Marginal
Security Review

Cantina Managed review by:
Jeiwan, Security Researcher
Gmhacker, Associate Security Researcher

February 24, 2024

Contents

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity	Description
Critical	*Must* fix as soon as possible (if already deployed).
High	Leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
Medium	Global losses <10% or losses to only a subset of users, but still unacceptable.
Low	Losses will be annoying but bearable. Applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.
Gas Optimization	Suggestions around gas saving practices.
Informational	Suggestions around best practices or readability.

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Marginal is a permissionless spot and perpetual exchange that enables leverage on assets with an Uniswap V3 Oracle.

From Feb 12th to Feb 20th the Cantina team conducted a review of v1-periphery on commit hash 1d4c6a63. The team identified a total of **16** issues in the following risk categories:

- Critical Risk: 1
- High Risk: 2
- Medium Risk: 4
- Low Risk: 1
- Gas Optimizations: 3
- Informational: 5

3 Findings

3.1 Critical Risk

3.1.1 Uniswap liquidity can be stolen during migration

Severity: Critical Risk

Context: V1Migrator.sol#L39

Description: The V1Migrator.migrate() function lets users migrate their Uniswap V3 liquidity positions to Marginal. On behalf of a user, the function removes liquidity from a Uniswap position, withdraws tokens, adds them to a Marginal pool and refunds unspent token amounts. Since Uniswap positions are wrapped in NFT tokens, the above operations require the `V1Migrator` contract to be approved by a token owner to manage the token.

However, the function doesn't check that the migration is initiated by the token owner. As a result, a malicious actor (e.g. an MEV bot) can back-run the Uniswap position token approval transaction to steal the tokens backing the position. The malicious actor can call the `migrate()` function and specify their address as the owner of the liquidity in the Marginal pool (V1Migrator.sol#L103) and/or receive the refunded amounts (which are sent to the caller: V1Migrator.sol#L119-L137).

Recommendation: In the `V1Migrator.migrate()` function, consider reverting if the caller is not the owner of the Uniswap token to be migrated. The owner of a token can be obtained via the ERC721.ownerOf() function.

Marginal: Agreed. Fixed in commit 0a4f4847.

Cantina Managed: Fixed as recommended.

3.2 High Risk

3.2.1 `Router` doesn't refund unspent ETH after swapping

Severity: High Risk

Context: Router.sol#L277, Router.sol#L308

Description: Router allows swapping of ETH for ERC20 tokens. The difference between selling ETH and an ERC20 token is that the contract can compute and request from the user the exact amount of ERC20 tokens to sell, but, when selling ETH, the user has to send the entire amount when making the call (i.e. before the actual amount was computed in the contract).

However, there's a scenario when sent ETH is consumed partially and not refunded: in exact output swaps, positive slippage can improve the price of the swap, making the swap more profitable for the user. But since this is an exact output swap, the improved price will result in less ETH sold in the swap. The unused ETH will remain in the contract and can be withdrawn by anyone else.

Recommendation: In `Router.exactOutputSingle()` and `Router.exactOutput()` functions, consider returning unspent ETH to the caller at the end of the functions. The `PeripheryPayments.refundETH()` function can be used for that.

Marginal: Agreed. Fixed in commit 5f60a945. Also refunded when adding liquidity, since this issue also occurs there I think.

Cantina Managed: Fixed as recommended.

3.2.2 Read-only reentrancy allows the transfer of ownership of a position token right before it gets burned

Severity: High Risk

Context: NonfungiblePositionManager.sol#L319-L339

Description: `NonfungiblePositionManager.burn()` is used to settle a margin position and burn the token representation in the manager. The `rewards` Ether gets transferred to the `recipient` input address before the position gets settled and the token gets burnt. If there's a market leveraging the `NonfungiblePositionManager` position token, with a normal pattern requesting the user to transfer the position token to the contract, the attacker can successfully burn the token but deposit it into that market before/during a `burn`, extracting value this way.

Here's a simple breakdown:

1. User calls `NonfungiblePositionManager.burn`. It will eventually call `pool.settle`.
2. The pool will transfer Ether to the `recipient` address. Having the execution control, the recipient transfer the token to a third-party platform to sell it or use as collateral.
3. The position gets settled in the pool.
4. The position token is burned.

The same attack vector can be used in `NonfungiblePositionManager.ignite()`, which also burns the token after settling the position in the pool.

Recommendation: Delete the position from the `_positions` mapping and burn the respective token before calling `pool.settle`, both in `burn` and in `ignite`.

Marginal: Agreed. Fixed in commit a8a34dc9.

Cantina Managed: Fixed in commit a8a34dc9 as recommended.

3.3 Medium Risk

3.3.1 Swap fee amount miscalculation during pool initialization

Severity: Medium Risk

Context: PoolInitializer.sol#L248

Description: The PoolInitializer.initializePoolSqrtPriceX96() function initializes a Marginal pool by performing as swap so that the current price of the pool after the swap equals the desired price. The token amounts for the swap are computed to reach the desired price specified in the `params.sqrtPriceX96` argument (PoolInitializer.sol#L236-L240). Since the computed amounts don't account for the swap fee, the fee is computed separately and added to the input token amount (PoolInitializer.sol#L244-L250).

However, the swap fee computation uses a wrong formula: it subtracts the fee from the computed input amount, which doesn't include the fee:

1. PoolInitializer.sol#L244-L250:

```
amountSpecified += int256(
    SwapMath.swapFees(
        uint256(amountSpecified),
        PoolConstants.fee,
        false // @audit lessFee == false
    )
);
```

2. SwapMath.sol#L61:

```
function swapFees(
    uint256 amount,
    uint24 fee,
    bool lessFee
) internal pure returns (uint256) {
    return (!lessFee ? (amount * fee) / 1e6 : (amount * fee) / (1e6 - fee));
}
```

As a result, the final `amountSpecified` value will be lower than expected, the swap will consume fewer input tokens and won't reach the desired initial price.

Recommendation: Consider setting the `lessFee` parameter to `true` when computing the swap fee during pool initialization in `PoolInitializer.initializePoolSqrtPriceX96()`.

Marginal: Agreed. Fixed in commit 127e93f4.

Cantina Managed: Fixed as recommended.

3.3.2 Liquidation rewards can be used to repay debt, causing a revert

Severity: Medium Risk

Context: NonfungiblePositionManager.sol#L391-L392, PeripheryPayments.sol#L79-L82

Description: The NonfungiblePositionManager.ignite() function is used to settle a position and repay its debt by selling a portion of the margin. The function first calls MarginalV1Pool.settle() to initiate a settlement. During the settlement, the function sells margin to buy enough debt tokens to repay the position's debt (PositionManagement.sol#L292-L295). During the swapping, it invokes the internal PeripheryPayments.pay() to transfer the margin tokens to the pool. The implementation of the function allows paying with ETH when the contract's balance of ETH is enough for the swap:

```
if (token == WETH9 && address(this).balance >= value) {
    // pay with WETH9
    IWETH9(WETH9).deposit{value: value}(); // wrap only what is needed to pay
    IWETH9(WETH9).transfer(recipient, value);
}
```

Due to this, there's a possibility that the liquidation reward withdrawn from the Marginal pool during settlement is used to repay the debt. In this scenario, it'll be wrapped in WETH and sold in the swap to repay the debt; the contract's balance of ETH will be reduced by the sold amount.

However, the `NonfungiblePositionManager.ignite()` returns the liquidation reward to the caller (NonfungiblePositionManager.sol#L391-L392) and reverts if there's not enough ETH balance (PeripheryPayments.sol#L45-L46). As a result, in a scenario when the liquidation reward of a position is used to repay the debt, the transaction will revert due to the reduced liquidation reward.

Recommendation: In the `PositionManagement.flash()` function, consider wrapping the entire contract's balance of ETH in WETH after the settlement call. Consider also removing the `sweepETH` invocation in the `NonfungiblePositionManager.ignite()` function. Besides freeing from sweeping leftover ETH, wrapping to WETH also allows to use liquidation rewards to settle unsafe positions before they were liquidated.

Marginal: Agreed and fixed in commit 45c7aa1f.

Took your original recommendation of wrapping all ETH in the contract in the `marginalV1SettleCallback()` when `address(this) == decoded.payer` (i.e. in flash call). In case the margin token is *not* WETH, I then make a follow up call to `unwrapWETH9()` for the WETH balance remaining in the contract at the end of the `flash()` function, to sweep liquidation rewards to the recipient in ETH.

Cantina Managed: Fixed as recommended.

3.3.3 Incompatibility of liquidations with the pool contract

Severity: Medium Risk

Context: NonfungiblePositionManager.sol#L398

Description: The NonfungiblePositionManager.grab() function is used to liquidate positions in underlying pool contracts. The function, however, has multiple issues:

1. It fails to detect if a position has already been liquidated via the pool's `liquidate()` function. An arbitrary position can be liquidated directly via the pool it was created in.
2. It doesn't delete the liquidated position and doesn't burn the respective token. There's also no separate function to burn tokens of liquidated positions.

3. Its gas consumption is higher than that of the pool's `liquidate()` function, which reduces the reward for the liquidator.

Due to the inability to improve the above issues, it doesn't seem reasonable to implement liquidations in the periphery contracts.

Recommendation:

1. Consider removing the `NonfungiblePositionManager.grab()` function since it cannot be reliably integrated with pool-level liquidations.
2. Consider recommending liquidators use the pool-level liquidation function instead of the periphery one. The `NonfungiblePositionManager` contract can still be used by liquidators to check the safety status of positions via the positions() function.
3. Additionally, consider adding a function to `NonfungiblePositionManager` that allows to burning of a liquidated position's token. The function should be only allowed to be called by the token's owner or an approved address. The function intends to allow token holders to burn liquidated and unused tokens.

Marginal: Agreed. Removed `NPM.grab()` and associated quoter function in commit a0b594f6.

Liquidators can `multicall` liquidate positions through pools by e.g. running our liquidator bot.

Cantina Managed: Fixed as recommended.

3.3.4 Not sending enough Ether to some `NonfungiblePositionManager` endpoints can lead to some loss of user funds

Severity: Medium Risk

Context: NonfungiblePositionManager.sol#L216-L223

Description: The functions `lock` and `free` of the `NonfungiblePositionManager` contract can receive Ether to be used as the new margin value of a position. When the pool calls marginalV1AdjustCallback, the contract will use the Ether to pay for the new margin by wrapping it into WETH9 and then transferring it.

If a user is trying to lock by paying Ether but actually pays less than what the pool callback will ask for, there might be cases where the user loses more money than wanted. The fact that `params.marginIn` is not what the user should pay (it should be `params.marginIn + position.margin`) is a factor that increases the likelihood of someone sending less ether than what is actually needed.

Let's say a user has 1 ether of WETH9 as `margin.position`:

1. The user sends `marginIn` as 1 Ether in the `lock` call, but sends only 1 Ether as `msg.value`.
2. `pool.adjust` will send 1 Ether of WETH9 to the recipient, and will ask for 2 Ether of WETH9 by executing the callback.
3. In the `marginV1AdjustCallback` function, the amount owed is 2 Ether, the token is WETH9 and the payer is the user. The `pay` function will check if `address(this).balance >= value`, but because it is `false`, it will proceed to doing `WETH.transferFrom(user, pool, amountOwed)` through `TransferHelper.safeTransferFrom`.
4. If the user does own WETH9 and has given WETH9 allowance to the manager, 2 Ether of WETH9 will be transferred from the user to the pool to complete the `adjust` call.
5. There's no Ether sweeping in the end of `lock`, so the ether will be left in the contract ready to be taken, and the user lost money.

The same potential loss can happen in `free`. The `burn` and `ignite` functions also have in theory the same vulnerability, but the confusion leading to less Ether being sent to the functions seems less probable to happen. Likewise, function PoolInitializer.initializePoolSqrtPriceX96 also has the same potential issue.

Recommendation: Make sure the ether being sent to the call is enough for the needed payment in advance. Alternatively, the contract could wrap whatever ether is available and try to use `WETH9.transferFrom` to only cover for the missing difference.

Marginal: Agreed. Fixed in commit fa08cbee.

- `params.marginIn` on `NPM.lock(params)` now specifies the *exact* amount in user must pay to manager when adding margin.
- `params.marginOut` on `NPM.free(params)` now specifies the *exact* amount out user receives from manager when removing margin.
- Removed `payable` from `NPM.free` and `NPM.ignite` to avoid any issues with native ETH being sent to contract for these functions.

Will fix `burn` in a separate commit that should also solve issue "`refundETH` is more gas-efficient than `sweepETH`".

Cantina Managed: Fixed in fa08cbee. The root cause is fixed by making `NPM.lock()` and `NPM.free()` non-payable. Additionally, the logic of payments was changed so that the margin amount specified by the user is the exact amount of tokens that is transferred from the user. The `NPM` contract acts as an intermediary to transfer the existing margin back into the pool.

3.4 Low Risk

3.4.1 Position management functions don't sort token addresses

Severity: Low Risk

Context: NonfungiblePositionManager.sol#L147-L152, NonfungiblePositionManager.sol#L229-L234, NonfungiblePositionManager.sol#L269-L274, NonfungiblePositionManager.sol#L310-L313, NonfungiblePositionManager.sol#L364-L369, NonfungiblePositionManager.sol#L410-L415

Description: NonfungiblePositionManager.mint(), NonfungiblePositionManager.lock(), NonfungiblePositionManager.free(), NonfungiblePositionManager.burn(), NonfungiblePositionManager.ignite() and NonfungiblePositionManager.grab() construct a `PoolAddress.PoolKey` instance using the order of token addresses as specified by the caller. However, as per the implementation of PoolAddress.getPoolKey(), token addresses should be sorted.

As a result, if tokens are provided in a different order, the transaction will revert. Since the order of token addresses is not enforced and sorting is not done in the periphery contract, the revert will happen deeper in the call stack (in one of the `MarginalV1Pool` functions) with a different error message that will be confusing to the caller (e.g. Amount1LessThanMin).

Also, when tokens are provided in a wrong order, the params.zeroForOne parameter might not match the order of tokens, which will also cause a revert in the pool contract.

Recommendation: Consider using PoolAddress.getPoolKey() to instantiate all `PoolKey`'s in the mentioned functions to allow successful transaction execution independent of the order of token addresses. In the `NonfungiblePositionManager.mint()` function, consider removing the `params.zeroForOne` parameter and inferring its value from the order of the tokens (as specified by the caller), similarly to how it's done during swapping.

Marginal: Acknowledged. Going to keep it as is given revert.

Cantina Managed: Acknowledged.

3.5 Gas Optimization

3.5.1 Duplicating pool address validation

Severity: Gas Optimization

Context: PositionManagement.sol#L85-L91, PositionManagement.sol#L160-L166, PositionManagement.sol#L217-L223, PositionManagement.sol#L251-L257, PositionManagement.sol#L367-L373

Description: PositionManagement.open(), PositionManagement.adjust(), PositionManagement.settle(), PositionManagement.flash() and PositionManagement.liquidate() are internal functions that are called from NonfungiblePositionManager to open and manage positions. Besides other parameters, all of the functions take the following input parameters: `token0`, `token1`, `maintenance`, and `oracle`. In all of the cases, these parameters are used to instantiate a `PoolAddress.PoolKey` and get the pool address by the key (PositionManagement.sol#L42-L46).

However, the pool address is already obtained and validated in the external functions that call the internal ones. E.g. NonfungiblePositionManager.mint() obtains the pool address and validates it before calling the `open()` function.

As a result, since getting a pool address requires making a call to the core factory contract (PoolAddress.sol#L49-L54), the duplicated code will incur additional gas costs on each position management operation.

Recommendation: In the internal functions, consider taking the pool address as an input parameter, instead of taking `token0`, `token1`, `maintenance`, and `oracle`. Ensure the address provided to the functions is properly obtained from the factory contract.

Also, notice that the `PositionManagement.flash()` function needs the `token0` and `token1` parameters for other purposes, so the two parameters should still be passed to the function.

Marginal: Acknowledged. Going to keep it as is.

Cantina Managed: Acknowledged.

3.5.2 `refundETH` is more gas-efficient than `sweepETH`

Severity: Gas Optimization

Context: NonfungiblePositionManager.sol#L200-L201

Description: In the NonfungiblePositionManager.mint() function, `sweepETH()` is invoked to refund the entire contract's ETH balance to the caller:

```
// sweep any excess ETH from escrowed rewards to sender at end of function to avoid re-entrancy with fallback
sweepETH(0, msg.sender);
```

In this exact scenario, i.e. when refunding the entire balance, the refundETH() function is more gas-efficient because it doesn't check for the minimal amount.

Recommendation: In `NonfungiblePositionManager.mint()`, consider using `refundETH()` instead of `sweepETH()` to refund leftover ETH balance.

Marginal: Agreed. Fixed in commit 72b01c49.

Also fixed issue "Liquidation rewards can be used to repay debt, causing a revert" by calling `refundETH()` at the end of `settle()` in position mgmt contract, in case funding causes miscalculation of debt for value sent in. User now can always send more than estimated, and guarantee to get excess back like with `NPM.mint()`.

Cantina Managed: Fixed as recommended.

3.5.3 Unnecessary usage of `!` in a ternary operator condition in `PositionAmounts`

Severity: Gas Optimization

Context: PositionAmounts.sol#L25

Description: In `PositionAmounts.getLiquidityForSize`, the `reserve` value is set using a conditional ternary operator, with the condition being `!zeroForOne`. But if we switch the outcomes of the conditional, we can save gas by removing the `!` operator.

Recommendation: Replace the condition with the following:

```
uint256 reserve = zeroForOne ? reserve1 : reserve0;
```

Marginal: Acknowledged. Going to keep it as is.

Cantina Managed: Acknowledged.

3.6 Informational

3.6.1 Unused immutable value

Severity: Informational

Context: PeripheryImmutableState.sol#L18

Description: The PeripheryImmutableState abstract contract, that's inherited from by other contracts in the scope, defines the `deployer` immutable that's set to the `MarginalV1PoolDeployer` address. However, this immutable is never used. The deployer contract is not used by the periphery contracts as well.

Recommendation: Consider removing the `deployer` immutable from `PeripheryImmutableState` and the constructors of the contracts that inherit from it.

Marginal: Agreed. Fixed in commit 734b4cef.

Cantina Managed: Fixed as recommended.

3.6.2 Functions return input values without changing them

Severity: Informational

Context: PositionManagement.sol#L121

Description: The NonfungiblePositionManager.mint() function takes a margin value as an input parameter and passes it to the internal invocation of the PositionManagement.open() function. The `PositionManagement.open()` returns the margin without changing it (PositionManagement.sol#L121), and `NonfungiblePositionManager.mint()` returns it to the caller (NonfungiblePositionManager.sol#L165).

As per the implementation of `MarginalV1Pool`, the value of the margin cannot be changed when opening a position. Returning the value in `NonfungiblePositionManager.mint()` can make callers mistakenly believe that it can be changed and force them to validate it, making integrations with the contract a little more confusing.

Recommendation: Consider not returning `margin` from `NonfungiblePositionManager.mint()` and `PositionManagement.open()`.

Marginal: Acknowledged. Going to keep as is given current FE work.

Cantina Managed: Acknowledged.

3.6.3 ERC721's `tokenURI` method returns an empty string for all tokens in `NonfungiblePositionManager`

Severity: Informational

Context: NonfungiblePositionManager.sol#L93

Description: The `tokenURI` method is defined through the OpenZeppelin's ERC721 implementation, which is inherited by the `NonfungiblePositionManager` contract. Due to `_baseURI` not being overwritten, all token URIs will be empty strings.

Recommendation: Consider producing a URI describing each position manager token.

Marginal: Agreed. Will implement before launch.

Cantina Managed: Acknowledged.

3.6.4 `PeripheryImmutableState` contract inherited twice by `PoolInitializer`

Severity: Informational

Context: PoolInitializer.sol#L29-L32

Description: The `PeripheryImmutableState` abstract contract is inherited by the `PoolInitializer` contract. But `PoolInitializer` inherits the `LiquidityManagement` abstract contract, which already inherits `PeripheryImmutableState`.

Recommendation: Remove the explicit inheritance of `PeripheryImmutableState` in `PoolInitializer`.

Marginal: Acknowledged. Going to keep it as is.

Cantina Managed: Acknowledged.

3.6.5 Constant `MINIMUM_LIQUIDITY` is not a valid `liquidityBurned` value in `PoolInitializer`

Severity: Informational

Context: PoolInitializer.sol#L130-131

Description: In `PoolInitializer.createAndInitializePoolIfNecessary()`, uninitialized pools must burn a certain amount of liquidity. The `MINIMUM_LIQUIDITY` constant implies that it is still an acceptable value, which is also reinforced by the fact that the error raised by an invalid `liquidityBurned` input is called `LiquidityBurnedLessThanMin`. But the transaction reverts with that error if the `liquidityBurned` input is equal to the minimum liquidity constant.

Recommendation: Consider replacing the <= operator with <.

Marginal: Potentially confusing naming, but constrained by the <= choice in `v1-core` as well, so will keep as is.

Cantina Managed: Acknowledged.