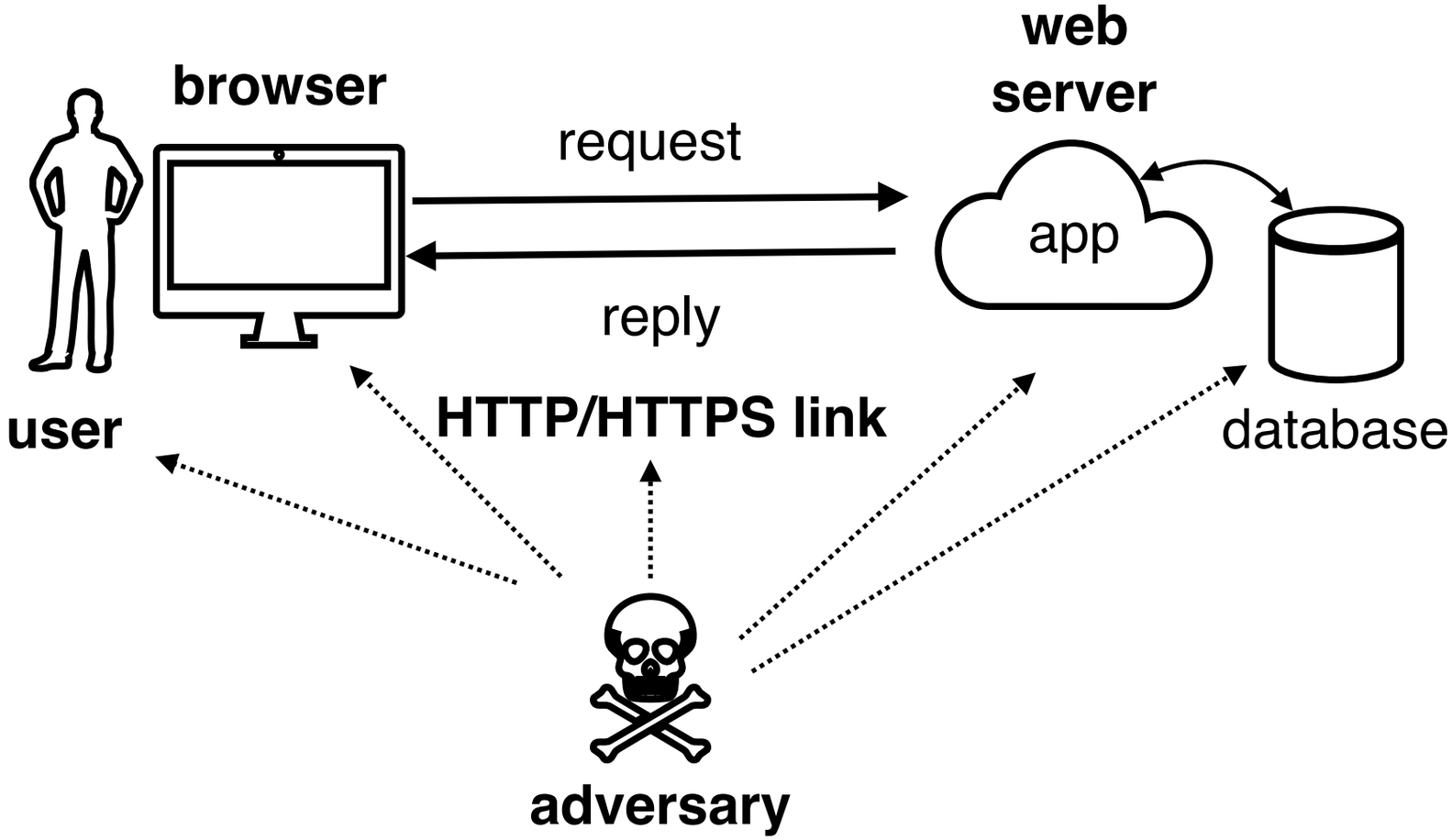
Web application vulnerabilities

Questões de Segurança em Engenharia de Software (QSES)
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, edrdo@dcc.fc.up.pt

Introduction

Web applications



- Basic aspects
  - Browser and server communicate through HTTP or HTTPS (HTTP: plain-text, HTTPS = HTTP over encrypted TLS connection)
  - Server-side features: Dynamic HTML generation, business logic, persistence layer (e.g., SQL database)
  - Client-side: renders HTML, executes scripts.
- **Q:** What may an adversary do?

HTTP requests

browser → **web server**

GET request — arguments are encoded in URL

```
GET http://127.0.0.1:8081/vulnerabilities/xss_r/?
  name=Eduardo&user_token=64e7a89cf687e5b53c4115f899ec438b HTTP/1.1
Proxy-Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_4) AppleWebKit/537.36
   (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/
   apng,*/*;q=0.8
Referer: http://127.0.0.1:8081/vulnerabilities/xss_r/
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Cookie: PHPSESSID=lhtuupa7c6jl5v3ekdjp63nv56; security=impossible
Host: 127.0.0.1:8081
```

Browser id

Referrer URL

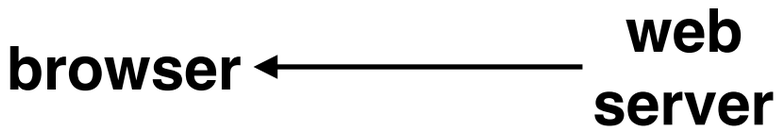
Cookie stored in browser

POST request

```
POST http://127.0.0.1:8081/login.php HTTP/1.1
Content-Length: 88
. . .
Referer: http://127.0.0.1:8081/login.php
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Cookie: PHPSESSID=lhtuupa7c6jl5v3ekdjp63nv56; security=impossible
Host: 127.0.0.1:8081

username=admin&password=password&Login=Login&user_token=ddafd9974dfb2b686c99fa1b36e2
823d
```

POST request — arguments are encoded in the request body

HTTP replies



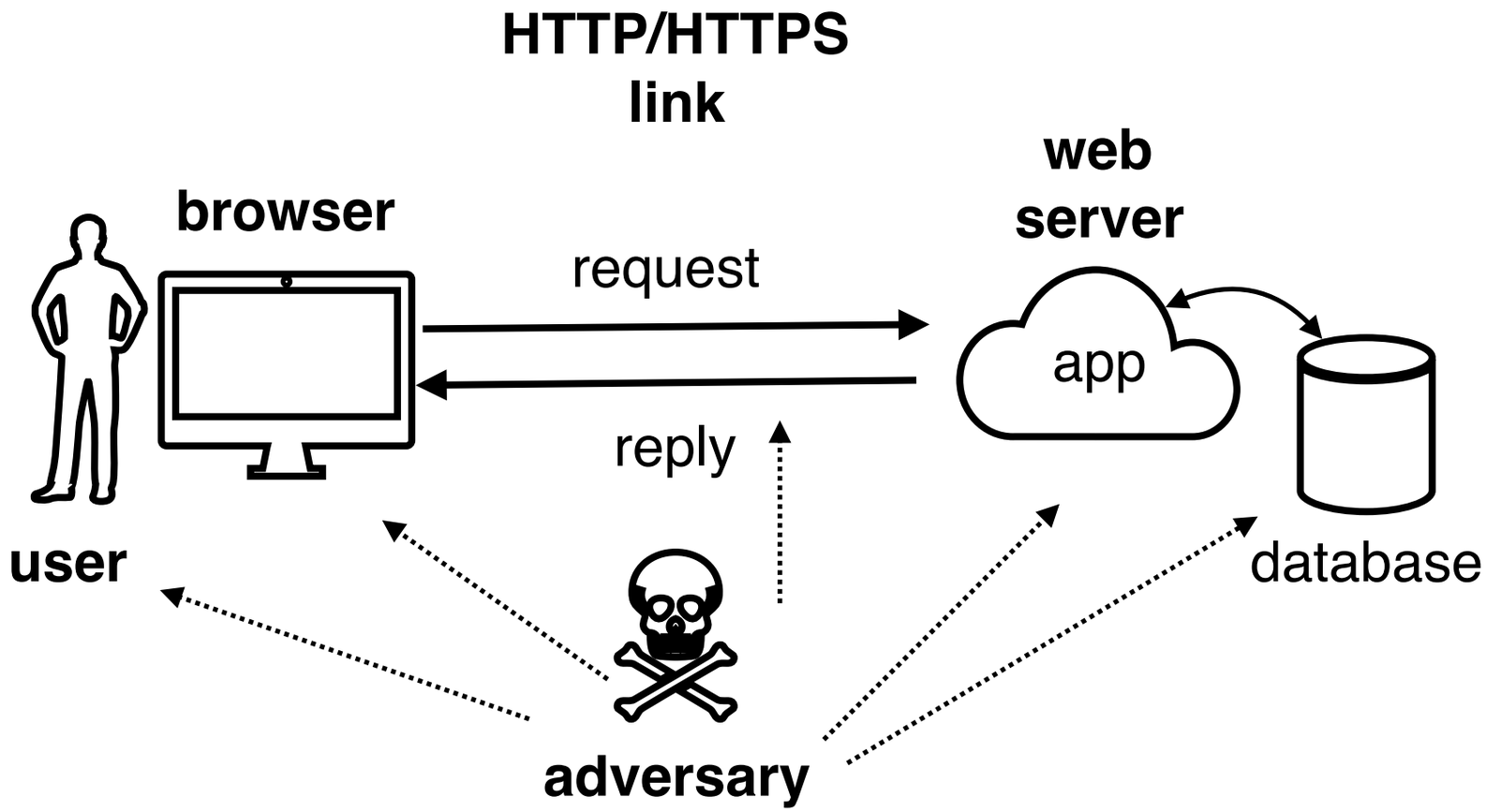
HTTP version id + status code + info message

```
HTTP/1.1 200 OK
Date: Mon, 15 Oct 2018 14:44:24 GMT
Server: Apache/2.4.10 (Debian)
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
Vary: Accept-Encoding
Content-Length: 1567
Content-Type: text/html;charset=utf-8

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-strict.dtd">

<html >
. . .
</html>
```

Content description

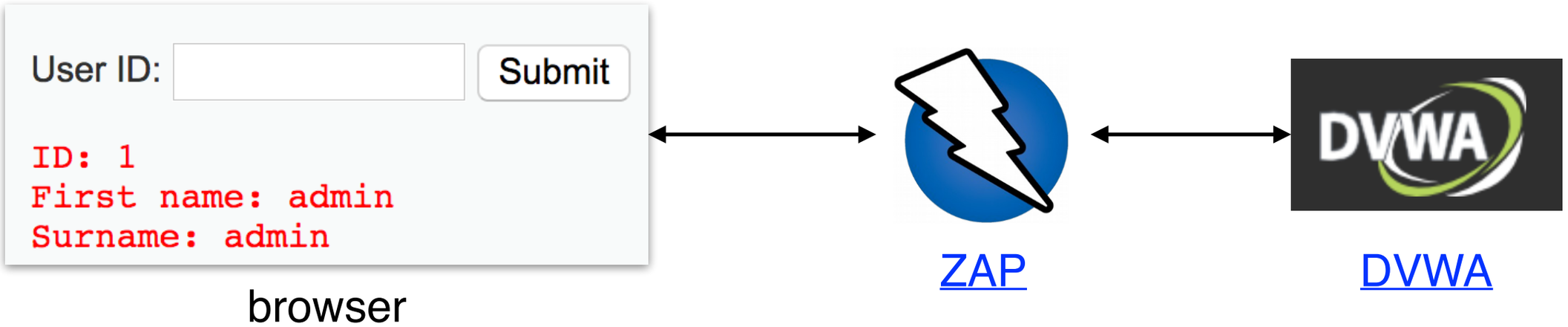
data (HTML in this case

Vulnerabilities



- We will look at the following types of vulnerability that are web-application specific (this list is far from exhaustive):
  - Cookie related vulnerabilities
  - Cross-Site Request Forgery (CSRF)
  - Cross-Site Scripting (XSS)
- Injection vulnerabilities discussed before (like SQLi or OS command injection) are also very common in web applications, but not specific to them.

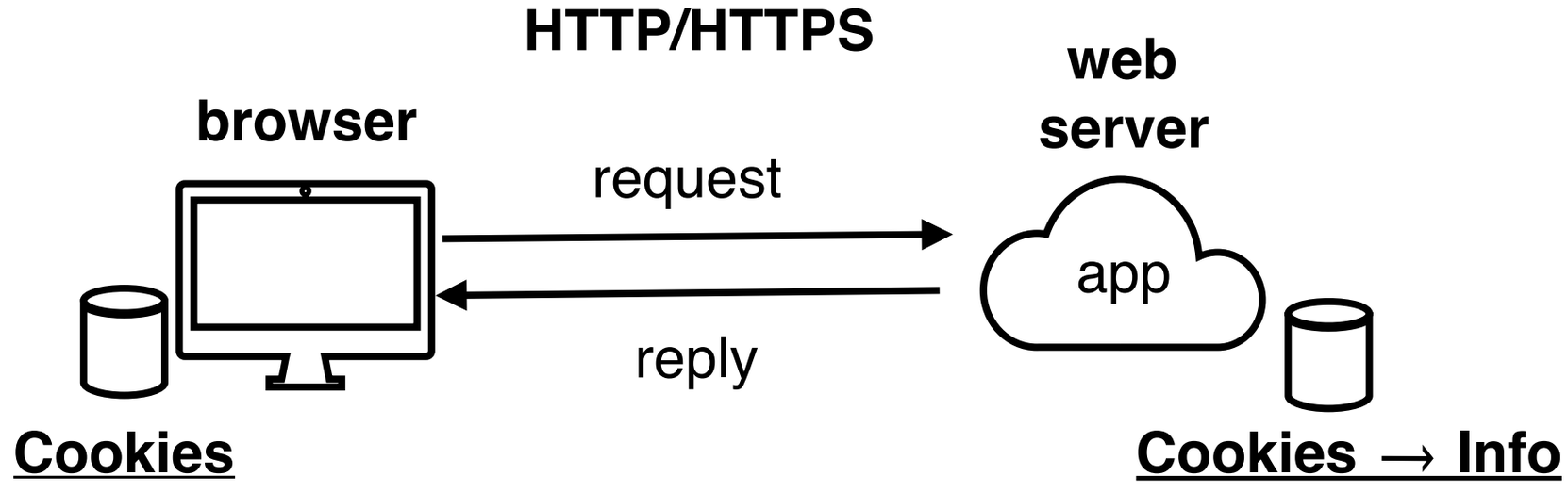
Running example



- To illustrate some of the concepts and vulnerabilities we will make use of:
  - DVWA again as an example of vulnerable application and different strategies for enforcing security, and the SonarCloud analysis of the DVWA source code
  - ZAP: the OWASP Zed Attack Proxy to inspect HTTP traffic, passively probe for vulnerabilities, or for active pen-testing attacks.

Cookie-related vulnerabilities

Cookies



- **HTTP is stateless**
  - Session = set of request-reply interactions possibly using the same connection (or not)
  - HTTP does not maintain state however => it merely echoes request headers issued by browser/web server.
- State is typically maintained through **cookies**. These are issued by the web server and stored by the browser. Some common uses are:
  - **authentication cookies, also called session cookies** — identifying logged-on users and the corresponding session
  - **tracking cookies** — used to track users as they navigate the web
  - maintaining info regarding user interactions (e.g. shopping baskets)
- Other mechanisms such as the URL query string or hidden form fields may be used to maintain state, but are typically less convenient and prone to ad-hoc logic.

Cookie setup

Cookie to set

Name: `PHPSESSID`
Value: `eib49g3fajovj3165e0uvv2gn1`
Attributes: `path=/`

```
HTTP/1.1 302 Found
Date: Mon, 15 Oct 2018 14:44:24 GMT
Server: Apache/2.4.10 (Debian)
Set-Cookie: PHPSESSID=eib49g3fajovj3165e0uvv2gn1; path=/
. . .
```

subsequent GET request includes cookie

```
GET http://localhost:8081/login.php HTTP/1.1
. . .
Cookie: PHPSESSID=eib49g3fajovj3165e0uvv2gn1; security=impossible
Host: localhost:8081
```

- **Server**: emits cookie, a key-value pair with possible additional atributes.
- **Client**: stores it and transmits it in subsequent connections to the same server (`Cookie` header in the fragment above).
- In the example: key = `PHPSESSID` , value = `eib49g3fajovj3165e0uvv2gn1` and an attributes is set : path = /

Cookie definition

```
Set-Cookie: 1P_JAR=2018-10-15-15;
            Expires=Wed, 14-Nov-2018 15:42:07 GMT;
            Path=/;
            Domain=.google.pt
```

- Server in this case indicates that the cookie
  - is **named** 1P_JAR and has **value** 2018-10-15-15
  - has an **expiration time** (MaxAge attribute can also be used):
    - ✦ it is set using Expires “14-Nov-2018 15:42:07 GMT” — **the cookie will not be deleted once the browser exits** — so this is a **persistent cookie** ;
    - ✦ **cookies without expiration time are deleted once a browser session is terminated and are called session cookies**
    - ✦ servers can send an expiration time in the past to delete the cookie
  - should be sent for any requests specified by the **domain/path** setting, i.e., “.google.pt" + “/“ in this case. This will match “<ANY>.google.pt/<ANY>
- Reference — RFC 2965 (HTTP State Management Mechanism)

Cookie definition (cont.)

```
Set-Cookie: Name=Value; … etc … ;
            Secure; HttpOnly
```

- The special **Secure** and **HttpOnly** attributes have no associated values.
- **Secure**: forces cookie to be transmitted using only secure encrypted channels, i.e. HTTPS is allowed but HTTP is not.
- **HttpOnly**: does not expose the cookie other than through HTTP(S) interactions. In particular, this means that Javascript code in web pages cannot access the cookie through the Document Object Model (DOM).

Attack surface

- Cookies may :
  - … be predictable (e.g. session ids)
  - … contain/leak confidential data (e.g. passwords)
  - … be persistent with a large expiration time
  - … be read and modified by Javascript if HttpOnly flag is not set
  - … may be intercepted by a “man-in-the-middle” on a HTTP connection if the secure flag is not set
- This may help a number of attacks
  - Session hi-jacking (next)
  - Cross-site request forgery attacks (also discussed in this class)

Session hijacking

- Basic scenario:
  - Adversary steals an authentication cookie, with a long (enough) expiration time.
  - Adversary may then impersonate a legitimate user (spoofing).
  - … and materialize other threats afterwards.
- How can the cookie be stolen?
  - A human may access your PC and the web browser data.
  - MITM attacks are feasible if cookie is sent over plain HTTP (allowed when secure flag not set). More complex MIM attacks are also possible, e.g. DNS cache poisoning may allow adversary to impersonates host of interest, letting a browser send cookies for the site's domain willingly.
  - By exploiting vulnerabilities on the browser or server side that leak the cookie information (e.g. injection of Javascript code that reads the cookie value).
  - By predicting the actual value of the cookie. An authentication cookie should be produced by a high quality random number generator and sufficiently long.

Session hijacking story — Twitter

- In 2013, Twitter used an authentication cookie that facilitated session hijacking:
  - The cookie persisted even after user logged out and did not expire.
  - So the same cookie value was used in every session for the same user.
  - More details ; other similar vulnerabilities here and here
- If an adversary stole an authentication cookie once, it could impersonate the user at stake *indefinitely*.
- Vulnerability instantiates CWE-539 — Information Exposure Through Persistent Cookies and CWE-384 - Session Fixation
- **Defenses**
  - Cookie should be deleted after user logs off. This deals with session fixation.
  - Regarding persistency, application may use session cookies (non-persistent). This compromises usability though, since user must log in again after closing the browser.
  - Limited persistency is a common compromise by having an expiration time set.

Session hijacking story — Firesheep

- Firesheep (2010)
  - A Firefox extension that sniffs traffic in WiFi networks (in particular public WiFi networks!)
  - Vulnerability classes explored - CWE-614: “Sensitive Cookie in HTTPS Session Without 'Secure' Attribute”
  - Login typically encrypted using HTTPS, but authentication cookie subsequently transmitted over plain HTTP. Session hijacking could then proceed at will for Facebook, Twitter, ….
- Preventions — by design:
  - **Security-sensitive cookies should be set with the Secure attribute**; they should not allowed to be transmitted over HTTP.
  - In many cases, such as for session id cookies, **the HttpOnly attribute should also be used** to avoid data leaks through the DOM.
  - **Sites should use HTTPS uniformly**.
- Some mitigations
  - Extensions like HTTPS Everywhere may be used to transforms HTTP onto HTTPS requests.
  - VPNs generally protect against sniffing/lack of encryption (FireSheep illustrates well that one should generally beware of public WiFi networks).

Some CWE vulnerability classes …

- CWE-1004: Sensitive Cookie Without 'HttpOnly' Flag
  - Cookies without HttpOnly flag are accessible by scripts in a web page through the DOM.
  - Cookies with HttpOnly flag are only handled by the browser.
- CWE-315: Cleartext Storage of Sensitive Information in a Cookie
  - in particular usernames and passwords !
- CWE-565: Reliance on Cookies without Validation and Integrity Checking

Detecting possible cookie vulnerabilities — static analysis

```
if( isset( $_POST[ 'id' ] ) ) {
        setcookie( 'id', $_POST[ 'id' ]);
```

Make sure that this cookie is written safely. See Rule — 4 years ago — L12

Security Hotspot — To Review — Not assigned — cwe, owasp-a3, sans-top25-porous

Make sure creating this cookie without the "secure" flag is safe here. See Rule — 4 years ago — L12

Security Hotspot — To Review — Not assigned — cwe, owasp-a2, owasp-a3, sans-top2…

- Check the SonarCloud issues in detail here

Detecting possible cookie vulnerabilities — pen-testing

Cookie No HttpOnly Flag

URL: https://localhost:9443/
Risk: Low
Confidence: Medium
Parameter: JSESSIONID
Attack:
Evidence: Set-Cookie: JS

```
HTTP/1.1 200 OK
Date: Wed, 17 Oct 2018 14:24:54 GMT
Content-Type: text/html;charset=utf-8
Set-Cookie: JSESSIONID=node0z0uhkvefshdt1qbg1ruyraltz0.node0;Path=/;Secu
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Content-Length: 7702
Server: Jetty(9.4.8.v20171121)
```

Cookie Without Secure Flag

URL: https://localhost:9443/LoginValidator
Risk: Low
Confidence: Medium
Parameter: privilege
Attack:
Evidence: Set-Cookie: privilege
CWE ID: 614

```
Date: Wed, 17 Oct 2018 14:27:08 GMT
Set-Cookie: privilege=user;HttpOnly
Expires: Thu, 01 Jan 1970 00:00:00 GMT
```

Secure cookie programming

```
Cookie privilege = new Cookie(key, value);
privilege.setHttpOnly(true);
privilege.setSecure(true);
privilege.setMaxAge(3600);
response.addCookie(privilege);
```

(Java fragment)

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

(PHP function prototype
for setcookie and example usage)

```
setcookie("dvwaSession", $cookie_value, time()+3600, "/vulnerabilities/
weak_id/", $_SERVER['HTTP_HOST'], true, true);
```

- Cookie attributes can be set programatically …

Bad cookie usage may be “obvious” but can be missed by automated detection !

```
Cookie username = new Cookie("username", user);
Cookie password = new Cookie("password", pass);
response.addCookie(username);
response.addCookie(password);
```

- In this fragment from a Java application (Java Vulnerable Lab), cookies are set with sensitive information like the user name and his password and in plain-text (an instance of CWE-315 — Cleartext Storage of Sensitive Information in a Cookie). This is a design flaw.
- Static-analysis and pen-testing tools will not detect context-dependent vulnerabilities such as this one. They may at most signal this to be security-sensitive, and note the lack of the Secure and HttpOnly flags.
- Defense in this case: we should definitely not store these items in cookies, the use of plain-text format makes matter even worse. Session id should map to an user name in the server internal logic, and indicate that user is principle authenticated.

Cross-site scripting (XSS)

CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Weakness ID: 79
Abstraction: Base

Status: Usable

Presentation Filter: Complete

Description

Description Summary

The software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.

Extended Description

Cross-site scripting (XSS) vulnerabilities occur when:

1. Untrusted data enters a web application, typically from a web request.
2. The web application dynamically generates a web page that contains this untrusted data.
3. During page generation, the application does not prevent the data from containing content that is executable by a web browser, such as JavaScript, HTML tags, HTML attributes, mouse events, Flash, ActiveX, etc.
4. A victim visits the generated web page through a web browser, which contains malicious script that was injected using the untrusted data.
5. Since the script comes from a web page that was sent by the web server, the victim's web browser executes the malicious script in the context of the web server's domain.
6. This effectively violates the intention of the web browser's same-origin policy, which states that scripts in one domain should not be able to access resources or run code in a different domain.

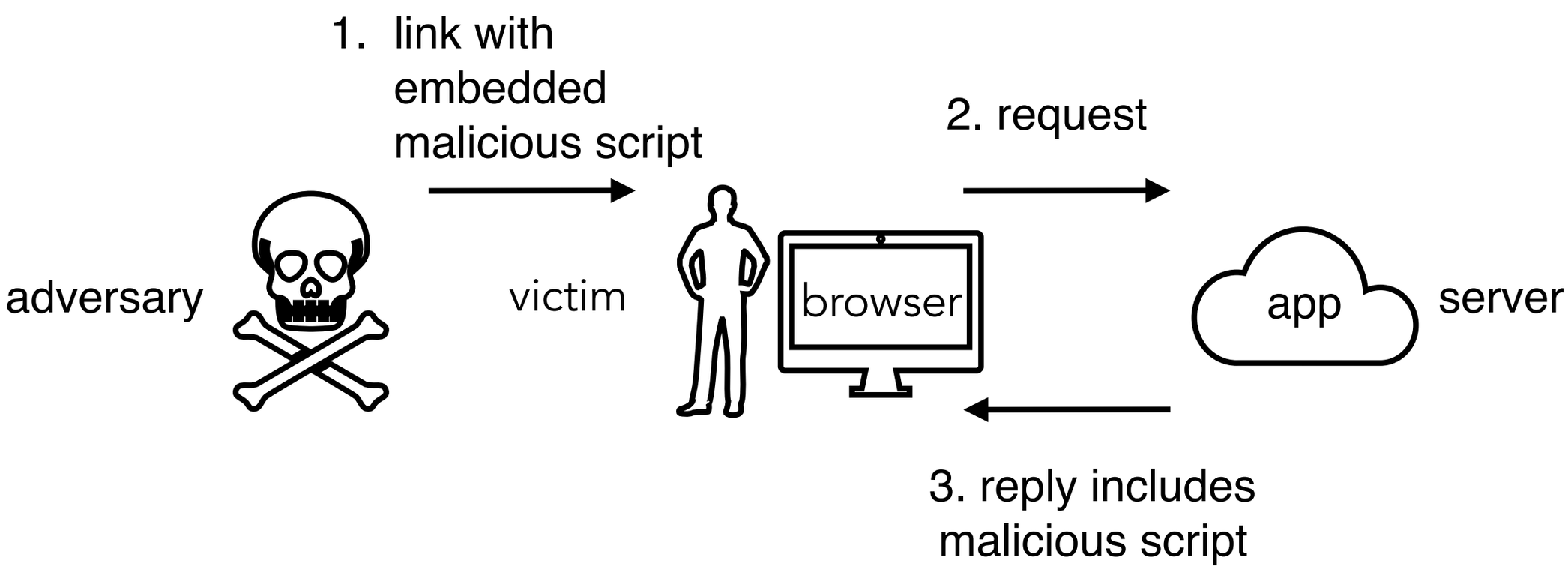
Nature	Type	ID	Name	V
ChildOf	C	74	Improper Neutralization of Special Elements in Output Used by a Downstream Component ('Injection')	699 1000 1003

http://cwe.mitre.org/data/definitions/79.html

XSS attacks and the SOP

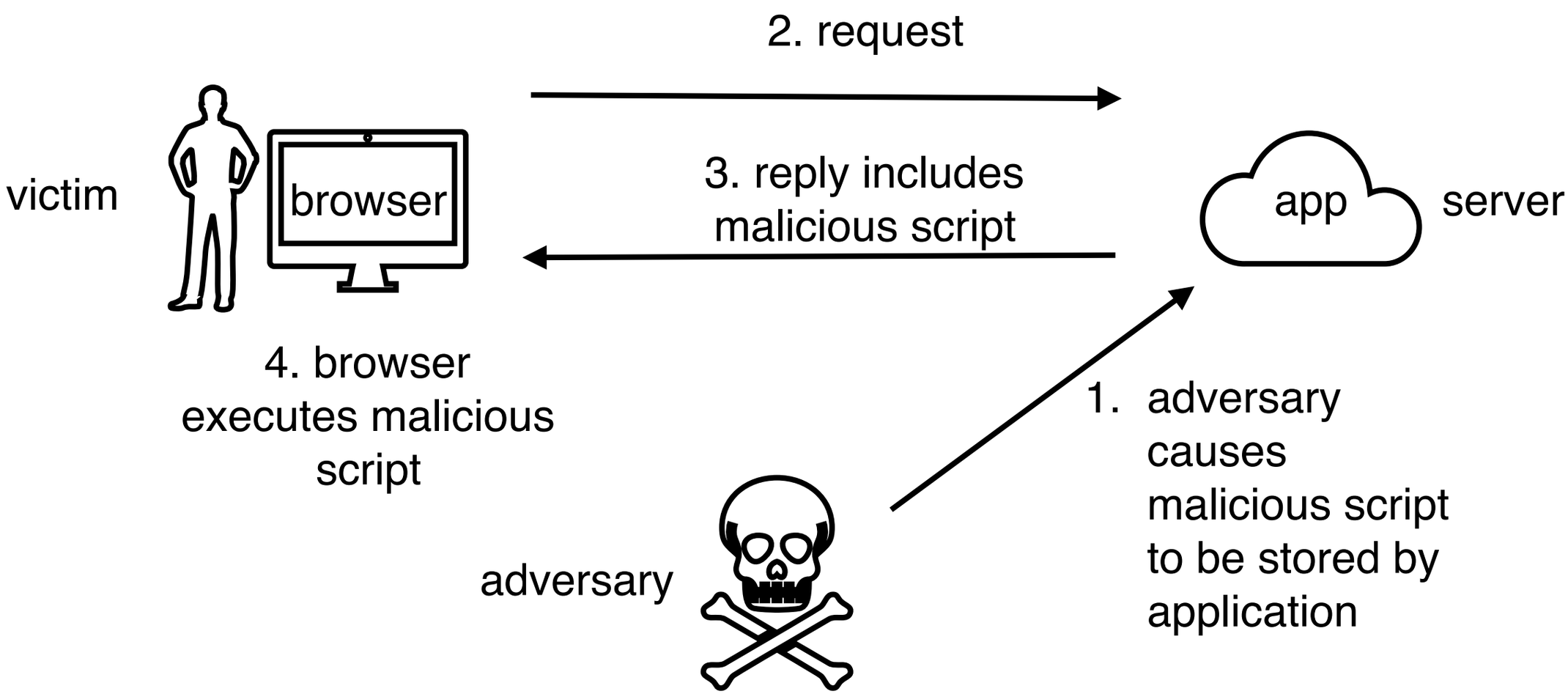
- **Common attack pattern**
  - Malicious input is supplied to a web application server encoding an executable script (e.g. through malicious link in email).
  - The server includes the script in the dynamic generation of a web page, possibly immediately (**reflected XSS**) or later (**stored XSS**).
  - Browser renders the page and executes the script.
- **XSS and the Same Origin Policy (SOP)**
  - SOP dictates that only scripts received from the same origin as the web page have access to the web page's DOM data.
  - XSS attacks "conform" to the SOP, since malicious content is loaded from the same origin.

Reflected XSS



- Malicious code delivered to an user through a link e.g. embedded in an email, web page, …
- Server reply “reflects” malicious script that is executed on the victim’s browser.

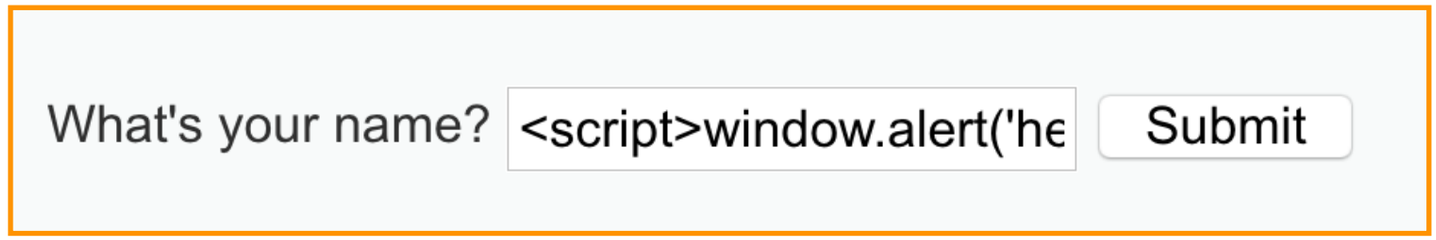
Stored XSS



- Malicious script stored by adversary exploiting a server-side vulnerability, then propagated to client browsers.

Reflected XSS — DVWA example

browser



request

server

```
echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
```

response

browser

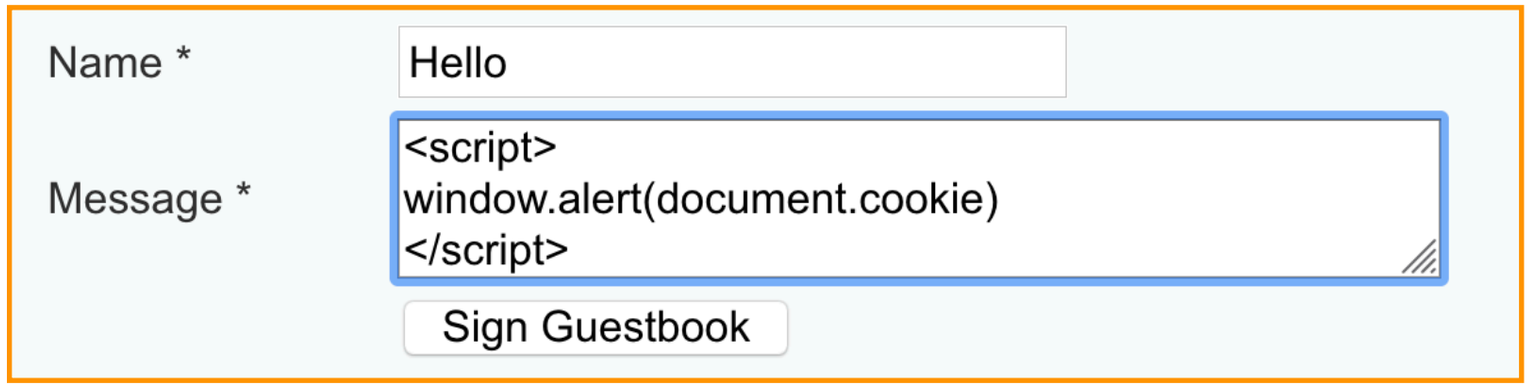
- Example above
  - DVWA set with low security level
  - Manual test illustrated, but CSRF-style malicious link could be easily crafted (note that a GET request is used).

Stored XSS — DVWA example

browser

request

server stores script




```
$query  = "INSERT INTO guestbook ( comment, name )
            VALUES ( '$message', '$name' );";
$result = mysql_query( $query )
```

subsequent request by browser

```
$query  = "SELECT name, comment FROM guestbook";
while( $row = mysqli_fetch_row( $result ) ) {
    …
}
```

- Script stored in the database and echoed back by the server for execution in the browser in subsequent visits.

Famous XSS attack — Samy XSS worm

- Samy attack on MySpace — a few quotes from "Ajax prepares for battle on the dark side", by Quinn Norton, Guardian, 2006
  - "Samy created **Ajax code on his MySpace site that ran automatically when anyone looked at his profile**. Because Ajax can interact with pages users never see, his code pressed all the relevant buttons to **add Samy to the victim's friends**, and added the words "but most of all, samy is my hero" to their page. **Finally, the code pasted itself into the victim's profile, so that any MySpace user viewing the victim's page would have their page infected**. MySpace users were unaware their computers were doing anything unusual."
  - "**The code - strictly speaking, a cross-site scripting worm - spread exponentially.** Within 24 hours Samy had a million emails from MySpace users "wanting" to be his friend and to whom he was their "hero". **MySpace was forced to shut down and make changes to stop Samy's code spreading.** The MySpace Worm, as it came to be called, served as an alarming example of what malicious hackers could do, even if they only had access to your browser."

Other XSS attacks

- Hackers still exploiting eBay's stored XSS vulnerabilities in 2017, Paul Mutton, NetCraft.com, 2017
  - "*All of the attacks stem from the fact that **eBay allowed fraudsters to include malicious JavaScript in auction descriptions.***"
- Email attack exploits vulnerability in Yahoo site to hijack accounts, Lucian Constantin, PCWorld, 2013
  - *"The same-origin policy is usually enforced per domain. […] **However, depending on the cookie settings, subdomains can access session cookies set by their parent domains.** This appears to be the case with Yahoo, where the user remains logged in regardless of what Yahoo subdomain they visit, including developer.yahoo.com.*
  - *"The rogue JavaScript code […] **forces the visitor's browser to call developer.yahoo.com with a specifically crafted URL […]**"*

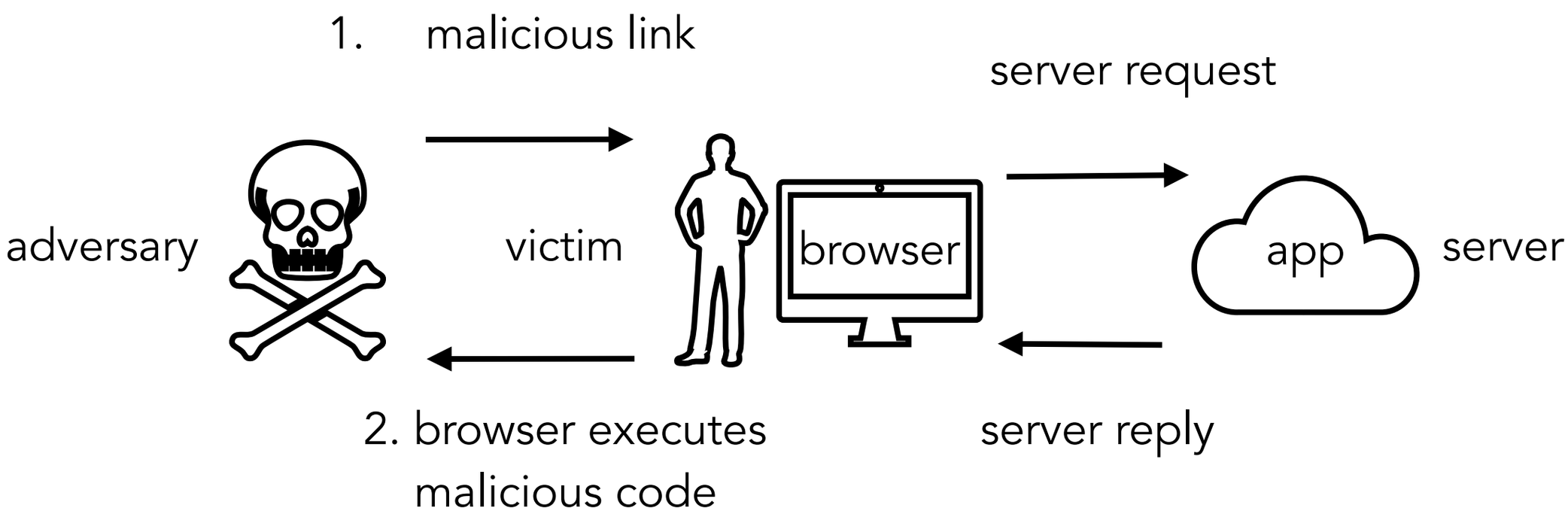
Prevention by input validation and output encoding

```
$message = htmlspecialchars( $message );
$name = htmlspecialchars( $name );
```

```
<script> ──────────► &lt;script&gt;
```

- **Preventing XSS — Server side**
  - **Input validation**: disallow/sanitise malicious input using conventional techniques, e.g. “escape” functions.
  - **Output encoding**: server sanitizes data before sending it, employing similar techniques.
  - DVWA example: high / impossible security levels in DVWA use the htmlspecialchars PHP function to escape HTML both for input sanitization and output encoding.

DOM-based XSS



- Malicious code may also be delivered in several forms: untrusted Javascript library, email links, etc
- Even if server interaction may play some role in serving the adversary's purpose, **DOM-based XSS takes effect directly on the client by manipulating the DOM model — malicious code need not be emitted by the server.**

DOM — Document Object Model

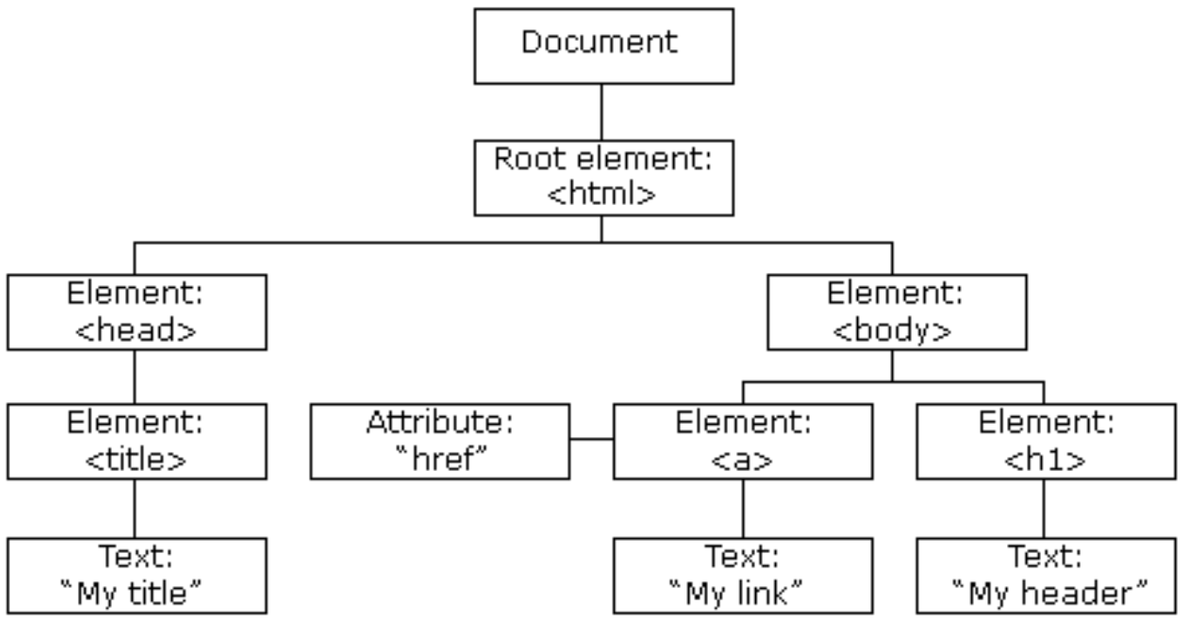


Image source: W3 Schools

```
<!DOCTYPE html>
<html> event
 <head>…</head>
 <body>
  <header>
   <div class="wrapper">
    <table>
     <tbody>
      <tr>
       <td valign="top">…</td>
       <td width="30">…</td>
       <td>
        <h2>Questões de Segurança em Engenharia de Software</h2>
        <h2>2018/19</h2>
        <p>
         <a href="http://msi.dcc.fc.up.pt">Mestrado em Segurança Informática
         </a>
         <br>
         <a href="http://www.dcc.fc.up.pt">
         Departamento de Ciência de Computadores</a>
         <br>
         <a href="http://www.fc.up.pt">
         Faculdade de Ciências da Universidade do Porto</a>
```

Firefox DOM Inspector

- The Document-Object model (DOM) is a tree-abstraction for documents:
  - an HTML (but also XML, XHTML, SVG, …) document is treated as a tree structure where in each node is an object representing a part of the document.
  - tree nodes can be visited, created/added/deleted, and have associated attributes like event handlers and styles.
  - A W3C standard until 2004, now maintained by the WHATWG group — check the live document for the current DOM specification.
  - Browsers represents HTML document in an internal structure similar to the DOM - major browsers use the WebKit Webcore component for that purose

Javascript DOM API — outlook

- A brief overview of some of the functionality in the Javascript DOM API … accessible through **`document`**, the top-level object that represents the DOM:
  - Basic attributes:
    - `title referrer URL hash cookie readyState`
  - Page elements:
    - `head body forms scripts links`
    - `getElementById(elementId)`
    - `querySelector(cssSelector)`
  - Modification methods:
    - `write(anything) writeln(anything)`: append output to the document
    - `createElement()` `createEvent()` `execCommand()` `addEventListener()`
- A wide attack surface for malicious scripts!

DOM-based XSS — simple example

Malicious script

```
<script>
  // Get malicious input from query string and unescape it
  var pos = document.URL.indexOf("evil=")+9;
  var evilScript=document.URL.substring(pos,document.URL.length);
  // Make it take effect
  document.write(unescape(evilScript));
</script>
```

Code injection possible using:

```
queryStringAttack.html?evil=<script>. . . </script>
```

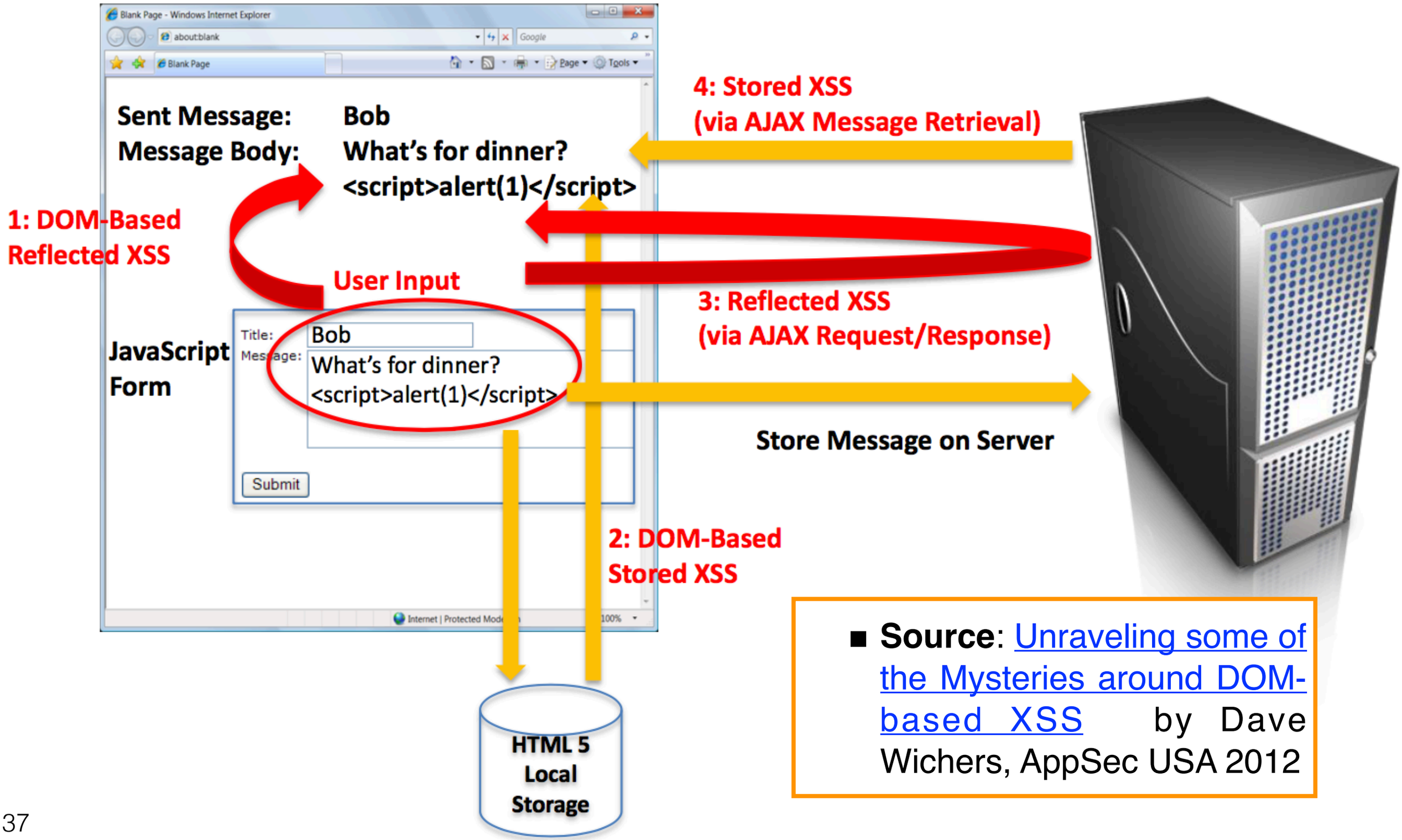
- Query string associated to HTML page. No server interaction for triggering the exploit.
- Anchor '#' (i.e., document.hash instead of document.URL) also exploitable in similar

DOM-based XSS — another example

```
// Store
localStorage.setItem("lastname", "Smith");
// Retrieve
document.getElementById("result").innerHTML = localStorage.getItem("lastname");
```

- HTML 5 introduced local browser storage of key-value pairs (like cookies), an extra attack surface.
  - One more facility to store sensitive or malicious data that can be controlled programatically. Above: a “predictable” usage example from w3schools … (!) How can it go wrong?
  - WebSQL and IndexedDB also allow structured databases, though the adoption of one or the other has not been peaceful (both out of HTML 5). Firefox only supports IndexedDB, Safari and Google also support WebSQL.
- Extra attack-surface: CSS-based vulnerabilities
  - CSS Exfil
  - Microsoft Internet Explorer Cascading Style Sheets Remote Code Execution Vulnerability

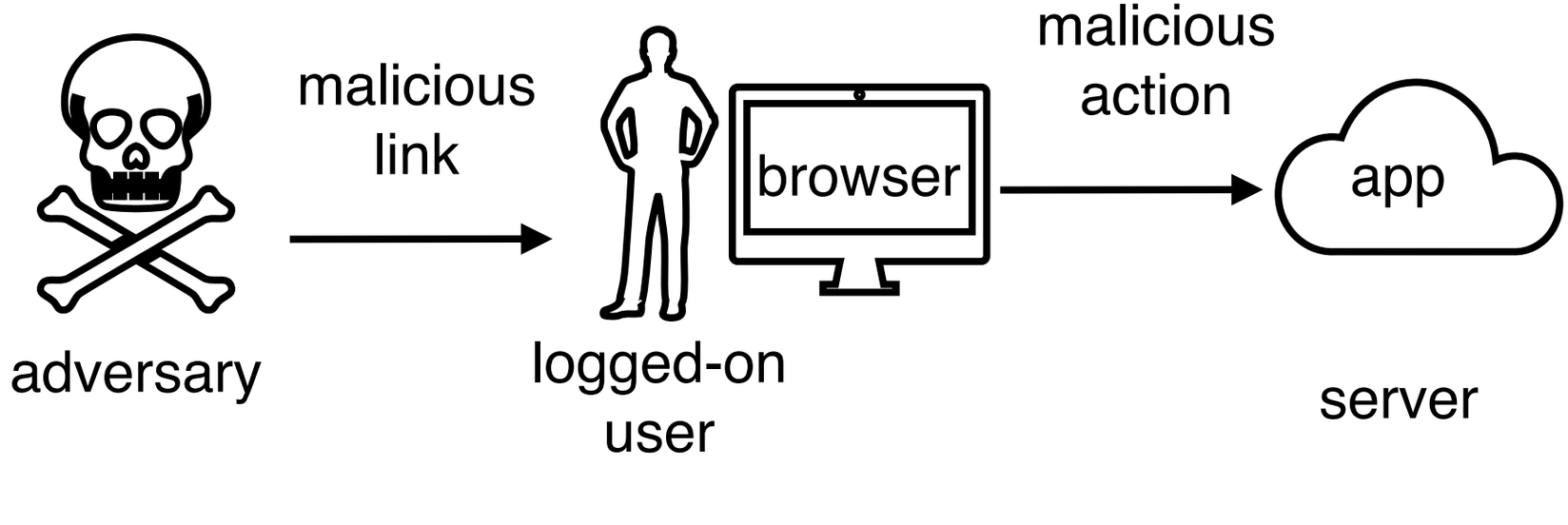
XSS types compared



- **Source**: Unraveling some of the Mysteries around DOM-based XSS by Dave Wichers, AppSec USA 2012

Cross-site request forgery (CSRF)

CSRF — general description



- CWE-352 - Cross-Site Request Forgery (CSRF)
  - "*The web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.*"
- Most common attack pattern:
  - User has an authenticated session for a web application.
  - Adversary tricks user into executing some malicious action, e.g. by clicking a link sent by email or provided in a web site controlled by the adversary.
  - Malicious actions are executed in the server as if intended by the user.

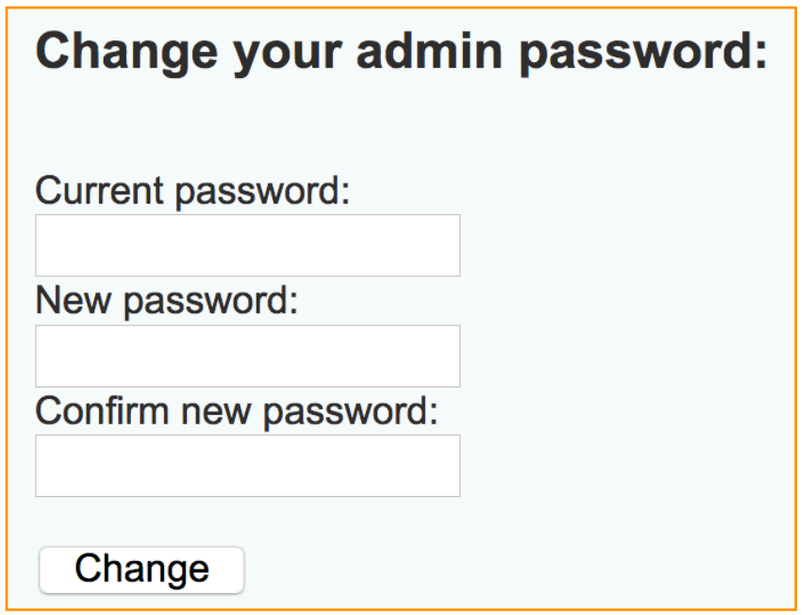
XSS vs CSRF

- XSS
  - Trust relation: client trusts the server
  - Attacker tries to affect what the server sends to the client / what runs on the client.
- CSRF
  - Trust relation: server trusts the client
  - Attacker tries to affect what the client sends to the server / runs on the server.

Example CSRF attacks — Gmail, 2010

- GMail Service CSRF Vulnerability (2010)
  - “*GMail is vulnerable to CSRF attacks in the "Change Password" functionality. The only token for authenticate the user is a session cookie, and this cookie is sent automatically by the browser in every request.*”
  - “*An attacker can create a page that includes requests to the "Change password" functionality of GMail and modify the passwords of the users who, being authenticated, visit the page of the attacker.*”
  - *“The attack is facilitated since the "Change Password" request can be realized across the HTTP GET method.”* [it suffices to craft a malicious link with an appropriate query string]

CSRF example — DVWA



- As in the Gmail example, the password change functionality is at stake.

- Security levels:
  - **low**: no protections, a simple malicious link may be used to change the password
  - **medium**: HTTP request header **Referrer**; Referrer link may also be forged by an adversary. Alternatively, malicious link can however be accomplished by exploiting a XSS vulnerability (hint: try the message forum; more discussion on this next).
  - **high**: automatically generated **anti-CSRF token** included in hidden form field - token is attached to session but not the request itself however …
  - "**impossible**": anti-CSRF token + request for current password confirmation

Some CSRF mitigations

- Allow only POST requests
  - GET requests allows a CSRF attack through a simple link.
  - POST requests may anyway be trivially defined for instance by HTML forms that are presented to the user.
- HTTP **`Referer`** field is used to indicate origin of request. A basic protection is to check it on the server side. The value may be absent however, frequently with the good motivation of preventing user tracking or leaking data to untrusted sites.
  - **`<meta name="referrer">` in HTML header** or the **`Referrer-Policy`** HTTP header may inhbit it.
  - Most browsers may be configured to omit the referrer information for cross-site requests.
  - A common referrer policy is to hide the information when making HTTP requests from content loaded through HTTPS.
  - **Note**: due to a typo in the original HTTP specification, the header is called **Referer** rather than **Referrer**.
- **Hardened application logic**: re-authentication or 2FA schemes for critical operations.
- **Use of synchronization tokens (next).**

Anti-CSRF token in DVWA

```
// Check Anti-CSRF token
checkToken( $_REQUEST[ 'user_token' ],
            $_SESSION[ 'session_token' ], 'index.php' );

// Do the passwords match?
if( $pass_new == $pass_conf ) {
      . . .
}
// Regenerate Anti-CSRF token
generateSessionToken();
```

```
<form action="#" method="GET">
  . . .
  <input type='hidden' name='user_token'
    value='d842e88752fd9991fb4dbcfa35649ae4' />
</form>
```

- Expected value of anti-CRSF token is checked first. Operation does not proceed on a token mismatch.
- The token gets regenerated with a new value once operation is complete.
- Javascript / XSS-based exploit possible - check here for an example

Anti-CSRF token in DVWA (cont.)

```
function generateSessionToken() { # Generate a brand new (CSRF) token

  if( isset( $_SESSION[ 'session_token' ] ) ) {
   destroySessionToken();
  }

  $_SESSION[ 'session_token' ] = md5( uniqid() );

}
```

- The generation function is in itself weak (it would not be appropriate for session ids for example):
  - MD5 is a weak cryptographic-hash function.
  - the uniqid() PHP function is predictable and does not in fact guarantee a unique ID - “[it] gets an unique identifier based on the current time in microseconds“ and “does not guarantee uniqueness of return value”.
- However, since DVWA regenerates the token after each request and the id is based on the scale of micro-seconds, it is hard to predict it quickly enough (1 million possibilities per second).

Synchronizer token pattern

- **Synchronizer Token Pattern**
  - State changing operation uses a token (different from the session id), generated through a cryptographically-secure random generator by the server, and that is unique per session (server associates token to the session id).
  - Token value is embedded in the web page or within a cookie.
  - Token mismatch in a future request inhibits state-changing operation.
- For enhanced security:
  - Token can be regenerated after each request (as in DVWA) or at least have a short expiration time.
  - Use different tokens per request/operation rather than for the entire session. This may hinder usability though (e.g. Back button reverts to a page with a invalid token).
- The synchronizer token pattern requires a state to be maintained explicitly. Stateless alternatives exist:
  - Encrypted-token scheme (check is based on sucessful decryption).
  - “Double-submit” cookie: generate nonce and include it both in the response body and in a header field.

CSRF: further reference

- Barth et al., "Robust Defenses for Cross-Site Request Forgery", CCS'08
- OWASP Cross-Site Request Forgery Prevention