hexens x Zealous

Security Review Report for Zealous Swap

May 2025

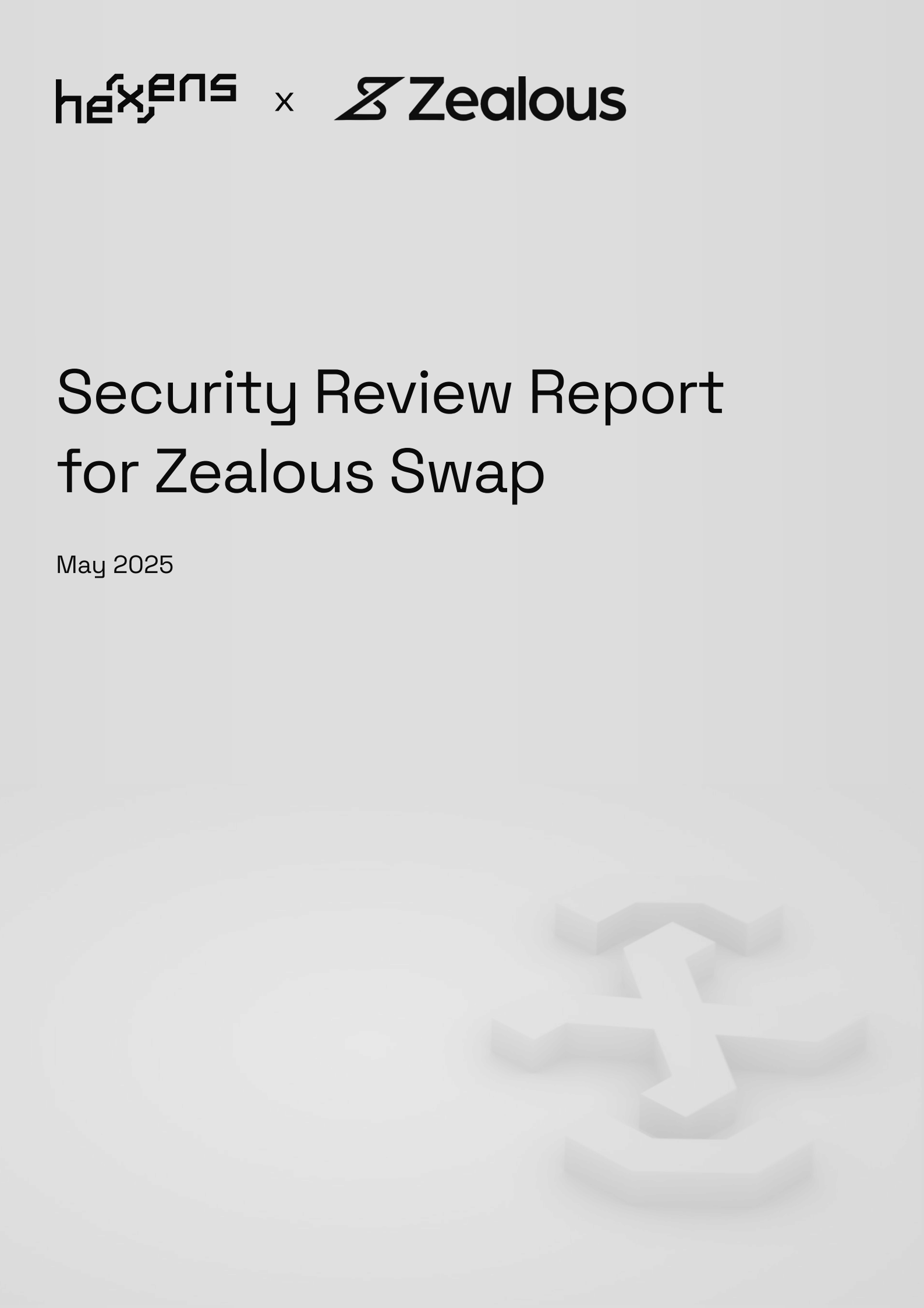


Table of Contents

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 covers ZealousSwap, the first AMM-based decentralized exchange designed specifically for the Kaspa ecosystem. The platform features an innovative NFT-based fee system tailored to address the unique challenges of this emerging blockchain network.

Our two-week security assessment involved a thorough review of five smart contracts.

During the audit, we discovered a high-severity vulnerability that could allow the first liquidity provider (LP) to steal tokens from other users. In addition, we identified five medium-severity issues, two low-severity issues, and two informational findings.

All reported issues were addressed by the development team and subsequently verified by our team.

As a result, we can confidently state that both the security and overall code quality of the protocol have significantly 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/louissaadgo/zealous-swap-contracts
- ZealousSwapFactory.sol
- ZealousSwapRouter.sol
- ZealousSwapNFTStaking.sol
- ZEALInfinityPool.sol
- ZealousSwapFarms.sol

📌 Commit: `ff2b794fbcc6fb0dc1ffac0808d7ef06d7574cf6`

🔗 https://github.com/louissaadgo/zealous-swap-contracts/blob/2ead1e488dfb4d85391b2326631c1ce6b7bf4d5b/contracts/WKAS.sol

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

🔗 https://github.com/louissaadgo/zealous-swap-contracts

📌 Commit: `024059b3cd607e80cce0bee061927fc9442e9d0c`

▪ 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.

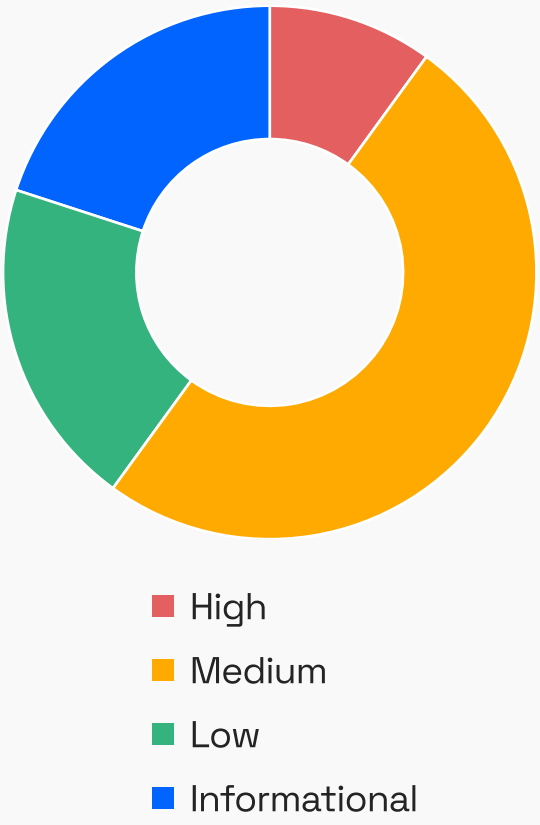
- **Issue Symbolic Codes**

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	0
High	1
Medium	5
Low	2
Informational	2
Total:	**10**



6. Weaknesses

This section contains the list of discovered weaknesses.

ZLS1-2 | First Depositor Can Front-Run and Steal the Next User's Stake

Fixed

Severity:	Probability:	Impact:
High	Likely	High

Path:

contracts/ZEALInfinityPool.sol#L196-L197
contracts/ZEALInfinityPool.sol#L274-L287

Description:

The **ZEALInfinityPool.stake()** function mints xZeal shares in exchange for Zeal tokens. The number of shares minted is calculated as follows:

```
1. currentRate = (totalStaked + totalRewards).mulDiv(PRECISION_FACTOR,
xSupply);

2. xMint = _amount.mulDiv(PRECISION_FACTOR, currentRate);
```

Since the calculation uses rounding down, **xMint** will be zero if **_amount * PRECISION_FACTOR < currentRate**. An attacker can exploit this by inflating the exchange rate using the **addRewards()** function, effectively making the next user's stake lose half of value.

Exploit Scenario (Assuming **zealPerBlock = 0**)

1. Initial Stake:
   - The attacker stakes **minStakeAmount**, receiving the same amount in xZeal.
   - State:
     - totalStaked = minStakeAmount
     - xZeal.totalSupply() = minStakeAmount
     - getExchangeRate() = 1e18

2. Partial Unstake:
    - The attacker unstakes **minStakeAmount - 1** xZeal.
    - State:
        - totalStaked = 1
        - xZeal.totalSupply() = 1
        - getExchangeRate = 1e18
3. Victim Prepares to Stake:
    - Alice (a regular user) submits a transaction to stake **X** Zeal.
4. Front-Run with Rewards:
    - The attacker front-runs Alice by calling **addRewards(X / 2)**.
    - This increases **totalStaked** without changing **xZeal.totalSupply**, causing the exchange rate to spike.
    - State:
        - totalStaked = 1 + X / 2
        - xZeal.totalSupply() = 1
        - getExchangeRate() = (1 + X / 2) * 1e18
5. Alice's Stake Executes:
    - Alice's minted shares:
      xMint = X * 1e18 / ((1 + X / 2) * 1e18) = X / (1 + X / 2)
        - Due to rounding, **xMint = 1**

Outcome

The attacker eventually calls **unstake()** to reclaim their Zeal tokens, they will receive:

```
    (X + 1 + X / 2) / 2

= (X / 2 + 1) + (X / 4 - 0.5)
```

Here:

- **(X / 2 + 1)** represents the attacker's initial investment in the strategy.
- **(X / 4 - 0.5)** is the net profit gained from front-running and draining the next user's stake.

```
uint256 xMint = _amount.mulDiv(PRECISION_FACTOR, currentRate);

require(xMint > 0, "Stake amount too small");
```

```
totalRewards += _amount;
totalManualRewards += _amount;

uint256 xSupply = xZealToken.totalSupply();
uint256 newRate;

if (xSupply == 0) {
    newRate = currentRate;
} else {
    newRate = (totalStaked + totalRewards).mulDiv(
        PRECISION_FACTOR,
        xSupply
    );
}
```

Remediation:

Consider burning an initial amount of xZeal tokens (representing dead shares), similar to the approach used in the **ZealousSwapPair** contract.

ZLS1-1 | All users can benefit from a discounted fee instead of the regular fee when performing a swap

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/ZealousSwapPair.sol#L285-L311

Description:

The function **ZealousSwapPair.swap()** facilitates token swaps and takes an input parameter **actualUser**, representing the true initiator of the swap (typically the caller of the router contract). This **actualUser** is used to determine fee eligibility—specifically, whether the user qualifies for a discount—within lines 287 to 291. If eligible, the swap fee is reduced from **regularFee** to **discountFee**.

```
function swap(
    uint amount0Out,
    uint amount1Out,
    address to,
    bytes calldata data,
    address actualUser /// @audit There is no guarantee that actualUser is the
actual caller
) external lock {
    ...

    {
        // Scope for reserve{0,1}Adjusted to avoid stack-too-deep errors
        if (discountManagerContract != address(0)) {
            isDiscountEligible = IZealousSwapDiscountManager(
                discountManagerContract
            ).isDiscountEligible(actualUser);
        }
        uint balance0Adjusted = balance0.mul(10000).sub(
            (
                balance0 > _reserve0 - amount0Out
                    ? balance0 - (_reserve0 - amount0Out)
                    : 0
            ).mul(isDiscountEligible ? discountFee : regularFee)
        );
```

```
        uint balance1Adjusted = balance1.mul(10000).sub(
            (
                balance1 > _reserve1 - amount1Out
                    ? balance1 - (_reserve1 - amount1Out)
                    : 0
            ).mul(isDiscountEligible ? discountFee : regularFee)
        );
        require(
            balance0Adjusted.mul(balance1Adjusted) >=
                uint(_reserve0).mul(_reserve1).mul(10000 ** 2),
            "ZealousSwap: K"
        );
    }

    ...
```

However, the function does not enforce that **actualUser** is the true caller of the swap. As a result, an attacker could bypass the router and call **swap** directly, supplying a discount-eligible address as **actualUser**—even if that address is not the one actually executing the transaction. This allows them to illegitimately benefit from a reduced swap fee.

Remediation:

Consider implementing a whitelist of approved router addresses. If the caller of the **swap** function is not in the whitelist, enforce that **msg.sender** must match **actualUser**. This ensures that only trusted routers can pass a different address as **actualUser**, preventing misuse of the discount mechanism.

```
++ mapping(address => bool) public isWhitelistedRouter;

function swap(
    uint amount0Out,
    uint amount1Out,
    address to,
    bytes calldata data,
    address actualUser
) external lock {
++  if (!isWhitelistedRouter[msg.sender]) {
++      require(
++          msg.sender == actualUser,
++          "Incorrect actual user"
++      );
++  }

    ...
}
```

ZLS1-3 | An attacker can set the exchange rate of ZEALInfinityPool to a very large or unlimited value

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/ZEALInfinityPool.sol#L192-L194
contracts/ZEALInfinityPool.sol#L85-L99

Description:

In the ZEALInfinityPool contract, the exchange rate between staked Zeal tokens and minted xZeal tokens always increases, using the **highestExchangeRate** variable as a minimum bound.

```
function getExchangeRate() public view returns (uint256) {
    uint256 xSupply = xZealToken.totalSupply();
    if (xSupply == 0) {
        return highestExchangeRate; // Return highest rate if no supply
    }
    uint256 currentRate = (totalStaked + totalRewards).mulDiv(
        PRECISION_FACTOR,
        xSupply
    );

    return
        currentRate > highestExchangeRate
            ? currentRate
            : highestExchangeRate;
}
```

```
function stake(uint256 _amount) external nonReentrant {
    ...
    if (xSupply == 0) {
        currentRate = PRECISION_FACTOR;
    } else {
        currentRate = (totalStaked + totalRewards).mulDiv(
            PRECISION_FACTOR,
            xSupply
        );
```

```
    }

    currentRate = currentRate > highestExchangeRate
        ? currentRate
        : highestExchangeRate;
    ...
```

However, even when the current supply is zero, the contract still uses the previous **highestExchangeRate** as the minimum bound. This allows an attacker to repeatedly stake and unstake when the supply is zero, inflating the exchange rate, especially when rewards are added.

With no cost, the attacker can inflate the exchange rate to an extremely large or even unlimited value, effectively preventing others from staking into the contract (DoS).

Example:

1. Initially, **currentRate = highestExchangeRate = PRECISION_FACTOR** (1e18).
   The attacker stakes 1e18 Zeal tokens and receives 1e18 xZeal shares.
2. The attacker then adds 1e18 Zeal tokens as rewards. Now, **highestExchangeRate = currentRate = 2e18**.
   The attacker unstakes 1e18 xZeal shares and receives the full 2e18 Zeal tokens, because the exchange rate for unstaking is set to **highestExchangeRate** when the remaining xZeal supply is zero.
3. Next, the attacker stakes 1e18 Zeal tokens again but receives only 0.5e18 xZeal shares, since the highestExchangeRate is now 2e18.
   The attacker can then add 1e18 Zeal tokens as rewards and unstake the 0.5e18 xZeal shares, doubling the exchange rate to 4e18.
4. By repeating this process (with just 2e18 Zeal tokens), the attacker can increase the **highestExchangeRate** to a very large value, making it so that any new Zeal token stakes will receive zero shares — effectively causing a denial-of-service (DoS) on the pool.

Remediation:

Consider reseting **highestExchangeRate** to its default value whenever the total supply returns to zero.

ZLS1-5 | Compilation failed due to incompatible Solidity versions

Fixed

Severity:	Probability:	Impact:
Medium	Unlikely	Medium

Path:

contracts/ZealousSwapPair.sol#L1
contracts/interfaces/IZealousSwapDiscountManager.sol#L1

Description:

The contract **ZealousSwapPair** imports the interface **IZealousSwapDiscountManager**. However, there is a version mismatch: **ZealousSwapPair** is written for Solidity **0.5.16**, while IZealousSwapDiscountManager requires Solidity version >=0.8.20. This version discrepancy causes a compiler error when attempting to compile **ZealousSwapPair**.

```
pragma solidity =0.5.16;
```

```
pragma solidity ^0.8.20;
```

Remediation:

Consider changing the solidity version of the interface **IZealousSwapDiscountManager** to **0.5.16**.

ZLS1-10 | lastRewardBlock Should Be Updated to Current Block When Resuming Emissions

Fixed

Severity:	Probability:	Impact:
Medium	Rare	High

Path:

contracts/ZEALInfinityPool.sol#L300-L304

Description:

When emissions are paused and later resumed via the **setEmissionsPaused(false)** function in the **ZEALInfinityPool** contract, the **lastRewardBlock** should be updated to the current block number at the time of resumption. Failing to do so results in reward accrual during the paused period, which is incorrect.

Consider the following scenario:

1. At block **100**, the contract is configured as follows:
   - zealPerBlock = 10
   - totalRewards = 1000
   - lastRewardBlock = 100
   - xZealToken.totalSupply() > 0
   - emissionsPaused = false
2. At block **200**, the owner calls **setEmissionsPaused(true)**:
   - **_updateEmissions()** is invoked in line 301:
     - blocksSince = 200 - 100 = 100
     - pending = 100 * 10 = 1000
     - totalRewards = 2000
     - lastRewardBlock = 200
   - **emissionsPaused** is set to true in line 302.
3. At block **300**, the owner calls **setEmissionsPaused(false)**:
   - **_updateEmissions()** exits early due to **emissionsPaused** being **true**, so **lastRewardBlock** remains at **200**.
   - **emissionsPaused** is now set to false.
4. Still at block 300, a user calls stake(), which triggers _updateEmissions():
   - blocksSince = 300 - 200 = 100
   - pending = 100 * 10 = 1000
   - totalRewards = 3000
   - lastRewardBlock = 300

The issue is the rewards were incorrectly accrued for the paused period (block 200 to 300), even though emissions were halted. This happens because **lastRewardBlock** wasn't updated when emissions resumed, causing **_updateEmissions()** to include the paused period in its reward calculations.

```
function setEmissionsPaused(bool _paused) external onlyOwner {
    _updateEmissions();
    emissionsPaused = _paused;
    emit EmissionsPaused(_paused);
}
```

Remediation:

When **setEmissionsPaused(false)** is called, **lastRewardBlock** should be explicitly set to the current block number to ensure that no rewards are emitted for the paused interval.

ZLS1-4 | User Can Bypass Locking Period Without Penalty Using emergencyWithdraw

Fixed

Severity:	Probability:	Impact:
Medium	Likely	Medium

Path:

contracts/ZealousSwapFarms.sol#L424-L440

Description:

When a user deposits tokens, they can't withdraw them until the locking period has passed. This is enforced in the **withdraw** function with the following check:

```
function withdraw(uint256 _pid, uint256 _amount) public nonReentrant {
  -- snip --
  require(
      block.number >= user.lastInteraction + lockingPeriod,
      "ZealousSwapFarms: Tokens are still locked"
  );
```

However, the **emergencyWithdraw** function does not have this restriction. This means a user can bypass the locking period and immediately withdraw their tokens by calling **emergencyWithdraw**.

```
function emergencyWithdraw(uint256 _pid) public nonReentrant {
    require(_pid < poolInfo.length, "ZealousSwapFarms: Invalid pool ID");

    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];

    uint256 amount = user.amount;
    require(amount > 0, "ZealousSwapFarms: No LP tokens staked");

    user.amount = 0;
    user.rewardDebt = 0;
    pool.totalDeposited = pool.totalDeposited - amount;
    pool.lpToken.safeTransfer(address(msg.sender), amount);
```

```
        emit EmergencyWithdraw(msg.sender, _pid, amount);
    }
```

Remediation:

To prevent users from bypassing the locking mechanism using **emergencyWithdraw**, introduce a penalty mechanism. For example, deduct a percentage (5–10%) from the withdrawn amount when **emergencyWithdraw** is used.

ZLS1-6 | Missing validation of lastRewardBlock when adding a new pool in ZealousSwapFarms

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

contracts/ZealousSwapFarms.sol#L143-L148

Description:

In the **ZealousSwapFarms::add()** function, if it reactivates an old pool, the **lastRewardBlock** of that pool is set to the maximum of **block.number** and the global **startBlock**. However, if it adds a new pool, the **lastRewardBlock** is specified via input, and there is no validation to ensure that it is greater than **block.number** or **startBlock.**

```
function add(
    uint256 _allocPoint,
    IERC20 _lpToken,
    bool _withUpdate,
    uint256 _lastRewardBlock
) public onlyOwner {
  ...

  uint256 lastRewardBlock = _lastRewardBlock;
  if (_lastRewardBlock == 0) {
    lastRewardBlock = block.number > startBlock
        ? block.number
        : startBlock;
  }

  ...
```

Therefore, a small **lastRewardBlock** might be set for a new pool, causing rewards to be distributed unfairly, as early liquidity providers would receive most of the rewards. This happens because a large time lapse is counted the first time **updatePool** is triggered with this small lastRewardBlock value.

Remediation:

The **lastRewardBlock** of a new pool should be checked after it is set.

For example:

```
    uint256 lastRewardBlock = _lastRewardBlock;
    if (_lastRewardBlock == 0) {
        lastRewardBlock = block.number > startBlock
            ? block.number
            : startBlock;
    }
+   require(
+       lastRewardBlock >= block.number &&
+       lastRewardBlock >= startBlock,
+       "ZealousSwapFarms: Too small lastRewardBlock"
+   );
```

ZLS1-9 | Calling ZealousSwapFarms.setRewardToken() May Result in Inflated or Incorrect Staker Payouts

Fixed

Severity:	Probability:	Impact:
Low	Rare	Medium

Path:

contracts/ZealousSwapFarms.sol#L442-L452

Description:

The **setRewardToken()** function in **ZealousSwapFarms** allows the farm owner to update the reward token. However, this change can be made without first distributing all pending rewards in the previous token. As a result, stakers may unintentionally receive rewards in the new token, potentially leading to incorrect or inflated payouts.

For example:

- Alice has 1,000 USDC in unclaimed rewards (**pendingReward(some_pid, Alice) = 1000 USDC**).
- The owner calls **setRewardToken()** to change the reward token from USDC to BTC.
- When Alice later calls **claim(some_pid)**, she receives 1,000 BTC instead of 1,000 USDC, creating a massive unintended profit.

This flaw could lead to significant financial discrepancies.

```
function setRewardToken(IERC20 _newRewardToken) public onlyOwner {
    require(
        address(_newRewardToken) != address(0),
        "ZealousSwapFarms: Invalid token address"
    );

    massUpdatePools();

    rewardToken = _newRewardToken;
    emit RewardTokenUpdated(address(_newRewardToken));
}
```

Remediation:

When updating the reward token for a pool, we recommend deploying a new farm contract and instructing users to migrate their stakes to the new contract to prevent reward inconsistencies.

ZLS1-7 | Redundant check of the NFT owner in the stakeNFT() function

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

contracts/ZealousSwapNFTStaking.sol#L310-L313

Description:

In the **ZealousSwapNFTStaking** contract, this owner check of the NFT from line 310 to 313 in the **stakeNFT()** function is redundant:

```
require(
    IERC721(nftContract).ownerOf(nftID) == msg.sender,
    "ZealousSwapNFTStaking: Not token owner"
);
```

This is because the function already checks the NFT's owner before and after the transfer, as shown in line 325.

```
address beforeOwner = IERC721(nftContract).ownerOf(nftID);
IERC721(nftContract).transferFrom(msg.sender, address(this), nftID);
address afterOwner = IERC721(nftContract).ownerOf(nftID);
require(
    afterOwner == address(this) && beforeOwner == msg.sender,
    "ZealousSwapNFTStaking: NFT transfer failed"
);
```

Remediation:

Remove the redundant check.

ZLS1-8 | Redundant Boundary Checks in ZealousSwapFactory.setFeesForAllPairs()

Fixed

Severity:	Probability:	Impact:
Informational	Very likely	Informational

Path:

contracts/ZealousSwapFactory.sol#L49-L53
contracts/ZealousSwapFactory.sol#L66-L70
contracts/ZealousSwapPair.sol#L46-L50

Description:

Lines 49–53 of the **setFeesForAllPairs()** function in the **ZealousSwapFactory** contract perform boundary validation on **_regularFee** and **_discountFee** before calling **IZealousSwapPair(allPairs[i]).setFees(_regularFee, _discountFee)** at line 56. However, the **setFees()** function in the **ZealousSwapPair** contract already enforces the same boundary checks (lines 46–50). As a result, these validations are executed twice, leading to unnecessary gas consumption.

A similar redundancy exists in the **ZealousSwapFactory.setFeesForSpecificPairs()** function as well.

```
require(_regularFee <= 100, "ZealousSwap: FEE_TOO_HIGH");
require(
    _discountFee <= _regularFee,
    "ZealousSwap: DISCOUNT_HIGHER_THAN_REGULAR"
);
```

Remediation:

Consider removing the fee boundary check within the function **setFeesForAllPairs()** and **setFeesForSpecificPairs()**.

```
    function setFeesForAllPairs(uint _regularFee, uint _discountFee) external {
        require(msg.sender == feeToSetter, "ZealousSwap: FORBIDDEN");
--      require(_regularFee <= 100, "ZealousSwap: FEE_TOO_HIGH");
--      require(
--          _discountFee <= _regularFee,
--          "ZealousSwap: DISCOUNT_HIGHER_THAN_REGULAR"
--      );

        for (uint i = 0; i < allPairs.length; i++) {
            IZealousSwapPair(allPairs[i]).setFees(_regularFee, _discountFee);
        }
    }

    function setFeesForSpecificPairs(
        uint _regularFee,
        uint _discountFee,
        address[] calldata _pairsToUpdate
    ) external {
        require(msg.sender == feeToSetter, "ZealousSwap: FORBIDDEN");
--      require(_regularFee <= 100, "ZealousSwap: FEE_TOO_HIGH");
--      require(
--          _discountFee <= _regularFee,
--          "ZealousSwap: DISCOUNT_HIGHER_THAN_REGULAR"
--      );


        for (uint i = 0; i < _pairsToUpdate.length; i++) {
            IZealousSwapPair(_pairsToUpdate[i]).setFees(
                _regularFee,
                _discountFee
            );
        }
    }
```

hexens x Zealous