QuillAudits

Audit Report
January, 2022

For

NOMADLAND

Contents

Contents

Overview

NomadLand is a BEP20 token, which provides features like minting, burning and freezing of tokens.

Scope of the Audit

The scope of this audit was to analyse NomadLand smart contract's codebase for quality, security, and correctness.

NomadLand Contract:
https://bscscan.com/address/0x59Cde41a855682349edaEA221169d9b686687748#code

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 multiply
- 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.timestampMultiple 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 BEP-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
Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis, Theo.

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	8	5
Closed	0	0	0	0

Functional Testing Results

Some of the tests performed are mentioned below

- User should be able to burn its tokens PASS
- Owner should be able to mint new tokens PASS
- Owner should not be able to mint new tokens if finishMinting is called PASS
- releaseOnce should revert if there exist no frozen tokens for the user or freeze time has not elapsed PASS
- releaseAll should revert if there exist no frozen tokens for the user or freeze time has not elapsed FAIL
- Should not transfer if the contract is paused PASS
- Owner should be able to transfer ownership to a new owner PASS
- Should be able to release frozen tokens once the freeze time has elapsed. PASS

Issues Found

High severity issues

No issues were found.

Medium severity issues

No issues were found.

Low severity issues

- **Not comply with BEP20 standard completely**

  BEP20 standard makes it mandatory for the tokens to define a function as getOwner, which should return the owner of the token contract.


5.1.1.6 getOwner

```
function getOwner() external view returns (address);
```

- Returns the bep20 token owner which is necessary for binding with bep2 token.
- NOTE - This is an extended method of EIP20. Tokens which don't implement this method will never flow across the Binance Chain and Binance Smart Chain.

  However, the token contract doesn't implement/define any such function, as a result the token may not flow across the Binance Chain and Binance Smart Chain, as stated by BEP20 interface documentation.

  Recommendation
  Consider adding the getOwner function.

  Status: Acknowledged

- **Old Solidity Compiler**

  Contract uses a very old solidity compiler as **0.4.23**, which contains some well known bugs.

  Recommendation
  Use the latest compiler version in order to avoid bugs introduced in older versions.

  Status: Acknowledged

- **Missing two step process to change privileged role Owner**

  When privileged roles are being changed, it is recommended to follow a two-step approach:
  1) The current privileged role proposes a new address for the change
  2) The newly proposed address then claims the privileged role in a separate transaction. This two-step change allows accidental proposals to be corrected instead of leaving the system operationally with no/ malicious privileged role. (Ref: Security Pitfalls: 162)

  However, token contract uses Ownable contract to define a privileged role owner, which uses a one step process to transfer/renounce ownership, as a consequence the owner may accidently lose control over the operational logic of the contract.

  Recommendation
  Consider switching to a two-step process for transferring ownership and an optional time-margin for renouncing ownership

  Status: Acknowledged

- **Unsafe Arithmetic Calculation**

  Although, contract uses SafeMath library which defines functions to perform safe arithmetic calculations, still some instances of unsafe arithmetic calculations have been reported which are prone to integer overflows. However the likelihood of such an event to occur is low.

  Some of the instances found during the audit are:
  [#L408] super.balanceOf(_owner) + freezingBalance[_owner];

  [#L511] tokens += balance;

  Recommendation
  Consider using SafeMath to avoid risks of unsafe arithmetic calculations like Integer Overflows

  Status: Acknowledged

- **Lack of Proper Documentation**

  Documentation describes what (and how) the implementation of different components of the system does to achieve the specification goals. Without documentation, a system implementation cannot be evaluated against the specification for correctness and one will have to rely on analyzing the implementation itself. (Ref. Security Pitfalls: 137)

  However, Operational logic for many functions was not well documented.

  Status: Acknowledged

- **Overuse of Public Visibility**

  Contract Consts defines some constants with values to be used by MainToken for initialization. However, the visibility used for these constants is public.

  Solidity implicitly creates a public getter function for each public variable, as a result, overuse of public visibility increases the bytecode and makes it expensive to deploy.

  Recommendation
  Consider switching from public to internal visibility unless absolutely necessary.

  Status: Acknowledged

- **Unexpected Functional Behaviour/Outcome**

  A user can use functions releaseOnce and releaseAll to release their frozen tokens once the freeze time has elapsed.

  releaseOnce allows a user to release first available frozen tokens, and reverts if there are no frozen tokens available, however releaseAll still executes even if there are no frozen tokens available for a user or the freeze time has not elapsed, hence may produce unexpected results or behaviour for users.

Recommendation
Consider reviewing and verifying the operational and business logic.

Status: Acknowledged

- **block.timestamp dependance**

  Contract uses block.timestamp to calculate freeze time for freezing tokens and releasing tokens, which may be manipulated by miners in order to bypass checks implemented by the contract.

  Recommendation
  Avoid using block.timestamp for calculating time values.

  Status: Acknowledged

Informational issues

- **Public** functions never used by the contract internally should be declared **external** to save gas

  Status: Acknowledged

- **Unnecessary modifier**

  Contract MintableToken defines a modifier hasMintPermission as

```
352     modifier hasMintPermission() {
353       require(msg.sender == owner);
354       _;
355     }
356
```

  However, MintableToken is derived from base contract Ownable, which defines a modifier onlyOwner with the same operational logic as

```
301     modifier onlyOwner() {
302       require(msg.sender == owner);
303       _;
304     }
```

  Hence, modifier hasMintPermission is unnecessary implemented and modifier onlyOwner can be used in place of it without harming any operational logic and improving code readability.

Recommendation
Consider removing the hasMintPermission modifier and using onlyOwner instead.

Status: Acknowledged

- **Unnecessary checks**

Contract MainToken restricts token transfers to happen only in the not paused state by implementing checks as:

```
703     function transferFrom(address _from, address _to, uint256 _value) public returns (bool _success) {
704       require(!paused);
705       return super.transferFrom(_from, _to, _value);
706     }
707
708     function transfer(address _to, uint256 _value) public returns (bool _success) {
709       require(!paused);
710       return super.transfer(_to, _value);
711     }
```

However, MainToken inherits from the base contract Pausable which already defines modifier for the same operational logic as:

```
604     modifier whenNotPaused() {
605       require(!paused);
606       _;
607     }
```

Hence, the checks can be replaced with the modifier whenNotPaused in order to improve the code readability.

Recommendation
Consider removing the checks and using whenNotPaused modifier instead.

Status: Acknowledged

- **Centralization Risk**

Contract mints the complete token liquidity and transfers the ownership to a hardcoded address defined with constant TARGET_USER, which imposes certain centralization risks as below:

1. In case of an incorrect address, the complete liquidity will be minted to an unintended address.

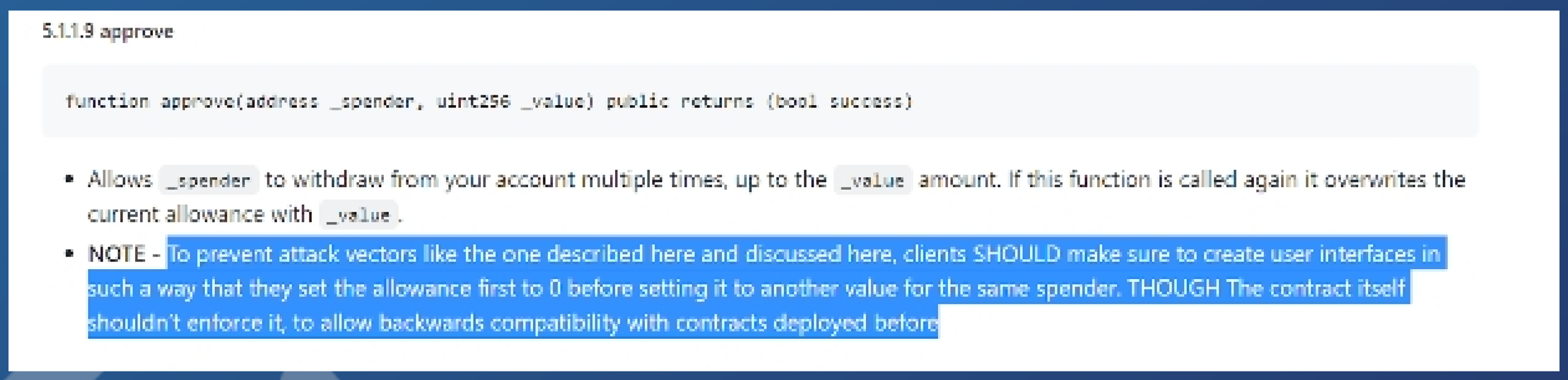
2. If the private key is lost, the deployer will immediately lose control over the privileged role ownerOnly functions.

Status: Acknowledged

- **BEP20 approve race condition**

The standard ERC20 implementation contains a widely-known racing condition in its approve function, wherein a spender is able to witness the token owner broadcast a transaction altering their approval and quickly sign and broadcast a transaction using transferFrom to move the current approved amount from the owner's balance to the spender. If the spender's transaction is validated before the owner's, the spender is able to spend their entire approval amount twice.

As the BEP20 standard is proposed by deriving the ERC20 protocol of Ethereum, the race condition exists here as well.


5.1.1.9 approve

```
function approve(address _spender, uint256 _value) public returns (bool success)
```

- Allows `_spender` to withdraw from your account multiple times, up to the `_value` amount. If this function is called again it overwrites the current allowance with `_value`.
- NOTE - To prevent attack vectors like the one described here and discussed here, clients SHOULD make sure to create user interfaces in such a way that they set the allowance first to 0 before setting it to another value for the same spender. THOUGH The contract itself shouldn't enforce it, to allow backwards compatibility with contracts deployed before

Reference

1. https://eips.ethereum.org/EIPS/eip-20
2. https://github.com/binance-chain/BEPs/blob/master/BEP20.md#5119-approve

Status: Acknowledged

Closing Summary

Several issues of low severity were found during the audit, which the Auditee has acknowledged. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

Quillhash audit is not a security warranty, investment advice, or an endorsement of the NomadLand contract. 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 Nomadland team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

Audit Report
January, 2022

For

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com