CHAPTER 4 State-Based Attacks

What's In This Chapter?

The concept of **state**, or the ability to remember information as a user travels from page to page within a site, is an important one for Web testers. The Web is stateless in the sense that it does not remember which page a user is viewing or the order in which pages may be viewed. A user is always free to click the Back button or to force a page to reload. Thus, developers of Web applications must take it upon themselves to code state information so they can enforce rules about page access and session management. This chapter contains a series of attacks that will help determine if your Web application does this important task correctly and securely.

This chapter presents the most common and notorious Web vulnerabilities.

Introduction

All Web sites have a designated "home" or "default" page that Web designers intend as the starting point for visitors. From that start page, users can navigate the various pages of the site by clicking hyperlink objects embedded in the various pages of the site. Hyperlinks can be text, images, or other objects on the page.

This is the way it is supposed to work anyway. The problem is that the Web has no built-in mechanism that specifies which sequence of Web pages and forms are presented to the user. This aspect of the Web is called **statelessness** to denote that each page is delivered to users without knowledge of where the users were previously or restrictions about where they can go next. Users can simply type in the URL of the page they want to load, skipping the start page and any other page they do not need to view.

If restrictions about page access are important, it is up to the Web application to enforce this.

Statelessness is ideal when browsing for information (or **surfing**, as it has become commonly known), but more has been demanded of the Web than surfing static, standalone pages, and statelessness can lead to any number of failures and security violations. Imagine surfing past the pages where credit card numbers are entered and going directly to the page where the receipt is displayed—obviously not something you want your own Web application to do!

The burden of including state information in a Web application falls squarely on the shoulders of the Web developer and the tools for adding such state information to a Web application are not particularly sophisticated.

The first option is using **forms** and **CGI parameters**, which allow the transfer of small amounts of data and information to be passed from page to page, essentially allowing the developer to bind together pairs of pages. More sophisticated state requirements mean that data needs to be stored, either on the client or the server, and then made available to various pages that have to check these values when they are loaded.

For example, we may store a flag on the server that indicates whether a user has entered a valid credit card. The Web application will then only allow the purchase pages to be loaded (and the purchase to be confirmed) if that flag is set to the correct value.

Shopping carts, purchase history, shipment tracking, and other such features require some state to be made available to the Web application. These features and the need to store state in general (and attacks on that state data) are the subject of this chapter.

ATTACK 6 Hidden Fields

One of the most basic ways of preserving state in Web pages is to hide data in the page. That way as a user browses pages, state information can be carried along, allowing the Web application to give the user a smooth browsing experience.

The most common ways of doing this are to place data in hidden form fields or to append data as CGI parameters to hyperlinks. Both methods have the same effect, but hidden fields are less obvious to the user.

When a form is submitted to the Web server, each of the form fields is passed to the server as `GET` or `POST` parameters. (Don't worry about these at the moment. We look at these in detail in the next attack.) But it's not only the fields that the user can see that are passed; hidden fields are passed, too, and the Web application can read them just like normal fields, and understand whatever data they contain. Developers sometimes favor hidden fields because they are easy to include at design time. Hidden fields have two other benefits. First, nontechnical people can maintain them in applications like FrontPage, Dreamweaver, and so on. Second, they are not obvious to a casual user.

The problem is that hackers are not casual users. They can and will read hidden fields. If the information these fields contain is useful in an attack, you can safely assume that hackers will use it that way.

You can store numerous things in hidden form fields. Not all of them are state related, but you should treat them with suspicion when they are discovered. The basis of this attack is to look for hidden fields within forms, analyze what they are used for, and try to change their values in ways that would benefit an attacker.

WHEN to Apply This Attack

The easiest way to determine if this attack is possible is to view the source of the page and search for the string `"hidden"`. Most form elements follow this structure:

```
<input name="is" value="1234" ... >
```

along with the possibility of other, additional attributes. The type `"hidden"` is one such attribute that appears in the source of a Web page, as follows:

```
<input name="id" value="1234" type="hidden">
```

The most primitive way of modifying these form elements is to save the page locally (using File, Save As in your browser while the page is displayed) and remove the `"type=hidden"` text from the source (remembering, as always, to change any relative links to absolute links so that everything still points to the correct location when you reload the locally saved copy of the page). This effectively changes the hidden field to a standard text box, which you can see and modify directly in the browser.

An alternative way of identifying hidden fields is to use the browser's Document Object Model (DOM). Both Internet Explorer and Firefox have programming interfaces that allow developers to query the document within the browser and change some of its attributes. This functionality was originally intended for dynamic HTML so that scripting languages like JavaScript or VBScript could implement dynamic UI functionality, as described in Chapter 3, "Attacking the Client."

Consider the DOM code that follows, which iterates over a document in Internet Explorer and prints the names and values of all hidden fields:

```
using System;
using mshtml; // access to IE's DOM
```

```
IHTMLElementCollection tags;  // interface to HTML
 document

// iterate through all HTML tags
tags = HTMLDocument.all;
foreach (IHTMLElement tag in tags)
{
  // Is the current tag an input tag?
  if (string.Compare(tag.tagName,?INPUT?,true) == 0)
  {
    // cast to an input tag
    IHTMLInputElement inputTag =
 (IHTMLInputElement)tag;

    // Is it a hidden input field?
    if (inputTag.type==?hidden?)
    {
      Console.Write(?hidden form field
 ??+inputTag.name+???+
          ?found. Value is ??+ inputTag.value+?? ?

      // change the field value here
      //  inputTag.value=="somevalue";

    }
  }
}
```

It is straightforward to modify this code to change any of the hidden field values to whatever value an attacker considers advantageous.

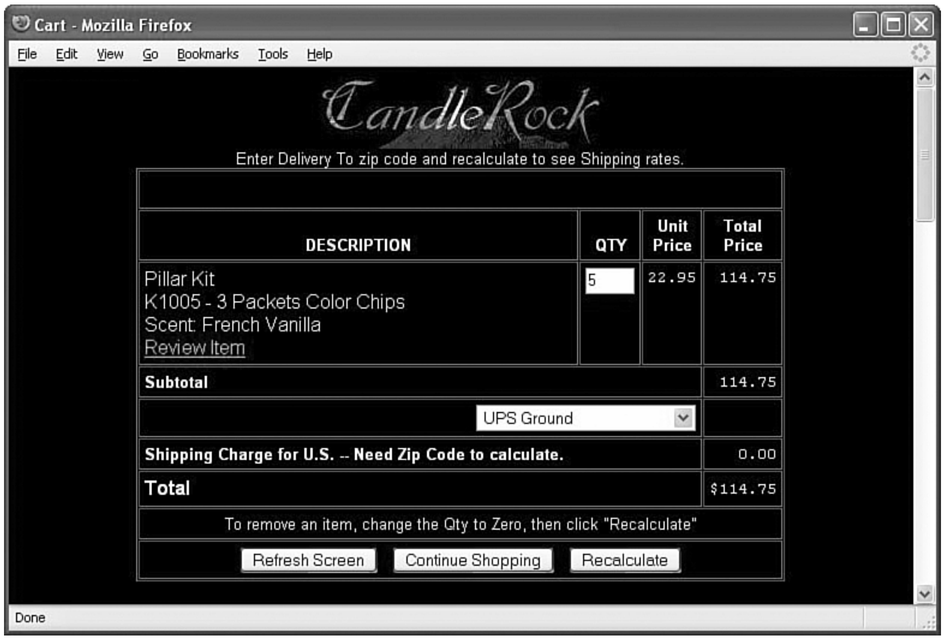
If you don't want to write the code yourself, the PageSpy tool on the CD in the back of this book uses this technique to list the hidden fields on a page and allow changes—all from a simple graphical user interface (GUI).

HOW to Perform This Attack

There is no easy recipe for this attack; it all depends on what hidden fields you find on the page and the data they contain. The most universally useful advice is to change values of hidden form fields and see what happens as subsequent pages load. This should make problems with hidden fields apparent. Consider the following example.

A really naive mistake that early Web developers made often and that people still make today is saving product information on a page and passing that information to subsequent pages as in the application shown in Figure 4-1. For example, as in Figure 4-2, we may want to save a product's price in a hidden field to help the server calculate totals as the user browses a site. If an attacker recognizes this field and modifies it, he can reduce the price of the product to whatever he likes.

FIGURE 4-1 An e-commerce application.

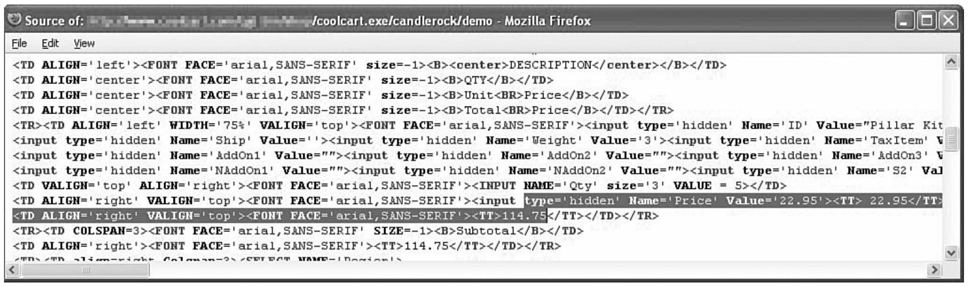


Enter Delivery To zip code and recalculate to see Shipping rates.

DESCRIPTION	QTY	Unit Price	Total Price
Pillar Kit K1005 - 3 Packets Color Chips Scent: French Vanilla Review Item	5	22.95	114.75
Subtotal			114.75
UPS Ground			
Shipping Charge for U.S. -- Need Zip Code to calculate.			0.00
Total			$114.75

To remove an item, change the Qty to Zero, then click "Recalculate"

FIGURE 4-2 Viewing the source of the application reveals a hidden field with the item's price. What would happen if we changed this value?


Source of: [redacted]/coolcart.exe/candlerock/demo - Mozilla Firefox
File Edit View

```
<TD ALIGN='left'><FONT FACE='arial,SANS-SERIF' size=-1><B><center>DESCRIPTION</center></B></TD>
<TD ALIGN='center'><FONT FACE='arial,SANS-SERIF' size=-1><B>QTY</B></TD>
<TD ALIGN='center'><FONT FACE='arial,SANS-SERIF' size=-1><B>Unit<BR>Price</B></TD>
<TD ALIGN='center'><FONT FACE='arial,SANS-SERIF' size=-1><B>Total<BR>Price</B></TD></TR>
<TR><TD ALIGN='left' WIDTH='75%' VALIGN='top'><FONT FACE='arial,SANS-SERIF'><input type='hidden' Name='ID' Value="Pillar Kit
<input type='hidden' Name='Ship' Value=''><input type='hidden' Name='Weight' Value='3'><input type='hidden' Name='TaxItem' V
<input type='hidden' Name='AddOn1' Value=""><input type='hidden' Name='AddOn2' Value=""><input type='hidden' Name='AddOn3' V
<input type='hidden' Name='NAddOn1' Value=""><input type='hidden' Name='NAddOn2' Value=""><input type='hidden' Name='S2' Val
<TD VALIGN='top' ALIGN='right'><FONT FACE='arial,SANS-SERIF'><INPUT NAME='Qty' size='3' VALUE = 5></TD>
<TD ALIGN='right' VALIGN='top'><FONT FACE='arial,SANS-SERIF'><input type='hidden' Name='Price' Value='22.95'><TT> 22.95</TT>
<TD ALIGN='right' VALIGN='top'><FONT FACE='arial,SANS-SERIF'><TT>114.75</TT></TD></TR>
<TR><TD COLSPAN=3><FONT FACE='arial,SANS-SERIF' SIZE=-1><B>Subtotal</B></TD>
<TD ALIGN='right'><FONT FACE='arial,SANS-SERIF'><TT>114.75</TT></TD></TR>
```

This is really the idea: Watch for information in all hidden fields, and ask yourself whether an attacker would find the information advantageous.

Another important thing to note is that hidden fields are data passed from a client machine to a Web server. Because hidden fields have no data type associated with them, changing their values to be illegal, overly long strings and special characters may result in crashing or otherwise adversely affecting the Web server.

Finally, you can use hidden fields to store data such as the previous page visited or the last selected action. This data can ensure that users follow the required flow of the application and don't jump to pages they shouldn't be able to access. Hidden fields can also store session information, as we shall see in a later attack.

HOW to Protect Against This Attack

Avoid hidden fields wherever possible, and most especially on information like price, quantity, page sequence, and other information you do not want your users to change. Before using these fields for anything, evaluate the data that the field contains for its security risk. Where you use hidden fields, limit their exposure by obfuscating the field name (for example, by using something less obvious than "price" or "password") and encrypting or hashing the value to something less recognizable to the attacker.

This technique, however, relies on security by obscurity, and is almost always broken over time. Something named `cX24y` is no more secure than something named `price`, but it is harder to tell what the former is and determine if it is important. If you do use hidden fields for something (they are not entirely evil—a common usage is to include them in search forms so the script that performs the functionality knows how to "brand" or frame the results), ensure that the data is what you expect. Attackers can and will modify these values.

ATTACK 7 CGI Parameters

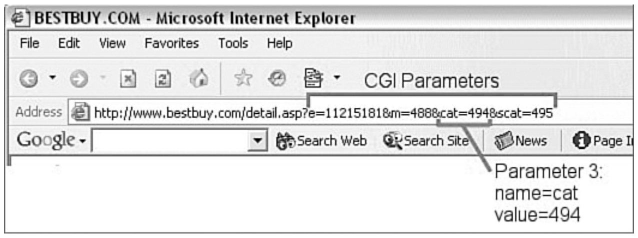
Although hidden form fields are a good way of passing data between pages, there is a big drawback in using this method: The user has to submit a form to an "action handler," usually by pressing a button. It may seem like a small point, but users are more used to clicking on hyperlinks or images for their navigation than form Submit buttons.

CGI parameters are ideal for this task. After the parameters reach the server they are accessed in the same way as form fields. (See the difference between `GET` and `POST` form methods in the next sidebar, "The Difference Between `GET` and `POST` Parameters.") You easily can attach CGI parameters to any hyperlink.

WHEN to Perform This Attack

CGI parameters are passed in a page request's URL after the ? character and are name-value pairs separated by & characters.

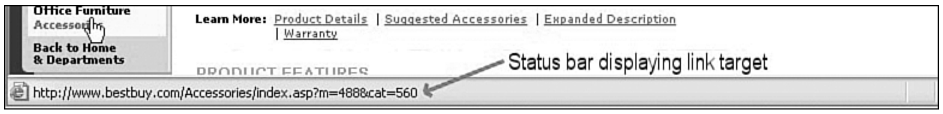
FIGURE 4-3 Example of CGI parameters in a browser's address bar.



http://www.bestbuy.com/detail.asp?e=11215181&m=488&cat=494&scat=495

It's easy to tell if the current page uses CGI parameters, because they will be clearly shown on the browser's address bar. Links from a given page and their parameters should display on the browser's status bar if this functionality is enabled.

FIGURE 4-4 CGI parameters in the status bar when a user hovers over a link.


Office Furniture
Accessories
Back to Home
& Departments
Learn More: Product Details | Suggested Accessories | Expanded Description
| Warranty
PRODUCT FEATURES
Status bar displaying link target

http://www.bestbuy.com/Accessories/index.asp?m=488&cat=560

Other than their location, we attack CGI parameters in the same way that we attacked hidden fields.

HOW to Perform This Attack

There is no single way of performing this attack. From an attacker's point of view, it all depends on what parameters he sees being passed from page to page and what their values are. As with the previous attack, we have to consider what advantage the information contained in the parameters represents to an attacker.

Begin by browsing your target site and noting the address bar. Also use your mouse to hover over clickable objects and note the URL that's usually shown at the bottom of the screen in the information bar. The data in a URL after the question mark are CGI parameters. We need to understand what the data represents and whether its exposure would benefit an attacker.

You can modify CGI parameters by editing the page's HTML, as in the hidden forms attack earlier, but for `GET` parameters, it is usually much easier to request a target page, change the values in the browser's address bar, and request the page again. There are many attacks against CGI parameters, all of which overlap with other attacks discussed in this book.

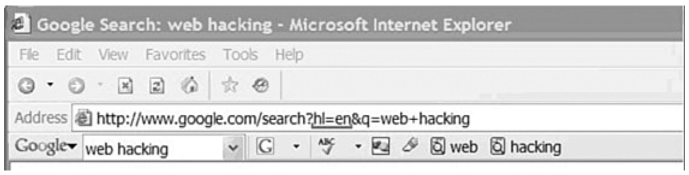
For example, if a parameter looks like it's to be used to select an item from a database (that is, the URL looks something like http://www.companytotest.com?item=1234), try changing the value and seeing what happens.[1]

This effectively asks the database for a different record than the one originally requested. Perhaps this is not a severe security risk in most circumstances, but imagine if the request was for a patient's record in a health-care provider's online system. You've just breached the patient's privacy in the worst sort of way. This is exactly the situation we are trying to prevent, so apply this attack in a creative way, and make sure these bugs are reported and fixed before your site goes live.

It helps to consider the common uses of CGI parameters, so let's spend some time talking about them.

CGI parameters are often used to pass user preferences. Take Google, for example. If you look at any Google search, you'll see the `hl` parameter, which specifies what language to "brand" Google, as shown in Figure 4-5.

FIGURE 4-5 User preference parameters.

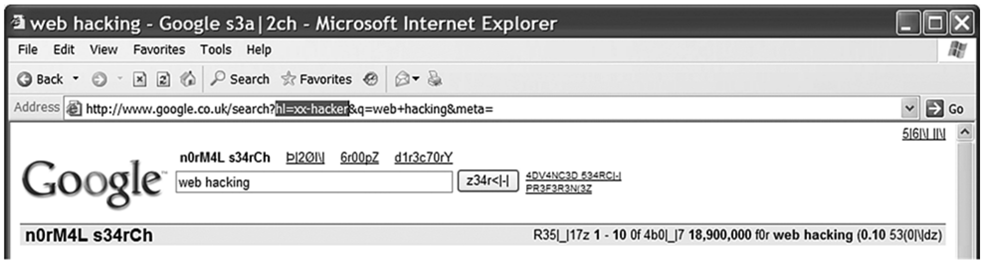


Google Search: web hacking - Microsoft Internet Explorer

Address http://www.google.com/search?hl=en&q=web+hacking

What happens if that parameter is changed, say to `'ru'`? In this case, Google changes its output to Russian. Changing the parameter to `xx-hacker` results in Figure 4-6.

FIGURE 4-6 Modifying user preference parameters.



web hacking - Google s3a|2ch - Microsoft Internet Explorer

Address http://www.google.co.uk/search?hl=xx-hacker&q=web+hacking&meta=

n0rM4L s34rCh

R35|_|17z 1 - 10 0f 4b0|_|7 18,900,000 f0r web hacking (0.10 53(0|\|dz)

1 We might also want to inject SQL statements or script tags to perform SQL injection and cross-site scripting attacks. Those topics will be discussed in later chapters.

Another common use of CGI parameters is to keep track of which pages a user has navigated successfully. For example, some pages might be restricted to users who have been through a registration or authentication process. These parameters often have short names (single characters aren't uncommon) and can carry the values of 1 (true/on) or 0 (false/off). Modifying the value may fool the Web application into believing that the attacker has already registered.

Because Web applications are notoriously difficult to debug (attaching a debugger and single stepping through code isn't easy), some developers add hidden debug parameters to their application. When these parameters are present, the developers send additional output to the browser, often giving a trace of internal application details such as database connections, SQL queries, and variable states.

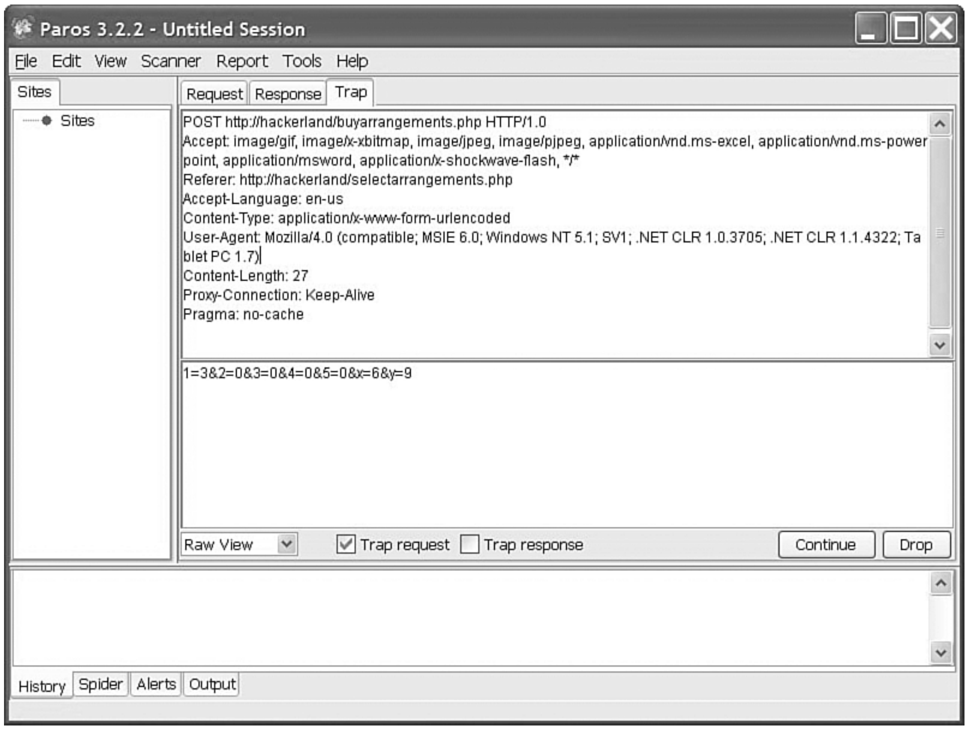
In normal use, these parameters aren't present, so the end user is none the wiser. Adding `&debug=on`, `&debug=1`, or `&debug=true` to the end of the list of CGI parameters (order of parameters generally isn't important, but commonly debug parameters are appended after existing ones) is a simple test to see if the developer has added this debug functionality. However, it's much easier to look at the application code to see if there are `if (debug)…` statements. Say that instead of using simple Boolean values, the developer uses a "magic" number, like 3141592654, to turn debug mode on. Using manual, black-box testing, you may never discover this number—looking at the source is much easier.

So far, we've talked about CGI parameters passed in the browser's address bar, which are known as `GET` parameters. We also mentioned `POST` parameters, which you'll be learning more about in the upcoming sidebar titled "The Difference Between `GET` and `POST` Parameters." `POST` parameters are not as obvious to the end user, or as easy to change, and are passed to the Web server in a slightly different way than `GET` parameters. This means that we cannot as easily modify them using techniques we have introduced thus far; we must use something to help us. Enter Paros Proxy 0, the authors' favorite Web testing tool.

Paros is described more fully in Appendix C, "Tools," but it allows you to see and modify all HTTP traffic to and from the Web server.

Numerous types of data are passed using CGI parameters. CGI is one of the *only* mechanisms of passing data to subsequently loaded pages. Therefore, a comprehensive list of attacks is impossible, and testers need to carefully consider how each parameter may be misused. CGI parameters are the delivery vector for most other attacks (cross-site scripting, SQL injection, directory traversal, and so on) that we will be discussing. That's why knowing what parameters there are and how to change them is important.

FIGURE 4-7 Paros proxy.


Paros 3.2.2 - Untitled Session
File Edit View Scanner Report Tools Help
Sites
Sites
Request Response Trap
Raw View
Trap request
Trap response
Continue
Drop
History Spider Alerts Output

```
POST http://hackerland/buyarrangements.php HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/vnd.ms-excel, application/vnd.ms-power
point, application/msword, application/x-shockwave-flash, */*
Referer: http://hackerland/selectarrangements.php
Accept-Language: en-us
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.0.3705; .NET CLR 1.1.4322; Ta
blet PC 1.7)
Content-Length: 27
Proxy-Connection: Keep-Alive
Pragma: no-cache
```

```
1=3&2=0&3=0&4=0&5=0&x=6&y=9
```

HOW to Protect Against This Attack

Perhaps the best advice to defend against this attack and many other attacks that originate on the client machine is to parse all input for validity. (You may want to refer to the sidebar "Validating Input" in the previous chapter for an in-depth discussion.)

The Difference Between `GET` and `POST` Parameters

Generally, the parameters you'll see passed to a Web server are `GET` parameters—those you can see on the address bar. However, there's another method of passing parameters known as `POST`. Unless client-side code (JavaScript, applets, and so on) generates `POST` requests, these requests are only sent via forms. (If you look at the `<form>` tag, you'll often see an `action="post"` attribute.) But before we go into the difference between the two parameter-passing mechanisms, let's address why there are two ways to accomplish the same thing.

The HTML specification gives the usage advice that `GET` requests are **`idempotent`** operations (basically just receiving information—thus "`GET`"), and `POST` operations should be for anything else that may involve some state change in the application, such as updating a database, sending e-mail, or ordering a product. There might be other reasons for using `POST` over `GET`, such as when sending large quantities of data, because some Web servers do not like receiving more than 8KB in the URL, but 1KB is a more realistic limit. However, the reason for this distinction is that the browser should not resend a `POST` request (for example, if the user clicks the Back button, resubmits a form, or reloads a page) without informing the user first. Just imagine that you're ordering a product, there's a delay, and you click the Order button once more—have you just submitted two orders or one? Some other significant differences exist, but we discuss them in later attacks.

There's also a technical difference between `GET` and `POST` values. Whereas `GET` parameters are passed with the URL, `POST` parameters are sent as part of the body of the request (that is, not in the HTTP Headers section—see Figure 4-7). Also, the byte count of the parameters plus all data is calculated and passed in the `content-length` HTTP header. Although most Web servers are lenient about mismatches between the specified size and actual size of `POST` parameters, lazy attackers don't update the `content-length`. That's why this is sometimes a good way to determine if the request has been significantly tampered with.

ATTACK 8 Cookie Poisoning

Cookies are small files of textual data that a Web application writes on a client's hard drive. The Web application can then reuse this data on subsequent visits to the site from that same computer. This allows the Web site to remember a visitor and offer him customized or personalized content based on the information stored in the cookie.

When people talk about **cookie poisoning**, it's mostly in the context of session hijacking (another attack described later in this chapter). However, there's much more to cookies than just session identifiers.

Cookies are delivered in four forms that are the combination of two settings: persistent or nonpersistent, and secure or nonsecure. The browser places persistent cookies on the client hard disk until their expiry date. In contrast, the browser destroys nonpersistent cookies (which are stored only in memory) as soon as it closes. The secure setting for a cookie, though, is a bit misleading. The cookie itself is not secured or encrypted in any way, but it is a directive to the browser to send this cookie *only* over secure transport, which is HTTP over SSL (HTTPS).

Although the data within a cookie is an obvious place to attack, cookies also have the ability to expire after a specified date. This functionality often ensures that users reidentify themselves after a period of time or sets some time limit on accessing a resource. For example, a credit report might be valid for only 30 days.

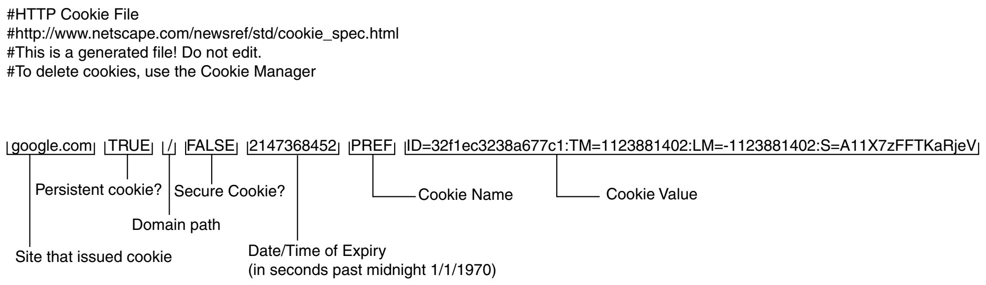
WHEN to Apply This Attack

Like it or loathe it, users are deluged with cookies whenever they use the Web. You can set up all browsers to warn users when a cookie is written to their hard drive, but software like CookiePal (http://www.kburra.com/cpal.html) or CookieCrusher (http://www.thelimitsoft.com/cookie/) gives users more fine-grain control over what cookies they accept or reject and how they view the cookies they have on their computer. Firefox has a lot of this functionality built in.

HOW to Perform This Attack

Cookies are stored in predefined locations, with predefined formats, so modifying their data manually is easy. In Firefox/Netscape, cookies are stored in a `cookies.txt` file with a format shown in Figure 4-8.

FIGURE 4-8 Netscape cookie format.



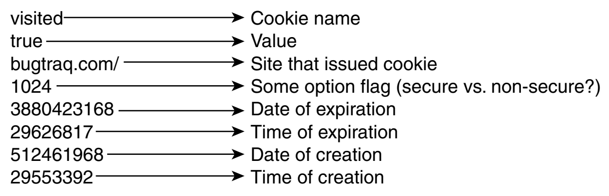
```
#HTTP Cookie File
#http://www.netscape.com/newsref/std/cookie_spec.html
#This is a generated file! Do not edit.
#To delete cookies, use the Cookie Manager

google.com  TRUE  /  FALSE  2147368452  PREF  ID=32f1ec3238a677c1:TM=1123881402:LM=-1123881402:S=A11X7zFFTKaRjeV
```

Internet Explorer stores its cookies in c:/documents and setting/%USERNAME%/cookies/ as individual text files in a format that needs some explanation.

Each text file in the `cookies` directory is formatted as `username@sitename[1].txt`. Therefore, if Joe visited Amazon.com, all his cookies for that site would be stored in the file `joe@amazon[1].txt`, and on rare occasions, the file would have the `[2]` postfix. Cookies inside the file are separated by `*` on a single line, with the cookies formatted as shown in Figure 4-9.

FIGURE 4-9 Internet Explorer cookie format.


visited → Cookie name
true → Value
bugtraq.com/ → Site that issued cookie
1024 → Some option flag (secure vs. non-secure?)
3880423168 → Date of expiration
29626817 → Time of expiration
512461968 → Date of creation
29553392 → Time of creation

What's interesting about this cookie format is not the name, value, or domain attributes, but the way the time and dates are stored.

Rather than storing the creation and expiry timestamps as the number of seconds past midnight January 1, 1970 (the most common format), Internet Explorer uses increments of 100 nanoseconds (10^{-7} seconds) since January 1, 1601. Why Microsoft had to use such a fine-grained scale or go back as far as the 1600s is beyond us, but it fits nicely into a 64-bit number. When you're saving cookies, however, this 64-bit value is broken into two sections: time and date. Although the numbers seem difficult to interpret, it's possible to deduce the date and time from them. The bottom number of the pair is the most significant because it shows time and date in units of 429.4967296 seconds since January 1, 1601. The top number shows the time since the last unit of 429.4967296 seconds has passed, in units of 10^{-7} seconds.

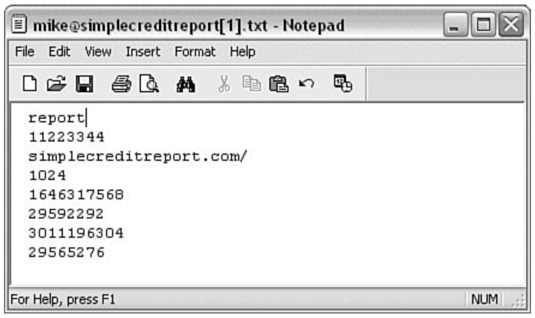
For example, suppose that we check our credit rating with a fictitious site, Simplecreditrating.com. We are given our credit rating report online, but it expires in 30 days. Simplecreditrating.com enforces this policy by issuing a cookie with the report ID that expires in a month. We can find the cookie here:

```
c:/documents and settings/mike/cookies/mike@
 simplecreditreport[1].txt
```

Now we can open the cookie in WordPad, as shown in Figure 4-10.

If we change the `29592292` value to `29598326`, we can access the report for an extra 30 days. The designer of this Web site probably didn't intend for us to do that.

FIGURE 4-10 Cookie for a sample credit report application.


mike@simplecreditreport[1].txt - Notepad
File Edit View Insert Format Help
For Help, press F1
NUM

```
report
11223344
simplecreditreport.com/
1024
1646317568
29592292
3011196304
29565276
```

It's not only the expiry timestamp of a cookie that we can change. We can also change the value part of the cookie. We can change the report reference number, `11223344`, to another value in an attempt to read someone else's credit report.

Some Web applications have a "remember me" functionality, where return visitors are automatically logged in or presented with custom content. Because cookies are the only way to store state information on the client across sessions, this is the obvious place to look to try to break this kind of functionality. Viewing cookies when this functionality is available can reveal usernames and passwords, or "magic" identifiers that are supplied to the Web server in lieu of a user having to authenticate. All of these are attractive targets for attackers.

In Attack 11, we'll look at a related method whereby an attacker can steal cookies.

HOW to Protect Against This Attack

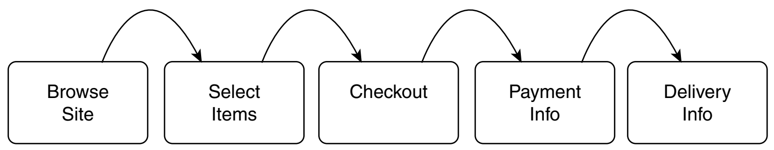
Designers of the Web never intended cookies to be secure. Cookies were to be an extension to HTTP that gave it some aspect of client-side state. However, because cookie files are the *only* way to store state information across browser sessions, Web designers have used them considerably and will likely continue to use them.

If your Web application is relying on cookies to enforce expiration or you really have to store sensitive data on the client, consider encrypting the cookie. And don't rely on the cookie's own expiry date, because that's easy to tamper with.

ATTACK 9 URL Jumping

Because the Web is inherently stateless, users can jump to any page they want to by typing the Universal Resource Locator (URL) in the browser's address bar and pressing Enter. Developers of Web applications often don't want to allow users this level of freedom because they might have a sequence of operations that users have to follow, as depicted in Figure 4-11.

FIGURE 4-11 Common flow of functionality in an e-commerce application.



If the user were allowed to jump directly from the Checkout page to the Delivery Info page, he may be able to receive his goods without paying for them. This is only one such example. There are many occasions in a system where one operation has to take place before another (for example, logging in before reading an e-mail message, or selecting a group before posting a forum message). The purpose of this attack is to identify actions in a Web application that should be sequenced and attempt to jump into, around, or over certain steps by browsing directly to them.

WHEN to Apply This Attack

This attack often requires some understanding of the Web application and exactly what it implements. You may want to go back to the page map we developed in Chapter 2, "Gathering Information on the Target," and think about sequences of pages or operations and the implications of jumping from page to page without clicking the links that the application provides.

Begin by browsing the application as a legitimate, well-behaved user, and note the addresses of pages visited along with their sequence. Using this list, randomly enter addresses and see if the application produces meaningful error messages or disallows access to specific pages.

HOW to Perform This Attack

For a poorly developed Web application, this attack is a task of reconnaissance followed by entering page addresses into the browser's address bar. However, good Web developers understand the problem of users breaking out of page sequences. As one means of protection against this attack, these developers may compare the last visited page against the one a user *should* have come from.

Developers can achieve this protection technique with any of the following methods:

- Using hidden fields or CGI parameters to store a page address or identifier
- Using cookies to store last visited pages or identifiers
- Comparing where the user should have come from with the HTTP-REFERER field

The first method, using hidden fields or CGI parameters with page addresses, is the most insecure method because it is subject to the attacks described earlier. It really only stops unsophisticated attackers; nonetheless, developers use the technique because of the simplicity of including the hidden data at design time. It's relatively easy to change the field's value or even to add a required hidden field where necessary (in either the HTML source or by capturing the page request using a proxy).

The second method, using cookies to store the last visited page, is slightly more secure because cookies (especially temporary ones[2]) are harder to modify as they are passed in the HTTP header—a place that users can't control through the browser.

FIGURE 4-12 Request for a page. Note the referer header.

```
GET /articles/news/today.asp HTTP/1.1
Accept: */*
Accept-Language: en-us
Connection: Keep-Alive
Host: localhost
Referer: http://www.myhomepage.com/links.asp
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT
 5.0)
Accept-Encoding: gzip, deflate
```

[2] Temporary cookies are ones that expire when the browser closes. Generally, they are only stored in memory, not on the hard drive where they are easier to locate and edit. But don't get the idea that temporary cookies are secure. They aren't. However, they do require a more sophisticated attacker who has more advanced debugging tools.

FIGURE 4-13 The associated response. Note the server setting a cookie value.

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Thu, 13 Jul 2000 05:46:53 GMT
Content-Length: 2291
Content-Type: text/html
Set-Cookie: chocolatechip=DR2EO53DNSK2EMM5K2LSLJ5NEKE;
 path=/
Cache-control: private

<HTML>
[…html markup follows…]
```

Also note that in the HTTP request data, the referer field carries the address of the page that initiated the request and may be used instead of setting an explicit cookie.

In fact, it's pretty easy to change the HTTP header. In modifying the referer header and the cookie, you can use proxy tools such as Paros to change the cookie's or the referer's value. You can also perform page requests manually, as we will show in future attacks.

Regardless of which method the Web developer chooses to implement or how you decide to attack it, the principle is the same: Request a page that a user should not be able to jump directly to, and see if you can view it. If not, modify the values of hidden fields, cookies, or the referer to try to force it the hard way. If you see the page, you have a potential attack scenario and a bug report to write.

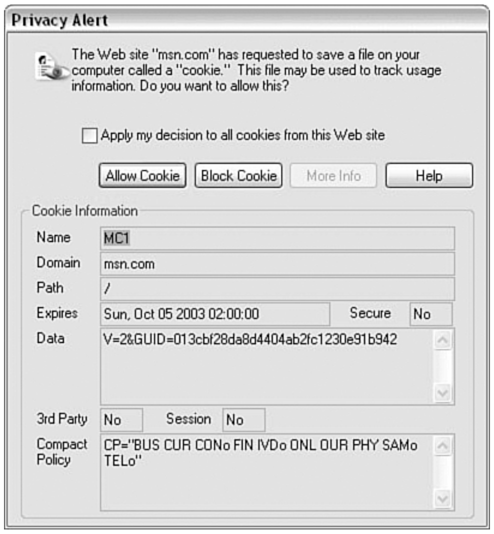
HOW to Protect Against This Attack

There is no other way to protect against this attack except by restricting the sequence in which you can view pages. This obviously requires storing the last visited page, but as mentioned earlier, you can store this information in numerous places, some of which an attacker can access.

The most secure place for the last visited page to be stored is on the server, because users only have control over information on the client machine and the information that the browser sends over the network. Many Web application servers can store variables on the server (ColdFusion, Java Servlets, ASP, PHP, and so on), but this requires the use of session variables and opens up the possibility of session hijacking attacks, covered later in this chapter.

If there is one preferred method of storing the last visited page (without server-side support), it would be in the HTTP-REFERER field. That field is not more secure than the others, but when a Web application sends cookies to the user's browser, it's a signal that something interesting is being stored,[3] as shown in Figure 4-14.

FIGURE 4-14 Cookie warning in Internet Explorer.


Privacy Alert

The Web site "msn.com" has requested to save a file on your computer called a "cookie." This file may be used to track usage information. Do you want to allow this?

☐ Apply my decision to all cookies from this Web site

Cookie Information

Name	MC1
Domain	msn.com
Path	/
Expires	Sun, Oct 05 2003 02:00:00
Secure	No
Data	V=2&GUID=013cbf28da8d4404ab2fc1230e91b942
3rd Party	No
Session	No
Compact Policy	CP="BUS CUR CONo FIN IVDo ONL OUR PHY SAMo TELo"

Utilizing the HTTP-REFERER is less likely to alert an attacker to its use because it is sent with every page request. Some proxies, however, may strip this information as a privacy precaution, so applications may not be able to rely on it. Manually typing a URL in the address bar also prevents it from being sent.

To protect against the risk of users tampering with data that has to be stored on the client, consider encrypting the data with a well-known standard and restrict storing the encryption/decryption keys on the server. (Be extremely suspicious of roll-your-own cryptography. We talk about attacking crypto in Chapter 8, "Authentication.")

[3] By default, Internet Explorer is set to allow all cookies. To change this functionality, go to Tools, Internet Options. Click on the Privacy tab, and then select Advanced.

ATTACK 10 Session Hijacking

Of all the state-based attacks that have been discussed thus far, session hijacking has the most exposure in the Web development literature. The reason for this is simple: You can use session management to solve a lot of the problems of storing state in a Web application. The issue is that if you do it incorrectly, it is open to attack.

Session management works by each user having a unique identifier that travels with him during his use of a Web site. This generally occurs with the server issuing a number to each new user on the initial home page of the site. All further requests would include this identifier so that Web applications can distinctly identify users and store their associated state information on the server.

You can use several methods to break session management by swapping the session identifier of one user with the session identifier of another user. The methods are as follows:

- Modifying data randomly, hoping to become another user
- Figuring out the sequence of unique identifiers that the site uses
- "Fixing" the session identifier of another user

Session identifiers are presented to the server as hidden fields, appended to URLs, or stored in cookies. Storing the session identifier in a cookie and then passing it to the server as each page is loaded is the most common. Session hijacking is the culmination of all the attacks that have been presented in this section.

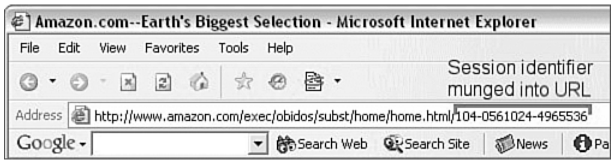
WHEN to Apply This Attack

The most obvious way of identifying when to apply this attack is when you see a cookie sent to the browser. The cookie must contain some session identifier. A recent survey of Web sites showed that the following are the most common names for session cookies:

- `ASPSESSIONID`
- `JSESSIONID`
- `PHPSESSID`
- `CFID`
- `CFTOKEN`

However, treat with suspicion any cookie that uses the moniker "ID" or looks like a unique number. When cookies are unavailable (some users disable them), you can append an identifier as a CGI parameter or insert it (munge it) into the page URL as Amazon.com does.

FIGURE 4-15 Session ID munging when cookies are not available.


Amazon.com--Earth's Biggest Selection - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Session identifier munged into URL
Address
Google
Search Web
Search Site
News

http://www.amazon.com/exec/obidos/subst/home/home.html/104-0561024-4965536

HOW to Perform This Attack

The attacker's objective in session hijacking is to masquerade as another legitimate user by using that person's identifying credentials—the session identifier. The most common way of achieving this is to steal that user's session identifier by various means. (The cross-site scripting attack is often associated with this goal, although monitoring network traffic is another avenue.) However, as we shall see later, it is also possible to "give" a user a compromised session.

Poorly implemented session handlers open the door to guessing previous or future session identifiers. The most obvious is where IDs are allocated sequentially, so the next person to visit the application will get the n+1 (or some other identifiable pattern) value. Therefore, we should first try to gather a number of session identifiers and see if we can find a pattern that will allow us to predict what identifiers a Web site will use for future and past visitors.

If we know or can figure out a session identifier, we can replace the value of the session variable (hidden field, CGI parameter, or cookie) with another valid one and then request a page again. However, with or without this knowledge, it may be necessary to try the attack several times as legitimate users log into and out of the Web site.

Some Web applications may provide helpful error messages when an invalid session is requested, which helps with this attack, but the main clue that the attack is successful is when personalized information of another user appears, as in Figure 4-16.

Another form of attacking session management is called **session fixation**. It is subtly different from session hijacking because hijacking suggests that there is something *in-place* to take. Session fixation occurs when the session ID is stolen before a legitimate user ever gets it. The attacker can then take the session from the legitimate user any time it is advantageous to do so.

FIGURE 4-16 Personalized information in a Web application.

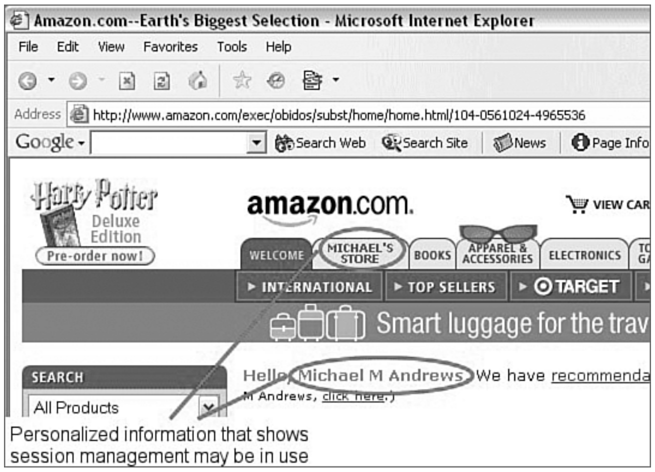
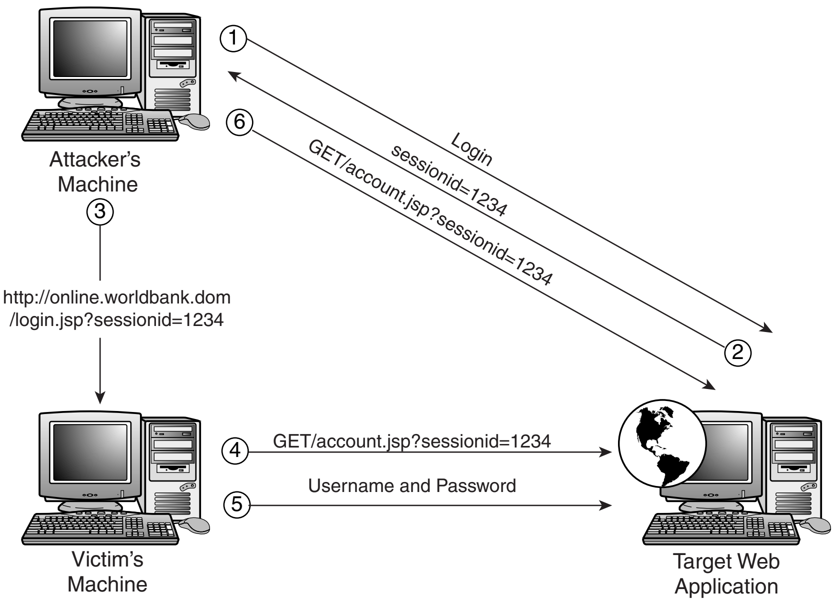

Amazon.com--Earth's Biggest Selection - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.amazon.com/exec/obidos/subst/home/home.html/104-0561024-4965536
amazon.com.
WELCOME
MICHAEL'S STORE
BOOKS
APPAREL & ACCESSORIES
ELECTRONICS
INTERNATIONAL
TOP SELLERS
TARGET
Smart luggage for the trav
SEARCH
All Products
Hello, Michael M Andrews. We have recommenda
M Andrews, click here.)
Personalized information that shows session management may be in use

FIGURE 4-17 Setting up a session fixation attack.


Attacker's Machine
1
Login
2
sessionid=1234
6
GET/account.jsp?sessionid=1234
3
http://online.worldbank.dom /login.jsp?sessionid=1234
4
GET/account.jsp?sessionid=1234
5
Username and Password
Victim's Machine
Target Web Application

As Figure 4-17 shows, this attack works by an attacker either generating a compromised session, or, depending on the session management mechanism, providing a link to the Web site with the session identifier already provided in the URL. The unsuspecting user logs in or clicks on the link and uses the Web application. Because the session is a valid one, the legitimate user doesn't notice a difference. However, the critical problem is that the attacker now knows the legitimate user's session identifier and can assume his identity.

The most probable targets for session fixation attacks are Webmail-type systems. This is because taking over a user's shopping cart doesn't really achieve anything (an attacker is unlikely to want to pay for someone else's goods!), but being able to read or send another user's e-mail after he has finished is a conceivable attack.

HOW to Protect Against This Attack

Session management is a necessity of Web applications, and if done correctly, it can be an effective protection mechanism against a number of attacks, including session hijacking. That's why it's typical for Web developers to utilize sessions, despite their security implications. Here's some advice about doing it right.

Protection of a session needs to focus on the unique session identifier because it is the only thing that distinguishes users. If the session ID is compromised, attackers can impersonate other users on the system.

The first thing is to ensure that the sequence of identification numbers issued by the session management system is unpredictable; otherwise, it's trivial to hijack another user's session. Having a large number of possible session IDs (meaning that they should be very long) means that there are a lot more permutations for an attacker to try.

Developers also need to pay attention to the random qualities (those that are nonsequential and hard to guess) of chosen individual IDs so that an attacker cannot easily determine the algorithm used to generate the session IDs.

Taking care to generate good session IDs is just the beginning. After you've generated the ID, you must protect it, which is a concept called **session management**. Good session management consists of the following:

- Using cookies for storing session values.

  Cookies are generally more difficult to modify than hidden fields or CGI parameters. You can protect them by using mechanisms like setting the secure flag (so they cannot be "sniffed" unencrypted on the network). In addition, you can restrict cookies to a particular site or even a section of a site (using the path attribute of the cookie[4]), or set them to expire automatically.
- Not allowing users to choose their own session identifiers.

  Some session management systems allow users to reactivate their session if they have a valid session ID but it has been expired. There is no

[4] The path attribute is not a completely trustworthy mechanism. It's just one more tool in a Web developer's arsenal.

good reason why an existing session should be reactivated because a new session can be created with a different session identifier but the same stored state. If an attacker discovers that session identifiers are being reused, he can gather a number of valid ones and have an immediate advantage in a session fixation attack.

- Ensuring that each user gets a "clean" session identifier number with each visit and revisits to your site.

  Users should get a new session number each time they visit your site, because that makes the attacker's job of giving them a compromised ID irrelevant. You can check this by comparing the referring page against the URL of the site. If they are different, you should create a new session identifier. However, a downside to this is that it might break the "remember me" and "single-click shopping" that some e-commerce sites use.

- Time-out session identifiers so someone cannot reuse them after a predetermined period of time.

  Storing session variables on the server allows the Web application to keep track of what sessions have been created and when. If no one has used a session for a specified period (based on user activity or a predefined time), you should expire it. This gives the attacker a smaller window of opportunity to guess (or brute force) valid session identifiers.

- Allowing users to log out and clear their session.

  When a user logs out, this action should invalidate identification numbers from both the client and the server. Not only should it clear the current sessions, but it should clear all other sessions that the users may have initiated but have failed to log out of because of forgetfulness (browsing away from the site) or some other issue like server failure.

- Utilizing the HTTP referer field to identify multiple clients browsing with the same ID.

  If the Web application can "track" users through the site and has clear paths of browsing that users follow, it's possible to discover situations where two or more people are using the same identifier. The basic idea is to know the correct page sequence of the site. If a request for a page that should not be accessible is received, then either a URL-jumping attack is in progress, or another user is using the same session identifier and is out of step with the original user. In both situations, the session identifier should be invalidated.

- Ensuring that session cookies are sent only over secure channels to prevent them from being captured in transit.

  You wouldn't want credit card numbers being sent in clear text across the network, and because session identifiers are indirect references to users' information, you should protect them equally. Because cookies are sent with every request matching a specified domain and path, it's easy for them to be inadvertently sent over a nonencrypted channel where an attacker may be listening. Therefore, you should set the secure flag for all session identifier cookies to ensure that they are sent only over HTTPS.

Even with these precautions, there's the possibility of an attacker discovering a current session ID by "stealing" a cookie through cross-site scripting, so protecting against that attack is a crucial facet of protecting against this one. Cross-site scripting is a topic for another chapter.

References

http://www.parosproxy.org

http://www.securityspace.com/s_survey/data/man.200507/cookieReport.html

http://www.dutchduck.com/help/cookies explorer/faq/

http://www.acros.si/papers/session_fixation.pdf