HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: GotBit
Date: July 25th, 2022

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Document

Name	Smart Contract Code Review and Security Analysis Report for GotBit
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	ERC20 token; Vesting
Platform	EVM
Language	Solidity
Methods	Manual Review, Automated Review, Architecture review
Website	no
Timeline	15.07.2022 - 25.07.2022
Changelog	25.07.2022 - Initial Review

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Table of contents

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Introduction

Hacken OÜ (Consultant) was contracted by GotBit (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report presents the findings of the security assessment of the Customer's smart contracts.

Scope

The scope of the project is smart contracts in the repository:

Initial review scope

Repository:
https://github.com/GotBit/UKAN
Technical Documentation:
Type: Technical description
https://github.com/GotBit/UKAN

Type: Functional requirements
https://github.com/GotBit/UKAN

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/contracts/Vesting.sol
SHA3: 4da2d47faf6e4e4964e9779bce59f98dddbe04348169dc529ae911961728828b

File: ./contracts/contracts/units/Token.sol
SHA3: f9f807999ecd8d805364e4e5dd9705873cf38574e82db203cc76252a112ce8c9

Second review scope

Repository:
https://github.com/GotBit/UKAN
Commit:
1383886880bdf9b3d4c2c2ee3599a561932b2f2d
Technical Documentation:
Type: Technical description
https://github.com/GotBit/UKAN

Type: Functional requirements
https://github.com/GotBit/UKAN

Integration and Unit Tests: Yes
Contracts:
File: ./contracts/contracts/Vesting.sol
SHA3: 4da2d47faf6e4e4964e9779bce59f98dddbe04348169dc529ae911961728828b

File: ./contracts/contracts/units/Token.sol
SHA3: 78158f13fb482bb99a9ea55888be6e24b4c99e3034dbf35cfc18029f9f048333

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to assets loss or data manipulations.
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 assets loss or data manipulations.
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets that cannot have a significant impact on execution.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Executive Summary

The Score measurements details can be found in the corresponding section of the methodology.

Documentation quality

The total Documentation Quality score is **7** out of **10**. Technical and functional requirements are provided. Tokenomic is not provided.

Code quality

The total CodeQuality score is **10** out of **10**. Deployment and basic user interactions are covered with tests.

Architecture quality

The architecture quality score is **10** out of **10**.

Security score

As a result of the second audit, the code does not contain any issues. The security score is **10** out of **10**.

All found issues are displayed in the “Findings” section.

Summary

According to the assessment, the Customer's smart contract has the following score: **9.7**.

Hacken OÜ
Parda 4, Kesklinn, Tallinn,
10151 Harju Maakond, Eesti,
Kesklinna, Estonia
support@hacken.io

Checked Items

We have audited provided smart contracts for commonly known and more specific vulnerabilities. Here are some of the items that are considered:

Item	Type	Description	Status
Default Visibility	SWC-100 SWC-108	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	SWC-101	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed
Outdated Compiler Version	SWC-102	It is recommended to use a recent version of the Solidity compiler.	Passed
Floating Pragma	SWC-103	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed
Unchecked Call Return Value	SWC-104	The return value of a message call should be checked.	Not Relevant
Access Control & Authorization	CWE-284	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed
SELFDESTRUCT Instruction	SWC-106	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant
Check-Effect-Interaction	SWC-107	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed
Uninitialized Storage Pointer	SWC-109	Storage type should be set explicitly if the compiler version is < 0.5.0.	Not Relevant
Assert Violation	SWC-110	Properly functioning code should never reach a failing assert statement.	Passed
Deprecated Solidity Functions	SWC-111	Deprecated built-in functions should never be used.	Passed
Delegatecall to Untrusted Callee	SWC-112	Delegatecalls should only be allowed to trusted addresses.	Not Relevant
DoS (Denial of Service)	SWC-113 SWC-128	Execution of the code should never be blocked by a specific contract state unless it is required.	Passed
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed
Authorization through	SWC-115	tx.origin should not be used for authorization.	Passed

tx.origin			
Block values as a proxy for time	SWC-116	Block numbers should not be used for time calculations.	Passed
Signature Unique Id	SWC-117 SWC-121 SWC-122	Signed messages should always have a unique id. A transaction hash should not be used as a unique id.	Passed
Shadowing State Variable	SWC-119	State variables should not be shadowed.	Passed
Weak Sources of Randomness	SWC-120	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant
Incorrect Inheritance Order	SWC-125	When inheriting multiple contracts, especially if they have identical functions, a developer should carefully specify inheritance in the correct order.	Not Relevant
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	All external calls should be performed only to trusted addresses.	Not Relevant
Presence of unused variables	SWC-131	The code should not contain unused variables if this is not justified by design.	Passed
EIP standards violation	EIP	EIP standards should not be violated.	Not Relevant
Assets integrity	**Custom**	Funds are protected and cannot be withdrawn without proper permissions.	Passed
User Balances manipulation	**Custom**	Contract owners or any other third party should not be able to access funds belonging to users.	Passed
Data Consistency	**Custom**	Smart contract data should be consistent all over the data flow.	Passed
Flashloan Attack	**Custom**	When working with exchange rates, they should be received from a trusted source and not be vulnerable to short-term rate changes that can be achieved by using flash loans. Oracles should be used.	Not Relevant
Token Supply manipulation	**Custom**	Tokens can be minted only according to rules specified in a whitepaper or any other documentation provided by the Customer.	Not Relevant
Gas Limit and Loops	**Custom**	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There should not be any cases when execution fails due to the block gas limit.	Passed
Style guide violation	**Custom**	Style guides and best practices should be followed.	Passed
Requirements Compliance	**Custom**	The code should be compliant with the requirements provided by the Customer.	Passed

Environment Consistency	**Custom**	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Passed
Tests Coverage	**Custom**	The code should be covered with unit tests. Test coverage should be 100%, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Passed
Stable Imports	**Custom**	The code should not reference draft contracts, that may be changed in the future.	Passed

System Overview

UKAN is a mixed-purpose system with the following contracts:

- *Token* – a simple ERC-20 token that mints all initial supply to a deployer. Additional minting is not allowed.
- *Vesting* – a contract that creates vesting once for 1 or more wallets. During the deployment contract process, the user should specialize the distributing token. This contract allows add users for vesting schedules. The user can check how many tokens are available for claiming. The user can claim his reward if there are enough tokens on the Vesting contract, at any time.

Privileged roles

- The owner of the *Vesting* contract can add users for vesting schedules.

Risks

- **Total token supply** can not be verified due to the lack of requirements provided by the Customer.

Findings

Critical

No critical severity issues were found.

High

1. Calculation in different units

Token decimals are provided as a constructor parameter, though the volume of minted tokens is specified in Ethers (10^18).

Total supply can differ from expected when decimals differ from 18 points.

File: ./contracts/contracts/utils/Token.sol

Contract: Token

Recommendation: Use *amount * 10 ** decimals_* instead of *Ethers*

Status: Fixed (Revised commit: 1383886880bdf9b3d4c2c2ee3599a561932b2f2d)

Medium

No medium severity issues were found.

Low

1. No messages in require conditions

The require condition can be used to check for conditions and throw an exception if the condition is not met. It is possible to provide a message string for require. If a string argument for require is not provided, it will revert with empty error data, not even including the error selector.

Vesting contract constructor's require statement is missing error messages.

This can lead to harder tests and debugging processes.

File: ./contracts/contract/Vesting.sol

Contract: Vesting

Function: constructor

Recommendation: Add error messages to require conditions.

Status: Fixed (Revised commit: 1383886880bdf9b3d4c2c2ee3599a561932b2f2d)

2. Potential zero division

In the case of 'addUser' will receive zero value for 'newDuration' , 'current.duration' in 'available' function will be equal to zero.

This can lead to an error.

File: ./contracts/contract/Vesting.sol

Contract: Vesting

Function: available

Recommendation: Implement a zero division prevention check or add a require check statement for ‘duration’ in the constructor.

Status: Fixed (Revised commit: 1383886880bdf9b3d4c2c2ee3599a561932b2f2d)

3. **Potential zero division**

In the case of ‘addUser’ will receive zero value for ‘newDuration’ or ‘slicePeriod’ , ‘current.duration’ or ‘current.slicePeriod’ in the ‘available’ function will be equal to zero.

This can lead to an error.

File: ./contracts/contract/Vesting.sol

Contract: Vesting

Function: available

Recommendation: Implement a zero division prevention check or add a require check statement for ‘duration’ in the constructor.

Status: Fixed (Revised commit: 1383886880bdf9b3d4c2c2ee3599a561932b2f2d)

4. **Redundant mathematical operation**

The mathematical operation ‘comparison’ is redundant inside the ‘available’ function.

File: ./contracts/contract/Vesting.sol

Contract: Vesting

Function: available

Recommendation: Remove redundant mathematical operations. Use comparison once: block.timestamp > current.startTime + current.duration. If this comparison is true, then return current.amount - current.claimed; If false, continue function execution. This makes code more readable and reduces Gas consumption.

Status: Fixed (Revised commit: 1383886880bdf9b3d4c2c2ee3599a561932b2f2d)

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed by the best industry practices at the date of this report, with 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).

The audit makes no statements or warranties on the security of the code. It also cannot be considered a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements. 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 the security of smart contracts.

Technical Disclaimer

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