**CSCE 629 Lab 3**

**Winter 2019**

**Buffer Overflows / Exploitation**

**Assigned: Lesson 9, 17 Jan**

**Due: Lesson 15, 29 Jan, 1400**

You must include these questions in your submitted solution. In other words, your submission must include the question listed followed by your solution with the answer clearly indicated (e.g., put a box or circle around the final answer). You will work with your partner and submit one solution.

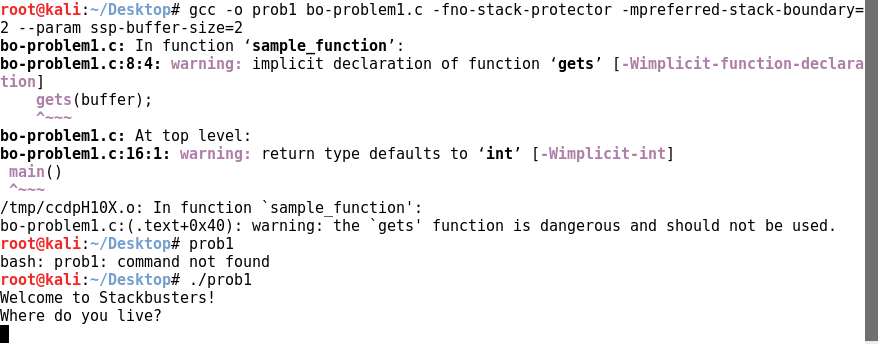
**Buffer Overflow**

**Use Kali-Linux-2016.2 for problems 1 and 2. Newer versions of Kali will not work.**

1. Consider the program *bo-problem1.c* on the file server. It is similar to programs we discussed in class. Your assignment is to crash the program using a buffer overflow technique.

1. Assuming no canary, draw a diagram (memory map) showing the contents of the stack just before the **gets** instruction is executed. This diagram should look very similar to those on the course slides.

**We compiled the program using the “gcc” command below. Note, in this screenshot we called the executable “prob1” however in later screen shots we refer to the program as “overflow.” This was to accommodate troubleshooting in GDB.**



**We filled in the table below using the GDB debugger.**

Memory Location | Value (low -> high bytes in each row) | Label

0xbffff404

0xbffff408

0xbffff40c

0xbffff410

0xbffff414

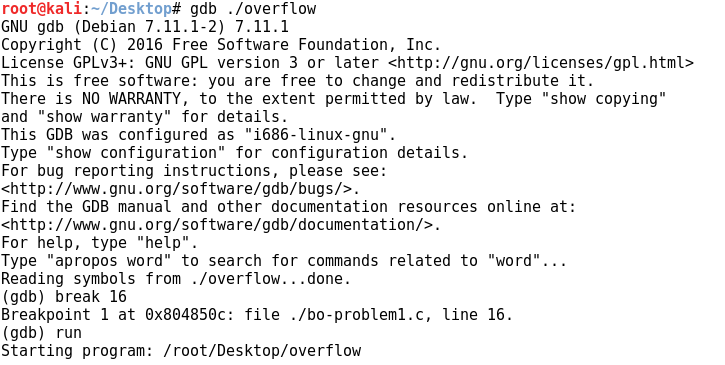
Buffer

Stored Instruction Pointer

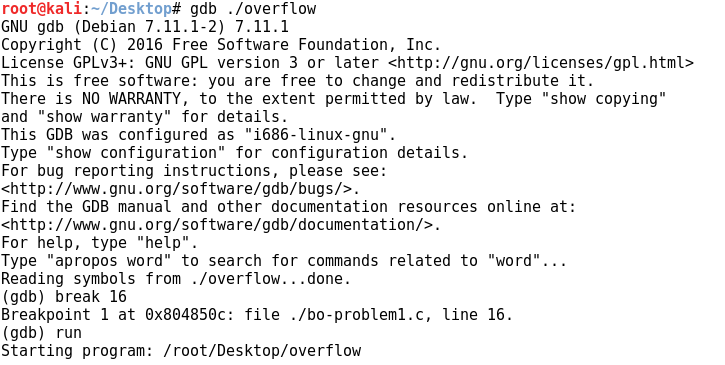
Main’s EBP

|  |  |  |  |
| --- | --- | --- | --- |
| 03 | 03 | 30 | 00 |
| 03 | 03 | 03 | 03 |
| 03 | 03 | 03 | 03 |
| bf | ff | f4 | 18 |
| 08 | 04 | 85 | 1e |

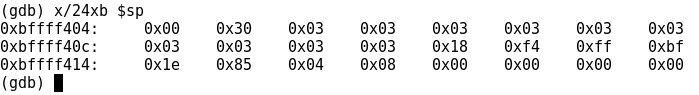
**We used the following gdb command to execute the program in debugging mode.**



**The “break 16” command to set a breakpoint at line 16. This stopped the program right before the “gets()” command was executed.**



**This command showed us the contents of the 24 bytes following the stack pointer during program executing (just before the gets command). We used the data to complete the table above.**

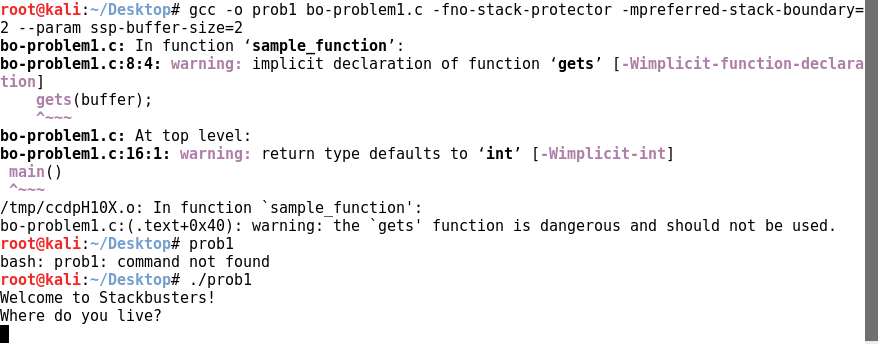


b. Predict the minimum number of characters the user must enter in order to crash the program. Remember, there is always a null character (\x00) appended to the end of strings. In other words, when you enter six B’s, seven bytes are actually stored in memory: BBBBBB\x00.

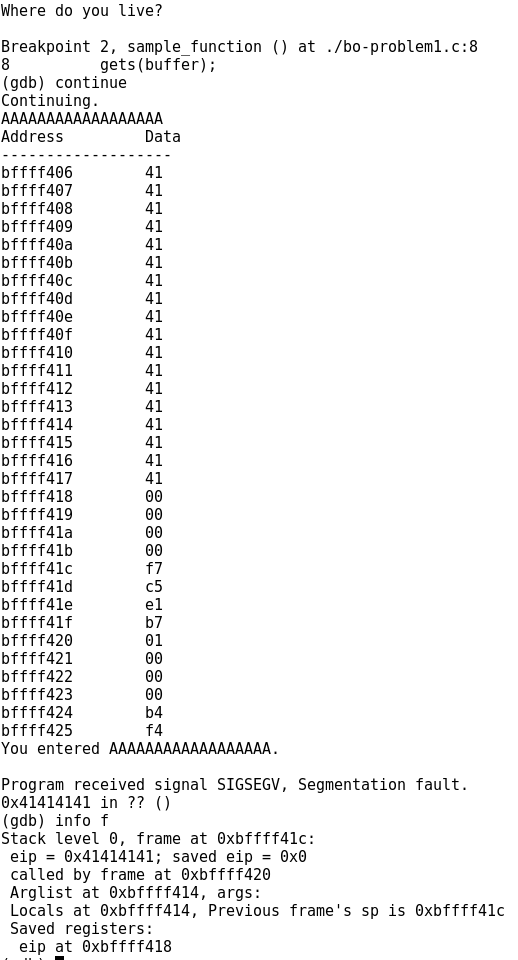
**Entering 14 bytes will crash the program. The first 10 written bytes will fill the buffer, the next 4 bytes will overwrite the stored (main’s) ebp, and the final byte (null terminator) will overwrite one byte in the stored instruction pointer. If we enter 18 bytes, this will allow us to redirect execution to a location of our choosing (the program will try to jump to the location provided in bytes 15-18 of the input).**

c. Compile the code in Linux using the **–fno-stack-protector** option to disable the canary (stack protection). Depending on the operating system (and version) you are using, this switch may not work and you’ll have to account for the canary in your answer. Also use **–mpreferred-stack-boundary=2** and **–-param ssp-buffer-size=2** to effectively turn off boundary alignment and set the minimum buffer size to be protected to 2 bytes.

**This screenshot shows successful compilation of the prog using the instructions above and successful execution:**



**The following screenshot shows the input of 18 bytes of the letter ‘A’ into the program. You can see at the bottom that it causes a segmentation fault.**



Null character following 18 bytes of input

Stored return pointer (4 Bytes)

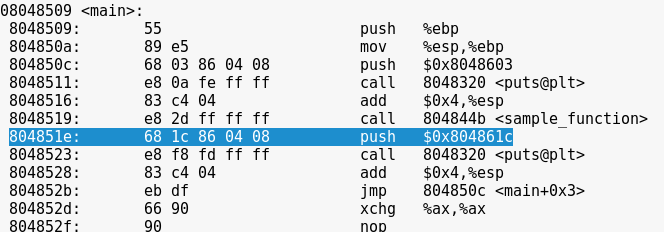
Stored EBP (4 Bytes)

Allocated Buffer (10 Bytes)

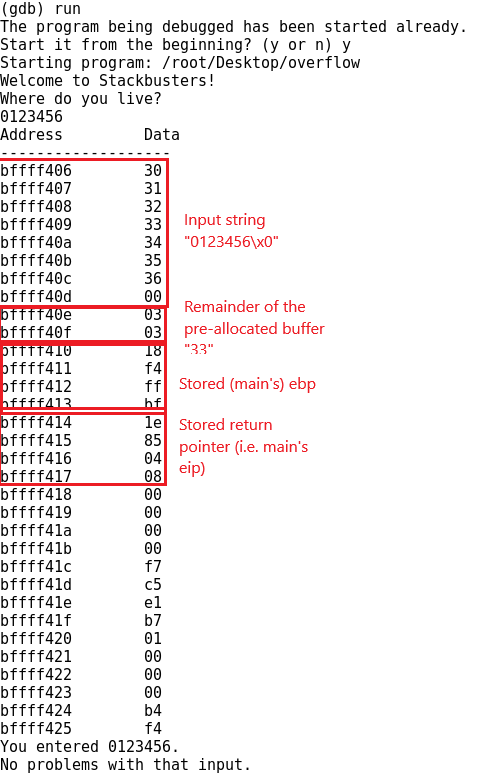
Input (18 Bytes)

d. Disassemble the executable (**objdump –d a.out**) to determine the return address that will be stored on the stack. What is the return address?

**The return address is 0x0804851e**

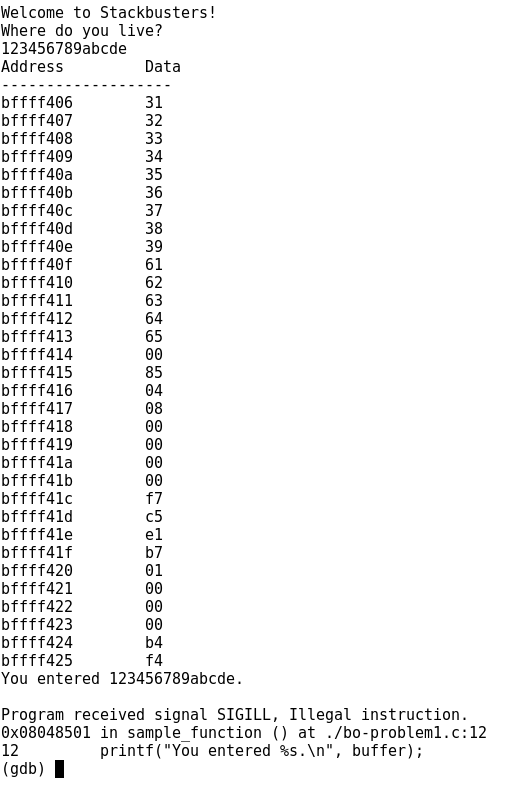


e. Execute your code and enter “**0123456**”. On your screenshot of the results, circle each field listed in part (a) above (e.g., buffer, canary if it exists, SFP, RP). Do you see the return pointer in the correct location?



f. Enter progressively longer input strings until the program “crashes”. That is, enter **12345678**, then **12345679**, then **123456789a**, then **123456789ab** etc. What is the minimum number of characters required to crash the program? Ensure your screenshot shows the crashed program with your input visible.

**The minimum string to cause a crash is “123456789abcde\x0**



1. Buffer (10 Bytes)
2. Stored EBP (4 bytes)
3. Null character (15th byte written to the stack)

g. Explain why it takes that number of characters to crash the program. Explain how the program crashed; in other words, how did the program become unstable? **Be very specific.**

As mentioned before, the minimum string to cause a crash is “123456789abcde\x0”. Including the null terminator, this consumes 15 bytes on the stack. There are 10 bytes allocated by the buffer for the string (1 screenshot above), then 4 bytes allocated by the program for the stored ebp (2 screenshot above). Finally, the next 4 bytes are the stored instruction pointer (e.g. the return pointer) (3 in screenshot above).

By writing 15 bytes, we write the minimum number of bytes necessary to overwrite at least one byte of the return pointer. Although gdb reports that the illegal instruction occurs at line 12, this is misleading. As can be seen from the output, the string “you entered 123456789abcd.” does successfully print; this means that the SIGILL is raised by the program’s attempt to execute line 13, which is the return statement.

As such, after the printf statement successfully completes, the program loads the (modified) stored instruction pointer into the EIP register and then attempts to execute the opcode stored at that memory address. However, since we’ve modified (at least) one byte of the instruction pointer, then instead of executing the instruction at 0x804851e, the program attempts to execute whatever is found at memory location 0x8048500 (4) -- which is in the middle of a call opcode (see below), and is therefore an illegal instruction. This causes the system to raise a SIGILL signal.



2. Consider the program *bo-problem2.c* on the file server. We will use it to investigate the behavior of the stack, crash the program, and actually change a variable value using a buffer overflow. The last of these is probably the most pernicious.

a. Compile the code in Linux using the **–fno-stack-protector** option to disable the canary (stack protection). Depending on the operating system (and version) you are using, this switch may not work and you’ll have to account for the canary in your answer. Also use **–mpreferred-stack-boundary=2** and **–-param ssp-buffer-size=2** to effectively turn off boundary alignment and set the minimum buffer size to be protected to 2 bytes.

We completed this step just like step (a) in the first problem using “twoverflow” instead of “prob1.” A screenshot is provided in that answer if needed.

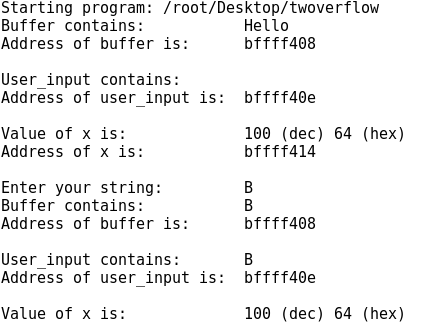
b. Execute the program and enter the letter B when prompted. Record the addresses of **x**, **user\_input** and **buffer** on the lines below.

**Buffer: bffff408**

**user\_input: bffff40e**

**x: bffff414**

****

****

c. Draw a memory map similar to the following example illustrating where each byte of the variables is stored in memory. You may shorten your addresses as shown below in red. Remember to include the SFP and return pointer.

|  |  |
| --- | --- |
| Memory Location (one byte) | Variable stored at that location |
| **bffff408 (sp)** | Buffer[0] = 0x42 = ‘B’ (strcpy(input)) |
| 09 | Buffer[1] = 0x00 = Null terminator |
| 0a | Buffer[2] = 0x6c= ‘l’ (pre-allocated) |
| 0b | Buffer[3] = 0x6c= ‘l’ (pre-allocated) |
| 0c | Buffer[4] = 0x6f = ‘o’ (pre-allocated) |
| 0d | Buffer[5] = 0x00 = Null terminator (pre-allocated) |
| 0e | User\_input[0] = 0x42 = ‘B’ (input) |
| 0f | User\_input[1] = 0x00 = Null terminator |
| 10 | User\_input[2] = 0x00 = Null (random) |
| 11 | User\_input[3] = 0x30 = ‘0’ (random) |
| 12 | User\_input[4] = 0xfb (random) |
| 13 | User\_input[5] = 0xb7 (random) |
| 14 | x = 0x64 = 100 (pre-allocated) |
| 15 | = 0x00 = Null = second byte of 4 byte int allocation |
| 16 | = 0x00 = Null = third byte of 4 byte int allocation |
| 17 | = 0x00 = Null = fourth byte of 4 byte int allocation |
| **bffff418** | ebp |

d. Execute the program several more times in order to complete the following table. Stop executing the program once you see a segmentation fault. Record the contents (not the addresses) of the three variables. The input column (first column) indicates how many B’s to enter when prompted.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input  (# of B’s)** | **buffer** | **user\_input** | **x (in hex)** |
| 4 | BBBB | BBBB | 0x64000000 (in LE) |
| 5 | BBBBB | BBBBB | 0x64000000 |
| 6 | BBBBBB |  | 0x00000000 |
| 7 | BBBBBBB | B | 0x42000000 |
| 8 | BBBBBBBB | BB | 0x42420000 |
| 9 | BBBBBBBBB | BBB | 0x42424200 |
| 10 | BBBBBBBBBB | BBBB | 0x42424242 |
| 11 | BBBBBBBBBBB | BBBBB | 0x42424242 |
| 12 | BBBBBBBBBBBB | BBBBBB | 0x00424242 |
| 13 | BBBBBBBBBBBB | BBBBBB | 0x42004242 |
| 14 | SEGFAULT |  |  |

e. Comment on the behavior of the three variables after you enter the B’s. That is, how are the three variables affected? I’m looking for a deep understanding of how the program is affecting memory. **Be very specific.**

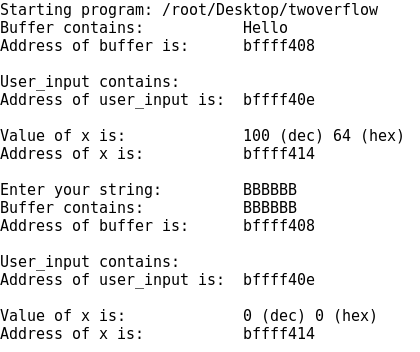
**Until the buffers are overrun at 6 x ‘B’, the program behaves as normal. Once “BBBBBB” (or similar string of length 6) is provided as input, the program attempts to write 7 bytes to the 6 byte user\_input buffer (“BBBBBB\x00”). This immediately causes x to be overwritten, because our input string consumes the 6 allocated user\_input buffer locations spanning from bffff40e - 413, and the 7th byte in user\_input overwrites memory location bffff414 with its value, which is the null terminator.**

**Once the 7 bytes are written, then “strcpy(buffer, user\_input)” copies those 7 bytes into the “buffer” buffer. However, since the “buffer” buffer sits on top of the user\_input buffer in the stack, then the 7th byte - “0x00” - actually gets written into the first byte of the user\_input buffer.**

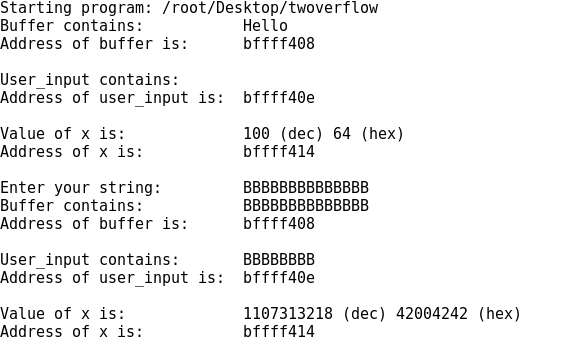
**In terms of the picture below, the strcpy() command overwrites user\_input[0] at bffff40e with the final character of the input string - “0x00”.**

**Finally the printf commands each run:**

* **The buffer printf command functions as expected since the first character in the buffer is appropriately located at bffff408 and it continues interpreting the input as a string until it encounters 0x00 at bffff40e.**
* **The printf command operating on user\_input[] begins by reading the location defined by &user\_input = bffff40e. However, since strcpy() overwrote that segment with “0x00” then the printf command immediately terminates, having found a string of length 0.**
* **Finally, the printf command operating on x works appropriately, if unexpectedly. It returns the integer-interpreted value of the 4 bytes spanning bffff414 - bffff418, however the least significant byte representing the DEC 100 was overwritten by the initial user\_input with “0x00”, so the integer value displayed is 0x00000000 = DEC 0.**



**As the number of ‘B’s increases, the behavior is more-or-less the same except that “B”s get pushed into lower buffers in the stack, instead of just null terminators. Eventually, this leads to x being interpreted as some number of 0x42 bytes followed by 0x00 bytes (from the initial overflowed input) and to user\_input being interpreted as containing len(input\_string) - 6 ‘B’s as the strcpy() command pushes each ‘B’ beyond the 6th into user\_input’s space, with a null terminator at the end of the overflowing ‘B’s.**



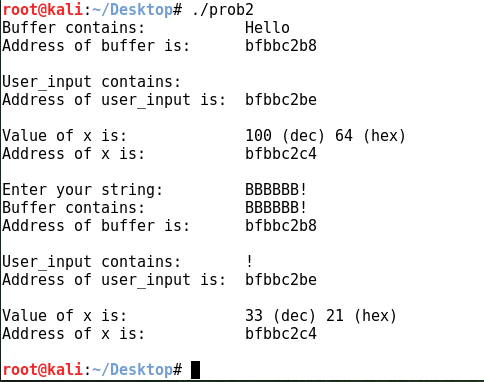
f. Why does the program generate a Segmentation fault? Recall that **main()** is actually a function called by the operating system.

**At the point of user input, there are 6 bytes allocated for that purpose and 4 additional bytes allocated below it for the integer x, below that are 4 bytes for the stored ebp, and below that are 4 bytes for the stored instruction pointer (which returns execution to the c runtime). As such, to overwrite the first byte of the stored instruction pointer requires writing 6 + 4 + 4 + 1 bytes, and an input string of 14 ‘B’s has length 15 (including the ending null terminator). This makes the variable x equal to ‘42004242’ in hex as shown above.**

g. Assuming this program is used by an e-commerce site and x represents the cost of a hub in dollars, your mission is to pay exactly $33 (in decimal) instead of $100. Discuss how to accomplish this diabolical feat and then do it!!!!!!!!! Ensure your screenshot is legible.

**Decimal 33 = 0x21 = ‘!’ (ASCII). So by constructing a 7 character input string with 6 arbitrary characters and ending in ‘!”, we should be able to cause the bytes “0x21000000” to be written to the integer bytes on the stack which will interpret those bytes as the least significant bytes of a 4 byte int 0x21000000 = 33 (in decimal).**

**This screenshot shows that we input an exclamation point (!) just after six ‘B’s (BBBBBB!) This overflowed the buffer and then put the decimal number 33 into the value of x on the stack.**



**Exploitation:**

3. Using techniques you’ve seen, you will attempt to learn the contents of a flag file on a computer owned by Mullins Movies, Music, and Machines Inc. (M4I). Your reconnaissance indicates this computer is on the CDN network. The following questions will help guide your exploitation. Besides describing how you completed a task, provide the exact commands you used as well as screenshots of your results for each step. Be sure to only attack your target machine; verify by checking the computer name. You may not use Armitage for this question. Remember your target machine name ends with your team number (e.g., Target3). The following question will help guide your quest…

a. How did you learn the target computer name?

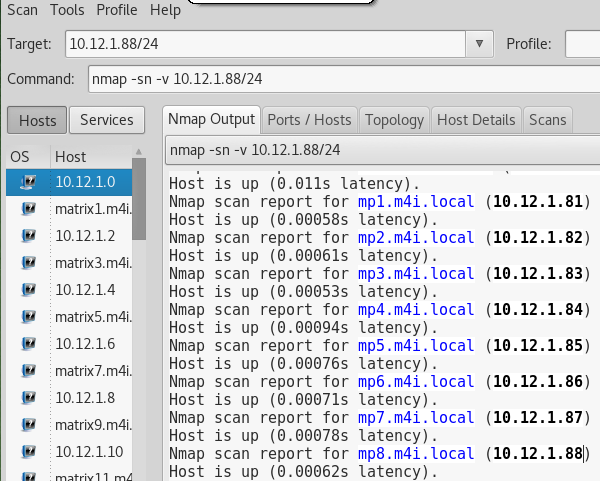
**Following the hyperlink on the Twitter bird to the right of the question in the lab instructions. We navigated to “https://twitter.com/DrEvilM4I”**



**At this site Dr. Evil had posted “10.12.1.88” as an interesting IP for Lab 3 Question 3 :)**

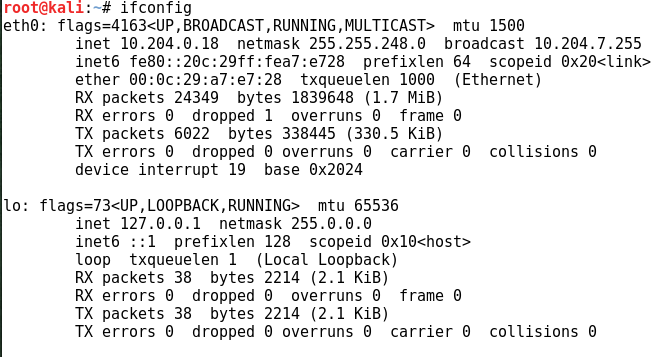


**We found our target’s IP and hostname by scanning the subnet of IP 10.12.1.88 (/24). The scan we performed in Zenmap showed the hosts that were “up” and provided their hostnames (if applicable). Given that the “interesting IP” is associated with “mp8.m4i.local” then we assume our target is mp4.m4i.local, which has an IP address of 10.12.1.84.**



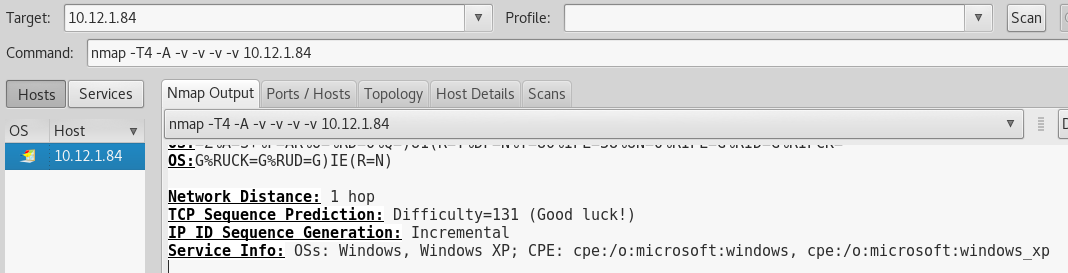
b. How did you identify the IP addresses of your target and the attacker (you)?

**I found my IP my running “ipconfig” from the command line. I am attacking from 10.204.0.18.**



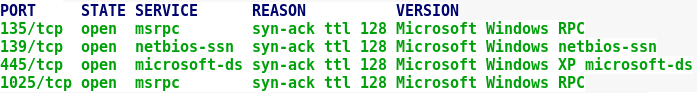
c. What is the target operating system?

**Running a more comprehensive nmap scan on the target IP shows a guess of the operating system as “Windows XP” (perfect…)**

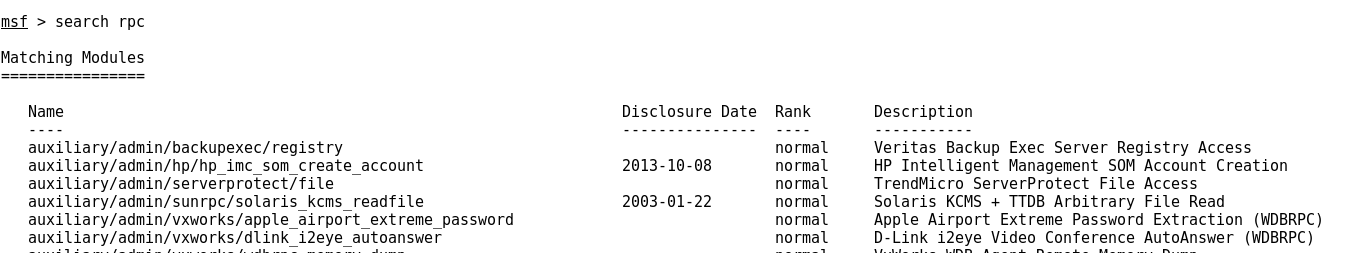


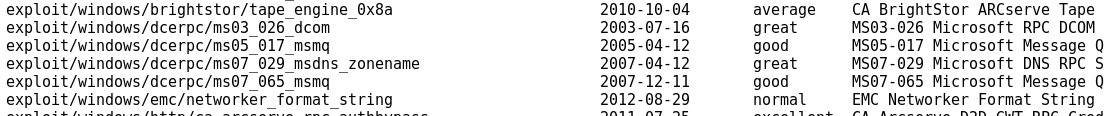
d. How did you exploit the machine? Describe exactly how you exploited the machine. What vulnerability did you use? For example, the exploit uses a buffer overflow in application XYZ.

**The scan above showed several open ports that could be attacked. We started with RPC on port 135.**

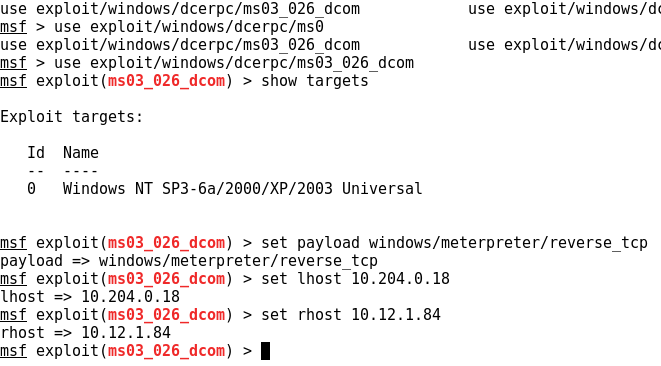


We used the “msfconsole” command to open Metasploit and looked for a vulnerability on one of the 4 open ports.

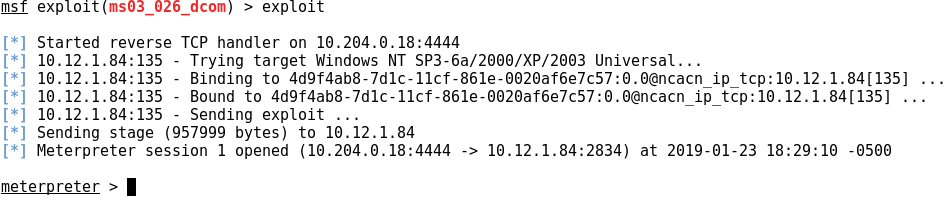




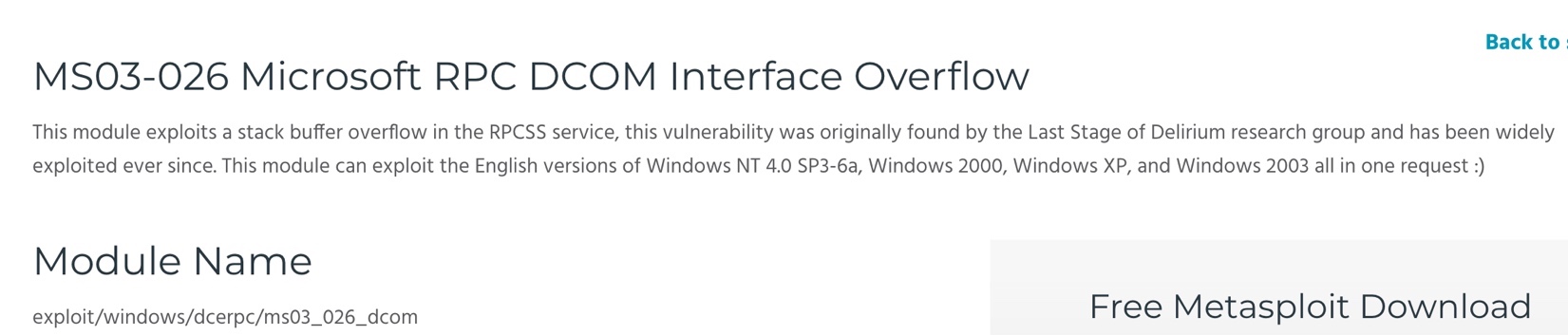
We believe ms03\_026\_dcom will work as it is a “great” exploit. We used the following commands to setup the attack. The “show targets” command revealed that we were on the right track as Windows XP was one of the targets listed.



**IT WORKED!! We executed the exploit command and it returned a Meterpreter shell. The DCOM exploit**



According to Rapid7, this exploit performs a stack buffer overflow in the RPCSS service. The RPCSS service is the Service Control Manager for COM and DCOM servers. We accessed this site to find the information: <https://www.rapid7.com/db/modules/exploit/windows/dcerpc/ms03_026_dcom>

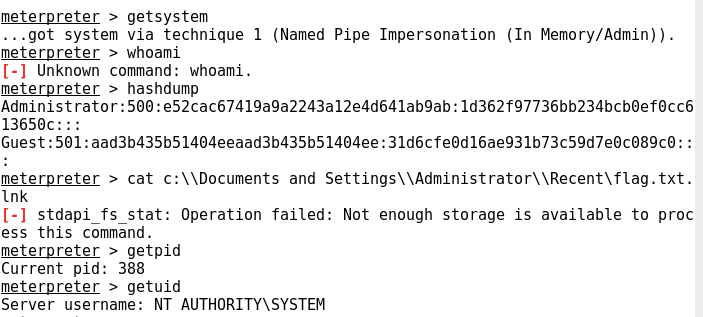


e. How did you establish a command shell from the target back to the attacker?

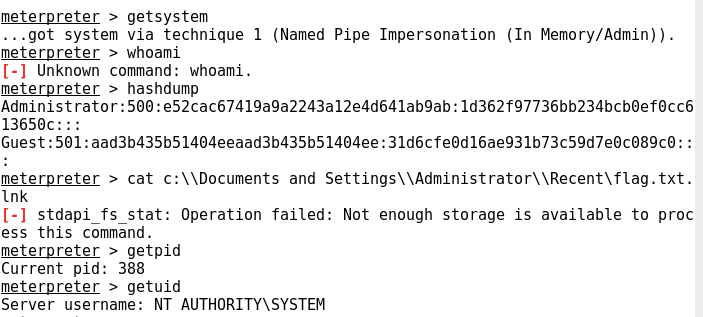
Used reverse TCP shell with Meterpreter to connect from an OS assigned port back to our machine. It did this because we specified that it should use a reverse shell. This technique makes the attack more likely to succeed if there is a firewall blocking incoming connections.

f. How did you search a computer for a specific file name?

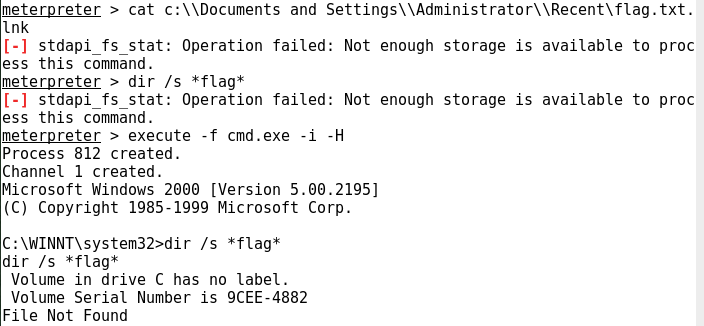
**First, we used the “getsystem” command to gain System privileges:**



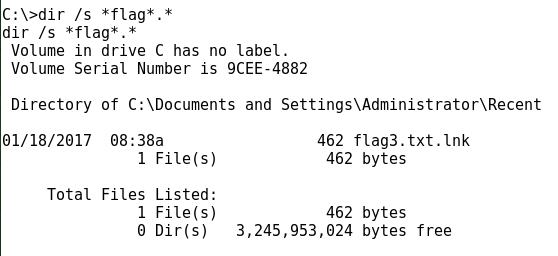
**Next, we checked our current level of access. The “getuid” command showed us that we have System level (full) access to the computer.**



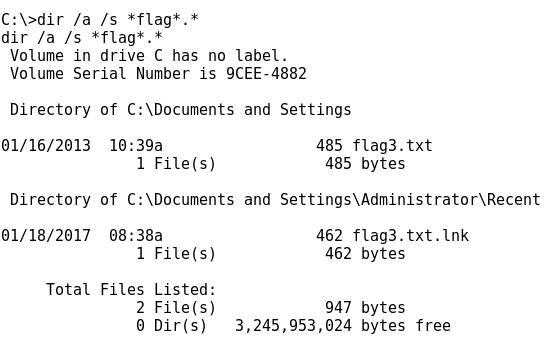
We then tried to use the “dir /s \*flag\*” command to find the flag file. However, this did not work because there was “not enough storage available to process the command.” We then used a Meterpreter command to open a command prompt on the machine. Using the same “dir” command produced no results.



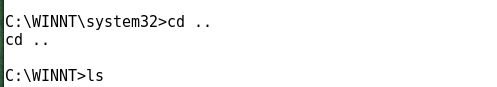
**We then added another “.\*” to account for other file extensions and found a link to a “flag3.txt” file in the Administrator’s recent items. This clued us in that there might be hidden files on the computer.**

****

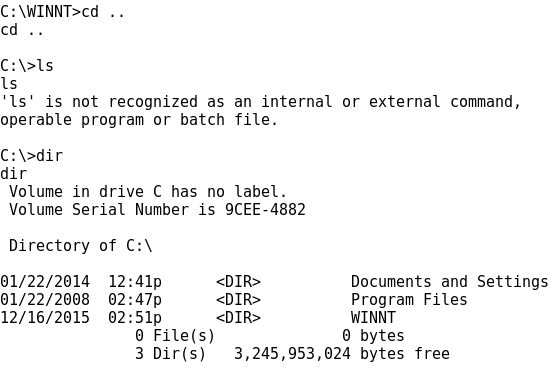
**Once we added the “/a” to look all files (including hidden files). We found the files in the “C:\Documents and Settings” folder.**



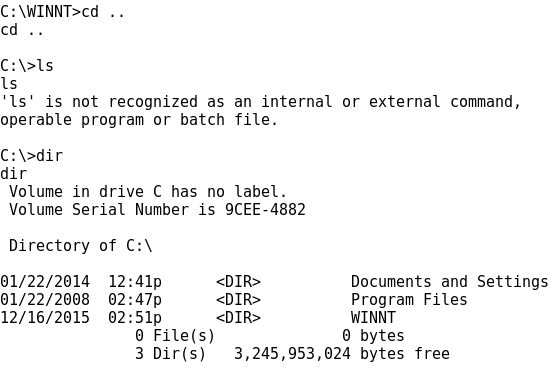
**We traversed to the folder listed above using a series of “cd” commands:**



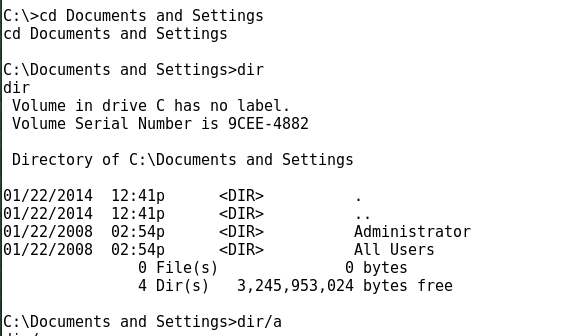








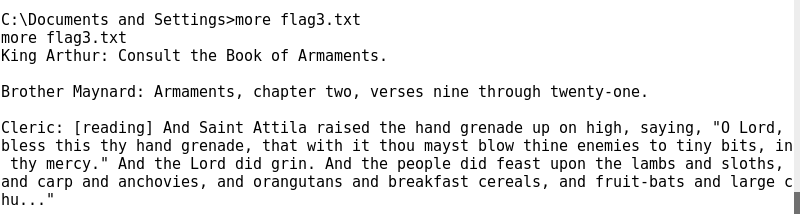






g. How did you display the contents of a file?

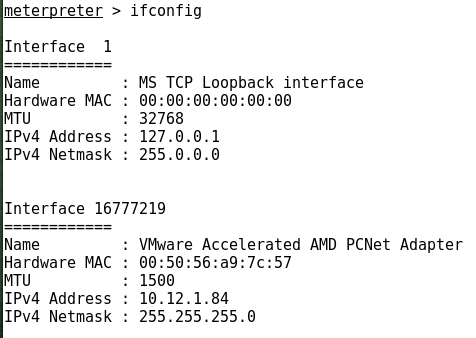
**Once we were in the correct folder, we used the “more” command to display the file to the screen.**



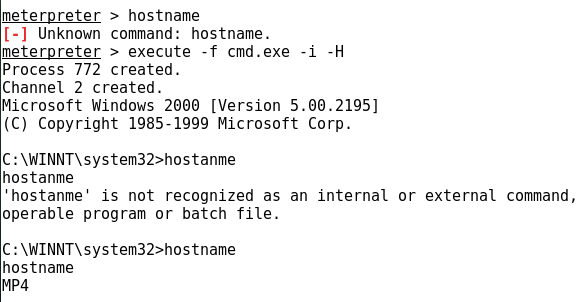


h. Display the hostname and IP address within your Metasploit console.

**We used the “ifconfig” command to show the IP address (10.12.1.84)**



**We could not find a hostname command in Meterpreter, so we opened the command prompt again and used the “hostname” command. The hostname is “MP4”**

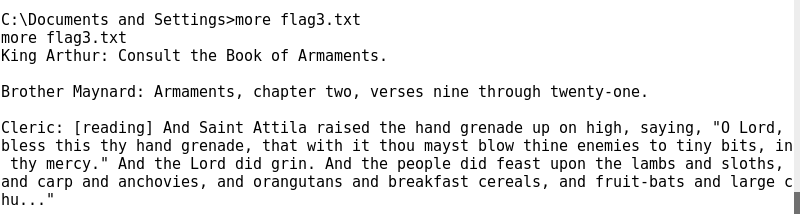


i. If you have made it this far, you can now answer the burning question… what is the message in the file?

**King Arthur: Consult the Book of Armaments.**

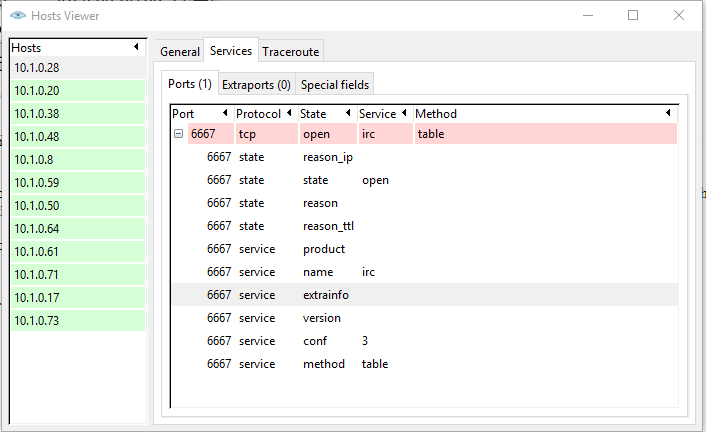
**Brother Maynard: Armaments, chapter two, verses nine through twenty-one.**

**Cleric: [reading] And Saint Attila raised the hand grenade up on high, saying, “O Lord, bless this thy hand grenade, that with it thou mayst blow thine enemies to tiny bits, in thy mercy.” And the Lord did grin. And the people did feast upon the lambs and sloths, and carp and anchovies, and orangutans and breakfast cereals, and fruit-bats and large chu…”**

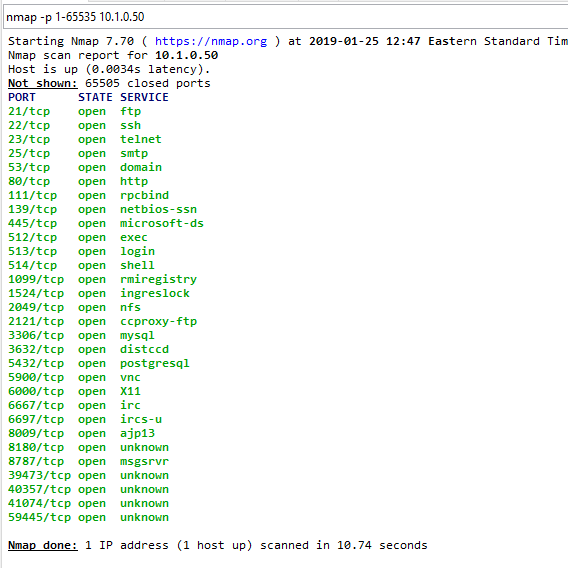


4. Your target is a machine on the CDN that is listening on port 6667. What are the contents of the two flag files? What is the operating system? Verify you are attacking your team’s computer. If you exploit the machine, describe how the vulnerability is exploited. For example, the exploit uses a buffer overflow in application XYZ. You must demonstrate how to gain access to the target in two ways: one may be with Metasploit/Armitage but the second may not use Metasploit/Armitage. You may not use the same exploit for the second access.

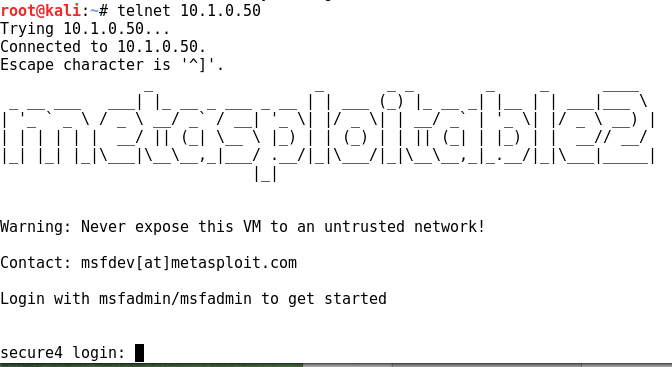
**Using the information given on the board in the GECO lab we used NMAP to scan 10.1.0.0/21 for a PC with port 6667 open. We used this command: “nmap -p 6667 -T4 -v 10.1.0.0/21” and then filtered the results in Zenmap to just show those machines with port 6667 open. All the target machines are listed on the left.**



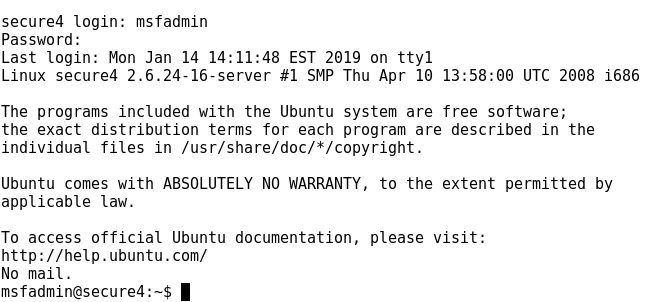
**We used this command to rescan those specific IPs to reveal all the open ports and protocols (*nmap -sS -sU -T4 -A -v -PE -PP -PS80,443 -PA3389 -PU40125 -PY -g 53 --script "default or (discovery and safe)" 10.1.0.0,8,17,20,28,38,48,50,59,61,64,71,73).* The screenshot below shows the scan of just one of the computers.**

****s

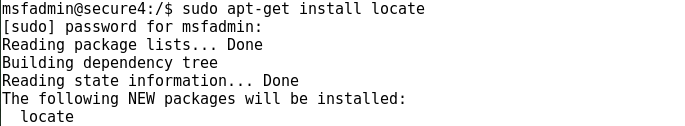
**We noticed telnet was open on port 23 for all machines with port 6667 open (highlighted above). We tried to login via telnet and saw that it listed “secure#” by the login (with # being a number). We assumed that this refered to the team number. Searching through different IPs (using telnet login prompts) we finally found that 10.1.0.50 listed “secure4” by the login prompt.**

****

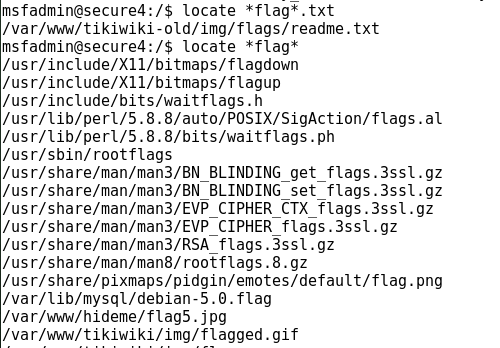
**Using the login information provided on the telnet login screen we access the computer (Username: msfadmin, Password: msfadmin)**

****

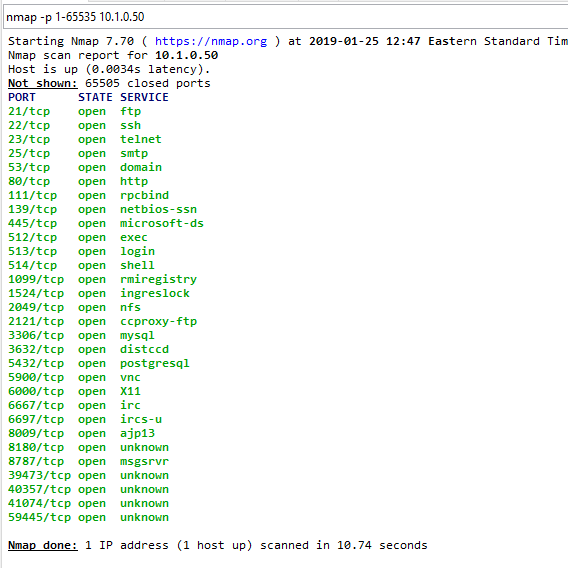
**We used the command “sudo apt-get install locate” in order to use the locate command to find the files:**

****

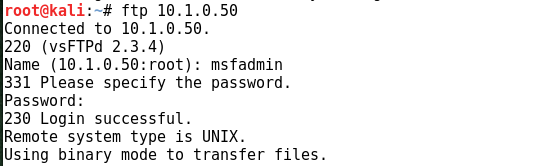
**The “locate \*flag\*” command showed many different files and folders with “flag” in the name. The only promising files was “/var/www/hideme/flag5.txt”**

****

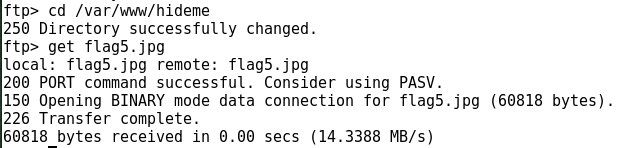
**Because we didn’t know any commands to download or transfer a file within telnet, and because port 21 was opened on the target we tried to connect to ftp on port 21.**

****

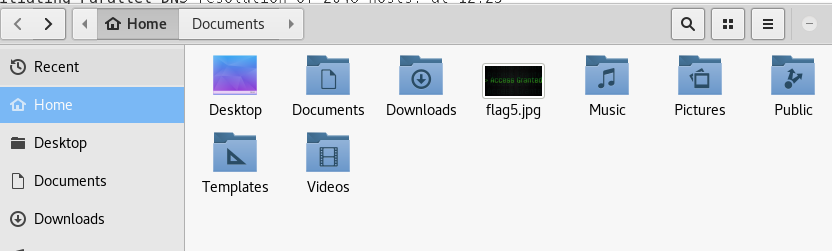
**We attempted to login using the ftp 10.1.0.50 command in a terminal window from our attacking machine. We used the same credentials as telnet (msfadmin / msfadmin).**

****

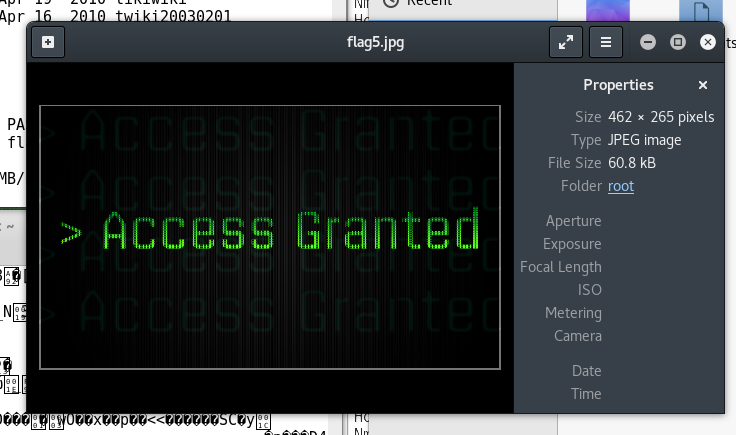
**Once we gained access, we navigated to the correct folder and used the “get” command to transfer the file to our attacking machines home folder.**

****

**The picture file “flag5.jpg” appeared:**

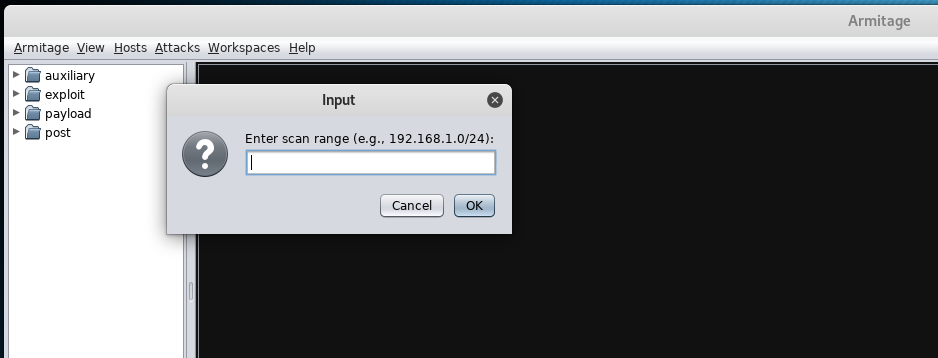
****

**The message in the picture is “Access Granted”**

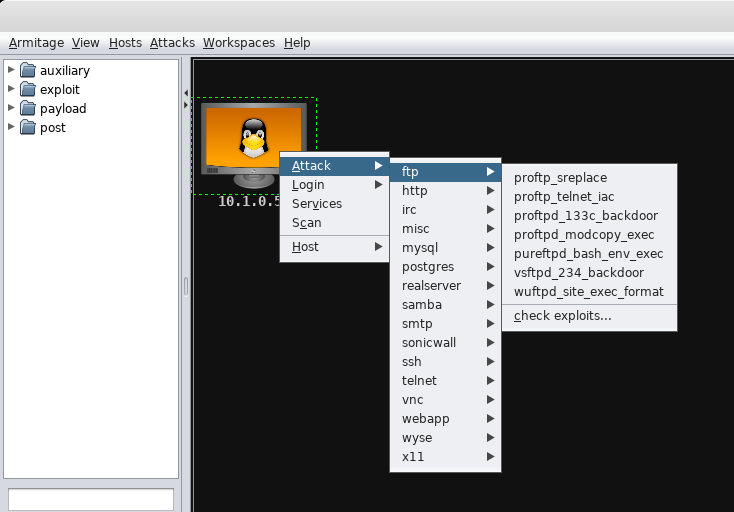
****

**For the second method of access, we used Armitage so that we could launch a command shell on the target and (hopefully) gain root access to search for the other hidden file.**

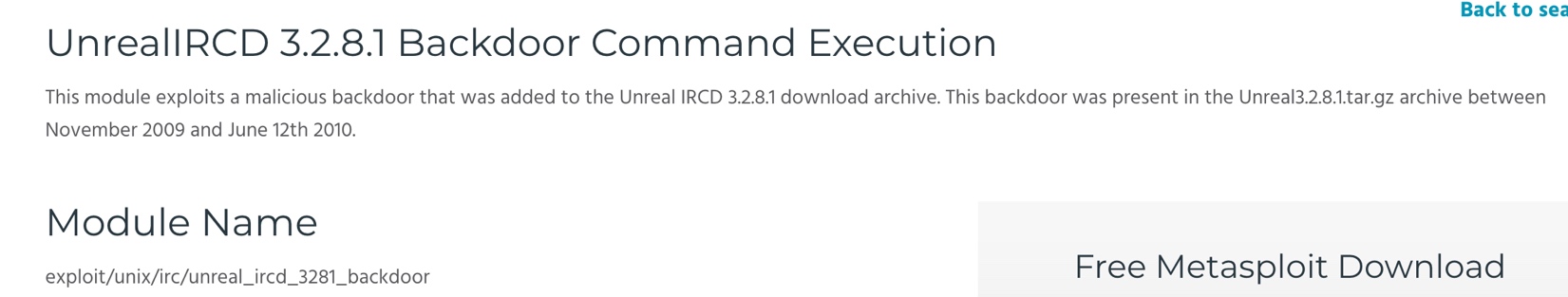
**We opened Armitage and selected “Hosts” and “Scan” and input “10.1.0.50” as the scan range.**

****

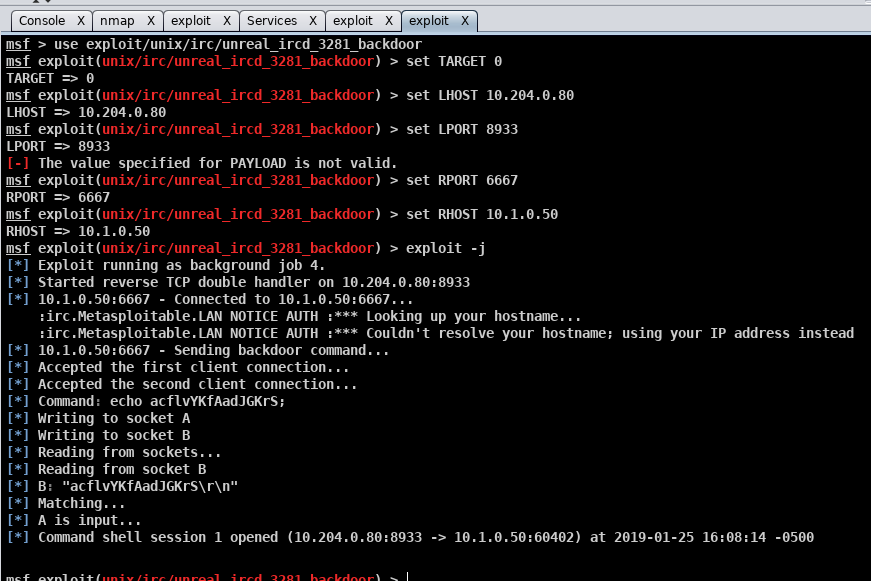
**This revealed the Linux computer below. We then used the “Attacks” menu to analyze all the possible exploits that are in the realm of the possible against the target. You can see from the screenshot that there are a lot of things to attack.**

****

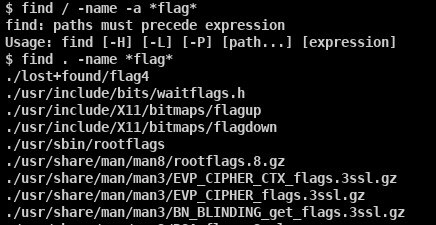
**We chose to exploit the “irc” service from the dropdown list. We used the “unix/irc/unreal\_ircd\_3281\_backdoor” exploit which exploits a malicious backdoor that was in one of the Unreal IRCD. The following snapshot is taken from the Rapid7 database (**[**https://www.rapid7.com/db/modules/exploit/unix/irc/unreal\_ircd\_3281\_backdoor**](https://www.rapid7.com/db/modules/exploit/unix/irc/unreal_ircd_3281_backdoor)**):**

****

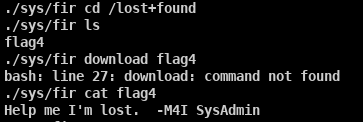
**Armitage filled in many of the commands below and then attempted the exploit which was successful!**

****

**Using the command shell that was opened in Armitage we searched for the flag file again using “find / -name -a \*flag\*” The ‘-name’ specifies that we are searching for a specific folder/file name and the ‘-a’ says we want to display all files (even hidden ones). The command revealed the flag4 file in the “./lost+found” folder.**

****

**We were able to access this directory and printed the contents of the flag4 file to the command prompt using the “cat” command.** **(Note: it looks like our command shell was acting strange because it was printing “./sys/fir” instead of the “$” when we first exploited the computer)**

****

**The message of the file is “Help me I’m lost. -M4I SysAdmin”**

**General Observations**

How long did it take you to complete this lab? **5 Hours**

Was it an appropriate length lab? **Yes. We understand buffer overflows much better now.**

What corrections and or improvements do you suggest for this lab? Please be very specific, and if you add new material, provide the exact wording and instructions you would give to future students in the new lab handout. You may cross out and edit the text of the lab on previous pages to make minor corrections/suggestions.

**None.**