Exploiting a Local Client Application

Exploiting the VUPlayer Application

In this Lab we want to write an exploit to achieve arbitrary code execution using a vulnerability in the VUPlayer application. This type of exploit is considered a local exploit, since you can only speak to the vulnerable application through the running system. The application is vulnerable to a memory corruption (buffer over) which is found when loading a malicious playlist. Therefore, in order to exploit the application, we need to create a fake and malicious playlist that we can use to achieve our exploitation purposes.

Use the code below to continue.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
buffer = "\x41" * 3000
payload=buffer
f = open ("bad.m3u", "w")
f.write(payload)
f.close()
```

Task #1: Crashing the Application

Open the file using the Python IDLE and just hit the "F5" button. This should create the bad.m3u file. Start VUPlayer either from within Immunity Debugger or by attaching it to the debugger and then load the playlist in the application.

Deliverable #1: Did the application crash? (provide proof)

Deliverable #2: Could you see any "A"s in any of the registers? Where? Explain your findings.

Task #2: Configuring Mona and Locating Offsets

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 much bytes do we need to send to control EIP. This could be done in different ways, but in this lab we will be using a tool called "mona" to help us answer that question.

Now while in Immunity Debugger, type the following in the command field:

```
!mona
```

Deliverable #3: What did you get and in which window was this result found in?

To make locating the files that mona will generate, I prefer to configure mona to store the results in a directory that I can access easily, that is why, I am going to configure mona to store the results on a directory on the desktop named "mona". In order to do that, we need to run the following command:

```
!mona config -set workingfolder C:\Users\user1\Desktop\mona
```

If you got a successful message, proceed to the next part, if not ask your instructor for help.

A) Now run the following command:

```
!mona suggest
```

Deliverable #4: Go to the newly created directory. Did you find anything there? Explain.

Now since we have mona configured, let us create a unique pattern which we can use to test the offset location. Ask your instructor for more details if needed or not clearly understood. In order to achieve that, we will be using the following command:

```
!mona pattern_create 3000
```

Deliverable #5: Where should you find the results and what are they?

What we need to do now, is send this unique code instead of the 3000 "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 generate another playlist.

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.

Deliverable #6: 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 mona 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):

```
!mona pattern_offset 41414141
```

Deliverable #7: What is the offset that you found?

Super! We found how much 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
2. Second part will be the four "B"s
3. Third part will be the padding we need to reach 3000 bytes, but let us use "C"s instead of A or B, so we can recognize them

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

REPEAT

Run the python code again to create the malicious playlist, load VUPlayer in Immunity, start it, and then load the playlist. 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. We will be doing them many times!

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

Deliverable #8: What are the four bytes that have overwritten EIP?

Also,....

Deliverable #9: 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: you can also 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? Should I manually write it? Or what should I do? There are many ways to answer that question, but for sure one of the answers in our current case is not going to be "write our own instruction". We will be using an instruction which is already found in the memory address space of the application.

To achieve our goal we can also use mona, but this time I want to show you a quick way of achieving that using Immunity Debugger. All you need to do is right click in the disassembly instructions pane and then go to "Search for" → "All commands in all modules" and then enter the command we want to search for which is "JMP ESP".

What this does is, Immunity Debugger will search through all of the loaded executables (EXEs and DLLs) for the instruction you wanted. It will then give you the results in a single page. We can use any of those addresses marked in GREEN.

Deliverable #10: Do you have an idea why we cannot use any of those marked in RED?

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. When the breakpoint has been successfully added, it will highlight the address with the color cyan.

Now **REPEAT** and then continue… Did Immunity Debugger stop at the breakpoint? If YES, proceed, if NO, please ask 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" * (3000-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. All you need to do now is hit the F9 button and this should instruct the debugger to continue execution.

Now, if you minimize Immunity Debugger and check…
Deliverable #11: Did you find the message box? What was the message?

One important question that requires some thinking!!!.......
Deliverable #12: How would you deliver such an exploit to the victim?

Deliverable #13: Please reflect on what you learned from this lab.

Congrats on your first memory corruption (BoF) exploitation…