Smart Contract Testing Handbook

Sascha Kubisch

April 5, 2023

1 Introduction

This handbook provides step-by-step instructions and essential concepts for testing the Carbon Credits Marketplace smart contracts using Hardhat. It covers unit and integration testing, relevant commands, and best practices.

2 Prerequisites

Before starting the testing process, ensure that the following tools and libraries are installed:

- Node.js (v14.x or later)
- npm (v6.x or later)
- Hardhat
- Solidity compiler (solc)

3 Unit Testing

Unit testing is the process of testing individual components or functions of a smart contract in isolation. It helps identify potential bugs and ensures that each function behaves as expected.

3.1 Writing Unit Tests

Hardhat uses the Mocha testing framework and Chai assertion library to write and execute unit tests. In your project, create a `test` folder and add test scripts for each smart contract.

A typical test script includes:

- Importing the necessary libraries and contracts
- Setting up the testing environment, such as deploying the contracts and creating test accounts

- Defining test cases using Mocha's `describe` and `it` functions
- Writing assertions using Chai's `expect` function

3.2 Running Unit Tests

To run the unit tests, navigate to the project directory and execute the following command:

```
npx hardhat test
```

Hardhat will automatically run all test scripts located in the `test` folder and display the results.

4 Integration Testing

Integration testing focuses on the interactions between smart contracts and ensures that they work together correctly. It helps identify issues with contract interfaces, event emissions, and overall system behavior.

4.1 Writing Integration Tests

Integration tests are similar to unit tests but focus on the interactions between multiple smart contracts. Create a separate test script for integration tests, such as `test/integrationTests.js`, and follow these steps:

- Import the necessary libraries and contracts
- Set up the testing environment, including deploying and configuring all relevant contracts
- Define test cases that involve interactions between multiple contracts
- Write assertions to verify contract state changes and event emissions

4.2 Running Integration Tests

To run the integration tests, navigate to the project directory and execute the following command:

```
npx hardhat test test/integrationTests.js
```

Hardhat will run the specified test script and display the results.

5 Testing Best Practices

When testing smart contracts, follow these best practices to ensure robust and reliable code:

- Test all public and external functions, as well as any critical internal functions
- Verify that functions revert when expected, such as when preconditions are not met or input is invalid
- Test edge cases and boundary conditions, such as maximum and minimum values or empty inputs
- Ensure that events are emitted correctly and include the expected data
- Verify that contract state changes are consistent with the intended behavior

6 Designing Unit Tests Based on Smart Contract Auditing Best Practices

Unit testing is a critical aspect of smart contract development, ensuring that individual components work as expected. Following best practices for designing unit tests based on smart contract auditing can help minimize vulnerabilities and improve code quality. This section provides a detailed guide to design unit tests based on auditing best practices.

6.1 Understand the Importance of Unit Testing

Thoroughly testing smart contracts is essential to ensure their security and functionality. Smart contracts often handle digital assets and are immutable once deployed; thus, it is crucial to identify and fix vulnerabilities before deployment.

6.2 Test for Common Vulnerabilities

Design unit tests that specifically target known vulnerabilities in smart contracts, such as reentrancy, integer overflows, and front-running attacks. By doing so, you can identify potential security issues early and make necessary adjustments to the code.

6.3 Test All Functions and Components

Each function and component of the smart contract should have corresponding unit tests. It is crucial to test both the expected behavior (positive test cases) and unexpected behavior (negative test cases) for each function.

6.4 Isolate Each Test

Each unit test should focus on a single function or component and should not rely on the results of other tests. This approach helps ensure that each test is independent, and any failures can be traced back to a specific function.

6.5 Use a Test-Driven Development Approach

Test-driven development (TDD) involves writing unit tests before writing the actual code. This approach encourages better code design, reduces the likelihood of introducing errors, and helps ensure that tests cover all aspects of the code.

6.6 Test Edge Cases

Edge cases are scenarios that may not be immediately apparent but can still cause problems or vulnerabilities in the code. When designing unit tests, consider edge cases, such as extremely large or small inputs, empty strings, or unusual transactions.

6.7 Ensure Code Coverage

Aim for a high level of code coverage when designing unit tests. Code coverage is the percentage of code that is executed during testing. A higher code coverage means a lower likelihood of undetected vulnerabilities.

6.8 Check Gas Usage

Design unit tests that check for excessive gas usage in your smart contract functions. Excessive gas usage can lead to higher transaction costs and even failed transactions if gas limits are exceeded.

6.9 Perform Fuzz Testing

Fuzz testing involves generating random inputs to test the smart contract functions, aiming to identify unexpected behavior or vulnerabilities. Incorporate fuzz testing into your testing suite to improve the robustness of your smart contracts.

6.10 Continuously Improve and Update Tests

As the smart contract code evolves, the associated unit tests should also be updated to reflect changes. Regularly review and update your tests to ensure that they remain relevant and effective in identifying vulnerabilities.

By following these best practices for designing unit tests based on smart contract auditing, you can improve the security and reliability of your smart contracts. Adequate testing can help identify vulnerabilities before deployment, minimizing the risk of loss or theft of digital assets.

7 Running Tests on a Local Testnet

This section provides a step-by-step guide to running the Carbon Credits Marketplace smart contract tests on a local testnet using Hardhat.

7.1 Prerequisites

Before starting the testing process, ensure that the following tools and libraries are installed:

- Node.js (v14.x or later)
- npm (v6.x or later)
- Hardhat

7.2 Setup Hardhat Project

1. Open the terminal and navigate to your project folder.
2. Execute the following command to initialize a new Hardhat project:

   sql Copy code

   ```
   npx hardhat
   ```

3. Follow the prompts and select "Create an empty hardhat.config.js" when asked.

7.3 Configure Hardhat

1. Open the `hardhat.config.js` file in your project folder and configure it with the following settings:

   css Copy code

   ```
   require("@nomiclabs/hardhat-waffle");

   module.exports = {
     solidity: "0.8.4",
     networks: {
       localhost: {
         url: "http://127.0.0.1:8545"
       }
     }
   };
   ```

   Adjust the Solidity version to match your smart contracts.

7.4 Install Dependencies

1. Install the necessary dependencies for testing using the following command:

   ruby Copy code

   ```
   npm install --save-dev @nomiclabs/hardhat-waffle ethers
   ```

7.5 Create Test Files

1. In your project folder, create a new folder named `test` and place your test script files inside.

7.6 Start Local Testnet

1. Open a new terminal window and navigate to your project folder.
2. Execute the following command to start a local Ethereum testnet:

   arduino Copy code

   ```
   npx hardhat node
   ```

3. The local testnet will start, and a list of available test accounts will be displayed. Take note of the private keys and addresses for later use.

7.7 Run Tests

1. In the original terminal window, navigate to your project folder.
2. Execute the following command to run the test scripts:

   arduino Copy code

   ```
   npx hardhat test
   ```

3. Hardhat will execute the tests and display the results in the terminal.

8 Deploying Smart Contracts to TRON Testnet

This section provides step-by-step instructions for deploying the Carbon Credits Marketplace smart contracts to the TRON testnet.

8.1 Prerequisites

Before starting the deployment process, ensure that the following tools and libraries are installed:

- Node.js (v14.x or later)
- npm (v6.x or later)
- TronLink Chrome extension
- TronBox

8.2 Setup TronLink Wallet

1. Install the TronLink Chrome extension from the Chrome Web Store.
2. Create a new wallet or import an existing one using your private key or mnemonic phrase.
3. Request test TRX from the TRON Faucet to fund your wallet with testnet tokens.
4. Ensure that the TronLink wallet is connected to the Shasta testnet.

8.3 Configure TronBox

1. In your project folder, create a new file named `tronbox.js`.
2. Open the `tronbox.js` file and configure the TronBox settings, specifying the Shasta testnet and your TronLink wallet address:

   vbnet Copy code

   ```
   module.exports = {
     networks: {
       shasta: {
         privateKey: 'YOUR_PRIVATE_KEY',
         userFeePercentage: 30,
         feeLimit: 1e8,
         fullHost: 'https://api.shasta.trongrid.io',
         network_id: '*'
       }
     }
   };
   ```

   Replace `YOUR_PRIVATE_KEY` with the private key of your TronLink wallet.

8.4 Compile Smart Contracts

1. In your project folder, create a new folder named `contracts` and place your Solidity smart contract files inside.
2. Open the terminal and navigate to your project folder.
3. Execute the following command to compile the smart contracts:

   arduino Copy code

   ```
   tronbox compile
   ```

8.5 Deploy Smart Contracts

1. In the terminal, run the following command to deploy the smart contracts to the Shasta testnet:

   arduino Copy code

   ```
   tronbox migrate --network shasta
   ```

2. TronBox will deploy the smart contracts and display the contract addresses. Save these addresses for future reference.

8.6 Interact with Smart Contracts

1. Use the deployed contract addresses and TronWeb library to interact with the smart contracts from your application or TronLink wallet.
2. Ensure that you are connected to the Shasta testnet while interacting with the deployed contracts.

9 Conclusion

This comprehensive handbook has provided you with the essential knowledge and tools to develop, deploy, and test smart contracts on the TRON network. By following the step-by-step instructions, folder structures, and test scripts presented in this guide, you can create and deploy secure and efficient smart contracts. Moreover, the handbook emphasizes the importance of unit testing and integration testing, as well as adherence to best practices for smart contract auditing, to ensure the reliability and security of your contracts.

As you progress in your smart contract development journey, remember to stay up-to-date with the latest developments in the field, and continuously refine your skills and knowledge. Blockchain technology and smart contract platforms continue to evolve rapidly, and staying informed about these changes will help you maintain a competitive edge.

We hope that this handbook has provided you with valuable insights and a strong foundation for your smart contract development journey. Armed with

this knowledge, you are now well-equipped to create, deploy, and test smart contracts that have the potential to revolutionize various industries and create new opportunities in the decentralized ecosystem. Good luck, and happy coding!