Alexander Lotero
Offensive Security
Spring 2023

Week 4 - Reverse Engineering 2

This week's challenges were "Bridge of Death" and "dora."

Bridge of Death

The first challenge this week was "Bridge of Death" (BOD). Much like the previous week's challenges, BOD provided us with a compiled binary for download and a short description: "Good luck, brave Sir Lancelot." Our first step upon downloading the binary was to run it locally to get a sense of how it executes. Doing so greets us with an introductory message as well as the first question: "what is your name?" Because we were just poking around, we typed "Lancelot" which reveals the next question. As we will describe later in this write-up, the first question does not actually test anything, the program will accept any input. The second question, "what is your quest?" appears to accept two inputs followed by a press of the enter key to distinguish between the two. Random guesses at the expected inputs reveals the error/fail behavior of the program: "Auuuuuuuugh!" is printed to the screen.

It was at this point that we opened the program in Ghidra(1). When Ghidra's decompilation process completes, we are able to view what ultimately became the five important functions: *main()*, *question1()*, *question2()*, *func2()*, and *question3()*. As alluded to earlier, the *question1()* function does not actually perform any sort of comparison on our input, meaning there is no actual "right answer" for question1. Next, *question2()* confirms that the program expects two inputs that it then calls a recursive function *func2()* with and uses to determine *question2*'s return value. Because we faced a similar case of a recursive function in the previous week's "recursion" challenge, we made some minor tweaks to our *recursion.py* file and used it to solve for these two inputs. Ultimately, we copied the arithmetic within *func2/question2* and used a nested for loop to solve for Input1 and Input2 which we found to be 10 and 10 (see included *bodquestion2.py* file included with this submission). We verified these values by entering them at the question2 prompt while executing *bridge_of_death* locally. As

expected, we do not see the error/fail output reported above and instead receive the question3 prompt, thus proving we have found the correct values.

The prompt for the third question is as follows: “what is the air-speed velocity of an unladen swallow?” By examining the function in Ghidra, we see that once again the program is expecting two inputs which we shall refer to as UserInput1 and UserInput2 for the remainder of the write-up. Additionally, the variable tested and iterated on within the first if statement of the function (inside the do…while loop) will henceforth be referred to as ExitCondition:

```
ExitCondition = 1;
do {
  if (9 < ExitCondition) {

...

    ExitCondition = ExitCondition + 1;
  } while( true );
}
```

Note, when testing for a proper solution to *question3()* in the *main()* function, it is indicated that *question3()* must return 0 in order to be considered as “answered correctly.” Within *question3()* there are two if statements, or cases, in which the return value will be set to 1 and the question will be considered as “answered incorrectly”:

1. If either of our two inputs exceed 0xff (256 in decimal), or
2. If the following element within the forestOfEwing memory offset is not equal to ExitCondition:
   *(char)forestOfEwing[(ulong)UserInput1 * 0x100 + (ulong)UserInput2]*[1]
3. Additionally, we cannot write over the stack canary.

These requirements ultimately mean we must find two values that are both between 1 and 256, and satisfy the following equation:

*forestOfEwing[(UserInput1 * 256) + UserInput2] == ExitCondition*

Now as we can see in the images above, each iteration of the loop will increase the value of ExitCondition by 1, so each time we are asked for input we need to solve for the inputs such that once arithmetic is performed, the corresponding element in forestOfEwing is equal to the current value of ExitCondition. For example, in the first iteration we need to find an occurrence of the value 1 within forestOfEwing, in the second iteration we need to find an occurrence of the value 2, and so on. To do so we used the find function within gdb(2) while stopped at a breakpoint in *question3()* like so:

[1] (char)forestOfEwing[(ulong)UserInput1 * 0x100 + (ulong)UserInput2] = forrestOfEwing[(UserInput1 * 256) + UserInput2]

```
gef➤  find 0x555555554000, 0x7fffffff000, 0x09
0x55555555480c
0x555555556eb8
0x555555557eb8
0x55555555e4e2 <forestOfEwing+25794>
```

The image above shows the case of finding the occurrence of 9 for the ninth iteration. Once we have identified which offset within forestOfEwing corresponds to which value needed for each iteration, we next need to solve for each iteration of UserInput1 and UserInput2 to reach that offset. We created a python script to solve for each of these values (*bridge_of_death_forestSolver.py* is included as part of this submission). The output of our script, in this case corresponding to the ninth iteration like the image above appears like so:

```
This is for offset  25794
This is UserInput1:  100
This is UserInput2:  194

Execution Time: --- 0.032080888748168945 seconds ---
```

We now have our inputs to satisfy the *question3()* requirements and avoid the if statements that cause an “answered incorrectly” return to main. We test our inputs by connecting and submitting our values to the live CTFd server. We verify that we have identified the correct values and completed the challenge by receiving our flag:

```
┌──(kali㉿kali)-[~/Documents/OffensiveSecurity/Week4]
└─$ nc offsec-chalbroker.osiris.cyber.nyu.edu 8005
Please input your NetID (something like abc123): aal562
hello, aal562. Please wait a moment ...
Stop! Who would cross the Bridge of Death must answer me these questions three, ere the other s
ide he see.

What is your name?
My name is Sir Lancelot of Camelot.
What is your quest?
10
10
kek
What is the air-speed velocity of an unladen swallow?
203
125
29
64
159
68
68
55
75
163
171
172
1
83
115
15
100
194
Right. Off you go.
Here's your flag, friend: flag{@_W1tch_W3'v3_G0t_@_W1tch!!!!!!!!_a565e1a1a07b}
```

Note, our write-up attempts to get straight to the point. Much time was spent figuring out each of these steps, but we chose not to discuss deadends and incorrect solutions (e.g. deploying angr(3) to solve for *question3()*) here. Further elaboration is available upon request.

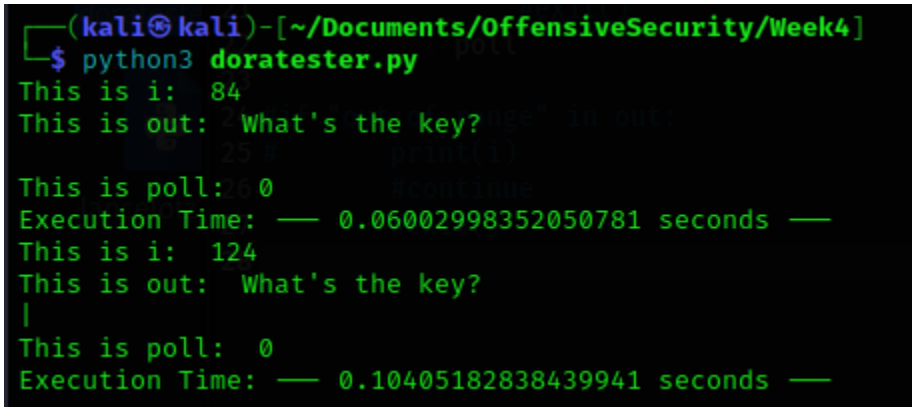
Dora

The second challenge this week was “dora.” Once again we are given a compiled binary to download, *dora.htm*. Like before, we start by observing how the program executes when run locally, and it is quite simple. All we get is a simple prompt “What is the key?” Trying different numbers typically leads to a segmentation fault or an illegal hardware instruction message being printed as the program terminates.

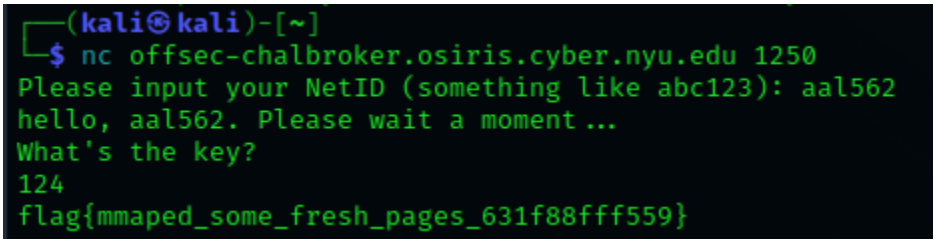
Thus, our next step was to examine the program in Ghidra (1). Doing so reveals that outside of the *get_number()* function also found in “Bridge of Death,” the only useful function that Ghidra appears to identify is a *main()* function. Within *main()* we see several variables defined, a variable that we ultimately renamed “Flag” reserves 4096 bits in memory, 80 bytes of which are initialized to seemingly random values. User provided input is then taken via the *get_number()* function - we renamed this variable UserInput. UserInput is then tested - if it is less than 0 or greater than 255, an “out of range” message is printed and the program is terminated. Next, we arrive at what was ultimately the most important piece of code for solving the challenge: a for loop that executes 80 times. With each of the 80 executions, the corresponding byte in “Flag” is replaced by the result of the byte being XORed with UserInput. Once this loop completes, the 80 bytes in memory at &Flag are cast to a function pointer and executed, hence the segmentation faults and illegal hardware instructions when we tested arbitrary numbers earlier - the program is attempting to interpret these values as pointers to a function.

Thus we have our task: find a value, between 1 and 255 that when entered and XORed with the bytes in Flag, will generate a valid function call, specifically one that provides us with our flag. To do so, we wrote a short python script, *doratester.py* (available as part of this submission), utilizing the “subprocess” library to start “an instance” of *dora.htm* and repeatedly try different values as the key. Specifically, we have a for loop that begins at 0 and iterates by 1 until 255, trying every value in between (the valid range defined in the program, see above). We track whether the input was correct or not using the subprocess library’s “poll” method (4). With poll we can check if the subprocess (our instance of *dora.htm*) terminated if *poll != 0* - i.e. we encountered

the segmentation fault or illegal hardware instruction error, or if the input was accepted as correct if *poll = 0*. Running our script provides us with two “correct” inputs:



We tested both of these values via the netcat connection to the CTFd server. 84 is some kind of false positive, but I am unsure as to why. 124 on the other hand, was the correct answer and when sent to the server provided us with out flag indicating our completion of the challenge:



Note, much of our work for this challenge is expressed through the *doratester.py* Python file. If any step of our process remains unclear, I encourage you to give it a look.

References

1. *https://ghidra-sre.org/*
2. *https://www.sourceware.org/gdb/*
3. *https://angr.io/*
4. *https://docs.python.org/3/library/subprocess.html*