SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: TrustSwap
Date: January 10, 2023

This report may contain confidential information about IT systems and the intellectual property of the Customer, as well as information about potential vulnerabilities and methods of their exploitation.

The report can be disclosed publicly after prior consent by another Party. Any subsequent publication of this report shall be without mandatory consent.

Document

Name	Smart Contract Code Review and Security Analysis Report for TrustSwap
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	ERC20 token; Vesting
Platform	Casper
Language	Rust
Methodology	Link
Website	https://trustswap.com
Changelog	27.12.2022 - Initial Review 10.01.2023 - Second Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by TrustSwap (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 consists of smart contracts in the repository:

Initial review scope

Repository	https://github.com/trustswap/team-finance-casper/tree/main/control_erc20_contract
Commit	ce6fb3b4
Whitepaper	Link
Functional Requirements	Link
Technical Requirements	Link
Contracts	File: ./src/address.rs SHA3:b17196cc268b96b13195654e293df85a4d94d0bfa7452de47ee130dff02284e7 File: ./src/constants.rs SHA3 91a9f5d4844e9d5e2d1c0a01ac80f4c8784e5c8ca111d6a8237d8eabd3245385 File: ./src/interact_token.rs SHA3:5c169e1a17f4aecb8acaeeb7826842b217f4dffe98ea906fd6fc7dc1a3e31e68 File: ./src/lib.rs SHA3:96a91987cf0beec69475373821d142cd09111859a2e7e1539531f3edbb17e2ad File: ./src/main.rs SHA3:273561348e6506cf2b9fb95aa315976537ad434deea1ec6e1b347adcdb444d7d File: ./src/utils.rs SHA3:1a6fba5f8c14ecdfbb7995445b83435c26e08b57538abf5715ff897b2e305dd8 File: ./src/vest.rs SHA3:06b204d1a314c5d747e9e832933b4d572abfaa07df8287a86ecd819f15efc652

Second review scope

Repository	https://github.com/trustswap/team-finance-casper/tree/main/control_erc20_contract
Commit	1494d88
Contracts	File: ./src/address.rs SHA3:b17196cc268b96b13195654e293df85a4d94d0bfa7452de47ee130dff02284e7 File: ./src/constants.rs SHA3 00b62b6e5ff88ce1661c553f3a4efd1d17250a6258c5593c644cd4f38dfac1a3

	File: ./src/interact_token.rs SHA3:cf960b9483671122eb096c4e970a88ee14a5581c46520410fc874b24a8a4bb42 File: ./src/lib.rs SHA3:6025f0bee87d6ae511f17f18c1f2f327c38c7d408994989f3810f8f347e342ba File: ./src/main.rs SHA3:5f9a620aa6c7bacc1874a2124207e4d83525a597d8398bdf3261c87f9453a672 File: ./src/utils.rs SHA3:ba090425d6cd11c2826e564d4025bac42594d3452c40f355d2400a303deaee3b File: ./src/vest.rs SHA3:f008b35b170be240d208565cef6b031ddd3a77aa41577253effbbcf9aab1daa3

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 asset 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 code quality

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 **2** out of **10**.

- Functional requirements are partially missing.
- Superficial functional description is provided
- Superficial technical description is provided.

Code quality

The total Code Quality score is **6** out of **10**.

- The development environment is not configured.
- Code is not covered with comments.
- Clippy errors are partially not fixed.

Test coverage

Code coverage of the project is **0%** (branch coverage).

- Deployment and basic user interactions are not covered with tests.
- Negative case coverage is missing.
- Interactions with several users are not tested.

Security score

As a result of the second audit, the code contains **1** medium issue and **1** low severity issue. The security score is **9** 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: **7,7**.

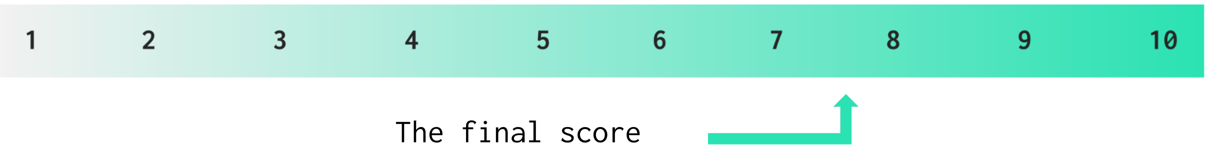

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
27 December 2022	8	1	0	0
10 January 2023	1	1	0	0

Checked Items

We have audited the Customers' smart contracts for commonly known and specific vulnerabilities. Here are some items considered:

Item	Description	Status
Default Visibility	Functions and state variables visibility should be set explicitly. Visibility levels should be specified consciously.	Passed
Integer Overflow and Underflow	If unchecked math is used, all math operations should be safe from overflows and underflows.	Passed
Outdated Compiler Version	It is recommended to use a recent version of the Rust compiler.	Passed
Unchecked Call Return Value	The return value of a message call should be checked.	Failed
Access Control & Authorization	Ownership takeover should not be possible. All crucial functions should be protected. Users could not affect data that belongs to other users.	Passed
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed
Shadowing State Variable	State variables should not be shadowed.	Passed
Weak Sources of Randomness	Random values should never be generated from Chain Attributes or be predictable.	Not Relevant
Calls Only to Trusted Addresses	All external calls should be performed only to trusted addresses.	Passed
Presence of Unused Variables	The code should not contain unused variables if this is not justified by design.	Passed
EIP Standards Violation	EIP standards should not be violated.	Not Relevant
Assets Integrity	Funds are protected and cannot be withdrawn without proper permissions or be locked on the contract.	Passed
User Balances Manipulation	Contract owners or any other third party should not be able to access funds belonging to users.	Passed

Data Consistency	Smart contract data should be consistent all over the data flow.	Passed
Flashloan Attack	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	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	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.	Not Relevant
Style Guide Violation	Style guides and best practices should be followed.	Failed
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Failed
Environment Consistency	The project should contain a configured development environment with a comprehensive description of how to compile, build and deploy the code.	Failed
Secure Oracles Usage	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	The code should be covered with unit tests. Test coverage should be sufficient, with both negative and positive cases covered. Usage of contracts by multiple users should be tested.	Failed
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Not Relevant

System Overview

The purpose of The vesting smart contract is for users to send tokens to the smart contract for it to lock and hold them for the specified time period, and releasing those tokens to the recipient according to the time schedule set by the user initially. Each lock has 3 basic functionalities: transfer, extend, and unlock. transfer changes the recipient of the unlocked tokens to another user. extend increases the lock time of the lock. unlock withdraws tokens based on the amount of time passed, relative to the lock schedules.
List of smart contracts:

- *VestContract* – The main vesting smart contract, where *lock*, *extend_lock*, *transfer_lock* and *claim* functionality is implemented.
- InteractToken - ERC20 functionality helper functions for the vesting smart contract.

Privileged roles

- Recipient - vesting token owner
- Signer/Caller - smart contract action signer or caller, who pays for transactions

Risks

- No stopping functionality provided for the smart contract, if something goes wrong or bad actors are found, the vesting contract can’t be stopped by an admin.
- Smart contract owner is not defined, and when the contract is invalid or needs an update or migration, no one would be able to do that.

Findings

🟥🟥🟥🟥 Critical

No critical severity issues were found.

🟧🟧🟧 High

No high severity issues were found.

🟨🟨 Medium

1. Unchecked Transfer Call

When the result of a transfer call is not checked, there is a risk that calling fails, but the result will be considered as successful and the smart contract state would be updated accordingly, so this can lead to an invalid contract state or funds loss/lock.

InteractTokens utilizes ‘*runtime::call_contract’* which will return () if the stored contract calls revert, it means the result should be considered as failed and no further actions should be taken, but this check is missing in the smart contract.

Path: ./src/vest.rs : lock(), claim()

Recommendation: Check the result of ‘*runtime::call_contract*’ if revered or not.

Status: Reported

🟦 Low

1. Redundant Names in Struct Init

Redundant field names in struct initialization, fields in struct literals should be shorthand. If the field and variable names are the same, the field name is redundant.

Path: ./src/vest.rs : lock():ln240

Recommendation: Remove redundant names from struct initialization.

Status: Fixed (Revised commit: 1494d88)

2. Needless Range Loop

Needless looping over the range of 0..len of some collection just to get the values by index.

Path: ./src/vest.rs : pack():ln62

Recommendation: Remove range loop use ‘for i in id_bytes’.

Status: Fixed (Revised commit: 1494d88)

3. Manual Slice Copy

Manually copying items between slices could be optimized by having a *‘memcpy’*. Manual copy is not as fast as a *‘memcpy’*.

Path: ./src/vest.rs : pack_schedule():ln101, ln105

Recommendation: try replacing the loop with: `res[..8].copy_from_slice(&release[..8]);`

Status: Fixed (Revised commit: 1494d88)

4. Unnecessary `let` binding

let-bindings should not be subsequently returned, It is extraneous code. Remove it to make your code more rusty.

Path: ./src/vest.rs : unpack():ln130, unpack_schedule():ln239

Recommendation: Remove let binding and return structure.

Status: Fixed (Revised commit: 1494d88)

5. Unnecessary Object Initialization

Writing `&Vec` instead of `&[_]` involves a new object where a slice will do.

Requiring the argument to be of a specific size makes the function less useful without any benefit; slices in the form of &[T] or &str usually suffice and can be obtained from other types.

Path: ./src/vest.rs : unpack_recipient():ln164

Recommendation: Replace ‘*&Vec<u8>*’ with ‘*&[u8]*’

Status: Fixed (Revised commit: 1494d88)

6. Needless Borrowing

The expression ‘*&runtime::get_caller().to_string().as_str()*’ creates a reference which is immediately dereferenced by the compiler.

Path: ./src/vest.rs : claim():ln361, ln364, ln373

Recommendation: Remove the unnecessary reference from ‘*&runtime::get_caller().to_string().as_str()*’.

Status: Fixed (Revised commit: 1494d88)

7. Unoptimized Empty String Check

Some structures can answer .is_empty() much faster than calculating their length. Using .is_empty(), is recommended as it is cheaper., It makes the intent clearer than a manual comparison in some contexts.

Path: ./src/vest.rs : is_valid_entry():ln391

Recommendation: Use is_empty where possible.

Status: Fixed (Revised commit: 1494d88)

8. Calls to `push` immediately after creation

If the Vec is created using with_capacity this will only lint if the capacity is a constant and the number of pushes is greater than or equal to the initial capacity.

If the Vec is extended after the initial sequence of pushes and it was initialized by default then this will only lint after at least four pushes. This number may change in the future.

Path: ./src/vest.rs : update_storage():ln432, update_storage_transfer_lock():ln465

Recommendation: Use the vec![] macro as it’s more performant and easier to read than multiple push calls.

Status: Reported

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing 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 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, the Consultant cannot guarantee the explicit security of the audited smart contracts.