Marginal v1 lbp

Security Review

Cantina Managed review by:
Defsec, Security Researcher
Jeiwan, Security Researcher

September 11, 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 any asset with an Uniswap V3 Oracle.

From Aug 27th to Sep 3rd the Cantina team conducted a review of marginal-v1-lbp on commit hash 6a000d85. The team identified a total of **28** issues in the following risk categories:

- Critical Risk: 0
- High Risk: 3
- Medium Risk: 4
- Low Risk: 8
- Gas Optimizations: 2
- Informational: 11

3 Findings

3.1 High Risk

3.1.1 `MarginalV1LBLiquidityReceiver.mintMarginalV1()` refunds to treasury, instead of liquidity owner

Severity: High Risk

Context: MarginalV1LBLiquidityReceiver.sol#L636

Description: The MarginalV1LBLiquidityReceiver.mintMarginalV1 function deposits funds as liquidity to a Marginal pool. At the end, the function refunds all unspent funds to the treasury address:

```
// refund any left over unused amounts from uniswap v3 and marginal v1 mints
uint256 balance0 = balance(token0);
uint256 balance1 = balance(token1);
if (balance0 > 0)
    pay(token0, address(this), params.treasuryAddress, balance0);
if (balance1 > 0)
    pay(token1, address(this), params.treasuryAddress, balance1);
```

However, it should refund them to the address that deployed the LB pool, provided the initial liquidity, and seeded the receiver.

The amounts of tokens left in the receiver contract after minting liquidity in Marginal can be significant. Depositing to Uniswap (which is required to deposit to Marginal) and Marginal requires balanced liquidity, which depends on the current prices in the pools. Due to multiple factors (i.e. delayed depositing to Uniswap/Marginal, delayed LB pool finalization, partial initial liquidity swapping, etc...), the liquidity in the receiver contract can be balanced differently. As a result, depositing to Uniswap and Marginal will leave unused liquidity in the receiver.

Recommendation: In the `MarginalV1LBLiquidityReceiver.mintMarginalV1()` function, consider refunding unspent tokens to the address that deposited the initial liquidity in the LB pool and the receiver.

Marginal: Fixed in commit 6e992b68. Added `address refundAddress` to receiver params in liquidity receiver. This is the address that then receives unspent receiver funds.

Cantina Managed: Fixed.

3.1.2 Unbalanced or single-sided liquidity leaves tokens locked in `MarginalV1LBLiquidityReceiver`

Severity: High Risk

Context: MarginalV1LBLiquidityReceiver.sol#L448

Description: After `MarginalV1LBLiquidityReceiver.mintUniswapV3()` was called, some amount of tokens can be left locked in the contract without the ability to be withdrawn or deposited to Marginal.

The MarginalV1LBLiquidityReceiver.mintUniswapV3() function is used to deposit funds to Uniswap V3 after a pool was finalized. The funds are deposited as pool liquidity at the current price of the pool, which requires that the deposited amounts are balanced, relative to the price.

Since there can be a delay between the LB pool finalization and depositing to Uniswap V3, the current price in the Uniswap pool can be different from the final price of the LB pool. As a result, the amounts deposited to Uniswap will be balanced differently, resulting in a leftover of one of the amounts (Uniswap takes the smaller liquidity when minting, thus not asking for more tokens). It won't be possible to deposit this leftover to Marginal via the MarginalV1LBLiquidityReceiver.mintMarginalV1(), because it also requires a balanced liquidity.

However, the receiver contract doesn't allow to refund of tokens left after providing liquidity to Uniswap. Thus the tokens will be left in the contract.

In an alternative and rare scenario, when an LB pool is finalized with the full initial liquidity (i.e. no swaps were made in the contract), the entire liquidity in the receiver contract is single-sided. Since Uniswap doesn't allow adding single-sided liquidity at the current price, the call to `MarginalV1LBLiquidityReceiver.mintUniswapV3()` will fail, leaving the entire liquidity locked in the contract. `MarginalV1LBLiquidityReceiver.mintMarginalV1()` will also fail because it requires that liquidity was previously added to Uniswap.

Impact: There's a risk of locking all or a portion of liquidity in the receiver contract.

Likelihood: The vulnerability impacts the normal flow of events in the protocol, but not all scenarios. E.g. when there's no price difference between the LB and the Uniswap pools, there will be no leftover token amounts.

Recommendation: Consider implementing a refunding functionality to allow users to withdraw tokens after providing liquidity to Uniswap V3. It might be also useful to allow withdrawals even before liquidity is provided to Uniswap and Marginal.

Marginal: Fixed in commit 10e31504. Added `freeReserves()` function that can be called after `params.lockDuration` seconds have passed since `receiver.notifyRewardAmounts` was called. The function zeroes reserve state and transfers (token0, token1) balances to the `params.refundAddress`.

Cantina Managed: Fixed.

3.1.3 Specified amount is not updated after clamping, causing a lock of funds

Severity: High Risk

Context: MarginalV1LBPool.sol#L450

Description: The value of `amountSpecified` in the `MarginalV1LBPool.swap()` function is not updated after the price is clamped. The clamping narrows the price movement during the swap, but the user still pays the full specified amount. The extra tokens cannot be withdrawn by the supplier and are left locked in the contract.

The swapping logic in the MarginalV1LBPool.swap() function allows swapping only until the either the lower or the upper price is reached. The target price of a swap is computed using the SqrtPriceMath.sqrtPriceX96NextSwap() function so that it's not lower than the lower price and not higher than the upper price.

However, after a price was clamped, the specified input amount was not recalculated (MarginalV1LBPool.sol#L306-L307, MarginalV1LBPool.sol#L328-L329):

```
if (!zeroForOne) {
    amount0 = !exactInput ? amountSpecified : amount0; // in case of rounding issues
    amount1 = exactInput ? amountSpecified : amount1;
    // ...
} else {
    amount1 = !exactInput ? amountSpecified : amount1; // in case of rounding issues
    amount0 = exactInput ? amountSpecified : amount0;
    // ...
```

Clamping reduces the price movement required to achieve the target price, thus also reducing the required input amount. However, the user still pays the full `amountSpecified`.

The final price is set to the clamped price (MarginalV1LBPool.sol#L351-L352):

```
// lbp done if reaches final sqrt price
_state.finalized = (_state.sqrtPriceX96 == sqrtPriceFinalizeX96);
```

When finalizing and burning liquidity, the amounts of tokens withdrawn from the pool are computed within the lower and upper price range (MarginalV1LBPool.sol#L449-L455):

```
// amounts out adjusted for concentrated range position price limits
(amount0, amount1) = RangeMath.toAmounts(
 liquidityDelta,
 _state.sqrtPriceX96,
 sqrtPriceLowerX96,
 sqrtPriceUpperX96
);
```

But since the trader pays more tokens than required to move the price between the boundaries of the range, one of the computed amounts will be smaller than the amount of tokens paid by the trader. It won't be possible to withdraw the difference and it will remain in the contract.

Impact: A portion of tokens paid by users is locked in the contract and cannot be withdrawn during pool finalization.

Likelihood: The vulnerability impacts all swaps that reach the final price, but since finalizing pools doesn't require reaching it, the likelihood is medium.

Proof of concept: The following proof of concept demonstrates that the specified input amount is paid in full when clamping happens:

- `tests/functional/pool/test_pool_swap.py`:

```
@pytest.mark.parametrize("init_with_sqrt_price_lower_x96", [False])
@pytest.mark.focus
def test_pool_swap__updates_state_with_exact_input_zero_for_one_to_range_tick_clamping(
pool_initialized,
callee,
sqrt_price_math_lib,
swap_math_lib,
liquidity_math_lib,
sender,
alice,
token0,
token1,
chain,
init_with_sqrt_price_lower_x96,
):
pool_initialized_with_liquidity = pool_initialized(init_with_sqrt_price_lower_x96)
state = pool_initialized_with_liquidity.state()

sqrt_price_lower_x96 = pool_initialized_with_liquidity.sqrtPriceLowerX96()
sqrt_price_finalize_x96 = pool_initialized_with_liquidity.sqrtPriceFinalizeX96()

zero_for_one = True
sqrt_price_limit_x96 = MIN_SQRT_RATIO + 1

# calc amounts in/out for the swap with first pass on price thru sqrt price math lib
sqrt_price_x96_next = sqrt_price_lower_x96
(amount0, amount1) = swap_math_lib.swapAmounts(
state.liquidity,
state.sqrtPriceX96,
sqrt_price_x96_next,
)
amount_specified = int(amount0 * 1.001)  # extra buffer

# compute the target price, as it's done in Pool.swap
sqrt_price_x96_next_computed = sqrt_price_math_lib.sqrtPriceX96NextSwap(
state.liquidity,
state.sqrtPriceX96,
zero_for_one,
amount_specified,
)

# update the oracle
block_timestamp_next = chain.pending_timestamp
tick_cumulative = state.tickCumulative + state.tick * (
block_timestamp_next - state.blockTimestamp
)

state.blockTimestamp = block_timestamp_next
state.tickCumulative = tick_cumulative

# update state price
state.sqrtPriceX96 = sqrt_price_x96_next
state.tick = calc_tick_from_sqrt_price_x96(sqrt_price_x96_next)
state.finalized = sqrt_price_x96_next == sqrt_price_finalize_x96

tx = callee.swap(
pool_initialized_with_liquidity.address,
alice.address,
zero_for_one,
amount_specified,
sqrt_price_limit_x96,
sender=sender,
)

assert pool_initialized_with_liquidity.state() == state
assert state.finalized

# the swap price was clamped: the computed price is below the actual price
```

```
assert sqrt_price_x96_next_computed < state.sqrtPriceX96

# the input amount wasn't clamped and remained as specified by the caller
swap = tx.decode_logs(pool_initialized_with_liquidity.Swap)[0]
assert amount_specified == swap.amount0
assert amount1 == swap.amount1
```

The following proof of concept demonstrates that there are leftover tokens after pool finalization:

- `tests/functional/test_supplier_finalize_pool.py`:

```
@pytest.mark.parametrize("fee_protocol", [10])
@pytest.mark.parametrize("init_with_sqrt_price_lower_x96", [False])
@pytest.mark.focus
def test_supplier_finalize_pool__finalizes_pool_leftover(
factory,
supplier,
receiver_and_pool_finalized,
token0,
token1,
sender,
admin,
finalizer,
chain,
fee_protocol,
init_with_sqrt_price_lower_x96,
):
factory.setFeeProtocol(fee_protocol, sender=admin)
(receiver, pool_finalized_with_liquidity) = receiver_and_pool_finalized(
init_with_sqrt_price_lower_x96
)
assert (
pool_finalized_with_liquidity.sqrtPriceInitializeX96() > 0
)  # pool initialized
assert pool_finalized_with_liquidity.totalSupply() > 0

state = pool_finalized_with_liquidity.state()
assert state.finalized is True
assert state.feeProtocol == fee_protocol

(receiver_reserve0, receiver_reserve1) = (receiver.reserve0(), receiver.reserve1())
assert (
receiver_reserve0 > 0
if init_with_sqrt_price_lower_x96
else receiver_reserve1 > 0
)

assert sender.address != finalizer.address
deadline = chain.pending_timestamp + 3600
params = (
pool_finalized_with_liquidity.token0(),
pool_finalized_with_liquidity.token1(),
pool_finalized_with_liquidity.tickLower(),
pool_finalized_with_liquidity.tickUpper(),
pool_finalized_with_liquidity.blockTimestampInitialize(),
deadline,
)
tx = supplier.finalizePool(params, sender=sender)

total_supply = pool_finalized_with_liquidity.totalSupply()
assert total_supply == 0

state = pool_finalized_with_liquidity.state()
assert state.liquidity == 0

assert token0.balanceOf(pool_finalized_with_liquidity.address) == 422484836 # !!! leftover
assert token1.balanceOf(pool_finalized_with_liquidity.address) == 1
```

Recommendation: In the `MarginalV1LBPool.swap()` function, consider using both computed amount swap amounts after the target price was clamped.

Marginal: Fixed in commit 052fec79.

Cantina Managed: Fixed.

3.2 Medium Risk

3.2.1 `MarginalV1LBLiquidityReceiver.mintUniswapV3()` updates reserves incorrectly

Severity: Medium Risk

Context: MarginalV1LBLiquidityReceiver.sol#L411

Description: The MarginalV1LBLiquidityReceiver.mintUniswapV3() function deposits funds into a Uniswap pool as liquidity. The available amounts of tokens are stored in the `reserve0`/`reserve1` state variables, that are reduced by the function (MarginalV1LBLiquidityReceiver.sol#L407-L415):

```
uint256 amount0UniswapV3 = (_reserve0 * params.uniswapV3Ratio) / 1e6;
uint256 amount1UniswapV3 = (_reserve1 * params.uniswapV3Ratio) / 1e6;

_reserve0 -= amount0UniswapV3;
_reserve1 -= amount1UniswapV3;

// update reserves
reserve0 = _reserve0;
reserve1 = _reserve1;
```

However, the actual reserves taken from the contract to deposit to Uniswap can be smaller because Uniswap can take fewer tokens than desired (e.g. due to the price difference between the final LB pool price and the current price in the Uniswap pool). The actual token amounts deposited to Uniswap are returned from the IUniswapV3NonfungiblePositionManager.mint() function call below:

```
(
    tokenId,
    liquidity,
    amount0,
    amount1
) = IUniswapV3NonfungiblePositionManager(
    uniswapV3NonfungiblePositionManager
).mint(
        IUniswapV3NonfungiblePositionManager.MintParams({
            // ...
        })
    );
```

As a result, less tokens will be deposited to Marginal because, in the MarginalV1LBLiquidityReceiver.mintMarginalV1() function, the available token amounts are read from the reserve variables:

```
(uint256 _reserve0, uint256 _reserve1) = (reserve0, reserve1);
if (_reserve0 == 0 && _reserve1 == 0) revert InvalidReserves();
```

Recommendation: In the `MarginalV1LBLiquidityReceiver.mintUniswapV3()` function, when updating the reserves, consider subtracting the `amount0`/`amount1` values returned by the `IUniswapV3NonfungiblePositionManager.mint()` function, not `amount0UniswapV3`/`amount1UniswapV3`.

Marginal: Fixed in commit e6dab824. Also added an `uninitialized` flag for more robust check in `MarginalV1LBLiquidityReceiver.sol::initialize` whether receiver has been initialized.

Cantina Managed: Fixed.

3.2.2 Zero token ID and shares in free functions of `MarginalV1LBLiquidityReceiver`

Severity: Medium Risk

Context: MarginalV1LBLiquidityReceiver.sol

Description: In the `MarginalV1LBLiquidityReceiver` contract, the `freeUniswapV3` and `freeMarginalV1` functions have a logical error where the token ID and shares are set to zero before they are used, resulting in incorrect transfers and emissions of zero values.

The issue occurs in both `freeUniswapV3` and `freeMarginalV1` functions:

1. In `freeUniswapV3`:

```
info.tokenId = 0;
info.blockTimestamp = 0;
uniswapV3PoolInfo = info;

// ... later in the function
IUniswapV3NonfungiblePositionManager(
    uniswapV3NonfungiblePositionManager
).transferFrom(address(this), recipient, info.tokenId); // info.tokenId is always 0 here

emit FreeUniswapV3(info.poolAddress, info.tokenId, recipient); // info.tokenId is always 0 here
```

2. Similarly in `freeMarginalV1`:

```
info.shares = 0;
info.blockTimestamp = 0;
marginalV1PoolInfo = info;

pay(info.poolAddress, address(this), recipient, info.shares); // info.shares is always 0 here

emit FreeMarginalV1(info.poolAddress, info.shares, recipient); // info.shares is always 0 here
```

---

1. No tokens being transferred in `freeUniswapV3`, effectively locking the NFT in the contract.
2. No shares being transferred in `freeMarginalV1`, resulting in a loss of user funds.
3. Incorrect event emissions with zero values for token ID and shares.

Recommendation: Cache the token ID and shares before setting them to zero.

Marginal: Fixed in commit 2a39af91.

Cantina Managed: Fixed.

3.2.3 ETH Dust in `createAndInitializePool`

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `createAndInitializePool` function is marked as `payable`, but it doesn't handle any potential dust ETH that might be sent along with the transaction. This could lead to ETH being trapped in the contract or potentially stolen by an attacker.

```
function createAndInitializePool(
    CreateAndInitializeParams calldata params
)
    external
    payable
    checkDeadline(params.deadline)
    returns (
        address pool,
        address receiver,
        uint256 shares,
        uint256 amount0,
        uint256 amount1
    )
{
    // ... function implementation ...
}
```

1. ETH sent to this function could become trapped in the contract.
2. An attacker could potentially extract any dust ETH left in the contract.
3. Users might lose small amounts of ETH unintentionally.

Recommendation: Implement a mechanism to refund any unused ETH at the end of the function. This can be done by calling the `refundETH()` function that already exists in the `PeripheryPayments` contract.

Marginal: Fixed in commit 3191c5d7.

Cantina Managed: Fix is verified.

3.2.4 Lack of slippage protection in liquidity provision

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The current implementation of `mintUniswapV3` and `mintMarginalV1` functions sets the minimum amount parameters (`amount0Min` and `amount1Min`) to zero. This lack of slippage protection could potentially expose the contract to sandwich attacks during liquidity provision.

In both `mintUniswapV3` and `mintMarginalV1` functions:

```
amount0Min: 0,
amount1Min: 0, // TODO: issue for slippage?
```

The worst-case scenario would result in less liquidity in Uniswap v3 and Marginal v1 pools, with excess funds being swept to the `treasuryAddress`.

Recommendation: Add a parameter to `receiverParams` to limit slippage. This would allow for dynamic slippage protection based on the specific needs of each transaction.

Marginal: Going to pass and keep as zero with the worst case simply being more funds swept to params refund address. The issue with dynamic slippage protection would be that we could either choose:

1. The LBP price as the price to reference for amount slippage tolerance.
2. The Uniswap v3 pool spot price from `slot0()`.

The former (1.) would not be manipulable, but if enough time has passed and price has moved significantly since the LBP if the pool already exists, it could cause the mint functions to revert and no liquidity to be deployed.

The latter (2.) is manipulable and doesn't really provide any more protection than setting mins to 0.

So in either case it seems safer to simply keep mins as 0 and simply sweep any excess unspent funds to the refund address.

Cantina Managed: Acknowledged.

3.3 Low Risk

3.3.1 `onlyPoolSupplier` modifier can be bypassed

Severity: Low Risk

Context: MarginalV1LBLiquidityReceiverDeployer.sol#L25

Description: The MarginalV1LBLiquidityReceiverDeployer.deploy() function allows to deploy a `MarginalV1LBLiquidityReceiver` contract. The access to the function is restricted by the onlyPoolSupplier modifier that allows calling the function only to the supplier contract:

```
modifier onlyPoolSupplier(address pool) {
    if (msg.sender != IMarginalV1LBPool(pool).supplier())
        revert Unauthorized();
    _;
}
```

The pool address is passed to the `MarginalV1LBLiquidityReceiverDeployer.deploy()` function by the caller, thus the restriction can be bypassed if the contract at the `pool` address implements the `supplier()` method and returns the address of the `MarginalV1LBLiquidityReceiverDeployer.deploy()` caller.

Recommendation: Consider storing the address of the supplier contract in an immutable instead of trusting a pool to return it.

Marginal: Fixed in commit 79c48742.

Cantina Managed: Fix is verified.

3.3.2 Insufficient constructor parameters validation in `MarginalV1LBLiquidityReceiver`

Severity: Low Risk

Context: MarginalV1LBLiquidityReceiver.sol#L170

Description: The MarginalV1LBLiquidityReceiver.checkParams validates the constructor parameters of the contract. However, `params.treasuryAddress` and `params.lockOwner` are not validated:

1. `params.treasuryAddress` receives fees from the rewards, thus it should never be the zero address.
2. `params.lockOwner` withdraws liquidity from Uniswap and Marginal, it should never be the zero address (except for when liquidity is locked indefinitely).

Recommendation: Consider adding the missed checks to `MarginalV1LBLiquidityReceiver.checkParams()`.

Marginal: Fixed in commit 4eade4df. Still not checking `lockOwner` is zero address to allow users to burn the LPs if they wish.

Cantina Managed: Fix is verified.

3.3.3 `MarginalV1LBPool.mint()` unnecessarily rounds up both amounts

Severity: Low Risk

Context: MarginalV1LBPool.sol#L390

Description: The MarginalV1LBPool.mint() function mints the initial liquidity in a pool. After computing the necessary token amounts, it rounds them both up (MarginalV1LBPool.sol#L382-L390):

```
// amounts in adjusted for concentrated range position price limits
(amount0, amount1) = RangeMath.toAmounts(
 liquidityDelta,
 _state.sqrtPriceX96,
 sqrtPriceLowerX96,
 sqrtPriceUpperX96
);
amount0 += 1; // rough round up on amounts in when add liquidity
amount1 += 1;
```

However, as per the logic of the pool contract, the initial liquidity can be added only to either of the tokens. Thus one of the amounts is always 0 and is not expected to be transferred from the caller. Rounding it forces the user to always pay 1 wei.

Recommendation: Consider improving the rounding logic so that only the required token amount is rounded.

Marginal: Fixed in commit b0b8e35a.

Cantina Managed: Fix is verified.

3.3.4 `MarginalV1LBPool.Swap` event should contain finalization status

Severity: Low Risk

Context: MarginalV1LBPool.sol#L357

Description: The MarginalV1LBPool.swap() function is used to swap tokens in the pool. When the final price is reached, the function sets the `finalized` flag in the state (MarginalV1LBPool.sol#L351-L352). However, the value of `finalized` is not emitted in the `Swap` event.

As a result, it won't be simple for analytic tools to track pools that have reached the final price and that should finalized.

Recommendation: Consider adding a boolean `finalized` field to the `MarginalV1LBPool.Swap` event and setting it to the value of `_state.finalized` in the `MarginalV1LBPool.swap()` function.

Marginal: Fixed in commit cd8a4b2b.

Cantina Managed: Fix is verified.

3.3.5 Missing tokens order check in `MarginalV1LBPoolDeployer.deploy()`

Severity: Low Risk

Context: MarginalV1LBPoolDeployer.sol#L16

Description: The MarginalV1LBPoolDeployer.deploy() function is used to deploy instances of the `MarginalV1LBPool` contract. The function takes necessary parameters and passes them to the constructor of `MarginalV1LBPool`.

However, the order of the token addresses (`token0` and `token1`) is not validated either in `MarginalV1LBPoolDeployer.deploy()`, or in the `MarginalV1LBPool`'s constructor.

As a result, a pool with a wrong order of `token0` and `token1` addresses can be deployed. If not noticed by the deployed, this can result in a swapping at the wrong price and a loss of funds.

Recommendation: In either the `MarginalV1LBPoolDeployer.deploy()` function or the constructor of `MarginalV1LBPool`, consider adding a check to ensure that `token0` and `token1` parameters are sorted correctly. Alternatively, consider restricting access to `MarginalV1LBPoolDeployer.deploy()` only to the factory contract.

Marginal: Ordering happens in `MarginalV1LBFactory.sol::createPool` and would consider interacting directly with the pool deployer contract simply unsafe. All "*canonical*" pools are deployed through interacting with the canonical factory contract:

```
/// @inheritdoc IMarginalV1LBFactory
function createPool(
    address tokenA,
    address tokenB,
    int24 tickLower,
    int24 tickUpper,
    address supplier,
    uint256 blockTimestampInitialize
) external returns (address pool) {
    (address token0, address token1) = tokenA < tokenB
        ? (tokenA, tokenB)
        : (tokenB, tokenA);

    if (
        getPool[token0][token1][tickLower][tickUpper][supplier][
            blockTimestampInitialize
        ] != address(0)
    ) revert PoolActive();

    pool = IMarginalV1LBPoolDeployer(marginalV1LBDeployer).deploy(
        token0,
        token1,
        tickLower,
        tickUpper,
        supplier,
        blockTimestampInitialize
    );
```

Cantina Managed: Acknowledged. Since `MarginalV1LBPoolDeployer.deploy()` is public, it can be called directly, allowing incorrectly ordered tokens.

3.3.6 `supplier` field in `PoolCreated` event is not indexed

Severity: Low Risk

Context: MarginalV1LBFactory.sol#L36

Description: The MarginalV1LBFactory.PoolCreated event is emitted when a new pool is created. The event contains the `supplier` field, which is the address that supplied the initial liquidity and that is allowed to finalize the pool. Since users can specify arbitrary suppliers, they might need to find events by the supplier address quickly. However, since the field is not indexed in the event, this won't be possible without fetching all `PoolCreated` events from a node.

Recommendation: Consider indexing the `supplier` field of the `MarginalV1LBFactory.PoolCreated` event.

Marginal: Fixed in commit c45d5c3e.

Cantina Managed: Fix is verified.

3.3.7 Lock duration can be set to 0

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `MarginalV1LBLiquidityReceiver` contract, the `lockDuration` parameter can be set to an already "0" during contract construction, potentially blocking unlocking of liquidity.

The issue occurs in the constructor of the `MarginalV1LBLiquidityReceiver` contract:

```
constructor(
    address _factory,
    address _marginalV1Factory,
    address _WETH9,
    address _pool,
    bytes memory data // receiver parameters encoded
)
    PeripheryImmutableState(_factory, _marginalV1Factory, _WETH9)
    MarginalV1LBReceiver(_pool)
{
    deployer = msg.sender;
    ReceiverParams memory params = abi.decode(data, (ReceiverParams));
    checkParams(params);
    receiverParams = params;
}
```

The problem arises because:

1. The `lockDuration` is part of the `ReceiverParams` struct, which is set during contract creation.
2. A mistaken deployment could set `lockDuration` to 0 or a very small value.

Recommendation: Implement a check in the constructor to ensure the `lockDuration` is different than zero.

Marginal: If `lockDuration == 0` then the `lockOwner` can simply free liquidity immediately via the `freeUniswapV3` and `freeMarginalV1` functions, which is ok.

Cantina Managed: Acknowledged, intended behavior due to comment.

3.3.8 Inconsistent factory address usage in `MarginalV1LBLiquidityReceiverDeployer`

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: In the `MarginalV1LBLiquidityReceiverDeployer` contract, there's an inconsistency in how the factory address is obtained and used. The contract stores a `marginalV1Factory` address in its state, set during construction. However, in the `deploy` function, it retrieves the factory address from the pool instead of using the stored address:

```
address factory = IMarginalV1LBPool(pool).factory();
```

Recommendation: Use the stored `marginalV1Factory` address instead of retrieving it from the pool.

Marginal: `marginalV1Factory` is for the `v1-core` contract factory whereas `factory` here is the factory for LBP contracts in this repo. Different factories.

Cantina Managed: Acknowledged. Was thinking If we can set it in the constructor.

3.4 Gas Optimization

3.4.1 Pool initialization can be simplified

Severity: Gas Optimization

Context: MarginalV1LBPool.sol#L155

Description: The MarginalV1LBPool.initialize() function takes a price `sqrtPriceX96` and sets it as the initial price of the pool. Since the price can only be either `sqrtPriceLowerX96` or `sqrtPriceUpperX96` (which are set in the constructor) there's no need to take an actual price: a boolean flag can be taken instead. E.g., if the flag is called `useLowerPrice` and is set to `true`, `sqrtPriceLowerX96` is used as the initial price, and vice versa. This way, users don't need to compute and pass a price, the price validation checks are not necessary, and the price and tick initialization can be simplified to read values from the respective immutables.

Recommendation: Consider simplifying the `MarginalV1LBPool.initialize()` function by taking a boolean flag instead of an actual price, `sqrtPriceX96`.

Marginal: Agreed but the `MarginalV1LBSupplier.sol::createAndInitializePool` function abstracts this away so going to keep as is. L118:

```
// initialize pool since not initialized yet
uint160 sqrtPriceX96 = TickMath.getSqrtRatioAtTick(params.tick);
uint160 sqrtPriceLowerX96 = IMarginalV1LBPool(pool).sqrtPriceLowerX96();
uint160 sqrtPriceUpperX96 = IMarginalV1LBPool(pool).sqrtPriceUpperX96();

// calculate liquidity using range math
uint128 liquidity = LiquidityAmounts.getLiquidityForAmounts(
    sqrtPriceX96,
    sqrtPriceLowerX96,
    sqrtPriceUpperX96,
    sqrtPriceX96 == sqrtPriceLowerX96 ? params.amountDesired : 0, // reserve0
    sqrtPriceX96 == sqrtPriceLowerX96 ? 0 : params.amountDesired // reserve1
);

(shares, amount0, amount1) = IMarginalV1LBPool(pool).initialize(
    liquidity,
    sqrtPriceX96,
    abi.encode(MintCallbackData({poolKey: poolKey, payer: msg.sender}))
);
```

Cantina Managed: Acknowledged. `MarginalV1LBPool.initialize()` is public and can be called without using the supplier contract.

3.4.2 Unnecessary copying in memory

Severity: Gas Optimization

Context: V1LBRouter.sol#L113

Description: In functions V1LBRouter.marginalV1SwapCallback(), V1LBRouter.exactInputInternal(), and V1LBRouter.exactOutputInternal, the calldata arguments `data.tokenIn` and `data.tokenOut` are unnecessarily copied to variables `tokenIn` and `tokenOut`, which consumes gas on the memory allocations.

Recommendation: In the linked functions, consider using `data.tokenIn` and `data.tokenOut` directly, without copying them in memory.

Marginal: Agreed but going to keep as is to keep as close to original router forked code as possible.

Cantina Managed: Acknowledged.

3.5 Informational

3.5.1 Unnecessary deadline requirements

Severity: Informational

Context: MarginalV1LBSupplier.sol#L90

Description: Some functions unnecessarily require transactions to be minted within a deadline:

1. MarginalV1LBSupplier.createAndInitializePool(): the function deploys new contracts and doesn't rely on an external state, thus there are no time-sensitive dependencies on state values.
2. MarginalV1LBSupplier.finalizePool(): the function finalizes a pool, which is possible at any moment after satisfying the finalization criteria; there's no need to allow finalizing pool only before a deadline.

Recommendation: Consider removing the `checkDeadline()` modifier from the mentioned functions.

Marginal: Fixed in commit 93d35833.

Cantina Managed: Fixed.

3.5.2 Tokenizing liquidity is not necessary

Severity: Informational

Context: MarginalV1LBPool.sol#L418

Description: The MarginalV1LBPool.mint() function is used to add the initial liquidity to a pool. The function tokenizes the added liquidity and mints the respective amount of ERC20 tokens. However, this is not necessary:

1. Liquidity can be added only once.
2. The only owner of liquidity is the supplier.
3. Finalizing burns the entire liquidity, so selling or transferring out a portion of LP tokens will disallow finalizing.

Recommendation: Consider removing the liquidity tokenization feature from the pool contract.

Marginal: Agree but want to keep as close to forked `v1-core/MarginalV1Pool.sol::mint` code as possible, so keeping as is.

Cantina Managed: Acknowledged.

3.5.3 `MarginalV1LBPool.mint()` can be simplified as it's only called once

Severity: Informational

Context: MarginalV1LBPool.sol#L375

Description: The MarginalV1LBPool.mint() function is used to mint liquidity in a pool. The logic of the function handles two cases: when minting the initial liquidity and when minting additional liquidity. However, the pool doesn't allow minting additional liquidity: the `MarginalV1LBPool.mint()` function is private and is only called during initialization.

Recommendation: Consider simplifying the `MarginalV1LBPool.mint()` function to handle only the initial minting and always set the minimum liquidity delta to `MINIMUM_LIQUIDITY`, as well as skip the shares computation.

Marginal: Agreed but want to keep as close as possible to forked `v1-core/MarginalV1Pool.sol::mint` code so will keep as is.

Cantina Managed: Acknowledged.

3.5.4 `IMarginalV1LBFinalizeCallback.marginalV1LBFinalizeCallback()` is not necessary

Severity: Informational

Context: MarginalV1LBPool.sol#L214

Description: The MarginalV1LBPool.finalize() function is used to finalize pools. At the end, the function calls the IMarginalV1LBFinalizeCallback.marginalV1LBFinalizeCallback() callback on the caller address. However, the callback doesn't require its implementor to take any actions: it only serves to notify the caller.

Instead of calling the callback, the function can just return the execution control to the caller and let it perform any actions it needs to do. This will save gas, free the caller from implementing an unnecessary callback, and make the handling of pool finalization simpler.

Recommendation: Consider removing the `IMarginalV1LBFinalizeCallback.marginalV1LBFinalizeCallback()` call in the `MarginalV1LBPool.finalize()` function.

Marginal: Fixed in:

- Commit 6cc60bf9.
- Commit e3a736af (edits to use amounts returned since `pay` already assumes standard ERC20s).

3.5.5 Unused amount caching

Severity: Informational

Context: V1LBRouter.sol#L48

Description: The `amountInCached` variable and the `DEFAULT_AMOUNT_IN_CACHED` constant are not used in the `V1LBRouter` contract. They are intended to be used in multi-pool exact output swaps, but since the contract implements only single-pool swaps, there's no need to have them.

Recommendation: Consider removing the `amountInCached` variable and the `DEFAULT_AMOUNT_IN_CACHED` constant from the `V1LBRouter` contract.

Marginal: Going to keep as is given used in `exactOutputSingle` just to keep consistent with original router forked code.

Cantina Managed: Acknowledged. The variables are still unused.

3.5.6 Fee calculation can lead to protocol revenue loss

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `rangeFees` function in the `RangeMath` library has one issue:

1. Potential loss of fees for small amounts or low fee rates due to integer division. The issues occur in the `rangeFees` function:

```
function rangeFees(
    uint256 amount0,
    uint256 amount1,
    uint8 fee
) internal pure returns (uint256 fees0, uint256 fees1) {
    fees0 = (amount0 * fee) / 1e4;
    fees1 = (amount1 * fee) / 1e4;
}
```

   For small amounts or low fee rates, the integer division by 1e4 can result in rounding down to zero, effectively losing the fee. For example:

   - If `amount0 = 9999` and `fee = 1`, the calculation `(9999 * 1) / 1e4 = 0` results in no fee being charged.

Recommendation: Implement a more robust fee calculation method with safeguards.

Marginal: For most tokens of interest (>= 6 decimals) this shouldn't be a big issue.

Cantina Managed: Acknowledged. Agreed that for most tokens this shouldn't be a big issue.

3.5.7 Loss of finalization capability in MarginalV1LBSupplier

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: In the `MarginalV1LBSupplier` contract, there is an issue where the ability to finalize a pool could be permanently lost if the finalizer address is set to the zero address (0x0). If the finalizer is set to the zero address, either accidentally or maliciously, the pool cannot be finalized by any address.

Recommendation: Prevent setting the finalizer to the zero address in `createAndInitializePool`.

Marginal: Fixed in commit 7ff0e7ee.

Cantina Managed: Fixed.

3.5.8 Limited Support for standard ERC20 tokens only

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The current implementation of the supplier and receiver contracts supports only standard ERC20 tokens. For non-standard tokens, the notified reward amount might not accurately reflect the actual balance received due to rebasing or fee-on-transfer mechanisms.

Recommendation: Implement additional balance checks before and after transfers.

Marginal: Agreed, but viewing supplier and receiver contracts as "*periphery*", so ok for now.

Cantina Managed: Acknowledged.

3.5.9 Use `Ownable2Step` for access control

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `MarginalV1LBFactory` contract executes a single-step ownership change, which may be prone to accidental transfers of access control. The `Ownable2Step` variant of the `Ownable` contract can protect such accidental transfers.

Recommendation: It is recommended to use `Ownable2Step` for access control.

Marginal: Going to keep as `Ownable`.

Cantina Managed: Acknowledged.

3.5.10 Unsolved TODOs

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: There are TODO comments in the files, most of which are questions related to this security review. However, it is recommended to remove all TODO comments before completing the review.

> ./receiver/MarginalV1LBLiquidityReceiver.sol:428: // TODO: check finite full tick range math ./receiver/MarginalV1LBLiquidityReceiver.sol:458: amount1Min: 0, // TODO: issue for slippage? ./receiver/MarginalV1LBLiquidityReceiver.sol:552: // TODO: fix logic for liquidity burned? use quoter? ./receiver/MarginalV1LBLiquidityReceiver.sol:559: uint128 liquidityBurned = PoolConstants.MINIMUM_LIQUIDITY ** 2; // TODO: validation around minimum liquidity? ./receiver/MarginalV1LBLiquidityReceiver.sol:591: amount1Min: 0, // TODO: issue for slippage? ./receiver/MarginalV1LBLiquidityReceiver.sol:619: amount1Min: 0, // TODO: issue for slippage?

Recommendation: It's not recommended to include TODO in production contracts. They should be fixed or removed.

Marginal: Fixed in commit 5adfa313.

Cantina Managed: Fix is verified. but todos are not solved.

3.5.11 Unnecessary and potentially risky `Multicall` functionality in `V1LBRouter`

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: The `V1LBRouter` contract inherits from Multicall, which provides a `multicall` function allowing multiple function calls in a single transaction. The presence of Multicall functionality introduces unnecessary complexity and potential attack vectors, such as:

- Unintended reuse of ETH across multiple operations.
- Potential manipulation of contract state between calls in a multicall transaction.
- Increased attack surface without clear benefits for the router's primary functions.

```
contract V1LBRouter is
    IV1LBRouter,
    IMarginalV1SwapCallback,
    PeripheryImmutableState,
    PeripheryPayments,
    PeripheryValidation,
    Multicall,
    SelfPermit
{}
```

Recommendation: Remove the inheritance from `Multicall`.

Marginal: Going to keep `Multicall` in router in case users wish to swap on multiple pools in a single call.

- Unintended reuse of ETH across multiple operation

Use of ETH confined to `pay` and `refundETH` in periphery payments. Both use `address(this).balance` and *not* `msg.value`.

- Potential manipulation of contract state between calls in a multicall transaction.

`V1LBRouter.sol` is effectively stateless.

- Increased attack surface without clear benefits for the router's primary functions.

Agreed except the benefit is multiple router swaps at once.

Cantina Managed: Acknowledged.