HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

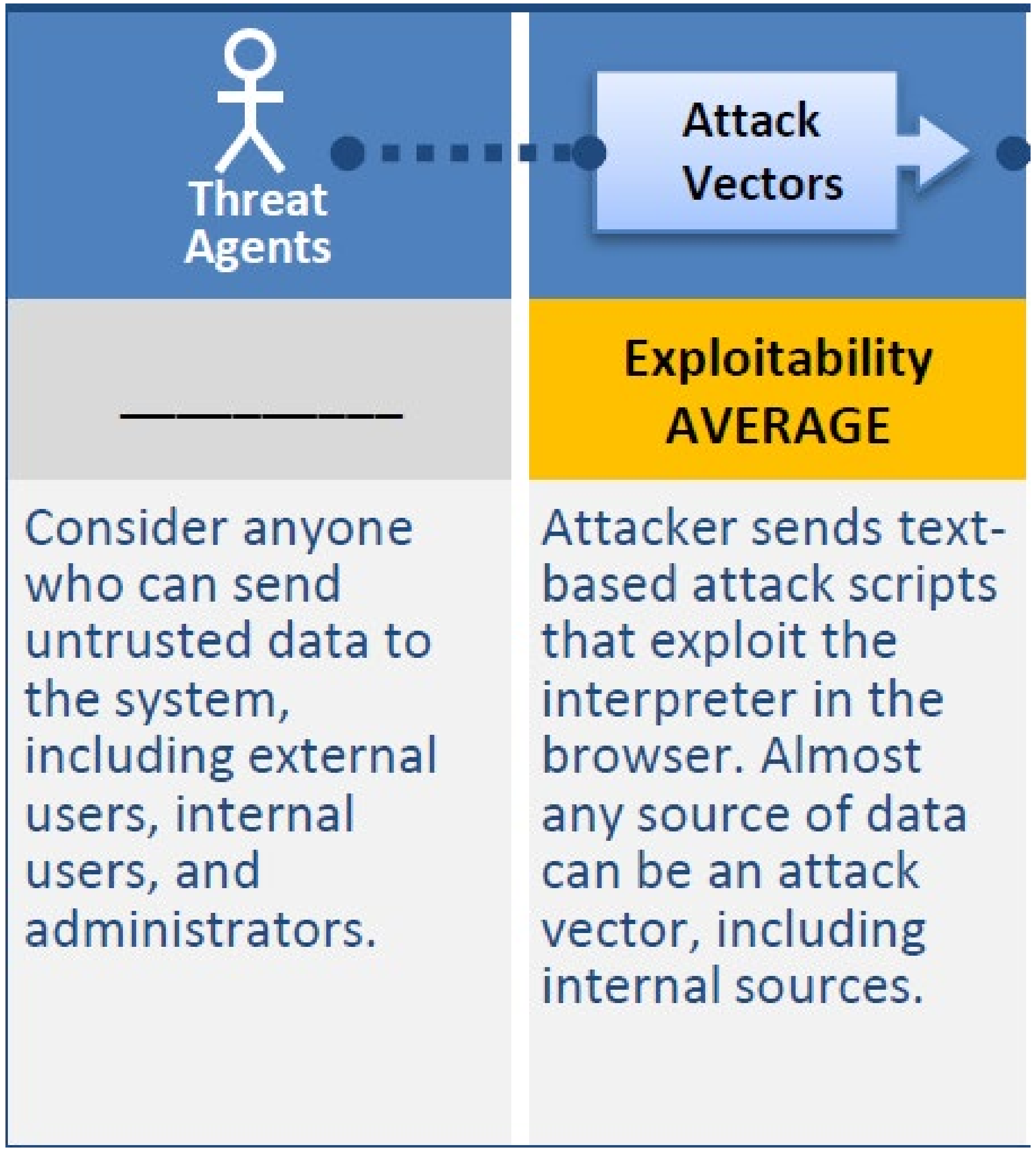
FHO Fachhochschule Ostschweiz

Cross Site Scripting

XSS

2019

Cross Site Scripting

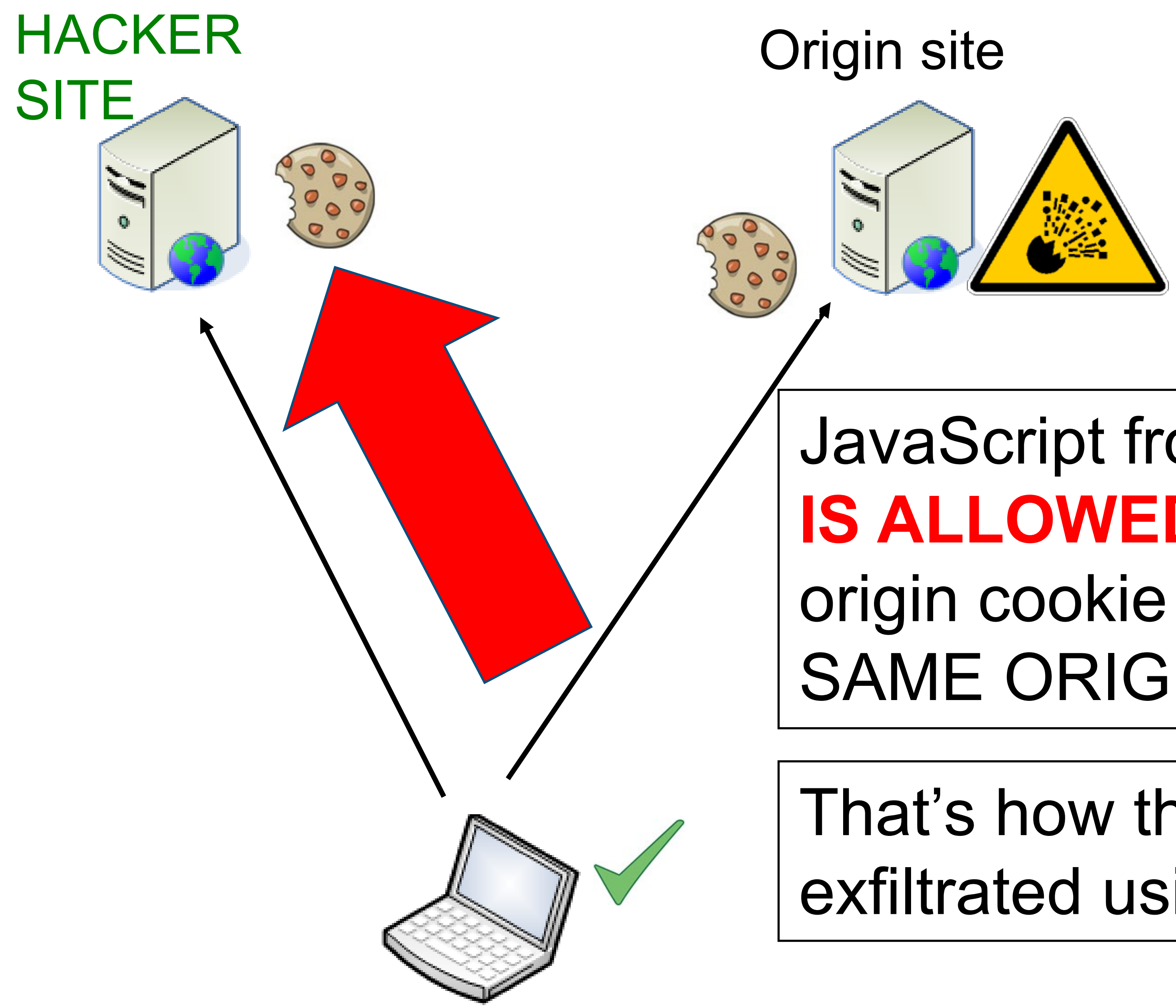


Threat Agents	Attack Vectors
________	Exploitability AVERAGE
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends text-based attack scripts that exploit the interpreter in the browser. Almost any source of data can be an attack vector, including internal sources.

XSS is the most prevalent web application security flaw.

XSS flaws occur when an application includes user supplied data in a page sent to the browser without properly validating or escaping that content.

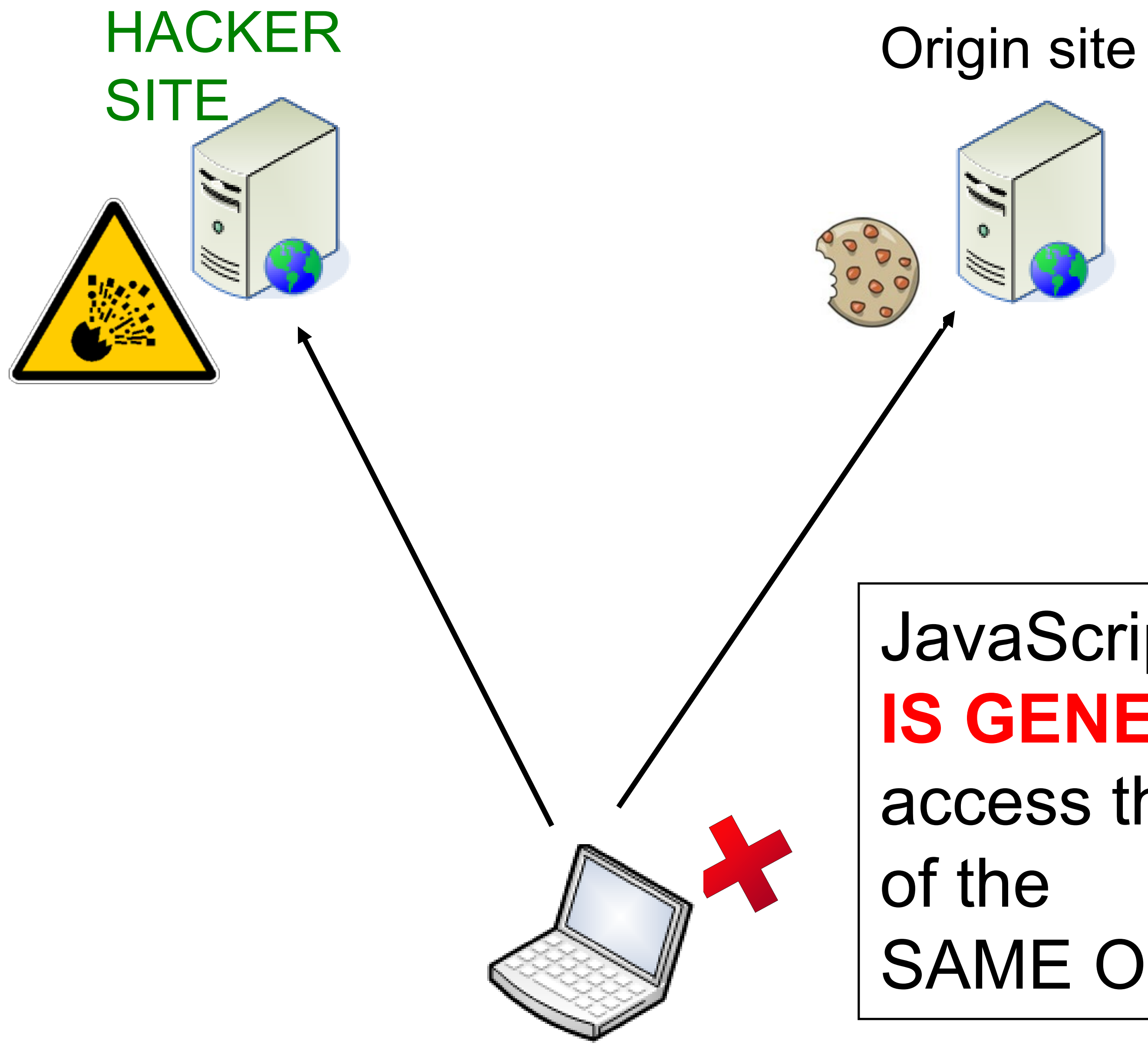
XSS



JavaScript from ORIGIN SITE **IS ALLOWED** to access the origin cookie because of the SAME ORIGIN POLICY

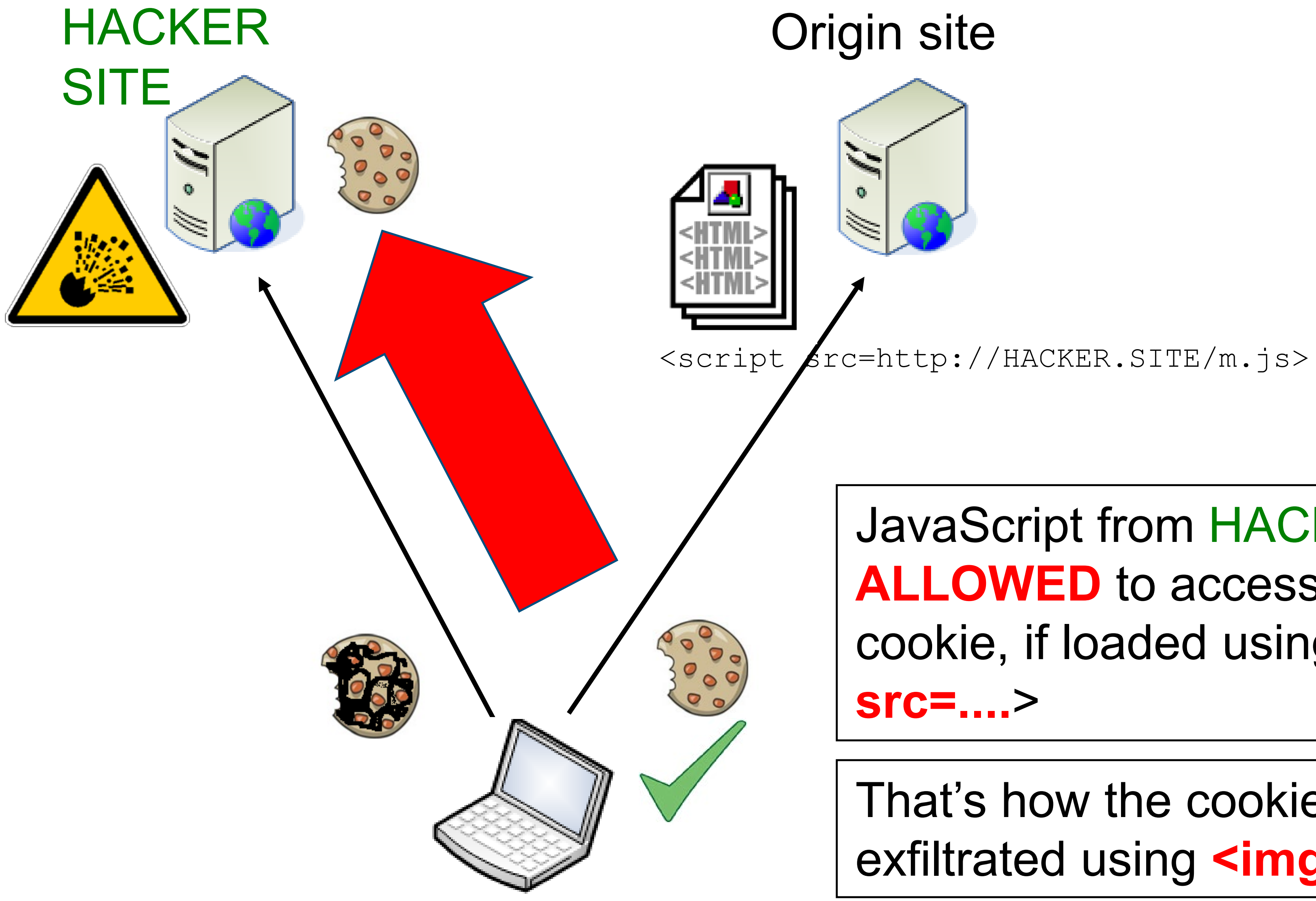
That's how the cookie is exfiltrated using **<img src…>**

XSS



JavaScript from HACKER SITE **IS GENERALLY DENIED** to access the origin cookie because of the
SAME ORIGIN POLICY

XSS with <script src>



JavaScript from HACKER SITE site **IS ALLOWED** to access the origin cookie, if loaded using <**script src=....**>

That's how the cookie is exfiltrated using **<img src…>**

Causes of XSS Flaws

Failure of the application to properly sanitize output to the user’s browser.

Improper trust of of user supplied data.

```
$a = $_GET['search'];
print 'Your search results '.$a;
```

Effects of XSS

Theft of session cookies

Arbitrary HTML or Javascript injection

Exploit injection

Keystroke Logging

BeEF & Metasploit can be used to show effects of XSS

Testing XSS Vulnerabilities

Play around with different test strings in request parameters

```
<script>alert('asdf')</script>
<script>alert(document.cookie)</script>
"<script>alert(document.cookie)</script>
"><script>alert(document.cookie)</script>
'><script>alert(document.cookie)</script>
<img src="http://www.bla.com/image.gif"
 onLoad="alert(document.cookie)">
```

Analyze the page that is being returned

- Search for the presence of test strings
- Check how the characters get filtered or changed
- Find out the problem why XSS does not work and try new test strings

Type of XSS attacks

Stored

- The injected script is permanently stored on the target server (e.g., via forum or blog posts)

Reflected

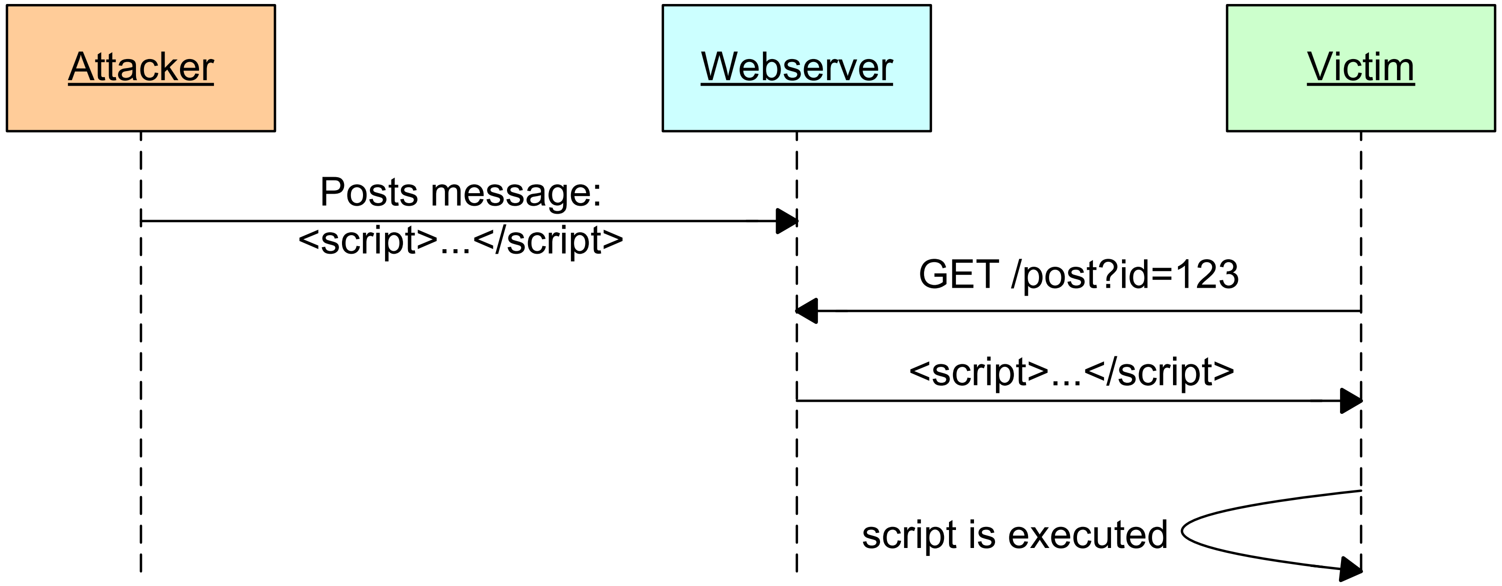
- The script is not stored on the target server
- Attacker usually needs to construct a malicious URL

DOM based

- Attacker also needs to construct a malicious URL
- But the parameter of this URL is not processed by the web server, but executed by the browser directly

Stored XSS

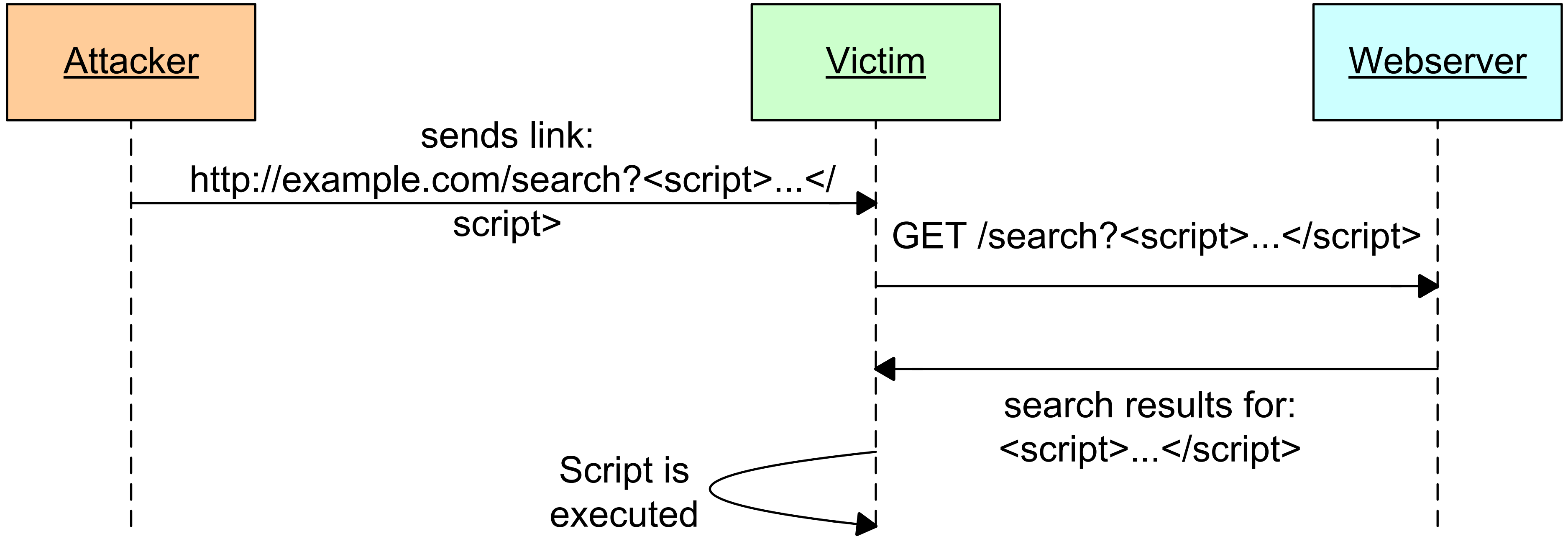
- What is stored XSS?
  - Data provided by a web client is stored in a database. This data is then presented to the user un-encoded
  - Malicious script is rendered more than once
  - XSS worms are based on stored XSS vulnerabilities
  - Typical example: message board



Reflected XSS

What is reflected XSS?

- Data provided by a web client is used immediately by server-side code to generate a page of results for that user
- Attacker has to send a crafted link to the victim
- Typical example: search form


Attacker
Victim
Webserver
sends link: http://example.com/search?<script>...</script>
GET /search?<script>...</script>
search results for: <script>...</script>
Script is executed

DOM Based XSS

Unlike the stored and reflected XSS, DOM based XSS does not require the web server to receive the malicious XSS payload.

Instead, in a DOM-based XSS, the attack payload is embedded in the DOM object in the victim’s browser used by the original client side script, so that the client side code runs in an “unexpected” manner.

DOM Based XSS

Example page

```
  <HTML><TITLE>Welcome!</TITLE>
Hi <SCRIPT>
var pos = document.URL.indexOf("name=") + 5;
document.write(document.URL.substring(pos,document.URL.length));
</SCRIPT>
</HTML>
```

Works fine with this URL

```
  http://www.example.com/welcome.html?name=Joe
```

But what about this one?

```
  http://www.example.com/welcome.html?name=
<script>alert(document.cookie)</script>
```

Countermeasures
Cross Site Scripting

XSS Prevention

Solution 1: Secure Programming

- Convert user input into HTML entities
- when storing user input (input encoding)
- when loading data from db (output encoding)

Solution 2: HTTP Response Header

- X-XSS-Protection (against reflected XSS)

Solution 3: Cookie Secure

- This will prevent JS from accessing document.cookie

Solution 4: CSP (Content Security Policy)

- script-src ‘self’

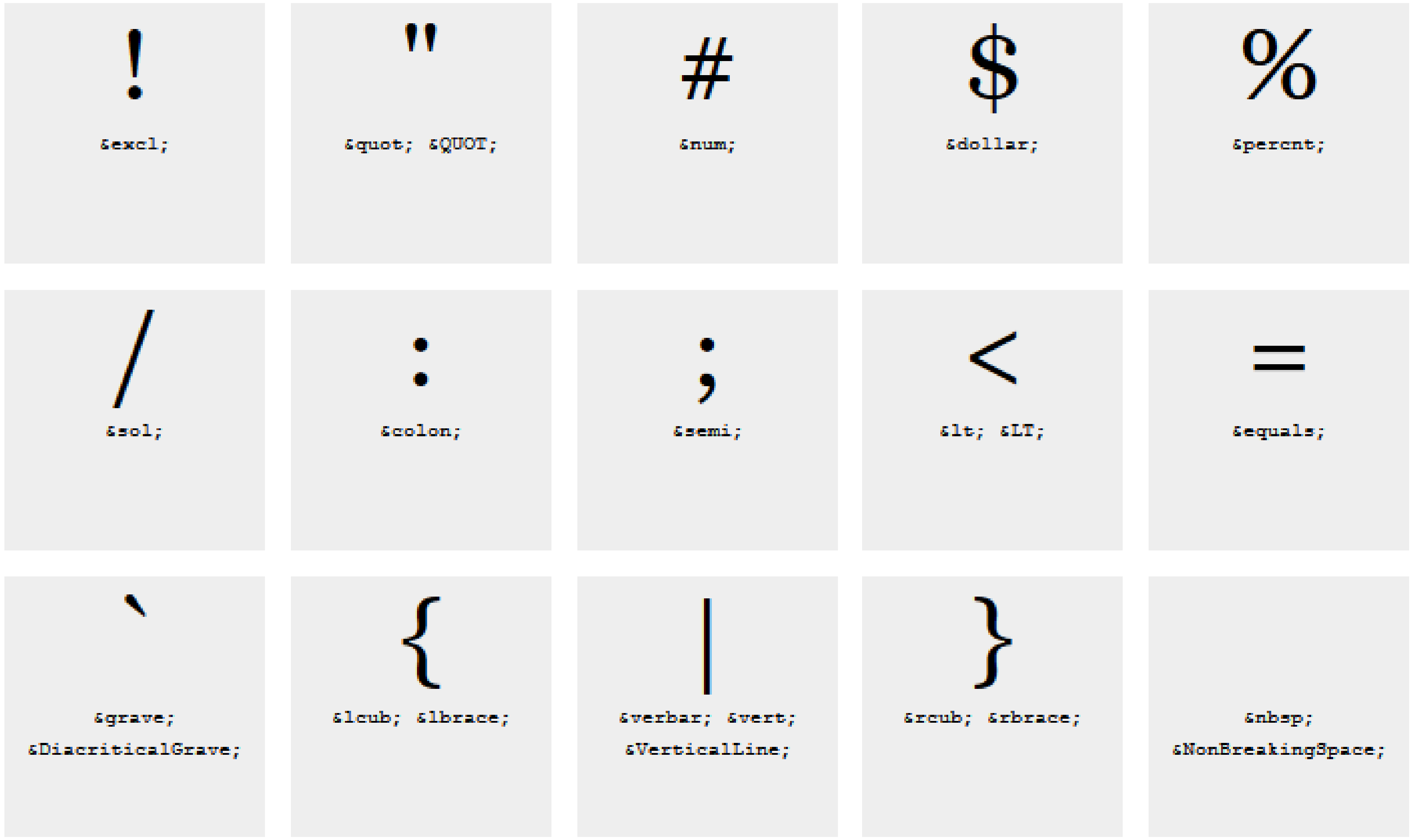
Solution 5: WAF (Web Application Firewall)

- Second Line of Defense

Solution 1: Secure Programming

Convert user provided input or insecure output into HTML entities

```
<  →  &lt;
>  →  &gt;
"  →  &quot;
'  →  &apos;
```



! &excl;	" &quot; &QUOT;	# &num;	$ &dollar;	% &percnt;
/ &sol;	: &colon;	; &semi;	< &lt; &LT;	= &equals;
` &grave; &DiacriticalGrave;	{ &lcub; &lbrace;	| | | &VerticalLine;	} &rcub; &rbrace;	&NonBreakingSpace;

https://dev.w3.org/html5/html-author/charref

Filtering Strings is not recommended!!!

Not recommended!!!! Use HTML entities instead

Filtering Approach

- Input validation on characters
  - Do not accept "dangerous" characters (e.g. <)
  - Delete "dangerous" characters from request
  - Transform "dangerous" characters into HTML entities
- Input validation on strings / tags
  - Do not accept "dangerous" tags (e.g. <script>)
  - Delete "dangerous" tags from request
  - Transform "dangerous" tags into HTML entities

XSS Prevention in PHP

Use htmlentities();

```
VULNERABLE
Replace echo $_POST['message’]

SECURE
htmlentities($_POST['message'], ENT_COMPAT, 'UTF-8');;
```

This escapes the user provided input into html entities

```
< => &lt;
> => &gt;
...
```

XSS Prevention in NodeJS

Enable Template Engine Auto Escape (server.js)

```
swig.init({
    root: __dirname + "/app/views",
    autoescape: true // default value
});
Escape Ouput using “sanitizer” based on Google Caja
sanitizer.escape(poisonous_var);
```

Escape Ouput using “sanitizer” based on Google Caja

```
sanitizer.escape(poisonous_var);
< => &lt;
> => &gt;
...
```

Input Validation: Check incoming data for valid contents.

XSS Prevention in AngularJS

By default, AngularJS handles output encoding in a secure way:

- The directive “ng-bind” simply performs context-sensitive encoding on all data:

```
<span ng-bind="helloMessage"></span>
```

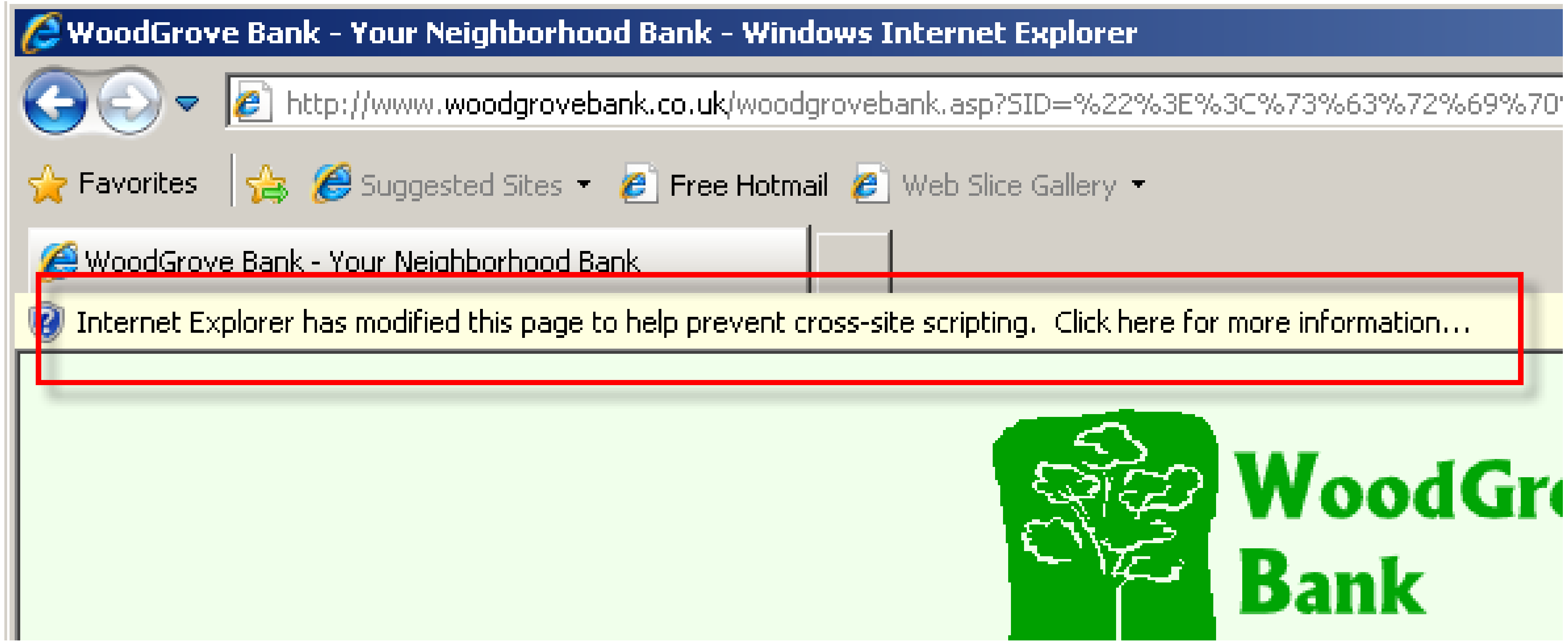
Example:

`<script>alert(1)</script>` is automatically converted to `&lt;script&gt;alert(1)&lt;/script&gt;` by AngularJS for safe integration into the HTML context

Solution 2: Client Side Prevention

Internet Explorer

- Anti XSS filter built in since version 8
- Active by default
- Only helpful for reflected XSS attacks


WoodGrove Bank - Your Neighborhood Bank - Windows Internet Explorer

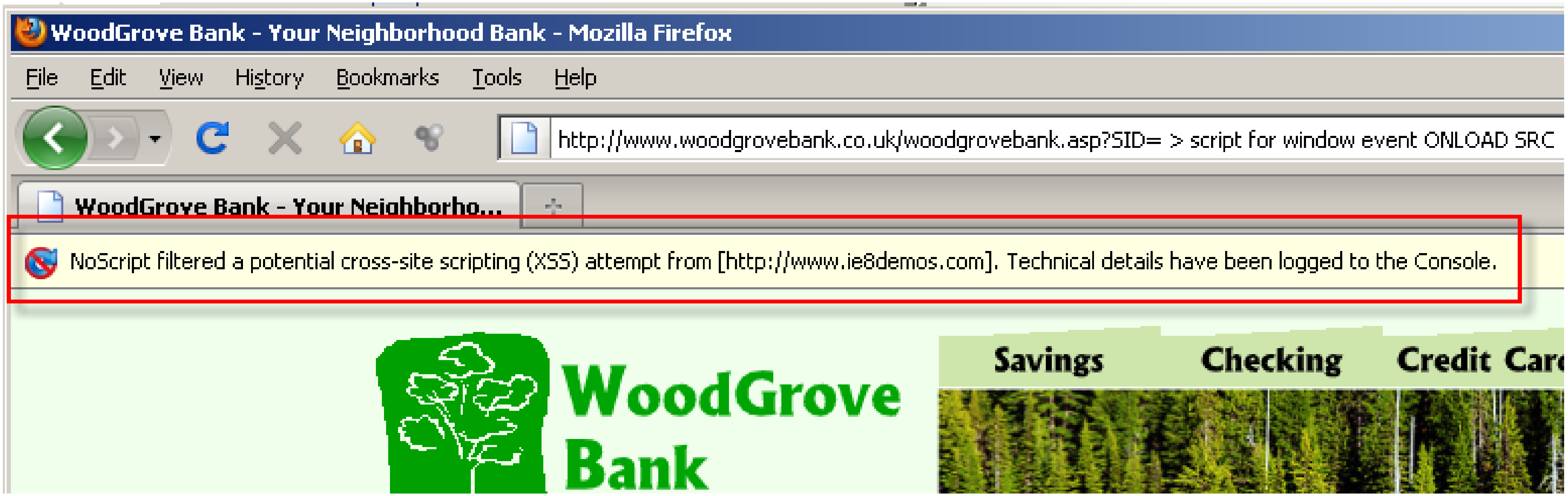
http://www.woodgrovebank.co.uk/woodgrovebank.asp?SID=%22%3E%3C%73%63%72%69%70

Internet Explorer has modified this page to help prevent cross-site scripting. Click here for more information...

Solution 2: Client Side Prevention

Firefox

- NoScript plugin is very powerful
- Possible to forbid JavaScript in general
- Other features include XSS prevention (reflected), Clickjacking prevention, forbid active content (Java, Silverlight, Flash,...)


WoodGrove Bank - Your Neighborhood Bank - Mozilla Firefox

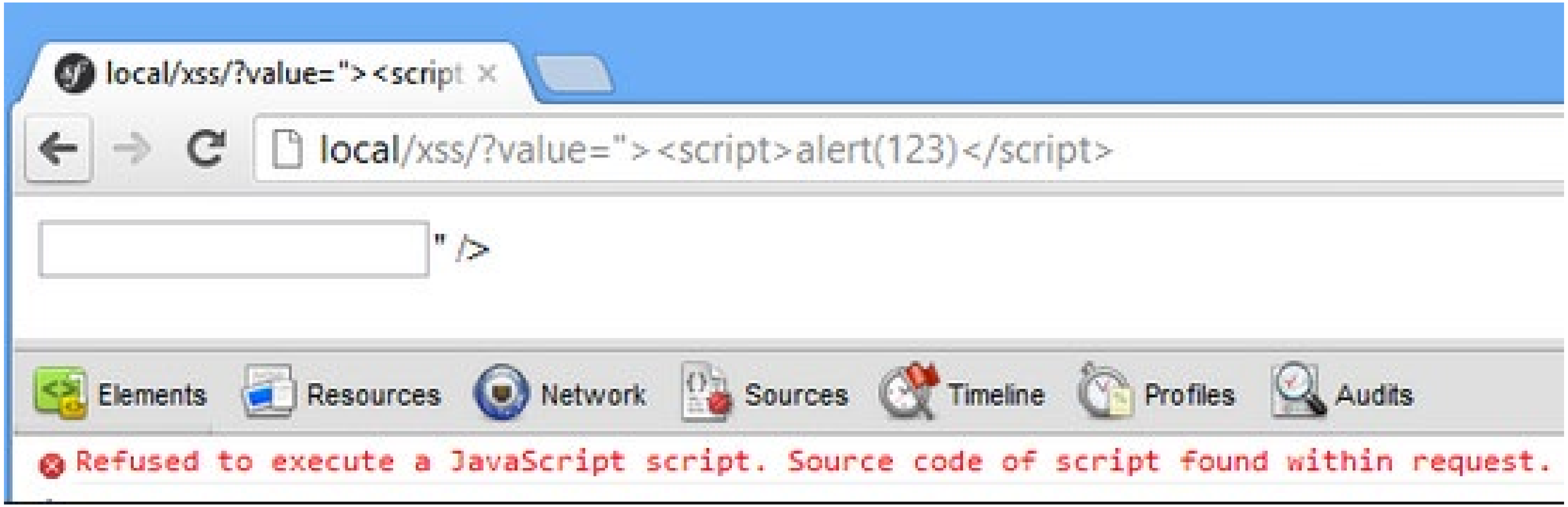
http://www.woodgrovebank.co.uk/woodgrovebank.asp?SID= > script for window event ONLOAD SRC

NoScript filtered a potential cross-site scripting (XSS) attempt from [http://www.ie8demos.com]. Technical details have been logged to the Console.

Solution 2: X-XSS-Protection

X-XSS-Protection: 1; mode=block

- Asks Internet Explorer >8 to render a blank page if a XSS attempt is detected
- Stricter policy than by default



```
local/xss/?value="><script>alert(123)</script>
```

" />

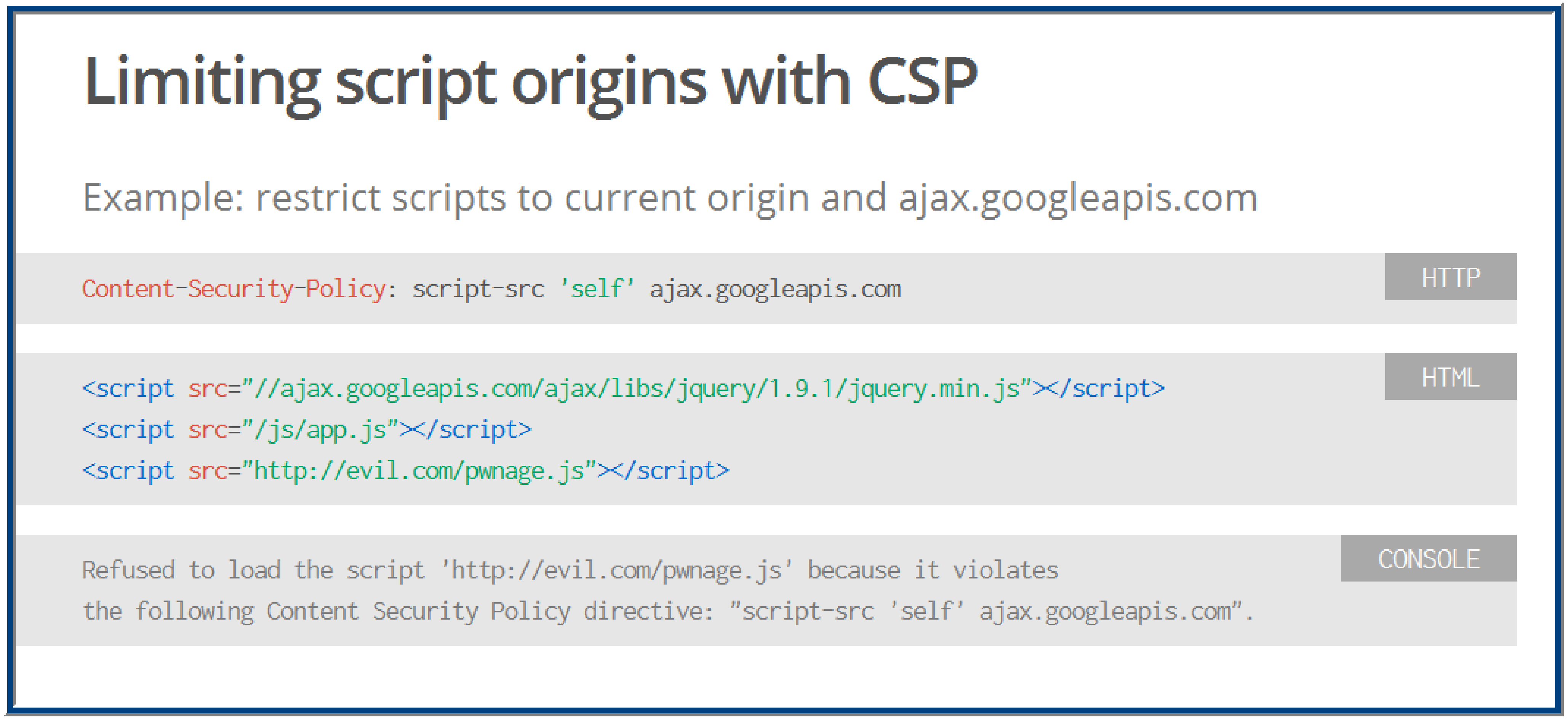
```
Refused to execute a JavaScript script. Source code of script found within request.
```

Solution 3: HttpOnly

Session Hijacking Mitigation

- Set Cookie with HttpOnly
- If set, document.cookie is empty()

Solution 4: Content Security Policy



Limiting script origins with CSP

Example: restrict scripts to current origin and ajax.googleapis.com

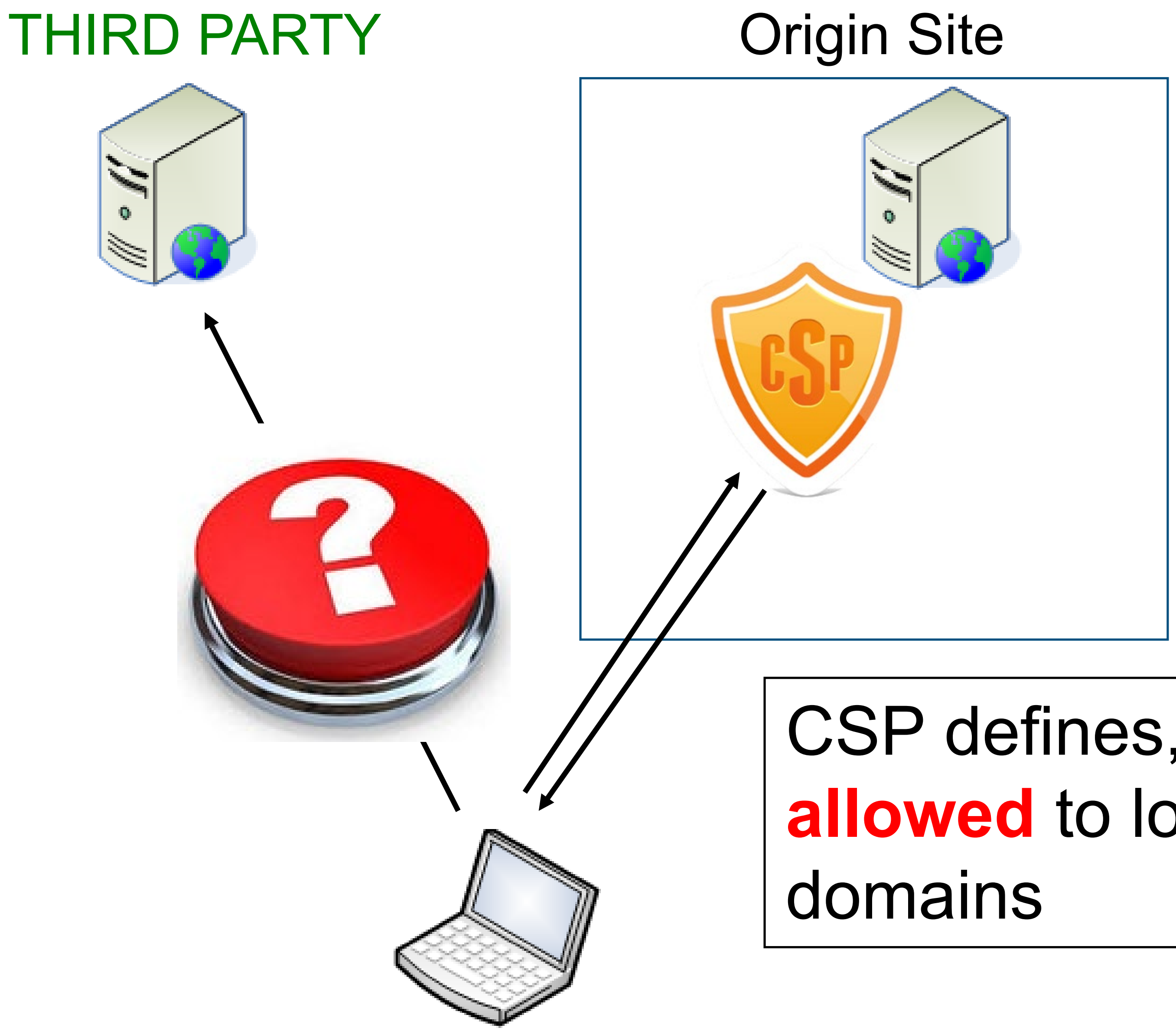
```
Content-Security-Policy: script-src 'self' ajax.googleapis.com
```

```
<script src="//ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"></script>
<script src="/js/app.js"></script>
<script src="http://evil.com/pwnage.js"></script>
```

```
Refused to load the script 'http://evil.com/pwnage.js' because it violates
the following Content Security Policy directive: "script-src 'self' ajax.googleapis.com".
```

http://benvinegar.github.io/csp-talk-2013/

Content Security Policy



Please view the full blown CSP theory (later in this course)

CSP defines, what the browser is **allowed** to load from other domains

COMPASS
SECURITY