CSCI 476: Lab 05 – Team 14

Nathan Stouffer Mitchell Faris Emilia Bourgeois Michaela Lightner

February 26, 2020

Level 1

Level 1 begins by modifying the shell code that we want to execute on the host system. Since we are Team 14, we must create a file called "team14.jpg" in the "/tmp/CTF" directory on the host system. Our modified shell code is depicted in Figure 1. It should be noted that all array elements are 4 characters long since each character takes up one of the four available bytes in a memory location. Additionally, all the values are placed in reverse order since they are processed on a stack (which operates on Last In, First Out).

Figure 1: Shellcode to create file named "team14.jpg"

During the competition, we did not take screenshots of Levels 1 and 2. We just used the same file on each level so we do not have our code was overwritten by the code for Level 3. So we substitute the screenshots with an entirely written explanation.

In Level 1, we know all the necessary information to carry out the buffer overflow attack. That is, we know the memory addresses of both &buffer how large buffer is (from which we can compute the distance to $ebp). This makes the buffer overflow attack quite easy since we can place the return address (which jumps to our NOPs) exactly 8 bytes above $ebp. This will run our malicious shell code.

Level 2

As mentioned in Level 1, we do not have a screenshot of our code for Level 2. We instead provide an example screen shot similar to what we used in the competition but likely with different values for &buffer and it's size.

Level 2 introduced a new challenge in that we do not know the exact size of buffer (although we do know the value of &buffer). To overcome this lack of knowledge we used a brute force technique. Let D be the distance from &buffer to $ebp. Since we know the value of &buffer, we just attack the host system by placing the target memory address at "D + 8" (distance to $ebp and then 8 bytes to the return address). Then we used a while-loop to adjust D to a new possible value. This attack succeeded, albeit, not cleanly.

Figure 2 shows how we would have written our solution if the value of &buffer was 0xBFFFDC50 and the possible sizes of buffer were 360 to 580. Note that we set the bounds of what we attempt for D slightly outside of the actual values for D to make sure that our attack succeeds (in case we made a small calculation error).

Figure 2: Example of Level 2 Python script

```
65
66 ###################################################################
67 D = 350                          # You need to change this value
68 while (D < 600):
69         target_address = 0xBFFFDC50 + D + 8
70         # Fill the return address field with the address of the shellcode
71         content[D:D+4] = (target_address).to_bytes(4, byteorder='little')
72         D += 4
73
```

Level 3

Level 3 introduces a new challenge in that we are given a range of possible values for both &buffer and the size of buffer. One strategy would be to run a brute force attack where we launch a query for each combination of &buffer and its size. However, there is a cleaner and much more efficient way to carry out this attack.

The ranges given to us are 0xBFFFDC00 to 0xBFFFDFFF for &buffer and about 320 to 560 for the size of buffer (we did not record the exact sizes of buffer but our code in Figure 3 bounds the size of buffer with the variable D).

Figure 3: Python script for Level 3

```
62
63 # Fill the content with NOP's
64 content = bytearray(0x90 for i in range(2000))
65
66 ###################################################################
67 D = 320
68 while (D < 560):
69         target_address = 0xBFFFDC00 + D + 1050
70         content[D:D+4] = (target_address).to_bytes(4, byteorder='little')
71         D += 4
72 ###################################################################
73
```

We now describe how we set up the malicious input. Figure 3 shows a while-loop that populates the "content" array (which is what we send as input). The problem that we need to solve is that we do not know where $ebp will be in relation to the beginning of our input. To overcome this, we just populate any *possible* location of $ebp with a valid return address.

We break the explanation of this code into two cases. One where we assume that &buffer is at 0xBFFFDC00 and one where we allow &buffer to exist anywhere in the range described above.

Let's begin with the case where we know exactly where &buffer is. It should be noted that this case is exactly the same as Level 2, however, we give this explanation since we did not solve Level 2 using this method. We know that the size of the array is greater that 320, so $ebp will never be less that 320 bytes away from &buffer. This is why we initialize D to 320. If we set the target address to "0xBFFFDC00 + D," then we will be telling the program to return to wherever $ebp is. But we want to jump to our NOP sled, so we must add an offset. Choose 50 to be the offset the we populate the range of indices [320:560] with "0xBFFFDC00 + D + 50" (which changes as D increments by 4).

Let us now consider the case where we do not know the exact address of &buffer. It should be noted that 50 was chosen arbitrarily as the offset in the previous case. There is no need to jump to a NOP that is close to $ebp. To get around not knowing the exact memory address, we could just jump to a memory that is guaranteed to have a NOP. That is, a memory location that could not be overwritten given a range of 0xBFFFDC00 to 0xBFFFDFFF. The subtraction 0xBFFFDFFF - 0xBFFFDC00 gives 1023 (in decimal). So we need only choose a value larger than 1023 and smaller that the size of the "content" array (which is 2000). It should be noted that we incorporate the size of buffer by incrementing the variable D. Populating the content array at [320:560] with "0xBFFFDC00 + D + 1050" guarantees that we land at a memory location with NOP and slide directly to our malicious code.

So we passed Level 3!

Level 4

We spent the remainder of class time working to solve Level 4. In Level 4, we are given no hints, so we don't have a notion of how large buffer is or where &buffer is. We were not able to pass this Level. As such, this section describes our best attempt to pass Level 4.

We first noted that the size of buffer was likely to be less than 2000 since the array we were using to send our malicious input has length 2000. This is not an assumption we would make in the real world, but we felt comfortable assuming this since this was an in class competition and that length was in the code we received. We then reasoned that our best option was to implement our solution Level 3, but for a whole bunch of starting memory addresses. Figure 4 shows the nested while-loop that generates the content array and attempts the attack. This did not succeed.

Figure 4: Python script for Level 4

```
62
63 # Fill the content with NOP's
64 content = bytearray(0x90 for i in range(2000))
65
66 ################################################################
67 mem = 0xBFFFD000
68 while (mem < 0xBFFFDFFF):
69         D = 50
70         mem += 4
71         while (D < 760):
72                 target_address = mem + D + 1050
73                 content[D:D+4] = (target_address).to_bytes(4, byteorder='little')
74                 D += 4
75 ################################################################
76
```

After the competition, we realized that we could have adapted our strategy. In the nested while-loop in Figure 4, there was no need to increment the memory address by 4 since our while loop that populates the "content" array accounts for 760 memory addresses. We could have incremented the variable "mem" by 760, and be just as effective as the code that we used. This would have traversed the memory of the host system much faster and could have possibly completed Level 4.

Another way that we could have improved our strategy was setting a higher loop guard for D. We only checked 760 memory values but we could have checked much higher values since the size of "content" was 2000.

Overall, we could have done a better job planning out our strategy for Level 4.