Exploiting a Remote Network Application

Exploiting the miniHTTPd Web Application Server

In this Lab we want to write an exploit to achieve arbitrary code execution using a vulnerability in the miniHTTPd web application server. This type of exploit is considered a remote exploit, since you can achieve this by speaking to the vulnerable application over the network. This means you must have network access to the system running the vulnerable application server. This application is also vulnerable to a memory corruption (buffer over) which is found when sending a long URL. Therefore, in order to exploit the application, we need to create an HTTP request with a malicious long URL that we can use to achieve our exploitation purposes.

Use the code given to start your exploitation process. This time since we will be sending the malicious code over the network, we will be using Kali Linux.

Note:

Before doing anything related to exploitation, make sure both your Kali Linux and Windows machine can communicate with each other. We can prove that by sending a ping request from Windows to Kali Linux, or the opposite (requires an update to the Windows firewall).

Task #1: Crashing the Application

Start the miniHTTPd either from within Immunity Debugger or by attaching it to the debugger and then from your Kali Linux machine run the exploit code like this:

```
# python exploit1.py
```

A) Did the application crash?

B) Could you see any “A”s in any of the registers? Where? Explain your findings.

Task #2: Locating Offsets using MSF

It turned out that we have control over EIP and our payload is somehow injected there. What we need to do now, is locate the offset to the EIP overwrite. In other words, we need to find how many bytes we need to send to control EIP. In the previous lab we did this using mona, this time let us use our Kali Linux system.

Now while on your Kali Linux system, type the following in the command field:

```
msf-pattern_create -l 9332
```

C) Is this something similar to what we got in our previous lab?

What we need to do now, is send this unique code instead of the 9332 “A”s. Restart the application from within Immunity or close and run it again, whichever works best for the application and for you. Do all the adjustments you need to our starting python code and run it again to send the malicious payload:

```
# python exploit2.py
```

Note: I recommend you rename the code to exploit2.py and so on for each change you do, this will help you later on understand your progress.

If everything you did correctly, then now when the application crashes, it would show a different four bytes value. This time it is not those 4 “A”s as we saw at the beginning.

D) What are the four bytes that have overwritten EIP this time?

Note down those four bytes, we need them to calculate the offset. This time we will be using another MSF command to find the offset. This could be done using the command below but with one adjustment to the command and that is to use the value you found in (F):

```
msf-pattern_offset -q  41414141
```

E) What is the offset that you found?

Super! We found how many bytes we need to overwrite EIP.

Up to this point, do you have a good understanding of what we are doing?
If not PLEASE ask your instructor…

Task #3: Code Adjustments and Payload Alignment

This time let us adjust the python code to overwrite EIP with four “B”s instead and make sure that our calculations are correct. We need to do three things here:

1. First part of payload will be “A”s of value you found in (E)
2. Second part will be the four “B”s
3. Third part will be the padding we need to reach 9932 bytes, but let us use “C”s instead of A or B, so we can recognize them

F) Write down below all the adjustments you made. Ask for help if needed.

REPEAT
Load miniHTTPd in Immunity, start it, and then send the payload from Kali. I am going to call this sequential number of tasks by uppercase REPEAT. Therefore, whenever you see in this document the word **REPEAT**, just do these steps again, which are:

1. Load and start miniHTTPd in Immunity Debugger
2. Send the payload using the python exploit#.py code

Now, I assume your application crashed, but the question is:

G) What are the four bytes that have overwritten EIP?

Also,....

H) Can you find any of the data you sent in any of the CPU registers? Where? Which did you find there, was it the “A”s or “B”s or was it the “C”s?

If everything went as we planned, then this means we have control over EIP and this was repeatable through the number of times we tested this. Let us move on to the next phase of our exploitation process.

Note: *reminder to check the rules of exploitability found in “Bug Hunting” slide #20*.

Task #4: Finding a Jumping Address

Now, since the code we sent is somehow located at the top of the stack, this means if we want to execute our payload, we need to find a "JMP ESP" instruction. This instruction when executed, will lead to EIP to point to the top of the stack and execute the code found there. The question now is how to find such an instruction?

In our previous lab we used Immunity Debugger to help us achieve this goal, this time let us use mona. Use the following command:

```
!mona jmp -r esp
```

Or we can specifiy which modules to search in by doing:

```
!mona jmp -r esp -m kernel32.dll
```

Or we can say, we want to search all of the loaded executables (remember DLLs are also executables). So we can do:

```
!mona jmp -r esp -m *
```

This will create a file within our configured mona directory on the desktop (assuming you configured mona in Lab #7). Open the document and ask your instructor to explain a little of the results found if you didn't understand them. We can use any of those addresses that don't have a null terminator (\x00) in them.

I) Do you have an idea why we cannot use any of those with a NULL Terminator?

This time let us update the python code to include the memory address of the JMP ESP we found. We can do that in different ways. Let us assume the address was "0xDEADBEEF" and since little endian is used, then we can either do:

```
eip = "\xEF\xBE\xAD\xDE"
```

OR

```
eip = struct.pack("<I", 0xDEADBEEF)
```

Use whichever is best for you, it won't matter, as both do the same thing. In Immunity Debugger use the goto memory address location to locate the address you are using and make sure you toggle a breakpoint there.

Now **REPEAT** and then continue...

J) Did Immunity Debugger stop at the breakpoint? If YES, proceed, if NO, please for help.

Task #5: Running Arbitrary Code

Everything is working perfectly as we want. Now let us adjust our exploit to run a message box that proves our exploitation was successful.

Add the code found in the payload.txt (shellcode) file given to your code and do the proper adjustments to proceed. Do not worry about how this code was generated, we will come to that, plus many others later, for now we want to complete our PoC.

One thing I would recommend before running your code, is to add some No Operation (0x90) instructions before the buffer holding the payload is reached. Sometimes this is best for alignment purposes and to make sure we jump into the right landing zone to run our injected code. Therefore, the final structure should be similar to this:

```
buffer = "\x41" * OFFSET
eip = struct.pack("<I", 0xDEADBEEF)
buf =  "" # → here should be the given payload
nops="\x90"*10
pad="\xCC" * (9332-len(buffer+eip+nops+buf))
payload=buffer+eip+nops+buf+pad
```

REPEAT and now run the code again…

If you were successful with your python code, then Immunity Debugger must have stopped at the EIP address we used. All you need to do now is hit the F7 button on your keyboard which is telling Immunity Debugger to continue to the NEXT instruction. This should bring you to the beginning of the payload (shellcode) we injected. You can go through the NOPS by using the F7 again for a couple of times (how much?) and then when reaching the shellcode, just hit the F9 button to instruct the debugger to continue execution.

Now, if you minimize Immunity Debugger and check…

K) Did you find the message box? What was the message?

Another question to ask:

L) What adjustments do you need to do in order to run this exploit without a debugger?

Congrats on your 2nd memory corruption (BoF) exploitation...