Software Security Course

Web Application Security

Dimitrios A. Glynos
{ daglyn at unipi.gr }

Department of Informatics
University of Piraeus

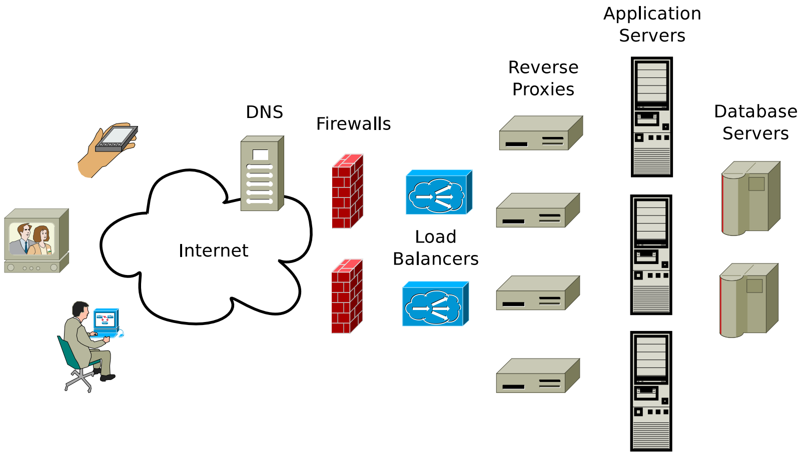
Part I

Web Applications

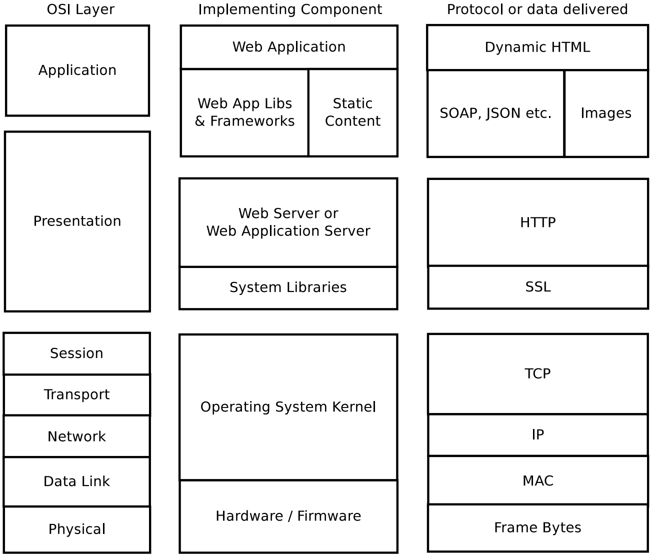
Terminology

- Web server
- Web application
- Web browser (aka browser)
- HTTP Header
- Cookie
- Mobile app
- Web application server
- Database server
- DNS server

Classic Web Application Architecture



Web Application Stack



The protocol stack

- IP for packet routing
  - IP information is processed by routing component of OS kernel
- TCP for reliable data transport
  - TCP data is forwarded by OS kernel to browser / web server / mobile app socket
- SSL for transport confidentiality, data integrity and peer authentication
  - Implemented as library code, used in browser / web server / mobile app
- HTTP for web transactions + content delivery
  - can be library code, used in browser / web server / mobile app
- Application-layer protocols for communication with web services (SOAP, JSON etc.)
  - can be library code, used in
    - JavaScript of browser applications
    - code of web applications + mobile apps

A typical web request

1. User enters `https://domain.net` to browser
2. Browser makes DNS request and resolves `domain.net` to IP 1.1.1.1
3. Browser starts SSL negotiation with the service on port 443 of IP 1.1.1.1
4. Browser verifies the server certificate chain
5. Browser sends HTTP request through the SSL communication channel

   ```
   GET / HTTP/1.1
   Host: domain.net
   ...
   ```

6. Server responds with the content of the page through the SSL channel

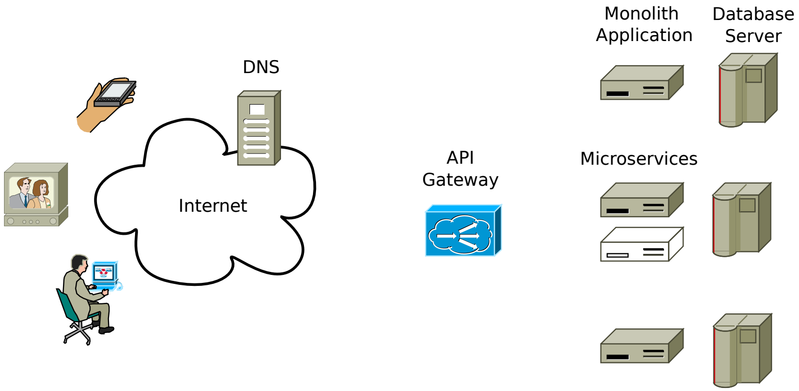
   ```
   HTTP/1.1 200 OK
   Content-Length: 131
   ...
   ```

7. Browser makes further requests for other content that needs to be displayed within the page (images etc.)
8. Browser finishes rendering the page

Sessions

- HTTP is stateless
- But applications require state!
  - The web application keeps a session object to track a user's session
  - Each session object is linked to a Session ID (a random number)
  - The web application passes the Session ID to the client
    - Usually by means of a **cookie** parameter
    - Server header
      ```
      Set-Cookie: PHPSESSID=ec370dcbdc1bc7326c0eae19942e900f;
      expires=Wed, 17 Apr 2024 05:49:35 GMT; Max-Age=86400;
      path=/; domain=localhost; HttpOnly
      ```
    - Browser header
      ```
      Cookie: PHPSESSID=ec370dcbdc1bc7326c0eae19942e900f
      ```
  - Each time the client wishes to perform a transaction within the same session it transmits the relevant Session ID to the web application
- By stealing a user's session ID an attacker would be able to impersonate that user to the server

Modern Web Application Stack



Microservices - Going back to stateless

- A *monolith* is a web application where all functionalities could be delivered by a single instance of the web application
- A *microservice* is an instance that serves a specific functionality of the web application
- Keeping HTTP transactions stateless has its benefits. We could easily scale up, by servicing requests through *multiple instances* of each microservice.
- Each type of *microservice* may communicate with its own database.
- User requests are typically authenticated through a *signed token*
- The API gateway provides a REST API to clients, hiding the microservice interactions

User Authentication

- HTTP provides for
  - Basic authentication
    - username, password is sent to server
    - password is kept hashed on the server
  - Digest Authentication
    - Server keeps client's password in original form
    - Server challenges client with nonce
    - Client sends username, hash(password, nonce)
- Most web applications implement their own authentication
  - Username and password are sent to login page
  - Server checks password against hashed (?) form in database
  - If password is verified, an *authenticated* session object (or a signed token) is created for the user

Note: Automated browser reactions

- Each time the browser visits a domain, it *automatically* sends
  - cookies it holds for this domain
  - Basic and Digest Authentication values (in the Authorization header, more on this later)
- Therefore if a web resource forces a browser to communicate with a domain, the browser *replays* the above information without the user's explicit consent

Authorization

- Check if an incoming request is tied to a session / token with the right privileges before proceeding with the action described in the request.
- Example authorization checks:
  - Is the session ID / token signature valid?
  - Does the session ID belong to a logged in user?
  - Is the session / token connected to an administrative account?
  - Is the user session in the required state (e.g. address details have been verified) for this action to occur?

Web Application Attack Surface

	Routing	**Transport***	**Application**
Client	MAC spoofing DNS spoofing BGP attacks ...	Eavesdropping Session cookie theft MITM attack ...	Browser bug exploitation XSS Clickjacking ...
Server	MAC spoofing DNS spoofing Bad FW config ...	SYN DoS Reflective DoS Padding oracle attack ...	Authentication bypass CSRF SQL injection ...

- **Transport** here covers all the non-routing functionality that is responsible for delivering data *as is* to the browser and web application.

Web Application Security

- Many of our every day processes have moved to a web service implementation
- Web applications are processing the data of millions of users
- There are ongoing attacks to every layer of the web application stack
- Proactive security
  - Development best practices
  - Code Audits + Security Testing
  - Web Application Firewalls
  - Contracts for DoS incident response by ISPs

Part II

OWASP Top 10 Vulnerabilities

OWASP Top 10

- OWASP : Open Web Application Security Project
  - not-for-profit organization focused on improving the security of web applications
  - `https://www.owasp.org`
- OWASP Top 10 project
  - Yearly listing of most critical web application security flaws

OWASP Top 10

1. Injection
2. Broken Authentication and Session Management
3. Cross-Site Scripting (XSS)
4. Insecure Direct Object References
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-Site Request Forgery (CSRF)
9. Using Components with Known Vulnerabilities
10. Unvalidated Redirects and Forwards

Injection

- Untrusted data is sent to an interpreter as part of a command or query
- The hostile data trick the interpreter into executing unintended commands or accessing unauthorized data
  - SQL injection
  - Blind SQL injection
  - PHP file inclusion
  - OS command injection
  - LDAP injection
  - XPATH injection
  - ...

SQL Injection

Example code:

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

// stmt is a previously instantiated Statement object

ResultSet rs = stmt.executeQuery(query);
```

Example trigger:

```
https://domain.net/custView?id=' or '1'='1
```

`rs` will get all data found in customers table!

Injection

Recommendations

- Use prepared statements with parameterized queries
- Use a white list if only specific parameter values should be allowed

Broken Authentication and Session Management

- Leaked (or enumerable) usernames
- Credentials can be changed / guessed via ’recover password’
- Predictable session IDs
- Session IDs in URLs
- Logout not working correctly (e.g. session IDs are not invalidated after timeout)
- Session IDs don’t change after successful login
- Credentials / Session IDs are transmitted in cleartext (no SSL/TLS)

Broken Authentication and Session Management

Example

1. Authenticated user A provides user B with a link to an item which they just bought from an e-shop
2. The link happens to contain the session ID of the first user: `https://domain.net/items?id=CAMERA100&sess=CB233240FACF1423`
3. The session ID enables user B to operate the web application as if she is user A (i.e. make payments, order more items etc.)

Broken Authentication and Session Management

Recommendations

- Authentication forms must not report whether a username is correct or not
- Use a well tested framework for generating session IDs, password recovery tokens etc.
- Store session IDs in cookies
- Protect cookie with ’HttpOnly’ (i.e. not accesible via browser JavaScript) and ’secure’ flag (only transmittable over SSL)
- Change session ID after login
- Invalidate session ID on the server-side after timeout & logout

Cross-Site Scripting (XSS)

Web application presents unsanitized content (think malicious JavaScript) to victim browser

- Stored XSS
- Reflected XSS
- DOM-based XSS

Cross-Site Scripting (XSS)

- Reflected XSS example

```
page += "Listing products of category " +
         request.getParameter("cat");
```

- Trigger: A victim user visits the URL below

```
https://domain.net/list?cat=<script>...</script>
```

Cross-Site Scripting (XSS)

A strategy for defending against XSS

- Are you generating HTML pages or HTML fragments on the server side dynamically?
  - Anything that should appear as text should have HTML special characters be replaced by HTML literals (e.g. '>' becomes '&gt;') on the server side
  - Anything that should be retained in HTML form must first be sanitized through a library like HTMLpurifier on the server side
- Is the web server supplying JSON data to the browser front-end code?
  - If the data is to be presented as text, use a DOM element's `text` node (e.g. `element.text = ...`) to hold the data
  - If the data is to be presented as HTML, it must first be sanitized by a front-end library like DOMPurify
- If the item cannot be adequately controlled (e.g. an uploaded SVG shown inline), you can use CSP rules to enforce which will be considered as valid JavaScript sources in the page

Insecure Direct Object References

- Access to an object is readily provided (to valid sessions)
- Example

```
https://domain.net/user_profile?ID=123124
```

Insecure Direct Object References

Recommendations

- Define access rights for each accessible object
- Use indirect (but unguessable) object references - refer to the object through a UUIDv4 mapping (rather than its database ID)
    - `1940e17d-bb62-4805-b9e0-7b60c539ad9c` $\rightarrow$ database ID 123124
- Use per user (or per session) indirect object references
    - IDs that map only to objects that are valid for the session context

Security Misconfiguration

Example

- An attacker finds a default installation of a CMS
  1. Locates the admin panel at the default location
  2. Logs in with the default credentials

Security Misconfiguration

Recommendations

- distribute applications with safe defaults (e.g. randomly generated passwords for default accounts)
- perform proactive hardening when rolling out systems
- verify security of web application configuration through a security testing procedure

Sensitive Data Exposure

Example

1. Web application stores passwords in unsalted MD5 form
2. Attacker uses SQL injection vulnerability to retrieve the hashes
3. Using a rainbow table the attacker collects the original form of the passwords
4. Attacker now has access to the web platform content
5. May try the same user passwords on other services (mail etc.)
6. May collect sensitive user data

Sensitive Data Exposure

Recommendations

- Create a threat model for the application
- Store only necessary data
- Follow security policies for the storage of sensitive data

Missing Function Level Access Control

- Example: Any authenticated user can add a new user
  `https://domain.net/adm/add_user?name=john&group=admins`
- Recommendation: gather all authorization policy rules in one place (table) and control access to all components of the web application through that policy

Cross-Site Request Forgery (CSRF)

1. Victim authenticates to web application
2. Application relies on authenticated request to fulfil a state-changing user request X
    - In some applications, an authenticated request is a mere presence of the session cookie which is transferred **involuntarily**
3. Attacker tricks the victim in visiting a malicious page that triggers the above request

Cross-Site Request Forgery (CSRF)

Example

- An e-shop changes user address details in the following way

```
GET /shop/address_change?new_address=58
...
Cookie data
...
```

- ID 58 references an address record in the database
- The attacker being a customer of the shop knows his address record id (YY)
- The attacker sends spam emails with the following URL

```
https://domain.net/shop/address_change?new_address=YY
```

- If an authenticated user visits the link his mailing address will automatically be changed!

Cross-Site Request Forgery (CSRF)

Recommendations

- Require an unpredictable, unique per request token
    - Token will be delivered by server (CSRF token)
    - Note: token can be stolen if page has an XSS vulnerability!
- Stateless solution: OWASP signed double submit cookie approach (copy a server-signed secret value held in a cookie, to a header value)
- Lower the exposure: Use the cookie attribute `SameSite=Strict` to allow only content from the valid domain to trigger a session cookie submission

Using Components With Known Vulnerabilities

1. A CMS uses component XYZ that has a known vulnerability
2. An attacker exploits the vulnerability and gains access to the hosting server

Using Components With Known Vulnerabilities

- Track the updates and security advisories on all used components
- Establish an update policy
- Use as little customized code as possible

Unvalidated Redirects and Forwards

- Attacker uses a site’s URL redirection functionality to send users to a malicious website (e.g. for phishing)

```
https://popular.com?redir=http://myphishing.com/page
```

Unvalidated Redirects and Forwards

Recommendations:

- Avoid providing the redirection functionality
- Don’t use parameters in redirect pages
- See if parameter belongs to whitelist (if a parameter is strictly required)