

HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: VitalTale
Date: 10 October, 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 VitalTale
Approved By	Grzegorz Trawiński Lead Solidity SC Auditor at Hacken OÜ
Tags	ERC20 token
Platform	EVM
Language	Solidity
Methodology	Link
Website	N/A
Changelog	20.09.2023 - Initial Review 10.10.2023 - Second Review

Table of contents

Introduction	4
System Overview	4
Executive Summary	5
Risks	6
Checked Items	7
Findings	10
Critical	10
High	10
Medium	10
M01. Missing Input Validations: setPaymentConfig()	10
Low	10
L01. Missing Events on Critical State Updates	10
Informational	11
I01. Floating Pragma	11
I02. Solidity Style Guide Violation: Order Of Layout	11
I03. Unused Function Arguments	12
I04. Functions That Should Be Declared View	12
I05. Functions That Should Be Declared External	12
Disclaimers	14
Appendix 1. Severity Definitions	15
Risk Levels	15
Impact Levels	16
Likelihood Levels	16
Informational	16
Appendix 2. Scope	17

Introduction

Hacken OÜ (Consultant) was contracted by VitalTale (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.

System Overview

VitalTale is a protocol with the following contracts:

- *VITALTALE* – simple ERC-20 token with custom minting mechanism. Minting is allowed for the user with the `MINTER_ROLE` after the mint config is set by the `DEFAULT_ADMIN_ROLE`. It has the following attributes:
 - Name: `VITALTALE`
 - Symbol: `VTTL`
 - Decimals: 18
- *AntiSnipe* – a whitelist contract for blocking ERC-20 transfers performed by non-whitelisted addresses.

Privileged roles

- The `DEFAULT_ADMIN_ROLE` of the *VITALTALE* contract can update mint config and payment config, enable/disable antisnipe (whitelist) feature and change the antisnipe contract address.
- The `MINTER_ROLE` of the *VITALTALE* contract can perform mintings after the mint config is set by the admin.
- The owner of the *AntiSnipe* contract can add and remove addresses from the whitelist.

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 **7** out of **10**.

- Functional requirements are provided.
- Technical description is provided.
- Natspec comments are not included.

Code quality

The total Code Quality score is **9** out of **10**.

- Solidity Style Guide is not being followed (I02)

Test coverage

Code coverage of the project is **100%** (branch coverage).

- Tests are not required for projects with less than 250 lines of code.

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: **9.5**. The system users should acknowledge all the risks summed up in the risks section of the report.



The final score 

Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
20 September 2023	1	1	0	0
10 October 2023	0	0	0	0



Risks

No risks were identified.

Checked Items

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

Item	Description	Status	Related Issues
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.	Not Relevant	
Outdated Compiler Version	It is recommended to use a recent version of the Solidity compiler.	Passed	
Floating Pragma	Contracts should be deployed with the same compiler version and flags that they have been tested thoroughly.	Passed	
Unchecked Call Return Value	The return value of a message call should be checked.	Not Relevant	
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	
SELFDESTRUCT Instruction	The contract should not be self-destructible while it has funds belonging to users.	Not Relevant	
Check-Effect-Interaction	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Passed	
Assert Violation	Properly functioning code should never reach a failing assert statement.	Passed	
Deprecated Solidity Functions	Deprecated built-in functions should never be used.	Passed	
Delegatecall to Untrusted Callee	Delegatecalls should only be allowed to trusted addresses.	Not Relevant	
DoS (Denial of Service)	Execution of the code should never be blocked by a specific contract state unless required.	Passed	

Race Conditions	Race Conditions and Transactions Order Dependency should not be possible.	Passed	
Authorization through tx.origin	tx.origin should not be used for authorization.	Passed	
Block values as a proxy for time	Block numbers should not be used for time calculations.	Passed	
Signature Unique Id	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	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	
Incorrect Inheritance Order	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	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.	Passed	
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. Contracts shouldn't rely on values that can be changed in the same transaction.	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.	Passed	
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.	Passed	
Style Guide Violation	Style guides and best practices should be followed.	Failed	I02
Requirements Compliance	The code should be compliant with the requirements provided by the Customer.	Passed	
Environment Consistency	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	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.	Passed	
Stable Imports	The code should not reference draft contracts, which may be changed in the future.	Passed	

Findings

Critical

No medium severity issues were found.

High

No high severity issues were found.

Medium

M01. Missing Input Validations: `setPaymentConfig()`

Impact	High
Likelihood	Low

The function `setPaymentConfig()` changes the stored payment config (fee receiver and fee percentage), but the function does not validate the inputted values.

The function `payment()` will perform transactions relying on the payment config. There are two cases where transactions can fail if the payment config is invalid:

- `_paymentConfig.receiver` is `address(0)` - fee transfer will fail, as transferring to the `address(0)` is not valid;
- `_paymentConfig.fee` is above 10000 - despite there is documentation that fee has two decimals, the fee percentage is not validated by the setter and values above 10000 can be provided.

Path: VITALTALE.sol: `setPaymentConfig()`.

Recommendation: Add proper validation to the `setPaymentConfig()` function.

Found in: [N/A]

Status: Fixed (Revised commit: N/A)

Remediation: The `setPaymentConfig()` now implements input validation. The receiver can not be set to `address(0)`. The fee must be set between 0 and 10000 inclusively.

Low

L01. Missing Events on Critical State Updates

Impact	Low
Likelihood	Medium

Critical state changes should emit events for tracking things off-chain.

This can lead to inability for users to subscribe events and check what is going on with the project.

Path: VITALTALE.sol: setMintConfig(), setPaymentConfig().

Recommendation: Emit events on critical state changes.

Found in: [N/A]

Status: Fixed (Revised commit: N/A)

Remediation: All mentioned functions now emit adequate events.

Informational

I01. Floating Pragma

The project uses floating pragmas `^0.8.9`.

This may result in the contracts being deployed using the wrong pragma version, which is different from the one they were tested with. For example, they might be deployed using an outdated pragma version, which may include bugs that affect the system negatively.

Path: *

Recommendation: Consider locking the pragma version whenever possible and avoid using a floating pragma in the final deployment. Consider known bugs (<https://github.com/ethereum/solidity/releases>) for the compiler version that is chosen.

Found in: [N/A]

Status: Fixed (Revised commit: N/A)

Remediation: The floating pragma is now locked on 0.8.19 version.

I02. Solidity Style Guide Violation: Order Of Layout

Inside each contract, library or interface, use the following order:

1. Type declarations
2. State variables
3. Events
4. Errors
5. Modifiers
6. Functions
 - a. constructor
 - b. initializer (if exists)
 - c. receive function (if exists)
 - d. fallback function (if exists)
 - e. external

- f. public
- g. internal
- h. private

Path: VITALTALE.sol.

Recommendation: Change order of layout to fit [Official Style Guide](#).

Found in: [N/A]

Status: **Reported**

Remediation: No change in VITALTALE.sol file was observed since initial review.

I03. Unused Function Arguments

The function `assureCanTransfer()` arguments `from`, `to` and `amount` are never used.

Path: AntiSnipe.sol: `assureCanTransfer()`.

Recommendation: Remove unused variables, completely or partially (just letting the types and excluding the variable names, for interface compatibility purposes).

Found in: [N/A]

Status: **Fixed** (Revised commit: N/A)

Remediation: The unused variables are now partially removed from the `assureCanTransfer()` function leaving only the types in function signature.

I04. Functions That Should Be Declared View

The function `_beforeTokenTransfer()` does not change the state of the network and should be declared “view”.

Path: VITALTALE.sol: `_beforeTokenTransfer()`.

Recommendation: Change function to view.

Found in: [N/A]

Status: **Fixed** (Revised commit: N/A)

Remediation: The `_beforeTokenTransfer()` function is now declared as “view”.

I05. Functions That Should Be Declared External

Functions that are only called from outside the contract should be defined as external. External functions are much more Gas efficient compared to public functions.

Paths:

www.hacken.io

```
VITALTALE.sol:      mint(),      isMintStarted(),      isMintEnded(),  
getMintPeriod(),  getMintConfig(),  setMintConfig(),  payment(),  
getPaymentConfig(), setPaymentConfig();
```

```
AntiSnipe.sol:      addToWhitelist(),      removeFromWhitelist(),  
assureCanTransfer().
```

Recommendation: Change function visibilities to external.

Status: Fixed (Revised commit: N/A)

Remediation: All mentioned functions are now declared as external.

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.

Appendix 1. Severity Definitions

When auditing smart contracts Hacken is using a risk-based approach that considers the potential impact of any vulnerabilities and the likelihood of them being exploited. The matrix of impact and likelihood is a commonly used tool in risk management to help assess and prioritize risks.

The impact of a vulnerability refers to the potential harm that could result if it were to be exploited. For smart contracts, this could include the loss of funds or assets, unauthorized access or control, or reputational damage.

The likelihood of a vulnerability being exploited is determined by considering the likelihood of an attack occurring, the level of skill or resources required to exploit the vulnerability, and the presence of any mitigating controls that could reduce the likelihood of exploitation.

Risk Level	High Impact	Medium Impact	Low Impact
High Likelihood	Critical	High	Medium
Medium Likelihood	High	Medium	Low
Low Likelihood	Medium	Low	Low

Risk Levels

Critical: Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.

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.

Medium: Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.

Low: Major deviations from best practices or major Gas inefficiency. These issues won't have a significant impact on code execution, don't affect security score but can affect code quality score.

Impact Levels

High Impact: Risks that have a high impact are associated with financial losses, reputational damage, or major alterations to contract state. High impact issues typically involve invalid calculations, denial of service, token supply manipulation, and data consistency, but are not limited to those categories.

Medium Impact: Risks that have a medium impact could result in financial losses, reputational damage, or minor contract state manipulation. These risks can also be associated with undocumented behavior or violations of requirements.

Low Impact: Risks that have a low impact cannot lead to financial losses or state manipulation. These risks are typically related to unscalable functionality, contradictions, inconsistent data, or major violations of best practices.

Likelihood Levels

High Likelihood: Risks that have a high likelihood are those that are expected to occur frequently or are very likely to occur. These risks could be the result of known vulnerabilities or weaknesses in the contract, or could be the result of external factors such as attacks or exploits targeting similar contracts.

Medium Likelihood: Risks that have a medium likelihood are those that are possible but not as likely to occur as those in the high likelihood category. These risks could be the result of less severe vulnerabilities or weaknesses in the contract, or could be the result of less targeted attacks or exploits.

Low Likelihood: Risks that have a low likelihood are those that are unlikely to occur, but still possible. These risks could be the result of very specific or complex vulnerabilities or weaknesses in the contract, or could be the result of highly targeted attacks or exploits.

Informational

Informational issues are mostly connected to violations of best practices, typos in code, violations of code style, and dead or redundant code.

Informational issues are not affecting the score, but addressing them will be beneficial for the project.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Initial review scope

Repository	N/A, smart contracts were obtained from Goerli Testnet.
Commit	N/A
Whitepaper	N/A
Requirements	Link
Technical Requirements	N/A
Contracts Addresses	https://goerli.arbiscan.io/address/0x3d1454Ab6C304822a8680726C0B368bCF2e2eF58#code https://goerli.arbiscan.io/address/0xc2D19c3A64D3fd2998cDcc480d38046cF3D847b5#code
Contracts	File: VITALTALE.sol SHA3: cb4946c05910ffd902b763d53bec7bc31cb24bd57ea330857853db50887d6e4a File: Antisnipe.sol SHA3: f4ba7bb8488970aa48f4dc457e4391cd79c9c840006e6faa5739caa931a8b555

Second review scope

Repository	N/A, files were provided as a zipped package.
Commit	N/A
Whitepaper	N/A
Requirements	Link
Technical Requirements	N/A
Contracts Addresses	https://arbiscan.io/address/0xB0f778A62d0DA8d7a6Ac03d390c47f2D85c386c1
Contracts	File: VITALTALE.sol SHA3: 0594ff681d1146c7e6e1dda63958ad313d7e18e3824fe3fd956ccc77a10f1115 File: Antisnipe.sol SHA3: 7e3989040a3b44a1dba6a1707b5092ccf44b9e781d62152a7dca00f857a793b1