Maverick V2 Supplemental Security Review

Report Version 1.0

April 26, 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 Supplemental smart contracts during the period from March 18th, 2024 to March 26th, 2024.

Overview

Project Name	Maverick V2 Supplemental contracts
Repository	https://github.com/maverickprotocol/maverick-v2
Commit hash	47287a62e15ca8d4bcabf7e0b6757debb5d10593
Resolution	175f8c39b19df69134add3aa8a2a042ce3047763
Methods	Manual review

Timeline

-	March 18th, 2024	Audit kick-off
v0.1	March 30th, 2024	Preliminary report
v1.0	April 26th, 2024	Mitigation review

Scope

All files in *v2-supplemental/contracts/*, excluding:
v2-supplemental/contracts/libraries/LiquidityUtilities.sol
v2-supplemental/contracts/MaverickV2PoolLens.sol
v2-supplemental/contracts/MaverickV2Quoter.sol
v2-supplemental/contracts/PositionImage.sol

Issues Found

High risk	0
Medium risk	0
Low risk	0
Informational	6

5 System overview

The Maverick V2 Supplemental protocol consists mainly of the MaverickV2Router and MaverickV2LiquidityManager contracts. They provide helper function for users to interact with the core AMM protocol's pools and more easily manage their liquidity positions.

The MaverickV2LiquidityManager helps users mint either NFT liquidity positions or boosted positions which are fungible. The MaverickV2Router supports both push-based and callback-based swaps. Push-based swaps help avoid using a callback function and are generally more gas efficient, but are only suitable for exact-input swaps since the swap's *amountIn* should be deposited into the pool before the swap execution. On the other hand, callback-base swaps make use of the implemented *maverickV2SwapCallback* in the router which transfers the tokens to the pool during the swap which is more suitable for exact-output swaps.

There are numerous libraries helping with encoding and decoding of parameters used within the protocol as well as abstract contracts defining common functionalities. Transactions batching is supported by the router, liquidity manager and MaverickV2Position contracts by inheriting from the Multicall. The contracts also support ERC-2612 permit (both the old and new standard interface) to make approving tokens easier for users.

5.1 Codebase maturity

Code complexity - *Satisfactory*
The code is straightforward and consists only of helper methods that users may use to interact with the core AMM protocol. There is some level of inheritance, but no heavily nested function calls.

Security - *Strong*
No significant issues were identified during the security review. The team has planned multiple sequential audits to further ensure security.

Decentralization - *Strong*
The protocol is immutable and has no privileged admin roles that can affect users funds or contract's availability and solviency.

Testing suite - *Strong*
The test suite is comprehensive, comprising full unit tests and multiple fuzzing tests.

Documentation - *Satisfactory*
There are sets of NatSpec and inline code comments that thoroughly describe implementation details, functions' behaviors and requirements as well as security considerations and key invariants.

Best practices - *Strong*
The code is well-written adhering to all best practices.

5.2 Privileged actors

- None.

5.3 Threat model

What in the Maverick V2 Supplemental protocol contracts has value in the market?

Any approved funds to the router and liquidity manager contracts. LP tokens and assets held and managed by the NFT and boost positions.

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

- Approved tokens to the router or liquidity manager getting drained.
- Unauthorized user getting access to the permissioned MaverickV2Position functions like removing liquidity and burning tokens.
- Incorrect pool or malicious user getting access to a permissioned callback function.
- User not receiving their appropriate share of a boosted position or misaccounting in general.
- Denial-of-Service on withdrawal/redeeming/burning of NFT or boosted positions, i.e. not being able to remove liquidity.
- Stealing deposited token during middle of a multi-call transaction via sweep.
- Incorrect encoding/decoding of input parameters leading to unexpected execution of functions.
- Slippage check not working in router.
- Impersonating a payer.
- Incorrectly accounting for bin balances in boosted position.
- Over- or under-minting/burning tokens for a BP.

5.4 Useful resources

The following resources were used throughout the audit to expedite the understanding phase of the codebase:

- *Maverick V2 Core Protocol*

6 Findings

6.1 Informational

6.1.1 Dirty bytes after decoding of packed elements

Severity: *Informational*

Context: PackLib.sol#L59-L64, PackLib.sol#L138,

Description: There are two places in the library contract where dirty bytes are left/injected after decoding of packed elements.

The first function is *PackLib.unpackInt32Array* which accepts an encoded/packed array of int32 variables as type *bytes* and returns their unpacked version after decoding the bytes in *_unpackArray*:

```
for {
    let i := 0
    let arrayCursor := add(array, 0x20) // skip array length
    let packedCursor := packedPointer
} lt(i, arrayLength) {
    // Loop until we reach the end of the array
    i := add(i, 1)
    arrayCursor := add(arrayCursor, 0x20) // increment array pointer by one word
    packedCursor := add(packedCursor, elementBytes) // increment packed pointer by
        one element size
} {
    mstore(arrayCursor, shr(padLeft, mload(packedCursor))) // unpack one array
        element
}
```

The problem here lies in the *for* loop body where the actual elements unpacking happens. Since each element's bits are shifted to the right with the needed padding, the rest *padLeft* number of bits will be 0-bits. However, this shouldn't be the case when the packed *int32* element is a negative number because the rest of the bits should actually be all 1s as that's how negative numbers are represented:

```
// packing and unpacking -1 would store:
// 0x000000000000000000000000000000000000000000000000ffffffffffffffff

// at the arrayCursor, while it should store:
// 0xffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff

// (since this is the representation of int32(-1))
```

The other function returning a result with dirty bits is *toAddressAddressBoolUint128Uint128*:

```
addr1 := and(0xffffffffffffffffffffffffffffffffffffffff, mload(add(_bytes, 20)))
addr2 := and(0xffffffffffffffffffffffffffffffffffffffff, mload(add(_bytes, 40)))
temp := mload(add(_bytes, 41))
amount1 := mload(add(_bytes, 57))
amount2 := mload(add(_bytes, 73))
```

Here, *amount1* and *amount2* are both of type *uint128*. However, the according mask is not applied, hence the variables contain dirty bits when returned.

After tracing the call trees and checking where the dirty variables are used, the conclusion is that there is no practical issue as they are never used in assembly again or if they are, they have been casted before that, effectively cleaning the dirty bits explicitly.

Recommendation: Despite the lack of an exploitable scenario, consider adding the appropriate masks in order to prevent from potential vulnerabilities in the future where the same functions are reused.

Resolution: Partially resolved. The second issue is fixed by clearing the dirty bits stored in *amount1* and *amount2*. The first one is acknowledged by the team as there is currently no impact.

6.1.2 Not all tokens use a max allowance value of type(uint256).max

Severity: *Informational*

Context: SelfPermit.sol#L98

Description: The *SelfPermit* library implements a set of 4 functions that execute a permit for the caller. The *selfPermitAllowedIfNecessary* calls the old ERC-2616 standard function interface on the passed token only if the allowance is not *type(uint256).max* (since this interface does not specify an amount, but general approval for whole balance):

```
if (IERC20(token).allowance(msg.sender, address(this)) < type(uint256).max)
    selfPermitAllowed(token, nonce, expiry, v, r, s);
```

A potential problem is that some tokens like UNI and COMP use *type(uint96).max* instead of *type(uint256).max* as their maximum allowance. Therefore, if such token is to be used, the *selfPermitAllowedIfNecessary* will not behave correctly. However, UNI implements the new ERC-2612 permit interface, so this function would not be used with it, and COMP does not support permit at all. Since there are no other well-known and widely used tokens we assign just *Informational* severity.

Recommendation: The fix for this issue is non-trivial. Given that there are no well-known instances of such tokens and that the issue is not critical even if such token is used, this behavior may just be acknowledged or documented.

Resolution: Acknowledged.

6.1.3 Re-entrancy vulnerability attack surface

Severity: *Informational*

Context: MaverickV2Position.sol#L42,

Description: There are two spots where the code does not completely adhere to the Checks-Effect-Interactions pattern and there's increased risk of a re-entrancy attack vector: *MaverickV2Position.mint* and *_removeLiquidityAndUpdateBalances*.

In *MaverickV2Position.mint*, *ERC721.safeMint* is used which calls a callback function on the recipient. No severe attack vector could be identified here. The only concern is that a user may reenter into the mint function again and mint a higher tokenId first (i.e. start minting position #1, reenter, mint position #2 first, go back to minting #1), which would only lead to unexpected order of emitted events.

In *_removeLiquidityAndUpdateBalances*, the concern is that the *binBalances* storage array is updated only after *pool.removeLiquidity* is executed which makes an untrusted call to the *tokenA* and/or *tokenB* which could be ERC777 tokens. Since the transfer happens after the bin balances have been updated in the pool, but are not yet updated in the boosted position, a malicious user may theoretically reenter into the position or any other contract that relies on the *binBalances* value. One such function is the *_removeLiquidityAndUpdateBalances* itself. However, the reentrancy guards on the MaverickV2 pool itself will prevent this from happening. No other meaningful vector could be identified except read-only reentrancy in a potential third-party protocol that uses the *binBalances* array for some purpose.

Recommendation: Consider whether following the Checks-Effect-Interactions pattern here would be more appropriate to ensure the *binBalances* are correctly tracked. Consider also documenting this behavior in the form of a code comment if no other mitigation is taken.

Resolution: Resolved. The recommendation was implemented.

6.1.4 Insufficient input validation in boosted positions factory

Severity: *Informational*

Context: MaverickV2BoostedPositionFactory.sol#L46-L49

Description: The *MaverickV2BoostedPositionFactory* allows users to create either static or dynamic boosted positions based on the kind of the bin IDs.

The problem is that it is not checked whether each of the passed bins' kinds is the same as the *kind* parameter used to initialize the position. Therefore, it is possible to create a pool with bins and kind that do not match.

Recommendation: Consider checking that each bin's kind is the same as the one passed.

Resolution: Resolved. The recommendation was implemented.

6.1.5 Funds may be swept during a multi-call if an untrusted external call is made

Severity: *Informational*

Context: Payment.sol#L37-L44

Description: The router and liquidity manager contracts inherit from the Multicall abstract contract in order to support transactions batching. They also inherit from the Payment abstract contract meaning that any funds held within these contracts can be swept.

Although it is explicitly mentioned that these contract should not hold any funds unless in a multi-call, a security concern when it comes to this type of design is that any untrusted external call made as part of the batch can reenter and sweep the funds that are temporarily held by the contract.

Recommendation: Consider whether this could become a problem. No specific attack vector could be identified with the current version of the contracts.

Resolution: Acknowledged.

6.1.6 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. Typographical mistakes on the following lines - *L18*, *L19*, *L25*, *L32*, *L42*, *L46*, *L26*, *L21*, *L9*.
2. The functions defined by the *IChecks* and *IState* interfaces could be marked as *view*.
3. The returned variable's name in *_getBinIdBytes* is *ratiosBytes* while it should be *binIdsBytes*.
4. *BytesLib.slice* will not revert with a custom error as expected in case of an overflow of the length, but will instead throw an EVM error for arithmetic overflow.
5. *BytesLib.toBool* and *BytesLib.toAddressAddressBoolUint128Uint128* could use a named return parameter and *eq(temp, 1)* in the assembly block rather than doing the boolean conversion in Solidity.
6. The *padLeft* variable in *PackLib._unpackArray* should be called *padRight*.
7. *PoolInspection.userSubaccountBinReserves* can return early (after the *while* loop) in case *userBinLpBalance* is 0 to save gas.
8. The *INFT.PositionTokenIdInvalid* custom error is never used.
9. Consider exposing public getter functions for *tokenAScale* and *tokenBScale* in the Maverick V2 pools to save gas for other protocols trying to obtain these values via: *Math.scale(tokenX().decimals());*. This is done in *PoolInspection.userSubaccountBinReserves*.
10. The *IRatioBinIdGetter* interface is not used.
11. The comment in *Payment.pay* is redundant.
12. The *Nft* abstract contract does not need to inherit *ERC721* explicitly since it is already inherited by *ERC721Enumerable*.
13. *Nft.tokenURI* does not need to execute *_requireOwned* as it is already done in *ownerOf* called on the next line.
14. Add *&& payer == address(this)* to the first check in *Payment.pay*.

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

Resolution: Resolved.