**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.

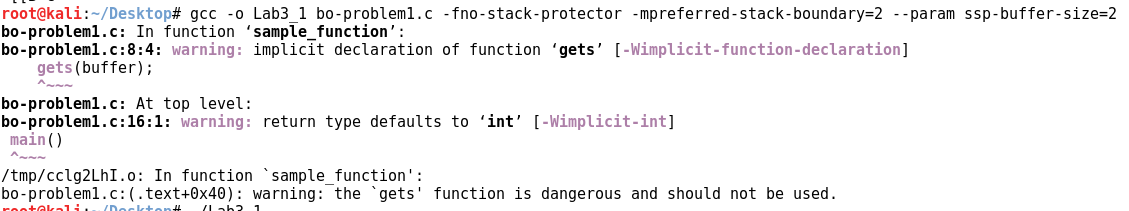
1. Assuming no canary, draw a diagram (memory map) showing the contents of the stack just before the **gets** instruction is executed. This diagram should look very similar to those on the course slides.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Memory Bank | |  |  |  |
|  |  |  |  |  |  |
|  |  | 03 | 03 | Buffer |  |
| 03 | 03 | 03 | 03 | Buffer |  |
| 03 | 03 | 03 | 03 | Buffer |  |
| f8 | 04 | bd | bf | EBP |  |
| 1e | 85 | 04 | 08 | return pointer | |

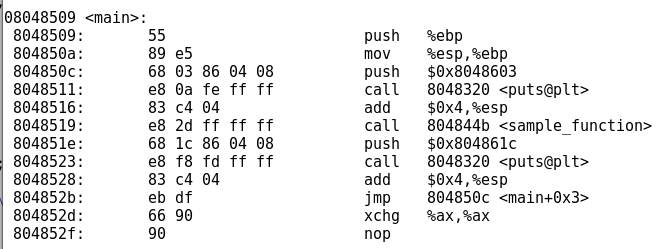
1. Predict the minimum number of characters the user must enter in order to crash the program. Remember, there is always a null character (\x00) appended to the end of strings. In other words, when you enter six B’s, seven bytes are actually stored in memory: BBBBBB\x00.

* 14, because the return pointer starts 15 bytes from the beginning of the buffer. Thus, we need a total of 14 characters input, plus the null character byte to reach the start (in memory) of return pointer.

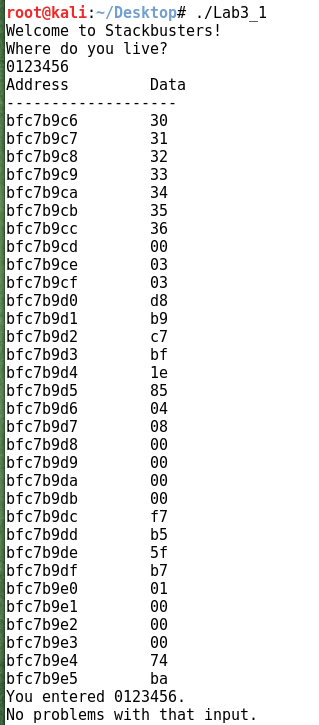
c. Compile the code in Linux using the **–fno-stack-protector** option to disable the canary (stack protection). Depending on the operating system (and version) you are using, this switch may not work and you’ll have to account for the canary in your answer. Also use **–mpreferred-stack-boundary=2**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



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 0x804851e, the address of the instruction immediately following the sample\_function call.

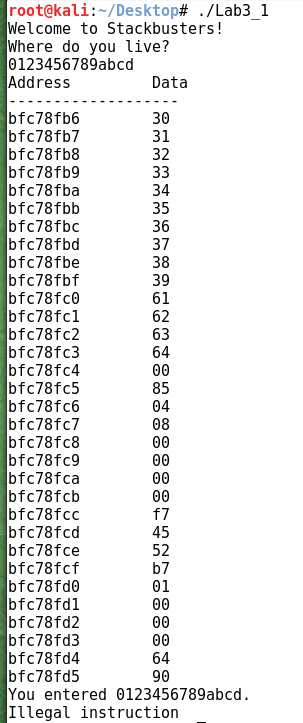


e. Execute your code and enter “**0123456**”. On your screenshot of the results, circle each field listed in part (a) above (e.g., buffer, canary if it exists, SFP, RP). Do you see the return pointer in the correct location?

The data/buffer is shown outlined in red. 0123456 is hex 30-36, followed by the null byte 00. The next 4 bytes are EDP, and are outlined in green. The return pointer 0x0804851e is then shown circled in yellow. The return pointer is where it was expected to be.

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 program will crash for an input of any length greater than or equal to 14. The yellow outline shows the null byte where the return address is, as well as the response from the program informing us that it was an illegal instruction.



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.**

* It takes 14 or more characters because at that length, the return pointer is overwritten by data from the user. This causes the program to not know where to return to after finishing sample\_function. In this example, because we entered 14 bytes, the first byte in memory of the return pointer 0x1e, is overwritten by the null character 0x00. This is shown on the screenshot of the page above, outlined in yellow.

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.

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 \_= BFF1EDC8\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**user\_input = BFF1EDCE­ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**x = BFF1EDD4 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

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.

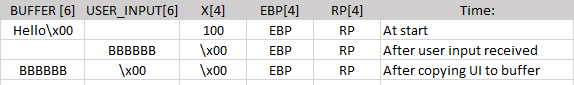
|  |  |
| --- | --- |
| Memory Location (one byte) | Variable stored at that location |
| BFF1EDC8 | Buffer[0] |
| C9 | Buffer[1] |
| CA | Buffer[2] |
| CB | Buffer[3] |
| CC | Buffer[4] |
| CD | Buffer[5] |
| CE | User Input[0] |
| CF | User\_Input[1] |
| D0 | User\_Input[2] |
| D1 | User\_Input[3] |
| D2 | User\_Input[4] |
| D3 | User\_Input[5] |
| D4 | X[0] |
| D5 | X[1] |
| D6 | X[2] |
| D7 | X[3] |
| D8 | EBP[0] |
| D9 | EBP[1] |
| DA | EBP[2] |
| DB | EBP[3] |
| DC | RP[0] |
| DD | RP[1] |
| DE | RP[2] |
| DF | RP[3] |

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.

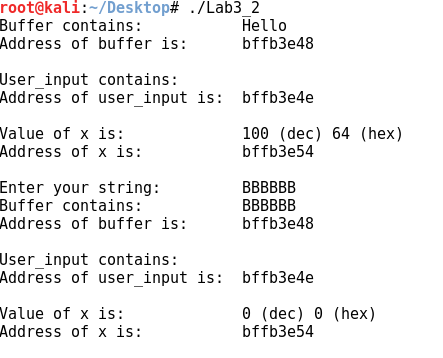
|  |  |  |  |
| --- | --- | --- | --- |
| **Input  (# of B’s)** | **buffer** | **user\_input** | **x (in hex)** |
| 4 | BBBB | BBBB | 64 |
| 5 | BBBBB | BBBBB | 64 |
| 6 | BBBBBB |  | 00 |
| 7 | BBBBBBB | B | 42 |
| 8 | BBBBBBBB | BB | 4242 |
| 9 | BBBBBBBBB | BBB | 424242 |
| 10 | BBBBBBBBBB | BBBB | 42424242 |
| 11 | BBBBBBBBBBB | BBBBB | 42424242 |
| 12 | BBBBBBBBBBBB | BBBBBB | 42424200 |
| 13 | BBBBBBBBBBBBB | BBBBBBB | 42420042 |
| 14 | BBBBBBBBBBBBBB | BBBBBBBB | 42004242 |

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.**

The behavior of the variables is a result of how the program receives data. It receives the user input using gets(). This function has no limits to the data length, so it writes the user’s input into memory starting at the address of User\_Input[0], until it reaches a null byte.

The program then copies the data from User\_Input into Buffer using strcopy(), which copies everything from the start of User\_Input until it reaches a null byte. The below chart shows the memory contents of the Buffer, User\_Input, X, EBP, and RP with memory addresses increasing from left to right.

As shown above, the user entered “BBBBBB”, which put 6 Bs into the User\_Input, and placed its null byte at the start of variable X. When the program copied User\_Input into Buffer, it copied “BBBBBB\0”, which overwrote the first byte of user input with a null byte. Thus, when the program output the results, it showed “Buffer = BBBBBB”, “User Input = “, and “Value of X = 0“.



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

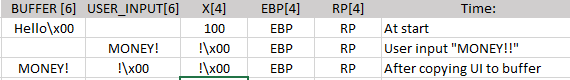
In the below table, memory addresses increase from left to right.

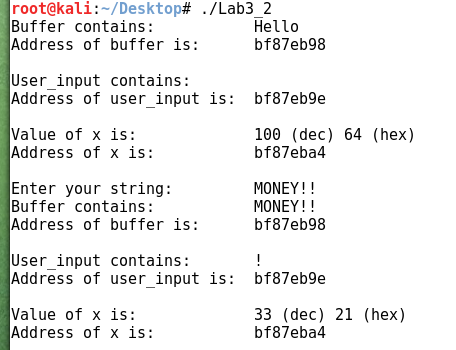


As discussed in part e, the methods used by the program to receive data place no limits on the length of input. Thus, when the user input 14 Bs, it actually placed 14 B’s followed by a null byte, starting at the address of User\_Input[0]. This caused the null byte to overwrite the first byte of main’s return address, causing the seg fault. The above screenshot shows how this appears in memory, with the user’s input outlined in red.

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.

To accomplish this, we simply need to overwrite the X variable with the value of 33. By inputting “MONEY!!” as the user input, this overwrites the value of X with “!\x00”. The ascii code for “!” is 33; thus, decimal 33 is stored in X. This process is shown in the table below, and the output is shown below outlined in red.



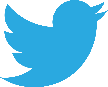


**Exploitation**

**You may not use Armitage for question 3. You may use Armitage for question 4.**

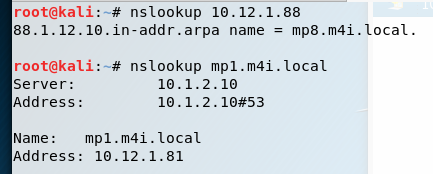
During this course (including the remaining labs and final project), you may never change permissions on files or folders. This includes sharing a drive/folder such that anyone can gain access. Also, you may never start services that provide access to the target such as FTP, HTTP, RDP, etc. This includes creating a service that is password protected.

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…

[](https://twitter.com/DrEvilM4I)

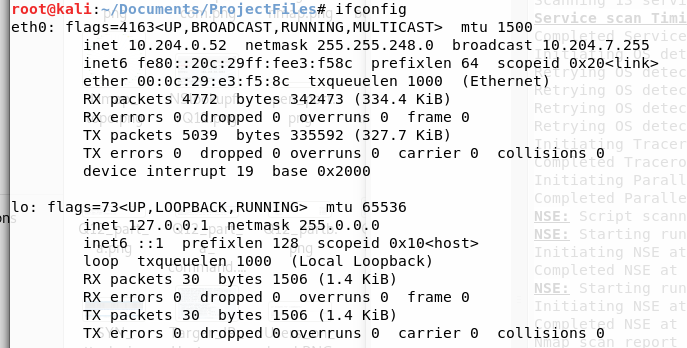
a. How did you learn the target computer name?

* We clicked the Twitter link to the right, taking us to a tweet informing us that “10.12.1.88” was an interesting IP for Lab 3.
* Ran nslookup on that IP address, learning that its alias was mp8.m4i.local.
* Ran nslookup again looking for mp1.m4i.local: giving us the IP address 10.12.1.81 verifying that is the proper target computer name.



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

* Target IP: 10.12.1.81
  + Learned target IP address using “nslookup mp1.m4i.local” (shown above in red)
* Attacker IP: 10.204.0.52
  + Learned by running ifconfig on our machine shown below.



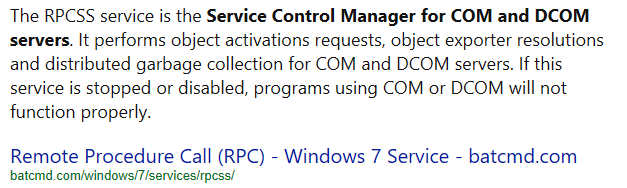
c. What is the target operating system?

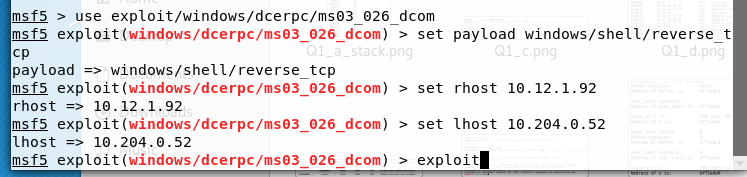
* Target OS is Microsoft Windows 2000 – on the picture above Dr. Mullins’ door.

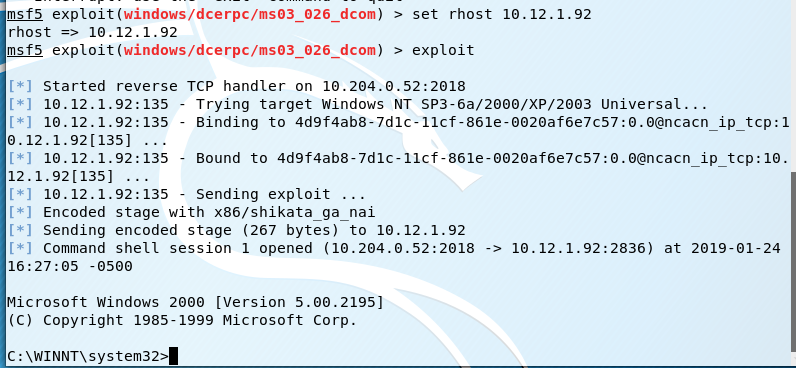
d. How did you exploit the machine? Describe exactly how your exploited the machine. What vulnerability did you use? For example, the exploit uses a buffer overflow in application XYZ.

* Searched the rapid7.com database for exploits on Windows 2000 machines.



* Selected exploit, as shown above, works for Windows 2000.
* Used Metasploit console to use the exploit, with a payload of windows/shell/reverse\_tcp.
* This vulnerability uses a buffer overflow in the RPCSS service, a service which performs “object activations requests, object exporter resolutions, and distributed garbage collection for COM and DCOM servers”
* Execute the exploit:

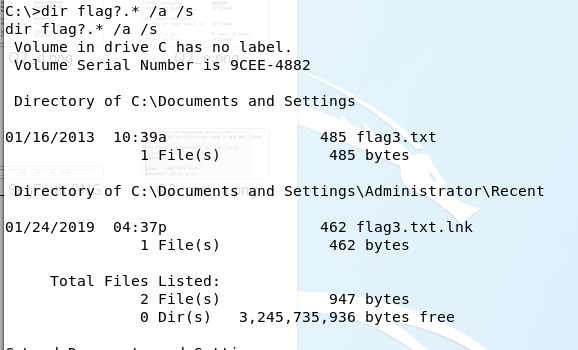


e. How did you establish a command shell from the target back to the attacker?

* We established a command shell from the target back to our attacking machine by using a reverse\_tcp payload, shown in Part D outlined in red.

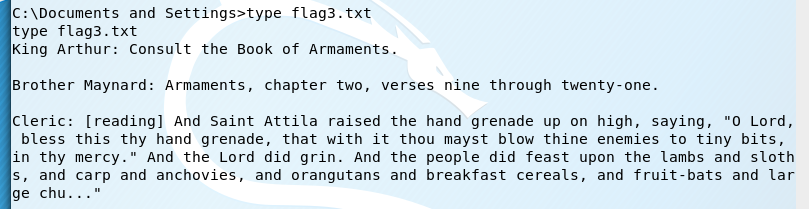
f. How did you search a computer for a specific file name?

* From the C: directory, we used the command “dir flag?.\* /a /s”, which searches all files, including hidden files, for flag, a number, and any file extension.



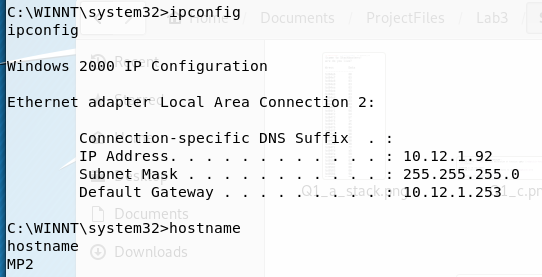
g. How did you display the contents of a file?

* Used the command “type flag3.txt”

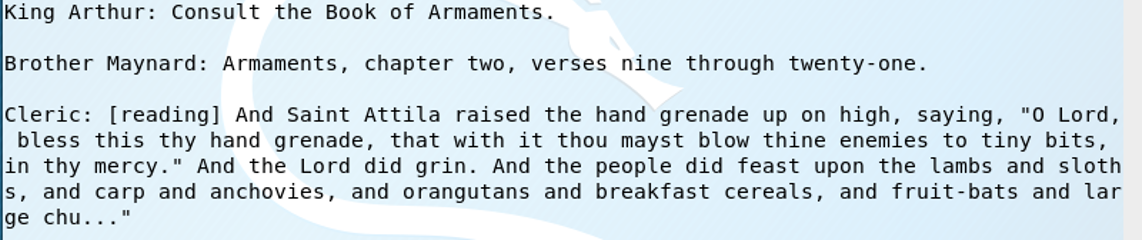


h. Display the hostname and IP address within your metasploit console.

* Hostname: MP2
* IP Address: 10.12.1.92
* Both are shown below outlined in red:



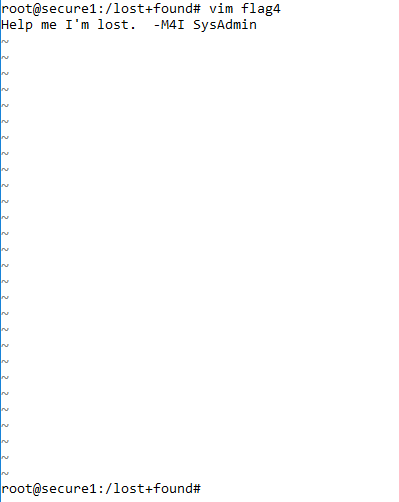
i. If you have made it this far, you can now answer the burning question… what is the message in the file?

* The message in the file is shown below:

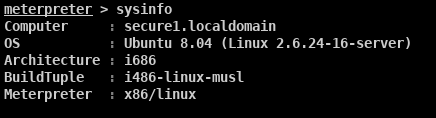
4. Your target is a machine on the CDN that is listening on port 6667. What are the contents of the two flag files? What is the operating system? Verify you are attacking your team’s computer. If you exploit the machine, describe how the vulnerability is exploited. For example, the exploit uses a buffer 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.

**Solution:**

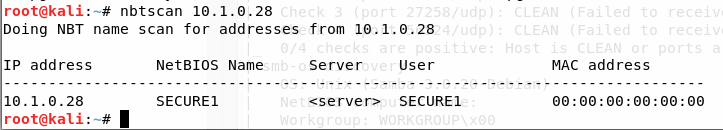
* Contents of Flag4.txt:



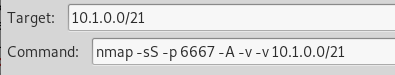
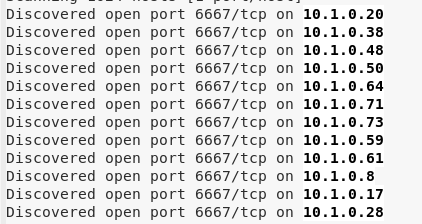
* Contents of Flag5:
* Operating System: Ubuntu 8.04



* Verification that 10.1.0.28 is our team’s computer:

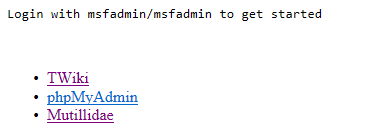


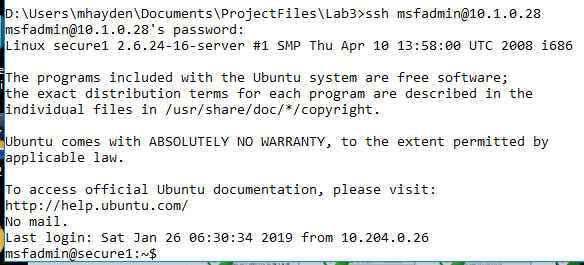
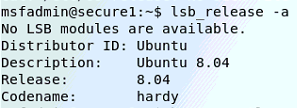
**Steps for finding target computer:**

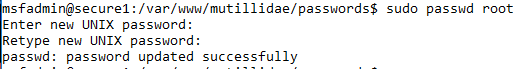
* Used nmap to scan the IP addresses on the subnet 10.1.0.0/21, looking for an open port 6667.
* Found the following list of devices with open port 6667:
* Used nbtscan IP for each IP address listed to find server names.
* Output for 10.1.0.28 shown above with associated name “SECURE1”

**Exploit 1: Recon + SSH**

* Went to the target IP in a web browser to do reconnaissance, where we found an initial username and password.

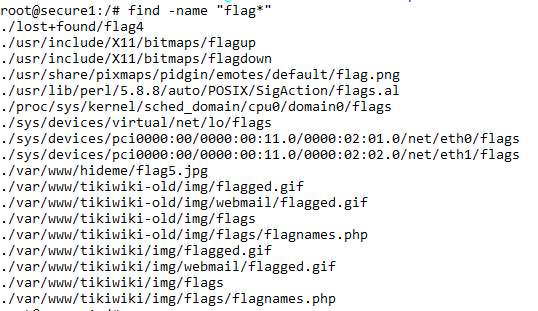


* Used ssh to login to the target IP using username – msfadmin, and password –msfadmin
* Learned operating system using “lsb\_release -a”
* Attempted to access locked directories, but did not have root access. Thus, we switched the root password to “password” from the msfadmin ssh terminal:

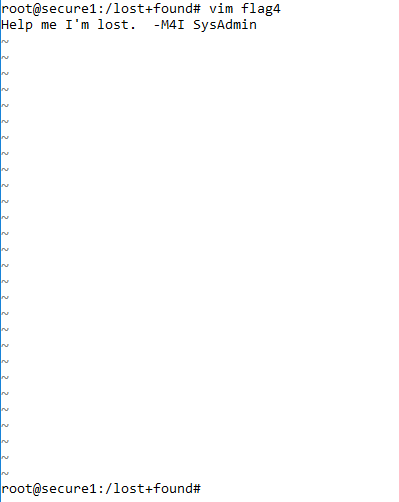


* Pivoted from msfadmin to root using “su –“ and entering the password “password”:

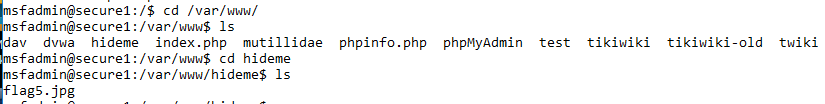


* Searched for the flag files from the root home directory:
* Navigated to the flag files using the paths from the search:

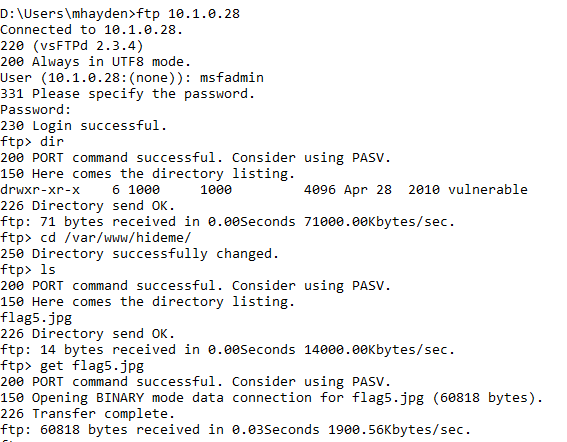
**Flag 4:** contents found using vim in the terminal:



**Flag 5:**

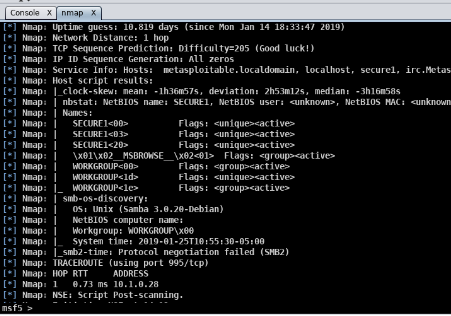
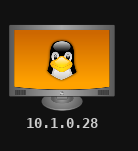
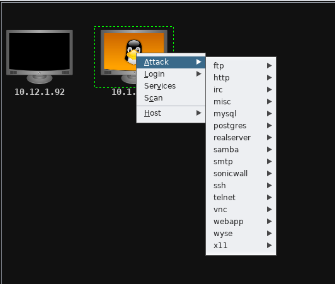
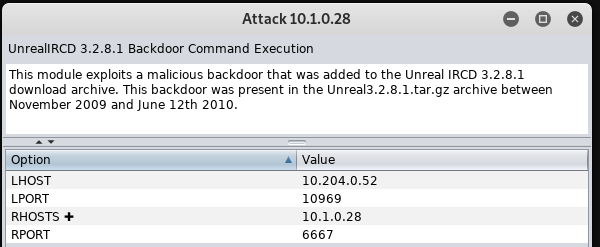


* Used an ftp session from my host computer to retrieve the file:

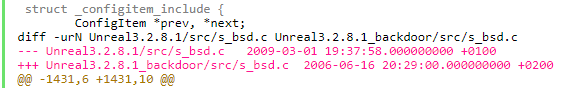


**Exploit 2: Armitage**

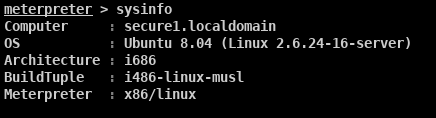
**Steps:**

* Performed nmap scan of target 10.1.0.28 in Armitage console; after which the target appeared in Armitage as shown below.
* Selected attack from the Attack list associated with the target machine:
* Exploit selected:

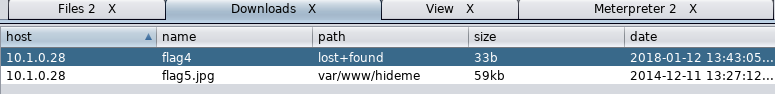
This exploit utilizea a backdoor in Unreal IRCD, which is a common IRC server software. As described in <https://blog.stalkr.net/2010/06/unrealircd-3281-backdoored.html>, “The backdoor is very simple: basically a few lines with defines and macros in order to have a **system()** on socket data if this data begins with special word **AB**. As shown below, the backdoor code switches which **s\_bsd.c** file is used. Also, as shown above, the exploit targets port 6667, the same port which was scanned/verified open in the reconnaissance/scanning phase.



* Once the machine was exploited, we opened a shell, and used the “shell to meterpreter” to gain a meterpreter shell from the existing command shell. This used the command **“use post/multi/manage/shell\_to\_meterpreter”**.
* Once we had a meterpreter shell, we checked the operating system:



* We then opened a “Browse Files” window, navigated to the appropriate directories for the “Flag” files, and downloaded them. Below is the downloads window in Armitage after the files were downloaded:



**General Observations**

How long did it take you to complete this lab?

* This lab took around 7-8 hours to complete. We spent 2-3 hours working on the buffer overflow portion, learning how the buffer overflow functioned and fully understanding what was happening. We spent 4-5 hours on Questions 3 and 4 due to unfamiliarity with the tools we were using and learning how to apply them.

Was it an appropriate length lab?

* Yes, it would have been faster had we had prior experience with the tools, but working through the challenges gave us a better understanding.

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 for this lab.