Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server.

David Litchfield
(david@ngssoftware.com)
8th September 2003

Abstract
This paper presents several methods of bypassing the protection mechanism built into Microsoft's Windows 2003 Server that attempts to prevent the exploitation of stack based buffer overflows. Recommendations about how to thwart these attacks are made where appropriate.

Introduction

Microsoft is committed to security. I've been playing with Microsoft products, as far as security is concerned, since 1997 and in the past year and a half or two I've seen a marked difference with some very positive moves made. In a way they had to. With the public relations crisis caused by worms such as Code Red Microsoft needed to do something to stem the flow of customers moving away from the Windows OS to other platforms. Microsoft's Trustworthy Computing push was born out of this and, in my opinion, I think we as consumers are beginning to see the results; or ironically not see them - as the holes are just not appearing as they would if the security push wasn't there. We have, of course, seen at least one major security hole appear in Windows 2003 Server, this being the DCOM IRemoteActivation buffer overflow discovered by the Polish security research group, the Last Stages of Delirium [http://www.lsd-pl.net]. We will see more; but I am confident that the number of security vulnerabilities that will be discovered in Windows 2003 Server will be a fraction of those found in Windows 2000. Acknowledging that there have been holes found and that, yes, more will come to light in the future this paper is going to look at how, currently, the stack based protection built into Windows 2003 Server to protect against buffer overflow vulnerability exploitation can be bypassed. The development of this mechanism is one of the right moves made in the direction of security.

An Overview of Windows 2003 Stack Protection

Windows 2003 Server was designed to be secure out of the box. As part of the security in depth model adopted by Microsoft for their latest Windows version a new stack protection mechanism was incorporated into their compiler that was intended to help mitigate the risk posed by stack based buffer overflow vulnerabilities by attempting to prevent their exploitation. Technically similar to Crispin Cowan's StackGuard, the Microsoft mechanism places a security cookie (or canary) on the stack in front of the saved return address when a function is called. If a buffer local to that function is overflowed then, on the way to overwriting the saved return address, the cookie is also overwritten. Before the function returns the cookie is checked against an authoritative version of the cookie stored in the .data section of the module where the function resides. If the cookies do not match then it is assumed that the buffer has been overflowed and the process is stopped. This security mechanism is provided by Visual Studio .NET - specifically the GS flag which is turned on by default.

Currently the stack protection built into Windows 2003 *can* be defeated. I have engineered two similar methods that rely on structured exception handling that can be used *generically* to defeat stack protection. Other methods of defeating stack protection are available, but these are dependent upon the code of the vulnerable function and involve overwriting the parameters passed to the function.

How is the cookie generated?

When a module is loaded the cookie is generated as part of its start up routine. The cookie has a high degree of randomness which makes cookie prediction too difficult, especially if the attacker only gets one opportunity to launch the attack. This code represents the manner in which the cookie is generated. Essentially the cookie is the result of a bunch of XOR operations on the return values of a number of functions:

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

int main()
{
        FILETIME ft;
        unsigned int Cookie=0;
        unsigned int tmp=0;
        unsigned int *ptr=0;
        LARGE_INTEGER perfcount;

        GetSystemTimeAsFileTime(&ft);
        Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;
        Cookie = Cookie ^ GetCurrentProcessId();
        Cookie = Cookie ^ GetCurrentThreadId();
        Cookie = Cookie ^ GetTickCount();
        QueryPerformanceCounter(&perfcount);
        ptr = (unsigned int)&perfcount;
        tmp = *(ptr+1) ^ *ptr;
        Cookie = Cookie ^ tmp;
        printf("Cookie: %.8X\n",Cookie);
        return 0;
}
```

The cookie is an unsigned int and once it has been generated it is stored in the .data section of the module. The .data section memory is writable and one of the attacks we'll look at later will involve overwriting this authoritative cookie with a known value and overwriting the stack cookie with the same value. A good recommendation to Microsoft would be to mark the 32 bits of memory where this cookie is stored as read-only using VirtualProtect. This will prevent the attack I've just mentioned.

An in-depth look at what happens if the cookies do not match.

If a local buffer is overflowed and the cookie is overwritten what happens? Well, when the check is performed and it fails the code checks to see if a security handler has been defined. This security handler exists so that the program's developers can take actions such as log things before terminating the process if they see fit. A pointer to this handler is stored in the .data section of the module. Later on we shall see how this can pose a problem. It is not necessary to define a security handler and, being honest, it is probably safer not to as any code that runs after the buffer is overflowed could be compromised. It is, in my opinion unsafe to do anything other than to trap to the kernel and terminate the process there and then. Even if no security handler has been defined there is still a raft of code that is executed by the process. The default steps taken if the cookies do not match are to set the UnhandledExceptionFilter to 0x00000000 and call the UnhandledExceptionFilter function. This starts the shutdown of the process. The UnhandledExceptionFilter function performs a number of tasks before the process is finally killed. Amongst other things, this function loads the faultrep.dll library and calls the ReportFault function. Every time you see a "Report this fault to Microsoft" popup - this is that function at work. As I said a few moments ago it is too dangerous to run any code after an overflow has occurred. Later we'll look at how even the actions performed by the UnhandledExceptionFilter function can be abused to gain control of an overflowed process. Whilst I am aware that the "Report a fault" has been invaluable to Microsoft and the bug fixing process, in my opinion, Microsoft should introduce a new kernel function, an NtTerminateProcessOnStackOverflow if you will, that deals with such situations and this should be called as soon as the cookie check fails. Subverting a kernel function from user mode is going to be difficult to say the least and provides less "opportunities" than running termination code in user mode.

Windows and Exception Handling

Exception handling is built into the Windows operating system and helps make applications more robust. When a problem occurs such as a memory access violation or divide by zero error an exception handler can be coded to deal with such situations and allow the process to recover and continue execution as normal. Even if the program developer has not set up any exception handling every thread of every process has at least one handler that is setup on thread initialization. Information about exception handlers is stored on the stack in an EXCEPTION_REGISTRATION structure and a pointer to the first EXCEPTION_REGISTRATION structure is stored in the Thread Environment Block.

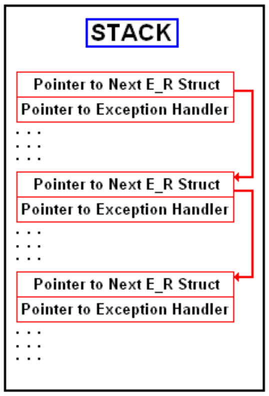
STACK
Pointer to Next E_R Struct
Pointer to Exception Handler
Pointer to Next E_R Struct
Pointer to Exception Handler
Pointer to Next E_R Struct
Pointer to Exception Handler

Figure 1.

As can be seen in Figure 1 the EXCEPTION_REGISTRATION structure has two elements, a pointer to the next EXCEPTION_REGISTRATION structure and a pointer to the exception handler. When overflowing stack based buffers on Windows platforms one of the mechanisms available to the attacker to gain control of the process is to overwrite this EXCEPTION_REGISTRATION structure – setting the pointer to the handler to a block of code or an instruction that will redirect the flow of execution back to the buffer.

Microsoft has recognized this as a problem and with Windows 2003 Server structured exception handling has been modified in an attempt to prevent such attacks. Exception handlers are now registered and the addresses of any handlers in a module are stored in the Load Configuration Directory of the module. When an exception occurs, before the handler is executed its address is checked against the list of registered handlers and it will not be executed if no match is found. If however the address of the handler, as taken from the EXCEPTION_REGISTRATION structure, is outside the address range of a loaded module then it is *still* executed. A final point to note is that if the address of the handler is on the stack then it will *not* be executed. This prevents attacks where the pointer to the exception handler is overwritten with a direct stack address. If the address of the “handler” is a heap address then it will be called however.

This provides the attacker with two good possibilities and a third, more difficult and code/process dependent option. On overflowing a local buffer if, as well as overwriting the cookie and the saved return address, an attacker overwrites the EXCEPTION_REGISTRATION structure then they can attempt to gain control of the process by either setting the pointer to the handler to an already registered handler and abusing that to gain control or alternatively, overwrite the pointer to the handler with an address that is outside the range of a loaded module and points to a block of code or instruction that allows the attacker to get back into their buffer. As we’ll see these two options provide two generic mechanisms for bypassing the stack protection of Windows 2003

Server. The third option is to “load” a buffer on the heap, if possible, with exploit code and overwrite the pointer to the handler with the address of the heap buffer.

Defeating Stack Protection through Structure Exception Handling

Before we can abuse structured exception handling as a mechanism to defeat stack protection we need to generate an exception before the cookie is checked. There are two ways this might happen – one we can control and the other is dependent of the code of the vulnerable function. In the latter case, if we overflow other data, for example parameters that were pushed onto the stack to the vulnerable function and these are referenced before the cookie check is performed then we could cause an exception here by setting this data to something that *will* cause an exception. If the code of the vulnerable function has been written in such a way that no opportunity exists to do this, then we have to attempt to generate our own exception. We can do this by attempting to write beyond the end of the stack. This will generate an exception. See Figure 2.

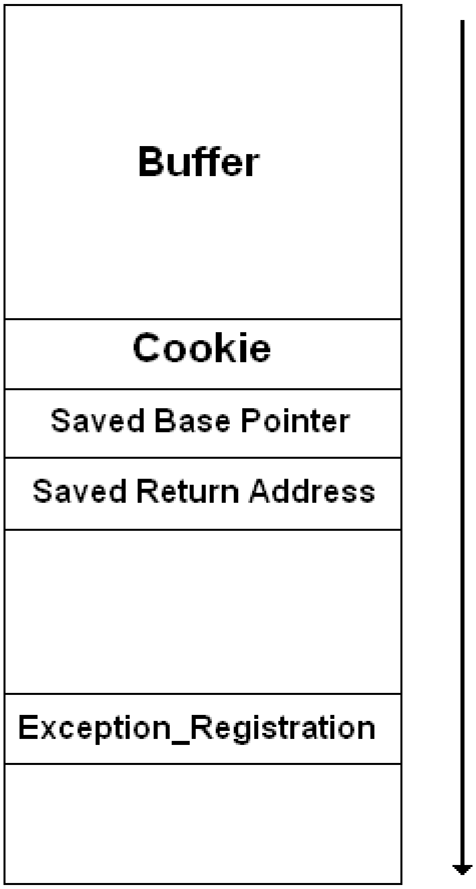

Figure 2.

Overwriting the pointer to the exception handler with an address outside the address range of a loaded module.

Recall that, before an exception handler is executed, its address is first checked against a list of registered handlers. If the address of the exception handler is not in the address range of any loaded module then it is called. Why this is so I do not know. My recommendation to Microsoft would be to not to do this. It provides an attacker with a method to bypass any protection. All an attacker needs to do is find an instruction or block of instructions that will redirect execution back into the user supplied data at an address not within the range of a module. If we examine the state of the stack and registers when an exception handler has been called we can see that none of the registers contain the address of anywhere in the user supplied data. If any did then all the attacker would need to do would be to find a “call register” or “jmp register” instruction – but none of the registers do point anywhere useful. If, however, we examine the state of the stack on the calling of an exception handler we can find many pointers to the EXCEPTION_RGISTRATION structure that has been overflowed:

ESP	+ 8	+ 14	+ 1C	+ 2C	+44	+50
EBP	+ 0C	+ 24	+ 30	- 04	- 0C	- 18

Table 1.

If an attacker can find an instruction that executes a

```
CALL DWORD PTR [ESP+NN]
CALL DWORD PTR [EBP+NN]
CALL DWORD PTR [EBP-NN]
JMP DWORD PTR [ESP+NN]
JMP DWORD PTR [EBP+NN]
JMP DWORD PTR [EBP-NN]
```

where NN is one of the offsets found in Table 1 above, at an address outside of the address range of a loaded module then it will be executed on the event of an exception. This will redirect the flow of execution to the “owned” exception registration structure. As the first 32 bits of this structure is a pointer to the next structure the attacker can load this with code that will perform a short jump over the pointer to the “handler” as in Figure 3.

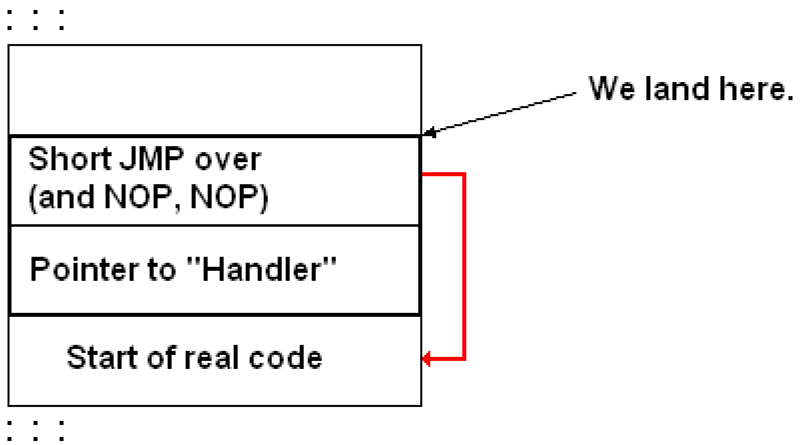

Figure 3.

The other alternative is to find a “pop reg, pop reg, ret” instruction block outside of the address range of a loaded module. This works as we can find a pointer to the EXCEPTION_REGISTRATION structure at ESP + 8. Two pops of the stack will leave this pointer at ESP so when the RET is executed we land back in the structure. Again we would need to perform the short jump. One can search the address space of a vulnerable process for such instruction blocks by injecting a small bit of code that will load a DLL that contains the code to do the search. See Appendix A. The output of the code in Appendix A shows three interesting possibilities where suitable instructions can be found outside of the address range of a loaded module. Firstly, there is a “CALL DWORD PTR[EBP+30h] at address 0x001B0B0B and two sets of “pop, pop, ret” at 0x7FFC0AC5 and 0x7FFC0AFC.

These three addresses are all outside the range of a loaded module. The CALL DWORD PTR[EBP+30h] at 0x001B0B0B isn’t actually an instruction. It’s just the right bytes that form this instruction – 0xFF5530. If we analyze the memory of a process using vadump.exe (Virtual Address Dump - http://www.microsoft.com/windows2000/techinfo/reskit/tools/existing/vadump-o.asp) we can see that this is within the address range of the memory mapped data file, UNICODE.NLS. Whilst UNICODE.NLS is mapped in some processes it is not in others. The base address that UNICODE.NLS is loaded at appears to be the same for each instance of the same given process, even across boots and across systems, but may vary from different process to process. For example UNICODE.NLS will always be mapped at 0x00260000 for a given a console application but at 0x001A0000 for svchost.exe. The CALL DWORD PTR[EBP+0x30] can be found 0x10B0B bytes offset from the start of the base.

PROCESS	BASE ADDRESS	NUMBER
SERVICES.EXE	0x001A0000	-
LSASS.EXE	0x001A0000	-
SVCHOST.EXE	0x001A0000	All 8 instances
SPOOLSV.EXE	0x001A0000	-
MSDTC.EXE	0x001A0000	-
DFSSVC.EXE	0x001A0000	-
WMIPRVSE.EXE	0x001A0000	All 2 instances
W3WP.EXE	0x001A0000	-
CSRSS.EXE	0x00270000	-
SMSS.EXE	UNICODE.NLS Not Mapped	-
INETINFO.EXE	UNICODE.NLS Not Mapped	-
CIDAEMON.EXE	UNICODE.NLS Not Mapped	-
CISVC.EXE	UNICODE.NLS Not Mapped	-

Table 2.

Table 2 lists some of the services that could be found running on a Windows 2003 Server machine. As can be seen, most of the services that do map UNICODE.NLS do so at address 0x001A0000 locating the “CALL DWORD PTR [EBP+0x30]” instruction at 0x001B0B0B.

The other two addresses, that is those where we find the POP, POP, RET instruction block, are slightly different. Vadump shows these to be “UNKNOWN_MAPPED” but there is a problem with these addresses. Whilst they are the same in every process – even across reboots – they are not at the same location on every system. These addresses, 0x7FFC0AC5 and 0x7FFC0AFC, were taken from a Sony VIAO PCG-R6000 laptop but a Dell running Windows 2003 Server has these instructions at address 0x7FFC0731 and 0x7FFC0914. A Sony VIAO PCG-Z1RSP laptop has these instructions at address 0x7FFC063A and 0x7FFC0C16. A VMWare image of Windows 2003 Server has a pop, pop, ret at 0x7FFC05A4. The location of these addresses seem to be hardware related but due to the instability of the location of these instructions they would be useful for local attacks only; unless of course you knew the exact spec of the hardware of the remote system. This means that an attacker would need to rely on the “CALL DWORD PTR [EBP+0x30]” after first verifying the address of UNICODE.NLS in the vulnerable process on one of their own systems. As this address is more than likely to contain a NULL instead of jumping forward the attacker would need to jump backwards. This NULL could cause a few problems in certain cases, however. One can guarantee an exception with an overflow by attempting to write beyond the end of the stack. With this NULL though, generating an exception in such a way won’t be possible, unless of course the overflow is Unicode in nature as we saw in the DCOM overflow. If the overflow is not Unicode in nature then the attacker must either hope that the code access violates on its own with a variable overwrite (see the Code Specific section) or find an instruction (block) at an address that does not contain a NULL.

The current way of handling exceptions is too dangerous. It is not safe to call an “exception handler” if it is outside of the address range of a loaded module; there is too much scope for abuse and allows an attacker to bypass the stack protection. My best guess here is that there maybe some Microsoft application that does place the code of an exception handler outside the address range of a loaded module; onto the heap for example. Perhaps we’re not just dealing with technology problems here – there may be political issues at play as well. If, for example, one of the Microsoft Office applications placed an exception handler on the heap, then modifying the way structured exception handling currently works would break it; and this, politically, would not do. If this is the case then it is standing in the way of providing a more restrictive solution – which is what we need.

Abusing an Existing, Registered Exception Handler

Assuming there were no such instruction(s) outside the address range of a module is there any scope to abuse an existing, registered handler to bypass stack protection? The answer to this is currently is yes. There is a registered handler in the Load Configuration Directory of NTDLL.DLL with an address of 0x77F45A34. If we disassemble this exception handler we find the following pertinent code:

```
77F45A3F   mov ebx,dword ptr [ebp+0Ch]
..
77F45A61   mov esi,dword ptr [ebx+0Ch]
77F45A64   mov edi,dword ptr [ebx+8]
..
77F45A75   lea ecx,[esi+esi*2]
77F45A78   mov eax,dword ptr [edi+ecx*4+4]
..
77F45A8F   call eax
```

At address 0x77F45A3F the value pointed to by EBP+0Ch is moved into EBX. This value is a pointer to the “owned” EXCEPTION_REGISTRATION structure. The DWORD value pointed to 0x0C bytes past EBX is then moved into ESI. As we've overflowed the EXCEPTION_REGISTRATION structure and beyond it we control this DWORD, and consequently, we “own” ESI. Next, the DWORD value pointed to 0x08 bytes past EBX is moved into EDI. Again, we control this. The effective address of ESI + ESI * 2 (equivalent to ESI * 3) is then loaded into ECX. As we own ESI we can guarantee the value that goes into ECX. Then the address pointed to by EDI, which we also own, added to ECX * 4 + 4, is moved into EAX. EAX is then called. As we completely control what goes into EDI and ECX (through ESI) we can control what is moved into EAX and therefore can direct the process to execute our code. The only difficulty is finding an address that holds a pointer to our code. We need to ensure that EDI+ECX*4+4 matches this address so the pointer to our code is moved into EAX and then called. For this to succeed all that is needed is a pointer to the “owned” EXCEPTION_REGISTRATION structure. We can find this on the stack, as it is linked through another EXCEPTION_REGISTRATION structure.

As an example of this let’s say there’s a pointer to the owned EXCEPTION_REGISTRATION structure at address 0x005CF3E4. If the attacker were to set 0x0C bytes past their EXCEPTION_REGISTRATION structure to 0x40001554 (this will go into ESI) and 0x08 bytes past it to 0x005BF3F0 (this will go into EDI) then after all the multiplication and addition they’re left with 0x005CF3E4. The DWORD value pointed to by this is then moved into EAX and then called. On EAX being called the attacker lands in their EXCEPTION_REGISTRATION structure at what would be the pointer to the next EXCEPTION_REGISTRATION structure. If they were to put code in here that performs a short jmp 14 bytes from the current location then they would jump over the crud they've needed to set to get execution to this point. (Incidently, the values used here are taken from one of the exploits I wrote for the DCOM overflow on Windows 2003 Server; hence the use of NULLs in the addresses. This overflow was Unicode in nature. For an ANSI based overflow the values would need to be adjusted accordingly to do away with any NULLs.)

The location of this pointer needs to be stable however – and if the address that this pointer can be found at wanders around then this will not work.

Whilst most of the registered handlers just perform a jump to __except_handler3 in msvcrt.dll or are simply copies of it, this handler at 0x77F45A34 is not the only registered handler that can be abused in this fashion. Other modules use this same handler. In terms of protecting against abuse of existing handlers Microsoft should carefully examine every handler to see if it can be subverted in a similar fashion and modified accordingly to prevent it.

Heap Loading

We already know that if the address of the handler is on the stack it will not be called. This is not true of the heap however. This provides a third alternative but it is completely code dependent. Essentially the method involves loading a heap based buffer with exploit code and overwriting the pointer to the exception handler with an address that points to the buffer on the heap. Needless to say that if the address of this buffer is not predictable, that is, the address wanders too much from process instance to process instance then this method cannot be used effectively.

Attacks that employ this method can be defeated by calling exception handlers if and only if they exist within the address range of a loaded module.

Code Specific Methods

This section describes some of the methods that can be used when the code of the vulnerable function allows it.

Consider the following function

```
        int foo(char **bar)
        {
            char *ptr = 0;
            char buffer[200]=””;
            strcpy(buffer,bar);
            ..
            ..
            *bar = ptr;
            return 0;
    }
```

Before this function returns the address of bar is updated to point to ptr. If we examine the assembly of this we have:

```
    MOV ECX, DWORD PTR [EBP+8] // **bar
    MOV EDX, DWORD PTR [EBP-8] // ptr
    MOV DWORD PTR [ECX],EDX
```

As we can see the address of bar is moved into ECX, the value of ptr is moved into EDX, which is then moved into the address pointed to by ECX. We can see in the C source that there is an unsafe call to strcpy and a local buffer could be overflowed. If an attacker were to overflow this buffer, overwrite the cookie and then overwrite the saved return address then they would begin to overwrite the parameters that were passed to the function. This means an attacker is in control of the parameters. With the pointer update at the end of this function an attacker is in the position to overwrite an arbitrary address with whatever the value of ptr is; this occurs before the security cookie checking code is executed. This presents an attacker with several opportunities of attack to bypass stack protection. Before we look at these we should question whether this is a likely scenario, to which the answer is yes. If we disassemble many of the Windows functions we can see this kind of action taking place fairly regularly. Indeed, the LSD DCOM buffer overflow suffered from exactly this problem.

Double Cookie Overwrite

In such overflows where a similar arbitrary overwrite occurs an attacker could use this as an opportunity to overwrite the authoritative version of the cookie stored in the .data section. Providing the value that it is overwritten with is predictable or consistent then the attacker would overwrite the stack copy of the cookie with the same value so when the cookie checking code executes the cookie matches and a normal “saved return address overwrite” exploit would work. Preventing this kind of attack is as easy as marking the 32bits of .data section where the authoritative cookie is stored as read only memory with a call to VirtualProtect:

```
    VirtualProtect(&Cookie,0x04,PAGE_READONLY,&old);
```

This should be executed as soon as the cookie has been set. [N.B. The smallest amount of memory VirtualProctect can protect is a page – that is 4K bytes. The cookie needs to located at the start of a page boundary and no other data that needs to be modified can be stored on the same page.]

Security Handler Overwrite

Recall that, if the cookies do not match, a check is made to see if a security handler has been defined. If one has then it is called. With such a scenario as described in the function above, an attacker could overwrite the address of the handler if one has been defined or set one if a one has not been. Provided the pointer pointed to attacker supplied data then this would be a viable option. With regards to the DCOM overflow there was a complication that prevented such an attack. The Unicode buffer had to start with “\\” (0x5C005C00) as it needed to be a UNC path to trigger the overflow.

The bytes 0x5C005C00, when translated to assembly is

```
POP ESP
ADD BYTE PTR[EAX+EAX+N]
```

where N is the next byte in the buffer. As EAX+EAX+N was never writable the process would access violate and the attacker would lose the chance to gain control.

To prevent such attacks there would be two options. One would be to not have a "security handler" facility and just terminate the process. The other would be to mark the 32 bits of the .data section where the pointer to the handler exists as read only once it has been set.

```
VirtualProtect(&sec_func,0x04,PAGE_READONLY,&old);
```

This would prevent such an attack. [Again, the same warning needs to be applied: VirtualProctect modifies the permissions of a whole page. The cookie and the pointer to the security handler can be place on the same page and then protected. No other data can be set on this page.]

Replacing the Windows System Directory

Again, if the buffer pointed to by the pointer is controlled by the attacker it is possible to overwrite the pointer to the Windows system directory, stored in the .data section of kernel32.dll. If the cookies don't match then, remember, the UnhandledExceptionFilter function is called. This function calls the GetSystemDirectoryW function which uses this pointer to get the directory. This is then copied to a buffer to which faultrep.dll is concatenated to. This library is then loaded and the ReportFault function called. By replacing the pointer to the Windows system directory with a pointer to the user supplied buffer an attacker could have the process load their own faultrep.dll that exported a ReportFault function and place in there whatever code they see fit.

Ldr Shim function pointer overwrites

Many of the Ldr* functions, that are called by functions such as LoadLibrary and FreeLibrary check to see if a function pointer has been set and calls it if one has been. This is related to the Shim engine. As the UnhandledExceptionFilter function calls both LoadLibraryExW and FreeLibrary an attacker has the opportunity to set these function pointers which would then be called allowing the attacker to gain control. For example, FreeLibrary calls LdrUnloadDll in NTDLL.DLL and if we examine the code of this function we can find:

```
77F5337A      mov eax, [77FC2410]
77F5337F      cmp eax, edi
77F53381      jne 77F53134
..
77F53134      push esi
77F53135      call eax
```

This moves the value at address 0x77FC2410, in the .data section of NTDLL.DLL, into EAX and then compares it with 0; EDI is 0. If EAX is not 0, in other words a function pointer is set at 0x77FC2410, ESI is pushed onto the stack and the function is called. By setting this function pointer an attacker could gain control.

Both this as an attack vector and the replacing of the pointer to the system directory are both difficult to defend against and there are hundreds of other similar attack vectors. We can't go around VirtualProtecting everything. The best approach would be to simply create a new kernel function NTTerminateProcessOnStackOverflow, as indicated earlier and immediately call it if the cookies do not match. This function could be responsible for the important “Report a Fault” stuff. As we can see there are just too many ways of subverting the stack based protecting mechanism to not to do this, in my opinion. If this were implemented then the attacker would need to revert to causing exceptions and as we’ve dispatched with those particular attacks we’re left with a more robust stack protection mechanism.

Wrapping Up

This paper has shown that currently there exists several methods to bypass the stack protection of Windows 2003 Server and there may yet be other ways. That said, if the recommendations made in this paper were adopted the number of possible avenues available to bypass the protection will be severely limited. There are a few, highly specific cases that could occur where these suggestions would be of no consequence. For example consider the following code snippet

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

int main(int argc, char *argv[])
{
        HMODULE Lib=NULL;
        unsigned int FuncAddress = 0;
        int err = 0;

        Lib = LoadLibrary("msvcrt.dll");
        FuncAddress = GetProcAddress(Lib,"printf");
        err = DoStuff(argv[1],FuncAddress);
        return 0;
}

int DoStuff(char *buf, FARPROC fnk)
{
        char buffer[20]="";
        strcpy(buffer,buf);
        (fnk)(buffer);
        __asm add esp,4
        return 0;
}
```

Here a pointer to the printf function is passed as one of the parameters to the DoStuff function and then is called. If the local buffer in DoStuff is overflowed beyond the cookie and the saved return address the function parameters are overwritten. Thus when the “printf” function is called the attacker can gain control. This is a highly manufactured situation, however, and is not likely to exist “in the wild” but that is not the point though. The point is that there may always be issues like this that can allow an attacker to gain control despite any protection offered by the operating system. As such, it is always advisable never to rely on such protection. The old adage “prevention is better than the cure” springs to mind; there is definitely no substitute for secure coding practices.

Appendix A

The two code listings can be used to inject code into another process to load a DLL and execute a function. This function searches the address space of the process for the bytes that form instructions that can be used to bypass stack protection using Structured Exception Handling. When compiled run from a command line

```
C:\>inject pid code.dll code
```

The results will be output to a file called results.txt.

```
/* Inject.c – compile C:\>cl /TC inject.c */

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

DWORD __stdcall RThread(pfnktstuff);
#define MAXINJECTSIZE 4096

typedef struct fnktstuff_t
{
        unsigned int LoadLibraryAddress;
        unsigned int GetProcAddressAddress;
        unsigned char Library[260];
        unsigned char Function[260];
} fnktstuff;

DWORD __stdcall RThread( fnktstuff *pfs)
{
        __asm{
                mov edi,dword ptr[esp+08h]      // Get Pointer to pfs
                push edi
                lea esi, dword ptr [edi+8]      // Get Pointer to Library
                push esi                        // Push onto the stack
                call dword ptr[edi]             // call LoadLibrary
                cmp eax,0
                je end
                pop edi
                lea esi, dword ptr [edi+268]
                push esi
                push eax
                call dword ptr[edi+4]
                cmp eax,0
                je finish
                call eax
                jmp finish
end:
                pop edi
finish:


                }
```

```
    return 0;
}

int main(int argc, char *argv[])
{
    HANDLE hProcess=NULL;
    HANDLE hRemoteThread=NULL;
    int ProcessID=0;
    void *FunctionAddress=0;
    void *data=NULL;
    HMODULE k=NULL;
    fnktstuff fs;

    if(argc !=4)
        return printf("C:\\>%s PID Library Function\n",argv[0]);

    k = LoadLibrary("kernel32.dll");
    fs.GetProcAddressAddress = (unsigned int)GetProcAddress(k,"GetProcAddress");
    fs.LoadLibraryAddress = (unsigned int)GetProcAddress(k,"LoadLibraryA");
    ZeroMemory(fs.Library,260);
    ZeroMemory(fs.Function,260);
    strncpy(fs.Library,argv[2],256);
    strncpy(fs.Function,argv[3],256);
    ProcessID = atoi(argv[1]);
    if(ProcessID == 0)
        return printf("ProcessID is 0\n");
    hProcess = OpenProcess(PROCESS_ALL_ACCESS,FALSE,ProcessID);
    if(hProcess == NULL)
        return printf("OpenProcess() failed with error: %d\n",GetLastError());
    printf("Process %d opened.\n",ProcessID);
    FunctionAddress = VirtualAllocEx(hProcess,0,MAXINJECTSIZE
        ,MEM_COMMIT,PAGE_EXECUTE_READWRITE);
    if(FunctionAddress == NULL)
    {
        printf("VirtualAllocEx() failed with error: %d\n",GetLastError());
        CloseHandle(hProcess);
        return 0;
    }
    printf("RemoteAddress is %.8x\n",FunctionAddress);
    data = (unsigned char *) VirtualAllocEx(hProcess,0,MAXINJECTSIZE,
        MEM_COMMIT,PAGE_READWRITE);
    if(data == NULL)
    {
        printf("VirtualAllocEx() failed with error: %d\n",GetLastError());
        CloseHandle(hProcess);
        return 0;
    }
    printf("DataAddress is %.8x\n",data);
    WriteProcessMemory(hProcess, FunctionAddress, &RThread, MAXINJECTSIZE, 0);
    WriteProcessMemory(hProcess, data, &fs, sizeof(fnktstuff), 0 );
    hRemoteThread = CreateRemoteThread(hProcess,NULL,0,
        FunctionAddress,data,0,NULL);
    if(hRemoteThread == NULL)
    {
        printf("CreateRemoteThread() failed with error: %d\n",GetLastError());
        CloseHandle(hProcess);
        return 0;
    }
    CloseHandle(hRemoteThread);
    CloseHandle(hProcess);
    return 0;
}
```

The code of the DLL:

```
// Compile: C:\>cl /LD code.dll

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

int __declspec (dllexport) code()
{
FILE *fd = NULL;
unsigned char *p = 0;
unsigned char *c = (char *)0x7fffffff;

fd = fopen("results.txt","w+");
if(!fd)
    return 0;

while(p < c)
{

    __try{
    if ((p[0] > 0x57) && (p[0] < 0x5F) && (p[1] > 0x57) && (p[1] < 0x5F) && (p[2] > 0xC1) &&
(p[2] < 0xC4))
        fprintf(fd,"pop, pop, ret found at\t\t\t%.8X\n", p );
    else if (p[0] == 0xFF)
    {
        if(p[1] == 0x55)
        {
            if(p[2] == 0x30)
                fprintf(fd,"call dword ptr[ebp+30] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24)
                fprintf(fd,"call dword ptr[ebp+24] found at\t\t%.8X\n", p );
            else if(p[2] == 0x0C)
                fprintf(fd,"call dword ptr[ebp+0C] found at\t\t%.8X\n", p );
            else if(p[2] == 0xFC)
                fprintf(fd,"call dword ptr[ebp-04] found at\t\t%.8X\n", p );
            else if(p[2] == 0xF4)
                fprintf(fd,"call dword ptr[ebp-0C] found at\t\t%.8X\n", p );
            else if(p[2] == 0xE8)
                fprintf(fd,"call dword ptr[ebp-18] found at\t\t%.8X\n", p );
        }
        else if(p[1] == 0x65)
        {
            if(p[2] == 0x30)
                fprintf(fd,"jmp dword ptr[ebp+30] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24)
                fprintf(fd,"jmp dword ptr[ebp+24] found at\t\t%.8X\n", p );
            else if(p[2] == 0x0C)
```

```
                fprintf(fd,"jmp dword ptr[ebp+0C] found at\t\t%.8X\n", p );
            else if(p[2] == 0xFC)
                fprintf(fd,"jmp dword ptr[ebp-04] found at\t\t%.8X\n", p );
            else if(p[2] == 0xF4)
                fprintf(fd,"jmp dword ptr[ebp-0C] found at\t\t%.8X\n", p );
            else if(p[2] == 0xE8)
                fprintf(fd,"jmp dword ptr[ebp-18] found at\t\t%.8X\n", p );
        }
        else if(p[1] == 0x54)
        {
            if(p[2] == 0x24 && p[3] == 0x08)
                fprintf(fd,"call dword ptr[esp+08] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24 && p[3] == 0x14)
                fprintf(fd,"call dword ptr[esp+14] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24 && p[3] == 0x1C)
                fprintf(fd,"call dword ptr[esp+1C] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24 && p[3] == 0x2C)
                fprintf(fd,"call dword ptr[esp+2C] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24 && p[3] == 0x44)
                fprintf(fd,"call dword ptr[esp+44] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24 && p[3] == 0x50)
                fprintf(fd,"call dword ptr[esp+50] found at\t\t%.8X\n", p );
        }
        else if(p[1] == 0x64)
        {
            if(p[2] == 0x24 && p[3] == 0x08)
                fprintf(fd,"jmp dword ptr[esp+08] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24 && p[3] == 0x14)
                fprintf(fd,"jmp dword ptr[esp+14] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24 && p[3] == 0x1C)
                fprintf(fd,"jmp dword ptr[esp+1C] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24 && p[3] == 0x2C)
                fprintf(fd,"jmp dword ptr[esp+2C] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24 && p[3] == 0x44)
                fprintf(fd,"jmp dword ptr[esp+44] found at\t\t%.8X\n", p );
            else if(p[2] == 0x24 && p[3] == 0x50)
                fprintf(fd,"jmp dword ptr[esp+50] found at\t\t%.8X\n", p );
        }
    }
    }

    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        p += 0x00000100;
    }

    p++;
}

fclose(fd);
return 0;
}
```