Quantstamp Security Assessment Certificate

May 27th 2022 — Quantstamp Verified

Bitcoin.com

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

Executive Summary

Type	Token Distribution Contract
Auditors	David Knott, Senior Research Engineer Alejandro Padilla Gaeta, Research Engineer Rabib Islam, Research Engineer
Timeline	2022-04-22 through 2022-05-27
EVM	Arrow Glacier
Languages	Solidity
Methods	Architecture Review, Unit Testing, Computer-Aided Verification, Manual Review
Specification	None
Documentation Quality	Low
Test Quality	High

Source Code

Repository	Commit
verse-token-contracts	0309a48
verse-token-contracts	99bc608
verse-token-contracts	3950501

Total Issues	**7** (5 Resolved)
High Risk Issues	0 (0 Resolved)
Medium Risk Issues	0 (0 Resolved)
Low Risk Issues	**2** (1 Resolved)
Informational Risk Issues	**4** (4 Resolved)
Undetermined Risk Issues	**1** (0 Resolved)

All Branches Covered

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).
Fixed	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

Quantstamp has performed a security audit of the `VerseToken` and `VerseClaimer` contracts and has identified 7 security issues ranging from Low to Undetermined risk levels. Additionally, we have found 3 documentation and 7 best practice issues. We recommend addressing all issues before deploying the smart contracts in production.

ID	Description	Severity	Status
QSP-1	Transfers To Zero Address	⌄ Low	Acknowledged
QSP-2	Undocumented Token Distribution Funding	⌄ Low	Fixed
QSP-3	Missing Constructor Argument Validation	○ Informational	Fixed
QSP-4	Gas Usage / Loop Concerns	○ Informational	Fixed
QSP-5	Untrusted Token Double Claim	○ Informational	Fixed
QSP-6	Signature Malleability	○ Informational	Fixed
QSP-7	Unrestricted Burns	? Undetermined	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 v0.8.3

Steps taken to run the tools:

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

Findings

QSP-1 Transfers To Zero Address

Severity: *Low Risk*

Status: Acknowledged

File(s) affected: `VerseToken.sol`

Description: Operations in the `VerseToken` contract do not verify that the addresses tokens are sent to are not `address(0)`. In particular, this might be a problem for the `_transfer` function as any tokens sent by mistake to the `address(0)` will be irrecoverable.

Recommendation: Validate that the `VerseToken::L196(transfer)` and `VerseToken::L212(transferFrom)` operations are not transferring tokens to `address(0)`.

Update: The Bitcoin.com team acknowledges the issue and states that they have chosen not to check for `address(0)` token sends to save gas and that wallets interfacing with the `VerseToken` contract are expected to check for `address(0)` token sends.

QSP-2 Undocumented Token Distribution Funding

Severity: *Low Risk*

Status: Fixed

File(s) affected: `VerseClaimer.sol`

Description: Before participants can be enrolled (and have their tokens scraped), the `VerseClaimer` contract must have enough `VerseToken`s assigned to it; otherwise calls to `VerseHelper::L75(_checkVerseBalance)` will fail and participants will not be able to claim their tokens. It is unclear how and when the `VerseClaimer` will receive these funds.

Recommendation: Add technical and end-user documentation specifying how `VerseClaimer` will be funded with `VerseToken`s.

Update: `VerseToken`'s `constructor` was modified to instantiate the `VerseClaimer` contract and mint `_initialSupply` directly to it.

QSP-3 Missing Constructor Argument Validation

Severity: *Informational*

Status: Fixed

File(s) affected: `VerseClaimer.sol`

Description: `VerseClaimer::L24(constructor)` takes in `_verseTokenAddress` and `_merkleRoot` as arguments but does not check that `_verseTokenAddress` is a smart contract and that `_merkleRoot` is not a null byte string. The incorrect setting of any of `VerseClaimer::L24(constructor)`'s arguments would require a redeployment of `VerseClaimer`.

Recommendation: Modify `VerseClaimer::L24(constructor)` to require `_verseTokenAddress` to be a contract address and `_merkleRoot` to not be a null byte string. Consider using OpenZeppelin's `isContract()` function to perform the contract check.

QSP-4 Gas Usage / Loop Concerns

Severity: *Informational*

Status: Fixed

File(s) affected: `VerseClaimer.sol`

Related Issue(s): SWC-126, SWC-134

Description: Gas usage is a main concern for smart contract developers and users, since high gas costs may prevent users from wanting to use the smart contract. Even worse, some gas usage issues may prevent the contract from providing services entirely. For example, if a loop requires too much gas to finish processing, then it may prevent the contract from functioning correctly entirely.
The `VerseClaimer::L61(enrollRecipientBulk)` performs unbounded iteration over multiple arrays which may lead `VerseClaimer::L61(enrollRecipientBulk)` calls to run out of gas. Furthermore, the usage of multiple arrays increases the chances of caller error.

Recommendation: Add an upper bound to the number of enrollments that `VerseClaimer::L61(enrollRecipientBulk)` can process. Additionally, modify `VerseClaimer::L61(enrollRecipientBulk)` to receive a `struct` instead of a `list` for each parameter to increase readability and safety of usage.

Update: An upper bound of 10 enrollments was added to `enrollRecipientBulk`.

QSP-5 Untrusted Token Double Claim

Severity: *Informational*

Status: Fixed

File(s) affected: `VerseClaimer.sol`

Description: This issue has been classified as informational as it is only relevant if a malicious ERC20 is used as `VerseClaim`'s `verseToken` instead of the `VerseToken` smart contract.
A reentrancy exploit can occur when external contract calls are made. To protect against reentrancy, it is recommended to order one's functions by:

- **checks** - Perform validation checks.
- **effects** - Update all contract state.
- **interactions** - Make external contract calls.

The above ordering protects one's functions from reentrancy because by the time external contract calls are made and execution is given to a potentially untrusted contract, all contract state is already in the most up-to-date state.
Double claims are protected against by requiring `keeperList[_recipient].keeperTill` to be equal to 0. `VerseClaimer::L125(_allocateTokens)` performs an external contract call to `verseToken` to check `VerseClaimer`'s token balance prior to setting `keeperTill`. If an attacker can acquire control of execution from `verseToken` they will be able to re-claim by calling `_enrollRecipient` multiple times before `keeperTill` is set.

Recommendation: Set `keeperTill` before checking `VerseClaimer`'s token balance to align with the checks/effects/interactions pattern. The balance check call should also use `staticcall` instead of `call` as it calls a non-state changing function.

Update: `VerseClaimers`'s `_allocateTokens`'s `_checkVerseBalance` external contract call was moved to be called after all the contract's internal state has been updated and is now in line with the checks/effects/interactions pattern.

QSP-6 Signature Malleability

Severity: *Informational*

Status: Fixed

File(s) affected: `VerseToken.sol`

Description: EIP-2 allows signature malleability for `ecrecover()`. `VerseToken::L233(permit)` accepts malleable signatures, allowing a signature different from the one an `owner` signed to be successfully submitted. This would make the permitting transaction hash different from the one the permit signer is expecting.

Recommendation: Modify `VerseToken::L233(permit)`'s `owner` signature check to require unique signatures by requiring signature s-values to be in the lower half of the potential range. The Ethereum Yellow paper (EY) defines s-value's full range as:

```
* 0 < s < secp256k1n ÷ 2 + 1 (EY 311)
* With secp256k1n = 115792089237316195423570985008687907852837564279074904382605163141518161494337 (EY 313)
```

Open Zeppelin's ECDSA contract which checks signature uniqueness can be used as a reference.

Update: `VerseToken`'s `permit` checks for and reverts if a signature's s-value is in the upper half of its potential range. This disables signature malleability.

QSP-7 Unrestricted Burns

Severity: *Undetermined*

Status: Acknowledged

File(s) affected: `VerseToken.sol`

Description: `VerseToken::L78(burn)`is not restricted; anyone can call it to burn their tokens. While this functionality is okay by itself, the Verse leadership's supply mechanism might be affected if too many tokens are burnt without their consent (the Verse white paper describes that there is a carefully designed supply mechanism planned).

Recommendation: Evaluate if the planned supply mechanism can be impacted by individual users burning their own tokens. If it is a problem, restrict `VerseToken::L78(burn)`method access to a whitelist of users controlled by Verse.

Update: The Bitcoin.com team acknowledges the issue and states that they intend for anyone to be able to burn tokens.

Automated Analyses

Slither

Slither has detected 105 results out of which most have been filtered out as false positives. The true positives have been incorporated in the findings above.

Code Documentation

1. There's no documentation anywhere in the codebase. While most of the code is quite straightforward, it is still a good practice to document the code for improved readability and maintainability. We recommend following the NatSpec Format.
2. Add documentation explaining why overflow/underflow protection is not necessary for `VerseToken`'s unchecked blocks.
3. Modify `VerseClaimer`'s minimum and maximum calculations to use OpenZeppelin's `min` and `max` functions to increase readability.

Update: No Code Documentation related code changes were found in the re-audit.

Adherence to Best Practices

1. Most of the implementation of the project can be replaced with code from existing audited third-party libraries. For example, most of `VerseToken`'s functionality can be replaced with OpenZeppelin's ERC20 implementation (`VerseToken` would have to inherit from the `ERC20.sol` contract). Similarly, `MerkleProof.sol` can be replaced with a call to OpenZeppelin's own implementation.
2. It would be more legible to use a require statement instead of revert on `VerseHelper::L27`. That way both the condition and the error message would be part of the same statement.
3. If the `scrapeAmount` on `L200` of the `VerseClaimer` contract is 0 then execution beyond it is a no operation (no-op). Add validation on `L203` to save gas by preventing the no-op from being executed.
4. Modify `VerseClaimer` to use OpenZeppelin's `IERC20` interface to interact with `VerseToken` instead of manually calculating the function signatures and using the low-level `call` keyword. The usage of `call` makes the code less readable and is more error prone. If there is a reason to use `call` add technical documentation explaining the reason.
5. The `ERC20` standard's `approve` functionality is vulnerable to double spending when an existing allowance is being updated. If a transaction claiming the old allowance is included before the transaction updating the allowance then both the old allowance and the new allowance can be claimed. Add end-user documentation explaining the double-spend vulnerability and recommending users use `decreaseAllowance` and `increaseAllowance` when modifying a non-zero `allowance`.
6. Modify `VerseToken::L233(permit)` to accept EIP-2098 compact signatures where the signature's $\mathbf{r}$, $\mathbf{s}$ and $\mathbf{v}$ values are stored compactly within 64 bytes.

Best Practices Found in Re-Audit

1. `VerseClaimer` contract `constructor` only validates that the `_minimumTimeFrame` is larger than `0`. Therefore, it can be set to impractically low values, thereby allowing tokens to be scrapped, for all intents and purposes, immediately (e.g., if we set the value to 1 the tokens would be available to be scrapped a second afterwards). Add validation to `VerseClaimer`'s constructor checking that `_minimumTimeFrame` is greater than a preconfigured minimum time frame value constant.

Update: No Best Practices related code changes were found in the re-audit.

Test Results

Test Suite Results

```
Contract: VerseClaimer
  Token Claimer
     ✓ should be able to store merkle root
     ✓ should be able to store token address
     ✓ should set createTime as blocktime (39ms)
     ✓ should revert if token address is invalid (247ms)
     ✓ should revert if timeframe is invalid (123ms)
  Token Claiming Functionality
     ✓ should be able to enroll (129ms)
     ✓ should be able to enroll bulk (110ms)
     ✓ should not allow to enroll the same recipient twice (80ms)
     ✓ should not allow to enroll zero time entry (40ms)
     ✓ should not allow to enroll invalid values (162ms)
     ✓ should increase totalRequired after each entroll (99ms)
     ✓ should decrease totalRequired after each scrape (201ms)
     ✓ should store correct values (99ms)
     ✓ should be able enroll and scrape tokens generate two events (60ms)
     ✓ should throw error if working with dead token (74ms)
     ✓ should not allow to create allocations if contract is not funded (253ms)
     ✓ should be able to query remaining balance locked (50ms)

Contract: VerseToken
  Token Initial Values
     ✓ should have correct token name
     ✓ should have correct token symbol
     ✓ should have correct token decimals
     ✓ should have correct token supply
     ✓ should return the correct balance for the given account
     ✓ should return the correct allowance for the given spender
  Token Transfer Functionality
     ✓ should transfer correct amount from walletA to walletB
     ✓ should revert if not enough balance in the wallet
     ✓ should reduce wallets balance after transfer (39ms)
     ✓ should emit correct Transfer event
     ✓ should update the balance of the recipient when using transferFrom (68ms)
     ✓ should deduct from the balance of the sender when using transferFrom (73ms)
     ✓ should revert if there is no approval when using transferFrom
     ✓ should revert if the sender has spent more than their approved amount when using transferFrom (53ms)
  Token Approval Functionality
     ✓ should assign value to allowance mapping
     ✓ should emit a correct Approval event (41ms)
     ✓ should allow to increase allowance (73ms)
     ✓ should allow to decrease allowance (79ms)
     ✓ should not change allowance if its at maximum (118ms)
  Burn Functionality
     ✓ should reduce the balance of the wallet thats burnng the tokens (40ms)
     ✓ should not allow to burn more than user have (77ms)
     ✓ should deduct the correct amount from the total supply (43ms)
  Permit Functionality
     ✓ revert if invalid permit() (53ms)
     ✓ should allow to adjust allowance through valid permit() (76ms)
     ✓ revert if invalid deadline (46ms)


  42 passing (9s)
```

Code Coverage

`npm run coverage` was ran to determine test coverage.

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Lines
contracts/	100	100	100	100	
MerkleProof.sol	100	100	100	100	
VerseClaimer.sol	100	100	100	100	
VerseHelper.sol	100	100	100	100	
VerseToken.sol	100	100	100	100	
All files	**100**	**100**	**100**	**100**	

Changelog

- 2022-04-22 - Initial report
- 2022-05-17 - Re-audit
- 2022-05-27 - Test coverage increase

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.