CHAPTER 16

# Fault Injection

*Fault injection* technologies have been used for more than half a century to verify the fault tolerance of hardware solutions. Fault injection systems are currently used to test the machinery in the cars we drive, the engines in the airplanes that fly us, and even the heating elements that warm our coffee. These systems inject faults through the pins of integrated circuits, via bursts of EMI, by altering voltage levels, in some cases, even through the use of radiation. These days every major hardware manufacturer employs some sort of fault injection system within their testing process.

As our technologies transcend from analog to digital, the amount of software in use grows at an exponential rate. The question that should be asked is: What tools do we have that will test the dependability of our software?

During the past decade, several fault injection solutions have been developed to detect serious problems in enterprise software. Many of these software-based fault injections solutions were created during the course of several research grants sponsored by the Office of Naval Research (ONR), Defense Advanced Research Project Agency (DARPA), National Science Foundation (NSF), and the Digital Equipment Corporation (DEC). Software fault injections systems such as `DEPEND`, `DOCTOR`, `Xception`, `FERRARI`, `FINE`, `FIST`, `ORCHESTRA`, `MENDOSUS`, and `ProFI` have demonstrated that fault injection technologies can be used to successfully enumerate a variety of faults in enterprise software applications.

Several of these solutions were each designed to help solve the same problem—to offer a resource to the software development community that will allow them to test the fault tolerance of their software. Few solutions in the public and private sectors have been designed specifically to discover security holes in targeted software. As the importance of security grows daily, so does the need for technologies to help improve the security of the software we use.

Fault testing tools are used every day by Quality Assurance (QA) engineers to test their assigned software for potential weaknesses. One of the most useful skills that QA engineers can possess is the ability to incorporate automation into their toolkits. Software security auditors could learn much from modern QA techniques. Most talented security auditors rely on manual auditing techniques, primarily reverse engineering and source auditing, to discover potential security problems in software products. While these skills are useful, if not required, in a successful auditor, the ability to develop automated auditing technologies is also important. By using the knowledge discovered during reversing, software testers can quickly configure their auditing applications to audit software while they perform other auditing tasks. This type of multitasking allows an auditor to perform the work of hundreds, if not thousands, of other software auditors in a fraction of the time.

One of the best facets of fault testing is that every mistake you make during the development of your solution may actually increase the success of your testing. A mistake in your development is one of the most serendipitous things that you can do. If you went back and made a list of all the programming mistakes you've made over time and built a test for each into your fault-testing application, you could easily break the majority of enterprise server software products.

Building a fault injection solution will motivate you to learn the attack classes to such depth that you will understand them at a much simpler level. With each new attack class you learn or discover, you will pick up tricks and techniques that will help you understand the other classes. What you learn can make your auditing suite even more powerful. The best part is that by using automation, you can even find world-shattering security holes while you sleep.

In this chapter we will design and implement a fault injection solution to discover security flaws within network server software products that operate over an application protocol-based network medium. This fault injection system, which we'll call RIOT, closely resembles a system designed in January 2000 that was used to discover several highly publicized vulnerabilities such as those exploited by the Code Red virus. Using RIOT, we demonstrate the effectiveness of fault testing by enumerating some of these security flaws in our target application, Microsoft's Internet Information Server (IIS) 5.0.

## Design Overview

The building blocks of our fault injection system are shown in Figure 16-1. Most fault injection systems can be broken down and categorized in a similar manner. We will discuss each of these components in depth throughout this chapter; later we will put the pieces together and build RIOT.

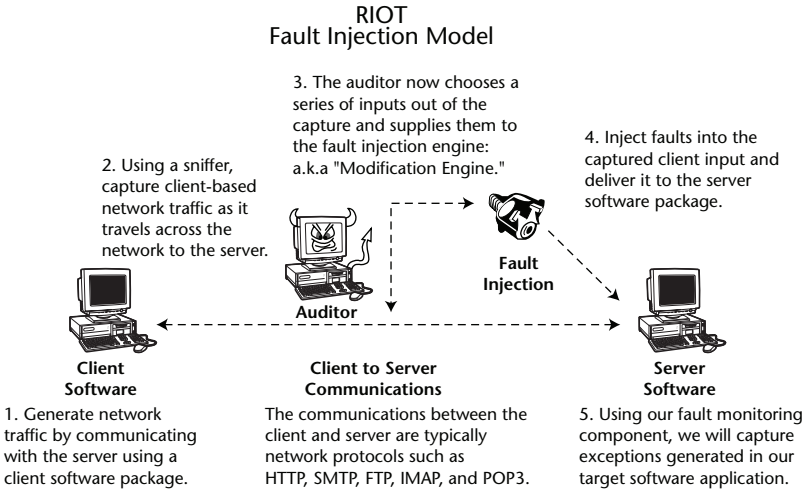


**Figure 16-1:** RIOT Fault Injection Model

## Input Generation

Various mediums can be used to gather input for our fault injection. We will mention a few in this section, but as you'll discover, many others are available. Our input can be divided into different test supplements. Each supplement will seed the generation of a series of tests. The amount and type of data in our input will determine what tests will be performed. While our input can be gathered without any regard to its contents, our effectiveness at discovering flaws in our target software will dramatically increase if we supply input that was used to communicate with esoteric and untested software features.

For our examples we will focus on application protocol input, such as the first client state of an `HTTP` transaction. We could begin gathering input for our tests by capturing network traffic from browser sessions with a production Web server. Let's assume we captured the following client request while monitoring local network traffic:

```
GET /search.ida?group=kuroto&q=riot HTTP/1.1
Accept: */*
```

```
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Host: 192.168.1.1
Connection: Keep-Alive
Cookie: ASPSESSIONIDQNNNNTEG=ODDDDIOANNCXXXXIIMGLLNNG
```

Someone versed with the HTTP protocol at an intermediate level may notice that the extension .ida is not a standard file extension. After we conduct a little research using our favorite search engine, we discover that this extension is part of a poorly documented feature available through an ISAPI filter installed with many versions of the IIS Web server.

**NOTE Any feature that is difficult to learn, difficult to use, and difficult to like is an excellent place to begin looking for security problems. If the feature steers your attention away from the primary functionality of the program, it most likely had the same effect on its developers and testers—before it was hustled out to meet the demands of persistent customers.**

The preceding example will be supplied to the fault injection component of our test application. This fault injection component will inject faults (bad or unexpected input) into the input data by modifying it. The input that we supply to our fault injection component will greatly affect the spectrum of our tests. The quality of our input will also greatly affect our tests. If we supply input that is directly invalid, then we will spend most of our time auditing error-handling routines in our target application. For this reason, we want to spend a significant amount of time carefully gathering our input. Depending on the amount of input gathered, we may also want to manually verify the quality of it before we begin a full-scale test.

Various methods can be used to gather input that we can supply to a fault injection solution. The input method, or combination of methods, we choose will depend on the type of fault testing we will conduct.

### *Manual Generation*

Manual input generation can be very time consuming, but it generally yields the best results. We can manually create our input data using our editor of choice, saving each created test as a separate file in a directory. We can write a simple function in our program to examine this directory and read each of the test inputs, passing them one at a time onto our fault injection component. We'll use this method in our example fault injection application RIOT. We could also store our created inputs in a database or include them directly in our application. Saving inputs directly to a file saves us the trouble of building custom data structures to organize them, record their size, and handle their contents.

### Automated Generation

For simple protocols such as HTTP, we may want to generate our input. We can do this by studying the protocol and designing an algorithm to generate potential input. Input generation is extremely useful for cases where we want to test a large range of a protocol, but we don't want to manually create all the input data. During my testing experience I found that automated input generation was helpful when dealing with simple protocols that had a very reliable structure, such as most application protocols. When working with protocols that are much more dynamic in nature, offering many layers and several states, automated input generation may not be the optimal input method. A bug in an automated input generation may not present itself until several hours after you've begun testing. If you aren't closely monitoring the input as it is being generated, you may not notice that the generated input is problematic.

### Live Capture

A few solutions, such as ORCHESTRA, offer the ability to inject faults directly into existing protocol communications. This method is very effective when testing complex state-based protocols. The only downfall is the requirement of the user to define the protocol so that alterations can be made to guarantee the successful data delivery. For example, if you alter the size of data within a protocol message, you may be required to also update various length fields to mirror the changes you've made. One of the few groups to overcome this similar problem was the group of researchers who developed ORCHESTRA; they used protocol stubs to define necessary characteristics of the protocol.

### "Fuzz" Generation

During the late 1980s and early 1990s, three researchers—Barton Miller, Lars Fredriksen, and Bryan So—conducted a study on the integrity of common Unix command-line utilities. During a thunderstorm late one night, one of the researchers was attempting to use some standard Unix utilities over a dial-up connection. Due to the line noise, seemingly random data was sent to the Unix utilities instead of what he was typing in his shell. He noticed that many of the programs would core dump when he tried to use them because of this random data. Using this discovery, the three researchers developed *fuzz*, a program designed to generate pseudo-random input data that could be used to test the integrity of their applications. Fuzz-input generation has now become a part of many fault injection suites. If you would like to learn more about fuzz, visit the archive at `http://www.cs.wisc.edu/~bart/fuzz/`.

Many current auditors believe that using fuzz input is like shooting bats in the dark. During the course of the fuzz project, these three researchers discovered

integer overflows, buffer overflows, format bugs, and generic parser problems in a wide array of applications. It should be noted that a few of these attack classes did not become publicly known and accepted until more than a decade after this research.

## Fault Injection

In the previous section, we discussed methods to generate input to be used by our fault injection component. In this section we'll talk about modifications we can make against our input that will generate faults, such as exceptions, in the application we will be testing.

This phase of the process is what really defines the solution. Although the methods used to gather input remain similar across all fault injection solutions, the methods used to inject faults and the types of faults that are injected are dramatically different. Some of the fault injection solutions require source access so that modifications can be made to the program being tested that will allow the auditor to gather information at runtime. Because our fault injection suite is targeted against closed-source applications, we will not need to modify the application in any way; we will only modify input data normally passed to the target application.

## Modification Engines

Once our collected input is processed and passed to our modification engine, we can begin inserting faults into the input data. We'll need to keep a virgin copy of the input data in memory that we can acquire, modify, and deliver for each iteration in our engine. In this case an iteration is simply one sequence of injecting a fault and delivering the modified input to the target application. The sample modification engine that is provided with this book at `http://www.wiley.com/go/shellcodershandbook` is geared toward the discovery of buffer overflow vulnerabilities. This engine will break the input stream apart into elements, insert a fault into each element (in this case a variable-sized buffer of data), and finally send it to the target application. The sample engine we will use is different from other fault injection systems previously developed. Instead of blindly inserting faults in a sequential fashion, our engine will examine the input and determine where to insert faults based on the contents of the input. The sample engine will also mimic inserted faults to their surrounding data so that during our auditing we will not be circumvented by common input sanitization schemes. These and other differences will dramatically increase the efficiency of our fault testing.

If you've ever written a fault injection application, you've probably iterated through data sequentially, injected faults, and delivered them to the target software application. Without any optimizations on the fault injection logic, you probably noticed that each test session required much time to complete,

and that many unnecessary tests were performed. By making a few simple alterations to our fault injection logic, we can dramatically minimize the amount of tests that need to be performed. Let's take the sample input stream:

```
GET /index.html HTTP/1.1
Host: test.com
```

We'll assume that we've just begun a test, and our index is positioned at the first byte of our `HTTP` method `G`. During our first iteration we'll insert a fault at this position. We'll then deliver this modified input to our target. After delivery is complete, we'll insert our next fault at the same position and deliver the modified input. This will continue until we've cycled through the possible faults we can inject. Our next step is to advance our index to the next position, the second byte of the `HTTP` method, or `E`. We'll repeat the modification and deliver each fault as we did with the previous index position. In other words, for each position we will perform a modification for every fault we offer. If we have 10 inputs, each with 5,000 possible positions, and our engine offers 1,000 faults, we should be able to buy a flying car when this test is done.

Instead of sequentially inserting faults throughout the entire input stream, we can separate our data into elements using delimiter logic. Then we insert our faults at the offset of each element instead of at every single offset in our input. The preceding sample input stream has a method, URI, protocol version, header name, and a header value. To break it down even further, we should also notice the file extension on the URL, the major and minor versions of the protocol, and even the country code or DNS root of the hostname. Building support into our auditing suite by hand for every element of every protocol we want to test is a horrendous task. Luckily, there is much simpler way to accomplish it.

### *Delimiting Logic*

When developers create their parsers, they rarely if ever create markers that fall into the alphabetic or numeric system. Markers are usually viewable symbols such as `#` or `$`.

To explain this concept, let's look at the following. If we used the value `1` to separate the protocol major and minor versions, how could we determine the protocol version in the following input stream?

```
GET /index.html HTTP/111
```

If we used alphanumeric values as parse symbols, how could we name or describe our data? We can read information with the use of special symbols, such as the period in this case:

```
GET /index.html HTTP/1.1
```

The balance and frequency of the distribution and separation of elements in information is the basis of communication. Imagine how difficult it would be to read this chapter if we removed every non-alphanumeric value—no spaces, no periods, nothing but letters and numbers. One of the great things about the human mind is that we can make decisions based on what we've learned throughout our lifetime. So, we could look at this unorganized mess of information and in time determine what's valid and what isn't. Unfortunately, the software used by our infrastructures is not as intelligent. We must format our information using the appropriate protocol standard so that we can communicate with the appropriate software.

Formatted data used in application protocols relies primarily on the concept of *delimiting*. Delimiters are usually printable ASCII values that are non-alphanumeric. Let's take another look at the sample input stream; this time we'll escape characters normally not visible by a `\`:

```
GET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
```

Notice that every element in the sample protocol input stream is separated by a delimiter. The method is delimited by a space, the URI is delimited by a space, the protocol name is delimited by a forward slash, the major version by a period, the minor by the a carriage return and new line; the header name is delimited by a colon, and the header value follows, delimited by two carriage returns and new lines. So, merely by inserting our faults around special symbols in our input stream, we can fault test nearly every element of the protocol without knowing anything about them. We should insert our faults before and after these special symbols, so we won't have problems auditing assignments or boundaries inside our input stream.

A sample run with ten iterations using the fault `EEYE2003` would produce the following faulted input streams.

Sequential fault injection:

```
EEYE2003GET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GEEYE2003ET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GEEEYE2003T /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GETEEYE2003 /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /EEYE2003index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /iEEYE2003ndex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /inEEYE2003dex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indEEYE2003ex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indeEEYE2003x.html HTTP/1.1\r\nHost: test.com\r\n\r\n
```

Fault injection using delimiter logic:

```
GETEEYE2003 /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
```

```
GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /EEYE2003index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indexEEYE2003.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.EEYE2003html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.htmlEEYE2003 HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html EEYE2003HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTPEEYE2003/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTP/EEYE20031.1\r\nHost: test.com\r\n\r\n
```

We can see the performance increase even in this small input stream. In a system that uses an average of several thousand data streams, with a nearly infinite number of possible faults, this optimization can save us weeks, months, if not years of testing time. Anybody can write something that will find a security hole within a few years; very few can write something to do it in five minutes.

### *Getting around Input Sanitization*

Now that we've talked about where we'll be inserting our faults, let's talk about the actual faults we'll be inserting. Assume that in our first modification engine focused toward the discovery of buffer overflow vulnerabilities, we chose to use one fault. Our one fault is a 1024-byte buffer filled with the character `X`. By using this one fault we may find a few problems in our target software package, but it is very unlikely because of size limiting and content limiting. If we can't get our fault-injected data past the initial input sanitization that our target software performs, we'll spend most of our test time bouncing off of error handling routines.

Our target applications will usually limit the initial size of each element of the protocol. For example, the `HTTP` method may be limited in size to 128 bytes, but later the method takes the input stream and copies it into a static buffer of 32 bytes. Because the fault we chose to inject is 1024 bytes (well over the 128-byte limit) the target software application will drop the input stream and return an error. Our fault will never be delivered to the vulnerable buffer.

We could use a spectrum of buffer sizes, for example 1 through 1024, with 1-byte increments. Given an input stream with several hundred elements, each of which we'll be injecting with our 1024 possible faults, the time required to perform this type of test may be unreasonable. Therefore, a spectrum of automatically generated sizes is not the most appropriate.

When dealing with closed-source applications, you can often learn a lot by looking at how developers have implemented certain data structures such as buffer sizes. When auditing a closed-source `HTTP` server, you can analyze the source code of several open source server packages such as Apache, Sendmail, and Samba. Greping through the source, you can determine what common buffer sizes are used. You will discover that the majority of the buffer sizes are

multiples of powers of 2, starting at 32; for example 32, 64, 128, 256, 512, 1024, and so on. Others are powers of 10. The rest are based on the same scheme but have a variable number added to or subtracted from them. This variable-sized number is usually between 1 and 20.

Using these statistics, you can create a table of buffer sizes that could potentially trigger a majority of buffer overflow vulnerabilities. Add a small delta before and after the buffer sizes to account for any common additions noticed in variable declarations. An effective method of confirming that we have good fault input is to run tests against vulnerable software that is known to contain particular buffer overflow vulnerabilities. Using the table of buffer sizes, you will find that the table of input will be able to reproduce each buffer overflow in the target software. Instead of having 70,000 possible fault injections per protocol element, we now have approximately 800.

Enterprise software applications will often verify input content before passing into internal routines to avoid potential problems. This behavior isn't directly a result of secure programming but it makes the discovery and exploitation of security holes slightly more difficult. If we want to audit the darker corners of a software product, where many undiscovered vulnerabilities exist, we'll need to navigate around the various content restrictions. Often fields will be limited to numbers, uppercase letters, or encoding schemes. The C functions `isdigit()`, `isalpha()`, `isupper()`, `islower()`, and `isascii()` are generally used to restrict content.

If we inject a fault that contains non-numeric data into a protocol element that can only hold numeric data, the software product will return an error after noticing that a call to `isdigit()` failed. By injecting faults that mirror surrounding data, we can navigate around the majority of these restrictions. We can inject a fault filled with the byte value of the surrounding protocol element. Let's compare a normal fault injection session with a fault injection session that injects faults to mirror surrounding data.

A sample run with the buffer size 10 would produce the following fault injected input streams:

```
GETTTTTTTTT /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET //////////index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET           /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /iiiiiiiiiiindex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indexxxxxxxxxxx.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.hhhhhhhhhhhtml HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.htmmmmmmmmmm HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HHHHHHHHHHTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTPPPPPPPPPP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTP/11111111111.1\r\nHost: test.com\r\n\r\n
```

## Fault Delivery

Current hardware fault injection equipment delivers faults by altering voltage levels, injecting data using test pins, and even bursts of EMI. Faults can be delivered to software over any medium from which the software application accepts input data. In the Windows operating system, this could be through the filesystem, registry, environment variables, Windows messages, LPC ports, RPC, shared memory, command-line arguments, or network input, as well as a variety of other mediums. The most vital communication mediums used by today's software applications are the TCP/IP network protocols. Using these protocols, we can communicate with vulnerable software products from the far reaches of the globe. In this section we'll discuss some methods and guidelines for delivering faults over network protocols.

Delivery is initiated from the modification engine. For each iteration of our modification engine, we will deliver the modified input to our target application using a suite of network functions designed to deliver data over TCP/IP connections. After we modify our input data, we will then deliver our modified data as follows:

1. Create network connection to target application.
2. Send our modified input data over the created connection.
3. Wait momentarily for a response.
4. Close the network connection.

## Nagel Algorithm

The *Nagel* algorithm, enabled by default in the Windows IP stack, will delay the transmission of smaller datagrams until enough have been delayed that they can be grouped together. Because our tests are created, delivered, and monitored as separate entities, we want to disable this by setting the `NO_DELAY` flag.

## Timing

The issue of timing is difficult to resolve. Many would choose to have very flexible timing to allow the server to respond. Others may choose to trim on timing to reduce test time. RIOT is configured somewhere in between. It is recommended that you have configurable timing so that it best suits the software product you choose to audit. Server applications that will not respond unless you've sent valid input should most likely be permitted a very short timeout. Server applications that always respond regardless of the type of request

should be permitted a higher timeout. The optimal solution would be to write your own timing algorithm that configures timing dynamically during audit initialization.

### Heuristics

We've always been a fan of software that can adjust itself to fit the situation. Though basic heuristics is far from real artificial intelligence, it's an interesting step in the right direction and offers fault injection an added edge. *Heuristics* is the science of communicating and watching responses to educate the communicator. If you want to incorporate simple heuristics into your fault injection suite, simply add support for a call back directly after the receive portion of your delivery code. You could start by examining server responses for custom error codes. When your auditing application receives an error, such as the Internal Server Error, you could set a flag so that auditing would temporarily become more aggressive until the response changes back. While these types of Web server errors may occur because of poor configuration or the failure to initialize a feature, they can also occur due to the corruption of the process address space.

### Stateless versus State-Based Protocols

We can break down protocols into two classes—*stateless* protocols and *state-based* protocols. Stateless protocols are very easy to audit—all we do is funnel our fault data at the remote server application and monitor its behavior. State-based protocols are a little more difficult to audit. Few if any fault injection solutions offer the ability to audit complex state-based protocols. This problem stems from the complexity of protocol negotiation. Software products often incorporate complex client-to-server protocols that require negotiation so granular that simple logical analysis cannot reproduce it.

Few researchers have been able to develop successful state-based auditing systems that work entirely on logical analysis of protocol data. The only solutions to this problem require additional code and/or complex protocol stubs that define each state of the protocol.

## Fault Monitoring

*Fault monitoring*, a step that's often grossly overlooked, is a crucial part of fault testing. The majority of fault injection projects developed by the academic community detect failures in an application only if it crashes or dumps its core. Enterprise applications are almost always built with a strong fault tolerance using exception handling, signal handling, or any other fault handling

available from the overlying operating system. By monitoring our faults using the operating system's debugging subsystem, we can detect many faults that were previously overlooked.

## Using a Debugger

If you are interactively fault testing, a debugger will suit your needs. Choose your debugger and attach the process of the software product you are auditing. Many debuggers are configured by default to catch only exceptions that are not handled by the process; for example, unhandled exceptions. Other debuggers allow you to catch only unhandled exceptions. If your debugger is capable of catching exceptions before they are passed to the application "first chance," we recommend you enable this feature for every type of exception that you want to monitor. The most important exceptions to monitor are access-violation exceptions. Access violations are generated when a thread in the process attempts to access an address that isn't valid in the address space of the process. These violations are often seen when data structures designated to reference memory are corrupted during the operation of the program.

## FaultMon

Unfortunately very few debuggers out there will allow you to log exceptions and automatically continue operation. For this reason, we've provided FaultMon, a utility written by Derek Soeder, a member of the eEye research group, on *The Shellcoder's Handbook* Web site (`http://www.wiley.com/go/shellcodershandbook`). To use FaultMon, simply open a command prompt and issue the process ID for the application for which you want to monitor exceptions. Each time an exception is generated, FaultMon will display information about the exception to the console.

```
21:29:44.985  pid=0590 tid=0714  EXCEPTION (first-chance)
     ------------------------------------------------------------------
            Exception C0000005 (ACCESS_VIOLATION writing [0FF02C4D])
     ------------------------------------------------------------------
       EAX=00EFEB48: 48 00 00 00 00 00 F0 00-00 D0 EF 00 00 00 00 00
       EBX=00EFF094: 41 00 41 00 41 00 41 00-02 00 41 00 41 00 41 00
       ECX=00410041: 00 00 00 A8 05 41 00 0F-00 00 00 F8 FF FF FF 50
       EDX=77F8A896: 8B 4C 24 04 F7 41 04 06-00 00 00 B8 01 00 00 00
       ESP=00EFEAB0: 38 25 F9 77 70 EB EF 00-94 F0 EF 00 8C EB EF 00
       EBP=00EFEAD0: 58 EB EF 00 89 AF F8 77-70 EB EF 00 94 F0 EF 00
       ESI=00EFEB70: 05 00 00 C0 00 00 00 00-00 00 00 00 B4 69 CC 68
       EDI=00000001: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
       EIP=00410043: 00 A8 05 41 00 0F 00 00-00 F8 FF FF FF 50 00 41
         --> ADD [EAX+0F004105],CH
      ------------------------------------------------------------------

Continue? y/n:
```

Here we see a sample exception that was captured by FaultMon during a RIOT test. The interactive option was set to `-i`. By having the interactive option set, we can pause between exceptions and examine the state of the program.

## Putting It Together

We've also included on *The Shellcoder's Handbook* Web site the source code and compiled Win32 version of the sample fault injection application, RIOT. To see RIOT in action, simply copy RIOT and FaultMon from the CD-ROM to a folder on your computer. We'll perform a sample test using the input we reviewed earlier in this chapter.

```
GET /search.ida?group=kuroto&q=riot HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Host: 192.168.1.1
Connection: Keep-Alive
Cookie: ASPSESSIONIDQNNNNTEG=ODDDDIOANNCXXXXIIMGLLNNG
```

Don't worry about creating a test for this; it's already set up and ready to go. Open two command shells (`cmd.exe`). The first command prompt must be opened on the server running the potentially vulnerable Web server you want to test. In this first command shell, run FaultMon and supply it with the process ID of the Web server that is running in the background. If you are running IIS 5.0, use the process ID of `inetinfo.exe`. If the process ID were `2003`, you would type the following command into your shell:

```
faultmon.exe –i 2003
```

As FaultMon starts, you should see a series of events displayed. You can ignore these events—they are related to FaultMon initialization and are irrelevant to our testing. Now that FaultMon is running and monitoring events, let's open another shell on our attacker machine.

The second shell should be opened on the machine where RIOT is located. In the second command shell, start RIOT by entering the target IP address of the host you are auditing as well as the port number on which the Web server is listening. If the IP address of the Web server is 192.168.1.1 and the port on which it is listening is 80, issue the following command:

```
riot.exe –p 80 192.168.1.1
```

The input files supplied with RIOT will allow you to rediscover various buffer-overflow vulnerabilities that have been found in enterprise Web servers. If you choose to audit a Microsoft Windows 2000 server with an early service pack, you may just rediscover the security flaw that lead to the success of the Code Red worm.

Each file in the input folder contains input data for a particular test. RIOT will start with test ID 1 and increment until it runs out of tests. You can edit these files and create your own tests as you'd like. There is also source code included that should give you a nice framework to start with. Happy hunting.

## Conclusion

In this chapter, you learned about the concept of fault injection, which is closely related to fuzzing. We demonstrated how to create faults with a new application, RIOT, and monitor the results on the targeted application with FaultMon.