Maverick V2 Security Review

Report Version 1.1

March 13, 2024

Conducted by:

George Hunter, Independent Security Researcher

Table of Contents

1 About George Hunter

George Hunter is a proficient and reputable independent smart contract security researcher with over 50 solo and team security engagements contributing to the security of numerous smart contract protocols in the past 2 years. Previously held roles include Lead Smart Contract Auditor at Paladin Blockchain Security and Smart Contract Engineer at Nexo. He has audited smart contracts for clients such as LayerZero, Euler, TraderJoe, Maverick, Ambire, and other leading protocols. For security audit inquiries, you can reach out on Telegram or Twitter at *@georgehntr*.

2 Disclaimer

Audits are a time-, resource-, and expertise-bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can reveal the presence of vulnerabilities **but cannot guarantee their absence**.

3 Risk classification

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

3.1 Impact

- **High** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - involves a small loss of funds or affects a core functionality of the protocol.
- **Low** - encompasses any unexpected behavior that is non-critical.

3.2 Likelihood

- **High** - a direct attack vector; the cost is relatively low compared to the potential loss of funds.
- **Medium** - only a conditionally incentivized attack vector, with a moderate likelihood.
- **Low** - involves too many or unlikely assumptions; offers little to no incentive.

3.3 Actions required by severity level

- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

4 Executive summary

George Hunter was engaged by Maverick to review the Maverick V2 AMM protocol during the week of February 5th, 2024, and from February 19th to March 1st, 2024.

Overview

Project Name	Maverick V2
Repository	https://github.com/maverickprotocol/maverick-v2
Commit hash	f8f40dd7a3b30be68ff980db3dbd9e35a90a143e
Resolution	cb6edc89312f0cfff6ab008d2568d122b421b938
Methods	Manual review

Timeline

-	February 1st, 2024	Audit kick-off
v0.1	March 8th, 2024	Preliminary report
v1.0	March 10th, 2024	Mitigation review
v1.1	March 13th, 2024	Final commit update

Scope

v2-amm/contracts/*
v2-common/contracts/libraries/*

Issues Found

High risk	0
Medium risk	0
Low risk	3
Informational	1

5 System overview

Maverick v2 AMM is a dynamic distribution AMM where LPs can choose to have their liquidity automatically move to stay near the price. The outward-facing functionality of the v2 AMM is almost identical to v1, but the underlying mechanisms have changed significantly to lower gas costs and allow for more flexible liquidity movement options.

5.1 Codebase maturity

Code complexity - *Moderate*

The protocol exhibits a high level of complexity and innovation. It encompasses numerous parameters, both input, and state variables, which determine the execution path of each swap. There is a multitude of mathematical equations that necessitate thorough testing and verification.

Security - *Satisfactory*

No significant issues were identified during the security review. The team has planned multiple sequential audits to further ensure security. However, the complexity introduces a heightened risk of overlooking vulnerabilities amidst the numerous state transitions.

Decentralization - *Strong*

The protocol is immutable and possesses minimal privileged role addresses, solely tasked with setting the protocol fee ratio and claiming accumulated fees.

Testing suite - *Satisfactory*

The test suite is comprehensive, comprising full unit tests and multiple fuzzing tests. Considering the nature and complexity of the protocol, it is advisable to undergo additional testing by experts in formal verification and invariant testing.

Documentation - *Strong*

The protocol's whitepaper thoroughly describes all features and mathematical formulas.

Best practices - *Strong*

The code is well-written and highly optimized.

5.2 Privileged actors

- Factory owner - has the ability to update the protocol fee ratio and collect the accumulated protocol fees.
- Accessor of a permissioned pool - a proxy (e.g. a router contract) that deployers of permissioned pools can set upon deployment to be the only address with access to the pool's functionalities.

5.3 Threat model

What in Maverick V2 protocol has value in the market?

Liquidity providers' LP tokens and tokens reserves.

What are the worst-case scenarios for the Maverick V2 protocol and its users?

- Draining a liquidity pool's reserves.
- Improper accounting of user funds allowing an adversary to steal value from the protocol or other users.

- Bricking a core functionality (Denial-of-Service) - adding/removing liquidity, swapping and migrating bins.
- Unauthorized access to a permissioned pool.

What are the main entry points and non-permissioned functions?

- `MaverickV2Pool.addLiquidity()` - Adds liquidity to the bins of the passed ticks that belong to a certain kind and mints LP tokens corresponding to the recipient's share in those bins.

- `MaverickV2Pool.removeLiquidity()` - Burns LP tokens from the caller's specified position and withdraws the corresponding amount of `tokenA` and/or `tokenB`.

- `MaverickV2Pool.swap()` - Executes a swap in the pool, updating the reserves of the passed ticks and bins, moving the passed bins in their respective directions, and executing the callback function on the caller if provided.

- `MaverickV2Pool.migrateBinUpStack()` - Migrate bins up the linked list of merged bins so that its `mergeId` is the current active bin.

- `MaverickV2PoolFactory.create()` - Creates a non-permissioned pool and configures it with the provided parameters and protocol fee ratio.

- `MaverickV2PoolFactory.createPermissioned()` - Creates a permissioned pool that only allows access to an `accessor` address and configures it with the provided parameters and protocol fee.

5.4 Useful resources

The following resources were used throughout the audit to expedite the understanding phase of the codebase and verify the defined invariants and requirements.

- *Maverick V2 white paper*
- *Maverick V1 protocol contracts*
- *Uniswap V3 contracts*

6 Findings

6.1 Low

6.1.1 An adversary can front-run a pool creation and set an invalid starting activeTick

Severity: *Low*

Description: The MaverickV2PoolFactory uses *create2* to deploy the pools smart contract on a predefined deterministic address which is computed via the parameters used to configure the pool like `fee`, `tickSpacing`, `lookback`, etc. There is only one parameter used to initialize the pool that is not included in the create2 salt computation arguments and this is the initial `activeTick`. It is correctly not included since we don't want a pool's address to change based on the initial price upon deployment.

However, the problem comes from the fact that there is no input validation performed on the `activeTick`. Therefore, a malicious user can front-run the deployment of any pool and set the starting `activeTick` as high as *type(int32).max* or as low as *type(int32).min* which would misconfigure the pool as these values are far outside the valid tick range a pool:

```
uint256 constant MAX_TICK = 322378; // max price 1e14 in D18 scale
int32 constant MAX_TICK_32 = int32(int256(MAX_TICK));
int32 constant MIN_TICK_32 = int32(-int256(MAX_TICK));
```

Recommendation: Consider limiting the `activeTick` value in the constructor of MaverickV2Pool.

Resolution: Acknowledged. Cleint's comment: A malicious user can set `activeTick` to an unreasonable value, but limiting to the MIN/MAX tick is not an effective countermeasure as the attacker would still have created a poorly-priced pool. A legitimate user can simply modify the fee/width/lookback slightly to create a pool that has an appropriate activeTick. Creating pools is expensive for the attacker with no upside other than causing grief, so this is an unlikely attack in practice and one that has an easy countermeasure.

6.1.2 Math.limitTick incorrectly limits the passed tick value

Severity: *Low*

Description: The `Math.limitTick` method is used to ensure that the tick values used in the protocol are in correct range of [-322378,322378]. The problem is that it checks the tick value itself rather than it multiplied by the `tickSpacing` corresponding pool is using as is done in `TickMath.subTickIndex`:

```
function subTickIndex(uint256 tickSpacing, int32 _tick) internal pure returns (
    uint32 subTick) {
    subTick = Math.abs32(_tick);
    subTick *= uint32(tickSpacing); // @audit not done in ‘Math.limitTick‘
    if (subTick > MAX_TICK) {
        revert TickMaxExceeded(_tick);
    }
}
```

Recommendation: Consider passing the tick multiplied by the `tickSpacing` to `Math.limitTick`.

Resolution: Resolved. Client's comment: After more consideration, it is clear that there is no need to validate the `tickLimit` in swap. If a user uses a `tickLimit` beyond the limit, their swap will just revert with an out of gas error. We removed this function and its use in the swap function. This has the additional benefit of saving gas on swaps.

6.1.3 Accumulated protocol fees may be lost if the owner renounces ownership

Severity: *Low*

Description: The `renounceOwnership` function has been overriden by the MaverickV2PoolFactory to implement the following check:

```
function renounceOwnership() public override(IMaverickV2FactoryAdmin, Ownable)
    onlyOwner {
    if (protocolFeeRatio != 0) revert FactoryProtocolFeeOnRenounce(protocolFeeRatio)
        ;
    super.renounceOwnership();
}
```

The purpose of this check is to not allow the factory owner to renounce ownership while protocol fees are still accumulating as there will be no one able to withdraw them. However, this does not completely mitigate the problem as if the owner renounces ownership of the factory and has not claimed any pending fees, they will remain locked forever.

Recommendation: Consider whether this would be a problem. A solution would be to implement a method that allows anyone to withdraw any pending fees after the ownership is renounced with a recipient set to some fee collector address.

Resolution: Acknowledged. The factory owner will have to take this into account if they ever renounce ownership.

6.2 Informational

6.2.1 Typographical mistakes, code-style suggestions and non-critical issues

Severity: *Informational*

Description: The contracts contain one or more typographical issues, code-style suggestions and non-critical issues. In an effort to keep the report size reasonable, we enumerate these below:

1. A wrong assumption is made in `TransferLib` regarding calldata decoding on the called contract:

```
// Append the "to" argument. erc20 transfer has address as the
// first argument, so only the first 160bits of ‘to‘ are read
// therefore we do not need to mask even though bits 161-256 may be
// dirty.
```

If the `to` address contains dirty bits, the `transfer` (and `transferFrom`) call will revert. Consider either implementing the needed mask on the `to` (and `from`) address or directly reusing the solmate version of the library by importing it.

2. Missing `iszero(extcodesize(token))` in TransferLib - a well-known issue in the solmate's SafeTransfer library is that if the token contract address is actually an EOA (has no deployed code) the transfer will succeed. This could theoretically be exploited by creating a pool for a token

that still does not exist on-chain, but its deployment address is known in advance (e.g. it is deployed via `create2` on multiple blockchains). However, it causes no impact to the Maverick V2 AMM protocol since the only tokens used are the ones passed via the factory's `deploy` function which only executes with existing token contracts.

3. There is no need to place the if-revert statements in an unchecked block in TransferLib.
4. `Math.msb`, `Math.lsb` and `Cast.toUint40` are never used.
5. `uniqueBinsCheck` can be renamed to `uniqueTicksCheck` as it uses ticks IDs rather than bins.
6. Unsafe downcast in `Math.floor`.
7. The comment `Equivalent to require(denominator != 0 && (x == 0 || (x * y)/ x == y))` in `Math.mulDivDown` is not completely correct. Consider removing the `denominator != 0 &&` part.
8. `denominator := add(denominator, 1)` can be rewritten to `denominator := 1` in `Math.mulDivDown` and `Math.mulDivUp`.
9. `UPDATE_INTERVAL_MIN` is not needed.
10. Redundant line in `Delta.sol` - `self.swappedToMaxPrice = delta.swappedToMaxPrice;`.
11. Typo in the code license of Bin.sol - `UNLICENSE` -> `UNLICENSED`.
12. Several structs can be removed to improve readability - `IMaverickV2Pool.BinState` + `params` in `addLiqudity`, `removeLiquidity` and `swap`.
13. The white paper mentions that `bin.tickBalance` should be updated by `deltaLpBalance` instead of `deltaTickBalance`.
14. `_createChecks` checks that the pool fee is not **>=** to the `MAX_POOL_FEE` while `_checkProtocolFeeRatio` checks just **>**. Consider whether this is the intended behavior in both places.
15. `kinds` is checked only if it contains the static mode, but should also be checked if it contains any invalid modes (e.g. 0b00010001) by verifying it is less than 16 (i.e. that only 4 bits at most are occupied).
16. Users should not be able to create permissioned pools with `accessor` set to the null address. Consider implementing a sanity check.
17. A zero-address check can be added for the `user` parameter in `addLiquidity`.
18. The NatSpec comment for `lookback` is incorrect - should use `D2` instead of `D18`.
19. There is no explicit check whether the passed `binId` is valid in `migrateBinUpStack` and `removeLiquidity`. Consider implementing a sanity check.
20. Commented-out line and function - `// TwaState internal twa;` and `getTwa()`.
21. `addLiquidityByReserves` uses `mulDivDown` while `deltaTickBalanceFromDeltaLpBalance` uses `mulDivCeil`. Consider whether these are the intended rounding directions in both places.
22. Swap `params` can be stored in `calldata` instead of `memory`.
23. Consider whether reverting would be more appropriate in `limitTick` if the passed tick is out-of-range.

24. `Math.floor(-214748364800)` returns `-2147483648`, but `Math.floor(-214748364801)` returns `2147483647` due to underflow.
25. There is no need to explicitly check if a pool already exists in `create` and `createPermissioned`. If a pool with the same parameters has already been deployed, the *create2* operation will revert (in `Deployer`/`DeployerPermissioned.deploy`).

Recommendation: Consider fixing the above typographical issues, code-style suggestions and non-critical issues.

Resolution: Resolved.