Zellic

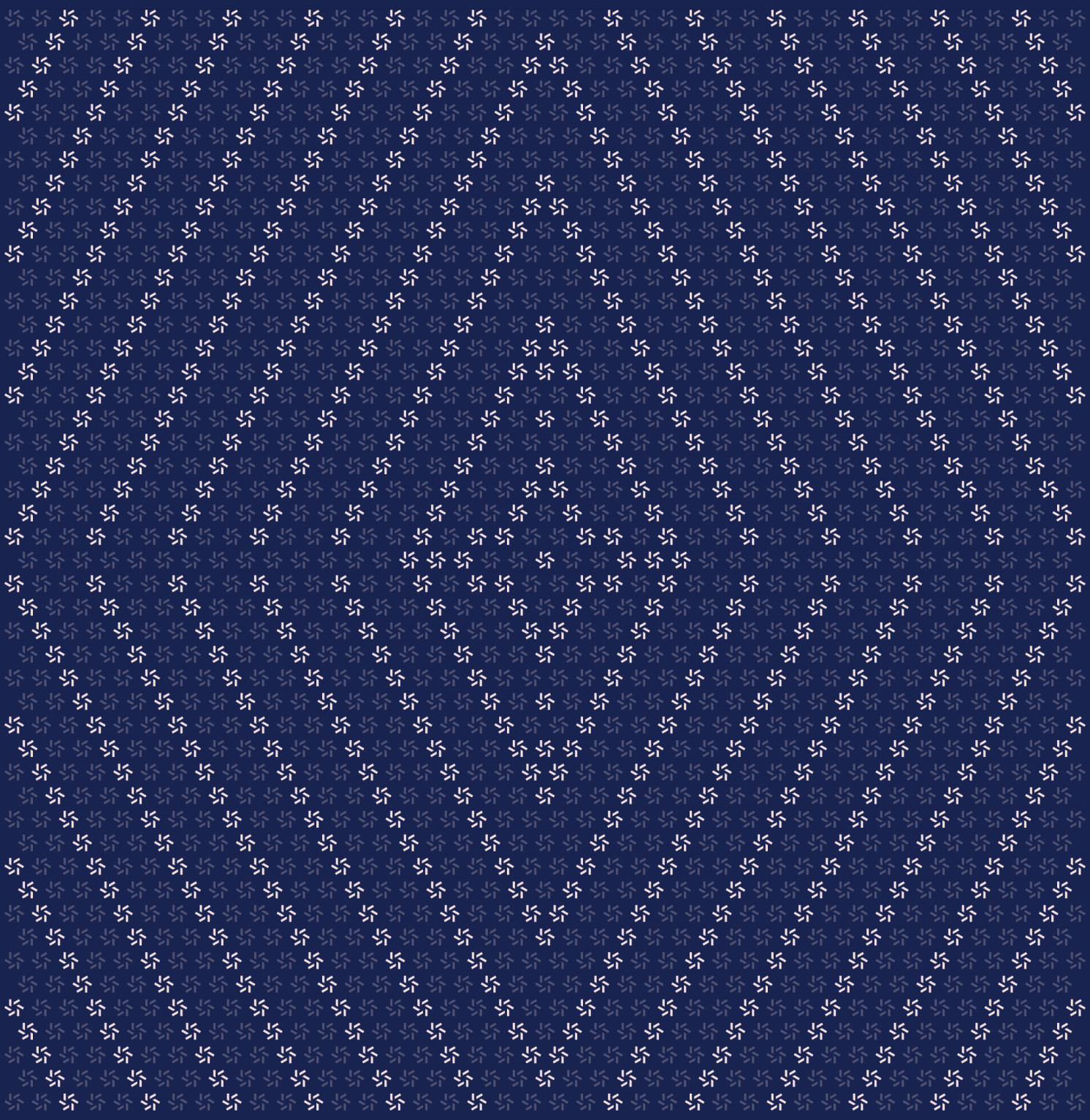
Prepared for
Patrick Lung
Brian Weickmann
Coordination Labs

Prepared by
Katerina Belotskaia
Quentin Lemauf
Zellic

December 9, 2025

Megapot (V2)

Smart Contract Security Assessment



Contents

About Zellic **4**

1. Overview **4**

1.1. Executive Summary 5

1.2. Goals of the Assessment 5

1.3. Non-goals and Limitations 5

1.4. Results 5

2. Introduction **6**

2.1. About Megapot (V2) 7

2.2. Methodology 7

2.3. Scope 9

2.4. Project Overview 9

2.5. Project Timeline 10

3. Detailed Findings **10**

3.1. Single-ticket–subscription spamming increases keeper load 11

3.2. Inconsistent normal ball range for new subscriptions 12

3.3. Subscriptions execute once per drawing instead of daily 13

3.4. Nonstandard ERC-20 token usage 15

3.5. Insufficient fund handling causes batch-execution revert via underflow 16

4. Discussion **16**

4.1. Code-quality improvements 17

4.2. Lack of validation in the `pickAuto` function 18

5. Threat Model 18

5.1. Module: BatchPurchaseFacilitator.sol 19

5.2. Module: JackpotAutoSubscription.sol 26

5.3. Module: JackpotRandomTicketBuyer.sol 33

5.4. Module: TicketPicker.sol 35

6. Assessment Results 39

6.1. Disclaimer 40

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 Labs from November 28th to December 2nd, 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 for funds to be stolen by a third party or by keepers?
- Are there any revert cases when buying tickets that are not handled gracefully (which is most likely due to parameter changes at the protocol level)?
- Is there any way that state can be accidentally corrupted due to faulty business logic?

1.3. Non-goals and Limitations

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

- Jackpot contract
- 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.

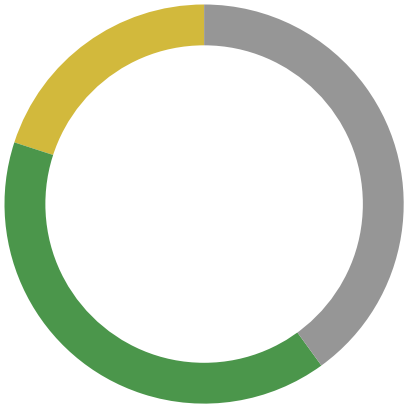
1.4. Results

During our assessment on the scoped Megapot (V2) contracts, we discovered five findings. No critical issues were found. One finding was of medium impact, two were of low impact, and the remaining findings were informational in nature.

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

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	2
Informational	2



2. Introduction

2.1. About Megapot (V2)

Coordination Labs contributed the following description of Megapot (V2):

> Megapot is a 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	`4c142af48f5337fcd57f2a1dc720f11135566f04`

Programs	`BatchPurchaseFacilitator.sol`
	`JackpotAutoSubscription.sol`
	`JackpotRandomTicketBuyer.sol`
	`lib/TicketPicker.sol`
	`lib/KeeperManager.sol`
	`lib/BuyTicketsHelpers.sol`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of one person-week. The assessment was conducted by two consultants over the course of three calendar days.

Contact Information

The following project manager was associated with the engagement:

Pedro Moura
Engagement Manager
pedro@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Katerina Belotskaia
Engineer
kate@zellic.io ↗

Quentin Lemauf
Engineer
quentin@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

November 28, 2025	Kick-off call
November 28, 2025	Start of primary review period
December 2, 2025	End of primary review period

3. Detailed Findings

3.1. Single-ticket–subscription spamming increases keeper load

Target	JackpotAutoSubscription		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The `createSubscription` function allows users to create a subscription for automatic ticket purchases for every drawing. This function validates that the specified number of tickets does not exceed the `maxDailyTickets` limit. However, there is no minimum required number of tickets per subscription.

As a result, an attacker can create a very large number of single-ticket subscriptions across many addresses.

Impact

This does not decrease the attacker's odds compared to batching tickets, but spamming many single-ticket subscriptions forces the keeper to loop through more addresses every drawing. Keepers will batch work via `executeSubscriptions(address[] calldata)` to cut transaction count, yet gas still scales roughly linearly with total subscriptions and may require multiple batches, increasing keeper cost and operational load.

Recommendations

Enforce a minimum ticket amount per subscription to prevent or discourage single-ticket–subscription spamming.

Remediation

This issue has been acknowledged by Coordination Labs. The team wants to preserve the ability for users to purchase one ticket per day over a defined period. If this is abused, they will consider making changes.

3.2. Inconsistent normal ball range for new subscriptions

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

Description

Subscription validation uses the current global `normalBallMax` instead of the drawing's snapshotted `ballMax`. When `normalBallMax` is lowered after a drawing starts, static picks that exceed the new max are silently converted into a dynamic ticket for the entire subscription. If `normalBallMax` is later raised, previously created static tickets remain valid, and in all cases the resulting ticket keeps the same odds as any other valid ticket.

Impact

- When the owner reduces `normalBallMax` mid-drawing, users paying for static numbers can unknowingly receive a dynamic ticket instead. Their intended static combination is discarded if it exceeds the lowered max, which can create confusion (e.g., believing a static pick was played when it was not).
- There is no odds disadvantage for the converted ticket — the risk is loss of user intent/expectation and unclear UX around when static picks are downgraded to dynamic.

Recommendations

Document this edge case so users understand that static picks above the current `normalBallMax` will be converted to dynamic tickets if the limit is lowered mid-drawing.

Remediation

This issue has been acknowledged by Coordination Labs. The team intentionally validates against the global max to preserve user-selected numbers across the subscription lifetime, accepting the UX tradeoff that static picks above a lowered max are converted to dynamic (odds unchanged).

3.3. Subscriptions execute once per drawing instead of daily

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

Description

According to the documentation, the `createSubscription` function allows users to create a subscription for daily ticket purchases.

```
/**
* @notice Creates an auto-subscription for daily ticket purchases
* @dev Funds the subscription in USDC and records the number of dynamic (random)
*      and static (user-defined) tickets to buy per drawing for a number of
    days.
*      Static tickets are validated and stored once. The subscription becomes
*      immediately eligible for the current drawing by setting
    `lastExecutedDrawing`
*      to `currentDrawingId - 1`.
*/
function createSubscription(
    address _recipient,
    uint64 _totalDays,
    uint64 _dynamicTicketCount,
    IJackpot.Ticket[] calldata _userStaticTickets,
    address[] calldata _referrers,
    uint256[] calldata _referralSplit
)
    external
    nonReentrant
{
    // [...]
}
```

However, the `executeSubscriptions` function enforces the subscription execution once per drawing, not per day, since `_determineExecutionAction()` skips execution when `lastExecutedDrawing == currentDrawingId`. As a result, a subscription cannot run more than once before the drawing rolls over.

```
function _determineExecutionAction(
    address _recipient,
    uint256 _currentDrawingId,
    uint256 _ticketPrice,
    uint256 _maxReferrers
)
    internal
    view
    returns (ExecutionAction)
{
    // [...]
    if (subscription.lastExecutedDrawing == _currentDrawingId)
    return ExecutionAction.SKIP_ALREADY_EXECUTED;
    // [...]
}
```

This creates a mismatch between the documented/expected behavior (daily execution over a number of days) and the actual behavior (single execution per drawing, regardless of duration).

Impact

Users who expect their subscription to buy tickets daily over a specified number of days may actually receive only one batch of tickets per drawing, even if that drawing lasts several days.

Recommendations

Align the implementation with the documented behavior, or update the documentation to clearly reflect the current once-per-drawing behavior.

Remediation

This issue has been acknowledged by Coordination Labs, and a fix was implemented in commit 98986f95 ↗.

3.4. Nonstandard ERC-20 token usage

Target	Multiple contracts		
Category	Business Logic	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The BatchPurchaseFacilitator, JackpotAutoSubscription, and JackpotRandomTicketBuyer contracts call `approve` directly on the USDC token to allow the Jackpot contract to transfer the exact amount of USDC during ticket purchase.

```
usdc.approve(address(Jackpot), totalCost);
```

While this works for USDC, the `approve` call can revert for USDT (or any other nonstandard ERC-20 token). It requires allowance to be set to zero before setting a new value; otherwise, calling `approve` with a nonzero current allowance will revert.

Impact

Because the code assumes a fully ERC-20–compliant token usage, swapping usage of USDC to the USDT token will cause approvals to fail, breaking the ticket-purchase flow and any functionality that relies on updating allowances.

Recommendations

To support USDT and other nonstandard tokens, use OpenZeppelin's SafeERC20 library that handles nonstandard tokens.

Remediation

This issue has been acknowledged by Coordination Labs.

3.5. Insufficient fund handling causes batch-execution revert via underflow

Target	JackpotAutoSubscription		
Category	Business Logic	**Severity**	Informational
Likelihood	Low	**Impact**	Informational

Description

The `executeSubscriptions` function is supposed to execute a batch of subscriptions for the current drawing. The `_determineExecutionAction` function determines the execution action for each subscription, and if some subscription cannot be executed, it is skipped or closed. However, it does not properly handle cases where a subscriber's remaining USDC balance is positive but below the cost of a draw. As a result, underfunded subscriptions are tried to be executed and revert because of the underflow. Additionally, `_executeSubscription` performs `remainingUSDC -= totalCost` without checking `remainingUSDC >= totalCost` and reverts silently without any error message.

Impact

A single subscription with insufficient funds can block all remaining subscription executions in a batch.

Recommendations

Consider implementing the following changes:

- Add a check to `_determineExecutionAction` to close the subscription if the remaining USDC balance is less than the cost of a draw.
- Add a check to `_executeSubscription` to revert with a clear error message if the remaining USDC balance is less than the cost of a draw.

Remediation

This issue has been acknowledged by Coordination Labs.

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. Code-quality improvements

These following notes on code quality are not directly security related.

Remove unused `EnumerableSet` helpers

JackpotAutoSubscription imports `EnumerableSet` and applies the address-set mix-in even though the contract never declares a set — helpers—keeper management lives in KeeperManager, not in this contract:

```
contract JackpotAutoSubscription is Ownable2Step, ReentrancyGuardTransient,
    KeeperManager {
    using SafeERC20 for IERC20;
@>  using EnumerableSet for EnumerableSet.AddressSet;
    // ...
}
```

Dropping the unused `using EnumerableSet for EnumerableSet.AddressSet;` (and its import) keeps the contract leaner and avoids pulling in helpers that are not exercised in this file.

Simplify dynamic ticket copy loop

Using an index that spans the full static and dynamic length, `_createTicketArray` copies dynamic tickets into the concatenated array:

```
for (uint256 i = filteredStaticTickets.length; i < _userStaticTickets.length
    + _dynamicTicketCount; i++) {
    tickets[i] = dynamicTickets[i - filteredStaticTickets.length];
}
```

Because `dynamicTickets.length` already equals `_userStaticTickets.length + _dynamicTicketCount - filteredStaticTickets.length`, the clearer pattern below avoids recomputing the upper bound each iteration and makes the offset explicit:

```
for (uint256 i = 0; i < dynamicTickets.length; ++i) {
    tickets[i + filteredStaticTickets.length] = dynamicTickets[i];
```

```
}
```

The behavior is unchanged while readability and intent is improved slightly.

4.2. Lack of validation in the `pickAuto` function

The `pickAuto` function in the TicketPicker library does not perform direct validation ensuring that `_normalMax >= 5`. Although the documentation states that `_normalMax` must be at least 5, this requirement is not enforced inside the library itself.

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: BatchPurchaseFacilitator.sol

Function: `cancelBatchOrder()`

This function allows users to cancel an unexecuted order or remaining USDC tokens if the order has been partially executed. The full `batchOrders[msg.sender].remainingUSDC` amount will be transferred to the user.

Branches and code coverage

Intended branches

- The expected amount of USDC tokens have been transferred to the caller.
  - ☑ Test coverage
- `batchOrders[msg.sender].remainingUSDC` has been reset to zero after execution.
  - ☑ Test coverage
- The `BatchOrderCancelled` event has been emitted.
  - ☑ Test coverage

Negative behavior

- `batchOrders[msg.sender].remainingUSDC` is zero.
  - ☑ Negative test

Function call analysis

- `this._cancelBatchOrder(msg.sender, ExecutionAction.CANCEL_USER_REQUESTED) -> SafeERC20.safeTransfer(this.usdc, _recipient, refundAmount)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the current balance of the contract is less than the remaining USDC.

Function: `createBatchOrder(address _recipient, uint64 _dynamicTicketCount, IJackpot.Ticket[] _userStaticTickets, address[] _referrers, uint256[] _referralSplit)`

This function allows any caller to create orders for the ticket purchase on behalf of any user. The caller address should provide funds to cover the tickets' cost. This function checks if the Jackpot is not locked (Jackpot can be unlocked in the future, so locking is not permanent).

Also, `BuyTicketsHelpers.validateTickets` reverts if `_userStaticTickets.normals` contains repeated numbers. The recipient can cancel using the `cancelBatchOrder` function and get refunded.

Inputs

- `_recipient`
  - **Control**: Full control.
  - **Constraints**: Should not have any active orders.
  - **Impact**: The address of the receiver of the tickets.
- `_dynamicTicketCount`
  - **Control**: Full control.
  - **Constraints**: The sum of `_dynamicTicketCount` and `_userStaticTickets.length` should be more than `minimumTicketCount`.
  - **Impact**: Total number of dynamic (random) tickets to include in this batch order. The user does not specify their numbers; they are generated randomly at execution time.
- `_userStaticTickets`
  - **Control**: Full control.
  - **Constraints**: The sum of `_dynamicTicketCount` and `_userStaticTickets.length` should be more than `minimumTicketCount`. The duplicated numbers are not allowed within a single static ticket, and `bonusball` should not be more than `_drawingState.bonusballMax`.
  - **Impact**: Array of user-defined static tickets, where each ticket specifies its own fixed normal numbers and bonusball.
- `_referrers`
  - **Control**: Full control.
  - **Constraints**: Length of the `_referrers` should be equal to the `_referralSplit.length`.
  - **Impact**: Array of referrer addresses.
- `_referralSplit`
  - **Control**: Full control.
  - **Constraints**: Total sum should be equal to the `PRECISE_UNIT`.

- **Impact**: Weights corresponding to `_referrers` addresses.

Branches and code coverage

Intended branches

- The new order has been created successfully.
  - ☑ Test coverage
- The order has been executed and created successfully again for the same `_recipient`.
  - ☐ Test coverage
- The total cost has been transferred to the contract successfully.
  - ☑ Test coverage
- `BatchOrderCreated` has been emitted.
  - ☑ Test coverage

Negative behavior

- The static ticket contains duplicate numbers.
  - ☑ Negative test
- The total number of tickets are less than `minimumTicketCount`.
  - ☐ Negative test
- The caller does not have enough USDC tokens to cover the total cost.
  - ☐ Negative test
- `batchOrders[_recipient].remainingUSDC != 0`.
  - ☑ Negative test
- Jackpot is locked.
  - ☑ Negative test
- The `_referrers` number is greater than the maximum.
  - ☑ Negative test
- `_recipient` is zero address.
  - ☑ Negative test
- The static ticket has invalid number of balls.
  - ☑ Negative test
- The static ticket has a number greater than the maximum.
  - ☑ Negative test
- The static ticket has zero bonusballs.

☑ Negative test

- The number of bonusballs is greater than the bonusball maximum.

☑ Negative test

- The sum of `_referralSplit` is not equal to the `PRECISE_UNIT`.

☑ Negative test

- The length of the `_referralSplit != _referrers`.

☑ Negative test

- Jackpot is not initialized.

☑ Negative test

Function call analysis

- `this.jackpot.currentDrawingId()`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the ID of the current drawing.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the ID is zero.
- `this.jackpot.getDrawingState(currentDrawingId)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** Returns the state of the current drawing. The current Jackpot can be locked, and then the order cannot be created.
  - **What happens if it reverts, reenters or does other unusual control flow?** There are no problems here.
- `SafeERC20.safeTransferFrom(this.usdc, msg.sender, address(this), totalCost)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if allowance is not enough or the balance of the caller is less than `totalCost`.

Function: `executeBatchOrder(address _recipient, uint256 _maxTicketsPerBatch)`

If the `_currentDrawingId` is not equal to the user's `orderDrawingId`, the order will be canceled. If the Jackpot is locked, the order will be canceled. If number of referrers of the order exceeds the global `maxReferrers`, the order will be canceled. If the `remainingTickets` is less than or equal to `_maxTicketsPerBatch`, the order will be fully executed and removed; otherwise, it will be partially executed. In the case of execution, it generates the dynamic tickets within the current drawing range and initiates the tickets' purchase in the Jackpot contract.

In the case of execution, the `Jackpot.buyTickets` will be called. It reverts

- if the current Jackpot is locked (`jackpotLock` can be locked at any moment by the owner so the order becomes unexecutable);
- if the `prizePool` is zero;
- if `allowTicketPurchases` is false;
- if the provided `_tickets` length is zero;
- if the `_recipient` address is zero;
- if `_referrers.length != _referralSplit.length`;
- if `_referrers.length > maxReferrers` (the `maxReferrers` can be updated by the owner at any moment so the order becomes unexecutable even if it was valid during order creation); and
- if approve is not enough to transfer cost (but ticket costs in `buyTickets` should be the same since the price is not updatable).

If `Jackpot.buyTickets` reverts, the recipient can execute `cancelBatchOrder`, so funds are not stuck in the contract.

Inputs

- `_recipient`
  - **Control**: Full control.
  - **Constraints**: Should have an existing order.
  - **Impact**: The address of the user on behalf of which order will be executed and who will receive tickets.
- `_maxTicketsPerBatch`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: The number of tickets that will be bought for every order.

Branches and code coverage

Intended branches

- The order has been canceled successfully if `_currentDrawingId != orderDrawingId`.

  ☑ Test coverage
- The order has been canceled successfully if the Jackpot is locked.

  ☑ Test coverage
- The order has been canceled successfully if the number of referrers of the order exceeds the global `maxReferrers`.

  ☑ Test coverage
- If `remainingTickets <= _maxTicketsPerBatch`, the order has been executed fully and deleted.

  ☑ Test coverage
- Full execution of the order, `totalCost` has been transferred to the Jackpot, and the full order state has been reset.

  ☑ Test coverage
- Full execution of the order that had been executed partially before.

  ☑ Test coverage
- Partial execution, the order state has been updated correctly, and `totalCost` has been transferred to the Jackpot.

  ☑ Test coverage
- `BatchOrderExecuted` has been emitted.

  ☑ Test coverage
- The order has been executed partially, and all static tickets have been already bought, the second time only dynamic tickets should be executed.

  ☑ Test coverage
- The order has been executed partially, and part of dynamic tickets have been executed before.

  ☑ Test coverage

Negative behavior

- The recipient does not have order.

  ☑ Negative test
- The caller is not the keeper.

  ☑ Negative test

Function call analysis

- `this._determineExecutionAction(_recipient, this.jackpot.currentDrawingId(), _maxTicketsPerBatch) -> this.jackpot.getDrawingState(_currentDrawingId)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the state of the current drawing.
    - **What happens if it reverts, reenters or does other unusual control flow?** There are no problems here.
- `this._determineExecutionAction(_recipient, this.jackpot.currentDrawingId(), _maxTicketsPerBatch) -> this.jackpot.maxReferrers()`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Returns the current global `maxReferrers`. If the number of referrers exceed this `maxReferrers`, the order will be closed.
    - **What happens if it reverts, reenters or does other unusual control flow?** The order will be canceled if the `maxReferrers` is less than the number specified in the order.
- `this._executeBatchOrder(_recipient, _maxTicketsPerBatch) -> TicketPicker.pickAuto(++this.executionNonce, dynamicTicketCount, drawingState.ballMax, drawingState.bonusballMax)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** Generates the dynamic ticket numbers. Five numbers per ticket, which are limited by the `ballMax` for the regular balls and `bonusballMax` for the bonus ball.
    - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if `dynamicTicketCount` is zero.
- `this._executeBatchOrder(_recipient, _maxTicketsPerBatch) -> this.usdc.approve(address(this.jackpot), totalCost)`
    - **What is controllable?** N/A.
    - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value.
    - **What happens if it reverts, reenters or does other unusual control flow?** There are no problems here. This approve is necessary for the jackpot contract to transfer funds during ticket purchase.
- `this._executeBatchOrder(_recipient, _maxTicketsPerBatch) -> this.jackpot.buyTickets(tickets, _recipient, batchOrder.referrers, batchOrder.referralSplit, BatchPurchaseFacilitator.BATCH_PURCHASE_TELEMETRY)`

- **What is controllable?** `_recipient`.
- **If the return value is controllable, how is it used and how can it go wrong?** Returns the IDs of purchased tickets.
- **What happens if it reverts, reenters or does other unusual control flow?** Transfers total costs for the tickets and reverts if the current contract does not have enough USDC tokens.

- `this._cancelBatchOrder(_recipient, ExecutionAction.CANCEL_TOO_MANY_REFERRERS) -> SafeERC20.safeTransfer(this.usdc, _recipient, refundAmount)`

  - **What is controllable?** `_recipient`.
  - **If the return value is controllable, how is it used and how can it go wrong?** There is no return value here.
  - **What happens if it reverts, reenters or does other unusual control flow?** Reverts if the current balance of the contract is less than the `refundAmount`.

5.2. Module: JackpotAutoSubscription.sol

Function: `createSubscription(address _recipient, uint64 _totalDays, uint64 _dynamicTicketCount, IJackpot.Ticket[] _userStaticTickets, address[] _referrers, uint256[] _referralSplit)`

This is a user entry point to create a subscription with `x` static tickets and `y` dynamic tickets for the next `n` drawings. The total subscription value is paid in advance, and USDC is transferred to the contract at the current ticket price. If the ticket price changes, any remaining balance in the subscription will be refunded.

Inputs

- `_recipient`
  - **Control**: Full control.
  - **Constraints**: Cannot already have a subscription ongoing.
  - **Impact**: Determines the address linked to the new subscription.
- `_totalDays`
  - **Control**: Full control.
  - **Constraints**: Must be nonzero.
  - **Impact**: Determines the number of drawings this subscription will run.
- `_dynamicTicketCount`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Determines the number of random tickets in the subscription,

generated via pseudorandomness on chain.

- `_userStaticTickets`
  - **Control**: Full control.
  - **Constraints**: Normals and bonusball must be in range; invalid static tickets are replaced with dynamic tickets.
  - **Impact**: Purchases fixed tickets every drawing.
- `_referrers`
  - **Control**: Full control.
  - **Constraints**: Length is capped.
  - **Impact**: Addresses that receive referral fees.
- `_referralSplit`
  - **Control**: Full control.
  - **Constraints**: Total should be equal to `PRECISE_UNIT`.
  - **Impact**: Fee share allocated to each referrer.

Branches and code coverage

Intended branches

- Creates a subscription when inputs are valid (no existing subscription, nonzero `_totalDays`, ticket count within `maxDailyTickets`, referrals valid) and records balances/tickets.

  ☑ Test coverage

Negative behavior

- Zero recipient, zero duration, and zero or excessive ticket counts are rejected.

  ☑ Negative test
- There is no active subscription present.

  ☑ Negative test
- Tickets are well-formed and in range.

  ☑ Negative test
- Referral data is in range, and there is no mismatching in splits.

  ☑ Negative test

Function call analysis

- `this.jackpot.currentDrawingId()`

- **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.jackpot.getDrawingState(currentDrawingId)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** It is the current drawing state, populated when the previous one is closed, with global state and dynamic values (`bonusMax`).
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this._validateBuyTicketParams(drawingState, _recipient, _userStaticTickets, _referrers, _referralSplit) -> this.jackpot.normalBallMax()`
  - **What is controllable?** Returns an owner-defined value in the Jackpot 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?** N/A.
- `this._validateBuyTicketParams(drawingState, _recipient, _userStaticTickets, _referrers, _referralSplit) -> BuyTicketsHelpers.validateTickets(_userStaticTickets, ballMax, _drawingState.bonusballMax, _recipient)`
  - **What is controllable?** Only `_userStaticTickets` and `_recipient` are 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.
- `this._validateBuyTicketParams(drawingState, _recipient, _userStaticTickets, _referrers, _referralSplit) -> BuyTicketsHelpers.validateReferrers(this.jackpot, _referrers, _referralSplit)`
  - **What is controllable?** Both `_referrers` and `_referralSplit` are 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.
- `this.jackpot.ticketPrice()`

    - **What is controllable?** Returns owner-defined value in the Jackpot 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?** N/A.
- `SafeERC20.safeTransferFrom(this.usdc, msg.sender, address(this), totalCost)`
    - **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: `executeSubscriptions(address[] _subscriptions)`

This is a keeper entry point that processes a batch of autosubscriptions for the current drawing. For each recipient, it determines the `ExecutionAction` (execute, skip, cancel) based on drawing ID, ticket price, and referral cap and then either buys tickets, skips, or cancels with a refund. Static tickets that fall out of range for the current drawing are replaced 1:1 with dynamic tickets so the per-drawing ticket count is preserved.

Inputs

- `_subscriptions`
    - **Control**: Keeper control.
    - **Constraints**: N/A.
    - **Impact**: Executes associated subscriptions.

Branches and code coverage

Intended branches

- Executes active subscriptions, deducts USDC, mints static/dynamic tickets (including dynamic-only and static-only cases), and updates balances.
    - ☑ Test coverage
- Handles execution life-cycle transitions, completes and removes subscriptions on the final day (`EXECUTE_AND_CLOSE`), filters out-of-range static tickets into dynamic ones, and processes multiple recipients in one call.
    - ☑ Test coverage

Negative behavior

- Caller is a keeper.
  - ☑ Negative test

Function call analysis

- `this.jackpot.currentDrawingId()`
  - **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.jackpot.getDrawingState(currentDrawingId)`
  - **What is controllable?** N/A.
  - **If the return value is controllable, how is it used and how can it go wrong?** It is the current drawing state, populated when the previous one is closed, with global state and dynamic values (`bonusMax`).
  - **What happens if it reverts, reenters or does other unusual control flow?** N/A.
- `this.jackpot.maxReferrers()`
  - **What is controllable?** Returns an owner-controlled value.
  - **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._executeSubscription(currentDrawingId, currentDrawingState, recipient) -> this._createTicketArray(userStaticTickets, subscription.dynamicTicketCount, _currentDrawingState.ballMax, _currentDrawingState.bonusballMax) -> this._validateAndFilterStaticTickets(_userStaticTickets, _ballMax, _bonusballMax)`
  - **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._executeSubscription(currentDrawingId, currentDrawingState, recipient) -> this._createTicketArray(userStaticTickets, subscription.dynamicTicketCount, _currentDrawingState.ballMax, _currentDrawingState.bonusballMax) -> this._generateDynamicTickets(finalDynamicTicketCount, _ballMax, _bonusballMax) -> TicketPicker.pickAuto(++this.ticketPickerNonce,`

`_dynamicTicketCount, _ballMax, _bonusballMax)`

  - **What is controllable?** Only `_dynamicTickerCounts` is partially controlled by the user while subscribing.
  - **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._executeSubscription(currentDrawingId, currentDrawingState, recipient) -> this.usdc.approve(address(this.jackpot), totalCost)`
  - **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._executeSubscription(currentDrawingId, currentDrawingState, recipient) -> this.jackpot.buyTickets(tickets, _recipient, subscription.referrers, subscription.referralSplit, JackpotAutoSubscription.AUTO_SUBSCRIPTION_TELEMETRY)`
  - **What is controllable?** `tickets`, `_recipient`, `subscription.referrers`, and `subscription.referralSplit` are user-controlled while subscribing.
  - **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._cancelSubscription(recipient, ExecutionAction.CANCEL_TOO_MANY_REFERRERS) -> SafeERC20.safeTransfer(this.usdc, _recipient, refundAmount)`
  - **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: `cancelSubscription()`

This function allows any user to cancel a subscription linked to their address.

Branches and code coverage

Intended branches

- The subscription data is deleted, and the remaining USDC is refunded.

  ☑ Test coverage

Negative behavior

- The caller has an associated subscription.

  ☑ Negative test

Function call analysis

- `this._cancelSubscription(msg.sender, ExecutionAction.CANCEL_USER_REQUESTED) -> SafeERC20.safeTransfer(this.usdc, _recipient, refundAmount)`
  - **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: `_executeSubscription(uint256 _currentDrawingId, IJackpot.DrawingState _currentDrawingState, address _recipient)`

This is an internal helper that executes one recipient's subscription for a drawing. It builds the ticket array (static plus generated dynamic tickets, replacing out-of-range static tickets), deducts cost from the subscription, approves USDC to the Jackpot, and calls `buyTickets`.

Inputs

- `_currentDrawingId`
  - **Control**: Set by the Jackpot contract — the keeper cannot choose arbitrarily.
  - **Constraints**: Must match the drawing being processed.
  - **Impact**: Recorded in `lastExecutedDrawing`.
- `_currentDrawingState`
  - **Control**: Provided by Jackpot — params ultimately owner-defined.
  - **Constraints**: Must correspond to `_currentDrawingId`.
  - **Impact**: Supplies `ticketPrice`, `ballMax`, and `bonusballMax` for ticket creation and pricing.
- `_recipient`
  - **Control**: Keeper-supplied from the batch list.

- **Constraints**: Should reference an active subscription with sufficient `remainingUSDC`.
- **Impact**: Drives which subscription is executed and who receives tickets.

Branches and code coverage

Intended branches

- Deducts cost, updates `lastExecutedDrawing`, and calls `buyTickets`.
  - ☑ Test coverage
- Handles static and dynamic, dynamic-only, static-only, and static tickets filtered to dynamic when out of range.
  - ☑ Test coverage

5.3. Module: JackpotRandomTicketBuyer.sol

Function: `buyTickets(uint256 _count, address _recipient, address[] _referrers, uint256[] _referralSplitBps, bytes32 _source)`

This function pulls USDC from the caller, generates `_count` random tickets for the current drawing via `TicketPicker.pickAuto(++nonce, ...)` and forwards them to `Jackpot.buyTickets`.

Inputs

- `_count`
  - **Control**: Full control.
  - **Constraints**: Must be greater than zero — gas and `totalCost` scale with this value.
  - **Impact**: Determines ticket quantity and total USDC pulled.
- `_recipient`
  - **Control**: Full control.
  - **Constraints**: Must be nonzero.
  - **Impact**: Receives minted tickets.
- `_referrers`
  - **Control**: Full control.
  - **Constraints**: Forwarded to Jackpot — must align with `_referralSplitBps` and meet Jackpot validation (length cap, nonzero, sum to 1e18).
  - **Impact**: Drives referral payouts — invalid values revert in Jackpot.
- `_referralSplitBps`

    - **Control**: Full control.
    - **Constraints**: Must match `_referrers` length and sum to `PRECISE_UNIT` as enforced by Jackpot.
    - **Impact**: Defines referral weightings — invalid splits revert downstream.
- `_source`
    - **Control**: Full control (telemetry tag).
    - **Constraints**: N/A.
    - **Impact**: N/A.

Branches and code coverage

Intended branches

- Increments the nonce, transfers the exact USDC cost and mints the requested ticket count.
    - ☑ Test coverage
- Reverts on `_count == 0`.
    - ☑ Test coverage
- Reverts on `_recipient == address(0)`.
    - ☑ Test coverage

Function call analysis

- `this.jackpot.currentDrawingId()`
    - **What is controllable?** Jackpot-owned state.
    - **If the return value is controllable, how is it used and how can it go wrong?** Drives which drawing state/ticket prices are used — an incorrect value could over- or under-charge or mint into the wrong drawing.
    - **What happens if it reverts, reenters or does other unusual control flow?** Propagates revert — no external state changes have occurred yet.
- `this.jackpot.getDrawingState(currentDrawingId)`
    - **What is controllable?** Jackpot-owned state.
    - **If the return value is controllable, how is it used and how can it go wrong?** Provides `ticketPrice`, `ballMax`, and `bonusballMax` — misconfigured values affect cost or cause downstream reverts in ticket generation.
    - **What happens if it reverts, reenters or does other unusual control flow?** Reverts the transaction before token transfers.
- `SafeERC20.safeTransferFrom(this.usdc, msg.sender, address(this), totalCost)`

- **What is controllable?** Token contract behavior — caller approval and balance.
- **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.approve(address(this.jackpot), totalCost)`
  - **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.
- `TicketPicker.pickAuto(++this.nonce, _count, drawingState.ballMax, drawingState.bonusballMax)`
  - **What is controllable?** `_count` from caller and ball/bounce bounds from Jackpot state and internal nonce increment.
  - **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.jackpot.buyTickets(tickets, _recipient, _referrers, _referralSplitBps, _source)`
  - **What is controllable?** All params derive from caller inputs plus generated tickets.
  - **If the return value is controllable, how is it used and how can it go wrong?** Jackpot validates referrals and ticket formats, any failure reverts.
  - **What happens if it reverts, reenters or does other unusual control flow?** Revert rolls back transfers/approvals; jackpot could attempt callbacks, but this function is `nonReentrant` to protect reentry into itself.

5.4. Module: TicketPicker.sol

Function: `pickAuto(uint256 _nonce, uint256 _count, uint8 _ballMax, uint8 _bonusballMax)`

This function produces a batch of tickets using on-chain entropy (`chainid`, contract address, `block.prevrandao`, previous blockhash), domain-separated with `PICK_AUTO_DST` and caller-provided nonce, then delegates to `pickMultipleTickets`.

Inputs

- `_nonce`

- **Control**: Caller provided.
- **Constraints**: Should be unique per call, especially within the same block, to avoid collisions.
- **Impact**: Mixes into seed; repeating can produce deterministic batches.

- `_count`
  - **Control**: Caller provided.
  - **Constraints**: Must be greater than zero or reverts via `pickMultipleTickets`; large values are limited by gas.
  - **Impact**: Batch size — drives downstream loop and gas usage.
- `_ballMax`
  - **Control**: Caller/drawing config.
  - **Constraints**: Must allow five unique normals — forwarded to `pickMultipleTickets`.
  - **Impact**: Bounds normal ball selection for all tickets.
- `_bonusballMax`
  - **Control**: Caller/drawing config.
  - **Constraints**: Must be in (0, 255]; enforced downstream.
  - **Impact**: Bounds bonusball selection for all tickets.

Branches and code coverage

Intended branches

- Generates `_count` tickets with valid normals/bonusball bounds. Each ticket has five unique normals within [1, `normalMax`] and a bonusball within [1, `bonusballMax`].
  - ☑ Test coverage
- Deterministic for identical nonces.
  - ☑ Test coverage
- Different nonces produce different batches.
  - ☑ Test coverage

Negative behavior

- Invalid `_count` or bounds revert via `pickMultipleTickets` / `pickSingleTicket` / `pickBonusball` rather than emitting biased tickets.
  - ☑ Negative test

Function: `pickBonusball(uint256 _seed, uint256 _max)`

This function generates a uniformly distributed bonusball in [1, `_max`] using domain-separated Keccak256 and rejection sampling to avoid modulo bias.

Inputs

- `_seed`
  - **Control**: Full control.
  - **Constraints**: N/A.
  - **Impact**: Drives randomness.
- `_max`
  - **Control**: Full control.
  - **Constraints**: Must be in (0, 255].
  - **Impact**: Defines inclusive upper bound.

Branches and code coverage

Intended branches

- Returns value in [1, `_max`].
  - ☑ Test coverage

Negative behavior

- Reverts if `_max` is zero or greater than 255.
  - ☑ Negative test

Function: `pickMultipleTickets(uint256 _seed, uint256 _count, uint8 _ballMax, uint8 _bonusballMax)`

This function generates a batch of tickets by deriving per-ticket seeds from a base seed and delegating to `pickSingleTicket` for each.

Inputs

- `_seed`
  - **Control**: Full control.
  - **Constraints**: Reuse yields deterministic batches.
  - **Impact**: Base for per-ticket seeds — affects entire batch randomness.

- `_count`
    - **Control**: Full control.
    - **Constraints**: Must be greater than zero.
    - **Impact**: Length of returned array and loop iterations.
- `_ballMax`
    - **Control**: Full control.
    - **Constraints**: Must allow five unique normals (≥ 5), passed to `pickSingleTicket`.
    - **Impact**: Bounds normal ball selection.
- `_bonusballMax`
    - **Control**: Full control.
    - **Constraints**: Must be in (0, 255], enforced in `pickBonusball` via `pickSingleTicket`.
    - **Impact**: Bounds bonusball selection for all tickets.

Branches and code coverage

Intended branches

- Generates `_count` tickets, each satisfying normal/bonusball bounds.
    - ☑ Test coverage
- Deterministic for identical base seeds.
    - ☑ Test coverage
- Different base seeds yield different batches.
    - ☑ Test coverage

Negative behavior

- Zero `_count` reverts.
    - ☑ Negative test

Function: `pickSingleTicket(uint256 _seed, uint8 _normalMax, uint8 _bonusballMax)`

This function constructs a single ticket by drawing five unique normals via Fisher-Yates rejection sampling and a bonusball via `pickBonusball`, using deterministic entropy per seed.

Inputs

- `_seed`
  - **Control**: Full control.
  - **Constraints**: Reused seeds yield identical tickets; quality governs unpredictability.
  - **Impact**: Drives both normals and bonusball generation.
- `_normalMax`
  - **Control**: Full control.
  - **Constraints**: Must be ≥ 5 so `FisherYatesRejection.draw` can sample five distinct numbers within [1, `_normalMax`].
  - **Impact**: Upper bound for normals.
- `_bonusballMax`
  - **Control**: Full control.
  - **Constraints**: Must be in (0, 255], enforced by `pickBonusball`.
  - **Impact**: Upper bound for bonusball.

Branches and code coverage

Intended branches

- Produces tickets with five unique normals within bounds and a bonusball within bounds.
  - ☑ Test coverage
- Deterministic for identical seeds.
  - ☑ Test coverage
- Different seeds yield different tickets.
  - ☑ Test coverage

6. Assessment Results

During our assessment on the scoped Megapot (V2) contracts, we discovered five findings. No critical issues were found. One finding was of medium impact, two were of low impact, and the remaining findings were informational in nature.

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.