CryptoBears
Due: March 3, 2019

Contents

1 Introduction 1

2 Setup 2

2.1 Installation 2

2.2 Frameworks and Packages 2

2.2.1 Truffle 2

2.2.2 npm 2

3 Assignment 2

3.1 Tokens 2

3.1.1 ERC20 2

3.1.2 ERC721 2

3.1.3 Support Code 2

3.2 Smart Contracts 3

3.2.1 BearBucks.sol 3

3.2.2 CryptoBears.sol 3

3.2.3 Commit-Reveal Flip 3

3.3 Crowdsale 4

3.4 Reentrancy 4

3.5 Vulnerabilities 4

3.6 Getting Started 5

3.7 Testing 5

4 Grading 6

5 Handing In 6

1 Introduction

Welcome to CryptoBears! You will be implementing two contracts to represent your beloved Brown University on the Ethereum Network.

One contract is a CryptoBear, contained in the CryptoBears.sol file. Like any mammalian beasts of the blockchain, CryptoBears need to eat to survive. Their primary method of eating is to get BearBucks, an ERC20 token contained in the BearBucks.sol file, which you will implement. BearBucks are won by betting, and the CryptoBears need to know how to correctly place bets with one another on the blockchain.

This assignment will be using a few Solidity paradigms that are essential to smart contract programming. Luckily, we’ve provided this Solidity Guide that has all the Solidity you’ll need to know for this project. Make sure to give it a read before you begin implementing. You can find the guide here. 
2 Setup

2.1 Installation

Install the support code by running `cs1951L_install CryptoBears` in your shell.

You will also need to run `npm install` followed by `export PATH=$(npm bin):$PATH`

Test that everything was installed correctly by running `truffle version`

If the install didn’t work properly, the message ”command truffle not found” will output instead of the current version of Truffle.

2.2 Frameworks and Packages

2.2.1 Truffle

Truffle is a smart contract development framework that helps with the deployment and testing. It deploys the smart contracts through two .js file located in the migrations directory. The file truffle.js is used to configure the truffle framework. In particular, your smart contracts will be deployed on a private Ethereum blockchain on your computer set up by the tool ganache. When the smart contract are compiled by Truffle, a new directory /build is made to store the EVM bytecode.

2.2.2 npm

npm is a package manager for JavaScript used by Truffle. The files `package.json` and `package-lock.json` are used by npm to handle the project’s dependencies - Truffle and ganache.

3 Assignment

You will be working with several files inside the `contracts` directory

3.1 Tokens

3.1.1 ERC20

Your BearBucks will be represented as ERC20 tokens. This is the standard for fungible tokens, meaning each token has the same value, similar to currency. You will need to implement the token functions in `ERC20.sol`

3.1.2 ERC721

CryptoBears will be implemented as ERC721 tokens, which are non-fungible. That is, each token has a unique value associated to it. Some non-fungible assets include houses, artworks, bonds, and seat-specific concert tickets, since each asset of the same type has unique value. You will need to implement the functions in `ERC721.sol`

3.1.3 Support Code

ERC20 and ERC721 industry standard code is open source and available for anyone to use and implement. We recommend attempting to implement these tokens yourself to familiarize yourself with how they work, but in practice it is good to use the accepted standard instead of trying to
implement the tokens yourself. For this reason, we are providing you with the link to the open source solution and you are allowed to copy these files verbatim. The repository can be found [here](link).

### 3.2 Smart Contracts

#### 3.2.1 BearBucks.sol

In BearBucks.sol, you will be implementing the BearBucks contract. The BearBucks contract keeps track of the total amount each address has in active bets at a given time. This contract allows users to place and remove bets.

There is a vulnerability in the smart contract that we will test for in our test suite. To fix this vulnerability, you will need to override the inherited function `approve(...)`. To override functions, use the same function signature and add any additional code you may consider necessary, with the function using `super.approve(...)` as a return call. You may find fixing the vulnerability easier after finishing the rest of the project.

#### 3.2.2 CryptoBears.sol

In CryptoBears.sol, you will be implementing the CryptoBears contract. A CryptoBear needs to eat BearBucks to survive but also needs to bet BearBucks to get more BearBucks. An important distinction to make is that as a CryptoBear is a smart contract it does not actually have a BearBucks balance. That balance is instead held by the Ethereum address both contracts are owned by. If a CryptoBear’s ownership is transferred to another address, no BearBucks are transferred with it.

For CryptoBears to bet against each other, they will use the Commit-Reveal Flip commitment scheme.

#### 3.2.3 Commit-Reveal Flip

After a long and bitter Providence winter, you and another starving CryptoBear want to place bets for BearBucks. You decide to flip a coin and bet on the outcome. To simulate a coin, you will both map your IDs to a bet and XOR them, letting the flip value be the parity of the XORed bets.

By storing your bets in a mapping, who ever bets second can see what the first Bear bet and can now place a bet to ensure a victory. Instead, to ensure neither party knows what the other will bet, the Commit-Reveal Flip scheme is used.

The CryptoBear contract uses six data structures for betting:

- `_bets` - a mapping that, for each bear, stores all possible opponents and the amount of Bear-Bucks the bear is currently betting against that opponent
- `_commitments` - a mapping of all the hashes of the owners’ bets
- `_committed` - a mapping of whether or not each owner has placed a bet
- `_secrets` - a mapping of the actual bets
- `_revealed` - a mapping of if each bet has been revealed
- `_bears` - an array of Bear structs. Each instance of a CryptoBear is represented by a Bear struct that holds its unique properties. The unique ID of each bear is the index of the Bear struct in the `_bears` array
The commit-reveal flip scheme uses hashes of the bets instead of the actual bets to verify that both parties committed to a bet. Since a hash is unique for each input, the actual bet can easily be verified after both parties have committed. Now that we know everyone bet without knowing anyone else’s bets, we can reveal the bets and securely flip.

3.3 Crowdsale

A crowdsale lets users purchase tokens using Wei, the smallest subunit of ether. You will be implementing functions to purchase both CryptoBear tokens as well as BearBucks tokens.

After each transaction, the Wei spent should be transferred to the wallet’s address while the unused Wei should be sent back to the buyer.

3.4 Reentrancy

There is one vulnerability we are requiring you to exploit. Since the crowdsale contract returns unused wei to msg.sender, if we send calls to the crowdsale contract from another smart contract, we can code that smart contract to call the crowdsale contract again from within the fallback function.

This is called reentrancy and is a famous type of smart contract hack. The DAO hack stole $250 million in Ether using reentrancy and caused a hard fork of the ethereum blockchain. The javascript test for this vulnerability is already included at the end of VulnerabilityTests.js, but you still need to code the attacking smart contract ReentrancyExploit.sol. This contract should have one payable function called attack() that takes in no arguments. This is called from the hacker’s address inside the javascript test. The rest of the smart contract is up to you. A successful attack will cause the javascript test to fail. In order to perform the attack, you will very likely need to adjust your crowdsale contract to make it vulnerable.

Once you have the attack working (which means the test is failing), you should comment out the vulnerable code and replace it with fixed code that prevents the attack from working. The test should now pass. Your comments in BearCrowdsale.sol should clearly explain which lines should be commented/uncommented in order to run the exploit, as the TA’s will be testing this. Your comments\README should also explain why the exploit works on the vulnerable code and not on the fixed code. Note that just making the reentrancy test fail is not enough, you should also explain why it fails (i.e. “balance is twice what it should be”, “contract balance drained” etc.)

3.5 Vulnerabilities

Checking for security vulnerabilities is by far the hardest and most important part of smart contract development. We are giving you the chance to practice identifying and exploiting smart contract vulnerabilities by offering extra credit to students who find exploitable bugs in their project. In order to count, vulnerabilities must not already be checked for in the provided testing suite. If your project passes all tests and still has a vulnerability, it will count. If your project does not pass all tests, then you should give a compelling explanation in your README of how the vulnerability is not checked for in the test suite.

Any vulnerability you find should be explained in your README. Each vulnerability you find will receive up to 4% extra credit. To get the full 4%, you must add a new test to the file VulnerabilityTests.js that demonstrates the vulnerability. The stencil provides three examples of how to write smart contract tests in javascript using the truffle framework, the last of which shows how one might write a test to prove a vulnerability exists (This test exploits a potential vulnerability that is
also checked for by other tests in the testing suite).

Since we haven’t taught you any javascript, the TA’s will be very generous in explaining how to write tests in hours and on piazza. Note that most tests in the test suite use a lot of custom code stored in Util.js that you do not need to use, so they may not be the best examples. You do not need to fix any vulnerabilities you find, but should attempt to explain how you might fix them in your README. If you are unable to write a test to exploit your vulnerability, a good written explanation may earn partial credit.

We have deliberately left several vulnerabilities in the smart contracts for you to find, but will be very pleased if you find more. If you’re not sure whether you’ve found a vulnerability, it’s probably a good idea to ask the TA’s on piazza or in hours (make sure to make piazza posts private).

### 3.6 Getting Started

We recommended that you begin working on the contracts in the following order: ERC20.sol, ERC721.sol, BearBucks.sol, CryptoBears.sol, BearCrowdsale.sol, ReentrancyExploit.sol.

### 3.7 Testing

Before running the test suite, you must first spin up a private blockchain on your computer with ganache. To do this, open a new terminal tab and run the command `ganache-cli`. Wait till you see ‘Listening on 127.0.0.1:8545’. This tells you the blockchain is ready to go. Ganache sets up a blockchain exactly like Ethereum’s except your computer is the only node in the network. It also preloads a list of 10 addresses with balances of 100 eth each and exposes their private keys so that truffle can use them to send transactions.

**Tip:** If you ever get any errors about their not being enough wei/gas/funds for a transaction. Try closing the terminal running the ganache-cli and running it again from a new terminal.

Now that you have ganache up and running, you can run the test suite with by opening up a new terminal and running: `truffle test`. Remember that you must be running on the same machine as ganache in order for the test suite to work. If you run into issues with ganache or truffle not being found try running `export PATH=$(npm bin):$PATH` again.

When correctly implemented, the test suite takes up to 5 minutes to run. So, we recommend running only the test file(s) relevant to the smart contract you are currently working on at a given time. The relevant files should be clear from their name (i.e. BearBucksNegativeTests.js).

To run a specific test file, run `truffle test/NameOfTestFile.js`

We recommend aiming to pass PositiveTests for the contract first, then moving on to NegativeTests for that contract. Try to pass all the tests for a contract before moving on to the next contract, as several contracts inherit from one another.
4 Grading

<table>
<thead>
<tr>
<th>Category</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality (Passing Tests)</td>
<td>70%</td>
</tr>
<tr>
<td>Comments / README / Style</td>
<td>10%</td>
</tr>
<tr>
<td>Reentrancy Functionality</td>
<td>15%</td>
</tr>
<tr>
<td>Reentrancy Explanation</td>
<td>5%</td>
</tr>
<tr>
<td>Vulnerabilities</td>
<td>4% Extra Credit Each</td>
</tr>
</tbody>
</table>

5 Handing In

To hand in CryptoBears, run:

cs1951l_handin CryptoBears