Secure Programming Lecture 12: Web Application Security III

David Aspinall

7th March 2017

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

Anonymous hacks UoE (2012)

Most notable were dumps from Michigan State, Manchester University and the University of Edinburgh who all appear to store at least some passwords without encryption in plain text.

At the University of Edinburgh's biology department, not only were the plain text passwords leaked, but so were the secret question answers. The dump from the physics department at Manchester University revealed full names and corresponding email addresses and passwords, which were, in many cases, simply the user's first name.

[See #ProjectWestWind - Today's Education!]

Starbucks iPhone payments (2014)

Coffee mogul Starbucks is under fire over their data security practices after it was discovered that the company’s iOS payment app does not encrypt customers’ login information.

Daniel Woods pointed out that Starbucks iPhone app stored passwords in the clear. After wider publicity they eventually fixed it.

SOHO Pharming (2014)

See InfoSecurity article

Missing function-level access control (A7)

This is OWASP’s term for not authorizing properly the operations, i.e., functions, that the web app implements. This is separate from handling authorization of

- pages (apps might have several functions per page)
- external objects, e.g., files on filesystem

Common mistake:

- Hiding navigation links to “unauthorized” sections
- . . . assuming (wrongly) this prevents non-authorized users visiting them
- e.g., no AdminPage link if not logged in as an admin
OWASP authorization advice

▶ Have well-specified policy
  ▶ if non-trivial, separate it from code
▶ Manage authorization in a separate module
  ▶ have a single route through code
  ▶ can trace to make sure authorization happens
▶ Make authorization checks for each function
▶ Use deny-by-default policy

Unvalidated redirects and forwards (A10)

Web apps often allow redirections which
▶ send users off-site with a polite message
▶ reroute them immediately


or forwards which
▶ redirect internally to different parts of the same site

http://www.example.com/login.jsp?fwd=admin.jsp

Question. What’s the security risk here?

Giving attackers legitimacy

▶ Attackers can craft URLs that fool users:
  www.example.com/redirect.jsp?url=www.evilhacker.com

These kind of open redirect links are favourites for phishing attacks, especially as ultimate destinations can be concealed in URL encodings. However, this may not directly harm www.example.com.

So, preventing open redirects is a typical example of a community wide desirable security measure (like older cases in network security: open mail relays, ICMP broadcast, etc.): good practice of all provides security for others.

Avoiding redirects and forwards

1. Don’t use them
2. Use them but not with user-supplied parameters.
3. If user-supplied parameters must be used, use an indirection (index value)

Another solution is to do the generation (& validation) of external links statically.

Question. Why is static generation of external links still not bullet-proof?
Information leakage
Reminder from your first security course:
Confidentiality is defined as the unauthorized learning of information.
Presumes notions of
- authorization likely based on an access control policy likely based on authentication.

Information leakage
Information can be learned in a variety of ways:
- direct exposure
  - display on a public web page
  - may be inadvertent on part of the user
- indirect/inferred exposure
  - programming mistakes showing wrong info
  - display of consequences of info
- leakage through side-channel attacks
  - e.g., timing attacks
- (via) offline mechanisms
  - social engineering to get passwords, etc

The Privacy Crisis
Some security researchers and commentators have said:
“Privacy is dead, get over it!”
We can’t stop people intentionally sharing personal information, but it is our job to ensure:
- good policy: advise users about it and its impacts
- good programming: don’t leak data accidentally
- good design: don’t force users to leak data
- good UX/UI: so users understand what they’re doing
Of course there may be competing incentives to expose personal data (Q. for example?).
But it is better that data handling practices are defined and explained clearly in privacy policies for end users.

Example: search data
Google
- may save your web search history forever
  - gives users the option to turn this off; then searches are partially “anonymized” after 18 months.
If you haven’t looked at this before, visit:
  - https://history.google.com/history/
Anonymization of data is very difficult or impossible in general: many examples of linkage attacks have recovered identities.

Example: voice control
Apple’s Siri keeps voice data for up to two years:
Here’s what happens. Whenever you speak into Apple’s voice activated personal digital assistant, it ships it off to Apple’s data farm for analysis. Apple generates a random number to represent the user and it associates the voice files with that number. This number — not your Apple user ID or email address — represents you as far as Siri’s back-end voice analysis system is concerned. Once the voice recording is six months old, Apple “disassociates” your user number from the clip, deleting the number from the voice file. But it keeps these disassociated files for up to 18 more months for testing and product improvement purposes.
Wired’s article from 2013, attempting to clarify Apple’s behaviour.

Example: an amazing mind reader
See the video campaign by Safe Internet Banking in Belgium.
Exercise. What has this got to do with banking? And why should it have anything to do with it?
Privacy by Design

Privacy by design (PbD) is a methodology introduced by the Information and Privacy Commissioner of Ontario in the 1990s. It has 7 foundational principles:

1. Be proactive and preventative, not remedial
2. Make privacy the default setting
3. Put privacy into the design
4. Encourage win-win (not privacy-security trade-off)
5. Provide full lifecycle protection (end-to-end)
6. Be transparent and open about practices
7. Be respectful and user-centric

This process is encouraged in the new EU GDPR General Data Protection Regulation, coming into law from 2016.

Basic strategy for sensitive data handling

0. Define your policy, devise requirements
1. Label the data parts at least informally
2. Sanitize to remove sensitive parts/meta-data
3. Follow the data through the app and check

Check questions for data flow:
- is the data stored as plain text long term (e.g. backups)?
- is the data transmitted as plain text?
- are encryption algorithms strong enough?
- are browser security directives/headers set appropriately?

OWASP advice for sensitive data:

1. Don’t store unnecessarily
2. Encrypt appropriately (e.g., large keys if long term)
3. Use known strong encryption algorithms
4. Manage passwords properly
5. Disable auto-completion on forms, disable caching

Regulation

If you manipulate or store user data you have (legal) responsibilities for managing it properly. For example:

- DPA UK Data Protection Act
  - organisations must register and data must be kept “safe and secure”. The ICO prosecutes and fines.
- GDPR set to come into EU state laws 2017–
  - breaches notified, rights of erasure, data portability
- Finance: Payment Card Industry Data Security Standard
  - requirements for anyone who processes card data
  - larger merchants are audited
- Health: HIPPA (in the US)

Given the scale and frequency of data lossage, regulation/enforcement increasing. (Expect security companies to push for and profit from this.)

Cross Site Request Forgery (CSRF) (A8)

- Exploit browser’s trust relationship with a web site
  - local intranet web site (home router admin, …)
  - banking or email site user is logged into
  - browser is authorized to connect here
- Attacker triggers malicious action
  - get user to open malicious link
  - browser undertakes action on target site

Question. How does CSRF differ from XSS?

CSRF in pictures
CSRF in code

Alice is logged in to the (hypothetical) GeeMail web mail system.

She sends an email with this form:

```html
<form action="http://geemail.com/send_email.htm" method="GET">
  Recipient's Email address: <input type="text" name="to">
  Subject: <input type="text" name="subject">
  Message: <textarea name="msg"></textarea>
  <input type="submit" value="Send Email">
</form>
```

Example GET request

Which sends a request like this:

```
http://geemail.com/send_email.htm?to=bob@example.com
&subject=hello&msg=What%27s+the+status+of+that+proposal%3F
```

Attacker’s cross-site request

Now Mallory just needs Alice to visit a site which loads an image link, e.g., by putting a fake image on his own blog:

```html
<img src="http://geemail.com/send_email.htm?to=charlie@geemail.com
&subject=Hi
&msg=My+email+address+has+been+stolen">
```

and Alice’s browser will send an Email!

Confused Deputy problem

CSRF is a case of the Confused deputy problem.

The issue motivates the use of capabilities.


Avoiding CSRF problems

In general: tricky. Need some way to assure the server code that the request has come from intended place.

- Referer header not tamper proof, may be absent
- Session ID cookie sent based on destination
- The Same Origin Policy restricts client-side code

Best strategy: use a good framework that provides built-in protection.

But how do they work?

- Don’t use GET for any (sensitive) state change
  - basic starting point
- Use a “double cookie” trick, repeated in POST
  - set a secure secret session ID in a cookie
  - submit it in cookie and hidden field on form
  - server-side, check fields identical
  - this works but has drawback (Q. what?)
- Use a special CSRF token in POST
  - secure random number (challenge) for each login
  - send this with POST and check server-side
  - save state: generate using hmac from session ID

See Robust defenses for cross-site request forgery, Barth et al, ACM CCS 2008.
Same-origin Policy (browser-side isolation)

The same-origin policy is a standard browser-side mechanism to protect simultaneously running web applications from one another. It restricts access to:

- DOM (i.e., representation of the document)
- APIs for web access (XMLHttpRequest)
- Cookies, HTML5 local storage APIs
to pages from the same domain, i.e., protocol-host-port.

JavaScript code can override the defaults by performing limited manipulation of document.domain.

Browser sandboxing enhances this (e.g., in Chrome, separate tabs/frames run in separate processes).

Access control: Cross-Origin Resource Sharing

Modern browsers provide XMLHttpRequest JS API.

- script code sends messages back to the originating site, receives XML (or JSON, HTML, etc).

The same-origin policy became too restricted leading to hacks and workarounds and drafts for improvement including Cross-Origin Resource Sharing (CORS).

CORS uses new HTTP headers to allow responses to indicate that they can be shared, for example:

Access-Control-Allow-Origin: http://www.example.com

or

Access-Control-Allow-Origin: *

See the messaging standard part of HTTP for links to details.

Misconfiguration (A5)

The whole web app stack must be secured (Q. Why?)

- Make sure up to date wrt security patches
- OS, Web/App Server, DBMS, app framework, libs…
- Disable unnecessary features
- Default accounts, demo pages, debug interfaces
- Use minimum privilege settings
- Security settings for frameworks and libraries
- Ensure error handling doesn’t expose info
- Have a repeatable security config process
- An app-specific checklist to work through
- Proactive deployment of updates
- Uniform configuration for devel, QA, deployment

Inherited vulnerabilities (A9)

Big cause of real-life vulnerabilities:

- patch was available but not installed

- Need to stay up-to-date
- Can be harder than might be hoped
  - no unique clearing house (but CVE, NVD v.~good)
  - unsatisfiable compatibility dependencies
- Future: expect to see more automated tools

Review questions

Authorization

- How might you design a web based admin front-end to a database server, supposing that the server provides a collection of databases and its own notion of user and authorization?

Redirection

- What is an open redirect in a web application and why is it undesirable?

Handling sensitive data

Describe three ways that a web application programming flaw might result in stored private user data being leaked.

CSRF

- Draw a picture showing how a CSRF attack might work against an online banking user. What might an attacker be able to do?
- What does a defence mechanism against CSRF need to be able to do?

Deployment

- Give three recommendations for securing the deployment environment (server, database, application framework) for a web app.
This lecture contained material from:

- the OWASP Top 10

as well as papers and other sources cited.