BLADE

SMART CONTRACTS REVIEW

zokyo

May 27th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING BLADESWAP

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 0 High issues: 0 points deducted
- 0 Medium issuei: 0 points deducted
- 4 Low issues: 4 acknowledged = - 8 points deducted
- 10 Informational issues: 10 acknowledged = 0 points deducted

Thus, 100 - 8 = 92

TECHNICAL SUMMARY

This document outlines the overall security of the Bladeswap smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Bladeswap smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Bladeswap team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

Auditing Strategy and Techniques Applied 5

Executive Summary 7

Structure and Organization of the Document 8

Complete Analysis 9

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Bladeswap repository:
Repo: https://github.com/Bladeswap/contracts/blob/master/contracts/pools

Last commit - e1ed74c29b4f49d8af1e12e25b5934c434977faf

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- StableSwapPool.sol
- StableSwapPoolFactory.sol
- XYKPool.sol
- XYKPoolFactory.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Bladeswap smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team has performed a security audit of the provided codebase. The contracts submitted for auditing are well-crafted and organized. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

The StableSwapPool.sol contract allows for the creation of a pool with two tokens. The velocore__execute function permits a smart contract, onlyVault, which the users will use to execute functions within the StableSwapPool contract. In addition to performing several necessary calculations for the correct operation of the code, the main functionality of the velocore__execute function is to simulate swaps. It allows users to swap tokens for token0 or token1 and also for LP tokens.

The XYKPool.sol contract also allows for the creation of a pool with two tokens and swapping between the tokens including LP token based on the formula of xy=k. The velocore__execute function permits a smart contract, onlyVault, which the users will use to execute functions within the XYKPool contract.

As well Contracts: StableSwapPoolFactory.sol and XYKPoolFactory.sol are factory contracts designed to facilitate the creation and management of StableSwapPool and XYKPool contracts, respectively. These factory contracts streamline the deployment of new pool instances by generating new contracts as needed and maintaining a registry of all created pools.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed.Acknowledged means that the issue was sent to the Bladeswap team and the Bladeswap team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	`fee1e9` can be set to a higher value than expected	Low	Acknowledged
2	Possible Denial of Service due to `lastWithdrawTimestamp != block.timestamp` when calling Bladeswap__execute()	Low	Acknowledged
3	Missing sanity checks for important variables	Low	Acknowledged
4	`lastWithdrawTimestamp` used instead of `lastTradeTimestamp`	Low	Acknowledged
5	Variable Shadowing Issue in getPools Function	Informational	Acknowledged
6	Inefficient Read from storage slot in setFee function	Informational	Acknowledged
7	Centralization Risks	Informational	Acknowledged
8	Wrong amount in event emission	Informational	Acknowledged
9	Unused parameter	Informational	Acknowledged
10	Use custom errors instead of `require`	Informational	Acknowledged
11	Incomplete Deployment Scripts and Compilation Issues Around Unit Tests	Informational	Acknowledged
12	Reading a storage variable in the event	Informational	Acknowledged
13	Setting unused local variables	Informational	Acknowledged
14	Missing Natspec	Informational	Acknowledged

`fee1e9` can be set to a higher value than expected

The function `setFee()` in the `StableSwapPool.sol` and `XYKPool.sol` contracts checks the new fee to ensure that it is equal or lower than `0.1e9`:

```
function setFee(uint256 fee1e9_, uint256 decayRate_) external authenticate {
        require(fee1e9 <= 0.1e9);
        fee1e9 = uint32(fee1e9_);
        emit FeeChanged(fee1e9 * uint256(1e8));
    }
```

It can be seen that the `require` checks the value of `fee1e9` instead of the input parameter `fee1e9_`.

Recommendation:

Fix the `require` statement:

```
require(fee1e9_ <= 0.1e9);
```

Client comment: Acknowledged. This is a typo. Fee1e9 is managed by authorized entities, so the possibility of DOS is minimal.

Possible Denial of Service due to `lastWithdrawTimestamp != block.timestamp` when calling Bladeswap__execute()

When a call to the `Bladeswap__execute()` function in the `StableSwapPool.sol` and `XYKPool.sol` contracts is executed there is a `require` that checks if `lastWithdrawTimestamp != block.timestamp`:

```
if (lastWithdrawTimestamp != block.timestamp) {
            feeMultiplier = 1e9;
}
```

If lastWithdrawTimestamp is not equal to block.timestamp then feeMultiplier is set to 1e9.

Then, the function execution reaches a point where a call to `_exchange_for_t1()`, `_exchange_for_t0()` or `_exchange_for_lp()` is executed. These 3 functions receives an input (the last one) as `fe1e9 * feeMultiplier`.

The mentioned functions later call `_exchange()`:

```
function _exchange(
      int256 a_0,
      int256 a_1,
      int256 a_k,
      int256 b_1,
      int256 d_k,
      int256 fee
  ) internal view returns (int256) {
      int256 b_k = a_k - d_k;
      require(b_k >= 2);

      if (a_k <= b_k) {
          b_1 -= (SignedMath.max(((a_k * b_1) / b_k) - a_1, 0) * fee) / 1e18;
      } else if (a_k > b_k) {
          b_1 -= (SignedMath.max(b_1 - ((b_k * a_1) / a_k), 0) * fee) / 1e18;
      }

      int256 b_0 = _y(b_k, b_1);

      if (a_k <= b_k) {
          b_0 +=
              (SignedMath.max(((a_k * b_0) / b_k) - a_0, 0) * fee) /
              (1e18 - fee);
      } else if (a_k > b_k) {
          b_0 +=
              (SignedMath.max(b_0 - ((b_k * a_0) / a_k), 0) * fee) /
              (1e18 - fee);
      }

      return b_0 - a_0 + 1;
  }
```

If `fe1e9` is set to `1e9`, which can not be done by calling the `setFee()` function but directly in the constructor, the `fee` passed as last parameter in `_exectute`, will be the result of 1e9*1e9, which is equal to 1e18. This leads to a scenario where the denominator for the computation of b_0 will be equal to 0 and a division by 0 will revert causing a denial of service.

Recommendation:
Implement checks that `int256(uint256(fee1e9 * feeMultiplier))` does not exceed 1e17.

Client comment: `fee1e9` is managed by authorized entities, so the possibility of DOS attack is not significant.

LOW-3 | ACKNOWLEDGED

Missing sanity checks for important variables

There are some important variables in the `StableSwapPool.sol` and `XYKPool.sol` contract that are not checked before being set in the `constructor()` function and also in the functions used for changing their values.

- setDecay():

```
function setDecay(uint256 decayRate_) external authenticate {
        decayRate = uint32(decayRate_);
        emit DecayChanged(decayRate);
    }
```

- setFee():

```
function setFee(uint256 fee1e9_, uint256 decayRate_) external authenticate {
        require(fee1e9 <= 0.1e9);
        fee1e9 = uint32(fee1e9_);
        emit FeeChanged(fee1e9 * uint256(1e8));
    }
```

Recommendation:
Add checks to ensure that the values set are between the desired thresholds.

`lastWithdrawTimestamp` used instead of `lastTradeTimestamp`.

The `Bladeswap__execute()` function in `StableSwapPool.sol` and `XYKPool.sol` implements an 'if statement' which checks if `lastWithdrawTimestamp` is equal to `block.timestamp`.

```
if (lastWithdrawTimestamp != block.timestamp) {
            feeMultiplier = 1e9;
 }
```

`lastWithdrawTimestamp` is never initialized so it will always be `0` and this check will never be met. It could have been confused with `lastTradeTimestamp`.

Recommendation:
Consider using `lastTradeTimestamp` instead of `lastWithdrawTimestamp` is it the desired behavior.

Client comment: lastWithdrawTimestamp is the desired behavior, and we acknowledge it is not being set. This however does not cause a serious problem.

INFORMATIONAL-1 | ACKNOWLEDGED

Variable Shadowing Issue in getPools Function

Location: XYKPoolFactory.sol/StableSwapPoolFactory.sol

The return variable `pools` in the `getPools` function shadows the state variable `pools`, which can lead to confusion and potential errors in the code.

Recommendation:
It is recommended to rename the return variable pools to a different name to avoid shadowing the state variable. This will improve code readability and reduce the risk of unintended errors caused by variable shadowing.

INFORMATIONAL-2 | ACKNOWLEDGED

Inefficient Read from storage slot in `setFee` function

Location: XYKPoolFactory.sol/StableSwapPoolFactory.sol

The `setFee` function in XYKPoolFactory.sol/StableSwapPoolFactory.sol reads the value of `fee1e9` from storage twice, which can be optimized for efficiency by reading the value of the calldata `fee1e9_` instead of `fee1e9` from the storage slot.

Recommendation:
Modify the `setFee` function to directly use the value of `fee1e9_` passed as a function parameter instead of assigning it to the storage variable `fee1e9` before using it. This will eliminate the need to read the value from storage twice and improve gas efficiency.

Client comment: This function is called very infrequently, by authorized entities only, therefore gas cost is not a concern here. Also, solc would probably be able to optimize them

INFORMATIONAL-3 | ACKNOWLEDGED

Centralization Risks

System critical state can be modified (setFee and deploy pools in factory contracts) by an “authenticated” address . It should be made sure that these authenticated addresses are multisigs with a timelock to ensure that privileges are not centralised to one address.

Recommendation:
Make sure authenticated addresses are multisigs with a timelock.

Client comment: Acknowledged and we’ve set timelock contract already.

INFORMATIONAL-4 | ACKNOWLEDGED

Wrong amount in event emission

The function `setFee` in the `StableSwapPool.sol` and `XYKPool.sol` contracts emits a `FeeChanged` event every time a new fee is set. This event emits the new amount but is wrongly set.

```
emit FeeChanged(fee1e9 * uint256(1e8));
```

The event emits `fee1e9 * uint256(1e)` as the new fee amount instead of emitting `fee1e9`.

The same event is emitted in the constructor.

Recommendation:
Change the event amount to:

```
emit FeeChanged(fee1e9);
```

Client comment: Acknowledged. This is a typo. It should be fee1e9 * 1e9 instead. We’ll calculate accordingly when we need to index it in off-chain.

INFORMATIONAL-5 | ACKNOWLEDGED

Unused parameter

The function `setFee` in the `StableSwapPool.sol` and `XYKPool.sol` contracts takes `decayRate_` as parameter but it is not needed as it is not used.

```
function setFee(uint256 fee1e9_, uint256 decayRate_) external authenticate {
        require(fee1e9 <= 0.1e9);
        fee1e9 = uint32(fee1e9_);
        emit FeeChanged(fee1e9 * uint256(1e8));
    }
```

Recommendation:
It's recommended that unused parameters are left unnamed to be more compatible with interfaces. This can also assist in improving readability for example:

```
function setFee(uint256 fee1e9_, uint256) external authenticate {...}
```

Client comment: This is intentionally added to make it compatible with some interfaces

INFORMATIONAL-6 | ACKNOWLEDGED

Use custom errors instead of `require`

In Solidity v0.8.4, a significant improvement was implemented with the incorporation of custom errors. These errors offer a more efficient method, in terms of gas usage, for communicating failure reasons within your smart contracts compared to using revert strings. This advancement helps in decreasing both deployment and runtime expenses.

StableSwapPool.sol implements a require statement that can be optimized by using custom errors.

Recommendation:
Consider replacing require statements in favor of custom errors for the purposes of gas optimizations.

INFORMATIONAL-7 | ACKNOWLEDGED

Incomplete Deployment Scripts and Compilation Issues Around Unit Tests

To ensure a smooth deployment, it is essential that all components are thoroughly completed and tested beforehand, including deployment scripts. Failure to do so may result in significant issues during deployment. Additionally, it should be noted that the tests did not compile, indicating potential underlying problems that need to be addressed before proceeding.

INFORMATIONAL-8 | ACKNOWLEDGED

Reading a storage variable in the event

In the setDecay() and setFee() function, DecayChanged() and FeeChanged() events read storage variables even if local variables exist for the same value.

Recommendation:
Read decayRate_ and fee1e9_ instead of decayRate and fee1e9 in the events.

INFORMATIONAL-9 | ACKNOWLEDGED

Setting unused local variables

In the _excessInvariant() function, a_0 and a_1 are not set but not used at all.

Recommendation:
Remove the line setting a_0 and a_1.

Missing Natspec

The contract exhibits a lack of documentation, hindering its readability, maintainability, and auditability.

Recommendation:
Consider adding Natspec to the contracts.

	StableSwapPool.sol StableSwapPoolFactory.sol XYKPool.sol XYKPoolFactory.sol
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Bladeswap team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Bladeswap team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo