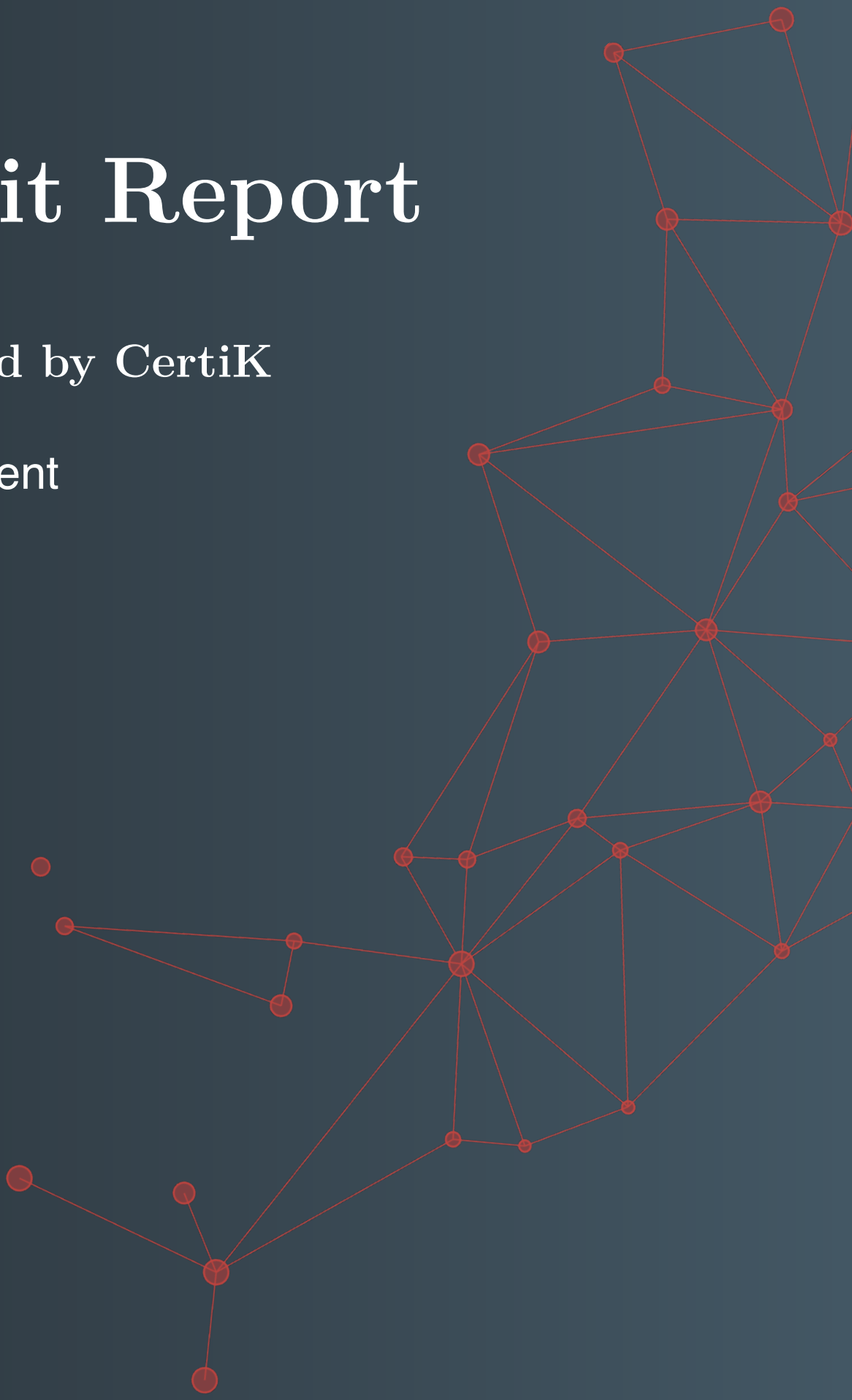
CERTIK

Audit Report

Produced by CertiK

for Quoxent



Contents

Disclaimer

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Verification Services Agreement between CertiK and Quoxent (the "Company"), or the scope of services/verification, and terms and conditions provided to the Company in connection with the verification (collectively, the "Agreement"). This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes without CertiK's prior written consent.

About CertiK

CertiK is a technology-led blockchain security company founded by Computer Science professors from Yale University and Columbia University built to prove the security and correctness of smart contracts and blockchain protocols.

CertiK, in partnership with grants from IBM and the Ethereum Foundation, CertiK's mission of every audit is to apply different approaches and detection methods, ranging from manual, static, and dynamic analysis, to ensure that projects are checked against known attacks and potential vulnerabilities. CertiK leverages a team of seasoned engineers and security auditors to apply testing methodologies and assessments to each project, in turn creating a more secure and robust software system.

CertiK has served more than 100 clients with high quality auditing and consulting services, ranging from stablecoins such as Binance's BGBP and Paxos Gold to decentralized oracles such as Band Protocol and Tellor. CertiK customizes its engineering tool kits, while applying cutting-edge research on smart contracts, for each client on its project to offer a high quality deliverable. For more information: https://certik.io.

Executive Summary

This report has been prepared for **Quoxent** to discover issues and vulnerabilities in the source code of their **QUO ERC-20 Smart Contract** as well as any contract dependencies that were not part of an officially recognized library. A comprehensive examination has been performed, utilizing Dynamic Analysis, Static Analysis, and Manual Review techniques.

The auditing process pays special attention to the following considerations:

- Testing the smart contracts against both common and uncommon attack vectors.
- Assessing the codebase to ensure compliance with current best practices and industry standards.
- Ensuring contract logic meets the specifications and intentions of the client.
- Cross referencing contract structure and implementation against similar smart contracts produced by industry leaders.
- Thorough line-by-line manual review of the entire codebase by industry experts.

Testing Summary

SECURITY LEVEL

Smart Contract Audit

This report has been prepared as a product of the Smart Contract Audit request by Quoxent.

This audit was conducted to discover issues and vulnerabilities in the source code of Quoxent's QUO ERC-20 Smart Contract.

TYPE	Smart Contract
SOURCE CODE	https://rinkeby.etherscan.io/address/0x20138894ac3288a16b53515752ec068c21e695d3#code
PLATFORM	EVM
LANGUAGE	Solidity
REQUEST DATE	July 24, 2020
DELIVERY DATE	Aug 20,, 2020
METHODS	A comprehensive examination has been performed using Dynamic Analysis, Static Analysis, and Manual Review.

Review Notes

Introduction

CertiK team was contracted by the Quoxent team to audit the design and implementation of their QUO token smart contract and its compliance with the EIPs it is meant to implement.

The audited source code link is:

- Token Source Code: https://rinkeby.etherscan.io/address/0x20138894ac3288a16b53515752ec068c21e695d3#code

The goal of this audit was to review the Solidity implementation for its business model, study potential security vulnerabilities, its general design and architecture, and uncover bugs that could compromise the software in production.

The findings of the initial audit have been conveyed to the team behind the contract implementations and the source code is expected to be re-evaluated before another round of auditing has been carried out.

Documentation

The sources of truth regarding the operation of the contracts in scope were minimal although the token fulfilled a simple use case we were able to fully assimilate. To help aid our understanding of each contract's functionality we referred to in-line comments and naming conventions.

These were considered the specification, and when discrepancies arose with the actual code behaviour, we consulted with the Quoxent team or reported an issue.

Summary

The codebase of the project is a typical EIP20 implementation with additional support for a locking mechanism, a staking mechanism and a set of administrators.

Certain optimization steps that we pinpointed in the source code mostly referred to coding standards and inefficiencies, however **1 minor and 1 critical severity vulnerability were identified during our audit**. Quoxent's team urgently alleviated the two findings, along with almost all of the informational ones.

The codebase of the project strictly adheres to the standards and interfaces imposed by the OpenZeppelin open-source libraries and as such its typical ERC-20 functions **can be deemed to be of high security and quality, however the custom functionality built on top of it possessed flaws** we identified. Quoxent's team considered our references and opted to change their codebase to match our recommendations.

Recommendations

Overall, the codebase of the contracts should be refactored to assimilate the findings of this report, enforce linters and / or coding styles as well as correct any spelling errors and mistakes that appear throughout the code **to achieve a high standard of code quality and security**.

Findings

Exhibit 1

TITLE	TYPE	SEVERITY	LOCATION
Unlocked Compiler Version	Language Specific	Informational	All “pragma” statements

[INFORMATIONAL] Description:

The smart contract “pragma” statements regarding the compiler version indicate that version 0.6.2 or higher should be utilized.

Recommendations:

We advise that the compiler version is locked at version 0.6.2 or whichever Solidity version higher than that satisfies the requirements of the codebase as an unlocked compiler version can lead to discrepancies between compilations of the same source code due to compiler bugs and differences.

Exhibit 2

TITLE	TYPE	SEVERITY	LOCATION
Visibility Specifiers	Language Specific	Informational	Quoxent: L734, L735, L743

[INFORMATIONAL] Description:

The specified lines of code contain contract-level variable declarations with no form of visibility specifier defined.

Recommendations:

We advise that a proper visibility specifier is provided for those variables to ensure that they are securely exposed, if at all, to other contracts.

Exhibit 3

TITLE	TYPE	SEVERITY	LOCATION
Inefficient Greater-Than Comparison w/ Zero	Optimization	Informational	Quoxent: L802, L803

[INFORMATIONAL] Description:

The lines above conduct a greater-than “>” comparison between unsigned integers and the value literal “0”.

Recommendations:

As unsigned integers are restricted to the positive range, it is possible to convert this check to an inequality “!=” reducing the gas cost of the functions.

Exhibit 4

TITLE	TYPE	SEVERITY	LOCATION
Struct Optimization	Ineffectual Code	Informational	Quoxent: L737 - L741

[INFORMATIONAL] Description:

The “Metadata” struct is meant to retain the number of stakes a user has conducted, the timestamps that each stake occurred as well as whether the address is locked and unable to transfer funds.

Recommendations:

Within Solidity, every struct is split into 32-byte segments and subsequently stored. As each read and write operation on the EVM costs gas and subsequently money, it is more optimal to optimize data structures such as structs to utilize the least 32-byte slots possible.

In the case of “Metadata”, it is possible to reduce the “uint256” deposits variable to a “uint248” to allow room for the 8-bit “bool” variable lock. This would reduce the number of slots a “Metadata” struct requires from 3 to 2, significantly reducing gas cost across the board.

Exhibit 5

TITLE	TYPE	SEVERITY	LOCATION
Unusual Naming Conventions	Coding Style	Informational	Quoxent: L745, L751

[INFORMATIONAL] Description:

The function modifiers defined in the contract are prefixed with an underscore whilst this is not what is suggested by the Solidity styling guide.

Recommendations:

We advise that the underscore be omitted to conform to a uniform styling guide.

Exhibit 6

TITLE	TYPE	SEVERITY	LOCATION
Dynamic Variable to Constant	Coding Style	Informational	Quoxent: L760, L805

[INFORMATIONAL] Description:

The total supply of the token is calculated by offsetting 360 million by 18 decimal places, the number of decimals supported by the token. Additionally, the time offset for staking is simply hard-coded.

Recommendations:

As the token retains the default number of decimals imposed by ERC20 which is 18, we advise that the total supply is instead set as a “constant” in a similar fashion to “DEPOSIT” and “REWARD”. With regard to the staking offset, we simply advise that it directly be stored to a “constant” variable.

Exhibit 7

TITLE	TYPE	SEVERITY	LOCATION
Contextual Message Sender Invocations	Optimization	Informational	Multiple Places

[INFORMATIONAL] Description:

The function "_msgSender()" is meant to retrieve the current function invocator. This function costs gas to execute.

Recommendations:

We advise that the result of "_msgSender()" is instead stored to an in-memory variable that is subsequently utilized as we have observed multiple redundant invocations to "_msgSender()" within the codebase.

Exhibit 8

TITLE	TYPE	SEVERITY	LOCATION
Staking System Exploitation	Logical	Critical	Quoxent: L787 - L797, L799 - L813

[CRITICAL] Description:

The current staking implementation is possible to be exploited for a user to mint an unlimited amount of tokens as the checks imposed by the redemption are invalid.

As an example, a user would create an arbitrary number of deposits and then redeem the first one he made.

This would cause the “timeStamp” of the deposit to be zeroed out but would still allow the user to set the redemption index in the “redeem” function to the already redeemed stake.

As long as they retain the total deposits above 1, they will be able to redeem an unlimited number of tokens.

Please see the accompanying exploit contract which was provided along with the report for a reproducible example of the exploit.

Recommendations:

A solution to this would be to either ensure that the timestamp of the index is not zero, which it can’t be under any logical circumstance, or to only allow redemption of the oldest stake in the array.

Exhibit 9

TITLE	TYPE	SEVERITY	LOCATION
Redundant Mapping Lookups	Optimization	Informational	Quoxent: L787 - L813

[INFORMATIONAL] Description:

The result of the "_metadata[_msgSender()]" lookup can be stored into a variable instead of dynamically evaluating it on each access.

Recommendations:

We advise that the result is indeed stored to an in-memory variable as this will significantly reduce the gas cost of the functions.

Exhibit 10

TITLE	TYPE	SEVERITY	LOCATION
Inexistent Error Messages	Coding Style	Informational	Quoxent: L746, L747, L753, L789, L801, L802, L839

[INFORMATIONAL] Description:

It is a generally accepted coding practice to add error messages to all types of “require” invocations to aid in the debugging of the application.

Recommendations:

We advise that proper error messages are provided for these statements.

Exhibit 11

TITLE	TYPE	SEVERITY	LOCATION
Administrator Management	Logical	Minor	Quoxent: L832 - L847

[MINOR] Description:

The current implementation of the administrator management system is flawed in the sense that it is possible for an administrator to overcome all other administrators with relative ease.

Recommendations:

Judging by the way the functions relating to administrators are worded, the presence of a “superadmin” which is currently the first administrator of the contract exists.

As a result, we advise that the system is revamped to utilize a single superadmin, the creator of the contract, and a set of delegated administrators that are meant to interface with users and be able to block malicious users using the locking functions.