ZEALYNX SECURITY

Web3 Security & Smart Contract Development

MONADEX

Security Assessment

September 18th, 2024 – Prepared by Zealynx Security

ZealynxSecurity@protonmail.com

Sergio	@Seecoalba
Bloqarl	@TheBlockChainer
Cholakov	@cholakovv
00xWizard	@00xWizard

Contents

1. About Zealynx

Zealynx Security, founded in January 2024 by Bloqarl and Sergio, specializes in smart contract audits, development, and security testing using Solidity and Rust. Our services include comprehensive smart contract audits, smart contract fuzzing, and formal verification test suites to enhance security reviews and ensure robust contract functionality. We are trusted by clients such as Glif, Shieldify, AuditOne, Bastion Wallet, Side.xyz, Possum Labs, and Aurora (NEAR Protocol-based).

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our blog, Bloqarl's blog, and Sergio's blog.

Zealynx has achieved public recognition, including a Top 5 position in the Beanstalk Audit public contest, rewarded with $8k. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can’t guarantee there are no vulnerabilities. It’s a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

A time-boxed independent security assessment of the Monadex protocol was done by Zealynx Security, with a focus on the security aspects of the application's implementation. We performed the security assessment based on the agreed scope, following our approach and methodology. Based on our scope and our performed activities, our security assessment revealed **2 Critical**, **2 High**, **6 Medium**, and **6 Low** severity security issues.

3.2 About Monadex

Website : Monadex
Docs: Monadex.gitbook

Monadex is an ecosystem-focused and community-driven decentralized exchange built on Monad. With the goal of making the DEX experience fun again on a super-fast chain, Monadex introduces a set of cutting-edge features that cater to the interests of both mature traders and degens alike.

Monadex Labs is building the next liquidity hub on Monad by offering its users a seamless trading experience, attractive LP yield, and new homemade features to propel the user experience to the next level, all while enjoying the benefits of Monad.

3.3 Audit Scope

The code under review is composed of multiple smart contracts written in Solidity language and includes 1021 nSLOC- normalized source lines of code. The core contracts on the scope are the following:

- MonadexV1Router
- MonadexV1RafflePriceCalculator
- MonadexV1Entropy
- MonadexV1Library
- MonadexV1Types
- MonadexV1Pool

4. Audit Methodology:

Approach

During our security assessments, we uphold a rigorous approach to maintain high-quality standards. Our methodology encompasses thorough **Fuzz** and **Invariant Testing** and meticulous **manual security review**.

Throughout the smart contract audit process, we prioritize the following aspects to uphold excellence:

1. **Code Quality:** We diligently evaluate the overall quality of the code, aiming to identify any potential vulnerabilities or weaknesses.
2. **Best Practices:** Our assessments emphasize adherence to established best practices, ensuring that the smart contract follows industry-accepted guidelines and standards.
3. **Documentation and Comments:** We meticulously review code documentation and comments to ensure they accurately reflect the underlying logic and expected behavior of the contract.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

Over the course of 2 weeks, Monadex team engaged with Zealynx Security to review their smart contracts. In this period of time, a total of **16** issues were found.

Summary of Findings

Vulnerability	Severity
[C-1] Nearly impossible for any user to register to the raffle	Critical
[C-2] Incorrect Confidence Calculation in Ticket Minting	Critical
[H-1] Multiple decimal precision issues in ticket calculation	High
[H-2] Missing Price Update Leads to User Being Unable to Receive Tickets	High
[M-1] Denial of Service Risk due to Unbounded Loop in Raffle Registration	Medium
[M-2] Locked funds on Raffle contract when sending higher value than the fee on requestRandomNumber	Medium
[M-3] Potential removal of unclaimed winning tokens	Medium
[M-4] Token Removal During Ongoing Raffle	Medium
[M-5] Incorrect ticket calculation for Tokens with fee-on-transfer	Medium
[M-6] Incorrect use of confidence value gets users less tickets	Medium
[L-1] Single-Step Ownership Transfer used in contracts	Low
[L-2] Unable to modify _minimumParticipants, s_multipliersToPercentages and s_winningPortions	Low
[L-3] Permit doesn't follow OpenZeppelin's recommendation to handle front runs	Low
[L-4] Supported Tokens should be limited to 5 or 6 tops to avoid high gas consumption	Low
[L-5] Unable to Remove Tokens with Non-Zero Balance	Low
[L-6] User uninformed of unused tickets after registering for raffle	Low

7. Findings

7.1. Critical Severity Findings

[C-1] Nearly impossible for any user to register to the raffle

Description

The *register()* function in the MonadexV1Raffle contract contains a critical logic error that prevents any user from successfully registering for the raffle. The function incorrectly checks the user's balance against the number of tickets to burn, causing it to revert in all scenarios.

- If ticketsToBurn < balance, the function reverts, preventing registration even when the user has sufficient balance.
- If ticketsToBurn >= balance, the function will likely revert in the _burn call due to insufficient balance.

Impact

This vulnerability completely breaks the core functionality of the raffle system, making it impossible for any user to register. As a result, the entire contract becomes non-functional, effectively creating a denial of service for the raffle feature. If deployed, this critical flaw would likely lead to a significant loss of user trust and damage to the protocol's reputation, as users would be unable to participate in the raffle despite potentially having purchased tickets.

Mitigation

Correct the balance check logic:

```
function register(uint256 _amount) external notZero(_amount) returns (uint256) {
// ... (previous checks)
    uint256 balance = balanceOf(msg.sender);
    uint256 ticketsToBurn = slotsToOccupy * RANGE_SIZE;
    if (ticketsToBurn > balance) {
        revert MonadexV1Raffle__NotEnoughBalance(ticketsToBurn, balance);
    }
// ... (rest of the function)
}
```

[C-2] Incorrect Confidence Calculation in Ticket Minting

Description

In the *calculateTicketsToMint* function of the MonadexV1Library contract, there is a critical error in the calculation of the confidence value. The function is using *_pythPrice.price* instead of *_pythPrice.conf* for the confidence calculation. This mistake leads to an incorrect subtraction of the confidence from the price, resulting in zero or negative ticket amounts.

Incorrect code:

```
uint256 confidence = _pythPrice.expo < 0
    ? uint256(uint64(_pythPrice.price)) * decimals / 10 ** uint256(uint32(-1 * _
pythPrice.expo))
    : uint256(uint64(_pythPrice.price)) * decimals * 10 ** uint256(uint32(-1 * _
pythPrice.expo))
```

The current implementation:

```
uint256 ticketsToMint = (price - confidence) * _amount / _pricePerTicket;
```

Impact

The function will always calculate zero or negative tickets to mint, causing all ticket purchase transactions to revert with *MonadexV1Raffle__ZeroTickets* error. The raffle system becomes non-functional as no user can purchase tickets.

Mitigation

Correct code should use *_pythPrice.conf*:

```
uint256 confidence = _pythPrice.expo < 0
    ? uint256(uint64(_pythPrice.conf)) * decimals / 10 ** uint256(uint32(-1 * _p
ythPrice.expo))
    : uint256(uint64(_pythPrice.conf)) * decimals * 10 ** uint256(uint32(-1 * _p
ythPrice.expo));
```

7.2. High Severity Findings

[H-1] Multiple decimal precision issues in ticket calculation

Description

The *calculateTicketsToMint* function in the MonadexV1Library contract contains multiple critical issues related to decimal precision handling. These issues stem from incorrect assumptions about token decimals, price representations, and Pyth oracle data formatting.

Key issues identified:

1. Hardcoded 18 decimal places assumption for all tokens.
2. Incorrect representation of the ticket price without proper decimal scaling.
3. Mishandling of Pyth oracle data, which may not always be in 18 decimal precision.

Impact

Users could receive orders of magnitude more or fewer tickets than intended, especially for tokens with decimals other than 18.

The raffle system's economy could be completely destabilized, with some users gaining a massive unfair advantage. If more tickets are minted than intended, the prize pool could be drained much faster than designed.

Mitigation

- Modify the function to accept and use the specific decimal places of each token:

```
function calculateTicketsToMint(
    uint256 _amount,
    PythStructs.Price memory _pythPrice,
    uint256 _pricePerTicket,
    uint8 _tokenDecimals
) internal pure returns (uint256) {
    uint256 adjustedAmount = _amount * 10**(18 - _tokenDecimals);
// ... rest of the calculation
}
```

- Ensure the PRICE_PER_TICKET constant is correctly scaled:

```
uint256 internal constant PRICE_PER_TICKET = 1e18;// $1 with 18 de
cimal places
```

- Use Pyth's utility functions to correctly interpret price data:

```
import "@pythnetwork/pyth-sdk-solidity/PythUtils.sol";

uint256 price = PythUtils.convertToUint(_pythPrice.price, _pythPri
ce.expo, 18);
uint256 confidence = PythUtils.convertToUint(_pythPrice.conf, _pyt
hPrice.expo, 18);
```

[H-2] Missing Price Update Leads to User Being Unable to Receive Tickets

Description

The _getTicketsToMint function in the contract does not call updatePriceFeeds() before retrieving the price data from the Pyth oracle using getPriceNoOlderThan(). This function fetches the latest available price within a specified time window, but without updating the price feed beforehand, the contract may be using stale data. As a result, the process of receiving tickets after a swap fails.

According to the *Pyth official documentation*, it is recommended to call updatePriceFeeds() before using price data to ensure it is fresh and up-to-date.

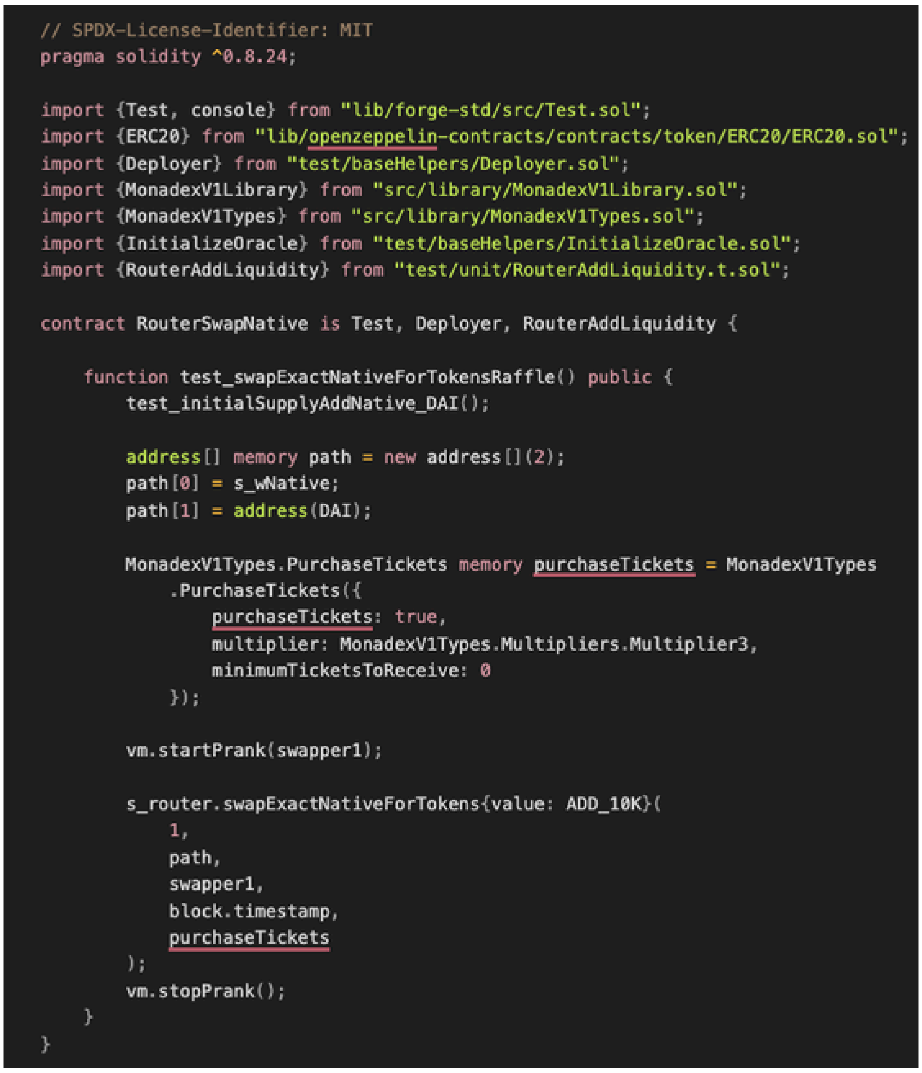
```
PythStructs.Price memory price = IPyth(i_pyth).getPriceNoOlderThan(
            config.priceFeedId,
            config.noOlderThan);
```

Impact

Not calling updatePriceFeeds() causes the function to always revert, making users unable to receive tickets.

Proof of Concept

Create a new file and add the following test:



```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.24;

import {Test, console} from "lib/forge-std/src/Test.sol";
import {ERC20} from "lib/openzeppelin-contracts/contracts/token/ERC20/ERC20.sol";
import {Deployer} from "test/baseHelpers/Deployer.sol";
import {MonadexV1Library} from "src/library/MonadexV1Library.sol";
import {MonadexV1Types} from "src/library/MonadexV1Types.sol";
import {InitializeOracle} from "test/baseHelpers/InitializeOracle.sol";
import {RouterAddLiquidity} from "test/unit/RouterAddLiquidity.t.sol";

contract RouterSwapNative is Test, Deployer, RouterAddLiquidity {

    function test_swapExactNativeForTokensRaffle() public {
        test_initialSupplyAddNative_DAI();

        address[] memory path = new address[](2);
        path[0] = s_wNative;
        path[1] = address(DAI);

        MonadexV1Types.PurchaseTickets memory purchaseTickets = MonadexV1Types
            .PurchaseTickets({
                purchaseTickets: true,
                multiplier: MonadexV1Types.Multipliers.Multiplier3,
                minimumTicketsToReceive: 0
            });

        vm.startPrank(swapper1);

        s_router.swapExactNativeForTokens{value: ADD_10K}(
            1,
            path,
            swapper1,
            block.timestamp,
            purchaseTickets
        );
        vm.stopPrank();
    }
}
```

Run the test with the following command:

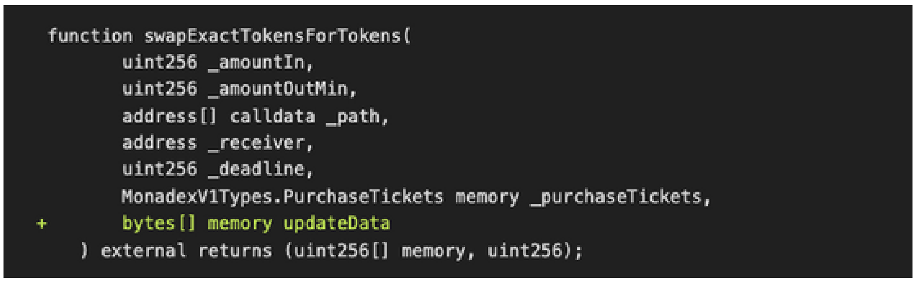
forge test --mt test_raffle -vvvv

Mitigation

To mitigate this issue, ensure that the price feed is updated before retrieving the price from the oracle. This can be achieved by calling updatePriceFeeds() with the appropriate price update data before retrieving the price. Keep in mind that this updateData should be obtained off-chain using the Hermes API as recommended by the Pyth official documentation and passed as an argument to the function.

Here's an example of how to implement this:

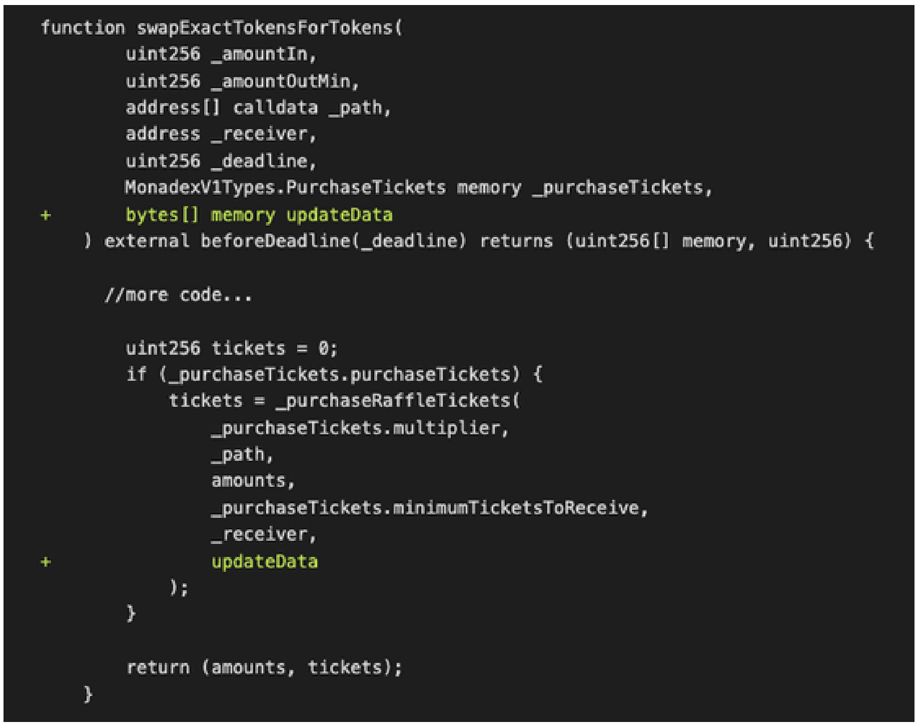
1. Update the swap functions in the IMonadexV1Router interface to have an extra updateData parameter:



```
 function swapExactTokensForTokens(
        uint256 _amountIn,
        uint256 _amountOutMin,
        address[] calldata _path,
        address _receiver,
        uint256 _deadline,
        MonadexV1Types.PurchaseTickets memory _purchaseTickets,
+       bytes[] memory updateData
    ) external returns (uint256[] memory, uint256);
```

Do the same thing with the other swap functions

2. Add the extra updateData parameter to the swap functions in the MonadexV1Router contract as well and call _purchaseRaffleTickets with the updateData parameter:



```
function swapExactTokensForTokens(
        uint256 _amountIn,
        uint256 _amountOutMin,
        address[] calldata _path,
        address _receiver,
        uint256 _deadline,
        MonadexV1Types.PurchaseTickets memory _purchaseTickets,
+       bytes[] memory updateData
    ) external beforeDeadline(_deadline) returns (uint256[] memory, uint256) {

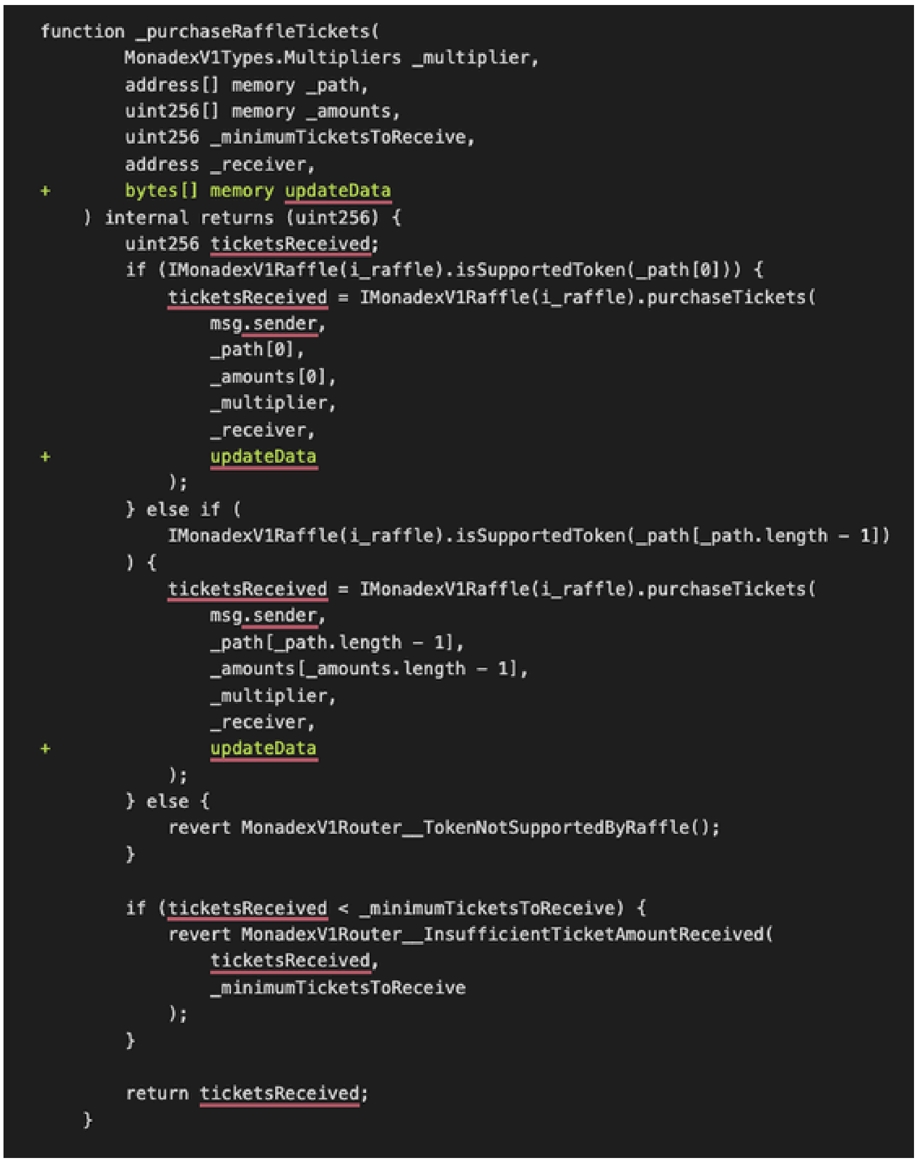
      //more code...

        uint256 tickets = 0;
        if (_purchaseTickets.purchaseTickets) {
            tickets = _purchaseRaffleTickets(
                _purchaseTickets.multiplier,
                _path,
                amounts,
                _purchaseTickets.minimumTicketsToReceive,
                _receiver,
+               updateData
            );
        }

        return (amounts, tickets);
    }
```

Do the same thing with the other swap functions

3. Add the extra *updateData* parameter to the *_purchaseRaffleTickets* function and call the purchaseTickets from the MonadexV1Raffle contract with the *updateData* parameter:

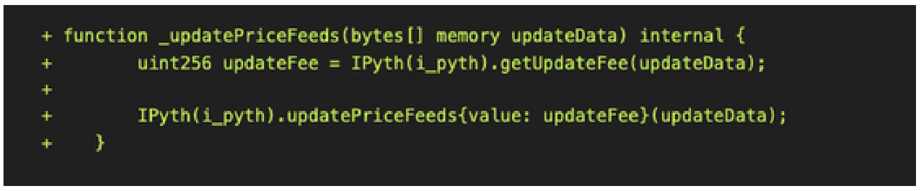


```
function _purchaseRaffleTickets(
        MonadexV1Types.Multipliers _multiplier,
        address[] memory _path,
        uint256[] memory _amounts,
        uint256 _minimumTicketsToReceive,
        address _receiver,
+       bytes[] memory updateData
    ) internal returns (uint256) {
        uint256 ticketsReceived;
        if (IMonadexV1Raffle(i_raffle).isSupportedToken(_path[0])) {
            ticketsReceived = IMonadexV1Raffle(i_raffle).purchaseTickets(
                msg.sender,
                _path[0],
                _amounts[0],
                _multiplier,
                _receiver,
+               updateData
            );
        } else if (
            IMonadexV1Raffle(i_raffle).isSupportedToken(_path[_path.length - 1])
        ) {
            ticketsReceived = IMonadexV1Raffle(i_raffle).purchaseTickets(
                msg.sender,
                _path[_path.length - 1],
                _amounts[_amounts.length - 1],
                _multiplier,
                _receiver,
+               updateData
            );
        } else {
            revert MonadexV1Router__TokenNotSupportedByRaffle();
        }

        if (ticketsReceived < _minimumTicketsToReceive) {
            revert MonadexV1Router__InsufficientTicketAmountReceived(
                ticketsReceived,
                _minimumTicketsToReceive
            );
        }

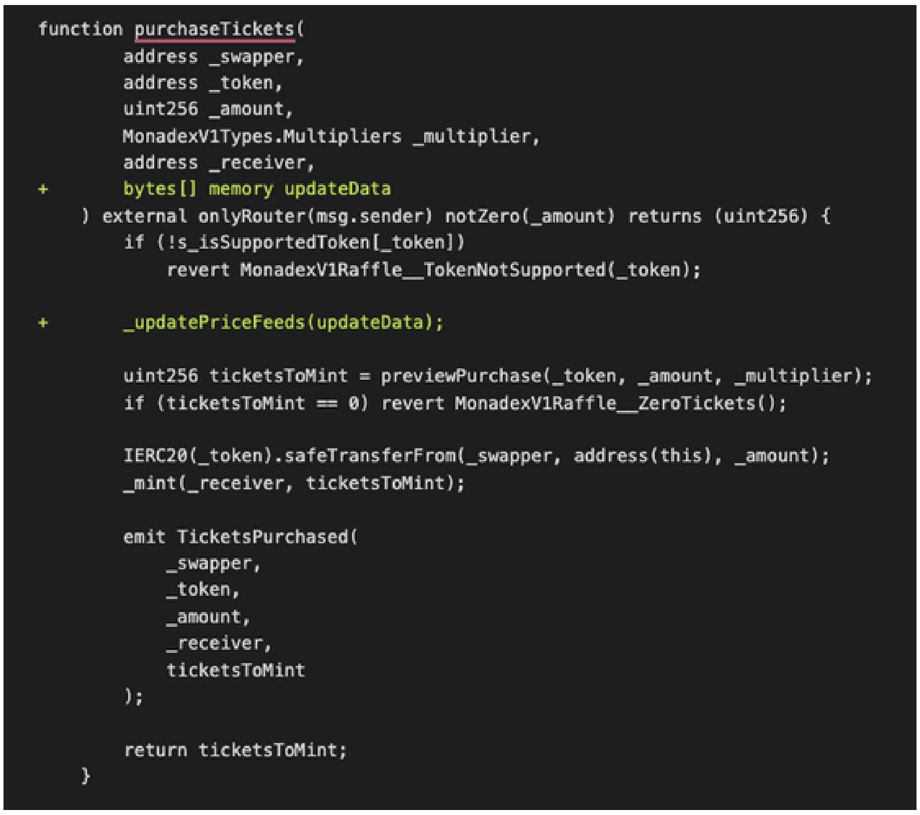
        return ticketsReceived;
    }
```

4. Create an *_updatePriceFeeds* function in the MonadexV1Raffle contract:



```
+ function _updatePriceFeeds(bytes[] memory updateData) internal {
+       uint256 updateFee = IPyth(i_pyth).getUpdateFee(updateData);
+
+       IPyth(i_pyth).updatePriceFeeds{value: updateFee}(updateData);
+   }
```

5. Add the extra *updateData* parameter to the *purchaseTickets* function and call *_updatePriceFeeds* before calling *previewPurchase*, which calls *_getTicketsToMint*, where the *getPriceNoOlderThan* function is called:



```
function purchaseTickets(
        address _swapper,
        address _token,
        uint256 _amount,
        MonadexV1Types.Multipliers _multiplier,
        address _receiver,
+       bytes[] memory updateData
    ) external onlyRouter(msg.sender) notZero(_amount) returns (uint256) {
        if (!s_isSupportedToken[_token])
            revert MonadexV1Raffle__TokenNotSupported(_token);

+       _updatePriceFeeds(updateData);

        uint256 ticketsToMint = previewPurchase(_token, _amount, _multiplier);
        if (ticketsToMint == 0) revert MonadexV1Raffle__ZeroTickets();

        IERC20(_token).safeTransferFrom(_swapper, address(this), _amount);
        _mint(_receiver, ticketsToMint);

        emit TicketsPurchased(
            _swapper,
            _token,
            _amount,
            _receiver,
            ticketsToMint
        );

        return ticketsToMint;
    }
```

7.3. Medium Severity Findings

[M-1] Denial of Service Risk due to Unbounded Loop in Raffle Registration

Description

The register function in the MonadexV1Raffle contract contains an unbounded loop that poses a Denial of Service (DoS) risk.

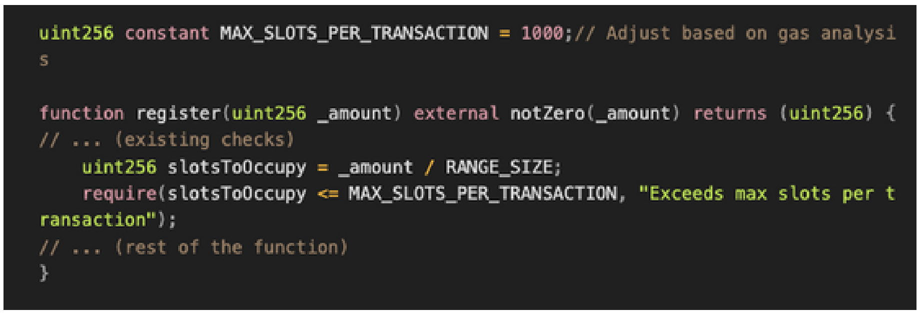
The number of loop iterations is directly controlled by the user input _amount, without an upper bound. This could allow a malicious user or even a legitimate user to force the function to exceed the block gas limit, causing a Denial of Service.

Impact

1. Denial of Service: An user can submit a transaction with a very large _amount, causing the function to consume excessive gas and potentially block the execution which will end up reverting.
2. Contract Lockup: If the registration period is time-sensitive, this vulnerability could prevent legitimate users from registering before the deadline.
3. Economic Damage: Failed transactions due to out-of-gas errors can result in lost gas fees for users attempting to register.

Mitigation

Implement a maximum limit on the number of slots that can be registered in a single transaction:



```
uint256 constant MAX_SLOTS_PER_TRANSACTION = 1000;// Adjust based on gas analysi
s

function register(uint256 _amount) external notZero(_amount) returns (uint256) {
// ... (existing checks)
    uint256 slotsToOccupy = _amount / RANGE_SIZE;
    require(slotsToOccupy <= MAX_SLOTS_PER_TRANSACTION, "Exceeds max slots per t
ransaction");
// ... (rest of the function)
}
```

This mitigation prevents potential DoS attacks by limiting the loop's maximum iterations, ensuring the function's gas consumption remains within acceptable limits.

[M-2] Excess Fees Locked in Raffle Contract

Description

The *requestRandomNumber* function in the MonadexV1Raffle contract accepts payment for requesting a random number but does not provide a mechanism to refund excess fees. Users must calculate the exact fee required, and any excess amount sent will be locked in the contract.

Impact

1. Users may lose funds if they send more ETH than the required fee.
2. Excess funds will accumulate in the contract without a way to retrieve them.

Mitigation

Implement a refund mechanism for excess fees. And add a function that allows users to query the current fee

[M-3] Potential removal of unclaimed winning tokens

Description

The MonadexV1Raffle contract's removeToken function currently lacks a mechanism to handle unclaimed winnings when removing a supported token. This oversight creates a significant vulnrability that could lead to the permanent loss of user funds and undermine the integrity of the raffle system.

In the current implementation, when a token is removed from the supported list, any unclaimed winnings denominated in that token become effectively trapped within the contract. The contract maintains a record of these winnings in the s_winnings mapping, but once the token is no longer supported, users have no way to claim these funds.

Impact

Users with unclaimed winnings in the removed token will lose access to their rightfully earned prizes. This represents a direct financial loss for participants, which could range from negligible amounts to potentially significant sums.

Mitigation

Implement a check for unclaimed winnings before token removal. Or alternatively, implement a migration mechanism for unclaimed winnings

[M-4] Token Removal During Ongoing Raffle

Description

The MonadexV1Raffle contract currently allows the owner to remove supported tokens at any time using the *removeToken* function. This function lacks safeguards to prevent token removal during an active raffle cycle. The ability to alter the set of supported tokens mid-raffle introduces a significant vulnerability that could compromise the integrity and fairness of the entire raffle system.

When a raffle is ongoing, participants have already committed their tokens and received entries based on the current set of supported tokens. Removing a token during this critical period could have far-reaching consequences on the raffle's mechanics, prize distribution, and overall fairness.

Impact

There are a few concerning things:

Firstly, removing a token that participants have already used to enter the raffle could invalidate their entries or alter their chances of winning unfairly. For instance, if a popular entry token is removed, it could disproportionately affect a large number of participants, effectively nullifying their participation without recourse.

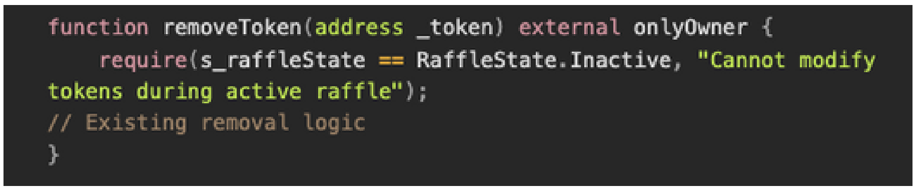
Secondly, the raffle's prize pool and distribution mechanism may be compromised. If the removed token was meant to be part of the prize, winners could find themselves unable to claim their rewards in the promised token, leading to disputes and potential loss of funds.

Moreover, this vulnerability opens the door to potential manipulation. A malicious or compromised owner could strategically remove tokens to influence the outcome of the raffle, favoring certain participants over others. This power to alter the fundamental rules of the raffle mid-game severely undermines the trustworthiness and perceived fairness of the entire system.

Mitigation

Addressing this vulnerability requires implementing strict safeguards and improving the overall architecture of the raffle system. Here's a comprehensive approach to mitigate this issue:

1. Implement a robust raffle state management system. This system should clearly define different stages of the raffle (e.g., Inactive, RegistrationOpen, RegistrationClosed, DrawingWinners, Completed) and enforce strict rules about what actions are allowed in each state.
2. Modify the removeToken function to include a state check:



```
function removeToken(address _token) external onlyOwner {
    require(s_raffleState == RaffleState.Inactive, "Cannot modify
tokens during active raffle");
// Existing removal logic
}
```

[M-5] Incorrect ticket calculation for Tokens with fee-on-transfer

Description

The *purchaseTickets* function in the MonadexV1Raffles contract does not account for tokens that implement a fee-on-transfer mechanism. These tokens deduct a fee from the transfer amount, resulting in the recipient receiving less than the amount specified in the transfer.

The current implementation calculates the number of tickets to mint based on the input *_amount*:

```
function removeToken(address _token) external onlyOwner {
    require(s_raffleState == RaffleState.Inactive, "Cannot modify
tokens during active raffle");
// Existing removal logic
}
```

However, it then transfers the tokens using:

```
IERC20(_token).safeTransferFrom(_swapper, address(this), _amount);
```

For fee-on-transfer tokens, the actual amount received by the contract will be less than *_amount*. This discrepancy leads to an incorrect calculation of tickets, where users receive more tickets than they should based on the actual amount of tokens received by the contract.

Impact

1. Users of fee-on-transfer tokens would receive more raffle tickets than they should, giving them an unfair advantage over users of standard tokens.
2. The raffle system would consistently overvalue fee-on-transfer tokens, potentially leading to economic exploits or imbalances in the protocol.
3. The contract's internal accounting of received tokens versus issued tickets would be inaccurate, potentially causing issues in other parts of the system.

Mitigation

To address this issue, implement a two-step process for purchasing tickets with fee-on-transfer tokens:

- First, transfer the tokens and calculate the actual amount received:

```
uint256 balanceBefore = IERC20(_token).balanceOf(address(this));
IERC20(_token).safeTransferFrom(_swapper, address(this), _amount);
uint256 balanceAfter = IERC20(_token).balanceOf(address(this));
uint256 actualAmount = balanceAfter - balanceBefore;
```

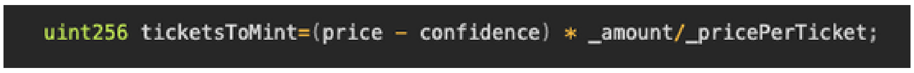
- Then, use the actualAmount to calculate the number of tickets to mint:

```
uint256 ticketsToMint = previewPurchase(_token, actualAmount, _multip
lier);
```

[M-6] Incorrect use of confidence value gets users less tickets

Description

The *calculateTicketsToMint* function in the MonadexV1Library contract incorrectly uses the confidence value from the Pyth oracle in its ticket minting calculation. According to Pyth documentation, the confidence value represents the uncertainty or possible error range in the reported price and should be used to assess the reliability of the price data, not for direct arithmetic operations.



```
uint256 ticketsToMint=(price - confidence) * _amount/_pricePerTicket;
```

This calculation subtracts the confidence from the price, which is not the intended use of the confidence value and leads to inaccurate ticket minting calculations.

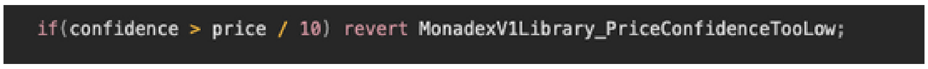
Impact

Users receive fewer tickets than they should for their input amount. In typical market conditions, this could result in 1-10% fewer tickets being minted.

During periods of high market volatility when confidence values are higher, the underminting becomes more severe. In extreme cases, it could result in significantly fewer tickets being minted.

Mitigation

Remove the confidence subtraction from the price calculation and use the confidence value to assess the reliability of the price feed rather than in the calculation itself. For instance:



```
if(confidence > price / 10) revert MonadexV1Library_PriceConfidenceTooLow;
```

Here the code ensures that the confidence value is within an acceptable range (e.g., 10% of the price). If the confidence value is too high, indicating high uncertainty in the price, the transaction will revert.

7.3. Low Severity Findings

[L-1] Single-Step Ownership Transfer used in contracts

Description

The MonadexV1Raffle and MonadexV1Factory contracts currently inherits from OpenZeppelin's *Ownable* contract, which implements a single-step ownership transfer pattern.

While functional, this pattern lacks a confirmation step for the new owner, which could potentially lead to issues if an incorrect address is provided during transfer.

Impact

The contract could become locked, with no address able to execute onlyOwner functions. This would affect:

- The ability to add or remove supported tokens
- Updates to price feed configurations
- Adjustments to raffle parameters or fees

Mitigation

Consider replacing *Ownable* with OpenZeppelin's *Ownable2Step*. This implements a two-step ownership transfer process, adding an extra layer of safety.

[L-2] Inability to modify Raffle parameters from constructor

Description

The MonadexV1Raffle contract lacks setter functions to modify critical parameters *_minimumParticipants*, *s_multipliersToPercentages*, and *s_winningPortions*.

These parameters are set during contract deployment and cannot be altered afterwards, limiting the contract's adaptability to changing requirements or market conditions.

Impact

The contract cannot adapt to changing needs or optimize raffle mechanics over time.

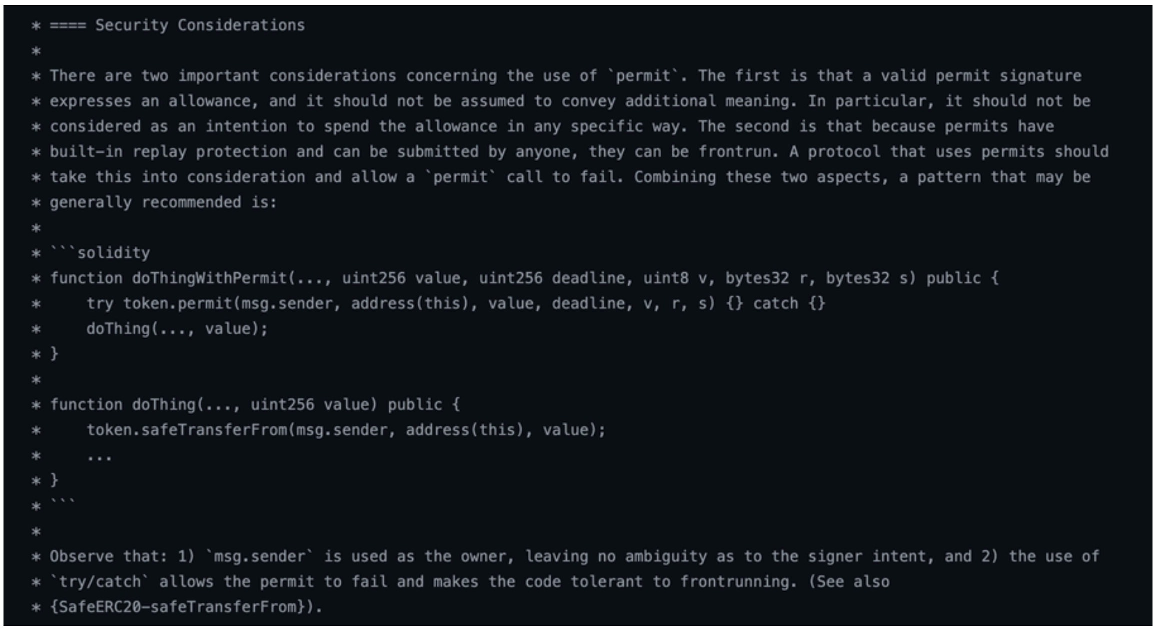
Mitigation

Implement setter functions for these parameters.

[L-3] Permit doesn’t follow OpenZeppelin’s recommendation to handle front runs

Description

The *removeLiquidityWithPermit* and *removeLiquidityNativeWithPermit* functions in the contract use the permit function of ERC20 tokens without proper error handling. According to OpenZeppelin's latest recommendations, permit calls should be wrapped in a try-catch block to handle potential failures gracefully.



```
* ==== Security Considerations
*
* There are two important considerations concerning the use of `permit`. The first is that a valid permit signature
* expresses an allowance, and it should not be assumed to convey additional meaning. In particular, it should not be
* considered as an intention to spend the allowance in any specific way. The second is that because permits have
* built-in replay protection and can be submitted by anyone, they can be frontrun. A protocol that uses permits should
* take this into consideration and allow a `permit` call to fail. Combining these two aspects, a pattern that may be
* generally recommended is:
*
* ```solidity
* function doThingWithPermit(..., uint256 value, uint256 deadline, uint8 v, bytes32 r, bytes32 s) public {
*     try token.permit(msg.sender, address(this), value, deadline, v, r, s) {} catch {}
*     doThing(..., value);
* }
*
* function doThing(..., uint256 value) public {
*     token.safeTransferFrom(msg.sender, address(this), value);
*     ...
* }
* ```
*
* Observe that: 1) `msg.sender` is used as the owner, leaving no ambiguity as to the signer intent, and 2) the use of
* `try/catch` allows the permit to fail and makes the code tolerant to frontrunning. (See also
* {SafeERC20-safeTransferFrom}).
```

Current implementation:

```
IERC20Permit(pool).permit(msg.sender, address(this), value, _params.deadline, _p
arams.v, _params.r, _params.s);
```

This implementation doesn't account for potential failures of the permit function, which could lead to transaction reverts in cases where the permit has already been used or in front-running scenarios.

Impact

The current implementation may cause unnecessary transaction failures in the following scenarios:

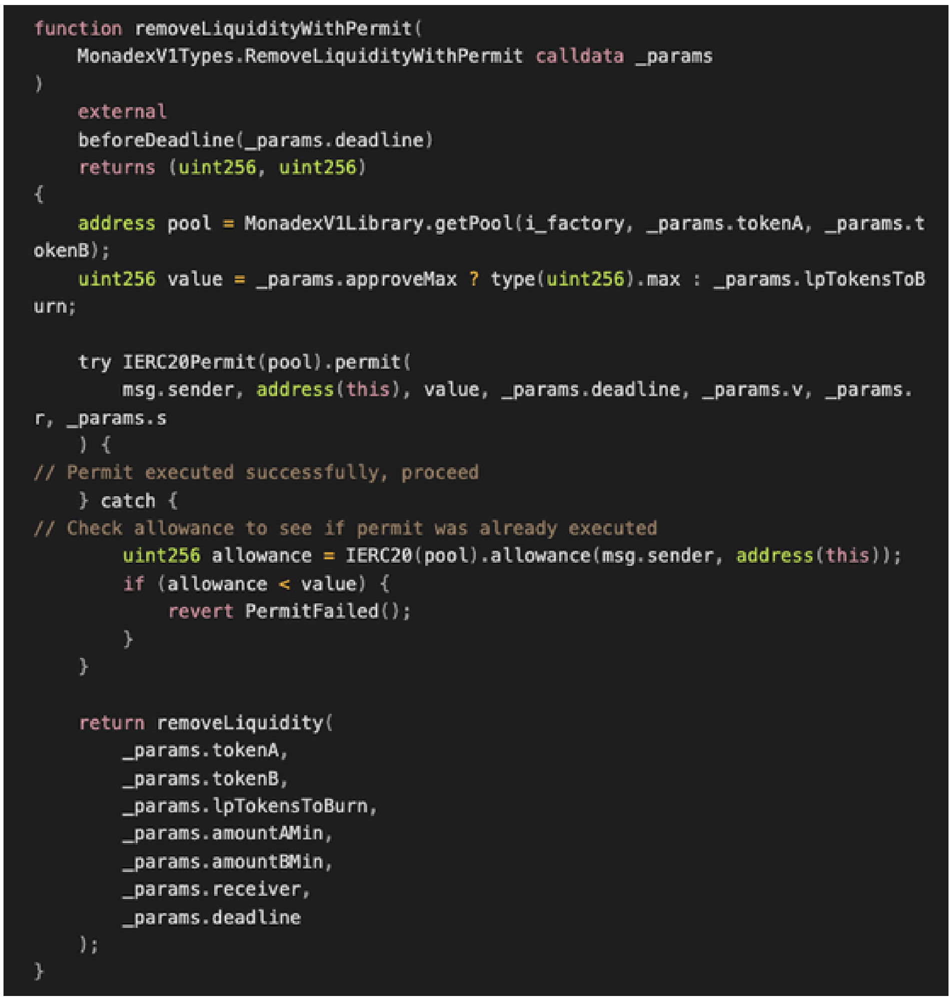
1. Front-running: If a permit is front-run, the subsequent transaction will fail entirely.
2. Replay: If a user accidentally tries to use the same permit twice, the second transaction will fail.
3. Smart Contract Wallets: Some smart contract wallets cannot produce permit signatures, potentially limiting functionality for these users.

These issues do not pose a direct security threat but may lead to a suboptimal user experience and potential gas waste for failed transactions.

Mitigation

Implement a try-catch block around the *permit* call and add a fallback allowance check. This approach allows the function to proceed if the permit has already been executed or fails for other reasons, as long as the necessary allowance is in place.

Update the *removeLiquidityWithPermit* and *removeLiquidityNativeWithPermit* functions to use the following pattern:



```
function removeLiquidityWithPermit(
    MonadexV1Types.RemoveLiquidityWithPermit calldata _params
)
    external
    beforeDeadline(_params.deadline)
    returns (uint256, uint256)
{
    address pool = MonadexV1Library.getPool(i_factory, _params.tokenA, _params.t
okenB);
    uint256 value = _params.approveMax ? type(uint256).max : _params.lpTokensToB
urn;

    try IERC20Permit(pool).permit(
        msg.sender, address(this), value, _params.deadline, _params.v, _params.
r, _params.s
    ) {
// Permit executed successfully, proceed
    } catch {
// Check allowance to see if permit was already executed
        uint256 allowance = IERC20(pool).allowance(msg.sender, address(this));
        if (allowance < value) {
            revert PermitFailed();
        }
    }

    return removeLiquidity(
        _params.tokenA,
        _params.tokenB,
        _params.lpTokensToBurn,
        _params.amountAMin,
        _params.amountBMin,
        _params.receiver,
        _params.deadline
    );
}
```

And apply the same pattern to removeLiquidityNativeWithPermit.

[L-4] Supported Tokens should be limited to 5 or 6 tops to avoid high gas consumption

Description

The current implementation of the raffle system does not limit the number of supported tokens. This could lead to high gas consumption during the weekly draw, potentially making the system inefficient or unusable as more tokens are added over time.

Impact

As the number of supported tokens grows, gas costs for operations like the weekly draw could become prohibitively expensive.

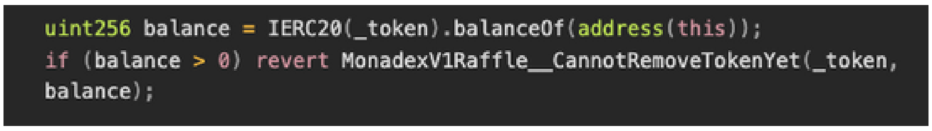
Mitigation

Implement a hard cap on the number of supported tokens (suggested 5-6 tokens maximum).

[L-5] Unable to Remove Tokens with Non-Zero Balance

Description

The MonadexV1Raffle contract's *removeToken* function includes a safety check that prevents the removal of a token if the contract holds any balance of that token. While this check is intended to prevent accidental removal of valuable assets, it introduces a significant vulnerability that could potentially lock the contract into supporting problematic or malicious tokens indefinitely.



```
uint256 balance = IERC20(_token).balanceOf(address(this));
if (balance > 0) revert MonadexV1Raffle__CannotRemoveTokenYet(_token,
balance);
```

This seemingly prudent check can backfire in several scenarios, effectively trapping the contract with tokens it cannot remove, regardless of the necessity or urgency of their removal.

Impact

1. If a token is later discovered to be malicious or compromised, the contract has no way to remove it from the supported list as long as any balance remains. This could expose users to ongoing risk
2. Tokens with implementation bugs that prevent transfers or have other issues cannot be removed, potentially affecting the raffle's operations or user interactions.
3. Even tiny, non-transferable dust amounts of a token can prevent its removal, leading to a bloated and potentially misleading list of supported tokens.

Mitigation

Add a force removal function for emergency situations or another mechanism to avoid such scenarios.

[L-6] Supported Tokens should be limited to 5 or 6 tops to avoid high gas consumption

Description

The current implementation of the raffle ticket system in the MonadexV1Raffle contract may result in users having unused tickets due to the rounding that occurs when calculating the number of slots a user can occupy.

The contract divides the user's ticket amount by the range size and then multiplies it back, which can lead to some tickets being unused in the raffle. However, the contract does not explicitly inform users about these unused tickets, which may remain in their balance.

Impact

Users may be unaware that they have unused tickets after participating in a raffle. This could lead to confusion about their actual ticket balance. That also means that users might miss opportunities to use these tickets in future raffles.

Mitigation

- Implement a mechanism to calculate and track unused tickets:

```
uint256 balance = IERC20(_token).balanceOf(address(this));
if (balance > 0) revert MonadexV1Raffle__CannotRemoveTokenYet(_token,
balance);
```

- Return the number of unused tickets to the user when they register for a raffle:

```
function register(uint256 _amount) external notZero(_amount) retur
ns (uint256, uint256) {
// ... existing code ...
    return (ticketsToBurn, unusedTickets);
}
```

- Emit an event that includes information about unused tickets:

```
emit Registered(msg.sender, ticketsToBurn, unusedTickets);
```