Quantstamp Security Assessment Certificate

QUANTSTAMP VERIFIED SECURITY CERTIFICATE

November 12th 2020 — Quantstamp Verified

eXRD Token

This security assessment was prepared by Quantstamp, the leader in blockchain security

Executive Summary

Type	Audit
Auditors	Luís Fernando Schultz Xavier da Silveira, Security Consultant Kacper Bąk, Senior Research Engineer Fayçal Lalidji, Security Auditor
Timeline	2020-10-27 through 2020-10-28
EVM	Muir Glacier
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
eXRD-token	37f62d7

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.

Total Issues	4 (0 Resolved)
High Risk Issues	0 (0 Resolved)
Medium Risk Issues	0 (0 Resolved)
Low Risk Issues	0 (0 Resolved)
Informational Risk Issues	4 (0 Resolved)
Undetermined Risk Issues	0 (0 Resolved)

4 Unresolved
0 Acknowledged
0 Resolved

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

We have discovered no vulnerabilities but have raised a few informational issues and made some suggestions for code improvement. The scope of the audit was the following contracts:

- `contracts/eXRD.sol`
- `contracts/Vault.sol` (diff from the previous audit)
- `contract-dependencies/Roles.sol`
- `contract-dependencies/OwnerRole.sol`
- `contract-dependencies/MinterRole.sol`
- `contract-dependencies/ERC20Mintable.sol`

ID	Description	Severity	Status
QSP-1	Privileged Roles and Ownership	○ Informational	Unresolved
QSP-2	Clone-and-Own	○ Informational	Unresolved
QSP-3	Unlocked Pragma	○ Informational	Unresolved
QSP-4	Allowance Double-Spend Exploit	○ Informational	Unresolved

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 0.6.13
- Mythril v0.22.10

Steps taken to run the tools:

1. Install the Slither tool: `pip3 install slither-analyzer`
2. Run Slither from the project directory: `slither path/to/contract`
3. Install the Mythril tool: `pip3 install mythril`
4. Run the Mythril tool on each contract: `myth analyze path/to/contract`

Findings

QSP-1 Privileged Roles and Ownership

Severity: *Informational*

Status: Unresolved

File(s) affected: `eXRD.sol`, `Vault.sol`

Description: The owners and minters of the eXRD contract can mint tokens arbitrarily, with owners being able to add new minters and owners. Furthermore, the owner of the Vault contract has almost complete control over allocations, being able to even revoke them. The funding account also has the power to deny the payment of allocations.

Recommendation: We recommend adequately informing users about this centralization of power.

QSP-2 Clone-and-Own

Severity: *Informational*

Status: Unresolved

File(s) affected: `Roles.sol`, `OwnerRole.sol`, `MinterRole.sol`, `ERC20Mintable.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.

Recommendation: 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.

QSP-3 Unlocked Pragma

Severity: *Informational*

Status: Unresolved

File(s) affected: `Roles.sol`, `OwnerRole.sol`, `MinterRole.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. Furthermore, this version should be the same among the files.

QSP-4 Allowance Double-Spend Exploit

Severity: *Informational*

Status: Unresolved

File(s) affected: `eXRD.sol`

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

Exploit Scenario: An example of an exploit goes as follows:

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

Here is the output from Slither we deemed relevant:

```
MinterRole._removeMinter(address).index (contract-dependencies/MinterRole.sol#37) is a local variable never initialized
OwnerRole._removeOwner(address).index (contract-dependencies/OwnerRole.sol#43) is a local variable never initialized
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-local-variables
renounceMinter() should be declared external:
    - MinterRole.renounceMinter() (contract-dependencies/MinterRole.sol#25-27)
addOwner(address) should be declared external:
    - OwnerRole.addOwner(address) (contract-dependencies/OwnerRole.sol#25-27)
renounceOwner() should be declared external:
    - OwnerRole.renounceOwner() (contract-dependencies/OwnerRole.sol#29-31)
    - Vault.release(uint256) (contracts/Vault.sol#397-432)
getAllocationInGroup(uint256,uint256) should be declared external:
    - Vault.getAllocationInGroup(uint256,uint256) (contracts/Vault.sol#442-451)
getBeneficiaryAllocations(address,uint256) should be declared external:
    - Vault.getBeneficiaryAllocations(address,uint256) (contracts/Vault.sol#453-466)
getNbrOfAllocations(address) should be declared external:
    - Vault.getNbrOfAllocations(address) (contracts/Vault.sol#468-470)
getRegisteredBeneficiaries(uint256) should be declared external:
    - Vault.getRegisteredBeneficiaries(uint256) (contracts/Vault.sol#472-474)
getIsRegistered(address) should be declared external:
    - Vault.getIsRegistered(address) (contracts/Vault.sol#476-478)
getUnlockedPercentages(uint256) should be declared external:
    - Vault.getUnlockedPercentages(uint256) (contracts/Vault.sol#480-482)
getGroup(uint256) should be declared external:
    - Vault.getGroup(uint256) (contracts/Vault.sol#484-495)
getGroupCount() should be declared external:
    - Vault.getGroupCount() (contracts/Vault.sol#497-499)
getFundingAccount() should be declared external:
    - Vault.getFundingAccount() (contracts/Vault.sol#501-503)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
Vault.revoke(address,uint256) (contracts/Vault.sol#124-155) ignores return value by token.transfer(fundingAccount,toBeReleased) (contracts/Vault.sol#152)
Vault.revokeBeneficiariesInGroup(address[],uint256) (contracts/Vault.sol#160-213) ignores return value by token.transfer(fundingAccount,totalAmount) (contracts/Vault.sol#212)
Vault.revokeGroup(uint256) (contracts/Vault.sol#218-243) ignores return value by token.transfer(fundingAccount,totalAmount) (contracts/Vault.sol#242)
Vault.addAllocations(address[],uint256[],uint256[]) (contracts/Vault.sol#292-358) ignores return value by token.transferFrom(fundingAccount,address(this),totalAmount) (contracts/Vault.sol#357)
Vault.release(uint256) (contracts/Vault.sol#397-432) ignores return value by token.transfer(msg.sender,releasable) (contracts/Vault.sol#431)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

Mythril

Mythril found no issues with the contracts.

Code Documentation

1. File `ERC20Mintable.sol`, line 11: that is not the case.

Adherence to Best Practices

1. See the unattended items in the Adherence to Best Practices Review in our previous report.
2. Consider using `SafeMath.add` in lines 205, 236 and 352 of `Vault.sol`.
3. File `Vault.sol`: consider checking that `group_ < groupCount` in `getUnlockedPercentages` and `getGroup`.
4. Have `_owners.remove(account)` be at the beginning of `_removeOwner` in `OwnerRole.sol` so the error diagnostic is better if there are no owners. The same holds for `_minters.remove(account)` in function `_removeMinter` of file `MinterRole.sol`.

Test Results

Test Suite Results

The test infrastructure is not working and we were unable to run the tests.

Code Coverage

We could not run code coverage tools as the test infrastructure is not working.

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

```
aff548ab807d9f24845b03f6c01ecad24d771eb5ea31a5ee2c0b886e092023b6 ./xrd-token/contracts/Vault.sol
1b36680daba2bd7bc9494a363792ea47b477dad831b0db907611075df9122de5 ./xrd-token/contracts/eXRD.sol
56fa39d1c88f681db4b1bc3ee92d963630f7efea7b644d6fa0cd66510b2ca96b ./xrd-token/contract-dependencies/MinterRole.sol
9bdbfa011bff477690c4282dc4a6b0639dce178ea4945cf6fcbfdeadec58d383 ./xrd-token/contract-dependencies/Roles.sol
8b072fe6c464f2fac8cfeb6caeb888304379812ccfef69225032c14d7efe6a91 ./xrd-token/contract-dependencies/OwnerRole.sol
63bf32c5de539260dec4ee22a564a9208d61b2518e3ec0581e4b97695d48b2f2 ./xrd-token/contract-dependencies/ERC20Mintable.sol
```

Changelog

- 2020-10-28 - Initial report

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.