Alexander Lotero

Offensive Security

Spring 2023

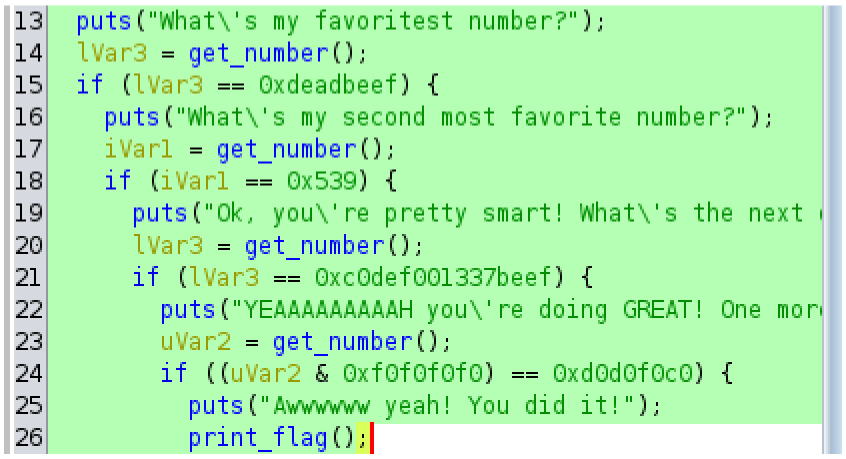
Week 3 - Reverse Engineering 1

This week's challenges included "Numerix," "strops," "recurse," and "Postage." For credit, I completed "Numerix" and "recurse."

Numerix

The first challenge I completed this week was "Numerix." This challenge provided us with an htm file "numerix.htm" and a netcat command to connect to the server to solve the challenge. Upon connecting via netcat and entering our netid at the prompt, a short greeting is printed to our terminal as well as the prompt "what's my favoritest number?" "Can you get them from my diary?" In this case, their diary is the "numerix.htm" file. Double-clicking the file to open it with the default app opened our Firefox browser to what appears to be random, useless text. However, because this week is all about reverse engineering, we proceeded by opening the file with Ghidra, the NSA's open-source reverse engineering suite (1).

Next, upon importing the file into Ghidra, we hoped to start by getting familiar with the program flow. To do so, we began by finding the *main* function within the *Functions* folder of the *Symbol Tree* tab. Lucky for us, upon selecting *main*, in addition to the main Ghidra window showing the function in assembly language, the "decompile" window is populated with *main* function written in C:



```
  puts("What\'s my favoritest number?");
  lVar3 = get_number();
  if (lVar3 == 0xdeadbeef) {
    puts("What\'s my second most favorite number?");
    iVar1 = get_number();
    if (iVar1 == 0x539) {
      puts("Ok, you\'re pretty smart! What\'s the next
      lVar3 = get_number();
      if (lVar3 == 0xc0def001337beef) {
        puts("YEAAAAAAAAAH you\'re doing GREAT! One mor
        uVar2 = get_number();
        if ((uVar2 & 0xf0f0f0f0) == 0xd0d0f0c0) {
          puts("Awwwwww yeah! You did it!");
          print_flag();
```

At this point we can see what numbers the program is expecting to receive and mark as correct in order to continue execution.

The first number that the program is expecting, “what’s my favoritest number?,” is *0xdeadbeef*. Attempting to enter this hexadecimal value as-is returns an incorrect message and our connection is terminated. At this point we made the correct assumption that our responses must be delivered in base-10 (hexadecimal). Thus, the solution here is a simple conversion from hex to decimal, we used (2) to get the following *0xdeadbeef → 3735928559*. We can conclude that our answer was accepted because the program then prompts us for a second number.

The second number, “what’s my second most favorite number?,” is *0x539*. Much like the first, a simple conversion gives us *0x539 → 1337*. Again, our answer is accepted and we are greeted with a prompt for a third number.

The third number, “Ok, you’re pretty smart! What’s the next one?,” is *0xc0def001337beef*. Once again, a simple conversion gives us *0xc0def001337beef → 868613086753832687*. Another prompt is then provided.

The fourth and final number, “YEAAAAAAAAAH you're doing GREAT! One more!,” is a bit trickier, *(uVar2 & 0xf0f0f0f0) == 0xd0d0f0c0*. Unlike before, this is not a simple case of converting to decimal, rather we have to solve for A in *A&B=C* given B=*0xf0f0f0f0* and C=*0xd0d0f0c0* where “&” is the bitwise AND operator. Note, this is not a simple case of algebraic “solve for x” in view of the fact that bitwise AND is irreversible because “information is lost by applying the operation” (3). Consider the following:

A = ?

B = 0110

C = 0010

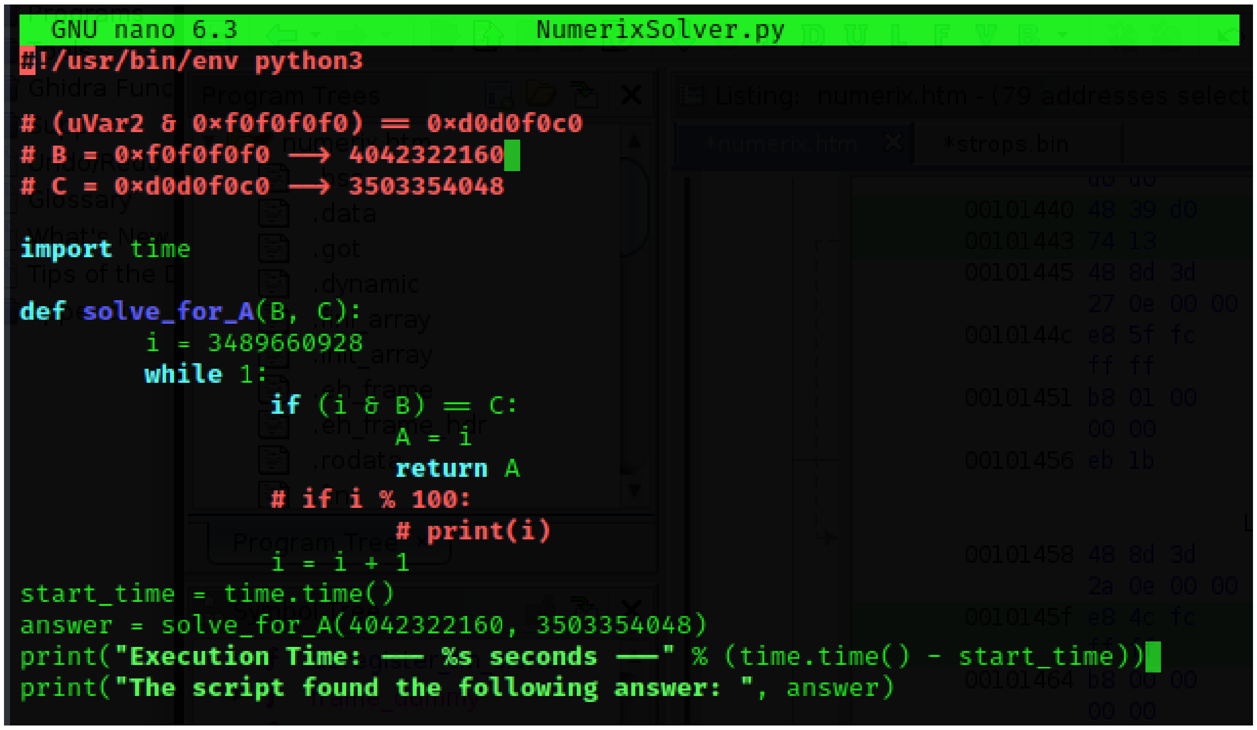
“A” can be any of the following: 0010, 1011, etc. The presence of a “0” in “C” leaves multiple possibilities for what “A” could be. Thus, at this point we chose to create a python script to ultimately guess at “A.” First, some steps we took before writing the script to make the writing process easier and include a shorter execution time:

1. Converted our “B” and “C” to decimal, respectively: *B=0xf0f0f0f0 → 4042322160*, *C=0xd0d0f0c0 → 3503354048*
2. Converted our “B” and “C” to binary to see if we can narrow our search for the correct answer:

```
                          A = 1101????1101????1111????1100????
Bitwise AND (&)           B = 11110000111100001111000011110000
----------------------------------------------------------------------
                          C = 11010000110100001111000011000000
```

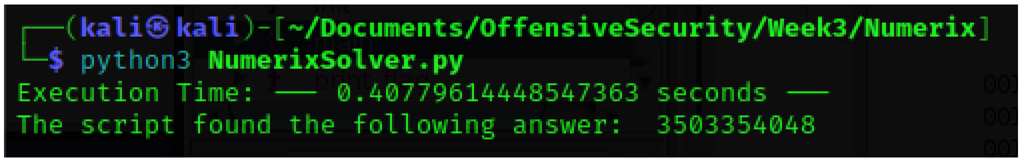
Given that only the “0” digits in “C” leave any ambiguity for the values of the corresponding digit in “A,” we can conclude that our “A” value is *1101????1101????1111????1100????*, so we shall start our brute-force search at *11010000000000000000000000000000 → 3489660928 (in decimal)*, because we know the correct answer is at least this large.

We put together the following python script for local testing:



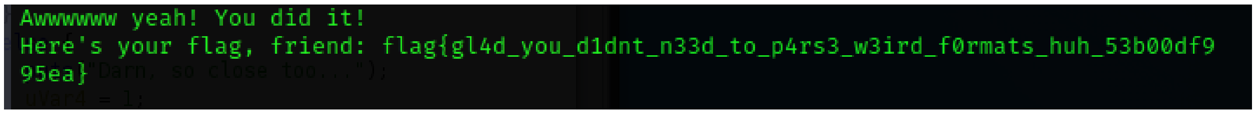
```
  GNU nano 6.3                    NumerixSolver.py
#!/usr/bin/env python3

# (uVar2 & 0xf0f0f0f0) == 0xd0d0f0c0
# B = 0xf0f0f0f0 --> 4042322160
# C = 0xd0d0f0c0 --> 3503354048

import time

def solve_for_A(B, C):
        i = 3489660928
        while 1:
                if (i & B) == C:
                        A = i
                        return A
                # if i % 100:
                        # print(i)
                i = i + 1
start_time = time.time()
answer = solve_for_A(4042322160, 3503354048)
print("Execution Time: --- %s seconds ---" % (time.time() - start_time))
print("The script found the following answer: ", answer)
```

We create a function *solve_for_A()* with a while loop that will run until the correct value of “A” is found and returned. Beginning with our starting value described above, we test to see if a given value bitwise AND with “B” is “C,” iterating by 1 each time. We include the python “time” library to track our execution time. Once the correct value for A is found it is returned and printed to our terminal like so:



```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week3/Numerix]
└─$ python3 NumerixSolver.py
Execution Time: --- 0.40779614448547363 seconds ---
The script found the following answer:  3503354048
```

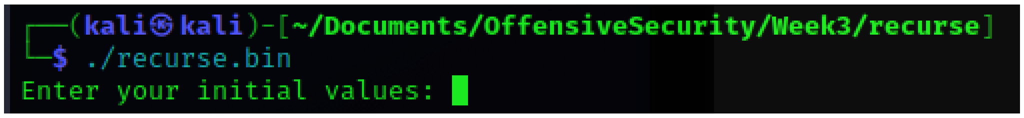
We enter the output of our script as the answer to the program’s final question to be rewarded with a “You did it!” message as well as our flag:



```
Awwwwww yeah! You did it!
Here's your flag, friend: flag{gl4d_you_d1dnt_n33d_to_p4rs3_w3ird_f0rmats_huh_53b00df9
95ea}
```

Recurse

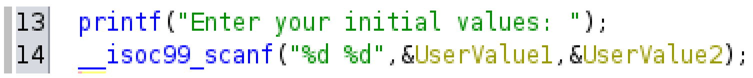
The second challenge that I solved for credit this week was “recurse.” Much like “Numerix” before it, recurse provides us with a brief snippet of introductory text, a netcat command to connect to CTFd on this challenge’s port, and a compiled *recurse.bin* binary file. The first step was to execute the binary, either locally or via netcat, and observe its behavior. We are greeted with the following prompt:



```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week3/recurse]
└─$ ./recurse.bin
Enter your initial values: 
```

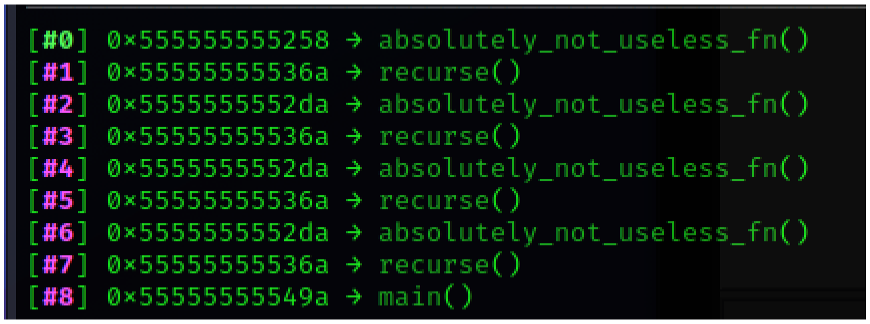
The program expects the user to input an unspecified number of “values.” An incorrect answer receives a simple “Nope!” message printed to the terminal and the program is terminated.

It was at this point that we opened the file within Ghidra (1). Upon importing the binary and waiting for Ghidra to complete its analysis, we select the *main* function identified within the “Symbol Tree” window. Within the *main* function, to make our lives a bit easier, we rename some of the function’s variables from the generic Ghidra defaults to something easier for us to identify as we step through the program:



```
printf("Enter your initial values: ");
__isoc99_scanf("%d %d",&UserValue1,&UserValue2);
```

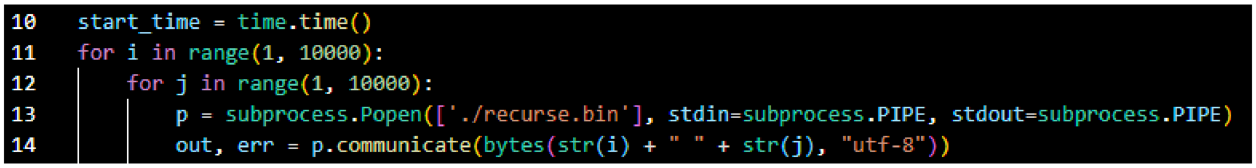
Here, we note that the “unspecified number of values” reported previously is actually two, the program is looking for two initial values. Further, the *%d* symbols found in the *scanf* function also tells us that the two initial values are C integers. Next, we attempted to step through the executing program, both by following along with the assembly and pseudo-C in Ghidra, and with the gef extension (4) for the gdb debugger. After some time attempting to follow two randomly guessed initial values through execution, we found ourselves discourage by the growing number of jumps between the *recurse()* and *absolutely_not_useless_fun()* functions:



```
[#0] 0x555555555258 → absolutely_not_useless_fn()
[#1] 0x55555555536a → recurse()
[#2] 0x5555555552da → absolutely_not_useless_fn()
[#3] 0x55555555536a → recurse()
[#4] 0x5555555552da → absolutely_not_useless_fn()
[#5] 0x55555555536a → recurse()
[#6] 0x5555555552da → absolutely_not_useless_fn()
[#7] 0x55555555536a → recurse()
[#8] 0x55555555549a → main()
```

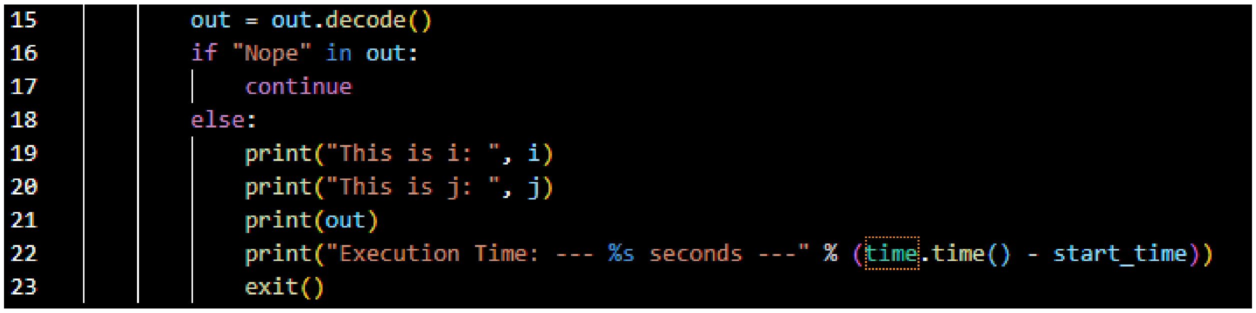
I got to the point that I decided that I was spending too much time going deeper into this rabbit hole without feeling any closer to understanding what the correct initial values should be. As I put it at the time, “walking through the code with two ‘guesses’ for the values just seems to tell me that they are the wrong values, it does not seem to tell me what the right values might be.”

It was at this point that I pivoted to adopt a new approach. While this challenge indicates reverse engineering, I instead attempted brute forcing the values using a python script against the local binary. Available with this assignment submission, my script features a nested *for* loop to try every combination of two integers between 1 and 10,000. Ten thousand felt like a suitably large guess at the time, and while it worked, I now know that it was much larger than it needed to be. Within the loops, my script uses the python subprocess library to launch the local binary and when prompted enters the values of the two iterator values of the *for* loops as seen below:



```
start_time = time.time()
for i in range(1, 10000):
    for j in range(1, 10000):
        p = subprocess.Popen(['./recurse.bin'], stdin=subprocess.PIPE, stdout=subprocess.PIPE)
        out, err = p.communicate(bytes(str(i) + " " + str(j), "utf-8"))
```

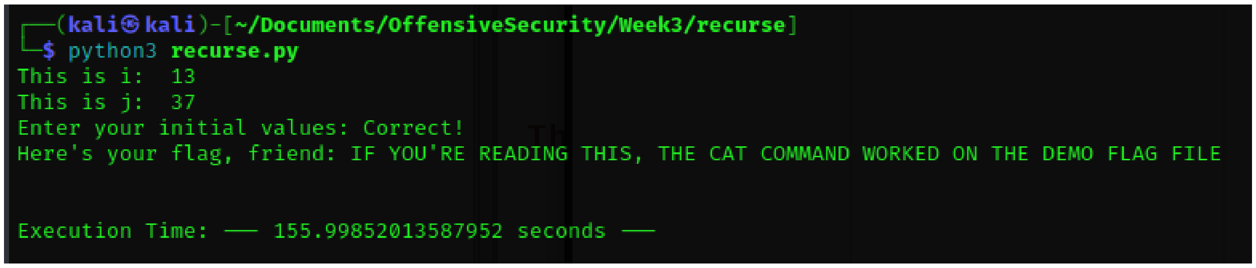
Initial value combinations are tested one-by-one starting with (1, 1) and technically ending with (10000, 10000), but as you will see, the correct values are found before reaching that point. Next, our script decodes the binary’s response to our guessed values and detects if the string “Nope” is found within the output - indicating that they were not the correct combination of initial values. If “Nope” is found, the loop continues. Otherwise, if “Nope” is not detected, then we have found the correct initial values:



```
            out = out.decode()
            if "Nope" in out:
                continue
            else:
                print("This is i: ", i)
                print("This is j: ", j)
                print(out)
                print("Execution Time: --- %s seconds ---" % (time.time() - start_time))
                exit()
```

Note, we calculate execution time to indicate that even with our oversized range of 1 to 10,000, our script still identifies the correct values within a reasonable amount of time.

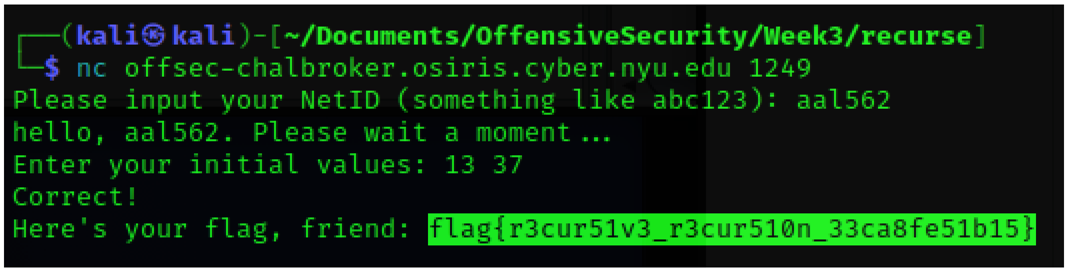
Finally, we execute our script and observe the output:



```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week3/recurse]
└─$ python3 recurse.py
This is i:  13
This is j:  37
Enter your initial values: Correct!
Here's your flag, friend: IF YOU'RE READING THIS, THE CAT COMMAND WORKED ON THE DEMO FLAG FILE


Execution Time: —— 155.99852013587952 seconds ——
```

As you can see, within roughly two-and-a-half minutes, our script identifies the correct initial values as 13 and 37. Ignore the “Here’s your flag” message as the binary is set to retrieve the flag from “/flag.txt” (as found by running the *strings* command on the binary), which when run on our local system is finding a test flag.txt file from a previous week’s challenge.

Finally, we netcat into the recurse challenge and try our newly discovered initial values to receive the flag and complete the challenge:



```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week3/recurse]
└─$ nc offsec-chalbroker.osiris.cyber.nyu.edu 1249
Please input your NetID (something like abc123): aal562
hello, aal562. Please wait a moment...
Enter your initial values: 13 37
Correct!
Here's your flag, friend: flag{r3cur51v3_r3cur510n_33ca8fe51b15}
```

References

1. *https://ghidra-sre.org/*
2. *https://www.rapidtables.com/convert/number/hex-to-decimal.html*
3. *https://stackoverflow.com/questions/45651753/is-it-possible-to-solve-equations-of-bit-wise-operators*
4. *https://github.com/hugsy/gef*
5.