SMART CONTRACT AUDIT

zokyo

January 2th 2023 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passes security qualifications to be listed on digital asset exchanges.

TECHNICAL SUMMARY

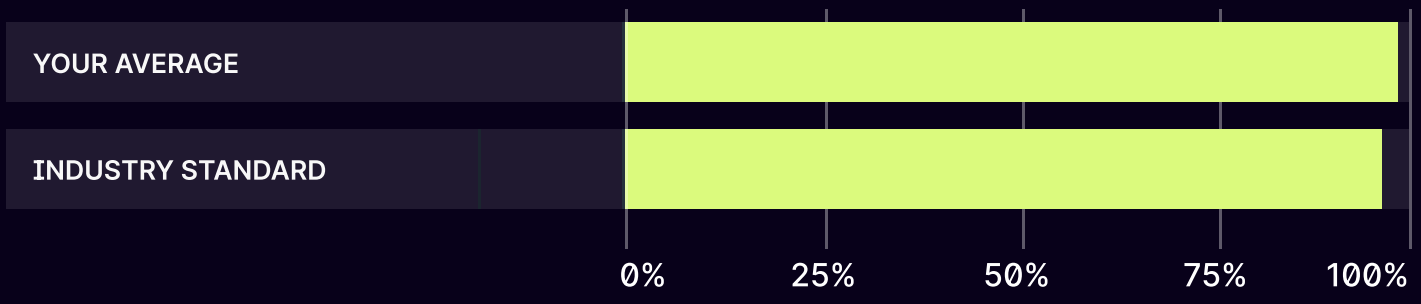
This document outlines the overall security of the OT smart contracts evaluated by the Zokyo Security team.

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

Contract Status

There was 1 critical issue found during the audit. (See Complete Analysis)

Testable Code



99.21% 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 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 OT team put in place a bug bounty program to encourage further active analysis of the smart contract.

Table of Contents

Auditing Strategy and Techniques Applied 3

Executive Summary 4

Structure and Organization of the Document 5

Complete Analysis 6

Code Coverage and Test Results for all files written by Zokyo Security 16

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the OT repository:
https://github.dev/1kxexchange/1kx-v1

Last commit: 54f4b95f372ab4176eea16cdb7e51571ff7c393

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

- FlashLoan.sol
- VariableBorrow.sol
- Curve.sol
- Liquidity.sol
- Osd.sol
- Swap.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 resent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of OT smart contracts. 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 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 logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contracts by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

There was one critical issue found during the audit, alongside 9 with low severity and some informational issues. All the mentioned findings may have an effect only in case of specific conditions performed by the contract owner and the investors interacting with it. They are described in detail in the “Complete Analysis” section.

Contracts are well written and structured. Some findings during the audit shall have an impact on contract performance and security, so it is not fully production-ready, but minor editions by the development team shall make it ready for production.

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” depending on whether they have been fixed or addressed. 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	User's fund can be transferred out without consentr	Critical	Resolved
2	Unchecked possible zero address in Liquidity.sol	Low	Resolved
3	Unchecked possible zero address in VariableBorrow.sol	Low	Resolved
4	Lock solidity pragma	Low	Unresolved
5	Incorrect value comparison for uint8 type in VariableBorrow.sol	Low	Resolved
6	Unchecked possible zero address in VariableBorrow.sol	Low	Resolved
7	Unchecked possible zero address in Swap.sol	Low	Resolved
8	Centralization risk	Low	Unresolved
9	Unchecked return value	Low	Resolved
10	Argument might be risky	Low	Unresolved
11	Change public functions to external where possible to save gas	Informational	Resolved
12	Remove hardhat console imports from the code	Informational	Unresolved
13	Shadowing inherited state variables	Informational	Resolved

#	Title	Risk	Status
14	Shadowing inherited state variables	Informational	Resolved
15	Use of reason strings instead of custom errors	Informational	Unresolved
16	Prefixed variable name	Informational	Resolved
17	Save gas by rearranging variables in the struct Pool declaration	Informational	Resolved

CRITICAL | RESOLVED

User's fund can be transferred out without consent

In contract Swap, the function **settleFutureProfit(address token, uint256 amount, address from)** can transfer funds from any user to itself. Anyone can call **settleFutureProfit(...)** with address of any other user who has approved the spender as Swapl and the tokens can be moved without the consent of the user. A bot can also monitor the approvals on public networks/mem pools and call **settleFutureProfit(...)** leading to the draining of user funds. Also, the user does not get any liquidity token in return, making the tokens immovable from the contract.

Step 1: List token in Swap.sol
Step 2: User approves Swap.sol to spend tokens
Step 3: Anyone can call **settleFutureProfit(token, amount, address_from_step_2)** and transfer tokens of the address from step 2 to Swap.sol.

Recommendation:
Fix the logic in the contract so that no user can transfer tokens of another address and provide liquidity tokens in case the user transfers their own tokens so that tokens can be retrieved.

LOW | RESOLVED

Unchecked possible zero address in Liquidity.sol

In contract Liquidity.sol, at line 21 inside the constructor, the _token parameter can be zero and is not checked. In case the address is zero, the Liquidity token would not function properly and there is no fallback method to set the correct implementation after deployment.

Recommendation:
Add a sanity check for the to address variable to not be zero and revert otherwise.

LOW | RESOLVED

Unchecked possible zero address in VariableBorrow.sol

In contract VariableBorrow.sol, at lines 93 and 94 inside the constructor, the _swap and _oracle parameters can be zero and are not checked.

Recommendation:
Add a sanity check for the to address variable to not be zero and revert otherwise.

LOW | UNRESOLVED

Lock solidity pragma

Contracts should be deployed with the same compiler version and flags that they have been tested the most with. Lock the pragma to a specific version, since not all the compiler versions support all the features.

Recommendation:
Lock pragma to version at least `0.8.12` to support all the features.

Note #1:
Minor change has been made, but it did not fix/address the issue.

LOW | RESOLVED

Incorrect value comparison for uint8 type in VariableBorrow.sol

In contract VariableBorrow.sol, at line 521, 522 and 523 in function updateProtocolRevenue inside the require statements uint types are checked to be greater than or equal to `0`, which is redundant as uint types can never be less than `0`. The checks are meant to ensure that those values are in a certain range e.g [`0`, `100`].

Recommendation:
Remove the greater than or equal comparison and leave only the less than or equal to the range upper bound

LOW | RESOLVED

Unchecked possible zero address in VariableBorrow.sol

In contract VariableBorrow.sol, at line 539 in function setOracle the _oracle parameter can be zero and is not checked. This can lead to reverted transactions throughout the contract as the oracle variable is used in multiple instances.

Recommendation:
Add a sanity check for the to address variable to not be zero and revert otherwise.

LOW | RESOLVED

Unchecked possible zero address in Swap.sol

In contract Swap.sol, at line 933 in function setPriceFeed the setPriceFeed parameter can be zero and is not checked. This can lead to reverted transactions throughout the contract as the priceFeed variable is used in multiple instances.

Recommendation:
Add a sanity check for the to address variable to not be zero and revert otherwise.

LOW | UNRESOLVED

Centralization risk

There are multiple instances where the contract owner can change protocol parameters that affect the behavior of the contracts. These include calls to several functions like updating asset details, protocol revenue and general setters/mutators.

Recommendation:
Consider using multisig wallets and having a governance module

Note #1:
No response to address that the project owners are planning to use multisig wallets to operate the contracts or not.

LOW | RESOLVED

Unchecked return value

In contract VariableBorrow.sol - return value of `_updateInterest` is not checked in almost all its occurrences.

Recommendation:
Wrap the result of the call in a require statement or handle return according to the desired behavior for each particular case.

Note #1:
`updateInterest` no longer returns value

LOW | UNRESOLVED

Argument might be risky

In contract VariableBorrow.sol - in methods `borrow` and `repay` the argument `to` mostly referring to `msg.sender` adds a dimension that can be exploited if things are not perfectly implemented in `Router` contract. The case in which the `to` is not `msg.sender` exists if `router = msg.sender`. But implementation of the router contract is not the concern of this audit's scope. If the router is exploited by an attacker to bypass the `require` check, severe consequences might take place that will lead to attackers exploiting others investors' collateral. In this contract presumably we have a securely implemented router, hence severity is low, but a simple coding refactor is recommended.

Recommendation:
It is a better coding practice and more secure to isolate those external methods in this discussed scenario. Create methods exposed for EOA callers without `to` for `borrow` and `repay`. Also, implement other methods restricted for the router callers that include the `to` address. As for the logic itself, implement it in `internal` methods to be invoked by the external methods, suggested saving gas of course.

Note #1:
No change done to address this.

INFORMATIONAL | RESOLVED

Change public functions to external where possible to save gas

In contract Flashloan.sol, at line 18 function flashLoan is declared as public but never used for internal calls, no reason to be public.

Recommendation:
Change function from public to external

INFORMATIONAL | UNRESOLVED

Remove hardhat console imports from the code

In contracts Swap.sol and VariableBorrow.sol the hardhat/console.sol library is imported and never used.

Recommendation:
Remove all occurrences of hardhat/console.sol imports from production code

INFORMATIONAL | RESOLVED

Shadowing inherited state variables

In contract Liquidity.sol, in function lpName, the variable symbol is a shadowing variable with the same name from inherited contract ERC20.

Recommendation:
Rename function parameter symbol.

INFORMATIONAL | RESOLVED

Shadowing inherited state variables

In contract Liquidity.sol, in function lpSymbol, the variable symbol is a shadowing variable with the same name from inherited contract ERC20.

Recommendation:
Rename function parameter symbol.

INFORMATIONAL | UNRESOLVED

Use of reason strings instead of custom errors

Inside the contracts the reason strings are used inside require statements as revert messages. An alternative would be to use custom errors. This contributes to reducing contract size and reducing overall gas cost.

Recommendation:
Change reason strings to custom errors.

INFORMATIONAL | RESOLVED

Prefixed variable name

In contract Swap.sol, at line 102 the variable of type IBorrowForSwap is prefixed with the "$" sign. This is not considered naming best practice in Solidity and is not ideal for readability.

Recommendation:
Remove the "$" prefix

Save gas by rearranging variables in the struct Pool declaration

The structs used in the contract use storage slots in EVM. The arrangement of variables defined in the *struct* affects the storage slots required. All reads and writes to storage in Solidity are handled in 32-byte increments. Solidity tightly packs variables where possible, storing them within the same 32 bytes. The best use of storage available slots is possible by re-arranging the variables in the *struct Pool* such that multiple variables can be stored in the same storage slot.

Recommendation:
Re-organize variables *struct Pool* declaration for efficient use of storage slot.
Suggested declaration as below:

```
struct Pool {
    uint256 reserve;
    uint256 lastRatioToken;
    uint256 lastRatioOsd;
    uint256 osd;
    uint256 revenueOsd;
    // status? paused?
    uint256 createdAt;
    uint16[3] feeRates; // 150/1000000 = 0.15%
    IERC20 token;
    Liquidity liquidity;
    bool rebalancible;
    bool usePriceFeed;
    // type => rate
    // 0: default: used for token <=> osd
    // 1: used for token
    // 2: stable
    uint8 feeType;
    uint8 revenueRate; // 70/100 = 70%
    uint8 tokenDecimal;
}
```

	FlashLoan.sol VariableBorrow.sol Curve.sol Liquidity.sol Osd.sol Swap.sol
Re-entrancy	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 OT in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Hardhat testing framework.

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

VariableBorrow

updateAsset
✓ updateAsset (619ms)
borrow
✓ borrow (468ms)
repay
✓ repay (860ms)
liquidate
✓ liquidate (891ms)
✓ Should liquidate low CR (560ms)
updateProtocolRevenue
✓ updateProtocolRevenue (210ms)
updateInterest
✓ updateInterest (332ms)
maxFlashLoan
✓ maxFlashLoan
extractProtocolRevenue
✓ extractProtocolRevenue (493ms)
FlashLoan
✓ Should flash loan (508ms)
getAssetsView
✓ getAssetsView
setRouter
✓ setRouter (73ms)
getAccountDebt
✓ getAccountDebt (232ms)
getCollaterals
✓ getCollaterals

liquidatable
✓ liquidatable (215ms)
getDebt
✓ getDebt (195ms)
getAssetsView
✓ getAssetsView
setOracle
✓ setOracle (71ms)
Viewers
✓ Should getMaxAmountOfBorrow() (496ms)
✓ Should getMaxAmountOfRepay() (227ms)

Curve

✓ Should be able to get constant price
✓ Should be able to get constant product out
✓ Should be able to get value out
✓ Should be able to get value out
✓ Should be able to get amount out (55ms)
✓ Should be able to get amount out

Liquidity

✓ Should be deployed correctly
✓ Should be able to get decimals
✓ Should be able to mint (49ms)
✓ Should be able to burn (74ms)

Osd

✓ Should be deployed correctly
✓ Should be able to add minters (81ms)
✓ Should be able to mint (69ms)
✓ Should be able to burn (124ms)

Swap

✓ Should be deployed correctly
✓ Should be able to able to set borrow (59ms)
✓ Should be able to able to set price feed (53ms)
✓ Should be able to list token (389ms)
✓ Should be able to add liquidity (331ms)
✓ Should be able to remove liquidity (511ms)
✓ Should be able to get liquidity in (419ms)
✓ Should be able to get liquidity out (349ms)
✓ Should be able to get pool fee policy (45ms)
✓ Should be able to get pool info
✓ Should be able to update pool (458ms)
✓ Should be able to get fee rate (286ms)

✓ Should be able to swapin (1604ms)
✓ Should be able to swapout (1975ms)
✓ Should be able to get token price
✓ Should be able to view liquidity out amount when pool uses price feed (483ms)
✓ Should be able to get debt (315ms)
✓ Should be able to swap osd (324ms)
✓ Should be able to swap token directly (425ms)
✓ Should be able to get amount out (961ms)
✓ Should be able to get amount in (931ms)
✓ Should be able to settle future profit (488ms)
✓ Should be able to get pool reserve (189ms)
✓ Should be able to get pool state (351ms)
✓ Should be able to get pool availability (402ms)
✓ Should be able to get price ratio (636ms)
✓ Should be able to get protocal revenue extract (706ms)
✓ Should be able to repay (570ms)
✓ Should be able to borrow (730ms)

SwapRouter

✓ Should be able to swap out (296ms)
✓ Should be able to swap out when WETH is token in (590ms)
✓ Should be able to swap out when WETH is token out (651ms)
✓ Should be able to swap in (156ms)
✓ Should be able to swap in when weth is token in (243ms)
✓ Should only be able to receive from weth contract (294ms)
✓ Should be able to swap in when weth is token out (432ms)
✓ Should be able to add liquidity (381ms)
✓ Should be able to remove liquidity (627ms)
✓ Should be able to get WETH address
✓ Should revert on non existing function call (194ms)

74 passing (3m)

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

FILE	% STMTS	% BRANCH	% FUNCS	% LINES	% UNCOVERED LINES
FlashLoan.sol	100	75	100	100	
VariableBorrow.sol	98.28	95.71	100	98.28	380,384,445,446
Curve.sol	100	100	100	100	
Liquidity.sol	100	100	100	100	
Osd.sol	100	100	100	100	
Swap.sol	99.65	93	100	99.65	718
swapRouter.sol	100	100	100	100	
All Files	**99.21**	**95.18**	**100**	**99.21**	

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

zokyo