Assignment #4

CPEN 442

November 2, 2015

SwagMasterFlex
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada

Github Repository *(with solutions)*:
https://github.com/EmilMaric/CPEN_442/tree/master/Assignment_4

I. Question #1

Student #:
`33657115`

Hash:
`LTF6A2CBE86F0354722C9B377E9E187885D1DE6B5D`

Bits of entropy
$\sim$`13 bits` = $ln(10^4)/ln(2)$

Password found:
`1531`

CPU/GPU time spent:
Less than a second

I started off by examining the number of characters in the hashed password. The count came out to 42 characters (168 bits), which didn't match any of the common hashing algorithms. Next, I observed that the first two characters of my hash were non-hex, while the other 40 characters were hex characters.

This led me believe that the first two characters of the hash were stored as the salt, and the following 40 characters were the result of hashing the password using **SHA1**, since **SHA1** produces a 20-character hash (40-character hex-representation).

To crack the password, I ran **oclHashcat** [2], a password cracker that runs on the GPU. It took less than a second for **oclHashcat** to crack the password.

II. Question #2

Student #:
`33657115`

Hash:
`CS0DF2939D499DCC46A693653DB281FDBB016B502B`

Bits of entropy
$\sim$`37 bits` = $ln(76^6)/ln(2)$

Password found:
`y%jtqf`

CPU/GPU time spent:
3 mins, 30 secs
(70,748,995,584 combinations tried)

Since the format of the hash is the same as `Question #1`, I made the same assumption as before. Namely, that the hash function used to hash the password was **SHA1**.

Again, I ran **oclHashcat** in order to crack the password. It took the program about three and a half minutes to find the password.

III. Question #3

Found Password:
`#l!Je*iDiHBbB5O#_Bm`

Bits of Entropy:
$\sim$37 bits = $ln(76^{(}19))/ln(2)$

In order to begin, I disassembled the binary using **IDAPro** [1]. This allowed me to step through the assembly code in order to find the password.

The first thing I noticed was that the program was denying access if the length of the string was not nineteen characters (0x13) long. Next, I inputted a nineteen-character long password, which allowed me to progress further through the code.

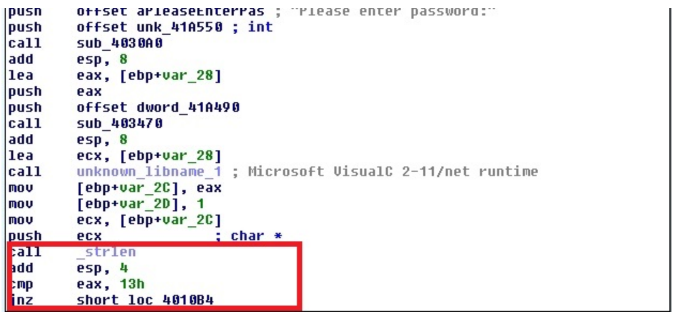

Fig. 1. Result of `strlen()` being compared with `0x13`

At this point, I noticed that the program was fetching one character at a time from my inputted password string and offset `0x419051`, and comparing these two values. I assumed that this was the password that the program was looking for. I extracted 19 characters from this offset location, and inputted this result on the next program iteration. This password finally granted me access.

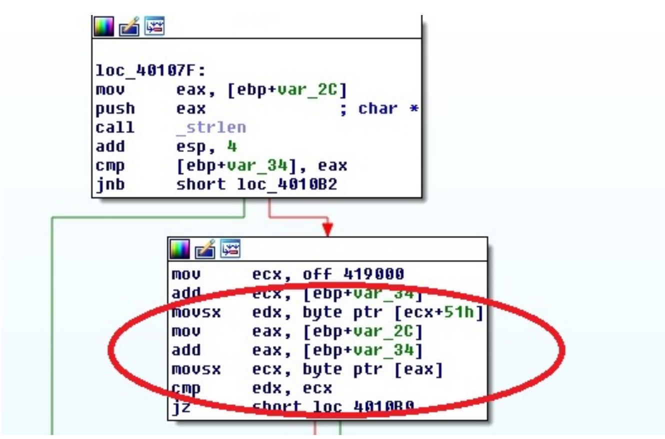

Fig. 2. My input string being compared with the stored password

In order to patch the program, I changed the value of `%ebp + var_2D` to always be `1`, indicating that the correct password was entered. When the value of `%ebp + var_2D` was tested (or compared to `0`), it would always pass through, and grant me access.

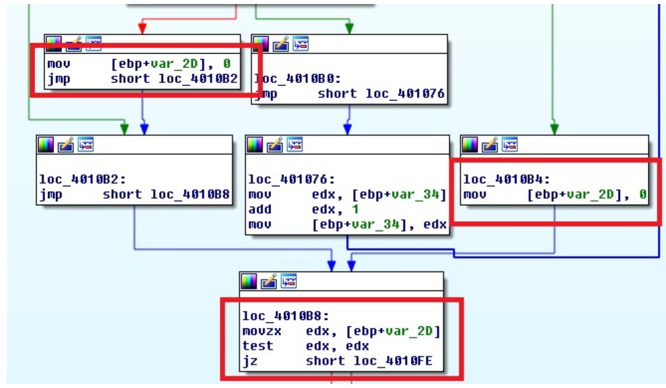

Fig. 3. Assigning `%ebp + var_2D` to `1` prevents the program from jumping to the "`Access Denied`" block

IV. QUESTION #4

Student #:
`33657115`

Hash:
`929a2e98651FD2DDE3629F8DB8F0BDC32B673BE4`

Found Password:
`M7rUAV`

CPU/GPU time spent:
2 mins, 53 secs

The process to find the required password for this problem was very similar to `Question #3`. Once again I used **IDAPro**. The presence of a **SHA1** function led me to believe that password was stored as a hash. The only parameter to the **SHA1** function was the input password itself, which led me to assume that no salt was used to hash the password.

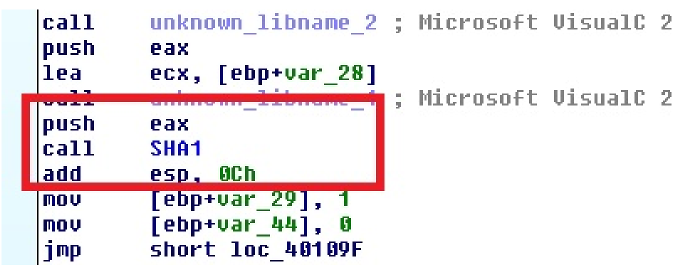

Fig. 4. `SHA1` function shown with parameters (notice no salt passed)

I noticed that my hashed input string was being compared with a string located at `0x4141F3`. I extracted 20 characters from this location and took it to be the password hash. Next, I entered the hash into **oclHashcat** and it generated the corresponding plaintext password.

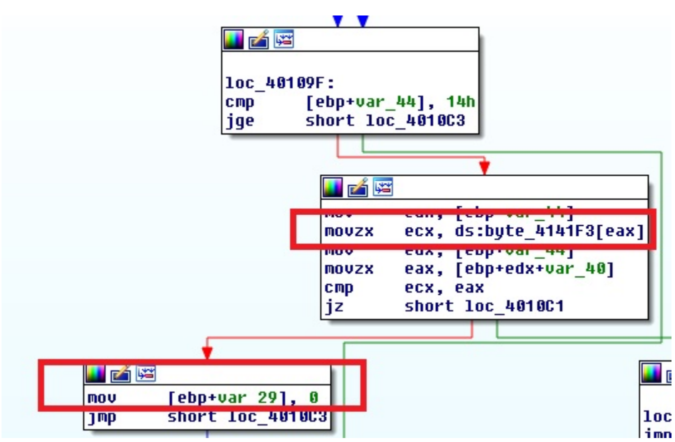

Fig. 5. My hashed input being compared with the hashed password. Also shows the instruction that I patched

To patch the program, I changed the value of `%ebp + var_29` to always be `1`. **Fig. 5** highlights where I changed the value of this variable from `0` to `1`. This allowed the program to always grant access, regardless of whether the password was correct or not.

In order to allow the user to change the password of the program, I created a Python program that asked the user for the new password, hashed it, and then overwrote the old hashed password, which was stored in static memory in the binary, with the new hashed password. The next time the user ran the program, they would need to enter the new password in order to gain access.

REFERENCES

[1] *IDAPro*. 2015. URL: https://www.hex-rays.com/products/ida/ (visited on 10/29/2015).

[2] Jens 'atom'Steube. *oclHashcat*. 2015. URL: http://hashcat.net/oclhashcat/ (visited on 10/29/2015).