VELOCORE

SMART CONTRACT AUDIT

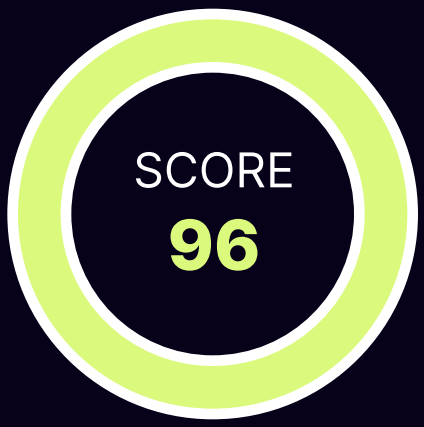
zokyo

August 14th 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 Velocore smart contracts evaluated by the Zokyo Security team.

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

Contract Status

There were 0 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 Velocore 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

The source code of the smart contract was taken from the Velocore repository: https://github.com/velocore/audit

Last commit - 179d215082383454881f6475bca10ca54af77495

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

- AdminFacet.sol
- Diamond.yul
- SwapFacet.sol
- VaultStorage.sol
- Pool.sol
- SatelliteUpgradeable.sol linear-bribe
- PoolWithLPToken.sol
- SingleTokenGauge.sol
- Satellite.sol
- ConstantProductLibrary.sol
- ConstantProductPool.sol
- ConstantProductPoolFactory.sol
- LinearBribe.sol
- LinearBribeFactory.sol
- VC.sol
- VeVC.sol
- WombatPool.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's Security Team has followed best practices and industry-standard techniques to verify the implementation of Velocore smart contracts. To do so, the code is reviewed line-by-line by our smart contract developers, documenting any issues as they are 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 contracts by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

No critical vulnerabilities were identified during the audit but were found issues in levels of severity—high, medium, and low—along with some informational issues. These details are elaborated upon in the "Complete Analysis" segment.

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 Velocore team and the Velocore team is aware of it, but they have chosen to not solved 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	Double spending through Double Entry Point Tokens	High	Resolved
2	Transfer design prone to DOS	High	Acknowledged
3	External call to transfer ETH might fail because of memory returned data	High	Resolved
4	Unchecked delegatecall result	Medium	Resolved
5	Initializers are missing lock mechanism	Medium	Resolved
6	Allowance frontrunning results in double-spending	Medium	Resolved
7	Diamond implementation does not follow standard	Low	Resolved
8	Incorrect assessment of delegatecall result	Low	Acknowledged
9	Redundant logic for 'view' function	Low	Acknowledged
10	Add return statement	Informational	Resolved
11	Transaction executed successfully in _execute parameter edge case.	Informational	Resolved
12	Debugging statements left in code	Informational	Resolved
13	Require statements without error messages	Informational	Acknowledged
14	Vulnerable to phishing	Informational	Resolved

Double spending through Double Entry Point Tokens

In contract SwapFacet, method _execute is vulnerable to a read-only reentrance through the convert flow (opType == 2), if the same balance is read from the same tokens twice in the for loop the same delta will be calculated twice creating a double spending and allowing the attack to steal assets from the vault, normally this flow is not possible because the _sort functionality is taking care of eliminating the duplicates from the tokens list, however, this flow is not taking into consideration the Double Entry Point tokens (see TUSD and SNX as some examples).
Scenario:

1. Alice select opType = 2 here the opDest is her special contract.
2. opAmount for DVT (token name as example) will be `0` for both addresses to just continue the operation and skip the transfer (we took the most straight-forward case)
3. DVT balance for Vault is `100` tokens so in both iterations values `100` will be saved in balances array.
4. External call to opDst happens, here Alice contract send `100` DVT tokens to Vault, so vault balance is now `200` DVT
5. In the next food loop where the delta is calculated (difference between balanceAfter and balanceBefore) will be `200 - 100 = 100`
6. In the first iteration `100` tokens are added to the cumDelta for the respective token, then in the second iteration because there was not transfer yet delta will be again `100` so value `100` will be added again in the cumDelta token for the second address (which represents the same balances but Vault have no way to know that)
7. After that the settlement will happen as the last operation of the execute function and in the first iteration `100` tokens will be send (the modified delta and basically the initial tokens that Alice send to the Vault through velocore_convert) and in the second iteration again `100` tokens will be send of the same tokens through the different address and this will represent the profit for the attack
8. Now attacker have `200` DVT and Vault `0` DVT.

Recommendation:

In the case of conversions do the tokens settlement in the loop iterations so that when balanceOf() will be called again (for the second iteration) the balance will be updated already.

Note #1: It is worth noticing that the pattern of double entry point tokens is not common in the EVM ecosystem and is not something that is usually implemented.

HIGH-2 | ACKNOWLEDGED

Transfer design prone to DOS

The Vault contract through execute function is executing different user operations while tracking the user virtual balances and as a last step is setting the user balances by taking care of the tokens transfer, from user to vault or vice versa. There is a problem in this design because a user can carry, as an example, multiple swap operations and when the settlement will happen if one of those settlement will fail because it managed to by DOS by an external actor the user transaction will be reverted because 1 of maybe 20 transfers failed

Recommendation:

1. Do the settlement of tokens/virtual balances in try-catch blocks.
2. If one of the settlement transfer fails, reset the pool balance for that particular pool.

HIGH-3 | RESOLVED

External call to transfer ETH might fail because of memory returned data

The Token library is used as a functionality wrapper to work with all different types of tokens that the protocol supports, ERC20, ERC721, ERC1155 and ETH (or the native cryptocurrency of the EVM compatible chain) . For the transfer of ETH (native tokens) the contract is doing a simple call to the destination address and after that is checking if the transfer have succeeded, Token.sol L#224. There is a problem with this approach, even if in the business logic you have do not use the returnedData field, that value will still be copied into memory and that can cause a DOS as it will revert with out of gas if the blob returned is to big.

Recommendation:

Do the low level call using assembly (YUL) as in that because it will completely ignore the returnedData

MEDIUM-1 | RESOLVED

Unchecked delegatecall result

In contract Diamond.yul at line 30, inside the constructor, there is a delegatecall performed back to the caller contract (AdminFacet) which executes the initializeFacet. The result of the delegate call is not checked for success/failure.

Recommendation:
Add a check for the delegatecall result

MEDIUM-2 | RESOLVED

Initializers are missing lock mechanism

Contracts Ve and veVC are missing a lock mechanism for the initialize function. It is part of best practices and recommandations to lock all your initializers, the lock mechanism can be added either through the intializer modifier from openzeppelin or by adding a simple lock variable that will be turned to true after the initializer will be first called and everytime somones wants to call the initialize function the first operation from the initialize function will be a check against the lock variable value.

Recommendation:
Add a lock mechanism for the initializers

MEDIUM-3 | RESOLVED

Allowance frontrunning results in double-spending

Contract PoolWithLpToken is vulnerable to the ERC20 approve and double spend front-running attack. In this attack, a token owner authorizes another account to transfer a specific amount of tokens on their behalf, and in the case that the token owner decides to change that allowance amount, the spender could spend both allowances by front running the allowance-changing transaction.

Recommendation:
Add increaseAllowance and decreaseAllowance functions to mitigate this attack

LOW-1 | RESOLVED

Diamond implementation does not follow standard

The current diamond implementation does not follow the standard as it has elements missing and some flows have been altered to produce the same results in a different way than described in the standard. The purpose and rationale of writing the Diamond contract entirely in yul is understandable, but the gas/memory-ops savings don't offset the risks associated with the increased complexity. Besides the fact that elements are missing, such as LibDiamond, DiamondLoupeFacet, there's different ways to delegatecall the target implementation, which appears unnecessary.

Recommendation:
1.Rewrite the Diamond contract in solidity and only use yul for the fallback delegatecalls to implementations
2. Add the helper facets such as DiamondLoupe, DiamondCut and use a straightforward way of mapping implementation addresses to selectors (i.e LibDiamond)

LOW-2 | ACKNOWLEDGED

Incorrect assessment of delegatecall result

In contract Diamond.yul at lines 49-50 and 80-81, after performing delegate calls, the execution result is checked in an incorrect way. It reverts if `success` i.e non-zero value, returning data for any other case. Throughout the contract there are other delegate calls performed, where the result of the execution is evaluated using a switch statemenet, where for the result of `0` it reverts, returning for all other cases. This is the correct approach, as delegatecall returns `0` on error.

Recommendation:
Change the result evaluation to the correct approach, as seen at lines 66-72

LOW-3 | ACKNOWLEDGED

Redundant logic for 'view' function

In the Diamond.yul contract, there are different flows for calling a view and executing a function. Both perform a delegatecall to a target contract, the difference being that for the `view` flow, the destination address is read from calldata, in case the first 4 bytes of the calldata are equal to `view`. Also, if the `implementation` address is not valid for a function call, the contract creates the calldata needed for the `view` flow and performs a delegate call to self, with the new destination address being a bitwise_not of `implementation`. This is confusing and can lead to undefined behavior, as it relies on calldata and assumes a view call can be performed. The same applies for having separate flows for `view` and `normal` calls, as this can't really be enforced other than checking the calldata selector, which can be altered given it is user input. This contract should simply delegate calls to its different facets.

Recommendation:
1. Keep a single flow for delegatecalls, regardless whether it's a `view` or `normal` call
2. Remove the "fallback" to view calls if the implementation loaded from storage is not "valid". This adds complexity and confusion as it deviates from the contract's purpose

INFORMATIONAL-1 | RESOLVED

Add return statement

In the function spotPrice inside the contract VelcoreLens.sol, there are multiple conditions that influence the parameters with which the function will be called recursively. However, if the function does not call itself, there is no return to be executed on this branch.

Recommendation:
To follow the best practices, add a return statement at the end of the function.

INFORMATIONAL-2 | RESOLVED

Transaction executed successfully in _execute parameter edge case.

In contract SwapFacet, function _execute logic choose the type of operation that will be performed based on the opType variable, the value for opType can be 0,1,2,3,4 each different integer represents a different operation (swap, convert, stake, etc..). However if the opType value will be for example 5, the transaction will be executed successfully and the execute function will do nothing, in this case it will be recommend to fail early to not cost users to much gas.

Recommendation:
Fail the executio early by first checking the opType is between some boundaries and only after that check what that opType is, as an example if it is a value smaller then 5 simply revert the transaction.

INFORMATIONAL-3 | RESOLVED

Debugging statements left in code

In the whole project there are multiple lines of code that invoke console.log functions that only have a role during the debugging phase, for production code, it will be recommended to remove them.

Recommendation:
Remove all console.log function calls

INFORMATIONAL-4 | ACKNOWLEDGED

Require statements without error messages

In the whole project there are multiple require statements that do not have an error message associated with it, it would be recommended to have an error message associated with all the require statements.

Recommendation:
Add error messages to all require statements.

INFORMATIONAL-5 | RESOLVED

Vulnerable to phishing

Contract AdminFacet is vulnerable to phishing as it is using tx.origin, we understand that is because of the forge deploy script however we will urge you to change the deployment mechanism and change the logic so it will not be depended of tx.origin anymore.

Recommendation:
Change the logic dependency of tx.origin between AdminFacet and deployment mechanism.

	AdminFacet.sol **Diamond.yul** **SwapFacet.sol** **VaultStorage.sol** **Pool.sol** **SatelliteUpgradeable.sol linear-bribe** **PoolWithLPToken.sol** **SingleTokenGauge.sol** **Satellite.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

	ConstantProductLibrary.sol **ConstantProductPool.sol** **ConstantProductPoolFactory.sol** **LinearBribe.sol** **LinearBribeFactory.sol** **VC.sol** **VeVC.sol** **WombatPool.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

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

zokyo