SMART CONTRACTS REVIEW

zokyo

August 6th 2024 | v. 1.0

Security Audit Score

PASS

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

ZOKYO AUDIT SCORING VAULTKA

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:
- 2 Critical issues: 2 resolved = 0 points deducted
- 0 High issues: 0 points deducted
- 2 Medium issues: 1 resolved and 1 acknowledged = - 4 points deducted
- 7 Low issues: 7 resolved issues = 0 points deducted
- 6 Informational issues: 6 resolved and 1 acknowledged = 0 points deducted

Thus, 100 - 4 = 96

TECHNICAL SUMMARY

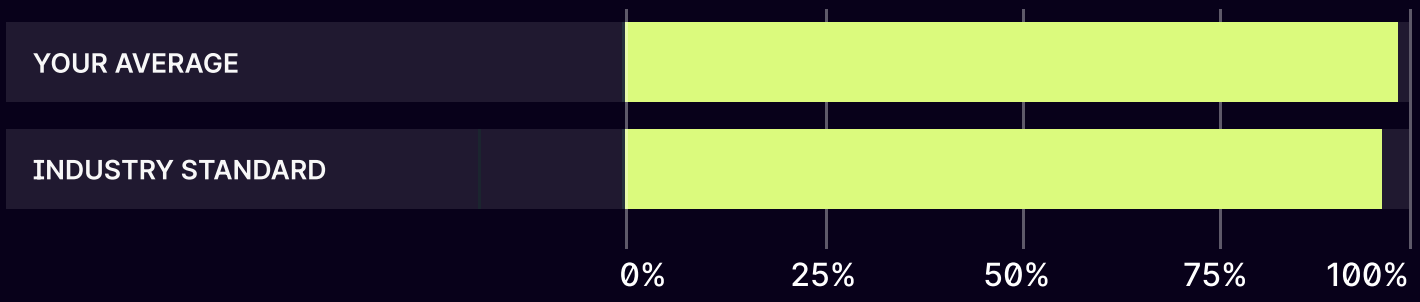
This document outlines the overall security of the Vaultka smart contract/s evaluated by the Zokyo Security team.

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

Contract Status

There were 2 critical issues found during the review. (See Complete Analysis)

Testable Code



The code coverage is 82.5%.

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 Solana 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 contract/s.

Table of Contents

Auditing Strategy and Techniques Applied 5

Executive Summary 7

Structure and Organization of the Document 8

Complete Analysis 10

Code Coverage and Test Results for all files written by Zokyo Security 18

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Vaultka repository:
Repo: https://github.com/Vaultka-Project/vaultkarust

Last commit -cf151df47e0a86299f297349de3fea6c4736636c

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

- Strategy and lending programs

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Vaultka 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.	03	Thorough manual review of the codebase line by line.
02	Cross-comparison with other, similar smart contract/s by industry leaders.	04	Thorough manual review of the codebase line by line.

METHODOLOGY

Our security audit methodology for Solana smart contracts developed with the Anchor framework employs a multi-faceted approach to ensure comprehensive security assurance. We integrate principles from established frameworks like OWASP and MITRE ATT&CK, adapting them to the unique characteristics of Solana and Anchor.

Automated Testing & Analysis:

- State Consistency: Rigorously verify the consistent state transitions of accounts and program data on the Solana blockchain, ensuring proper rollback mechanisms in case of transaction failures.
- Anchor IDL Verification: Validate the accuracy and security of Anchor's interface definition language (IDL), ensuring alignment between client-side interactions and on-chain program logic.
- Arithmetic Safety: Test for robust prevention of integer overflow and underflow vulnerabilities within Anchor instructions and data structures.
- Input Validation: Enforce strict parameter validation on all CPI (Cross-Program Invocation) calls and external data inputs to mitigate injection attacks and unexpected program behavior.
- Error Handling: Assess comprehensive error handling within Anchor programs, ensuring appropriate responses to potential failures, including those arising from CPI interactions or external dependencies.
- Account Discrimination: Verify the proper discrimination of accounts within Anchor programs, preventing unauthorized access or manipulation of sensitive account data.
- Seed Security: Evaluate the security of account seeds used in Anchor programs, ensuring they are generated randomly and securely stored to prevent unauthorized account creation or compromise.
- Secure Coding Practices: Enforce adherence to established Rust and Anchor security best practices, informed by guidelines from OWASP and Solana-specific recommendations.

METHODOLOGY

Meticulous Code Review:

- Scope: In-depth manual examination of the Rust codebase, focusing on potential security weaknesses, logic flaws, and common vulnerabilities specific to Solana and Anchor. Our review incorporates best practices from frameworks like OWASP and MITRE ATT&CK.
- Anchor-Specific Concerns: Pay particular attention to the proper use of Anchor macros, constraints, and account derivations, as well as potential vulnerabilities related to CPI interactions and account cross-contamination.

Formal Verification (Where Applicable):

- Critical Functionality: Explore the application of formal verification techniques, when feasible, to mathematically prove the correctness of core smart contract functions on the Solana blockchain, providing a high level of assurance.

Collaborative Audit Process:

- Testnet Deployment: Execute targeted tests on a Solana testnet environment, replicating real-world transaction scenarios to evaluate smart contract behavior under realistic conditions.
- Audit Trail: Diligently document all findings, including identified vulnerabilities, code weaknesses, and recommended mitigations, creating a transparent and actionable audit report.

Commitment to Excellence

- This methodology reflects our dedication to comprehensive security assessments of Solana smart contracts developed with the Anchor framework. Our goal is to uncover and remediate vulnerabilities, protecting your projects and users from potential exploits in the dynamic Solana ecosystem.

Executive Summary

The Zokyo team conducted a security audit of the Rust smart contract codebase developed using the Anchor framework for the Solana blockchain. While the contracts demonstrate core functionality, the audit revealed opportunities for improvement in code structure and testability. Several critical vulnerabilities were identified and successfully remediated during the audit process.

Key areas for potential refactoring include:

- Code Organization: Implementing a more modular design would enhance readability and maintainability.
- Function Granularity: Refactoring overloaded functions into smaller, more focused units would improve testability and isolation of logic.

A detailed breakdown of findings and recommendations can be found in the “Complete Analysis” section below.

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.

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	Position_amount never updated after withdrawal	Critical	Resolved
2	Debt could be relieved without repayment	Critical	Resolved
3	Centralized control in the program	Medium	Acknowledged
4	User should be both `keeper` and `owner` to be able to set `to_be_liquidated = true`	Medium	Resolved
5	Price multiplied by ten	Low	Resolved
6	Duplicated check	Low	Resolved
7	The event doesn't have an address	Low	Resolved
8	The event doesn't have an address	Low	Resolved
9	unused account field	Low	Resolved
10	use `sub_lamports` and `add_lamports`	Low	Resolved
11	Incorrect error message	Low	Resolved
12	Code quality	Informational	Acknowledged
13	Unnecessary variables	Informational	Resolved
14	Path prefix not necessary	Informational	Resolved
15	Unsigned check for less than zero	Informational	Resolved
16	Unnecessary variables	Informational	Resolved
17	Multiple calls to the same field	Informational	Resolved

CRITICAL-1 | RESOLVED

Position_amount never updated after withdrawal

File: strategy/programs/vaultka/src/lib.rs
Description: The function `request_deposit` sets initiates the "PositionInfo" with `position_amount=0`. Then, the function `execute_deposit` updates it to be equal to `received_amount` right before the end of the function execution. The `request_withdraw` uses this value to determine the amount to be withdrew and, actually does send tokens to the keeper. There's also no check if the position_info.is_in_withdraw_request is already `true`, which makes it possible to call it again many times always getting the tokens.

Recommendation:
re-check the logic in those four function, it seems to be totally corrupted. Try to isolate functionality and make sure to document all checks you needed

CRITICAL-2 | RESOLVED

Debt could be relieved without repayment

File: lending/programs/water/src/lib.rs: 614
Description: the `repay` function increases a `vault_sol_balance` and decreases a `borrowed_amount` without receiving any payments

Recommendation:
add a Transfer function to get the payment from the borrower

MEDIUM-1 | ACKNOWLEDGED

Centralized control in the program

Description: Both programs (strategy and vault) have centralized control. In the Strategy program, an admin account can manually specify any amount within either execute_withdraw or execute_deposit, while in the Vault program, any whitelisted account can borrow assets and repay the debt without paying assets back.

Recommendation:
update methods to decrease centralization

MEDIUM-2 | RESOLVED

User should be both `keeper` and `owner` to be able to set `to_be_liquidated = true`

File: strategy/programs/vaultka/src/lib.rs: 902 and 913
Description: In the `request_withdraw` function in the case when "dtv" is greater or equals to the "strategy.dtv_limit", there is a check for the `accounts.user` to be the strategy keeper. and right after that, on the line 913 it checks the same user to be an owner of the position

Recommendation:
Make sure the owner and the keeper could be the same account for this case to work

LOW-1 | RESOLVED

Price multiplied by ten

File: strategy/programs/vaultka/src/lib.rs: 692, 705
Description: In the `request_deposit` function there are two statement: `sol_price = sol_price * 10` and `jlp_price = jlp_price * 10`, that do not make any sense.

Recommendation:
Please make sure if it's needed and if so, add comments to explain this adjustment.

LOW-2 | RESOLVED

Duplicated check

File: strategy/programs/vaultka/src/lib.rs: 907 and 913
Description: In the `request_withdraw` function in the case when "dtv" is less than "strategy.dtv_limit", there is a check for the `accounts.user` to be equal to `position_info.user`, but right after that it is checked again.

Recommendation:
Remove duplicated check

LOW-3 | RESOLVED

The event doesn't have an address

File: lending/programs/water/src/lib.rs: 345
Description: The function `set_whitelisted_user` fires an event `AddedWhitelistedUser` with only one boolean flag: "whitelisted: true" which makes no sense.

Recommendation:
Include the whitelisted address for the event.

LOW-4 | RESOLVED

The event doesn't have an address

File: lending/programs/water/src/lib.rs: 361
Description: The function `disable_whitelisted_user` fires an event `DisabledWhitelistedUser` with only one boolean flag: "whitelisted: true" which makes no sense.

Recommendation:
Include the whitelisted address for the event.

LOW-5 | RESOLVED

unused account field

File: lending/programs/water/src/lib.rs: 171
Description: The `Deposit` structure defines an account `program_authority` with an `AUTHORITY_SEED`, but it is never used

Recommendation:
make sure you don't need this account in the deposit function and remove it from the structure

LOW-6 | RESOLVED

use `sub_lamports` and `add_lamports`

File: lending/programs/water/src/lib.rs: 539,540,543,544,601,602
Description: The anchor has functions to sub/add lamports, which are more ergonomic, performant, and have additional checks.

Recommendation:
use anchor-provided functions to sub/add lamports to accounts

LOW-7 | RESOLVED

Incorrect error message

File: lending/programs/water/src/lib.rs: 628
Description: the code is checking for the `whitelisted.whitelisted` but the error thrown is `InvalidBorrowAmount`

Recommendation:
make sure the thrown errors are correct

INFORMATIONAL-1 | ACKNOWLEDGED

Code quality

Description: The code quality of the project is bad. The code is not well structured and the code is not well commented.
Functions are overly complex and the code is not modular. The code itself is very hard to read, not self-explanatory, and hard to maintain.

Recommendation:
Refactor the code to make it more readable and maintainable. Use proper naming conventions and break down the code into smaller functions.

INFORMATIONAL-2 | RESOLVED

Unnecessary variables

File: strategy/programs/vaultka/src/lib.rs: 712,713
Description: In the `request_deposit` function there are two variables introduced: `jlp_p` and `sol_p` that are excessive.

Recommendation:
Change the declaration of initial `jlp_price` and `sol_price` as `u128` and you'll never need excessive vars.

INFORMATIONAL-3 | RESOLVED

Path prefix not necessary

File: lending/programs/water/src/lib.rs: 399,425,430
Description: The code uses the following constructions:
`anchor_lang::solana_program::program::invoke`, while the `anchor_lang` is already imported, so it can be omitted.

Recommendation:
omit `anchor_lang` in the declaration when it has already been imported. ie:
`solana_program::program::invoke`

INFORMATIONAL-4 | RESOLVED

Unsigned check for less than zero

File: lending/programs/water/src/lib.rs: 421,502,571,618,622
Description: The unsigned int variable could not be less than zero by the definition

Recommendation:
compare it only for the equality, i.e.: `if deposit_amount == 0 {`

INFORMATIONAL-5 | RESOLVED

Unnecessary variables

File: lending/programs/water/src/lib.rs: 712,713
Description: In the `borrow` function, an excessive variable is introduced: `utilization_rate`.

Recommendation:
Either make `get_utilization_rate` to return Result<u64>, or use `vault_data.max_utilization_rate` as u128 in the comparison

INFORMATIONAL-6 | RESOLVED

Multiple calls to the same field

File: lending/programs/water/src/lib.rs: (numerous places)
Description: We identified that through the entire code, there are innumerable places when the same field is accessed multiple times (i.e., "ctx.accounts.lending_account") without creating a local variable or even a local variable but in the same context, the field is being accessed for the read again

Recommendation:
use local variables when there are multiple reads from the same struct field

CONTRACTS		
Transaction Ordering	Check for any logic that could be exploited by manipulating the order of transactions	PASS
Timestamp Reliance	Identify areas where code depends on timestamps, and consider potential vulnerabilities related to time manipulation	PASS
Integer Calculations	Scrutinize integer operations to prevent overflow or underflow errors	PASS
Rounding Errors	Look for places where rounding might lead to unintended consequences	PASS
Denial of Service (DoS)	Examine if the code has weaknesses that could be exploited to disrupt service or cause unexpected resource consumption	PASS
Access Controls	Verify that access to sensitive functions and data is properly restricted and that permissions are implemented correctly	PASS
Centralization	Assess whether any single entity or component has excessive control, creating a potential point of failure	FAIL
Business Logic Integrity	Ensure that the code accurately reflects the intended business rules and specifications	PASS
Code Duplication	Minimize redundant code to reduce the attack surface and improve maintainability	PASS
Gas Usage	Analyze gas consumption patterns to predict potential DoS scenarios or unexpected cost implications	PASS
Token Minting	If applicable, verify that token minting mechanisms are secure and cannot be abused	PASS
Call Return Values	Ensure that return values from external calls are always checked for errors	PASS
Capability Flow	Carefully track how capabilities are transferred and used to prevent unauthorized actions	PASS
Witness Type Usage	If applicable, verify that witness types are used correctly to enforce constraints	PASS

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As part of our work assisting Vaultka in verifying the correctness of their contract/s code, our team was responsible for writing unit tests.

Running vaultka/tests/unit.rs (target/debug/deps/unit-b8f5285d3ca48bf2)
running 28 tests
test tests::test_set_jito_price ... **ok**
test tests::test_set_jito_price_below_slippage ... **ok**
test tests::test_set_fee_receivers ... **ok**
test tests::test_set_jito_price_not_keeper ... **ok**
test tests::test_set_fee_receivers_not_admin ... **ok**
test tests::test_set_jito_price_over_slippage ... **ok**
test tests::test_set_admin_not_admin ... **ok**
test tests::test_initialize ... **ok**
test tests::test_set_admin ... **ok**
test tests::test_set_fee_receivers_fees_over_5000 ... **ok**
test tests::test_set_jito_price_zero ... **ok**
test tests::test_set_slippage_control ... **ok**
test tests::test_set_slippage_control_not_admin ... **ok**
test tests::test_set_strategy_params ... **ok**
test
tests::test_set_strategy_params_fixed_fee_split_greater_than_950_and_mfee_percent_greater_th
an_5000 ... **ok**
test tests::test_set_strategy_params_dtv_limit_greater_than_990000000 ... **ok**
test tests::test_set_strategy_params_dtv_limit_less_than_500000000 ... **ok**
test tests::test_set_strategy_params_jito_mint_zero ... **ok**
test tests::test_set_strategy_params_jito_sol_feed_empty ... **ok**
test tests::test_set_strategy_params_keeper_ata_is_zero ... **ok**
test tests::test_set_strategy_params_keeper_fees_less_than_100000 ... **ok**
test tests::test_set_strategy_params_keeper_fees_greater_than_10000000 ... **ok**
test tests::test_set_strategy_params_leverage_less_than_1 ... **ok**
test tests::test_set_strategy_params_leverage_greater_than_16 ... **ok**
test tests::test_set_strategy_params_maturity_time_less_than_zero ... **ok**
test tests::test_set_strategy_params_not_admin ... **ok**
test tests::test_set_strategy_params_sol_jito_feed_empty ... **ok**
test tests::test_set_strategy_params_keeper_is_zero ... **ok**
test result: **ok**. 28 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.01s

```
Running water/tests/unit.rs (target/debug/deps/unit-7595b2fd4e8f9ecb)
running 34 tests
test tests::test_borrow_signer_not_whitelisted ... ok
test tests::test_disable_whitelisted_user_caller_not_owner ... ok
test tests::test_deposit_zero_deposit_amount ... ok
test tests::test_borrow_utilization_rate_exceeded ... ok
test tests::test_borrow_zero ... ok
test tests::test_borrow_sol_balance_exceeded ... ok
test tests::test_deposit ... ok
test tests::test_borrow ... ok
test tests::test_deposit_total_shares_greater_than_zero ... ok
test tests::test_disable_whitelisted_user ... ok
test tests::test_gift_sol_to_vault ... ok
test tests::test_gift_sol_to_vault_caller_not_keeper ... ok
test tests::test_initialize_vault ... ok
test tests::test_repay ... ok
test tests::test_repay_amount_zero ... ok
test tests::test_repay_bad_debt ... ok
test tests::test_repay_borrower_not_whitelisted ... ok
test tests::test_set_fee_params_caller_not_owner ... ok
test tests::test_repay_debt_value_zero ... ok
test tests::test_set_fee_params_withdraw_fee_greater_than_5000 ... ok
test tests::test_set_fee_params ... ok
test tests::test_set_max_util_rate_caller_not_owner ... ok
test tests::test_set_fee_params_withdraw_receiver_is_zeroed ... ok
test tests::test_set_max_util_rate_value_exceeds_990000000 ... ok
test tests::test_set_vault_keeper ... ok
test tests::test_set_vault_keeper_caller_not_owner ... ok
test tests::test_set_whitelisted_user_caller_not_owner ... ok
test tests::test_set_whitelisted_user ... ok
test tests::test_set_max_util_rate ... ok
test tests::test_set_vault_keeper_keeper_is_zeroes ... ok
test tests::test_withdraw ... ok
test tests::test_withdraw_above_maximum ... ok
test tests::test_withdraw_zero ... ok
test tests::test_withdraw_exceeds_sol_balance ... ok
test result: ok. 34 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s
```

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

FILE	% STMTS	% BRANCH	% FUNCS	% LINES
vaultka/src/lib.rs	65	60	65	65
water/src/lib.rs	100	100	100	98
All Files	**82.5**	**80**	**82.5**	**81.5**

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.

zokyo