HACKEN

SMART CONTRACT CODE REVIEW AND SECURITY ANALYSIS REPORT

Customer: Marsha Foundation
Date: 16 November, 2023

Document

Name	Smart Contract Code Review and Security Analysis Report for Marsha Foundation
Audited By	Pukhno Bohdan | Solidity SC Auditor at Hacken OÜ
Approved By	Yves Toiser | Solidity SC Auditor at Hacken OÜ
Tags	ERC20 token
Platform	EVM
Language	Solidity
Methodology	Link
Website	https://www.marshafoundation.org/
Changelog	03.11.2023 - Initial Review 16.11.2023 - Second Review

Table of contents

Introduction

Hacken OÜ (Consultant) was contracted by Marsha Foundation (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

MarshaToken – it is a simple ERC-20 token that mints all initial supply to the deployed contract. Additional minting is not allowed.
It has the following attributes:

- Name: *MARSHA+*
- Symbol: MSA
- Decimals: 18
- Total supply: 8 billion tokens.

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.

Test coverage

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

Security score

As a result of the audit, the code contains **0** 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**. The system users should acknowledge all the risks summed up in the risks section of the report.

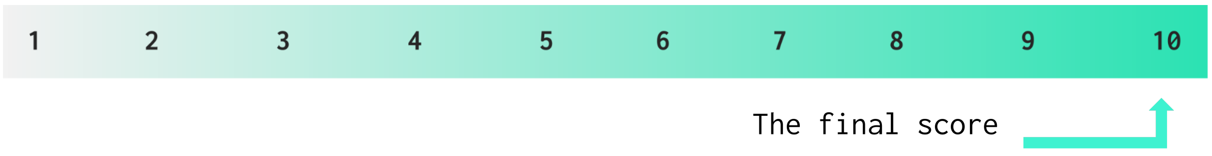


Table. The distribution of issues during the audit

Review date	Low	Medium	High	Critical
3 November 2023	2	1	0	1
16 November 2023	0	0	0	0

Risks

- If community tokens are moved to a different address, it will no longer be possible to call the *burnIfNeeded()* function, nor will the annual burning of community tokens be feasible.

Findings

■■■■ Critical

C01. Funds lock because of denial of transfer service

Impact	High
Likelihood	High

The *transfer()* function will be locked after the 3 years period, because the deployed contract will have no tokens to distribute.

The contract initially mints the entire initial supply of tokens to itself and proceeds to transfer 63.5% of these tokens to various recipients while setting up a time lock for the distribution of the remaining 36.5% after a 3-year period.

The *teamRewardAfter3Years()* function is responsible for distributing the outstanding percentage of tokens from the contract to different recipients, and it can only be invoked once after the 3-year period. After this initial call, any further invocations of this function will result in an error message "ERC20: transfer amount exceeds balance" because the contract will have no tokens left, as all the initial supply was distributed to recipients.

This issue arises because the *teamRewardAfter3Years()* function is triggered each time the *transfer()* or *transferFrom()* function is called.

Therefore transfers of the token will not be available, because each transfer will throw an "*ERC20: transfer amount exceeds balance*" error.

Impact:

This can lead to denial of transfer service: users will not be able to transfer tokens anymore.

Path: ./contracts/MarshaPlus.sol : transfer()

./contracts/MarshaPlus.sol : transferFrom()

./contracts/MarshaPlus.sol : teamRewardAfter3Years()

Recommendation: Remove the *teamRewardAfter3Years()* function from these two functions (*transfer()* and *transferfrom()*) to avoid blocking the correct flow. Additionally, it is possible to implement a pull over push pattern for the *teamRewardAfter3Years()* function.

Found in: 3194e22652cca3e7fa13485676a05be066c28ae4

Status:

Fixed (Revised commit: 5c1834d2996fb1915c53f33289c54d31b10a7b03)

■■■ High

No high severity issues were found.

■■ Medium

M01. Overriding inherited functions violation

Impact	Medium
Likelihood	Medium

The contract overrides the *transfer()* and *transferFrom()* functions, introducing extra functionality. In general, these functions are expected to return boolean values to signify the outcome of the function call. Nevertheless, in the current implementation, these overridden functions return their own boolean values instead of the boolean values returned by the corresponding functions in the superclass.

Impact:

Every token transfer will have a significant additional Gas cost. The transfer functionality will be dependent on the good implementation of the two functions that they call, which will present a risk. In this specific case, the issue described above in C01 will cause a total denial of service..

Path: ./contracts/MarshaPlus.sol : transfer(), transferFrom()

Recommendation: Do not override transfer() nor transferFrom() functions, change *teamRewardAfter3Years()* and *burnIfNeeded()* to external functions.

Found in: 3194e22652cca3e7fa13485676a05be066c28ae4

Status:

Fixed (Revised commit: 5c1834d2996fb1915c53f33289c54d31b10a7b03)

■ Low

L01. Missing zero address validation

Impact	Medium
Likelihood	Low

The zero address validation check is not implemented for the following function: *constructor()*

Setting one of aforementioned parameters to zero address (0x0) results in breaking main business flow.

Impact:

If an address passed into the *constructor()* would be the zero address, some amount of tokens will not be distributed correctly and will be lost.

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

Recommendation: Implement a zero address validation for the given parameters.

Found in: 3194e22652cca3e7fa13485676a05be066c28ae4

Status:

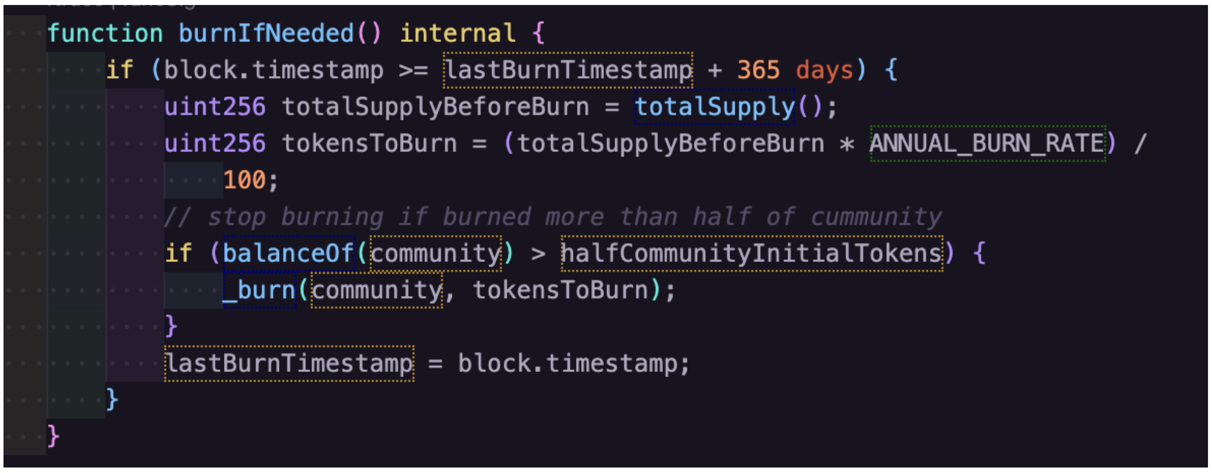
Fixed (Revised commit: 5c1834d2996fb1915c53f33289c54d31b10a7b03)

L02. Incorrect state variables updating

Impact	Low
Likelihood	Medium

The *lastBurnTimestamp* variable can only be updated if the *burnIfNeeded()* function is called, but the burn itself will only occur if the community token balance is greater than or equal to half of the original amount of minted coins.

This variable will be updated every year, regardless of whether tokens were burned or not.



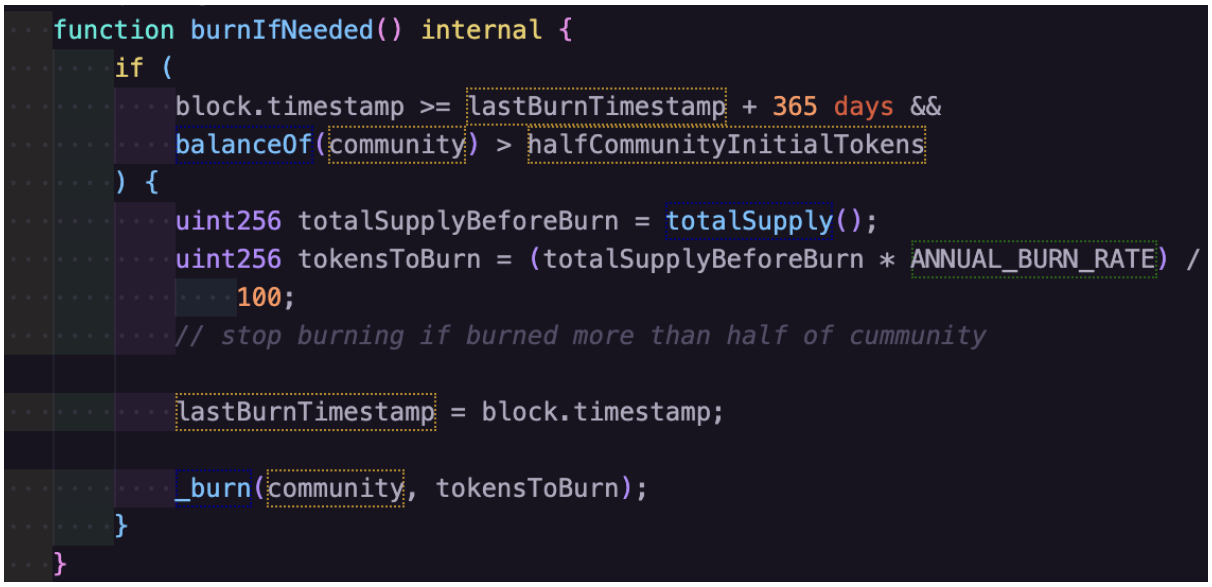
```
function burnIfNeeded() internal {
    if (block.timestamp >= lastBurnTimestamp + 365 days) {
        uint256 totalSupplyBeforeBurn = totalSupply();
        uint256 tokensToBurn = (totalSupplyBeforeBurn * ANNUAL_BURN_RATE) /
            100;
        // stop burning if burned more than half of cummunity
        if (balanceOf(community) > halfCommunityInitialTokens) {
            _burn(community, tokensToBurn);
        }
        lastBurnTimestamp = block.timestamp;
    }
}
```

Impact:

This issue can lead to misunderstanding of contract state variables.

Path: ./contracts/MarshaPlus.sol : burnIfNeeded()

Recommendation: Combine both if statements in the *burnIfNeeded()* function to avoid redundant calls if these statements are not equal to true:



```
function burnIfNeeded() internal {
    if (
        block.timestamp >= lastBurnTimestamp + 365 days &&
        balanceOf(community) > halfCommunityInitialTokens
    ) {
        uint256 totalSupplyBeforeBurn = totalSupply();
        uint256 tokensToBurn = (totalSupplyBeforeBurn * ANNUAL_BURN_RATE) /
            100;
        // stop burning if burned more than half of community

        lastBurnTimestamp = block.timestamp;

        _burn(community, tokensToBurn);
    }
}
```

Found in: 3194e22652cca3e7fa13485676a05be066c28ae4

Status:

Fixed (Revised commit: 5c1834d2996fb1915c53f33289c54d31b10a7b03)

Informational

I01. Floating pragma used in the contract

The project uses floating pragma ^0.8.0.

Impact:

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: ./contracts/MarshaPlus.sol

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: 3194e22652cca3e7fa13485676a05be066c28ae4

Status:

Fixed (Revised commit: 5c1834d2996fb1915c53f33289c54d31b10a7b03)

I02. State variables can be declared immutable

Compared to regular state variables, the gas costs of constant and immutable variables are much lower. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed.

Variables' values are set in the constructor.
These variables: *community*, *charity*, *foundation*, *development*, *marketing*, *investors*, *legal*, *expansion*, *halfCommunityInitialTokens* can be declared immutable.

Impact:

This will result in increased Gas cost during the deployment.

Path: ./contracts/MarshaPlus.sol

Recommendation: Declare the mentioned variables as immutable.

Found in: 3194e22652cca3e7fa13485676a05be066c28ae4

Status:

Fixed (Revised commit: 5c1834d2996fb1915c53f33289c54d31b10a7b03)

I03. State variable default visibility is not set

Labeling the visibility explicitly makes it easier to catch incorrect assumptions about who can access the variable.

The *halfCommunityInitialTokens* variable has no declared visibility.

Path: ./contracts/MarshaPlus.sol

Recommendation: Variables can be specified as being public, internal or private. Explicitly define visibility for all state variables.

Found in: 3194e22652cca3e7fa13485676a05be066c28ae4

Status:

Fixed (Revised commit: 5c1834d2996fb1915c53f33289c54d31b10a7b03)

I04. Style guide violation

Contract readability and code quality are influenced significantly by adherence to established style guidelines. In Solidity programming, there exist certain norms for code arrangement and ordering. These guidelines help to maintain a consistent structure across different contracts, libraries, or interfaces, making it easier for developers and auditors to understand and interact with the code.

The suggested order of elements within each contract, library, or interface is as follows:

1. Type declarations
2. State variables
3. Events

4. Modifiers
5. Functions

Functions should be ordered and grouped by their visibility as follows:

1. Constructor
2. Receive function (if exists)
3. Fallback function (if exists)
4. External functions
5. Public functions
6. Internal functions
7. Private functions

Within each grouping, *view* and *pure* functions should be placed at the end.

Furthermore, following the Solidity naming convention and adding NatSpec annotations for all functions are strongly recommended. These measures aid in the comprehension of code and enhance overall code quality.

Path: ./contracts

Recommendation: Consistent adherence to the official Solidity style guide is recommended. This enhances readability and maintainability of the code, facilitating seamless interaction with the contracts. Providing comprehensive NatSpec annotations for functions and following Solidity's naming conventions further enrich the quality of the code.

This issue will be fixed automatically by removing the two overridden functions as recommended in C01 and M01.

Found in: 3194e22652cca3e7fa13485676a05be066c28ae4

Status:

Fixed (Revised commit: 5c1834d2996fb1915c53f33289c54d31b10a7b03)

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	https://github.com/IziumenkoViacheslav/MarshaPlusSolidityContract
Commit	3194e22652cca3e7fa13485676a05be066c28ae4
Whitepaper	https://www.marshafoundation.org/
Requirements	https://www.marshafoundation.org/_files/ugd/4c311c_dfc3c09361d549e1b30ad927f80fe2b0.pdf
Technical Requirements	-
Contracts	File: ./MarshaPlus.sol SHA3: cf2c0ed316850cc249bf9eec145111f790b8aa08cbf3122a6f6a7909633ed535

Second review scope

Repository	https://github.com/IziumenkoViacheslav/MarshaPlusSolidityContract
Commit	5c1834d2996fb1915c53f33289c54d31b10a7b03
Whitepaper	https://www.marshafoundation.org/
Requirements	https://www.marshafoundation.org/_files/ugd/4c311c_dfc3c09361d549e1b30ad927f80fe2b0.pdf
Technical Requirements	https://github.com/IziumenkoViacheslav/MarshaPlusSolidityContract/blob/master/README.md
Contracts	File: ./MarshaPlus.sol SHA3: 78be093cad7cb337dc027894d447f1a98f5b7bc0c819db89c262ee7c27196ded