

## Session Management

- Every HTTP request coming from a client is a standalone entity as far as the receiving web server is concerned
- By default, there is no state information retained by the server: the server forgets who we are, where we are, whether we're authenticated etc.
- This is not ideal when a complex web application is involved, it may have multiple users at various locations on the server and it needs to differentiate between them
- We need a means of associating a unique session ID with a user and using it to track her around the application
- This is session management

## Session Management

- Many development frameworks take care of all session management details for free
- Most likely, you will ultimately code real world applications using technologies that look after session management for you
- However, here our aim is to understand the “low level” issues involved in securely implementing session management and so we assume no such framework is available to you

## Session Authentication

- Session management is closely tied to authentication
- A unique session ID is associated with each authenticated user
- Typically, a user is presented with a login form and once authenticated some identifying session token is sent to her browser
- This session token is subsequently included with each HTTP request sent to the server
- The session token should be used to index a server-side database where the real information about that user is stored

## Cookies

- Now a requirement in order to use many online applications
- Originally designed to hold non-security-sensitive data such as date of last visit, user preferences etc.
- There are four cookie types:
  1. Persistent and secure
  2. Persistent and non-secure
  3. Non-persistent and secure
  4. Non-persistent and non-secure
- A persistent cookie is stored in a text file on the client and is valid until its expiry date

## Cookies

- A non-persistent cookie lives in RAM until the browser is closed or the domain that sent the cookie destroys it
- A non-persistent cookie is not tamper proof
- Secure cookies will only be sent over HTTPS (SSL); non-secure cookies will be sent over both HTTP and HTTPS
- Only transport security is ensured: the web server cannot trust secure cookie contents
- Cookies represent a problem where a computer is shared e.g. in an Internet café since, after one user connects to a cookie-setting site, subsequent users may use that user's cookies, bypassing normal authentication requirements

## Cookies

- Cookies are not shared across DNS domains so JavaScript from `http://www.x.com` cannot access/set cookies from `http://www.y.com`
- Cookie submission can be restricted to particular pages on the web server e.g. it is possible to specify that a cookie be sent only to URLs in or below a certain directory
- Cookies are created by two main methods:
  1. Set by the server in HTTP headers
  2. Set by JavaScript sent to client machine
- Browsers can hold up to twenty cookies per domain with a per domain cookie data limit of 4kB

## Cookies

- Once the cookie is set, all HTTP requests to the domain will be accompanied by a copy of the cookie
- Along with a value, a cookie has the following fields:
  1. Domain: The domain that set and can read/write the cookie
  2. Flag: Boolean indicating if all machines within a domain can access the cookie
  3. Path: If `/computing/darragh` then the cookie is only sent with HTTP requests in or below that directory
  4. Secure: Boolean indicating if a HTTPS connection is required before sending the cookie
  5. Expiration: Unix time the cookie expires (no expiration implies non-persistent)
  6. Name: The name of the cookie

## Session Tokens

- Putting a token in a cookie can tie a user to a session
- Clearly, we do not want a session token to be predictable  
e.g. if *A* is authenticated and assigned  $SID = 1$ , then *B*, who is next, should not be assigned  $SID = 2$ : if an attacker knows your session ID then, to the server, she is you
- Tokens can also be passed in URLs and hidden fields
- Tokens should be user-unique, non-predictable and immune to reverse engineering
- The session ID space should be large enough (in terms of length and character set) to prohibit brute force attacks

## Session Management Schemes

- A session token that does not expire allows an attacker unlimited time to brute force it e.g. the “Remember me” option available on many sites may store a cookie on the client machine indefinitely
- To reduce the risk of brute forcing, session tokens should be regenerated regularly by the server e.g. every 20 minutes
- A session token time-out is essential (usually after some period of inactivity)
- Many sites monitor for brute force password attacks but do not check for brute force session ID attacks

## Session Management Schemes

- Could deploy booby-trapped session IDs to detect and thwart such attacks, through IP blocking perhaps
- Critical actions, e.g. money transfer, should always require re-authentication as is common with online banking
- A session token captured in transit leaves that session open to hijack and so HTTPS/SSL must be employed to securely transmit session tokens
- Non-persistent cookies disappear when the browser is closed
- In an Internet café, however, often the browser is never closed so an obvious logout option must be provided that will destroy authenticating cookies

## Session Hijacking

- When an attacker guesses, steals or brute forces an authenticated user's session ID they are free to impersonate that user, hijacking their session
- Such attacks are most easily carried out in an Internet café setting where non-expired, URL-based session IDs may live on in browser history
- Provide a logout mechanism that clears server-side tokens, use non-persistent, short-lived tokens and do not store session IDs in URLs

## Session Hijacking

- Remember, just because someone has a valid session ID does not mean they are who a server thinks they are: always re-authenticate before critical actions
- More information in *A Guide to Building Secure Web Applications and Web Services* available at <http://www.owasp.org>
- Many resources available from OWASP including top ten web security vulnerabilities and tools like WebScarab and WebGoat

## Input Validation

- One of the most common security weaknesses exhibited by web applications is their failure to properly validate client data
- Never trust any data coming from the client
- We look at various client-side data sources and formats, client-side tampering techniques and defence mechanisms
- Specific threats include SQL-injection and HTML-injection (also referred to as cross-site scripting or XSS) attacks

## Input Validation

- We look at the following input sources:
  1. URL query strings
  2. Form fields
  3. Cookies
  4. HTTP headers

## URL Query Strings

- If a HTML form submits user data through the GET method then all form attribute:value pairs appear in the URL and tampering with them is both tempting and trivial
- For example if an online bank allows a user to select one of her accounts and submit her selection using GET, she can trivially try to access others by tampering with the URL
- Data in the URL can not be trusted and actions such as account selection must be authenticated against the session token though a database lookup
- Data indirection is a technique that helps further secure handling data emanating from an un-trustworthy client

## Form Fields

- Form fields come as drop downs, text boxes, radio buttons and hidden fields which are not rendered by the web browser but are submitted to server
- All form fields including hidden ones can be manipulated on the client-side irrespective of the submission method
- For an attacker it is simply a matter of saving and editing the source followed by submission of malicious data
- For the same reason, attempted input validation on the client-side (e.g. setting a text field's `maxlength` attribute) is useless from a security standpoint
- Client-side code is for aesthetics only

## Form Fields

- If server-generated input to the web server is required then two options are available to the web developer:
  1. She can employ a message authentication code (MAC); the MAC is generated by passing a server-side secret along with the server-generated input through a hash function; when the form is submitted the server can verify that the server-generated input it contains hashes to the MAC and has not been tampered with
  2. Alternatively, she may use symmetric encryption; simply encrypt the server-generated input and send it along with the form; when the form is submitted decrypt it before use

## Cookies

- Both persistent and non-persistent cookies can be manipulated on the client-side
- Therefore no authorisation-affecting data should be stored in a cookie, only an identifying token generated within a sparsely populated key space