MiloTruck

Rodeo Finance

Security Review

January, 2024

Contents

Introduction **2**
- About MiloTruck 2
- Disclaimer 2

Risk Classification **3**
- Severity Level 3
- Impact 3
- Likelihood 3

Executive Summary **4**
- About Rodeo Finance 4
- Overview 4
- Scope 4
- Issues Found 4

Findings **5**
- Summary 5
- High Severity Findings 6
  - H-01: `onERC721Received()` doesn't validate the `from` address 6
- Medium Severity Findings 7
  - M-01: `_earn()` isn't called before `_mint()` and `_burn()` 7
  - M-02: Avoid using `balanceOf` for deposit amounts in `quoteAndSwap()` 8
  - M-03: Missing functionality to withdraw spNFT from `nitroPool` 9
  - M-04: `NFTPool.withdrawFromPosition()` reverts when the spNFT is deposited in a nitro pool 10
  - M-05: Missing call to `NitroPool.harvest()` in `_earn()` 11
  - M-06: `_earn()` cannot handle nitro pools with two reward tokens 11
  - M-07: xGRAIL rewards are not distributed to users 12
  - M-08: TWAP is less resistant to manipulation on Arbitrum 12
  - M-09: `tokenId` is not reset in `unstake()` when the spNFT is burned 13
- Low Severity Findings 14
  - L-01: `twapPeriod` should never exceed `int32.max` 14
- Informational Findings 14
  - I-01: `UniProxy.deposit()` contains override deposit checks 14

Introduction

About MiloTruck

MiloTruck is an independent security researcher who specializes in smart contract audits. Currently, he works as a Senior Auditor at Trust Security and Associate Security Researcher at Spearbit. He is also one of the top wardens on Code4rena.

For private audits or security consulting, please reach out to him on:

- Twitter - @milotruck

You can also request a quote on Code4rena or Cantina to engage them as an intermediary.

Disclaimer

A smart contract security review **can never prove the complete absence of vulnerabilities**. Security reviews are a time, resource and expertise bound effort to find as many vulnerabilities as possible. However, they cannot guarantee the absolute security of the protocol in any way.

Risk Classification

Severity Level

	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	High	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- High - Funds are **directly** at risk, or a **severe** disruption of the protocol's core functionality.
- Medium - Funds are **indirectly** at risk, or **some** disruption of the protocol's functionality.
- Low - Funds are **not** at risk.

Likelihood

- High - Highly likely to occur.
- Medium - Might occur under specific conditions.
- Low - Unlikely to occur.

Executive Summary

About Rodeo Finance

Rodeo is a decentralized finance protocol that allows its community to earn yield on a range of managed and passive investment strategies on Arbitrum and the greater DeFi ecosystem. At its core, Rodeo offers Boosted Yield Farming and Structured Yield Products.

Boosted Yield Farming allows liquidity providers and farmers who seek boosted yield to stake their yield bearing assets for use in the Rodeo Farms, thus maintaining exposure to the underlying asset and earning additional yields in the farms.

Structured Yield Products allow Rodeo to innovate on top of existing DeFi infrastructure to create novel products to boost, leverage, automate and capture the best DeFi opportunities and narratives.

Overview

Project Name	Rodeo Finance (Camelot V3 Strategy)
Project Type	Yield, Camelot V3 Integration
Repository	https://github.com/rodeofi/rodeo
Commit Hash	6832bf08243e7e64b74a51dcbff51a002a1b1faf

Scope

- contracts/src/strategies/StrategyCamelotV3.sol

Issues Found

Severity	Count
High	1
Medium	9
Low	1
Informational	1

Findings

Summary

ID	Description	Severity
H-01	onERC721Received() doesn't validate the from address	High
M-01	_earn() isn't called before _mint() and _burn()	Medium
M-02	Avoid using balanceOf for deposit amounts in quoteAndSwap()	Medium
M-03	Missing functionality to withdraw spNFT from nitroPool	Medium
M-04	NFTPool.withdrawFromPosition() reverts when the spNFT is deposited in a nitro pool	Medium
M-05	Missing call to NitroPool.harvest() in _earn()	Medium
M-06	_earn() cannot handle nitro pools with two reward tokens	Medium
M-07	xGRAIL rewards are not distributed to users	Medium
M-08	TWAP is less resistant to manipulation on Arbitrum	Medium
M-09	tokenId is not reset in unstake() when the spNFT is burned	Medium
L-01	twapPeriod should never exceed int32.max	Low
I-01	UniProxy.deposit() contains override deposit checks	Informational

High Severity Findings

H-01: `onERC721Received()` doesn't validate the `from` address

Description

The `onERC721Received()` callback only checks if `msg.sender` is the NFT pool address, and assigns `tokenId` if so:

StrategyCamelotV3.sol#L297-L299

```
function onERC721Received(...) external returns (bytes4) {
    if (msg.sender == address(nftPool) && tokenId == 0) {
        tokenId = _tokenId;
    }
```

However, this allows attackers to set their own spNFT as `tokenId` by sending it directly to the `StrategyCamelotV3` contract.

An attacker can cause a loss of funds for users by:

- Create an spNFT with a maximum `lockDuration` (currently 183 days)
- Sending it directly to the strategy contract before the first deposit, which sets `tokenId` to the spNFT in `onERC721Received()`
- All subsequent deposits with `_mint()` will call `NFTPool.addToPosition()` with the attacker's spNFT, which resets the lock duration to its maximum
- As such, users will be unable to withdraw their funds with `_burn()` since the lock duration will never expire, causing `withdrawFromPosition()` to always revert here

Recommendation

Consider checking the `from` address in `onERC721Received()`:

```
function onERC721Received(address, address from, ...) external returns (bytes4) {
    if (from != address(0) && from != address(nitroPool) && from != previousStrategy) {
        revert InvalidFrom();
    }

    if (msg.sender == address(nftPool) && tokenId == 0 ) {
        tokenId = _tokenId;
    }
```

This ensures that the spNFT can only be from:

- `address(0)` - Minted by the NFT pool in `NFTPool.createPosition()`
- `nitroPool` - Sent from the nitro pool when calling `NitroPool.withdraw()`
- `previousStrategy` - Sent from a previous Camelot V3 strategy during migration

Rodeo Finance: Acknowledged. We will perform the first deposit into the strategy to ensure `tokenId` does not belong to a malicious spNFT.

Medium Severity Findings

M-01: `_earn()` isn't called before `_mint()` and `_burn()`

Description

In the strategy, `_earn()` is used to harvest rewards from the NFT and nitro pools, while `_mint()` and `_burn()` are called by users to deposit and withdraw funds respectively.

However, since `_earn()` is not called before user deposits and withdrawals are processed in `_mint()` and `_burn()`, the share calculation in both functions will not factor in accrued rewards that have not been harvested yet.

An attacker can exploit this to steal the strategy's rewards with a flashloan as such:

- Borrow a huge flashloan of `targetAsset`.
- Call `_mint()` to deposit all his `targetAsset`.
- Call `_earn()` to harvest rewards and deposit them into the strategy's position.
- Call `_burn()` to withdraw all his shares. Since the attacker used a flashloan, he will hold most of the shares in the strategy. As such, most of the harvested rewards accrued to his shares and will be withdrawn by him.
- Repay the flashloan.

Additionally, users that do not call `_earn()` before `_burn()` will lose out on rewards that have not been harvested yet.

Recommendation

Call `_earn()` before performing any logic in `_mint()` and `_burn()`:

```
function _mint(...) internal override returns (uint256) {
    _earn();
    // Some code here
}

function _burn(...) internal override returns (uint256) {
    _earn();
    // Some code here
}
```

This ensures that all rewards will be harvested and re-deposited into the strategy's position before processing any user's deposit/withdrawal.

Rodeo Finance: Acknowledged. `earn()` is permissioned and will be called at random times to prevent attackers from attempting to sandwich the call to `earn()` as described above. Note that front-running is also not possible since the protocol is deployed on Arbitrum.

M-02: Avoid using `balanceOf` for deposit amounts in `quoteAndSwap()`

Description

When depositing into Gamma's hypervisors through `UniProxy`, the amount of `token0` and `token1` deposited must be within a certain ratio, as seen from here. Therefore, `quoteAndSwap()` uses `UniProxy.getDepositAmount()` to adjust the token amounts before depositing.

However, it uses `token.balanceOf()` to fetch the adjusted token amounts after swapping:

StrategyCamelotV3.sol#L240-L241

```
amt0 = hypervisor.token0().balanceOf(address(this));
amt1 = hypervisor.token1().balanceOf(address(this));
```

This allows an attacker to DOS deposits by directly sending `token0` or `token1` to the contract:

- Assume the following:
  - The deposit ratio for `token0` : `token1` must be around 1
  - `token0` is USDC and `token1` is WETH
- An attacker directly sends 500 USDC to the contract.
- A user deposits 4000 USDC. In `quoteAndSwap()`:
  - His deposit is split into 2000 USDC and 1 WETH (assume 1 WETH = 2000 USDC)
  - Since `balanceOf()` is used, `amt0` includes the amount sent by the attacker and becomes 2500 USDC
  - The user's deposit fails as his `token0` : `token1` ratio is out of the acceptable range

Recommendation

Use the token amounts returned by `quoteAddLiquidity()` and the amount returned from the swap:

StrategyCamelotV3.sol#L234-L241

```
if (trgtAst == token0) {
    uint256 balanceBefore = hypervisor.token1().balanceOf(address(this));
    swap(trgtAst, address(hypervisor.token1()), path, toLp1, slp);

    amt0 = toLp0;
    amt1 = hypervisor.token1().balanceOf(address(this)) - balanceBefore;
} else {
    uint256 balanceBefore = hypervisor.token0().balanceOf(address(this));
    swap(trgtAst, token0, path, toLp0, slp);

    amt0 = hypervisor.token0().balanceOf(address(this)) - balanceBefore;
    amt1 = toLp1;
}
```

Rodeo Finance: Fixed in commit 5208af9 as recommended.

M-03: Missing functionality to withdraw spNFT from `nitroPool`

Description

The admin is able to deposit the strategy's spNFT into a nitro pool with `setNitroPool()` for additional rewards.

However, when the nitro pool's reward period expires, there is no functionality to withdraw the spNFT from that nitro pool. Currently, the only way to withdraw is to call `_exit()`, which should only be used to migrate to a new strategy contract.

As such, once the strategy's spNFT has been deposited into a nitro pool, it can never be moved to another nitro pool.

Recommendation

Modify `setNitroPool()` to handle spNFT withdrawals:

StrategyCamelotV3.sol#L82-L86

```
function setNitroPool(address _nitroPool) external auth {
    if (tokenId == 0) revert TokenIdNeededFirst();

    if (_nitroPool == address(0)) {
        nitroPool.withdraw(tokenId);
    } else {
        nftPool.safeTransferFrom(address(this), _nitroPool, tokenId, "");
    }

    nitroPool = INitroPool(_nitroPool);
}
```

Rodeo Finance: Fixed in commit f170a71 as recommended.

M-04: `NFTPool.withdrawFromPosition()` reverts when the spNFT is deposited in a nitro pool

Description

The admin is able to deposit the strategy's spNFT into a nitro pool with `setNitroPool()` for additional rewards.

However, when an spNFT is deposited in a nitro pool, users are not allowed to withdraw from the spNFT, as seen in this callback.

This will cause `unstake()` to revert as it calls `withdrawFromPosition()` directly without withdrawing the spNFT from the nitro pool first:

StrategyCamelotV3.sol#L150-L152

```
function unstake(uint256 amount) internal {
    nftPool.withdrawFromPosition(tokenId, amount);
}
```

Therefore, when the strategy's spNFT is deposited in a nitro pool, users will be unable to withdraw their funds.

Recommendation

Withdraw the spNFT from `nitroPool` before calling `withdrawFromPosition()`, and re-deposit it afterwards:

```
function unstake(uint256 amount) internal {
    if (address(nitroPool) != address(0)) {
        nitroPool.withdraw(tokenId);
    }

    nftPool.withdrawFromPosition(tokenId, amount);

    if (address(nitroPool) != address(0)) {
        nftPool.safeTransferFrom(address(this), address(nitroPool), tokenId, "");
    }
}
```

Rodeo Finance: Fixed in commit 0961992 and commit 38da4c4 as recommended.

M-05: Missing call to `NitroPool.harvest()` in `_earn()`

Description

The admin is able to deposit the strategy's spNFT into a nitro pool with `setNitroPool()` for additional rewards.

However, `_earn()` does not call `NitroPool.harvest()` before accruing the strategy's rewards. As such, rewards from the nitro pool will be left unclaimed and will not be redistributed to depositors.

Recommendation

Call `nitroPool.harvest()` in `_earn()` as such:

StrategyCamelotV3.sol#L158

```
  nftPool.harvestPosition(tokenId);
+ nitroPool.harvest();
```

Rodeo Finance: Fixed in commit fb1baf5 as recommended.

M-06: `_earn()` cannot handle nitro pools with two reward tokens

Description

When the strategy's spNFT is deposited in a nitro pool, it will receive the following tokens as rewards:

- GRAIL and xGRAIL tokens from the NFT pool.
- Up to two reward tokens from the nitro pool.

However, the admin can only configure two reward tokens to be accrued in `_earn()`, namely `rewardToken1` and `rewardToken2`. This is problematic for nitro pools with two reward tokens, since the contract is not designed to handle two reward tokens + GRAIL from the NFT pool.

Recommendation

In `_earn()`, add logic to handle GRAIL alongside `rewardToken1` and `rewardToken2`:

```
uint256 grailBalance = GRAIL.balanceOf(address(this));
if (strategyHelper.value(GRAIL, grailBalance) > 1e18) {
    GRAIL.approve(address(strategyHelper), grailBalance);
    strategyHelper.swap(GRAIL, asset, grailBalance, slp, address(this));
}
```

This is because the strategy contract will always receive GRAIL as rewards, regardless of whether the spNFT is staked in a nitro pool or not.

Rodeo Finance: Fixed in commit 588475f by adding a third reward token.

M-07: xGRAIL rewards are not distributed to users

Description

When the strategy's spNFT is staked in a NFT pool, it will receive xGRAIL rewards alongside GRAIL.

However, `_earn()` does not contain any logic to handle xGRAIL rewards from the NFT pool. As such, they will remain stuck in the contract and will not be distributed to users.

Recommendation

Although xGRAIL is non-transferable, Camelot's documentation describes a few ways to utilize xGRAIL.

For example, consider adding functions for the admin to call `xGRAIL.redeem()` and `xGRAIL.finalizeRedeem()`. This allows the admin to redeem xGRAIL for GRAIL, which can then be swapped to `targetAsset` and deposited into the strategy's position in `_earn()`.

Rodeo Finance: Fixed in commit 7688fa2 by adding admin functions to call `xGRAIL.redeem()` and `xGRAIL.finalizeRedeem()`.

M-08: TWAP is less resistant to manipulation on Arbitrum

Description

The `valueLiquidity()` function takes the TWAP `sqrtPriceX96` value to calculate the strategy's current liquidity in the pool:

StrategyCamelotV3.sol#L251-L258

```
uint32 period = twapPeriod;
uint32[] memory secondsAgos = new uint32[](2);

secondsAgos[0] = period;
secondsAgos[1] = 0;

(int56[] memory tickCumulatives,,,) = hypervisor.pool().getTimepoints(secondsAgos);
uint160 midX96 = TickMath.getSqrtRatioAtTick(int24((tickCumulatives[1] - tickCumulatives[0]) /
int32(period)));
```

However, TWAP is less resistant to price manipulation on L2s since they tend to have a higher block frequency. More specifically, Arbitrum produces one block every 0.25s as compared to Ethereum's one block per 12s.

As such, it is less costly for an attacker to control more blocks in a `twapPeriod`, making it easier to manipulate `sqrtPriceX96`.

Recommendation

Consider setting `twapPeriod` to a higher value to be more resistant against TWAP manipulation.

Rodeo Finance: Fixed in commit 1ad73b8 by increasing `twapPeriod` to `0.5 days`.

M-09: `tokenId` is not reset in `unstake()` when the spNFT is burned

Description

When a position does not have any amount left after `NFTPool.withdrawFromPosition()` is called, its corresponding spNFT will be burned:

NFTPool.sol#L848-L852

```
if (position.amount == 0) {
    // destroy if now empty
    _lpSupplyWithMultiplier = _lpSupplyWithMultiplier.sub(position.amountWithMultiplier);
    _destroyPosition(tokenId, position.boostPoints);
} else {
```

Note that `_destroyPosition()` burns the position's spNFT. However, `unstake()` does not account for this:

StrategyCamelotV3.sol#L150-L152

```
function unstake(uint256 amount) internal {
    nftPool.withdrawFromPosition(tokenId, amount);
}
```

Therefore, if `withdrawFromPosition()` is ever called by a user to withdraw all of the strategy's remaining position, the strategy's spNFT will be burned but `tokenId` will not be updated.

This will DOS all future deposits as `_mint()` will attempt to call `NFTPool.addToPosition()` with a non-existent spNFT, causing it to revert.

Recommendation

In `unstake()`, reset `tokenId` to `0` when the `amount` is equal to the strategy's entire position:

```
  function unstake(uint256 amount) internal {
      nftPool.withdrawFromPosition(tokenId, amount);
+     if (amount == totalManagedAssets()) {
+         delete tokenId;
+     }
  }
```

Rodeo Finance: Fixed in commit d5bb2f1 by resetting `tokenId` to `0` in `stake()` when `totalManagedAssets()` is `0`. This also prevents the issue of someone sending their own spNFT directly to the strategy contract, as mentioned in H-01.

Low Severity Findings

L-01: `twapPeriod` should never exceed `int32.max`

Description

`valueLiquidity()` contains an unsafe cast of `period` from a `uint32` to `int32`:

StrategyCamelotV3.sol#L258

```
uint160 midX96 = TickMath.getSqrtRatioAtTick(int24((tickCumulatives[1] - tickCumulatives[0]) /
int32(period)));
```

Therefore, if `twapPeriod` ever exceeds `int32.max`, the type cast above will overflow, causing `valueLiquidity()` to return incorrect values.

Recommendation

Check that `twapPeriod` cannot be set to larger than `int32.max` in `setTwapPeriod()`:

StrategyCamelotV3.sol#L78-L80

```
  function setTwapPeriod(uint32 newTwapPeriod) external auth {
+     if (newTwapPeriod > type(int32).max) revert TwapPeriodTooLong();
      twapPeriod = newTwapPeriod;
  }
```

Rodeo Finance: Fixed in commit 98369ce as recommended.

Informational Findings

I-01: `UniProxy.deposit()` contains override deposit checks

`clearDeposit()`, which is called by `UniProxy.deposit()` to check the deposit amounts of `token0` and `token1`, can be overridden by the admin to perform the following checks:

Clearing.sol#L142-L149

```
if (p.depositOverride) {
  if (p.deposit0Max > 0) {
    require(deposit0 <= p.deposit0Max, "token0 exceeds");
  }
  if (p.deposit1Max > 0) {
    require(deposit1 <= p.deposit1Max, "token1 exceeds");
  }
}
```

If the admin of `UniProxy` ever sets `deposit0Max` or `deposit1Max` to small values, attempting to deposit funds into the hypervisor in `_mint()` might revert.