Secure Programming Lecture 10: Web Application Security (HTTP, OWASP)

David Aspinall

Informatics @ Edinburgh

Outline

Roadmap

In labs and the next few lectures we'll look at **web application security** including

- some of the main weakness categories
- the vulnerabilities that arise,
- and better programming to avoid them.

To understand things, we'll start from some necessary basics of web technology.

Before that, we'll examine a community-driven list of common weaknesses.

Outline

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, OWASP produce taxonomies of weaknesses and coding guidelines.

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

OWASP Top 10

OWASP Top 10 list 2021

- A1 Broken Access Control
- A2 Cryptographic Failures
- A3 Injection
- A4 Insecure Design
- A5 Security Misconfiguration
- A6 Vulnerable and Outdated Components
- A7 Identification and Authentication Failures
- A8 Software and Data Integrity Failures
- A9 Security Logging and Monitoring Failures
- A10 Server-Side Request Forgery

The list is compiled using data for found problems but also from a community survey, to include newer, emerging problem types. Ranking uses CVSS scores for exploitability and impact.

Primarily for **awareness.**

See https://owasp.org/Top10

2021 OWASP Top 10 overview

We'll take a quick look at the 2021 OWASP Top 10 to define each of them at a high level:

- **Definition** what the category means
- **Causes** the general causes of the problem
- **Effects** the typical kind of effects seen

In more detail, each weakness category is mapped onto many more specific CWEs. We'll look at specific examples later.

Question. What is a risk if we focus only on the OWASP Top 10?

A1 Broken Access Control

Broken Access Control

Users can act outside their intended permissions.

- **Causes**: access control policy is wrong or can be bypassed.
- **Effects**: information disclosure, modification, destruction.

A2 Cryptographic Failures

Cryptographic Failures

Lack of cryptographic protection or bad use of cryptography.

- **Causes**: failure to protect data in transit or at rest, use of deprecated or buggy methods.
- **Effects**: data disclosure.

A3 Injection

Injection

User-supplied data is not validated, filtered or sanitized.

- **Causes**: using unsafe APIs, manually assembled commands or queries, lack of defensive resource controls
- **Effect**: data disclosure and modification, remote code execution.

A4 Insecure Design

Insecure Design

Missing or ineffective design of security controls.

- **Causes**: architectural weaknesses arising from misunderstanding threats, using insecure design patterns.
- **Effects**: immediate lack of security, enabling other attacks.

A5 Security Misconfiguration

Security Misconfiguration

Errors in configuration of services or web functions.

- **Causes**: insecure-by-default values unchanged; application stack configurations not secured; degraded security enabled.
- **Effects**: attacks are easier than they should be.

A6 Vulnerable and Outdated Components

Vulnerable and Outdated Components

Out of data and unpatched components, client or server side, including nested app dependencies, engine, database, OS, C libraries.

- **Causes**: starting from old versions or not implementing updates; no regular scanning; misconfigurations.
- **Effects**: opens vulnerabilities to well known or easily discoverable exploits.

A7 Identification and Authentication Failures

Identification and Authentication Failures

User or machine identities are not properly established, or authentication mechanisms are missing or weak.

- **Causes**: broken or missing certificate or SSO checks; mismanagement of session IDs or other credentials; allowing automated attacks.
- **Effects**: attackers can access user accounts and information, perhaps opening way to elevation-of-privilege.

A8 Software and Data Integrity Failures

Software and Data Integrity Failures

Lack of integrity checks on critical software or data downloads and updates, or extended dependencies outwith framework controls.

- **Causes**: unsigned software or unchecked deserialised data; untrustworthy software repositories; compromised build/deployment machines.
- **Effects**: opens attack surface for adjacent malware or tampering with application operation.

A9 Security Logging and Monitoring Failures

Security Logging and Monitoring Failures

Lack, misconfiguration or insufficiency of logging, missing important information or logging without output filtering.

- **Causes**: security-relevant events such as successful or failed logins are not logged; messages are inadequate; no support for realtime monitoring and response.
- **Effects**: long-lived compromises can go unnoticed; mass data breaches and denial-of-service made easier for attacker and harder to diagnose.

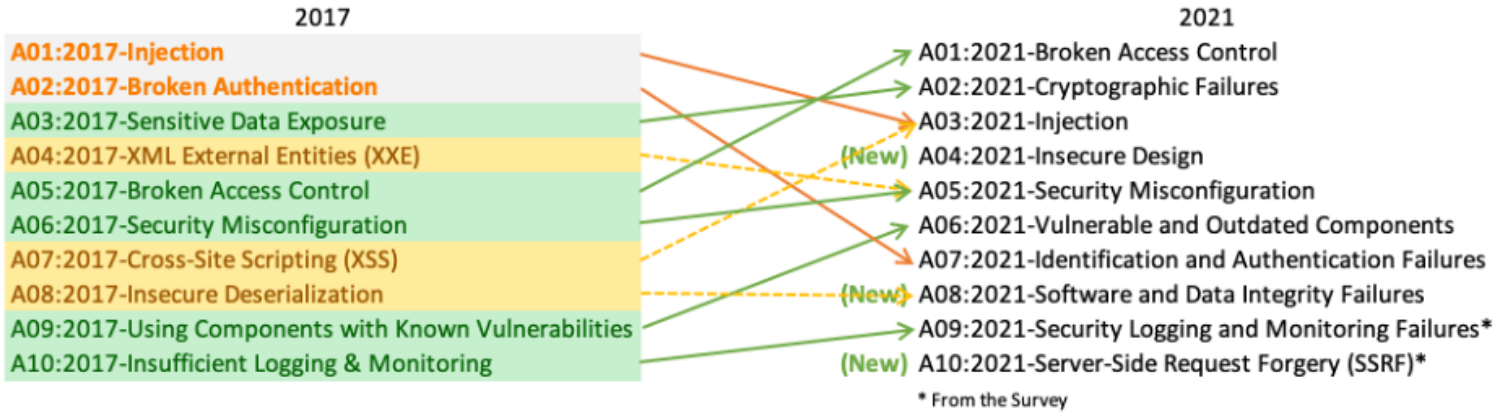
A10 Server-Side Request Forgery

Server-Side Request Forgery (SSRF)

A web-application fetches a remote resource without properly validating URLs.

- **Causes**: web application allows users to add links to other places or content, URLs can be crafted to make malicious requests or for reconnaissance.
- **Effects**: attacker learns about web app or server network architecture, causes actions on other servers or accesses sensitive data.

Changes from 2017 to 2021



Outline

Outline

Web apps: a myriad of methods!

There are many *so* ways of programming web applications:

- Low-code and no-code methods (*Retool*, *LANSA*, . . .)
- Serverless cloud functions (*AWS Lambda*, *GCF*, *Heroku*, . . .)
- Microservice architecture (*Node.js*, *Spring Boot*, . . .)
- Web Application Framework (*Rails*, *Django*, . . .)
- Content Management System (*Joomla*, *Drupal*, . . .)
- Wiki (*MediaWiki*, *Confluence*, . . .)
- Blog (*Wordpress*, . . .)

Web application frameworks

For bespoke applications which don’t fit into other categories, it’s common to use a web app framework.

Graphic from Distinguished.io

Web application frameworks

Choose a programming language, choose a web framework, choose security mechanism. . .

Language	Choices	Security provision methods
PHP	19	per-framework; ACLs, RBAC, OpenID
Java	18	>10, builtin/plugin: Spring, OpenID, RBAC
Python	14	per-framework
JavaScript	5	limited

Wikipedia's handy Comparison of server-side web frameworks which lists over 10 languages, almost 100 frameworks.

Question. How would you choose which framework to use? How would you know how to fix security issues for someone else's choice?

What’s underneath all this?

Knowing what is happening underneath these frameworks is important to understand fundamentally how web security provisions work (or don’t).

It’s also useful to learn about to study the detail of web exploits.

Similarly, we looked at assembler code and CPU execution for C applications, to understand what was *really* going on “under the bonnet” and how low-level code attacks work.

Outline

HTTP

Let’s start at the beginning.

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
  - HTTP/2 in RFC 7540 (mainly efficiency)
- Messages are text based, in lines (Unix: CR+LF)
- *Stateless* client-side design
  - quickly became a problem, hence **cookies**
- **Note**: HTTP is entirely separate from HTML!
  - HTTP headers not HTML <HEAD>
  - HTML is text format for web *content*

HTTP is based around 4 request methods: GET, POST, PUT, and DELETE.

HTTP communication

HTTP is a client-server protocol.

- Client initiates TCP connection, usually
  - port 80 for plain text HTTP
  - port 443 for HTTP over TLS (HTTPS)
- 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 newer HTML5 WebSockets
  - WebSockets allow low-latency interactivity
- In HTTP/2 **Server Push** can pre-emptively send additional responses
  - Idea: anticipate subsequent requests
  - semantic equivalence but caching behaviour subtle

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 (less simplified)

```
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 (less simplified)

```
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
...
```

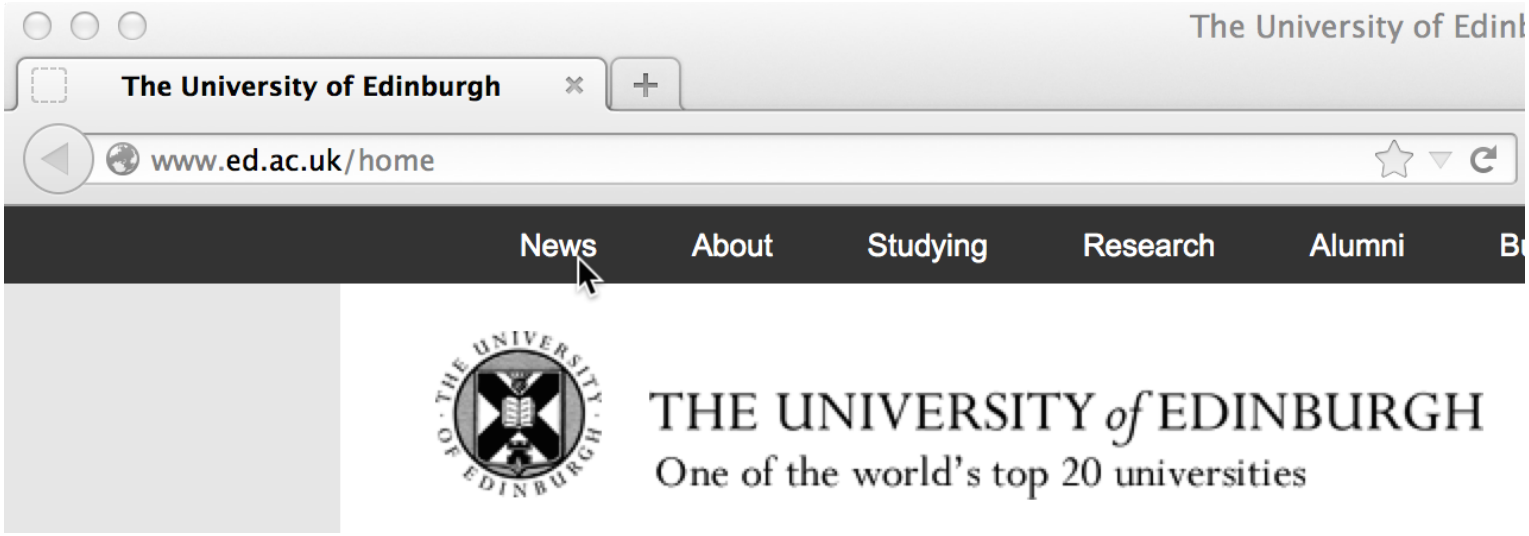
Client-side security doesn't exist

- 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.

Client side security doesn't exist right now, in the future *trusted computing* mechanisms may be able to provide it.

Increasingly web servers are protected by other checks and layers which reject requests that are out-of-the-ordinary.

Referer header

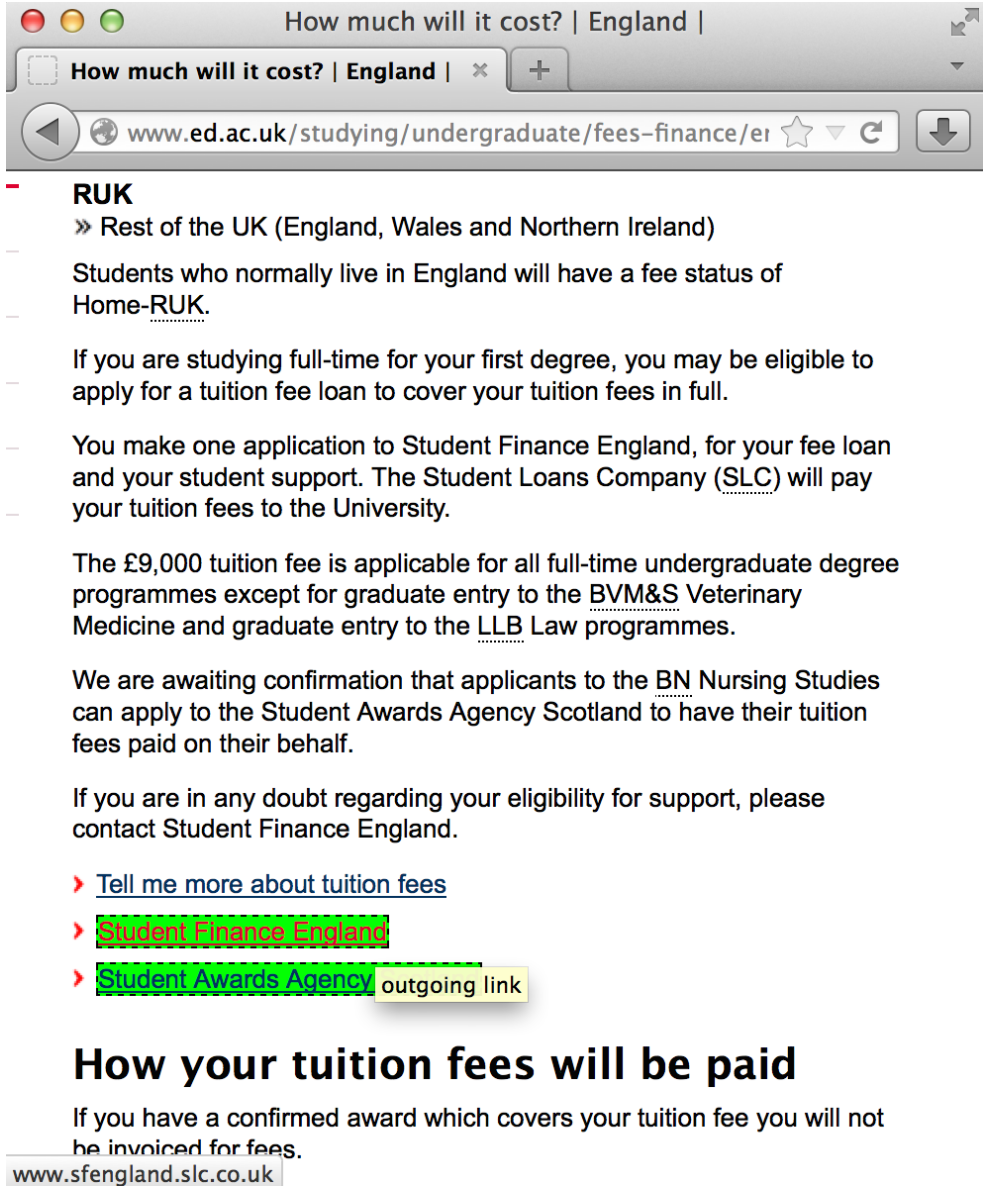



```
GET /news/ HTTP/1.1
Host: www.ed.ac.uk
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.9; rv:27.0) Geck
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://www.ed.ac.uk/home
Connection: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

When to use POST instead of GET

- For sensitive data, *always* use P0ST
  - helps with confidentiality but not enough alone
- For large data, use P0ST
  - URLs should be short (e.g., <=2000 chars)
  - longer URLs cause problems in some software
- For actions with (major) side effects use P0ST
  - mainly correctness; many early web apps wrong

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

Outline

Review questions

OWASP Top 10

- What is the purpose of OWASP and its Top 10 list?

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 P0ST rather than GET. What security guarantees does it provide?

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 and the OWASP resources.