HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: WhiteBIT
Date: August 25th, 2022

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

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 WhiteBIT
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU Noah Jelich | Senior Solidity SC Auditor at Hacken OU
Type	Vesting
Platform	EVM
Network	Ethereum
Language	Solidity
Methods	Manual Review, Automated Review, Architecture Review
Website	https://whitebit.com/ua
Timeline	19.07.2022 - 25.08.2022
Changelog	22.07.2022 - Initial Review 02.08.2022 - Second Review 25.08.2022 - Third 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 WhiteBIT (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/whitebit-exchange/wbt-token/
Commit:
bbdf662650a2127da8dae1c87284b070cfd9c70f
Technical Documentation:
Type: Technical description

Type: Functional requirements
Integration and Unit Tests: Yes
Contracts:
File: ./escrow-private/contracts/Escrow.sol
SHA3: d59a2a458ae347d9e9d288ffbc81085391ac15e669fdc64ebefc9c47d5a01f05

File: ./escrow-private/contracts/interfaces/IERC20.sol
SHA3: a334b22db83ae9db74ee4f498532345b7de99e7c84947f684a8154bcd0ac4db6

File: ./escrow-private/contracts/Ownable.sol
SHA3: 2cdb58d27bdbb7607735ba2057e66ee526e2fafc42de2469705ec28e7b9e3db9

File: ./escrow-private/contracts/testToken.sol
SHA3: 364e9c82011434187716b8a9dafe94053b8431159f0b1da1cf5dccbc617f655c

Second review scope

Repository:
https://github.com/whitebit-exchange/wbt-token/
Commit:
f98af086d186098a06e5a0cf57a32ae1d43b0d82
Technical Documentation:
Type: Technical description

Type: Functional requirements
Integration and Unit Tests: Yes
Contracts:
File: ./escrow-private/contracts/CustomOwnable.sol
SHA3: c242b28e85874b2e9bd1ee6062bf72a6d2087876f3b21af9e0fffab67db46bdb

File: ./escrow-private/contracts/Escrow.sol
SHA3: 1c23fae7a66d1475b6aed509acc1677967dd32044a0d05d8d00f1eb7399d30a6

File: ./escrow-private/contracts/interfaces/IERC20.sol
SHA3: a334b22db83ae9db74ee4f498532345b7de99e7c84947f684a8154bcd0ac4db6

File: ./escrow-private/contracts/testToken.sol
SHA3: 364e9c82011434187716b8a9dafe94053b8431159f0b1da1cf5dccbc617f655c

Third review scope

Repository:
https://github.com/whitebit-exchange/wbt-token/

Commit:
7303a11b04a6ede4aa36bb9b25b25725c3315283

Technical Documentation:
Type: Technical description

Type: Functional requirements

Integration and Unit Tests: Yes

Contracts:
File: ./escrow-private/contracts/CustomOwnable.sol
SHA3: c242b28e85874b2e9bd1ee6062bf72a6d2087876f3b21af9e0fffab67db46bdb

File: ./escrow-private/contracts/Escrow.sol
SHA3: 897ed5d7f0fe5a64fac1b230014619870df41367568cb55024564e60f164a0c7

File: ./escrow-private/contracts/interfaces/IERC20.sol
SHA3: a334b22db83ae9db74ee4f498532345b7de99e7c84947f684a8154bcd0ac4db6

File: ./escrow-private/contracts/testToken.sol
SHA3: 364e9c82011434187716b8a9dafe94053b8431159f0b1da1cf5dccbc617f655c

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

Documentation quality

The total Documentation Quality score is **10** out of **10**. Functional requirements and a technical description were provided.

Code quality

The total CodeQuality score is **10** out of **10**. Most of the code follows official language style guides. Unit tests were provided.

Architecture quality

The architecture quality score is **9** out of **10**. The architecture is clear. The development environment was provided, but the instructions on how to deploy the code were not provided.

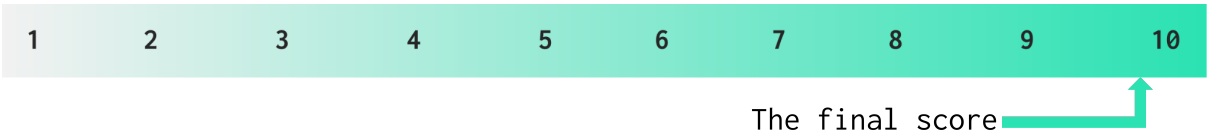
Security score

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: **9.9**.



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.	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.	Passed
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.	Failed
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	SWC-115	tx.origin should not be used for	Passed

through tx.origin		authorization.	
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	Signed messages should always have a unique id. A transaction hash should not be used as a unique id. Chain identifier should always be used. All parameters from the signature should be used in signer recovery	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.	Not Relevant
Calls Only to Trusted Addresses	EEA-Level-2 SWC-126	All external calls should be performed only to trusted addresses.	Passed
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.	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	Passed

		data stored on the contract. There should not be any cases when execution fails due to the block Gas limit.	
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.	Failed
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, that may be changed in the future.	Passed

System Overview

WBT: Private Sale Vesting is a vesting project with the following contracts:

- *Escrow* — is a vesting contract that allows the distribution of WBT tokens (the contract allows the distribution of any tokens). The time ranges for 5 schedules during which tokens are distributed are defined during the contract deployment.

  Before the schedules start, the owner adds the information about the vesting for each user (address and the amount of tokens). The amount is divided into 5 schedules (per 20% for each schedule) and is available to be transferred to the user by the contract owner at the appropriate time.
- *CustomOwnable* — is a contract with the owner access mechanism, inherited by the *Escrow* contract.
- *testToken* — is an ERC-20 token contract used for testing purposes.
- *IERC20* — is an interface for ERC-20 tokens, used in the *Escrow* contract, inherited by the *testToken* contract.

Privileged roles

- The owner of the *Escrow* contract can add the information about the vesting for users before the distribution starting time, and remove the information for the vesting anytime. The owner triggers the vest (tokens distribution).

Risks

- The tokens distribution, the correctness of amounts, and the receivers can not be guaranteed, as the contract owner adds and removes the information for the distribution tokens (users and amounts), and triggers the distribution.
- There are contracts in the repository not included in the audit scope.

Findings

■■■■ Critical

No critical severity issues were found.

■■■ High

1. **Denial of Service vulnerability**

   The tokens are transferred in loops that run over all the schedules and all the users.

   If the number of schedules or users to get tokens are large enough to make the Gas required for executing the for loop exceed the block Gas limit, the *vest* function will be inoperable.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow

   Function: vest

   Recommendation: Limit the number of iterations.

   Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

2. **Funds lock**

   Tokens are transferred to the contract to be vested, but there is no mechanism for their withdrawal.

   Therefore, if the vesting information was deleted (*purgeAccount* function) or extra funds were sent, it would not be possible to withdraw them.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow

   Recommendation: Add the function for extra funds withdrawal (*IERC20(token).balanceOf(address(this) - totalVestedBalance*).

   Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

■■ Medium

1. **Failing tests**

   4 of the test cases are failing with the *"TypeError: escrow.countSchedules is not a function"* message.

   Recommendation: Ensure that all the test cases are passing.

   Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

2. **Unoptimized loops usage**

   The function *addVestingSchedule* uses a loop with updating *totalVestedBalance* state variable.

   This will lead to Gas losses.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow.sol

   Function: addVestingSchedule

   Recommendation: Cache *totalVestedBalance* variable to a memory variable *_totalVestedBalance* and update *totalVestedBalance* after the loop execution.

   Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

3. **Unoptimized loops usage**

   It is checked if the *block.timestamp* value is less than all the schedules (*_times*) in the *constructor* inside a loop.

   This functionality is redundant, because only the first schedule may be checked for this condition, leading to Gas losses.

   If the *block.timestamp* value is less than the *VestingSchedulesRemained[i]* value, the *continue* statement is used, and the loop runs all the next iterations.

   This functionality is redundant and leads to Gas losses, as if the *block.timestamp* value is less than the *VestingSchedulesRemained[i]*, it is less than all the next *VestingSchedulesRemained* values.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow.sol

   Functions: constructor, vest

   Recommendation: Check if the *block.timestamp* value is less than *_times[0]* outside a loop in the *constructor*. Use a *break* statement if the *block.timestamp* value is less than the *VestingSchedulesRemained[i]* in the *vest* function.

   Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

4. **Missing events emit on changing important values**

   It is recommended to emit events after changing important values. This will give everyone the ability to be noticed after such changes easily.

   File: ./escrow-private/contracts/Escrow.sol

Contract: Escrow.sol

Functions: appendVestingEntry, purgeAccount

Recommendation: Implement event emits after changing contract values.

Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

5. **Checks-Effects-Interaction pattern violation**

   The state variables are updated after the external calls.

   This can lead to reentrancies, race conditions, or denial of service vulnerabilities.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow.sol

   Functions: vest, appendVestingEntry

   Recommendation: Implement functions according to the Checks-Effects-Interactions pattern.

   Status: Fixed (Revised commit: 7303a11b04a6ede4aa36bb9b25b25725c3315283)

6. **Unoptimized loops usage**

   The function `vest` uses a loop with the `j <= MAX_ACCOUNTS_COUNT` condition. If there are fewer accounts (`accountMapperCounter`), the loop will have redundant iterations.

   This will lead to Gas losses.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow.sol

   Function: vest

   Recommendation: Use the number of accounts for the loop condition.

   Status: Fixed (Revised commit: 7303a11b04a6ede4aa36bb9b25b25725c3315283)

Low

1. **Modification of a well-known contract**

   Imported or copy-pasted contracts (like `SafeMath`, `Context`, `Ownable`, etc.) should not be modified to keep the code clear.

   The Base64 is a modified version of the OpenZeppelin implementation of the contract of the same name.

   File: ./escrow-private/contracts/Ownable.sol

Contract: Ownable

Recommendation: Delete the modifications.

Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

2. **Use of hard-coded values**

   The value for *percentDistributionPerMonth* in the *appendVestingEntry* functions is hardcoded.

   Using hardcoded values is not a good practice.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow

   Function: appendVestingEntry

   Recommendation: Convert the hardcoded value into the constant.

   Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

3. **Use of "uint" keyword**

   The contract uses *uint* data type.

   Although *uint* and *uint256* are the same, a recommended way is to use *uint256*, as it increases the readability of the code.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow

   Recommendation: Replace the *uint* data types with the *uint256*.

   Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

4. **Redundant use of "Add Assignment" operator**

   The value for *totalVestedAccountBalance* is set using the "Add Assignment" operator (*+=*).

   This value is set only once, and the use of the "Simple Assignment" operator (=) would save Gas.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow

   Function: appendVestingEntry

   Recommendation: Use the "Simple Assignment" operator for the *totalVestedAccountBalance* variable value setting.

Status: Fixed (Revised commit: 7303a11b04a6ede4aa36bb9b25b25725c3315283)

5. **Use of hard-coded values**

   The required value for *times* (5) in the *constructor* is hardcoded.

   Using hardcoded values is not a good practice.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow

   Function: constructor

   Recommendation: Convert the hardcoded value into the constant.

   Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

6. **Block values as a proxy for time using**

   The contract uses *block.timestamp* for time calculations. It is not precise and safe.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow

   Functions: constructor, appendVestingEntry, vest

   Recommendation: It is recommended to avoid using *block.timestamp* in the time calculations. Alternatively, it is safe to use oracles.

   Status: Mitigated. The Customer comment: “We do not use the timestamp value for calculations, block.timestamp deviation of a few seconds can only result in the vest allocation function becoming available a couple of seconds earlier or later, which does not pose a threat to the security of user funds and the contract as a whole”.

7. **Division before multiplication**

   Since Solidity only supports whole number division, the order of operations between multiplication and division is important.

   Division before multiplication can lead to rounding errors.

   File: ./escrow-private/contracts/Escrow.sol

   Contract: Escrow

   Function: appendVestingEntry (line 121)

   Recommendation: Perform the multiplication before the division.

   Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

8. **Missing zero address validation**

Address parameter (*_token*) is being used without checking against the possibility of 0x0.

This can lead to unwanted external calls to 0x0.

File: ./escrow-private/contracts/Escrow.sol

Contract: Escrow

Function: constructor

Recommendation: Implement zero address checks.

Status: Fixed (Revised commit: f98af086d186098a06e5a0cf57a32ae1d43b0d82)

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.