Zellic

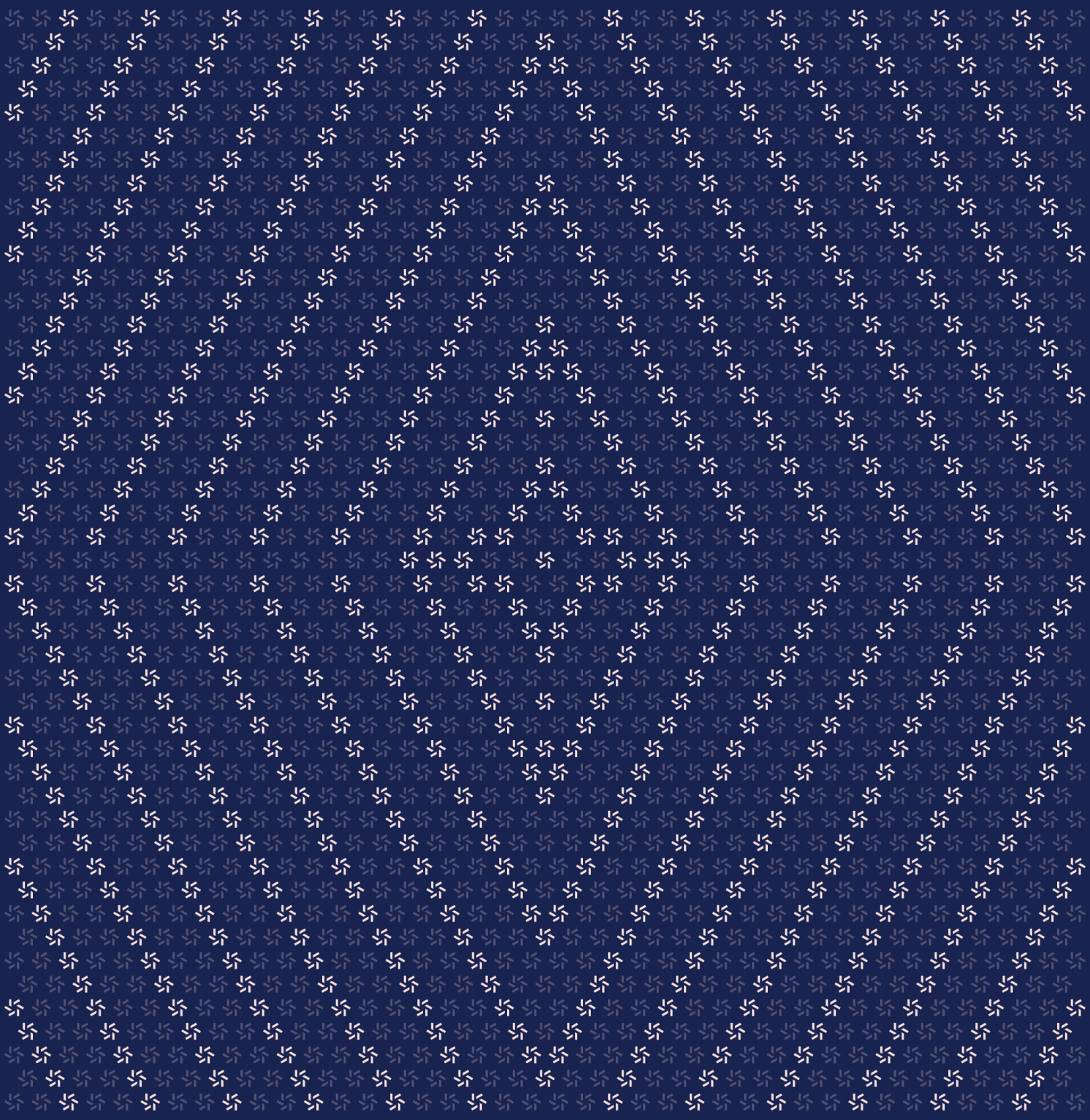
Prepared for
Patrick Lung
Brian Weickmann
Coordination Inc

Prepared by
Filipe Alves
Quentin Lemauf
Ayaz Mammadov
Zellic

October 17, 2025

Megapot v2

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Coordination Inc from October 6th to October 16th, 2025. During this engagement, Zellic reviewed Megapot v2's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Is there any way to drain funds in the jackpot via LP or referrer deposit/withdraw flows?
- Is there any way to drain the jackpot by falsifying tickets?
- Is the jackpot truly fair, or can it be exploited (i.e., is randomness being correctly used and creating truly random outputs)?
- Is there any way that the jackpot could end up being -EV for LPs? Can a minimum amount of edge be guaranteed?
- Is there any way the jackpot could end up undercollateralized via accounting errors — either business logic (i.e., not accounting for all ticket winners) or rounding errors (especially accrued over time, rounding should be conservative with respect to collateralization)?
- Is there any way that LPs, referrers, or users could not be paid out what they are owed due to faulty state tracking or math (i.e., not all ticket winners are accounted for)?
- Is there any way to lock funds in the jackpot for any user — winners, referrers, LPs?
- Is there any way that the jackpot could potentially get stuck and be unable to progress to the next drawing?
- Can EIP-712 signatures be exploited as part of the bridging manager to either steal someone's tickets or otherwise interfere with accounting, either from signature replays or attempting hash collision?
- Is the case where the guaranteed payouts exceed the total value of the pool adequately handled?
- Is all the bitpacking logic sound? Are there any potential boundary errors that could arise either between the lower bits where the normals are or the higher bits where powerball must be less than `255 - normalBall Max`?
- Can admin changes (e.g., `ticketPrice`, `normalBallMax`, fees) made mid-drawing create inconsistent states or violate expectations for players/LPs?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Megapot v2 contracts, we discovered 12 findings. One critical issue was found. Three were of high impact, one was of medium impact, and seven were of low impact.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Coordination Inc in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	3
Medium	1
Low	7
Informational	0

2. Introduction

2.1. About Megapot v2

Coordination Inc contributed the following description of Megapot v2:

> Megapot is jackpot protocol that matches risk-on LPs seeking high yields with users looking to take long odds bets.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood.

We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Megapot v2 Contracts

Type	Solidity
Platform	EVM-compatible
Target	megapot-v2-contracts
Repository	https://github.com/coordinationlabs/megapot-v2-contracts ↗
Version	`c8b14683caa8f32c713ee9bf974534cb92791e4b`

Programs	`lib/*.sol`
	`mock/*.sol`
	`GuaranteedMinimumPayoutCalculator.sol`
	`Jackpot.sol`
	`JackpotBridgeManager.sol`
	`JackpotLPManager.sol`
	`JackpotTicketNFT.sol`
	`ScaledEntropyProvider.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 2.6 person-weeks. The assessment was conducted by three consultants over the course of 1.8 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

Pedro Moura
Engagement Manager
pedro@zellic.io

The following consultants were engaged to conduct the assessment:

Filipe Alves
Engineer
filipe@zellic.io

Quentin Lemauf
Engineer
quentin@zellic.io

Ayaz Mammadov
Engineer
ayaz@zellic.io

2.5. Project Timeline

The key dates of the engagement are detailed below.

October 6, 2025	Kick-off call
October 6, 2025	Start of primary review period
October 16, 2025	End of primary review period

3. Detailed Findings

3.1. Randomness can be manipulated

Target	Jackpot.sol		
Category	Business Logic	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The function `runJackpot` used to start the jackpot flow and eventually gather randomness for finding the winning tickets does so by calling into the ScaledEntropyProvider. The ScaledEntropyProvider is a contract meant for use not only by Megapot but by all users. It acts as an abstraction layer over Pyth to generate randomness and then scale it to generate specific numbers, which will be interpreted as the normal ball / powerball. The randomness is generated in another block, and then a callback to the jackpot is done.

```
function runJackpot() external payable noEmergencyMode {
    ...
    entropy.requestAndCallbackScaledRandomness{value: fee}(
        entropyGasLimit,
        setRequests,
        address(this),
        this.scaledEntropyCallback.selector,
        bytes("")
    );
    ...
}
```

```
function scaledEntropyCallback(
    bytes32,
    uint8[][] memory _randomNumbers,
    bytes memory
)
    external
    onlyEntropy
    nonReentrant
```

There is a permission check that `msg.sender` is the ScaledEntropyProvider. However, the ScaledEntropyProvider itself is not permissioned. Consequently, anyone can request randomness and point the callback to the Jackpot contract; this is especially problematic as the ScaledEntropyProvider takes parameters that allow for reducing the range of randomness.

Specifically, this is done during the `entropyCallback` after Pyth returns randomness to the ScaledEntropyProvider.

```
function entropyCallback(uint64 sequence, address /*provider*/,
    bytes32 randomNumber) internal override {
    PendingRequest memory req = pending[sequence];
    if (req.callback == address(0)) revert UnknownSequence();

    delete pending[sequence];

    uint8[][] memory scaledRandomNumbers = _getScaledRandomness(randomNumber,
    req.setRequests);
```

As such, there is a possible transaction-sandwiching situation where a malicious user can create a malicious request pointed to the jackpot with parameters to force the randomness generated to result in a specific number, allowing them to control the powerball numbers.

The order of operations would be as such:

1. Jackpot calls `runJackpot`, a request for randomness.
2. A malicious user sends a TX request to ScaledEntropyProvider, targeted towards Jackpot.
3. Jackpot receives the malicious user's randomness request first.
4. ScaledEntropyCallback is run with manipulated numbers.

Impact

Malicious users can win the jackpot every time by manipulating the numbers in the powerball.

Recommendations

Bind some information to the randomness request that allows Jackpot to identify where the randomness response from ScaledEntropyCallback originates. This could be done using the sequence number returned by `ScaledEntropyCallback.requestAndCallbackScaledRandomness`. Alternatively, ScaledEntropyCallback could store and send the caller of a randomness request in the callback. Either method would work in eliminating malicious randomness requests.

Remediation

This was remediated in commit 88b43e4061123b95797fe985e1614ed64a3e1179 ↗ by removing the ability to target callbacks from `ScaledEntropyProvider`. All requests for randomness are returned to the `msg.sender`

3.2. Incorrect payout distribution when prize pool equals minimum payout allocation

Target	GuaranteedMinimumPayoutCalculator		
Category	Business Logic	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

The `JackpotWinningManager::calculateAndStoreDrawingUserWinnings` function contains a logic flaw where tier-based premium payouts are eliminated in the rare edge case when the prize pool exactly equals the minimum payout allocation. While the likelihood of this occurring is very low, when it happens, all winners in minimum payout tiers receive only the flat `minPayout` amount regardless of their tier, while winners in non-minimum tiers receive nothing.

```
function calculateAndStoreDrawingUserWinnings(
    uint256 _drawingId,
    uint256 _prizePool,
    uint8 _normalMax,
    uint8 _powerballMax,
    uint256[] memory _uniqueResult,
    uint256[] memory _dupResult
) external onlyJackpot returns (uint256 totalPayout) {
    [...]
    for (uint256 i = 0; i < TOTAL_TIER_COUNT; i++) {
        tierWinners[i] = tierWinningTickets;
        if (tierInfo.minPayoutTiers[i]) {
            minimumPayoutAllocation += tierWinningTickets
    * tierInfo.minPayout;
        }
    }

    totalPayout = _calculateAndStoreTierPayouts(
        _drawingId,
      minimumPayoutAllocation > _prizePool ? _prizePool : _prizePool -
          minimumPayoutAllocation,
      minimumPayoutAllocation > _prizePool ? 0 : tierInfo.minPayout,
        tierWinners,
        _uniqueResult,
        _dupResult
    );
```

```
}
```

When `_prizePool == minimumPayoutAllocation`, the function passes `_remainingPrizePool = 0` to `_calculateAndStoreTierPayouts`:

```
function _calculateAndStoreTierPayouts(
    uint256 _drawingId,
    uint256 _remainingPrizePool,
    uint256 _minPayout,
    uint256[TOTAL_TIER_COUNT] memory _tierWinners,
    uint256[] memory _uniqueResult,
    uint256[] memory _dupResult
) internal returns(uint256 totalPayout) {

    for (uint256 i = 0; i < TOTAL_TIER_COUNT; i++) {
        if (_tierWinners[i] != 0) {
            uint256 premiumTierPayoutAmount = _remainingPrizePool * tierInfo.
                premiumTierWeights[i] /
                  (PRECISE_UNIT * _tierWinners[i]);

            uint256 tierPayout = tierInfo.minPayoutTiers[i] ?

    _minPayout + premiumTierPayoutAmount :  premiumTierPayoutAmount;

            tierPayouts[_drawingId][i] = tierPayout;
            totalPayout += tierPayout * (_uniqueResult[i] + _dupResult[i]);
        }
    }
}
```

This causes `premiumTierPayoutAmount` to become zero for all tiers. As a result, winners will receive a payout equal to either the `_minPayout` argument when a tier is eligible to receive a minimum payout or 0 otherwise.

Impact

The tier-based reward structure collapses when the prize pool equals the minimum payout allocation. Jackpot winners receive only the flat minimum payout instead of their expected tier-weighted premium, while non-minimum tier winners receive nothing despite having valid winning tickets. Critically, if the jackpot tier is not eligible for minimum payout, the jackpot winner receives nothing despite matching all numbers.

Recommendations

Modify the calculation to guarantee a minimum amount is always allocated for premium distribution. Instead of allowing `_remainingPrizePool` to become zero when `prizePool == minimumPayoutAllocation`, reserve a portion of the prize pool for tier-weighted premium payouts to prevent the reward structure from collapsing.

Additionally, apply a tier-based weighting strategy to minimum payouts instead of using a flat `_minPayout` for all eligible tiers. Minimum payouts should be distributed proportionally according to tier hierarchy to ensure higher-tier winners receive proportionally larger minimum payouts than lower-tier winners.

Remediation

This issue has been acknowledged by Coordination Inc, and a fix was implemented in commit `d68a772d` ↗.

3.3. Arbitrary calls

Target	JackpotBridgeManager		
Category	Business Logic	**Severity**	Critical
Likelihood	Low	**Impact**	High

Description

In the case in which `_claimedAmount` is 0, a malicious user can do any arbitrary calls with any arbitrary calldata.

```
function _bridgeFunds(RelayTxData memory _bridgeDetails,
    uint256 _claimedAmount) private {
    ...
    (bool success,) = _bridgeDetails.to.call(_bridgeDetails.data);
    ...
    if (preUSDCBalance - postUSDCBalance != _claimedAmount)
    revert NotAllFundsBridged();
    ...
}
```

Impact

Any approval given to the JackpotBridgeManager can be stolen. However, in practice, no users should give approval to the JackpotBridgeManager, and any approval given by a bridge should be atomically used.

Recommendations

Ensure that `_claimedAmount` cannot be 0, and ideally enforce a whitelist on `bridgeDetails.to`.

Remediation

This issue was remediated in commit 642a64a8 ↗ by ensuring that claim amount cannot be 0.

However, this remediation may be partial. While it prevents allowance stealing in the vast majority of tokens, there remain edge cases where arbitrary calls could still be exploited. For example, a token with an exposed `multicall` function could potentially allow an attacker to:

1. First invoke a function that moves USDC from the `JackpotBridgeManager` to satisfy the balance check
2. Then invoke a function to steal any granted allowances from that token

Such an exploit would require specific conditions:

- A token with external allowances granted to the `JackpotBridgeManager`
- The same token having sufficient logic complexity (like `multicall`) to bypass USDC balance checks while stealing allowances
- The attacker having a winning ticket to trigger the bridge function

While no popular ERC20 tokens with these characteristics were immediately identified, the possibility of such tokens existing or being created in the future represents a residual risk. Consider implementing additional safeguards such as a whitelist for `_bridgeDetails.to` addresses to fully eliminate arbitrary call risks.

3.4. Incorrect tier calculation using global normalBallMax instead of drawing-specific ballMax

Target	Jackpot.sol		
Category	Business Logic	**Severity**	High
Likelihood	High	**Impact**	High

Description

`Jackpot::claimWinnings` contains a logic error where it uses the global `normalBallMax` variable instead of the drawing-specific `ballMax` value from `drawingState` when calculating the tier ID for winning tickets. This causes incorrect tier identification when the `drawingState[drawingId].ballMax` is different from the global `normalBallMax`, potentially resulting in incorrect payout amounts.

```
function claimWinnings(uint256[] memory _userTicketIds)
    external nonReentrant {
    [...]
    for (uint256 i = 0; i < _userTicketIds.length; i++) {
        [...]

    uint256 tierId = _calculateTicketTierId(ticketInfo.ticket, drawingState
          [drawingId].winningTicket, normalBallMax);
        jackpotNFT.burnTicket(ticketId);

        uint256 winningAmount = payoutCalculator.getTierPayout(drawingId,
    tierId);
        uint256 referrerShare
    = _payReferrersWinnings(ticketInfo.referralScheme, winningAmount);

        totalClaimAmount += winningAmount - referrerShare;
        emit TicketWinningsClaimed(
            msg.sender,
            drawingId,
            ticketId,
            tierId / 2,
            (tierId % 2) == 1,
            winningAmount - referrerShare
        );
    }
```

```
        usdc.transfer(msg.sender, totalClaimAmount);
    }
```

Impact

When the drawing's `ballMax` differs from the global `normalBallMax`, tickets are incorrectly unpacked causing wrong tier identification and incorrect payout amounts.

Recommendations

Replace the global `normalBallMax` with the drawing-specific `ballMax` value stored in `drawingState[drawingId]`:

```
uint256 tierId = _calculateTicketTierId(ticketInfo.ticket, drawingState[
    drawingId].winningTicket, normalBallMax);
uint256 tierId = _calculateTicketTierId(ticketInfo.ticket, drawingState[
    drawingId].winningTicket, drawingState[drawingId].ballMax);
```

Remediation

This issue has been acknowledged by Coordination Inc, and a fix was implemented in commit 250b1977 ↗.

3.5. Insufficient test coverage

Target	Project Wide		
Category	Protocol Risks	**Severity**	Medium
Likelihood	N/A	**Impact**	Medium

Description

When building a complex contract with multiple moving parts (state changes) and dependencies, comprehensive testing is essential. This includes testing for both positive and negative scenarios on every function that touches the contract's state.

Positive tests should verify that each function's side effect is as expected, while negative tests should cover every revert, preferably in every logical branch.

The test coverage for this project should be expanded to include all contracts, and all functions — not just surface-level functions. It is important to test the invariants required for ensuring security and also verify any mathematical properties from the project's specification. Additionally, testing cross-chain function calls and transfers is recommended to ensure the desired functionality.

Impact

Good test coverage has multiple effects:

- It finds bugs and design flaws early (preaudit or prerelease).
- It gives insight into areas for optimization (e.g., gas cost).
- It displays code maturity.
- It bolsters customer trust in the product.
- It improves understanding of how the code functions, integrates, and operates — for developers and auditors alike.
- It increases development velocity long-term.

The last point may seem contradictory given the time investment to create and maintain tests. To expand upon it, tests help developers trust their own changes. It is difficult to know if a code refactor — or even just a small one-line fix — breaks other code if there are no tests. This is especially true for new developers or those returning to the code after a prolonged absence.

Tests have your back here. They are an excellent indicator that the existing functionality was most likely not broken by a change to the code. Without a comprehensive test suite, the above benefits are lost. This increases the likelihood of bugs and vulnerabilities going unnoticed until after deployment, which can be costly and damaging to the project's reputation.

Recommendations

We recommend building a rigorous test suite, including end-to-end tests, that includes all contracts to ensure that the system operates securely and as intended.

Remediation

This issue has been acknowledged by Coordination Inc.

3.6. Nonce overflow in FisherYatesRejection prevents drawing completion with large range sizes

Target	FisherYatesRejection		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	Low

Description

The `FisherYatesRejection::draw` function uses a `uint8` for the `nonce`, which can overflow when the algorithm requires more than 255 iterations. The nonce is incremented both during rejection sampling (when random values fall outside the acceptable range) and at the end of each loop iteration. If the total number of increments exceeds 255, Solidity's overflow protection causes the transaction to revert.

```
function draw(
    uint8 minRange,
    uint8 maxRange,
    uint256 count,
    uint256 seed
) external pure returns (uint8[] memory result) {
    require(count <= maxRange - minRange + 1, "Too many draws");

   uint8 nonce = 0;

    // Fisher-Yates shuffle with rejection sampling
    for (uint256 i = rangeSize - 1; i > 0; i--) {
        uint256 rand;
        while (true) {
            rand = uint256(keccak256(abi.encode(seed, nonce)));
            uint256 limit = (MAX_UINT / (i + 1)) * (i + 1);
            if (rand < limit) {
                rand = rand % (i + 1);
                break;
            }
           nonce++;
        }
        [...]
       nonce++;
    }
    [...]
```

```
}
```

While the current Jackpot implementation uses small range sizes (making overflow unlikely), the FisherYatesRejection library is designed as a generic, reusable component for use across different projects. However, with a sufficiently large `rangeSize` (e.g., > 255), the nonce will overflow purely from loop iterations, making the library unsuitable for general-purpose use.

Impact

This has direct and indirect impacts.

- **Direct impact on Megapot.** While extremely unlikely due to small range sizes in the current implementation, a remote but real chance exists for nonce overflow if the combination of range size and rejection sampling iterations exceeds 255. When this occurs, `ScaledEntropyProvider::entropyCallback` fails, leaving the Jackpot contract locked until manual admin intervention via `unlockJackpot`, halting all protocol operations. Critically, this prevents legitimate winners from being determined — a potential winning ticket is denied its payout through no fault of the holder, as the drawing cannot complete.
- **Indirect impact on library reusability.** The library is designed as a generic component for reuse across different projects, but the `uint8` nonce limitation makes it fundamentally broken for general-purpose use. For any implementation using `rangeSize > 255`, nonce overflow is guaranteed.

Recommendations

Change the `nonce` type from `uint8` to `uint256` to prevent overflow:

```
function draw(
    uint8 minRange,
    uint8 maxRange,
    uint256 count,
    uint256 seed
) external pure returns (uint8[] memory result) {
    require(count <= maxRange - minRange + 1, "Too many draws");

    uint8 nonce = 0;
    uint256 nonce = 0;

    for (uint256 i = rangeSize - 1; i > 0; i--) {
        [...]
        nonce++;
    }
    [...]
```

```
}
```

Remediation

This issue has been acknowledged by Coordination Inc, and a fix was implemented in commit 8952253b ↗.

3.7. Inconsistent cap validation allows LP pool to exceed maximum capacity

Target	JackpotLPManager		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `JackpotLPManager::setLPPoolCap` function only validates the new cap against `lpPoolTotal`, ignoring `pendingDeposits`, which will be added when the drawing settles. This is inconsistent with `JackpotLPManager::processDeposit`, which correctly checks `lpPoolTotal + pendingDeposits` to prevent exceeding the cap. This validation gap allows the invariant `lpPoolTotal <= lpPoolCap` to be violated when an admin reduces the cap while deposits are pending.

```
function processDeposit(uint256 _drawingId, address _lpAddress,
    uint256 _amount) external onlyJackpot() {
   uint256 totalPoolValue = lpDrawingState[_drawingId].lpPoolTotal +
       lpDrawingState[_drawingId].pendingDeposits;

    if (_amount + totalPoolValue > lpPoolCap) revert JackpotErrors.ExceedsPoolC
       ap();
    [...]
}
[...]
function setLPPoolCap(uint256 _drawingId, uint256 _lpPoolCap)
    external onlyJackpot() {
    if (_lpPoolCap < lpDrawingState[_drawingId].lpPoolTotal) revert
        InvalidLPPoolCap();
     lpPoolCap = _lpPoolCap;
}
```

Impact

This vulnerability breaks the core invariant `lpPoolTotal <= lpPoolCap`, allowing the LP pool to exceed its capacity and undermining risk-management controls. Once violated, the protocol enters an inconsistent state where new deposits are rejected but the pool remains over capacity until withdrawals restore the limit.

Recommendations

Validate against the complete future state of the pool:

```
function setLPPoolCap(uint256 _drawingId, uint256 _lpPoolCap)
    external onlyJackpot() {
   uint256 totalPoolValue = lpDrawingState[_drawingId].lpPoolTotal +
       lpDrawingState[_drawingId].pendingDeposits;


    if (_lpPoolCap < lpDrawingState[_drawingId].lpPoolTotal) revert InvalidLPPo
       olCap();
   if (_lpPoolCap < totalPoolValue) revert InvalidLPPoolCap();
    lpPoolCap = _lpPoolCap;
}
```

Additionally, consider emitting events for better admin visibility into cap changes and current pool state.

Remediation

This issue has been acknowledged by Coordination Inc, and a fix was implemented in commit 44700db7 ↗.

3.8. Missing reentrancy protection in `Jackpot::runJackpot`

Target	Jackpot		
Category	Coding Mistakes	**Severity**	Low
Likelihood	High	**Impact**	Low

Description

The `Jackpot::runJackpot` function lacks a `nonReentrant` modifier, creating an inconsistency with the contract's security pattern where all other external state-changing functions use reentrancy protection. The function refunds excess fees to `msg.sender` via a low-level call before completing execution, creating a reentrancy window.

```
function runJackpot() external payable noEmergencyMode {
    DrawingState storage currentDrawingState = drawingState[currentDrawingId];
    if (currentDrawingState.jackpotLock) revert JackpotErrors.JackpotLocked();
    if (currentDrawingState.drawingTime >= block.timestamp)
    revert JackpotErrors.DrawingNotDue();

    _lockJackpot();

    uint32 entropyGasLimit = entropyBaseGasLimit + entropyVariableGasLimit
    * uint32(currentDrawingState.powerballMax);
    uint256 fee = entropy.getFee(entropyGasLimit);
    if (msg.value < fee) revert JackpotErrors.InsufficientEntropyFee();
    if (msg.value > fee) {
        // Refund excess - creates reentrancy opportunity
       (bool success, ) = msg.sender.call{value: msg.value - fee}("");
        require(success, "Transfer failed");
    }
    [...]
}
```

While the function sets `jackpotLock` via `_lockJackpot()` before the external call, preventing traditional same-function reentrancy, the absence of `nonReentrant` is inconsistent with the contract's security pattern and increases the potential attack surface for cross-function reentrancy scenarios.

Impact

The missing `nonReentrant` modifier creates an inconsistency with the established security standard where all other external state-changing functions employ reentrancy protection. This opens a theoretical window for both same-function reentrancy (prevented by the `jackpotLock` check) and cross-function reentrancy attacks. While no current functions are vulnerable due to proper `jackpotLock` checks and adherence to the checks-effects-interactions pattern, this inconsistency unnecessarily increases the attack surface and suggests a potential oversight in the contract's security implementation.

Recommendations

Add the `nonReentrant` modifier to maintain consistency with the contract's security standard:

```
function runJackpot() external payable noEmergencyMode {
function runJackpot() external payable nonReentrant noEmergencyMode {
    DrawingState storage currentDrawingState
   = drawingState[currentDrawingId];
    if (currentDrawingState.jackpotLock)
   revert JackpotErrors.JackpotLocked();
    if (currentDrawingState.drawingTime >= block.timestamp)
   revert JackpotErrors.DrawingNotDue();

    _lockJackpot();
    [...]
}
```

Remediation

This issue has been acknowledged by Coordination Inc, and a fix was implemented in commit `56cd7cca` ↗.

3.9. Bridge griefing

Target	JackpotBridgeManager		
Category	Business Logic	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

When the user calls the function `claimWinnings`, the user's winnings will be bridged to a bridge of their choice when the function `_bridgeFunds` is invoked.

```
function _bridgeFunds(RelayTxData memory _bridgeDetails,
    uint256 _claimedAmount) private {
    uint256 preUSDCBalance = usdc.balanceOf(address(this));
    (bool success,) = _bridgeDetails.to.call(_bridgeDetails.data);

    if (!success) revert BridgeFundsFailed();
    uint256 postUSDCBalance = usdc.balanceOf(address(this));

    if (preUSDCBalance - postUSDCBalance != _claimedAmount)
    revert NotAllFundsBridged();

    emit FundsBridged(_bridgeDetails.to, _claimedAmount);
}
```

There is a check that the bridged amounts were fully exactly bridged. This could cause issues with integrations or open up possible griefing attacks where an external callback could donate 1 WEI of USDC to cause the check to fail. Any bridge that does not take exact amounts, or possibly leaves dust due to decimal conversions or other miscellaneous factors, could also revert.

Impact

This could lead to potential griefing attacks that could cause user funds to not be bridged, though this is unlikely as the user specifies the bridge and there is no incentive. Alternatively, it could lead to difficulty integrating with bridges that do not bridge exact amounts.

Recommendations

Ensure that the bridges that will be integrated account for these details.

Remediation

This issue has been acknowledged by Coordination Inc, and they provided the following response:

> We prefer stricter balance checks and are confident our bridging provider can support the current logic.

3.10. Division by zero in LP accumulator freezes jackpot after full withdrawals

Target	JackpotLPManager		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

The `JackpotLPManager::processDrawingSettlement` function blindly divides by `currentLP.lpPoolTotal` when updating the accumulator for drawings `> 0`. If the previous drawing settled with `newLPValue == 0`, the next drawing is initialized with `lpPoolTotal = 0`. Settling that drawing therefore reverts due to division by zero, halting `Jackpot::scaledEntropyCallback` and keeping the drawing locked until manual intervention (`Jackpot::unlockJackpot`).

This zero state arises naturally when LPs collectively withdraw their entire position. During the settlement of drawing n, the pool balance is recomputed as `newLPValue = postDrawLpValue + currentLP.pendingDeposits - withdrawalsInUSDC`. When the withdrawals consume everything that was left (and no fresh deposits arrive), this arithmetic legitimately yields `newLPValue == 0`. The next drawing then inherits that zero via `Jackpot::_setNewDrawingState`, which seeds the manager with `JackpotLPManager::initializeDrawingLP(currentDrawingId, _newLpValue);`.

```
function _setNewDrawingState(...) internal {
    ...
    jackpotLPManager.initializeDrawingLP(currentDrawingId, _newLpValue);
    ...
}
```

Once drawing $n + 1$ settles, the accumulator update still executes `newAccumulator = (drawingAccumulator[_drawingId - 1] * postDrawLpValue) / currentLP.lpPoolTotal;`, so the division by zero reverts and bricks the callback.

```
function processDrawingSettlement(
    ...
)   ... {
    LPDrawingState storage currentLP = lpDrawingState[_drawingId];
    uint256 postDrawLpValue = currentLP.lpPoolTotal + _lpEarnings
    - _userWinnings - _protocolFeeAmount;

    if (_drawingId > 0) {
       newAccumulator = (drawingAccumulator[_drawingId - 1] * postDrawLpValue)
    / currentLP.lpPoolTotal;
```

```
            drawingAccumulator[_drawingId] = newAccumulator;
        }

        uint256 withdrawalsInUSDC = currentLP.pendingWithdrawals * newAccumulator
        / PRECISE_UNIT;
        newLPValue = postDrawLpValue + currentLP.pendingDeposits
        - withdrawalsInUSDC;
    }
```

Impact

Once `lpPoolTotal` hits zero, the jackpot settlement will revert, so the drawing stays locked, ticket sales and LP flows revert, and only manual owner action plus liquidity restores the system.

Recommendations

Treat a zero pool as a special case by resetting the accumulator without division before processing withdrawals:

```
if (_drawingId > 0) {
    newAccumulator = (drawingAccumulator[_drawingId - 1] * postDrawLpValue) /
        currentLP.lpPoolTotal;
    if (currentLP.lpPoolTotal == 0) {
        newAccumulator = PRECISE_UNIT;
    } else {

      newAccumulator = (drawingAccumulator[_drawingId - 1] * postDrawLpValue)
            / currentLP.lpPoolTotal;
    }
      drawingAccumulator[_drawingId] = newAccumulator;
}
```

Remediation

This issue has been acknowledged by Coordination Inc, and a fix was implemented in commit 49e6b50e ↗.

3.11. Lack of safe ERC-20 functions

Target	Project Wide		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

In many areas of the project, nonsafe functions are used. For example, `transfer` and `transferFrom` are used instead of their safer counterparts `safeTransfer` and `safeTransferFrom`.

Impact

Some ERC20s just return false when a transfer fails instead of reverting. Using SafeERC20 wrappers enforces a revert in those cases, keeping behavior consistent, preventing silent failures, and avoiding divergent state or stranded funds.

```
function finalizeWithdraw() external nonReentrant noEmergencyMode {
        uint256 withdrawableAmount
    = jackpotLPManager.processFinalizeWithdraw(currentDrawingId, msg.sender);
@>      usdc.transfer(msg.sender, withdrawableAmount);
    }
```

Recommendations

Use the safe counterparts.

Remediation

This issue was remediated in commit 01bea4b2f0b8b6fc6a9ab70f360af1970fcb1734 ↗ by using the safe counterparts.

3.12. USDT allowance quirk

Target	Project Wide		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

As of the time of writing, this project is considering using or deploying with USDT as a supported asset. In many areas of the project, USDT's allowance quirks are not accounted for:

```
function _bridgeFunds(RelayTxData memory _bridgeDetails,
    uint256 _claimedAmount) private {
    if (_bridgeDetails.approveTo != address(0)) {
        usdc.approve(_bridgeDetails.approveTo, _claimedAmount);
    }
```

USDT requires that allowance is `0` before an approval is done to prevent a double-spend attack on allowances.

Impact

We have not identified any areas where allowances are not fully spent. However, if there is any remaining allowance when invoking `approve`, then the invocations of `approve` may fail.

Recommendations

Set the allowance to zero before invoking `approve`.

Remediation

This issue has been acknowledged by Coordination Inc, and they provided the following response:

> We have noted this in case we decide to make a future USDT version of the protocol - for now the focus is USDC.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Minimum payout allocation and prize-pool distribution

The minimum payout for a powerball combination is 1 USDC regardless of its tier, whether it is the jackpot or the minimum guaranteed tier (e.g., a two normal-ball combo).

If the prize pool does not have enough funds to guarantee minimum payouts, then it foregoes them completely, instead opting for scaling the prize pool according to the premium weights of each tiers.

In the case in which minimum payouts (~650K USDC) are covered by the price pool, then the remaining funds are distributed according to the weights of the tiers.

This leaves an unexpected situation when the the prize pool barely covers the minimum allocation. In this worst-case scenario where there is no premium funds at all, with a prize pool of around ~650K, a jackpot winner and the smallest guaranteed payout tier would have the same payout of 1 USDC.

It is important that this quirk of the Powerball is explained and clear for users/LPs. Specifically this case, where the Jackpot winner's prize grows when there is no minimum guaranteed prizepool (~650k USDC).

However, the moment the minimum guaranteed payout breakpoint is hit (~650k USDC) the Jackpot winner's drops to almost nothing (1 USDC) and has to grow again.

5. Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm.

Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1. Module: Jackpot.sol

Function: `buyTickets(Ticket[] _tickets, address _recipient, address[] _referrers, uint256[] _referralSplit, bytes32 _source)`

This function handles a full ticket order, sanity-checks the batch, charges the caller in USDC, mints NFTs to the recipient, and records referral payouts when asked.

Inputs

- `_tickets`
  - **Control**: Provided by the caller.
  - **Constraints**: Must include at least one entry; every ticket needs five distinct normals within `[1, ballMax]` plus a powerball within `[1, powerballMax]`.
  - **Impact**: Determines the tickets and duplicate detection.
- `_recipient`
  - **Control**: Provided by the caller.
  - **Constraints**: Cannot be the zero address.
  - **Impact**: Receives ownership of all freshly minted ticket NFTs.
- `_referrers`
  - **Control**: Provided by the caller.
  - **Constraints**: Length must equal `_referralSplit` and must not exceed `maxReferrers`, and every entry must be nonzero.
  - **Impact**: Decides who accrues referral balances for this purchase.
- `_referralSplit`
  - **Control**: Provided by the caller.
  - **Constraints**: `PRECISE_UNIT`-scaled weights — each must be nonzero, and the total must equal `PRECISE_UNIT`.
  - **Impact**: Defines how the referral fee is divided among `_referrers`.
- `_source`
  - **Control**: Provided by the caller.
  - **Constraints**: N/A.

- **Impact**: Emitted on `TicketPurchased` to tag the order source.

Branches and code coverage

Intended branches

- Validation succeeds, USDC transfers, NFTs mint, and drawing counters update.
  - ☑ Test coverage
- No referral path (`_referrers` empty) — referral share rolls into LP earnings.
  - ☑ Test coverage
- The combo tracker catches duplicates and updates the prize pool as intended.
  - ☑ Test coverage

Negative behavior

- Reverts when sales are off.
  - ☑ Negative test
- Reverts on malformed ticket or referral data.
  - ☑ Negative test

Function call analysis

- `this.usdc.transferFrom(msg.sender, address(this), ticketsValue)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.jackpotNFT.mintTicket(_recipient, ticketId, currentDrawingId, packedTicket, _referralSchemeId)`
  - **What is controllable?** `_recipient`, `_referralSchemeId`, and encoded ticket data come from user inputs that passed validation.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A as there are no hooks on this NFT minting.

Function: `claimWinnings(uint256[] _userTicketIds)`

This function lets a winner cash out. For each ticket, it confirms ownership, ensures the drawing is finished, burns the NFT, looks up the tier payout, and wires the remaining USDC to the caller. Every ticket is handled independently, so one bad entry fails the entire claim.

Inputs

- `_userTicketIds`
  - **Control**: Provided by the caller.
  - **Constraints**: The array cannot be empty, and each ticket must exist, be owned by the caller, and reference a drawing strictly less than `currentDrawingId`.
  - **Impact**: Selects which tickets' NFT are burned and determines the final payouts.

Branches and code coverage

Intended branches

- Tickets are validated and burned, the referral share is distributed, and USDC is transferred.
  - ☑ Test coverage
- Multi-ticket claims settle each ticket in a loop and accumulate the payout.
  - ☑ Test coverage

Negative behavior

- Reverts when `_userTicketIds` is empty.
  - ☑ Negative test
- Reverts if a ticket is not owned by the caller or belongs to an unfinished drawing.
  - ☐ Negative test

Function call analysis

- `this.jackpotNFT.getTicketInfo(ticketId)`
  - **What is controllable?** `ticketId` is derived from user input `_userTicketIds`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Defines the ticket that will be refunded.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

- `IERC721(address(this.jackpotNFT)).ownerOf(ticketId)`
    - **What is controllable?** `ticketId` is derived from user input `_userTicketIds`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.jackpotNFT.burnTicket(ticketId)`
    - **What is controllable?** `ticketId` is derived from user input `_userTicketIds`.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.payoutCalculator.getTierPayout(drawingId, tierId)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.usdc.transfer(msg.sender, totalClaimAmount)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `lpDeposit(uint256 _amountToDeposit)`

This function lets an LP move fresh USDC into the system. It rejects the call if the drawing is locked or the protocol is in emergency mode; otherwise, it pulls funds and hands the amount to the LP manager for accounting.

Inputs

- `_amountToDeposit`
    - **Control**: Provided by the caller.
    - **Constraints**: N/A.
    - **Impact**: Specify the amount deposited in the pool.

Branches and code coverage

Intended branches

- Drawing is open, funds are transferred, and the LP manager records the deposit.
  - ☑ Test coverage

Negative behavior

- Reverts when the drawing is locked or emergency mode is enabled.
  - ☑ Negative test

Function call analysis

- `this.usdc.transferFrom(msg.sender, address(this), _amountToDeposit)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.jackpotLPManager.processDeposit(currentDrawingId, msg.sender, _amountToDeposit)`
  - **What is controllable?** Inputs come from function arguments (user-controlled) and contract state.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `initiateWithdraw(uint256 _amountToWithdrawInShares)`

This function moves part of an LP's consolidated balance into the pending-withdrawal bucket so it can be cashed out once the drawing settles (final withdraw happens from $n + 1$ drawing). It is only allowed when the drawing is unlocked and the protocol is not in emergency mode — the LP manager enforces share limits.

Inputs

- `_amountToWithdrawInShares`
  - **Control**: Provided by the caller.

- **Constraints**: Must be greater than zero — LP manager verifies the caller shares.
- **Impact**: Determines how many shares are reclassified as pending withdrawal.

Branches and code coverage

Intended branches

- Drawing is not locked, emergency mode is off, and shares are queued for withdrawal.
  - ☑ Test coverage

Negative behavior

- Reverts when the drawing is locked or emergency mode is active.
  - ☑ Negative test
- Reverts on zero amount.
  - ☐ Negative test
- Reverts if the LP manager rejects the request.
  - ☑ Negative test

Function call analysis

- `this.jackpotLPManager.processInitiateWithdraw(currentDrawingId, msg.sender, _amountToWithdrawInShares)`
  - **What is controllable?** `currentDrawingId` is state-determined, and `_amountToWithdrawInShares` is user-controlled.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `emergencyWithdrawLP()`

When the system is stuck, LPs can forcefully unwind every position they have. This function lets the LP manager compute the amount and wire the resulting USDC straight to the caller.

Branches and code coverage

Intended branches

- Emergency mode is on, and the LP manager releases funds.

  ☑ Test coverage

Negative behavior

- Reverts if emergency mode is off.

  ☑ Negative test

Function call analysis

- `this.jackpotLPManager.emergencyWithdrawLP(currentDrawingId, msg.sender)`
  - **What is controllable?** `DrawingId` is state-determined.
  - **If the return value is controllable, how is it used and how can it go wrong?** Used to transfer this specific amount to the user.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.usdc.transfer(msg.sender, withdrawableAmount)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `finalizeWithdraw()`

This function cashes out the caller's matured withdrawals. It leans on the LP manager to total up anything pending or claimable then transfers that USDC to the LP. Emergency mode blocks the call.

Branches and code coverage

Intended branches

- The LP manager reports an amount (possibly zero) and transfers it out.

  ☑ Test coverage

Negative behavior

- Reverts under emergency mode.

  ☑ Negative test

Function call analysis

- `this.jackpotLPManager.processFinalizeWithdraw(currentDrawingId, msg.sender)`
    - **What is controllable?** `currentDrawingId` is state-determined.
    - **If the return value is controllable, how is it used and how can it go wrong?** Used to transfer this specific amount to the user.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.usdc.transfer(msg.sender, withdrawableAmount)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `emergencyRefundTickets(uint256[] _userTicketIds)`

This function provides refunds for tickets in the active drawing when emergency mode is engaged. Each ticket is burned, and the caller receives either the full ticket price or the price minus any referral amount already paid out.

Inputs

- `_userTicketIds`
    - **Control**: Provided by the caller.
    - **Constraints**: Must not be empty — every ticket needs to be owned by the caller and come from `currentDrawingId`.
    - **Impact**: Determines which tickets are burned and how much USDC is refunded.

Branches and code coverage

Intended branches

- Emergency mode is on, tickets are validated, and a refund is transferred.
    - ☑ Test coverage

Negative behavior

- Reverts on empty input, mismatched drawing.

- ☑ Negative test
- Tickets' `drawingId` mismatches the actual `drawingId`.
  - ☐ Negative test

Function call analysis

- `this.jackpotNFT.getTicketInfo(ticketId)`
  - **What is controllable?** `ticketId` is derived from user input `_userTicketIds`.
  - **If the return value is controllable, how is it used and how can it go wrong?** Defines the ticket that will be refunded.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `IERC721(address(this.jackpotNFT)).ownerOf(ticketId)`
  - **What is controllable?** `ticketId` is derived from user input `_userTicketIds`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.jackpotNFT.burnTicket(ticketId)`
  - **What is controllable?** `ticketId` is derived from user input `_userTicketIds`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.usdc.transfer(msg.sender, totalRefundAmount)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `claimReferralFees()`

This function pays out any referral balance accrued by the caller.

Branches and code coverage

Intended branches

- Positive balance, entry is cleared, and USDC is sent.
  - ☑ Test coverage

Negative behavior

- Reverts when the caller has no fees to claim.
  - ☑ Negative test

Function call analysis

- `this.usdc.transfer(msg.sender, transferAmount)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `runJackpot()`

This is a permissionless entry point that locks the drawing and pays the entropy provider once the scheduled drawing time has elapsed. It computes the required gas budget, refunds any overpayment, and registers callback so settlement can continue asynchronously (via `scaledEntropyCallback`).

Branches and code coverage

Intended branches

- Fees are refunded to the caller and paid for Pyth network.
  - ☑ Test coverage
- A request is crafted and sent to Pyth network via EntropyProvider.
  - ☑ Test coverage

Negative behavior

- Reverts if the drawing is already locked or if it is too early.
  - ☑ Negative test
- Reverts when the caller underpays the entropy fee (refund transfer can also revert if the caller rejects ETH).
  - ☑ Negative test

Function call analysis

- `this.entropy.getFee(entropyGasLimit)`
  - **What is controllable?** N/A, `entropyGasLimit` is crafted from state var defined by the admin.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `msg.sender.call{value: msg.value - fee}("")`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The bool `success` is checked and reverts in consequence.
- `this.entropy.requestAndCallbackScaledRandomness{value: fee}(entropyGasLimit, setRequests, address(this), this.scaledEntropyCallback.selector, bytes(""))`
  - **What is controllable?** All inputs are precomputed by the admin using the contract settings for the actual drawing.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** The revert propagates when `requestV2` on the Pyth network fails.

Function: `scaledEntropyCallback(bytes32 _requestId, uint8[][] _randomNumbers, bytes _callbackData)`

This is a settlement callback from ScaledEntropyProvider, which derives the Pyth randomness into five normal balls and one powerball. Once randomness arrives, the function checks that the drawing remains locked, computes winners and payouts, moves the protocol fee, updates the LP manager, and rolls into the next drawing.

Inputs

- `_requestId`
  - **Control**: Supplied by the entropy provider.
  - **Constraints**: Not used by the contract.
  - **Impact**: N/A.
- `_randomNumbers`

- **Control**: Fully controlled by the entropy provider.
- **Constraints**: Must be a two-row matrix (row 0: five normal balls, row 1: one powerball).
- **Impact**: Determines the winning ticket and derived payouts.

- `_callbackData`
  - **Control**: Supplied by the entropy provider.
  - **Constraints**: Unused — any bytes are accepted.
  - **Impact**: N/A.

Branches and code coverage

Intended branches

- Drawing is locked, and randomness is validated.
  - ☑ Test coverage
- Winners and payout totals are computed.
  - ☑ Test coverage
- The protocol fee is transferred.
  - ☑ Test coverage
- The LP manager settles, and the next drawing is initialized.
  - ☑ Test coverage

Negative behavior

- Reverts when the drawing is unlocked (`JackpotNotLocked`).
  - ☑ Negative test

Function call analysis

- `this.jackpotLPManager.processDrawingSettlement(currentDrawingId, currentDrawingState.lpEarnings, drawingUserWinnings, protocolFeeAmount)`
  - **What is controllable?** Only this contract can invoke it; parameters come from the current drawing state and freshly computed totals.
  - **If the return value is controllable, how is it used and how can it go wrong?** Provides the updated LP total and accumulator; bad data would skew future prize pools.
  - **What happens if it reverts, reenters or does other unusual control flow?** A revert leaves `jackpotLock` asserted so the drawing stays sealed; `nonReentrant` on the callback blocks reentry into settlement logic.

5.2. Module: ScaledEntropyProvider.sol

Function: `requestAndCallbackScaledRandomness(uint32 _gasLimit, SetRequest[] _requests, address _callback, bytes4 _selector, bytes _context)`

This is an entry point to request randomness from Pyth, and it registers the settlement callback, expected to be called by the jackpot when `runJackpot` is executed. It validates the callback metadata, ensures the caller paid at least the quoted fee, pushes the request to Pyth via `requestV2`, and caches the request parameters for the async callback.

Inputs

- `_gasLimit`
  - **Control**: Provided by the caller.
  - **Constraints**: Must be high enough to cover the callback execution; values feed directly into the Pyth fee quote.
  - **Impact**: Defines the gas for the Pyth request.
- `_requests`
  - **Control**: Provided by the caller.
  - **Constraints**: `_validateRequests` enforces a nonempty list, `minRange <= maxRange`, and `samples > 0`.
  - **Impact**: Describes each set of scaled draws that will be generated when entropy is delivered.
- `_callback`
  - **Control**: Provided by the caller.
  - **Constraints**: Must be a nonzero address.
  - **Impact**: The contract that receives the randomness from Pyth.
- `_selector`
  - **Control**: Provided by the caller.
  - **Constraints**: Must be nonzero.
  - **Impact**: Function selector invoked on `_callback` with the generated randomness.
- `_context`
  - **Control**: Provided by the caller.
  - **Constraints**: N/A.
  - **Impact**: Arbitrary metadata stored with the request.

Branches and code coverage

Intended branches

- The fee check passes (`msg.value >= getFee(_gasLimit)`).
  - ☑ Test coverage
- An entropy request is submitted to Pyth with the provided gas limit.
  - ☑ Test coverage
- The pending request metadata is stored under the returned sequence.
  - ☑ Test coverage

Negative behavior

- `_callback == address(0)` reverts.
  - ☑ Negative test
- `msg.value < getFee(_gasLimit)` reverts.
  - ☑ Negative test
- Invalid selector or malformed set requests revert.
  - ☑ Negative test

Function call analysis

- `this.getFee(_gasLimit)` → `this.entropy.getFeeV2(this.entropyProvider, _gasLimit)`
  - **What is controllable?** `_gasLimit` comes from the caller, while `entropyProvider` is owner-managed.
  - **If the return value is controllable, how is it used and how can it go wrong?** The fee quote from Pyth is expected to reflect to a precise fee to avoid a later revert.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.entropy.requestV2{value: msg.value}(this.entropyProvider, _gasLimit)`
  - **What is controllable?** The caller controls `_gasLimit`, and `entropyProvider` is configured by the owner.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the sequence ID that keys the pending request — incorrect responses would desync pending storage.
  - **What happens if it reverts, reenters or does other unusual control flow?** The Pyth contract is trusted not to reenter.

Function: `entropyCallback(uint64 sequence, address, bytes32 randomNumber)`

This is an internal hook invoked by the Pyth entropy contract once randomness is ready. It loads the cached request and expands the raw entropy into the requested sets of numbers.

Inputs

- `sequence`
  - **Control**: Supplied by Pyth entropy.
  - **Constraints**: Must correspond to an entry in `pending` — unknown IDs revert.
  - **Impact**: Keys the stored request metadata and ensures the right consumer is notified.
- `provider`
  - **Control**: Ignored.
  - **Constraints**: N/A.
  - **Impact**: N/A.
- `randomNumber`
  - **Control**: Provided by Pyth entropy.
  - **Constraints**: Assumed uniform 32-byte entropy value.
  - **Impact**: Seed for deriving every random sample.

Branches and code coverage

Intended branches

- The pending request is located via `sequence`.
  - ☑ Test coverage
- Scaled random numbers are derived from the raw entropy.
  - ☑ Test coverage
- The consumer callback executes successfully.
  - ☑ Test coverage

Negative behavior

- Missing or reused `sequence` reverts.
  - ☐ Negative test

Function call analysis

- `req.callback.call(abi.encodeWithSelector(req.selector, sequence, scaledRandomNumbers, req.context))`
    - **What is controllable?** Destination, selector, and context were initially provided by the requesting contract.
    - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
    - **What happens if it reverts, reenters or does other unusual control flow?** The Pyth contract is trusted not to reenter.

5.3. Module: JackpotLPManager.sol

Function: `processDeposit(uint256 _drawingId, address _lpAddress, uint256 _amount)`

This is a deposit entry point called from `Jackpot::lpDeposit` when new LP funds arrive for the active and actual drawing. It caps liquidity by checking `lpPoolCap`, rolls any prior drawing pending deposit into shares, and then records the actual LP deposit for the next settlement.

Inputs

- `_drawingId`
    - **Control**: Determined by Jackpot with the active drawing.
    - **Constraints**: N/A.
    - **Impact**: Determines which accumulator snapshot will later price the deposit when consolidated.
- `_lpAddress`
    - **Control**: Jackpot routes the deposit on behalf of the LP (`msg.sender` from jackpot).
    - **Constraints**: N/A.
    - **Impact**: Keys into `lpInfo` for deposit consolidation and share tracking.
- `_amount`
    - **Control**: Jackpot forwards the amount transferred in USDC.
    - **Constraints**: Check if `lpPoolCap` is not exceeded.
    - **Impact**: Increments both the LP's pending deposit and the drawing's aggregate pending deposits.

Branches and code coverage

Intended branches

- Amount fits under cap, and `_lpAddress lpInfo` is incremented as expected.
  - ☑ Test coverage
- Prior pending deposit (`_lp.lastDeposit.drawingId < _drawingId`) is consolidated into shares.
  - ☑ Test coverage

Negative behavior

- Non-Jackpot callers are rejected by `onlyJackpot`.
  - ☑ Test coverage
- Exceeding the pool cap reverts with the cap error.
  - ☑ Test coverage

Function: `processInitiateWithdraw(uint256 _drawingId, address _lpAddress, uint256 _amountToWithdrawInShares)`

This function moves part of an LP's consolidated share balance into the pending-withdraw queue for the current drawing. It settles any earlier deposits into shares and converts older pending withdrawals into claimable USDC, so this request stays focused on the current drawing.

Inputs

- `_drawingId`
  - **Control**: Jackpot determines which drawing is processed for the withdrawal.
  - **Constraints**: N/A.
  - **Impact**: Used to determine the accumulator snapshot used during consolidation.
- `_lpAddress`
  - **Control**: Jackpot routes the withdraw on behalf of the LP (`msg.sender` from jackpot).
  - **Constraints**: N/A.
  - **Impact**: Determines which LP balance is modified.
- `_amountToWithdrawInShares`
  - **Control**: By the user from `Jackpot::initiateWithdraw`.
  - **Constraints**: Must not exceed `lp.consolidatedShares` (total LP shares).
  - **Impact**: Deducts from consolidated shares and increments drawing pending

withdrawals.

Branches and code coverage

Intended branches

- Sufficient shares, consolidation of deposits/withdrawals occurs, and state mirrors request.

  ☑ Test coverage
- Prior `lastDeposit` migrates into shares before withdrawal — add regression verifying share math.

  ☑ Test coverage
- `_consolidateWithdrawals` converts older requests into claimable USDC.

  ☑ Test coverage
- Multiple low-amount withdraw initiations accumulate correctly into the final withdrawal.

  ☑ Test coverage

Negative behavior

- Non-Jackpot callers are rejected by `onlyJackpot`.

  ☑ Test coverage
- Requests above the available share balance revert.

  ☑ Test coverage

Function: `processFinalizeWithdraw(uint256 _drawingId, address _lpAddress)`

This function is called from `Jackpot::finalizeWithdraw`. It acts as the follow-up step in the withdrawal flow after `processInitiateWithdraw` runs during a prior drawing. It finishes an LP withdrawal by sweeping any leftover shares into the claimable pile and clearing the state in consequences.

Inputs

- `_drawingId`
  - **Control**: Jackpot determines which drawing is processed for the withdrawal.
  - **Constraints**: N/A.
  - **Impact**: Used to determine the accumulator snapshot used during consolidation.

- `_lpAddress`
    - **Control**: Jackpot routes the withdraw on behalf of the LP (`msg.sender` from jackpot).
    - **Constraints**: N/A.
    - **Impact**: Determines which LP balance is modified.

Branches and code coverage

Intended branches

- Consolidation happens, the `LpWithdrawFinalized` event fires, and the contract returns the amount expected.
    - ☑ Test coverage
- The second call after payout reverts.
    - ☐ Missing test

Negative behavior

- Non-Jackpot callers are rejected by `onlyJackpot`.
    - ☑ Test coverage
- Zero claimable balance reverts.
    - ☑ Test coverage

Function: `emergencyWithdrawLP(uint256 _drawingId, address _user)`

In emergency scenario, the Jackpot contract flips this emergency switch, allowing to cash out every piece of an LP's position: current deposits, consolidated shares, pending withdrawals, and claimable balance. The contract global state is updated in consequences.

Inputs

- `_drawingId`
    - **Control**: Jackpot determines which drawing is processed for the emergency withdrawal.
    - **Constraints**: N/A.
    - **Impact**: Determines the `drawingId` when the Jackpot was flipped in emergency mode.
- `_user`
    - **Control**: Jackpot passes the LP getting rescued (`msg.sender` from Jackpot).

- **Constraints**: N/A.
- **Impact**: Read, cash out, and clear this LP struct.

Branches and code coverage

Intended branches

- Only refund pending deposits and adjust drawing totals correctly.
  - ☑ Test coverage
- Share conversion, pending-withdraw math, and state cleanup all behave as expected.
  - ☑ Test coverage
- Handles cases where `pendingWithdrawal.amountInShares > 0`.
  - ☑ Test coverage
- Empty/zeroed positions do not break accounting or cause underflow.
  - ☑ Test coverage

Negative behavior

- Non-Jackpot callers are rejected by `onlyJackpot`.
  - ☑ Test coverage
- Check for underflow when totals are tiny.
  - ☐ Missing test

Function: `processDrawingSettlement(uint256 _drawingId, uint256 _lpEarnings, uint256 _userWinnings, uint256 _protocolFeeAmount)`

Once a drawing wraps, this function begins with the previous pool balance, adds LP earnings, subtracts player prizes and protocol fees, factors in pending deposits and withdrawals, and returns both the updated LP total and the next accumulator snapshot.

Inputs

- `_drawingId`
  - **Control**: Jackpot determines which drawing is processed for the drawing settlement.
  - **Constraints**: N/A.
  - **Impact**: Determines the `drawingId` when the Jackpot settles and updates to the accumulator associated.
- `_lpEarnings`

- **Control**: Jackpot forwards LP revenue from the round.
- **Constraints**: N/A.
- **Impact**: Pulled in the pool when computing `postDrawLpValue`.

- `_userWinnings`
  - **Control**: Jackpot reports total prizes paid to players.
  - **Constraints**: N/A.
  - **Impact**: Pulled out of the pool when computing `postDrawLpValue`.
- `_protocolFeeAmount`
  - **Control**: Jackpot submits the protocol's cut.
  - **Constraints**: N/A.
  - **Impact**: Pulled out of the pool when computing `postDrawLpValue`.

Branches and code coverage

Intended branches

- Drawing `0` skips the accumulator update yet still returns the correct `newLPValue`.
  - ☑ Test coverage
- Standard rounds recompute the accumulator as expected.
  - ☑ Test coverage
- Cover situations where `lpPoolTotal` could be zero.
  - ☐ Missing test

Negative behavior

- Non-Jackpot callers are rejected by `onlyJackpot`.
  - ☑ Test coverage

5.4. Module: GuaranteedMinimumPayoutCalculator.sol

Function: `calculateAndStoreDrawingUserWinnings`

This function, while being external, is only called by Jackpot after it has received randomness in its ScaledEntropyCallback method. It is responsible for calculating the total winning combos per tier, iterating tiers, and verifiying if they need to be accounted for in the minimum guaranteed payout. Then, it distributes the remaining prize pool across the premium weights.

Inputs

- `_drawingId`

- **Control**: None.
- **Constraints**: N/A.
- **Impact**: The drawing ID, the cycle.

- `_prizePool`
  - **Control**: Partial.
  - **Constraints**: None.
  - **Impact**: The prize pool, which can be influenced by ticket purchases.
- `normalMax`
  - **Control**: None.
  - **Constraints**: 1 to 69.
  - **Impact**: Normal-ball maximum (69).
- `_powerballMax`
  - **Control**: None.
  - **Constraints**: 1 to 26.
  - **Impact**: Powerball maximum (26).
- `_uniqueResult`
  - **Control**: None.
  - **Constraints**: N/A.
  - **Impact**: Used to determine if a combination is a winner.
- `_dupResult`
  - **Control**: None.
  - **Constraints**: N/A.
  - **Impact**: Used to determine distribution of funds.

Branches and code coverage

Intended branches

- Calculate the correct payout for a minimum-payout tier.
  - ☑ Test coverage
- Calculate the correct payout for a no-minimum-payout tier.
  - ☑ Test coverage
- Calculate the correct payout for a minimum-payout tier with premium funds remaining.
  - ☑ Test coverage
- Calculate the correct payout for a no-minimum-payout tier with premium funds remaining.
  - ☑ Test coverage

- All the tests before but with duplicate winners.

  ☑ Test coverage

Function: `_calculateTierTotalWinningCombos`

This function is responsible for calculating the total number of possible winning combos for a certain tier (e.g., tier 8 would calculate how many winning four normal-ball combinations without powerballs there are). This is done to calculate what the minimum payout should be.

Inputs

- `_matches`
  - **Control**: None.
  - **Constraints**: N/A.
  - **Impact**: The number of normal-ball matches.
- `normalMax`
  - **Control**: None.
  - **Constraints**: 1 to 69.
  - **Impact**: Normal-ball maximum (69).
- `_powerballMax`
  - **Control**: None.
  - **Constraints**: 1 to 26.
  - **Impact**: Powerball maximum (26).
- `_powerBallMatch`
  - **Control**: None (derived internally based on the tier being evaluated).
  - **Constraints**: N/A.
  - **Impact**: Whether the powerball is matching or not, it adds more combinations.

Branches and code coverage

Intended branches

- Calculate the correct payout for a minimum-payout tier.

  ☑ Test coverage
- Calculate the correct payout for a no-minimum-payout tier.

  ☑ Test coverage
- Calculate the correct payout for a minimum-payout tier with premium funds remaining.

☑ Test coverage

- Calculate the correct payout for a no-minimum-payout tier with premium funds remaining.

  ☑ Test coverage

- All the tests before but with duplicate winners.

  ☑ Test coverage

5.5. Module: JackpotBridgeManager.sol

Function: `buyTickets`

This is the entry point for buying tickets from the bridge relay.

Inputs

- `_tickets`
  - **Control**: None.
  - **Constraints**: N/A.
  - **Impact**: Tickets.
- `_recipient`
  - **Control**: None.
  - **Constraints**: Not `address(0)`.
  - **Impact**: The recipient.
- `_referrers`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The referrers.
- `_referralSplitBps`
  - **Control**: None.
  - **Constraints**: Must add up to 1e18.
  - **Impact**: The distribution of the referral fee across referrers.
- `source`
  - **Control**: None.
  - **Constraints**: N/A.
  - **Impact**: Tracking identifier.

Branches and code coverage

Intended branches

- Update the user's tickets and ticket owner.
  - ☑ Test coverage
- Set the BridgeManager as the ticket owner on the Jackpot contract.
  - ☑ Test coverage
- Correctly transfer the USDC from the buyer to the Jackpot contract via the manager.
  - ☑ Test coverage
- All the tests before but with duplicate winners.
  - ☑ Test coverage

Negative behavior

- Emergency mode test.
  - ☐ Negative test
- Reentrancy check test.
  - ☐ Negative test

Function call analysis

- `jackpot.ticketPrice()`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** The price of the ticket.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `jackpot.currentDrawingId()`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** The `currentDrawingId`.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `usdc.transferFrom(msg.sender, address(this), ticketCost)`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** No return.
  - **What happens if it reverts, reenters or does other unusual control flow?**

N/A.

- `usdc.approve(address(jackpot), ticketCost)`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** No return.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `jackpot.buyTickets(_tickets, address(this), _referrers, _referralSplitBps, _source)`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** No return.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.

Function: `claimWinnings`

This is the entry point for claiming winnings from the bridge relay.

Inputs

- `_userTicketIds`
  - **Control**: None.
  - **Constraints**: N/A.
  - **Impact**: The tickets to claim.
- `_bridgeDetails`
  - **Control**: None.
  - **Constraints**: None.
  - **Impact**: The bridge to call with calldata.
- `_signature`
  - **Control**: Full.
  - **Constraints**: None.
  - **Impact**: The signature to enforce that the owner of these tickets is attempting to claim winnings.

Branches and code coverage

Intended branches

- Transfers tokens.
  - ☑ Test coverage
- Owner/signature check.
  - ☑ Test coverage
- Calls the bridge with the supplied bridge details.
  - ☑ Test coverage

Negative behavior

- Emergency mode test.
  - ☐ Negative test

Function call analysis

- `usdc.balanceOf(address(this))`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** Not controllable — used for USDC balance check.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts might cause issues with bridge integrations; see Finding 3.9. ↗.
- `jackpot.claimWinnings(_userTicketIds)`
  - **What is controllable?** `_userTicketIds`.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `usdc.balanceOf(address(this))`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** Not controllable — used for USDC balance check.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts might cause issues with bridge integrations; see Finding 3.9. ↗.
- `usdc.approve(_bridgeDetails.approveTo, _claimedAmount)`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** N/A.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts potentially with USDT; see Finding 3.12. ↗.
- `usdc.approve(_bridgeDetails.approveTo, _claimedAmount)`

- **What is controllable?** Nothing.
- **If the return value is controllable, how is it used and how can it go wrong?** N/A.
- **What happens if it reverts, reenters or does other unusual control flow?** Reverts potentially with USDT; see Finding 3.12. ↗.

- `usdc.balanceOf(address(this))`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** Not controllable — used for USDC balance check.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts might cause issues with bridge integrations; see Finding 3.9. ↗.
- `_bridgeDetails.to.call(_bridgeDetails.data)`.
  - **What is controllable?** `_bridgeDetails` and `_bridgeDetails.data`.
  - **If the return value is controllable, how is it used and how can it go wrong?** the boolean call success var, which revert if the call failed.
  - **What happens if it reverts, reenters or does other unusual control flow?** There is a reentrancy risk here and also an arbitrary execution risk; see Finding 3.3. ↗.
- `usdc.balanceOf(address(this))`
  - **What is controllable?** Nothing.
  - **If the return value is controllable, how is it used and how can it go wrong?** Not controllable — used for USDC balance check.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts might cause issues with bridge integrations; see Finding 3.9. ↗.

6. Assessment Results

During our assessment on the scoped Megapot v2 contracts, we discovered 12 findings. One critical issue was found. Three were of high impact, one was of medium impact, and seven were of low impact.

We recommend focusing on a thorough end-to-end test suite and full integration coverage prior to going to production.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.