Perl In Secure Web Development

Jonathan Worthington (jonathan@jwcs.net)

August 31, 2005

Perl is used extensively today to build server side web applications. Using the vast array of modules on CPAN, one can easily put together web applications that do a massive range of tasks. What is all too often neglected is that the application is accessible to everyone on the internet, including people who intend to make the application do things it decidedly was not supposed to do.

1 About Security

Security is about protecting assets from a malicious and intelligent adversary. An asset might be a customer or order database on an e-commerce site, the posts in a personal guest book or messages that relate to an innovative product a company is developing that are posted on a password-protected message board system. The malicious adversary may be a competitor on an espionage mission, a SPAMmer looking for yet another way to send their junk emails without getting caught or a teenager who likes to show off by defacing websites.

1.1 What is being protected?

After identifying the assets that need to be protected, the next question is what about them needs protecting? Is it secrecy (protection against unauthorised viewing), integrity (protection against unauthorised modification) or availability (protection against attacks that result in a denial of service)?

1.2 Security Is Relative

Even if a web application itself is perfectly secure (which is unlikely in large systems), the web server software or the operating system may have security holes that are yet to be discovered. Even if they don't, there is still the possibility of social engineering or mounting a physical attack on the system. A system can be made more secure or less secure, but not perfectly secure.

1.3 The Economics Of Security

The bad news is that the odds are stacked against you from the start. You need to find and close all of the security holes in your web application. Your adversary only needs to find one of them (more specifically, one useful in performing the kind of attack they desire).

The good news is that the adversary has limited resources, so a web application can be "secure enough" by being hard enough to attack that the adversary has expended their resources before they succeed. The resources an adversary has will usually be related to the value of the asset. It is unlikely anyone is going to spend an entire month trying to deface a personal guest book (insufficient motivation resources). At the same time, it's unlikely that an adversary looking to steal a competitor's customer database is going to give up after 10 minutes if there is potentially a big financial benefit available if the attack succeeds. The amount of effort put into protecting assets accessible through a web application and the web application itself depends on how much value it is perceived to have.

2 The Importance Of Validation

Validation involves checking that data given as input to the web application contains what was expected and then appropriately handling the situation when it does not. Lack of or insufficient validation is likely the single largest cause of exploits in web applications.

2.1 What's Allowed, Not What's Denied

In general, code that validates data should check that the data contains what is known to be valid rather than checking that it does not contain what is known to be invalid. Checking the data contains what is known to be valid is safe by default - the worst that can happen is that something that should have been accepted is rejected. If the data is checked to ensure it does not contain all the things known to be invalid, then there is a risk of missing one of the invalid cases, which could lead to a security hole.

```
# Good - we only accept valid phone numbers here.
if ($phone !~ /^[\d\s()-]+$/) { error(); }

# Bad - we try and protect against insertion of HTML tags to
# avoid XSS attacks, but potentially miss other problems with
# the data.
if ($phone =~ /<|>/) { error(); }
```

2.2 To Make It Safe Or To Abort?

The obvious and safe thing to do when data fails validation is to report this back to the user and halt the script. In some cases, however, it may be more desirable to correct the data so that it passes validation. For example, if the data was detected to contain the "<" or ">" characters then it could be seen as an attempt to carry out an XSS attack. If what was being validated was the "message" field in the "Post New Message" script of a message board system running on a mathematics site, the use of these characters is likely legitimate.

Instead of complaining, these characters can simply be made safe by replacing them with their HTML escape codes "&lt;" and "&gt;". Also known as escaping or sanitising, this is sometimes done automatically to all data that enters the web application from the user.

2.3 A Note On Client Side Validation

Client side validation does little to protect web applications. It can be bypassed by turning off client-side scripting if it is in the way of an attack. Furthermore, it is dangerous to rely on any restrictions that a web browser places on what data can be sent to a server side web application. By constructing a simple script, a request containing any data an adversary desires can be formed and sent. Thinking about possible attack strategies involves thinking outside of the web browser.

3 Some Common Attacks

While it would be impossible to list all of the attacks that might be attempted against a web application, what follows are some of the most common types of attack and how to protect against them.

3.1 Injection Attacks

An injection attack take advantage of a lack of validation to change the behaviour of a web application.

3.1.1 SQL Injection

Many web applications connect to a database and perform SQL queries on it. A typical login script may contain a snippet of code like the one below.

```
my $sth = $dbh->prepare("
    SELECT userid, usertype
    FROM   users
    WHERE  login = '$form{'login'}' AND pass = '$form{'pass'}'
");
$sth->execute;
if ($sth->rows == 0) {
    # Invalid login...give error message.
} else {
    # Valid login...fetch details, etc.
}
```

Imagine that `$form{'pass'}` is not validated and any arbitrary value can be placed into it. It is possible to insert characters that are part of the SQL syntax and change the behaviour of the query. For example, if `$form{'pass'}` were to contain "`' OR '' = '`", then the SQL query becomes the following:

```
    SELECT userid, usertype
    FROM   users
    WHERE  login = '$form{'login'}' AND pass = '' OR '' = ''
```

The WHERE clause of this query will always evaluate to true (as an empty string is always equal to an empty string) and therefore every row in the users table is returned. In this event, most web applications will just look for the one row they expected and assume that the correct login and password were provided. Therefore the authentication mechanism can be bypassed. Worse, this attack could be extended to allow an attacker to log in as a particular user or user type without knowing the password.

A number of solutions exist. One is to sanitise all values, replacing every `'` with `\'`. However, the \ itself has special meaning in SQL syntax and it is possible to formulate an attack using the ability to insert arbitrary backslashes while relying on the script to escape quotes. Thus \ should be escaped to \\ and then the quotes escaped. Alternatively, each of these could be escaped to HTML `&#xx;` style codes.

```
for (keys %form) {
    $form{$_} =~ s/\\/\\\\/g;
    $form{$_} =~ s/'/\\'/g;
}
```

A better solution for drivers that support it is to use place holders and get the DBI to do the escaping, as demonstrated below.

```
# Put question marks in place of variables we'd use.
my $sth = $dbh->prepare("
    SELECT userid, usertype
    FROM   users
    WHERE  login = ? AND pass = ?
");

# Specify variables here, and DBI handles all escaping for us.
$sth->execute($form{'login'}, $form{'pass'});
```

Queries executed using the `do` method of a database handle need equal care, or a delete one record query could become a delete the entire table query. These can be re-written to use prepare and execute so place holders can be used.

3.1.2 Path Injection

Sometimes data provided to the script is substituted into a file path.

```
open FILE, ``< data/$form{'userid'}.dat'';
```

If `$form{'userid'}` is not validated properly then the system could allow access to arbitrary files. Putting a `../` in a path allows the directory tree to be traversed upwards; any file ending in ".dat" on the server that the script has permissions to read (or write, if a write operation is being performed) can be opened instead of the one the developer of the web application intended.

If the file path contains no extension, then any file can be accessed. If there is an unwanted extension, it is sometimes possible for an attacker to pass in a NULL character (ASCII 0). This means that when the string is passed to the operating system I/O functions, written in C or C++, then anything beyond the NULL may not be seen.

The solution is to ensure anything that is placed into "open" statements has been validated. Again, validating inclusively is best; it is safe by default, whereas trying to catch attempts to do directory traversal is not and still leaves the possibility of a shell injection attack.

3.1.3 Shell Injection

Shell injections are much like path injections apart from the data that has not been validated is passed directly to the shell for evaluation. Any data used in `‘backticks‘`, `system`, `exec` and `open` that has not been validated will almost certainly lead to an attack of this style being possible. It is more serious than a path injection in that it directly enables arbitrary code execution on the server hosting the web application.

For example, imagine a program contained the following code.

```
my $feedback = ‘python log_parser.py $logpath‘;
```

The variable $logpath could be set to " blah ; rm -rf /* ", with obvious bad consequences. Other more subtle attacks could include installation of a trojan and emailing copies of the site source and/or other data files to the attacker.

The open statement provides an even more subtle way for this kind of attack to take place. As a contrived example, imagine the following program was written.

```
# Get path.
print "Enter path: ";
my $path = <>;
chop $path;

# Display file.
open FILE, "$path";
print while <FILE>;
close FILE;
```

If $path contains "rm *.pl |" then the pipe character on the end will cause the script to execute what precedes it in the shell.

The solution, once again, is in properly validating incoming data. The open example in particular shows how a lack of understanding of the semantics of the implementation language can cause weak validation conditions to lead to unforeseen security issues.

3.1.4 Mail Header Injection

A script vulnerable to a mail header injection attack is a SPAM senders delight. Many scripts that directly use a sendmail program to send emails have code like much like this:

```
open MAIL, "| /usr/sbin/sendmail -t"';
print MAIL "To: $toaddr\n";
print MAIL "From: $fromaddr\n";
print MAIL "Subject: $subject\n\n";
# Send body of email.
```

Generally scripts like this are pretty careful not to fall into the obvious trap and validate $toaddr so that the script can not be abused and made to send email to anyone. A more subtle attack is possible if $fromaddr or $subject go unvalidated, however. Imagine that $subject can be given an arbitrary value. An extra header could be inserted by making it contain a line break, for example:

```
MAKE YOUR WIFE SCREAM WITH HUGE MORTGAGE PAYMENT ENLARGEMENTS
Bcc: lots@of.us get@th.is
```

This is particularly nasty as the recipient of emails sent to them through the To address of the script have no indication that the junk email has been sent to anyone besides them.

3.2 Cross Site Scripting (XSS)

XSS could be seen simply as Yet Another Injection Attack, but it is worth separating this one out as the attack directly involves other users of the web application.

The majority of web applications accept data from users and later display that data on generated pages; in many cases the data is displayed to other users of the web application. If the data the user provided contained HTML tags then, unless proper validation and/or sanitisation takes place, these will be placed into generated pages. This enables a malicious user to to change the page layout (useful for defacement), present false data and insert arbitrary client side scripts into the page. Doing the last of these is known as an XSS attack.

If there is a JavaScript (or other client-side scripting language) vulnerability then an insecure web application can be turned into a trojan distribution mechanism. A more commonly exploited feature of client side scripting is that scripts have access to the cookies for the site that the script was served from. These can be accessed and covertly sent to another server. If the cookies contain authentication related data or other personal data about a user then this could allow an adversary to pretend to be the user that was authenticated, hijacking their session and performing operations as if they were that user.

It may seem that escaping the `<` and `>` characters is enough. This is a good start, but unfortunately it is insufficient. Imagine a link directory accepts a URL from a user then inserts it into an anchor tag, for example “`<a href="$url">$title</a>`”. If `$url` contained “`javascript:some malicious code`” then when the link was clicked the malicious code would be executed. The solution is to introduce validation that only allows valid URLs.

Finally, web applications should ensure that anything that is being placed into a page inside quotes does not itself contain quotes. Otherwise insertion of arbitrary attributes is possible, almost always allowing client side scripts to be inserted. Validation is, once again, the key.

3.3 Attacks On Systems With Multiple Users

Frequently a web application has a set of objects associated with a particular user that the user is allowed to manipulate but other users are not. For example, a link directory may intend to allow users to only be able manipulate links that they have added.

A common construction involves a page containing a list of the links that the currently logged in user owns, with a link to edit each one. A query to select the links may look like the one below.

```
SELECT id, title, url description
FROM   listings
WHERE  userid = $auth_user{'userid'}
```

A link to edit a listing will probably be generated as follows.

```
<a href="listing_edit.pl?id=$id">Edit</a>
```

The edit listing script therefore receives a listing ID, fetches it and displays the current data in a form, allowing it to be edited. Once it has been edited, the user submits the form and the data is saved. An incorrect, but seemingly valid update query may look as follows.

```
UPDATE listings
SET    title = '$title',
       url = '$url',
       description = 'description'
WHERE  id = $listingid
```

The problem here is that, unless any other checks are done, any user can edit and update any listing they desire as the edit listing script fails to check that the listing is associated with the user attempting to update it before the update is allowed.

The solution is to ensure that these checks are done. An experimental security feature that has been implemented by the author was to write a wrapper for the DBI that checks that when a query is done on one of a given list of tables, a given field is involved in the WHERE clause. This can, in development, catch a few mistakes of this kind.

3.4 Denial Of Service Attacks

Often the availability of an asset or of the web application as a whole is important. Denial of service attacks attempt to make the application unavailable to other users by monopolising a limited resource. That resource might be bandwidth, disk space, available memory or available processing power.

Preventing denial of service attacks is something that needs to be considered throughout the application stack. What follows is just a small handful of web application level issues that can enable or contribute towards denial of service attacks.

- Failing to check the length of input data can lead to larger than expected resource consumption. For example, an image upload script that fails to check the size of the uploaded images can be attacked by uploading a number of gigabyte sized “images”. Also, when a large amount of processing is done on data, the time taken will usually be proportional to the quantity of data, so a number of much larger than expected requests given simultaneously can affect the web application’s responsiveness towards other requests.

- A related issue to input size is that the time or memory used by many algorithms does not grow linearly with the size of the input. This should influence the maximum allowed input size.

- Some algorithms have degenerate cases. For example, quicksort using the first element as the pivot will have quadratic order rather than the expected pseudo linear order for certain sets of data (for example, where the data is already sorted or reverse sorted).

Conclusions

Clearly validation is at the heart of writing secure web applications. Perl offers some help in providing taint mode, enabled by starting `perl` with the `-T` flag. This marks any variables that have come from an untrusted source as “tainted” and prevents them from being used in certain unsafe operations, including `open` statements, backticks and `system` statements. Instead, a pattern match has to be done against the tainted variable, then any captures (`$1`, `$2`, etc) will be untainted and can be used with unsafe operations. The idea is to force the programmer to do some kind of validation, though it is up to the programmer to use a regex that is not too liberal.

However, taint mode will not protect against all attack vectors. Of the few attacks described in this paper it only really deals with path and shell injection. There is still much work to do in addition to using taint mode in order to develop more secure web applications.

The author of this paper has naively built web applications vulnerable to many of the attacks described in this paper. Today he can only look back in horror at some of his early work and know that, despite a large number of changes to the way he develops web applications and security bug-hunting efforts, there are inevitably still some vulnerabilities lurking in production systems today. Though not a security expert, he has been able to greatly improve the security of his web applications by having an awareness of some of the potential issues, and writes this paper in hope of helping others do the same.