Zellic

Maverick

Smart Contract Security Assessment

May 7, 2022

Prepared for:

Bob Baxley and Alvin Xu

Maverick Protocol

Prepared by:

Aaron Esau and Ayaz Mammadov

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Introduction

1.1 About Maverick

Maverick is a DeFi ecosystem that brings open, transparent, and efficient markets to everyone. At the heart of this ecosystem is an Automated Market Maker (AMM) powered by an innovative automated liquidity placement mechanism. Maverick offers effortless efficiency to liquidity providers, competitive pricing to traders, and zero-maintenance market making to projects.

1.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these "shallow" bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, etc. as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, etc.

Complex integration risks. Several high-profile exploits have been the result of not any bug within the contract itself, but rather an unintended consequence of its interaction with the broader DeFi ecosystem. We perform a meticulous review of all of the contract's possible external interactions, and summarize the associated risks; for example: flash loan attacks, oracle price manipulation, MEV/sandwich attacks, etc.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines, or code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, etc.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document, rather than impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their importance may differ. This varies based on numerous soft factors, such as our clients' threat models, their business needs, project timelines, etc. We aim to provide useful and actionable advice to our partners that consider their long-term goals, rather than simply a list of security issues at present.

1.3 Scope

The engagement involved a review of the following targets:

Maverick Contracts

Repository	https://github.com/nexuslabsdev/swap-contracts/
Versions	`cd330409c4b526f061429adfe123c581649d5988`
Type	Solidity
Platform	EVM-compatible

1.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants, for a total of 4 person-week. The assessment was conducted over the course of 2 calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-Founder
jazzy@zellic.io

Stephen Tong, Co-Founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Aaron Esau, Engineer
aaron@zellic.io

Ayaz Mammadov, Engineer
ayaz@zellic.io

1.5 Project Timeline

The key dates of the engagement are detailed below.

April 25, 2022	Start of primary review period
April 26, 2022	Kick-off call
May 7, 2022	End of primary review period
TBD	Closing call

1.6 Disclaimer

This assessment does not provide any warranties on finding all possible issues within its scope; i.e., the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees on any additional code added to the assessed project after our assessment has concluded. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program. Finally, this assessment report should not be considered financial or investment advice.

2 Executive Summary

Zellic conducted an audit for Maverick Protocol from April 25th to May 7th, 2022 on the scoped contracts and discovered 4 findings. Fortunately, no critical issues were found. We applaud Maverick Protocol for their attention to detail and diligence in security in the development of Maverick.

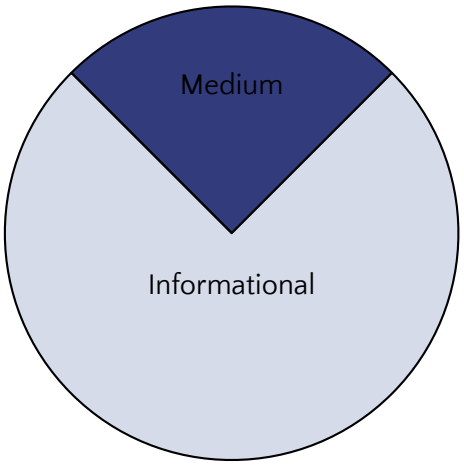
Of the 4 findings, 1 was of low severity, and the remaining findings were informational in nature. Additionally, Zellic recorded its notes and observations from the audit for Maverick Protocol's benefit at the end of the document.

Zellic thoroughly reviewed Maverick codebase to find discrepancies between the code implementation and the whitepaper's description of math, or any technical issues outlined in the Methodology section of this document. Specifically, taking into account Maverick's threat model, we focused heavily on issues that would break core invariants like manipulating swap prices or denial of service by desynchronizing base/quote accounting.

Our general overview of the code is that it was very well-organized and structured. The code coverage is high and tests are included for the majority of the functions. The code's documentation was poor, although all processes were accurately described in the whitepaper. The code was usually easy to comprehend with a pre-existing understanding of Maverick protocol.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	0
Informational	3



3 Coverage

We carefully reviewed Maverick for attack scenarios archetypal to projects like it.

3.1 Major Attack Scenarios

Attack Type	Description	Analysis	Likelihood
Re-entrancy	The impact of re-entrancy cannot be understated as it is one of the principal causes of exploits in the DeFi space.	Fortunately, there were no instances of reen-trancy with security impact in the AMM pools.	None
Flash Loan Attacks	A flash loan attack could be used to sandwich a trader and steal money from victims without risk to the attacker.	The pool has an estimator system specifically de-signed so that traders and market participants get an approximation of mar-ket costs accounting for potential predatory mar-ket forces.	None
Liquidity Ma-nipulation	Liquidity manipula-tion through oracles and other effects that can cause wild market irregularities—causing staked users and liq-uidity providers to lose capital.	After testing with our fuzzing suite and a de-tailed manual overview, we found no practical liquidity manipulation that pose a security risk.	Low

Centralization Risk	Centralization risk has been one of the most expensive issues in the DeFi space. Security incidents involving even a single private key have been fatal.	In the Maverick AMM pool, the centralization risks are limited to a factory owner that can set the pool into emergency mode, where the capital in the pool can still be withdrawn. There are no other centralization risks as there are no other owner-only functions.	None
Freezing of Funds	Fund lock-ups have affected many contracts. When funds are frozen, they are permanently and irrecoverably lost. This is usually caused by faulty business logic.	We did not discover any problems that would lead to a lock-up of funds.	None
Market Manipulation & Griefing	A common issue with sophisticated AMMs is the potential to execute orders for other users at unfair prices. This could be due to sandwiching trades or other avenues of manipulating internal bookkeeping.	Fortunately, we were not able to find such bugs in the Maverick AMM, and we noted that Maverick implmented functions such as spread fees to stop predatory arbitrage interactions when large capital movements happen, and so on.	None

3.2 Code Maturity Assessment

Category	Description	Detection
Error Handling	Maverick's AMM mechanisms contain several stringent checks to validate input and filter invalid requests. This is important it limits the impact of business logic errors.	Satisfactory
Roles and Access Controls	The AMM pool has very little to no centralization. There is no need for roles or any access controls.	Optimal
Customization & Composability	The Maverick pools expose many settings to fine-tune and properly setup one's AMM pool. It also demonstrates excellent compasibility, as seen in the `ZeroExRouter.sol` extension	Optimal
Project Structure	The project is well structured. It is easy to upgrade individual contracts.	Optimal
Events & Logging	We found the emission of events sufficient to thoroughly follow and understand the order of events. These events are useful to help diagnose the root cause of potential peculiarities. They also provide information for statistics for anyone wanting to study the AMM.	Optimal
Documentation	There is a lot of code implementing math described by the whitepaper. Comments linking areas in the white paper to actions in the code would improve comprehensibility.	Needs Improvement
Test Suite Evaluation	The test suite is certainly large and covers a variety of situations and cases. We applaud the initiative of Maverick for preemptively setting up security tools such as slither, eth-security-toolbox and even going as far as setting up fuzzing tests for echidna.	Optimal

Arithmetic and Precision	Maverick uses the PRBMath library which uses an 18-decimal standard representation for its operations. The ordering of operations was found to be sufficient, although incorrect in certain places. There may be small rounding errors in certain cases as documented in the findings section.	Medium
Gas Optimization	We consider the level of gas optimization in Maverick's AMM pool to be satisfactory. There were no boundless loops or extreme accessing of storage variables. They also employed other techniques such as using a cache for inputs.	Satisfactory

4 Code Overview

4.1 Contract `Pool` in `Pool.sol`

Function Name	Visibility	State Access	Modifiers
`initialize`	External	Read / Write	`initializer`
`getTwapParameters`	External	Read-only	`override`
`claimProtocolFees`	External	Read / Write	`override`
`getState`	External	Read-only	`override`
`addLiquidity`	External	Read / Write	`override`
`removeLiquidity`	External	Read / Write	`override`
`swap`	External	Read / Write	`override`
`_executeSwap`	Internal	Read / Write	–
`_computeBaseForQuoteBin`	Internal	–	–
`_computeQuoteForBaseBin`	Internal	–	–
`_calculateTwau`	Internal	Read-only	–
`_getAssetsInActiveBin`	Internal	Read-only	–
`_getAssetsInActiveBinLUT`	Internal	–	–
`_getAssetsInActiveBinCustom`	Internal	Read-only	–
`_getActiveInd`	Internal	Read-only	–
`_getActiveIndLUT`	Internal	–	–
`_swapBaseForQuote`	Internal	Read / Write	–
`_boundaryCheck`	Internal	–	–
`_swapQuoteForBase`	Internal	Read / Write	–
`_claimProtocolFees`	Internal	Read / Write	–
`_quoteTokenBalance`	Internal	Read-only	–
`_baseTokenBalance`	Internal	Read-only	–

`_addFeeToEscrow`	Internal	Read / Write	–
`_removeLiquidity`	Internal	Read / Write	–
`_setLiquidity`	Internal	Read / Write	–
`_spreadFee`	Internal	Read-only	–
`_updateU`	Internal	Read / Write	–
`triggerEmergencyMode`	External	Read / Write	–
`emergencyClaimProtocolFees`	External	Read / Write	–
`emergencyRemoveLiquidity`	External	Read / Write	–

4.2 Contract `Router` in `Router.sol`

Function Name	Visibility	State Access	Modifiers
`addLiquidity`	External	Read / Write	`valid(params.deadline) override`
`removeLiquidity`	External	Read / Write	`valid(params.deadline) override`
`swap`	External	Read / Write	`valid(params.deadline) override`
`swapEthForToken`	External	Read / Write	`valid(params.deadline) payable override`
`swapTokenForEth`	External	Read / Write	`valid(params.deadline) override`
`estimateSwap`	External	Read-only	`override`

4.3 Contract `Estimator` in `Estimator.sol`

Function Name	Visibility	State Access	Modifiers
`swap`	Public	Read-only	–
`getEstimationParameters`	External	Read-only	–
`price`	Public	Read-only	–
`_estimationParameters`	Internal	Read-only	–
`_estimateSwap`	Internal	Read-only	–

`_boundaryCheck`	Internal	–	–
`_estimateSwapBaseForQuote`	Internal	Read-only	–
`_estimateSwapQuoteForBase`	Internal	Read-only	–
`_computeBaseForQuoteBin`	Internal	–	–
`_computeQuoteForBaseBin`	Internal	–	–
`_getActiveInd`	Internal	–	–
`_getActiveIndLUT`	Internal	–	–
`_calculateTwau`	Internal	Read-only	–
`_getAssetsInActiveBinCustom`	Internal	–	–
`_getAssetsInActiveBin`	Internal	–	–
`_getAssetsInActiveBinLUT`	Internal	–	–
`_addFeeToEscrow`	Internal	Read-only	–
`_spreadFee`	Internal	Read-only	–

4.4 Contract `ZeroExRouter` in `ZeroExRouter.sol`

Function Name	Visibility	State Access	Modifiers
`sellTokenForToken`	External	Read / Write	`override`
`sellEthForToken`	External	Read / Write	`payable override`
`sellTokenForEth`	External	Read / Write	`override`
`getSellQuote`	External	Read-only	`override`

4.5 Contract `Factory` in `Factory.sol`

Function Name	Visibility	State Access	Modifiers
`setClaimer`	Public	Read / Write	–
`setOwner`	Public	Read / Write	–

`create`	External	Read / Write	–
`lookup`	External	Read-only	–
`generateKey`	Public	–	–
`_initializePool`	Internal	Read / Write	–

5 Detailed Methodology

We checked the code for bugs using the steps listed in the section 1.2, but this section will more thoroughly document the ideas we explored and invariants we tried to break.

5.1 Manual Static Analysis

Reentrancy is not a major concern because the AMM does not implement flash loans. The calls to the asset pools allowed reentrancy, but it does not pose a security risk. This is because if a malicious actor controls the token, they can already `mint` or `burn`. Furthermore, the code uses non-reentrancy locks on many pool functions.

We also analyzed the code for potential overflows and cast truncation. The code uses a safe cast library in some places that revert when an integer cast truncates data. This attack scenario may be feasible, but for any ERC-20 token with 18 decimals of precision (the default decimal amount), it would take around 20 orders of magnitude to overflow an `int128`. We believe this attack scenario is unlikely to occur in real-world conditions.

We checked the code for bugs that can be triggered by passing valid inputs which nevertheless corrupt the AMM's internal bookkeeping through rounding errors. We noted our findings in the Detailed Findings section.

Finally, we verified the robustness of the pool mechanics. That is, we ensured the implementation follows the specification provided by the whitepaper such that no oddities occur when calculating the swap invariant, amount of base, or other variables.

There is currently no conclusive way to statically verify the implementation, other than to thoroughly read the code and compare the mechanics of the implementation with the specification.

5.2 Automated Static Analysis

For the sake of comprehensiveness, we employed industry-standard static analysis tools, like Slither. Fortunately, our automated analyses did not uncover any notable issues. We would also like to note that the Nexus Labs implemented a Slither test in `package.json`.

5.3 Symbolic Execution and SMT Checking

We attempted to run the Mythril contract analyzer on the contracts. However, the contracts are very complex, and the analyzer never completed due to the classic state explosion problem faced by symbolic execution techniques. There is a large number of operations and many loops, resulting in an exponentially large number of states to explore. In the industry, this is an active research question currently undergoing extensive study.

Running the estimator and pool through the Solidity compiler's SMTChecker to formally verify the correctness of their relationship was also not feasible due to similar issues. As of the time of writing, the SMTChecker is not able to unroll/inline loops in the contracts, rendering it practically unusable for this engagement.

However, as we discuss in the next section, we did apply fuzzing tests to strengthen the contracts' level of assurance.

5.4 Fuzzer Dynamic Analysis

As part of our dynamic analyses, we implemented several end-to-end fuzzing tests. These Echidna tests are simple yet effective.

In our tests, to maximize coverage, we added multiple effects functions to alter the pool state in every way possible:

- Add a random but valid amount of base and quote liquidity to the pool.
- Remove a random but valid amount of liquidity to the pool.
- Send a random amount of assets to the pool.

Additionally, we ran this test against a variety of pool parameters and token decimals to ensure correctness regardless of pool configuration.

As the fuzzer generates random inputs in the range of the size of the data type, to make the quantities of input more reasonable, we constrained the inputs using the modulo operator. In doing so, we more accurately represented real transactions or feasible token amounts that may be used in attacks.

The property function `property_swap_test` checks that a call to `swap` does not revert. Because we constrained the inputs, any reversion is indicative of a coding flaw.

Whenever a reversion was encountered, the seed was recorded and the testcase was minimized. Then, we attempted to reproduce the reversion using hardhat tests in the repository.

Finally, we thoroughly investigated the reversion to find the root cause. If we discov-

ered the root cause was a fuzzer test misconfiguration, we tweaked the fuzzer. If the fault was a real issue, we worked together with Maverick Protocol to understand its' impact and exploitability.

In addition to the property function detecting reversions, we checked to ensure the quantity of swap output is equal to the estimate provided by `Estimator.sol`. Because the pool is running in a controlled environment and slippage is not possible as our tests run synchronously, any discrepancy between prediction and actual swap output is indicative of a bug.

Fortunately, we did not find a way to cause a price discrepancy. We did, however, discover several peculiarities using our fuzzing tests that we made note of in the Detailed Findings and Discussion sections.

Running Zellic's Fuzzing Tests

We provided Nexus Labs with `FuzzingContract.sol`, a flattened version of all of the relevant contracts, and `echidna.config.yaml`, which is the configuration file needed to run the fuzzer.

We slightly modified the contracts from those in the repository to combat bugs in the fuzzer, echidna. However, to ensure the results of our tests are valid, no logic was modified.

The following command runs the fuzzer:

```
echidna FuzzingContract.sol --contract SwapTest --config ./echidna.config
    .yaml
```

6 Detailed Findings

6.1 Rounding Issues Caused by Loss of Precision

- **Target**: Pool
- **Category**: Coding Mistakes
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Medium

Description

In some places, the contract performs division before multiplication on fixed-point 18-decimal numbers, increasing the significance of rounding errors. In other places, the order of operations cannot be changed, but there is still a precision loss.

For example, the following code causes the `q0` to become smaller before causing it to be larger. As the type is fixed-point at 18 decimals, it is safer to have the value become larger before dividing to avoid rounding earlier in the operations:

```
activeInfo.L = activeInfo.q0.div(su).mul(params.lambdas0);
```

We observed a loss of precision leading to strange behavior in a few cases. For example, the asset fee and protocol fee in `_addFeeToEscrow` can be bypassed using a `baseIn` low enough to cause `assetFee` to be 0. Assuming `txFee` multiplier is `0.01e18`, the `assetFee` (and `protcolFee`) will be 0 for any `baseIn` less than or equal to 49.

Impact

It is possible to make swaps without paying fees. Realistically, gas prices make attacks abusing this rounding bug impractical. However, all math operations on the fixed-point numbers may incur small amounts of precision loss—meaning there may be other side effects of precision loss.

Side effects should only be observable for extreme values (e.g. extremely small numerators, or extremely large denominators).

Recommendation

In general, we recommend ordering operations so that numbers may increase before decreasing (i.e. multiply before dividing) where possible.

For the fee bypass example, we recommend requiring a minimum `txFee` and `protoco`

lFee of 1 to ensure that both fees are always collected:

```
if (isQuote) {
    if (spreadFee < 0) {
        txFee = (int256(currentState.fee) * 1e14) - spreadFee;
    } else {
        txFee = int256(currentState.fee) * 1e14;
    }
    txFee = BasicMath.min(0.999e18, txFee);
    assetFee = amountIn.mul(txFee);
    int256 protocolFee = assetFee.mul(
        int256(currentState.protocolFeeRatio) * 1e16
    );
    require(txFee > 0 && protocolFee > 0);
    quoteBalance += int128(assetFee - protocolFee);
    cache.quoteBalance = quoteBalance;
    quoteProtocolFeeEscrow += int128(protocolFee);
} else {
    if (spreadFee > 0) {
        txFee = (int256(currentState.fee) * 1e14) + spreadFee;
    } else {
        txFee = int256(currentState.fee) * 1e14;
    }
    txFee = BasicMath.min(0.999e18, txFee);
    assetFee = amountIn.mul(txFee);
    int256 protocolFee = assetFee.mul(
        int256(currentState.protocolFeeRatio) * 1e16
    );
    require(txFee > 0 && protocolFee > 0);
    baseBalance += int128(assetFee - protocolFee);

    cache.baseBalance = baseBalance;
    baseProtocolFeeEscrow += int128(protocolFee);
}
return amountIn - assetFee;
```

Remediation

This issue was acknowledged by \CustomerName{} and a fix is pending.

6.2 Interfaces Inaccurately Describe Contracts

- **Target**: Project-Wide
- **Category**: Code Maturity
- **Likelihood**: Informational
- **Severity**: Informational
- **Impact**: Informational

Description

Certain interfaces do not accurately reflect contracts and their public APIs.

For example, `Pool.sol` contains emergency functions that are essential if a security incident were to happen, but the `Pool` interface does not reference the functions.

Impact

Interactions with smart contracts may be more difficult; it is a composability issue for future developers who want to build upon or understand the codebase.

Recommendation

We recommend adding all exposed/public APIs to interfaces such that they accurately reflect the underlying code.

Remediation

This issue was acknowledged by \CustomerName{} and a fix is pending.

6.3 Lack of Documentation

- **Target**: Pool.sol
- **Category**: Code Maturity
- **Likelihood**: Informational
- **Severity**: Informational
- **Impact**: Informational

Description

While the whitepaper accurately describes the project and its mechanisms, there are certain areas in the code where a reaffirmation of the mechanisms would help with comprehensability. For example:

```
if (binInd == 0) {
    info.spHigh = cache.u.div(int256(currentState.w) * 1e14).sqrt();
    info.spLow = cache
        .u
        .div(int256(currentState.w) * 1e14)
        .div(int256(currentState.k) * 1e16)
        .sqrt();
} else if (binInd == 1) {
    info.spHigh = cache.u.mul(int256(currentState.w) * 1e14).sqrt();
    info.spLow = cache.u.div(int256(currentState.w) * 1e14).sqrt();
} else {
    info.spHigh = cache
        .u
        .mul(int256(currentState.w) * 1e14)
        .mul(int256(currentState.k) * 1e16)
        .sqrt();
    info.spLow = cache.u.mul(int256(currentState.w) * 1e14).sqrt();
}
```

A comment clarifying that variables `spLow` and `spHigh` are the bounds of bins would help readers more quickly understand the code.

Impact

Code maturity is very important in high-assurance projects. Undocumented code may result in developer confusion, potentially leading to future bugs should the code be modified later on.

In general, a lack of documentation impedes the auditors' and external developers' ability to read, understand, and extend the code. The problem is also carried over if

the code is ever forked or reused.

Recommendation

We recommend adding more comments to the code—especially comments that tie operations in code to locations in the whitepaper, and brief comments to reaffirm developers' understanding.

Remediation

This issue was acknowledged by \CustomerName{} and a fix is pending.

6.4 Lack of Pool Parameter Validation

- **Target**: Project-Wide
- **Category**: Code Maturity
- **Likelihood**: Informational
- **Severity**: Informational
- **Impact**: Informational

Description

The pool initializer does not validate the inputted parameters. In the fuzzing tests that Nexus Labs provided to us, for example, the `protocolFeeRatio` parameter was `1.1e2` (110%). The invalid parameter caused subtle accounting errors that we did not catch until modifying the tests to make them stricter.

Impact

An uninformed pool deployer may unintentionally configure invalid settings that cause the pool to be vulnerable or not function properly.

Recommendation

Add `require()` statements to ensure all pool parameters are valid.

Remediation

Nexus Labs noted that only their pool factory can deploy pools, and it is their responsibility to deploy with the correct settings.

7 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

We admire Nexus Labs for their good test coverage and use of the fuzzing suite echidna. Their automatic liquidity-concentrating AMM is certainly innovative in the DeFi space.

As described in the Detailed Methodology section, as part of our analysis of the smart contracts, we implemented an end-to-end fuzzing suite to verify that the estimator—which is used to approximate prices/trades for the pool—was in fact in sync with the pool. We also employed a series of random fuzzing tests to determine the robustness of the pool for extreme imbalances and extreme (but valid) values.

During our fuzzing, we found several edge cases that either rendered the pool unusable or threw off internal bookkeeping.

7.1 Dependence on `block.timestamp`

The spread fee—which relies on the configurable time value (default 3600 seconds)—is increasingly vulnerable to timestamp manipulation by miners (Reference) for smaller `T` values. This dependence may allow certain arbitrageurs to abuse the `block.timestamp` to become the winners when a large U-moving transaction occurs.

We believe that for some pools where the underlying assets are volatile, a low value is optimal for `T` so that the AMM is up to date with moving market prices. Those cases where the `T` values are low, the pool may be susceptible to `block.timestamp` manipulation.

For the default `T` value of 1 hour, we estimate that the spread fee is <1 percent manipulable by miners, based on the "15-second Rule". Thus, we believe the security risk stemming from this dependence is limited.

7.2 Invalid Pool Parameters in Maverick's Tests

We observed that, in the fuzzing tests provided by Maverick, there are invalid (i.e. out-of-bounds) values. For example, the `protocolFeeRatio` setting is configured as `1.1e2` which results in the occurence of negative numbers in certain areas:

```
int256 protocolFee = assetFee.mul(
    int256(currentState.protocolFeeRatio) * 1e16
);
baseBalance += int128(assetFee - protocolFee);
```

The above code—with the invalid `protocolFeeRatio`—takes 110% of the `assetFee` and subtracts it from `assetFee`, resulting in a negative number. This lowers the internal `quoteBalance/baseBalance`. In extreme cases when the internal balances of quote/base are small and the `amountIn` is large enough, the 10% negative offset of the protocolFeeRatio would lower the internal balances to negative ranges—later resulting in reverts.

7.3 Casting to `int128` May Truncate Numbers

When the incoming quantities are large enough, several places in the code may truncate bytes from the numbers resulting in overflow-like behavior; that is, some numbers may become negative when cast into `int128` leading to a revert if passed into `sqrt`. The solution to this issue is to always use safe casting functions such as those implemented in `libraries/Cast.sol`.

7.4 Loss of Precision in Decimal Scaling

Scaling of varying decimals resulted in the loss of precision in very small inputs. For example, when doing a small-enough swap with a 24-decimal token where the `quoteBalance/baseBalance` is scaled down to 18 decimals using standard PRB math, the `quoteBalance/baseBalance` remains the same resulting in a 0 value of `amountIn`:

```
function _quoteTokenBalance(State memory currentState)
    internal
    view
    returns (int256)
{
    return
        SafeTransfer
            .fromScale(
                IERC20(quote).balanceOf(address(this)),
                currentState.quoteDecimals
            )
            .toInt256();
}
```

```
amountIn = _quoteTokenBalance(currentState) - totalQuoteBalance;
```

An `amountIn` value of 0 causes a revert with the reason string `Pool: INSUFFICIENT_SWAP`.