from one `view` function that is never used so it could be safely removed.

Majority Games: Initially we removed it from the `Prompt` struct in commit 3644561. However it was later added back in commit 581a98d as it was required to resolve issue `Prompt::finalizedAnswer is never set`.

Cyfrin: Verified.

7.5.13 `DefaultSession::assertResults` should verify input `sessionId` belongs to a game associated with its instance

Description: `DefaultSession::assertResults` doesn't verify that the input `sessionId` belongs to a game associated with that instance of `DefaultSession`.

But different games can be associated with different instances of DefaultSession; see `SessionManager::getSessionStrategy`.

Consider this scenario:

- there are 2 games G1 and G2, each associated with a different instance of `DefaultSession` DS1 and DS2 but both DS1 and DS2 are associated with the same instance of `SessionManager`
- a good user calls `DS1::assertResults` with valid results to assert the results for G1
- a malicious user copies the exact inputs and calls `DS2::assertResults` with valid results to also assert the results for G1

In this state both DS1 and DS2 can have recorded winners for G1, even though DS2 isn't the correct strategy for G1.

This doesn't appear to be further abusable as `SessionManager::claimRewards` always gets the correct strategy instance and prevents winners from claiming more than once, but it doesn't seem like a good idea to allow this.

Recommended Mitigation: `DefaultSession::assertResults` should verify input `sessionId` belongs to a game associated with its instance.

Majority Games: Fixed in commit 462c01a.

Cyfrin: Verified.

7.5.14 `getReactionTime` is returning the `reactionDeadline` even if the user didn't participate in the game

Description: The `getReactionTime function` retrieves the reaction time for a player on a specific question:

```
function getReactionTime(uint256 questionId, address player) public view returns (uint256) {
        return _getReactionTime(questionId, player);
    }
    function _getReactionTime(uint256 questionId, address player) internal view returns (uint256) {
        return reactions[questionId][player].baseReaction.timestamp == 0
            ? revealedQuestions[questionId].reactionDeadline <------
            : reactions[questionId][player].baseReaction.timestamp - revealedAt[questionId];
    }
```

As you can see, if a user has never participated in the game, the function returns the maximum `reactionDeadline` for the question instead of reverting for a user who didn't commit a response or participate in the game.

Impact: The `getReactionTime` function returns an incorrect value for a user who didn't commit a response or participate in the game.

Recommended Mitigation: Consider return 0 or revert in `getReactionTime` if a user has never participated in a game.

Majority Games: This is the intended behavior but we updated. the natspec to make this explicit now.

Cyfrin: Verified.

7.6 Gas Optimization

7.6.1 Use `uint128` to pack `DepositManager::protocolFee`, `maxCreatorFee` into the same storage slot

Description: `DepositManager::protocolFee`, `maxCreatorFee` (and the same fields inside the `GamePool` struct) will always be < `BASIS_POINTS=10000`, so they can be declared as `uint128` to pack both of them into the same storage slot.

This means that functions such as `DepositManager::getRewards` which read both of them can perform only 1 storage read instead of 2.

Majority Games: Fixed in commit adedfc2.

Cyfrin: Verified.

7.6.2 Cache identical storage reads and only write to storage once

Description: Reading from storage is expensive; cache identical storage reads to prevent re-reading identical values from storage. Writing to storage is also expensive; increment cached values then write to storage only once when processing is complete:

- `DepositManager.sol`:

```
// cache `pool.token` in `sponsorGame` saves 1 storage read
135:         emit GameSponsored(gameId, msg.sender, pool.token, amount);
136:         SafeERC20.safeTransferFrom(IERC20(pool.token), msg.sender, address(this), amount);

// cache `gamePools[gameId].token` in `_claimReferralReward` saves 1 storage read
142:         SafeERC20.safeTransfer(IERC20(gamePools[gameId].token), msg.sender, referralReward);
143:         emit ReferralRewardClaimed(gameId, msg.sender, gamePools[gameId].token, referralReward);

// cache `pool.token`, `pool.totalCollectedAmount * pool.creatorFee / BASIS_POINTS`,
// `pool.totalCollectedAmount * pool.protocolFee / BASIS_POINTS` in `_distributeFees`
// to save 6 storage reads
152:             pool.token,
154:             pool.totalCollectedAmount * pool.creatorFee / BASIS_POINTS,
156:             pool.totalCollectedAmount * pool.protocolFee / BASIS_POINTS
158:         SafeERC20.safeTransfer(IERC20(pool.token), creator, pool.totalCollectedAmount *
↪  pool.creatorFee / BASIS_POINTS);
160:            IERC20(pool.token), protocolTreasury, pool.totalCollectedAmount * pool.protocolFee /
↪  BASIS_POINTS

// cache `sponsorAmounts[sponsor][gameId]` before the `require` check
// in `_refundSponsorFunds` saves 1 storage read, caching `pool.token` afterwards
// also saves 1 storage read
165:         require(sponsorAmounts[sponsor][gameId] > 0, AlreadyRefunded(sponsor, gameId));
167:         uint256 amount = sponsorAmounts[sponsor][gameId];
170:         emit RefundSponsorFunds(gameId, sponsor, pool.token, amount);
171:         SafeERC20.safeTransfer(IERC20(pool.token), sponsor, amount);

// cache `pool.ticketPrice` in `_refundEntryFee` saves 2 storage reads
183:         require(pool.totalCollectedAmount >= pool.ticketPrice, NotEnoughFunds(pool.token,
↪  pool.totalCollectedAmount));
186:         SafeERC20.safeTransfer(IERC20(pool.token), player, pool.ticketPrice);
187:         pool.totalCollectedAmount -= pool.ticketPrice;

// cache `pool.token`, `pool.ticketPrice` in `_payEntryFee` saves 7 storage reads
193:             IERC20(pool.token).balanceOf(player) >= pool.ticketPrice,
196:                 token: pool.token,
197:                 balance: IERC20(pool.token).balanceOf(player),
198:                 required: pool.ticketPrice
201:         SafeERC20.safeTransferFrom(IERC20(pool.token), player, address(this), pool.ticketPrice);
202:         pool.totalCollectedAmount += pool.ticketPrice;
```

```
203:         referralRewards[gameId][Registry(registry).referrers(player)] += pool.ticketPrice *
↪  REFERRER_FEE;
```

- QuestionManager.sol:

```
// cache `nextQuestionId` to save 3 storage reads per loop iteration in `_commitQuestions`
// writing to storage is also expensive so ideally only want to write to storage once when updating
// nextQuestionId. Do it like this to be much more efficient:

// cache prior to loop
uint256 nextQuestionIdCache = nextQuestionId;

for (uint256 i; i < _questionHashes.length; i++) {
    require(_questionHashes[i] != bytes32(0), InvalidQuestionHash(_gameId, i));

    // use cached value to save identical storage reads
    require(
        questionCommitment[nextQuestionIdCache].promptHash == bytes32(0),
        QuestionAlreadyCommitted(_gameId, nextQuestionIdCache)
    );
    gameQuestions[_gameId].push(nextQuestionIdCache);
    questionCommitment[nextQuestionIdCache] =
        PromptInitData({promptHash: _questionHashes[i], promptStrategy: _promptStrategies[i]});
    emit QuestionCommitted(_gameId, nextQuestionIdCache, _questionHashes[i], _promptStrategies[i]);

    // increment cache at end of each loop iteration
    nextQuestionIdCache++;
}

// once loop finished, write to storage once
nextQuestionId = nextQuestionIdCache;
```

- SessionManager.sol:

```
// cache `game.startTime` in `startAndRevealGameQuestion` saves ` storage read
402:         require(block.timestamp >= game.startTime, GameHasNotStartedYet(game.startTime,
↪  block.timestamp));
404:             block.timestamp <= game.startTime + revealGracePeriod,

// cache `questions.length` in `endGame` saves `questions.length - 1` storage reads
438:         for (uint256 i = 0; i < questions.length; i++) {

// cache `games[_gameId].state` in `cancelGame`, cancelGameIfCreatorMissing` saves 1 storage read
452:             games[_gameId].state != SessionState.Cancelled,
456:             games[_gameId].state != SessionState.Concluded,
465:             games[_gameId].state != SessionState.Cancelled,
469:             games[_gameId].state != SessionState.Concluded,
```

- DefaultSession.sol:

```
// cache `assertion.winners.length` in `recordResults`
172:         for (uint256 i = 0; i < assertion.winners.length; ++i) {

// cache `assertion.totalXPs[i], assertion.totalTimes[i]` in `recordResults`
// also at L180 instead of `assertion.sessionId` can use input `sessionId` as they
// were asserted equal at L164
177:                     questionIds[j], winner, assertion.totalXPs[i], assertion.totalTimes[i]
181:                 SessionResult({placement: i + 1, xp: assertion.totalXPs[i], time:
↪  assertion.totalTimes[i]});
```

- MajorityChoicePrompt.sol:

```
// cache `revealedAt[_questionId]` in `commitReaction`
```

```
// same thing applies in `TriviaChoicePrompt` & `SPBinaryPrompt` `commitReaction` function
106:         require(revealedAt[_questionId] != 0, QuestionNotRevealed(_questionId));
112:             revealedAt[_questionId] + revealedQuestions[_questionId].reactionDeadline >
↪  block.timestamp,
```

Majestic Games: Fixed in commit 4e56c11.

Cyfrin: Verified.

7.6.3 Prefer `calldata` to `memory` for external read-only function inputs

Description: Prefer `calldata` to `memory` for external read-only function inputs:

- `SessionManager::createGame` & `QuestionManager::_commitQuestions`:

```
224:         bytes32[] memory _promptHashes,
225:         address[] memory _promptStrategies,

45:     function _commitQuestions(uint256 _gameId, bytes32[] memory _questionHashes, address[] memory
↪  _promptStrategies)
```

- `DefaultSession::setXPTiers`:

```
100:     function setXPTiers(uint256 gameId, uint256[] memory _xpTiers) external {
```

Majestic Games: Fixed in commit be290a6.

Cyfrin: Verified.

7.6.4 In Solidity don't initialize to default values

Description: In Solidity don't initialize to default values:

```
session/DefaultSession.sol
125:         for (uint256 i = 0; i < questionIds.length; ++i) {
172:         for (uint256 i = 0; i < assertion.winners.length; ++i) {
174:             for (uint256 j = 0; j < questionIds.length; ++j) {

QuestionManager.sol
50:         for (uint256 i = 0; i < _questionHashes.length; i++) {

SessionManager.sol
154:     bool public livenessRequired = false;
159:     bool public creationSunsetted = false;
248:         for (uint256 i = 0; i < _promptStrategies.length; i++) {
366:         for (uint256 i = 0; i < questionIds.length; i++) {
438:         for (uint256 i = 0; i < questions.length; i++) {
474:         for (uint256 i = 0; i < questions.length; i++) {
549:         for (uint256 i = 0; i < _gameIds.length; i++) {

prompt/TriviaChoicePrompt.sol
108:         for (uint256 i = 0; i < questionIds.length; i++) {

offchain/uma/SessionResultAsserter.sol
121:         for (uint256 i = 0; i < addresses.length; i++) {
133:         for (uint256 i = 0; i < data.length; i++) {

reward/ProportionalToXPReward.sol
44:         for (uint256 i = 0; i < winners.length; ++i) {
50:         for (uint256 i = 0; i < winners.length; ++i) {
65:         for (uint256 i = 0; i < winners.length; ++i) {

reward/FixedRanksReward.sol
```

```
57:         uint256 totalPoints = 0;
58:         for (uint256 i = 0; i < _rankedRewards.length; ++i) {
76:         for (uint256 i = 0; i < winners.length; ++i) {
```

Majestic Games: Fixed in commit 6686df5.

Cyfrin: Verified.

7.6.5 Perform input-related checks prior to reading storage

Description: Since reading from storage is expensive, it is more efficient to "fail fast" by performing input-related checks prior to reading storage:

- `SessionManager.sol`:

```
// in `createGame`, perform this check before all the others
require(_promptStrategies.length == _promptHashes.length, ArrayLengthMismatch());
```

Majority Games: Fixed in commit 02b8fd8.

Cyfrin: Verified.

7.6.6 More efficient implementation of `SessionManager::joinGame` via better storage packing

Description: `SessionManager::joinGame` performs these 4 storage reads:

```
// reads games[_gameId].state up to 2 times
require(
    games[_gameId].state == SessionState.Created || games[_gameId].state == SessionState.Ongoing,
    InvalidGameState(SessionState.Created, games[_gameId].state)
);
// reads games[_gameId].numContestants once
require(
    games[_gameId].numContestants < maximumContestants,
    TooManyContestants(maximumContestants, games[_gameId].numContestants)
);
// reads games[_gameId].verificationRequired once
if (games[_gameId].verificationRequired) {
    require(isVerificationApproved[msg.sender], NotVerified(msg.sender));
}
```

The `Game` struct can be refactored to pack `state`, `numContestants` and `verificationRequired` into the same storage slot like this:

```
struct Game {
    uint256 gameId;
    uint256 startTime;
    uint256 endTime;
    address sessionStrategy;
    address rewardStrategy;
    uint256 originalStartTime;
    address creator;
    address creatorfeeReceiver;
    uint32 numContestants;
    SessionState state;
    bool verificationRequired;
}
```

Then all 3 can be read inside `SessionManager::joinGame` through just one storage read:

```
Game storage gameRef = games[_gameId];
(uint32 numContestants, SessionState state, bool verificationRequired)
```

```
    = (gameRef.numContestants, gameRef.state, gameRef.verificationRequired);

// remaining checks/processing follows as normal
```

Majority Games: Fixed in commit c7eafa2.

Cyfrin: Verified.

7.6.7 Use `uint32` for timestamps for better storage packing

Description: The maximum value of `uint32` is 4294967295 which is 2106/02/07 - likely far longer than required by this protocol! Using `uint32` instead of `uint256` for timestamps and making sure those variables are adjacent to each-other can result in significantly reducing the amount of storage slots required:

- `SessionManager::Game::startTime, endTime, originalStartTime`
- `SessionManager::minimumStartDelay, maxGameDuration, revealGracePeriod, livenessDuration`
- `DefaultSession::SessionResult::time`

Majority Games: Fixed in commit 5902894.

Cyfrin: Verified.

7.6.8 Don't copy entire `Assertion` struct from `storage` to `memory` in `DefaultSession::assertionResolvedCallback`

Description: The `Assertion` struct is defined as:

```
struct Assertion {
    uint256 sessionId;
    string resultCid;
    string calculationCid;
    address asserter;
    bool resolved;
    address[] winners;
    uint256[] totalXPs;
    uint256[] totalTimes;
}
```

It is very inefficient to copy this entire struct from `storage` to `memory`. Yet `DefaultSession::assertionResolvedCallback` does exactly this even though it only needs 4 fields:

```
if (assertedTruthfully) {
    assertions[assertionId].resolved = true;
    Assertion memory dataAssertion = assertions[assertionId];
    emit DataAssertionResolved(
        dataAssertion.sessionId,
        dataAssertion.resultCid,
        dataAssertion.calculationCid,
        dataAssertion.asserter,
        assertionId
    );
    recordResults(assertions[assertionId].sessionId, assertionId);
```

Recommended Mitigation: Use a storage reference like this:

```
        if (assertedTruthfully) {
            assertions[assertionId].resolved = true;
-           Assertion memory dataAssertion = assertions[assertionId];
+           Assertion storage dataAssertion = assertions[assertionId];
            emit DataAssertionResolved(
                dataAssertion.sessionId,
```

```
                dataAssertion.resultCid,
                dataAssertion.calculationCid,
                dataAssertion.asserter,
                assertionId
            );
```

Majority Games: Fixed in commit fc5e0fa.

Cyfrin: Verified.