SMART CONTRACTS REVIEW

zokyo

March 22nd 2024 | v. 1.0

Security Audit Score

PASS

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

ZOKYO AUDIT SCORING WOMBAT EXCHANGE

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.

HYPOTHETICAL 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
- 1 Medium issue: 1 resolved = 0 points deducted
- 2 Low issues: 2 acknowledged = - 4 points deducted
- 10 Informational issues: 9 resolved and 1acknowledged = 0 points deducted

Thus, 100 - 4 = 96

TECHNICAL SUMMARY

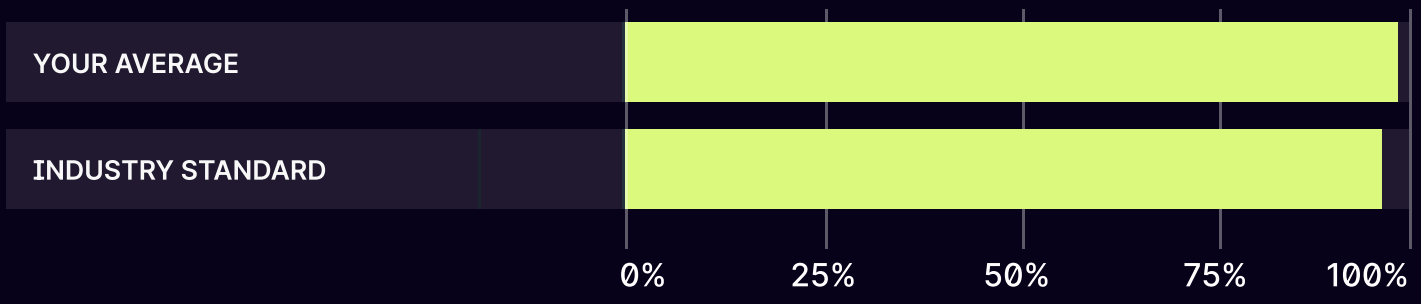
This document outlines the overall security of the Wombat Exchange smart contract/s evaluated by the Zokyo Security team.

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

Contract Status

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

Testable Code



99% of the code is testable, which is above the industry standard of 95%.

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 Wombat Exchange 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

The source code of the smart contract was taken from the Wombat Exchange repository:
Repo: https://github.com/wombat-exchange/wombat/blob/develop/contracts/wombat-core
Last commit: e51a7987ff5f64aea0f616c75c8f700c552ac56a

Fixes:
- https://github.com/wombat-exchange/wombat/pull/1176
- https://github.com/wombat-exchange/wombat/pull/1177

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

- VolatilePool.sol
- DynamicFeeHelper.sol
- RepegHelper.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 Wombat Exchange 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. Part of this work includes writing a test suite using the Foundry and Hardhat testing framework. 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.	03	Testing contract/s logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contract/s by industry leaders.	04	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.

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 Wombat Exchange team and the Wombat Exchange 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	The latest `Value` is not saved in the Write() method	Medium	Resolved
2	Method toLogScale should be internal	Low	Acknowledged
3	Method _getHaircutRate should check if the price anchor is set or not	Low	Acknowledged
4	Variable lastSwapTimestamp getting updated inside the loop	Informational	Resolved
5	Unnecessary condition check	Informational	Resolved
6	Use existing values to save gas	Informational	Resolved
7	Unsafe cast	Informational	Resolved
8	Unnecessary cast	Informational	Resolved
9	Pending To-do(s)	Informational	Resolved
10	Unused constants	Informational	Resolved
11	Typo/Wrong comment	Informational	Resolved
12	Unused imports	Informational	Resolved
13	Use specific Solidity compiler version	Informational	Acknowledged

The latest `Value` is not saved in the Write() method

In Library DynamicFeeHelper, the method write(...) is called to write data after each swap. Since the timepoint for a block can be written only once, this method has the following condition for early return:

```
PointHistory memory lastPoint = pointHistories[lastIndex];
        if (lastPoint.pointTimestamp == blockTimestamp) {
            // Early return if we've already written a timepoint this block
            lastPoint.value = value;
            return lastIndex;
        }
```

Here, lastPoint.value is being updated but lastPoint is of type `memory` so the value will not be persisted in the storage. Hence latest value will not be saved.

Since this `value` is the first step of the data flow and the swap method can be called multiple times in a block, this not being persistent can lead to unexpected results in further calculation.

Recommendation:
Update the `value` for lastIndex properly so it is persistent.

LOW-1 | ACKNOWLEDGED

Method toLogScale should be internal

In Library DynamicFeeHelper, the method toLogScale(...) has the following logic:

```
// bound the result
        if (result > type(int32).max) {
            return type(int32).max;
        } else if (result < type(int32).min) {
            return type(int32).min;
        } else {
            return int32(result);
        }
```

Here the result is greater than type(int32).max, then returned value is type(int32).max which incorrect if toLofScale() method is used directly. Any external contract relying on this value can consider the upper and lower bounds as the correct result and process accordingly which will lead to wrong calculations.

Recommendation:
Update to make this method internal and to be used only through safetoLogScale() method.

LOW-2 | ACKNOWLEDGED

Method _getHaircutRate should check if the price anchor is set or not

In Contract VolatilePool.sol, the method _getHaircutrate(...) calculates the volatility of fromAsset and toAsset if they are not price anchor.

Since priceAnchor is not set in the initialize() method, there is a possibility of it not being assigned and this will lead to volatility calculation even for priceAnchor asset if it is either fromAsset or toAsset which will lead to wrong haircut rate.

Recommendation:
Ensure that the price anchor is set before getHaircutrate can be calculated.

INFORMATIONAL-1 | RESOLVED

Variable lastSwapTimestamp getting updated inside the loop

In Contract VolatilePool.sol, the method _postSwapHook() has the following code:

```
for (uint256 i; i < assetCount; ++i) {
            . . .
            marketPricesLast[asset] = marketPrice;
            lastSwapTimestamp = block.timestamp;
        }
```

Since lastSwapTimestamp needs to be updated once in the _postSwapHook method, there is no need to update it in the loop.

Recommendation:
Update the variable lastSwapTimestamp outside the loop.

INFORMATIONAL-2 | RESOLVED

Unnecessary condition check

In Library RepegHelper, the method _getProposedPriceScales has the following check:

```
        require(normalizedAdjustmentStep <= WAD);
```

Since the normalizedAdjustmentStep can not be more than 0.2e18 as per the logic of the method `_getNormalizedAdjustmentStep`, this check is not needed here.

Recommendation:
Remove this unnecessary check.

INFORMATIONAL-3 | RESOLVED

Use existing values to save gas

In Contract VolatilePool.sol, the method _getHaircutRate() has the following logic:

```
        uint256 fromLiability = fromAsset.liability();
        uint256 toLiability = toAsset.liability();
        uint256 rFromAsset = fromLiability > 0 ?
uint256(fromAsset.cash()).wdiv(fromAsset.liability()) : WAD;
        uint256 rToAsset = toLiability > 0 ?
uint256(toAsset.cash()).wdiv(toAsset.liability()) : WAD;
```

Here fromLiability and toLiability are initialized but later on `fromAsset.liability()` and `toAsset.liability()` is used as well.

Recommendation:
Use already initialized variables.

INFORMATIONAL-4 | RESOLVED

Unsafe cast

In Contract VolatilePool.sol, the method _getHaircutRate() has the following logic:

```
return
            poolData.haircutRate +
            DynamicFeeHelper.getVolatilityHaircutRate(dynamicFeeConfig,
volatility.toInt256()) +
            DynamicFeeHelper.getImbalanceHaircutRate(dynamicFeeConfig,
int256(rFromAsset), int256(rToAsset));
```

Here, rFromAsset and rToAsset are cast to int256 unsafely.

Recommendation:
Use .toInt256() for casting rFromAsset and rToAsset.

INFORMATIONAL-5 | RESOLVED

Unnecessary cast

In Contract VolatilePool.sol, the method _postSwapHook() has the following logic:

```
            int256 value = DynamicFeeHelper.safeToLogScale((marketPrice *
1e18) / priceLast, dt);

            DynamicFeeHelper.write(dynamicFeeData[asset],
uint40(block.timestamp), int32(value));
```

Here, the value returned is of type int32 but unnecessarily cast to int256 and again cast to int32 on the next line.

Recommendation:
Remove the unnecessary cast and get the return value as int32.

INFORMATIONAL-6 | RESOLVED

Pending To-do(s)

There are pending To-do(s) in all the audited contracts. It is advised to resolve or remove those To-do(s) before proceeding with production deployment.

Recommendation:
Remove/Resolve the pending to-do(s).

INFORMATIONAL-7 | RESOLVED

Unused constants

In the library RepegHelper, the following constant is unused.

```
int256 private constant WAD_I = 10 ** 18;
```

Recommendation:
Remove the unused constant.

INFORMATIONAL-8 | RESOLVED

Typo/Wrong comment

- RepegHelper, l#80 has a typo.
- RepegHelper, l#207 has a wrong comment. It should be r* >= 1 as per whitepaper.
- DynamicFee, l#46 has a typo.

INFORMATIONAL-9 | RESOLVED

Unused imports

Following import in VolatilePool.sol is not used.

```
import '../../wombat-governance/libraries/LogExpMath.sol';
```

The following imports in RepegHelper are not used.

```
import '../interfaces/IRelativePriceProvider.sol';
```

```
import '../pool/PoolV4Data.sol';
```

The following import in DynamicFeeHelper is not used.

```
import '../interfaces/IAsset.sol';
```

Recommendation:
Removed unused imports.

Use specific Solidity compiler version

Audited contracts use the following floating pragma:

```
pragma solidity ^0.8.5;
```

It allows to compile contracts with various versions of the compiler and introduces the risk of using a
different version when deploying than during testing.

Recommendation:
Use a specific version of the Solidity compiler.

	VolatilePool.sol DynamicFeeHelper.sol RepegHelper.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

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting Wombat Exchange in verifying the correctness of their contract/s code, our team was responsible for writing integration tests using the Hardhat and Foundry testing framework.

The tests were based on the functionality of the code, as well as a review of the Wombat Exchange contract/s requirements for details about issuance amounts and how the system handles these.

VolatilePool - Utils

Queries

✓ estimateNewGlobalEquilCovRatio
✓ getMarketPrice
✓ quoteIdealSwapRate
✓ getNorm
✓ _getNormalizedAdjustmentStep
✓ _getCashValuesWithReserve
✓ _getProposedPriceScales
✓ _getProposedOraclePrices
✓ _getProposedOraclePrices should prevent oracle manipulation
✓ checkRepegCondition - verify adjustment step
✓ checkRepegCondition - verify r*
✓ setPriceAnchor
✓ getTimePointAt

Repeg

✓ Repeg should check repeg conditions are fulfilled
✓ Repeg should update price scale

Oracle

Repeg happens at txn 29, cov ratio = 1.000954599325946402

✓ attemptRepeg

Dynamic haircut

✓ addAsset, removeAsset, then addAsset again
✓ setReserveRateForRepegging
✓ swaps - low volatility
✓ swaps - high volatility

DynamicFeeHelper

Orignal value: BigNumber { value: "12182493960700000000" }
valueInLogScale: BigNumber { value: "671088639" }

✓ toLogScale & fromLogScale

```
✓ safeToLogScale
✓ volatilityOnRange
✓ getMeanVolatilityInWindow
✓ getVolatilityHaircutRate
✓ getImbalanceHaircutRate

26 passing (27s)

Running 9 tests for test/RepegHelper.t.sol:RepegHelperTest
[PASS] test_ShouldNotUpdateOraclePriceMoreThanOnceInSameBlock() (gas: 6703114)
[PASS] test_checkRepegAllConditions() (gas: 6746557)
[PASS] test_checkRepegCondition1() (gas: 6640691)
[PASS] test_checkRepegCondition2() (gas: 6833879)
[PASS] test_getCashValuesWithReserve() (gas: 6680933)
[PASS] test_getMarketPrice() (gas: 6462849)
[PASS] test_getNormalizedAdjustmentStep() (gas: 6644304)
[PASS] test_shouldRepegAndUpdatePriceScale() (gas: 6808419)
[PASS] test_updateOracle() (gas: 6698112)
Test result: ok. 9 passed; 0 failed; 0 skipped; finished in 2.83ms

Ran 1 test suites: 9 tests passed, 0 failed, 0 skipped (9 total tests)
```

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	%UNCOVERED LINES
VolatilePool.sol	100%	100%	100%	100%	
DynamicFeeHelper.sol	100%	100%	100%	100%	
RepegHelper.sol	97%	90%	93%	95%	
All Files	**99%**	**96.67%**	**97.66%**	**98.33%**	

We are grateful for the opportunity to work with the Wombat Exchange 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 Wombat Exchange team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo