

# CPEN 442

## Assignment 4

### Passwords

Agent47 (56406144)

#### 1 QUESTION 1

The first step to solve that question was realize that the size of the hash didn't match any of the known hash sizes. The closest size possible was **SHA1** but I had 2 extra characters.

**It6D209CE47FEC653F2947F0E3D09580130C05C3CC**

The two first characters aren't hexadecimal so they might be a salt. I ran all the possible pin combination using the salt perpended and after that I was able to find the solution.

The correct pin is: **2160**

The time necessary for finding the solution was:

real	0m0.020s
user	0m0.016s
sys	0m0.004s

#### 2 QUESTION 2

Just like in the previous question the hash was generated using **SHA1** and the first two symbols are the salt. For this question first I had to generate all the possible combinations with the alphabet provided by th question

abcdefghijklmnopqrstuvmwxyz  
 ABCDEFGHIJKLMNOPQRSTUVWXYZ  
 1234567890  
 !@#\$%^&\*()\_+==

For this I used the python library *itertools*. This library has a method called *product* that will generate all the possible combinations using an specific set of characters.

Because the number of possible combinations is:

$76^6 = 192699928576$  possible passwords

I though that searching for the password going forward and backward at the same time the chances that I would find the solution would be higher. For that I ran 2 different codes where the generated possible passwords were generated starting from 'aaaaaa' to the first terminal and starting from '======' for the second terminal. After around 6 hours processing my computer crashed and I lost the processes that were running. So I had to patch the software in a way that it would not test the previous generated passwords. Even though the program still had to spin lock in the for loop until the counter reaches 5 billion passwords because that was approximately the number of passwords tested before the crash.

I setup the terminals to run the code again and after a few hours the right password was founded.

The correct password for this question is: **NTdLWn2\_**

The approximated time necessary for finding the solution was:

real	539m16.173s
user	539m15.059s
sys	0m0.139s

#### 3 QUESTION 3

To solve this assignment I used the software **IDA pro** in order to analyze the executable file.

##### 3.1 a)

At first I was analyzing the code in the text mode. Using the search tool I could find the part of the code that prints the instructions to insert the password. I started analyzing the data from there. Analyzing the assembly code I could recognize that there is an loop that I suppose that is the loop that check for the integrity of the password comparing symbol by symbol the typed password and the hard coded password. I also noticed that there is a hard coded string in the assembly code. The variable can be found at the address **.rdata:004141A0**

the content of the hardcoded string is:

**t#eKyJ\_Zue!VjV4x-5dU)lnbSCISl##vb%Mq#JBrWMytûsgFe-UQHBIWd&\_aAOfp**

At first I thought that this was the correct password but using this as a password didn't work. I came back to the assembly code and first I saw that before the program starts checking for the integrity of the password it tests the length of the string that I inserted. This test is done by the following code:

```
call _strlen
add esp, 4
cmp eax, 0Ah
```

paying close attention you can see that the last command checks if the string has 0Ah length, that converting to decimal is 10 characters.

At this point I tried to insert the first 10 characters of the string, but it also didn't work.

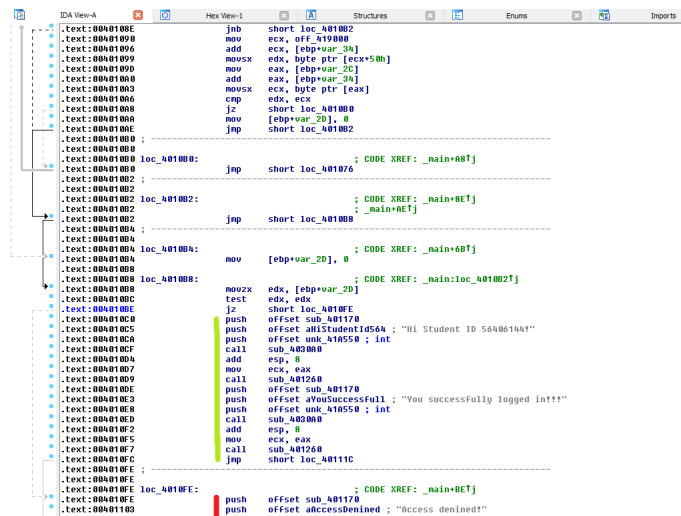
After a more detailed look at the code I also noticed that the inside the loop that checks for the integrity of the password the variable that receives the hard coded string has an offset. This can be seen at the following instructions:

```
mov ecx, off_419000
add ecx, [ebp+var_34]
movsx edx, byte ptr [ecx+50h]
```

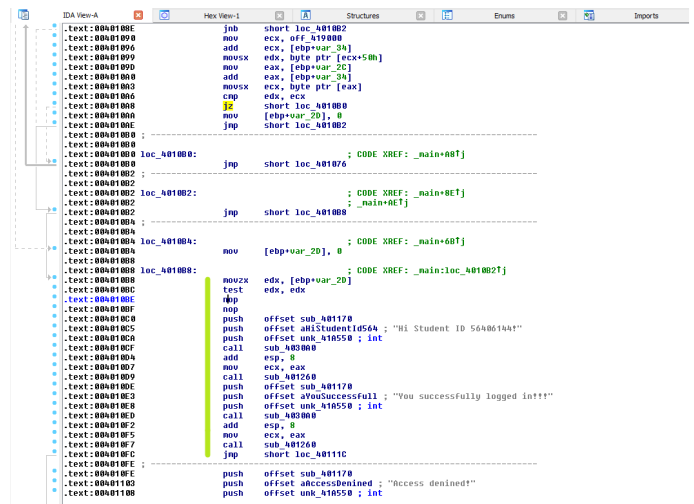
At the last line ecx (the variable that receives the hard coded string) is added with an offset of 50h witch is 80 if converted to base 10. considering this offset I could find that the correct password is: **jCng\$4RME**

### 3.2 b)

Looking at the text mode we can see that the ADA pro software has some arrows that shows the instructions flow. Analyzing the execution flow I could see that in the line **.text:004010BE** the program flow is redirected to the area where the program outputs that the password is incorrect (marked in RED). this can be seen in the following image:



To do not jump to the area of the code where the program inform that the password is wrong, I changed that line where it checks the program flow for two **nop** instructions. Doing that the program will jump to the section where the correct password would lead. It can be seen in the image below:



After the changes being made I exported the .dif file using the ADA software. To apply the changes I executed the C code available at [http://idabook.com/examples/chapter\\_14/ida\\_patcher.c](http://idabook.com/examples/chapter_14/ida_patcher.c). After compiled this C code will receive as input the .exe file and the .dif file and then merge the files. The .exe file will be modified so I had to make sure that I made a copy of the executable file.

## 4 QUESTION 4

### 4.1 a)

Just like in the question 3 the password is hard coded in the program. However in this example the password isn't stored as plain text, what is stored is the hash of the password using SHA1. The hash is stored in the address **.rdata:00414212** byte **414212**. In this case the has is stored in a sequence of hex values. Because the SHA1 only have 40 characters output only the first 20 hex values. For cracking the password in this questions I used the Hashcat software. Because I have access to a graphic card I used cudaHashCat.

The software can be downloaded for free at <http://hashcat.net/oclhashcat/>

To use that software I used the following steps:

Download the program

Unzip the file

Open CMD

Go to the hashcat directory file and run the following command:

```
.cudaHashcat64.exe -m 100 -a 3 -1 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZVWXYZ1234567890!@#%$&*()_+==' hash.txt
```

Here is the output of the hashcat program after cracking that hash

```
21f0e9b168b2d5870906f1b86693754c0ddae05c:7NeW0i
Status.....: Cracked
Input.Mode.....: Mask (?1?1?1?1?1) [6]
Hash.Target....: 21f0e9b168b2d5870906f1b86693754c0ddae05c
Hash.Type.....: SHA1
Time.Started...: Mon Nov 02 20:33:38 2015 (7 mins, 28
secs)
Speed.GPU.#1...: 198.8 MH/s
```

Recovered.....: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts  
 Progress.....: 89472892928/192699928576 (46.43%)  
 Rejected.....: 0/89472892928 (0.00%)  
 Restore.Point.: 15466496/33362176 (46.36%)  
 HWMon.GPU.#1...: 0% Util, 66c Temp, N/A Fan  
 Started: Mon Nov 02 20:33:38 2015  
 Stopped: Mon Nov 02 20:41:11 2015  
 The password for this file is: 7NeW0i

#### 4.2 b)

For this part a similar solution was used just like in the Question 3

The image below shows the code without the patch

```

.text:004010A3 jge short loc_4010C3
.text:004010A5 mov eax, [ebp+var_44]
.text:004010A8 movzx ecx, ds:byte_41A212[eax]
.text:004010AF mov edx, [ebp+var_44]
.text:004010B2 movzx eax, [ebp+edx+var_40]
.text:004010B7 cmp ecx, eax
.text:004010B9 jz short loc_4010C1
.text:004010BB mov [ebp+var_29], 0
.text:004010BF jmp short loc_4010C3
;-----
.text:004010C1 loc_4010C1: jmp short loc_401096 ; CODE XREF: _main+097j
;-----
.text:004010C3 loc_4010C3: ; CODE XREF: _main+037j
; _main+0F7j
.text:004010C3 movzx ecx, [ebp+var_29]
.text:004010C7 test ecx, ecx
.text:004010C9 jz short loc_401109
.text:004010CB push offset sub_401180
.text:004010CD push offset aHiStudentId544 ; "Hi Student ID 54466144"
.text:004010D0 push offset unk_41A550 ; int
.text:004010D4 call sub_4030D0
.text:004010D8 add esp, 8
.text:004010DE mov ecx, eax
.text:004010E0 call sub_401290
.text:004010E4 push offset sub_401180
.text:004010E8 push offset aYouSuccessfully ; "You successfully logged in!!"
.text:004010EC push offset unk_41A550 ; int
.text:004010F0 call sub_4030D0
.text:004010F4 add esp, 8
.text:004010FA mov ecx, eax
.text:004010FC call sub_401290
.text:00401100 jmp short loc_401127
;-----
.text:00401109 loc_401109: ; CODE XREF: _main+097j
.text:00401109 push offset sub_401180
.text:0040110D push offset aAccessDenied ; "Access denied!"
.text:00401111 push offset unk_41A550 ; int
.text:00401115 call sub_4030D0
.text:00401119 add esp, 8
.text:0040111F mov ecx, eax

```

Now using the patch to accept any password

```

.text:004010A3 jge short loc_4010C3
.text:004010A5 mov eax, [ebp+var_44]
.text:004010A8 movzx ecx, ds:byte_41A212[eax]
.text:004010AF mov edx, [ebp+var_44]
.text:004010B2 movzx eax, [ebp+edx+var_40]
.text:004010B7 cmp ecx, eax
.text:004010B9 jz short loc_4010C1
.text:004010BB mov [ebp+var_29], 0
.text:004010BF jmp short loc_4010C3
;-----
.text:004010C1 loc_4010C1: jmp short loc_401096 ; CODE XREF: _main+097j
;-----
.text:004010C3 loc_4010C3: ; CODE XREF: _main+037j
; _main+0F7j
.text:004010C3 movzx ecx, [ebp+var_29]
.text:004010C7 test ecx, ecx
.text:004010C9 jz short loc_401109
.text:004010CB push offset sub_401180
.text:004010CD push offset aHiStudentId544 ; "Hi Student ID 54466144"
.text:004010D0 push offset unk_41A550 ; int
.text:004010D4 call sub_4030D0
.text:004010D8 add esp, 8
.text:004010DE mov ecx, eax
.text:004010E0 call sub_401290
.text:004010E4 push offset sub_401180
.text:004010E8 push offset aYouSuccessfully ; "You successfully logged in!!"
.text:004010EC push offset unk_41A550 ; int
.text:004010F0 call sub_4030D0
.text:004010F4 add esp, 8
.text:004010FA mov ecx, eax
.text:004010FC call sub_401290
.text:00401100 jmp short loc_401127
;-----
.text:00401109 loc_401109: ; CODE XREF: _main+097j
.text:00401109 push offset sub_401180
.text:0040110D push offset aAccessDenied ; "Access denied!"
.text:00401111 push offset unk_41A550 ; int
.text:00401115 call sub_4030D0
.text:00401119 add esp, 8
.text:0040111F mov ecx, eax

```

After the changes being made I exported the .dif file using the ADA software. To apply the changes I executed the C code available at [http://idabook.com/examples/chapter\\_14/ida\\_patcher.c](http://idabook.com/examples/chapter_14/ida_patcher.c). After compiled this C code will receive as input the .exe file and the .dif file and then merge the files. The .exe file will be modified so I had to make sure that I made a copy of the executable file.

#### 4.3 c)

To solve that question I had to compute manually the hash of the desired password (for that a python script was used) and then insert that hash into the executable file changing the original hard coded hash value. In that way when the program computes the hash it will match with that was modified.

The lines where the code might be modified are represented in the image below:

```

.rdata:0041A212 byte_41A212 00 0F 0B ; DATA XREF: _main+0B7r
.rdata:0041A213 00 0E 9A ; T
.rdata:0041A214 00 0B 1B ; I
.rdata:0041A215 00 68 0A ; h
.rdata:0041A216 00 0B 21 ; t
.rdata:0041A217 00 00 5A ; +
.rdata:0041A218 00 67 0A ; h
.rdata:0041A219 00 00 00 ; 0
.rdata:0041A21A 00 00 00 ; 0
.rdata:0041A21B 00 00 00 ; 0
.rdata:0041A21C 00 00 00 ; 0
.rdata:0041A21D 00 00 00 ; 0
.rdata:0041A21E 00 00 00 ; 0
.rdata:0041A21F 00 00 00 ; 0
.rdata:0041A220 00 00 00 ; 0
.rdata:0041A221 00 00 00 ; 0
.rdata:0041A222 00 00 00 ; 0
.rdata:0041A223 00 00 00 ; 0
.rdata:0041A224 00 00 00 ; 0
.rdata:0041A225 00 00 00 ; 0
.rdata:0041A226 00 00 00 ; 0
.rdata:0041A227 00 00 00 ; 0
.rdata:0041A228 00 00 00 ; 0
.rdata:0041A229 00 00 00 ; 0
.rdata:0041A22A 00 00 00 ; 0
.rdata:0041A22B 00 00 00 ; 0
.rdata:0041A22C 00 00 00 ; 0
.rdata:0041A22D 00 00 00 ; 0
.rdata:0041A22E 00 00 00 ; 0
.rdata:0041A22F 00 00 00 ; 0
.rdata:0041A230 00 00 00 ; 0
.rdata:0041A231 00 00 00 ; 0
.rdata:0041A232 00 00 00 ; 0
.rdata:0041A233 00 00 00 ; 0
.rdata:0041A234 00 00 00 ; 0
.rdata:0041A235 00 00 00 ; 0
.rdata:0041A236 00 00 00 ; 0
.rdata:0041A237 00 00 00 ; 0
.rdata:0041A238 00 00 00 ; 0
.rdata:0041A239 00 00 00 ; 0
.rdata:0041A23A 00 00 00 ; 0
.rdata:0041A23B 00 00 00 ; 0
.rdata:0041A23C 00 00 00 ; 0
.rdata:0041A23D 00 00 00 ; 0
.rdata:0041A23E 00 00 00 ; 0
.rdata:0041A23F 00 00 00 ; 0

```

After modifying the hash code the same methodology of the Question 3 b) and Question 4 b) will be used. First the .dif file was exported using the ADA software. To apply the changes the C code available at [http://idabook.com/examples/chapter\\_14/ida\\_patcher.c](http://idabook.com/examples/chapter_14/ida_patcher.c) was executed. After compiled, this C code will receive as input the .exe file and the .dif file and then merge the files. The .exe file will be modified so I had to make sure that I made a copy of the executable file.

After this process the file will accept only the other the password that generated the hash computed earlier.

## 5 APPENDIX

All the codes for this project can be found at <https://github.com/faellacurcio/AssignmentPassword.git>

## REFERENCES

- [1] 14.1. hashlib Secure hashes and message digests [Online]. Available: <https://docs.python.org/2/library/hashlib.html>
- [2] string - Generate all possible passwords based on character mapping possible in 3 lines of python? [Online]. Available: <http://stackoverflow.com/questions/10082094/generate-all-possible-passwords-based-on-character-mapping-possible-in-3-line>
- [3] IDA: about [Online]. Available: <https://www.hex-rays.com/products/ida/>
- [4] hashcat - advanced password recovery [Online]. Available: <http://hashcat.net/hashcat/>
- [5] coder 32 edition —X86 Opcode and instruction reference [Online]. Available: <http://ref.x86asm.net/coder32.html>
- [6] [Online]. Available: [http://idabook.com/examples/chapter\\_14/ida\\_patcher.c](http://idabook.com/examples/chapter_14/ida_patcher.c)
- [7] dll used to run program 4 download. Available: <https://dl.dropboxusercontent.com/u/14053604/libeay32.dll>
- [8] [Online]. Available: