UNIVERSITÉ
LAVAL

FACULTÉ DES SCIENCES ET DE GÉNIE
Département d'informatique et de génie logiciel

Pavillon Adrien-Pouliot, local 3908
1065, avenue de la Médecine
Université Laval
Québec, Canada, G1V A06

BUFFER OVERFLOW VULNERABILITIES IN C AND C++

PAR

PATRICE LACROIX
ET
JULES DESHARNAIS

RAPPORT DE RECHERCHE
DIUL-RR-0803

DÉPARTEMENT D'INFORMATIQUE ET DE GÉNIE LOGICIEL
FACULTÉ DES SCIENCES ET DE GÉNIE

Pavillon Adrien-Pouliot
1065, avenue de la Médecine
Université Laval
Québec, QC, Canada
G1V 0A6

AOÛT 2008

Buffer Overflow Vulnerabilities in C and C++*

PATRICE LACROIX and JULES DESHARNAIS
Université Laval

August 7, 2008

Abstract

Buffer overflows are bugs that are often present in programs and often exploited by pirates. They have been known for a long time, but are still the source of major security problems today. A buffer overflow happens when some data are read from or written to a location that is not allocated for them. Usually, there are already other data in memory next to the area where an overflow happens, and these data are modified by the overflow. A pirate can thus cause an overflow and modify these data at will, and thus he can influence the rest of the execution of the program.

This survey describes the causes and consequences of this vulnerability. It explains the techniques used by pirates to exploit programs containing this kind of vulnerability. It also presents approaches and programs that are useful to avoid buffer overflows or limit their consequences.

1 Introduction

1.1 Motivation

Today, computers are everywhere and, more important, they have the ability to communicate with each other. They may be connected to a local network or to the Internet, and it is now rare that a computer is completely isolated. This is true for personal as well as enterprise computers.

With this increasing connectivity, interacting with a computer becomes easier and easier and many people use the network to gain illegal access to private data. There can be security problems when software is not conceived with security in mind, or if it is not implemented properly. Pirates often exploit these problems to gain access to information or to a computer they should not have access to. This survey studies one kind of vulnerability that is often exploited by pirates, namely, buffer overflows.

*Author's address: LSFM Research Group, Département d'informatique et de génie logiciel, Université Laval, Québec, QC G1K 7P4 Canada
e-mail: patrice.lacroix@ift.ulaval.ca; jules.desharnais@ift.ulaval.ca
This research was funded by Natural Sciences and Engineering Research Council of Canada (NSERC) and Defence Research and Development Canada (DRDC).

Since a few years ago, buffer overflows pose major security problems. They are the cause of a large percentage of recent vulnerabilities. Depending on the source of information and the dates, about 50% of reported vulnerabilities are caused by buffer overflows [CWP+00, PL04, Wag00]. In order to write secure programs, it is important to know more about buffer overflow vulnerabilities.

1.2 Scope

The literature concerning buffer overflows is plentiful. We could not possibly describe every variation of attacks and protection techniques and every tool that can help detecting buffer overflows. However, we want to present, for every approach, enough information to understand the ideas behind it, their advantages and their limitations.

We examine buffer overflow vulnerabilities mostly for source code written in C or C++, but we also look briefly at Java, a language not vulnerable to this problem in the traditional way that C and C++ are. Unless stated otherwise, demonstration programs were built with GCC 2.95.4 under Linux.

1.3 Buffer Overflows

According to [Wor03], a buffer is:

> a part of RAM used for temporary storage of data that is waiting to be sent to a device; used to compensate for differences in the rate of flow of data between components of a computer system.

Usually, when we talk about buffer overflows, we refer to the action of writing past the end of a buffer. For this work, *buffer overflow* has a larger meaning and covers all memory accesses outside the bounds of an array without regard for the way it is defined, allocated or accessed. A buffer overflow can thus happen as well when reading as when writing into an array.

We are particularly interested in the cases where a buffer overflow introduces a vulnerability that can be exploited maliciously. It is however not very useful to make this distinction for the following reasons:

1. To check that a buffer overflow is exploitable, we first have to know of its existence. It is however very hard to know if the execution of a program will cause a buffer overflow.

2. Once a possibility of buffer overflow is identified, it is usually relatively easy to correct it.

3. On the other hand, to determine if a buffer overflow is exploitable or not is much more complex and the answer may vary depending on the compiler and the options given for compilation, because languages like C and C++ do not define what happens at the time of a buffer overflow.

4. Even if a buffer overflow cannot be exploited in a malicious way, it can nevertheless have unwanted effects on a program.

However that may be, we will see that many approaches to solve the problem of buffer overflows try only to protect certain cases more susceptible to be vulnerable, but without proving that there really is vulnerability.

1.4 Plan of the Survey

Section 2 examines different causes and circumstances that can lead to buffer overflow vulnerabilities. Section 3 studies consequences of these vulnerabilities, mostly those that can compromise systems security. We will explain many ways to exploit buffer overflows. In particular, we will see in Section 3.3 how an attacker can force a vulnerable program to execute arbitrary code.

Section 4 looks at different approaches to detect buffer overflows. Section 4.1 covers approaches that require modifications to the source code, Section 4.2 is about testing techniques, Section 4.3 discusses possible modifications to the compiler and application libraries, Section 4.4 considers modifications to the operating system kernel, Section 4.5 presents run-time checks that can be implemented, and Section 4.6 deals with many kinds of static analysis techniques.

Section 5 presents our conclusion. The Appendix presents some programs that can be useful to detect or avoid buffer overflows or their consequences.

2 Causes of Vulnerabilities

When we use a language like C, it is easy to write a program that contains unintentional possibilities of buffer overflow. In fact, it is quite difficult to write a C program with an absolute certainty that buffer overflows are impossible [Wag00, p. 11]. Many factors contribute to this, notably the fact that the language does not enforce type safety, that it contains standard functions very difficult to use securely and, to a certain extent, a culture of laziness among C programmers.

2.1 Type Safety

This section is based on [CPM+98, CWP+00, MV00, Rit93].

Some languages are type-safe and are consequently immune to buffer overflow vulnerabilities. The C and C++ languages are not type-safe. A type-safe language does not allow using a variable in a way that is incompatible with its real type. Consequently, such a language cannot allow access to an element outside the bounds of an array since this element does not exist, and thus there is no way it can be used in a way compatible with its real type.

Contrary to C and C++, Java is a type-safe language, despite its syntax being similar to C. In order to be type-safe, Java had to give up some constructions existing in C such as pointers and functions with a variable number of parameters. Pointers are used inside the Java virtual machine to represent references to objects, but those pointers are not visible to Java programmers

and, above all, it is impossible to manipulate them in an arbitrary way with casts or arithmetic operations.

If there are type-safe languages, one can wonder why the C language is not. The explanation can be found in the origins of the language [Rit93]. C was designed as a high-level language for implementing the UNIX operating system. There was a need for a language rivalling with assembly in terms of performance. It is thus to generate speedier programs that there is no automatic array bounds validation. The programmer has to do it himself.

Since an operating system must interact closely with the computer on which it runs, pointer arithmetic and casting between integers and pointers allow accessing hardware in a relatively simple and efficient way.

Every buffer overflow can be seen as the consequence of a type-safety violation, but the following sections show, in practice, why buffer overflows happen.

2.2 Fixed-Size Buffer

This section is based on [Ale96, Smi97].

In some cases, a programmer defines a fixed-size buffer to store data from some source. This is frequent when it is possible to know the largest size of valid inputs at a certain point in the program. It is the case, for instance, when the user is asked to enter his choice from a menu where every entry can be selected with only one character.

However, nothing prevents the user from entering an invalid choice. In this case, if the programmer is not cautious and does not validate the size of input data, he risks introducing buffer overflow possibilities in his program.

In the C language, the problem is amplified by the fact that strings are delimited by null characters instead of having a distinct variable indicating length. The programmer thus has usually two stopping conditions to monitor when looping on a string in C, one to watch for the null character and the other to check if the index is inside the bounds of the array. By negligence, he can forget one of these conditions, as is the case in the program of Table 1.

In this program, `string` represents a variable-length string entered by the user and `buffer`, is a fixed-size buffer to make an upper case copy of the string. The program does not check if the size of `buffer` is exceeded, which produces a buffer overflow since the string is too large for the buffer.

An alternative would be to allocate the right amount of memory dynamically for buffer `buffer`. It is important to note that there is nothing intrinsically bad in using a fixed-size buffer. There are legitimate reasons to handle a limited amount of data, especially when it comes from an untrusted source. However, when using a fixed-size buffer, it is important to think about checking index range.

2.3 Interfaces Not Allowing Bounds Checking

This section is based on [MV00, Smi97, VBKM00, Wag00, Whe02b].

```
#include <ctype.h>

int main()
{
        /* The following string is larger than 100 */
        unsigned char *string =
                        "Someone with malicious intentions "
                        "could arrange things in such a way "
                        "that this string gets much "
                        "larger than what the programmer"
                        "thought he had to handle.";
        static unsigned char buffer[100];
        int i;
        /* Make an upper case copy */
        for (i=0; string[i]!=0; i++)
                buffer[i] = toupper(string[i]);
        buffer[i] = 0;
        return 0;
}
```

Table 1: Buffer overflow on a string. The program does not check that the index is inside the bounds of the array in the stopping condition of the loop.

```
#include <string.h>

int main()
{
        /* The following string is larger than 100 */
        unsigned char *string =
                        "Someone with malicious intentions "
                        "could arrange things in such a way "
                        "that this string gets much "
                        "larger than what the programmer"
                        "thought he had to handle.";
        static unsigned char buffer[100];
        /* Next instruction causes a buffer overflow */
        strcpy(buffer, string);
        return 0;
}
```

Table 2: Buffer overflow with `strcpy()`. The programmer does not ensure the buffer is large enough to hold the string.

The C library contains standard functions having interfaces not allowing array bounds checking. Here is a summary of functions considered dangerous because they do not allow bounds checking or do not force it. They are grouped by the particularity that renders them dangerous. When there are less dangerous equivalent functions, they are mentioned with their differences. The behaviors described are those of the GNU C Library (glibc) version 2.2, the C library most often used under Linux.

2.3.1 strcpy() and wcscpy()

Functions `strcpy()` and `wcscpy()` take a buffer and a string as parameters. `strcpy()` handles strings of type `char` while `wcscpy()` handles strings of type `wchar_t`. When calling them, we have to be sure the buffer is at least as large as the string.

In the program of Table 2, `string` represents once again a variable-length string entered by the user. There is a buffer overflow because no check is done to ensure that the size of the buffer is large enough to hold the string. It is possible to fix the problem by allocating enough memory dynamically, for instance with `strlen()` and `malloc()`.

An alternative to the use of functions `strcpy()` and `wcscpy()` is to replace them by `strncpy()` and `wcsncpy()` respectively. These functions take one more parameter, the buffer size, and never exceed this size. However, they have two

```
char buffer[10];
char *string = "Hello world!";
strncpy(buffer, string, 10);
/* Null character to end the string */
buffer[9] = '\0';
```

Table 3: Use of `strncpy()`

drawbacks. First, there can be performance problems if the buffer size is much larger than the string to copy. This is due to the fact that after the string is copied, the unused part of the buffer is filled with null characters. Next, the string risks not being terminated by a null character in the buffer if it is larger than the buffer size. This problem is much more important from a security standpoint because in C a string usually has to be terminated by a null character and, if it is not, the program risks looking for it outside the bounds of the buffer. Data from another variable could then appear in the string read from the buffer, which could be serious if it is a password or other confidential data. When `strncpy()` is used, it is important to overwrite the last character of the buffer with a null character as in Table 3.

2.3.2 strcat() and wcscat()

Functions `strcat()` and `wcscat()` are much like `strcpy()` and `wcscpy()`, respectively. However, the string to be copied is appended to the one already in the buffer instead of overwriting it. This means the buffer must be large enough to hold the string it already contains plus the one to be copied.

They also have alternatives, which are `strncat()` and `wcsncat()`. These functions do not have the performance problem of `strncpy()` and `wcsncpy()`. They also differ from these by the fact that the null character is always included at the end of the string. Finally, the size given to `strncat()` is not the buffer size, but represents the maximal number of characters that can be copied from the string. When counting the size required for a buffer, we thus have to add the length of the string already in the buffer, the maximal number of characters to be copied which is given as a parameter, and one for the null character that will end the string.

2.3.3 sprintf() and vsprintf()

Functions `sprintf()` and `vsprintf()` are more difficult to use correctly because they take as parameter a format string that is used to format the buffer. This string can contain conversion specifications to be replaced by values of variable length. For instance, a "`%s`" is replaced by a string. Its size must thus be taken

```
char buffer[10];
sprintf(buffer, "%.9s", "Hello world!");
sprintf(buffer, "%.*s", 9, "Hello world!");
```

Table 4: "Precision" of a string with sprintf

into account when counting the size required for the buffer.

An alternative is to specify a "precision" with a string conversion specification. This way, only the first characters of the string up to the given limit are considered. It is possible to give the precision directly in the format string or as a parameter. Table 4 shows the two ways it can be done.

It is important to note that the precision in the format string is specified after the dot. If the dot is removed, the value becomes the minimal width of the field and there is no protection against overflows. When the precision specification is used to limit the length of a string, it is important to take into account the rest of the format string when counting the required size for the buffer, in particular, the null character.

A "`%d`" can also take a variable length once formatted. If there is no additional constraint, the maximal length varies with the size of an `int`, which depends on the compiler and the processor for which code is generated. It is 6 characters for 16-bit `int`, 11 characters for 32-bit `int` and 20 characters for 64-bit `int`.

Moreover, format strings can have multiple conversion specifications. This complicates the calculation of the buffer size and renders errors more likely. If a calculation error makes the buffer too small, there is a risk of buffer overflow. It is thus important to be careful when choosing the size of a buffer corresponding to a format string.

Functions `snprintf()` and `vsnprintf()` are respectively equivalent to `sprintf()` and `vsprintf()`, but they are usually considered less dangerous because they take the size of the buffer as a parameter and observe it. Contrary to `strncpy()`, these functions always reserve the space required for a null character at the end of the buffer. In order to avoid buffer overflows, one obviously has to give the real size of the buffer. Another thing that may harm the use of these functions is that they are not available on all systems. They are present under Linux, but their absence poses a problem when trying to write code that has to work on different systems.

2.3.4 scanf() and its Friends

Functions in this category are `scanf()`, `fscanf()`, `sscanf()`, `vscanf()`, `vfscanf()`, `vsscanf()`, `wscanf()`, `fwscanf()`, `swscanf()`, `vwscanf()` and `vfwscanf()`.

```
char **p;
scanf("%as", p);
```

Table 5: Dynamically allocated string with `scanf()`

They also take a format string as a parameter, but this time, the format string indicates how to interpret and convert input data to be placed in some variables. We obviously have to make sure we give parameters corresponding to conversion specifications, but we also have to be careful about the size of buffers receiving strings. Since the exact length of strings is usually not known before they are read, it is always better to give the buffer size with a string conversion specification. For instance, instead of "`%s`", it is better to use "`%20s`" if the buffer has 20 characters.

With glibc, it is also possible to use the "`a`" flag to indicate that a buffer large enough to hold the complete string must be allocated dynamically. In this case, one has to pass the address that will receive a pointer to the newly allocated string as in Table 5. It is however important to note that this flag is not specified by the C standard [Int99]. Programs using it are not portable.

2.3.5 gets()

Function `gets()` is almost impossible to use correctly. It takes only one parameter: a pointer to a buffer. It reads a line of input from `stdin` and places it in the buffer without any check for overflow. The problem is that `stdin` is usually connected to user input and it is thus impossible to know in advance the amount of data that will be received. For this reason, it is recommended to never use `gets()` and instead use another function such as `fgets()` to which the buffer size is specified.

2.3.6 realpath() and getwd()

These functions both write a path in a buffer. `realpath()` writes the canonical form of a given path and `getwd()` writes the current working directory. They are not very difficult to use correctly. The buffer given to them only has to be at least `PATH_MAX` characters large to avoid overflows.

2.3.7 Other Dangerous Functions

Functions `getopt()` and `getpass()` may, in some cases, be dangerous because in some implementations of the C library they can lead to internal buffer overflows. The problem is thus more related to the implementation than the interface. Glibc does not have these problems. In particular, `getpass()` allocates as much memory as required to hold a complete line of input.

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

int main()
{
        char *string = "Hello world!";
        char buffer[8];
        strncpy(buffer, string, sizeof(buffer));
        printf("%d\n", strlen(string));
        return 0;
}
```

Execution of the program:

```
$ ./strlen
21
```

Table 6: Non-observance of `strlen()` interface

Functions `streadd()`, `strecpy()` and `strtrns()` should also be used cautiously, but they do not exist in glibc and thus cannot cause any problem.

We saw in this section that many of the standard functions of the C library are difficult to use correctly to avoid buffer overflows. There often exists replacing functions that allow specifying the size of buffers more easily, but programmers can be discouraged to use them because they do not always offer a consistent interface.

2.4 Non-Observance of an Interface

There is non-observance of an interface when one passes to a function parameters that do not satisfy its specification. The line can be fuzzy between this section and the last one. One could argue that overflow risks presented in the last section were all due to non-observance of an interface. Contrary to interfaces without bounds checking, in this section we deal with functions conceived to avoid buffer overflows, but used incorrectly.

For instance, `strlen()` takes a pointer to a string terminated by a null character. Passing to it an unterminated string is thus an error and causes a buffer overflow more often than not.

In the program of Table 6, the buffer is too small to receive the whole string. There is no buffer overflow while copying since function `strncpy()` observes the buffer size. However, this makes the buffer unterminated since the null character cannot be copied. This leads `strlen()` to overflow the buffer and give a result

```
#include <ctype.h>

void off()
{
        unsigned char buffer[8];
        unsigned char string[8] = "hello!!";
        int i;
        /* Upper case... */
        for (i=0; i<=8; i++)
                buffer[i] = toupper(string[i]);
}

int main()
{
        off();
        return 0;
}
```

Execution of the program:

```
$ ./off1
Segmentation fault
```

Table 7: Off-by-one error

different from the expected one. Indeed, there was clearly an access outside the buffer since 21 characters were counted while only 8 characters were copied.

2.5 Off-by-One Error

An off-by-one error happens when some calculation gives a result that is one unit greater or smaller than the right answer. This usually happens when the programmer makes a mistake in a formula or in a loop condition, because he forgets an item, often the null character, or counts one more, or he does not use the right index as the first element (1 vs. 0). We are interested in cases where an off-by-one error produces a buffer overflow. It happens mostly when dealing with indices.

The program of Table 7 shows a typical off-by-one error. In this program, the error is made by using the operator `<=` instead of `<`.

A simple off-by-one error is thus enough to crash a program. Moreover, we will see in Section 3 that these errors may cause serious vulnerabilities.

```
int main()
{
        int array[10];
        int *ptr = &array[0];
        int i;
        for (i=0; i<30; i++)
                array[i] = i;
        return *ptr;
}
```

Here is the execution:

```
$ ./fault1
Segmentation fault
```

Table 8: Abnormal termination produced by an invalid pointer value.

3 Consequences of Buffer Overflows

In this section, we will see the possible consequences of buffer overflows, and more specifically techniques to exploit them.

3.1 Abnormal Termination

An abnormal termination happens when a program does something serious and uncontrolled. Buffer overflows can cause abnormal termination in many different ways. One of these is when an overflow gives an invalid value to a pointer. When the pointer is used to access memory, the result is something called a segmentation fault. This is what happens in the program of Table 8.

3.2 Abnormal Execution

If buffer overflows can cause the termination of a program, thus producing a denial of service, there are also many possible consequences that are more serious.

If some data are placed in memory next to a buffer, they can be overwritten by an overflow of that buffer. Depending on the meaning of these data, the consequences can be more or less important. For example, overwriting a file name, a user identification, or the description of his access rights has obvious important consequences in a privileged program.

Other data can have indirect effects. This is the case, for instance, of a variable that indicates the remaining number of tries for a user to be authenti-

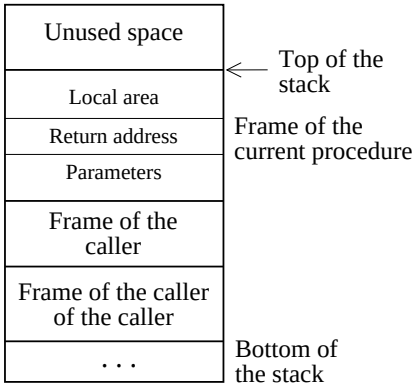

Figure 1: Run-time stack of a program. The (stack) frame of a function is made of its parameters, its return address and its local area.

cated before his account is deactivated. Such a variable could be used to get an unlimited number of tries before finding the right password.

Attack possibilities vary greatly, depending on data that can be overwritten by an overflow. The remainder of this section shows some of the principles that allow an attacker to control the execution of a program.

3.3 Arbitrary Code Execution

Among the possible consequences of buffer overflows, the most serious one is probably the execution of arbitrary code. This goes further than abnormal execution because the possibility to execute arbitrary code greatly simplifies the task of someone who wants to exploit a buffer overflow. It is not obvious at first sight how a buffer overflow, in an area that should contain data, can allow execution of arbitrary code. We will see that many techniques can be used to make this happen.

3.3.1 Overwriting the Return Address

Most programming languages allowing recursivity use a stack to handle the parameters, the return address and the local variables of a function. It is notably the case for the C and C++ languages.

On a function call, the caller first pushes the function parameters on the stack, then it pushes the return address before transferring control to the function. The return address is normally the address of the instruction right after the function call. At the beginning of its execution, the called function reserves space from the stack for its local variables. It also uses this space to save the value of some important registers that are modified by the function, but which have to be restored to their original value before it returns to its caller. Figure 1 illustrates the organization of the stack.

If one could modify at will a return address on the stack, he would be able to make the program transfer its execution to an arbitrary place in memory when it tries to return to its caller. A buffer overflow can overwrite a return address, but some conditions must be met.

First, the buffer must be located on the stack. Second, a return address has to be present after the buffer. While it is possible to overflow a buffer on the side where the buffer starts[1], it is rarer. Buffer overflows usually happen at the end of a buffer.

With this knowledge, we return to Figure 1, where we can see some indications about the top and the bottom of the stack, but no mention of the direction in which the addresses grow. The reason is simple, it depends on the architecture in use, notably the operating system and the processor. For the purpose of this survey, we will assume that the stack grows towards lower addresses, which is the most common case.

Suppose we have a buffer in the local area of a function. If this buffer can be overflowed, the first values that can be overwritten are other local variables following the buffer. Next, the return address can also be overwritten. Attackers thus want to overwrite the return address of the function to make their code execute when the function tries to return to its caller. To do this, they have three problems to solve.

Firstly, an attacker must succeed in placing his code[2] inside the memory of the target program. There are many ways to achieve this. Almost any source of input of the program is susceptible to be used by an attacker to insert code inside the program. For example, he can use the standard input stream, a file, a network connection or an environment variable. Depending on the method used by the program to handle the input, the attacker may not have absolute freedom on the code he can insert in the program. For example, if the code is treated as a string, it is usually not possible to embed null bytes anywhere but at the end of the string since they indicate the end of the string. In this case, the machine code must be written to avoid null bytes, which is usually not too difficult.

Secondly, the attacker has to know where his code will go in memory to overwrite the return address with the right value. This code can be among static data of the program, inside the heap or inside the stack; it does not matter. It is however important to know its address. An attacker can use a debugger and a copy of the program to find this information more easily. If he only has access to the source code, he can compile it to get an approximation of the address. If he does not have the executable or the source code, he must proceed by trial and error.

Many factors can make the exact address hard to guess for the attacker. For instance, source code compiled with different compilers or options can result in

[1] This is sometimes called an underflow, but we will not use this term because it is ambiguous.

[2] This code is often called *shellcode* since it often executes a shell with administrator privileges. The term *egg* is also used to refer to what is placed inside the overflowing buffer, which allows an attacker to obtain what he wants. It can be the shellcode, pointers, or other things.

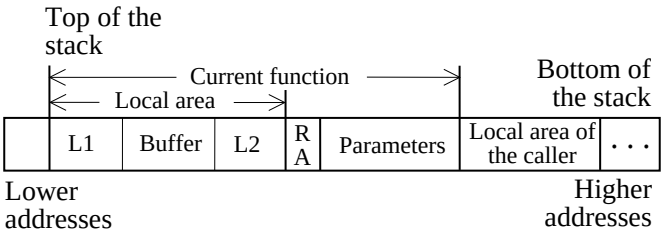

Figure 2: Stack growing towards lower addresses. To overwrite the return address (RA), Buffer has to overflow and overwrite L2, and then RA.

different executable programs. In addition, when memory is allocated dynamically, the address may be different from one execution to the other. Even stack addresses can change from one execution to the other, depending on the callers.

However that may be, an attacker can use a trick allowing an attack with only an approximate knowledge of the address of his code. Indeed, he only has to put instructions that do nothing in front of the code. Such instructions are called `NOP`[3]. This way, it is possible to use the address of any of these instructions as the return address and the attacker's code will be executed.

Thirdly, the attacker must know the position of the return address to overwrite relatively to the buffer that overflows. Figure 2 shows a more detailed view of the stack organization. The attacker who wants to overwrite the return address (RA) thus has to fill the buffer (Buffer), and then the remaining local area, from the end of the buffer to the return address (L2). At first sight, we could think he must know the exact size of the buffer and that of the remaining local area. However, in practice he does not have to. He can begin repeating the address of his code well before RA is reached and continue to do so well after it. He must however have an idea of the size of Buffer and L2 since the stack is not infinite and the program will crash before control is given to his code if there is an attempt to access memory outside the stack. On the other hand, if the return address is not reached, his code will not be executed. Between these two extremes, there is enough room to operate.

Note that nothing prevents using the buffer itself as the container for the code of the attack. With only one overflow, it is thus possible to insert the code in the program and overwrite a return address so that it points to this code. This method is explained in detail by Aleph One in [Ale96], who suggests creating an attack string that is 100 bytes longer than the buffer that will be overflowed. The string is built so that the code to execute is located in the middle, preceded by `NOP` instructions and followed by the return address, which is repeated until the end of the string, and which should point somewhere in the `NOP` instructions. Figure 3 shows the stack once the buffer has overflowed this way.

This method is not the only possible one. If a buffer is too small and the

[3] `NOP` is an abbreviation for *no operation*.

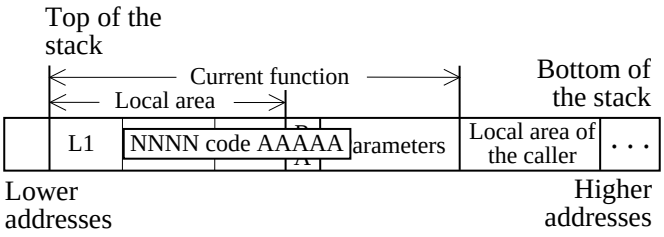

Figure 3: Stack with an overflowed buffer. N represents a `NOP` instruction and A represents an address that is inside the `NOP`s.

return address is too near to the buffer, one could put the `NOP` and the code after the return addresses, and thus overflow further after the end of the buffer. He only would have to be careful not to go beyond the stack limit.

The technique described in this section and its variations are very popular for pirates because they are relatively simple and they can be adapted to a large number of vulnerable programs. The following section presents alternatives that can be used in some situations where it is not possible to overwrite the return address, or at least not directly.

3.3.2 Overwriting the Saved Stack Frame Pointer

This technique allows exploiting a buffer overflow and executing arbitrary code in some particular situations where it is not possible to overflow a buffer by more than one byte. This may be the case when there is an off-by-one error in a program. It is described in detail in [klo99]. To understand how it works, it is necessary to know more about the code produced by a compiler for a function call in C or C++.

What follows is written for IA-32 processors, but it is also applicable to other little-endian[4] processors that have 32-bit words and a stack growing towards lower addresses.

Compilers usually reserve one of the processor registers to hold the address of the stack frame of the current function. This register, called `EBP`, usually points near the return address, so that the parameters of a function are at a positive offset of the stack frame pointer, and local variables in the local area of a function are at a negative offset. This way, the compiler can use a constant offset to access parameters and local variables. It would not be possible to do so if the stack pointer (`ESP`) was used directly, since its value can change during a function call.

The stack frame pointer thus has to be saved before it is initialized when entering a function. When leaving a function, it has to be restored so that the calling function has access to its stack frame. Table 9 shows instructions that make all this happen. In `funct()`, `EBP` is first saved on the stack, then

[4] A little-endian processor stores the least significant byte at the lower address. A big-endian processor stores the most significant byte at the lower address.

Here `main()` calls `funct(1, 2, 3)`. The arguments are pushed in the reverse order of appearance. This means that in memory they are in the order of appearance because the stack grows towards lower addresses.

```
0x8048431 <main+9>:     push   $0x3
0x8048433 <main+11>:    push   $0x2
0x8048435 <main+13>:    push   $0x1
0x8048437 <main+15>:    call   0x80483c0 <funct>
0x804843c <main+20>:    add    $0x10,%esp
```

The following is the prologue and the epilogue of `funct()`. It declares an array of 60 bytes in its local area and 2 local variables. More space is allocated for internal use by the compiler and for alignment.

```
0x80483c0 <funct>:      push   %ebp
0x80483c1 <funct+1>:    mov    %esp,%ebp
0x80483c3 <funct+3>:    sub    $0x58,%esp
...
0x8048424 <funct+100>:  leave
0x8048425 <funct+101>:  ret
```

It is important to know that the `leave` instruction is equivalent to:

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

Table 9: Typical assembly code corresponding to a function call and return in C.

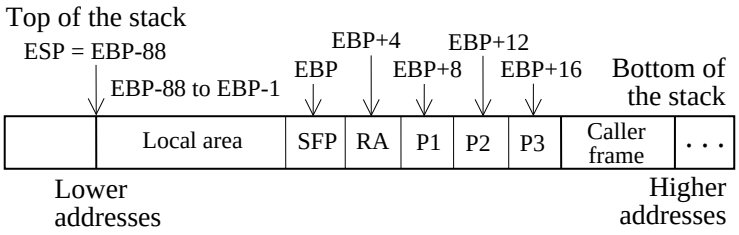

Figure 4: Detailed organization of a stack frame. P1, P2 and P3 represent the three parameters of the function, RA is the return address and SFP is the saved frame pointer, that is to say the old value of EBP.

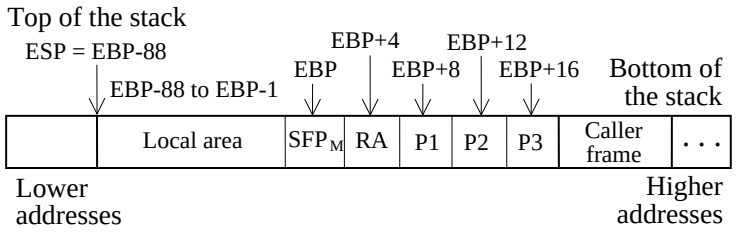

Figure 5: Organization of the stack frame of `funct()` after an overflow has overwritten the least significant byte of the saved frame pointer.

it is initialized to point to the top of the stack. After this, `ESP` is decreased, thus creating the local area. Eighty-eight (0x58) bytes are reserved. The `leave` instruction allows restoring the original values of `ESP` and `EBP` before the function returns to its caller.

Figure 4 gives a detailed picture of the stack after the execution of the `sub` instruction. On the stack, each parameter is usually 4 bytes wide. On the other hand, in the local area, variables are usually organized in a much more flexible manner. There are often structures and arrays.

Now, we will see how it is possible to cause the execution of arbitrary code with an overflow of only one byte. We assume that the buffer is located on the stack, right before the saved frame pointer, SFP. It is thus only possible to overwrite the least significant byte of the saved frame pointer. We call $\mathrm{SFP_M}$ the value of SFP with least significant byte modified. Figure 5 shows this situation.

When `funct()` terminates, the `leave` instruction is executed and the state of the stack is described by Figure 6. We note that `EBP` has taken the value of the modified stack frame pointer. If SFP had not been modified, `EBP` would point to the saved frame pointer of the caller, and everything would be correct. Since its least significant byte was modified, $\mathrm{SFP_M}$ (and thus `EBP`) can point to at most 255 bytes before or after SFP.

Figure 7 shows the stack state after the return to `main()` is complete. The control is correctly returned to `main()` and `ESP` has the right value. However,

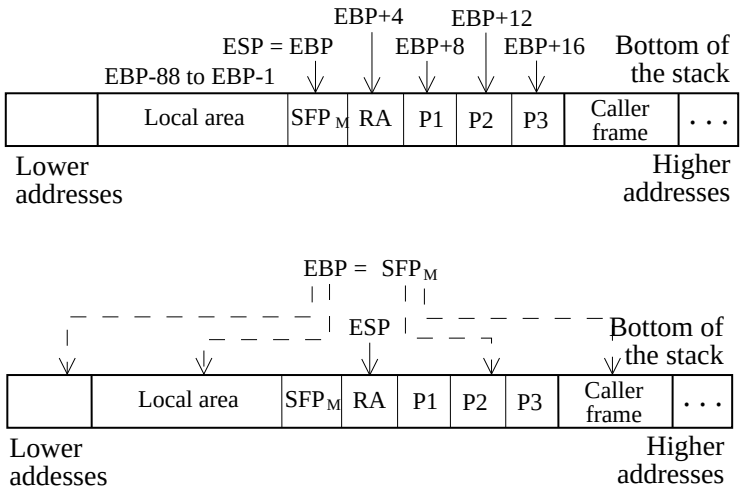

Figure 6: Result of the `leave` instruction with a modified saved frame pointer, explained with the equivalent instructions `mov %ebp, %esp` and `pop %ebp`. The part at the top of the figure shows the state after the `mov`. The part at the bottom shows the state at the end of the `pop`. The dashed arrows represent some of the addresses that `EBP` can take.

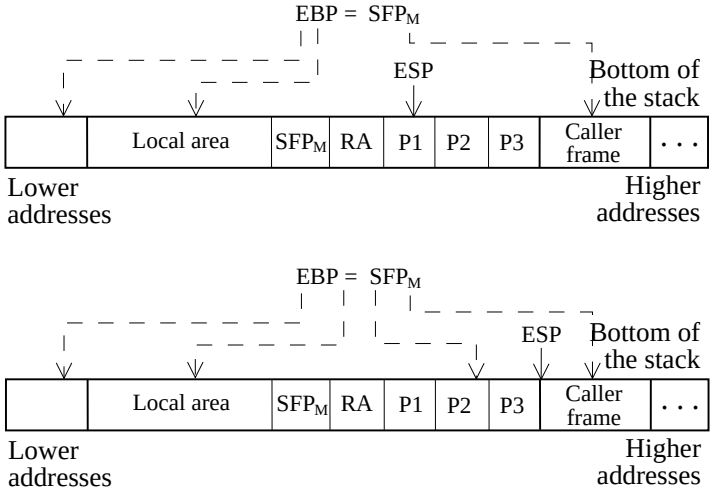

Figure 7: Stack state after returning to `main()`. The control is really returned to `main()` because `ESP` has the right value and the return address (RA) was not modified. The part at the top represents the state of the program after the `ret` instruction. At this time, `EIP`, the instruction pointer, has the value of RA. The part at the bottom shows the state of the program after the `add` instruction, which removes the parameters from the stack.

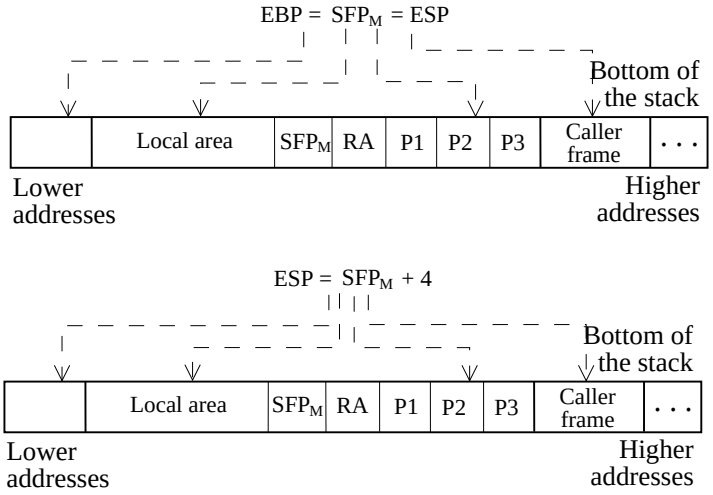

Figure 8: State of the stack at the end of `main()`, the calling function. As in Figure 6, the `leave` instruction is broken in two steps to better see what happens.

`EBP` keeps its wrong value, something that causes problems when `main()`, the caller of `funct()`, ends.

Indeed, function `main()` is not different from others and it returns to its caller in the same way that `funct()` does, that is to say, using `leave` and `ret` instructions. Figure 8 shows the effect of the leave instruction on the state of the program. The following instruction is `ret` and it has the effect of giving the control to the instruction at the address pointed to by `ESP`. So that the attack succeeds, the address at this place must be that of the attacker's code. As in the case of overwriting the return address, the code can be just about anywhere in memory, but it is often easier to put it in the buffer that overflows.

To sum up, one can exploit a buffer overflow of only one byte by overwriting the least significant byte of a saved frame pointer. It suffices to make it point 4 bytes before the location in memory where the address of the code to execute is stored. Two `leave` and `ret` sequences later, the code executes.

This technique is much more sensitive than the overwrite of the return address. Indeed, having only a partial control over the saved frame pointer, it is not possible to give it a completely arbitrary value. It may be that all values that it can take (+4) are addresses that cannot be controlled. In this case, it is not possible to control where the execution will continue.

In addition, once the control returns to the calling function, the stack frame pointer (`EBP`) is no longer valid. If the function does not return immediately, it can have a strange behavior if it tries to use its parameters or its local variables. It may also be possible that while trying to modify its variables, it modifies the code in the buffer that the attacker wants to execute. However, a meticulous at-

```
#include <stdio.h>

/* Declaration of a function pointer */
void (*hello_ptr)(const char *);

void english_hello(const char *name)
{
        printf("Hello %s!\n", name);
}

int main(int n, char **argv)
{
        /* Initialization of the function pointer */
        hello_ptr = english_hello;
        if (argv[1])
                /* Use of the function pointer */
                hello_ptr(argv[1]);
        return 0;
}
```

Table 10: Function call through a function pointer

tacker can sometimes build its attack so that the calling function sees "credible" values in place of its real parameters and variables. It is also possible to put the code to execute elsewhere than in this buffer, which gives more flexibility.

3.3.3 Overwriting a Function Pointer

When a buffer is allocated statically, or dynamically on the heap, it is usually not possible to overwrite a return address. However, other elements can be the target of an attack, and allow the execution of arbitrary code. Among them are function pointers. Indeed, the C and C++ languages allow calling a function through a function pointer. The program of Table 10 shows how it is done.

If the function pointer could be overwritten between its initialization and its use, it would be possible to execute arbitrary code instead of the function that should be referred to by the pointer at the time of use. In [Con99], the author explains in more detail how this is possible. As in the case of buffer overflows overwriting a return address, the easiest way for an attacker is often to insert the code to be executed directly in the buffer that overflows. The value to give to the function pointer is then the address of this buffer.

Contrary to what usually happens when a return address is overwritten, there can be much time elapsed between the moment at which the function is

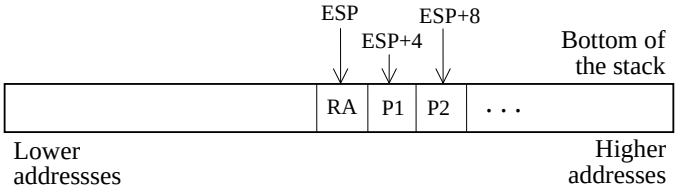

Figure 9: Stack at the beginning of the execution of a function

overwritten and the moment at which it is used. This can play against the attacker since an overflow often has to overwrite important variables before it reaches the targeted function pointer. For the attack to succeed, it is necessary that using of these variables does not make the program crash or modify the injected code or data.

3.3.4 Copying and Executing Arbitrary Code With a Double Return

The technique presented here is described in detail in [Woj98]. In [Ale96] and [Sol97a], the authors give more information about some ideas that it uses. This technique is a particular case of the overwrite of a return address that was presented in Section 3.3.1. Here, instead of returning directly to the address where the code to execute is located, the execution first returns to a function which copies the code to execute elsewhere in memory and then returns to the new copy of the code.

The first question that comes to mind is why one would want to copy some code already in memory before executing it? The answer is that some part of the memory can be non-executable. For instance, some operating systems do not allow code to be executed on the stack. The technique presented here allows an attacker to copy his code, already injected on the stack, to the heap.

To understand how it works, it is important to remember the organization of the stack when a function begins its execution. Figure 9 shows this organization. Usually, the calling function first pushes the parameters, then it gives control to the called function at the same time that it pushes the return address with the `call` instruction.

Here, instead, it is a `ret` instruction that is used to call a function able to copy some code, `strcpy()` for instance, that will be called. Right before this instruction is executed, the stack thus has to look like Figure 10. Thus CA must overwrite the old RA, the new RA must overwrite the old P1, the new P1 must overwrite the old P2, and the new P2 must overwrite the following value.

It is important to understand that in this figure, CA is the address of the function that should be called, but for the function executing the `ret` instruction, it is also its new return address.

In theory, to copy some code and execute it, it is enough to use:

- as CA: the address of `strcpy()`;

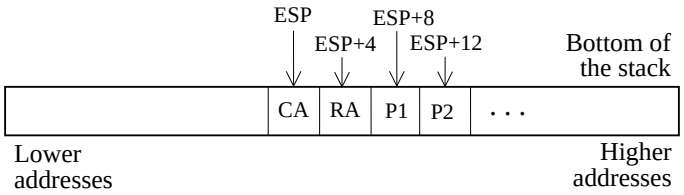

Figure 10: Stack before a "call through a `ret`". CA represents the address of the function that will be called, and RA represents the address where this function will return when it terminates.

- as RA: the address of any memory area writable and executable;
- as P1: the same address as for RA;
- as P2: the address of the code to execute, which is already in memory.

For P2, there is no problem if the address does not contain any null byte. If it does, it may be necessary to find some other location to hold the code. For P1 and RA it is possible to choose just about any valid address from the heap that does not have one null byte. This should be easy. It is important to note that the heap is executable more often than not.

If it is so important to avoid null bytes, it is because buffer overflows are often caused by the incorrect use of string-handling functions. If a null byte is present in the string that tries to exploit a buffer overflow, the part of the string after this null byte is ignored, and the overflow cannot be exploited correctly.

Things get more complicated with CA, which must take the address of `strcpy()`. This is because, as a protection measure against attacks of this kind, the patch from the Openwall Project [Sol02], which implements a non-executable stack for Linux, also modifies the address at which the C library is loaded in memory so that it is always under 01000000_{16}. This way, all addresses pointing to functions in the C library contain at least one null byte and it is not possible to use them when the overflow is produced using a function handling strings.

Fortunately (or unfortunately, depending on the point of view) it is possible to work around this protection. Indeed, while the C library can be loaded at addresses that all have at least a null byte, it is not the case for the main program, which is usually loaded at addresses with the most significant byte equal to 8. Both code and data of the program are loaded there. A part of the code of the program makes up the *Procedure Linking Table*, or PLT.

To understand the role of the PLT, it is necessary to know how linking is done with shared libraries under Linux. When a program has a reference to a symbol in a shared library, it is not resolved at compile time, but at run time. Indeed, when a program makes a call to a function in a shared library, the linker does not know in advance where in memory the library will be loaded, much less the position of the function within the library. It thus creates an entry in

the PLT for this function. Table 11 outlines how a function in a shared library is called.

This example demonstrates that it is possible to call a function of the C library without pointing to it directly. Using the PLT makes it possible to avoid working with null bytes. To have an entry in the PLT, a function of the C library only has to be called once, no matter where in the program.

The function `strcpy()` is not the only one that can be used to copy code from some place to another. `strncpy()`, `sprintf()`, `wcscpy()` and `memmove()` are only some of the functions of the C library that can be used to that effect.

To summarize this section, we saw a technique allowing the execution of arbitrary code even in the presence of two protection mechanisms (non-executable stack and modification of the address of the C library) against this kind of attack. These two mechanisms will be presented in more detail in Section 4.

3.3.5 Overwriting a Pointer and then the Structures of atexit()

The technique presented in this section is explained in more detail with other similar ones in [BK00]. Here the goal is to exploit a program even when the return address is protected against overwrite. Section 4 describes methods preventing or allowing detection of the overwrite of return addresses. The technique presented in Section 3.3.1 cannot be used in these situations. We saw in Section 3.3.3 that it was often possible to overwrite a function pointer. The technique presented here is an extension of this technique when a buffer overflow does not allow overwriting a function pointer, but rather a data pointer.

Many conditions must be met so that this technique can be used successfully by an attacker.

- There must be a buffer overflow.
- It must be possible to overwrite a pointer with the overflow.
- The pointer must be used as the destination of a copy operation after the overflow.
- The pointer must not be initialized between the overflow and the copy operation.
- The attacker must have control over data that are copied.

The program presented in Table 12 meets all these conditions. It certainly does not look like a real program. One would rather say it is devised specifically to be exploited! It indeed is, so that the working of the attack is clearer. In [BK00], the authors show more convincing examples of vulnerable programs. We observe that just about anything can happen between `strcpy()` and `strncpy()`, provided that `p` is not modified. It is also important to notice that `strcpy()` can be replaced by any function causing the overflow of `buffer` and that `strncpy()` can be replaced by any copy function.

This is the code corresponding to the call of function `strcpy()` of the C library when it is linked as a shared library:

```
0x80483f9 <main+9>:     push   $0x8048474
0x80483fe <main+14>:    push   $0x80495c0
0x8048403 <main+19>:    call   0x8048300 <strcpy>
```

We observe that the call is not made directly to the C library, but rather in the PLT:

```
0x8048300 <strcpy>:     jmp    *0x8049584
0x8048306 <strcpy+6>:   push   $0x18
0x804830b <strcpy+11>:  jmp    0x80482c0 <_init+40>
```

If we look at the value present at address 0x08049584, we notice that it corresponds to the address of the next instruction, the `push`. This value is part of a table called GOT, for *Global Offset Table*.

```
0x8049584 <_GLOBAL_OFFSET_TABLE_+24>:   0x08048306
```

The next `jmp` instruction gives the control to the dynamic linker. The dynamic linker then resolves the address of `strcpy()` and it saves this value in the GOT.

```
0x8049584 <_GLOBAL_OFFSET_TABLE_+24>:   0x001a4120
```

This way, on the next call to `strcpy()` via the PLT, the first `jmp` instruction will give the control directly to the C library, rather than to the dynamic linker.

We can also observe that Linux has the non-executable stack patch from the Openwall Project applied because the most significant byte of the address is 0.

Table 11: Function call in a shared library

```
#include <string.h>

int main(int n, char **argv)
{
        char *p;
        char buffer[20];

        p = buffer;

        strcpy(p, argv[1]);

        strncpy(p, argv[2], sizeof(buffer));

        return 0;
}
```

Table 12: Program allowing the overwrite of a pointer

The attack is carried out in two steps. Initially, the buffer overflow allows overwriting the pointer. It is given the value of the address of a function pointer. Then, in a copy operation, the function pointer is overwritten to point to the code of the caller.

For the first step, it is necessary to know the address of a function pointer to be used later. In [BK00], the authors suggest to overwrite a function pointer in the structures used by the function `atexit()`, among other things. This function allows the execution of other functions at the time the program terminates. Overwriting a function pointer stored in its internal structures thus makes it possible to execute additional code, provided the program terminates normally. It is useful to know that two functions are registered automatically at the beginning of a program, one to clean up the main program, and another one to clean up the C library.

In [BK00], the variable `fnlist` is used to find the position of the structure in memory. It has been replaced by variable `initial` in glibc, but it is not visible since it is not exported. It is however possible to find its address by following the execution of `atexit()` and comparing it to the source code of `glibc`. Table 13 explains how this is done.

When the pointer that will be overwritten is identified, it is with its address that the pointer used as a destination in a subsequent copy operation is overwritten. The arrow labeled 1° in Figure 11 indicates the expected result after this step.

For the second step, the pointer just overwritten is used again as the destination of a copy operation. Here, even if the program checks the bounds, it

First, it is necessary to identify the code that accesses this structure. It is `__exit_funcs` that contains a pointer to `initial` at the beginning of the execution of the program.

```
0x4004cf7a <__cxa_atexit+122>:  mov    0x8c0(%ebx),%eax
0x4004cf80 <__cxa_atexit+128>:  mov    (%eax),%esi
```

At this point, `eax` contains the address of `__exit_funcs` and `esi` contains the address of `initial`, which are as follows (the text following `(gdb)` is a command to the debugger gdb):

```
(gdb) info reg eax esi
eax            0x40134d1c       1075006748
esi            0x40139e40       1075027520
```

The content of the first 10 words of initial is:

```
(gdb) x/10 $esi
0x40139e40 <errno+928>: 0x00000000  0x00000002
0x40139e48 <errno+936>: 0x00000004  0x40009e50
0x40139e50 <errno+944>: 0x00000000  0x00000000
0x40139e58 <errno+952>: 0x00000004  0x08048500
0x40139e60 <errno+960>: 0x00000000  0x00000000
```

The goal is to overwrite the pointer at 0x40139e40 + 12 (i.e. 0x40009e50) or 0x40139e40 + 28 (i.e. 0x08048500).

Table 13: Address of the structures used by `atexit()`

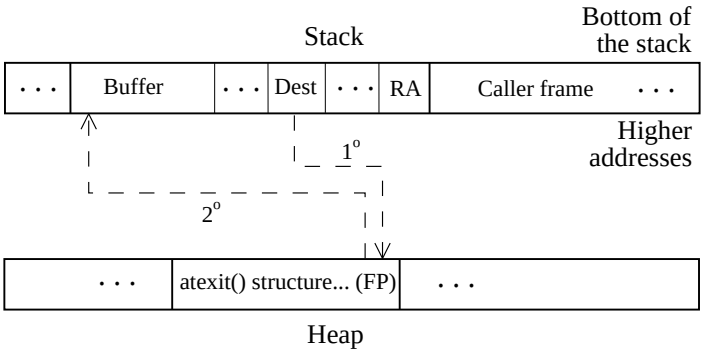

Figure 11: Before overwriting the function pointer FP, the destination pointer of a copy operation (Dest) must first be overwritten. The arrows represent pointers. In this figure, the attacker code is located in the buffer that is being overflowed.

does not prevent the exploitation since only 4 bytes have to be copied. This is because the pointer does not point to what it should anymore. The arrow labeled 2° in Figure 11 shows the result of this operation. The data source for the copy of course has to be under the control of the attacker so that he can give an arbitrary value to the function pointer.

To summarize this section, we saw an attack allowing bypassing another protection mechanism against buffer overflows. Even though this kind of attack is not applicable to all programs, it demonstrates that the protection technique is not infallible.

3.3.6 Other Targets to Execute Arbitrary Code

There are many other targets that can be overwritten to execute arbitrary code. Some of them are presented here summarily with references to more information about them.

Overwriting the vtable pointer In C++, the addresses of the virtual methods of a class are stored in a table called *vtable* and a pointer to this table is present in each instance of the class. It is often possible to overwrite this pointer so that it points to a table created by the attacker, who can then execute anything he wants when a virtual method is called.

This technique is described in [rix00]. This article also gives an example where the attacked program continues its execution normally after the attack ends.

Overwriting a jmp_buf In [Con99], the author suggests an alternative to overwriting a function pointer. It consists in overwriting the instruction pointer that is stored in a `jmp_buf`. This structure is used by the functions `setjmp()`

and `longjmp()` of the C library. The structure makes it possible to memorize part of the state of a program, including the stack pointer and the instruction pointer, and to go back to this state later. They are mostly used in error handling.

Overwriting a pointer and then a return address In [BK00], the authors present an alternative to the technique mentioned in Section 3.3.5. Instead of overwriting a function pointer used by `atexit()`, it is possible to overwrite directly the return address of a function. The idea is that if the overwrite is targeted directly at the return address instead of starting in a buffer, the StackGuard compiler will not detect it. It is important to note that at least one version of this compiler is immune to this kind of attack [Imm00]. Section 4 gives more details on how StackGuard works.

Overwriting a pointer and then the GOT Still in [BK00], yet another alternative is explained. This one requires about the same conditions as the other two to be applicable, but it has the feature of allowing bypassing the protection mechanisms of StackGuard, Stack Shield, and the patch preventing execution of code on the stack all at the same time.

It is done by overwriting the entry of the GOT corresponding to a function of the C library that is called right after the operation that causes the overflow. A pointer to the attacker's code is placed in the GOT and is immediately followed by this code. Everything is thus copied in one step.

Overwriting the pointer __exit_funcs In [Bou00], the author notices that it is sometimes possible to overwrite the pointer to the structure used by `atexit()` when the C library is linked statically. He explains that this structure matches perfectly with the one used by the operating system to pass arguments to a program. It is however important to know that the structure used by `atexit()` in glibc is different from the one described in [Bou00], and that this technique is thus probably not applicable under Linux. What is described in the article comes from the BSD world.

Overwriting the structures of malloc() and then anything Most `malloc()` implementations store management information about allocated memory blocks in-band, that is consecutive to a memory block. Among this information are pointers to implement a double linked list of memory blocks. In [Ano01], it is explained that since these pointers are stored in-band, they can be overwritten when a buffer overflow is possible in the heap. By choosing carefully the values used to corrupt pointers of the double linked list it is possible to trick a call to `free()` into overwriting an arbitrary location with an arbitrary value. Thus, the location overwritten can be any target that we have seen in this section, which can lead to execution of arbitrary code.

3.4 Other Types of Attack

In this section, we describe summarily certain types of attack that, while not allowing the execution of arbitrary code, are often as dangerous as the ones allowing it. Indeed, most programs contain much code that can be used maliciously. The following techniques explain how this code can be exploited.

3.4.1 Returning in the C Library

In [Sol97a], the author details a technique allowing to defeat the protection offered by his own patch (now part of the Openwall Project) preventing the execution of code on the stack. We mentioned it in Section 3.3.4. It works by modifying the return address so that it points to a function of the C library instead of the stack as in Section 3.3.1. The words following the return address on the stack are also modified so that the function can see "interesting" parameters.

The most common use of this technique is to execute a shell (/bin/sh) using `system()`. This can be done without having to inject `"/bin/sh"` in memory since the C library already contains this string. The execution of a shell allows an attacker to do just about anything he can imagine thereafter.

The article also mentions it is possible to call two functions in a row when the first one takes only one parameter.

It also explains a protection mechanism against this kind of attack, which consists in loading libraries at addresses with null bytes. This mechanism is explained in Section 4, but it is also mentioned in Section 3.3.4, which explains a technique that can defeat it.

3.4.2 Returning in the PLT, Overwriting the GOT and Returning to system()

In [Woj98], the author explains another way to exploit a program protected by the patch from the Openwall Project preventing the execution of code on the stack. Contrary to the technique explained in the previous section, this one works when the C library is loaded at an address containing a null byte. Contrary to the one presented in section 3.3.4, this one also works if the data areas are not executable either. On the other hand, it does not allow the execution of arbitrary code.

The technique consists in modifying the return address and the word following it so that the execution will return two times in a row in the PLT for the call of `strcpy()`. This way, null bytes in addresses of the C library are avoided. In the first call (or rather the first return), the source pointer (second parameter) points to a file name. This file name ends with the address of `system()` in the C library. The destination pointer is the address of the GOT entry for `strcpy()` adjusted so that it is overwritten by the address of `system()`, which is at the end of the file name to be copied.

In the second call (or return!) it is thus `system()` that is called instead of `strcpy()`. Its only parameter is the name of the file to be executed, which the

attacker took care to place in a temporary directory before the attack.

One could wonder why would one want to return two times in a row in such a complex way to finally call `system()`? Why not call `system()` directly by way of its PLT entry? The problem with this is that the program to be exploited may not use `system()`, and thus may not have a PLT entry for `system()`. On the other hand, this function is always present in the shared C library.

Another question could be how is it possible to put an address with a null byte in the file name, but not directly as a return address? This is because the most significant byte is the null one, and it is the last one on a little-endian processor. Since a parameter has to be placed after the return address, it is generally not possible to overwrite the return address with an address containing a null byte. Since nothing has to be added after the file name, it can end with a null byte.

Overwriting a pointer, and then the GOT to call system() In [BK00], the authors explain another technique allowing bypassing many protection mechanisms against buffer overflows. As in Section 3.3.6, many conditions have to be met so that this technique is applicable. It consists in overwriting the destination pointer of a copy operation, and then overwriting the GOT entry of a function called right after with the address of `system()`. This function has to take as parameter a string, which the attacker must control so he can decide what he wants to execute.

3.5 Published Exploits

Very often, when a buffer overflow vulnerability is discovered and published, a program to exploit it is published at the same time or not long afterward. Thus, there are now hundreds of publicly available exploits for buffer overflow vulnerabilities, which can be used as examples to build exploits for new vulnerabilities. Here are just a few of them [cla05, Fra01, Kae01].

4 Avoiding Vulnerabilities

There are many different solutions to avoid buffer overflow vulnerabilities. They differ mainly by:

- doing their work before the program is run, at run time, or both;
- requiring changes in the source code, the compiler or the operating system;
- being ad hoc or based on formal methods;
- avoiding overflows or only some consequence of them;
- possibly giving false negatives or false positives;
- adding a large run-time overhead or not.

This section talks about all the different techniques available, no matter how they are classified and whether they are efficient or not.

4.1 Solving the Problem at the Source

When a program is correct, without bugs, it does not have vulnerabilities such as buffer overflows. Experience shows that it would be unrealistic to expect programmers to write bug-free code. With languages such as C, it is easy to do something incorrect [CWP+00, Fry00, Wag00]. New vulnerabilities are discovered almost every day in different software. We are thus looking for more automated ways of finding or avoiding vulnerabilities.

4.1.1 Using an Immunized Language

Some languages, such as Java, are immune against buffer overflows. Even so, there can be unwanted consequences to an attempt to access elements outside the bounds of a buffer. Some approaches that we will see in this section can thus be useful even for languages that are immune against buffer overflows. For example, the program Wasp uses static analysis to detect accesses outside the bounds of arrays in Java programs.

Moreover, it is not always possible to use a language other than C. For instance, the choice of a programming language is often very limited when it comes to writing operating system modules. There are also hundreds of millions of lines of C code in existing software and it would not be realistic to rewrite all of them in another language [CWP+00, GBPdlHQ+02, Whe01].

There is also a dialect of the C language, called Cyclone [Cyc02], that was specially designed to avoid some problems such as buffer overflows. Despite its apparent similarity with the C language, it is in fact a different language and existing C programs cannot simply be recompiled to benefit from its advantages.

4.1.2 Protecting Sensitive Memory Areas

This section is based on [COR02, CPM+98, Cow00, Woj98].

In a program, some data are more sensitive than others, thus greater care could be taken to protect them. In particular, some data described in Section 3, those that can be used to exploit buffer overflows, could be considered sensitive.

Almost all processors that support virtual memory offer page-level protection against memory overwrite. It is possible to place on a given page only data that have to be protected. Once they are initialized, the operating system is asked to disallow write access to the whole page. If the data have to be modified later by the program, the operating system must first be notified before they can be modified. After, the read only status is restored by calling the operating system again. Data are thus protected against any unintentional modification by the program. For example, it is protected against a buffer overflow in the page preceding the one containing sensitive data. This way, many attacks can be stopped.

Performance is usually not impacted too much by this technique because it requires only two system calls to modify an arbitrary amount of data. However, it would be very much impacted if modifications were frequent and could not be grouped together.

Everything needed to implement this technique is already present in modern operating systems. No modification has to be done on compilers. However, the program must be modified to indicate its intents. For this reason, this technique is usually applied only to data that are really sensitive and that are not modified often.

Here is a non-exhaustive list of objects that are interesting to protect. Protecting those that are not directly under the control of a program would require modifications to the compiler or system libraries.

- Security related variables (authentication, rights descriptions, ...)
- Pointers
- Return addresses
- Saved frame pointers
- Global Offset Table (GOT)
- Virtual function pointer table
- `atexit()` function pointer table

In addition, the code of applications is usually protected by default against overwriting on most operating systems. This is to allow more efficient sharing of code segments between applications that use, say, a common library. Moreover, when the operating system is running out of memory, it can easily decide to drop a code page without first writing it to swap space since it can always retrieve it directly from the original file.

Even if the protection of sensitive memory areas has the potential to considerably reduce the consequences of buffer overflows, it cannot prevent them directly. The difficulty in protecting sensitive areas is to identify them all correctly. It not only requires deep knowledge about the program that must be protected, but also the operating system, the compiler, the runtime environment, and all libraries it uses.

4.1.3 Using Safer Library Functions

We saw in Section 2.3 that some of the functions of the C library expose an interface that makes it difficult to do bounds checking. Here are some alternate functions that are easier to use safely.

strlcpy() and strlcat() Functions `strlcpy()` and `strlcat()` are not standardized, but they offer an increasingly popular alternative to some functions of the C library. They were created by the OpenBSD project, which has as its prime goal the security of the operating system. These functions are consistent in the sense that they always take the total length of the buffer as parameter, they always end a string with a null character, and they always return the length of the string that would result if there were no truncation. Under Linux, they are most often available in the glib library under the names `g_strlcpy()` and `g_strlcat()`.

astring from libmib The library libmib contains a part called astring, which offers an alternative to some functions of the C library. Instead of taking a pointer to a buffer, this library asks for a pointer to a buffer pointer. This way, the library can allocate a buffer of the right size and return to the program a pointer to this buffer. This library is described in [Cav98].

4.2 Testing

It is certainly possible to discover problems caused by buffer overflow vulnerabilities by testing a program. However, it is often impossible to be sure that it does not have these problems only by testing it. The tests it undergoes usually concentrate on the parts for which it was well specified, while the source of the vulnerabilities often comes from unexpected cases, those that were poorly specified.

Even so, there are automated testing tools that can be useful to discover some vulnerabilities in programs. For instance, in 1995, fuzz helped identifying problems in the use of arrays and pointers in 24 programs from UNIX and its derivatives [MKL+95]. Fuzz generates random data as input for programs to be tested.

4.2.1 Fault Injection

In [GOM98], the authors propose a slightly different approach. Instead of trying to cause a buffer overflow while testing, the program is instrumented to simulate buffer overflows. The program is then monitored for unsafe behaviors. If an unsafe behavior is detected after a simulated overflow at some location in the program, it means that special care must be taken to ensure that there is no overflow at this location.

This approach is not very useful in practice since it never detects overflows, it never suggests possible fixes, and the program must be instrumented manually,

4.3 Modifications to the Compiler and its Supporting Libraries

Attacks using buffer overflows exploit the knowledge of the inner workings of a program generated by a compiler. Without modifying the C language, it is

possible to modify the structure of the code generated by the compiler in such a way that buffer overflows are avoided or cannot be exploited. This section describes some possibilities.

4.3.1 Protecting Return Addresses

When a buffer overflow is possible on the stack, the easiest target for an attacker is certainly the return address. It is always present and it is (almost) always used at the end of a function. This explains why some compilers take special measures to protect the return address of a function against overwriting. The different protection methods discussed in this section are explained in [CPM+98, Imm00].

Using a canary One way to protect the return address is to place some value called a *canary*[5] in memory right before it. Here we assume the stack grows towards lower addresses, as it is often the case. When entering a function, a canary is pushed on the stack. When exiting it, the value of the canary is checked. If it changed, the return address might also have been modified and it cannot be used. Thus the program is stopped.

We assume here that in order to overwrite the return address using a buffer overflow, an attacker first has to overwrite the canary. To be efficient, the protection mechanism must not allow the attacker to overwrite the canary with the same value. For this, the canary can be chosen in many different ways.

It is possible to use a 32-bit word with the value 0; it is called a null canary. String operations usually do not allow copying null characters. The overflow would thus be stopped before it reaches the return address. A variant of this is to use a value containing many terminating characters. For instance, in a 32-bit word, it is possible to place characters "`\0`", "`\r`", "`\n`" and "`\xff`". This is called a terminator canary. It increases the odds that a copy operation is stopped by the canary.

Another possibility is to use a random value that is generated at the start of the program. This way an attacker cannot know in advance the value he has to use to overwrite the canary. It is also possible to XOR a random canari with the return address to improve security [Imm00].

The use of canaries was introduced by StackGuard and this protection mechanism was then replicated in other compilers such as Stack-Smashing Protector (SSP) [EY00] and Visual C++ 7.0 when using option "`/Gs`" [RWM02].

Using a different stack To prevent buffer overflows from overwriting return addresses, it is possible to place them on a stack distinct from the main program stack. To avoid breaking calling conventions between functions, the return address is copied to a different stack when entering a function and the copy is used to check the value of the main stack just before returning to the caller. This is the technique used by Stack Shield [Ven00] and Return Address Defender (RAD) [CH01].

[5] Welsh miners brought canaries in cages to detect hazardous conditions. When a canary died, they knew they had to leave the mine.

Using assistance from the processor and the operating system The use of a canary does not prevent overwriting return addresses, it only makes it possible to detect the overwriting before it causes problems. With some cooperation from the operating system, it is possible to detect all write accesses to some memory location. This way, it is possible to stop the program as soon as the return address is modified.

On Pentium processors and its successors of the IA-32 family, two functionalities are available to detect a write access to a specific address. First, the processor has four debug registers that can specify an address generating an exception when it is read or written.

When these registers do not suffice, it is possible to make a page, which is usually 4 kB, only available for reading. Every write access then generates an exception and the operating system can check if it is a return address that is about to be overwritten.

The compiler thus has to insert code at the beginning and at the end of each function to inform the operating system to start and stop protecting a return address.

Both techniques are very costly in execution time since they require a call to the operating system, but the page-level protection has a much greater cost. This is because a 4 kB page on the stack does not only contain return addresses, but also parameters, local variables, and possibly arrays. Each write access to one of these elements generates an exception and must be checked by the operating system before it gives control back to the program. The use of debug registers does not cause this problem, but does not allow protecting more than four addresses at a time.

It is also StackGuard that introduced these protection techniques. To do this, it uses some functionality added to Linux by MemGuard. The authors of StackGuard have measured that protecting a real program using debug registers increases its execution time by a factor of up to 11. For page-level protection, the factor can reach 460. Since the slowdown of these techniques is much too high, newer versions of StackGuard do not implement these techniques anymore.

4.3.2 Modifying the Order of Variables on the Stack

To obtain a better protection, it is possible to protect other important variables in addition to return addresses. For instance, pointers are often an interesting target in attacks exploiting buffer overflows. The C language allows compilers to choose the ordering of the local variables of a function. A compiler can thus change it so that pointers are placed before arrays. In this case, a buffer overflow cannot overwrite a pointer in the same stack frame.

Pointers passed as parameters cannot be moved without breaking calling conventions between functions, but they can be copied among local variables when entering a function. This way the copy is protected against buffer overflows and the original is ignored. Thus, it does not matter if it is overwritten.

It is the project Stack-Smashing Protector (SSP) that introduced the idea of modifying the ordering of variables to increase protection [EY00]. However,

this approach cannot protect structures containing arrays and pointers since the C language prohibits changing the ordering of elements in a structure. In addition, it cannot protect pointers passed in variable arguments of a function such as `printf()`.

4.3.3 Using an Alternate Implementation of Libraries

The libraries used by default on a system, for instance the C library, are often implemented to support some specification while being as fast as possible. Sometimes it is possible to use an alternate version of these libraries that does more checks than strictly required in order to detect the bad behaviors of a program.

Replacing malloc() and its friends This section is based on [Per93].

It is possible to replace the functions handling memory allocation to help detect buffer overflows. These functions are `malloc()`, `free()`, `calloc()`, `realloc()` `memalign()` and `valloc()`. Memory can be allocated in such a way that it coincides with the beginning or the end of a memory page. If memory is allocated at the end of a page, the following page can be marked as inaccessible. This way, the processor signals a fault as soon as the program accesses memory after the allocated area. When an error is detected, a debugger can show precisely what instruction causes the overflow. If memory is allocated at the beginning of a page, overflows before the area can be detected in the same way.

There can be alignment problems in some cases if the size of memory to allocate is not an integral multiple of the native word size of the processor. Indeed, programs rightfully expect an aligned memory area, and the offsets in this area are chosen with this in mind. Luckily, compilers usually add filling bytes to structures to ensure that they are an integral multiple of the word size. The allocated memory is thus usually aligned.

This technique cannot detect all overflows. For instance, if the block of memory allocated corresponds to a structure, which in turn contains an array, this array can overflow without detection if the memory block does not overflow. Another disadvantage of this technique is that it requires much memory. Even when only a few bytes are requested by the program, two whole pages are used, one for data, the other as guard.

This technique is implemented in the library Electric Fence [Per93].

Replacing gets() and its friends This section is based on [Ale01, TS01b, Lib01, Sna97, Sna00, TS01a, Whe02b].

This section covers many functions of the C library. The goal is to avoid the worst possible consequences for functions not taking the size of a buffer as parameter. Among these functions, there is `gets()`, `strcpy()`, `sprintf()`, `sscanf()`, and their variations.

More precisely, we want to protect return addresses and saved frame pointers against overwriting. We saw in Section 3 that these addresses are often used as

a target in attacks exploiting buffer overflows. On the stack, the saved frame pointers form a linked list that can be used to identify the location of return addresses. They mark the limit between local variables and the parameters of a function. No array can extend over these values. It is thus possible at run time to check that these values are not crossed in an operation filling an array. While it does not avoid overflows, many attacks are stopped.

This technique is implemented under Linux in a library called libsafe [TS01b] which can be linked dynamically to programs. These programs do not have to be recompiled to benefit from the protection of this library. Under FreeBSD, the library libparanoia [Sna00] offers a similar protection.

This technique could also be used for functions to which the size of a buffer is passed to validate if it is plausible, but usually we trust a program that bothers passing the size of buffers.

This approach has its limits. First, it does not detect overflows in a general way, it only protects some special values on the stack. We saw in Section 3 that other values could be used as targets in an attack. Moreover, this technique does not offer any protection if a program is compiled without using frame pointers (`-fomit-frame-pointer`). It cannot do anything against overflows on the heap. In addition, the alternate libraries are not compatible with compilers using a different stack frame format, such as StackGuard.

4.3.4 Stack Growing Towards Higher Addresses

We saw that with most operating systems and most processors, the stack grows towards lower addresses. When the stack grows towards higher addresses, it is impossible to exploit a buffer overflow at the top of the stack since there is no data after this buffer. There are processors on which the stack grows natively towards higher addresses, and others supporting a stack growing in either direction. For example, VAX computers had a stack growing towards higher addresses [KS02].

Even if the stack grows towards higher addresses, it is still possible to exploit a buffer overflow that is not at the top of the stack. In particular, a buffer defined among the local variables of the caller of the current function can overflow and overwrite the return address of the current function. Figure 12 shows what happens.

Although a stack growing towards higher addresses prevents the exploitation of some overflows, its efficiency is limited. Moreover, implementing a stack growing towards higher addresses on a processor not supporting it natively would not be as fast. For these reasons, this approach is rarely used.

4.3.5 Protecting All Pointers from Overwrite

In [CBJW03], the authors explain that it is possible to protect all the pointers of a program, including return addresses, by encrypting them. Encryption is simply XOR with a secret key randomly chosen at the start of the program. Pointers are decrypted only when they are loaded into a register of the CPU.

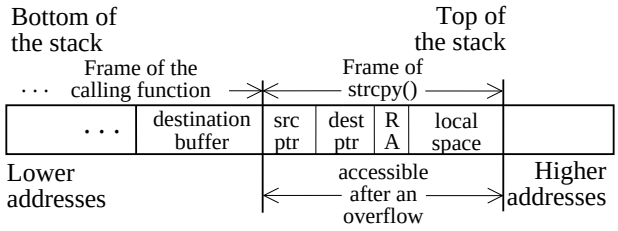

Figure 12: Overflow on a stack growing towards higher addresses. A function called `strcpy()` and passed it a pointer to a local buffer. If there is an overflow, the return address can be overwritten, as we saw in Section 3.3.1. The attack happens "faster" than with a stack growing towards lower addresses since the return to the attacker's code is made as soon as `strcpy()` returns.

When it is in a register, a pointer is safe since registers cannot be overwritten by a buffer overflow. A pointer can be overwritten in main memory, but since it will be decrypted before it is used, the attacker cannot control to which address it will point. Thus, this protection technique does not stop buffer overflows, but it tries to prevent their exploitation.

In [PL02], the main idea is the same, but it is applied only to function pointers, thus data pointers are not protected.

4.3.6 Run-Time Bounds Checking

The C and C++ languages allow the use of pointers in contexts completely disconnected from the declarations to which they refer. It renders run-time bounds checking very difficult for a compiler. For instance, a function can take as input a pointer to an integer. At run time, this integer can be a single variable or an element of an array. In the former case, arithmetic on this pointer leads to an undefined result, but in the latter case, the result is well defined as long as the pointer stays inside the referred array.

However that may be, the specification of these languages does not make it completely impossible to do bounds checking. It describes behaviors that have a well-defined result and others that have an undefined one. A compiler can thus be more or less restrictive on operations allowed on an array at run time.

Tracking buffers and checking arguments to library functions In [LC02], the authors explain a way to verify that library functions work on buffers that are large enough for the work they are requested to do. For this, the compiler must be modified so that information about static and automatic buffers is available at run time. Usually, this information, which is available during compilation, is kept only for debugging purpose and it cannot be seen by the application. Information about dynamically allocated memory is also made available by intercepting calls to `malloc()` and `free()`.

Calls to functions of the C library that are considered dangerous are also intercepted. With all the information from the compiler and from dynamic memory allocations, it is possible to check if a buffer given as parameter will overflow or not. In case of an overflow, the program can be aborted. If no overflow is possible, the real function is called. This approach can only detect overflows that would happen in library functions.

Tracking buffers and looking for possible overflows In [HB03], the authors present an approach that has a lot in common with the previous one. Instead of targeting the deployed version of the program, they target versions built for testing. They define *interesting functions* as the ones that can produce a buffer overflow. The main difference with the previous approach is that when an interesting function is called, the size of the destination buffer is compared to the size of the source buffer instead of the size of actual data. If it is smaller, a warning is logged even if there is no overflow for this particular call.

With this technique, there is no need to try triggering buffer overflows during testing, the important thing is that each interesting function gets called from all of its call sites in the program. This is called interesting function coverage. The drawback is that there can be many false positives.

Using bounded pointers This section is based on [ABS94, Jon95, JK97, McG98a, McG98b, McG99].

Pointers in C and C++ are usually represented by the address to which they point. This representation allows for the best possible performance since it can be used directly by the processor. However, it is not the only representation allowed by the language. Compilers can use any possible representation, as long as it does not violate the language specification.

In particular, a pointer can be represented by a tuple $<address, base, limit>$. Here, *address* is the address of the pointed variable, *base* is the address of the first element of the array and *limit* is the address of the end of the array, after the last element. This is called a bounded (or fat) pointer. If there is a pointer to a variable of type T, which is not part of an array, we have:

$$address = base = limit - \texttt{sizeof}(T)$$

When the address of an element of an array is initially generated, the compiler can always identify the beginning and the end of the array. Thus, it can also give the right value to the three fields of the pointer. Thereafter, no matter if the pointer is passed as parameter or returned by a function, the base and the limit of the array follow the pointer. When arithmetic operations are computed on a pointer, the compiler can check efficiently if the pointer respects the bounds of the object to which it points.

However, this technique has some disadvantages. First, many programs assume it is possible to put a pointer in an `int` variable. When pointers are represented by a tuple of 3 addresses, it is obviously not the case. Nevertheless,

these programs do not respect the language specification, and thus it is sensible to expect that they will not benefit from bounds checking.

Another disadvantage to the change of pointer representation is related to compatibility. If two compilers simply use the address as the pointer representation, the code generated by them can be linked together without problem. If they use a different representation, the code cannot be linked. In this case, an adaptation layer must be used to convert one representation to the other. The advantages of bounded pointers are lost. Moreover, passing as parameter a pointer to a pointer might not even work correctly. Libraries available only as binary code are thus more difficult to use. It is easier when the source code is available since they can be rebuilt.

The use of bounded pointers also has impacts on performance. This is understandable since the pointers are three times larger and bounds checking also takes a significant amount of time. The performance penalty is about 50% for well-optimized implementations of bounds checking using bounded pointers, and up to 200% otherwise.

There is an extension to GCC implementing this approach [McG00]. Another implementation stores yet more information in the pointer in order to detect all possible spatial and temporal memory access errors [ABS94]. Yet another implementation [XDS04] tries to be more efficient. It adds information about run-time types but it introduces some optimizations to disable bounded pointers when it is safe. However, it cannot handle all C programs.

Encoding integers and bounded pointers In [OSSY02], the authors explain how they changed the representation of integers and bounded pointers in order to allow casts between integers and pointers without losing information. Information about run-time types is kept and encoded pointers have a flag that indicates if their static type is the same as the run-time type of the memory they point to. The notion of virtual offset is introduced to improve compatibility with code that assumes that pointers and integers are 32-bit wide. A garbage collector is used to make sure no dangling pointer capture another object at the same location.

Decoupling bounds checking In [PF97], the authors suggest a way to decrease the cost of run-time bounds checking. Their idea is to use a second processor, which would be idle otherwise. To that end, a second, customized version of the program is created. Some kind of bounded pointer is used, but everything that is not related to bounds checking is removed from the program.

At run time, there are two processes for a program, the main process does the real work and the shadow process runs the customized version of the program to detect overflows. For events that cannot be reproduced such as user input, the shadow process must be informed of the result by the main process. It is important to note that this technique does not prevent buffer overflows, and does not stop attacks. It merely detects overflows.

Let us consider the following C code (which is correct).

```
int i[10];
int *p = i;
int *q;
memcpy(&q, &p, sizeof(int *));
p++;
```

If pointers are represented by descriptors, the result is incorrect because both pointers use the same pointer descriptor and both are incremented when only p should be.

Table 14: Problem with the representation of pointers with descriptors

Representing pointers using descriptors If the size of pointers cannot be changed, it is possible to represent them with a descriptor which is the index of an entry in a pointer table. This table can then contain bounded pointers, as they are defined in Section 4.3.6.

When a pointer is modified, for instance by incrementing it, its representation is unaltered, only the entry in the table is changed. Two different pointers cannot share the same descriptor. If it were allowed, incrementing one of them would also increment the other one, something we do not want. When a pointer is copied, its entry in the descriptor table is thus copied to the entry of the destination variable.

This approach has an important problem. The C language allows to copy pointers in many ways, and compilers cannot intercept them all. For the code presented in Table 14, the compiler cannot produce a correct program if pointers are represented by descriptors.

The use of `memcpy()` and all other functions that copy memory must thus be banned for copying a pointer. The C++ language also has this kind of restriction for copying objects (`struct` or `class`). This is because objects can have a complex copy semantic defined by the copy operator and the copy constructor. Pointers represented by descriptors are thus more like C++ objects than a primitive data type.

This approach for bounds checking is usually rejected because it is not compatible with a significant proportion of correct programs.

Tracking allocated memory areas and pointers at run time This section is mostly based on [Jon95, JK97].

A different approach to bounds checking is possible without modifying the representation of pointers. For this, it is necessary to keep, at run time, the definition of all allocated objects (memory areas). They are indexed in such a

way that it is fast to find the object of which an address is part of. An object can be static, dynamically allocated (`malloc()`) or on the stack. It corresponds to a declaration. For instance, if one declares an array of structures, which in turn contain arrays, then all of this is part of the same object. It is not possible to divide the area in many objects without risking associating many possible objects to one address.

Each operation on a pointer is modified to check if, after its application, the pointer is still in the range of the original object. For example, before incrementing a pointer, the object to which it points is noted, and after the incrementation a check is performed to ensure the new pointer has not passed the end of this object. If it has, it receives the special value `ILLEGAL`, which is different from the `NULL` pointer. This value is not strictly necessary and the program could be stopped as soon as it generates an illegal pointer. However, it seems that many programs generate illegal pointers without ever using them. The special value `ILLEGAL` thus allows stopping a program only when it uses such a pointer.

For many operations using two pointers, both must point to locations that are part of the same object, else the operation has no well-defined meaning. This is the case for operators `-`, `<`, `<=`, `>` and `>=`. The compiler thus adds this check and the program is also stopped if the pointers point to different objects.

The C language allows obtaining a pointer on the element immediately following the end of an array, even if it does not allow it to be dereferenced. This can be used to check a termination condition of a loop. These pointers thus cannot be represented by the value `ILLEGAL`. To avoid any ambiguity between a pointer to the element following the end of an array and the first of the following array, and to keep the accuracy of the checks, it is necessary to keep at least one unallocated byte of memory between objects.

Before dereferencing a pointer, the compiler always checks that the value is not `ILLEGAL` and that it is not after the end of the object.

Tracking objects at run time thus requires modifications to the allocation of memory at many places. Two aspects must be taken into account. All objects must be added to the global list of allocated objects when they are allocated, and they must be removed when they are unallocated. In addition, there must be at least one byte separating objects. This is easily done for dynamic allocations by modifying `malloc()` and `free()`. For static allocation, the compiler must be modified. For local variables, the most difficult is to correctly handle the scope of variables. They must be added and removed from the list of objects correctly, even when there are `gotos`. This is very similar to the handling of constructors and destructors in C++.

It is more difficult to introduce an unallocated byte between function parameters without breaking the calling convention. Luckily, the case most susceptible to lead to pointer arithmetic is when passing by value a structure containing an array. This is not very common.

To handle parameters correctly, without modifying calling conventions, it is possible to make a copy of local variables and use this copy instead of the parameter directly. This is the method described in [EY00] to protect against some

```
// The compiler ‘‘sees’’ an object for s1
struct {
        char c[4];
} s1[10];

// and another one for s2
struct {
        char c[4];
        int i;
} s2;

int main()
{
        // The accesses outside the bounds are not detected
        // because they are still inside the right ‘‘object’’
        s1[0].c[20] = 1;
        s2.c[4] = 2;
        return 0;
}
```

Table 15: Object tracking at run time may not be enough for correct array bounds checking.

buffer overflows. Obviously, this does not work when the number of arguments accepted by a function is variable.

This approach to do array bounds checking has its limits. In fact, it does not do array bounds checking, but object bounds checking. The program of Table 15 exhibits cases where bounds checking is not done correctly.

This approach also considerably reduces the performance of programs. For example, it has been implemented as an extension to GCC. Depending on the program executed, the execution time can be multiplied by a factor of up to 50.

In [RL04], the authors present another implementation of this approach. To improve performance, they offer the option of checking only pointers to characters. They believe security is not compromised by this. To improve compatibility with existing code, they introduce out-of-bounds objects to replace the value `ILLEGAL`. These objects allow tracking pointers that go outside the bounds of the object they refer to.

Segmentation This section is based on [Int00, Jon95, KS02].

Intel processors of the IA-32 family offer a segmentation mechanism allowing a fine division of data. Linux uses a memory model in which the application does not care about segments. In particular, code and data segments usually

cover the whole address space. Nonetheless, it would be possible to define a segment for each memory area. This way, results very similar to the ones of Section 4.3.6 could be achieved, but with much better performances using help from the processor.

Defining memory areas using segments requires modifications to the compiler. The operating system must also intervene since modifying a segment descriptor is a privileged operation. Contrary to the tracking of objects at run time, segmentation does not force keeping unused space between memory areas. However, the pointer representation must be modified to include a 16-bit segment selector in addition to a 32-bit offset. Pointers must thus be made larger. Another drawback of this approach is that it does not allow the use of more than 16 384 segments at a time. For programs handling a large amount of data, it is probably not enough. Thus, it would be important to reserve a global descriptor for areas that cannot be represented by their own descriptor while also allocating available descriptors "cleverly".

If this technique does not permit the detection of all buffer overflows, it ensures they are contained inside their area.

Keeping a memory map Another approach to do array bounds checking consists in keeping a map of allocated memory. It is described in [Gin98, HJ91, Jon95]. For each byte of memory used in a program, there is a corresponding bit indicating if it is allocated or not. As is the case in Section 4.3.6, memory allocation must be modified in the compiler and in `malloc()` and `free()` to keep the memory map up to date. It is also possible to keep free memory blocks between allocated areas to better detect overflows. They are called red-zones. In particular, some parts of the stack, such as return addresses and saved frame pointers, are considered red-zones because they are between the local variables of a function and it parameters. This allow detection of attacks overwriting the return address.

It is important to understand that this approach is less precise than the one of Section 4.3.6. With the latter, we are sure that pointers do not jump from one area to another one. Here, we only check that the pointers point to allocated memory. Each instruction accessing memory is thus modified to check the bit indicating if the memory is allocated or not.

This approach does not only allow the detection of unallocated memory, but also uninitialized memory. This requires a second bit indicating whether a byte is initialized or not. Thus, there are two bits for each byte, one to indicate if it can be read and the other to indicate if it can be written to. Valgrind implements this approach and can even detect the use of individual uninitialized bits by keeping 8 initialization bits for each byte.

It is also possible to use this approach directly on the machine code of a program when the source code is not available. The tools Purify and Valgrind do this by modifying or interpreting the machine code that has to be checked.

This approach thus has the advantage of working with almost all existing programs without any modification, but it does not permit the detection of all

buffer overflows. It also has a significant impact on the performance of programs. Purify slows down execution time by a factor between 2 and 5. This kind of tool is thus used mainly for debugging.

Checking unsafe pointers This section is based on [NMW02, YH03]. The idea is to add bounds checking at run time, but only when static analysis cannot prove that a memory access is safe. The two papers present different techniques to achieve this end.

In [NMW02] the authors explain how they used type inference to decide whether a run time check is required or not. They describe their type system and their type inference algorithm. They implemented it in CCured, a tool that is free software. CCured always ignores explicit memory deallocation and instead relies on a garbage collector to free memory.

In [YH03], the authors explain that their idea is to check at run time whether pointer dereferences always target appropriate locations or not. To minimize the overhead, some static analysis is done during compilation and no run-time check is added when it determines that a pointer is manipulated safely.

There are three steps to the static analysis. The first step is performing a flow-insensitive points-to analysis. The result of this analysis is used in the next two steps. The second step is identifying *unsafe pointers*. A pointer is considered unsafe when it may refer to invalid memory at run time, when it is dereferenced for writing and when it is passed as an argument to `free()`. The third step is identifying *tracked locations*: variables that may be, at some point during execution, an appropriate target of some unsafe pointer.

At run time, a mirror of the memory is used to tell whether some specific byte of memory is an appropriate target of some unsafe pointer. There is one bit in the mirror for every byte of memory. A location is marked as *appropriate* at the time of its allocation during execution. It is marked as *inappropriate* at the time of its deallocation. Before dereferencing an unsafe pointer or calling `free()`, a check is made to ensure that the pointer points to an appropriate location.

This approach as a lot in common with the one described in Section 4.3.6. The key difference is that, with the help of static analysis, it is possible to remove completely some run-time checks, and thus the overhead is smaller. Moreover, since relatively few locations are tracked at run time, a memory access error (i.e. pointing to the wrong location, whether appropriate or not) is more likely to be detected because locations that are not tracked are never marked as appropriate.

Another difference is that the technique described in this section does not check reads, only writes. For the tests that were done, the programs ran about 1.97 times slower on average when protected this way. This seems to be better than what CCured achieves.

4.4 Modifications to the Operating System

An operating system can help countering some of the attacks exploiting buffer overflows. It does so by changing the way it sets up virtual memory or the way

it handles special conditions.

4.4.1 Non-Executable Stack

This section is based on [Sol97b, Sol97a, Sol02, Woj98].

We already mentioned, for instance in Section 3.4.1, that it is possible to have a non-executable stack. This prevents attacks in which the code to execute is placed on the stack. It is important to note that the stack is the memory zone used most often by attackers. On UNIX operating systems, environment variables, arguments and all non-static local variables are on the stack.

The patch from the Openwall Project for Linux, which prevents execution of code on the stack, is very specific to processors of the IA-32 family. It works by breaking the symmetry in the definition of segments. Usually, under Linux, the code, the data and the stack segments (CS, DS and SS) all give access to the same memory. This patch changes the code segment so that it does not reach the stack, which is at the end of the address space.

However, the use of a non-executable stack has its drawbacks. Some legitimate programs place code on the stack to execute it. For example, when one uses an extension of GCC allowing definition of nested functions in C, the compiler must generate "trampolines" to allow the use of pointers to these nested functions. A trampoline is code that allows a nested function to get access to its context (local variables of the outer function). When a pointer to a nested function is generated, the trampoline, and thus the code, is placed on the stack. This mechanism does not work when the stack is not executable.

The patch is able to detect and emulate trampolines. A trampoline is detected when the instruction transferring control to code on the stack is a `call`, instead of a `ret`. If it is, a very limited number of instructions can be emulated in the trampoline.

Another problem caused by the non-executable stack exists in signal handling. When a signal is sent to a process, Linux places the code allowing returning to the main program on the stack. This does not work when the stack is non-executable. To solve this problem, the method used to return from a signal handler is modified, but this modification is not visible to applications, only to the kernel.

There are other programs that use the stack to place code. For instance, programs written in a functional language often use the stack to execute code. LISP and Objective C are two examples. Thus, there must be a mechanism to deactivate the protection against execution of code on the stack if we want to make these programs work correctly.

However that may be, preventing the execution of code on the stack does not prevent any buffer overflow. In the best case it prevents some attacks, but there are many ways to defeat this protection, as we have seen in Section 3.

4.4.2 Making Other Data Area Non-Executable

The material in this section is based on [Con99, dR03, Dup02, Mol03, Sta01, PaX00, PaX02, Woj01].

Arranging for data outside the stack to be non-executable is a bit more difficult. The stack is at the end of the address space of the program, and it is thus possible to shorten the code segment. However, data areas are usually placed between two code segments. This would not cause any problem if the processor allowed to mark each page individually as executable or not, but it was not the case for processors of the IA-32 family until recently [Mol04].

However that may be, three different techniques can be used to obtain similar results on older processors of the family. The first one consists in exploiting knowledge about the internal working of the processor. All processors supporting paging keep a small number of page entries in a cache memory to avoid accessing main memory on every access to a page. This cache memory is called translation look-aside buffer, or TLB. If the entry of a page is modified in main memory, the entry of the TLB remains unaltered.

For all processors of the IA-32 family and their clones, the processor has different TLB for code (ITLB) and data (DTLB). Although the two TLBs use the same source for page entries, it is possible to have one loaded while the other is not. The kernel of the operating system can also intercept the load of the ITLB or DTLB by marking a page not available or not accessible by the user. It can thus control precisely which pages are in the ITLB, and thus which ones can be executed.

This approach is implemented by PaX, an extension of Linux. It slows down a bit the execution of programs using many pages at a time, but usually, the performance loss is only 5% to 8%.

The second technique uses the segmentation mechanism instead of paging. This is the usual way to prevent execution of code on IA-32 processors. The problem is that Linux programs are generally not aware of segments. They use a memory model where the whole address space of a program, usually 3 gigabytes of virtual memory, is accessible from every segment without restriction. When using segmentation to prevent data from being executed, it is necessary to maintain the symmetry in the definition of the code and data segments so that existing programs can execute properly, that is to say, any page of physical memory must appear at the same offset in both segments. This means that a page requires two different definitions since it can be accessible as data, and not accessible as code. Defining two mappings for each page divides by two the virtual memory of a process. Thus, programs using large memory-mapped files can suffer from this technique.

Since this approach uses "real" functionalities of the processor to achieve its result, we can expect the performance to be better. The performance loss is indeed almost zero. It is implemented in kNoX and RSX.

In the third technique, the code segment is reduced in a way similar to that described for the stack in Section 4.4.1. It is much simpler that what PaX, kNoX or RSX do. However, for this to be effective, the executable code

has to be linked lower than data in the program address space. There is an implementation of this approach for Linux called exec-shield and another one in OpenBSD 3.4 called W$\hat{\text{X}}$. The OpenBSD implementation does more work than the Linux one to ensure that every data page is above the execution limit, and thus, it is more secure.

As for the stack, preventing execution of code in other data areas does not prevent any buffer overflow. It blocks some attacks, but it is possible to bypass this protection [Woj01].

4.4.3 Changing the Location of Libraries

This section is based on [Sol97a, Woj98, Woj01].

Many attacks use the fact that a library is always at the same location in memory. For example, the C library contains almost all the code an attacker could want to execute. However, to use this code, he has to give control to the exact location. Under Linux, it is possible to load a shared library anywhere in memory since its code is position independent.

Null byte in the address The presence of a null byte in the address of a library renders more difficult the attacks using the code of this library, such as the one described in Section 3.4.1. Indeed, for this kind of attack, the address of the function to call must be followed by its parameters on the stack. If the return address contains a null character and it is copied using a function such as `strcpy()`, the copy stops as soon as the null character is encountered and it is not possible to give arbitrary parameters to the function.

This protection measure was implemented for the first time in the patch from the Openwall Project, which prevents the execution of code on the stack. It is also present in kNoX, a patch also preventing execution of other modifiable data. The main program cannot be protected in this way because its code is not position independent.

Random address Another manner to protect against these attacks is to load the library at a random location. This approach is implemented in PaX. As for addresses with null bytes, the main program cannot be moved. It is important to note that there may be some ways for an attacker to discover at run time the address of a library [Woj01].

4.4.4 Modifying the Stack Location

Many attacks need the exact or approximate address of some variables or buffer on the stack. By randomly choosing the address at which the stack starts, it is possible to defeat many of these attacks. This approach is explained in [Ket98] with an example showing how this protection measure can be placed directly in the code of a program. PaX implements this feature in the Linux kernel to protect all programs without modification or recompilation [Woj01].

However, it does not offer an absolute protection since an attacker can try every possibility until he finds the right one. There are also other kinds of attacks that do not require the address of a variable on the stack.

4.4.5 Protecting Return Addresses on Sparc Processors

The techniques presented here have a lot in common with the ones of Section 4.3.1. An important difference is that they are implementated in the operating system instead of the compiler. They are based on [FS01].

Sun Microsystem's Sparc processor architecture implements function call differently than most other processor architectures. From the application point of view, each function sees 24 registers (the register window) of what appears to be an infinit number of registers available for functions calls (input parameters, local variables and output parameters). When a function is called the register window is shifted so that output parameters become input parameters, and new registers are available for output parameters and local variables. When a function returns, the register window is shifted back so that the calling function sees its own input registers, output registers and local variables again.

Obviously, the number of register is not infinit. It is in fact quite limited. The operating system kernel must save registers on the stack in memory if there is no register available for a function call. It must also load them back from memory on a function return if the registers of the calling function are no longer available. It is thus possible to modify the kernel so that it encrypts registers (XOR) with a per-process secret key before saving them. Then they are decrypted when loaded back.

Another possibility suggested is the implementation of a return address hash table. Most registers are saved directly on the stack, but return addresses receive a special treatment. They are saved in a hash table (indexed by the frame address) together with a random number, which takes the place of return addresses among saved registers. This allow better detection of attacks.

4.4.6 Protecting Return Addresses on Other Processors

For other processor architectures, which are not designed to allow the operating system to protect return addresses, it is possible, at least in theory, to modify them expressly to offer such protection without changing applications. This has been suggested three times [MKSL03, PL04, XKPI02].

The idea is that the processor can keep a copy of return addresses in memory that is not accessible to applications. It is called the secure return address stack (SRAS). When a function returns, if the address in the main stack differs from the address of the SRAS, the processor raises an exception and the operating system can log an attack attempt.

With this approach, the compiler might have to be modified in order to handle some kinds of non-local flow control. For example, exceptions and and `longjmp()` require special considerations.

4.5 Run-time Checks

This section describes other dynamic techniques that do not prevent overflow vulnerabilities but render them more difficult to exploit.

4.5.1 Principle of Least Privilege

It is possible to limit the consequences of possible buffer overflow vulnerability by executing a program that presents some risk with reduced privileges. For this, an unprivileged account of the operating system is often used to execute the program, particularly when it offers services on the Internet. For example, the Apache web server is often executed under an unprivileged account such as nobody.

This approach is not a general solution to the problems caused by buffer overflow vulnerabilities because it does not avoid them at all. It only prevents them from making too much damage. Moreover, it is of no use for programs that, by their nature, require some privileges to do their task.

Privilege elevation In [Pro03], the author explains how it is possible to elevate an application's privilege only for some system calls using a technique that will be described in Section 4.5.3. This can be used to apply the principle of least privilege.

4.5.2 Looking for Executable Code

In [TK02], the authors present an approach designed to protect applications offering services on the Internet, for example, web servers. Their idea is to look for executable code among data received from a client. It is considered normal to find some machine instruction in random data, but if the number of consecutive executable instructions is unusually high, it could be the code of an attack exploiting a buffer overflow. When it happens, the service can stop processing the request. This technique is called abstract payload execution.

4.5.3 Monitoring System Calls

Another approach consists in scrutinizing the actions of a program and intervening, for instance by stopping it, when it deviates from what it is supposed to do. The idea is that an attacker cannot do most of the malicious actions without making a system call[6]. Indeed, whether it is for reading or writing a file, establishing or using a network connection or executing a program, the operating system must intervene.

Protecting a program with this approach is achieved in two phases. In the first one, a policy for the program is built. This policy represents the system calls that are allowed for a program. The second one happens at run time. The trace of the program is checked in real time for conformance to the policy.

[6] A possible exception to this is the production of a denial of service by executing an infinite loop not containing any system call.

For example, Janus [GWTB96], Subterfugue [Sub02] and syscalltrack [Sys02] allow monitoring system calls and they make it possible to disallow some of them. These tools provide a mechanism for monitoring programs, but no policy telling whether a system call is acceptable or not. The next few sections discuss different approaches to build such a policy.

4.5.4 Obtaining Allowed System Calls from Test Runs

A simple technique to generate a base policy is to monitor a program in order to obtain the set of system calls it generates. This is suggested in [Pro03]. The policy can then be refined by the user.

4.5.5 Creating a Specification Manually

In [SU99], the authors propose a language to define a specification of programs, which they call regular expressions for events (REE). It deals with system calls as well as their arguments and it incorporates variables. It is much more expressive than regular expressions. This approach can allow a system call in a specific context, but not in other contexts, someting that can improve security.

4.5.6 Automatic Learning of a Finite-State Automaton

In [SBDB01], the authors describe an automatic learning algorithm that generate a finite-state automaton (FSA) from system call traces efficiently. The FSA can then be used to monitor a program. Since the learning process cannot ensure the FSA is perfect, the monitor must implement some technique to ignore isolated anomalies, otherwise the false-positive rate could be too high.

4.5.7 Creating a Model with Static Analysis

The ideas of this section are described in detail in [Wag00].

Instead of building a policy manually, empirically or from an external specification, it is proposed to build a model for the program automatically from its source code The source code of a program is statically analyzed to find out what system calls it should do at run time.

The model can be expressed in many different ways. It may represent a program very roughly, very precisely, or anything in between.

A trivial model A very simple model for a program could be the set of all system calls it can do. It has the advantage of being easy to check. For each system call done at run time, we can check if it is part of the set of authorized system calls. If it is not, the program can be stopped immediately. However, this type of model is not very precise because it does not take into account the ordering of system calls.

The callgraph model A more precise model can be built by considering the control flow graph of the program. This graph indicates for each instruction (node) of the program all its possible successors including those inside a function that is possibly called. Some instructions correspond to a system call and others do not. An instruction can have many successors, for example when there is an `if` or a loop. A return from a function also has one successor for each place from which the function is called.

From this graph, it is possible to build a nondeterministic finite automaton where the alphabet is the set of system calls. This automaton accepts a sequence of system calls if and only if there exists a path of the graph such that the sequence of nodes corresponding to a system call matches the sequence of system calls. We ignore nodes that do not make a system call.

When executing the program, we check if the sequence of system calls executed by the program is accepted by the automaton. It is not necessarily the case if the program is maliciously exploited. To do this verification, there is no need to wait for the complete sequence. The automaton can be simulated while the program runs and the program can be stopped as soon as the automaton reaches state Wrong.

This model never generates a false positive, that is to say, it never gives an indication that the program respects the model when it does not. The automaton is built in such a way that all possible execution traces of the program are covered. On the other hand, it also accepts many impossible traces. Indeed, after a function call, it permits a return to a different location from the one where the call originates. An attack that would follow one of these paths would not be detected.

Abstract stack model It is possible to use a nondeterministic pushdown automaton to obtain a more precise model. To that end, the stack of a pushdown automaton is used to ensure that, when a function ends, it returns to a point that matches the corresponding function call.

Even though it allows exploration of execution paths that do not match the real one, we have the assurance that the real one is always among possible paths. Thus, there can be no false positive.

This model, while much more precise than the one built with a finite automaton, is not perfect in the sense that it does not allow the detection of all intrusions. A meticulous attacker could build an attack that makes system calls in an order allowed by the automaton. Simulating a pushdown automaton is also much less efficient that a finite automaton. Depending on the program checked using this model, the overhead can be acceptable or not. For some programs, this model adds many minutes and even hours to the running time of the program that is checked for a single interactive action of the user. For these, a more efficient model must be used.

Directed graph model Another possible model can be built by considering, among all possible sequences of system calls, only windows of length k. A

directed graph can thus be built with nodes representing sequences of length $k-1$, and edges $(s_1 s_2 \dots s_{k-1}, s_2 s_3 \dots s_k)$ indicating that the program has at least one trace containing the sequence $s_1 s_2 \dots s_k$.

Checking a program using this kind of model is very efficient, but the problem is building the directed graph. The technique used by David Wagner is so inefficient that it prevented experimenting with sequences of more than two system calls. However that may be, this model can be interesting, particularly if the window could be enlarged.

4.5.8 Monitoring Control Flow Transfers

In [KBA02], the authors propose to monitor branches in programs instead of system calls, they call it program shepherding. It cannot prevent buffer overflows, but it can stop attacks that try to exploit them by changing control flow transfers. This can be done by interpreting a program instead of executing it natively. It is then possible to reduce overhead to a minimum by using a dynamic optimizer. This approach is implemented in a product called SecureCore.

4.6 Static Analysis

Static methods offer many advantages over dynamic ones. They allow obtaining results that are true for all possible executions of a program. However, checking whether a program can overflow its buffers is not only difficult, it is undecidable for the general case. Thus, there will always be false positives or false negatives for any program attempting to tell whether an arbitrary program can overflow a buffer.

There are many approaches using static analysis. Some of them only detect the overflows done by some functions, others detect overflows anywhere in the code. Some do their analysis one function at a time, others analyze a complete program. They can be distinguished by whether they produce false positives, false negatives or both, as well as the frequency of false diagnostics.

4.6.1 Lexical Analysis

Lexical analysis is not very powerful, it only recognizes tokens in the source code of a program. Analyzers using such an approach are looking for a sequence of tokens that can pose problem. For example, an analyzer can check whether there is a call to `strcpy()` and flag it unconditionally as a potential buffer overflow. It does not try to see if this function is used safely or not, it only indicates to the programmer that there may be a vulnerability and it incites him to check if the code is correct.

Tools using these techniques have the advantage of being fast because the analysis does not require many calculations. They usually give many false positives, that is to say they give indications that an overflow is possible when it is not. Also, they can never ensure that no overflow is possible. For example, they consider some functions as secure and they do not warn when they are not used

correctly. In addition, they do not detect an overflow caused by a bad condition in a loop.

The simplest tool to do lexical analysis is grep. It allows finding the lines matching a regular expression in a text file. For example, the command `grep -n strcpy test.c` finds all the lines of the file `test.c` that call the function `strcpy()`, but also those having a comment or a string containing "strcpy". Thus it is not very precise and there are many false positives. This is why there are tools trying to better filter the code most susceptible to cause problems.

Flawfinder, RATS, and ITS4 are software tools performing a more powerful lexical analysis to increase the precision of the results. Flawfinder [Whe02a] can detect potential buffer overflow problems, format string vulnerabilities, and race conditions in C and C++ programs. RATS too can detect potential buffer overflow problems, format string vulnerabilities, and race conditions in C, C++, Perl, PHP and Python programs. ITS4 [VBKM00] can detect potential buffer overflow problems, format string vulnerabilities, race conditions, and bad uses of functions generating pseudo-random numbers in C and C++ programs.

4.6.2 Annotation-Assisted Lightweight Static Checking

Evans and Larochelle [EL02, LE01] explain a method allowing, among other things, the detection of buffer overflows in C programs by using some kind of lightweight static checking. Their method avoids interprocedural analysis with the use of annotations, also called semantic comments, added by the programmer to the prototype of the functions. These annotations form a kind of contract for the behavior of a function. It is thus possible to check if the contract is respected by any party independently of others.

To check if a function can cause an overflow, we assume the values received as input respect the pre-conditions expressed in the annotations. Under these conditions, it is often possible to verify that the code of the function ensures the requirements that the post-conditions are met at the end of the function. If it does not, the programmer is informed.

Where there is a function call, we verify that the parameters passed to the function respect the pre-conditions and we assume that, after this point, the post-conditions of the annotations are met. If a pre-condition is not satisfied for a function call, the programmer is warned. Since the prototypes are equally visible for the compilation of a function and for the compilation of all modules calling it, we have the assurance that the analysis is correct, unless the annotations are modified between the compilation of a function and the compilation of its callers.

The tool doing this kind of analysis is called Splint and it is free software. Its analysis is not limited to buffer overflows. It also recognizes annotations to check the allocation and the deallocation of memory, format string vulnerabilities, and the use of null pointers.

The main annotations that allow buffer overflow detection are the operators `maxSet` and `maxRead`. `maxSet(t)` represents the maximal index of the array `t` for which a value can be assigned, while `maxRead(t)` represents the maximal

```
void strcpy (char *s1, char *s2)
     /*@requires maxSet(s1) >= maxRead(s2) @*/
     /*@ensures maxRead(s1) == maxRead (s2) @*/;

char * fgets (char *s, int n, FILE *stream)
     /*@requires maxSet(s) >= (n -1); @*/
     /*@ensures maxRead(s) <= (n -1) /\ maxRead(s) >= 0; @*/;
```

Table 16: Some of the annotations used by Splint. Only some of the annotations concerning buffer overflows are present to show the general principle. Operator `/\` represents conjunction.

index of array `t` that can be read. When used in a `requires` clause, they express pre-conditions, while in an `ensures` clause, they express post-conditions. It is also possible to use operators `minSet` and `minRead` that represent the minimal index for an array, for assignment and for reading respectively. Table 16 shows some annotations present by default in Splint.

To obtain useful results from Splint, it is not necessary to annotate all functions. First, Splint has annotated declarations for the standard C library. It is thus possible to find many overflows without adding any annotation. In addition, when there is no annotation, Splint uses default pre-conditions and post-conditions. These default annotations are defined so that they match as many functions as possible. It is thus possible to annotate only the functions for which default annotations are not right. Moreover, since Splint does a lightweight static analysis, and thus runs fast, we can run it once, modify some annotations, run it again and repeat the procedure as many times as required, the same way compilation errors are corrected with the help of the compiler.

The analysis done by Splint is neither sound nor complete. Indeed, it can cause false positives and false negatives. Sometimes, the analysis is not sufficient to determine that some code respects all the conditions. The analysis considers the control flow but only minimally. In addition, it cannot always check that the result of an expression is in a given range.

Other times, certain violations of the conditions imposed by the annotations can escape the analysis because of some simplifications. The analysis is done one instruction at a time and it does not consider that the value of a variable can change during an instruction. To analyze a loop, some heuristics are used to determine the number of times it can execute, but they do not ensure the result is correct. The analysis assumes that a loop causing a buffer overflow will show this behavior in its first or in its last iteration. Most often this is the case, but it is possible to build a loop overflowing a buffer while not making it apparent in the first and in the last iteration.

Splint never assures there is no overflow, but for most real programs, it

```
char s[20], *p, t[10];
strcpy(s, "Hello");
p = s + 5;
strcpy(p, " world!");
strcpy(t, s);
```

Table 17: The overflow of buffer `t` is not detected by BOON because the string `s` is modified by another pointer.

detects almost all of them. On the other hand, it gives many false positives that must be checked manually by the programmer. However, it is useful. For instance, for the program WU-FTPD, the addition of 22 annotations is enough to convince Splint that 92% of the 225 function calls usually considered dangerous are correct.

4.6.3 Abstract Data Type and Integer Range Analysis

In [WFBA00, Wag00], the authors consider strings as an abstract data type manipulated by the functions of the C library. The strings are modeled as a pair of integers indicating the allocated size of a string and its real length. For each manipulation of a string, one or more constraints are associated to these two integers.

The time spent for solving the constraint system with this algorithm is proportional to the number of constraints for cases that were observed. The result is, for each string, a range of possible values for the memory allocated and another one for the size of the string. To be sure there is no overflow, the upper bound of the range for the size must be smaller than or equal to the lower bound of the range for allocated memory.

The tool implementing this technique is called BOON. It is written in ML and only parts of it is free software. It has some important limitations. First, it can only detect overflows on strings, and not for arbitrary arrays. Moreover, only manipulations of strings by the functions of the C library are considered and not those modifying a string as a character array or using pointer dereferences. Another problem is the function `strncpy()` that is not modeled correctly. BOON does not handle the case of a result string not terminated by a null character.

The handling of pointers also poses problems because it is much simplified in BOON. For instance, BOON largely ignores the fact that two pointers can refer to the same string at the same time. It also totally ignores doubly indirected pointers, arrays of pointers, function pointers and `union`. All these deficiencies prevent the detection of some buffer overflows. Table 17 exhibits a buffer overflow that would not be detected by BOON.

The analysis done to generate constraints has the advantage of being fast,

but it lacks precision. This is why the number of false positives is rather high with BOON. First, the analysis does not take control flow into account, that is, the order of instructions and control structures is ignored. Second, this tool considers there is only one instance of each variable composing a structure instead of one for each instance of the structure. This can greatly enlarge the range of possible values seen by BOON as opposed to the real values it can take. However, this principle has the advantage of considering that two pointers to a structure can modify the same variable.

BOON has been used to find previously unknown overflows in some software used under Linux. Even though the number of false positives is high, it is about 15 times smaller than that obtained with the simple use of grep.

Improving static analysis In [GJC+03], the authors discuss their improvements to the analysis described in the previous section. In particular, they implement a more precise pointer analysis. They also achieve context-sensitive analysis using two techniques. The first one is by inlining constraints, but it cannot work with recursive function calls. The second one is by generating summary constraints for functions. Their approach to solve constraint is also different, it is based on linear programming. Even with these improvements, the rate of false-positives is still high.

4.6.4 Catching All String Errors with Integer Analysis

The approach presented in [DRS03] can detect all string manipulation errors. It reduces the problem of checking string manipulations to that of checking integer manipulations. It can use annotations, but it can never miss an error because a contract is too weak or too strong. In the worst case, there will be more false positives. When no contract is given, the tool uses an algorithm to compute an approximation of the strongest postcondition and the weakest liberal precondition.

CSSV must perform some rather precise static analysis to implement all this. It is important to know which pointers may be or must be aliases, else the number of false positives would be too high. CSSV is not publicly available. The results presented show that, in some cases, the time and memory required for the analysis and the number of false positives can be relatively high. However, it is difficult to tell how representative these results are, because they represent less than 1000 lines of code.

4.6.5 Reducing False Positives

The approach presented in [XCE03] allows checking millions of lines of existing code because it is fast and it is tuned to suppress common sets of false positives. It can use annotations but it does not require them because of the way the analysis is performed and because it uses statistical belief analysis to infer parts of them.

The analysis is interprocedural and path-sensitive, since otherwise it would not give precise enough results. It also needs to be aware of pointer aliases for buffers (and their offsets) to ensure that a certain class of errors is detected. First, the callgraph of the program to analyse is built. Functions are analysed in bottom-up order and cycles are broken in such a way that the function with the least number of callees is analysed first. When a function is analysed, its memory access constraints are summarized so that they can be checked when a call to it is encountered later.

The analysis of a function ends when all possible execution paths are analysed or after a pre-determined time limit, which defaults to 5 seconds. Obviously, there are some heuristics to handle loops.

The solver tracks as many relations as possible between variables. It is not the most powerful one, but it can be queried and updated in constant time. In general, it reports an error only when it can demonstrate that a memory access is unsafe.

This approach was implemented in a tool called ARCHER, which is not publicly available, and 2.6 millions lines of code in four large open-source programs were analysed. It found 160 potential errors and there were only 55 false positives. The analysis ran for about 6 hours.

4.6.6 Hybrid pointer Alias Analysis and IPSSA

In [LL03], the authors present another approach. Their goal is to catch errors that arise from inconsistencies around procedure boundaries and along exceptional control flow paths without producing too many false alarms. It uses a hybrid pointer alias analysis. The first component of this analysis is precise and it is path and context sensitive. It is used to track locations accessed from parameters and local variables by *simple paths*, that is to say, without iterated dereference or field access. The second component is more efficient but less precise because it is flow and context insensitive. It is used to track all other locations.

This approach uses an unsound assumption, which is that pointers passed into a procedure and locations that can be accessed by applying simple paths to these pointers are all distinct from each other. It has the dual benefit of speeding up the analysis and suppressing many false positives. It also matches well with how most programs are written. The analysis reports a warning when it concludes that this assumption is definitely violated.

Once the analysis is complete, an abstract representation of the program is generated. This representation is called IPSSA. It captures intraprocedural and inter-procedural definition-use chains for both directly and indirectly accessed memory locations. A tool that finds buffer overflows and format string vulnerabilities from this representation has been implemented. It found 14 security vulnerabilities in ten applications programs while producing only one false positive. The IPSSA representation could also be used to implement, in less time, a tool to detect different kinds of bugs in C programs.

4.7 Other Approaches to Protection

In this section, we covered all approaches to avoid buffer overflows or mitigate their consequences for which we could find publicly available information. We have not mentioned commercial tools that can help detecting buffer overflows at run time or statically, but for which there is not enough publicly available information to describe how they work. We believe most of them use techniques that are presented here.

5 Conclusion

In this survey, we have described what buffer overflows are and how they can be exploited. We have also described methods to detect this kind of vulnerability, to get the assurance that it is not present, and to avoid its exploitation.

In an ideal world, a static approach would give the assurance that a program does not have such vulnerabilities before executing it. Dynamic methods to avoid these vulnerabilities or limit their consequences would then be superfluous and the program could execute at full speed without risking compromising the security of a system.

However, in the real world, this is impossible. Thus there are cases where some static approach is better, there are cases where some dynamic approach is better, and there are other cases where the combination of many approaches is preferable.

The approaches using static analysis that we have studied all have important limitations. They either miss some errors, or they generate too many false positives to be useful when analysing large programs. The dynamic approaches give a result that is valid only for a limited number of executions of a program, and most of them cannot detect all buffer overflows. Those that can add a large overhead in execution time.

Moreover, there is still an important problem: what should be done when a vulnerability is detected? Usually, the best can be done is to stop the program since allowing it to continue could have consequences that are more serious. However, abruptly stopping a critical program can also lead to serious consequences. This question is still open and applies not only to C and C++, but also to Java and other languages for which there is full run-time bounds checking.

APPENDIX

This appendix presents different publicly available programs that can be useful to detect or avoid buffer overflows and/or some of their consequences. We did not test most of them, and the information in this section is based on available documentation. We classified them in five categories:

- static analysis;

- dialect of C;
- run-time bounds checking or overflow detection;
- testing tools;
- other dynamic checks.

A.1 Static Analysis Tools

This section describes the tools that use static analysis to detect buffer overflows.

Splint We described this program and its working in Section 4.6.2. In addition to detecting buffer overflows, Splint detects format string vulnerabilities and many misuses of the C language. The analysis is based on annotations that the programmer has to add to function declarations and is done one function at a time. Even though some errors can escape the analysis, most aspects of the C language are taken into account by Splint. Earlier versions of this program were called LCLint. `http://www.splint.org/`

RATS We discussed this program in Section 4.6.1. It does a lexical analysis that can detect "dangerous" constructions of C, C++, and a few other languages. This approach allows finding buffer overflows, format string vulnerabilities, and some other potential security problems. `http://www.fortify.com/security-resources/rats.jsp`

Flawfinder We discussed this program in Section 4.6.1. It does a lexical analysis that can detect "dangerous" constructions of the C and C++ languages. This approach allows finding buffer overflows, format string vulnerabilities, and some other potential security problems. `http://www.dwheeler.com/flawfinder/`

ITS4 We discussed this program in Section 4.6.1. It does a lexical analysis that can detect "dangerous" constructions of the C and C++ languages. This approach allows finding buffer overflows and some other potential security problems. `http://www.cigital.com/its4/`

BOON (Partly free) We discussed how this program works in Section 4.6.3. It can find buffer overflows by considering strings as an abstract data type manipulated by the functions of the C library. It does an interprocedural analysis, but not a very precise one. The detection of buffer overflows is done by solving a system of constraints on integer ranges representing the space allocated for the strings and their length. The overflows not caused by the call of a function of the C library are completely ignored. `http://www.cs.berkeley.edu/~daw/boon/`

Wasp This program is described in [Mat01]. It analyzes Java programs. Among other things, it can detect array overflows statically. `http://www.waspsoft.com/`

UNO This program is described in [Hol02]. It analyzes C programs. It can detect some array overflows statically, but for this many conditions have to be met. `http://spinroot.com/uno/`

A.2 Dialect of C

Cyclone It is a programming language based on C that does not allow buffer overflows and format string vulnerabilities. It was mentioned briefly in Section 4.1.1. `http://www.research.att.com/projects/cyclone/`

A.3 Run-Time Bounds Checking or Overflow Detection

GCC — Bounded Pointers It is an extension to GCC that modifies the representation of pointers so that run-time bounds checking can be efficient. The technique it uses is described in Section 4.3.6. `http://gcc.gnu.org/projects/bp/main.html`

Bounds Checking for C This is an extension of GCC that does run-time bounds checking on most pointers without any modification to their representation. The technique used is described in Section 4.3.6. `http://www.doc.ic.ac.uk/~phjk/BoundsChecking.html`

CCured This program analyses C source code and adds to it some run-time checks to avoid buffer overflows. It is described in Section 4.3.6. `http://manju.cs.berkeley.edu/ccured/`

StackGuard This is an extension to GCC protecting the return address of the functions on the stack. To do this, it uses canary values and assistance from the processor. It is described in Sections 4.3.1 and 4.3.1. Depending on versions and the configuration, the canary can be null, terminating or random. The protection with assistance from the processor (MemGuard) was available only in the first few versions because it is not efficient enough. StackGuard does not do real bounds checking and it does not avoid any overflow. It only prevents some attacks from working. To that end, the code generated by the compiler for the prologue and the epilogue of functions is altered.

Stack Shield It protects the return address of the functions on the stack. To do that, it uses an alternate stack. This technique is described in Section 4.3.1. However, the alternate stack is limited in size and, by default, it can only contain 256 elements. It means that if more than 256 function calls are nested, the last ones do not benefit from the protection. Stack Shield works by modifying the

assembly code generated when compiling code with GCC. It does not do any bounds checking and it does not avoid any overflow. It only prevents some attacks from working. This program is no longer maintained. `http://www.angelfire.com/sk/stackshield/`

Stack-Smashing Protector (SSP) It is an extension of GCC protecting return addresses and many other important data on the stack. To do this, it uses canaries and it reorders the variables on the stack. These techniques are described in Sections 4.3.1 and 4.3.2 respectively. It does no bounds checking and it does not prevent any overflow, but it can counter many attacks. It works by modifying the intermediate language code generated by GCC. It does not add any code when it can determine that the program will always behave correctly without these additions. However, the optimizer of GCC can remove some of the checks added by SSP when the optimization level asked is too high. In the past, this project was known as Propolice. `http://www.trl.ibm.com/projects/security/ssp/`

Libsafe This is a library replacing some of the functions of the C library. It works under Linux and it prevents return addresses from being overwritten in addition to format string vulnerabilities. The techniques it uses are described in Section 4.3.3. This library does not prevent all overflows, but it prevents the most obvious ones. `http://dag.wieers.com/rpm/packages/libsafe/` `http://www.research.avayalabs.com/project/libsafe/`

Libparanoia This is a library replacing some of the functions of the C library. It works under FreeBSD and it prevents return addresses from being overwritten. The technique it uses is described in Section 4.3.3. This library does not prevent all overflows, but it prevents the most obvious ones. `http://www.lexa.ru/snar/libparanoia/`

strlcpy() et strlcat() These functions offer alternate interfaces for handling strings, which help the programmer to avoid buffer overflows. They are described in Section 4.1.3.

Libmib — astring This is an alternate library doing allocation and reallocation of strings automatically to avoid buffer overflows. It is described in Section 4.1.3. `http://www.mibsoftware.com/libmib/astring/`

Checker This program is a debugging tool that can be used to find, at run time, if a program reads or writes outside allocated memory, or if it reads an uninitialized value. Thus, it can find some buffer overflows. The technique it uses is described in Section 4.3.6. It can also find memory blocks that are no longer referenced while still allocated. The program to be checked has to be instrumented using an extension of GCC. `http://www.gnu.org/software/checker/checker.html`

Valgrind This program is a debugging tool that can be used to find, at run time, if a program reads or writes outside allocated memory, or if it reads an uninitialized value. Thus, it can find some buffer overflows. The technique it uses is described in Section 4.3.6. It follows the use of memory bit by bit. It can also find memory blocks that are no longer referenced while still allocated and other kinds of errors. Moreover, it can simulate cache memory to profile its use. A JIT compiler allows instrumenting in real-time the machine code of the program to be checked (IA-32 to IA-32). `http://developer.kde.org/~sewardj/`

Rational Purify This program is a debugging tool that can be used to find, at run time, if a program reads or writes outside allocated memory, or if it reads an uninitialized value. Thus, it can find some buffer overflows. The technique it uses is described in Section 4.3.6. It follows the use of memory bit by bit. It can also find memory blocks that are no longer referenced while still allocated. The machine code of the checked program is modified dynamically at run time to keep a record of memory accesses. This software is not available under Linux, only under some other Unix platforms. `http://www-306.ibm.com/software/awdtools/purify/unix/`

Electric Fence It is a replacement library for the functions dealing with memory allocation. It allocates memory so that an access after (or before) any block of memory generates a memory fault. This technique is described in Section 4.3.3. `http://perens.com/FreeSoftware/`

A.4 Testing Tools

Fuzz This program generates random input for a program in order to find problems. It helped to identify many buffer overflows. It is mentioned in Section 4.2. `http://www.cs.wisc.edu/~bart/fuzz/fuzz.html`

BFBTester This tool tries to detect buffer overflows in the handling of arguments and environment variables of a program.
`http://bfbtester.sourceforge.net/`

A.5 Other Dynamic Checks

The programs in this section do not prevent buffer overflow vulnerabilities directly, but they can limit their consequences by preventing bad behaviors.

Linux kernel patch from the Openwall Project This patch to the Linux kernel prevents the execution of code on the stack and it modifies the address used to load shared libraries. These techniques are described in Sections 4.4.1 and 4.4.3. It also prevents other actions often done by pirates. `http://www.openwall.com/linux/`

PaX This extension of the Linux kernel allows marking individual pages as executable or not. It also locates shared libraries and the stack at random addresses so that it is more difficult to find the right address in an attack. These techniques are described in the first part of Section 4.4.2 and in Sections 4.4.3 and 4.4.4.

kNoX This patch for Linux kernel 2.2 prevents the execution of code located in all pages accessible for writing, and it modifies the address used to load dynamically-linked libraries. These techniques are described in Section 4.4.2 and in Section 4.4.3. It also prevents other actions often done by pirates. `http://isec.pl/projects/knox/knox.html`

RSX This extension of the Linux kernel allows marking individual pages as executable or not. The technique it uses is described in Section 4.4.2.

exec-shield This extension of the Linux kernel restricts the part of the address space of a program that can be executed by defining a code segment limit. It loads shared libraries in low memory in an attempt to have a code segment as small as possible. The technique it uses is described in Section 4.4.2. `http://people.redhat.com/mingo/exec-shield/`

Syscalltrack This software is, in large part, an extension of the Linux kernel, and it can be used to filter the system calls to the kernel for all programs of the system. It is mentioned briefly in Section 4.5.1. `http://syscalltrack.sourceforge.net/`

Janus This program can execute another program in a controlled environment, that is, with its actions checked for conformance to a security policy. It is mentioned briefly in Section 4.5.1. `http://www.cs.berkeley.edu/~daw/janus/`

Subterfugue It is an application framework to change the reality as seen by the programs. It can check system calls to discover what programs attempt to do or prevent them from doing some actions. It is mentioned briefly in Section 4.5.1. `http://subterfugue.org/`

References

[ABS94] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 290–301, New York, NY, USA, 1994. ACM Press.

[Ale96] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49):14, November 1996.

[Ale01] Aleph One. Bugtraq frequently asked questions, 2001.

[Ano01] Anonymous. Once upon a free()... *Phrack*, 11(57):0x09, August 2001.

[Bea01] Maxime Beaudoin. Données malicieuses: théorie et analyse. Technical Report DIUL-RR-0105, Département d'informatique, Université Laval, August 2001.

[BK00] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 10(56):0x05, November 2000.

[Bou00] Pascal Bouchareine. __atexit in memory bugs. Posted on VulnDev mailing list, Dec., December 2000.

[Cav98] Forrest J. Cavalier III. Libmib allocated string functions, 1998. `http://mibsoftware.com/libmib/astring/`.

[CBJW03] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, Berkeley, CA, USA, August 2003. USENIX Association.

[CH01] Tzi-cker Chiueh and Fu-Hau Hsu. Rad: A compile-time solution to buffer overflow attacks. In *21st International Conference on Distributed Computing*, page 409, Washington, DC, USA, April 2001. IEEE Computer Society.

[cla05] class101. Veritas backup exec 8.x/9.x remote universal exploit. Posted on BugTraq mailing list, January, January 2005. `http://www.securityfocus.com/archive/1/386754`.

[Con99] Matt Conover. w00w00 on heap overflows, January 1999. `http://www.w00w00.org/articles.html`.

[COR02] CORE Security Technologies. Multiple vulnerabilities in stack smashing protection technologies, April 2002. Apr. `http://www1.corest.com/corelabs/advisories/index.php`.

[Cow00] Crispin Cowan. Stackguard 1.21 vulnerability. Posted on BugTraq mailing list, Aug., August 2000.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, Berkeley, CA, USA, January 1998. USENIX Association.

[CWP+00] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, pages 1119–1129, Washington, DC, USA, 2000. IEEE Computer Society.

[Cyc02] Cyclone. Cyclone, 2002. `http://www.research.att.com/projects/cyclone/`.

[dR03] Theo de Raadt. i386 W^X. Posted on OpenBSD mailing list, Apr., April 2003.

[DRS03] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 155–167, New York, NY, USA, 2003. ACM Press.

[Dup02] Kasper Dupont. Why a stack with exec flag? Posted on comp.os.linux.security, June, June 2002.

[EL02] David Evans and David Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, 19(1):42–51, Jan/Feb 2002.

[EY00] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks, June 2000. IBM Research Division `http://www.trl.ibm.com/projects/security/ssp/main.html`.

[Fra01] Przemyslaw Frasunek. ntpd =< 4.0.99k remote buffer overflow. Posted on BugTraq mailing list, April, April 2001.

[Fry00] Niklas Frykholm. Countermeasures against buffer overflow attacks, November 2000. RSA Security `http://www.rsasecurity.com/rsalabs/node.asp?id=2011`.

[FS01] Mike Frantzen and Mike Shuey. StackGhost: Hardware facilitated stack protection. In *10th USENIX Security Symposium*, pages 55–66, Berkeley, CA, USA, August 2001. USENIX Association.

[GBPdlHQ+02] Jesús M. González-Barahona, Miguel A. Ortuño Pérez, Pedro de las Heras Quirós, José Centeno González, and Vicente Matellán Olivera. Counting potatoes: The size of Debian 2.2, January 2002. `http://opensource.mit.edu/papers/counting-potatoes.html`.

[Gin98] Tristan Gingold. Checker, 1998. `http://www.gnu.org/software/checker/checker.html`.

[GJC+03] Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In *CCS '03: Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 345–354, New York, NY, USA, 2003. ACM Press.

[GOM98] A. K. Ghosh, T. O'Connor, and G. McGraw. An automated approach for identifying potential vulnerabilities in software. In *IEEE Symposium on Security and Privacy*, pages 104–114, Washington, DC, USA, May 1998. IEEE Computer Society.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewe. A secure environment for untrusted helper applications. In *Proceedings of the 6th USENIX Security Symposium*, Berkeley, CA, USA, 1996. USENIX Association.

[HB03] Eric Haugh and Matthew Bishop. Testing C programs for buffer overflow vulnerabilities. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium*, Reston, VA, USA, February 2003. Internet Society.

[HJ91] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter 1992 USENIX Conference*, pages 125–136, Berkeley, CA, USA, 1991. USENIX Association.

[Hol02] Gerard J. Holzmann. UNO: Static source code checking for user-defined properties, 2002. Lucent Technologies `ftp://cm.bell-labs.com/cm/cs/what/uno/uno_cstr.pdf`.

[Imm97] Immunix. Immunix canary patch to GCC 2.7.2.2 — a buffer overflow exploit detector, December 1997.

[Imm00] Immunix. StackGuard mechanism: Emsi's vulnerability, 2000.

[Int99] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. International Organization for Standardization, Geneva, Switzerland, December 1999.

[Int00] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*, 2000.

[JK97] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs, 1997. Department of Computing Imperial College of Science, Technology and Medicine.

[Jon95] Richard W. M. Jones. A bounds checking C compiler, May 1995. Department of Computing Imperial College of Science, Technology and Medicine.

[Kae01] Michel Kaempf. [synnergy] - sudo vudo. Posted on BugTraq mailing list, June, June 2001.

[KBA02] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–206, Berkeley, CA, USA, 2002. USENIX Association.

[Ket98] Richard Kettlewell. Protecting against some buffer-overrun attacks, April 1998. `http://www.greenend.org.uk/rjk/random-stack.html`.

[klo99] klog. The frame pointer overwrite. *Phrack*, 9(55):08, September 1999.

[KS02] Paul A. Karger and Roger R. Schell. Thirty years later: Lessons from the multics security evaluation. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference*, page 119, Washington, DC, USA, 2002. IEEE Computer Society.

[LC02] Kyung-suk Lhee and Steve J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–88, Berkeley, CA, USA, 2002. USENIX Association.

[LE01] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *10th USENIX Security Symposium*, Berkeley, CA, USA, August 2001. USENIX Association.

[Lib01] Libsafe. Libsafe, 2001. `http://www.research.avayalabs.com/project/libsafe/`.

[LL03] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in C programs. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 317–326, New York, NY, USA, 2003. ACM Press.

[Mat01] Vincent Mathieu. Outils d'analyse statique. Technical Report DIUL-RR-0106, Département d'informatique, Université Laval, August 2001.

[McG98a] Greg McGary. Array bounds checking? Posted on egcs mailing list, May, May 1998.

[McG98b] Greg McGary. Bounds checking. Posted on egcs mailing list, May, May 1998.

[McG99] Greg McGary. Support array bounds checking. Posted on egcs-patches mailing list, June, June 1999.

[McG00] Greg McGary. Bounds checking in C and C++ using bounded pointers, August 2000. `http://gcc.gnu.org/projects/bp/main.html`.

[MKL+95] Barton P. Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of UNIX utilities and services, 1995. Computer Sciences Department, University of Wisconsin `ftp://ftp.cs.wisc.edu/paradyn/technical_papers/fuzz-revisited.pdf`.

[MKSL03] John P. McGregor, David K. Karig, Zhijie Shi, and Ruby B. Lee. A processor architecture defense against buffer overflow attacks. In *IEEE International Conference on Information Technology: Research and Education (ITRE 2003)*, pages 243–250, Washington, DC, USA, August 2003. IEEE Computer Society.

[Mol03] Ingo Molnar. Exec shield, new Linux security feature. Posted on linux-kernel mailing list, May, May 2003.

[Mol04] Ingo Molnar. Nx (no execute) support for x86. Posted on linux-kernel mailing list, Jun, June 2004.

[MV00] Gary McGraw and John Viega. Make your software behave: Preventing buffer overflows, March 2000. *IBM developerWorks*.

[NMW02] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, New York, NY, USA, 2002. ACM Press.

[OSSY02] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-safe ANSI-C compiler: An approach to making C programs secure (progress report). In *Proceedings of International Symposium on Software Security, Tokyo, Japan*, pages 25–36, New York, NY, USA, November 2002. Springer-Verlag.

[PaX00] PaX Team. Original design & implementation of PAGEEXEC, November 2000.

[PaX02] PaX Team. Why a stack with exec flag? Posted on comp.os.linux.security group, June, June 2002.

[Per93] Bruce Perens. Electric fence, 1993. `http://perens.com/FreeSoftware/`.

[PF97] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software: Practice and Experience*, 27(1):87–110, 1997.

[PL02] Changwoo Pyo and Gyungho Lee. Encoding function pointers and memory arrangement checking against buffer overflow attack. In *ICICS '02: Proceedings of the 4th International Conference on Information and Communications Security*, pages 25–36, New York, NY, USA, 2002. Springer-Verlag.

[PL04] Yong-Joon Park and Gyungho Lee. Repairing return address stack for buffer overflow protection. In *CF'04: Proceedings of the First Conference on Computing Frontiers*, pages 335–342, New York, NY, USA, 2004. ACM Press.

[Pro03] Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, Berkeley, CA, USA, August 2003. USENIX Association.

[Rit93] Dennis M. Ritchie. The development of the c language. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 201–208, New York, NY, USA, 1993. ACM Press.

[rix00] rix. Smashing C++ vptrs. *Phrack*, 10(56):0x08, May 2000.

[RL04] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, Reston, VA, USA, February 2004. Internet Society.

[RWM02] Chris Ren, Michael Weber, and Gary McGraw. Microsoft compiler flaw technical note, 2002. Cigital `http://www.cigital.com/news/mscompiler-tech.pdf`.

[SBDB01] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *SP '01: Proceedings of the IEEE Symposium on Security and Privacy*, page 144, Washington, DC, USA, 2001. IEEE Computer Society.

[Smi97] Nathan P. Smith. Stack smashing vulnerabilities in the UNIX operating system, May 1997.

[Sna97] Alexander Snarskii. Increasing overall security..., February 1997.

[Sna00] Alexandre Snarskii. Libparanoia, 2000. `http://www.lexa.ru/snar/libparanoia/`.

[Sol97a] Solar Designer. Getting around non-executable stack (and fix). Posted on BugTraq mailing list, Aug., August 1997.

[Sol97b] Solar Designer. Non-executable stack – final Linux kernel patch. Posted on linux-kernel mailing list, May, May 1997.

[Sol02] Solar Designer. Linux kernel patch from the Openwall Project, September 2002. `http://www.openwall.com/linux/`.

[Sta01] Paul Starzetz. Announcing RSX - non exec stack/heap module. Posted on BugTraq mailing list, June, June 2001.

[SU99] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the 8th USENIX Security Symposium*, pages 63–78, Berkeley, CA, USA, 1999. USENIX Association.

[Sub02] Subterfugue. Subterfugue, 2002. `http://subterfugue.org/`.

[Sys02] Syscalltrack. Syscalltrack, 2002. `http://syscalltrack.sourceforge.net/`.

[TK02] Thomas Toth and Christopher Kruegel. Accurate buffer overflow detection via abstract payload execution. In *Recent Advances in Intrusion Detection : 5th International Symposium*, pages 274–291, New York, NY, USA, October 2002. Springer-Verlag.

[TS01a] Timothy Tsai and Navjot Singh. Libsafe 2.0: Detection of format string vulnerability exploits, February 2001.

[TS01b] Timothy Tsai and Navjot Singh. Libsafe: Protecting critical elements of stacks. Technical Report ALR-2001-019, Avaya Labs, Avaya Inc., 233 Mt. Airy Rd., NJ 07920 USA, August 2001.

[VBKM00] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw. ITS4: A static vulnerability scanner for C and C++ code. In *ACSAC '00: Proceedings of the 16th Annual Computer Security Applications Conference*, page 257, Washington, DC, USA, 2000. IEEE Computer Society.

[Ven00] Vendicator. Stack Shield, 2000. `http://www.angelfire.com/sk/stackshield/`.

[Wag00] David A. Wagner. *Static analysis and computer security: New techniques for software assurance*. PhD thesis, University of California at Berkeley, 2000.

[WFBA00] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed System Security Symposium*, pages 3–17, Reston, VA, USA, February 2000. Internet Society.

[Whe01] David A. Wheeler. More than a gigabuck: Estimating GNU/Linux's size, July 2001. `http://www.dwheeler.com/sloc/`.

[Whe02a] David A. Wheeler. Flawfinder, 2002. `http://www.dwheeler.com/flawfinder/`.

[Whe02b] David A. Wheeler. Secure programming for Linux and Unix howto, 2002. `http://www.dwheeler.com/secure-programs/`.

[Woj98] Rafal Wojtczuk. Defeating Solar Designer non-executable stack patch. Posted on BugTraq mailing list, Jan., January 1998.

[Woj01] Rafal Wojtczuk. The advanced return-into-lib(c) exploits: PaX case study. *Phrack*, 11(58):0x04, December 2001.

[Wor03] WordNet. Wordnet, 2003. `http://www.cogsci.princeton.edu/~wn/`.

[XCE03] Yichen Xie, Andy Chou, and Dawson Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 327–336, New York, NY, USA, 2003. ACM Press.

[XDS04] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 117–126, New York, NY, USA, 2004. ACM Press.

[XKPI02] Jun Xu, Zbigniew Kalbarczyk, Sanjay Patel, and Ravishankar K. Iyer. Architecture support for defending against buffer overflow attacks. In *Second Workshop on Evaluating and Architecting System dependabilitY (EASY)*. Web, October 2002.

[YH03] Suan Hsi Yong and Susan Horwitz. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 307–316, New York, NY, USA, 2003. ACM Press.