Assignment #4

CPEN 442

Kaibo Ma (32400129)
Department of Electrical and Computer Engineering
University of British Columbia
Vancouver, Canada

I. Problem #1

mypwd:co5916
where co is the salt.

10^4 = 10,000
2^13 = 8192
2^14 = 1684

It is about 13~14 bits of entropy. I used john the ripper for this assignment. I was able to retrieve the first password in mere seconds running with 32 different forks. I found that 'co' is the salt.

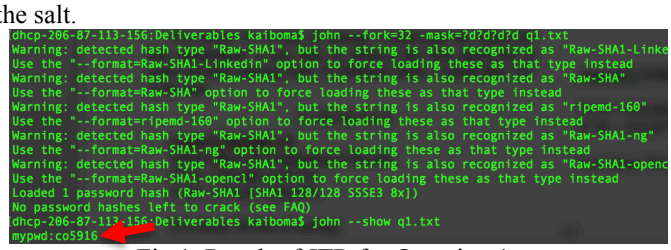

Fig 1. Result of JTR for Question 1.

II. Problem #2

mypwd:Dkqk&UvR
where Dk in the beginning is the salt
79^6 = 243,087,455,521
2^38 = 274,877,906,944

It is about 38 bits of entropy. Used the same program as question with 32 different forks and ran in mask mode. This specified the length and keyspace to narrow down the search. It took about 2 hours and 30 mins to complete on my Macbook. This would have been faster running off a GPU or a faster CPU.

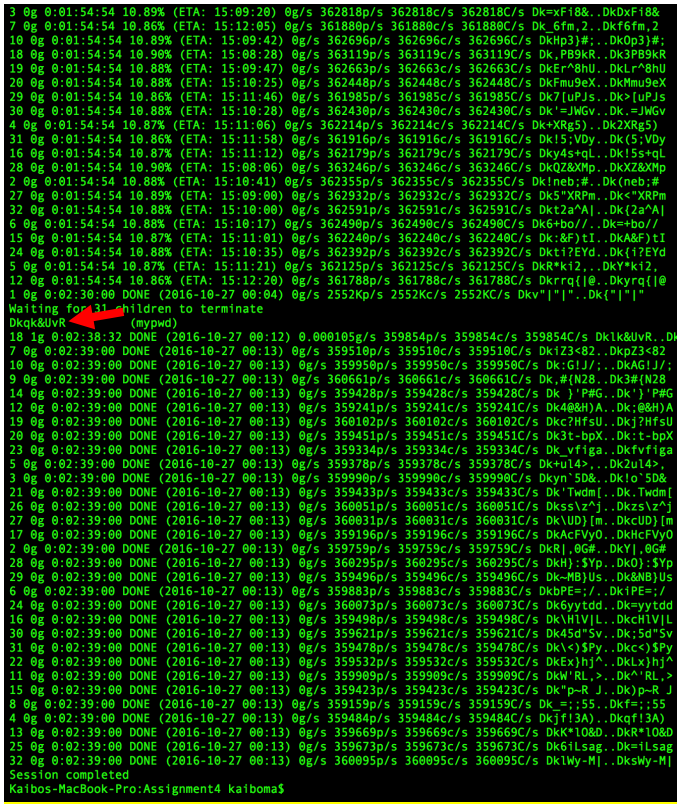

Fig 2. Result of JTR for Question 2.

III. Problem #3

mypwd:5Wo-u3ozhz8q1M^d

79^16 = 2.3016191^30
2^101 = 2.5353012^3

a) It is about 101 bits of entropy. I obtained the password by looking at the main function assembly code. I realized that it checks if your input string is the same length as a hard coded length of 10h which is 16.

```
push    ecx
call    sub_40CD40
add     esp, 4
cmp     eax, 10h
jnz     short loc_405424
```

Fig 3. The string length comparison check

Then the code will check with a hard coded randomized string inside .rdata. It will compare a certain offset and match your input string with that string.

```
mov     ecx, off_42D000
add     ecx, [ebp+var_14]
movsx   edx, byte ptr [ecx+20h]
mov     eax, [ebp+var_18]
add     eax, [ebp+var_14]
movsx   ecx, byte ptr [eax]
cmp     edx, ecx
jz      short loc_405420
```

Fig 4. The offset for comparison string

```
db 't0FZNjaquZjPS(1GeZZW!c6VHW%TqrB5Wo-u3ozhz8q1M^dz791H-)G8XdgHz1+G'
                ; DATA XREF: .data:off_42D000↓o
db '59)b_$2$yRyd0fjQVO!Mh(626pUZh^6X07)2)bD*huiEUD+S(NO0$Iz!crt5rc6G0'
db 'ZacjFUUk*',0
```

Fig 5. The whole comparison string where my password resides.

I used a script I created to help me find my password inside the string (attached findpw.py).

b) I noticed in the assembly that there is a "test" opcode followed by a "jz". I knew that it would either pass or fail. Hence I made the "jz" opcode to "nop" which would skip the jump and proceed with logging in successfully.

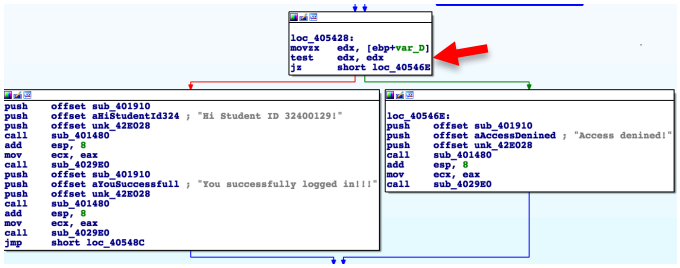

Fig 6. The checking condition for pass or fail

This is the .dif file I created with IDA:

This difference file has been created by IDA
32400129.program1.exe
0000482E: 74 90

IV. Problem #4

mypwd:U3H$ki
79^6 = 243,087,455,521
2^38 = 274,877,906,944
It is about 38 bits of entropy.

This is the .dif file I created with IDA:
This difference file has been created by IDA
32400129.program2.exe
00004859: 74 90

a) Unlike Question 3 this programs password is encrypted with SHA-1. I noticed from the assembly that it will check byte by byte in a lookup table imported by libeay32.dll. The password starts at the address referenced by 'byte_41F234' at address 41F234. Then in a counter loop of 20 times (this gave away that it was SHA1as it takes 20 bytes). In the following diagram, this is my SHA1 hashed password in the .rdata.

```
.rdata:0041F234 byte_41F234  db 0DCh          ; DATA XREF: _main+A8↑r
.rdata:0041F235              db  2Ch ; ,
.rdata:0041F236              db    1
.rdata:0041F237              db 0D9h ; +
.rdata:0041F238              db  64h ; d
.rdata:0041F239              db  30h ; 0
.rdata:0041F23A              db  11h
.rdata:0041F23B              db  4Ch ; L
.rdata:0041F23C              db 0F5h ; +
.rdata:0041F23D              db 0A2h ; +
.rdata:0041F23E              db 0ECh ; +
.rdata:0041F23F              db 0DBh ; +
.rdata:0041F240              db  98h ; +
.rdata:0041F241              db  38h ; 8
.rdata:0041F242              db  58h ; X
.rdata:0041F243              db 0F5h ; +
.rdata:0041F244              db  8Ch ; +
```

Fig 7. The hashed password value.

Once obtaining this value and copying it over to a .txt file I ran john the ripper on 32 forks. As a result, I obtained my password: U3H$ki which tested perfectly into the program provided. The brute force search took about5 hours and 25 minutes to complete.

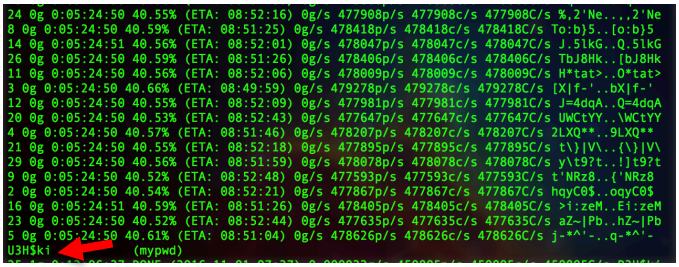

Fig 8. The resulting brute force search on the password.

b) Similar to question 3, the same 'test' and 'jz' functions are called. This allowed me to just take 'jz' written in 74h and change to 90h which is 'nop'.

c) To write a patch file to make our .exe file accept and password we desire. I created a python script that seeks out the correct address offset of the beginning of where the SHA1 password is stored. Then it will take the arguments you run with the python script and hash that value and overwrite the address location that contains the previous values. Image below shows how I run my script with argument 'qwerty' that patches the .exe file and use the new 'qwerty' password as input to .exe. The .exe accepts the new password:

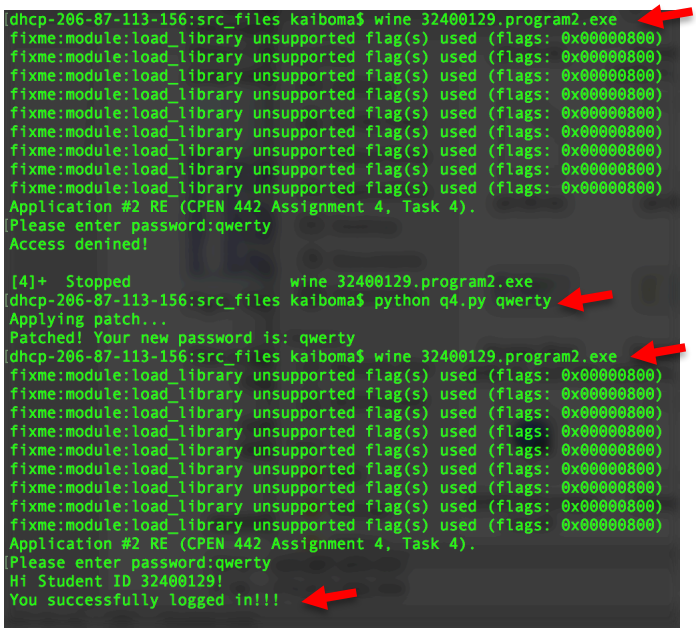

Fig 9. Successful usage of q4.py script that patches the .exe

```python
import hashlib
import array
from sys import argv,exit

def main():
    if len(argv) < 2:
        print "Please Enter Desired Password:"
        print "Usage: %s <password>" % (argv[0])
        exit(0)

    else:
        print "Applying patch..."
    try:
        f = open('32400129.program2.exe', 'r+b')
        sha1 = hashlib.sha1(argv[1]).hexdigest()
        f.seek(0x1E034)
        f.write(sha1.decode("hex"))
        f.close()
        print "Patched! Your new password is:", (argv[1])
    except Exception, e:
        print "Error..."
        exit(1)

if __name__ == "__main__":
    main()
```

Fig 10. My python script that patches the program to any password you want.

I have also attached q4.py along with the email.