Secure Programming (06-20010)
Chapter 4: Web Sessions

Christophe Petit

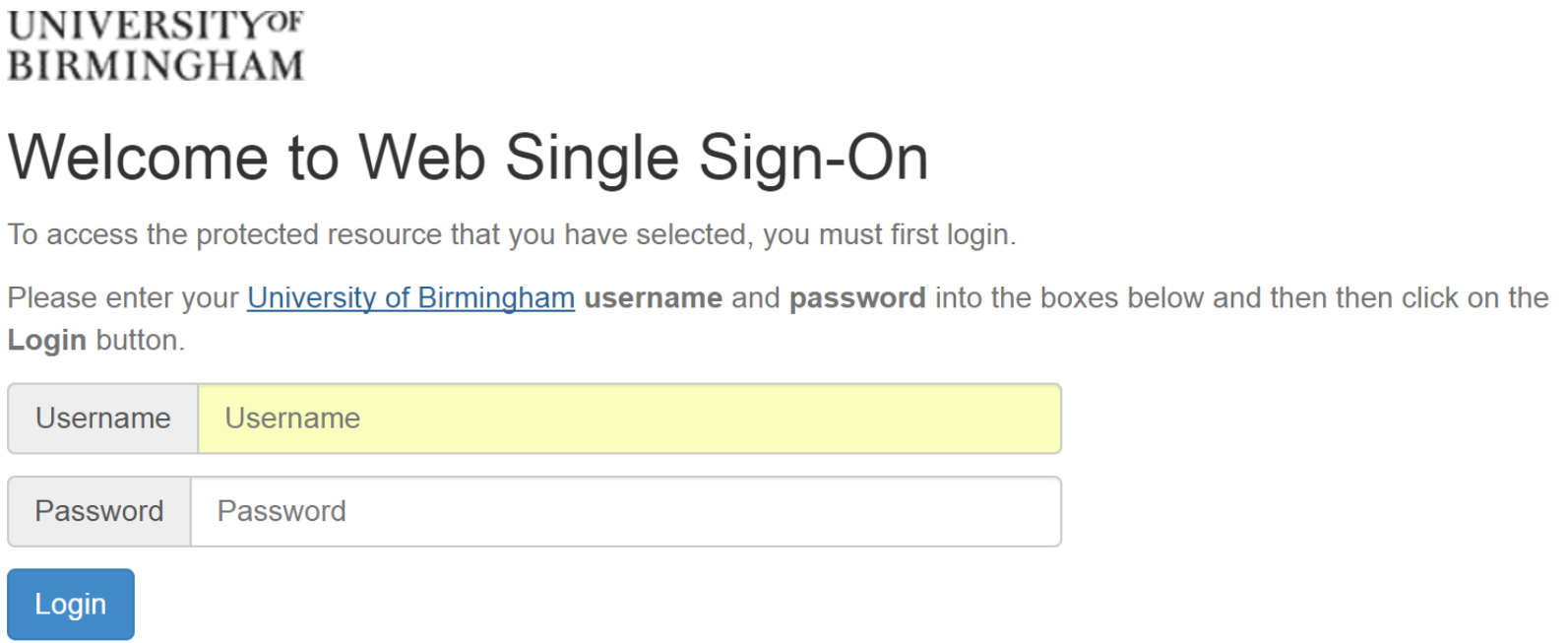
University of Birmingham

Lectures Content (tentative)

1. Introduction
2. General principles
3. Code injection (SQL, XSS, Command)
4. HTTP sessions
5. Unix Access Control Mechanisms
6. Race conditions
7. Integer and buffer overflows
8. Code review

Outline

Web Authentication



OWASP Top 10

...

2. Broken authentication and session management

...

8. Cross-Site Request Forgery (CSRF)

...

Outline

Outline

Outline

Authentication and Authorization

- **Authentication :**
  make sure request was really made by a specific user
- **Authorization :**
  ensure user allowed to execute a certain action

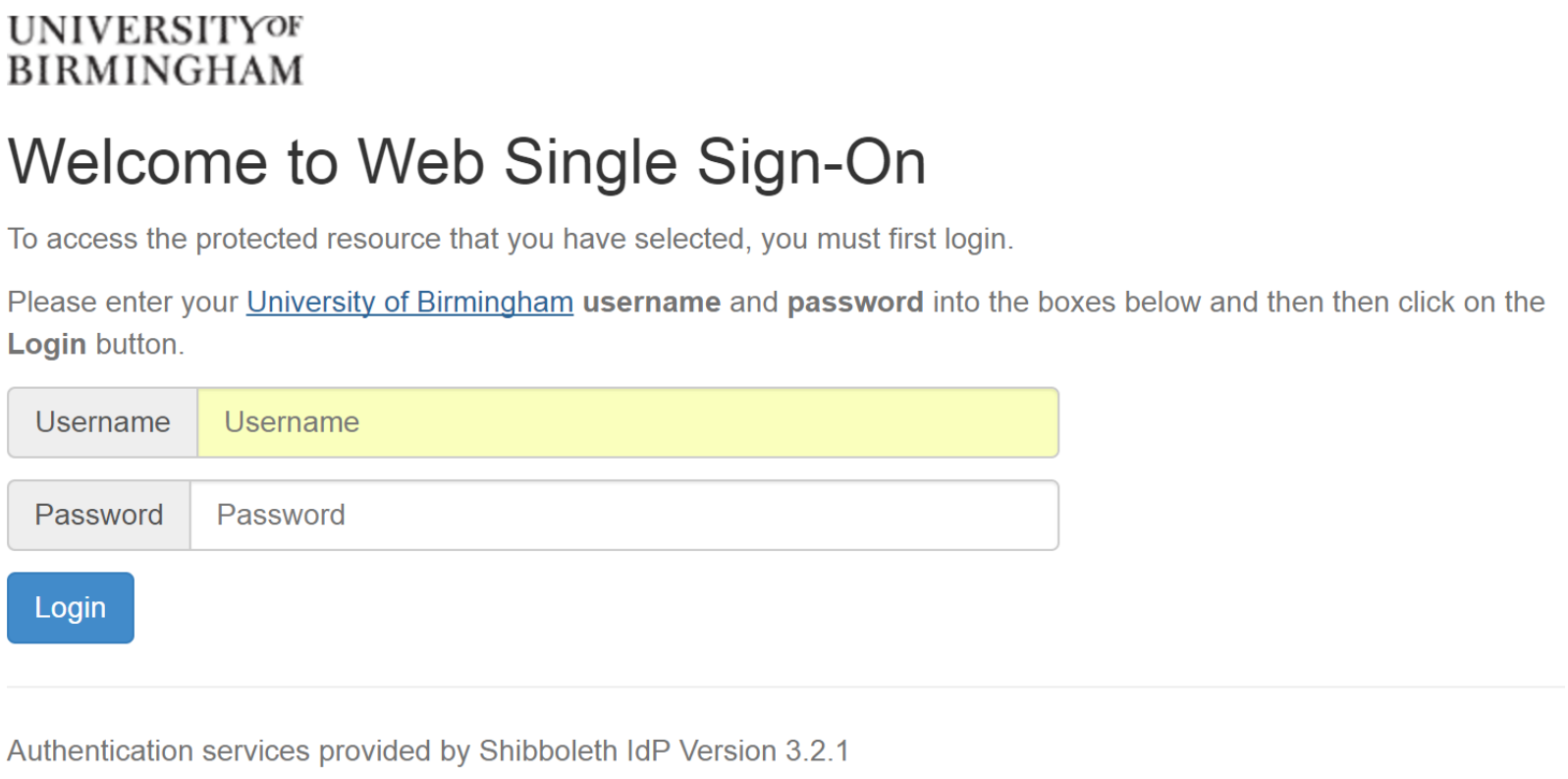
Authentication Methods

- Based on what you know : password
- Based on what you have : physical key or device, Public Key Certificate
- Based on what you are : biometrics
- Or a combination of several methods

Password Authentication Methods

- Basic authentication : send password in clear
- Digest authentication : hash password and send the hash
- Challenge-Response Protocol
  - Client initiates a connection ; Server sends a challenge ; Client responds ; Server checks response
  - Client's response is a cryptographically strong function of challenge and password
- HTTPS (HTTP over TLS)
  - Agree on a common secret session key
  - Encrypt password using the secret session key

Password Authentication Example



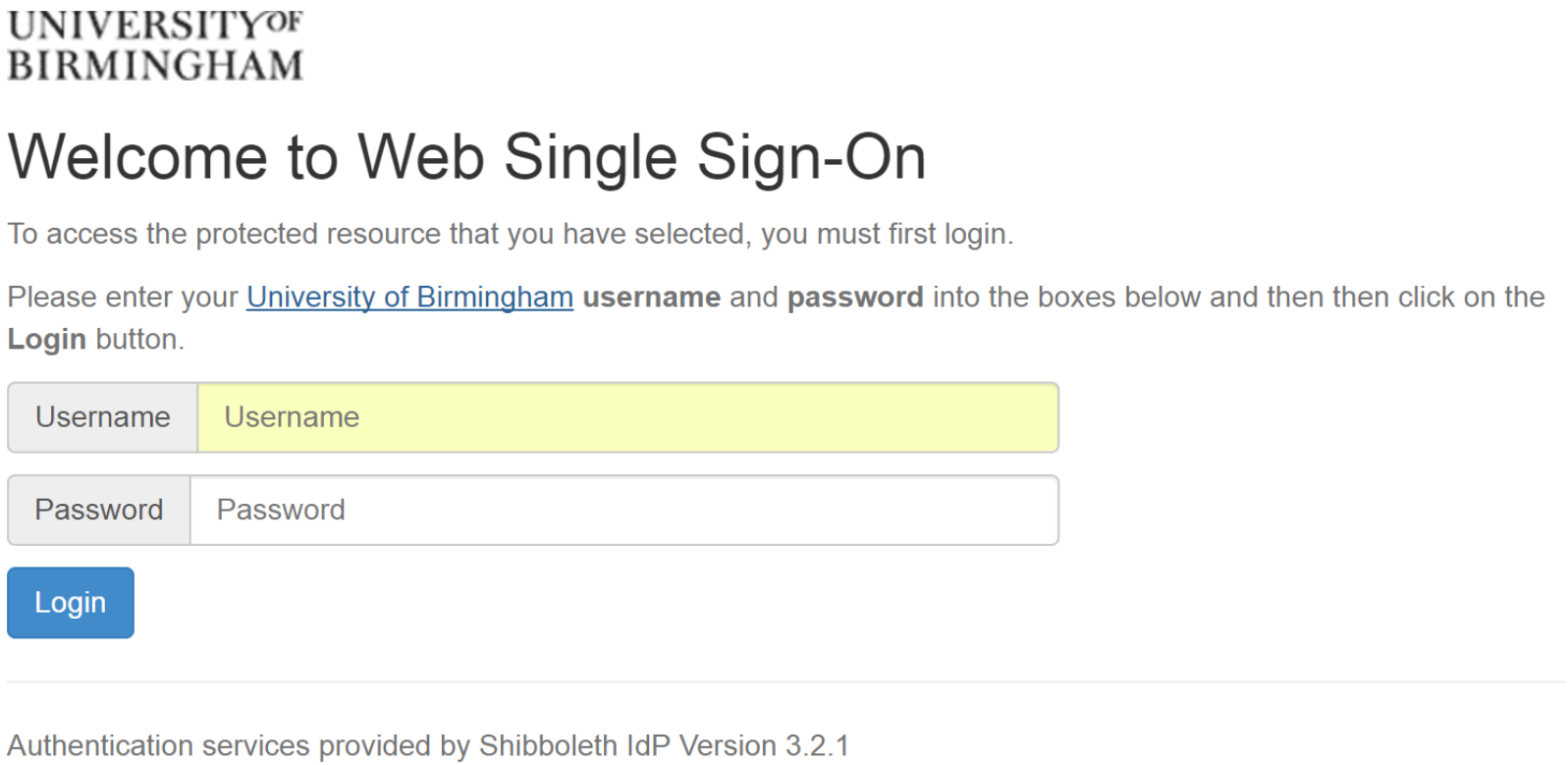
Hypertext Transfer Protocol (HTTP)

- Core data communication protocol for World Wide Web, built over TCP and IP
- Request-response protocol
  - Client connects to server to initiate a request
  - Server responds with answer to request (or some error message)
- Most common methods are GET and POST

Hypertext Transfer Protocol (HTTP)

- GET : request a specified resource
- HEADER : only request header of specified resource
- POST : post data on the specified web resource
- OPTIONS
- PUT
- DELETE
- TRACE
- CONNECT
- ...

UoB Single Sign-On



UoB Single Sign-On : source code

```
<form action="/idp/profile/SAML2/Redirect/SSO?execution=e1s1" method="post">
  <h2 class="form-signin-heading">Welcome to Web Single Sign-On</h2>


                    <p class="help-block">To access the protected resource that you have selected, you must first login.</p>

  <p class="help-block">Please enter your <a href="https://www.birmingham.ac.uk">University of Birmingham</a> <b>username</b> and
<b>password</b> into the boxes below and then then click on the <b>Login</b> button.</p>

  <div class="input-group col-md-8 col-xs-11">
    <span class="input-group-addon">Username</span>
    <input id="j_username" name="j_username" type="text" class="form-control" value="" placeholder="Username" required autofocus />
  </div>

  <p />
  <div class="input-group col-md-8 col-xs-11"">
    <span class="input-group-addon">Password</span>
    <input id="j_password" name="j_password" type="password" class="form-control" placeholder="Password" required>
  </div>


  <p />
  <div class="input-group">
    <button class="btn btn-primary"
            type="submit"
            name="_eventId_proceed" id="j_submit"
            value="Login"
            alt="Login"
            onClick="this.childNodes[0].nodeValue='Logging in, please wait...'" >Login</button>
  </div>

</form>
```

HTTP is Stateless

- From the server point of view, each new HTTP request is independent from previous ones
- Can make "persistent" or "keep-alive" connections, where multiple requests/responses sent with a single connection, but these time out after 1-2minutes
- For most applications we will need longer sessions

Sessions

- Aim : extend the length of a communication
- Idea :
  - Client makes a first connection to server
  - Server gives some session ID (for example, in a cookie) to the client
  - Session ID is then included in every message between server and client

How the Session ID is sent

- URL parameters
- URL arguments on GET requests
- Body arguments on POST requests such as HTML forms
- Cookies in standard HTTP headers
- Proprietary HTTP headers

Authenticated Sessions

- Client's credentials checked when session created, then session ID exchanged as before
- So after initial authentication the client is implicitly authenticated by the session ID
- I.e. : session ID grants same privileges as password
- Need to protect session ID
- Need to set an expiration time

Session ID in HTTP GET parameters

- Will be visible in the URL bar
  `http://mysite/login?jsessionid=123`
- Should the user decide to share a specific site by sharing the URL, he will also share his session
- Multiple sessions in a single browser are possible using different tabs

Session ID in HTTP cookie

- Not visible in the URL bar
- Automatically transmitted by the browser, no link rewriting is required
- Sharing URLs is not a problem
- Can specify a lifetime
- Can also be bound to the lifetime of the browser
- Best and most common option

Cookie HTTP Headers

- SetCookie header :

```
Set-Cookie: <name>=<value>[; <name>=<value>]...
[; expires=<date>][; domain=<domain_name>]
[; path=<some_path>][; secure][; httponly]
```

- Cookie header :

```
Cookie: <name>=<value> [;<name>=<value>]...
```

Cookies : PHP

setcookie

(PHP 4, PHP 5, PHP 7)
setcookie — Send a cookie

Description

```
bool setcookie ( string $name [, string $value = "" [, int $expire = 0 [, string $path = "" [, string $domain = "" [, bool
$secure = false [, bool $httponly = false ]]]]] )
```

setcookie() defines a cookie to be sent along with the rest of the HTTP headers. Like other headers, cookies must be sent *before* any output from your script (this is a protocol restriction). This requires that you place calls to this function prior to any output, including <*html*> and <*head*> tags as well as any whitespace.

Once the cookies have been set, they can be accessed on the next page load with the `$_COOKIE` array. Cookie values may also exist in `$_REQUEST` .

Cookies : JavaScript

Create a Cookie with JavaScript

JavaScript can create, read, and delete cookies with the **document.cookie** property.

With JavaScript, a cookie can be created like this:

```
document.cookie = "username=John Doe";
```

You can also add an expiry date (in UTC time). By default, the cookie is deleted when the browser is closed:

```
document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC";
```

With a path parameter, you can tell the browser what path the cookie belongs to. By default, the cookie belongs to the current page.

```
document.cookie = "username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 UTC; path=/";
```

UoB Single Sign-On : loging out

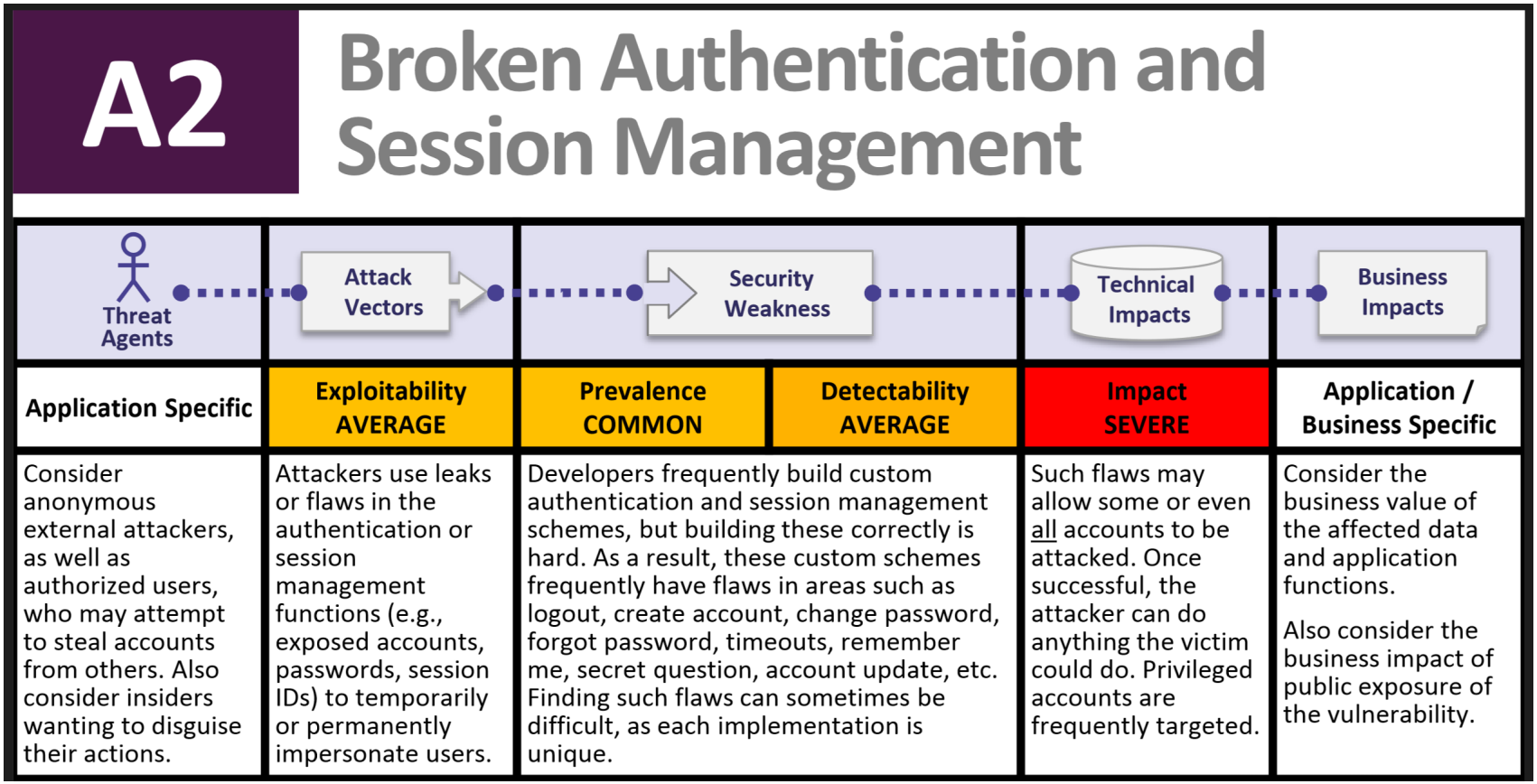
Canvas logout

Please close all browser windows to complete the logout process.

The University uses Shibboleth authentication to provide single sign-on access to Canvas, Library Account and other services. When you log in to one of these services, you are authenticated to access them all without having to log in again. The single sign-on mechanism continues to work even if you log out of a particular service and therefore you must close down the browser to prevent continued access to these services.

Outline

OWASP Top 10


A2
Broken Authentication and Session Management
Threat Agents
Attack Vectors
Security Weakness
Technical Impacts
Business Impacts
Application Specific
Exploitability AVERAGE
Prevalence COMMON
Detectability AVERAGE
Impact SEVERE
Application / Business Specific
Consider anonymous external attackers, as well as authorized users, who may attempt to steal accounts from others. Also consider insiders wanting to disguise their actions.
Attackers use leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to temporarily or permanently impersonate users.
Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, create account, change password, forgot password, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.
Such flaws may allow some or even all accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.
Consider the business value of the affected data and application functions.
Also consider the business impact of public exposure of the vulnerability.

Security Threats

- After initial authentication the client is implicitly authenticated by the session ID
- An attacker might try
  - **Session Hijacking :** steal your session ID and use it
  - **Cross-Site Request Forgery :** incite you to execute some action while you hold the session ID

Session hijacking

- Attacker recovers a session ID, and can then fully impersonate a victim
- Some techniques
  - Session fixation : give user a session ID to use
  - Session hijacking, via packet sniffing
  - Cross-site scripting
  - Malwares
- May target a specific user, or any user

Session hijacking countermeasures

- Do not send session IDs in clear, use encryption
- Use unpredictable IDs
- Make sure a fresh ID is generated at each login
- Check IP address source (not enough)
- Refresh session ID values regularly
- Client side : do not forget to logout

Useful cookie attributes

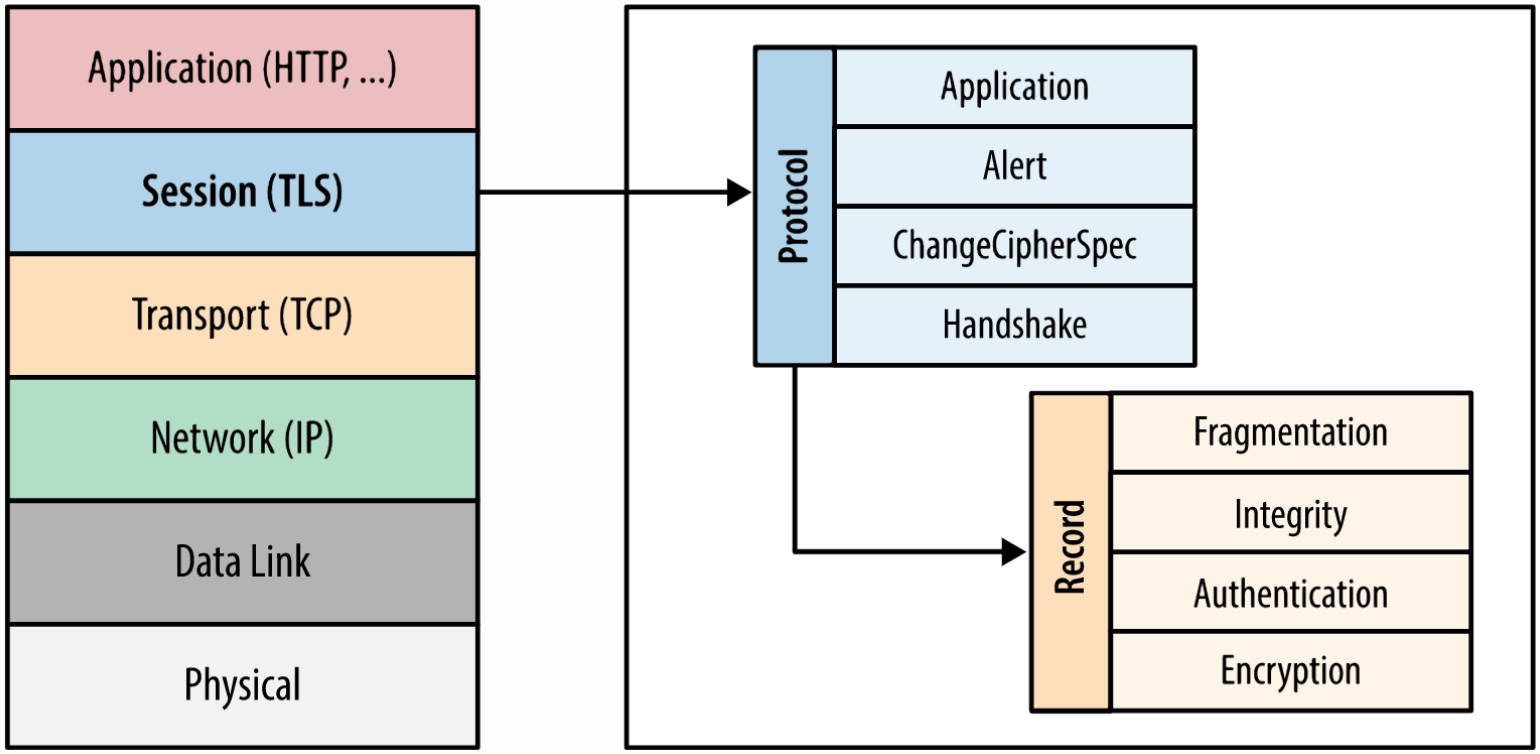
- HTTPonly : cookie cannot be accessed by scripts (helps against XSS attacks)
- SameSite : do not send the cookie with a request coming from another site
- Domain and Path : specify where the cookie can be sent (restrictive by default)
- MaxAge : otherwise cookie expires when browser is closed
- Secure : only send cookie through HTTPS

Outline

HTTPS

- HTTPS is HTTP over SSL/TLS
- TLS = Transport Layer Security
  - Handshake protocol, to agree on communication method and construct shared secret session keys
  - Record protocol, where session keys used to encrypt and authenticate all communications

HTTPS (2)



Picture credit : `hpbn.co/transport-layer-security-tls/`

Some Cryptography concepts

- Encryption algorithm
- Signature algorithm
- Hash function
- Key exchange protocol
- Public Key infrastructure

Crypto tool : encryption algorithm

- In fact three algorithms
  - Key generation algorithm
  - Encryption algorithm
  - Decryption algorithm
- Can be symmetric (same key to encrypt and decrypt) or asymmetric (public key to encrypt, private key to decrypt)
- Examples : AES, RSA, ElGamal
- Intuitively : given encryptions of two chosen messages, attacker should not be able to tell which is which, even if they can request decryptions of other ciphertexts

Crypto tool : signature algorithm

- In fact three algorithms
  - Key generation algorithm : for public,secret key pairs
  - Signature algorithm, using the secret key
  - Decryption algorithm, using the public key
- Examples : RSA, ElGamal
- Intuitively : attacker should not be able to produce a valid signature on any message, even if they can request signatures on other messages

Crypto tool : hash function

- Takes arbitrary large message inputs, and outputs some fixed size message digest
- Examples : SHA, MD5
- Security properties :
  - Cannot be inverted
  - Cannot find two messages with same digest
  - Produce “random-looking” outputs

Crypto tool : key exchange

- Interactive protocol between two parties
- Goal : establish a common *secret* key after exchanging *public* messages
- Example : Diffie-Hellman protocol

Public Key Infrastructure

- Until the seventies all cryptography was symmetric : same key used to decrypt and encrypt
- 1976 : invention of public key cryptography
  - Every party generates their pair of public,secret keys
  - Public key used to encrypt ; secret key used to decrypt
  - Secret key used to sign ; public key used to verify
- Public key certificate authenticates a public key : essentially a signature of the public key by a trusted certification authority

TLS handshake

RFC 5246 TLS August 2008

The TLS Handshake Protocol involves the following steps:

- Exchange hello messages to agree on algorithms, exchange random values, and check for session resumption.

- Exchange the necessary cryptographic parameters to allow the client and server to agree on a premaster secret.

- Exchange certificates and cryptographic information to allow the client and server to authenticate themselves.

- Generate a master secret from the premaster secret and exchanged random values.

- Provide security parameters to the record layer.

- Allow the client and server to verify that their peer has calculated the same security parameters and that the handshake occurred without tampering by an attacker.

TLS handshake

RFC 5246 TLS August 2008

```
      Client                                               Server

      ClientHello                  -------->
                                                      ServerHello
                                                     Certificate*
                                               ServerKeyExchange*
                                              CertificateRequest*
                                   <--------      ServerHelloDone
      Certificate*
      ClientKeyExchange
      CertificateVerify*
      [ChangeCipherSpec]
      Finished                     -------->
                                                [ChangeCipherSpec]
                                   <--------             Finished
      Application Data             <------->     Application Data
```

Figure 1. Message flow for a full handshake

* Indicates optional or situation-dependent messages that are not always sent.

TLS guarantees

- Server authentication
- Optionally, client authentication
- Data integrity
- Data confidentiality
- Optionally, forward security : past communications remain secure even if long term keys are compromised
- Assuming you choose strong cryptographic algorithms, and TLS protocols are sound

TLS : Deployement

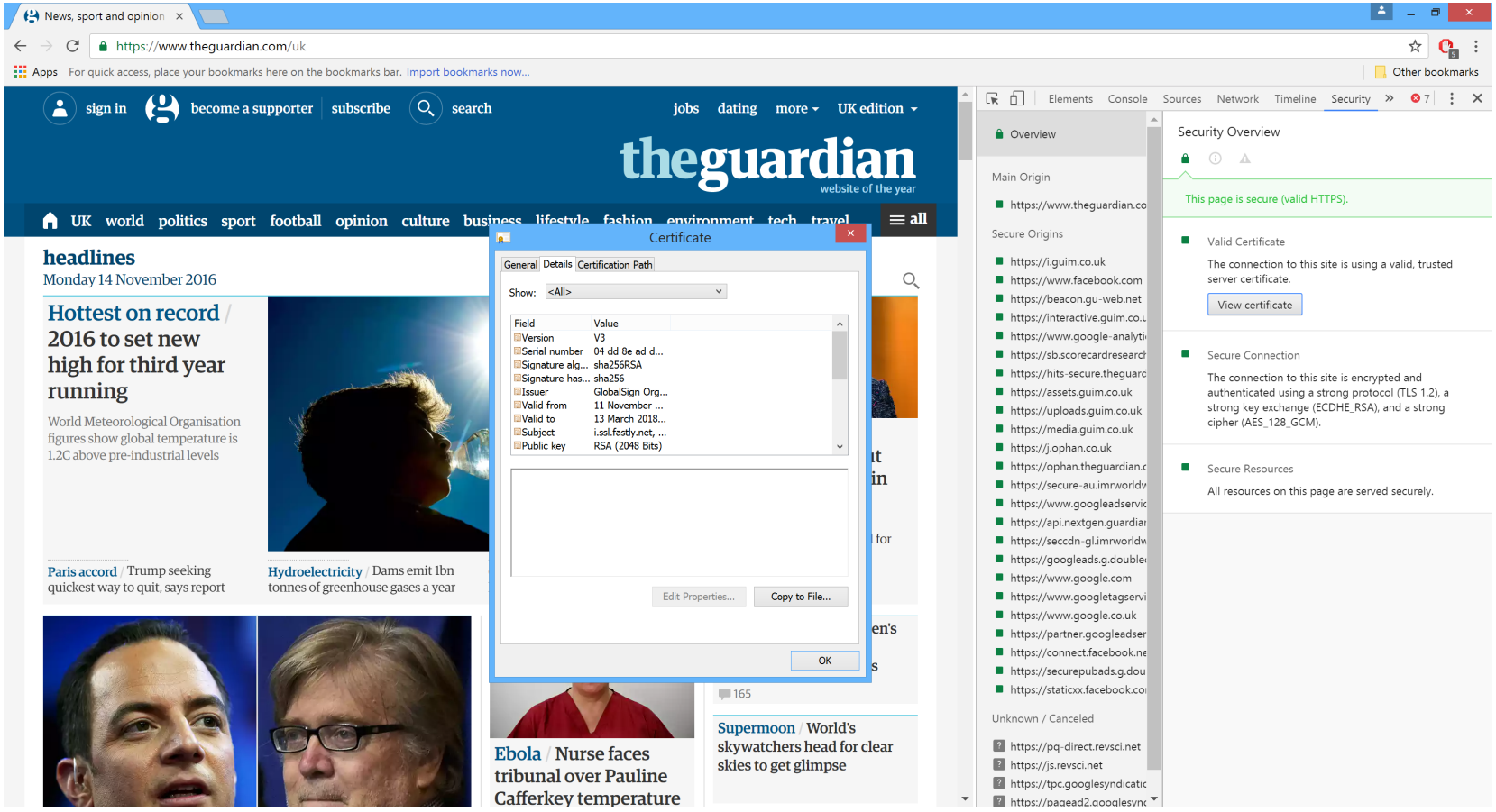
- Generate your private/public key pairs (for example using OpenSSL)
- Buy a certificate from a certificate authority (VeriSign, Entrust, GeoTrust, GoDaddy, . . .)
- Install your keys and certificate on your web server
- Enforce TLS use when needed
  - Use HTTP Strict Transport Security (HSTS)

```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

  - Set your cookies "secure"

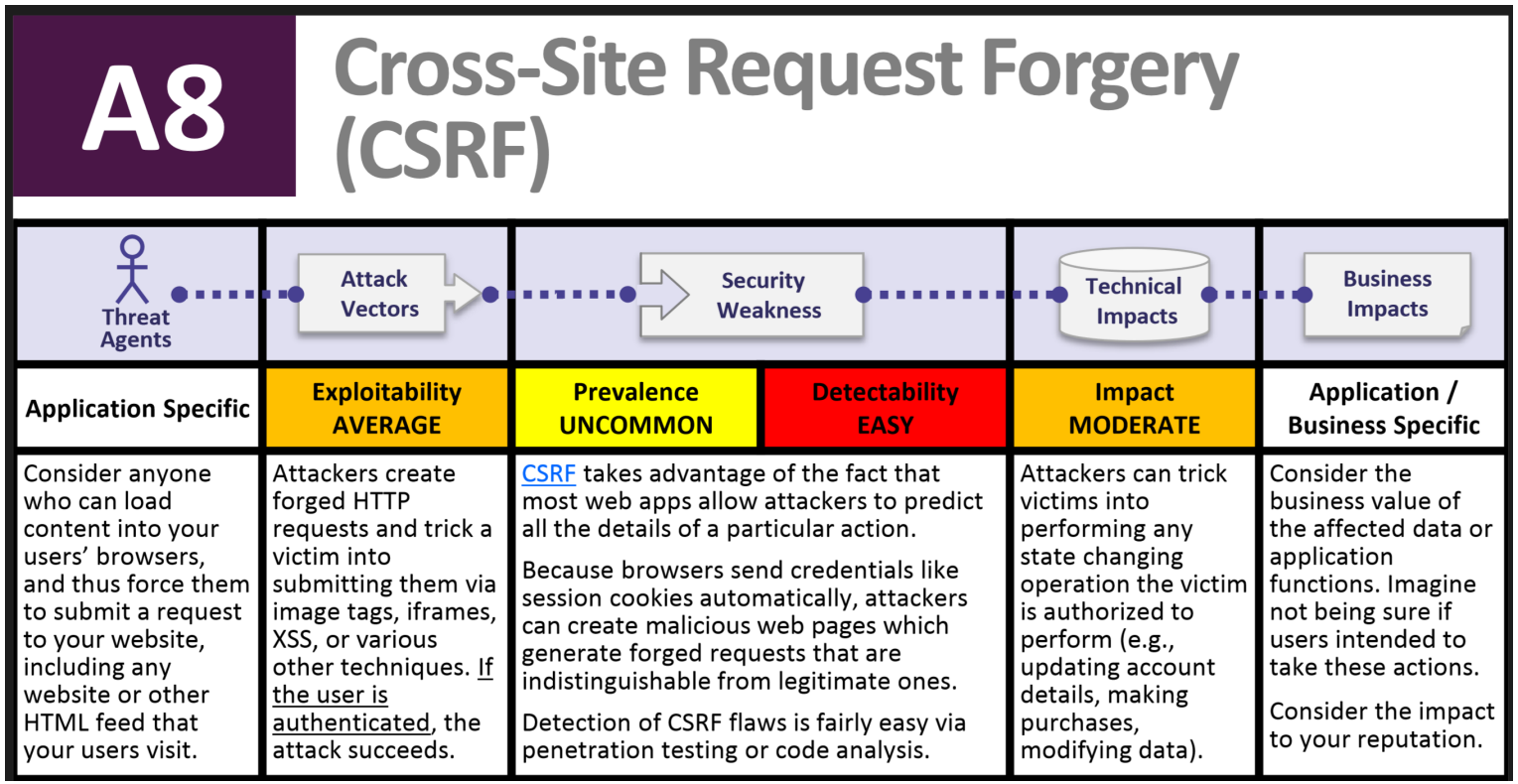
secure
Indicates that the cookie should only be transmitted over a secure HTTPS connection from the client. When set to **TRUE**, the cookie will only be set if a secure connection exists. On the server-side, it's on the programmer to send this kind of cookie only on secure connection (e.g. with respect to `$_SERVER["HTTPS"]`).

HTTPS example



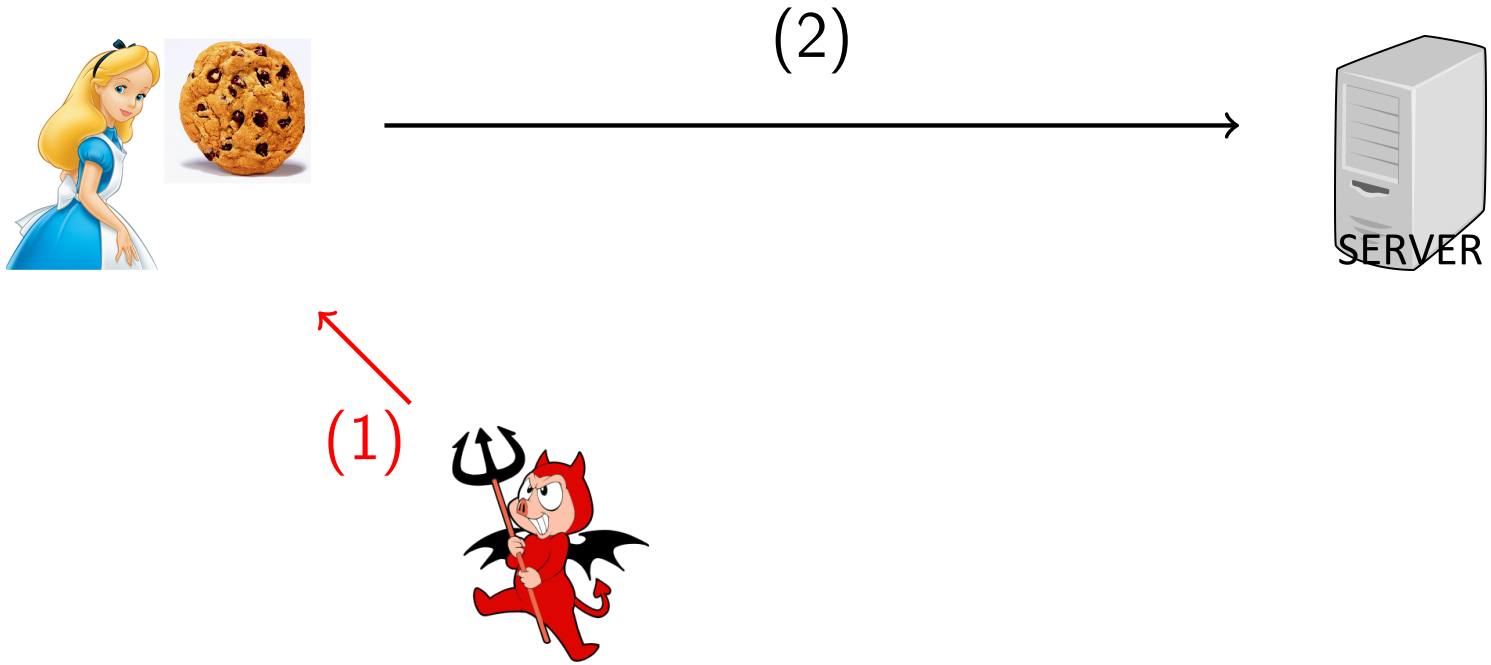
Outline

OWASP Top 10


A8
Cross-Site Request Forgery (CSRF)
Threat Agents
Attack Vectors
Security Weakness
Technical Impacts
Business Impacts
Application Specific
Exploitability AVERAGE
Prevalence UNCOMMON
Detectability EASY
Impact MODERATE
Application / Business Specific
Consider anyone who can load content into your users' browsers, and thus force them to submit a request to your website, including any website or other HTML feed that your users visit.
Attackers create forged HTTP requests and trick a victim into submitting them via image tags, iframes, XSS, or various other techniques. If the user is authenticated, the attack succeeds.
CSRF takes advantage of the fact that most web apps allow attackers to predict all the details of a particular action.
Because browsers send credentials like session cookies automatically, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones.
Detection of CSRF flaws is fairly easy via penetration testing or code analysis.
Attackers can trick victims into performing any state changing operation the victim is authorized to perform (e.g., updating account details, making purchases, modifying data).
Consider the business value of the affected data or application functions. Imagine not being sure if users intended to take these actions.
Consider the impact to your reputation.

Outline

Cross-Site Request Forgery (CSRF)



- Client has previously received a session ID from server
- Attacker tricks Client into making some request to server

Comparison with XSS

- Cross-site scripting exploits the trust a user has for a particular site
- Cross-Site Request Forgery exploits the trust that a site has in a user's browser

A web banking scenario

- A client connects to a server, say a banking application
- Server gives session cookie to Client
- To make a payment, Client visits some page from Server
  1. Client's browser makes HTTP request to get the page ; it also automatically adds the cookie
  2. Page is read by Client's browser ; it contains some form
  3. Client fills in the form and presses submit button
  4. Browser issues another HTTP request to Server ; it also automatically adds the cookie
  5. On receiving a request Server checks the cookie and proceeds with the payment

A web banking scenario

- To make a payment, Client visits some page from server
  1. ...
  5. On receiving a request Server checks the cookie and proceeds with the payment

- Two important observations
  - Server does not check whether Steps 1-4 took place before executing Step 5
  - When browser makes an HTTP request to the Server, browser automatically adds the cookie to the request

Cross-Site Request Forgery (CSRF)

- Client has previously received a cookie from Server
- Attacker sends Client some link, or some script embedded in a webpage, that will issue a request to Server
- Client's browser sends the request to Server ;
  it also automatically adds the cookie
- Server checks the cookie and then executes the request
- Client not aware of what is happening, in fact vulnerable Server page may never be loaded by their browser !

GET example (OWASP)

- Alice wants to send £100 to Bob using a vulnerable application *bank.com*
- She connects to the website, and she now has an authentication cookie
- A normal request would look like

  GET `http://bank.com/transfer.do?acct=BOB&amount=100` HTTP/1.1

GET example (OWASP)

- Attacker sends Alice an email including

```
<a href="http://bank.com/transfer.do?
   acct=EVIL&amount=100000">View my
   Pictures!</a>
```

- By clicking the link Alice effectively makes the request

- Alternatively, attacker includes a fake image

```
<img src="http://bank.com/transfer.do?
   acct=EVIL&amount=100000" width="0"
   height="0" border="0">
```

- Alice won't see image but her browser will nevertheless send the request to the bank

POST example (OWASP)

- A normal request would look like
  POST `http://bank.com/transfer.do` HTTP/1.1
  acct=BOB&amount=100
- Attacker makes Alice visit a page including

```
<body onload="document.forms[0].submit()">
<form action="http://bank.com/transfer.do" method="POST
    ">
<input type="hidden" name="acct" value="EVIL"/>
<input type="hidden" name="amount" value="100000"/>
</form>
```

Impact

- Money transfer, item purchase, password change, etc, for user accounts
- Full application compromise for admin accounts
- Favourite targets are community and payment websites
- Attack can be exercised behind a firewall
- Webmails particularly vulnerable : you will be logged in when you receive the malicious email

Cross-Site Request Forgery (OWASP)

- CSRF relies on the following :
  1. Browsers automatically send session information
  2. Attacker can predict some valid application URLs
  3. Session management only relies on information known by the browser
  4. Some HTML tags (for example IMG) cause automatic access to an HTTP(S) resource
- Points 1, 2, 3 essential for the vulnerability to exist
- Point 4 facilitates exploitation but not strictly required

Outline

Remember : web banking scenario

- To make a payment, Client visits some page from server
  1. ...
  5. On receiving a request Server checks the cookie and proceeds with the payment

- Two important observations
  - Server does not check whether Steps 1-4 took place before executing Step 5
  - When browser makes an HTTP request to the Server, browser automatically adds the cookie to the request

Prevention

- Server side : check that Steps 1-4 indeed took place
  - Same origin policy
  - Anti-CSRF Tokens
  - Challenge-Response mechanisms
- Client side : do not relay cross-site requests
  - Use SameSite attribute
- Prevent Cross-Site Scripting
- Educate users

Same origin policy

- Idea : ignore a request if it originates from a different site
- Check Origin and/or Referer HTTP headers to know where the request originated from
  - These are protected headers, can only be set by browsers
  - Origin is mandatory with HTTPS
  - Not mandatory otherwise
- Deny by default : reject request if headers are not there

Anti-CSRF tokens

- Include some randomly generated token in your form
- Token sent to legitimate Client with the page
- Ask Client to submit the token with the form
- Only accept requests with valid token

- Previous CSRF attacks defeated since Attacker does not know the token
- Server needs to store and manage the tokens
- Check OWASP's CSRFGuard and CSRFProtector

Challenge-Response Mechanisms

- Before the request is executed send a challenge to user, who has to respond with a valid answer
  - Re-authentication, CAPTCHA, one-time token
- Very good protection, but can affect user experience
- Keep only for sensitive requests

Prevent Cross-Site Scripting

- Cross-Site Scripting is not necessary for CSRF to work
- But XSS vulnerability can be used to defeat token and referer/origin based defenses
- XSS cannot defeat challenge-response defenses

Safety CSRF Tips for Users (OWASP)

- Logoff immediately after using a Web application
- Do not allow your browser to save username/passwords
- Use different browsers to access sensitive applications and to surf internet freely
- Plugins such as No-Script make POST based CSRF vulnerabilities difficult to exploit
- Mails and newsreaders integrated in browsers pose additional risks

SameSite cookie attribute

- Forbid browser to send cookies with cross-site requests
- Can be Strict or Lax
  - Strict mode prevents following a Facebook link from another webpage when you are logged in
- See `https://scotthelme.co.uk/csrf-is-dead/`

Outline

Summary

- HTTP is a stateless protocol, so session mechanisms are built on top of it
- Cookies must be protected to prevent Session hijacking ; otherwise attacker temporarly gets Client's provileges
- Cross-Site Request Forgery attacks (CSRF) exploit trust of a website on a client
- Main CSRF prevention : Same Origin Policies and Tokens

References and Acknowledgements

- Relevant OWASP and CWE articles
- `https://scotthelme.co.uk/csrf-is-dead/`
- While preparing these slides I also used teaching material developed by Erik Tew at the University of Birmingham (kindly provided to me). Some of my slides are heavily inspired from his (but blame me for any errors !)