hashlock.

Security Audit

Zeus Exchange 2nd (DeFi)

September 2025 info@hashlock.com.au Hashlock Pty Ltd

Table of Contents

CAUTION

THIS DOCUMENT IS A SECURITY AUDIT REPORT AND MAY CONTAIN CONFIDENTIAL INFORMATION. THIS INCLUDES IDENTIFIED VULNERABILITIES AND MALICIOUS CODE WHICH COULD BE USED TO COMPROMISE THE PROJECT. THIS DOCUMENT SHOULD ONLY BE FOR INTERNAL USE UNTIL ISSUES ARE RESOLVED. ONCE VULNERABILITIES ARE REMEDIATED, THIS REPORT CAN BE MADE PUBLIC. THE CONTENT OF THIS REPORT IS OWNED BY HASHLOCK PTY LTD FOR USE OF THE CLIENT.

Executive Summary

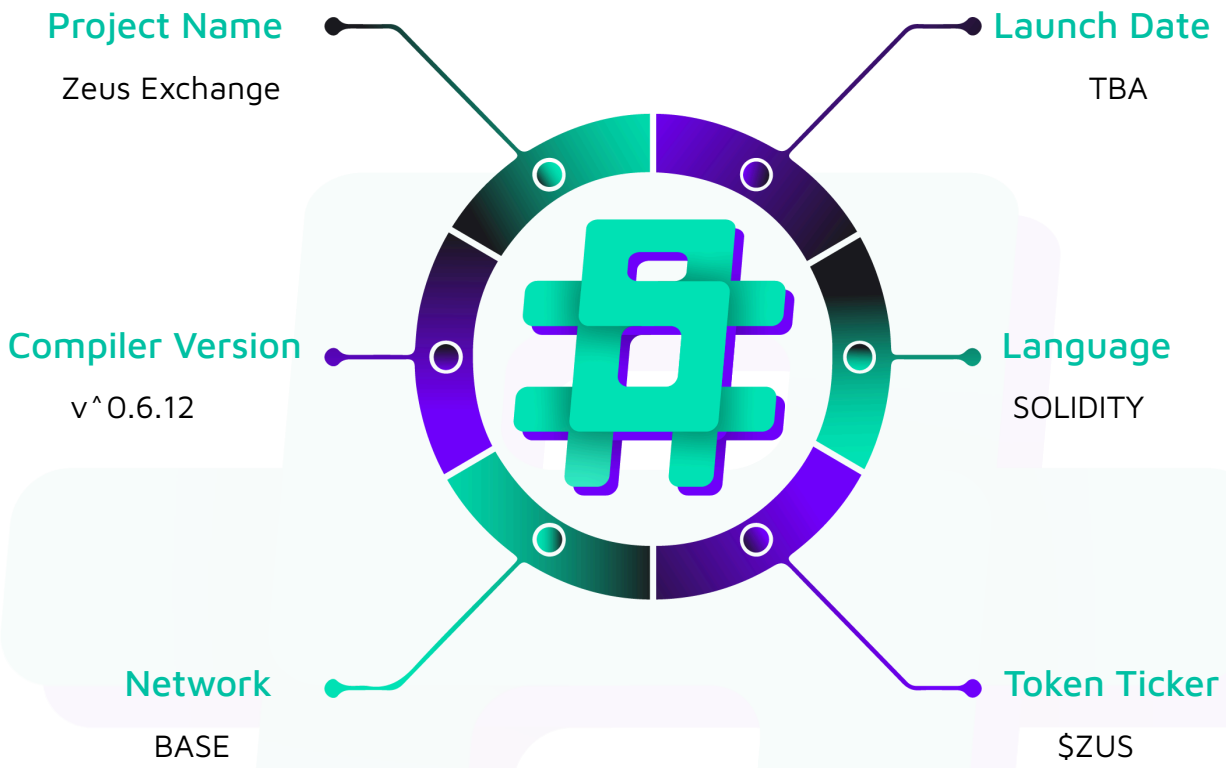
The Zeus Exchange team partnered with Hashlock to conduct a security audit of their smart contracts. Hashlock manually and proactively reviewed the code in order to ensure the project's team and community that the deployed contracts are secure.

Project Context

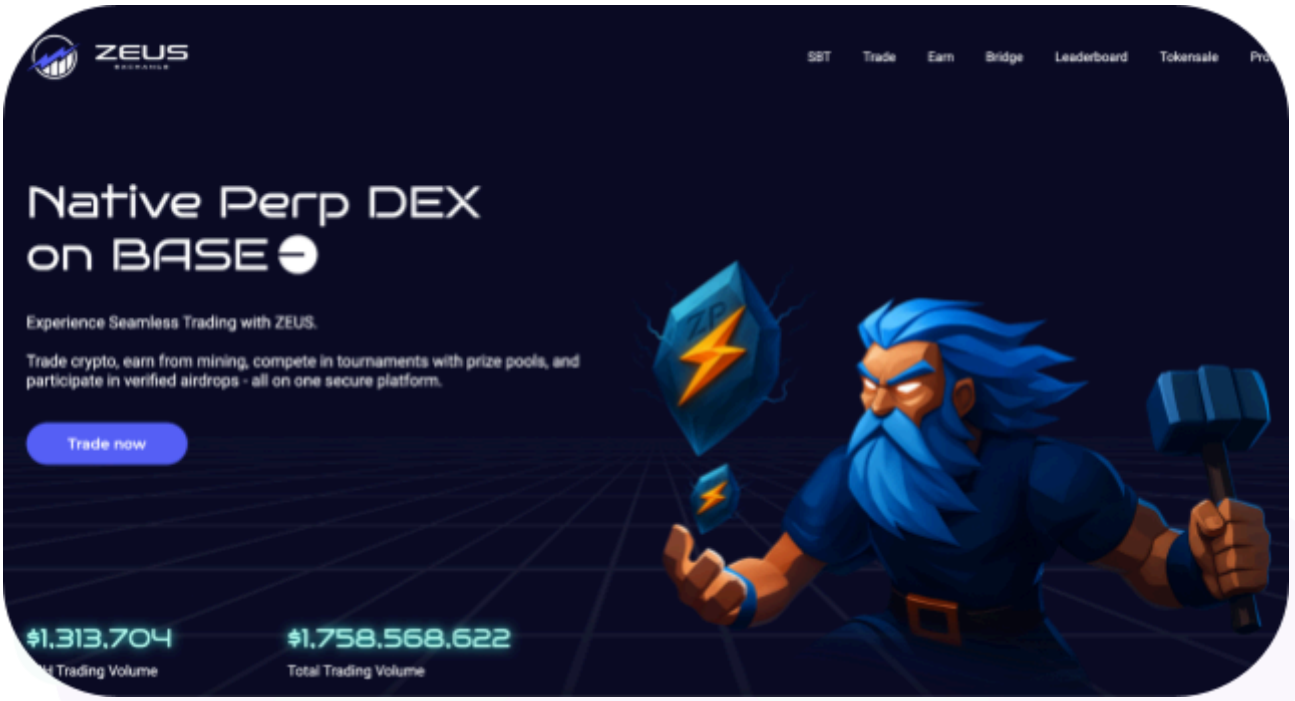
Zeus Exchange is a decentralized perpetual exchange (DEX) built on the Base blockchain. It enables both spot and perpetual (derivatives) trading directly from users' wallets without intermediaries. The platform sets itself apart by leveraging Soulbound Tokens (SBTs) to identify real users and fairly distribute 25% of monthly revenue as airdrops—combating airdrop abuse and promoting long-term engagement. With plans for multichain support, copy trading, AI agents, and RWA trading pairs, Zeus is positioning itself as a performance-driven, community-aligned player in DeFi's derivatives space.

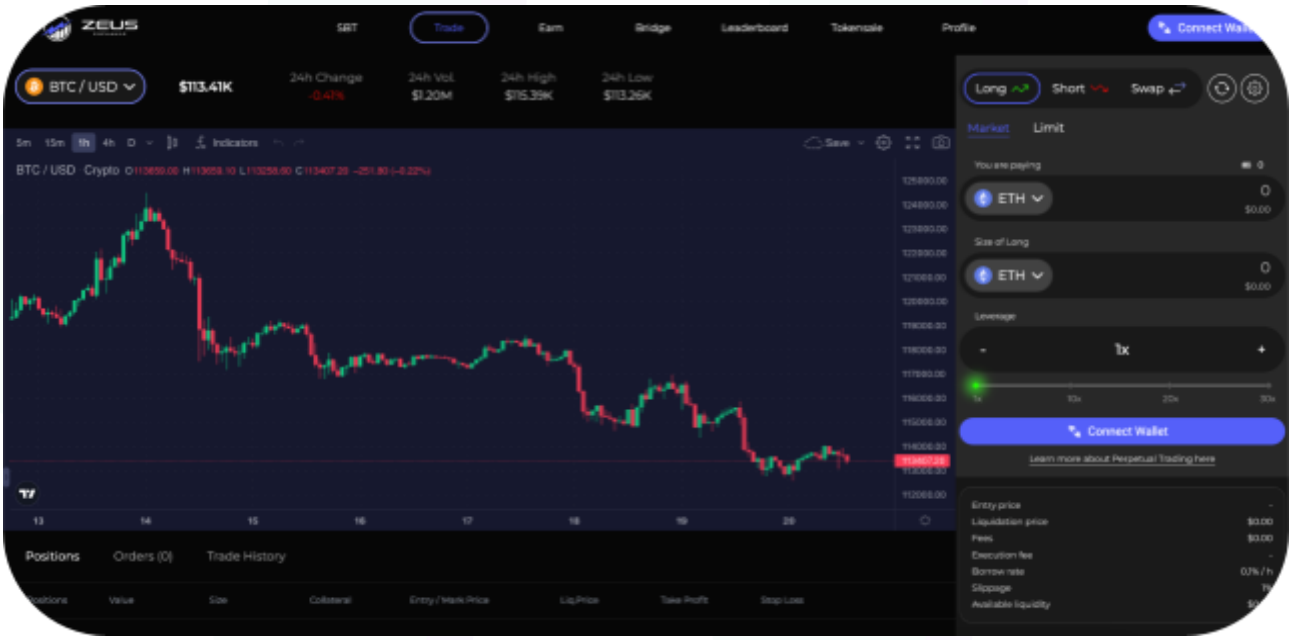
Project Name: Zeus Exchange
Project Type: DeFi
Compiler Version: ^0.6.12
Website: https://zeustrade.io/
Logo:

Visualised Context:



Project Visuals:





Audit Scope

We at Hashlock audited the solidity code within the Zeus Exchange project, the scope of work included a comprehensive review of the smart contracts listed below. We tested the smart contracts to check for their security and efficiency. These tests were undertaken primarily through manual line-by-line analysis and were supported by software-assisted testing.

Description	**Zeus Exchange Smart Contracts**
Platform	**Ethereum / Solidity**
Audit Date	**September, 2025**
Contract 1	BonusDistributor.sol
Contract 2	RewardDistributor.sol
Contract 3	RewardRouter.sol
Contract 4	RewardRouterV2.sol
Contract 5	RewardTracker.sol
Contract 6	StakeManager.sol
Contract 7	StakedZlp.sol
Contract 8	StakedZlpMigrator.sol
Contract 9	Vester.sol
Contract 10	ZusBalance.so
Audited GitHub Commit Hash	65fb549b4996bd3f4f81e1b837737f077f388f88
Fix Review GitHub Commit Hash	02d7fb6cb4bed9b910489186c9ec87beb6e71990

Security Rating

After Hashlock's Audit, we found the smart contracts to be **"Secure"**. The contracts all follow simple logic, with correct and detailed ordering. They use a series of interfaces, and the protocol uses a list of Open Zeppelin contracts.

The 'Hashlocked' rating is reserved for projects that ensure ongoing security via bug bounty programs or on chain monitoring technology.

All issues uncovered during automated and manual analysis were meticulously reviewed and applicable vulnerabilities are presented in the Audit Findings section. The list of audited assets is presented in the Audit Scope section and the project's contract functionality is presented in the Intended Smart Contract Functions section.

All vulnerabilities initially identified have now been resolved.

Hashlock found:

2 High severity vulnerabilities

3 Medium severity vulnerabilities

2 Low severity vulnerabilities

1 Gas Optimisation

6 QA

Caution: *Hashlock's audits do not guarantee a project's success or ethics, and are not liable or responsible for security. Always conduct independent research about any project before interacting.*

Intended Smart Contract Functions

Claimed Behaviour	Actual Behaviour
BonusDistributor.sol - Set bonus multiplier for token distribution - Update the last distribution timestamp - Set admin address - Withdraw tokens accidentally sent to the contract - Distribute bonus tokens based on time elapsed and supply | Contract partially achieves this functionality. - Distribution is only callable by rewardTracker, not by the admin directly - Missing validation for the BONUS_DURATION period enforcement - Bonus multiplier changes are only allowed after the first distribution	**Contract achieves this functionality.**
RewardDistributor.sol - Set tokens per interval for reward distribution - Update the last distribution timestamp - Set admin address - Withdraw tokens accidentally sent to the contract - Distribute reward tokens based on time elapsed | Contract partially achieves this functionality. - Distribution is only callable by rewardTracker, not by the admin directly - Cannot recover the ETH accidentally sent to the contract - Token per interval changes are only allowed after	**Contract achieves this functionality.**

the first distribution	
RewardRouter.sol - Stake ZUS and esZUS tokens - Unstake ZUS and esZUS tokens - Mint and stake ZLP with tokens or ETH - Unstake and redeem ZLP for tokens or ETH - Claim all rewards from staking - Claim specific esZUS or fee rewards - Compound rewards automatically	**Contract achieves this functionality.**
RewardRouterV2.sol - Stake ZUS tokens - Unstake ZUS tokens - Mint and stake ZLP with tokens or ETH - Unstake and redeem ZLP for tokens or ETH - Claim all rewards from staking - Claim specific fee rewards - Handle rewards with ETH conversion options - Signal transfer of staking assets - Accept transfer of staking assets - Allows admins to: - Initialize contract addresses once - Batch stake ZUS for multiple accounts - Stake ZUS for specific accounts - Withdraw tokens accidentally sent to contract	**Contract achieves this functionality.**
RewardTracker.sol - Stake deposit tokens to earn rewards - Unstake deposit tokens and receive them back - Claim accumulated rewards - Transfer staking tokens (ERC20 compliant) - Approve token spending (ERC20 compliant)	**Contract achieves this functionality.**

- View claimable reward amounts - Allows admins to: - Initialize contract with deposit tokens and distributor - Set deposit token validity - Enable private transfer/staking/claiming modes - Set handler permissions for privileged operations - Withdraw tokens accidentally sent to contract - Update reward distributions	
StakedZlp.sol - Transfer staked ZLP tokens between accounts - Approve token spending for transfers - View staked ZLP balances and allowances - View total supply of staked ZLP	**Contract achieves this functionality.**
StakedZlpMigrator.sol - Transfer staked ZLP tokens from specified sender to recipients - Disable the migrator contract permanently - Migrate staked ZLP through unstaking and restaking process	**Contract achieves this functionality.**
StakeManager.sol - Stake tokens for any account on any reward tracker	**Contract achieves this functionality.**
Vester.sol - Deposit esTokens to start the vesting process - Claim vested claimableTokens over time - Withdraw all remaining esTokens and pair tokens	**Contract achieves this functionality.**

- View claimable amounts and vesting status - Allows admins to: - Set handler permissions for privileged operations - Set maximum vestable amount restrictions - Withdraw tokens accidentally sent to contract - Transfer stake values between accounts - Set transferred rewards and deductions manually	
ZusBalance.sol - Transfer ZLP staking balances between accounts - Approve token spending for transfers - View allowance amounts	**Contract achieves this functionality.**

Code Quality

This audit scope involves the smart contracts of the Zeus Exchange project, as outlined in the Audit Scope section. All contracts, libraries, and interfaces mostly follow standard best practices and to help avoid unnecessary complexity that increases the likelihood of exploitation, however, some refactoring were recommended to optimize security measures.

The code is very well commented on and closely follows best practice nat-spec styling. All comments are correctly aligned with code functionality.

Audit Resources

We were given the Zeus Exchange project smart contract code in the form of GitHub access.

As mentioned above, code parts are well commented. The logic is straightforward, and therefore it is easy to quickly comprehend the programming flow as well as the complex code logic. The comments are helpful in providing an understanding of the protocol's overall architecture.

Dependencies

As per our observation, the libraries used in this smart contracts infrastructure are based on well-known industry standard open source projects.
Apart from libraries, its functions are used in external smart contract calls.

Severity Definitions

The severity levels assigned to findings represent a comprehensive evaluation of both their potential impact and the likelihood of occurrence within the system. These categorizations are established based on Hashlock's professional standards and expertise, incorporating both industry best practices and our discretion as security auditors. This ensures a tailored assessment that reflects the specific context and risk profile of each finding.

Significance	Description
High	High-severity vulnerabilities can result in loss of funds, asset loss, access denial, and other critical issues that will result in the direct loss of funds and control by the owners and community.
Medium	Medium-level difficulties should be solved before deployment, but won't result in loss of funds.
Low	Low-level vulnerabilities are areas that lack best practices that may cause small complications in the future.
Gas	Gas Optimisations, issues, and inefficiencies.
QA	Quality Assurance (QA) findings are informational and don't impact functionality. Supports clients improve the clarity, maintainability, or overall structure of the code.

Status Definitions

Each identified security finding is assigned a status that reflects its current stage of remediation or acknowledgment. The status provides clarity on the handling of the issue and ensures transparency in the auditing process. The statuses are as follows:

Significance	Description
Resolved	The identified vulnerability has been fully mitigated either through the implementation of the recommended solution proposed by Hashlock or through an alternative client-provided solution that demonstrably addresses the issue.
Acknowledged	The client has formally recognized the vulnerability but has chosen not to address it due to the high cost or complexity of remediation. This status is acceptable for medium and low-severity findings after internal review and agreement. However, all high-severity findings must be resolved without exception.
Unresolved	The finding remains neither remediated nor formally acknowledged by the client, leaving the vulnerability unaddressed.

Audit Findings

High

[H-01] BonusDistributor#tokensPerInterval - Infinite Bonus Duration Beyond 365 Days

Description

The `BonusDistributor` contract lacks time-based validation to enforce the intended 365-day bonus distribution period. Despite defining `BONUS_DURATION = 365 days`, the contract continues distributing bonus tokens indefinitely without checking if the bonus period has expired, leading to unlimited token emissions beyond the designed timeframe.

Vulnerability Details

The root cause lies in the `tokensPerInterval` and `pendingRewards` functions, which use `BONUS_DURATION` only as a mathematical divisor for calculating emission rates, but never validate whether the current time exceeds the intended bonus period. The contract treats `BONUS_DURATION` as a rate calculation parameter rather than an actual time constraint.

```
function tokensPerInterval() public view override returns (uint256) {
    uint256 supply = IERC20(rewardTracker).totalSupply();
    // BONUS_DURATION used only as divisor, no time validation
                                                                    return
supply.mul(bonusMultiplierBasisPoints).div(BASIS_POINTS_DIVISOR).div(BONUS_DURATION);
}

function pendingRewards() public view override returns (uint256) {
    if (block.timestamp == lastDistributionTime) {
        return 0;
    }
    uint256 supply = IERC20(rewardTracker).totalSupply();
    uint256 timeDiff = block.timestamp.sub(lastDistributionTime);
```

```
    // No check if current time > bonus start + BONUS_DURATION
                                                                    return
timeDiff.mul(supply).mul(bonusMultiplierBasisPoints).div(BASIS_POINTS_DIVISOR).div(BONUS_
DURATION);
}
```

Impact

This vulnerability results in unlimited bonus token emissions, significantly exceeding the protocol's intended token distribution schedule. Attackers or users can continue earning bonus rewards indefinitely, potentially draining the reward token reserves and causing severe inflation. The economic model becomes unpredictable, as the total bonus supply grows without bounds, undermining tokenomics and potentially devaluing the reward token.

Recommendation

Add a bonus start timestamp and implement time-based validation to enforce the 365-day limit:

```
+ uint256 public bonusStartTime;
  function setBonusMultiplier(uint256 _bonusMultiplierBasisPoints) external onlyAdmin {
    require(lastDistributionTime != 0, "BonusDistributor: invalid lastDistributionTime");
+   if (bonusStartTime == 0 && _bonusMultiplierBasisPoints > 0) {
+       bonusStartTime = block.timestamp;
+   }
    IRewardTracker(rewardTracker).updateRewards();
    bonusMultiplierBasisPoints = _bonusMultiplierBasisPoints;
    emit BonusMultiplierChange(_bonusMultiplierBasisPoints);
  }
  function tokensPerInterval() public view override returns (uint256) {
+     if (bonusStartTime > 0 && block.timestamp > bonusStartTime.add(BONUS_DURATION)) {
+         return 0;
+     }
            uint256 supply = IERC20(rewardTracker).totalSupply();              return
supply.mul(bonusMultiplierBasisPoints).div(BASIS_POINTS_DIVISOR).div(BONUS_DURATION);
  }
  function pendingRewards() public view override returns (uint256) {
      if (block.timestamp == lastDistributionTime) {
```

```
            return 0;
        }

+       if (bonusStartTime > 0 && block.timestamp > bonusStartTime.add(BONUS_DURATION)) {
+           return 0;
+       }
        uint256 supply = IERC20(rewardTracker).totalSupply();

        uint256 timeDiff = block.timestamp.sub(lastDistributionTime);

                                                                                return
timeDiff.mul(supply).mul(bonusMultiplierBasisPoints).div(BASIS_POINTS_DIVISOR).div(BONUS_
DURATION);
    }
```

Status

Resolved

[H-02] RewardTracker#transfer - ERC20 transfer updates ERC20 balance but not staking/deposit state, breaking unstake

Description

The contract allows holders to call `transfer` (and `transferFrom`) to move `RewardTracker` ERC20 tokens between accounts while only updating the ERC20 balances mapping. Staking-related state is not moved or adjusted. This desynchronization permits a user to transfer their ERC20 representation without transferring the on-chain staking rights, causing senders or recipients to become unable to unstake or claim correctly.

Vulnerability Details

The root cause is that `_transfer` updates only balances and emits Transfer, while `_unstake` and reward logic rely on `stakedAmounts` and `depositBalances`. Example vulnerable fragments:

```
function _transfer(address _sender, address _recipient, uint256 _amount) private {
    require(_sender != address(0), "RewardTracker: transfer from the zero address");
    require(_recipient != address(0), "RewardTracker: transfer to the zero address");

    if (inPrivateTransferMode) _validateHandler();

      balances[_sender] = balances[_sender].sub(_amount, "RewardTracker: transfer amount exceeds balance");
    balances[_recipient] = balances[_recipient].add(_amount);

    emit Transfer(_sender, _recipient, _amount);
}
```

Meanwhile `_unstake` burns ERC20 balance and checks stakedAmounts:

```
uint256 stakedAmount = stakedAmounts[_account];
require(stakedAmounts[_account] >= _amount, "RewardTracker: _amount exceeds stakedAmount");

stakedAmounts[_account] = stakedAmount.sub(_amount);

...
```

```
_burn(_account, _amount);
IERC20(_depositToken).safeTransfer(_receiver, _amount);
```

Because `_transfer` does not touch stakedAmounts nor depositBalances, the following inconsistent states are possible:

- Sender transfers ERC20 tokens away but still has non-zero `stakedAmounts` and non-zero `depositBalances`. After the transfer, the sender lacks the ERC20 balance to burn on unstake — burn amount exceeds balance revert.

Recipient receives ERC20 tokens but has `stakedAmounts == 0` and `depositBalances == 0`, so calling unstake reverts with `_amount` exceeds `stakedAmount`.

Proof of Concept

```
function testTransferPreventsUnstaking() public {
    // --- Setup and Preconditions ---
    // Generate test addresses for alice (original staker) and bob (token recipient)
    address alice = address(0x1);
    address bob = address(0x2);

    // Mint 1 ZUS token to alice for staking
    zus.mint(alice, 1e18);

    vm.startPrank(alice);
    // Alice approves and stakes 1 ZUS token, receiving 1 RewardTracker token
    zus.approve(address(stakedZusTracker), 1e18);
    stakedZusTracker.stake(address(zus), 1e18);

    // --- Exploit Execution ---
    // Alice transfers her entire RewardTracker token balance to bob
    // This updates balances mapping but NOT depositBalances or stakedAmounts
    stakedZusTracker.transfer(bob, 1e18);

    // --- Post-condition Check ---
    // Alice cannot unstake despite having staking rights (no balance left)
    vm.expectRevert("RewardTracker: burn amount exceeds balance");
    stakedZusTracker.unstake(address(zus), 0.5e18);
```

```
    // PoC succeeds if alice cannot unstake due to insufficient balance

    vm.stopPrank();

    vm.startPrank(bob);
    // Bob cannot unstake despite owning tokens (no staking rights recorded)
    vm.expectRevert("RewardTracker: _amount exceeds stakedAmount");
    stakedZusTracker.unstake(address(zus), 1e18);
    // PoC succeeds if bob cannot unstake due to zero stakedAmounts
    vm.stopPrank();
}
```

Impact

Users may lose the ability to withdraw deposits or claim rewards if ERC20 receipt tokens are transferred. The sender cannot unstake due to insufficient balances for burning, while the recipient holds tokens without valid staking records. This causes funds to become locked until governance intervention, with risk scaling to the total staked supply.

Recommendation

Two safe options (pick one based on intended design):

1. Prohibit peer-to-peer transfers — simplest and safest: disallow `transfer`/`transferFrom` unless handler. Add a guard preventing transfers while the account has a staking state.
2. Move staking state on transfer — more complex: when transferring ERC20, transfer corresponding `stakedAmounts`, `depositBalances` (per deposit token), `previousCumulatedRewardPerToken`, `cumulativeRewards`, and `averageStakedAmounts`. This requires careful handling to avoid breaking accounting and to handle partial transfers and multiple deposit tokens.

Status

Resolved

Medium

[M-01] BonusDistributor#distribute -Claimable vs Actual Distribution Mismatch

Description

The contract exposes `pendingRewards` as the on-chain view for how many bonus tokens are claimable, while `distribute` actually caps distribution to the contract's current `rewardToken` balance. `pendingRewards` calculates a theoretical amount based on time and supply, but does not consider the token balance held by the contract. As a result, callers (and integrators) can read a higher claimable amount than will be delivered by `distribute` when the contract lacks sufficient tokens.

Vulnerability Details

The root cause is a mismatch between the view calculation and the execution path: `pendingRewards` computes rewards from `elapsed` time, supply, and the multiplier, but never compares that computed value with `IERC20(rewardToken).balanceOf(address(this))`. Meanwhile, `distribute` silently clamps the transfer amount to balance:

```
// from pendingRewards()
uint256 supply = IERC20(rewardTracker).totalSupply();
uint256 timeDiff = block.timestamp.sub(lastDistributionTime);
return
timeDiff.mul(supply).mul(bonusMultiplierBasisPoints).div(BASIS_POINTS_DIVISOR).div(BONUS_
DURATION);

// from distribute()
uint256 amount = pendingRewards();
...
uint256 balance = IERC20(rewardToken).balanceOf(address(this));
if (amount > balance) amount = balance;
IERC20(rewardToken).safeTransfer(msg.sender, amount);
```

Because `pendingRewards` can return X while `distribute` sends min(X, balance), a discrepancy arises when `balance < X`. This leads to inconsistent on-chain accounting

and broken expectations for callers that rely on the view. The problem surfaces whenever the contract's `rewardToken` balance is lower than the theoretical accrued amount (e.g., tokens were withdrawn via `withdrawToken`, tokens were not funded, or external drain happened).

Impact

Users and integrators relying on `pendingRewards` for accounting, UI display, or composable logic will see an inflated claimable amount compared to what is actually transferred by distribute, potentially causing wrong accounting, UX confusion, failed payout expectations, or downstream logic errors; in the worst case this can result in failed economic assumptions or misreporting of earned rewards and the shortfall can be as large as the difference between the theoretical pending amount and the contract's `rewardToken` balance (i.e., up to the entire intended accrual amount or effectively the whole contract reward balance).

Recommendation

Make `pendingRewards` reflect available tokens (UI-friendly): clamp the returned value by the current `rewardToken` balance.

```
 function pendingRewards() public view override returns (uint256) {
                    ...
-                                                                          return
timeDiff.mul(supply).mul(bonusMultiplierBasisPoints).div(BASIS_POINTS_DIVISOR).div(BONUS_
DURATION);
+                                                  uint256       theoretical       =
timeDiff.mul(supply).mul(bonusMultiplierBasisPoints).div(BASIS_POINTS_DIVISOR).div(BONUS_
DURATION);
+        uint256 balance = IERC20(rewardToken).balanceOf(address(this));
+        return theoretical > balance ? balance : theoretical;
    }
```

Status

Resolved

[M-02] RewardTracker#_updateRewards -Rewards Locked When totalSupply Is Zero

Description

When `RewardTracker._updateRewards` calls the distributor's `distribute` while `totalSupply == 0`, any `blockReward` transferred to `RewardTracker` is never applied to `cumulativeRewardPerToken` (division by zero avoided) and therefore remains stuck in the `RewardTracker` contract. This occurs whenever the distributor distributes between reward updates and no stakers exist (e.g., early funding, all users unstaked, or temporary supply gap). The result is permanently or indefinitely locked reward tokens unless governance intervenes.

Vulnerability Details

`_updateRewards` always pulls rewards from the distributor (`IRewardDistributor(distributor).distribute()`), then only applies them if `supply > 0`. There is no handling path for `supply == 0 && blockReward > 0`, so funds remain in the contract balance but are never credited to any user. Relevant snippet:

```
uint256 blockReward = IRewardDistributor(distributor).distribute();

uint256 supply = totalSupply;
uint256 _cumulativeRewardPerToken = cumulativeRewardPerToken;

if (supply > 0 && blockReward > 0) {
                                                _cumulativeRewardPerToken                =
_cumulativeRewardPerToken.add(blockReward.mul(PRECISION).div(supply));
    cumulativeRewardPerToken = _cumulativeRewardPerToken;
}

// cumulativeRewardPerToken can only increase
if (_cumulativeRewardPerToken == 0) {
    return;
}
```

Because the function returns when `_cumulativeRewardPerToken == 0`, any `blockReward` received while `supply == 0` is not distributed and remains held by the contract, the distributor (and `pendingRewards` logic in some distributors) may consider those rewards distributed, creating a permanent mismatch between distributed tokens and recorded accounting.

Proof of Concept

```
function testRewardsLockedWithZeroSupply() public {
    // --- Setup and Preconditions ---
    // Generate test address for alice (future staker)
    address alice = address(0x1);
    // Verify RewardTracker starts with zero total supply
    assertEq(stakedZusTracker.totalSupply(), 0);
    // PoC succeeds if initial supply is zero
    // --- Exploit Execution ---
    // Time passes with zero supply, causing rewards to accumulate
    vm.warp(block.timestamp + 1 days);
    // Trigger pending rewards calculation while supply is still zero
    stakedZusDistributor.pendingRewards();
    // Mint ZUS tokens to alice for future staking
    zus.mint(alice, 1e18);
    // Alice stakes after rewards have already accumulated with zero supply
    vm.startPrank(alice);
    zus.approve(address(stakedZusTracker), 1e18);
    stakedZusTracker.stake(address(zus), 1e18);
    vm.stopPrank();
    // --- Post-condition Check ---
    // Alice has no claimable rewards despite accumulated tokens in contract
    assertEq(stakedZusTracker.claimable(alice), 0);
    // PoC succeeds if alice cannot claim any rewards
    // Reward tokens are locked in RewardTracker contract forever
    assertTrue(zus.balanceOf(address(stakedZusTracker)) > 0);
    // PoC succeeds if tokens are trapped in contract
}
```

Impact

If the distributor emits rewards while `totalSupply == 0`, all such rewards are transferred into the `RewardTracker` contract but never applied to users; those tokens are effectively locked until a privileged actor manually `recovers` them (e.g., via `withdrawToken`) or the protocol admin corrects the state. This can result in permanently stranded reward balances equal to the sum of all `blockReward` transfers that occurred during zero-supply windows — potentially large amounts (the full funded reward pool for periods with no stakers) — and causes misalignment between on-chain token balances and accounting, broken user UX, and loss of trust or token availability unless governance intervenes.

Recommendation

Immediate safe return: if `supply == 0` and `blockReward > 0`, return the `blockReward` to the distributor (or governance/reserve). This prevents locking and keeps accounting consistent.

Status

Resolved

[M-03] StakedZlp -Inconsistent State Management and Access Control Bypass

Description

The StakedZlp contract suffers from state management vulnerabilities due to unrestricted access to underlying `RewardTracker` contracts. Users can bypass cooldown periods, and partial staking creates inconsistencies between `balanceOf` calculations and transfer capabilities. This issue also occurs in the ZlpBalance contract.

Vulnerability Details

Three interconnected issues exist: users can bypass cooldown restrictions via direct `stakedZlpTracker.transfer` calls; `balanceOf` only checks `feeZlpTracker` while ignoring stakedZlpTracker-only positions; and `transfer` logic assumes complete staking but fails with partial staking. `RewardTracker` contracts default to `inPrivateTransferMode = false`, enabling unrestricted access.

```
// StakedZlp.balanceOf() - Only checks feeZlpTracker deposits
function balanceOf(address _account) external view returns (uint256) {
    return IRewardTracker(feeZlpTracker).depositBalances(_account, zlp);
    // Returns 0 if user staked directly to stakedZlpTracker only
}
// StakedZlp._transfer() - Assumes complete staking stack
function _transfer(address _sender, address _recipient, uint256 _amount) private {
    require(
                    IZLP(zlp).lastAddedAt(_sender).add(IZLP(zlp).cooldownDuration())   <=
block.timestamp,
        "StakedZlp: cooldown duration not yet passed"
    );
    // This unstaking sequence fails if sender only staked to stakedZlpTracker
     IRewardTracker(stakedZlpTracker).unstakeForAccount(_sender, feeZlpTracker, _amount,
address(this));
}
// RewardTracker.transfer() - No cooldown checks when inPrivateTransferMode = false
function transfer(address _recipient, uint256 _amount) external override returns (bool) {
    _transfer(msg.sender, _recipient, _amount); // Bypasses all StakedZlp protections
    return true;
}
```

Impact

These vulnerabilities completely undermine StakedZlp's security model, allowing users to bypass cooldown periods and creating state inconsistencies where `balanceOf` returns incorrect values. This enables front-running attacks, liquidity manipulation, broken user interfaces, failed transactions, and potential fund trapping in unusable states.

Recommendation

Initialize RewardTracker contracts in private mode during deployment and implement comprehensive state validation

Status

Resolved

Low

[L-01] Contract#withdrawToken - No native ETH rescue function (only ERC20)

Description

Several contracts implement `withdrawToken(address _token, address _account, uint256 _amount)`, which allows gov to recover ERC-20 tokens accidentally sent to the contract. However, there is no equivalent function to recover native ETH (msg.value) sent to these contracts. As a result, accidentally sent ETH can become stranded in the contract balance with no on-chain recovery path by governance.

Recommendation

Add a dedicated native-ETH rescue function guarded by `onlyGov` that uses a safe call pattern to forward ETH to a designated recipient; alternatively, add a single `withdraw(address _tokenOrZero, address _account, uint256 _amount)` that treats `_tokenOrZero == address(0)` as native ETH.

Status

Resolved

[L-02] StakedZlp, ZlpBalance#transferFrom - Missing Special Handling for uint256.max Allowance

Description

In the `transferFrom` function, there is no special handling for the case when the allowance is set to `type(uint256).max`. Normally, many ERC20 implementations skip decreasing the allowance in this scenario to save gas and prevent unnecessary state changes. The current implementation always decreases the allowance, which may be inefficient.

Recommendation

Consider skipping the allowance decrement when it is `type(uint256).max`:

```
- _approve(_sender, msg.sender, nextAllowance);
+ if (allowances[_sender][msg.sender] != type(uint256).max) {
+     _approve(_sender, msg.sender, nextAllowance);
+ }
```

Status

Resolved

QA

[Q-01] BonusDistributor#pendingRewards - Inconsistent Calculation Method

Description

The `pendingRewards` function re-implements the reward rate logic instead of reusing the `tokensPerInterval` calculation. This results in duplicated code paths that may diverge over time, creating inconsistencies between expected reward flow and the actual per-interval reward rate. Using a single source of truth (`tokensPerInterval`) would improve clarity and reduce the risk of logical drift.

Recommendation

Refactor `pendingRewards` to directly use `tokensPerInterval` for consistency and maintainability. Minimal suggested change:

```
-return
timeDiff.mul(supply).mul(bonusMultiplierBasisPoints).div(BASIS_POINTS_DIVISOR).div(BONUS_
DURATION);

+ return timeDiff.mul(tokensPerInterval());
```

Status

Resolved

[Q-02] RewardRouter#claim - Suggestion for Using Dedicated Claim Functions

Description

The claim function directly calls all underlying reward trackers instead of using the dedicated claimEsZus and claimFees functions. This duplication reduces readability and increases maintenance complexity, as changes to the dedicated claim functions would not automatically propagate to the claim. Using the existing helper functions would make the reward claiming process more intuitive and consistent.

Recommendation

Refactor claim to call the dedicated helper functions:

```
function claim() external nonReentrant {
    address account = msg.sender;
-   IRewardTracker(feeZusTracker).claimForAccount(account, account);
-   IRewardTracker(feeZlpTracker).claimForAccount(account, account);
-   IRewardTracker(stakedZusTracker).claimForAccount(account, account);
-   IRewardTracker(stakedZlpTracker).claimForAccount(account, account);
+   claimEsZus();
+   claimFees();
}
```

Status

Resolved

[Q-03] RewardRouter#handleRewards - Unused Function Parameter

Description

The handleRewards function includes a boolean parameter `_shouldClaimWeth`, which appears unnecessary, as the internal logic always assumes WETH claiming is desired and directly references `_shouldConvertWethToEth` to control ETH conversion. Keeping this parameter may confuse users and developers, suggesting optional behavior that does not exist, reducing code clarity and maintainability.

Recommendation

Remove the `_shouldClaimWeth` parameter and simplify the function:

```
- function handleRewards(
-     // bool _shouldStakeMultiplierPoints,
-     bool _shouldClaimWeth,
-     bool _shouldConvertWethToEth
- ) external nonReentrant {
+ function handleRewards(
+     // bool _shouldStakeMultiplierPoints,
+     bool _shouldConvertWethToEth
+ ) external nonReentrant {
-     if (_shouldClaimWeth) {
+     // WETH claiming is always performed
```

Status

Resolved

[Q-04] RewardTracker - Confusing Variable Name

Description

The `depositBalances` mapping is named in a way that may confuse developers, as it tracks deposited amounts per user and token but the current name does not clearly convey this. Ambiguous variable names can lead to misuse or misunderstandings when reading or interacting with the contract.

Recommendation

Rename the variable to improve clarity:

```
- mapping(address => mapping(address => uint256)) public override depositBalances;
+ mapping(address => mapping(address => uint256)) public override depositedBalances;
```

Status

Resolved

[Q-05] StakedZlp - Unnecessary Public Getter

Description

The allowances mapping already has a dedicated getter function, so declaring it public is redundant. Exposing it publicly without need can lead to confusion or unintended reliance on the storage layout. It is recommended to make it internal or private to align with best practices.

Recommendation

Update the visibility of the mapping:

```
- mapping(address => mapping(address => uint256)) public allowances;
+ mapping(address => mapping(address => uint256)) internal allowances;
```

Status

Resolved

[Q-06] ZlpBalance - File Name Mismatch

Description

The contract `ZlpBalance` is implemented in a file named `ZusBalance.sol`, which can cause confusion for developers and auditors. Maintaining consistent naming between the contract and the file is important for readability, maintainability, and to avoid deployment mistakes.

Recommendation

Rename the file to match the contract name.

Status

Resolved

Gas

[G-01] StakedZlp - Gas Optimization on stakedAmounts Reads

Description

The code repeatedly reads `stakedAmounts[_account]` multiple times within the same scope. Each SSTORE or SLOAD operation consumes gas, so reading the mapping once and caching the value in a local variable can reduce gas costs and improve efficiency.

Recommendation

Cache `stakedAmounts[_account]` in a local variable and reuse it:

```
- if (_account != address(0)) {
-     uint256 stakedAmount = stakedAmounts[_account];
-     ...
-     if (_claimableReward > 0 && stakedAmounts[_account] > 0) {
-                                  ...                  averageStakedAmounts[_account]    =
averageStakedAmounts[_account].mul(cumulativeRewards[_account]).div(
-             nextCumulativeReward
-         ).add(stakedAmount.mul(accountReward).div(nextCumulativeReward));
-         cumulativeRewards[_account] = nextCumulativeReward;
-     }}
+ if (_account != address(0)) {
+     uint256 stakedAmount = stakedAmounts[_account];
+     ...
+     if (_claimableReward > 0 && stakedAmount > 0) {
+                              ...                       averageStakedAmounts[_account]   =
averageStakedAmounts[_account].mul(cumulativeRewards[_account]).div(
+             nextCumulativeReward
+         ).add(stakedAmount.mul(accountReward).div(nextCumulativeReward));
+         cumulativeRewards[_account] = nextCumulativeReward;
+     } }
```

Status

Resolved

Centralisation

The Zeus Exchange project values security and utility over decentralisation.

The owner executable functions within the protocol increase security and functionality but depend highly on internal team responsibility.

Conclusion

After Hashlock's analysis, the Zeus Exchange project seems to have a sound and well-tested code base, now that our vulnerability findings have been resolved. Overall, most of the code is correctly ordered and follows industry best practices. The code is well commented on as well. To the best of our ability, Hashlock is not able to identify any further vulnerabilities.

Our Methodology

Hashlock strives to maintain a transparent working process and to make our audits a collaborative effort. The objective of our security audits is to improve the quality of systems and upcoming projects we review and to aim for sufficient remediation to help protect users and project leaders. Below is the methodology we use in our security audit process.

Manual Code Review:

In manually analysing all of the code, we seek to find any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behaviour when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our methodologies include manual code analysis, user interface interaction, and white box penetration testing. We consider the project's website, specifications, and whitepaper (if available) to attain a high-level understanding of what functionality the smart contract under review contains. We then communicate with the developers and founders to gain insight into their vision for the project. We install and deploy the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We undergo a robust, transparent process for analysing potential security vulnerabilities and seeing them through to successful remediation. When a potential issue is discovered, we immediately create an issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is vast because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, and then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyse the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the contract details are made public.

Disclaimers

Hashlock's Disclaimer

Hashlock's team has analysed these smart contracts in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in the smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

Due to the fact that the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Hashlock is not responsible for the safety of any funds and is not in any way liable for the security of the project.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to attacks. Thus, the audit can't guarantee the explicit security of the audited smart contracts.

About Hashlock

Hashlock is an Australian-based company aiming to help facilitate the successful widespread adoption of distributed ledger technology. Our key services all have a focus on security, as well as projects that focus on streamlined adoption in the business sector.

Hashlock is excited to continue to grow its partnerships with developers and other web3-oriented companies to collaborate on secure innovation, helping businesses and decentralised entities alike.

Website: hashlock.com.au
Contact: info@hashlock.com.au