MacroMillions Security Review

By Prism

Auditors

Lead Security Researcher: Jacobo Lansac (@Jacopod)

July 14, 2025

Contents

1 Findings Summary

Issue ID	Severity	Issue Title	Team Response
C1	Critical	An attacker can mint infinite raffle tickets by self-transferring tokens	Fixed
H1	High	An attacker can attempt to break the protocol by sending tickets to a USDC-blacklisted address	Fixed
H2	High	Missing access control in ‘setCanNotMint()‘ function allows attackers to deny PermaTickets minting	Fixed
M1	Medium	Insufficient block confirmations make the Raffle outcome vulnerable to reorgs	Fixed
M2	Medium	Flawed rebasing of ‘tokensForTicket‘ makes Units-for-ticket not constant over time allowing users to keep tickets while holding no MACRO after a period	Fixed
M3	Medium	UniswapRouter will cause MACRO transfers to revert because the MACRO token is the receiver and is also in the swap path	Fixed
M4	Medium	Raffle winners can be identified wrongly if the winner ‘ticketId‘ is a perma-ticket	Fixed
L1	Low	Lack of slippage protection in token swaps is vulnerable to sandwich attacks	Acknowledged
L2	Low	Unbounded gas consumption in ‘debase()‘ can halt the protocol if there are many pending epochs	Fixed
L3	Low	O(n) gas cost for token transfers can reach block gas limit for large transfers	Acknowledged
L4	Low	Imprecise percentage system doesn’t allow decimals in percentages	Fixed
L5	Low	If no tickets are minted in the first epoch, the raffle settlement will revert with Division by zero error and halt the raffle	Acknowledged
L6	Low	Rounding error in staking transfers	Acknowledged
L7	Low	Misleading return value when minting disabled	Fixed
L8	Low	Non-debasing addresses can keep tickets without holding any token balance	Fixed
L9	Low	Unbounded gas for-loop to handle non-debasing addresses could reach block gas limit	Acknowledged
L10	Low	Debasing mechanism uses compound interest instead of simple interest and the supply does not decrease by 0.125% every epoch	Fixed

2 Introduction

A time-boxed review of **MacroMillions Raffle** 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.

Security researchers hold no responsibility for the findings provided in this document. 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. This security review is focused solely on the security aspects of the smart contracts.

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

Summary

Project Name	MacroMillions
Repository	MacroMillions/macro-millions-contracts/
Review Commit	bd07fc1fac2a403ceec052cdd175aadae54f2aa5
Mitigation Commit	d3f5425bd85d982643737df89f2647073d0ab9e0
Type of Project	Raffle, Debasing ERC20
Audit Timeline	2025-06-29 - 2025-07-13
Methods	Manual Review, Testing

Issues Found

Critical Risk	1
High Risk	2
Medium Risk	4
Low Risk	10
Gas Optimizations	6
Informational	15

7.1 Files in scope

Files in scope	Solidity nSLOC	Notes
contracts/core/MACROStaking.sol	30	
contracts/core/MacroBurnFees.sol	79	
contracts/core/PermaTicketNFT.sol	146	ERC721Enumerable
contracts/core/RaffleSettle.sol	168	VRFV2PlusWrapperConsumerBase
contracts/core/MacroMillions.sol	548	ERC20, debasing
Totals	**971**	

8 Architecture review

8.1 Protocol Overview

A Raffle system where the chances of winning are proportional to the amount of token holding, mixed with some debasing mechanics.

Debasing ERC20 Token

- Debasing system with debasing index and a units system
- The ratio from units to balance is decreased to debase the token
- Periodic balance reduction creates deflationary pressure
- Users can stake to avoid debasing but lose raffle eligibility
- Certain contracts are excluded from debasing, which also removes raffle eligibility
- Ticket allocation remains constant despite balance reduction, i.e., units-per-token remains constant

Ticketing System

- Regular tickets based on token holdings
- PermaTickets provide permanent eligibility via NFT ownership
- Dead tickets system for non-eligible addresses
- Checkpoint system tracks ownership at specific timestamps

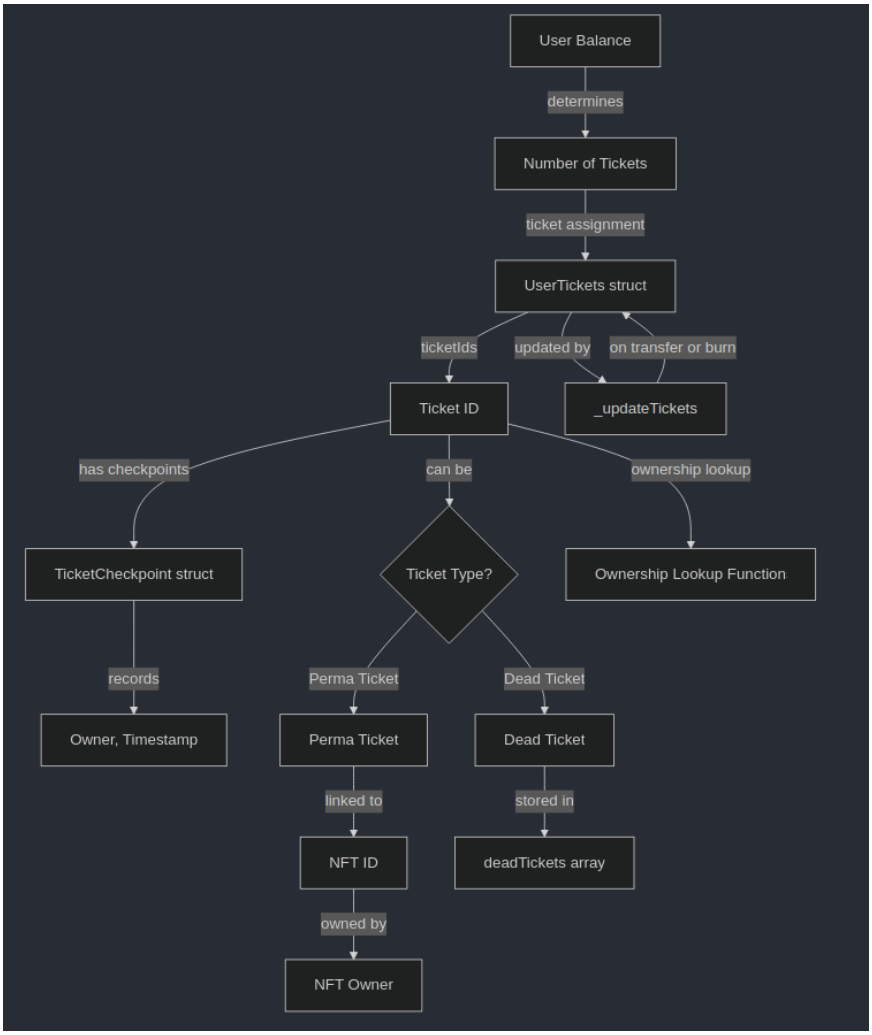

Figure 1: Ticket System Diagram

Fee Distribution

- Buy/sell fees on MACRO token, which are accumulated as MACRO
- Some MACRO is burned, and some is swapped for USDC and transferred to a Raffle splitter, which regulates how much goes to the Raffle
- In every raffle, 50% of USDC balance is allocated to the pot
- Out of that 50%, 20% is given to the team as fees (so 10% of the round funds)

Raffles

- Chainlink VRF for winner selection
- Aligned with debase epochs
- State transitions: Initialized → Awaiting Settlement → Randomness Requested → Settled

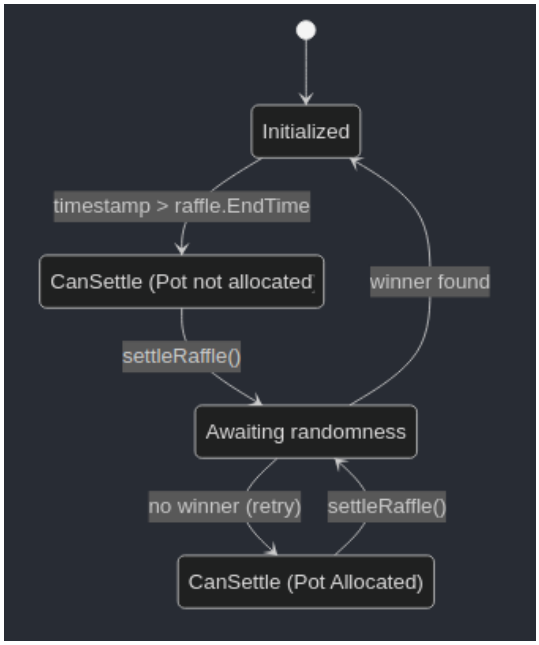

Figure 2: Raffle State Transitions

8.2 Protocol invariants

- Staked MACRO immune to debasing
- Constant ratio between units and tickets. If a user holds MACRO (no sells/buys/transfers) the ticket allocation should remain constant
- An account's probability of winning the raffle should be proportional to their MACRO holding at raffle settlement, except PermaTickets
- `totalTickets - permaTickets <= totalSupply / tokensForTicket`

8.3 Architecture Concerns

Tickets to token ratio complexity

The `tokensForTicket` rebasing mechanism could be simplified using `unitsForTickets`. The core required property is that the units-to-ticket ratio is constant, so there is no need to debase the supply AND the `tokensForTicket`. Instead `tokensForTicket` should be redefined as `unitsForTicket`. Simpler and closer to the actual requirement.

For-loops

Several contracts rely heavily in for-loops, (non-debasing addresses, ticket updates, etc) which makes it non-gas efficient. For instance, token transfers gas consumption scales linearly with the amount transferred. This would be a problem in eth mainnet, but as the contracts will be deployed on base, the issue is less critical.

Lack of tests

The repository has no test suite. This is a major concern as even the strongest manual review can miss bugs. A strong unittest suite is always recommended.

9 Findings

9.1 Critical Risk

9.1.1 [C1] - An attacker can mint infinite raffle tickets by self-transferring tokens

Location:

- Contract: MacroMillions
- Function: _updateTickets()

Summary

The ticket allocation system can be exploited to mint an unlimited number of raffle tickets through self-transfers. The `_updateTickets()` function calculates ticket changes based on balance differences before and after transfers, but lacks validation to prevent self-transfers. This allows attackers to manipulate the ticket counting mechanism resulting in an increased number of tickets owned with now increase in their MACRO balance.

Impact

An attacker can mint an infinite amount of raffle tickets by repeatedly transferring tokens to themselves, after setting up their balances in a specific (but trivial) way. This completely breaks the raffle fairness mechanism, allowing the attacker to have a near-100% probability of winning all raffle prizes while other legitimate participants have virtually no chance.

Technical Details

The vulnerability exists in the ticket calculation logic. When a transfer happens, the function estimates the new ticket-balance the sender is entitled to using the `tokensForTicket` before and after the transfer. The same is done for the receiver to know how many new tickets should be minted.

```
function _updateTickets(address _from, address _to, uint256 _amount) internal {
    UserTickets storage fromTickets = userTickets[_from];
    UserTickets storage toTickets = userTickets[_to];

    if (!isNonDebasingAddress[_from]) {
        uint256 fromBalanceBefore = balanceOf(_from);
        uint256 fromBalanceAfter = fromBalanceBefore - _amount;
        uint256 fromTicketsBefore = fromBalanceBefore / tokensForTicket;
        uint256 fromTicketsAfter = fromBalanceAfter / tokensForTicket;

        uint256 newDeadTickets = fromTicketsBefore - fromTicketsAfter;
        // Tickets are "killed" from sender
    }

    if (!isNonDebasingAddress[_to]) {
        uint256 toBalanceBefore = balanceOf(_to);
        uint256 toBalanceAfter = toBalanceBefore + _amount;
        uint256 toTicketsBefore = toBalanceBefore / tokensForTicket;
        uint256 toTicketsAfter = toBalanceAfter / tokensForTicket;

        uint256 additionalTickets = toTicketsAfter - toTicketsBefore;
        // New tickets are minted to receiver
    }
}
```

Note however, that if the amount can decrease the sender's ticket-balance by a different amount than the receiver's ticket balance. If an attacker self-transfers, he can manage to mint new tickets as receiver while no tickets are burner as the sender. Here is an example:

```
// INITIAL CONDITIONS
tokensForTicket = 10 // tokens
attackerBalance = 98 // tokens
attackerTickets = 9 // rounded down from 98/10

// ATTACK: the attacker self-transfers 5 tokens
fromBalanceBefore = 98 // 9 tickets before
fromBalanceAfter = 93  // 9 tickets after
newDeadTickets = 0     // no tickets removed!

toBalanceBefore = 98   // 9 tickets before
toBalanceAfter = 103   // 10 tickets after
additionalTickets = 1  // one new ticket minted out of thin air!!
```

Fix:

Add a check to prevent self-transfers:

```
function _updateTickets(address _from, address _to, uint256 _amount) internal {
+   require(_from != _to, "Self-transfers not allowed");
    // ... rest of the function
}
```

Team response:

The suggested requirement has been added in the `_transfer()` function.

9.2 High Risk

9.2.1 [H1] - An attacker can attempt to break the protocol by sending tickets to a USDC-blacklisted address

Contract: RaffleSettle.sol

Circule-USDC can blacklist accounts, making their USDC transfers revert (they can't send or receive), see etherscan-USDC-implementation.

In `RaffleSettle.fulfillRandomWords()`, the winner is found and the allocated prize is directly transferred to the winner (inside `_pullSettle()`). If this account is blacklisted, the transaction will revert. Chailink won't reattempt to fulfill randomness, and the contract will be in a locked state in which it is awaiting randomness, but Chainlink VRF won't reattempt to fulfill randomness. This is effectively a protocol halt.

Note that USDC is not required to participate in the raffle, only MACRO or permatickets. So, an attacker can purchase MACRO and transfer it to a blacklisted address. Then chance has a role, and if this blacklisted address wins a raffle, then the protocol is broken. An attacker can increase the chances of halting the protocol by sending more MACRO.

Note that there is a raffle every 6 hours and there may not be so many tickets in circulation, so the chances are non cero. But of course there is no economic incentive to do this, as it costs MACRO to the attacker.

```
    function _pullSettle(uint256[] memory _ticketNumbers) {
        // ...
        uint256 toSend = currentRaffle.amount;
        if (toSend > 0) {
            // @audit this will fail if the _winner is blacklisted by USDC
            IERC20(USDC).transfer(_winner, toSend);
        }
```

Impact:

An attacker can force an eventual protocol halt by sending tickets to a USDC blacklisted account. Low probability, but critical impact for the protocol.

Fix:

1- Easy option: Make sure the winner is not blacklisted. If it is, then keep the funds here for a future raffle. Caveat: this doesn't prevent if the USDC transfer fails for another reason.

```
    function _pullSettle(uint256[] memory _ticketNumbers) {
        // ...
        uint256 toSend = currentRaffle.amount;
-       if (toSend > 0) {
+       if ((toSend > 0) & (!USDC.isBlacklisted(_winner))) {
            IERC20(USDC).transfer(_winner, toSend);
        }
```

2- Robust option: make the winners pull the funds instead of sending them. Caveats: all other contracts holding tickets need to implement functions to pull funds.

Response:

The function `_pullRaffle()` skips the winner if it is blacklisted, and attempts to draw a new winner.

9.2.2 [H2] - Missing access control in `setCanNotMint()` function allows attackers to deny PermaTickets minting

Location:

- Contract: PermaTicketNFT
- Function: setCanNotMint()

Description

The `setCanNotMint()` function lacks proper access control, allowing any user to disable the minting functionality of Perma Tickets. This creates a denial-of-service vulnerability where malicious actors can prevent legitimate users from minting Perma Tickets anytime, at no cost.

Impact

An attacker can frontrun any legitimate minting transaction by calling `setCanNotMint()` first, causing all subsequent `mint()` calls to revert. This effectively denies the possibility of minting PermaTickets, breaking a core protocol functionality.

Code

```
/// @notice Set can NOT mint perma ticket
function setCanNotMint() external {
    require(canMint, "Already can NOT mint");
    canMint = false;
}
```

Recommendation

Add the `onlyOwner` modifier to restrict access to authorized users only:

```
-   function setCanNotMint() external {
+   function setCanNotMint() external onlyOwner {
        require(canMint, "Already can NOT mint");
        canMint = false;
}
```

9.3 Medium Risk

9.3.1 [M1] - Raffle winners can be identified wrongly if the winner `ticketId` is a perma-ticket

Location:

MacroMillions contract, `getPriorTicketOwner()` function

Description:

The `RaffleSettle` contract calls `getPriorTicketOwner()` to find out the owner of a perma-ticket NFT in case a perma-ticket wins a raffle. When querying historical ownership of perma-tickets, the function returns the **current** owner instead of the owner at the specified timestamp (the end of the raffle).

If a perma ticket is transferred after a raffle ends but before settlement, the new owner receives the prize instead of the rightful owner who held the ticket during the raffle period.

Impact:

Wrong winners receive raffle prizes, creating unfair outcomes and potential disputes. The rightful winner loses their prize despite owning the ticket at the raffle settlement.

Code:

```
    if (isPermaTicket[_ticketNum] && permaTickets[_ticketNum].permaTicketFrom <= _timestamp) {
        // This returns CURRENT owner, not owner at _timestamp
        return IERC721(permaTicketNFT).ownerOf(permaTickets[_ticketNum].nftId);
    }
```

Fix:

```
    if (isPermaTicket[_ticketNum] && permaTickets[_ticketNum].permaTicketFrom <= _timestamp) {
-       return IERC721(permaTicketNFT).ownerOf(permaTickets[_ticketNum].nftId);
+       return IERC721(permaTicketNFT).getPriorPermaOwner(permaTickets[_ticketNum].nftId, _timestamp);
    }
```

9.3.2 [M2] - UniswapRouter will cause MACRO transfers to revert because the MACRO token is the receiver and is also in the swap path

Location:

MacroMillions contract, `_swapTokensForUSDC` function

Description:

Uniswap V2 reverts when the `to` address is one of the contracts in the swap path. The current implementation sets the receiving address to `address(this)` (the MacroMillions contract), but since MACRO is one of the tokens being swapped, this will cause all swap transactions to revert.

Code:

```
uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
    tokenAmount,
    0, // accept any amount of USDC
    path,
    address(this), // This causes revert when MACRO is in the path
    block.timestamp
);
```

Impact:

All token swapping will fail, effectively halting fee processing. While this can be worked around by setting a very high `swapAmountAt` to prevent swaps, it requires manual intervention.

Response:

The receiver in the swap is now the `raffleSplitter`.

9.3.3 [M3] - Flawed rebasing of `tokensForTicket` makes Units-for-ticket not constant over time allowing users to keep tickets while holding no MACRO after a period

Location:

MacroMillions contract, `debase()` function.

Description:

A core invariant of the protocol is that if a user buys MACRO and doesn't buy/sell/transfer, he should be entitled to the same number of tickets over time.

This mechanic is implemented with a rebasing of the `tokensForTicket` that aims to counter the debasing of the supply. However, the asymmetry in increasing/decreasing percentages make the ration units-to-tickets **not constant**. More specifically, this causes approximately 0.5% decrease in the "unitsForTicket" every 14 days, meaning the same units gradually gain more ticket-buying power over time.

This breaks the invariant that `unitsForTicket` should remain constant and creates an exploitable scenario where an attacker can:

1. Purchase MACRO tokens and receive tickets (e.g., 100 tickets)
2. Wait for several epochs (e.g., 60 days) while `unitsForTicket` decreases
3. Sell all MACRO tokens, which should leave him with 0 tickets
4. Due to the decreased `unitsForTicket`, he ends up with 6 tickets despite holding no MACRO

Code:

```
// This calculation affects unitsForTicket over time
uint256 ticketDebaseAmt = tokensForTicket * debaseRate / DEBASE_DENOM;
uint256 newTicketAmt = tokensForTicket - ticketDebaseAmt;
```

Impact:

Users can exploit this to maintain raffle participation without holding MACRO tokens, reducing other participants' winning chances.

Fix:

The change is not a one-liner. I having `tokensForTicket` which needs to be counter-debased, use instead a `unitsForTicket`, which can be constant. As a user's units balance doesn't change unless you buy/sell/transfer, the ticket allocation wouldn't change either.

Response:

A constant variable named `UNITS_FOR_TICKETS` has been introduced, and a function `tokensForTickets()` which converts those units to a token balance with `balanceForUnits()`.

9.3.4 [M4] - Insufficient block confirmations make the Raffle outcome vulnerable to reorgs

Location:

RaffleSettle contract initialization

Description:

The raffle system uses only 3 block confirmations for Chainlink VRF requests, which may be insufficient for a high-value system handling USDC prizes. Base L2 has an average block time of 2 seconds, and the recommended confirmation count for high confidence finality is typically 12 blocks. This could lead to potential issues if the chain reorganizes before the raffle settles.

For reference, Circle (USDC issuer) uses 3-9 minutes (equivalent to 12 Ethereum blocks) for confirmations, which translates to approximately 90-270 Base blocks, see Circle-required-block-confirmations.

Code:

```
uint16 public requestConfirmations = 3;
// Should be increased to a higher value
```

Impact:

Chain reorganization could potentially affect raffle results, compromising the integrity of the raffle system.

Fix:

Increase the number of `requestConfirmations` in `RaffleSettle` contract.

Team response:

The `requestConfirmations` was increased to 15.

9.4 Low Risk

9.4.1 [L1] - Unbounded gas consumption in `debase()` can halt the protocol if there are many pending epochs

Location:

MacroMillions contract, `debase()` function.

Description:

The gas consumption in the `debase()` function is unbounded. If no debases happen for an extended period, the function could consume excessive gas and revert, effectively halting all token transactions and the entire protocol. The function uses a `while` loop that processes all missed debasing epochs, which could accumulate if the contract is inactive for a long time.

Code:

```
function debase() public {
    if (block.timestamp >= currentDebaseEpoch.endTime) {
        while(block.timestamp >= currentDebaseEpoch.endTime) {
            // Processing logic for each missed epoch
            // This loop could run many times if debasing hasn't occurred for a while
        }
    }
}
```

Impact:

High impact with low probability - if debasing hasn't occurred for a long time, all subsequent transactions could fail, halting the protocol until manual intervention.

Fix:

Consider adding an input parameter to `debase()` which allows a max number of debases. If too many epochs are pending, it is possible to run `debase()` in several transactions without causing out-of-gas errors that would halt the contract.

Response

Fixed, by adding a constant `MAX_DEBASES = 25`, and each call to `debase()` cannot debase more than that.

9.4.2 [L2] - Non-debasing addresses can keep tickets without holding any token balance

Location:

MacroMillions contract, `addNonDebasingAddress()` function.

Description:

When an address with a positive MACRO balance is added to the non-debasing list, they can transfer their tokens to another address while keeping their raffle tickets, essentially duplicating the number of tickets they own. This happens because non-debasing addresses don't have their tickets updated during transfers via the `_updateTickets` function.

This creates two problems:

1. The original holder keeps tickets and participates in raffles for free
2. The token recipient gets new tickets, artificially inflating the total ticket supply at the expense of other holders

Code:

```
    function _updateTickets(address _from, address _to, uint256 _amount) internal {
        UserTickets storage fromTickets = userTickets[_from];
        UserTickets storage toTickets = userTickets[_to];

        // for non-debasing addressses, the tickets are not removed
        if (\!isNonDebasingAddress[_from]) {
            // this piece deducts tickets corresponding to the transferred amount
            uint256 fromBalanceBefore = balanceOf(_from);
            uint256 fromBalanceAfter = fromBalanceBefore - _amount;
            uint256 fromTicketsBefore = fromBalanceBefore / tokensForTicket;
            uint256 fromTicketsAfter = fromBalanceAfter / tokensForTicket;
        }
    }
```

Impact:

Unfair advantage for certain addresses, reducing the winning chances of legitimate participants and creating artificial ticket inflation.

Fix:

Only addresses with zero balance can be marked as non-debasing.

```
    function addNonDebasingAddress(address _who) external onlyOwner {
+       require(balanceOf(_who) == 0, "non-zero balance");
        require(\!isNonDebasingAddress[_who], "Already Non Debasing Address");
        isNonDebasingAddress[_who] = true;
        nonDebasingAddresses.push(_who);
```

9.4.3 [L3] - Misleading return value when minting disabled

Location: PermaTicketNFT contract

Description:

When minting is disabled (`!canMint`), `currentMintCost()` returns 0 instead of a high value, which may mislead integrating components into thinking they have enough MACRO to mint.

Code:

```
function currentMintCost() public view returns (uint256) {
    if (!canMint) return 0; // Should return very high value instead
}
```

Impact:

Poor user experience as integrating components may attempt to mint and fail unexpectedly because their balance is enough to mint at 0 cost.

9.4.4 [L4] - Rounding error in staking transfers

Location: MACROStaking contract

Description:

MACRO transfers round down due to units-to-index conversion, creating a small insolvency where the sum of all user balances is slightly below the actual MACRO balance.

Impact:

The last staker to unstake won't be able to unstake his full balance, but only what remains in the contract balance which will be slightly lower.

Fix:

When staking, check the balance increase of the Staking contract, to increase the staked balance only by that amount (as if it was a fee-on-transfer token).

Response:

The team will always leave some staked tokens, so this situation should never happen.

9.4.5 [L5] - If no tickets are minted in the first epoch, the raffle settlement will revert with Division by zero error and halt the raffle

Location: RaffleSettle contract

Description:

If no tickets exist when `fulfillRandomWords` is called, the function will revert with division by zero. This can happen if no tickets are minted in the first epoch after deployment.

Impact:

Contract halt if the first raffle is triggered without any tickets, requiring redeployment.

Fix:

Make sure at least one ticket is minted before the first epoch ends:

- Mint a permaticket
- Transfer MACRO to an address (normal, debasing address)

Response:

Acknowledged, as the team will for sure mint tokens to non-debasing addresses right after deployment.

9.4.6 [L6] - Imprecise percentage system doesn't allow decimals in percentages

Location: RaffleSettle contract

Description:

The current percentage system doesn't allow decimal precision, using whole numbers only (100 = 100%). Consider using basis points for more precise percentages.

Code:

```
uint256 public constant PERCENT_DENOM = 100;
// Should consider using BASIS_POINTS = 10000 for more precision
```

Impact:

Limited precision in percentage calculations for raffle distributions.

Response

Fixed in a later commit than the mitigation review: 85382acf2ed24b1a484db4f90c175f4ca00b7c57

9.4.7 [L7] - O(n) gas cost for token transfers can reach block gas limit for large transfers

Location: MacroMillions contract

Description:

Every token transfer has a gas cost of O(n), where n is the number of tickets the sender has. Larger transfer amounts result in higher gas costs.

Code:

```
// This loop executes for each ticket being removed
for (uint256 i; i < newDeadTickets; i++) {
    // Ticket removal logic
}
```

Impact:

High gas costs for users with many tickets. Transferring very large amounts, which requires updating many tickets may reach the block-gas-limits and revert.

It could be bypassed by transferring smaller amounts, but at a very high gas cost anyways.

9.4.8 [L8] - Unbounded gas for-loop to handle non-debasing addresses could reach block gas limit

Location: MacroMillions contract

Description:

The debasing mechanism iterates through all non-debasing addresses in a loop, to increase their units, so that their final supply remains constant on every `debase()`. This is an unbounded for-loop, that iterates over all `nonDebasingAddresses`. If the array is long enough, this could potentially reach the block gas limit and halt the contract as debases would revert with out-of-gas errors.

Code:

```
// This loop processes all non-debasing addresses. Gas consuming.
uint256 totalAdditionalUnits;
for (uint256 i; i < nonDebasingAddresses.length; ++i) {
    address nonDebasingAddress = *nonDebasingAddresses*[i];
    // ... processing logic
}
```

Impact:

Low, as the intention is to have very few addresses as non-debasing, but it is important to point it out.

9.4.9 [L9] - Debasing mechanism uses compound interest instead of simple interest and the supply does not decrease by 0.125% every epoch

Location: MacroMillions contract

Description:

The debasing mechanism doesn't decrease the supply by a constant rate of 0.125% per epoch as intended. Instead, it applies compound interest, causing the debasing rate to decrease over time. After 1000 epochs (250 days), the actual supply will be roughly 2x higher than expected.

Code:

```
    uint256 previousDebaseIndex = debaseIndex;
    debaseIndex = debaseIndex * DEBASE_DENOM / (DEBASE_DENOM + debaseRate);
```

Impact:

The debasing rate is not constant but decreasing over time. If there's a target supply on a target date, this mechanism may not reach it.

Fix:

```
-   debaseIndex = debaseIndex * DEBASE_DENOM / (DEBASE_DENOM + debaseRate);
+   debaseIndex = debaseIndex * (debaseRate + DEBASE_DENOM) / DEBASE_DENOM
```

9.4.10 [L10] - Lack of slippage protection in token swaps is vulnerable to sandwich attacks

Description: Setting `amountOutMin` to zero in token swap functions removes slippage protection, exposing swaps to MEV sandwich attacks. Without a minimum output constraint, attackers can manipulate prices around the swap, extracting significant value and causing unpredictable losses.

Implications:

- **MacroBurnFees contract (`swapWinnings` function):** The swap from USDC to MACRO for perma ticket minting uses `amountOutMin = 0`, making winnings highly vulnerable to sandwich attacks. Large swaps can lose up to 100% of their value, directly harming protocol participants.

```
uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        totalToSwap,
        0, // No slippage protection
        path,
        address(this),
        block.timestamp
);
```

- **MacroMillions contract (`_swapTokensForUSDC` function):** Fee accumulation swaps also use `amountOutMin = 0`, risking significant MEV extraction. This can reduce the funds available for raffles and disrupt protocol mechanics.

```
uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        tokenAmount,
        0, // No slippage protection - vulnerable to sandwich attacks
        path,
        address(this),
        block.timestamp
);
```

Fix Considerations:

Mitigating this risk typically requires passing a minimum output value from off-chain, but this is challenging when swaps are triggered inside token transfers, as it may break ERC20 compliance. Alternatives include using an on-chain oracle to estimate expected output, though this adds complexity and maintenance overhead.

9.5 Gas Optimization

9.5.1 [G1] - Constant Function Call in Conditional

Location: MacroBurnFees.sol

The code calls `IERC20Metadata(USDC).decimals()` to get USDC decimals, but since USDC is a constant with known decimals (6), this should be hardcoded.

```
// Current implementation
if (totalToSwap > 5_000 * 10 ** IERC20Metadata(USDC).decimals()) {

// Recommended optimization
if (totalToSwap > 5_000_000_000) { // 5,000 * 10^6 (USDC has 6 decimals)
```

Impact: Eliminates external contract call overhead for a known constant value.

9.5.2 [G2] - Storage Variable Access in Loop

Location: MacroBurnFees.sol

The `permaTicketsMinted` storage variable is read on every loop iteration instead of being cached.

```
// Current implementation
for(uint i; i < _amount; i++) {
    uint256 modulo = (permaTicketsMinted + i) % 10; // Reads storage each time
}

// Recommended optimization
uint256 cachedMinted = permaTicketsMinted;
for(uint i; i < _amount; i++) {
    uint256 modulo = (cachedMinted + i) % 10; // Uses cached value
}
```

Impact: Minor gas savings per iteration, though less significant on Layer 2 networks like Base.

9.5.3 [G3] - Unnecessary Full Struct Loading in Binary Search

Location: MacroMillions.sol and PermaTicketNFT.sol

Binary search functions load entire checkpoint structs into memory when only the timestamp field is needed for comparison.

```
// Current implementation
TicketCheckpoint memory cp = ticketCheckpoints[_ticketNum][center];
if (cp.fromTimestamp == _timestamp) {
    // Only using timestamp field
}

// Recommended optimization
TicketCheckpoint storage cp = ticketCheckpoints[_ticketNum][center];
if (cp.fromTimestamp == _timestamp) {
    // Read only what's needed from storage
}
```

Impact: Reduces memory allocation and copying costs during checkpoint lookups.

9.5.4 [G4] - Inefficient Array Length Access in Loop

Location: MacroMillions.sol

The debase function reads `nonDebasingAddresses.length` from storage on every loop iteration instead of caching it in memory.

```
// Current implementation
for (uint256 i; i < nonDebasingAddresses.length; ++i) {
    // Loop body reads length from storage each time
}

// Recommended optimization
uint256 length = nonDebasingAddresses.length;
for (uint256 i; i < length; ++i) {
    // Loop uses cached length
}
```

Impact: Saves 2,100 gas per iteration by avoiding repeated storage reads.

9.5.5 [G5] - Redundant Field in UserTickets Struct

Location: MacroMillions.sol

The `UserTickets` struct contains a `numTickets` field that duplicates information already available through `ticketIds.length`. This redundant storage increases gas costs for every token transfer.

```
// Current implementation
struct UserTickets {
    uint256 numTickets;    // Redundant field
    uint256[] ticketIds;   // Already contains the count
}

// Recommended optimization
mapping(address => uint256[]) public userTickets; // Direct array mapping
```

Impact: Eliminates unnecessary storage writes during transfers, reducing gas costs significantly for frequent operations.

9.5.6 [G6] - Unnecessary Storage Variable for Debase Rate

Location: MacroMillions.sol

The `debaseRate` variable is set once in the constructor to 125 (representing 0.125%) and never updated afterward. Since this value is immutable, it should be declared as a `constant` to save gas on every read operation.

```
// Current implementation
uint256 public debaseRate; // initialized in constructor, never updated

// Recommended optimization
uint256 public constant DEBASE_RATE = 125; // 0.125%
```

Impact: Each storage read costs 2,100 gas. Using a constant reduces this to a simple code reference, saving gas on every debase operation.

9.6 Informational

9.6.1 [I1] - Unused linkAddress Variable

Contract: RaffleSettle.sol

The `linkAddress` variable is defined but never used in the contract.

```
//// @audit-info linkAddress is unused here
address public linkAddress = 0x88Fb150BDc53A65fe94Dea0c9BA0a6dAf8C6e196;
```

Impact: Unnecessary storage that serves no purpose in the contract's functionality.

9.6.2 [I2] - Unnecessary Receive Function

Contract: RaffleSettle.sol

The contract includes a receive function for Ether, but the contract doesn't manage Ether (it uses LINK tokens for randomness payments).

```
// @audit-info no need for this function as there is no ether management in this contract
//   the randomness is payed with LINK token.
// - impact: ether stuck if sent, as there is no way to get it back
receive() external payable {}
```

Impact: Could result in stuck Ether if accidentally sent to the contract, as there's no withdrawal mechanism.

9.6.3 [I3] - Redundant currentRaffle Storage

Contract: RaffleSettle.sol

The `currentRaffle` variable duplicates information already available in `raffles[raffleId]`.

```
//// @audit-info raffles[raffleId] carries the same information as currentRaffle, so the latter could be
↪  removed
mapping(uint256 => Raffle) public raffles;
```

Impact: Redundant storage that increases gas costs and contract complexity.

9.6.4 [I4] - Unnecessary Ceiling Cost Check

Contract: PermaTicketNFT.sol

The ceiling cost check in `currentMintCost()` is unnecessary since the function only decreases prices and other functions ensure prices never exceed the ceiling.

```
// @audit-info This ceiling check is unnecessary, as the currentStoredCost can never be higher than
↪  ceilingCost when reaching this point.
// this function only makes the price decay and does not increase the price.
// The function increasing the price is mintCostMultiple(), which already makes sure to never return a
↪  price higher than ceilingCost.
// So this check could be removed
if (trueCost > ceilingCost) return ceilingCost;
```

Impact: Unnecessary computation that wastes gas during price calculations.

9.6.5 [I5] - Unused totalBurnedForPermaTickets Variable

Contract: PermaTicketNFT.sol

The `totalBurnedForPermaTickets` variable is never read and could be derived off-chain using indexers.

```
//// @audit-info totalBurnedForPermaTickets is never read, and could be derived offchain with subgraph
↪  or other indexer
uint256 public totalBurnedForPermaTickets;
```

Impact: Unnecessary storage operations that increase gas costs without providing on-chain utility.

9.6.6 [I6] - Potential insufficient swap amount can cause reverts in `mintPermaTicket()`

Contract: MacroBurnFees.sol

If the mint cost exceeds $5000 worth of MACRO, the swap might not provide enough tokens, causing `mintPermaTicket` to revert.

```
// @audit-info if mintCost > 5000$ worth of MACRO, perhaps not enough is swapped.
// - impact: mintPermaTicket will revert, but it can be solved by calling swapWinnings() before
↪  mintPermaTicket()
// - severity: low as the probability of such a high price is low
// - fix: no fix needed. If this happens, the swapWinnings() function should be called before
↪  mintPermaTicket() to ensure enough MACRO is available
```

Impact: Low severity issue that can be resolved by calling `swapWinnings()` before `mintPermaTicket()` if MACRO price increases significantly.

9.6.7 [I7] - Redundant massApproval Function

Contract: MacroBurnFees.sol

The `massApproval()` function duplicates approvals already done in the constructor.

```
// @audit-info This function seems unnecessary as the mass approval is already done in the constructor
IERC20(USDC).approve(address(uniswapV2Router), type(uint256).max);
```

Impact: Provides no additional functionality since approvals are already handled in the constructor.

9.6.8 [I8] - Immutable Variable Opportunity

Contract: MacroBurnFees.sol

Multiple contracts have `MACRO` as a storage variable that can only be set once. These could be immutable variables set in constructors.

```
/// @audit-info Multiple contracts have MACRO as a storage variable that can only be set once. Consider
↪  having these ones as immutable variables, and passing the MACROS address in the constructors of all
↪  other contracts
require(MACRO == address(0), "Address already set");
MACRO = _MACRO;
```

Impact: Using immutable variables would reduce gas costs for reads and remove the need for setter functions.

9.6.9 [I9] - Unused permaModuloToTeam Constant

Contract: MacroBurnFees.sol

The `permaModuloToTeam` constant is defined but never used in the contract.

```
//// @audit-info The permaModuloToTeam is unused and can be removed
uint256 public constant permaModuloToTeam = 9;
```

Impact: Increases contract size without providing functionality.

9.6.10 [I10] - Missing Events in Setter Functions

Contract: MacroMillions.sol

Several state-changing setter functions lack event emissions for important state changes.

```
// @audit-info missing event in multiple state changing functions like setters
uint256 oldFreq = debaseFreq;
debaseFreq = _freq;
emit DebaseFreqUpdated(oldFreq, _freq);
```

Impact: Reduced transparency and monitoring capabilities for off-chain systems tracking contract state changes.

9.6.11 [I11] - Unsafe usage of ERC20 transfers ignoring outputs

Multiple contracts perform ERC20 transfers ignoring the return value.

```
// @audit-info ERC20 token transfers ignore output. Using safeTransfer library is recommended.
IERC20(USDC).transfer(raffleSplitter, IERC20(USDC).balanceOf(address(this)));
```

Impact: If the transfer fails, the transaction will not revert, potentially leading to inconsistent state.

Fix: Use SafeErc20 library for ERC20 token transfers

9.6.12 [I12] - Unnecessary checkpoint update logic when minting new tickets

Contract: MacroMillions.sol, in `_updateTickets()` function.

If there aren't enough `deadTickets` to mint for the receiver, the function `_updateTickets()` mints the `additionalTickets`. When minting these new tickets, the code includes logic to update existing checkpoints, but new tickets should never have existing checkpoints.

```
   for (uint256 x; x < additionalTickets; x++) {
       uint256 newTicket = nextTicketId;
       nextTicketId++;
       toTickets.numTickets++;
       toTickets.ticketIds.push(newTicket);

-      uint256 nCheckpoints = numTicketCheckpoints[newTicket];
-      if (nCheckpoints > 0 && ticketCheckpoints[newTicket][nCheckpoints - 1].fromTimestamp ==
↪  block.timestamp)
-      {
-          ticketCheckpoints[newTicket][nCheckpoints - 1].owner = _to;
-      } else {
-          ticketCheckpoints[newTicket][nCheckpoints] = TicketCheckpoint(block.timestamp, _to);
-          numTicketCheckpoints[newTicket] = nCheckpoints + 1;
-      }

+      ticketCheckpoints[newTicket][nCheckpoints] = TicketCheckpoint(block.timestamp, _to);
+      numTicketCheckpoints[newTicket] = nCheckpoints + 1;
   }
```

9.6.13 [I13] - Unreachable zero address check can be removed

Contract: MacroMillions.sol

The `burn()` function includes a zero address check that can never be triggered since `burn()` is called with `msg.sender` as input parameter. The requirement can be removed.

```
// @audit-info this can never happen, because `burn()` calls with `msg.sender` as account. So this
↪  requirement can be removed.
require(account \!= address(0), "ERC20: burn from the zero address");
```

Impact: Unnecessary gas consumption for a check that can never fail.

9.6.14 [I14] - Precision Loss in Non-Debasing Address Updates

Contract: MacroMillions.sol

During debase operations, non-debasing addresses undergo double rounding which may result in 1 wei precision loss.

```
// @audit-info this is rounded down twice, so the end result is likely 1wei less than the original. It
↪  would only be relevant if we had to calculate tickets for theses tokens
uint256 balance = 1e18 * previousUnits / previousDebaseIndex;
uint256 newUnits = unitsForBalance(balance);
```

Impact: While minimal (1 wei), this precision loss could accumulate over time for non-debasing addresses, though it's not relevant for ticket calculations.

9.6.15 [I15] - Redundant and unused data structures and variables can be removed

Contracts: MacroMillions.sol, MacroBurnFees.sol, PermaTicketNFT.sol, RaffleSettle.sol

Several contracts maintain variables or data structures that are either redundant, duplicate information, or are never read/used:

- `permaTicketArray` (MacroMillions.sol) - never read anywhere
- `isPermaTicket` mapping (MacroMillions.sol) - duplicates information in `permaTickets`
- `tokensForRaffle` (MacroMillions.sol) - calculated but never used
- `permaModuloToTeam` (MacroBurnFees.sol) - unused constant
- `totalBurnedForPermaTickets` (PermaTicketNFT.sol) - never read, can be derived off-chain
- `linkAddress` (RaffleSettle.sol) - unused variable

Additionally, some state variables duplicate information already available elsewhere:

- `currentDebaseEpoch` (MacroMillions.sol) - duplicates `debaseEpochs[debaseId]`
- `currentRaffle` (RaffleSettle.sol) - duplicates `raffles[raffleId]`

Impact: Unused and redundant variables/data structures increase contract size, gas costs, and complexity without providing additional functionality. Removing them will optimize the protocol and improve maintainability.