QuillAudits

Audit Report
February, 2022

For

ritestream

Contents

Overview

ritestream

ritestream is an eco-system platform for the creation, monetisation and consumption of film and TV content in Web3. ritestream's vision is to democratise the creator economy and generate revenues in the metaverse and via NFT's for creators and the community.

ritestream has 4 key pillars:
i) a launchpad for film/TV content whereby community can participate in the creation of film/TV project and earn revenues from these NFT's
ii) A consumer app for watching film/tv content
iii) An NFT Marketplace for Film/TV NFT's and
iv) A B2B film/tv distribution platform

Scope of the Audit

The scope of this audit was to analyze the ritestream token smart contract's codebase for quality, security, and correctness.

Date: 17th February, 2022 - 25th February, 2022

ritestream Contract: https://github.com/ritestream/ritestream-contract/blob/master/contracts/Token.sol

Branch: master

Commit: 01377a078b067ad19c4bdcf4da80426de045e5b0

Fixed In: bc2f2d7200adebb23ad52f9ae6f4bb44ece134a6

Checked Vulnerabilities

We have scanned the smart contract for commonly known and more specific vulnerabilities. Here are some of the commonly known vulnerabilities that we considered:

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Malicious libraries
- Compiler version not fixed
- Address hardcoded
- Divide before multiplying
- Integer overflow/underflow
- ERC20 transfer() does not return boolean
- ERC20 approve() race
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis
In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis
Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis
Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption
In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit
Slither, MythX, Truffle, Remix, Ganache, Solidity Metrics

Issue Categories

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

Risk-level	Description
High	A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.
Medium	The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.
Low	Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.
Informational	These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Number of issues per severity

Type	High	Medium	Low	Informational
Open	0	0	0	0
Acknowledged	0	0	0	0
Closed	1	1	3	3

Issues Found – Code Review / Manual Testing

High severity issues

1. Infinite Approval

[#L43] function setAllowanceWithSignature approves infinite tokens from from account to spender account. Spender can exhaust the funds in case the approver account has more than the approved amount.

Recommendation
The amount approved should be the amount that is included in the signature/method params.

```
29      function setAllowanceWithSignature(
30          address from,
31          address spender,
32          uint256 amount,
33          bytes calldata signature
34      ) public onlyOwner {
35          bytes32 messageHash = getMessageHash(from, spender, amount);
36          bytes32 ethSignedMessageHash = getEthSignedMessageHash(messageHash);
37
38          require(
39              recoverSigner(ethSignedMessageHash, signature) == from,
40              "Not authorized"
41          );
42
43          _approve(from, spender, type(uint256).max);
44          userNonces[from]++;
45      }
```

Status: Fixed
Auditor's Comment: The amount approved is the amount that was included in the signature.

Medium severity issues

2. Centralization Risk

[#L21] function burn performs burn operation of the amount on the address provided which can be called by the contract owner. This poses a risk for the token holders where their funds can be burnt by the contract owner at any time.

```
21      function burn(address account, uint256 amount) external onlyOwner {
22          _burn(account, amount);
23      }
```

Recommendation
We advise the client to handle the governance account carefully to avoid any potential hack. We also advise the client to consider the following solutions: with reasonable latency for community awareness on privileged operations; Multisig with community-voted 3rd-party independent co-signers; DAO or Governance module increasing transparency and community involvement;

Status: Fixed

Auditor's Comment: The function now burns the token from the sender's wallet.

Low severity issues

3. Floating Pragma

The contract makes use of the floating-point pragma ^0.8.11. Contracts should be deployed using the same compiler version and flags that were used during the testing process. Locking the pragma helps ensure that contracts are not unintentionally deployed using another pragma, such as an obsolete version that may introduce issues in the contract system.

```
1   // SPDX-License-Identifier: MIT
2   pragma solidity ^0.8.11;
3   pragma abicoder v2;
```

Recommendation
Consider locking the pragma version. It is advised that floating pragma not be used in production. Both truffle-config.js and hardhat.config.js support locking the pragma version.

Status: Fixed
Auditor's Comment: Pragma version is now Locked to 0.8.11

4. Renounce Ownership

Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities on his behalf. The renounceOwnership function is used in smart contracts to renounce ownership. Otherwise, if the contract's ownership has not been transferred previously, it will never have an Owner, which is risky.

Recommendation
It is advised that the Owner cannot call renounceOwnership without first transferring ownership to a different address. Additionally, if a multi-signature wallet is utilized, executing the renounceOwnership method for two or more users should be confirmed. Alternatively, the Renounce Ownership functionality can be disabled by overriding it.

Status: Fixed
Auditor's Comment: enounceOwnership() is now overridden in token contract.

5. Required Zero-Trust Policy

As the crucial aspect of the Ritestream system is the offline agreement and signature of the approver. If the attacker succeeds in convincing a token holder to sign a malformed message hash(which contains the attacker's desired parameters) with the help of phishing or social-engineering attacks, it may allow the attacker to transfer any desired number of tokens from the holder's account, hence it is necessary and required that every lender should follow a zero-trust policy and sign their message hashes by themselves.

Recommendation
Notify and announce the need for a zero-trust policy to the token holders.

Status: Fixed

Informational issues

6. **Public functions that are never called by the contract should be declared external to save gas. Function setAllowanceWithSignature() is declared public.**

   Status: Fixed
   Auditor's Comment: unction setAllowanceWithSignature() is Declared as External

7. **State Variable Default Visibility**

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

```
10        mapping(address => uint256) userNonces;
```

   Recommendation
   Variables can be specified as being public, internal or private. Explicitly define visibility for all state variables. Ref: https://swcregistry.io/docs/SWC-108

   Status: Fixed
   Auditor's Comment: State Variable Visibility is set to Internal

8. **Missing Docstrings**

   It is extremely difficult to locate any contracts or functions, as they lack documentation. One consequence of this is that reviewers' understanding of the code's intention is impeded, which is significant because it is necessary to accurately determine both security and correctness. They are additionally more readable and easier to maintain when wrapped in docstrings. The functions should be documented so that users can understand the purpose or intention of each function, as well as the situations in which it may fail, who is allowed to call it, what values it returns, and what events it emits.

Recommendation

Consider thoroughly documenting all functions (and their parameters) that are part of the contracts' public API. Functions implementing sensitive functionality, even if those are not public, should be clearly documented as well. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

Status: Fixed

Auditor's Comment: Natspecs for important functions are added

Functional Testing Results

Complete functional testing report has been attached below:

```
ERC20 Token
  ✓ Should return the correct decimal count
  ✓ Should burn tokens from deployer
  ✓ Should only allow deployer to mint/burn (67ms)
```

Some of the tests performed are mentioned below:

- ☑ should be able to burn the tokens if the msg.sender is owner
- ☑ should revert if the msg.sender is not owner while burning tokens
- ☑ should approve tokens from holder to spender account through signature
- ☑ should revert on setting user’s allowance with signature if not called by owner
- ☑ should revert on use of other user's signature while approving through signature

Closing Summary

Some issues of High, Medium and Low severity were found during the Audit,All of the Issues are now Fixed by the Auditee.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of ritestream. This audit does not provide a security or correctness guarantee of the audited smart contracts. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the ritesStream team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report
February, 2022

For

ritestream

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com