CTF Report

Troy Fulton
CSCE 451 200

June 3, 2019

Contents

1 Cheesy

For this challenge, I started by running the program in my Linux Virtual Machine to view the output. I recognized the base 64 encryption algorithm when I ran the executable reversing1.exe because I have seen before that when a string ends with "==," it tends to be base 64. I used https://codebeautify.org/base64-decode to verify that when I entered the strings printed from the program, I got:

FLAGflagFLAGflagFLAGflag
Can you recognize base64??
FLAGflagFLAGflagFLAGflag
You just missed the flag

Then, I opened the executable in IDA Pro using my Virtual Machine and went straight to the strings subview to see if I could find another string that ended in "==". Surely enough, the string "Z2lnZW17M2E1eV9SM3YzcjUxTjYhfQ==" is "gigem{3a5y_R3v3r51N6!}" in base 64, which was the flag.

2 Snakes over Cheese

From searching for the .pyc file type, I found that Reversing2.pyc is a compiled python file, meaning that they contain byte code for the python virtual machine to interpret (https://stackoverflow.com/a/2998228). This means that to compile it, someone had to use the py_compile library, which also means it can be "uncompiled" back to python source code. I installed the uncompyle library and decompiled the byte codes back to the following Python2.7 source code in my Virtual Machine:

```
1 # uncompyle6 version 3.2.5
2 # Python bytecode 2.7 (62211)
3 # Decompiled from: Python 2.7.15rc1 (default, Nov 12 2018, 14:31:15)
4 # [GCC 7.3.0]
5 # Embedded file name: reversing2.py
6 # Compiled at: 2018-10-07 15:28:58
7 from datetime import datetime
8 Fqaa = [102, 108, 97, 103, 123, 100, 101, 99, 111, 109, 112, 105, 108, 101, 125]
9 XidT = [83, 117, 112, 101, 114, 83, 101, 99, 114, 101, 116, 75, 101, 121]
10
11 def main():
12     print 'Clock.exe'
13     # Enter SuperSecretKey
14     input = raw_input('>: ').strip()
15     kUIl = ''
16     for i in XidT:
17         kUIl += chr(i)
18
19     if input == kUIl:
20         alYe = ''
21         for i in Fqaa:
22             alYe += chr(i)
23
24         print alYe
25     else:
26         print datetime.now()
27
28
29 if __name__ == '__main__':
30     main()
31 # okay decompiling reversing2.pyc
```

The comment on line 13 was entered after I solved the problem, but before that, I figured out that I needed to get this XidT string to match my input because I could see on lines 16 and 17 that these integers in that array were nothing more than ASCII. Using https://www.rapidtables.com/convert/number/ascii-hex-bin-dec-converter.html, I converted the characters to the ASCII String "SuperSecretKey" and entered that when running the program. Then, the python script spit out the flag: "flag{decompile}".

3 042

The first thing I recognized about this assembly file was that it was compiled for MacOS. As soon as I tried to compile it with GCC on my Linux VM, I was drowned in a sea of assembly errors. I started by meticulously going through each error one at a time and trying to make the assembly look more like it was assembled by GCC so that GCC could assemble it and spit out the answer. I knew that if I could run it, I would get the answer because line 143 has a formatted string "gigem{%s}", which I knew would give me the flag.

Thus, I first got rid of the MacOS headers and footers on lines 1, 2, 35 and 146. Then, I noticed I was still getting errors because of function calls such as "___stack_chk_fail" and "___stack_chk_guard" since they depended on the last line that I removed, which linked the symbols to the executable. Once I got them to compile, they became a problem again because when I tried to run the code after compiling, stack smashing was reported. For some reason, the compiler doesn't recognize the relative PC from the Global Offset Table seen from the command on line 62 movq ___stack_chk_guard@GOTPCREL(%rip), %r9

I fixed this by simply making the compiler call the ___stack_chk_fail function instead, and the problem magically went away. Once I made some minor changes to function names (such as changing "_main" to "main" and "_memset" to "memset@PLT"), I was able to produce the following output:

The answer: 1
Maybe it's this:5
gigem{A553Mb1YH...5...}

Where the "..." is replaced with garbage. I guessed that since it looks like it spells "Assembly", I could get rid of some of the end, and I was right. I believe this is probably due to the stack failure checking I got rid of, but since the program was able to run, it only produced a little bad output. I also guessed the answer from what I had seen in IDA when I loaded the executable:

```
call    _memset
mov     [rbp+s], 41h ; 'A'
mov     [rbp+var_F], 35h ; '5'
mov     [rbp+var_E], 35h ; '5'
mov     [rbp+var_D], 33h ; '3'
mov     [rbp+var_C], 4Dh ; 'M'
mov     [rbp+var_B], 62h ; 'b'
mov     [rbp+var_A], 31h ; '1'
mov     [rbp+var_9], 59h ; 'Y'
mov     [rbp+var_1C], 0
```

4 KeyGenMe

This puzzle was by far the hardest to solve. I loaded the executable to IDA and noticed from main that it was reading my input into a 64 byte buffer and checking its validity with verify_key before reading the file. Inside verify_key, I noticed that it always rejected strings that were less than 9 or more than 64 bytes because of the use of strlen and the techniques we have used in this class. I saw that the function enc took my string and encoded before passing it to strcmp() with "[OIonU2_<__nK<KsK". Thus, to get the string right, I knew I had to give the program a string (the "key") that would encrypt to this string.

Inside the enc function is a loop with a large body for manipulating each character of the string as can be seen below.

```
1  loc_960:
2  mov     eax, [rbp+var_10]
3  movsxd  rdx, eax
4  mov     rax, [rbp+s]
5  add     rax, rdx
6  movzx   eax, byte ptr [rax]
7  movsx   eax, al
8  # With the ith char in register A:
9  # Do something and put the result
10 # in register C
11 # Replace the ith char with reg C:
12 mov     eax, [rbp+var_10]
13 movsxd  rdx, eax
14 mov     rax, [rbp+var_8]
15 add     rax, rdx
16 mov     edx, ecx
17 mov     [rax], dl
18 # Set [rbp + var_11] to that value
19 mov     eax, [rbp+var_10]
20 movsxd  rdx, eax
21 mov     rax, [rbp+var_8]
22 add     rax, rdx
23 movzx   eax, byte ptr [rax]
24 mov     [rbp+var_11], al
25 add     [rbp+var_10], 1
```

Here I isolated everything except the bit manipulation. You can see from lines 1-7 that the loop iterator is [rbp + var_10], and that the string pointed to by [rbp + s]. Before this loop, malloc is called for a buffer of size 64, and the handle for it is put in [rbp+var_8], so at the end of the loop, that buffer is being written in the corresponding place with the encoded string, and the variable [rbp+var_11] is being set to the current value (important at later steps). With all that said, let's look at the guts:

```
27 lea     edx, [rax+12]  # add 12
28 movzx   eax, [rbp+var_11]
29 imul    eax, edx    # multiply by var_11
30 lea     ecx, [rax+17]  # add 17
31 # Do something crazy...
32 mov     edx, 0EA0EA0EBh
33 mov     eax, ecx
34 imul    edx
35 lea     eax, [rdx+rcx]
36 sar     eax, 6
37 mov     edx, eax
38 mov     eax, ecx
39 sar     eax, 31
40 sub     edx, eax
41 mov     eax, edx
42 # multiply by 70 and add 48
43 imul    eax, 70
44 sub     ecx, eax
45 mov     eax, ecx
46 lea     ecx, [rax+48]
```

I noticed that the first four lines and last four lines were simple and familiar, so I knew that it had to be something along the lines of:

int x = (input[i] + 12)*var_11 + 17;
x = // do something to x...
x = x*70 + 48;

To figure out what was going on, I Googled "keygen ctf" to see if there was ever a CTF in the past where this kind of algorithm was used. I found this:
https://github.com/x0r19x91/hack-con-solutions/blob/master/match.c.

Inside one of their loops, there is a mod operation, and when I looked at the compiled assembly, I noticed that there was the same pattern as in lines 32-41 of multiplying by some large constant then shifting right a few times to undo the damage. I wasn't sure how it exactly worked, so I just tried putting what I had into Godbolt with just some random mod operation, and it started showing me the right patters with the wrong numbers. Since I saw that the number was being multiplied by 70, I tried that, and I was right. The exact assembly from the problem was produced from the following source code:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/*
    Answer: gigem{k3y63n_m3?_k3y63n_y0u!}
 */

char key[] = "[OIonU2_<__nK<KsK";

char* enc(char* str) {
    char* buf = (char*) malloc(64);
    int len = strlen(str);
    char var_11 = 'H';
    for (int i = 0; i < len; ++i) {
        buf[i] = ((str[i] + 12)*var_11 + 17) % 70 + 48;
        var_11 = buf[i];
    }

    return buf;
}

bool verify_key(char* str) {
    if (strlen(str) > 9 && strlen(str) <= 64)
    {
        char* s2 = enc(str);
        char* s1 = key;
        return strcmp(s1, s2) == 0;
    }
    else return false;
}

```

The enc and verify_key functions exactly matched the assembly when compiled. Then, to figure out what a possible key could be (there are a couple because of the mod operation), I wrote the following code to basically run through all the possible values a character could take on, encrypt it, and check against the key:

```
33 int main()
34 {
35     char s[65];
36     /*
37     setvbuf(stdin, 0, 2, 0);
38     puts("\nPlease Enter a product key to continue: \n");
39     fgets(s, 65, stdin);
40     */
41     int len = strlen(key);
42     for (int i = 0; i < len-1; i++) {
43
44         // Find that character
45         char* temp = enc(s);
46         s[i] = 32;
47         s[i+1] = 0;
48
49         while (temp[i] != key[i]) {
50             free(temp);
51             s[i]++;
52             temp = enc(s);
53         }
54         free(temp);
55
56     }
57
58     if (verify_key(s))
59     {
60         printf("passed verification...\n");
61         if (fopen("flag.txt", "r") == 0) printf("Too bad it's only on the server!");
62     }
63     else {
64         printf("failed verification...\n");
65     }
66 }
```

The comments above the loop should pretty closely match what is in the main source code. Then the loop on lines 42-56 starts the character ' ' and increments through the ASCII sequence, encrypting the entire string each time, since each character depends on the character before it (because of [rbp + var_11] in enc()). I'm not sure why I had to set the bounds to len - 1 in the for loop, but when I did, I got the key:

$*Z2S”+’)””+’+$(

When I gave this to the remote server, it gave me back the flag:

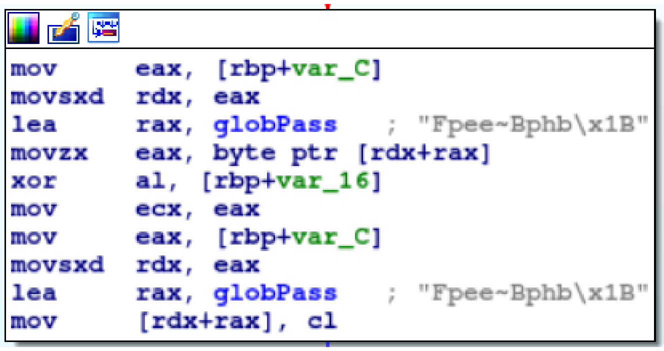
gigem{k3y63n_m3?_k3y63n_y0u!}

5 NoCCBytes

I first ran this program only to see it ask me for input. Then, I opened the ELF in IDA to see that the function check(char*) was reading a copy of the string I gave it. I was tempted to try ”Fpee Bphb\x1B” since it appeared in the code so many times, but I knew that eventually, the text I entered had to be manipulated in some way to match that. I recognized that the first code structure the string goes through in charck() is a meaningless nested for loop meant to check the string for 0xCC bytes by XORing with 85 and comparing to 153. Although it looks like it is adding i*j (where i and j are the indices of the inner and outer for loops) to a

saved number, it subtracts it once for every time it adds it.

Thus, all that matters in the check() function is what happens right before returning. The unchanging value previously mentioned is 0x11. At the end of the function, it can be seen in IDA that the function is XORing each character with 0x11.



This snippet comes from the last part of the last loop in check(), where all the action is happening. The first four lines load "Fpee Bphb\x1B" as a global variable. Then the fifth line xor's the ith character with 0x11, and the rest of the block just replaces that character in the local variable. Thus, to find the correct input to the program, I used the xor calculator at http://xor.pw/# to get the string "WattoSays" from the string "Fpee Bphb". Once I entered this, I was directed to go to the server, and once I queried the server using the nc command, I got back "gigem{Y0urBreakpo1nt5Won7Work0nMeOnlyMon3y}", the flag.

6 ReversingErirefvat

This puzzle was similar in spirit to the rock problem we did in class. I first attempted the same technique as 042, where I got rid of the MacOS headers and footers. It still took a lot of demangling to get the function names to compile. In particular, it was difficult to get the round() function to work because it depended on the math library. After I compiled the assembly with the "-lm" flag, it linked the math library automatically. Once I got it to compile to an executable and dissassembled it in IDA, it was much easier to look at.

Even in IDA, however, it was a little hard to look at, so I decided I would compile it with the debug flag and use gdb to step through the execution. In gdb, I set a breakpoint for the last label in the assembly, which was section LBB3_7. I ran the program until it came to the breakpoint and issued the following to see the strings on the stack:

x/96s $sp

And got the full stack trace at the end of the program. Here is an excerpt from that trace:

```
0x7fffffffddc4: "\006"
0x7fffffffddc6: ""
0x7fffffffddc7: ""
0x7fffffffddc8: "\377\265\360\226ppuvc"
0x7fffffffddd2: ""
0x7fffffffddd3: "\031ip"
0x7fffffffddd7: ""
0x7fffffffddd8: "\031\002\r\003"
0x7fffffffdddd: ""
0x7fffffffddde: ""
0x7fffffffdddf: ""
---Type <return> to continue, or q <return> to quit---
0x7fffffffdde0: "abcdefghijklmnopqrstuvxyz"
0x7fffffffddfa: "UUUU"
0x7fffffffddff: ""
0x7fffffffde00: "\360\336\377\377\377\177"
0x7fffffffde07: ""
0x7fffffffde08: ""
0x7fffffffde09: "\301\236\232\277\342ك\360HUUUU"
```

At 0x7fffffffddc8, I could see the string that was manipulated in the loop I noticed in IDA. I found it by simply counting up from the stack in the trace from gdb. I knew the offset from the stack because the string being manipulated in the executable was in [rbp + var_3D], meaning that all I had to do was go from $sp - 96 and add 0x3D - 4 (for the offset from the base pointer). That put me right on the letter 'u' at 0x7fffffffddce. Thus, I found "gigem{uvc}" was the answer, which makes sense because the first loop in the assembly has a limit of 3, as opposed to the second loop, which has a limit of 5.

Just a disclaimer: I found that this was a ROT 13 when I was trying to reverse the functions called on each letter (thus "Erirefvat" becomes "Reversing"). I just never did the work of figuring out what the 3-character string was before it was rotated, although according to rot13.com, it should be "hip."