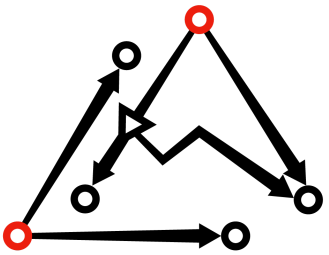

Least Authority

PRIVACY MATTERS

PiSwap Protocol Smart Contracts

Security Audit Report

PiSwap Protocol

Final Audit Report: 20 May 2022

Table of Contents

- Overview
  - Background
  - Project Dates
  - Review Team
- Coverage
  - Target Code and Revision
  - Supporting Documentation
  - Areas of Concern
- Findings
  - General Comments
    - System Design
    - Code Quality
    - Documentation
    - Scope
  - Specific Issues & Suggestions
  - Suggestions
    - Suggestion 1: Perform Multiplications Before Division
    - Suggestion 2: Define Functions Appropriately
    - Suggestion 3: Require Oracle Length in Function setOracleLength to be Consistent with Whitepaper
    - Suggestion 4: Implement Property-Based Testing
- About Least Authority
- Our Methodology

Overview

Background

PiSwap Protocol has requested that Least Authority perform a security audit of the PiSwap Protocol Smart Contracts.

Project Dates

- **April 6 - April 26**: Initial Code Review *(Completed)*
- **April 29**: Delivery of Initial Audit Report *(Completed)*
- **May 18-29**: Verification Review *(Completed)*
- **May 20**: Delivery of Final Audit Report *(Completed)*

Review Team

- Alicia Blackett, Security Researcher and Engineer
- Nicole Ernst, Security Researcher and Engineer
- Ahmad Jawid Jamiulahmadi, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the PiSwap Protocol smart contracts followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repository is considered in-scope for the review:

- PiSwap Protocol Smart Contracts:
  https://github.com/PiSwapProtocol/PiSwap-core

Specifically, we examined the Git revision for our initial review:

`e81aa5f7b6d968e6e15e4f7d26dbe230f54d1055`

For the verification, we examined the Git revision:

6e85f63b66aea5494a1064c95e76edaf064130fd

For the review, this repository was cloned for use during the audit and for reference in this report:

PiSwap Protocol Smart Contracts:

https://github.com/LeastAuthority/PiSwap-core

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- PiSwap Protocol Whitepaper: https://docs.google.com/document/d/1B69RdEEovy2JXgLP8avUy09hAtH2--_dTBaSLMzqoIs/edit

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Common and case-specific implementation errors;
- Adherence to the specification and best practices;
- Adversarial actions and other attacks on the smart contracts;
- Potential misuse and gaming of the smart contracts;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service (DoS) and other security exploits that would impact the intended use or disrupt the execution of the smart contracts;
- Vulnerabilities in the smart contract code;
- Protection against malicious attacks and other ways to exploit the smart contracts;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

PiSwap protocol proposes an approach to NFT trading whereby a market comprising 2 liquidity pools (bull and bear) is created, which allows users to take positions on the expected value of an underlying NFT. This mechanism aims to determine a spot price for NFTs that is reflected by the market's desire to hold bull or bear tokens for that asset. Once the protocol has sufficient levels of liquidity, functionality is enabled to allow the smart contract to buy and sell NFTs from users.

Our team examined the functions interacting with the ETH vaults to check for any logic errors that could lead to the funds being drained by a malicious actor and could not identify any vulnerabilities. We checked external smart contract calls for exploitable reentrancy opportunities and could not identify any. We also checked the mathematical operations performed in the smart contract overflow or underflow errors and did not find vulnerabilities.

System Design

Our team examined the PiSwap protocol system and found that security has been taken into consideration in the overall design of the system, as demonstrated by the appropriate implementation of the checks-effects interaction pattern and reentrancy guards, the utilization of extensively audited and well tested libraries, and comprehensive unit test coverage.

Code Quality

The PiSwap protocol codebase is very well organized and adheres to best coding practices. We identified an instance where the ordering of the arithmetic operations is not consistent with recommended best practice in Solidity. Specifically, multiplication operations must be performed before division operations

where possible, in order to avoid precision errors. We recommend adhering to best practice in the ordering of arithmetic operations (Suggestion 1).

Furthermore, functions in the codebase are not identified according to best practice. Functions must be defined according to their purpose within the system to improve readability and to remove any ambiguity regarding the intended capabilities of the function and how the function can be accessed. We recommend that functions be defined in adherence to recommended practice (Suggestion 2).

Tests

The PiSwap protocol smart contracts implement a test suite with almost 100% coverage. This is commendable and significantly enhances the ability to identify implementation errors. However, the project utilizes complex mathematical formulas, and while unit tests provide some indication that they work correctly, we recommend that property based testing be used to verify correct functionality (Suggestion 4).

Documentation

The project documentation provided for this review sufficiently described the overall system, each of the components, and how those components interact with each other. However, we identified an instance in the implementation that is not consistent with the documentation.

The PiSwap protocol implements an on-chain oracle to register NFT prices. The oracle is an array of price entries and the average of the price recorded over the last 60 blocks is taken to prevent an attacker from manipulating the average price. The whitepaper included in the project documentation states that the number of blocks can be increased, if found to be insufficient. However, a function in the `PiSwapRegistry` smart contract (`setOracleLength`) checks for a value far smaller than this. We recommend requiring the minimum value to be the same as in the white paper (Suggestion 3).

Code Comments

The PiSwap protocol smart contracts contain code comments, which describe the intended behavior of security critical functions and components that adhere to NatSpec guidelines for Solidity code comments.

Scope

The scope of this security audit contained all security critical components of the system. However, our team did not make an assessment of the protocol's ability to achieve its economic objectives as defined in the whitepaper or test any assumptions that the economic model is built upon. The financial primitives introduced by the project have not seen wide use on-chain before. While direct economic exploits of the economic assumptions of the protocol were out of scope for this audit, they did introduce challenges to reasoning about the logic of the code. We recommend that the PiSwap team perform an audit of the system's ability to achieve its economic objectives.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Suggestion 1: Perform Multiplications Before Division	Resolved

Suggestion 2: Define Functions Appropriately	Resolved
Suggestion 3: Require Oracle Length in Function setOracleLength to be Consistent with Whitepaper	Resolved
Suggestion 4: Implement Property-Based Testing	Unresolved

Suggestions

Suggestion 1: Perform Multiplications Before Division

Location

contracts/lib/market/PiSwapLibrary.sol#L136-L139

Synopsis

In the `lockedEth` function, there are multiple cases with division performed before multiplication. In some cases, this might result in the Solidity integer division truncating the quotient and causing loss of precision.

Mitigation

We recommend reordering the arithmetics to perform possible multiplications before divisions as follows:

```
int256 numerator = int256(
                ((tokenReserve ** 2) * ethReserve) / ONE
                + MAX_SUPPLY * (MAX_SUPPLY * STRETCH_FACTOR  * ethReserve *
tokenReserve / (ONE * ONE)).sqrt()
                - (MAX_SUPPLY * ethReserve * tokenReserve) / ONE
                - (MAX_SUPPLY * STRETCH_FACTOR * tokenReserve) / ONE
            );
```

Status

The PiSwap team responded that because the two parameters `ethReserve` and `tokenReserve` can receive large values, it is not possible to perform multiplications before division due to integer overflow.

Verification

Resolved.

Suggestion 2: Define Functions Appropriately

Location

Defined Internal:

contracts/lib/market/PiSwapLibrary.sol#L25

contracts/lib/registry/BeaconProxyOptimized.sol#L27

contracts/lib/registry/BeaconProxyOptimized.sol#L44

contracts/lib/registry/BeaconUpgradeable.sol#L32

contracts/lib/registry/OwnedUpgradeable.sol#L22

Defined Public:

contracts/PiSwapMarket.sol#L56

contracts/PiSwapMarket.sol#L307

Synopsis

Several of the above referenced functions are defined as `internal`. However, they are not being used in any of the derived smart contracts. Others are defined as `public` even though they should actually be defined as `external`. It is considered best practice to define function access modifiers based on where the function is going to be used. This helps improve the readability of the code and makes it easier to identify incorrect assumptions about who can call the function.

Mitigation

We recommend defining the referenced internal and public functions, in the function definition, as private and external respectively by replacing the `internal` keyword with `private` and the `public` keyword with `external`.

Status

The PiSwap team has implemented the mitigation as suggested.

Verification

Resolved.

Suggestion 3: Require Oracle Length in Function setOracleLength to be Consistent with Whitepaper

Location

contracts/PiSwapRegistry.sol#L148

Synopsis

The PiSwap protocol whitepaper states that a block length of 60 is necessary to sufficiently reduce the risk of price manipulation. However, in the function `setOracleLength`, the oracle block length can be set to a value greater or equal to 5.

Mitigation

We recommend that the implementation of a `require` statement in `setOracleLength` check for a value greater or equal to the value stated in the whitepaper.

Status

The PiSwap team stated that the oracle length is set lower for testing purposes but will be increased in accordance with the documentation for deployment to mainnet.

Verification

Resolved.

Suggestion 4: Implement Property-Based Testing

Location

contracts/lib/market/PiSwapLibrary.sol#L88-L91

contracts/lib/market/PiSwapLibrary.sol#L136-L139

Synopsis

The PiSwap protocol smart contracts code contains calculations in multiple places that might lead to unexpected behavior, in certain edge cases, as a result of loss of precision. While the codebase has sufficient unit test coverage, additional test coverage is needed to catch potential unintended behavior.

Mitigation

We recommend using property-based testing, in addition to unit testing, to increase the probability of identifying such errors.

Status

The PiSwap team plans to increase test coverage. However, at the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.