Quantstamp Security Assessment Certificate

October 13th 2021 — Quantstamp Verified

Mars4 ERC20

This audit report was prepared by Quantstamp, the leader in blockchain security.

Executive Summary

Type	ERC20
Auditors	Jan Gorzny, Blockchain Researcher Hisham Galal, Research Engineer Roman Rohleder, Research Engineer
Timeline	2021-10-05 through 2021-10-05
EVM	London
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	None
Documentation Quality	Low
Test Quality	Low
Source Code	

Repository	Commit
erc20	08a5ce2

Total Issues	**5** (0 Resolved)
High Risk Issues	0 (0 Resolved)
Medium Risk Issues	0 (0 Resolved)
Low Risk Issues	**5** (0 Resolved)
Informational Risk Issues	0 (0 Resolved)
Undetermined Risk Issues	0 (0 Resolved)

0 Unresolved
5 Acknowledged
0 Resolved

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for client's reputation or serious financial implications for client and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental for the client's reputation if exploited, or is reasonably likely to lead to moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low-impact in view of the client's business circumstances.
Informational	The issue does not post an immediate risk, but is relevant to security best practices or Defence in Depth.
Undetermined	The impact of the issue is uncertain.

Unresolved	Acknowledged the existence of the risk, and decided to accept it without engaging in special efforts to control it.
Acknowledged	The issue remains in the code but is a result of an intentional business or design decision. As such, it is supposed to be addressed outside the programmatic means, such as: 1) comments, documentation, README, FAQ; 2) business processes; 3) analyses showing that the issue shall have no negative consequences in practice (e.g., gas analysis, deployment settings).
Resolved	Adjusted program implementation, requirements or constraints to eliminate the risk.
Mitigated	Implemented actions to minimize the impact or likelihood of the risk.

Summary of Findings

The Mars4 ERC20 is a slight modification to the ERC20 standard code written by OpenZeppelin. A number of small issues are present, which result from development processes (QSP-1, QSP-2) or unavoidable concerns of using a token on Ethereum (QSP-3). QSP-4 and QSP-5 are avoidable and should be addressed before the contract is deployed. Two best practices could also be implemented. Although the modifications to the base system are very small, the project has no tests, and therefore the changes are not covered by tests. Quantstamp, as always, strongly recommends adding tests to safeguard against unpredictable code.

ID	Description	Severity	Status
QSP-1	Unlocked Pragma	Low	Acknowledged
QSP-2	Clone-and-Own	Low	Acknowledged
QSP-3	Race Conditions / Front-Running	Low	Acknowledged
QSP-4	Greedy Contract	Low	Acknowledged
QSP-5	Allowance Double-Spend Exploit	Low	Acknowledged

Quantstamp Audit Breakdown

Quantstamp's objective was to evaluate the repository for security-related issues, code quality, and adherence to specification and best practices.

Possible issues we looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Mishandled exceptions and call stack limits
- Unsafe external calls
- Integer overflow / underflow
- Number rounding errors
- Reentrancy and cross-function vulnerabilities
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting

Methodology

The Quantstamp auditing process follows a routine series of steps:

1. Code review that includes the following
   i. Review of the specifications, sources, and instructions provided to Quantstamp to make sure we understand the size, scope, and functionality of the smart contract.
   ii. Manual review of code, which is the process of reading source code line-by-line in an attempt to identify potential vulnerabilities.
   iii. Comparison to specification, which is the process of checking whether the code does what the specifications, sources, and instructions provided to Quantstamp describe.
2. Testing and automated analysis that includes the following:
   i. Test coverage analysis, which is the process of determining whether the test cases are actually covering the code and how much code is exercised when we run those test cases.
   ii. Symbolic execution, which is analyzing a program to determine what inputs cause each part of a program to execute.
3. Best practices review, which is a review of the smart contracts to improve efficiency, effectiveness, clarify, maintainability, security, and control based on the established industry and academic practices, recommendations, and research.
4. Specific, itemized, and actionable recommendations to help you take steps to secure your smart contracts.

Toolset

The notes below outline the setup and steps performed in the process of this audit.

Setup

Tool Setup:

- Slither v0.6.6

Steps taken to run the tools:

Installed the Slither tool: `pip install slither-analyzer` Run Slither from the project directory: `slither .`

Findings

QSP-1 Unlocked Pragma

Severity: Low Risk

Status: Acknowledged

File(s) affected: `Mars4ERC20.sol`

Description: Every Solidity file specifies in the header a version number of the format `pragma solidity (^)0.4.*`. The caret (^) before the version number implies an unlocked pragma, meaning that the compiler will use the specified version *and above*, hence the term "unlocked".

Recommendation: For consistency and to prevent unexpected behavior in the future, it is recommended to remove the caret to lock the file onto a specific Solidity version.

QSP-2 Clone-and-Own

Severity: Low Risk

Status: Acknowledged

File(s) affected: `Mars4ERC20.sol`

Description: The clone-and-own approach involves copying and adjusting open source code at one's own discretion. From the development perspective, it is initially beneficial as it reduces the amount of effort. However, from the security perspective, it involves some risks as the code may not follow the best practices, may contain a security vulnerability, or may include intentionally or unintentionally modified upstream libraries. Rather than the clone-and-own approach, a good industry practice is to use the Truffle framework for managing library dependencies. This eliminates the clone-and-own risks yet allows for following best practices, such as, using libraries.

Recommendation: Use the appropriate dependencies, rather than copy the code directly.

QSP-3 Race Conditions / Front-Running

Severity: Low Risk

Status: Acknowledged

File(s) affected: `Mars4ERC20.sol`

Description: A block is an ordered collection of transactions from all around the network. It's possible for the ordering of these transactions to manipulate the end result of a block. A miner attacker can take advantage of this by generating and moving transactions in a way that benefits themselves.

Recommendation: Make sure users are aware of this ubiquitous Ethereum issue.

QSP-4 Greedy Contract

Severity: Low Risk

Status: Acknowledged

File(s) affected: `Mars4ERC20.sol`

Description: A greedy contract is a contract that can receive ether which can never be redeemed.

Recommendation: In accordance with best practices, to prevent tokens being accidentally stuck in the ERC20 contract itself, it is recommended to prevent the transferal of tokens to the contracts address. This can be achieved, by i.e. adding require statements to transfer functions, similar to `require(to != address(this));`.

QSP-5 Allowance Double-Spend Exploit

Severity: Low Risk

Status: Acknowledged

Description: As it presently is constructed, the contract is vulnerable to the allowance double-spend exploit, as with other ERC20 tokens.

Exploit Scenario:

1. Alice allows Bob to transfer `N` amount of Alice's tokens (`N>0`) by calling the `approve()` method on `Token` smart contract (passing Bob's address and `N` as method arguments)
2. After some time, Alice decides to change from `N` to `M` (`M>0`) the number of Alice's tokens Bob is allowed to transfer, so she calls the `approve()` method again, this time passing Bob's address and `M` as method arguments
3. Bob notices Alice's second transaction before it was mined and quickly sends another transaction that calls the `transferFrom()` method to transfer `N` Alice's tokens somewhere
4. If Bob's transaction will be executed before Alice's transaction, then Bob will successfully transfer `N` Alice's tokens and will gain an ability to transfer another `M` tokens
5. Before Alice notices any irregularities, Bob calls `transferFrom()` method again, this time to transfer `M` Alice's tokens.

Recommendation: The exploit (as described above) is mitigated through use of functions that increase/decrease the allowance relative to its current value, such as `increaseAllowance()` and `decreaseAllowance()`.
Pending community agreement on an ERC standard that would protect against this exploit, we recommend that developers of applications dependent on `approve()` / `transferFrom()` should keep in mind that they have to set allowance to 0 first and verify if it was used before setting the new value. Teams who decide to wait for such a standard should make these recommendations to app developers who work with their token contract.

Automated Analyses

Slither

Slither found no serious issues. Those present involved the trustworthiness of the relatively new version of Solidity used (8) and the suggestion to change the visibility of many functions to `external`.

Adherence to Best Practices

1. `Mars4ERC20.sol`: For better disambiguation and explicitness the mint amount in line 508 should be considered to be changed from `_mint(msg.sender, 4_000_000_000 ether);` to `_mint(msg.sender, 4_000_000_000 * 10**18);`.

Test Results

Test Suite Results

There are no tests.

Code Coverage

There are no tests and therefore no lines of code are covered by tests.

Appendix

File Signatures

The following are the SHA-256 hashes of the reviewed files. A file with a different SHA-256 hash has been modified, intentionally or otherwise, after the security review. You are cautioned that a different SHA-256 hash could be (but is not necessarily) an indication of a changed condition or potential vulnerability that was not within the scope of the review.

Contracts

`8d0e012fb96e0e97c249050c53dbd8058f2d0dd875a097eb2b70b15a2b1d3224 ./contracts/Mars4ERC20.sol`

`4fd6092bdfa8b42f19d535c5ac69c4323b0b894717c699e58d5552eeabd04cd4 ./contracts/Migrations.sol`

Changelog

- 2021-10-05 - Initial report [`08a5ce2`]

About Quantstamp

Quantstamp is a Y Combinator-backed company that helps to secure blockchain platforms at scale using computer-aided reasoning tools, with a mission to help boost the adoption of this exponentially growing technology.

With over 1000 Google scholar citations and numerous published papers, Quantstamp's team has decades of combined experience in formal verification, static analysis, and software verification. Quantstamp has also developed a protocol to help smart contract developers and projects worldwide to perform cost-effective smart contract security scans.

To date, Quantstamp has protected $5B in digital asset risk from hackers and assisted dozens of blockchain projects globally through its white glove security assessment services. As an evangelist of the blockchain ecosystem, Quantstamp assists core infrastructure projects and leading community initiatives such as the Ethereum Community Fund to expedite the adoption of blockchain technology.

Quantstamp's collaborations with leading academic institutions such as the National University of Singapore and MIT (Massachusetts Institute of Technology) reflect our commitment to research, development, and enabling world-class blockchain security.