Secure Programming Lecture 10: Web Application Security

David Aspinall

23rd February 2016

OWASP

The Open Web Application Security Project is a charity started in 2001, to promote mechanisms for securing web apps in a non-proprietary way.

They have local chapters worldwide; the Scotland chapter sometimes meets in Appleton Tower.

Like CERT and Mitre, they produce taxonomies of weaknesses and coding guidelines.

Their most well known output is the OWASP Top 10 list of weaknesses in web applications.

OWASP Top 10 list 2013

- 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

The list is compiled using data from application security consultancy companies, said to comprise over 500k vulnerabilities.

OWASP Top 10 list 2013

- 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

In the next few lectures and lab session we'll look at web app security and some of the main weakness categories.

Overview

We start by going back to basics.

Even if you program web sites using a high-level

- Web Application Framework (*Rails*, *Django*, . . .)
- Content Management System (*Joomla*, *Drupal*, . . .)
- Wiki (*MediaWiki*, *Confluence*, . . .)
- Blog (*Wordpress*, . . .)
- . . . anything

knowing what is happening underneath is important to understand how security provisions work (or don't).

[Similarly, we looked at assembler code and CPU execution for C applications, to understand what was *really* going on]

HTTP

HTTP = Hyper Text Transfer Protocol

- Protocol used for web browsing
  - and many other things by now (**Q.** Why?)
- Specifies messages exchanged
  - HTTP/1.1 specified in RFC 2616
  - request methods: `GET`, `POST`, `PUT`, `DELETE`
- Messages are text based, in lines (Unix: CR+LF)
- *Stateless* client-side design
  - quickly became a problem, hence **cookies**
- NB: HTTP is entirely separate from HTML!
  - HTTP headers not HTML `<HEAD>`
  - HTML is text format for web *content*

HTTP communication

HTTP is a client-server protocol.

- Client initiates TCP connection (usually port 80)
- Client sends HTTP request over connection
- Server responds
  - may close connection (HTTP 1.0 default)
  - or keep it *persistent* for a wee while
- Server never initiates a connection
  - except in recent HTML5 WebSockets
  - WebSockets allow low-latency interactivity
  - expect to see rise in use and security issues. . .

HTTP GET message (simplified)

```
GET / HTTP/1.1
Host: www.bbc.co.uk
User-Agent: Mozilla/5.0
Accept: text/html
Accept-Language: en-US,en;q=0.5
```

HTTP GET message (full)

```
GET / HTTP/1.1
Host: www.bbc.co.uk
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:27.0) Geck
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
```

HTTP Response (simplified)

```
HTTP/1.1 200 OK
Server: Apache
Content-Type: text/html; charset=UTF-8
Date: Wed, 19 Feb 2014 14:30:42 GMT
Connection: keep-alive

<!DOCTYPE html> <html lang="en-GB" > <head> < !-- Barlesque 2.60.1 -->
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<meta name="description" content="Explore the BBC, for latest news,
sport and weather, TV &amp;amp; radio schedules and highlights, with
nature, food, comedy, children's programmes and much more" />
...
```

HTTP Response (full)

```
HTTP/1.1 200 OK
Server: Apache
Etag: "c8f621dd5455eb03a12b0ad413ab566f"
Content-Type: text/html
Transfer-Encoding: chunked
Date: Wed, 19 Feb 2014 20:12:34 GMT
Connection: keep-alive
Set-Cookie: BBC-UID=a583d...4929Mozilla/5.0; expires=Sun, 19-Feb-18 20
X-Cache-Action: HIT
X-Cache-Hits: 574
X-Cache-Age: 50
Cache-Control: private, max-age=0, must-revalidate
X-LB-NoCache: true
Vary: X-CDN

d1c
<!DOCTYPE html>
...
```

Note: cache fingerprint; chunked transfer; **cookie**; cache directives.

Client != Browser

```
[dice]da: telnet www.bbc.co.uk 80
Trying 212.58.244.71...
Connected to www.bbc.net.uk.
Escape character is '^]'.
GET / HTTP/1.0
Host: www.bbc.co.uk
Accept: text/html, text/plain, image/*
Accept-Language: en
User-Agent: Handwritten in my terminal
```

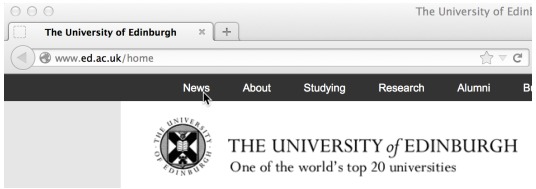
Client != Browser

```
HTTP/1.1 200 OK
Server: Apache
Content-Type: text/html
Date: Wed, 19 Feb 2014 14:26:00 GMT
...
```

Client-side security doesn't exist

- Any program can conduct HTTP(S) communications
- . . . URLs can be constructed arbitrarily
- . . . POST forms content also
- In server-side context, there are *no input validation guarantees* despite any client-side code.

Referer header

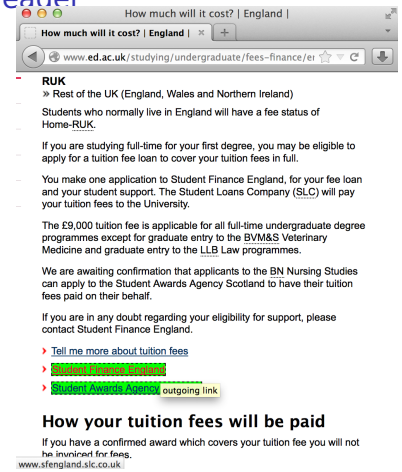



```
GET /news/ HTTP/1.1
Host: www.ed.ac.uk
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:27.0) Geck
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://www.ed.ac.uk/home
Connection: keep-alive
```

Referer header

Question. What immediate security issue arises from this header?

Referer header



Referer header

```
GET /loggedin/secretfile.html HTTP/1.1
Host: www.mycompany.com
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://www.mycompany.com/loggedin/
```

Don't rely on Referer header for access decisions!

- Flawed assumption made in old web apps: *user has navigated to a logged in area, therefore they must be logged in*
- But Referer is from client, cannot be trusted!
- Also risky because of TOCTOU
- and confuses authentication with authorization

Inputs via GET Request

```
http://www.shop.com/products.asp?name=Dining+Chair&material=Wood
```

- Input encoded into *parameters* in URL
- Bad for several reasons:
  - SEO optimisation: URL not canonical
  - cache behaviour (although not relevant for login)

Question. What's another reason this format is bad?

Inputs via GET Request

```
http://someplace.com/login.php?username=jdoe&password=BritneySpears
```

- URL above is visible in browser navigation bar!

POST Request

```
POST /login.php HTTP/1.0
Host: www.someplace.example
Pragma: no-cache

Cache-Control: no-cache
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.5a)
Referer: http://www.someplace.example/login.php
Content-type: application/x-www-form-urlencoded
Content-length: 49

username=jdoe&password=BritneySpears
```

- URL in browser:
  `http://www.someplace.example/login.php`

GET versus POST

- **GET** is a *request* for information
  - can be (transparently) resent by browsers
  - also may be cached, bookmarked, kept in history
- **POST** is an *update* providing information
  - gives impression that input is hidden
  - browsers may treat differently
- **neither provide confidentiality** without HTTPS!
  - plain text, can be sniffed
- in practice, GET often changes state somewhere
  - user searches for something, gets recorded
  - user has navigated somewhere, gets recorded
  - so shouldn't think GET implies functional

When to use POST instead of GET

- For sensitive data, *always* use POST
  - helps with confidentiality but not enough alone
- For large data, use POST
  - URLs should be short (e.g., <=2000 chars)
  - longer URLs cause problems in some software
- For actions with (major) side effects use POST
  - mainly correctness; many early web apps wrong

These are general guidelines. There are sometimes more complex technical issues to prefer GET.

Cookies: state in a stateless world

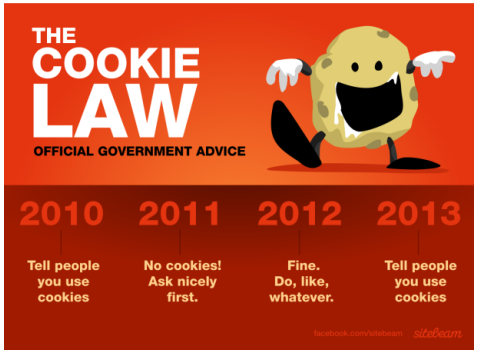
Some state is highly desirable between requests. E.g.,

- remember user's preferences, navigation point, . . .
- web applications: **user logged in**

However, also the less desirable. E.g.,

- advertising network **tracking ids**
- may be shared between websites
- profile user browsing behaviour
- hence **compromise privacy**
- also **risk of theft**
  - if browser/machine compromised
  - cookies passed in clear

Cookies and the law



See Sitebeam's cheeky infographic

Cookies in HTTP headers

- Specified in RFC6265
- Just ASCII plain text
  - Sent by server
  - **Stored in client** (database, filesystem, . . .)
  - Returned by client when visiting page again
- Cookies can be set by the server for a particular path/domain
  - then sent for any page matching
- Multiple cookies may be set and returned
- Cookies may have a **limited lifetime**
  - set by `Expires` or `Max-Age`

Setting cookies

Server -> User Agent

```
Set-Cookie: SID=31d4d96e407aad42; Path=/; Secure; HttpOnly
Set-Cookie: mylanguage=en-GB; Path=/; Domain=example.com
```

User Agent -> Server

```
Cookie: SID=31d4d96e407aad42; mylanguage=en-GB
```

Secure cookies?

RFC6265: *The Secure attribute limits the scope of the cookie to "secure" channels (**where "secure" is defined by the user agent**). When a cookie has the Secure attribute, the user agent will include the cookie in an HTTP request only if the request is transmitted over a secure channel (typically HTTP over Transport Layer Security (TLS) [RFC2818]).*

- . . . provided browser obeys this
- . . . not secure against active attacks
- still, no harm in using (defence in depth)

the HttpOnly attribute is similar, and forbids the browser from allowing JavaScript access to the cookie, in principle.

Expiry dates

Server -> User Agent

```
Set-Cookie: mylanguage=en-US; Expires=Wed, 09 Jun 2024 10:18:14 GMT
```

User Agent -> Server

```
Cookie: SID=31d4d96e407aad42; mylanguage=en-US
```

- Of course, no guarantee cookie is kept for 10 years. . .

Removing cookies

RFC6265: *To remove a cookie, the server returns a Set-Cookie header with an expiration date in the past. The server will be successful in removing the cookie only if the Path and the Domain attribute in the Set-Cookie header match the values used when the cookie was created.*

Server -> User Agent

```
Set-Cookie: lang=; Expires=Sun, 06 Nov 1994 08:49:37 GMT
```

User Agent -> Server

```
Cookie: SID=31d4d96e407aad42
```

- Again, no guarantee of what browser actually does
- . . . if indeed the same browser is being used

Session hijacking

Web apps use session IDs as a credential

- if an attacker steals a SID, he is logged in!

This is **session hijacking**

Many possible theft mechanisms:

- XSS, sniffing, interception
- or: calculate, guess, brute-force
- also **session fixation**
  - using same SID from unauthenticated to logged in
  - attacker grabs/sets SID before user visits site

Session hijacking defences

Web apps (or frameworks) should implement defences, and discard SIDs if something suspicious happens.

- Link SID to IP address of client
  - but problems if behind NAT, transparent proxies
  - ISP proxy pools mean need to use subnet, not IP
  - subnet may be shared with attacker!
- Link SID to HTTP Headers, e.g. User-Agent
  - but can be trivially faked. . . and usually guessed
  - . . . or captured (trick victim to visit recording site)

OWASP: I may be vulnerable if. . .

- User authentication credentials aren't protected when stored using hashing or encryption.
- Credentials can be guessed or overwritten through weak account management functions (e.g., account creation, change password, recover password, weak session IDs).
- Session IDs are exposed in the URL (e.g., URL rewriting).
- Session IDs are vulnerable to session fixation attacks.
- Session IDs don't timeout, or user sessions or authentication tokens, particularly single sign-on (SSO) tokens, aren't properly invalidated during logout.
- Session IDs aren't changed after for a new login.
- Passwords, session IDs, and other credentials are sent over unencrypted connections.

OWASP: How do I do things correctly?

Follow the detailed advice given in OWASP documents:

- Authentication Cheat Sheet
- Session Management Cheat Sheet

Or use a framework in which there is a strong degree of confidence that things have been done properly.

General Secure Programming advice: reuse believed-to-be-secure solutions as far as possible.

Outlook for web authentication/identity

We're likely to see more shared facilities (and a battle):

- Interoperable schemes, e.g. OWASP ASVS
- Perhaps using OAUTH, OpenID
- FIDO, an industry-led initiative

- Maybe Government identity verification, e.g., GOV.UK Verify.

Review questions

HTTP Headers

- Describe three possible vulnerabilities for a web application posed by an attacker who fabricates HTTP headers rather than using the web app running via a reliable browser.
- Explain the reasons for using POST rather than GET. What security guarantees does it provide?

Cookies

- Consider an online grocery merchant that uses a cookie to store the user's shopping basket, including the list of product IDs and their prices, encrypted using a secret key derived from the SID. What threats might be posed and by whom?

References

Some examples were adapted from:

- *Innocent Code: a security wake-up call for web programmers* by Sverre H. Huseby, Wiley, 2004.

as well as the named RFCs.