HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Corpus Ventures
Date: December 22nd, 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 Corpus Ventures
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	ERC20 token; Tokens Sale
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://corpus.ventures/
Changelog	01.12.2022 - Initial Review 22.12.2022 - Second 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 Corpus Ventures (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/corpus-ventures/tokenize.it-smart-contracts
Commit	5e9046b
Functional Requirements	Link
Technical Requirements	Link
Contracts	File: ./contracts/AllowList.sol SHA3: 4be83992b2fdbccf38917e0f3d2484ae0347c8ba4e5907a97a79ec0ca0ba1e67 File: ./contracts/ContinuousFundraising.sol SHA3: 591f5837b110329fc5efac63947c20a871a1c054d47b6f83565772e027d8a2e5 File: ./contracts/FeeSettings.sol SHA3: c04e00d15a274654044d4bf20e23cfe7e3934ec50470ef886940cc96a0cf4b70 File: ./contracts/PersonalInvite.sol SHA3: fcff711908bb5b4ad12861359421d4f5a4da3db9b346ac074b80a1c5faa69469 File: ./contracts/PersonalInviteFactory.sol SHA3: 451fcd2fd7ed39728fa4a3e64259b1e56c78ce65fcb4b4ce7808ce7cbed6943c File: ./contracts/Token.sol SHA3: 6e8e21afe976fad77dd6f088b742a76290366b0b13ce50828c6d075a7fca63aa

Second review scope

Repository	https://github.com/corpus-ventures/tokenize.it-smart-contracts
Commit	88f1908
Functional Requirements	Link
Technical Requirements	Link
Contracts	File: ./contracts/AllowList.sol SHA3: 24662e84be5c8ec7699284af79b3ea17f3f7896c977e724e78e2b9c31334046c File: ./contracts/ContinuousFundraising.sol SHA3: 50cee811b966a9b5ebc107f6e476354d87c8b099f62d3a6e8df894315ea4d66e File: ./contracts/FeeSettings.sol SHA3: 7a8d95c4848b3b0edd9dc0c01c53106c1514dcc1ce8b7040b8b63b8b456100da

	File: ./contracts/PersonalInvite.sol SHA3: f552c6258d955fc99e45017caf916ec0eaf90ebc14dabc5b155e0d470a517ba6 File: ./contracts/PersonalInviteFactory.sol SHA3: b118fac716687bd4e1c61dee2fde6e4d67fc0786616b8d141b72eab5eeacc498 File: ./contracts/Token.sol SHA3: 2c2a92bd7affd46b90383a95832cec8b21efa12552cb902c92cd5aed8fda6d63

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation by external or internal actors.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation by external or internal actors.
Medium	Medium vulnerabilities are usually limited to state manipulations but cannot lead to assets loss. Major deviations from best practices are also in this category.
Low	Low vulnerabilities are related to outdated and unused code or minor Gas optimization. These issues won't have a significant impact on code execution but affect the code quality

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

Executive Summary

The score measurement details can be found in the corresponding section of the scoring methodology.

Documentation quality

The total Documentation Quality score is **10** out of **10**.

- Functional requirements are provided.
- Technical description is provided.

Code quality

The total Code Quality score is **10** out of **10**.

- The development environment is configured.
- The official Solidity style guide is followed.

Test coverage

Test coverage of the project is **96.42%** (branch coverage).

- Deployment and basic user interactions are covered with tests.
- Positive and negative cases are covered with tests.

Security score

As a result of the audit, the code contains **no** 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: **10**.

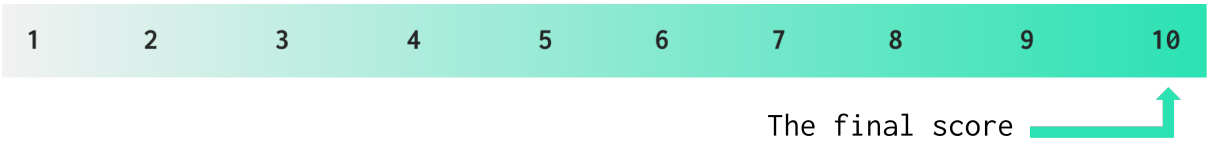

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
01 December 2022	4	0	0	0
22 December 2022	0	0	0	0

Checked Items

We have audited the Customers' smart contracts for commonly known and more specific vulnerabilities. Here are some items 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.	Not Relevant
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
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 required.	Passed
Race Conditions	SWC-114	Race Conditions and Transactions Order Dependency should not be possible.	Passed

Authorization through tx.origin	SWC-115	tx.origin should not be used for authorization.	Passed
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 EIP-155 EIP-712	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifiers should always be used. All parameters from the signature should be used in signer recovery. EIP-712 should be followed during a signer verification.	Not Relevant
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.	Passed
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.	Passed
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.	Not Relevant
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.	Passed

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
Secure Oracles Usage	**Custom**	The code should have the ability to pause specific data feeds that it relies on. This should be done to protect a contract from compromised oracles.	Not Relevant
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, which may be changed in the future.	Passed

System Overview

Tokenize.it is a protocol for tokenize real-life company shares with following contracts:

- ***AllowList*** – a contract used to manage a list of addresses and attest each address certain attributes.
- ***ContinuousFundraising*** – a contract that represents the offer to buy Tokens at a preset price.
- ***FeeSettings*** – used to manage fees paid to the tokenize.it platform.
- ***PersonalInvite*** – like ContinuousFundraising, represents an offer to buy Tokens at a certain price, but it is unique and created using specific buyer, receiver, amount, price, expiration and pair.
- ***PersonalInviteFactory*** – a factory contract used to deploy PersonalInvite contracts.
- ***Token*** – a contract based on ERC20 and ERC2771 standards with some extended functionalities to fit company tokenizations. It is pausable, burnable, has role-based access control and can whitelist/blacklist holders using AllowList by having specific requirements.

Privileged roles

- The owner of AllowList can update the status of the addresses in the map.
- The owner of ContinuousFundraising can change all pair data (input token, output token, input token receiver, etc) and pause.unpause the contract.
- The owner of FeeSettings can propose fee changes, execute fee changes and change the fee collector.
- The Token contract has role-based access control:
    - **REQUIREMENT_ROLE** – The role that has the ability to define which requirements an address must satisfy to receive tokens.
    - **MINTERADMIN_ROLE** – The role that has the ability to grant the minter role.
    - **MINTER_ROLE** – The role that has the ability to mint tokens. Will be granted to the relevant PersonalInvite and ContinuousFundraising contracts.
    - **BURNER_ROLE** – The role that has the ability to burn tokens from anywhere. Usage is planned for legal purposes and error recovery.
    - **TRANSFERERADMIN_ROLE** – The role that has the ability to grant transfer rights to other addresses.
    - **TRANSFERER_ROLE** – Addresses with this role do not need to satisfy any requirements to send or receive tokens.
    - **PAUSER_ROLE** – The role that has the ability to pause the token.

Findings

Critical

No critical severity issues were found.

High

No high severity issues were found.

Medium

No medium severity issues were found.

Low

1. **Floating Pragma**

   Locking the pragma helps ensure that contracts do not accidentally get deployed using, for example, an outdated compiler version that might introduce bugs that affect the contract system negatively.

   The project uses floating pragma ^0.8.0.

   Paths: all contracts

   Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment.

   Status: Fixed (Revised commit: 88f1908)

2. **Missing Zero Address Validation**

   Address parameters are being used without checking against the possibility of 0x0.

   This can lead to unwanted external calls to 0x0.

   Path: ./contracts/ContinuousFundraising.sol : constructor()

   Recommendation: Implement zero address checks.

   Status: Fixed (Revised commit: 88f1908)

3. **Unindexed Events**

   Having indexed parameters in the events makes it easier to search for these events using indexed parameters as filters.

   Paths: ./contracts/FeeSettings.sol, ./contracts/Token.sol, ./contracts/ContinuousFundraising.sol, ./contracts/PersonalInviteFactory.sol

   Recommendation: Use the “indexed” keyword to at least one of the event parameters.

   Status: Fixed (Revised commit: 88f1908)

4. Redundant Code Block

Redundant parts of the code create excessive Gas costs.

In both ContinuousFundraising and PersonalInvite constructors, there is the variable declaration of `uint256 fee` followed by an “if” statement that, if true, will set the value of `fee` to 0. That is redundant because, in Solidity, all variables are already initialized at the moment of creation. In this case, `fee` was already set to 0 and all the useful code was inside the “else” block.

Paths: ./contracts/ContinuousFundraising.sol, ./contracts/PersonalInvite.sol

Recommendation: Remove the redundant reassignment and improve the “if” statement readability.

Status: Fixed (Revised commit: 88f1908)

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 report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted to and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and 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.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

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, Consultant cannot guarantee the explicit security of the audited smart contracts.