HARMONY SECURITY
RESEARCH AND CONSULTANCY

www.harmonysecurity.com
info@harmonysecurity.com

Buffer Overflows

Defending against arbitrary code insertion and execution

By Stephen Fewer

Contents

1 Introduction

Buffer Overflows are one of the most common and potentially deadly forms of attack against computer systems to date. They allow an attacker to locally or remotely inject malicious code into a system and compromise its security. They arise out of poor programming practices and buggy software. First seen in the Internet Worm of 1987 they can occur on all major platforms (UNIX, Linux, Win32, and Solaris) and all major architectures (Intel, Alpha and MIPS). There are in fact many variations of buffer overflows but this paper will deal with the most basic and common type, i.e. stack based overflows. Other types include heap-based overflows, frame pointer overflows and adjacent memory space overflows. These however are beyond the scope of this paper. I will begin with observing how this vulnerability arises before assessing what methods are used to prevent such occurrences from being able to happen. Throughout this paper I will be using C code examples and x86 Assembly Language examples (coded in the Intel syntax, the AT&T syntax is messy). All examples are intended to work on a Win32 system but should be portable to Linux unless otherwise stated.

1.1 Where does the problem lie?

To understand and prevent this type of vulnerability we need to fully understand where the weaknesses in the system are. There are two key elements involved here:

- The stack and function calling.
- Memory I/O with no bounds checking.

1.1.1 The Stack

An executing program, further referenced in this paper as a process, uses the stack

primarily for passing arguments (pointers, variables) and function calling. Variables can be held in the stack but are also stored in other locations of a process depending how they are instantiated or allocated (Refer to section 4 Process Segments for a list of possibilities). The stack is an abstract space that uses a method known as Last in First out (LIFO). This means the last thing to be pushed onto the stack can be retrieved by popping it out. Pushing and Popping will automatically grow and shrink the stack. It is important to remember the stack grows downwards in memory as it gets bigger (see Figure 1). This is because the stack works backwards, the bottom of it is located at a higher memory address then the top.

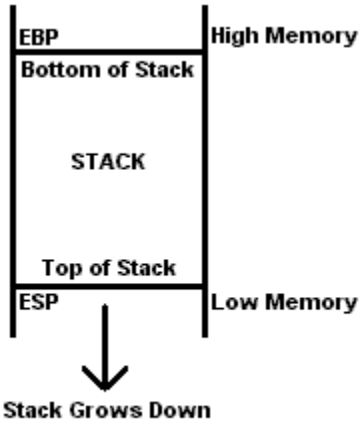

Figure 1

Variables declared inside a function are allocated space within the stack frame of that function. To understand what a stack frame is and how memory is allocated for a variable we can use the example C code in Listing 1 and its assembly language equivalent in Listing 2.

```
1: int main ()
2: {
3: char buffer[16];
3: return 0;
4: }
```

Listing 1

```
1: push ebp
2: mov ebp, esp
3: sub esp, 0x10
4: xor eax, eax
5: mov esp, ebp
6: pop ebp
7: ret
```

Listing 2

In Listing 1 we see a very simple C program which when run will allocate 16 bytes on the stack to a variable called buffer and then exit. This does not give us much information on how the stack is used so we must look into its assembly equivalent. Listing 2 shows us the same program, which will do exactly the same thing when run, but here we can see exactly what is going on inside the process space.

There are three registers used here, the base pointer (EBP), the stack pointer (ESP) and the accumulator register (EAX). The EBP is a static pointer within a stack frame and can be used to reference variables held within the stack. The ESP is a dynamic register that will grow and shrink as variables are allocated and de-allocated on the stack (Refer to Figure 1). The EAX is used normally to return values from a function. The first two lines are called the procedure prelude; this is what sets up a new stack frame. Line one saves the old base pointer by pushing it onto the stack. Line two moves the current stack pointer into the base pointer. Now a new stack frame has been set up. By executing line two the base of this new stack frame begins at the top of previous stack. Thus a stack frame can be considered a separate stack for the current function held on top of the callers stack

frame. Line three is where the 16 bytes are allocated for the variable "buffer". It increases the stack size by subtracting 16 bytes from the stack pointer (remember that the stack grows down and the ESP points to a lower memory address than the EBP, this is why we subtract the required amount). If we were to reference this variable we would point to its address in memory by referring to the EBP (this is possible because the base pointer is static.). For example: lea eax, [ebp-0x10]. If we executed this line EAX would point to our variable. Line four sets the functions return value to zero by xoring the EAX with itself, thus reducing it to zero. Lines five and six are known as the procedure prologue. Line five destroys the current stack frame by moving the base pointer into the stack pointer. This will set the ESP back to its original value before the function was called. Line six restores the EBP to its original value before the function was called. Finally in line seven the function after restoring the ESP and EBP returns.

Listing 3 (shown below) and its assembly equivalent in Listing 4 shows us a practical example of stack frames being created and destroyed as function foo() is called and returns. A quick overview of what happens is a new stack frame is created when main() is called. As foo() is called a separate stack frame is arranged on top of main()'s one.

```
1: void foo()
2: {
3: return;
4: }
5: int main ()
6: {
7: foo();
8: return 0;
9: }
```

Listing 3

```
1: push ebp
2: mov ebp, esp
3: pop ebp
4: ret
5: push ebp
6: mov ebp, esp
7: call 1
8: xor eax, eax
9: pop ebp
10: ret
```

Listing 4

It is extremely important to note that when a call is performed in assembly the current Instruction Pointer (EIP) is pushed onto the stack. The EIP is a pointer to the current line of code being executed and is the key to execution redirection. Thus a call in assembly is the same as:

```
push eip ; pushes the current EIP
onto the stack
jmp X ; jump to the function at
address X
```

This allows for a function to return to its caller as it can restore the old EIP. If we can overwrite the saved EIP we can redirect the processes execution flow to anywhere we want (e.g. our own malicious shellcode). The function foo() returns by using the saved EIP that was pushed onto the stack by the call in line seven. The function foo removes its stack frame in line two of Listing 4 and restoring main()'s stack pointers (EBP and EBP), thus restoring main()'s stack frame. When a return is performed in assembly it is the same as:

```
mov edx, dword[ebp+0x04] ; moves
the saved EIP into EDX
jmp edx ; jumps to the address in
EDX
```

1.1.2 No Bounds Checking

Writing to memory can be hazardous if there is no procedure employed to check how much data is being written and where it is being written to. For example if we have a buffer in memory which is 16 bytes long and we write 24 bytes to it we will overwrite the 8 bytes at the end which don't belong to the buffer. Listing 5 demonstrates this with a complete example using a C program.

```
 1: #include <stdio.h>
 2: #include <string.h>
 3:
 4: char overflowbuffer[24] =
        "CCCCCCCCCCCCCCCCCCCCCCCC";
 5:
 6: int main()
 7: {
 8:
 9: char one[16] = "AAAAAAAAAAAAAAA";
10: char two[16] = "BBBBBBBBBBBBBBB";
11:
12: printf("before...\n");
13: printf("one: %s\n", one);
14: printf("two: %s\n\n", two);
15:
16: strcpy(two, overflowbuffer);
17:
18: printf("after...\n");
19: printf("one: %s\n", one);
20: printf("two: %s\n\n", two);
21:
22: return 0;
23: }
```

Listing 5

Upon compiling and executing this application you will receive the output as show in Listing 6. Up until line fifteen the processes stack will look like Figure 2. Keep in mind that the stack works backwards so the two strings appear reversed (the dot at the end of each string is a null character, 0x00, which is appended to the end of a string to indicate its termination.). After executing line sixteen, the variable overflow buffer gets copied into memory starting at the beginning address of the variable two. However the variable overflow buffer is eight bytes longer than the variable two and will overflow into the variable one's memory space. The processes stack will now look like Figure 3. Printing out the two variables again after executing line fifteen verifies this.

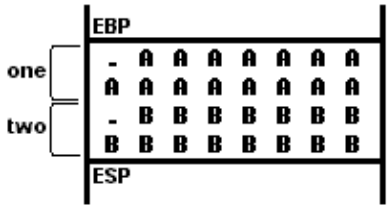

Figure 2

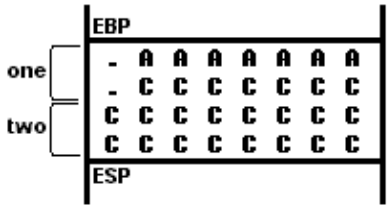

Figure 3

The reason we are able to do this is because most functions (notably in the C/C++ language) that write to memory perform no bounds checking. Also most compilers don't provide for bounds checking at compile time to warn developers. Therefore overflows are a very regular occurrence in real world applications. In the next section we will look how no bounds checking and the stack methodology can be used for malicious purposes.

```
before...
one: AAAAAAAAAAAAAAA
two: BBBBBBBBBBBBBBB
after...
one: CCCCCCC
two: CCCCCCCCCCCCCCCCCCCCCCCC
```

Listing 6

2 Smashing the Stack

Smashing the stack is the term used to refer to a stack based buffer overflow where the process execution flow is redirected into malicious shellcode. To do this we need to inject our own code into the processes memory space (usually into the current stack frame) and overwrite the saved EIP (which is located 4 bytes below the current frames EBP) to point to our own code so it will get executed. Listing 7 contains a program that will when executed overwrite the EBP and the EIP with our own values.

```
 1: #include <stdio.h>
 2: #include <string.h>
 4: void foo(char *ptr)
 5: {
 6: printf("\tinside foo()\n");
 7: char smallbuff[128];
 8: strcpy(smallbuff, ptr);
 9: printf("\tleaving foo()\n");
10: return;
11: }
12: int main()
13: {
14: printf("starting\n");
15: char bigbuff[160];
16: for(int i=0; i< 160; i++)
17: {
18: bigbuff[i] = 'A';
19: }
20: // These 4 bytes will overwrite
the saved EBP
21: bigbuff[128] ='B';
22: bigbuff[129] ='B';
23: bigbuff[130] ='B';
24: bigbuff[131] ='B';
25: // These 4 bytes will overwrite
the saved EIP
26: bigbuff[132] ='C';
27: bigbuff[133] ='C';
28: bigbuff[134] ='C';
39: bigbuff[135] ='C';
30: printf("calling foo()\n");
31: foo(bigbuff);
32: // We will never get here
33: printf("finished\n");
34: return 0;
35: }
```

Listing 7

Upon executing this code the processes instruction pointer will get overwritten to 0x43434343 (0x43 is 'C' in hexadecimal) and the base pointer will get overwritten to 0x42424242. For our own code to be executed we would insert it into the first 124 bytes of the buffer and point the EIP back into it.

3 Prevention

To render this class of vulnerability obsolete we must review what methods are most likely to succeed. There are two main options available.

- Protection at run time; kernel patches, modified run time environment.
- Protection at compile time; Modified compilers, secure functions.

3.1 Non Executable Stacks

By modifying the kernel we can set the stack segment of a process to become non-executable. Therefore if the execution flow was ever redirected inside the stack to execute code a general protection fault would occur causing the process to produce a core dump. Currently this method is only possible under operating systems such as UNIX and Solaris as system level modification is easily applied. Systems such as Windows NT would need a new kernel from Microsoft. This is a run time solution that does not solve the problem completely. The vulnerably process attacked will still crash out (like a denial of service attack) and ways around non-executable stacks have been discovered. It also ignores more advanced types of buffer overflows such as heap based overflows.

3.2 Random Stack Addresses

In order to execute malicious code injected into the stack the EIP must be pointed to the

stack, normally at ESP. If however we can make the stack address random every time a process runs, the attacker will not know where to point the instruction pointer. This is not completely true for Windows systems as it is often possible to point the EIP into a fixed address which contains a line of code similar to JMP ESP, which upon executing will jump to wherever the ESP is and begin executing from there. Therefore randomizing won't make a difference. This is only possible because DLL's are mapped to a fixed base address in a process memory space and it is easy to find an instruction such as JMP ESP to reference. Systems such as UNIX don't employ such a technique and, therefore, randomizing the stack address will help reduce the possibilities of an attack succeeding. This method can be employed at run time by the kernel or at compile time by the programmer. It still won't solve the problem and acts only as a deterrent, best used in conjunction with other methods.

3.3 Bounds Checking

As the heart of the vulnerability lies in being able to write past the bounds of a variable, one of the most effective methods to secure against this is to employ secure coding in the form of bounds checking. Bounds checking can be employed in the functions that access memory or in the compiler itself, which will modify the code as needed. An example of a function which uses bounds checking is the common string function strncpy(). It will only copy a certain amount of bytes unlike the function strcpy() which will copy as many as possible until a null byte is found. An example is given below.

```
strcpy(smallbuff, ptr); // insecure
coding!!!

strncpy(smallbuff, sizeof(smallbuff), ptr);
// secure coding :)
```

If bounds checking is employed at compile time the programmer can be warned of possible vulnerabilities and the code can be modified to protect against then. It is easier to employ this under platforms like Linux as they are open source and easily modified. An example is a modification for gcc (the Linux C/C++ compiler) which will compile insecure code and produce safer code. It is important to note that C/C++ is the most vulnerable language due to its ability to access and reference memory directly. Java completely avoids this entire class of vulnerability as it has sophisticated memory management algorithms that will not allow a user to overwrite areas of memory. However C/C++ is the most widely used development language and most operating systems are largely developed with it (e.g. UNIX, Linux and Windows).

4 Process Segments

When an application is compiled various parts are placed in certain segments depending what they are. They will then get mapped into a process space upon execution. There are five main segment types.

1. The stack segment is for the stack which is used for passing arguments and storing variables.

2. The code segment, also called the text segment, contains the compiled code.

3. The heap Segment is for variables dynamically allocated at run time, e.g.
   `char * buf = new char[malloc(i)];`

4. The Block Started by Symbol (BSS) Segment is for global variables un-initialized at compile time, e.g. `int i;`

5. The Data Segment is for global variables initialized at compile time, e.g. `int i=9;`

5 References

- Protecting Against Some Buffer Overrun Attacks - *Richard Kettlewell*
- Bounds Checking for C - *Richard Jones and Paul Kelly*
- Smashing The Stack For Fun And Profit - *Aleph One*
- Advanced buffer overflow exploit - *Taeho Oh*
- The Tao of Windows Buffer Overflow - *DilDog*
- Exploiting Windows NT 4 Buffer Overruns - *David Litchfield*