Quantstamp Security Assessment Certificate

November 12th 2020 — Quantstamp Verified

Radix eXRD vault

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 Kevin Feng, Blockchain Researcher Fayçal Lalidji, Security Auditor
Timeline	2020-09-23 through 2020-10-01
EVM	Muir Glacier
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Computer-Aided Verification, Manual Review
Specification	None
Documentation Quality	Medium
Test Quality	Low

Source Code

Repository	Commit
eXRD-token	37f62d7

Total Issues	**13** (10 Resolved)
High Risk Issues	0 (0 Resolved)
Medium Risk Issues	**2** (2 Resolved)
Low Risk Issues	**6** (5 Resolved)
Informational Risk Issues	**5** (3 Resolved)
Undetermined Risk Issues	0 (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

No major security issues were found. However, that is only if the contract owner understands the contract code well and is very careful with functions calls, as there are a number of usability issues we documented. The contract centralizes a great deal of power in its owner, which is not necessarily a problem provided users are well informed of this fact. We have also made several suggestions for code improvement, particularly with regards to adherence to best practices.

ID	Description	Severity	Status
QSP-1	Funds transfer from revoked allocation	Medium	Fixed
QSP-2	Gas Usage / `for` Loop Concerns	Medium	Mitigated
QSP-3	Possible return of funds to incorrect account	Low	Acknowledged
QSP-4	Gas Usage / `for` Loop Concerns	Low	Fixed
QSP-5	Potentially incorrect results from `releasableAmount`	Low	Fixed
QSP-6	Unlocking inexistent groups	Low	Fixed
QSP-7	Enabling of completed groups	Low	Fixed
QSP-8	Transacting to zero funding account	Low	Fixed
QSP-9	Better check for group existence	Informational	Fixed
QSP-10	Empty allocations	Informational	Fixed
QSP-11	Gas Usage / `for` Loop Concerns	Informational	Fixed
QSP-12	Centralization of power	Informational	Acknowledged
QSP-13	Allowance Double-Spend Exploit	Informational	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 0.6.13
- Mythril 0.22.9

Steps taken to run the tools:

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

Findings

QSP-1 Funds transfer from revoked allocation

Severity: *Medium Risk*

Status: Fixed

File(s) affected: `Vault.sol`

Description: In functions `revoke`, `revokeBeneficiariesInGroup` and `revokeGroup`, a check that the allocation being revoked has not been revoked before is missing. As a result, if the owner makes a mistake and revokes such an allocation more than once, the remaining funds to be released will be transferred from the contract to the funding account, possibly violating the invariant that the contract holds enough funds to cover all allocations.

Recommendation: Do not perform operations on already revoked allocations.

QSP-2 Gas Usage / `for` Loop Concerns

Severity: *Medium Risk*

Status: Mitigated

File(s) affected: `Vault.sol`

Description: Function `revokeGroup` uses too much gas even for small groups of allocations because it iterates over the whole set of allocations.

Recommendation: To mitigate this issue, instead of having a `registeredBeneficiaries` dynamic array, consider having a dynamic array of allocation references (beneficiary and allocation index) in the `Group` struct. When an allocation is added (to a group), push the reference there. Like this the function would only iterate over allocations in the group. We also recommend the operator of the contract to be made aware of this issue. To ensure gas consumption is under control and that the operator is aware, a range over the "array" `allocations` in struct `Group` could be passed to the `revokeGroup` function.

QSP-3 Possible return of funds to incorrect account

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `Vault.sol`

Description: If the funding account is changed after some allocations have been made, the new funding account will receive the tokens coming out of revocations even though they originally belonged to another funding account.

Recommendation: Make sure this is the expected behavior or pass the funding account to the constructor, so it is initialized exactly once.

QSP-4 Gas Usage / `for` Loop Concerns

Severity: *Low Risk*

Status: Fixed

File(s) affected: `Vault.sol`

Description: Function `addAllocations` should only push `beneficiary` onto `registeredBeneficiaries` if it has not yet been registered, or there will be unnecessary copies of it in the array.

Recommendation: Check that `!isRegistered[beneficiary]`.

QSP-5 Potentially incorrect results from `releasableAmount`

Severity: *Low Risk*

Status: Fixed

File(s) affected: `Vault.sol`

Description: The function does not check that the allocation hasn't been revoked. It arguably should return zero in such cases or throw.

QSP-6 Unlocking inexistent groups

Severity: *Low Risk*

Status: Fixed

File(s) affected: `Vault.sol`

Description: The function `unlock` should not work on groups that do not exist.

Recommendation: Check that `group < groupCount`.

QSP-7 Enabling of completed groups

Severity: *Low Risk*

Status: Fixed

File(s) affected: `Vault.sol`

Description: The function `enableGroup` should probably check that `unlockedPercentages[group] < 100` as the `unlock` function will disable completed groups.

QSP-8 Transacting to zero funding account

Severity: *Low Risk*

Status: Fixed

File(s) affected: `Vault.sol`

Description: Given that `addAllocations` checks that `fundingAccount != address(0)`, most likely the `revoke*` family of functions should also, as they transact with such account.

QSP-9 Better check for group existence

Severity: *Informational*

Status: Fixed

File(s) affected: `Vault.sol`

Description: Checking that `group < groupCount` is simpler than `bytes(groups[group].name).length`.

QSP-10 Empty allocations

Severity: *Informational*

Status: Fixed

File(s) affected: `Vault.sol`

Description: The function `addAllocations` should probably check that `amounts[i] > 0`.

QSP-11 Gas Usage / `for` Loop Concerns

Severity: *Informational*

Status: Fixed

File(s) affected: `Vault.sol`

Description: The `addAllocations` function could tally the amount to take from the funding account and perform a single transfer. The same principle applies to `revokeBeneficiariesInGroup` and `revokeGroup`.

QSP-12 Centralization of power

Severity: *Informational*

Status: Acknowledged

File(s) affected: `Vault.sol`

Description: The owner of the contract has power to unilaterally withhold tokens from beneficiaries.

Recommendation: We recommend properly informing users that can be beneficiaries of this fact.

QSP-13 Allowance Double-Spend Exploit

Severity: *Informational*

Status: Acknowledged

File(s) affected: `Vault.sol`

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

Under the assumption that `token` in `Vault.sol` is a trusted contract, Slither found no issues undocumented here.

Mythril

Mythril found no issues.

Adherence to Best Practices

1. The ERC20 standard specifies the return of a boolean from the `transfer` and `transferFrom` methods and that it should be checked. The `ERC20FixedSupply` contract always returns `true` from them (unless they revert), so there is no risk here. However, it would be considered best practice to check these return values in lines 120, 155, 190, 277 and 349.

2. When a group is added using `addGroup`, the group ID is not selected by the owner but automatically. Since the `GroupAdded` event does not keep track of this ID, it can be hard to track the added groups. Please note that functions dealing with groups receive an ID and not a name and also that there can be many groups with the same name.

3. It is customary for Solidity programs to name their function arguments with an underscore at the end. This convention helps code readability. This is specially true in the `addAllocations` function.

4. **(Fixed)** It should probably be checked that `beneficiaries[i] != address(0)` and/or `isRegistered[beneficiaries[i]]` in `revokeBeneficiariesInGroup`. Similar checks could be inserted in the `revoke` function.

5. **(Fixed)** The name of the argument to the `AllocationFullyReleased` event should probably be `allocationIndex` rather than `group` (see `L346`).

6. Checking that `_token != address(0)` in constructor could help prevent human error. Likewise for `account` in `setFundingAccount`.

7. The storage map `nbrOfAllocations` is redundant. For an address `beneficiary`, we have the invariant `nbrOfAllocations[beneficiary] == beneficiaryAllocations[beneficiary].length`.

8. Similarly, `groups` could be made into a dynamic array, making `groupCount` unnecessary.

9. The storage map `isRegistered` is also redundant because, as beneficiaries are only registered when allocations are made, we have the invariant `isRegistered[beneficiary] == true` if, and only if, `beneficiaryAllocations[beneficiary].length > 0`.

10. The storage map `unlockedPercentage` could have been integrated as a struct field in `Group`.

Test Results

Test Suite Results

All tests are passing. Note the warning about Truffle migrations.

```
Compiled 9 contracts successfully


  Contract: Vault
Your project has Truffle migrations, which have to be turn into a fixture to run your tests with Buidler
    Use cases
      Add allocation, vest 10%, release, vest 20%, release, revoke
         ✓ should add allocation (229ms)
         ✓ should be impossible to release any tokens
         ✓ should vest 10% (154ms)
         ✓ should be possible to release tokens after vesting (119ms)
         ✓ should be impossible to release again
         ✓ should vest 20% (175ms)
         ✓ should be possible to release tokens after vesting again (122ms)
         ✓ should revoke (115ms)

  Contract: ERC20FixedSupply
     ✓ should have the name XXRR
     ✓ should have 18 decimals
     ✓ should mint (50ms)

  Contract: Vault
    addAllocations
       ✓ should register allocations
       ✓ should set beneficiary as registered (140ms)
       ✓ should set nbrOfAllocations (129ms)
       ✓ should revert if sender is not the owner (115ms)
       ✓ should revert if beneficiary is the zero address (49ms)
       ✓ should revert if group is not registered (91ms)
    vest
       ✓ should vest for group 0 (12042ms)
       ✓ should vest for group 1 (10987ms)
       ✓ should vest for group 2 (9442ms)
    release
       ✓ should be able to release tokens after vesting (331ms)
       ✓ should be able to release all tokens after 100% is vested (447ms)
       ✓ should revert if allocation has been revoked (180ms)
       ✓ should revert if allocation index is invalid (42ms)
    revoke
       ✓ should revoke an allocation (77ms)
       ✓ should revert if not called by owner (42ms)

  Contract: Vault
    Gas tests
      adding allocations
         ✓ should add a lot of allocations (4633ms)


  27 passing (44s)

Done in 76.74s.
```

Code Coverage

We were unable to run code coverage tests. We strongly recommend the developers to perform this step.

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

```
449a6da8a797553832affa3fe74d6e1b942661f0989c05292bb939e0330673cc ./contracts/ERC20FixedSupply.sol

f882315963fbf57267b11d920ffd0104da9b65c23fc575152314be9e21d03afd ./contracts/Vault.sol
```

Changelog

- 2020-10-01 - Initial report
- 2020-10-08 - Final 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.