Secure Web Development In Perl

Jonathan Worthington

French Perl Workshop 2007

Secure Web Development In Perl

Overview

We'll look at both the theory and practice of security in web applications.

- **The Academic Bit**
  - Security analysis
  - The economics of security
- **The Practical Bit**
  - A range of common exploits
  - How to protect against them

Security Analysis

What is security?

Security is about protecting assets from a malicious and intelligent adversary.

- **Examples of assets:** a customer database, corporate secrets or internal data, posts in the guest book of a personal site.
- **Examples of adversaries:** a malicious competitor, a SPAMmer, an angsty teenager who thinks their skillz are 1337.

What is being protected?

After identifying assets, identify what it is about them that needs protecting.

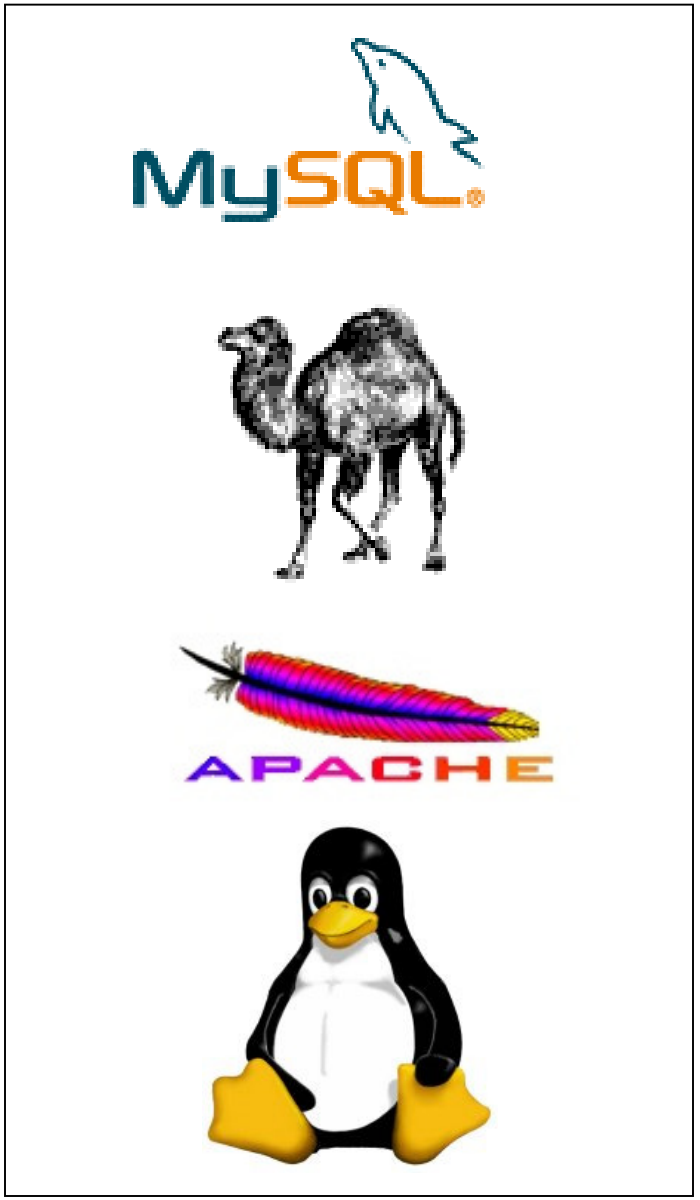
- **Secrecy:** protection against unauthorised access, viewing, copying, etc.
- **Integrity:** Protection against unauthorised modification and deletion.
- **Availability:** Protection against attacks that render the asset unusable.

Secure Web Development In Perl

Security is relative

Even if your web application were to be perfectly secure...

- The web server might have security holes
- The database server might be insecurely configured
- The OS might have security holes



Security is relative

And even if that's all perfectly secure, you've still social engineering to worry about.

- One stupid person can render millions of dollars worth of security software pointless
- One study found a scary number of people would reveal their password for chocolate!
- If that doesn't work, try money, identity theft, intimidation, sex…
- People are weak. Or stupid. Or both. :-)

Security is relative

Then there's the whole issue of physical security…

- If you have physical access to a machine, you've a lot more possibilities.
- Social engineering is one path to physical access.
- Just how secure is the datacenter where your servers are really? There was a story of a break-in at one in the news just last week.

The Economics Of Security

The Economics Of Security

The bad news: The odds are stacked against you.

- You have to find every security hole in the system and deal with it.
- An attacker need only find one that is suitable for the kind of attack they want to mount.

The Economics Of Security

The good news: The attacker has limited resources.

A system could be considered “secure enough” if an attacker is likely to spend all of their resources before they succeed in compromising your system.

The Economics Of Security

The resources an attacker has are usually related to the value of the asset that is being protected.

- It is unlikely anyone is going to spend an entire month trying to deface the guest book on your personal home page.
- It is also unlikely that an attacker looking to steal a competitor's customer database is going to give up after 10 minutes.

Solutions To Web Security Problems

The Practical Bit

Usually I present a problem, then its solution. However, we're going to look at a solution first, then some attack vectors, because…

- The vast majority of security issues I'm about to present have the same solution.
- The solution sounds boring, so I'll present it while everyone is still awake. :-)

Validation

Validation involves…

- Checking that data given as input to the web application contains what was expected.
- Appropriately handling the situation when it does not.

Lack of or insufficient validation is likely the single largest cause of exploits in web applications.

Doing Validation Right

Normally you should be doing **positive validation**.

- Check that the data contains **what is expected**, rather than checking if it contains things that could be dangerous.
- That way, the worst case is that something that should have been accepted is rejected.
- With negative validation, it is easy to miss something potentially dangerous.

Doing Validation Right

Example: positive vs. negative validation

```
# Good - we only accept $phone if it contains things
# that are valid in a phone number.
if ($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();
}
```

The Perl Angle On Validation

Perl makes validation quick and easy.

- For hand-made validation code, regular expressions are very useful, and more convenient to use in Perl than in many other languages.
- There are some useful CPAN Modules:
  - HTML::FormValidator
  - Data::Validate
  - …

The Perl Angle On Validation

Perl can also run in taint mode.

- Here, all data from outside the program is considered tainted. Copying data from a tainted variables will taint the destination.
- You have to validate, using a regex, to untaint a variable.
- Using a tainted variable in unsafe operations is an error.

Sanitizing Input

This involves making user input safe when it would otherwise fail validation.

- On a message board discussing maths, the < and > characters may be used frequently.
- However, we shouldn't just output them directly as they can mess up page layout or allow for a XSS attack (more on XSS attacks coming later).

Sanitizing Input

We transform dangerous input into something safe.

- In this case, we can turn all < into < and all > into > to make the input safe.

```
$input =~ s/</&lt;/g;
$input =~ s/>/&gt;/g;
```

- This can be done on all form data to make user input safer “by default”.
- Sanitization doesn’t imply no validation needed.

Thinking Outside The Browser

Client side validation is of little help when it comes to server side security.

- JavaScript can be turned off easily.
- Custom requests can be assembled, bypassing any browser constraints.
- For example, even if the browser shows a single line text entry field, a custom request could contain line breaks.

Exploits

The Exploits

The following slides present some of the most common exploits that web applications are vulnerable to.

- This will not be anything close to a complete list.
- Validation will come up as The Solution™ again and again.

Injection Attacks

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

- Usually these attacks take advantage of certain characters having special meanings.
- For example, the new line character has special meaning in a mail header – it denotes that another header follows.

SQL Injection Attacks

Many web applications use an SQL database for data storage.

- SQL is a language, and like any other language certain characters have special meanings.
- Not sanitizing or validating input that is placed into an SQL query potentially enables an attacker to change the meaning of a query.

SQL Injection Attacks

Imagine $form{‘pass’} is not validated or sanitized and is used as follows.

```
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.
}
```

SQL Injection Attacks

What happens if the user was to enter the following value for $form{‘pass’}?

' OR '' = '

SQL Injection Attacks

The SQL query will look like this:

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

- The WHERE clause will always evaluate to true!
- You will log in as the first user in the table.
- In multi-user systems, how often is the first account in the table an administrative one?

Preventing SQL Injection Attacks

One solution is to sanitize the data with a couple of simple substitutions.

- Obviously, need to sanitize the quote character, either to a HTML &#xx; style sequence or by putting a \ before it.
- Escaping with a backslash alone is not enough, however!
- Also need to sanitize the backslash, otherwise quotes can be escaped as the attacker wishes.

Preventing SQL Injection Attacks

The best solution for database drivers that support it is to use placeholders. This way, we pass the buck to the DBI to do the sanitizing.

```
# Put question marks in place of variables we want
# to substitute in.
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'});
```

Preventing SQL Injection Attacks

I've only presented examples of using the prepare/execute/fetch/finish method of doing SQL queries.

- $dbh->do(…) style queries present an equal risk; that "delete one thing" could easily be turned into a "delete everything".
- To use place holders, these need to be re-written as a prepare/execute/finish sequence.

Path Injection Attacks

Sometimes user supplied data is used to generate a file path.

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

- Substituting unvalidated data into a path may enable the attacker to modify the path and/or the filename to one of their choice.
- Using “../”, the attacker can move up the directory tree.

Path Injection Attacks – NULL Tricks

The file extension can really get in an attackers way if they wish to overwrite a file with a different one.

- If the path gets passed to a C routine, then putting a null character (code 0) in gives the attacker the ability to snip off the extension.

Preventing Path Injection Attacks

Validate everything that is to be substituted into a path or filename.

- Obviously, never let any slashes (forward or backward – remember Win32 and *NIX) through.
- And remember the NULL issue.
- This shows why positive validation is a Good Thing™ - will eliminate these 2 cases “by default”.

Shell Injection Attacks

Shell injections are much like path injections apart from the data that has not been validated is passed directly to the shell for evaluation.

- In Perl, this can happen with anything placed between backticks and passed to the functions system(…), exec(…) and open(…).
- Very dangerous - enables direct arbitrary code execution.

Shell Injection Attacks

Imagine if the following line was executed and $logpath was unvalidated.

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

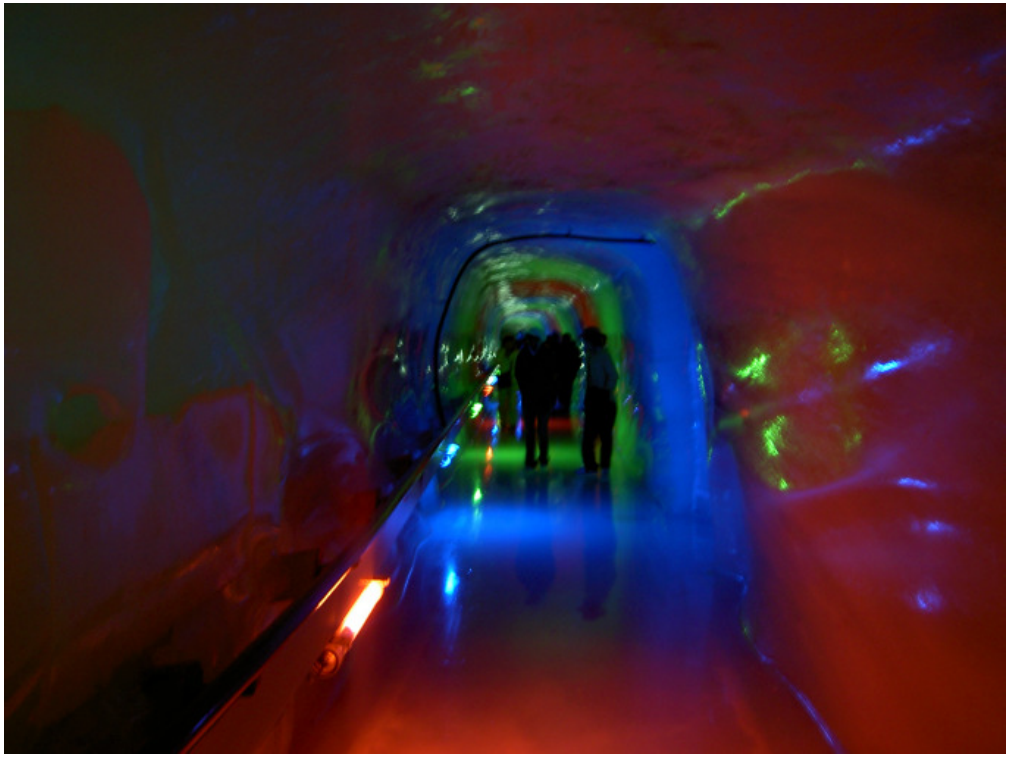
- Obvious potential damage; what if $logpath contained "**blah ; rm -rf /***"?
- More subtle attacks could include installation of a trojan and emailing copies of the site source and/or other data files to the attacker.

Shell Injection Attacks

The Perl open statement provides a sneaky way to do a shell injection attack.

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

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



Shell Injection Attacks

What happens if the user enters "**rm *.pl |**"?

- The pipe character has special meaning in an open statement.
  - The pipe at the end means "execute this command and read the input it gives".
  - The command is executed at the shell!
- **Once again, validation is the answer.**

Mail Header Injection Attacks

SPAM is suckful. So are some form mail scripts. Here's a typical snippet of mail sending code.

```
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.
```

Thankfully, today almost all scripts do not allow an arbitrary value to be in $toaddr. But how many scripts bother to validate $subject?

Mail Header Injection Attacks

Imagine that $subject can be given an arbitrary value.

- An extra header could be inserted by making $subject contain a line break, for example:

```
EVERY WOMAN LIKES A MAN WITH A BIG MORTGAGE!
Bcc: lots@of.us get@th.is
```

- Later headers can be curtailed by inserting a double line-break, allowing control over the message body too!

Mail Header Injection Attacks

The attack has sent the message to additional email addresses.

```
To: some@address.com
From: another@address.com
Subject: EVERY WOMAN LIKES MAN WITH A BIG MORTGAGE!
Bcc: lots@of.us get@th.is

Here comes the body of the email
```

Worst of all, the owner of the script won't know their script has been exploited until somebody reports the SPAM, as they don't see what is in the Bcc header.

XSS (Cross Site Scripting) Attacks

Sort of an injection attack, but directly involves other users of the web application.

- Most web applications accept data from users and later render this data to that user and other users.
- If this data is not validated properly, it could contain unwanted HTML tags, including <script> tags.
- These could be sent to other users.

XSS (Cross Site Scripting) Attacks

Why is being able to to insert HTML or run a script on other user's computers useful for malicious activity?

- Defacing sites, to damage reputation etc.
- Stealing cookies that relate to the site in question, which may contain session data.
- If there is a browser vulnerability about, an attack can be distributed by hijacking XSS vulnerable sites.

XSS (Cross Site Scripting) Attacks

Sanitizing the < and > characters is a good start, but not enough.

- Imagine a link directory that asks the user to input a URL and a name for the link.
- If these were being pulled from a database, the fetch and display loop could look like this:

```
while (my ($name, $url) = $sth->fetchrow_array()) {
    print qq{<a href="$url">$name</a>};
}
```

XSS (Cross Site Scripting) Attacks

Imagine that $url was not validated to ensure it really was a URL.

- It could contain "**javascript:alert('Ha')**", or something quieter and more useful.
- Alternatively, it could contain something like (including the quote): **" onClick="alert('Ha!')**
- Either of these will allow for execution of arbitrary JavaScript when the link is clicked.

Attacks On Multi-user Systems

Some web applications have a set of items associated with a particular user that they are allowed to manipulate, but other users are not.

- For example, a link directory may intend to allow users to only manipulate links that they have added.
- A bad assumption: if a user doesn't have a link to something, they can't access it.

Attacks On Multi-user Systems

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'}
```

Attacks On Multi-user Systems

A link to edit a listing will probably be generated for each link the user owns as follows:

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

The edit listing script therefore receives a listing ID, fetches the name and URL associated with the listing and displays the current data in a form, allowing it to be edited.

So far, so good.

Attacks On Multi-user Systems

Once the data has been edited it needs to be saved back to the database.

- The naïve update query looks like this:

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

- Unfortunately, this allows any user who can construct the appropriate HTTP request to edit the listing.

Attacks On Multi-user Systems

The solution is to always check that the listing belongs to the currently logged in user.

- Can do a separate query to explicitly do the check.
- Alternatively, include the condition of the listing being owned by the current user in the WHERE part of the UPDATE query and check the number of rows the query affected was not zero.

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.

Denial Of Service Attacks

Preventing DOS attacks needs work throughout the entire web stack, not just at the web application level. However, potential web application level issues exist.

- Failing to check for over-sized of input can lead to massive resource consumption.
- Large input sets of data will expose the growth rate of more complex algorithms.
- Some algorithms have de-generate cases.

Conclusion

Insecure Web Applications Exist

How do I know?

- When I started out in web development, some of my code was shockingly insecure.
- A number of popular web applications have been found to contain instances of vulnerabilities discussed in this talk.
- I've carried out attacks discussed in this talk against other people's web applications, successfully.

Conclusions

The main things to take away:

- Security is about protecting privacy, integrity or availability of assets from a malicious attacker.
- The effort expended to protect a system should relate to the cost of it being exploited.
- Validation, validation, validation!

Questions?

Merci!