Interested in learning more about security?

SANS Institute InfoSec Reading Room

Buffer Overflows for Dummies

Buffer Overflows are responsible for many vulnerabilities in operating systems and application programs, actually dating back to the famous Morris worm in 1988. Descriptions of buffer overflow exploitation techniques are, however, in many cases either only scratching the surface or quite technical, including program source code, assembler listings and debugger usage, which scares away a lot of people without a solid programming background. This paper tries to fill the gap between those two categories by striking a good...

Buffer Overflows for Dummies

Josef Nelißen
May 1, 2002
GSEC Practical Assignment Version 1.4

Summary

Buffer Overflows are responsible for many vulnerabilities in operating systems and application programs, actually dating back to the famous Morris worm in 1988. Descriptions of buffer overflow exploitation techniques are, however, in many cases either only scratching the surface or quite technical, including program source code, assembler listings and debugger usage, which scares away a lot of people without a solid programming background.

This paper tries to fill the gap between those two categories by striking a good balance between depth and breadth of the presentation, covering the stack smashing, frame pointer overwrite, return-into-libc, and heap based overflow techniques as well as possible countermeasures.

Table of Contents

Table of Figures

Typographic Conventions:

The first time a term is introduced it is emphasized by *italics*.

Terms and names which are related to the C programming language are provided in Courier New font, e.g. `main()`.

1 Introduction

Buffer Overflow based exploits are featured on all security related web sites and mailing lists. For example, the SANS Windows Security Digest dedicates a regular section to buffer overflows, stating

> Buffer overflows can generally be used to execute arbitrary code on the victim host; as such, they should be considered HIGH risk. Many buffer overflows are discovered each month. [16]

A recent CERT Security Improvement Feature backs this view:

> Even though the cause [The Morris Worm of 1988] was highly publicized, buffer overflows are still a major cause of intrusions today. In fact, the first six CERT® Coordination Center advisories issued in 2002 describe buffer overflow flaws in several common computer programs. [28]

Although many people are aware of the high risk posed by buffer overflows, insight into this topic is not exactly easy to gain. Many sources of information require a rather solid background in topics as the C programming language, assembler code and using a debugger on assembly level, e.g. [1], [19], or [24]. On the other hand some papers describe buffer overflows only on a quite abstract level, focusing mainly on the stack smashing technique, which we will describe in section 7.

This paper explains several common buffer overflow exploitation techniques. Its focus is to strike a good balance between depth and breadth of the presentation. Using this approach we will sometimes deliberately omit technical details or provide a simplified model. If technical details are desired, the references should provide a good starting point.

The underlying model of the paper are the C programming language and the Linux operating system on the widespread Intel x86 architecture, sometimes called IA32 as well, which includes in particular all Pentium and the corresponding AMD processors.

Actually all other current operating systems and architectures we are aware of are, however, vulnerable to buffer overflows too, but technical details will differ.

2 So what's a Buffer Overflow, after all?

The ultimate purpose of any program that runs on a computer is to process data of some kind. Most don't operate on fixed data, but on data that is ultimately provided by a user, possibly preprocessed in some fashion. The program needs to store the data somewhere in the computer's memory, and this is the point where the problems start. Many programmers implicitly assume that the user input will be "reasonable" to a certain extent. This assumption holds in particular for the length of character input, commonly called *strings*. It seems for instance unreasonable that the address of a web page will be longer than say 500 characters. In most cases programmers add some safety margin, e.g. multiply by 2 or even 10 and assume that this should now really be an upper bound on the length of any address that a web surfer might provide. Based on this assumption, the programmer of a web server reserves memory for a web page address that could hold up to 5'000 characters. This reserved memory space is commonly called a *buffer*[1].

[1] More precisely, a buffer (often called an array as well) is a segment in memory that holds a number of identical objects. In our example the objects are mostly characters, but they could be of any other pre- or programmer-defined type.

So far everything is more or less ok with this approach. Now comes the crucial part, namely: Is the implicit assumption of "reasonable" length explicitly checked?!? This means, does the programmer just assume "5'000 characters will do", and process the input as provided by the user in the address field of his browser without further question, or **will he explicitly check** that this address is not longer than 5'000 characters?

In many, many cases there is no check at all (due to laziness, guilelessness, an "aggressive" development schedule or simply obliviousness of the programmer) – the data is just processed as is[2]. So what happens if a user unintentionally or intentionally provides a web address with say 5'500 characters in it? After writing the 5'000 "good" characters in memory, and thus filling up the allocated buffer, the remaining 500 additional "bad" characters[3] will be placed in the memory following the buffer, resulting in a *buffer overflow.*

Please note that any method for providing user input to a program can be (ab)used for buffer overflow purposes. This includes, but is not limited to:

a) Data typed or pasted into text fields of the program's graphical user interface (as in the previous example of input to a web server)

b) Data sent to the program via a network

c) Data provided in a file

d) Data provided on the command line when invoking the program via a command line interpreter (any shell in Unix or cmd.exe in Windows)

e) Data provided in environment variables (e.g. %TMP% in Windows or $TMP in Unix – the directory used for temporary files)

f) …

Note that a), b), and c) are examples for buffer overflows that can probably be exploited remotely, while the others are targeting a local system.

Depending on a number of factors, including the method of memory reservation chosen by the programmer (see section 6 below), the internal memory organization of the computer running the program, and the actual input, the result of a buffer overflow may be:

a) Corruption of other program data, resulting in incorrect output for the input data, e.g. for a program doing some sort of calculation a wrong result (in our web server example, assuming that the provided web address is indeed valid, incorrect output would be for instance a "404 - Document Not Found" http-error).

b) Corruption of control flow structures of the program, generally leading to premature program termination (On Unix systems, this often triggers the error message "segmentation violation", on Windows Systems "general protection fault" or "access violation" – all meaning that the program tried to access memory locations outside the memory range reserved for it by the operating system).

c) Shutdown of the computer running the program, if the involved program was e.g. part of the operating system.

[2] Note that in many circumstances the best choice would actually be to dynamically allocate enough memory for the actual input data during program execution, which requires, however, slightly more work, has some other drawbacks (as discussed below) and is hence rarely done.

[3] How does the program recognize that it read all its data? Normally the end of the data is marked by special data values; for strings normally the `NULL` character with a decimal value 0. The processing of data stops with the first NULL character read, which terminates the input.

This certainly doesn't sound desirable, but actually you might be better off if you encounter only those cases! The sad truth is, however, that by carefully crafting the data causing the overflow, an attacker might

d) **Be able to execute code of his choice on the computer running the program!**

This threat is the reason why potential buffer overflows are not only annoying, but actually a huge threat to anyone running a vulnerable program!

To understand how case d) can be accomplished, we need to discuss some fundamentals of computer programs and architecture.

3 Programs, Libraries, and Binaries

Any of the programs we use day-by-day are built from constants, variables and instructions that deal with the data to be processed. The instructions are normally grouped in small units called *modules*, *(sub-)routines*, or, in the case of the C programming language, simply *functions*, which each implement a certain functionality. Fortunately many common tasks don't need to be programmed in detail, but are available as functions already in *libraries*; collections of useful routines offered both by the underlying operating system and from third parties.

A special program called a *compiler* is used to translate the high level instructions of a programming language like C into machine code – byte sequences that the targeted processor executes to accomplish the program's target. Note that you can by means of *disassemblers* convert the machine code back into so called *assembler code*, which is a very basic, not to say cryptic sort of programming language, but even so better readable by humans. Another program, called the linker (sometimes embedded in or otherwise directly invoked by the compiler) is used to integrate all parts of the program, including all referenced functions from libraries. The result of all those activities is commonly called a *binary*.

When the user invokes by whatever means the binary, the execution starts (by convention of the C programming language) at the machine code equivalent to the first line of code of the `main()` function (which hence needs to be present in any C program!). For any non-trivial program, `main()` will call another function, which in turn might call a third one, which may, after performing its task, transfer control back to the second function and so on and so on, until the binary either exits on its own behalf (e.g. by reaching finally the end of the `main()` function or calling the `exit()` function) or is explicitly closed by user input.

4 The Threat

Each execution of a binary has a certain context that determines which privileges are granted to this running instance of the binary. Type and granularity of privileges are highly dependent on the operating system and the configuration of the computer. Typical examples are the permission to read or write files, to open network connections on specific ports or to shut down other binaries or the whole computer. Often the context is solely dependent on the user starting the binary, meaning that the binary can basically perform all actions programmatically that the user could perform manually, and nothing more.

Some programs, however, require extended privileges to accomplish their task. Consider for instance a program that permits users to change their password. This program needs to be able to write the corresponding data into a password file or database. Under Unix Systems, the way to accomplish this is through the use of *suid* programs that use the privileges of the binary's file owner instead of the user invoking it.

Even more attractive targets in particular for remote attacks are, however, certain programs that are started already at boot time and run silently in the background all time. These daemons (Unix term) or services (Windows term) often run with extended privileges (which are for instance required on Unix systems for access to the server ports 1-1023). This and the "always on" property are a combination that suits many attackers.

Basically in the same category fall server applications like web servers, which share the "always on" property. In particular Microsoft's Internet Information Server (IIS) has a bad record of buffer overflow (and other) vulnerabilities see for instance [14].

So the goal of an attacker utilizing buffer overflow techniques is generally to abuse a binary (and thus its privileges – the more, the better) by running code of his choice on the target machine.

5 Basics of Computer Architecture

On an abstract level, any computer consists of

- a central processing unit, *CPU*, which does the real data processing,
- memory, that is sequentially numbered starting with address 0,
- some means for non-volatile storage of data – usually provided by hard disks, and
- input and output devices like keyboard, mouse, network cards, etc.

The CPU, often referred to as the computer's (main) processor (e.g. a Pentium 4), holds a number of so called *registers*, which can hold and process/manipulate data. One particular register, called the *instruction pointer*, *IP*, is responsible for keeping track of program execution by holding the address of the next instruction to be executed. Note that a *pointer* is generally an entity containing a memory address.

The smallest unit of information that a CPU and thus a computer can deal with is a *bit* (**Bi**nary Digi**t**). Dealing with single bits is, however, not exactly efficient. Hence very early in computer history the notion of a *word* was created, identifying the number of bits a CPU was able to deal with concurrently (still including the possibility to manipulate only one bit in a word). Early computers used *nibbles*, 4 bits, as words, later on *bytes*, 8 bits, were used, and currently the majority of systems (including all x86 and thus Pentium systems), deal with words of 32 bits. Actually a number of processors use already a word length of 64 bit, starting with DEC's/Compaq's Alpha processor already in 1993. Intel entered into the competition in 2001 with its Intel® Itanium™ processor series.

6 Memory Organization[4]

If a program is compiled and linked into a binary and then run, the different parts of it, namely program code, constants and data are placed in different segments of memory, as illustrated by Figure 1.

[4] Note that this section is mainly based on [24].

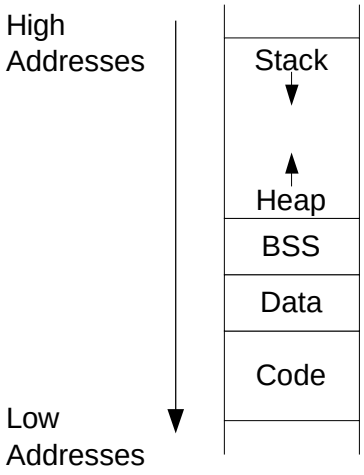


Figure 1: Memory layout of a binary (adapted from [24])

6.1 Code

First of all there is the Code segment, containing all those strange byte patterns whose meaning only the CPU and some blessed individuals can fully understand. This memory segment is read-only, which enables it to be shared by all users running the binary concurrently. So the Code segment is definitely no target for buffer overflows; any attempt to write into it will result in a memory access violation error and termination of the program.

Note that the other memory segments described below, which contain user data, are for obvious reasons not shared, but specifically created and reserved for each running instance of a binary.

6.2 Data and BSS

The Data and BSS segments contain global variables, which are accessible from any code in any function. The main difference between the Data and BSS segments is that the former contains variables which have been assigned an initial value in the program code, while uninitialized global variables reside in the BSS segment. Since variables shouldn't contain executable code, these memory segments are not executable; meaning that directing the instruction pointer of a binary by whatever means to them will cause premature program termination.

6.3 Stack

As you might already have guessed, the counterpart to global variables are local variables, which are used only in one function and thus only accessible by instructions in this function[5]. Local variables are placed in the memory segment called the *stack*, which makes dealing with them easy and efficient, as we will see below.

Computer scientists may try to impress you by calling a stack an abstract data type. Don't be intimidated[6]; although they are right, a stack is actually a nice, easy to comprehend thing. As stated in [23]:

[5] It is actually good programming practice to use as few global variables as possible in favor of local variables.

[6] You might just decide to strike back by innocently asking why on earth computer scientists can't even generally proof whether a program will on certain input data eventually stop or run forever. The fact that this is indeed not possible is known as the "Halting problem", proven the first time by Alan Turing in 1935 (see e.g. [34]). Trying to recall the proof and explain it wil l certainly keep your friend busy for some time and thus prevent him from bugging people. ☺

> The usual physical example of a stack is to be found in a cafeteria: a pile of plates or trays sitting on a spring in a well, so that when you put one on the top they all sink down, and when you take one off the top the rest spring up a bit.

Other names for a stack are *last-in-first-out* (LIFO) lists or *push down lists*. These terms describe one fundamental property of stacks: You are only able to access the topmost element, the top of the stack, and either move the element on top of it somewhere else and advance thus to the next element below, or put a new item on top of this element. These operations are known as *pop* and *push*, respectively, where pop takes as argument a destination where to put the popped element, while push uses the value that should be pushed as argument.

In our context, the stack is located at the end of the memory accessible by the program (cf. Figure 1) and growing downwards, so the topmost element is actually the one with the lowest address. Nevertheless it is still called the top. The CPU contains a special register that just keeps track of the top of the stack when it is growing and shrinking again. This register is called the *stack pointer*[7], SP.

Actually not only local variables are placed on the stack, but a number of other interesting objects as well – we'll cover these in a minute in the section "Smashing the stack".

Items that are pushed on the stack can, incidentally, only be put at *word boundaries*, meaning that the address must be a multiple of the word length. Hence if the program contains a local variable using only one byte, then nevertheless a full word is used to store this variable!

6.4 Heap

The final memory segment we need to cover is the *heap*. The heap is the memory area where you can allocate memory during the execution of a binary (by means of a system function called `malloc()`, **m**emory **alloc**ation). You (well, the programmer) can just say: "I now need 5'000 bytes of memory" and there it is, if you have been blessed by the operating system! This is particularly helpful if you can't predict how much space you will actually need, since this will depend on the input to the program (do you recall our discussion of fixed buffer length in the "So what's a Buffer Overflow, after all?" section? Great!). The counterpart to `malloc()` and the memory allocation is incidentally the `free()` function, which returns the memory to the operating system.

The heap is actually closely related to the already mentioned concept of a pointer in the C language, a memory address that holds no "real" data, but another memory address. Part of the magic of `malloc()` is that it provides you with the lowest address of the memory region you have been granted – how could you otherwise access it? Variables holding memory addresses are of pointer type, hence the address returned by `malloc()` is for future use stored in such a pointer. This can on the one hand be very useful, but has on the other hand been a constant source of various problems with C programs[8], in particular if pointers are not properly initialized or operations on pointers are done wrong.

[7] Note that the term used in the x86 processor family is actually *extended* stack pointer, %esi, where the term extended is used to indicate that this is a 32 bit architecture, no longer 16 bit (cf. section "The registers" in [24]).

[8] Note that the Java programming language has for reasons of safety and robustness explicitly disapproved pointers in its design!

7 Smashing the stack

7.1 Exploitation Technique

Equipped with the information provided in the previous sections we are now in the position to discuss the simplest or most straightforward buffer overflow attack, commonly called *stack smashing*.

The "classical" papers discussing these exploits first were published by "Mudge" in October 1995, see [19], and by "Aleph One" in the Phrack Magazine in November 1996, see [1]. Apart from these papers this section benefits again from [24] and [25], in particular regarding the figures.

To understand the fundamental technique of stack smashing we need to dig a little bit more into what happens when a function is called by another function, i.e. when a source code line like

`my_func(param1, param2, …, paramn);`

is executed, which uses local variables var_1, var_2, …, var_m and is called with certain values for its parameters `param1, param2, …, paramn`.

Before the call of `my_func()`, the stack appear as shown in Figure 2.

As discussed before, the stack pointer SP contains the address Y of the top of the stack, and the instruction pointer IP the address Z of the next machine code instruction to be executed, in this case the first of a series of instructions to actually prepare the call of `my_func()`. But what is the function of the BP register?

Although we stated above that the stack is used to store local variables, in truth it holds more information than that. Actually the whole context of a function is pushed onto the stack, containing its parameters, its local variables and some other information which we'll handle below. The whole bunch of memory dedicated to a function is called its *stack frame*.

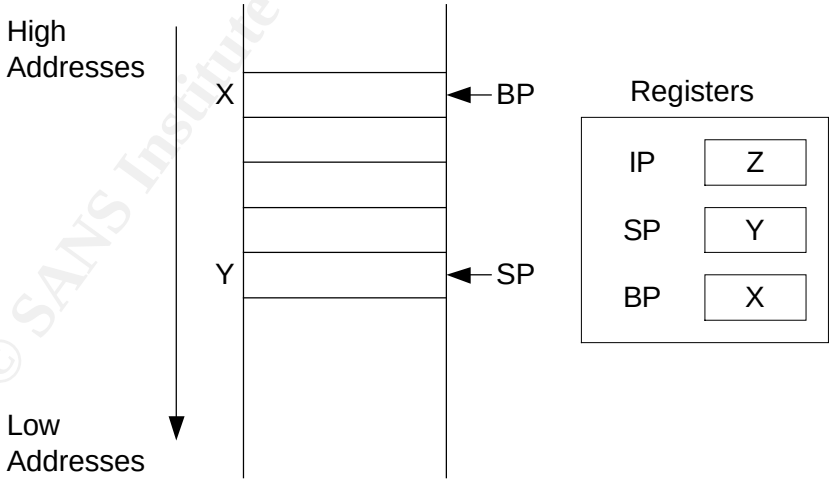


Figure 2: Stack before function call

Now the processor needs some means to address both the parameters and local variables in such a stack frame. As it turns out, one clever way to accomplish this is to identify a fixed location in each stack frame. This reference point is the contents of the BP register, which stands for *base pointer* (sometimes, i.e. for other processors, called *frame pointer* as well).

But back to our call to `my_func()`. The first thing to do is to pass the actual arguments, that is the values for $param_1$, $param_2$, …, $param_n$ to it. This is for obvious reasons actually the responsibility of the function calling `my_func()`. By convention the parameters are passed in reverse order, starting with $param_n$, down to $param_1$.

The next problem we are faced with is that after `my_func()` has done its job, we would like to continue the execution in the calling function. To accomplish this, we just push the address V of the next instruction after the call on the stack as well (this is the time to admit that such a stack is really a handy thing!).

Figure 3 shows how our stack looks now after parameter and return address pushing has been performed. Please note that this figure doesn't scale; actually the size required for each parameter does normally differ widely, depending on the type of each parameter.

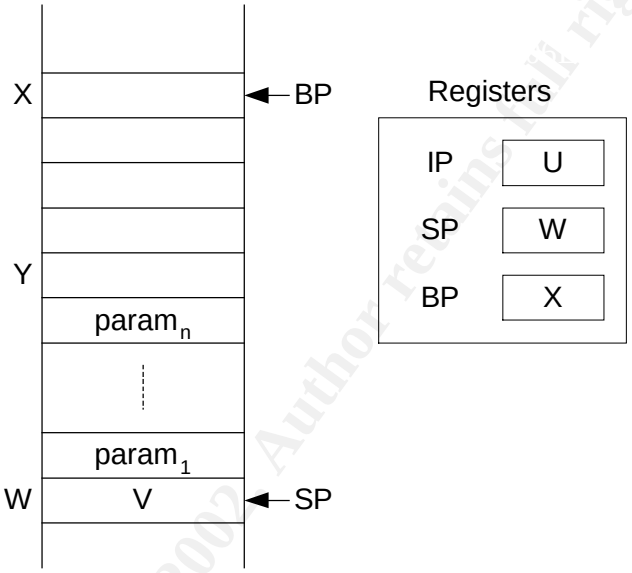


Figure 3: Stack after Pushing of Parameters and Return Address

The next instruction at address U is the actual call of `my_func()`, which just copies the address of the first instruction of it into the IP register.

Now finally comes the hour of fame of the called function! Before it can start its real work, however, it must make sure that

1. the old base pointer is stored by pushing it on the stack – note that this is a prerequisite for properly transferring control back to the calling function!
2. the base pointer register is updated by copying the value of the stack pointer register to the base pointer, and that
3. stack place for the local variables is reserved by sliding the stack pointer down according to the required memory size of all local variables, in our example for var_1, var_2, …, var_m.

Note that the first two steps are handled by the same machine code / assembler instructions for all function calls (although the actual values stored from or kept in the registers will of course differ). In step 3 only the number of words[9] to reserve on the stack differs between function calls.

[9] Recall from section 6.3 that the smallest addressable unit on the stack are actually words!

These three steps, which constitute the generic part at the beginning of all function calls, are called the *prolog* of a function.

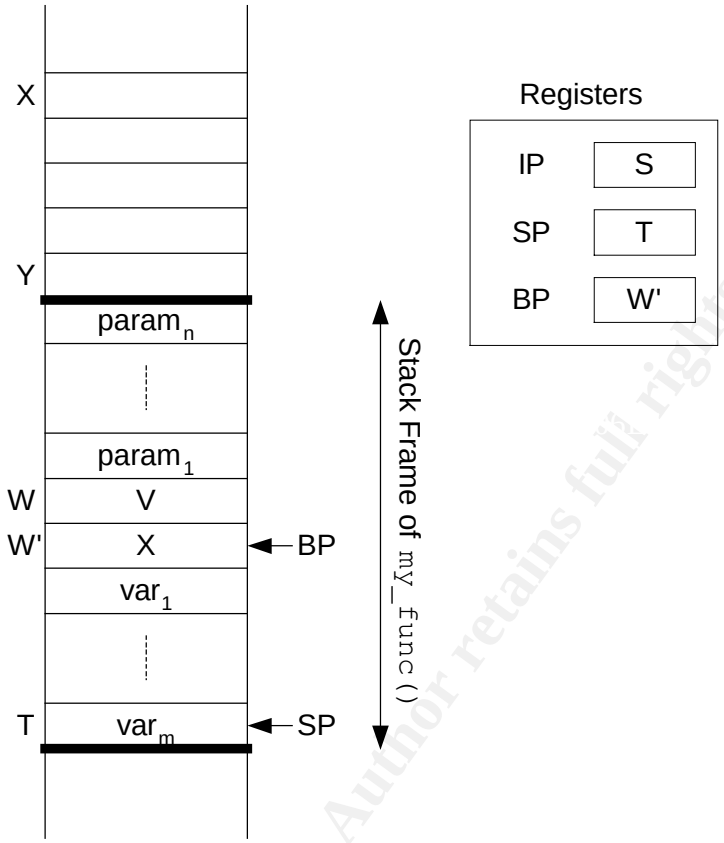


Figure 4: Stack after function prolog

Figure 4 shows the stack after the function prolog (again not explicitly indicating the different size requirement for the local variables). Note that the IP holds the address S of the first "real" instruction in the function and that the address W', to which BP points, is actually the value of W minus the number of bytes in a word.

Please note that the parameters and local variables of the function are arranged in a symmetric fashion above and below the base pointer. As a side effect, the relative offset of the parameters is positive with regard to BP, and the local variables have a negative offset[10].

As you might have guessed: If there is a prolog, then there must be an *epilog* as well. Its purpose is to clean up after the function has done its "real" work.

This involves again three steps:

1. To copy the value W' of the base pointer into the stack pointer, thus discarding the local variables.
2. Copy the saved value X of the base pointer location of the calling function back into BP (done by a pop instruction, since SP now points to the stack address holding this value). As a comparison of Figure 5 with Figure 3 shows, the stack is again back to the status it had before the function's prolog, if we neglect the possibly changed values of

[10] This enables experienced assembler programmers to distinguish between those two types of variables by just looking at the assembler instructions.

parameters and local variables of the called function. With regard to the registers, the only difference is that the instruction pointer contains now the address R.

3. R points to another pop operation which will copy the return address, V in our example, back to the instruction pointer.

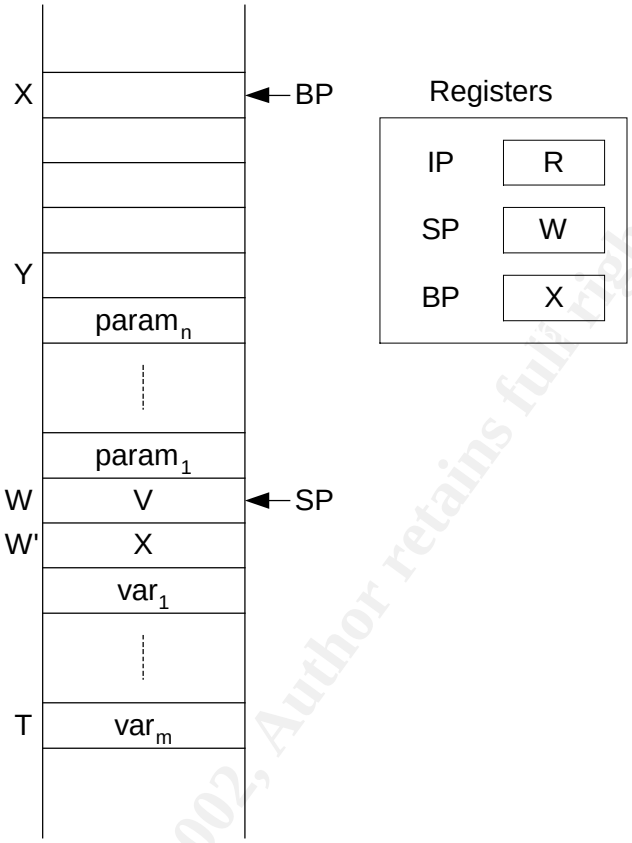


Figure 5: Stack after step 2 of the function epilog

The instruction at this address V (in the calling function – we are back!) will actually increment the stack pointer by the same amount it was effectively increased by pushing all parameters $param_1$, $param_2$, …, $param_n$ on the stack prior to the function call. With regard to the stack we have now eventually reached the same situation as shown in Figure 2 before the function call was made[11].

So far everything seems alright, so where is the problem or threat?

Imagine that one of the innocent looking local variables var_1, var_2, …, var_m might actually be a buffer of any kind. By convention buffers are placed in memory in such a way that the first item in the buffer has the lowest address. In other words, the buffer grows in the direction of higher addresses, upwards in our figures.

Imagine further that the length of input to this buffer isn't checked. The excess data will first overwrite other local variables, if any, and then proceed to provide new values for the stored values of the base pointer and the return address to jump to after the function epilog has been executed. Well, this offers indeed interesting opportunities! Imagine that we provide buffer data to the program that will overwrite this return address with the address of some carefully crafted machine code which we put into the buffer either after

[11] Note that the actual result or side effect of calling the function manifests in changed value of registers or global variables or e.g. by some output in the graphical user interface of the program.

the faked return address or before it[12]. If properly done, control flow of the attacked program will continue with this "injected" code after the epilog of `my_func()`!

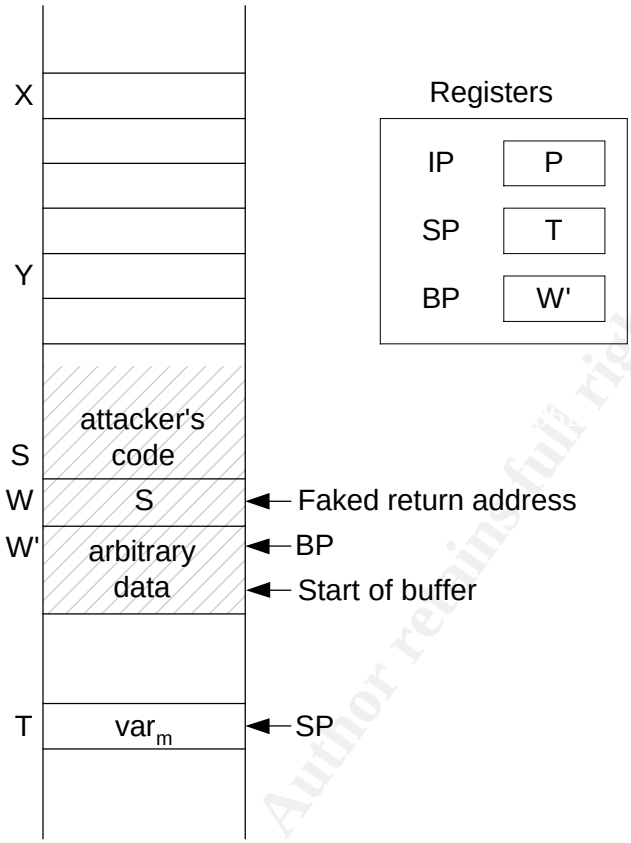


Figure 6: Stack after buffer overflow

Figure 6 shows the stack after the buffer overflow (whose extent is indicated by hatching) occurred, where the attacker's code was put after the faked return address.

On Unix systems, the most prevalent type of code used by attackers, is so called *shellcode*. A Shell is a program providing a command line interface to Unix Systems. The default shell program is the Bourne Shell, which is located under /bin/sh in the Unix filesystem. The goal of the shellcode is to execute the /bin/sh binary on behalf of the attacked binary, which provides the attacker with an interface to execute arbitrary commands under the privileges of the attacked program (recall our discussion in section 4 above).

Examples for such shellcode for various system are widely available on the Internet, e.g. in references [1], [19] or [24]. One of the advantages of shellcode is that it is usually quite small – the example provided in [24] actually fits into less than 60 bytes! This minimizes the risk that the buffer overflow data exceeds the attacked program's memory space or that other unwanted side effects (from the attacker's perspective) occur!

For Windows system, the term shellcode is used as well, but normally in a different meaning. As stated in [10]:

[12] Note that in this and the other examples of this section, injecting the code somewhere in memory and arranging to get it executed is done in one single step. This needs not necessarily be the case, but is obviously convenient, if possible!

> …the problem with most win32 remote overflow exploits stems from the payload, the current trend is to have the shellcode download an external file and execute.

This is probably due to the fact that exactly this approach was chosen in the "classical" paper [12] on exploiting buffer overflows in Windows.

Reference [10] shows, however, an example for shellcode that is the equivalent of the traditional shellcode for Unix systems for Windows systems.

In any case there are some obstacles for exploiting a buffer overflow vulnerability even if the shellcode as such is bulletproof, meaning that it would indeed provide a shell to the user if run in the context of a "normal" program on the target machine. These obstacles include, but are not limited to:

1. Guessing the right value to put into the faked return address
2. Guessing the location of the return address on the stack (W in our examples) relatively to the overflowed buffer (in case you don't have access to the source code of the vulnerable program and hence can't just look up the offset)
3. Ensuring that the shellcode doesn't contain any zeros, since string copying functions, which are responsible for the majority of buffer overflow vulnerabilities, will stop copying data (in this case the shellcode) after they encounter the first byte containing zero, interpreted as the special NULL character.

Issues 1 and 2 are normally handled simultaneously. The key idea is

1. to prepend a number of so called NOP instructions (short for **N**o **Op**eration – an instruction that does exactly this – nothing!) in front of the shell code and
2. to repeat the estimated (well, "guestimated" would be the more appropriate term) start address several times in the data used to overflow the buffer.

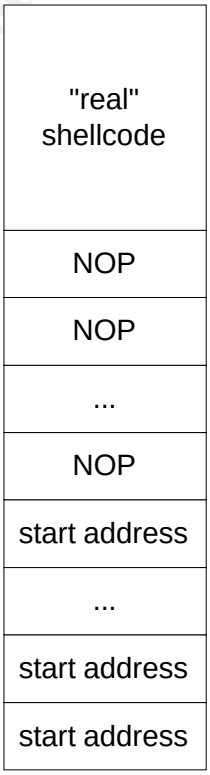


Figure 7: Buffer filled up for exploit (adapted from [25])

Figure 7 shows a buffer whose payload for an exploit is arranged according to this scheme. By this approach we have first increased the probability that the start address will indeed overwrite the return address of the stack frame. Moreover it is now no longer required to exactly hit the beginning of the "real" shellcode with the start address, but it is sufficient that the start address points somewhere into the NOPs, since the CPU will just advance through them till it reaches the "real" shellcode. More details on these techniques and other issues, which are beyond the scope of this paper, can be found e.g. in [1] and [25].

Issue 3 can be tackled e.g. by using register operations in the shellcode to generate zeros as register content and use it in the appropriate place; see [1] or [24] for a detailed discussion of this approach.

Another possibility is in some cases to "mask" part of the shellcode by e.g. setting a specific bit that was 0 for all bytes in the original data to 1 in all bytes and reversing this operation by the shellcode. This approach is discussed in [11].

Note that it is even possible to write shellcode for the x86 architecture that uses only bytes that are alphanumeric, meaning that the shellcode would look like garbage from digits and upper and lower case characters, when loaded into a text editor (see [26]). Such approaches have the additional advantage (for a possible attacker!) that they escape the shellcode identification routines of advanced intrusion detection systems that look for suspicious byte sequences contained in "standard" shellcodes.

7.2 Countermeasures

7.2.1 Software Development

Since the root cause of stack smashing vulnerabilities are programming flaws, proper care in software development is the ultimate line of defense. Recommendations include, but are not limited to

- Make security a top priority of your development efforts, even if this might conflict with the number of features you plan to implement or with the development schedule.
- Use appropriate code review techniques to ensure the quality of code (and try generally to improve the quality of your software development process in other important areas as well!)
- Use other programming languages like Java or Ada95, which are not vulnerable to buffer overflows, for a new product, or to port your existing application to such languages. If this is not feasible (as is sadly true for quite a large number of applications, which will still be around for years to come) then the measures listed in the following bullets should be applied.
- Perform proper bounds checking on all input data and in loops processing data (we will discuss possible problems with loops below in section 8.1).
- On top of using, if possible, appropriate technical means (like libsafe, see below), avoid the implicit problems of C-functions like `strcpy()` and `strcat()` and friends by properly[13] using their bounds checking equivalents, in this case `strncpy()` and `strncat()` which take as additional argument the maximum number of characters that will be copied.

[13] See the section on "Using n functions" in [25] and the examples in [33] for more information on proper and improper usage.

- Use dynamic variables, i.e. put the buffers on the heap instead of the stack. This trades the problem of stack smashing for that buffer in for an increase in lines of code, possible memory leaks and general problems with memory management. So this is more sort of a workaround than a real countermeasure. , Moreover, as concluded in [25]: "Again, this doesn't correct the problem, the exploit just becomes less trivial." We'll cover some related exploits in the section on Heap Overflows below.

7.2.2 Tools and Technical Means

An interesting approach to avoid a number of unsafe C functions on Linux is the Libsafe project of Avaya Labs Research (formerly Bell Labs), see [3]. Libsafe is a library which contains safe re-implementations of functions like `strcpy()`, `strcat()`, and `gets()`. After installing libsafe, calls to such function are intercepted and executed with libsafe's version instead of the default implementation. "Safe" basically means that the functions ensure that the saved base pointer will not be overwritten. Hence buffer overflows corrupting other local variables on the stack are still possible, but both the base pointer and the return address are save. Naturally, this protects only from buffer overflows for functions that are re-implemented in libsafe!

"Traditional" run-time memory usage tools like Rational's Purify or Compuware's (previously NuMega) BoundsChecker enable the test and debug of an application in its system test, including possible buffer overflows **if they occur in the test**. They provide in particular convenient and efficient support for locating the source of errors quickly.

Further on in the tools section, two compiler enhancements for the popular GNU C Compiler (GCC) for the Linux operating system are available:

The first one, StackGuard (see [6] and [8]), changes

1. the prolog of every function to place a so called *canary* value just below the saved return address, as shown in Figure 8 (compare to Figure 4!) in the hatched area and
2. checks in the function's epilog that this value hasn't been changed during the execution of the function. If the value has been changed, this will be flagged as a possible stack smashing attack before any malicious code will be executed.

The effectiveness of this approach relies on the fact that it is not possible for the attacker to guess the canary value and incorporate it into the attack code. To prevent this (or more precisely, to make this highly unlikely), two different options for canaries are employed. The first one is based on the usage of common string terminators of the C programming language, while the second one is based on randomization of the canary value (see [8] for details). Note that an additional layer of defense in an updated StackGuard version (see [9]) is reached by the fact that the randomized canary value depends as well on the return address which it should protect[14]. This enables the detection of changes to the return address that are not based on a (direct) buffer overflow, like the one discussed in [6].

[14] For those after the gory details: This is done by XORing the canary value with the return address both in the prolog and epilog of the function.

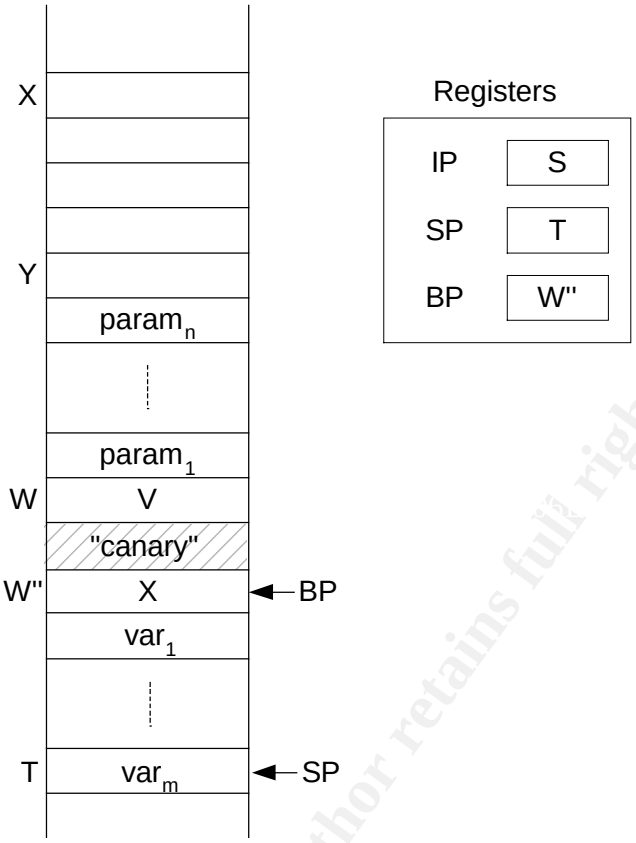


Figure 8: Stack after function prolog with StackGuard protection

The second GCC enhancement, Stack Shield (see [6]), basically stores away the value of the return address in the function prolog, and copies it back to the original memory location in the epilog, thus ensuring that the execution resumes in the calling function.

In the recent[15] version 7 of its Visual C/C++/.Net compiler, Microsoft has incorporated a /GS compiler option that can be seen as an adaptation of the StackGuard approach (though Microsoft claims that "both technologies were invented independently.", see [5]). A minor difference is that the so called *cookie* (equivalent to the canary in StackGuard) is placed before the saved base pointer and the saved return address, i.e. in analogy to Figure 8 the sequence would be "cookie, X , and V. See [4] for more details.

A comprehensive countermeasure exists on operating system level: **Disable execution of <u>any</u> code in the stack memory segment**, meaning that jumping to any shellcode put there will result in forced program termination. This option is available for a number of Unix flavors, including Linux (see either [31] or [32]) and Solaris (see [21]), but notably not for Microsoft's Windows operating systems[16].

As discussed in [8] and section 14.1 of [13], a non-executable stack has some side effects, which may however be negligible compared to the security improvement gained.

Note that neither the compiler enhancements nor the non-executable stack will be able to prevent general data corruption (see case a) on page 5).

[15] Released in February 2002
[16] Note that the "Code Red" .ida worm relied on an executable stack (see [22])!

7.2.3 Software users

Software users should, apart from favoring products from software companies that give security a high priority,

- frequently check for product patches and updates that eliminate vulnerabilities in programs
- subscribe to security information sources both from software suppliers and 3rd parties to keep up to date with detected vulnerabilities and act accordingly.

8 Off-by-one or frame pointer overwrite buffer overflows

8.1 Exploitation Technique

As we have seen in the last section, stack smashing vulnerabilities typically result from inappropriate or completely missing bounds checking on buffer data. As a consequence the exploit code has no predefined length limit[17].

Another common programming error in C, known as *off-by-one error*, is to exceed the buffer size by just one byte. Typically this happens in loops that try to process all buffer elements[18]. So this can duly be regarded as the minimal possible buffer overflow!

On a cursory look one might be tempted to assume that nothing reasonable can be done with this single byte. In fact, though, even this situation bears exploitation potential, as shown in reference [15], which provides the basis for the remainder of this section.

Imagine a scenario where the first local variable in the stack frame is a buffer, which is subject to an off-by-one error while reading or processing user input. If no padding occurs (meaning that there are some extra bytes added to the buffer due to the alignment of stack values on word boundaries), then this extra byte might – cf. Figure 4 – overflow one byte of the saved based pointer of the previous stack frame, the value X in Figure 4.

The good news is that the saved return address, the value V in Figure 4, is way out of reach, so there is no immediate way to execute any shellcode which might have been injected in the buffer.

The bad news is that only a small detour is required to achieve this goal. Let's first note that the x86 architecture is a little endian architecture, meaning that the four bytes that make up a word are stored in such a way that the byte with the lowest significance comes first or has the lowest address, respectively. If we take an analogy to the decimal system and imagine that numbers would be stored digitwise, then little endian would mean that 1234 would be stored with the digit 4 in the lowest and 1 in the highest memory location.

Let's further assume that the off-by-one overflow happens in a function `bad_func()` that is called from the function `good_func()`. Then the overflow enables us to change the **lowest** byte (corresponding to the digit 4 in our example) of the saved base pointer of `good_func()` in the previous stack frame. Why is this important? Imagine that the byte order would be reverse (which is, incidentally, called big endian). Then any modified value of the base pointer would point to an address way off the current context (in our example, the nearest values would be either 234 or 2234), which would most probably lie outside of

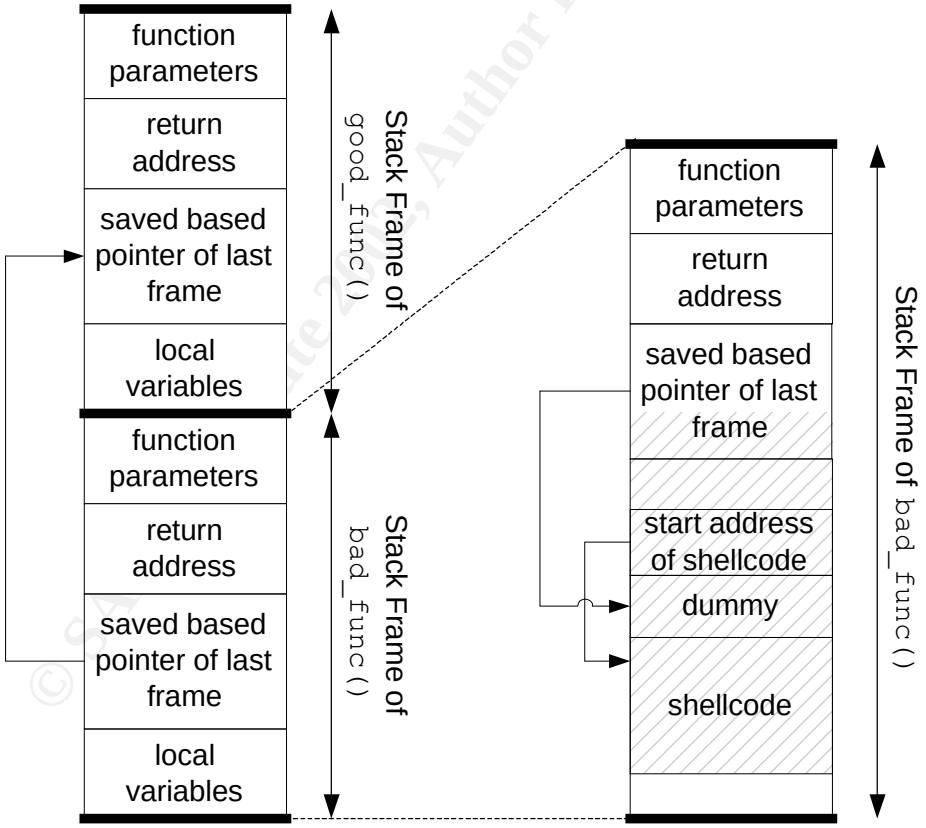
[17] Note, though, that smaller overflows have a greater chance to succeed since they reduce the risk to leave the program's memory space and end up with a memory access violation instead of a successful exploit.

[18] The most common example is to get the terminating condition for a `for` loop wrong, that is to use `for(i=0;i<=BUFSIZE;i++)` (note the <= operator!) instead of using the standard idiom `for(i=0;i<BUFSIZE;i++)` for processing all elements of the buffer.

the memory space of the running binary. But with the lowest byte at our discretion (meaning that the resulting number in our example would be 123x), we have a good chance to change the value of the base pointer to an address that is under our control, namely inside the buffer we are overflowing.

Since, as explained above, the base pointer is the reference to access both parameters and local variables of the function, the first result of changing the base pointer for the stack frame of `good_func()` will be that the instructions in `good_func()` following the call to `bad_func()` up to the end of the function will operate on the wrong data and thus produce garbage. If the instructions include pointer operations, the odds are that the binary will throw in the towel due to trying to access data outside its memory space (cf. case b) on page 5). From the security perspective, this must again be considered the better end, since the alternative is that we reach the function epilog of `good_func()`. As described on page 13 (see as well Figure 5), the third step of the epilog, the final pop instruction, copies faithfully the value just above the base pointer of the stack frame back to the instruction pointer!

So the only thing necessary to execute our shellcode in the overflowed buffer[19] is to change the value of the saved based pointer of `good_func()` to point to an address just one word below a memory location in the buffer where our (guestimated) start address for the shellcode is.



a) Before buffer overflow

b) After buffer overflow - saved base pointer is redirected to execute shellcode

[19] Provided that it hasn't been changed or overwritten in the meantime!

Figure 9: Exploiting off-by-one errors

Figure 9 illustrates the exploit, using arrows to indicate where memory addresses point to before and after the exploit, and indicating the buffer overflow area again by hatching.

8.2 Countermeasures

The countermeasures against this overflow attack are basically the same as for stack smashing attacks (see page 16), but it is at least one order of magnitude more difficult for programmers to take precautions or to spot vulnerabilities while reviewing code.

Note that the technical countermeasures – StackGuard, Stack Shield, Microsoft's /GS compiler switch and non-executable stack – are effective against this attack as well.

Incidentally, in this case the minor difference of canary/cookie setting between StackGuard and the /GS switch will result in earlier detection by the latter, namely already in the epilog of `bad_func()`, since instead of the targeted saved base pointer the cookie value will be changed, compared to a failed canary check for StackGuard in the epilog of `good_func()`.

9 Return-into-libc buffer overflows

In the light of the discussion in the sections 7.2.2 and 8.2 it looks as if non-executable stacks are a true bastion against any kind of buffer overflow exploits. You don't rely on the use of special compilers or compiler options, but just assert "Thy stack shalt not be executable!".

Sadly enough, this won't help much against the buffer overflow exploitation technique described in this section[20].

9.1 Exploitation Technique

Let's have again a look at Figure 3, which shows the stack just before the call to `my_func()`. From the perspective of `my_func()`, or in fact of any other called function, the stack contains first the return address it should use, and then the parameters as expected by the function. As we discussed in the section on the stack smashing exploit, the ultimate target of such an attack is to run shellcode to obtain a shell. Typically such shellcode relies on calling functions like `system()` or `execve()` on Unix systems or `WinExec()` on Windows systems[21]. These functions basically take as parameter a program name and/or path (more precisely, the memory address of such a string) and run it on behalf of the calling binary.

So an alternative to the direct stack smashing approach is to just fake the call to the desired function directly, which has the definite advantage that it can't be prevented by a non-executable stack, since the called function is a perfectly reasonable code address!

To accomplish such a fake, it suffices to overwrite the saved return address with the address of the targeted library function, insert an arbitrary dummy value for the return address of the function call just afterwards and adding the parameters for the library function (and indeed overwrite the (remaining) parameters of the original function with them).

[20] To be precise, this **alone** won't help much. See [30] and [31] for a combination of measures including non-executable stack that will prevent such exploits on Linux.

[21] Note that the name "return-into-libc" for this variant of buffer overflows stems from the fact that the quoted Unix functions are found in the libc system library.

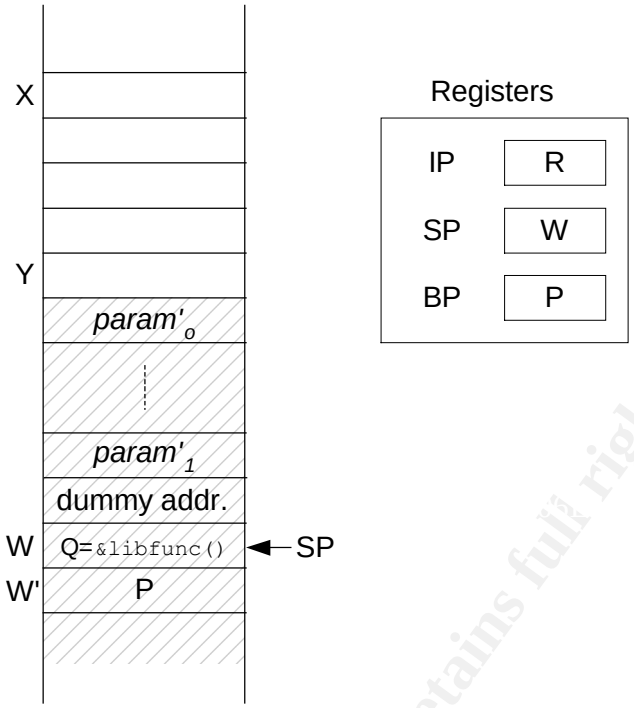


Figure 10: Return-into-libc exploit

Figure 10 shows the stack and register status after a return-into-libc buffer overflow (the extent of which is again indicated by hatching) just after step 2 of the function epilog – compare to the original situation in Figure 5! The start address Q of the targeted function `libfunc()` will be put into the instruction pointer by step 3 of the epilog – the pop instruction at address R. Note that compared to the situation in Figure 5 the parameters for the call to `libfunc()` are shifted upwards by one word due to the need to put a dummy return address for it on the stack. An interesting observation is, moreover, that the forged value P of the base pointer will be put again on the stack by the prolog of `libfunc()`.

Depending on the operating system, architecture and targeted function, obtaining/guessing the right address and putting the right parameters on the stack might not be particularly easy, but it can definitely be done.

References [20] and [30] provide detailed information on the gory details for Unix, and [18] an example exploit for Windows.

9.2 Countermeasures

Apart from the fact that you can't rely only on a non-executable stack as single protection measure, the same countermeasures as discussed in section 7.2 apply.

Actually the two implementations of non-executable stacks mentioned in section 7.2.2 contain already reinforcements to protect against return-into-libc exploits:

The Openwall kernel patch (see [30] and [31]) added a protection layer by ensuring that function addresses contain a zero byte, which prohibits buffer overflow attacks based on string functions, as the zero byte (interpreted as NULL) will stop the overflowing, meaning in this case that at the worst a function without any parameters can be called, which shouldn't pose a real threat to the system.

The PaX Linux kernel patch (see [32]) has added a feature that basically changes the address of library functions each time a binary is run. Although this approach certainly increases the security, it is not impossible to defeat it, as [20] has shown.

So we need in both cases to be aware of the fact that no measure can provide absolute security (or, more precisely, that there is no such thing as "absolute security"), but that a Defense-in-Depth approach is required.

10 Heap Overflows

10.1 Exploitation Technique

This paper would be incomplete without at least touching on the topic of heap overflows. Recall from section 6.4 that a program can request memory of a certain size on the heap by means of the `malloc()` function and needs to return this memory later on by the `free()` function.

Since you can put buffers on the heap as well, buffer overflows are again a concern. The implications of buffer overflows and the way to exploit them are, however, different from their stack related siblings. In particular there is neither a base pointer nor a saved return address to be found that could be (ab)used to directly (or in two stages, like in the frame pointer overwrite attack) jump to code the attacker has supplied.

Unfortunately this doesn't preclude exploits, see e.g. the recent problems with Microsoft's Internet Information Server, described in [14].

The basic scenario is again that you have a buffer, which is not properly protected by bounds checking, and some other interesting data of the program allocated behind this buffer's allocated memory space, that you would like to fill with data of your choice.

The "classical" paper on heap overflows, the first one that published actual exploits, is reference [7], which states,

> "When we refer to a 'heap-based overflow' in the sections below, we are most likely referring to buffer overflows of both the heap and data/bss sections."

So we need to take the data/bss section into account as well[22]! The paper presents first an example where the overflow was used to replace the string representing the name of a temporary file to which the program was writing. An exploit could e.g. provide the name of a configuration file for a security control like a firewall or an intrusion detection system. Depending on the level of input validation and/or error checking in the targeted application, this could lead to subsequent security breaches. More opportunities manifest if an attacker could on top of this influence the data written to the file. [7] demonstrates this by compromising the .rhosts file, which is used to establish trust relationships between Unix hosts.

The next example in [7], that comes a good deal closer to the stack overflow techniques described above, is to overwrite a *function pointer*. A function pointer contains the start address of a function. The type declaration of the function pointer in the program specifies the number and type of parameters the function expects and processes, and what type of return result, if any, it will deliver. Such function pointers are handy if you have a number of functions available that perform the same task, but would like to be able to specify the

[22] Recall, however, from section 6.2 that these memory segments are by default non-executable and are hence not suitable for placing shellcode!

actual function that should be used on the input data at run-time. Typical examples are sorting routines, since each sorting algorithm has its specific strengths and weaknesses.

An attacker, however, doesn't care much for the parameters etc. The feature of interest is that eventually the binary will jump to the address provided in the function pointer. So she will try to overwrite this pointer with the guestimated address of the shellcode that she inserted in the buffer (or some other memory location). Note that the techniques described in the section on stack smashing above (see Figure 7), can again be used, namely to increase the probability of a successful exploit by repeating the start address of the shellcode several times and prepending the real shellcode with NOP operations.

An extension of the function pointer overwrite technique to the C++ programming language is to overwrite or replace, respectively, an array of pointers holding the addresses of C++'s equivalent of functions, *class methods*. See [27] for more details.

Moreover the actual implementation of the heap memory management may be vulnerable to buffer overflows attacks as well. This is due to the storage of management information with or in between the chunks of memory allocated for a binary. If this management information is overwritten with carefully crafted data, eventually arbitrary memory locations may be overwritten, including again return addresses and function pointers, thus again enabling the execution of arbitrary code on the binary's behalf. Please see [2] for more details, which are way beyond the scope of this paper.

The same applies to another approach for overwriting arbitrary memory that is based on the so called *format string vulnerability* found in the default implementations of the `printf()` function and its relatives. See [29] for more details.

10.2 Countermeasures

From the programming and user perspective, the same countermeasures as discussed in the section on stack smashing apply again.

The appropriate technical measure to protect against code execution on the heap, is, not surprisingly, to make the heap non-executable. In this respect the PaX kernel patch for Linux systems (see again [32]) stands out, since it offers not only a non-executable stack, but can also render the heap non-executable!

Note though, that being able to overwrite a function pointer of suitable type **and** being able to tamper with the parameters used to call this function may still compromise a system hardened with PaX in a return-into-libc like fashion.

11 Conclusion

We have discussed several types of buffer overflow exploitation techniques, starting from simple stack smashing over frame pointer overwrites and return-into-libc to heap overflows, finally touching on advanced techniques how to use either heap management or format string vulnerabilities to write to arbitrary memory locations.

We have seen that certain technical countermeasures exist. Notably the Open Source Community has contributed to this with enhancements to the Linux Operating System and the GNU C Compiler. Microsoft has finally made a very first step to catch up with this by introducing protection against stack overflow exploits by means of the /GS compiler switch for its Visual C/C++/.NET compiler. It remains to be seen whether Microsoft's "Trustworthy Computing Initiative" will cover this area and trigger more support!

Although technical countermeasures surely help to diminish the threats posed by buffer overflows and should certainly be applied as part of the Defense-in-Depth approach, it must be clearly stated that the root cause for buffer overflows are programming errors and negligence. This can to a certain degree be attributed to the missing awareness of software developers regarding the problems buffer overflows can pose, and to laziness and normal error rates of developers.

To be fair, however, it should be pointed out that both software development companies and their customers are currently mainly interested in software functionality and short version cycles. The resulting enormous pressure on developers to deliver required functionality according to an "aggressive" schedule inevitably compromises quality – which includes security. As long as this attitude continues and (inherent) security of products is at most a nice-to-have feature instead of a strong plus in competition, we'll probably largely stick to the detect-and-fix loop for buffer overflows instead of tackling the roots of the problem.

So your vote counts – cast your ballot for prevention instead of cure!

12 Acknowledgements

The author thanks William T. Abrams for his helpful comments on proofreading this paper!

13 References

[1] "Aleph One." "Smashing The Stack For Fun And Profit." Phrack Magazine. Volume 7, Issue 49, File 14 of 16. 8 November 1996. URL: http://phrack.org/show.php?p=49&a=14 (24 April 2002).

[2] "anonymous." "Once upon a free()..." Phrack Magazine. Volume 11, Issue 57, File 9 of 18. 11 August 2001. URL: http://phrack.org/show.php?p=57&a=9 (29 April 2002).

[3] Avaya Labs Research. "Libsafe: Protecting Critical Elements of Stacks." 23 April 2002. URL: http://www.research.avayalabs.com/project/libsafe/ (28 April 2002).

[4] Bray, Brandon. "Compiler Security Checks In Depth". MSDN Article. February 2002. URL: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dv_vstechart/html/vctchcompilersecuritychecksindepth.asp (27 April 2002).

[5] Bray, Brandon. "In response to alleged vulnerabilities in Microsoft Visual C++ security checks feature." BugTraq Posting. 14 February 2002. URL: http://online.securityfocus.com/archive/1/256365 (27 April 2002).

[6] "Bulba and Kil3r (Mariusz Woloszyn)." "Bypassing StackGuard and Stack Shield." Phrack Magazine. Volume 10, Issue 56, File 5 of 16. 1 Mai 2000. URL: http://www.phrack.org/show.php?p=56&a=5 (26 April 2002).

[7] Conover, Matt. w00w00 Security Team. "w00w00 on Heap Overflows." January 1999. URL: http://www.w00w00.org/files/articles/heaptut.txt (28 April 2002).

[8] Cowan, Crispin. Wagle, Perry. Pu, Calton. Beattie, Steve. Walpole, Jonathan. "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade." 22 November 1999. URL: http://downloads.securityfocus.com/library/discex00.pdf (26 April 2002).

[9] Cowan, Crispin. "StackGuard Mechanism: Emsi's Vulnerability." 6 June 2000. URL: http://immunix.org/StackGuard/emsi_vuln.html (29 April 2002).

[10] "dark spyrit." "Win32 Buffer Overflows (Location, Exploitation and Prevention)." Phrack Magazine. Volume 9, Issue 55, File 15 of 19. 9 September 1999. URL: http://www.phrack.org/show.php?p=55&a=15 (25 April 2002).

[11] "DilDog." "Creating our Jumptable." Section in "The Tao of Windows Buffer Overflow." 1999. URL: http://www.cultdeadcow.com/cDc_files/cDc-351/page8.html (26 April 2002).

[12] "DilDog." "The Tao of Windows Buffer Overflow." 1999[23]. URL: http://www.cultdeadcow.com/cDc_files/cDc-351/ (25 April 2002).

[13] Fayolle, Pierre-Alain. Glaume, Vincent. "A Buffer Overflow Study - Attacks & Defenses." 27 March 2002. URL: http://www.enseirb.fr/~glaume/bof/report.html (28 April 2002).

[14] Hernan, Shawn V. "Microsoft Internet Information Server (IIS) vulnerable to heap overflow during processing of crafted '.htr' request by 'ISM.DLL' ISAPI filter." CERT Vulnerability Note VU#363715. 10 April 2002. URL: http://www.kb.cert.org/vuls/id/363715 (29 April 2002).

[15] "klog." "The Frame Pointer Overwrite." Phrack Magazine. Volume 9, Issue 55, File 8 of 19. 9 September 1999. URL: http://www.phrack.org/show.php?p=55&a=8 (26 April 2002).

[16] Kolde, Jennifer et. al. "The SANS Windows Security Digest - A Resource for Computer and Network Security Professionals." Volume 5, Number 3. Section 6.1.

[17] LaRock Decker, Nicole." Buffer Overflows: Why, How and Prevention." SANS Reading Room: Threats & Vulnerabilities. 13 November 2000. URL: http://rr.sans.org/threats/buffer_overflow.php (25 April 2002).

[18] Litchfield, David "Non-stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP." NGSSoftware Insight Security Research. 5 March 2002. URL: http://www.nextgenss.com/papers/non-stack-bo-windows.pdf (27 April 2002).

[19] "Mudge." "How to write Buffer Overflows." 20. October 1995[24]. URL: http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html (25 April 2002).

[20] "Nergal." "The advanced return-into-lib(c) exploits." Phrack Magazine. Volume 11, Issue 58, File 4 of 14. 28 December 2001. URL: http://www.phrack.org/show.php?p=58&a=4 (28 April 2002).

[21] Noordergraaf, Alex. Watson, Keith. "Solaris™ Operating Environment Security." Sun BluePrints™ OnLine. January 2000. URL: http://www.sun.com/software/solutions/blueprints/0100/security.pdf (28 April 2002).

[22] Permeh, Ryan. Maiffret, Marc. "Detailed analysis of the 'Code Red' .ida worm." 19 July 2001. File "Code-Red-Analysis.txt" in URL: http://www.eeye.com/html/advisories/codered.zip (27 April 2002).

[23] According to reference [17], the document was published April 1998. The Security Focus ONLINE Library Archive, however, states under http://online.securityfocus.com/library/766 a publishing year of 1999, which we adopted.

[24] Note that reference [17] actually states that this document was published in 1997, providing a different URL that is not valid anymore. The URL provided here states indeed both by document date and copy right notice that the document was at least first published in 1995.

[23] Raymond, Eric (Ed.). “Terms : The S Terms : stack .“ The Jargon Dictionary. Version 4.2.2. 20 August 2000. URL: http://info.astrian.net/jargon/terms/s/stack.html (24 April 2002).

[24] Raynal, Frédéric; Blaess, Christophe; Grenier, Christophe. “Avoiding security holes when developing an application - Part 2: memory, stack and functions, shellcode.” 27 April 2001. URL: http://www.linuxfocus.org/English/March2001/article183.shtml (22 April 2002).

[25] Raynal, Frédéric; Blaess, Christophe; Grenier, Christophe. “Avoiding security holes when developing an application - Part 3: buffer overflows.” 1 Mai 2001. URL: http://www.linuxfocus.org/English/May2001/article190.shtml (26 April 2002).

[26] “rix.” “Writing ia32 alphanumeric shellcodes.” Phrack Magazine. Volume 11, Issue 57, File 15 of 18. 11 August 2001. URL: http://www.phrack.org/show.php?p=57&a=15 (26 April 2002).

[27] “rix.” “Smashing C++ VPTRS.” Phrack Magazine. Volume 10, Issue 56, File 8 of 16. 1 Mai 2000. URL: http://phrack.org/show.php?p=56&a=8 (29 April 2002).

[28] Rogers, Larry. “Buffer Overflows – What Are They and What Can I Do About Them?” CERT Security Improvement Features. 5 April 2002. URL: http://www.cert.org/nav/index_green.html#feature (25 April 2002).

[29] “scut / team teso.” “Exploiting Format String Vulnerabilities.” Version 1.2. 1 September 2001. URL: http://teso.scene.at/articles/formatstring/ (29 April 2002).

[30] “Solar Designer.” “Getting around non-executable stack (and fix).” BugTraq Posting. 10 August 1997. URL: http://online.securityfocus.com/archive/1/7480 (27 April 2002).

[31] “Solar Designer.” “Linux kernel patch from the Openwall Project.” URL: http://www.openwall.com/linux/README (28 April 2002).

[32] “The PaX Team.” “Homepage of The PaX Team.” 2 April 2002. URL: http://pageexec.virtualave.net (28 April 2002).

[33] “twitch.” “Taking advantage of non-terminated adjacent memory spaces.” Phrack Magazine. Volume 10, Issue 56, File 14 of 16. 1 Mai 2000. URL: http://www.phrack.org/show.php?p=56&a=14 (26 April 2002).

[34] Walker, Henry M. “Limits of Computing.” Course CS105: “Problem Solving via Computer Programming” of the University of Richmond, VA. May 1999. URL: http://www.mathcs.richmond.edu/~jkent/courses/cs105_may_1999/limits.pdf (25 April 2002).

Last Updated: December 15th, 2017

Upcoming SANS Training

Click Here for a full list of all Upcoming SANS Events by Location

SANS Security East 2018	New Orleans, LAUS	Jan 08, 2018 - Jan 13, 2018	Live Event
SANS Amsterdam January 2018	Amsterdam, NL	Jan 15, 2018 - Jan 20, 2018	Live Event
Northern VA Winter - Reston 2018	Reston, VAUS	Jan 15, 2018 - Jan 20, 2018	Live Event
SEC599: Defeat Advanced Adversaries	San Francisco, CAUS	Jan 15, 2018 - Jan 20, 2018	Live Event
SANS Dubai 2018	Dubai, AE	Jan 27, 2018 - Feb 01, 2018	Live Event
SANS Las Vegas 2018	Las Vegas, NVUS	Jan 28, 2018 - Feb 02, 2018	Live Event
Cyber Threat Intelligence Summit & Training 2018	Bethesda, MDUS	Jan 29, 2018 - Feb 05, 2018	Live Event
SANS Miami 2018	Miami, FLUS	Jan 29, 2018 - Feb 03, 2018	Live Event
SANS Scottsdale 2018	Scottsdale, AZUS	Feb 05, 2018 - Feb 10, 2018	Live Event
SANS London February 2018	London, GB	Feb 05, 2018 - Feb 10, 2018	Live Event
SANS Southern California- Anaheim 2018	Anaheim, CAUS	Feb 12, 2018 - Feb 17, 2018	Live Event
SANS Secure India 2018	Bangalore, IN	Feb 12, 2018 - Feb 17, 2018	Live Event
SANS Dallas 2018	Dallas, TXUS	Feb 19, 2018 - Feb 24, 2018	Live Event
SANS Brussels February 2018	Brussels, BE	Feb 19, 2018 - Feb 24, 2018	Live Event
SANS Secure Japan 2018	Tokyo, JP	Feb 19, 2018 - Mar 03, 2018	Live Event
Cloud Security Summit & Training 2018	San Diego, CAUS	Feb 19, 2018 - Feb 26, 2018	Live Event
SANS New York City Winter 2018	New York, NYUS	Feb 26, 2018 - Mar 03, 2018	Live Event
CyberThreat Summit 2018	London, GB	Feb 27, 2018 - Feb 28, 2018	Live Event
SANS London March 2018	London, GB	Mar 05, 2018 - Mar 10, 2018	Live Event
SANS San Francisco Spring 2018	San Francisco, CAUS	Mar 12, 2018 - Mar 17, 2018	Live Event
SANS Secure Osaka 2018	Osaka, JP	Mar 12, 2018 - Mar 17, 2018	Live Event
SANS Secure Singapore 2018	Singapore, SG	Mar 12, 2018 - Mar 24, 2018	Live Event
SANS Paris March 2018	Paris, FR	Mar 12, 2018 - Mar 17, 2018	Live Event
SANS SEC460: Enterprise Threat Beta	OnlineCAUS	Jan 08, 2018 - Jan 13, 2018	Live Event
SANS OnDemand	Books & MP3s OnlyUS	Anytime	Self Paced