CHAPTER 19

# Instrumented Investigation: A Manual Approach

With all the talk about fuzzing, you might be led to believe that there's no place for manual investigation in the world of the modern bug hunter. The aim of this chapter is to show why that's not true, and that manual bug hunting is alive and well. We'll start with a discussion of the technique (such as it is) and then go through some examples of the thought processes and techniques behind the discovery of certain bugs. Along the way, we'll also address input validation in general and talk about some interesting ways to bypass it, because input validation often thwarts the research process, and a slightly deeper understanding can help to both make attacks more potent and increase understanding of defensive techniques.

## Philosophy

The idea behind our approach is to simplify the researcher's view of the system, allowing him or her to focus on the structure and behavior of the system from a technical security perspective rather than being led along some predefined path by vendor documentation or source code. It is more of an attitude and an approach than a specific technique, although you will need some basic skills. Our experience has been that this approach leads to the discovery of bugs that were "not thought possible" by the development teams—because they were too obvious, or obscured by the source code (for example, complex

C macro definitions), or because an interaction between components of the system had simply not been thought about. Throwing away the rulebook, so to speak, is a liberating thing.

The approach can best be summarized as follows:

- Attempt to understand the system without referencing its documentation and source code.
- Investigate likely areas of weakness. During the investigation, make use of system tracing tools to learn more about the behavior of the system and take note of where behaviors diverge—these points may not be obvious.
- Where differing behaviors are observed, attempt common forms of attack and observe the response.
- Continue until you've covered all the behaviors.

Perhaps a concrete example would clarify the process.

## Oracle extproc Overflow

Oracle issued Security Alert 57 regarding the extproc overflow—you can find the alert at `otn.oracle.com/deploy/security/pdf/2003alert57.pdf`. You can find the Next Generation Security Software (NGS) advisory at `www.ngssoftware.com/advisories/ora-extproc.txt`.

In September 2002, Next Generation Security Software performed a rigorous investigation of the Oracle RDBMS to look for security flaws, because the authors felt that the Oracle DBMS had been under-audited by the rest of the security community. David Litchfield had previously found a bug in the `extproc` mechanism, so we decided to take another look in that general area. It's important to understand the architectural details here that relate to how this first bug was discovered.

In general, advanced DBMSs support some dialect of Structured Query Language (SQL) that allows for more complex scripts and even procedures to be created. In SQL Server, this is called *Transact-SQL*, and allows things like `WHILE` loops, `IF` statements, and so on. SQL Server also allows direct interaction with the operating system via what Microsoft terms "extended stored procedures"—these are custom functions implemented in DLLs that can be written in C/C++.

Historically, there have been *many* buffer overflows in SQL Server's extended stored procedures, so it's logical to believe that analogous mechanisms in other DBMSs will suffer from similar problems. Enter Oracle.

Oracle offers a mechanism that is much richer than SQL Server's extended stored procedure mechanism in that it allows you to call arbitrary library

functions, not just functions conforming to some predefined specification. In Oracle, calls out to external libraries are called *external procedures*, and are carried out in a secondary process, `extproc`. `extproc` is offered as an additional Oracle service, and can be connected to in a similar manner to the database service itself.

Another important thing to understand is the Transparent Network Substrate (TNS) protocol. This is the part of the architecture that manages the Oracle process's communication with clients and with other parts of the system. TNS is a text-based protocol with a binary header. It supports a large number of different commands, but the general purpose is to start, stop, and otherwise manage Oracle services.

We looked at this `extproc` mechanism, and decided to instrument it to see what it did. We were running Oracle on the Windows platform, so we took all the standard sockets calls in the Oracle process and breakpointed them—`connect`, `accept`, `recv`, `recvfrom`, `readfile`, `writefile`, and so on. We then made a number of calls to external procedures.

David Litchfield discovered that when Oracle called out to an extended stored procedure, it used a series of TNS calls followed by a simple protocol to make the call. There was absolutely no authentication; `extproc` simply assumed that it must be Oracle talking on the other end of the connection. The implication of this is that if you can (as a remote attacker) direct `extproc` to call the library of your choice, you can easily compromise the server, by (say) running the `system` function in libc or `msvcrt.dll` on Windows. There are a number of mitigating factors, but in a default installation (before Oracle fixed this bug), this was the case.

We reported this to Oracle and worked with them to put out a patch. You can find the Oracle alert (number 29) at `otn.oracle.com/deploy/security/pdf/lsextproc_alert.pdf`. David Litchfield's advisory is available at `www.ngssoftware.com/advisories/oraplsextproc.txt`.

Because this area of Oracle's behavior is so sensitive (in terms of the security of the system), we decided to again review all the behaviors that relate to external procedures to see whether we could find anything more.

The way you make the aforementioned call is fairly simple—you can see the TNS commands by debugging Oracle and running the following script (from Litchfield's excellent "HackProofing Oracle Application Server" paper, which you can find at `www.ngssoftware.com/papers/hpoas.pdf`):

```
Rem
Rem oracmd.sql
Rem
Rem Run system commands via Oracle database servers
Rem
Rem Bugs to david@ngssoftware.com
```

```
Rem

CREATE OR REPLACE LIBRARY exec_shell AS
'C:\winnt\system32\msvcrt.dll';
/
show errors
CREATE OR REPLACE PACKAGE oracmd IS
PROCEDURE exec (cmdstring IN CHAR);
end oracmd;
/
show errors
CREATE OR REPLACE PACKAGE BODY oracmd IS
PROCEDURE exec(cmdstring IN CHAR)
IS EXTERNAL
NAME "system"
LIBRARY exec_shell
LANGUAGE C; end oracmd;
/
show errors
```

Then you run the procedure

```
exec oracmd.exec ('dir > c:\oracle.txt);
```

to kick off the actual execution.

Starting at the beginning, we tried the usual things in the `create or replace library` statement by manually plugging in queries and seeing what happened (in the debugger and in `FileMon`). Surprisingly, when we submitted an overly long library name:

```
CREATE OR REPLACE LIBRARY ext_lib IS 'AAAAAAAAAAAAAAAAAAAAAAAAA...';
```

and then called a function in it:

```
CREATE or replace FUNCTION get_valzz
   RETURN varchar AS LANGUAGE C
     NAME "c_get_val"
 LIBRARY ext_lib;

select get_valzz from dual;
```

something weird happened, not to Oracle itself, but apparently somewhere else—the connection was being reset—which would normally indicate some sort of exception. The odd thing was that it didn't occur in the Oracle process.

After looking at `FileMon` for a while, we decided to debug the TNS Listener (`tnslsnr`) process (which handles the TNS protocol and is the intermediary between Oracle and `extproc` when calling external procedures). Because the

tnslsnr process starts extproc, we used WinDbg, which allows for easy tracing of child processes. The process was a little involved:

```
1) Stop all oracle services
2) Start the oracle database service ('OracleService<hostname>')
3) From a command line session in the interactive desktop, start windbg
-o tnslsnr.exe
```

This causes WinDbg to debug the TNS Listener and any processes that the TNS Listener starts. The TNS Listener is running in the interactive desktop.

Sure enough, once we did this, we saw the magic exception in WinDbg:

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00ec0480 ecx=00010101 edx=ffffffff esi=00ebbfec
edi=00ec04f8
eip=41414141 esp=0012ea74 ebp=41414141 iopl=0         nv up ei pl zr na
po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010246
41414141 ??              ???
```

This was indicative of a vanilla stack overflow in extproc.exe. A quick set of tests revealed that the problem didn't affect only the Windows platform.

We have one vector to trigger the bug—via the create library statement. But, harking back to David's original extproc advisory, we recalled that it's possible to make calls to extproc directly as a remote attacker. We then coded up a quick harness to remotely trigger the overflow and discovered that it works the same way. We had found a remote unauthenticated stack overflow in Oracle. So much for "*Unbreakable*"!

Apparently, this vulnerability was introduced in the patch to the previous bug—the functionality introduced to log the request to run the external procedure is vulnerable to an overflow.

To summarize the process behind the discovery of these two bugs:

- We were aware of a probable architectural weakness, in that we knew that SQL Server had problems in this general area of functionality. We therefore considered it likely that Oracle would suffer from a similar problem, because accessing stored procedures is a difficult thing to do safely.
- Carefully instrumenting and tracing the behavior of Oracle led us to the possibility of executing external procedures without authentication—bug number one.
- Revisiting this area of functionality, we found that something strange happened with overly long library names (since the patch to the first extproc bug).

- We instrumented with debuggers and a file-monitoring tool (Russinovich and Cogswell's excellent `FileMon`) and identified the components in question.
- Upon debugging the components in question, we saw the exception indicating the stack overflow—bug number two.

At no point did we automate anything; it was all based on looking carefully at the construction of the system under test and disregarding the documentation, preferring instead to understand the system in terms of its instrumented behavior.

As a footnote to this exercise, Oracle has now included an excellent set of workaround information for these bugs in Alert 57 as well as a patch that fully addresses both issues.

# Common Architectural Failures

As we saw in the previous example, things tend to fail in similar ways. After you've been looking at advisories every day for a few years, you start to notice patterns and then go after them in your own research. It's probably useful to stop and consider those patterns for a moment, because they might provide ideas for your future research.

## Problems Happen at Boundaries

Although this isn't universally true, it's generally the case that security problems occur when there's a transition of some kind: from one process to another, from one technology to another, or from one interface to another. The following are a few examples of these.

### *A Process Calling into an External Process on the Same Host*

Good examples of this problem are the Oracle issue 57 described previously and the Named Pipe Hijacking issue found by Andreas Junestam and described in Microsoft Bulletin MS03-031. To see some interesting privilege elevation issues, use `HandleEx` or Process Explorer (from Sysinternals) to take a look at the permissions assigned to global objects (like shared memory sections) in Windows. Many applications don't guard against local attacks.

In Unix you'll find a whole family of problems relating to the parsing of command-line options and environment variables when a process calls out to some other process to perform some kind of function. Once again, instrumentation is helpful if you're looking in this area. In this case, ltrace/strace/truss are probably the best way to go.

### A Process Calling into an External, Dynamically Loaded Library

Again, Oracle and SQL Server provide multiple examples of this problem—the original `extproc` bug found by David Litchfield (Oracle alert 29) being one, and the many extended stored procedure overflows found in SQL Server being another.

Also, there are a very large number of problems in `ISAPI` filters in Microsoft IIS, including a Commerce Server component, the `ISM.DLL` filter, the `SQLXML` filter, the `.printer` ISAPI filter, and many more. One of the reasons these sorts of problems occur is that although people heavily audit core behaviors of a network daemon, they tend to overlook extensibility support.

IIS isn't alone in this. Take a look at the Apache `mod_ssl` off-by-one bug, as well as problems in `mod_mylo`, `mod_cookies`, `mod_frontpage`, `mod_ntlm`, `mod_auth_any`, `mod_access_referer`, `mod_jk`, `mod_php`, and `mod_dav`.

If you're auditing an unknown system, a soft spot can normally be found in this kind of functional area.

### A Process Calling into a Function on a Remote Host

This is also a minefield, although people tend to be more aware of the risks. The recent Microsoft Windows RASMAN RPC bug (MS06-025) shows that this kind of problem is still around. Most RPC bugs fit into this category, like the Sun UDP PRC DOS, the Locator Service overflow, the multiple MS Exchange overflows found by Dave Aitel, and the old favorite `statd` format string bug found by Daniel Jacobowitz.

## Problems Happen When Data Is Translated

When data is transformed from one form to another, it's often possible to bypass checks. This is actually a fundamental problem relating to translation between grammars. The reason this kind of problem (often called a canonicalization bug) is so prevalent is that it is exceptionally difficult to create a system in which programmable interfaces become less grammatically complex as you descend deeper into the call tree.

Formally, we could put it like this: Function `f()` implements a set of behaviors `F`. `f()` implements these behaviors by calling a function `g()`, passing it some of the input to `f()`. `g` implements the set of behaviors `G`. Unfortunately, set `G` contains behaviors that are undesirable to expose via `f()`. We call these bad behaviors `Gbad`. Therefore, `f()` must implement some mechanism to ensure that `F` contains none of `Gbad`. The only way that the implementer of `f()` can do this is to fully understand all of `G` and validate the inputs to `f()` to ensure that no combination of inputs results in any member of `Gbad`.

This is a problem for two reasons:

- Things almost always get more complex the lower you go down the call tree, so `f()` deals with too many cases.
- `g()` has the same problem, as do `h()`, `i()`, `j()`, and so on down the call tree.

For example, take the Win32 filesystem functions. You might have a program that accepts a filename. As far as it understands the concept of filenames, it assumes the following:

- A filename may have an extension at the end. Extensions are normally, but not always, three characters long, and are denoted by the final period (.) character in the filename.
- A filename may be a fully qualified path. If so, it starts with a drive letter, which is followed by a colon (:) character.
- A filename may be a relative path. If so, it will contain backslash (\) characters.
- Each backslash character signals a transition into a child directory.

This can be thought of as constituting a grammar for filenames as far as the program is concerned. Unfortunately, the grammar implemented by the underlying filesystem functions (like the Win32 API `CreateFile`) includes many other potentially dangerous constructs such as the following (this is not an exhaustive list):

- A filename can begin with a double-backslash sequence. If this is the case, the first `directory name` signifies a host on the network and the second an SMB `share name`. The `FileSystem` API will attempt to connect to this share using the (sniffable) credentials of the current user.
- A filename can also begin with a `\\?\` sequence, which denotes that it is a Unicode file path and is able to exceed the normal length limits imposed by the `FileSystem` API.
- A filename can begin with `\\?\UNC`, which will also trigger the Microsoft Share connection behavior described previously.
- A filename can begin with `\\.\PHYSICALDRIVE<n>`, where `<n>` is the zero-based index of the physical drive to open. This will open the physical drive for raw access.
- A filename can begin with `\\.\pipe\<pipename>`. The named pipe `<pipename>` will be opened.
- A filename can include a colon (:) character (after the initial drive letter sequence). This denotes an alternate data stream in the `NTFS` filesystem,

which is treated effectively as a distinct file, but which is not listed disparately in directory listings. The `:$DATA` file stream is reserved for the normal contents of the file.

- A filename can include (as a directory name) a `..` or `.` sequence. The former case signals a transition to the parent directory, the latter signifies that no directory transition should be performed.

Many other equally bizarre behaviors are possible. The point is, unless you're careful about input validation, you'll end up introducing problems, because the underlying API is likely to implement behaviors that you're unaware of. Therefore, from the attacker's perspective, it makes sense to understand these underlying behaviors and try to get at them through the defending input validation mechanisms.

Some of the real-world bugs that happen because of this sort of problem (not all shellcode, unfortunately) include the IIS Unicode bug, the IIS double-decode bug, the `CDONTS.NewMail` SMTP injection problem, PHP's `http://filename` behavior (you can open a file based on a URL), and the Macromedia Apache source code disclosure vulnerability (if you add an encoded space to the end of a URL, you get the source code). There are many more. Almost every source code disclosure bug fits into this category.

If you think about it, input validation is actually the reason why overflows are so harmful. The input to a function is interpreted in some underlying context. In the case of a stack overflow, the data that overflows the buffer is treated as a portion of a stack frame comprising data, Virtual Pointers (VPTRs), saved return addresses, exception handler addresses, and so on. What you might call a phrase in one grammar is interpreted as a phrase in a different one.

You could summarize almost all attacks as attempts to construct phrases that are valid in multiple grammars. There are some interesting defensive implications to this, in the fields of information theory and coding theory, because if you can ensure that two grammars have no phrases in common, you might (possibly) be able to ensure that no attack is possible based on a translation between the two.

The idea of interpretive contexts is a useful one, especially if you're dealing with a target that supports a variety of network protocols—such as a Web server that sends email or transfers data to a Web services server using a weird XML format. Just think "What parses this input?" and if you can correctly answer that question, you might be well on the way to finding a bug.

## Problems Cluster in Areas of Asymmetry

In general, developers tend to apply defensive techniques across a whole area of behaviors, using such things as length limits, checking for format strings, or

other kinds of input validation. One excellent way to find problems is to look for an area of asymmetry and explore it to find out what makes it different.

Perhaps a single HTTP header supported by a Web server appears to have a different length limit than all the others, or perhaps you notice a weird response when you include a particular symbol in your input data. Or possibly specifying a recently implemented Web method in Apache seems to change your error messages. Maybe your attempt at file-execution through a buggy Web server fails when you request cmd.exe, but would succeed on ftp.exe.

Taking note of areas that are different can tip you off to areas of a product that are less protected.

## Problems Occur When Authentication and Authorization Are Confused

*Authentication* is the verification of identity, nothing more. *Authorization* is the process of determining whether a given identity should have access to a given resource.

Many systems take great care over the former and assume that the latter follows. Worse, in some cases, there is seemingly no connection between the two—if you can find an alternative route to the data, you can access it. This leads to some interesting privilege elevation situations, such as the Oracle `extproc` example. You can also see it in Lotus Domino with the view ACL bypass bug (`www.ngssoftware.com/advisories/viewbypass.txt`), and in Oracle `mod_plsql` with the authentication bypass (`www.ngssoftware.com/papers/hpoas.pdf`—search for *authentication by-pass*). The Apache case-insensitive `htaccess` vulnerability (`www.omnigroup.com/mailman/archive/macosx-admin/2001-June/020678.html`) was another good example of what happens when another route is provided to sensitive data.

You can also see this type of problem in many Web applications. Because HTTP is inherently stateless, the mechanism used to maintain the state (a session ID) normally carries with it the authentication state. If you can somehow guess or reproduce the session ID, you can skip the authentication stage.

## Problems Occur in the Dumbest Places

If a particular bug hunt is becoming too technical and it's been a long day, don't be afraid to try the really obvious. Overly long usernames were the cause of these bugs among many:

- `www.ngssoftware.com/advisories/sambar.txt`
- `otn.oracle.com/deploy/security/pdf/2003Alert58.pdf`
- `www.ngssoftware.com/advisories/ora-unauthrm.txt`

- www.ngssoftware.com/advisories/ora-isqlplus.txt
- www.ngssoftware.com/advisories/steel-arrow-bo.txt
- cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0891
- www.kb.cert.org/vuls/id/322540

Generally, the authentication phase of a protocol is a good target for overflow and format string research for the obvious reason that if you can gain control prior to authentication, you need no username and password to compromise the server. Another couple of classic, unauthenticated remote root bugs are the `hello` bug found by Dave Aitel (`cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-1123`) and the SQL-UDP bugs found by David Litchfield (`www.ngssoftware.com/advisories/mssql-udp.txt`). We've also found products where obscure parts of various protocols were accessible without authentication—the check for authentication was simply skipped in some cases. In one spectacular case, the absence of authentication state was considered to make the user a superuser ("uid is 0 if there's no uid," combined with "a uid of 0 means you're root"). These areas make great targets, for obvious reasons.

# Bypassing Input Validation and Attack Detection

Understanding input validation and knowing how to bypass it are essential skills for the bug hunter. We'll give you a brief overview of the subject to help you understand where mistakes are made and provide you with some useful validation bypass techniques.

## Stripping Bad Data

People often use flawed regular expressions to try to limit (or detect) potential attacks. One common application is to strip out input that is known to be bad—if you are defending against SQL Injection you might, for example, write a filter that strips out SQL reserved words such as `select`, `union`, `where`, `from` and so on.

For instance, the input string

```
' union select name, password from sys.user$--
```

might become

```
' name, password sys.user$--
```

This produces an error. Sometimes you can bypass this error by recursively including bad data within itself, like this:

```
' uniunionon selselectect name, password frfromom sys.user$--
```

Each bad term is included within itself. As the values are stripped out, the enclosing bad data is reconstituted, leaving us with precisely the data we wanted. Obviously this only works when the known bad terms are composed of at least two distinct characters.

## Using Alternate Encodings

The most obvious way of bypassing input validation is to use an alternate encoding of your data. For instance, you might find that the way a Web server or Web application environment behaves depends on how you encode form data. The IIS Unicode encoding specifier `%u` is a good example. In IIS, these two are equivalent:

- `www.example.com/%c0%af`
- `www.example.com/%uc0af`

Another good example is treatment of whitespace. You might find that an application treats space characters as delimiters, but not TAB, carriage return, or linefeed characters. In the Oracle `TZ_OFFSET` overflow, a space will terminate the `timezone` specifier, but a TAB character will not. We wrote an exploit for this bug that ran a command, and we were having trouble specifying parameters in the exploit. We quickly modified the exploit to change all spaces to TABs, which worked fine, because most shells treat both spaces and TABs as delimiters.

Another classic example was an `ISAPI` filter that attempted to restrict access to an IIS virtual directory based on certain credentials. The filter would kick in if you requested anything in the `/downloads` directory (`www.example.com/downloads/hot_new_file.zip`). Obviously, the first thing to try in order to bypass it is this:

```
www.example.com/Downloads/hot_new_file.zip
```

which doesn't work. Then you try this:

```
www.example.com/%64ownloads/hot_new_file.zip
```

and the filter is bypassed. You now have full access to the downloads directory without authentication.

## Using File-Handling Features

Some of the techniques presented in this section apply only to Windows, but you can normally find a way around these kinds of problems on Unix platforms as well. The idea is to trick an application so that either:

- It believes that a required string is present in a file path.
- It believes that a prohibited string is not present in a file path.
- It applies the wrong behavior to a file if file handling is based on a file's extension.

### *Required String Is Present in Path*

The first case is easy. In most situations in which you can submit a filename, you can submit a directory name. In an audit we performed, we encountered a situation in which a Web application script would serve files provided that they were in a given constant list. This was implemented by ensuring that the name of one of the specified files:

- `data/foo.xls`
- `data/bar.xls`
- `data/wibble.xls`
- `data/wobble.xls`

appeared in the `file_path` parameter. A typical request might look like this:

```
www.example.com/getfile?file_path=data/foo.xls
```

The interesting thing is that when most filesystems encounter a parent path specifier, they don't bother to verify that all the referenced directories exist. Therefore, we were able to bypass the validation by making requests such as:

```
www.example.com/getfile?file_path=data/foo.xls/../../../etc/passwd
```

### *Prohibited String Not Present in Path*

This situation is a little trickier, and again, it involves directories. Let's say the file serving script mentioned in the preceding section allows us to access any file but prohibits the use of parent path specifiers (`/../`) and additionally prohibits access to a private data directory by checking for this string in the `file_path` parameter:

```
data/private
```

We can bypass this protection by making such requests as:

```
www.example.com/getfile?file_path=data/./private/accounts.xls
```

because the `/./` specifier does nothing in the context of a path. Doubling up on slashes ('data//private') can sometimes achieve a similar result.

### *Incorrect Behavior Based on File Extension*

Let's say that Web site administrators tire of people downloading their accounts spreadsheets and decide to apply a filter that prohibits any `file_path` parameter that ends in `.xls`. So we try:

```
www.example.com/getfile?file_path=data/foo.xls/../private/accounts.xls
```

and it fails. Then we try:

```
www.example.com/getfile?file_path=data/./private/accounts.xls
```

and it also fails.

One of the most interesting aspects of the Windows NT NTFS filesystem is its support for alternate data streams within files, which are denoted by a colon (:) character at the end of the filename and a stream name after that.

We can use this concept to get the account's data. We simply request:

```
www.example.com/getfile?file_path=data/./private/accounts.xls::$DATA
```

and the data is returned to us. The reason this happens is that the "default" data stream in a file is called `::$DATA`. We request the same data, but the filename doesn't end in the `.xls` extension, so the application allows it.

To see this for yourself, run the following on an NT box (in an NTFS volume):

```
echo foobar > foo.txt
```

Then run:

```
more < foo.txt::$DATA
```

and you'll see `foobar`. In addition to its ability to confuse input validation, this technique also provides a great way to hide data.

A bug in IIS a few years ago let you read the source of ASP pages by requesting something like:

```
www.example.com/foo.asp:::$DATA
```

This worked for the same reason.

Another trick relating to file extensions in Windows systems is to add one or more trailing dots to the extension. That would make our request to the file serving script become:

```
www.example.com/getfile?file_path=data/./private/accounts.xls.
```

In some cases, you'll get the same data. Sometimes the application will think the extension is blank; sometimes it will think the extension is `.xls`. Again, you can quickly observe this by running

```
echo foobar > foo.txt
```

then

```
type foo.txt.
```

or

```
notepad foo.txt.....
```

## Evading Attack Signatures

Most IDS systems rely on some form of signature-based recognition of attacks. In the shellcode field, people have already published much information about `nop`-equivalence, but I'd like to address the point here briefly, because it's important.

When you write shellcode, you can insert an almost infinite variety of instructions that do nothing in between the instructions meaningful to your exploit. It's important to bear in mind that these instructions need not actually do nothing—they just need to do nothing that is relevant to the state of your exploit. So for example, you might insert a complex series of stack and frame manipulations into your shellcode, interleaving the instructions with the actual instructions that make up your exploit.

It's also possible to come up with an almost infinite number of ways to perform a given shellcode task, such as pushing parameters onto the stack or loading them into registers. It's fairly easy to write a generator that takes one form of the assembler for an exploit and spits out a functionally identical exploit *with no code sequences in common*.

## Defeating Length Limitations

In some cases, a given parameter to an application is truncated to a fixed length. Generally, this is an attempt to guard against buffer overflows, but sometimes it's used in Web applications as a generic defense mechanism

against SQL Injection or command execution. There are a number of techniques that can help in this kind of situation.

### Sea Monkey Data

Depending on the nature of the data, you might be able to submit some form of input that expands within the application. For example, in most Web-based applications you wind up encoding double-quote characters as:

```
&quot;
```

which is a ratio of six characters to one.

Any character that is likely to be "escaped" in the input is a good candidate for this sort of thing—single quotes, backslashes, pipe characters, and dollar symbols are quite good in this respect.

If you're able to submit `UTF-8` sequences, submit overly long sequences, because they might be treated as a single character. You might be lucky and come across an application that treats all non-ASCII characters as 16 bits. You might then overflow it by giving it characters that are longer than this, depending on how it calculates string length.

`%2e` is the URL encoding for (.). However:

```
%f0%80%80%ae
```

and

```
%fc%80%80%80%80%ae
```

are also encodings of (.).

### Harmful Truncation—Severing Escape Characters

The most obvious application of this technique is to SQL Injection, although bearing in mind the earlier discussion of canonicalization, it's possible to come up with all sorts of interesting ways of applying the technique wherever delimited or escaped data is used. Running commands in perl is good for possibly injecting into an SMTP stream.

Essentially, if data is being both escaped and truncated, you can sometimes break out of the delimited area by ensuring that the truncation occurs in the middle of an escape sequence.

There is an obvious SQL Injection example: If an application that escapes single quotes by doubling them up accepts a username and password, the

username is limited to (say) 16 characters, and the password is also limited to 16 characters, the following username/password combination would execute the `shutdown` command, which shuts down SQL Server:

```
Username: aaaaaaaaaaaaaaa'
Password: ' shutdown
```

The application attempts to escape the single quote at the end of the username, but the string is then cut to 16 characters, deleting the "escaping" single quote. The result is that the password field can contain some SQL if it begins with a single quote. The query might end up looking like this:

```
select * from users where username='aaaaaaaaaaaaaaa'' and password='''
shutdown
```

Effectively, the username in the query has become:

```
aaaaaaaaaaaaaaa' and password='
```

so the trailing SQL runs, and SQL Server shuts down.

In general, this technique applies to any length-limited data that includes escape sequences. There are obvious applications for this technique in the world of perl, because perl applications have a tendency to call out to external scripts.

### *Multiple Attempts*

Even if all you can do is write a single value somewhere in memory, you can normally upload and execute shellcode. Even if you don't have space for a good exploit (perhaps you're overflowing a 32-byte buffer, although that's enough for `execve` or `winexec`, with space left over), you can still execute arbitrary code by writing a small payload into some location in memory. As long as you can do that multiple times, you can build up your exploit at some arbitrary location in memory, and then (once you've got the whole thing uploaded) trigger it, because you already know where it is. This technique is very similar to the excellent non-executable stack exploit technique when exploiting format string bugs.

This method might even be applicable to a heap overflow situation, although the target process would have to be very good at handling exceptions. You just use your "write anything anywhere" primitive with repeated attempts to build up your payload, and then trigger it by overwriting a function pointer, exception handler address, VPTR, or whatever.

### *Context-Free Length Limits*

Sometimes a given data item can be submitted multiple times in a given set of input, with the length limit applied to each instance of the data, but with the data then being concatenated into a single item that exceeds the length.

Good examples of this are the HTTP host header field, when taken in the context of Web Intrusion Prevention technologies. It's not unusual for these things to treat each header separately from the others. Apache (for example) will then concatenate the host headers into one long host header, effectively bypassing the host header length limit. IIS does something similar.

You can use this technique in any protocol in which each data item is identified by name, such as SMTP, HTTP parameters, form fields and cookie variables, HTML and XML tag attributes, and (in fact) any function-calling mechanism that accepts parameters by name.

## Windows 2000 SNMP DOS

Though not an exceptionally exciting bug, this issue illustrates the principles behind the instrumented investigation technique pretty well. You can find the relevant Microsoft Knowledge Base article at `support.microsoft.com/default.aspx?scid=kb;en-us;Q296815`, and the NGS advisory can be found at `www.ngssoftware.com/advisories/snmp-dos/`.

In a moment of boredom while testing some SNMP walk code (common to SNMP implementers), we decided to see whether we could cause an overflow in the Microsoft SNMP daemon. We went through the usual process of attaching a debugger, RegMon, and FileMon; taking a quick peek at `HandleEx` to see what resources the SNMP daemon had open; and using Performance Monitor to keep a track of what resources the SNMP process was using—and then we ran a few quick tests, firing off some manual requests with a malformed BER structure (lengths not corresponding correctly and so on). Little appeared to happen, so we took a peek at which SNMP OIDs were present when we walked the entire tree.

Again, nothing terribly interesting seemed to be present, but then when we went back into Performance Monitor we noticed that the daemon had apparently allocated about 30MB of memory.

Running another SNMP walk, the SNMP process again allocated a large amount of memory. We then stepped through the SNMP walk code, keeping a close eye on the amount of memory allocated by the SNMP process. We found that the problem appeared to occur when requesting printer-related values in the `LanMan mib`.

It turns out that a single SNMP request (that is, a single UDP packet) causes an allocation of 30MB. It's ridiculously easy (and very quick) to consume all

available memory this way, with a few thousand packets, and the entire server is crippled. No new processes will start, no new windows will be created, and if anyone attempts to log in (perhaps in order to attempt to shut down the SNMP service or the server itself), they will fail because the Microsoft Graphical Identification and Authentication (GINA), the DLL that controls logins, doesn't have enough memory available to create the dialogs it needs in order to obtain the user's credentials. The only way out is to power down.

So in this case, discovery of the bug was based on closely observing memory usage in the target process. If we hadn't been looking at the memory usage, we'd never have seen the bug.

## Finding DOS Attacks

The previous example illustrates another excellent technique for finding DOS attacks—closely monitoring resource utilization. Eliminating resource leaks is a difficult problem, and most large applications have leaks in one form or another. This is the sort of bug that's easy to spot with good instrumentation and almost impossible without it. So, how do we go about monitoring for this kind of thing?

In Linux, the `proc` tree is pretty informative (`man proc`); it lists files that a process has open (`fd`), memory regions that the process has mapped (`maps`), and the virtual memory size in bytes (`stat/vsize`). `statm` is also somewhat useful; it provides page-based memory status information.

In Windows, the story is slightly different. The standard task manager can be helpful in getting a rough idea of resource utilization, because you can fairly easily change the columns displayed in the `processes` tab. Useful things to look for are `handle count`, `memory usage`, and `vm size`.

A better way of monitoring resources in a process (if you're serious about your instrumentation) is the Windows Performance Monitor, which can be started by running `perfmon.msc` in Windows 2000 or via the Administrative Tools Start menu option.

Performance Monitor is an excellent source of numerical information about processes, because it allows you to create custom histograms including all the items you'd like to monitor in the process. This gives you a view of the resource usage over time, rather than just a spot count, making it easier for you to see patterns.

Useful counters to add to the chart when you're testing a specific process are generally found in the `process` performance object—such things as handle count, thread count, and the memory usage stats. If you monitor these numbers over time, you'll be much more likely to find resource leak DOS problems.

## SQL-UDP

The Slammer worm made use of this SQL-UDP bug. You can find the NGS advisory on it at `www.ngssoftware.com/advisories/mssql-udp.txt`.

In the course of a consultancy engagement for a client, NGS was asked by the client to look at the different protocols supported by SQL Server, because they formed a point of concern for the client. Specifically, the client had seen UDP traffic flying around the network and was aware of the possibility of forged UDP packets. The client was concerned about the security implications of this strange UDP-based protocol and wanted to clearly establish whether or not he should block this traffic within the networks. The team began to examine the protocol.

Based on information published by Chip Andrews relating to his splendid tool `sqlping`, the team was aware that by sending a single-byte UDP packet containing the byte `0x02`, the targeted SQL Server would respond with details of the protocols that would be used to connect to the various instances of SQL Server running on the host.

The obvious place to start was, therefore, looking at what other leading bytes in the packet did (`0x00`, `0x01`, `0x03`, and so on). The team instrumented various instances of SQL Server with `FileMon`, `RegMon`, debuggers, and so forth and started making requests.

David noticed (via `RegMon`) that when the first byte of the UDP packet was `0x04`, SQL Server attempted to open a registry key of the form:

```
HKLM\Software\Microsoft\Microsoft SQL
Server\<contents_of_packet>\MSSQLServer\CurrentVersion
```

The next thing to do was clearly to append a large number of bytes to the packet. Sure enough, SQL Server fell over with a vanilla stack overflow.

At this point it was pretty clear that the client should really think about blocking UDP 1434 throughout the network. The team continued, the investigation thus far having taken about five minutes.

Several other leading bytes exhibited interesting behaviors. `0x08` triggered a heap overflow when the lead byte was followed by a long string, colon, and then a number. `0x0a` caused the SQL Server to reply with a packet containing the single byte `0x0a`—therefore, you could easily set up a network utilization denial of service by forging the source address of one SQL Server and sending a packet with a `0x0a` in it to another SQL Server.

## Conclusion

To digress into the social issues around vulnerability research for a moment, the frightening thing about the SQL-UDP bug was the speed with which the stack overflow was found; literally five minutes of investigation was all it took. It was obvious to us that if we could find the bug this quickly, other, perhaps less responsible, people would also be very likely to find it and possibly use it to compromise systems. We reported the bug to Microsoft in the usual manner, and both we and Microsoft were extremely vocal about trying to get organizations to apply the patch and block UDP 1434 (it's only used if an SQL client is unsure of how to connect to an SQL Server instance).

Unfortunately, a large number of organizations did nothing about the bug and then, exactly six months after the patch was released, some (as yet unknown) individual decided to write and release the Slammer worm, causing significant Internet congestion and imposing an administrative headache on thousands of organizations.

While it's true that the Slammer worm could have been much worse, it was still depressing that people didn't protect themselves sufficiently to thwart it. It's difficult to see what security companies can do to prevent this kind of problem from occurring in the future. In all the most widely reported cases—Slammer, Code Red (based on the IIS `.ida` bug found by another of this book's co-authors, Riley Hassel), Nimda (the same bug), and the Blaster worm (based on the RPC-DCOM bug that The Last Stage of Delirium group found)—the companies involved worked responsibly with the vendors to ensure that a patch and good-quality workaround information was available before publishing information about the vulnerabilities. Yet, in each case someone built a worm that exploited the bug and released it, and caused massive damage.

It's tempting to stop researching software flaws when this kind of thing happens, but the alternative is far worse. Researchers don't create the bugs, they find them. Microsoft released 72 distinct security patches in 2002, 51 in 2003, 45 in 2004, 55 in 2005, and 78 in 2006. Many of these patches fixed multiple security bugs.

If you are a Linux user, don't be too dismissive of these problems. According to the US-CERT 2005 year-end survey, Linux had more vulnerabilities during that year than Windows—although this figure is somewhat difficult to pin down, because it depends on how you categorize issues. The SSH and Apache SSL and chunked-encoding bugs are good examples of Linux problems.

If you are a Macintosh user, you still cannot dismiss Microsoft with its viruses and worms and Linux with its SSH and SSL flaws and multitude of privilege elevation issues. The number of people actively researching and publishing flaws on the Mac platform is currently small, but just because no one is looking for bugs doesn't mean that they aren't there. Time will tell.

If you imagine a world in which no one had carried out any vulnerability research—either for legal reasons or because they just couldn't be bothered—all these phenomenally dangerous bugs would still be there and available for use by anyone who wanted to take control of our machines and networks for whatever reason. We would have little hope of defending ourselves against criminals, governments, terrorists, and even commercial competitors because of the absence of information. Because people have found these bugs, vendors have had to fix them, and we therefore have had some measure of defense.

Vulnerability research is simply a process of understanding what's running on your machine. Researchers don't create flaws where previously there were none; they simply shed light on what we (and our customers) are running in our networks. Hopefully, this book will help you understand the problems and further illuminate the subject.