Blockchain CTF
CSCI 5240 Final Project

Sam Patti, Ethan Pyke, Luke Morrissey

Overview

For our final project, we made a Blockchain inspired CTF. The CTF starts at https://blockchain-ctf.netlify.app. By solving each task you will receive a flag. After solving each level, you will be able to provide your answers as arguments and write to a smart contract where you will receive a token for your efforts.

The various levels teach the user about EVM byte code, Zero Knowledge Proofs, Proof of Work Consensus and Reentrancy Attacks!

The project can be fully experienced by just going through the link above and completing the challenges. The rest of this write up will detail solutions as well as how we expected people to go about finding the solutions.

We hope you enjoy the CTF!

Solutions are Spoiled Below!

LEVEL 1: WEB

Given the following EVM byte code:

00	34	CALLVALUE
01	80	DUP1
02	02	MUL
03	610100	PUSH2 0100
06	14	EQ
07	600C	PUSH1 0C
09	57	JUMPI
0A	FD	REVERT
0B	FD	REVERT
0C	5B	JUMPDEST
0D	00	STOP
0E	FD	REVERT
0F	FD	REVERT

User has to give a value for CALLVALUE to push onto the stack (we'll call this x) the stack now looks like this: [x 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]. That value is then duplicated and pushed on the stack so it looks like this: [x x 0 0 0 0 0 0 0 0 0 0 0 0 0 0]. Then, the top two elements of the stack are multiplied, consuming both elements, and the product is pushed to the stack. It then looks like this: [x*x 0 0 0 0 0 0 0 0 0 0 0 0 0 0]. Then, the hex value 0100 is pushed onto the stack so it now looks like this: [0x0100 x*x 0 0 0 0 0 0 0 0 0 0 0 0 0 0]. 0x0100 is the value 256 in decimal so the stack can be thought of as [256 x*x 0 0 0 0 0 0 0 0 0 0 0 0 0 0]. Next, EQ checks if the top two values are equal. If they are, it pushes a 1, if not it pushes a 0 and consumes the two other elements. Lastly, if the top value is 1, JUMPI will execute, otherwise it will fail and revert. So we need 256 and x*x to be equal which means x must be 16. We can click submit answer, enter 16 and click check and we see that it is correct and we can then download the remaining CTF files.

LEVEL 2: HARDER EVM BYTE CODE

We move on and see that we have another EVM byte code problem to solve. We are given the following code:

```
00      36      CALLDATASIZE
01      6003    PUSH1 08
03      10      LT
04      6009    PUSH1 09
06      57      JUMPI
07      FD      REVERT
08      FD      REVERT
09      5B      JUMPDEST
0A      34      CALLVALUE
0B      36      CALLDATASIZE
0C      02      MUL
0D      6008    PUSH1 36
0F      14      EQ
10      6014    PUSH1 14
12      57      JUMPI
13      FD      REVERT
14      5B      JUMPDEST
15      00      STOP
```

Here, we need to enter both a CALLDATASIZE and a CALLVALUE. CALLDATASIZE takes a hex value (0x……) and then converts it to bytes so for example the values 0x00000001 and 0xffffffff both convert to the value 4 since they are 4 bytes long. We see that the stack gets populated with our CALLDATASIZE and then the hex value 0x08 (8 in decimal) and then the LT instruction. This checks if the first stack value (0x08) is less than the second stack value (CALLDATASIZE). If true, it will push 1 to the stack, else 0. In order for this to be true, CALLDATASIZE needs to be longer than 8 bytes. After that, we push 0x09 to the stack as the jump destination and then jump there IF a 1 was pushed from the LT instruction. Assuming it was, we then are asked to enter a CALLVALUE which is pushed to the stack and then our CALLDATASIZE is pushed. We then multiply those two values, push 0x36, and check that the two are equal. 0x36 is decimal 54. If they are equal, we push a 1 and then a 0x14 for the jump destination. IF the values are equal and a 1 is pushed, then we successfully exit. So, we know that our CALLDATASIZE must be greater than 8 bytes and be multiplied by a CALLVALUE to equal 0x36 or decimal 54. If we enter a CALLDATASIZE that is 9 bytes (0x000000000000000001) and a CALLVALUE of 6, then we will pass this challenge. Run the execute binary (./execute) and enter those values.

LEVEL 3: PROOF OF WORK

In this level there are 5 blocks filled with transactions. The goal is to change one of the transactions on the first block but keep them as valid blocks. This will require you to find a nonce value of the first block that keeps it so there are still 4 leading zeros in the block hash, hence proof of work. This will change the block hash of the first block so you will thus have to recompute the nonce of all of the following blocks as well as they reference the hash of the previous block.

The first step is to verify that your environment is giving you the same hash values already calculated. Running the starter code will give you the hash of the unchanged first block(block0). This should be the same as the value of the prev_block in the second block(block1). I am using python 3.10.9 so if it doesn't work try matching that version.

After the environment is verified you can go ahead and write code to change the first transaction on the new block and then calculate the new nonce values to make these valid blocks. We are looking for the smallest nonce values so start at 0 and increase from there. After you find the nonce of the new first block, change the prev_hash of the second block to this new hash and then recompute the nonce for this block. Do this until you have found the nonce values of each block that have 4 leading zeros for the hash of the block.

The new nonce values are:

Block0: 72477
Block1: 42284
Block2: 2651
Block3: 74709
Block4: 69344

And the sum of these nonce values which is the flag for this part is then: 261465

LEVEL 4: ZERO KNOWLEDGE PROOF

We are given a lot of files here and most of them don't matter. The first step is to realize what is going on in the circuit. In foo(), there are two inputs and one output. This makes it trivial to fill in the "???". We know that in1 will be multiplied by in2 to get out. The only tricky part may be making sure they are in the right order. The final line should be:

```
out <== in1 * in2;
```

Once that is established, figuring out the rest of the circuit becomes just plugging in the values. After doing so, we see that the circuit is simply multiplying the three input values from the input.json file. Multiplying the values gives us 5240 (our class number). The last part is filling in the four blanks in the public.json file. Here, we already have our four important values (the product 5240, and the three inputs) and all we need to do is put them in the right order. The final public.json file should look like this:

```
[
    "5240",
    "10",
    "131",
    "4"
]
```

Once these two files are complete, run the verify binary (./verify) and get the flag.

LEVEL 5: REENTRANCY ATTACK

In this level you are given access to two solidity smart contracts and a two-page write up to learn how to interact with them on Remix IDE. Remix allows you to run solidity contracts on a sample blockchain that resets every time you open a workspace.

The first part of the challenge is just a tutorial on how to get set up and start working with the smart contract on Remix. TheStore.sol contract is fully written and given to the user. It is a very basic smart contract with deposit, withdraw, and getBalance functions.

The second part of the challenge notifies the user that there is a vulnerability in the way that TheStore.sol contract is written, and to write an attack function that drains the balance of TheStore.sol. This vulnerability is in the withdraw function which: checks the balance of the user, sends the funds to the user, then updates the balance of the user. The order in which these events happen are vulnerable to a reentrancy attack. This occurs because when TheStore.sol contract sends funds to the user, it hits a fallback function on the user's contract which re enters back into TheStore.sol contract. Most of the time this function is empty so it does nothing, but an attacker can actually write code in the fallback function that calls withdraw again. At this point, the funds would have been sent to the attacker's contract, but their balance would not have been updated yet. Therefore, the attacker will have made a loop which keeps calling the withdraw function in their contract's fallback function until TheStore.sol balance is drained. A code stub for the Attack.sol contract is pre-written, the user just has to write code for the fallback and attack functions. The correctly written code for these functions would look something like this:

```
receive() external payable {
    if (address(theStore).balance >= 1 ether) {
        theStore.withdraw();
    }
}

function attack() external payable {
    require(msg.value >= 1 ether);
    theStore.deposit{value: 1 ether}();
    theStore.withdraw();
}
```

NOTE: These types of vulnerabilities can be easily avoided if the withdraw function order instead: checks the balance of the user, updates the balance of the user, then sends the funds to the user.

At the end of the attack a "checkFlag" function has been pre written that checks the Attack contract's balance. If it is >= 10 ether (how much the setup made the user put into TheStore contract), then a string converted from a Byte32 snippet is revealed as the flag. (I know someone could just convert this online, but hopefully it would be easier to just write the attack function than do that).

LEVEL 6: MINTING A TOKEN

The reward token is a token we made and deployed to the Goerli testnet called Buff Coin. You can view the token on ether scan here: https://goerli.etherscan.io/address/0xa71b62cf39714eae8126bf0054a319870155dfeb. This token was written following this tutorial. The contract code has been verified so you can inspect the solidity code for details of implementation.

Once you have solved all the CTF levels you can claim your token by interacting with our smart contract directly on etherscan: https://goerli.etherscan.io/address/0xb5b587B005284c59662ED616B7Bf23de9dc23B29. Simply go to the contract -> write contract tab. You can connect your wallet and enter the answers to extract your token. This contract is also verified on etherscan so you can inspect it for details.

After you have claimed your token, you can import the Buff Coin to your wallet by referencing the contract that created it above.

The solutions to all the levels are as follows (without the quotes), so you can claim a token:

web_sol: “16”
evm_sol: “{3vm_15_c001}”
zero_knowledge_sol: “{w31c0m3_t0_Z3R0_KN013D63}”
proof_of_work_sol: “261465”
reentrancy_sol: “th15 i$ r33ntr4ncy”