Maverick PoolPosition Audit

Smart Contract Security Assessment

March 31, 2023

ABSTRACT

Dedaub was commissioned to perform a security audit of the PoolPosition component of the Maverick protocol. The audited contracts concerned the PoolPositions accounting and operation, their management and associated rewards mechanisms.

The PoolPosition component of Maverick allows protocols and market makers to create a "PoolPosition" (PP), which is a distribution of liquidity within a given pool. LPs can join this PP by adding assets proportional to the existing PP asset mix. PP LPs will collect only a portion of the fees generated by their liquidity, but they will be compensated with LP incentives that any party can add to the LP Rewards contract.

One critical vulnerability that could allow a malicious LP to mint an unlimited amount of LP shares was identified and fixed. A number of lower severity issues were also identified and patched, further improving the security and robustness of the system and achieving a high quality standard.

SETTING & CAVEATS

This audit report refers to the at the time private repository maverickprotocol/token-v1 of the Maverick protocol at commit `194be8ffec99a07aeb03fdd59534fed80968f3d2`.

The full list of audited contracts can be found bellow:

```
contracts/
├── phase2/
│   ├── IPoolPositionAndRewardFactorySlim.sol
│   ├── PoolPositionAndRewardFactorySlim.sol
│   ├── PoolPositionBaseSlim.sol
│   ├── PoolPositionDynamicDeployerSlim.sol
```

```
│   ├── PoolPositionDynamicSlim.sol
│   ├── PoolPositionStaticDeployerSlim.sol
│   └── RewardOpenSlim.sol
├── PoolPositionBase.sol
├── PoolPositionManager.sol
└── RewardBase.sol
```

The codebase, including all the audit fixes, was subsequently made available at the https://github.com/maverickprotocol/poolposition-v1 repo at commit 12f82cfc4ff2147507eff90247cb6b45c075105d. The PoolPositionBase contract was removed. The PositionInspector and PoolInformation contracts, which are part of the new repo, were not part of the audit.

Two auditors worked on the codebase for 4 days. Maverick Pool Positions are built upon the Maverick AMM, which was not part of this audit. Nevertheless, significant effort was put into understanding the bigger picture in order to minimize the risk of missing issues that require a full understanding of context.

The audit's main target is security threats, i.e., what the community understanding would likely call "hacking", rather than the regular use of the protocol. Functional correctness (i.e. issues in "regular use") is a secondary consideration. Typically it can only be covered if we are provided with unambiguous (i.e. full-detail) specifications of what is the expected, correct behavior. In terms of functional correctness, we often trusted the code's calculations and interactions, in the absence of any other specification. Functional correctness relative to low-level calculations (including units, scaling and quantities returned from external protocols) is generally most effectively done through thorough testing rather than human auditing.

VULNERABILITIES & FUNCTIONAL ISSUES

This section details issues affecting the functionality of the contract. Dedaub generally categorizes issues according to the following severities, but may also take other considerations into account such as impact or difficulty in exploitation:

Category	Description
CRITICAL	Can be profitably exploited by any knowledgeable third-party attacker to drain a portion of the system's or users' funds OR the contract does not function as intended and severe loss of funds may result.
HIGH	Third-party attackers or faulty functionality may block the system or cause the system or users to lose funds. Important system invariants can be violated.
MEDIUM	Examples: • User or system funds can be lost when third-party systems misbehave. • DoS, under specific conditions. • Part of the functionality becomes unusable due to a programming error.
LOW	Examples: • Breaking important system invariants but without apparent consequences. • Buggy functionality for trusted users where a workaround exists. • Security issues which may manifest when the system evolves.

Issue resolution includes "dismissed" or "acknowledged" but no action taken, by the client, or "resolved", per the auditors.

CRITICAL SEVERITY:

ID	Description	STATUS
C1	PoolPosition LP can mint an unlimited number of shares	**RESOLVED**

In function PoolPositionBaseSlim::mint there is no check that the fromTokenId is different from the tokenId in the pool.transferLiquidity call and at the same time there is no restriction on who can call PoolPositionBaseSlim::mint. This means that a malicious PoolPosition LP could reuse the liquidity of the PoolPosition, i.e., transfer it to itself by setting `fromTokenId` to `tokenId` without adding any new liquidity, but at the same time mint more PoolPosition LP shares to them as they are free to set mint's `to` parameter. Essentially the malicious LP would be able to mint an unlimited number of LP shares to them while completely diluting all other positions.

```
function mint(address to, uint256 fromTokenId, uint128 binZeroLpAddAmount)
    external override nonReentrant checkBin checkpointLiquidity
    returns (uint256 amountMinted)
{
    uint256 supply = totalSupply();
    amountMinted = supply == 0 ? ONE : Math.mulDiv(
        binZeroLpAddAmount, supply, pool.balanceOf(tokenId, binIds[0])
    );

    require(amountMinted != 0, "PP: zero mint");
    pool.transferLiquidity(
        fromTokenId, // Dedaub: function parameter which can be freely set
        tokenId, // Dedaub: storage variable representing the PoolPosition
        _binLpAddAmountRequirement(binZeroLpAddAmount)
    );
    // Dedaub: PoolPosition mints LP shares to to, a function parameter
    // that can be freely set by the attacker
    _mint(to, amountMinted);
}
```

The suggested prevention measure would be to disallow the `fromTokenId` parameter to be equal to the `tokenId` of the PoolPosition. An extra layer of security would be achieved if calls to PoolPositionBaseSlim::mint were restricted only to certain entities, e.g., just to the PoolPositionManager if the intent is for all LPs to go through it.

Great care should also be taken in future versions of the protocol to not allow PoolPositions to manage the liquidity of other PoolPositions (via approvals) as such functionality could be exploited to freely mint LP shares by transferring liquidity between PoolPositions unless further logic/restrictions, e.g., burning of LP shares, are implemented.

HIGH SEVERITY:

[NO HIGH SEVERITY ISSUES]

MEDIUM SEVERITY:

ID	Description	STATUS
M1	PoolPositionManager could be reentered to steal pending balances	**RESOLVED**

The PoolPositionManager contract has been developed to act similar to a router, trying to be as stateless as possible. However, any balances held in it between transactions are up for grabs by anyone via calls to the functions unwrapWETH9, sweepToken and refundETH. This property could prove dangerous when combined with any possibility of untrusted parties gaining control in the middle of a user transaction, as the untrusted party would be able to drain any temporary balances held in the contract.

PoolPositionManager allows batching of multiple actions including transfers to potentially untrusted recipients. If untrusted code is invoked at any point in the transfer, e.g., as the result of an `unwrapWETH9(amount, untrusted_recipient)` call, the code can re-enter the PoolPositionManager and claim any remaining balances.

```
function unwrapWETH9(
    uint256 amountMinimum, address recipient
) public payable override {
    uint256 balanceWETH9 = WETH9.balanceOf(address(this));
    require(balanceWETH9 >= amountMinimum, "Manager: Insufficient WETH9");
    if (balanceWETH9 != 0) {
        WETH9.withdraw(balanceWETH9);
        // Dedaub: _safeTransferETH can pass control to an untrusted party
        _safeTransferETH(recipient, balanceWETH9);
    }
}

function _safeTransferETH(address to, uint256 value) internal {
    // Dedaub: untrusted code could be called here
    (bool success, ) = to.call{value: value}(new bytes(0));
    require(success, "ETH transfer failed");
}
```

Such remaining balances could exist in the contract because the user intends to use them later or sweep them at the end of the batch of actions. The functions unwrapWETH9 and sweepToken (but not refundETH) of PoolPositionManager define the **amountMinimum** parameter that a user can use to protect themselves from malicious contracts stealing their tokens and having them receiving too little back. However, users might not be aware of this functionality or they might have trouble using it effectively due to the difficulty in estimating output amounts. We believe that a locking solution could be considered to make the execution of batches of actions bullet-proof against such reentrancy attacks.

LOW SEVERITY:

ID	Description	STATUS
L1	PoolPositionManager::mintPoolPosition does not revoke given approval	**RESOLVED**

The function PoolPositionManager::mintPoolPosition, which is called by the functions addLiquidityToPoolPosition and createPoolPositionAndAddLiquidity, approves the `poolPosition` for `_managerTokenId` so that the `poolPosition` can transfer the liquidity position minted to the PoolPositionManager to itself.

```
function mintPoolPosition(
    IPoolPositionSlim poolPosition,address recipient,uint256 lpTokenAmount
) private returns (uint256 mintedPoolPositionTokenAmount) {
    uint256 _managerTokenId = managerTokenId;
    // Dedaub: poolPosition is approved for _managerTokenId
    position.approve(address(poolPosition), _managerTokenId);
    mintedPoolPositionTokenAmount = poolPosition.mint(
        recipient, _managerTokenId, SafeCast.toUint128(lpTokenAmount)
    );
    // Dedaub: approval for _managerTokenId should be revoked by doing
    // position.approve(address(0), _managerTokenId);
    position.approve(address(poolPosition), _managerTokenId);
}
```

However, as can be seen in the above code snippet the approval is not successfully removed after the `poolPosition.mint(...)` call, as `address(0)` should be approved instead of `address(poolPosition)`. As the PoolPositionManager contract is not expected to hold any liquidity positions at the end of its execution we do not consider this issue a serious threat.

L2	PoolPositionBaseSlim burn functions have complex side-effects that could lead to loss of user funds	**RESOLVED**

Contract PoolPositionBaseSlim defines two ERC20Burnable-like burn functions, `burnFrom` and `burnFromToReserves`, that have more side-effects than the burn of tokens of an account.

```
function burnFrom(
    address account, uint256 toTokenId, uint256 lpAmountToUnStake
) external override nonReentrant {
    IPool.RemoveLiquidityParams[] memory params =
        _ppBurn(account, lpAmountToUnStake);
    pool.transferLiquidity(tokenId, toTokenId, params);
}

function burnFromToReserves(
    address account, address recipient, uint256 lpAmountToUnStake
) external nonReentrant returns (uint256 amountA, uint256 amountB) {
    IPool.RemoveLiquidityParams[] memory params =
        _ppBurn(account, lpAmountToUnStake);
    (amountA, amountB, ) =
        pool.removeLiquidity(recipient, tokenId, params);
}
```

The function burnFrom transfers the liquidity deposited into the underlying pool from the control of the current PoolPosition (`tokenId`) to another entity (`toTokenId`). Callers of `burnFrom` might incorrectly assume that when they specify another PoolPosition via the `toTokenId` their liquidity will be assigned to that PoolPosition (correct assumption) and LP shares of that PoolPosition will be minted to them (incorrect assumption), leading to lost funds. On the other hand, the complementary function `burnFromToReserves` unintuitively does what one might have expected from the function named `burnFrom`, as it just removes the liquidity from the pool and transfers it to the defined `recipient`. We believe that the naming of the aforementioned functions could be revisited to avoid misconceptions that could lead to loss of user funds. Extensive documentation of the intent and side-effects of each of these functions would be also very recommended.

L3	addLiquidityCallback access check is not effective	RESOLVED

The function PoolPositionManager::addLiquidityCallback implements the requirement `msg.sender == address(data.pool)` that does not actually protect from an attacker calling the function and manipulating any available balances, as the attacker can set `data.pool`, giving possibly a false sense of extra security.

```
function addLiquidityCallback(
    uint256 amountA, uint256 amountB, bytes calldata _data
) external {
    AddLiquidityCallbackData memory data =
        abi.decode(_data, (AddLiquidityCallbackData));
    require(factory.isFactoryPool(IPool(msg.sender)));
    // Dedaub: an attacker can set data.pool to whatever, even msg.sender
    require(msg.sender == address(data.pool));

    if (amountA != 0) {
        pay(data.tokenA, data.payer, msg.sender, amountA);
    }
    if (amountB != 0) {
        pay(data.tokenB, data.payer, msg.sender, amountB);
    }
}
```

Nevertheless, the PoolPositionManager contract is not expected to hold any tokens outside of a transaction's scope. Even if an attacker could intercept the execution flow of another user's transaction and call addLiquidityCallback, they would be stopped by the requirement `factory.isFactoryPool(IPool(msg.sender))`.

L4	Malicious deposits to RewardBase contract	DISMISSED

The RewardBase contract expects every transfer to it to be preceded by a call to `_notifyRewardAmount()`, in order to update the contents of the rewardData array.

However it is possible for a malicious attacker to transfer reward tokens which have already been whitelisted directly into this contract, without passing through `_notifyRewardAmount()`. A legitimate user could also cause this to occur by mistake.

When this occurs, certain variables inside the RewardData struct of the rewardData array will be smaller than they should actually be for the given reward token. This occurs because the amount parameter of _notifyRewardAmount is not added to the various quantities calculated by _notifyRewardAmount. Other variables in the struct will simply not be updated.

The variables inside the RewardData struct which are affected are the following:

- rewardData[rewardTokenIndex].rewardPerTokenStored
- rewardData[rewardTokenIndex].escrowedReward
- rewardData[rewardTokenIndex].rewardRate

Also as a result of _notifyRewardAmount not running, the following are not updated:

- rewardData[rewardTokenIndex].finishAt
- rewardData[rewardTokenIndex].updatedAt

There are a number of view functions which will thus report stale reward data:

- rewardInfo()
- earnedList()
- earned()

A number of stateful functions are also affected:

- _getReward() - not problematic because it is currently unused.
- updateAllRewards() - calls updateRewards(). Since the malicious deposit did not update the timestamps rewardData[rewardTokenIndex].finishAt and rewardData[rewardTokenIndex].updatedAt, the values are stale. If the deposit

had been notified correctly, the duration computed by the function could have been different. Thus when updateRewards() is called next, and calls deltaRewardPerToken(), the timeDiff could also have been different, leading to a different reward.

- recoverERC20() - this function returns remainingRewards to the owner. Now remainingRewards is calculated by subtracting the escrowedReward (which does not track the malicious deposit) from the balance of rewards (which includes the malicious deposit). Hence in this case, the owner would be able to withdraw more rewards than usually permitted by the protocol.
- A follow up call to _notifyRewardAmount() - this will also calculate the remainingRewards by subtracting the escrowedReward from the balance of rewards. It will then check whether the amount being deposited (this time legitimately) is greater than the remainingRewards, to determine whether the legitimate user can set the rate. However, thanks to the malicious deposit remainingRewards may be greater than expected. Therefore the legitimate user may not get to set the rate after all. This may be a problem if the legitimate user relies on the issuance of the NotifyRewardAmount event to determine what he needs to deposit in order to change the rate, as the malicious deposit competing with him is not visible to him through the event log.

L5	RewardBase exposes implementation details to the outside world	**RESOLVED**

The RewardBase contract exposes implementation details around the `tokenIndex`, `rewardData` and `globalActive` storage variables to the outside world. Even though we have not identified any concrete scenarios where this could make the protocol vulnerable to exploitation, we discuss below how we believe the contract could be made more robust.

Array `RewardBase::rewardData` and bitmap `RewardBase::globalActive` reserve their first element for the "empty" token, an empty value that is used as a placeholder so that index 0 can be used as a sentinel value in `RewardBase::tokenIndex`, i.e., if `tokenIndex[X] == 0`, `X` is not in `rewardData`. The RewardBase contract is implemented in such a way that the "empty" token data can be accessed by several functions, leading to partially empty returned data or execution reversions.

- The functions `rewardInfo` and `earnedList` returns an array of length `rewardData.length` where the element at position 0 is empty as it refers to the "empty" token. The returned array could omit the "empty" token and return only data regarding valid tokens. That would also simplify the logic of the consumers of these data.
- The functions `_getReward` and `_getRewardList` allow users to access and update the empty data if they provide 0 as the `rewardTokenIndex` parameter. This is possible partially because bit `0` of the `globalActive` bitmap is set in the constructor signifying that the "empty" token is *active*, which allows it to bypass the check in `_getReward`.
- The functions `removeStaleToken` and `recoverERC20` also allow accessing the "empty" token's data but the transaction eventually fails as a contract call is attempted on address 0.

L6	RewardBase does not emit events when reward tokens are added or removed	**RESOLVED**

The RewardBase contract does not emit an event when a new reward token is added or a stale one is removed. Certain users could benefit from such an addition as they would be able to monitor the contract for certain events and react appropriately.

CENTRALIZATION ISSUES:

It is often desirable for DeFi protocols to assume no trust in a central authority, including the protocol's owner. Even if the owner is reputable, users are more likely to engage with a protocol that guarantees no catastrophic failure even in the case the owner gets hacked/compromised. We list issues of this kind below. (These issues should be considered in the context of usage/deployment, as they are not uncommon. Several high-profile, high-value protocols have significant centralization threats.)

ID	Description	STATUS
N1	Reward factory's owner can drain the reward contract	**RESOLVED**
The function `RewardBase::recoverERC20` allows the owner of the factory contract to recover ERC20 tokens that are registered as reward tokens. What is more, as the staking token of the RewardBase contract can be listed/approved as a reward token by the factory's owner, the whole staking token balance in the contract could be drained if the owner's key got compromised. The protocol developers could consider disallowing the recovery of staking tokens as a possible mitigation of the aforementioned threat.		

OTHER / ADVISORY ISSUES:

This section details issues that are not thought to directly affect the functionality of the project, but we recommend considering them.

ID	Description	STATUS
A1	Removal of approved reward tokens is not supported	**DISMISSED**

The PoolPositionAndRewardFactorySlim contract implements functionality to add tokens as approved rewards via the addNewApprovedRewardToken function but does not support their removal via a removeApprovedRewardToken function in case the governors of the protocol decided that a token should not longer be approved.

ID	Description	STATUS
A2	AddLiquidityCallbackData can be shrinked to save gas	**RESOLVED**

The PoolPositionManager::AddLiquidityCallbackData struct can be shrinked to save gas, as when it is created in PoolPositionManager::addLiqudity the `tokenA` and `tokenB` parameters are set to pool.tokenA() and pool.tokenB() respectively while the pool is also saved in the struct.

```
AddLiquidityCallbackData memory data = AddLiquidityCallbackData(
    {tokenA: pool.tokenA(), tokenB: pool.tokenB(), pool: pool, payer: msg.sender}
);
```

The struct could be simplified to just the pool and the payer while the tokens could be retrieved when needed via the two appropriate calls to the pool.

```
struct AddLiquidityCallbackData {
    IPool pool;
    address payer;
}
```

A3	PoolPositionBase contract's initializeContract function can be called multiple times	**RESOLVED**

The PoolPositionBase contract has an initializeContract() function which sets the voteReward contract and can be called multiple times. If this function is a setter function it is recommended that it be named appropriately. If not, multiple initialisation should not be allowed just in case the approved callers are compromised.

The PoolPositionBase contract was removed from the current version of the Pool Positions system by the Maverick team.

A4	RewardBase::recoverERC20 allows recovering only tokens that are registered as reward tokens	**RESOLVED**

The function `RewardBase::recoverERC20` allows the owner of the factory contract to recover ERC20 tokens that are registered as reward tokens but not any other token. This would mean that if tokens of significant value were sent to the contract by mistake or due to any other currently unforeseeable reason, the only way to retrieve them would involve approving that token as a reward token. Of course, that could be infeasible or non-trivial due to the state of the contract at the time (the number of reward tokens is limited). The developers team could consider the implementation of a "recover" function that would handle non-reward tokens.

A5	PoolPositionBase is an abstract contract but the claimFeeToVoters function has empty implementation	**RESOLVED**

The PoolPositionBase contract is an abstract contract. It implements the `claimFeeToVoters()` function, but leaves the implementation empty. Since it is an abstract contract, it does not need to provide an implementation for this function, to be compliant with the IPoolPosition interface. This function can be left virtual so that concrete base classes of PoolPositionBase do not forget to implement it.

The PoolPositionBase contract was removed from the current version of the Pool Positions system by the Maverick team.

A6	Storage data could be deleted to save gas	**RESOLVED**

The function `RewardBase::_removeStaleToken` does not delete the `updatedAt` field of the `RewardData` struct stored in storage at `rewardData[rewardTokenIndex]`. Doing so would grant a gas refund.

A7	Unused storage variables	**RESOLVED**

There are a few storage variables that are not being used:

- `PoolPositionAndRewardFactorySlim::lpRewardToVoteReward`
- `PoolPositionAndRewardFactorySlim::postDeploy`
- `PoolPositionDynamicSlim::activeBins`

A8	Documentation of require in PoolPositionBaseSlim's constructor	**RESOLVED**

The PoolPositionBaseSlim contract has a constructor which performs the following check:

```
if ((isStatic && bin.kind != 0) || !(_binIds[i] > lastBinId))
    revert InvalidBinIds();
```

It is recommended to document clearly that the constructor expects the items in its `_binIds` array parameter to be in ascending order. This check is meant to rule out duplicate elements in this array, but has the side effect of reverting if these are not presented in ascending order.

A9	Struct with only one element in IPoolPositionAndRewardFactorySlim	**DISMISSED**

The IPoolPositionAndRewardFactorySlim interface has a RewardInfos struct containing just one element. This can be simplified by just using the underlying array instead of the struct.

```
// Dedaub: IPoolPositionAndRewardFactorySlim has the following struct:
struct RewardInfos {
    IReward.RewardInfo[] rewardInfoList;
}
```

A10	It is possible to create pool positions with identical liquidity distributions	**DISMISSED**

The `createPoolPositionAndRewards()` function of the PoolPositionAndRewardFactorySlim contract makes no checks on whether a pool position with identical ratios has already been created. In fact it is possible for users to create such positions by calling the `createPoolPositionFromDeltas()` function of the PoolPositionManager. This issue is being flagged here because having multiple positions with identical distributions in the same pool may cause liquidity to be dispersed among these positions, which may not be a desirable outcome for the protocol.

A11	Not all of the internal and private functions conform to the naming style	**RESOLVED**

There are several internal and private functions that do not conform to the naming style that requires their name starting with an underscore. A non-exhaustive list can be found below:

- PoolPositionBaseSlim::tokenBinReserves
- PoolPositionManager::mintPoolPosition
- PoolPositionManager::createPoolPositionFromDeltas
- PoolPositionManager::addLiquidity
- PoolPositionManager::pay

• PoolPositionBase::abTransfer		
A12	Incorrect code comment in RewardBase::RewardData	**RESOLVED**
The code comment describing the `resetCount` field of the `RewardData` struct in contract RewardBase is identical to that for the `rewards` field and does not make sense.		
A13	Inaccurate function name transferAndNotify	**RESOLVED**
The function `RewardBase::transferAndNotify` should be renamed to `notifyAndTransfer` as it first calls `_notifyRewardAmount` and then transfers the token from the `msg.sender` to the contract via a `safeTransferFrom` call.		
A14	Floating version pragma in contracts	**DISMISSED**
The floating version `pragma solidity ^0.8.0` allows contracts to be compiled with any version of the Solidity compiler ranging from 0.8.0 to 0.9.0. Even though versions might not differ drastically, floating pragmas should be avoided and the pragma should be fixed to the version that will be used for the contracts' deployment.		
A15	Compiler version and possible bugs	**DISMISSED**
The code can be compiled with Solidity versions ^0.8.0. According to the foundry.toml file of the codebase, version 0.8.17 is currently used which has some known bugs, which we do not believe affect the correctness of the contracts.		

DISCLAIMER

The audited contracts have been analyzed using automated techniques and extensive human inspection in accordance with state-of-the-art practices as of the date of this report. The audit makes no statements or warranties on the security of the code. On its own, it cannot be considered a sufficient assessment of the correctness of the contract. While we have conducted an analysis to the best of our ability, it is our recommendation for high-value contracts to commission several independent audits, a public bug bounty program, as well as continuous security auditing and monitoring through Dedaub Watchdog.

ABOUT DEDAUB

Dedaub offers significant security expertise combined with cutting-edge program analysis technology to secure some of the most prominent protocols in DeFi. The founders, as well as many of Dedaub's auditors, have a strong academic research background together with a real-world hacker mentality to secure code. Protocol blockchain developers hire us for our foundational analysis tools and deep expertise in program analysis, reverse engineering, DeFi exploits, cryptography and financial mathematics.