HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: OpenWare
Date: June 21st, 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 OpenWare
Approved By	Evgeniy Bezuglyi | SC Audits Department Head at Hacken OU
Type	ERC20 token
Platform	EVM
Language	Solidity
Methods	Manual Review, Automated Review, Architecture review
Website	https://www.openware.com/
Timeline	06.06.2022 - 21.06.2022
Changelog	09.06.2022 - Initial Review 21.06.2022 - Second Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by OpenWare (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/yellorg/smart
Commit:
448cac0f894b601167338437fe1abe2d8059e57f
Technical Documentation:
Type: Whitepaper (partial functional requirements provided)
Link

Type: Project documentation (Project overview)
Link

Type: Tokenomics description (superficial requirements provided)
Link

Integration and Unit Tests: Yes (in “test” directory)
Contracts:
File: ./contracts/yellow/Yellow.sol
SHA3: 99da529164f6b57889cf0f9933cddc37dea73d29b70d948fe2939aa7be35a239

Second review scope

Repository:
https://github.com/yellorg/smart/tree/fix/yellow-sc
Commit:
63bdcdbbb7a8d7ae5b21f1dcbbfa6badd4dc1c95
Technical Documentation:
Type: Whitepaper (partial functional requirements provided)
Link

Type: Project documentation (Project overview)
Link

Type: Tokenomics description (superficial requirements provided)
Link

Integration and Unit Tests: Yes (in “test” directory)
Contracts:
File: ./contracts/yellow/Yellow.sol
SHA3: d4a7ac593cf14b31923ceee6d6ff3202552cc6033930697c8ac06a0a5bab3e27

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Low	Low-level vulnerabilities are mostly related to outdated, unused, etc.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.

Executive Summary

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

Documentation quality

The total Documentation Quality score is **8** out of **10**. Functional requirements are superficial. A technical description is provided in code by NatSpec comments.

Code quality

The total CodeQuality score is **9** out of **10**. Code violates the style guide in case of functions order, code is covered with tests.

Architecture quality

The architecture quality score is **9** out of **10**. Some functions can be declared external.

Security score

As a result of the audit, the code contains **1** low severity issue. 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: **9.6**.

1
2
3
4
5
6
7
8
9
10
The final score

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.	Passed
Check-Effect-Interaction	SWC-107	Check-Effect-Interaction pattern should be followed if the code performs ANY external call.	Not Relevant
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.	Not Relevant
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.	Not Relevant
Race	SWC-114	Race Conditions and Transactions Order	Not Relevant

Conditions		Dependency should not be possible.	
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	Signed messages should always have a unique id. A transaction hash should not be used as a unique id.	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.	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.	Passed
Gas Limit and Loops	**Custom**	Transaction execution costs should not depend dramatically on the amount of data stored on the contract. There	Passed

		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.	Failed
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

- *Yellow* - ERC20 token with the logic inherited from OpenZeppelin contracts. Additional minting is allowed until the cap is reached.
  It has the following attributes:
  - Name: Yellow
  - Symbol: $Yellow
  - Total supply: 10 000 000 000 tokens.

Privileged roles

- The owner of the *Yellow* contract can pause and unpause the contract and mint tokens.

Findings

🟥🟥🟥🟥 Critical

No critical severity issues were found.

🟨🟨🟨 High

1. **Highly permissive owner`s access to burn tokens.**

The owner can burn the tokens of a certain address.

This can lead to the loss of users' funds.

Contracts: Yellow.sol

Function: burnFrom

Recommendation: Add owner`s permissions in the documentation or refactor highly permitted functions.

Status: Fixed (revised commit: 63bdcdb)

2. **Owner can stop all the project`s transactions.**

The owner can stop all the token transfers with a pause function.

This can lead to the users' funds manipulation.

Contracts: Yellow.sol

Function: init

Recommendation: Add owner`s permissions in the documentation.

Status: Mitigated (Customer added functionality description to NatSpec comment in code)

🟩🟩 Medium

Code documentation contradiction.

Some statements in code comments do not match the real code. Function *burnFrom* can be called only by BURNER_ROLE, while NatSpec comments specify that it can be called only by MINTER_ROLE.

This makes code hard to read and evaluate.

Contract: -

Functions: -

Recommendation: Refactor the code project documentation and add the code requirements.

Status: Fixed (revised commit: 63bdcdb)

🟦 Low

1. **Functions can be declared as external.**

To save Gas, public functions that are never called in the contract should be declared as external.

Contracts: Yellow.sol

Functions: mint, burn, pause

Recommendation: Declare mentioned functions as external.

Status: Reported

2. **Floating pragma.**

The project`s contracts use floating pragma ^0.8.0.

Contracts: Yellow.sol

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

Status: Fixed (revised commit: 63bdcdb)

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.