RD AUDITORS

Stabull Smart Contract, Code Review and Security Analysis Report

Customer: Stabull
Prepared on: 31st July 2024
Platform: Ethereum
Language: Solidity

rdauditors.com

Table of Contents

Disclaimer

This document may contain confidential information about its systems and intellectual property of the customer as well as information about potential vulnerabilities and methods of their exploitation.

The report containing confidential information can be used internally by the customer or it can be disclosed publicly after all vulnerabilities are fixed - upon the decision of the customer.

Document

Name	Smart Contract Code Review and Security Analysis Report of Stabull
Platform	Ethereum / Solidity
File 1	StakingFactory.sol
MD5 hash	f971209700fb37a4b1ef6121da57196b
SHA256 hash	1ca6096bd91d717ac11ca9d9800918c39b8129ea2f75eba0a80b08ac9a642bdb
File 2	StakingPool.sol
MD5 hash	e2a2349989f5ea6971245f092249818f
SHA256 hash	35d3ee2f9138c6b878d6162d8df972fdd461ace798a3d327e76b8e1f335f86d1
Date	31/07/2024

Introduction

RD Auditors (Consultant) were contracted by Stabull (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report represents the findings of the security assessment of the customer`s smart contract and its code review conducted between 12th - 31st July 2024.

This contract consists of two files.

Project Scope

The scope of the project is a smart contract. We have scanned this smart contract for commonly known and more specific vulnerabilities, below are those considered (the full list includes but is not limited to):

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Byte array vulnerabilities
- Style guide violation
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Unchecked external call - Unchecked math
- Unsafe type inference
- Implicit visibility level

Executive Summary

According to the assessment, the customer's solidity smart contract is now **Well-Secured.**

Automated checks are with smartDec, Mythril, Slither and remix IDE. All issues were performed by our team, which included the analysis of code functionality, the manual audit found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the audit overview section. The general overview is presented in the AS-IS section and all issues found are located in the audit overview section.

We found the following;

Total Issues	0
Critical	0
High	0
Medium	0
Low	0
Very Low	0

Code Quality

The libraries within this smart contract are part of a logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned to a specific address and its properties/methods can be reused many times by other contracts.

The Stabull team has not provided scenario and unit test scripts, which helped to determine the integrity of the code in an automated way.

Documentation

The hash of that file is mentioned in the table. As mentioned above, It's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic.

Comments are very helpful in understanding the overall architecture of the protocol. It also provides a clear overview of the system components, including helpful details, like the lifetime of the background script.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure. Those were based on well known industry standard open source projects and even core code blocks that are written well and systematically.

AS-IS Overview

StakingFactory.sol

File And Function Level Report

File 1: StakingFactory.sol

Contract: StakingFactory

Observation: Passed

Test Report: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	add	write	Passed	All Passed	No Issue	Passed
2	set	write	Passed	All Passed	No Issue	Passed
3	setRewardTokenPerBlock	write	Passed	All Passed	No Issue	Passed
4	setFundSource	write	Passed	All Passed	No Issue	Passed
5	inCaseTokensGetStuck	write	Passed	All Passed	No Issue	Passed
6	deposit	write	Passed	All Passed	No Issue	Passed
7	withdrawAll	write	Passed	All Passed	No Issue	Passed
8	claimReward	write	Passed	All Passed	No Issue	Passed
9	claimRewardMultiple	write	Passed	All Passed	No Issue	Passed
10	emergencyWithdraw	write	Passed	All Passed	No Issue	Passed
11	pendingRewardToken	write	Passed	All Passed	No Issue	Passed

12	stakedTokens Amount	write	Passed	All Passed	No Issue	Passed
13	withdraw	write	Passed	All Passed	No Issue	Passed
14	updatePool	write	Passed	All Passed	No Issue	Passed
15	massUpdateP ools	write	Passed	All Passed	No Issue	Passed
16	getMultiplier	read	Passed	All Passed	No Issue	Passed
17	safeRewardTo kenTransfer	internal	Passed	All Passed	No Issue	Passed
18	getRewardTo ken	internal	Passed	All Passed	No Issue	Passed

StakingPool.sol

File And Function Level Report

File 1: StakingPool.sol

Contract: StakingPool

Observation: Passed

Test Report: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	deposit	write	Passed	All Passed	No Issue	Passed
2	withdraw	write	Passed	All Passed	No Issue	Passed
3	unpause	write	Passed	All Passed	No Issue	Passed
4	setEntranceFe eFactor	write	Passed	All Passed	No Issue	Passed

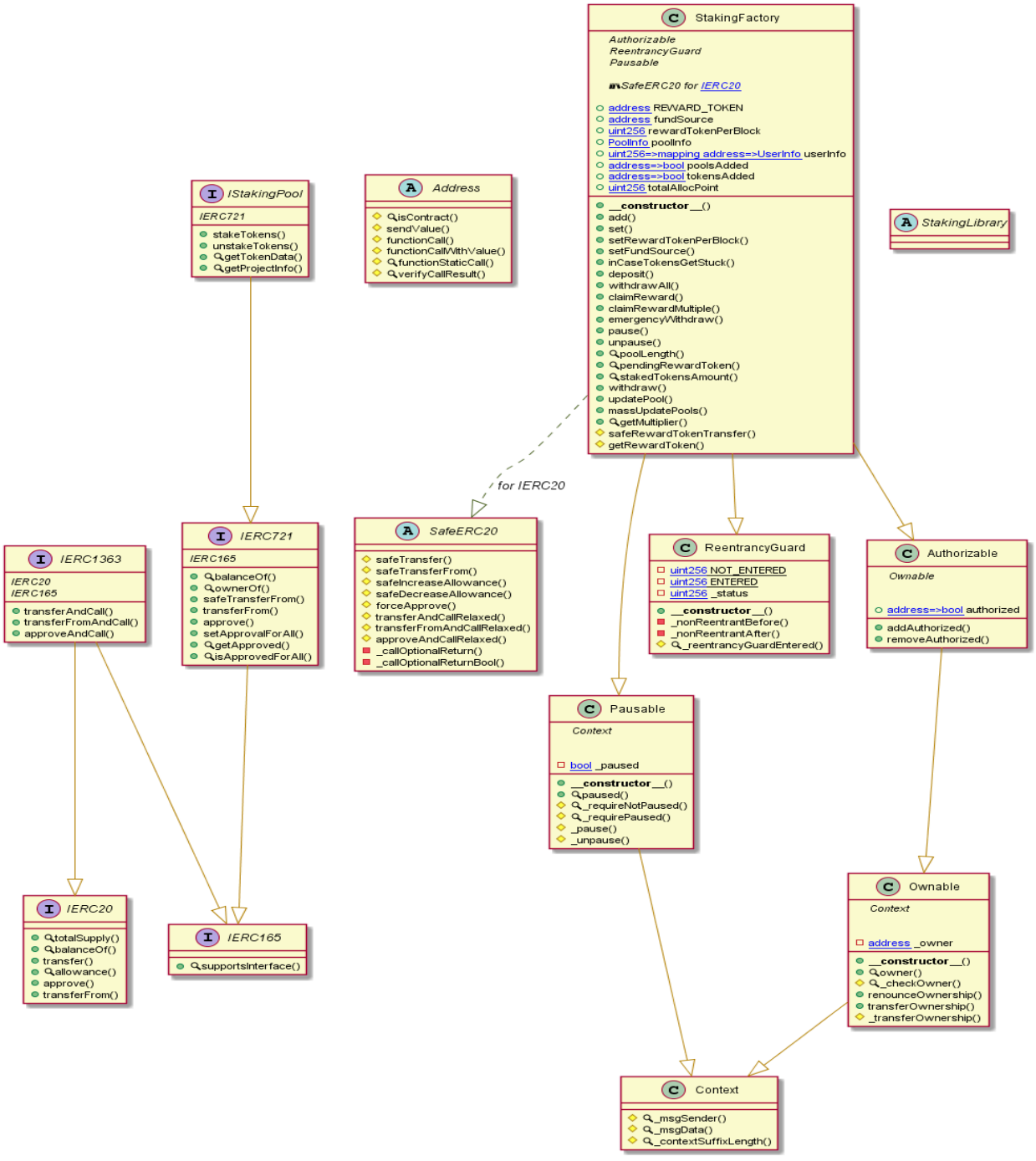
5	setExitFeeFactor	write	Passed	All Passed	No Issue	Passed
6	setGov	write	Passed	All Passed	No Issue	Passed
7	setFeeReceiver	write	Passed	All Passed	No Issue	Passed
8	inCaseTokensGetStuck	write	Passed	All Passed	No Issue	Passed
9	setWithdrawFeePeriod	write	Passed	All Passed	No Issue	Passed

Code Flow Diagram - StakingFactory

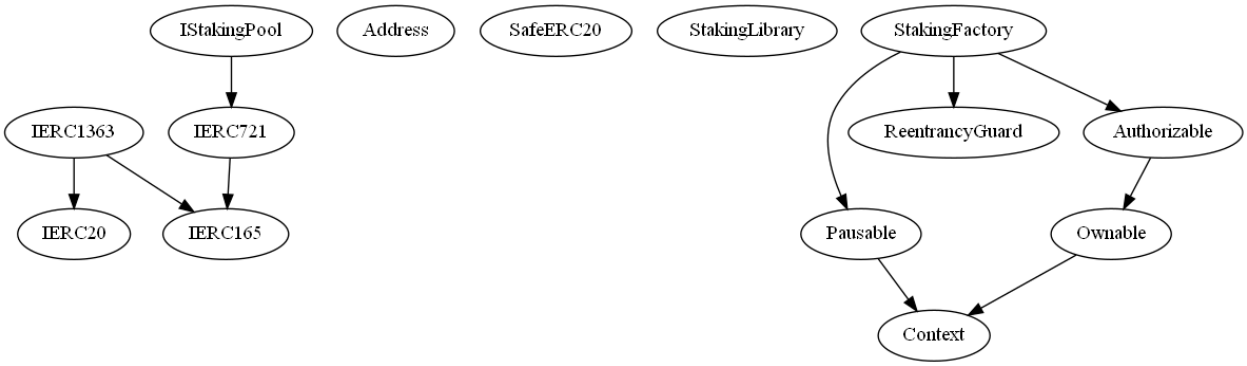
Please click on the link below to view the flow diagram:

https://drive.google.com/file/d/1U8QT-7zJA8hYxTY8IevzKIUDnXSnAV_R/view?usp=sharing

UML Diagram - StakingFactory



Inheritance Diagram - StakingFactory



Solidity Static Analysis

StakingFactory.sol

Check-effects-interaction:
Potential violation of Checks-Effects-Interaction pattern in StakingFactory.deposit(uint256,uint256): Could potentially lead to re-entrancy vulnerability.
Note: Modifiers are currently not considered by this static analysis.
Pos: 393:4:

For loop over dynamic array:
Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Additionally, using unbounded loops incurs a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.
Pos: 464:8:

For loop over dynamic array:
Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Additionally, using unbounded loops incurs a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.
Pos: 466:12:

Constant/View/Pure functions:
StakingFactory.emergencyWithdraw(uint256) : Potentially should be constant/view/pure but is not. Note: Modifiers are currently not considered by this static analysis.
Pos: 491:4:

Similar variable names:
StakingFactory.claimRewardMultiple(uint256[]) : Variables have very similar names "_pid" and

"_pids". Note: Modifiers are currently not considered by this static analysis.
Pos: 467:23:

No return:
StakingFactory.stakedTokensAmount(uint256,address): Defines a return type but never explicitly returns a value.
Pos: 543:4:

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
Pos: 183:8:

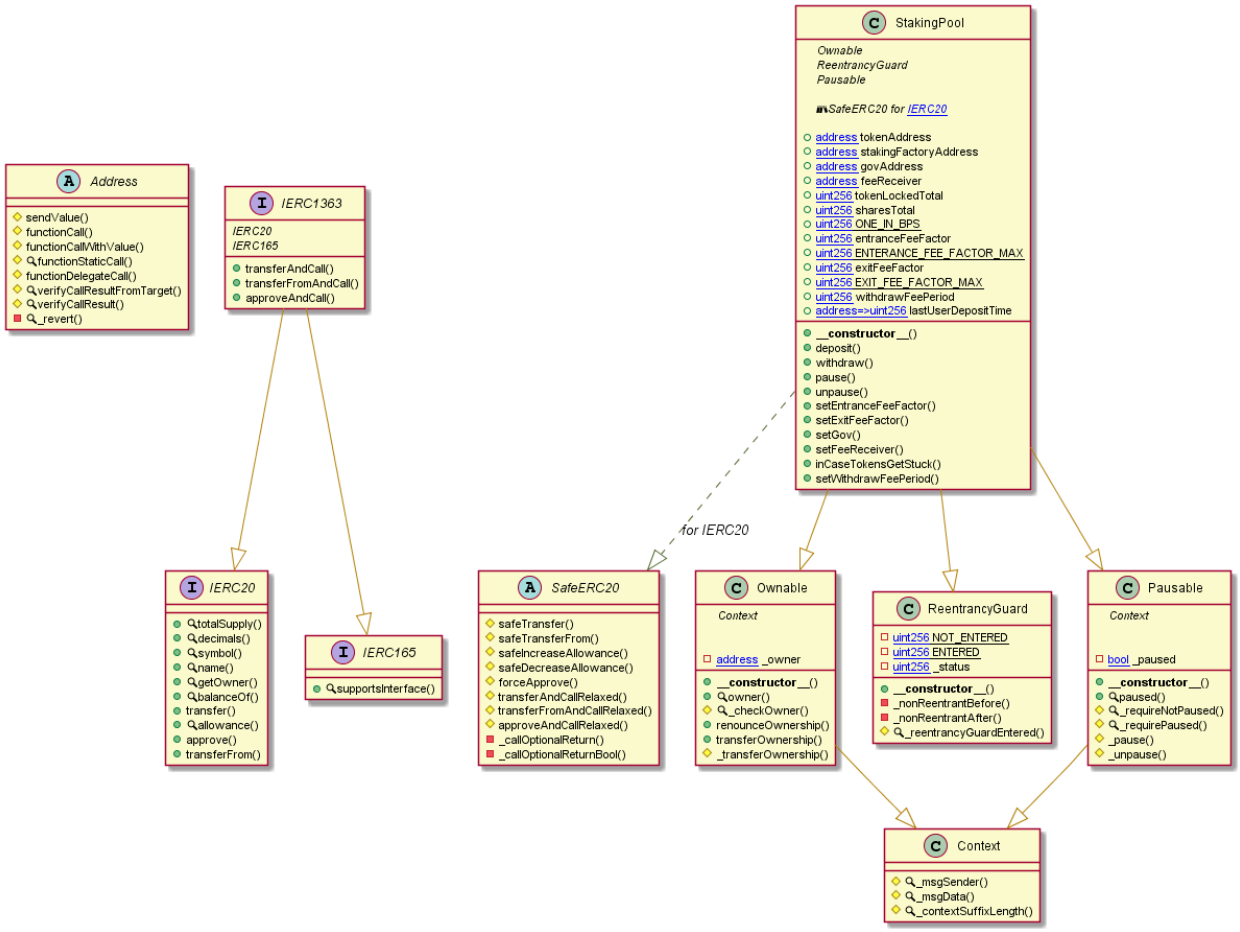
Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
Pos: 183:8:

Code Flow Diagram - StakingPool

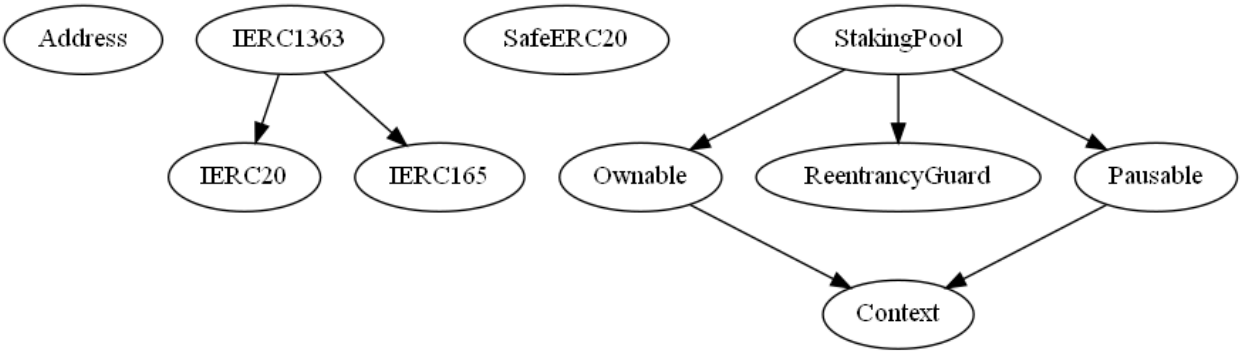
Please click on the link below to view the flow diagram:

https://drive.google.com/file/d/1icL79fxXy6AM640-u4ONwGIatRGMyCF1/view?usp=sharing

UML Diagram - StakingPool



Inheritance Diagram - StakingPool



Code Flow Diagram - Slither Results Log

StakingPool.sol

```
INFO:Detectors:
StakingPool.withdraw(address,uint256) (StakingPool.sol#874-916)
uses timestamp for comparisons
        Dangerous comparisons:
        - lastUserDepositTime[_userAddress] + withdrawFeePeriod >=
block.timestamp (StakingPool.sol#902-903)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#block
-timestamp
INFO:Detectors:
Pragma version0.8.20 (StakingPool.sol#3) necessitates a version too
recent to be trusted. Consider deploying with 0.8.18.
solc-0.8.20 is not recommended for deployment
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#incor
rect-versions-of-solidity
INFO:Detectors:
Parameter StakingPool.deposit(address,uint256)._userAddress
(StakingPool.sol#840) is not in mixedCase
Parameter StakingPool.deposit(address,uint256)._tokenAmt
(StakingPool.sol#841) is not in mixedCase
Parameter StakingPool.withdraw(address,uint256)._userAddress
(StakingPool.sol#875) is not in mixedCase
Parameter StakingPool.withdraw(address,uint256)._tokenAmt
(StakingPool.sol#876) is not in mixedCase
Parameter
StakingPool.setEntranceFeeFactor(uint256)._entranceFeeFactor
(StakingPool.sol#934) is not in mixedCase
Parameter StakingPool.setExitFeeFactor(uint256)._exitFeeFactor
(StakingPool.sol#944) is not in mixedCase
Parameter StakingPool.setGov(address)._govAddress
(StakingPool.sol#954) is not in mixedCase
Parameter StakingPool.setFeeReceiver(address)._feeReceiver
(StakingPool.sol#963) is not in mixedCase
Parameter
StakingPool.inCaseTokensGetStuck(address,uint256,address)._token
(StakingPool.sol#975) is not in mixedCase
Parameter
```

```
StakingPool.inCaseTokensGetStuck(address,uint256,address)._amount
(StakingPool.sol#976) is not in mixedCase
Parameter
StakingPool.inCaseTokensGetStuck(address,uint256,address)._to
(StakingPool.sol#977) is not in mixedCase
Parameter
StakingPool.setWithdrawFeePeriod(uint256)._withdrawFeePeriod
(StakingPool.sol#994) is not in mixedCase
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#confo
rmance-to-solidity-naming-conventions
INFO:Detectors:
StakingPool.stakingFactoryAddress (StakingPool.sol#760) should be
immutable
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#state
-variables-that-could-be-declared-immutable
INFO:Slither:StakingPool.sol analyzed (10 contracts with 93
detectors), 41 result(s) found
```

Solidity Static Analysis

StakingPool.sol

Check-effects-interaction:
Potential violation of Checks-Effects-Interaction pattern in StakingPool.withdraw(address,uint256): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.
Pos: 140:4:

Block timestamp:
Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.
Pos: 130:44:

Block timestamp:
Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.
Pos: 169:12:

Constant/View/Pure functions:
StakingPool.inCaseTokensGetStuck(address,uint256,address) : Potentially should be constant/view/pure but is not. Note: Modifiers are currently not considered by this static analysis.
Pos: 240:4:

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.
Pos: 123:28:

Data truncated:
Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.
Pos: 171:32:

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to lost tokens etc.
High	High level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g. public access to crucial functions.
Medium	Medium level vulnerabilities are important to fix; however, they cannot lead to lost tokens.
Low	Low level vulnerabilities are most related to outdated, unused etc. These code snippets cannot have a significant impact on execution.
Lowest Code Style/ Best Practice	Lowest level vulnerabilities, code style violations and information statements cannot affect smart contract execution and can be ignored.

Audit Findings

Critical:

No critical severity vulnerabilities were found.

High:

No high severity vulnerabilities were found.

Medium:

No medium severity vulnerabilities were found.

Low: (Resolved)*

1. Check “amount” for 0 before transferring the value in the “emergencyWithdraw” function.
2. Check “sharesRemoved” for 0 before transferring the value in “Withdraw” function.

Very Low:

No very low severity vulnerabilities were found.

*Status: The team has resolved these issues in the following commit: https://github.com/stabull/v1-staking/commit/5e470f3c891d90c8320b205704b2b97ebe6a829c

Conclusion

We were given a contract file and have used all possible tests based on the given object. So it is ready to go for production. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

The security state of the reviewed contract is “**Well-Secured**”.

Note For Contract Users

Technical auditing does not guarantee the project's ethical side.

Please do your due diligence before investing. Our audit report is never an investment advice.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyse the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinised by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

RD Auditors Disclaimer

The smart contracts given for audit have been analysed in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Because the total number of test cases are unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can’t guarantee explicit security of the audited smart contracts.

Email: info@rdauditors.com

Website: www.rdauditors.com