Table of Contents

Writing Secure Web Applications

http://advosys.ca/tips/web−security.html
Apr 13 2001

Contents:

Web application development is very different from other environments. Web clients and Internet communications pose many security problems not found in traditional client−server applications.

Web developers must know how web servers and web browsers interact, the nature of Internet communications, and the attacks most web applications undergo when they are made available on the Internet.

If you think your servers are secured by a firewall, think again. Security flaws in web applications easily bypass firewalls and other basic security measures. It's easy to unknowingly write a web application that allows outsiders access to files on the server, gather passwords and customer information, and even alter the application itself despite firewalls and other security you may have implemented.

This document presents common web application security problems. The examples used relate to Unix, Perl and the CGI.pm library running under Unix, arguably the most widely deployed language for web apps. Regardless, The flaws and concepts described apply to Java servlets, Cold Fusion, PHP, or any development language and environment.

Sanitize browser input:

All input from web browsers, such as user data from HTML forms and cookies, should be stripped of special characters and HTML tags as described in the following CERT advisories:

CERT advisory 97.25.CGI
http://www.cert.org/advisories/CA−1997−25.html
CERT advisory CA−2000−02
http://www.cert.org/advisories/CA−2000−02.html

There are two separate dangers with browser input data:

1. Input containing special characters such as ! and & could cause the web server to execute an operating system command or have other unexpected behaviour.

2. User input stored on the server, such as comments posted to a web discussion program, could contain malicious HTML tags and scripting code. When another user views the input, the code can be executed by the user's web browser.

Special characters in form input

Characters such as & > ! $ (sometimes called "meta characters") have special meaning to many operating systems. For example, both Unix and NT interpret the < (less than) symbol as meaning "read input from file".

When raw HTML form input is is used to open files, send mail or interact with the operating system in any way, malicious users can enter meta characters hoping they will be passed on to the underlying operating system to be interpreted, and possibly access files or execute commands.

Invisible characters are also a potential threat... especially the NULL character (ASCII zero). Programs written in C use the NULL character to mark the end of strings. However, languages like Perl do not. Inserting a NULL character in HTML form input can cause strings to be terminated early or be unrecognized by simple pattern matching filters. This "poisoned NULL" attack can cause interesting security problems when Web applications are built using both C and other languages. See Phrack magazine Vol 9 Issue 55 http://www.insecure.org/news/P55−07.txt for an excellent description of the problem.

Malicious stored HTML

Some web guestbooks and discussion boards allow users to format their comments with HTML tags such as <B></B> and <FONT SIZE>. This is pretty, but storing HTML can allow users to also insert Javascript, Java and DHTML tags. When another user reads the comments posted, the code can be executed on their browser. Malicious scripts can access and delete files, crash a user's computer or steal information from HTML fields.

Stored HTML tags don't threaten your web server security like metacharacters do. Instead, they threaten the users of your web application. The real danger is one of trust... an outsider is allowed to store potentially dangerous HTML on your server. When another user reads it, they and the security settings of their web browser trust the HTML originates from your organization. A Java applet or insecure Javascript functions that normally would be stopped could be allowed to run in the user's browser because it came from a trusted source.

Solutions

The best practice is to simply strip meta characters, invisible characters and HTML tags from user input.

When stripping nasty characters from user input, the safest way is to check the input against a list of *valid* characters, not a list of invalid ones. Why? It's too difficult to determine all possible invalid characters... just when you think you've thought of them all, a cracker invents an unexpected attack like "poisoned null" characters described above. It's also easier to simply check input against a list of characters A−Z and 0−9.

All input should be sanitized, not just fields used to open files or that are displayed back to other users. Any input can potentially percolate through to unexpected places. Even if you are certain a particular input field can not cause problems now, it may become possible in future revisions of the web application. Rather than try to guess what input could be dangerous, it is simpler and more effective to just sanitize all input immediately when received from the browser.

If your web application is written in Perl using the CGI.pm library, all form input fields can be sanitized at the beginning of the program with a routine similar to this:

```
$ok_chars = 'a-zA-Z0-9 ,-';
foreach $param_name ( param() ) {
    $_ = param($param_name);
    $_ =~ s/[^$ok_chars]//go;
    param($param_name,$_);
}
```

The above silently removes all characters except the ones listed in $ok_chars from all HTML form input fields collected by CGI.pm. This method takes care of shell metacharacters, the poisoned NULL attack and disables HTML tags.

A more thorough solution is raise an alarm instead of sliently stripping input. When invalid characters are detected, it would be better for a sanitize routine to alert a system administrator and log the error and IP of the user to a file. This way you will know when potential crack attempts are made and by who.

User input is not just limited to HTML form fields. HTML cookies and HTTP headers such as REMOTE_USER can also be manipulated by user. It's not difficult to write a Perl script that poses as a web browser and sends hacked versions of form input, cookies and HTTP headers. Never trust any input sent a browser.. all of it can be altered.

The above Perl code only removes characters from form input fields. Input from cookies and HTTP headers should also be sanitized in a simliar way before being used in a web application.

Use a data directory:

If your web applications write data files (such as a guestbook application or message board), don't put the data files in the cgi-bin directory. Any file in cgi-bin can be executed, including data files. Storing data files in /htdocs is not recommended either since any file in htdocs can be read through a web browser.

A good location for data files is a directory located outside both htdocs and cgi-bin. For example, a /data in your home directory with sub-directories for data files of each web application cannot be executed or read from a browser.

Here's an example layout of a web server home directory on Unix. Note separate sub-directories in /data to hold data files from each web app used on the site:

```
/cgi-bin
/htdocs
/data
   /catalog
   /webforum
   /webmail
```

Keeping all data files in a common location also makes it easier to manage the web site. You have separate locations for HTML files, CGI programs, and data. Separating the data files into sub-directories by application helps eliminate file naming problems, such as two different apps that create data files named "data.txt"

Avoid using a shell:

If you program in Perl, use Perl's system() call to run external programs instead of backticks (` `), as mentioned in the WWW security FAQ (http://www.w3.org/Security/Faq/www−security−faq.html).

Backticks in Perl are a convenient way to run external programs, carried over from shell programming. However, backticks cause Perl to spawn a Unix shell (or Windows command interpreter on that platform) then execute the program. Shells have powerful capabilities... by spawning a shell your application gives control to another application that is not as security−conscious as your own application. Stray metacharacters from user input could misdirect the shell into doing something nasty.

When using system(), separate each parameter in the system() call with commas. For example:

```
system("command", "arg1", "arg2");
```

This avoids a shell being spawned. If you string the program name together with the parameters, system() spawns a shell.

Not only is using system() more secure than the backtick method, it's more efficient... by not spawning a shell your application uses fewer system resources and runs faster.

'hidden' fields aren't:

Many CGI programs rely on so−called "hidden" form fields to store state information, settings and previous input data. However, HTML "hidden" fields are not hidden and not secure. Users can see them simply by viewing the HTML source of your form in their browser. It's easy for a user to change "hidden" fields... they only have to same the HTML form to their computer, edit the HTML then re−submit the form.

Contents of hidden fields should be sanitized and validated just like any other user input field. Hidden fields should not be used to set access modes or privileges for a CGI program (such as an 'admin' mode or 'paying user' privilege) without also using some form of user validation, such as password and username access restrictions.

The problem is demonstrated in detail in our paper "Preventing HTML form tampering" (http://advosys.ca/tips/form−tampering.html). A solution, complete with example Perl code is presented.

InfoSec Labs (http://www.infoseclabs.com/) also has a very detailed white paper on hidden form field vulnerabilities. See http://www.infoseclabs.com/mschff/mschff.htm.

If you write web applications in Perl, check out the CGI::EncryptForm module by Peter Marelas (available from CPAN http://www.cpan.org/). It's one way of encrypting hidden form field information to prevent users from changing hidden fields.

A better way of preserving state information and settings is to store data in a file or database on the server then use an HTTP cookie or unique URL ID to reference the file. This is more difficult to program, but important data stays on your server. The Perl module CGI::Persistent by Vipul Ved Prakash is one tool that makes this technique easier to implement.

Don't trust HTTP_REFERER:

Don't rely on the HTTP_REFERER CGI variable for authentication. Many CGI scripts rely on that variable to make sure the script is being called by itself. However, HTTP_REFERER is sent by the user's web browser, not the web server. It can be can be spoofed easily.

In addition, some browsers and users accessing your script from behind proxy servers and firewalls don't send an HTTP_REFERER variable at all.

For example, the privacy protection proxy server Junkbuster and many firewall proxies routinely strip HTTP_REFERER to prevent click tracing.

Use POST instead of GET:

HTML forms can be submitted using either GET or POST methods. POST is preferred, especially when sending sensitive information.

The GET method sends all form input to the web application as part of the URL. For example:

```
http://www.yourdomain.com/cgi-bin/cart.cgi?username=jsmith&password=happypup
```

When the web application is called using GET, the above input is visible on the browser's URL location window. However, a more dangerous problem is that URLs are logged in many places:

- The web server access log
- The web browser's disk cache and history file
- In firewall logs
- In proxy server and web cache logs such as Squid.

All this logging allows others to see the data sent from HTML forms using GET.

The POST method sends form input in a data stream, not part of the URL. The data is not not visible in the browser location window and is not recorded in web server log files.

The POST method is also more practical... there is a limit to how many characters can be sent using the GET method, but POST can send an almost unlimited amount of data from an HTML form.

However, even though POST information is generally not logged, like all other plain text information sent from a browser it can still be sniffed as it passes across the Internet. However, sniffing must be done in real time as information is sent across the Internet and requires the attacker to have physical access to the data lines between the web browser and web server. The risk of information being sniffed is far less than the risk of information being gathered from log files.

Validate on the server

Developers know they must check that form input fields contain the correct data type, that required input is not missing and to preform other simple sanity checks. However, a disturbing trend recently has been to use Javascript or Java applets to validate user input.

In a traditional web application, HTML form fields are checked when the user submits the HTML form to the web server. If a mandatory field is blank or has the wrong type of input, the web application typically sends back an HTML page describing the errors and lets the user must submit the form again.

Client−side validation lets this work be done in by the browser. Javascript or Java applets can be used to check inputs before the form is submitted to the web application, and not allow the form information to be sent until all fields are correct. This saves time and processing by the web application. It moves the overhead of input validation from the web application to the browser.

This technique saves load on the web server, but the problem is that anything running on the client end can never be trusted. Javascript in an HTML form can be changed or disabled by the end user very easily. Java applets can be disabled by the end user, or even decompiled and re−written.

It's easy for a knowledable user to save an HTML form, disable the embedded Javascript, then use the modified form to submit bad data back to the web application. When the application expects all input validation to have already been done by the web browser, and therefore doesn't double check the input, your web app can be compromised.

For this reason web applications should always validate form inputs on the server *even if they are also validated on the client*. Client−side input can **never** be trusted. Client−side validation should be used as a complement to server−side validation... a means of catching simple input mistakes to reduce the number of times the web server has to validate input. Client−side validation should **never** be trusted as a replacement for additional server−side validation.

Client−side validation is valuable... for highly loaded web sites it offloads a lot of work from the server to the browsers. However if server load is not a factor, it's probably not worth the trouble of writing both client−side and a server−side validation routines for HTML form input.

Use taint checking:

If your web application is written in Perl, turn on Perl's "taint checks". These attempt to ensure that data coming from outside the program cannot accidentally be used to affect something at the operating system level, such as an unchecked user input field being used to erase a file.

Perl taint checks can be turned on by adding the −T parameter to #! line of the program. For example:

```
#!/usr/bin/perl −T
```

Beware that Perl's taint check is far from foolproof. It is not a substitute for knowing web application security issues and writing secure programs. More information about Perl taint checking is found on the 'perlsec' man page (http://language.perl.com/info/documentation.html)

The principle used by Perl's taint checks can be used in other programming languages, but for other languages the check must be applied manually.

Don't use raw path and file names:

Never accept path or file names directly from an HTML form. Instead, use keywords that are *pointers* to actual names stored in a lookup table. In Perl, the most effective way to do a lookup table is by using associative arrays (hashes).

For example, in a Perl web app do NOT do this:

```
WRONG!> $datafile = param('datafilename');
WRONG!> $open DATAFILE $datafile or die;
```

Instead, do something like this:

```
BETTER> my %filelist = ( "name" => "/home/data/name.txt",
BETTER> "address" => "/home/data/address.txt" );
BETTER> $keyword = param('datafilename');
BETTER> open DATAFILE $filelist($keyword) or die;
```

Using the above method, HTML form input is never passed directly to the 'open' command. If a malicious user tries to pass a bad value (such as '/etc/passwd'), it will fail to find a match in the associative array. Lookup tables also prevent a cracker from using a poisoned NULL attack (see Sanitize browser input, above) to shorten strings.

For flexibility, the locations of the files can be pulled in from an external configuration file, rather than be hard−coded into the application. However, make sure the config file cannot be accessed by web users (ie. do not put it in the HTML directory) and that it cannot be changed by other users on the web server.

If your web application absolutely must be capable of opening ad hoc files based directly on browser input, rather than a predetermined list, never accept complete path and filenames from HTML form input fields. Your web application should at least prefix the input filename with an absolute path and strip slashes, backslashes, NULLs and double−dots from the input.

For example:

```
$datadir = '/sites/internet/data';
$datafile = param('datafilename');
sanitize($datafile);
open DATAFILE $datadir . $datafile or die;
```

Use absolute path and filenames:

Never assume a web app is being executed from a particular directory on the server. When opening files, whether for reading or writing, always use a fully−qualified absolute path and filename. For example, instead of referring to a file like this:

```
open '../../../data/mydata.txt'
```

Use a complete pathname such as:

```
open '/sites/internet/data/mydata.txt'
```

Specify the mode when opening a file:

When opening a file for reading only (such as a configuration file), *specify* that it should be opened read−only. Most programming languages allow modes to be specified when opening files: read−only, write, append, read/write etc. and the correct mode should always be specified. Do not rely on the default mode of the programming language… it may change in future versions.

For example, The Perl 'open' command defaults to read−only mode if no specific mode has been specified:

```
open DATAFILE '/sites/internet/data/mydata.txt';
```

However, future versions of Perl may change this, or Perl running on an unusual platform could use a different default. It is better to be specific:

```
open DATAFILE '</sites/internet/data/mydata.txt';
```

Log suspicious errors:

All applications should be written to trap errors. However, web applications are almost always attacked by crackers trying to discover flaws. It is a good idea to not only trap and recover from errors, but also to log events that may indicate an attack:

1. Before opening a file for reading, a web application should check that the file exists and is readable. If the filename was constructed based on input from an HTML form, attempts to access a non−existent file or one it doesn't have privileges to read can signal an attack.
2. If a web application was written to be called with the POST method, raise an alarm if someone calls it using other methods such as GET and PUT. Some web apps written to work when called by one method will fall apart when called by others. Even if your app handles with all methods properly, many crackers try this form of attack. Detecting it will warn you someone is trying to crack your application.
3. If certain HTML input fields should always be present in a form (such as hidden fields that store values), raise an alarm if the form is submitted without them. Missing fields could indicate a cracker is calling the web app from a saved version of the HTML form or from a script attempting to break the app.
4. For Unix−based web applications, any input containing '/etc/passwd' should set off alarm bells. For Windows applications, attempts to access registry files or .pwl files should be monitored. Input with ".." is always suspicious, indicating someone is trying a relative path attack described above.
5. Denial of service attacks (where a malicious user tries to overload a server or application) can be detected by watching for repeated access from the same IP address. More than one request per second from the same IP may indicate a denial of service attempt, or that a cracker is running an automated script attempting to break the application.

Look beyond the web application:

All the above recommendations are methods of locking the "barn door"... securing the way users normally access a web application.

However, securing the application itself is wothless if an attacker can simply telnet to the web server and crack the administrator's password. Obviously web security begins with having secure servers and networks in the first place.

Many organizatations make the mistake of thinking skilled application developers are natually also skilled system and network administrators. This is rarely the case. Just as being an application developer is a full time career, so is being a system administrator, network manager and security analyst.

Many of the well−publicized attacks on web sites targeted well−known operating system and network problems that were missed simply because the organization was focused on developing the web applications and design of the web site, or assumed their development team had all the answers.

Truly securing web applications requires a combined effort in many areas: application design, server management, and network management. Professionals specializing in one of these areas require a good knowledge of the others, but in the end each is a specialty that requires the attention of a focused individual or team.

Recommended reading:

World Wide Web security FAQ:
http://www.w3.org/Security/Faq/www−security−faq.html

CERT advisory 97.25.CGI:
http://www.cert.org/advisories/CA−1997−25.html

CERT advisory CA−2000−02
http://www.cert.org/advisories/CA−2000−02.html

Secure Programming for Linux and Unix HOWTO (David A. Wheeler):
http://www.dwheeler.com/secure−programs/

Hidden form field vulnerability white papers (InfoSec Labs):
http://www.infoseclabs.com/mschff/mschff.htm

Programming Perl (Second Edition):
O'Reilly Associates Inc ISBN 1−56592−149−6

Comments, suggestions, criticisms, additions to this document?
Please e−mail tips@advosys.ca

Latest version of this document available at http://advosys.ca/tips/web−security.html
Copyright © Advosys Consulting Inc. Ottawa Canada. All Rights Reserved.
Last modified Apr 13 2001

Advosys Consulting Inc.

Copyright and terms of use

Use of this document

Permission to use this document from the Advosys Consulting web site is granted, provided that (1) This notice notice appears in all copies, (2) use is for informational and non−commercial or personal use only and will not be copied, reprinted, or posted on any network, computer or broadcast in any media, and (3) no modifications of the document are made.

Educational institutions (specifically K−12, universities and community colleges) may reproduce the Documents for distribution in the classroom, provided that (1) the below copyright notice appears on all copies, and (2) the original Uniform Resource Locator ("URL") of the document on the Advosys Consulting web site appears on all copies.

Use of this document for any other purpose requires written permission of Advosys Consulting Inc.

Copyright Notice:
Copyright © Advosys Consulting Inc., Ottawa Ontario Canada.
All rights reserved.

Limitation of liability

Advosys Consulting take no responsibility for the accuracy or validity of any claims or statements contained in the Documents and related graphics ("the content") on the Advosys web site. Further, Advosys Consulting Inc. makes no representations about the suitability of any of the information contained in the content for any purpose. All such documents, related graphics, products and services are provided "as is" and without warranties or conditions of any kind. In no event shall Advosys Consulting Inc. be liable for any damages whatsoever, including special, indirect or consequential damages, arising out of or in connection with the use or performance of information, products or services available on or through the Advosys Site.

Trademarks

Product, brand and company names and logos used on the Advosys web site site are the property of their respective owners.