

Penetration Testing for Web Applications (Part Three)

Jody Melbourne and David Jorm 2003-08-20

In the [first installment](#) of this series we introduced the reader to web application security issues and stressed the significance of input validation. In the [second installment](#), several categories of web application vulnerabilities were discussed and methods for locating these vulnerabilities were outlined. In this third and final article we will be investigating session security issues and cookies, buffer overflows and logic flaws, and providing links to further resources for the web application penetration tester.

Cookies

Cookies are a mechanism for maintaining persistent data on the client side of a HTTP exchange. They are not part of the HTTP specification, but are a de-facto industry standard based on a specification issued by Netscape. Cookies involve the use of HTTP header responses to set values on the client side, and in client requests to provide these values back to the server side. The value is set using a 'Set-Cookie' header and returned using a 'Cookie' header. Take the following example of an exchange of cookies. The client requests a resource, and receives in the headers of the response:

```
Set-Cookie: PASSWORD=g0d; path=/; expires=Friday, 20-Jul-03 23:23:23 GMT
```

When the client requests a resource in path "/" on this server, it sends:

```
Cookie: PASSWORD=g0d
```

The browser is responsible for storing and retrieving cookie values. In both Netscape and Internet Explorer this is done using small temporary files; the security of these mechanisms is beyond the scope of this article, we are more concerned with the problems with cookies themselves.

Cookies are often used to authenticate users to an application. If the user's cookie is stolen or captured, an attacker can impersonate that user. There have been numerous browser vulnerabilities in the past that allow attackers to steal known cookies -- for more information on client-side cookie security, please refer to the cross-site scripting section in [part two](#) of this series.

Cookies should be treated by the developer as another form of user input and be subjected to the same validation routines. There have been numerous examples in the past of SQL injection and other vulnerabilities that are exploitable through manipulating cookie values. Refer to the [PHPNuke admin cookie SQL injection](#), and [Webware WebKit cookie overflow](#) vulnerabilities.

Session Security and Session-IDs

Most modern web scripting languages include mechanisms to maintain session state. That is, the ability to establish variables such as access rights and localization settings which will apply to every interaction a user has with the web application until they terminate their session. This is achieved by the web server issuing a pseudo-unique string to the client known as a Session ID. Then the server associates elements of data with this ID, and the client provides the ID with each subsequent request made to the application. Both PHP and ASP have in-built support for sessions, with PHP providing them via GET variables and Cookies, and ASP via Cookies only.

PHP's support for GET variable sessions is considered by all accounts an inferior mechanism, but is provided because not all browsers support cookies and not all users will accept cookies. Using this method, the Session ID is passed via a GET variable named PHPSESSID, provided in the query string of every request made. PHP automatically modifies all links at runtime to add the PHPSESSID to the link URL, thereby persisting state. Not only is this vulnerable to replay attacks (since the Session ID forms part of the URL), it trivializes it -- searching proxy logs, viewing browser histories or social engineering a user to paste you a URL as they see it (containing their Session ID) are all common methods of attack. Combine GET variable sessions with a cross site scripting bug and you have a simple way of forcing the disclosure of the Session ID. This is achieved by injecting javascript code which will post the document URL to a remote logging application, allowing the attacker to simply watch his logging application for the Session IDs to roll in.

The cookie method works in a similar manner, except the PHPSESSID (or Session-ID in the case of ASP) variable is persisted using a cookie rather than a GET variable. At a protocol level, this is just as dangerous as the GET method, as the Session ID can be logged, replayed or socially engineered. It is, however, obfuscated and more difficult to abuse as the Session ID is not embedded in the URL. The combination of cookies, sessions and a cross site scripting bug is just as dangerous, as the attacker need only post the document.cookie property to his logging application to extract the Session ID. Additionally, as a matter of convenience for the user, Session IDs are frequently set using cookies with either no expiry or a virtually infinite expiry

date, such as a year from the present. This means that the cookie will always exist at the client side, and the window of opportunity will be indefinitely replayable as the cookie has no expiry date.

There are also many, albeit less common, forms of user session management. One technique is to embed the Session ID string in an `<input type="hidden">` tag with a `<form>` element. Another is to use Session ID strings provided by the Apache webserver for user tracking purposes and as authentication tokens. The Apache project never intended these to be used for anything other than user tracking and statistical purposes and the algorithm is based on concatenation of known data elements on the server side. The details of Session ID bruteforcing and cryptographic hashing algorithms are beyond the scope of this article, but David Endler has provided a [good paper on this topic](#) (.pdf) if you are interested in reading more.

Session IDs are very much an Achilles' Heel of web applications, as they are simply tack-ons to maintain state for HTTP -- an essentially stateless technology. The penetration tester should examine in detail the mechanism used to generate Session IDs, how the IDs are being persisted and how this can be combined with client-side bugs (such as cross site scripting) to facilitate replay attacks.

Logic Flaws

Logic flaws are a broad category of vulnerability encompassing most bugs which do not explicitly fall into another category. A logic flaw is a failure in the web application's logic to correctly perform conditional branching or apply security. For example, take the following snippet of PHP code:

```
<?php
$a=false;
$b=true;
$c=false;
if ($b && $c || $a && $c || $b)
    echo "True";
else
    echo "False";
?>
```

The above code is attempting to ensure that two out of the three variables are set before returning true. The logic flaw exists in that, given the operator precedence present in PHP, simply having \$b equal to true will cause the if statement to succeed. This can be patched by replacing the if statement with either of the following:

```
if ($b && $c || $a && ($c || $b))  
if ($b && $c || $a && $c || $a && $b)
```

Logic flaws are difficult to identify from a blackbox testing perspective, and they more commonly make themselves apparent as a result of testing for another kind of vulnerability. A comprehensive code audit where the conditional branching logic is reviewed for adherence to program specification is the most effective way to trap logic flaws. An example of a logic flaw issue is the [SudBox Boutique login bypass vulnerability](#).

Binary Attacks

Web applications developed in a language that employs static buffers (such as C/C++) may be vulnerable to traditional binary attacks such as format string bugs and buffer overflows. Although code and content manipulation issues (such as SQL and PHP code injection) are more common, there have been numerous cases in the past of popular web applications with overflow vulnerabilities.

A buffer overflow occurs when a program attempts to store more data in a static buffer than intended. The additional data overwrites and corrupts adjacent blocks of memory, and can allow an attacker to take control of the flow of execution and inject arbitrary instructions. Overflow vulnerabilities are more commonly found in applications developed in the C/C++ language; newer languages such as C# provide additional stack protection for the careless developer. Recent examples of overflows in web applications include [mnoGoSearch](#) and [Oracle E-Business Suite](#).

Buffer overflows can often be located through black-box testing by feeding increasingly larger values into form inputs, header and cookie fields. In the case of ISAPI applications, a 500 error message (or time-out) in response to a large input may indicate a segmentation fault at the server side. The environment should first be fingerprinted to determine if the development language is prone to overflow attacks as overflows are more common to compiled executables than scripted applications. Note that most of the popular web development languages (Java,

PHP, Perl, Python) are interpreted languages in which the interpreter handles all memory allocation.

Format string attacks occur when certain C functions process inputs containing formatting characters (%). The printf/fprintf/sprintf, syslog() and setproctitle() functions are known to misbehave when dealing with formatting characters. In some cases, format string bugs can lead to an attacker gaining control over the flow of execution of a program. Refer to the [PWC.CGI vulnerability](#) for an example of this type of exploit in a web application.

Useful Testing Tools

A number of applications have been developed to assist the blackbox tester with locating web application vulnerabilities. While analysis of programmatic output is probably best accomplished by hand, a large portion of the blackbox testing methodology can be scripted and automated.

AtStake WebProxy

WebProxy sits between the client browser and the web application, capturing and decoding requests to allow the developer to analyze user interactions, study exploit techniques, and manipulate requests on-the-fly.

Home Page: <http://www.atstake.com/webproxy>

SPIKE Proxy

SPIKE proxy functions as a HTTP/HTTPS proxy and allows the blackbox tester to automate a number of web application vulnerability tests (including SQL injection, directory traversal and brute force attacks).

Home Page: <http://www.immunitysec.com/spike.html>

WebserverFP

WebserverFP is a HTTPD fingerprinting tool that uses values and formatting within server responses to determine the web server software in use.

Home Page: <http://www.astralclinic.com>

KSES

KSES is a HTML security filter written in PHP. It filters all 'nasty' HTML elements and helps to prevent input validation issues such as XSS and SQL injection attacks.

Home Page: <http://sourceforge.net/projects/kses>

Mieliekoek.pl

This tool, written by roelof@sensepost.com, will crawl through a collection of pages and scripts searching for potential SQL injection issues.

Download: <http://www.securityfocus.com/archive/101/257713>

Sleuth

Sleuth is a commercial application for locating web application security vulnerabilities. It includes intercept proxy and web-spider features.

Home Page: <http://www.sandsprite.com/Sleuth>

Webgoat

The OWASP Webgoat project aims to create an interactive learning environment for web application security. It teaches developers, using practical exercises, the most common web application security and design flaws. It is written in Java and installers are available for both *nix and Win32 systems.

Home Page: <http://www.owasp.org/development/webgoat>

AppScan

AppScan is a commercial web application security testing tool developed by Sanctum Inc. It includes features such as code sanitation, offline analysis, and automated scan scheduling.

Home Page: <http://www.sanctuminc.com/solutions/appscan/index.html>

Conclusion

Web applications are becoming the standard for client-server communications over the Internet. As more and more applications are 'web enabled', the number of web application security issues will increase; traditional local system vulnerabilities, such as directory traversals, overflows and race conditions, are opened up to new vectors of attack. The responsibility for the security of sensitive systems will rest increasingly with the web developer, rather than the vendor or system administrator.

In this series of articles we hope to have stressed the importance of user input validation and have demonstrated how all major web application security issues relate back to this concept. The best defense against input-manipulation attacks is to treat all input with a healthy dose of paranoia and the notion of "if not explicitly allowed, deny." Dealing with user complaints about non-permitted characters is always going to be less painful than a security incident stemming from unfiltered input.

Author Credit

View [more articles by Jody Melbourne and David Jorm](#) on SecurityFocus.

[Privacy Statement](#)

Copyright 2006, SecurityFocus