MacroMillions Security Review

By Prism

Auditors

Lead Security Researcher: Jacobo Lansac (@Jacopod)

July 31, 2025

Contents

1 Findings Summary

ID	Severity	Issue Title	Team Response
M1	Medium	Protocol settlement can halt when first/second degree referrals are blacklisted because the validation checks winner instead of referral addresses	Fixed
M2	Medium	Users with perma tickets have underrepresented ticket counts because ‘getNumPriorTicketsOwned()‘ excludes perma tickets	Fixed
M3	Medium	The referral system can be DOS because by attackers that front-run referrals to make targets ineligible	Fixed
L1	Low	Token transfers possible before trading is active because ‘transferTicketFrom()‘ bypasses ‘tradingActive‘ check	Acknowledged
L2	Low	Ambiguity between actual index 0 and non-existent ticket because ‘getCurrentIndex()‘ returns 0 for both cases	Fixed
L3	Low	Inconsistent tracking of ticket ownership for permaTicketNFT contract because ‘ticketOwnedCheckpoints‘ are not updated	Fixed

2 Introduction

A time-boxed review of **MacroMillions Referrals** smart contracts, developed by MacroMillions. The focus of this review is to identify security vulnerabilities, explain their root-cause and provide solutions to mitigate the risk.

Gas optimizations are not the main focus but will also be identified if found.

3 About Prism

Prism delivers specialized security solutions for blockchain and AI companies. We go beyond traditional audits, offering bespoke penetration testing, adversary simulation, and AI security solutions to meet the needs of every client. With tailored services and best-in-class expertise, we safeguard your business against the most sophisticated threats, allowing you to focus on innovation.

Learn more about us at prismsec.xyz

4 About MacroMillions

MacroMillions is a gamified raffle system where the chances of winning are proportional to your holdings of the MACRO token.

5 Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time and resource-bound effort to find as many vulnerabilities as possible, but there is no guarantee that all issues will be found. This security review does not guarantee against a hack. Any modifications to the code will require a new security review.

This review does not focus on the correctness of the happy paths. Instead, it aims to identify potential security vulnerabilities and attack vectors derived from an unexpected and harmful usage of the contracts. The devs are ultimately responsible for the correctness of the code and its intended functionality.

A security review is not an endorsement of the underlying business or product and can never be taken as a guarantee that the protocol is bug-free.

6 Risk classification

Severity level	Impact: High	Impact: Medium	Impact: Low
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

6.1 Impact

- **High**: leads to a significant material loss of assets in the protocol or significantly harms a group of users.
- **Medium**: only a small amount of funds can be lost (such as leakage of value) or a core functionality of the protocol is affected.
- **Low**: can lead to unexpected behavior with some of the protocol's functionalities that are not so critical.

6.2 Likelihood

- **High**: almost certain to happen, easy to perform or highly incentivized.
- **Medium**: only conditionally possible or incentivized, but still relatively likely
- **Low**: requires multiple unlikely conditions, or little-to-no incentive

6.3 Action required for severity levels

- **Critical**: Must fix as soon as possible (if already deployed)
- **High**: Must fix (before deployment if not already deployed)
- **Medium**: Should fix
- **Low**: Could fix

7 Executive Summary

Scope details

Project Name	MacroMillions
Type of Project	Raffle, Referral System
Repository	MacroMillions/macro-millions-contracts/
Review commit	bd07fc1fac2a403ceec052cdd175aadae54f2aa5
Mitigation commit	a9c951a900fd853581f50365c68fcbe8034dbb10
Audit Timeline	2025-07-23 - 2025-07-39
Methods	Manual Review, Testing

Files in scope	nSLOC	In scope
contracts/core/RaffleSplitter.sol	44	44 (full)
contracts/referral/Referrable.sol	30	30 (full)
contracts/referral/MacroReferral.sol	119	119 (full)
contracts/core/MacroMillions.sol	654	+175 lines, -3 lines
contracts/core/RaffleSettle.sol	277	+179 lines, -27 lines
contracts/core/PermaTicketNFT.sol	198	+73 lines, -13 lines
Totals	-	**620**

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	3

8 Architecture review

8.1 Architecture Overview

The changes included in the scope were:

- A referral system of first and second degree
- Dynamic cap of the raffle price
- Ticket transferring within MACRO contract (transferring also underlying tokens)

8.1.1 Referral System Overview

The referral system consists of two main contracts:

1. MacroReferral.sol - Core referral logic

- Two-tier referral structure: First-degree (direct) and second-degree (referrer's referrer)
- Fee rates: First-degree: 2.5% base, Second-degree: 1.25% base
- Multipliers: Perma ticket holders get 2x multiplier, Super ticket holders get 2x multiplier (stackable)

2. Referrable.sol - Abstract contract for integration

- Bonus ticket system: Referrers get 1 bonus ticket for every 10 tickets they refer
- Helper functions: _setReferral() and _ticketsReferred() for easy integration

8.1.2 Gas considerations

The ticketing logic relies heavily on `for` loops, which cause high gas consumption. While this is not a problem in cheap blockchains such as the intended deployment chain (Base), it is worth mentioning as the contract wouldn't be viable in ethereum mainnet.

9 Findings

9.1 Medium Risk

9.1.1 [M1] - Protocol settlement can halt when first/second degree referrals are blacklisted because the validation checks winner instead of referral addresses

Location:

RaffleSettle.sol - `_pullSettle()` function

Description:

The blacklist validation for first and second degree referrals incorrectly checks if the winner is blacklisted instead of checking if the referral addresses themselves are blacklisted. The code checks `!IUSDC(USDC).isBlacklisted(_winner)` when it should check `!IUSDC(USDC).isBlacklisted(feeAndReferral.firstDegree)` and `!IUSDC(USDC).isBlacklisted(feeAndReferral.secondDegree)`.

Impact:

If either the first/second referral addresses are blacklisted, the settlement process fails. As Chainlink VRF won't attempt a second call to `fulfillRandomWords()`, the Raffle contract will be halted forever.

Note that a true enemy of the protocol can increase the chances of halting by forcing his blacklist and using that wallet for referrals, as discussed in the first review of this protocol.

Mitigation:

Change the blacklist checks to validate the referral addresses instead of the winner address.

```
function _pullSettle(uint256 _seed) internal returns (bool _settled, address _winner, uint256
↪  _winningTicket) {
    // ...
-   if (feeAndReferral.firstDegree != address(0) && !IUSDC(USDC).isBlacklisted(_winner)) {
+   if (feeAndReferral.firstDegree != address(0) &&
↪   !IUSDC(USDC).isBlacklisted(feeAndReferral.firstDegree)) {
        // ...
    }
-   if (feeAndReferral.secondDegree != address(0) && !IUSDC(USDC).isBlacklisted(_winner)) {
+   if (feeAndReferral.secondDegree != address(0) &&
↪   !IUSDC(USDC).isBlacklisted(feeAndReferral.secondDegree)) {
        // ...
    }
}
```

9.1.2 [M2] - Users with perma tickets have underrepresented ticket counts because `getNumPriorTicketsOwned()` excludes perma tickets

Location:

MacroMillions.sol - `getNumPriorTicketsOwned()` function

Description:

The `getNumPriorTicketsOwned` function only counts regular tickets owned by a user at a specific timestamp but does not include the number of perma tickets owned. This results in an incomplete count of total tickets owned by a user.

Impact:

Users owning perma tickets will have their total ticket count underrepresented, potentially reducing benefits that are proportional to the number of tickets owned and affecting any calculations or rewards based on historical ticket ownership.

Since these ticket counts will be used for things like vote-counting, a misrepresentation of these tickets would affect the representation of the user.

Code Context:

```
function getNumPriorTicketsOwned(address _who, uint256 _timestamp) external view returns (uint256) {
    require(_timestamp < block.timestamp);

    uint256 nCheckpoints = numTicketsOwnedCheckpoints[_who];
    if (nCheckpoints == 0) {
        return 0;
    }
    // ... binary search logic ...
    return ticketOwnedCheckpoints[_who][lower].tickets;
    // Missing: perma tickets count
}
```

Mitigation:

Read the the number of perma tickets owned by `_who` at `timesamp` and add it to the returned value in all of the function returns.

9.1.3 [M3] - The referral system can be DOS because by attackers that front-run referrals to make targets ineligible

Location:

MacroReferral.sol - `canBeReferred()` function

Description:

The referral system can be denied by exploiting the `canBeReferred()` function logic. The function prevents users who are already masters (have referred someone) from being referred themselves. An attacker can DOS the referral system by front-running legitimate referral transactions and setting arbitrary users as referrers first, making them masters and preventing them from being referred by legitimate referrers.

Example attack:

- Alice gives bob a referral link
- An attacker is a competitor of Alice, and wants to avoid that Alice's referral is ever used
- When bob attempts to call `RaffleSettle.purchase()` (with Alice as referral), the attacker front-runs like this:
  - 1. Attacker creates a dummy account.
  - 2. Attacker calls `stake()` with the dummy account, passing bob as a referral. Now bob is first-degree referral of dummy.
- Then, when bob's transaction is processed, `RaffleSettle.purchase()` calls _setReferral(bob, Alice) which will read the firstDegreeReferral of msg.sender (bob), and will get address(0) as he hasn't set a referral yet (ok so far).
- However, in the next line inside _setReferral(), the call canBeReferred(msg.sender) will return False, because msg.sender (bob), is already a referrer (of dummy), so numFirstDegreeReferralsForReferer[bob] > 0.
- Therefore, the purchase() will continue assuming that the referral is address(0), instead of Alice.
- The attacker can repeat this process for all the bobs who try to use Alice's referral link, effectively Denying the referral benefits to Alice

Impact:

The referral system can be completely disabled for targeted users, disrupting the protocol's referral mechanism and preventing legitimate users from earning referral rewards.

Mitigation:

Review and redesign the referral system logic to prevent such manipulation. Consider implementing a time-based or stake-based protection mechanism.

Code Context:

```
function canBeReferred(address _who) public view returns (bool _canBeReferred) {
    // @audit an attacker can turn any user being referred into an active user, DOSing the referral
    ↪  action.
    bool activeUser = numFirstDegreeReferralsForReferer[_who] > 0;
    if (firstDegreeReferral[_who] == address(0) && !activeUser) return true;
}
```

Response: Fixed

Fixed by adding a mapping of enabled referrers. A referrer has to sign a tx setting himself as a referrer before his address can be used as a referral.

9.2 Low Risk

9.2.1 [L1] - Token transfers possible before trading is active because `transferTicketFrom()` bypasses `tradingActive` check

Location:

MacroMillions.sol - `transferTicketFrom()` function

Description:

The `transferTicketFrom` function allows transferring tokens before trading is active, bypassing the `tradingActive` requirement that is imposed in the normal `_transfer` function. This creates an inconsistency in the trading restrictions.

Impact:

Users can transfer tokens before they are officially allowed to trade, though the impact is limited since there won't be any circulating supply until trading is active.

Mitigation:

Add a requirement in `transferTicketFrom()` to revert before trading is active, similar to the check in `_transfer`.

Team Response

Acknowledged because there won't be any circulating supply before trading is active.

9.2.2 [L2] - Ambiguity between actual index 0 and non-existent ticket because `getCurrentIndex()` returns 0 for both cases

Location:

MacroMillions.sol - `getCurrentIndex()` function

Description:

When a ticket number doesn't exist, the `getCurrentIndex` function returns index 0, which is a valid index but not the correct one. This creates ambiguity between an actual index 0 and a non-existent ticket.

Impact:

The function provides conflicting output between actual index 0 and a non-existing ticket. However, the impact is limited since the only caller (`transferTicketFrom`) checks ticket ownership before calling this function.

Mitigation:

Return `type(uint256).max` or similar value to clearly indicate a non-existent ticket.

Code Context:

```
function getCurrentIndex(uint256 _ticketNum) public view returns (uint256 index) {
    if (_ticketNum >= nextTicketId) return type(uint256).max;
    address owner = getCurrentTicketOwner(_ticketNum);
    uint256[] memory _userTickets = userTickets[owner];

    uint256 length = _userTickets.length;
    for (uint256 i; i < length; i++) {
        if (_userTickets[i] == _ticketNum) return i;
    }
    // TODO Returns 0 by default, should return type(uint256).max
}
```

9.2.3 [L3] - Inconsistent tracking of ticket ownership for permaTicketNFT contract because `ticketOwnedCheckpoints` are not updated

Location:

MacroMillions.sol - `createPermaTicket()` function

Description:

When creating a perma ticket, the function updates `ticketCheckpoints` and `numTicketCheckpoints` but fails to update `ticketOwnedCheckpoints` and `numTicketsOwnedCheckpoints` for the permaTicketNFT contract. This creates inconsistency in tracking ticket ownership.

Impact:

Inconsistent tracking of ticket ownership for the permaTicketNFT contract, which could affect functions that rely on these checkpoints for historical data.

Mitigation:

Add updates to `ticketOwnedCheckpoints` and `numTicketsOwnedCheckpoints` when creating perma tickets for consistency.

Code Context:

```
function createPermaTicket(uint256 _nftId) external {
    // ...
    uint256 nCheckpoints = numTicketCheckpoints[newTicket_];
    if (nCheckpoints > 0 && ticketCheckpoints[newTicket_][nCheckpoints - 1].fromTimestamp ==
    ↪  block.timestamp) {
        ticketCheckpoints[newTicket_][nCheckpoints - 1].owner = permaTicketNFT;
    } else {
        ticketCheckpoints[newTicket_][nCheckpoints] = TicketCheckpoint(block.timestamp, permaTicketNFT);
        numTicketCheckpoints[newTicket_] = nCheckpoints + 1;
    }
    // TODO Missing: ticketOwnedCheckpoints and numTicketsOwnedCheckpoints updates
}
```