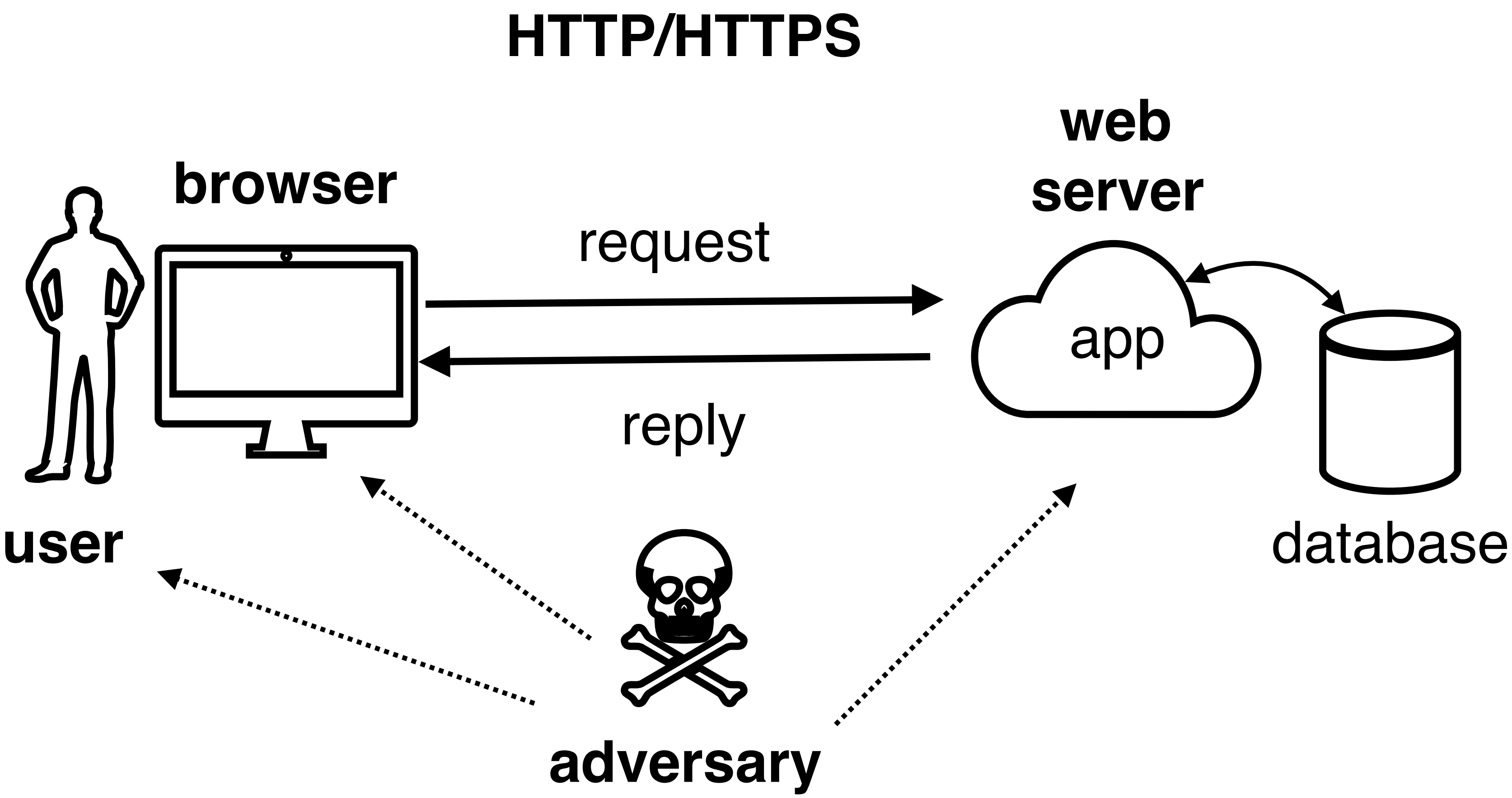
Web application vulnerabilities

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

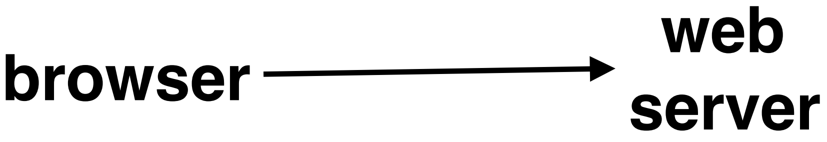
Web applications

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 do you think an adversary may try to do ? For what purpose?

HTTP requests



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

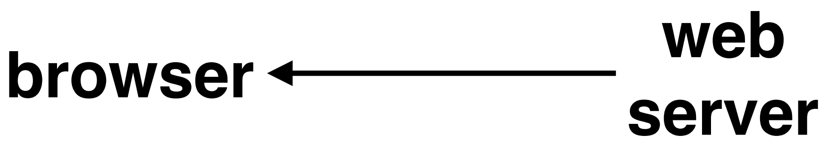
POST request

```
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

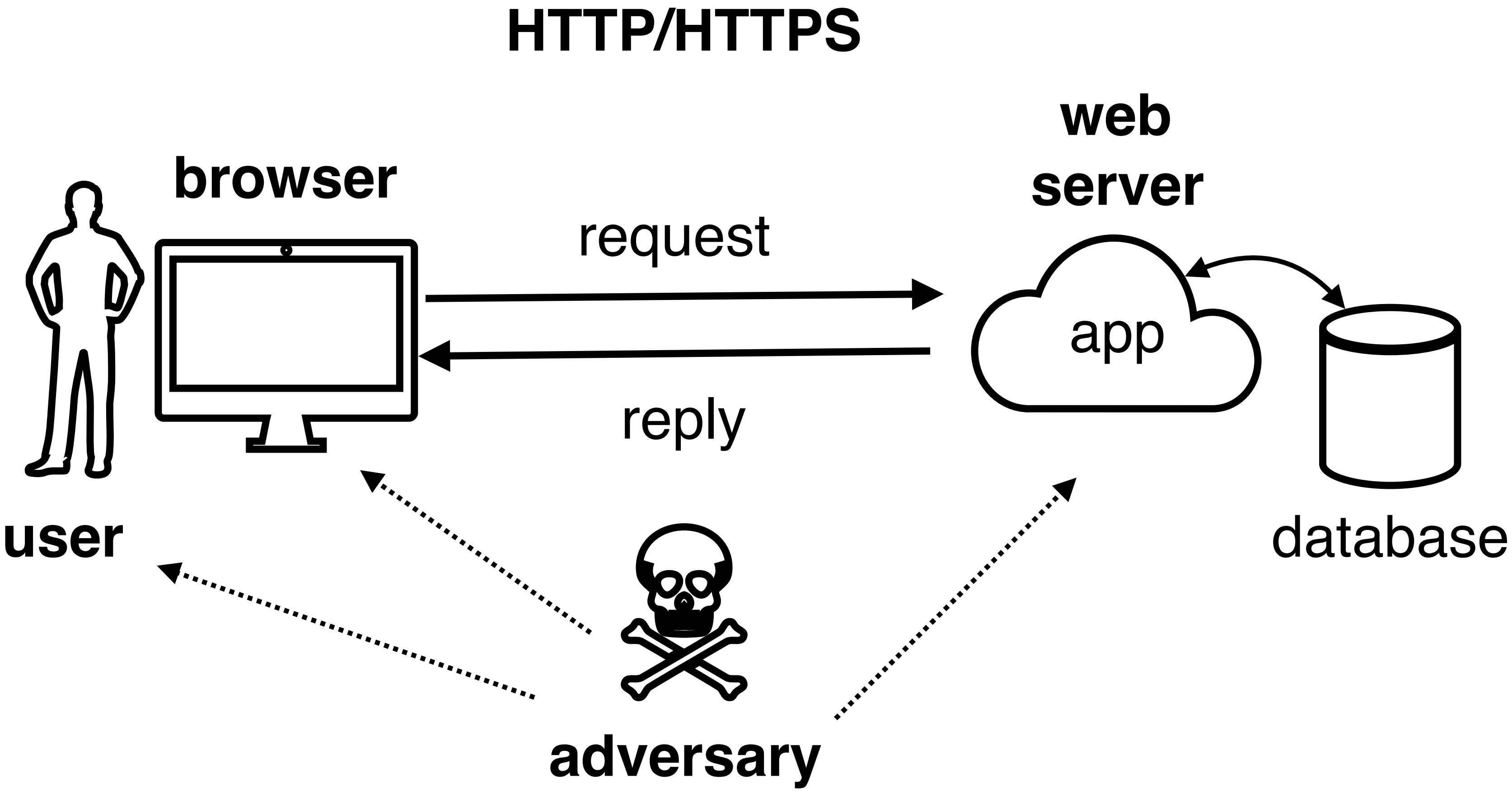
```
HTTP/1.1 200 OK
Date: Mon, 15 Oct 2018 14:44:24 GMT
Server: Apache/2.4.10 (Debian)
Expires: Tue, 23 Jun 2009 12:00:00 GMT
Cache-Control: no-cache, must-revalidate
Pragma: no-cache
Vary: Accept-Encoding
Content-Length: 1567
Content-Type: text/html;charset=utf-8

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

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

data (HTML in this case

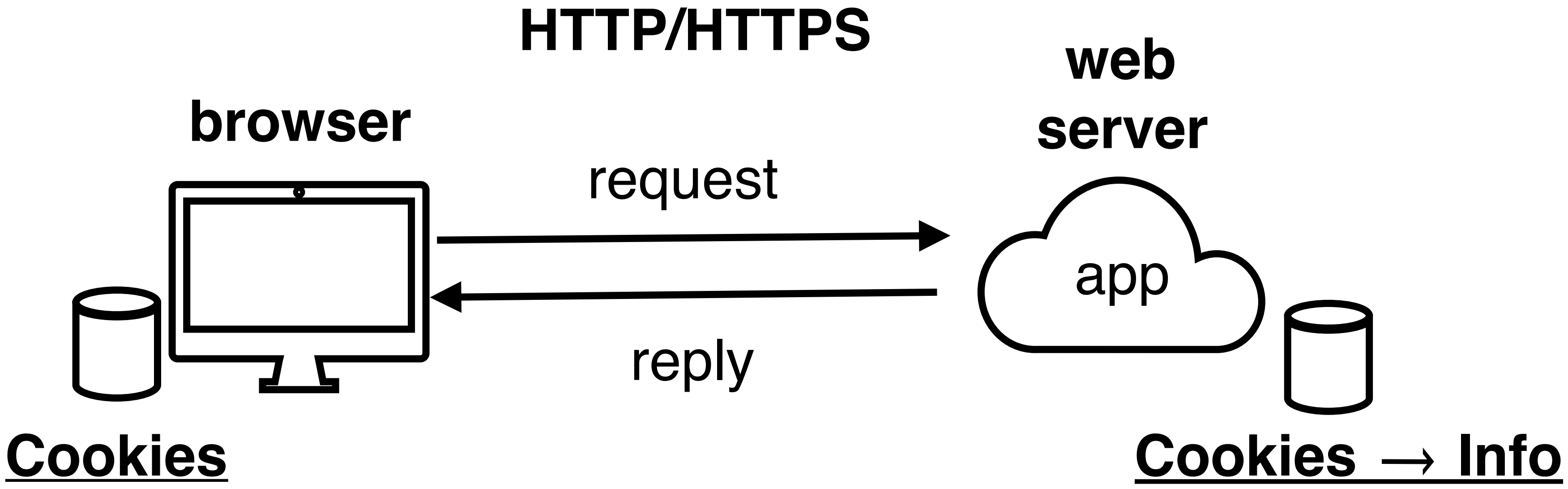
Vulnerabilities



- We will look at the following types of vulnerability, that are web-application specific:
  - Cookie related vulnerabilities
  - Cross-Site Request Forgery (CSRF)
  - Cross-Site Scripting (XSS)
- Injection vulnerabilities (last class) like SQL injection and OS command injection are also omnipresent in web applications (as we showed for DVWA) but not specific to them.

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 => 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** — identifying logged-on users
  - **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 semantics

```
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 set to “14-Nov-2018 15:42:07 GMT” — the cookie will not — so this is a **persistent cookie** ; cookies without expiration time are deleted once a browser session is terminated and are called **session cookies** (note: Expires and MaxAge attributes may both be used to specify expiration)
  - 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)

Attack surface

- Cookies:
  - … may be persistent and even not expire
  - … may be read and modified by Javascript
  - … may be intercepted by “man-in-the-middle”
  - … may be predictable (e.g. session ids)
  - … may contain/leak confidential data
- This may help
  - Session hi-jacking (next)
  - Cross-site request forgery attacks (also discussed in this class)

Session hijacking

- Basic scenario:
  - Adversary steals an authentication cookie.
  - Adversary may then impersonate a legitimate user (spoofing).
  - … and materialize other threats afterwards
- How can cookie be stolen?
  - By compromising the browser or server.
  - By prediction — to prevent this an authentication cookie should be truly random and sufficiently long.
  - MIM attacks facilitated if cookie is sent over plain HTTP.
  - More complex MIM attacks may employ DNS cache poisoning: adversary impersonates host of interest, browser sends cookies for the site's domain willingly.

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
- Defences
  - 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 session.
  - Alternatively, limited persistency is a common compromise by having an expiration time set.

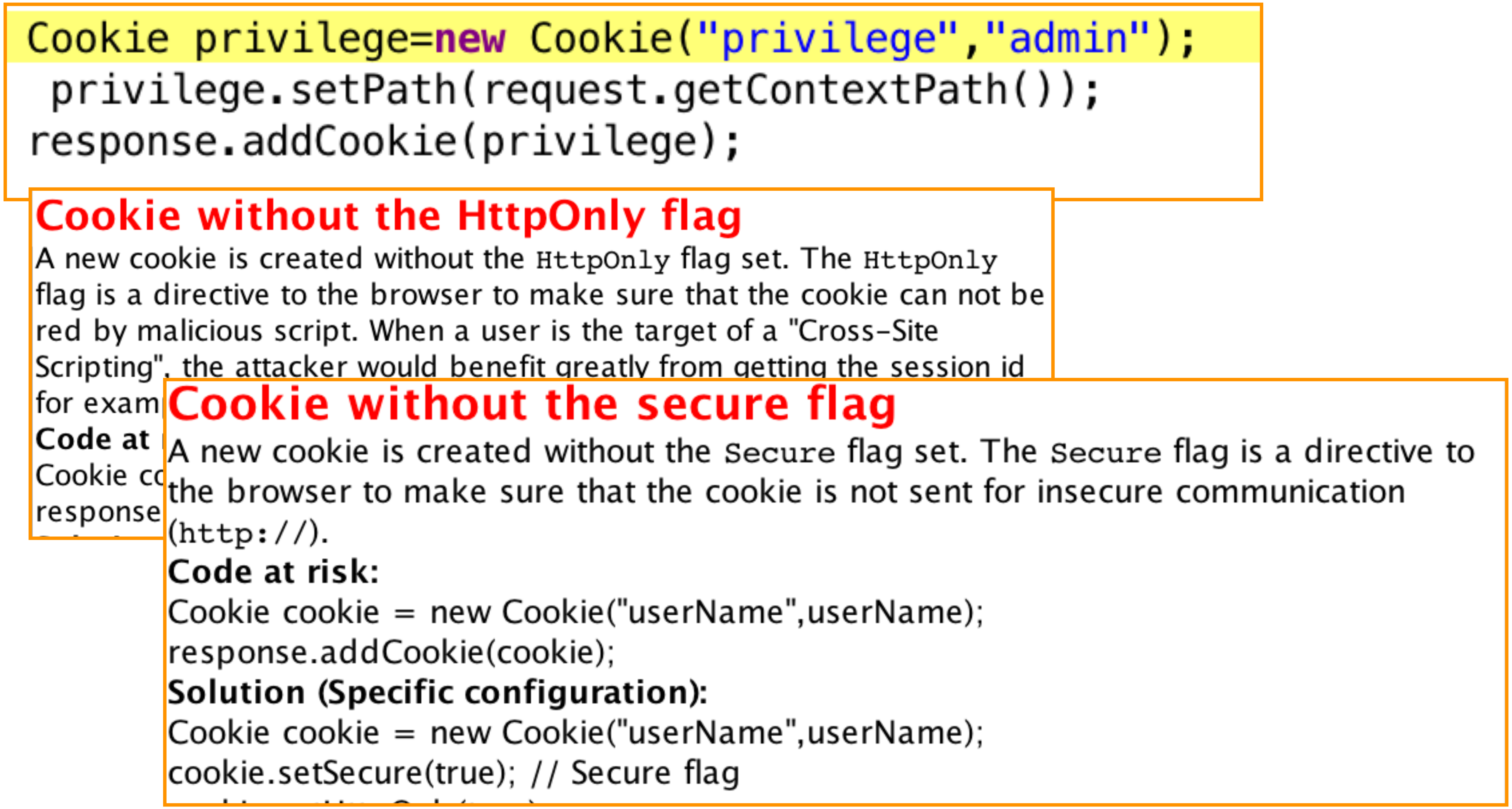
Session hijacking story — Firesheep

- 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 are not allowed to be transmitted over HTTP.
  - **Sites should use HTTPS uniformly**.
- 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).

Other common cookie-related vulnerabilities

- 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 (some) cookie vulnerabilities — static analysis



```
Cookie privilege=new Cookie("privilege","admin");
 privilege.setPath(request.getContextPath());
response.addCookie(privilege);
```

Cookie without the HttpOnly flag

A new cookie is created without the `HttpOnly` flag set. The `HttpOnly` flag is a directive to the browser to make sure that the cookie can not be red by malicious script. When a user is the target of a "Cross-Site Scripting", the attacker would benefit greatly from getting the session id for exam

Code at

Cookie co

response

Cookie without the secure flag

A new cookie is created without the `Secure` flag set. The `Secure` flag is a directive to the browser to make sure that the cookie is not sent for insecure communication (`http://`).

Code at risk:

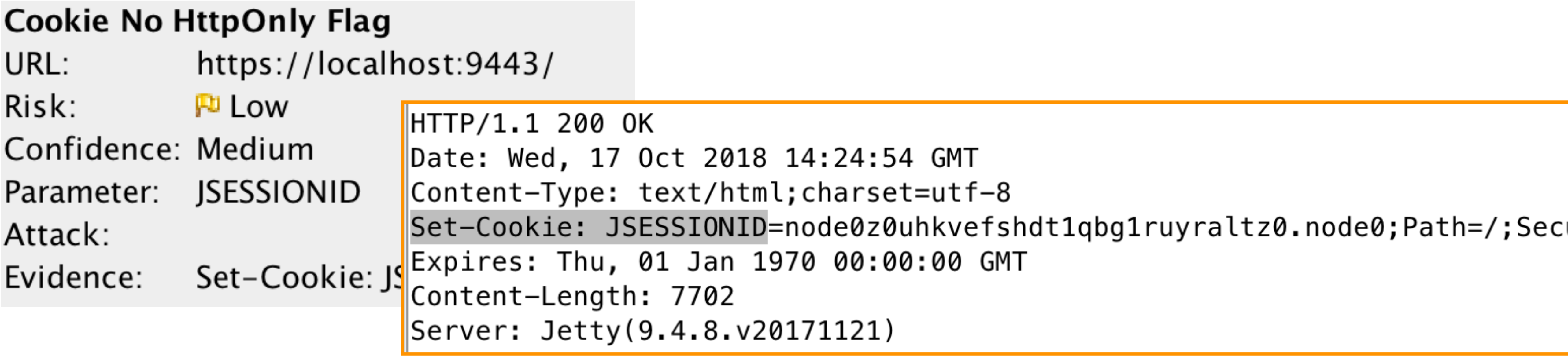
```
Cookie cookie = new Cookie("userName",userName);
response.addCookie(cookie);
```

Solution (Specific configuration):

```
Cookie cookie = new Cookie("userName",userName);
cookie.setSecure(true); // Secure flag
```

- SpotBugs + FindSecBugs scan over JavaVulnerableLab (application for 1st project).

Detecting (some) cookie vulnerabilities — pen-testing



Cookie No HttpOnly Flag

URL: https://localhost:9443/

Risk: Low

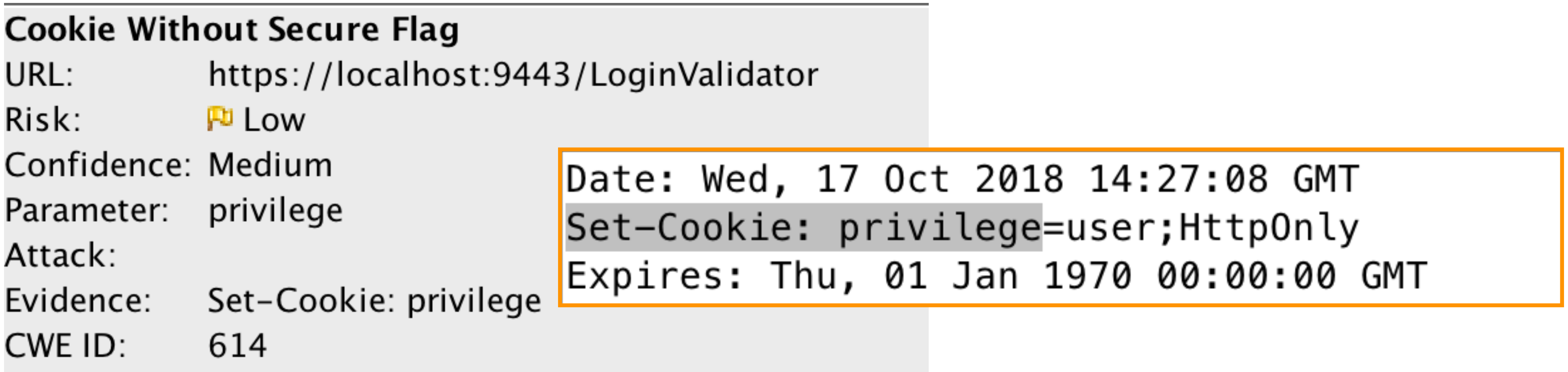
Confidence: Medium

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=/;Sec
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
```

- ZAP passive scan over JavaVulnerableLab (application for 1st project).

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 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).
- Typically, static-analysis and pen-testing tools will not detect context-dependent vulnerabilities such as this one. SpotBugs and ZAP will only notice the lack of the Secure and HttpOnly flags.
- Defence in this case: we should definitely not store these items of data in cookies, plain-text format makes matter even worse. Session id should map to an user name in the server internal logic, and indicates that user is principle authenticated (so why store password in a cookie?).,

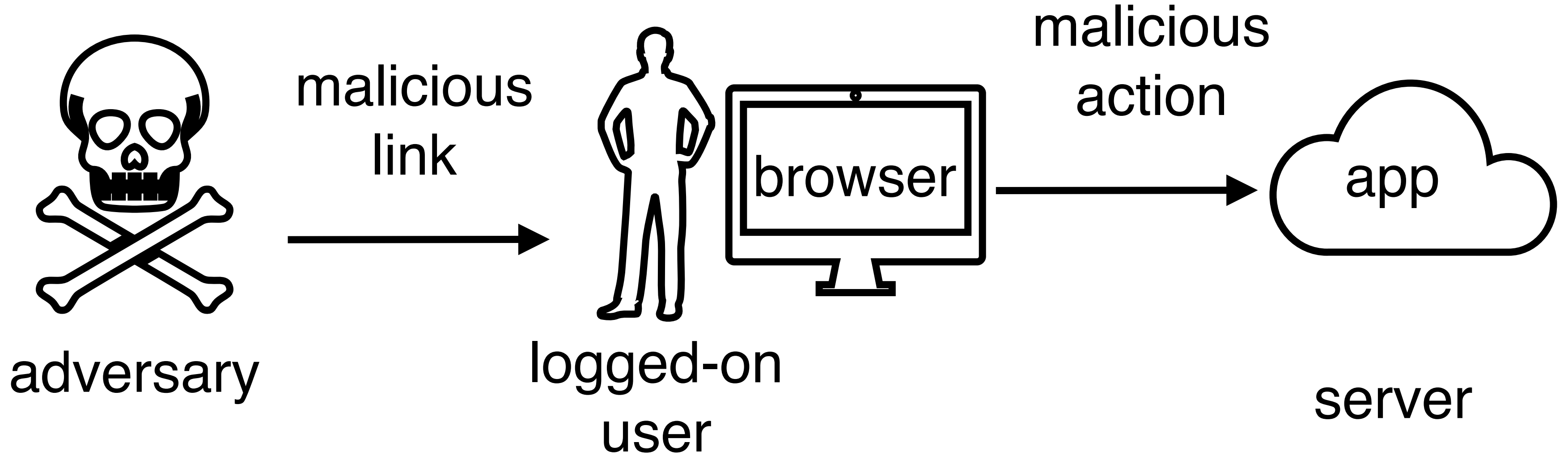
Secure cookie programming

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

- Cookie attributes can be set programatically for aspects such as expiration or the HttpOnly + Secure flags

Cross-site request forgery (CRSF)

CSRF — general description

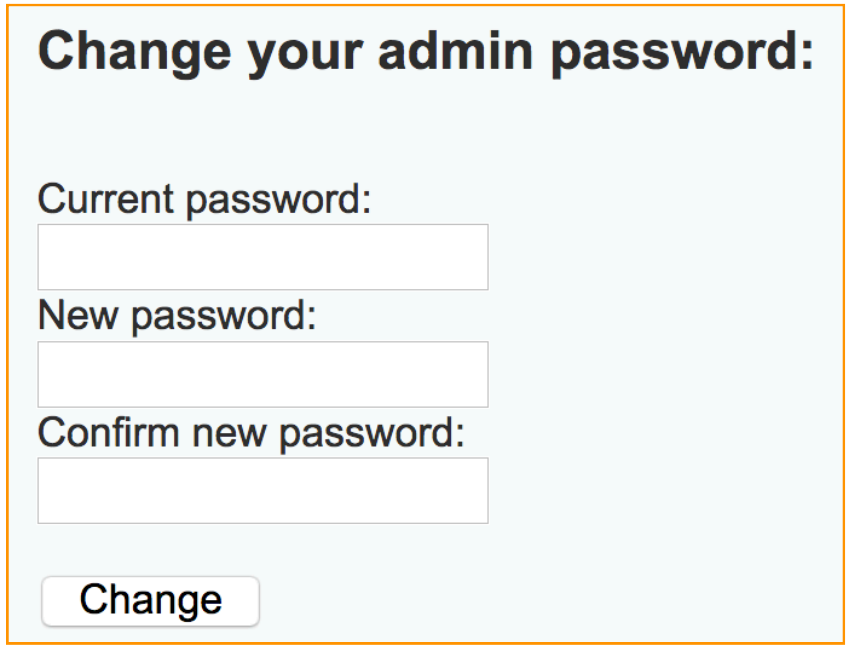


- CWE-352 - Cross-Site Request Forgery (CSRF)
  - “*The web application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.*”
- Common attack pattern:
  - User has an authenticated session for a web application.
  - Adversary tricks user into executing some malicious action, e.g. by clicking a link or opening a media file sent by email or through a maliciously embedded script (through cross-scripting, discussed later)
  - Malicious actions are executed as if intended by the user.

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



Change your admin password:

Current password:

New password:

Confirm new password:

- 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

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();
```

```
function generateSessionToken() {
      if( isset( $_SESSION[ 'session_token' ] ) ) {
              destroySessionToken();
       }
       $_SESSION[ 'session_token' ] = md5( uniqid() );
}
```

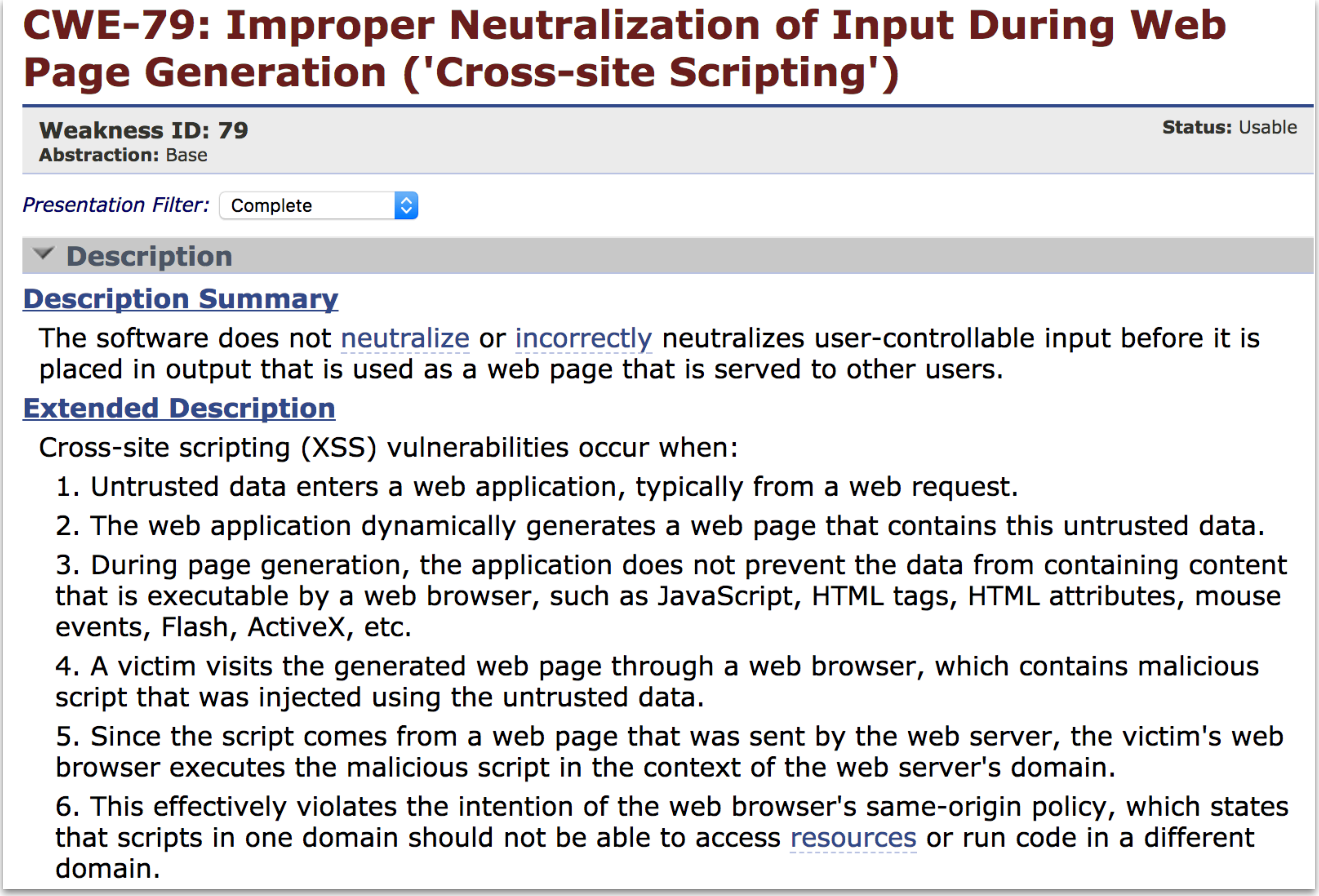
```
<form action="#" method="GET">
  . . .
  <input type='hidden' name='user_token'
    value='d842e88752fd9991fb4dbcfa35649ae4' />
</form>
```

- 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

Synchronizer token pattern

- **Synchronizer Token Pattern**
  - State changing operation uses a token, generated through a cryptographically-secure random generator , and that is unique per session.
  - Token mismatch inhibits state-changing operation.
- For enhanced security:
  - Token can be regenerated after each request (as in DVWA)
  - Use different tokens per request/operation rather than for the entire session (DVWA does *not* do this!). This May hinder usability though (e.g. Back button reverts to a page with a invalid token).
- Token-based mitigation is the most common and recommend mitigation techniques.
- Other alternatives the synchronizer token pattern, other forms of token-based mitigation can be used — e.g. check the OWASP Prevention Cheat Sheet.

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

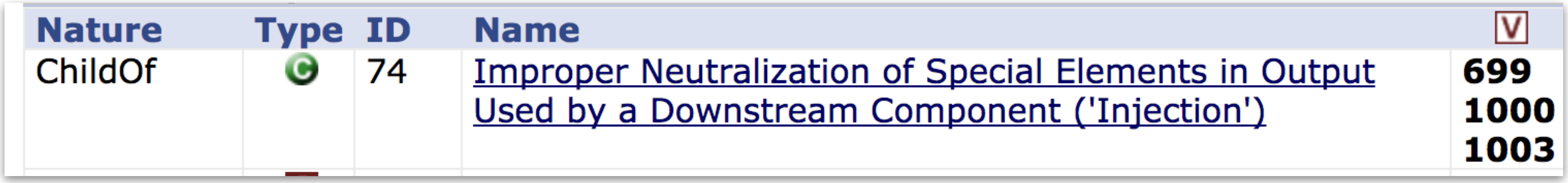
Description Summary

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

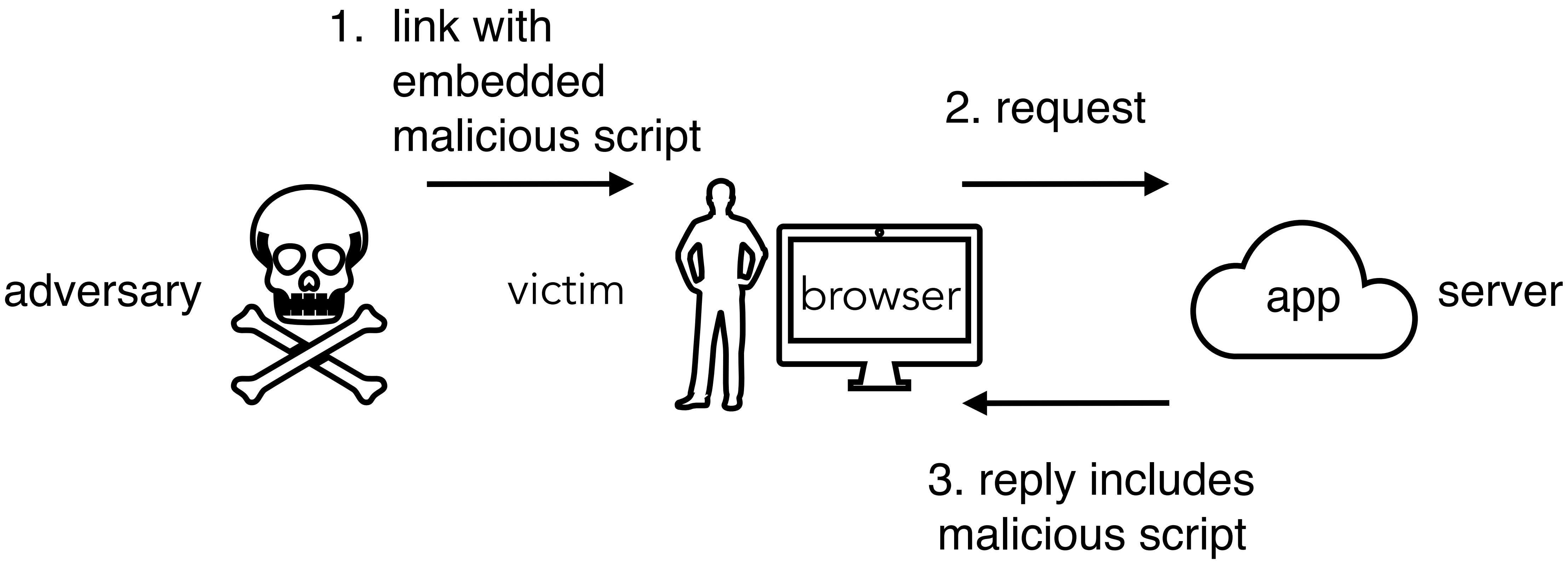
XSS attacks

- Steps
  - Malicious input is supplied to a web application (e.g. through malicious link in email), encoding an executable script.
  - Server includes the script in the dynamic generation of a web page.
  - Browser renders the page and executes the script.
- Two main types of XSS
  - **Reflected XSS**: malicious script is reflected immediately by server.
  - **Persistent XSS (also called Stored XSS)**: malicious script is stored by application at the server; web page is rendered some time later.

XSS and the Same Origin Policy

- Motivation for injecting the script in the server: can malicious scripts be loaded from a third-party site?
- The **Same Origin Policy** (SOP) dictates that only scripts received from the same origin as the web page have access to the web page’s DOM data.
- The point of XSS attacks is that scripts are treated as trusted since they originate from the same origin as the containing web page, hence the Same Origin Policy is not formally violated.

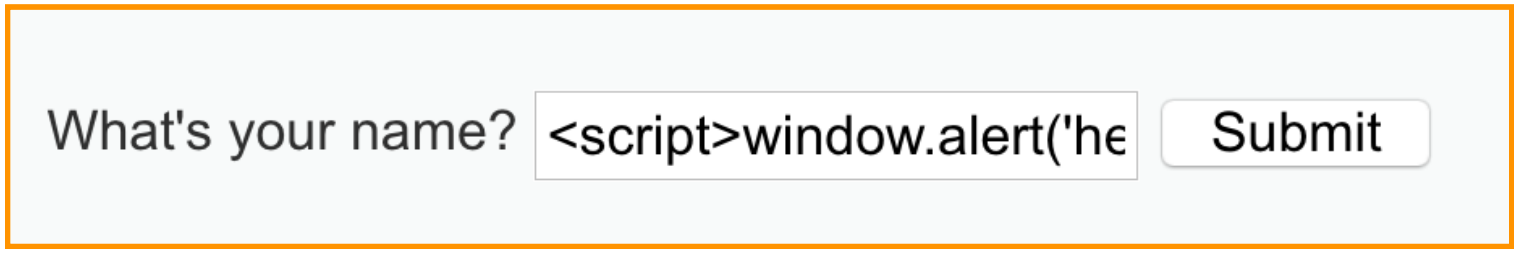
Reflected XSS


1. link with embedded malicious script
2. request
adversary
victim
browser
app
server
3. reply includes malicious script

- 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.

Reflected XSS — DVWA example

browser



What's your name? <script>window.alert('he

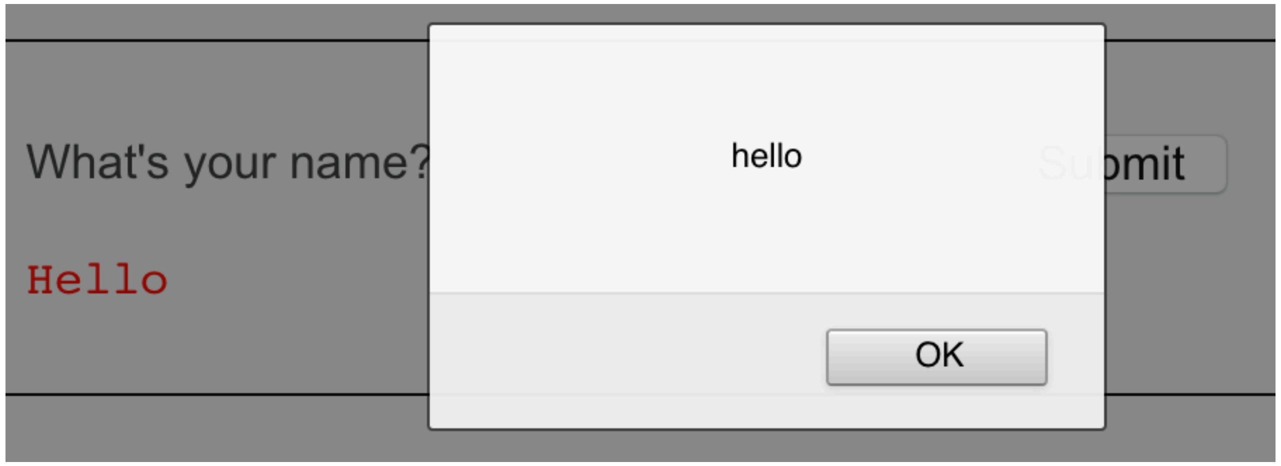
request

server

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

response

browser



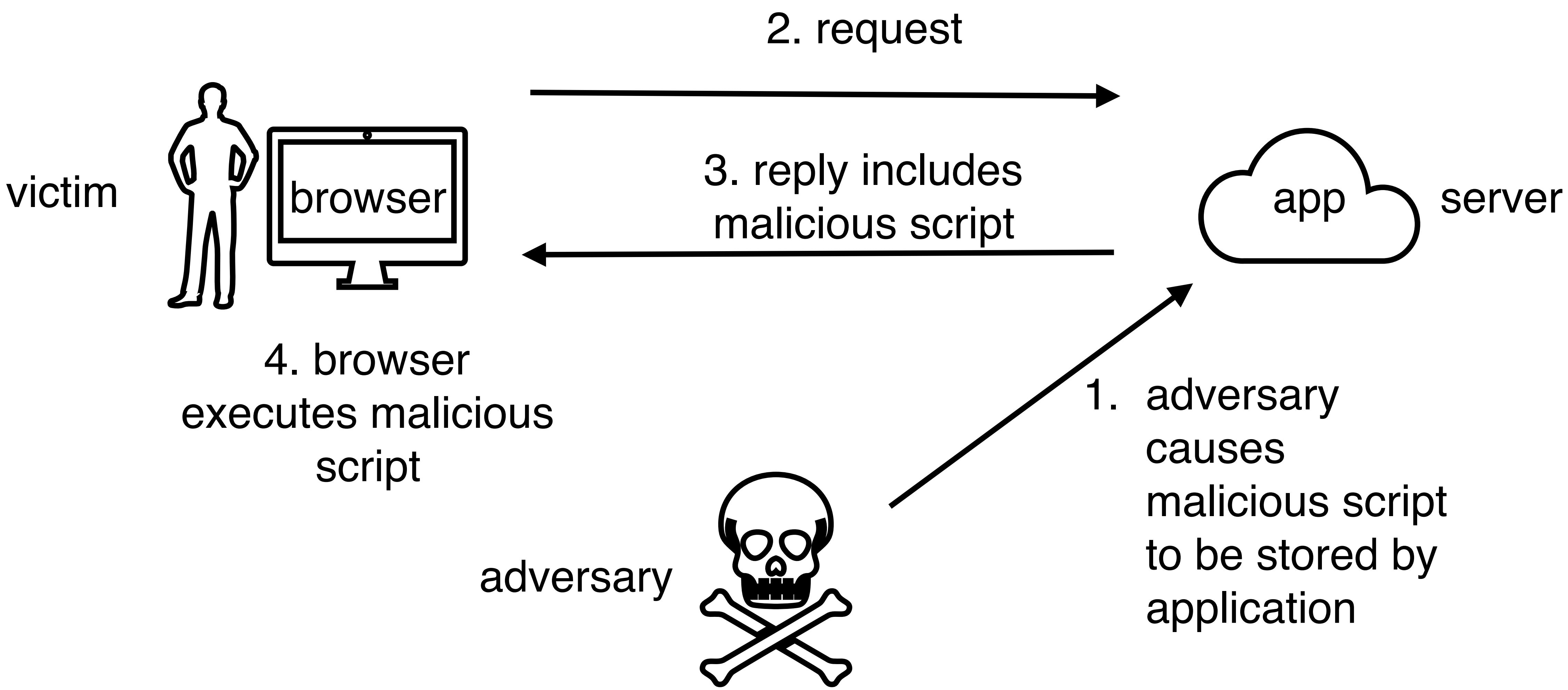
What's your name?

Hello

hello

- 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).

Persistent XSS


2. request
victim
browser
3. reply includes malicious script
app
server
4. browser executes malicious script
1. adversary causes malicious script to be stored by application
adversary

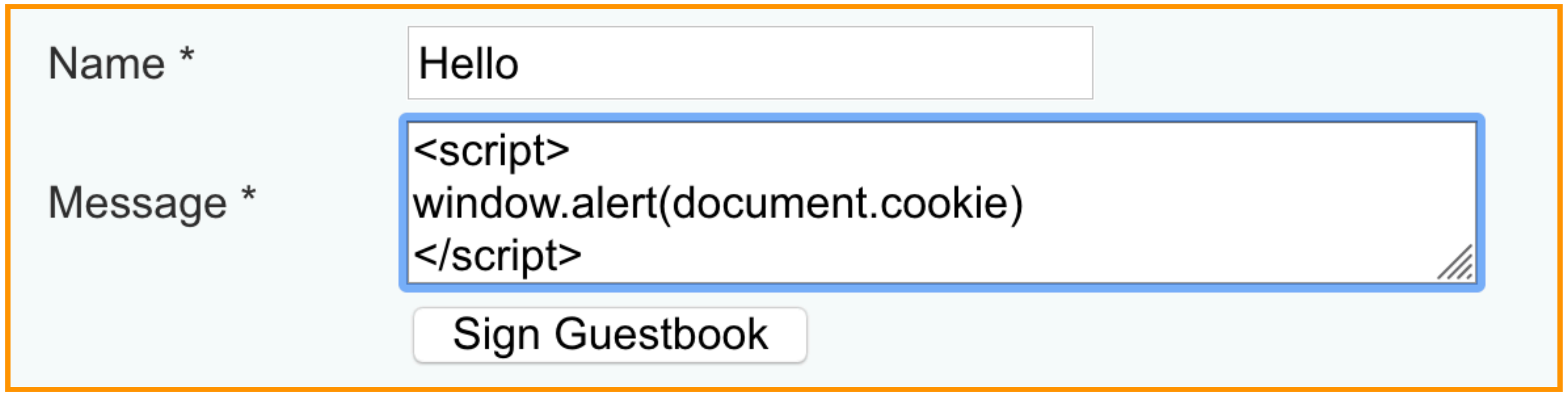
- Malicious script stored by adversary exploiting a server-side vulnerability, then propagated to client browsers.

Persistent XSS — DVWA example

browser

request

server
stores script



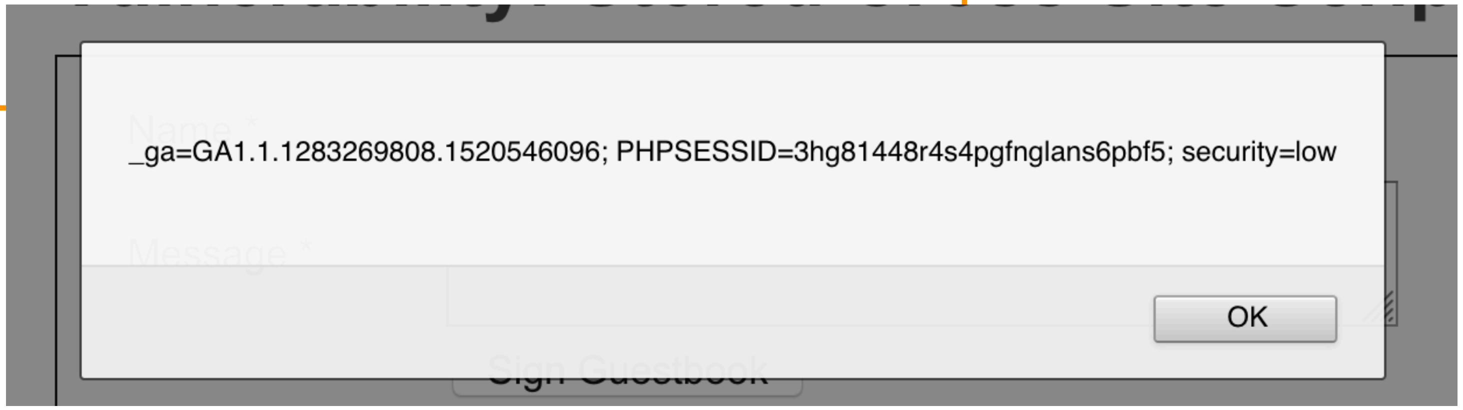
Name * Hello

Message * <script>
window.alert(document.cookie)
</script>

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

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

subsequent
request by browser



_ga=GA1.1.1283269808.1520546096; PHPSESSID=3hg81448r4s4pgfnglans6pbf5; security=low

- 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 […]***"

XSS vs CSRF

- XSS
  - Trust relation: client trusts the server
  - Attacker tries to affect what the server sends to the client.
- CSRF
  - Trust relation: server trusts the client
  - Attacker tries to affect what the client sends to the server.
- XSS and CSRF can of course be combined for an attack (one may enable the other for instance)

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.