



SMART CONTRACT AUDIT



July 25th 2023 | v. 1.0

Security Audit Score

PASS



TECHNICAL SUMMARY

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

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

Contract Status



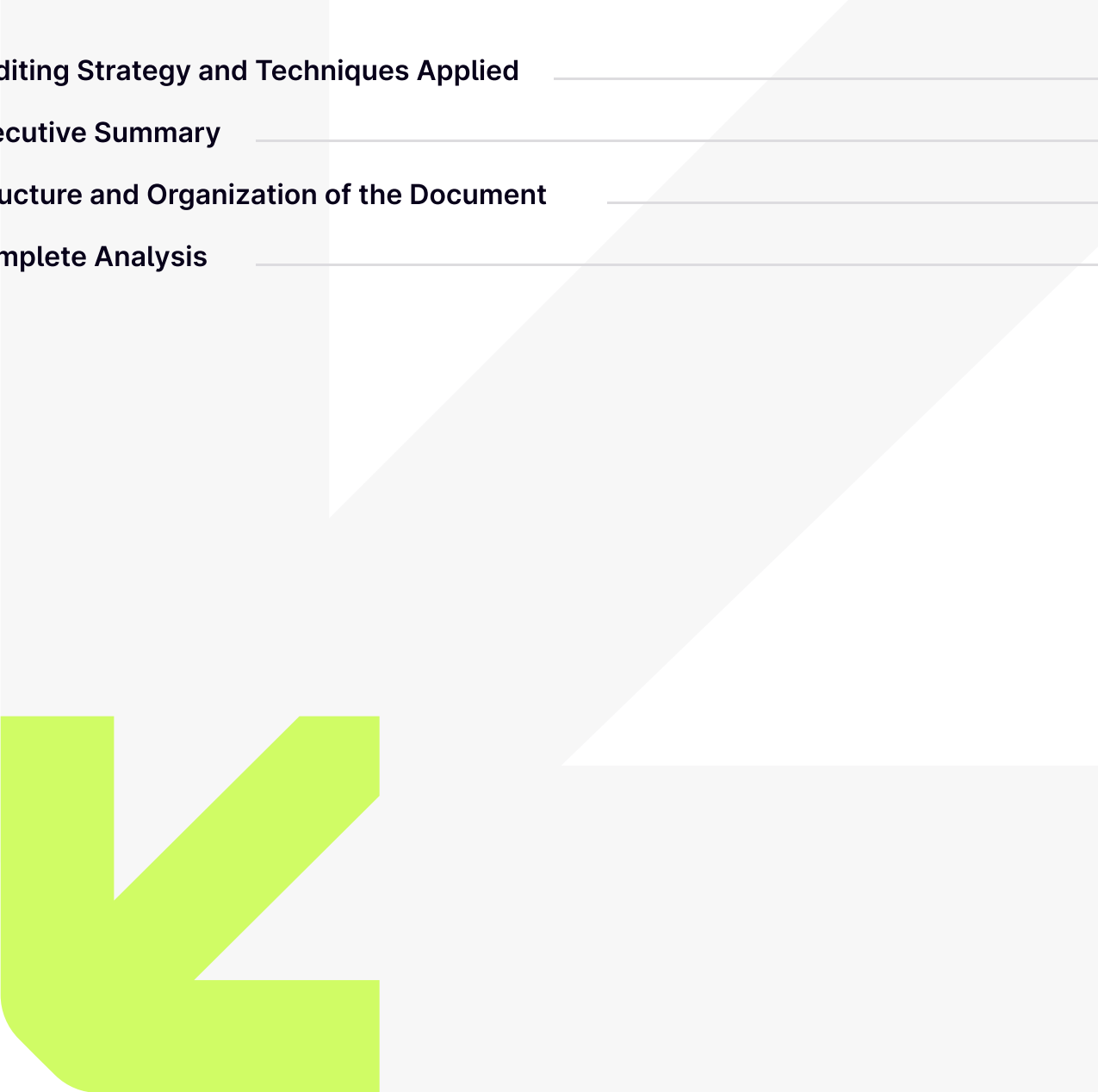
There were no critical issues found during the audit. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts 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 Vaultka team put in place a bug bounty program to encourage further active analysis of the smart contracts.



Table of Contents

Auditing Strategy and Techniques Applied	3
Executive Summary	4
Structure and Organization of the Document	5
Complete Analysis	6



AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Vaultka repository:
<https://github.com/Vaultka-Project/sake-contracts>

Last commit - [e7473f2ba2002e1c4ebd4075b864b22f603f4750](#)

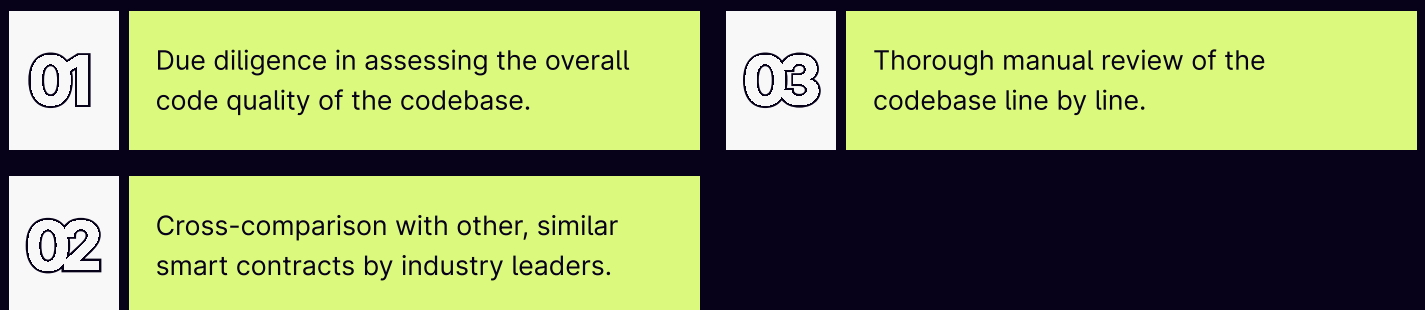
Within the scope of this audit, the team of auditors reviewed the following contract(s):

- /contracts/Sake.sol
- /contracts/WaterV2.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 Vaultka 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. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:










Executive Summary

During the audit, no critical issues were identified, alongside high, medium, and low severity issues, along with a few informational concerns. A detailed examination of these issues can be located 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 Vaultka team and the Vaultka 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. |  Low
The issue has minimal impact on the contract's ability to operate. |
|  High
The issue affects the ability of the contract to compile or operate in a significant way. |  Informational
The issue has no impact on the contract's ability to operate. |
|  Medium
The issue affects the ability of the contract to operate in a way that doesn't significantly hinder its behavior. | |

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	_userInfo.liquidated is not set when liquidatePosition is success	High	Resolved
2	Possible re-entrance	High	Resolved
3	Function argument re-assigned	High	Resolved
4	Use safeIncreaseAllowance instead of safeApprove	Medium	Resolved
5	Liquidation works while paused	Medium	Acknowledged
6	Using storage variables instead of memory ones	Low	Acknowledged
7	Inconsistency in function declarations	Low	Acknowledged
8	Modifier never used	Low	Resolved
9	Tautology or contradiction	Low	Resolved
10	Uncommented code	Informational	Acknowledged
11	Complex functions, repeating functionality	Informational	Acknowledged

_userInfo.liquidated is not set when liquidatePosition is a success

When the `liquidatePosition` is successful the _userInfo.liquidated is not set to true which will lead to repeated liquidations of the same position

Recommendation:

set _userInfo.liquidated to true

Possible re-entrance

In the contract SakeVault there is a `velaStakingVault` with a function `withdrawVlp` which has a not introduced functionality. In this case, this function has callbacks, that could potentially lead to re-entrance issues in all functions when it used: `closePosition` and `liquidatePosition`.

Recommendation:

make sure `withdrawVlp` does not call any external sources or protect `closePosition` and `liquidatePosition` from re-entrance attack

Function argument re-assigne

In the contract SakeVault at line #345 in the method `openPosition` the argument `user` is being re-assigned right at the beginning of the function making that argument not used at all.

Recommendation:

remove the argument from the function or review the assignment

Use `safeIncreaseAllowance` instead of `safeApprove`

While calling external contracts that consume your assets, try to avoid using `safeApprove` and use `safeIncreaseAllowance` instead. That will ensure you're not getting the 'SafeERC20: approve from non-zero to non-zero allowance' revert in the case when the external contract doesn't consume the entire allowance.

Recommendation:

replace `safeApprove` to `safeIncreaseAllowance` calls

Liquidation works while pause

The `liquidatePosition` function is functioning while the `SakeVault` is on pause

Recommendation:

check and confirm if this is a correct behavior.

Comment from the client: By design, we don't want bad debt at any time.

Using storage variables instead of memory ones

Using storage variable for multiple reading access (ex: in ``closePosition`` function) costs lots of gas. Consider using memory variables when multiple readings occur. Do not forget that copying excess fields from the storage to memory would also cost additional gas that you can save.

Recommendation:

changing the variable from storage to memory and replacing assignments directly to the storage variable will save you gas (only ``closePosition`` could save about 19k of gas).

Comment from the client: `_userInfo` is indeed reading but also writing hence the need for storage

Note #1: While still writing, there is a way to save tons of gas by only reading what you need once and writing only what was changed. As we mentioned earlier, a small fix in the `closePosition` function could save about 19k of gas

Inconsistency in function declarations

Right now there are 4 functions that accept both arguments user address and the id of position: ``getUpdatedDebtAndValue``, ``getCurrentPosition``, ``closePosition``, and ``liquidatePosition``. all of them but ``liquidatePosition`` accept position then user, while the ``liquidatePosition`` accepts user, then position.

Recommendation:

change the order of the arguments in the ``liquidatePosition`` function

Modifier never used

In the contract Water2 there is a modifier `onlyAdmin` and the state variable `admin` which is never used in the code.

Recommendation:

remove the modifier and state variable declarations.

Tautology or contradiction

In the contract, SakeVault in the method `updateFeeStrategyParams` there is a check for `_feeStrategy.maxFeeSplitSlope1` to be greater or equal to zero. However, the declared type of this element is `uint128` which certainly could not go down to zero

Recommendation:

review if that statement is what you really wanted to check for and if so, remove it because it is not needed there

Uncommented code

The code looks to be very complex, but it lacks comments.

Recommendation:

add comments to the code, as well as NatSpecs for each function

Complex functions, repeating functionality

The current function implementation is too complex and also often repeat the functionality already presented in other function.

Recommendation:

split the functionality into multiple functions, re-use functions with the same functionality

	/contracts/Sake.sol /contracts/WaterV2.so
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

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

