



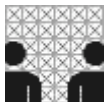
Universität Hamburg

# **SessionSafe: Implementing XSS Immune SessionHandling**

**ESORICS '06**

20. September 2006

**Martin Johns**



**Fachbereich Informatik**

**SVS – Sicherheit in Verteilten Systemen**



## Me, myself and I

- **Martin Johns**
- **johns at informatik.uni-hamburg.de**
- **Security researcher at the University of Hamburg**
- **Member of the secologic project**
  - ◆ **Research project carried out by SAP, Commerzbank, Eurosec and the University of Hamburg**
  - ◆ **Sponsored by the German Ministry of Technology (BMWV)**
  - ◆ **Goal: Improving software security**
  - ◆ **Visit us at <http://www.secologic.org>**



- **Web Application Session Management**
- **Cross Site Scripting**
- **Protection Approaches**
- **Conclusion**



- **Web Application Session Management**
- Cross Site Scripting
- Protection Approaches
- Conclusion



## Session management in http

- As http is stateless a web application has to implement its own session tracking mechanism
- An authenticated user gets assigned a session ID

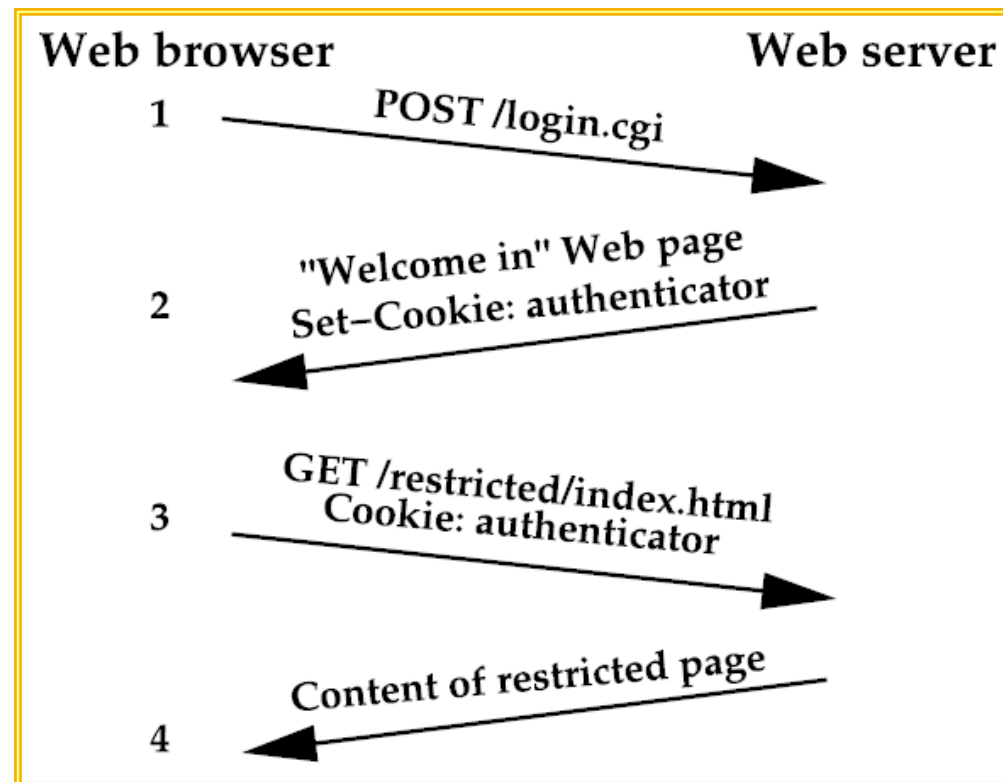
### Three common methods

- URL rewriting
- Form based session IDs
- Cookies

## Cookies are used to maintain state

- Stored on the client's browser
- Send to the server along with matching requests
- Example: A request to `www.example.org/index.html` will send all cookies that are stored for
  - ◆ `www.example.org` and
  - ◆ `example.org`
- Cookies stored for other (sub-)domains are not sent

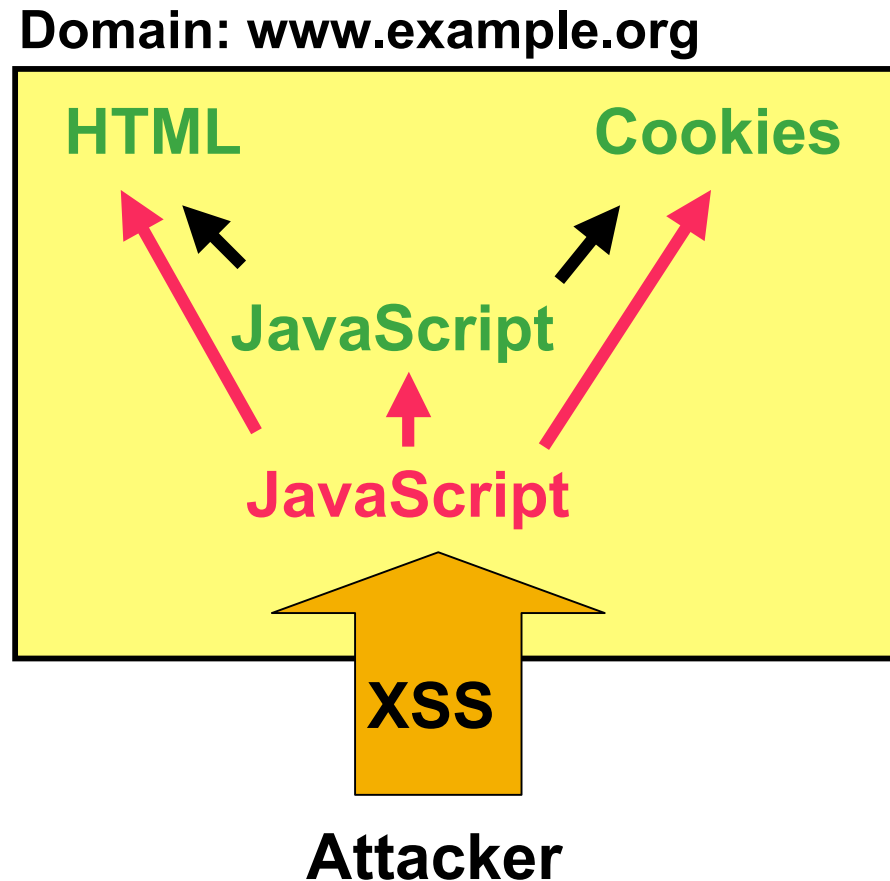
- The server sets a cookie at the client's browser after the authentication form
- This cookie is automatically included in all further requests
- The client's requests are treated as authorized as long as this cookie is valid





- Web Application Session Management
- **Cross Site Scripting**
- Protection Approaches
- Conclusion

- An attacker includes malicious JavaScript code into a webpage
- This code is executed in the victim's browser session

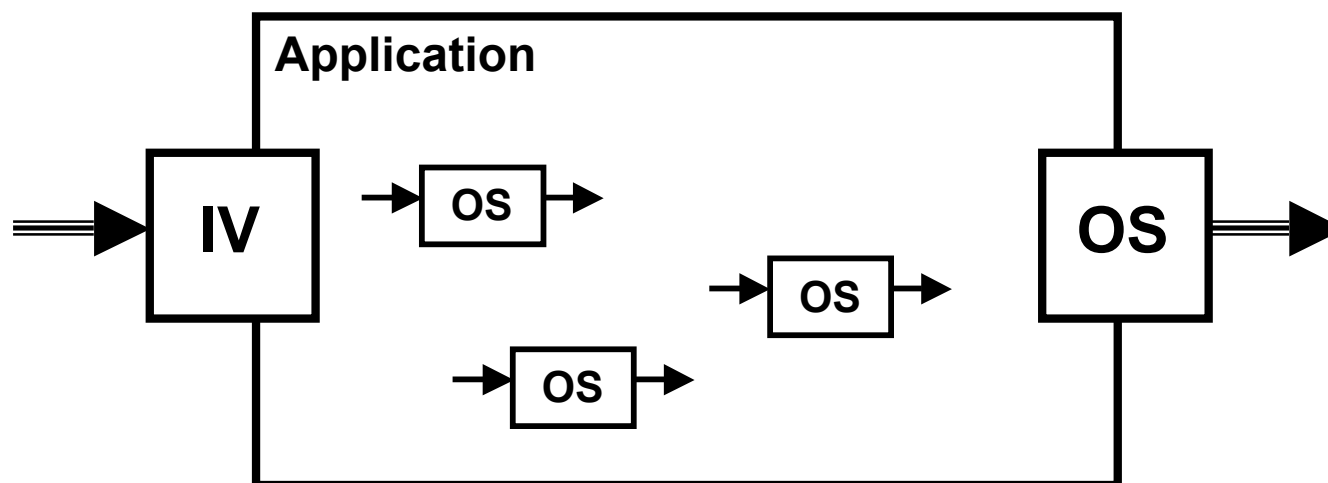


## Input Validation (IV):

- All untrusted input data is checked against predefined specification

## Output Sanitization (OS):

- “Cleaning” of HTML
  - ◆ Removing malicious data
  - ◆ Encoding malicious data





## This approach fails frequently

### Reported XSS vulnerabilities:

- In the first six months of 2006 over 550 XSS vulnerabilities were reported to BugTraq
- All major frameworks were affected (php, j2ee, asp.NET,...)

### Affected websites:

- Google (16.09.2006)
- MSN (25.06.2006)
- Yahoo (28.04.2006)
- PayPal.com (23.07.2006)
- Amazon.com (27.06.2006)
- Verisign (14.08.2006)
- ibm.com, sun.com, ebay.com,...



## Why does the approach fail?

### Input Validation (IV):

- IV is not always possible
- Has to be tailored to a specific scenario
- Scattered code

### Output Sanitization (OS):

- Context dependent
  - ◆ e.g., from data origin, data destiny, role of the application's user
  - ◆ Heterogeneous technologies lead to losing data's context

**IV/OS is not centralized enforceable**



## Why does the approach fail? (II)

Furthermore, there are XSS attacks that cannot be prevented this way:

- DOM based XSS
- Server induced XSS
  - ◆ Expect handler vulnerability

To make things worse:

- One single XSS vulnerability compromises the whole web application

# Can we defend a web application against session hijacking after a successful XSS attack?

### Approach:

- **Determining the available security mechanisms in web architecture**
- **Classification of XSS session hijacking attacks**
- **Investigation of the basic requirements of these attack classes**
- **Applying the security mechanisms to revoke those requirements**



- Web Application Session Management
- Cross Site Scripting
- **Protection Approaches**
- Conclusion



## The same origin policy

- Defines basic access rights in http
- Two elements have the “same origin” if the
  - ◆ protocol,
  - ◆ port,
  - ◆ and hostare the same for both elements
- Access rights
  - ◆ JavaScript has unlimited access to all elements, which match the script’s origin
  - ◆ Cookies are sent along with all requests with matching target

## JavaScript Closures



# XSS Session-Hijacking-Attack Classes

## Attack class

## Requirement

- **SessionID Theft**

- ◆ Leakage of the SID



- **Accessibility of the SID by the malicious JavaScript**

- Deferred Loading**

- **Browser Hijacking**

- ◆ Execution of the attack within the browser

- **Pre-knowledge of the application's URLs**

- One-Time URLs**



- **Background XSS Propagation**

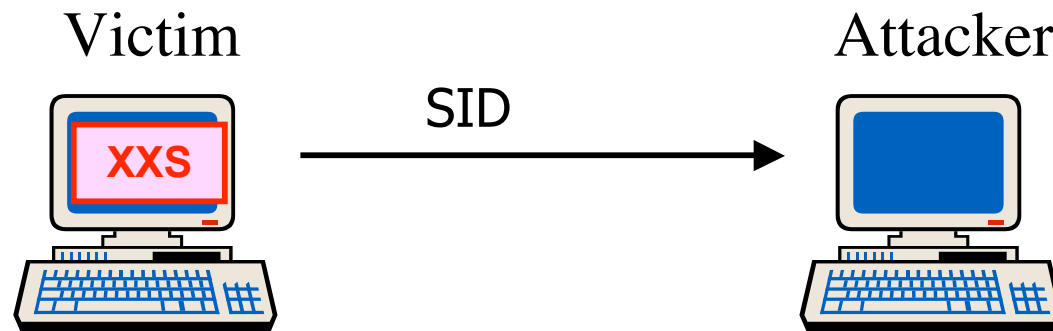
- ◆ Controlling the attack with an outside entity

- **Implicit trust between browser documents with the same origin**

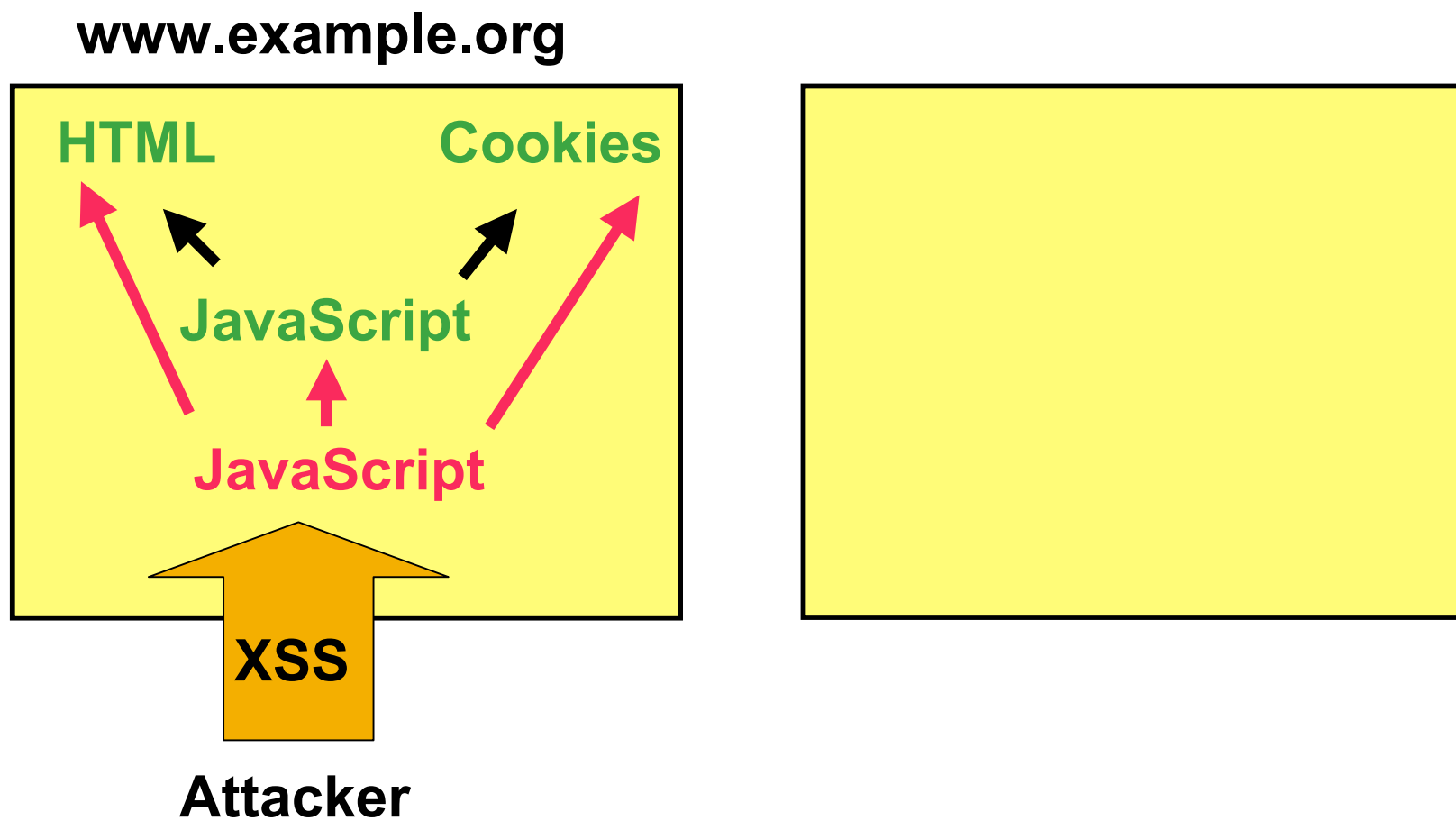
- Subdomain Switching**

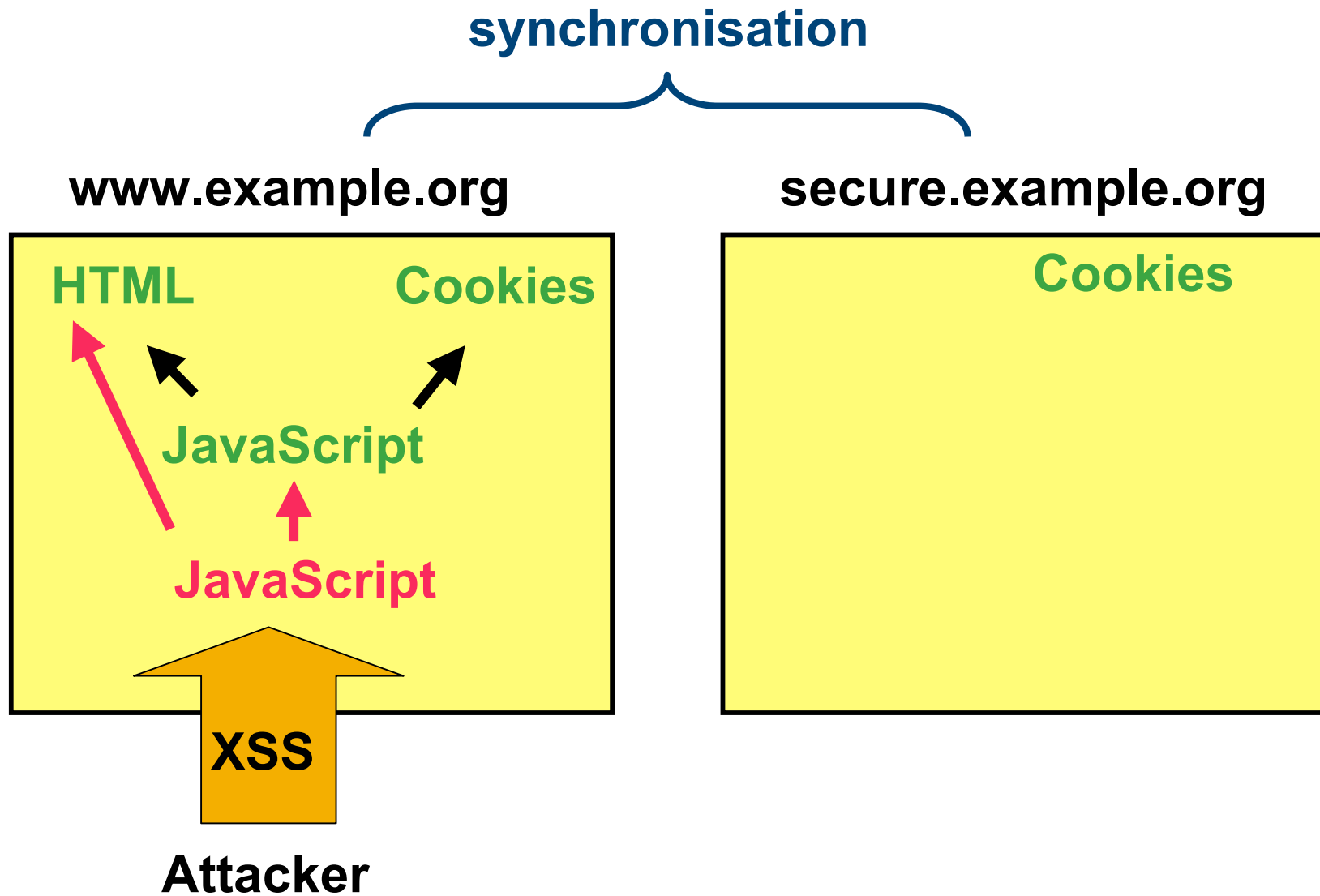
## This is XSS 101

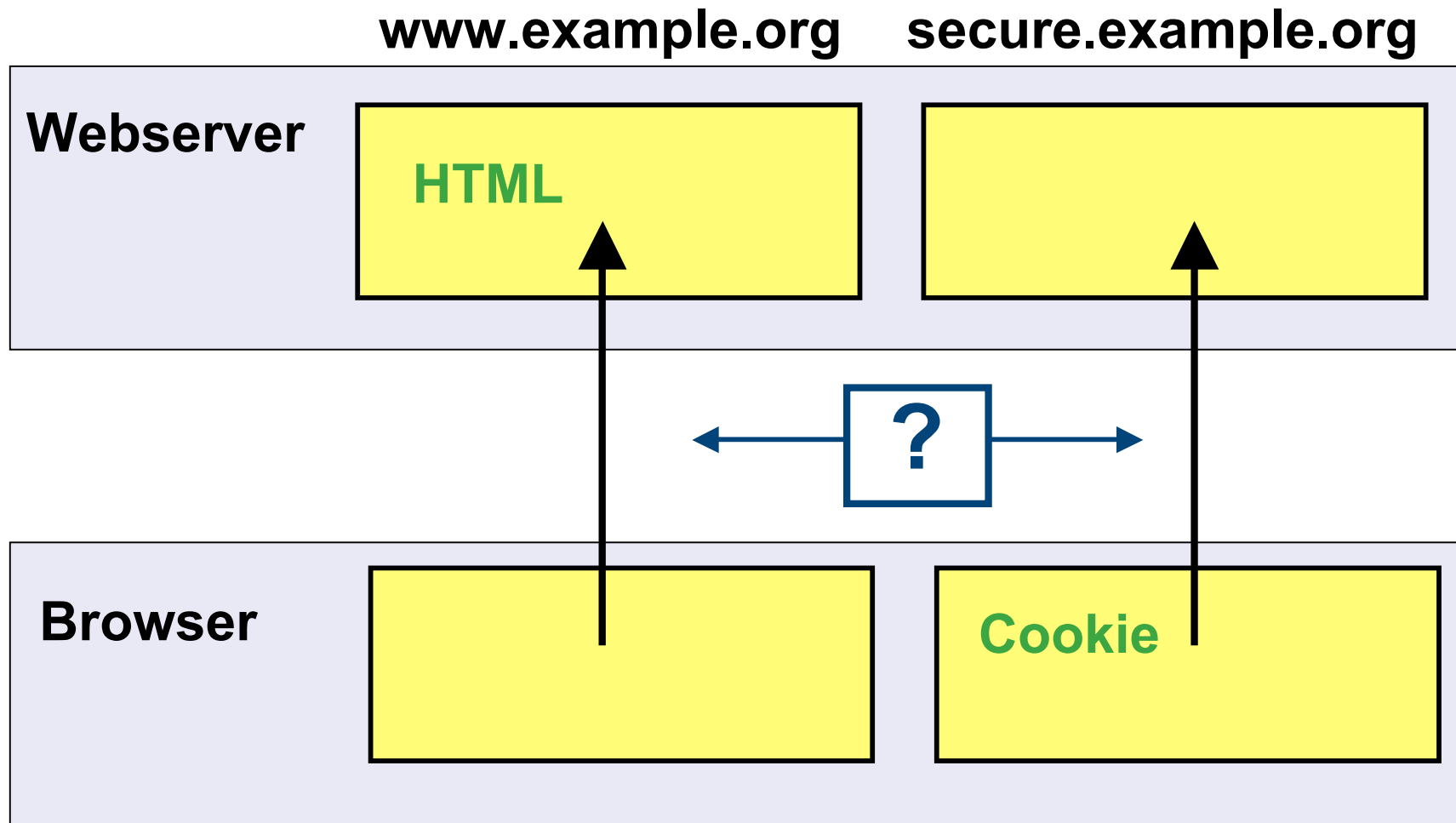
- The malicious JavaScript obtains the SID
  - ◆ All presented SID mechanisms (URL, Form, Cookie) are vulnerable
- and communicates it to the attacker.
- The attacker is now able to impersonate the victim



- Requirement: Malicious JavaScript has to be able to access the SID

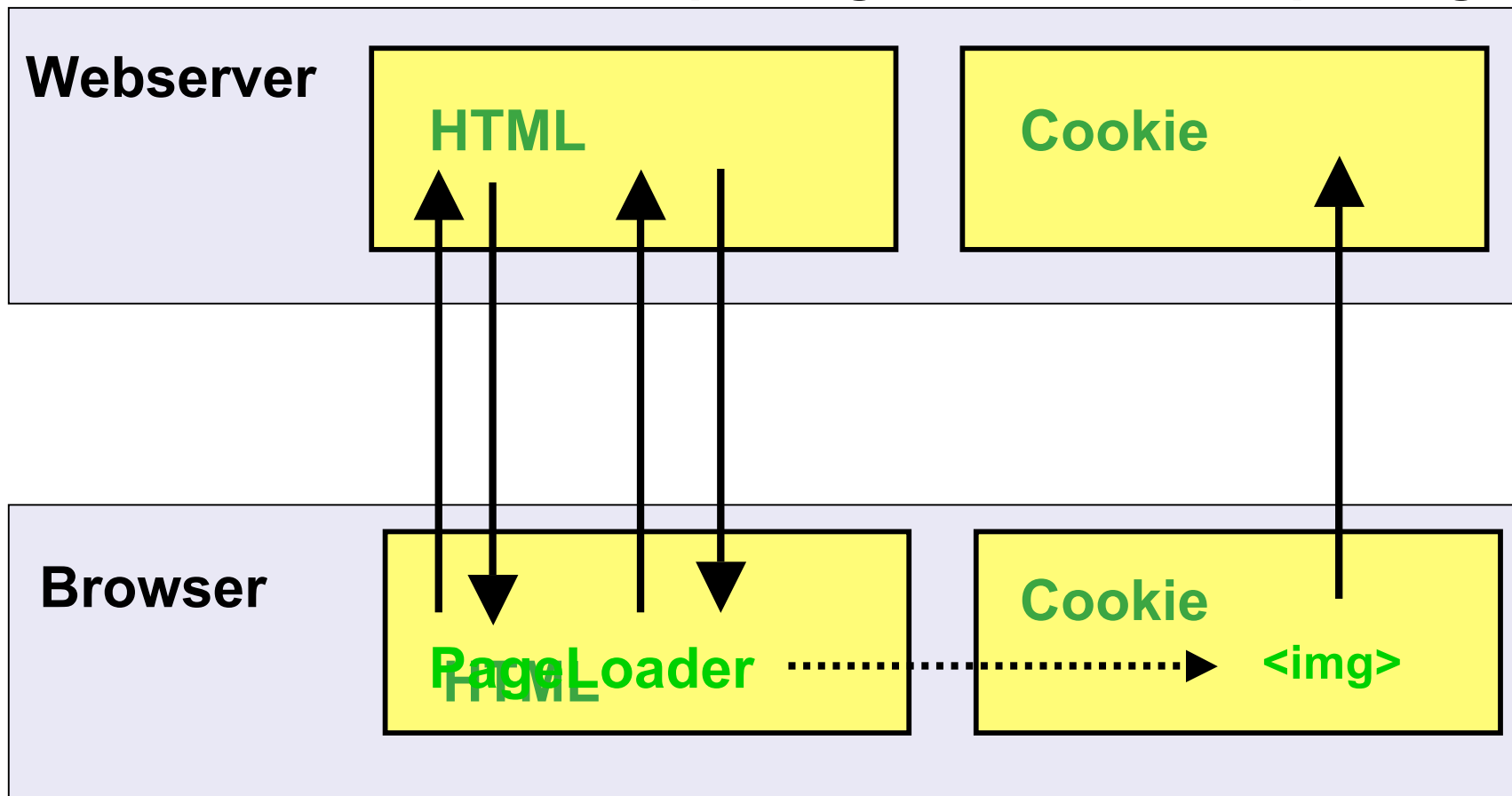




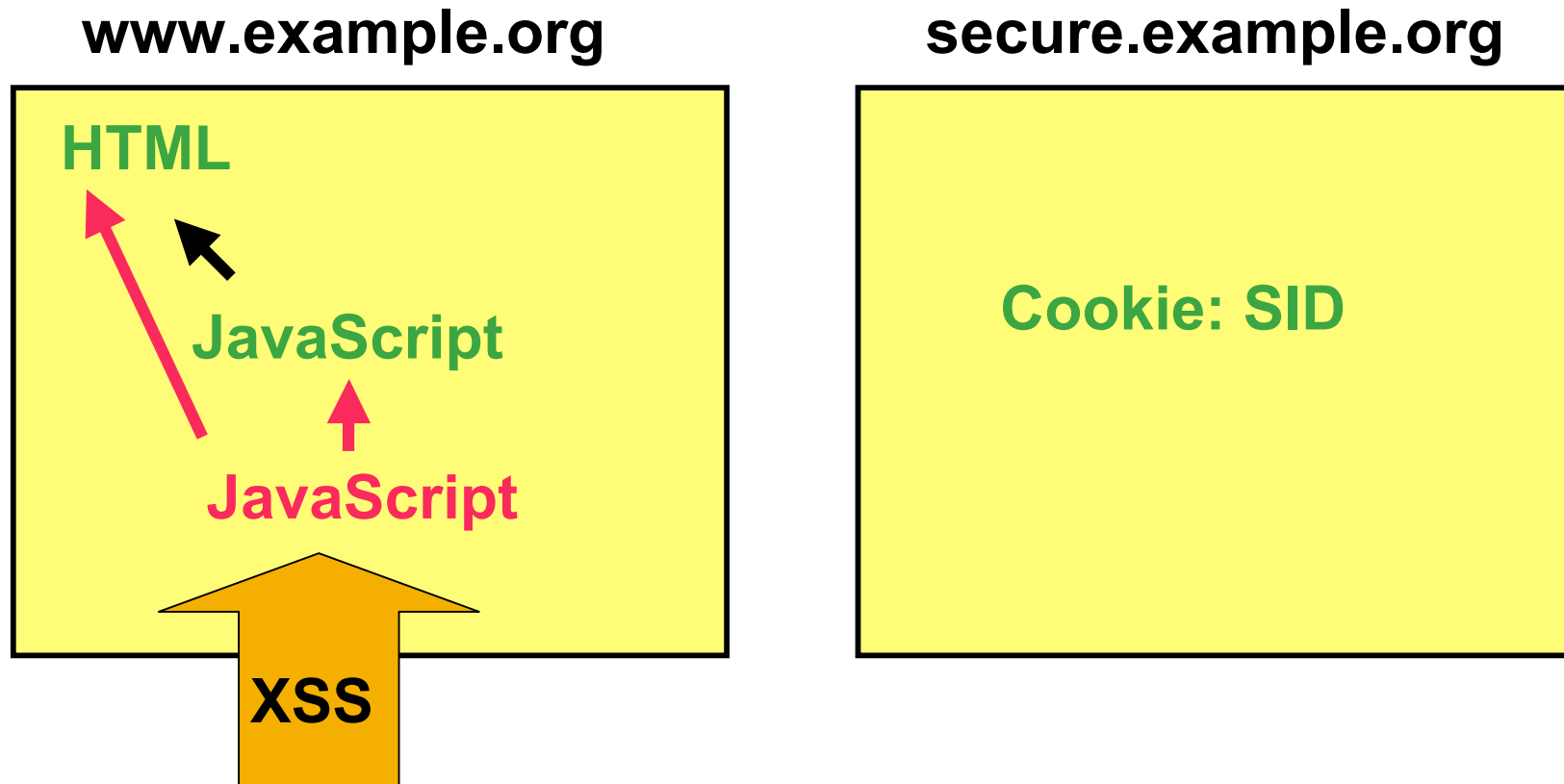


synchronisation

www.example.org    secure.example.org



- Because of the Deferred Loading process, the SID cookie and potential XSSs belong to different (sub)domains
- For this reason the malicious script has no access to the SID





## Implementation

- **Transparent implementation for J2EE**
- **Serverside proxy**
- **The proxy executes the Deferred Loading process before passing the requests to the server**



- Web Application Session Management
- Cross Site Scripting
- Protection Approaches
- **Conclusion**



### Implementation

- **Transparent server-side proxy for J2EE**

### Limitations

- **The XSS still has full control over the vulnerable webpage**
- **Further drawbacks:**
  - ◆ **AJAX has to be treated separately**
  - ◆ **One-Time URLs may break the “Back” and “Reload” button**

### Protection:

- **Good protection against “hidden” attacks**
- **XSS may still be able to steal passwords by spoofing authentication dialogs**
  - ◆ **Client Side SSL authentication could help**



**Thank you**

**Thanks for listening.**

**Questions? Comments?**