IAIK TU Graz

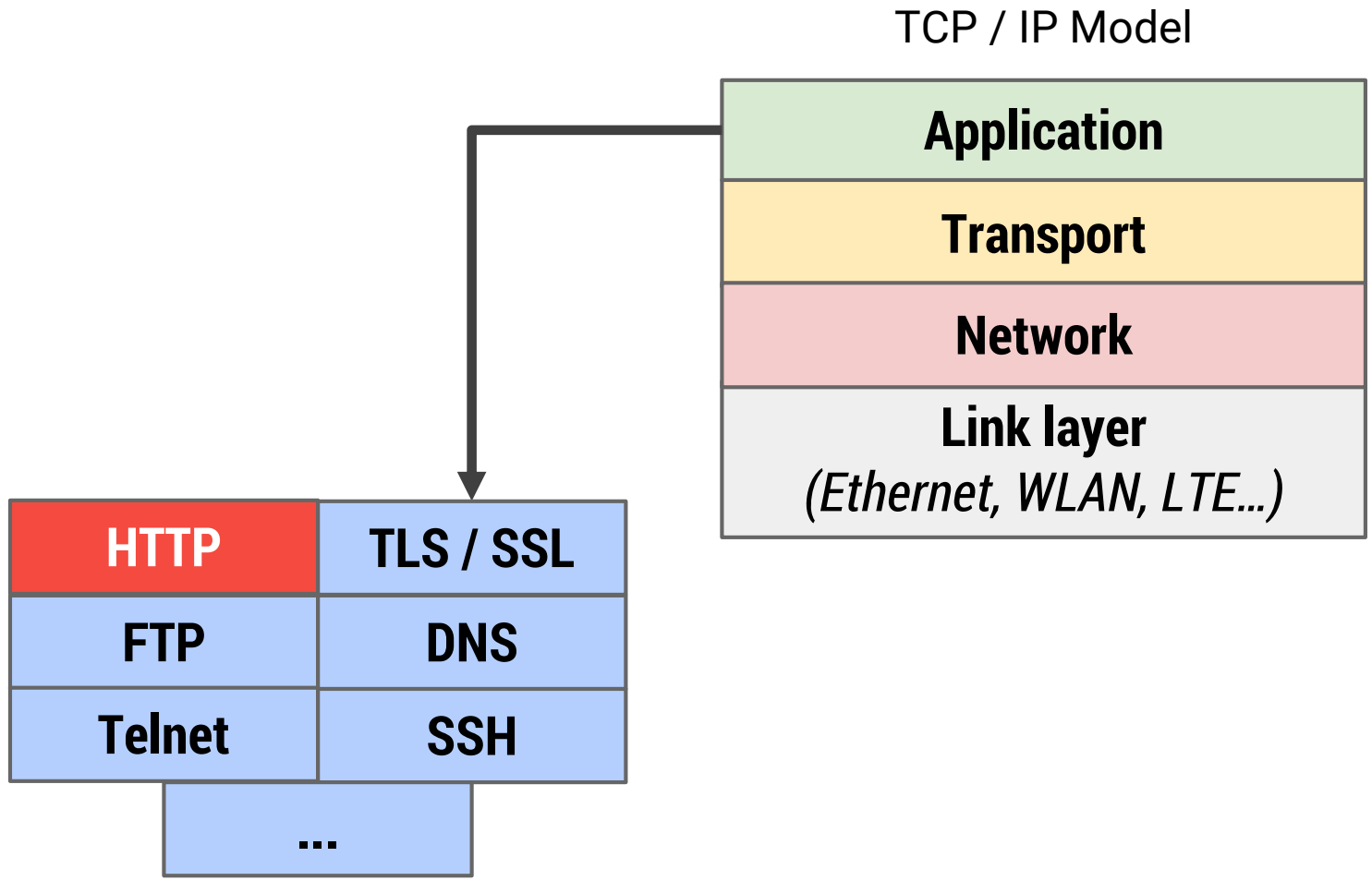
Web (Browser) Security

Information Security 2019

Johannes Feichtner
johannes.feichtner@iaik.tugraz.at

Outline

- HTTP Sessions
- Same Origin Policy
- Bypassing SOP
  - AJAX Proxy
  - JSONP
  - CORS
  - CSP
- Client-Side Attacks
  - Session Stealing
  - XSS & CSRF



Review: HTTP

- Simple (stateless) request / response protocol
  - Client opens TCP connection, requests document
  - Server responds with document
  - Client closes TCP connection
- Multiple versions (HTTP 0.9 – HTTP/2)
- Advanced communication
  - AJAX, COMET (AJAX with Long Polling), WebSockets

HTTP Sessions

Sessions in HTTP?

HTTP is stateless!

This means…

- Any request is considered unrelated to prior ones
- Server does not maintain session information
  - E.g. does not know if you are logged in or not within some web application

Q: Now how to (re-)identify users?

A: Session IDs!

= Unique identifier transmitted for each request / maintained by session

Session IDs

Comparable with short-lived access key to some resource

→ **Whoever knows the session ID has access (even without credentials)!**

Requirements

- Session ID should be randomly chosen, unique, large key space
- Not predictable or from weak random number generator

How to pass session IDs?

- Via rewritten URLs = Session ID in URL
- Via cookies = Stored in HTTP headers
- Via hidden tags in HTML pages
- Via tokens sent in header

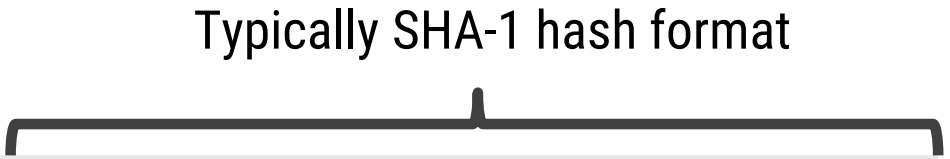
Which method is best / most secure?
Depends on implementation!

URL Rewriting

Idea

Encode session ID as parameter into URL

Example




```
https://iaik.tugraz.at/admin.php?logout=0&s=fa392522a05d07ed1512020627d976a2
```

→ To be sent with every request!

Problems:

- Webservers log requests → Session IDs also!
- Browser history contains login information
- Users who copy URLs also copy session IDs
- Session ID exposed in HTTP referer header

URL Rewriting

Problem

HTTP Requests typically send a referer field with originating URL

Example

When clicking a link on *http://wetter.orf.at/steiermark* the request to *news.orf.at* contains a referer header with the request origin

Consequence

If origin URL has a session ID, the referer leaks it to the clicked page!

```
GET / HTTP/1.1
Host: news.orf.at
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64; rv:46.0) Gecko/20100101 Firefox/46.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Referer: http://wetter.orf.at/steiermark/
Cookie: vote-2293=cast; vote-2563=cast; Vietnam2Session=127.0.0.129.27.152.15use6cc0fwxk
Connection: keep-alive
If-None-Match: "c8eVtjGDrkYV/bIQZfz+0w=="

HTTP/1.1 200 OK
Date: Fri, 20 May 2016 15:07:47 GMT
Server: Jetty(6.1.22)
X-Cache: HIT from localhost
ETag: "xbgxtPN12o+zM9ctUwXF0Q=="
Content-Length: 23726
Content-Type: text/html; charset=utf-8
X-Uncompressed-Size: 104961
Content-Encoding: gzip
Cache-Control: max-age=0
Expires: Fri, 20 May 2016 15:07:47 GMT
Accept-Ranges: none
Connection: close

..........}.r.G.....'lK1.H....YR")Y..CP..O.`....B.`..  .....>.......=..|.7..l~.U.
...H...'." t.oV.WY._....g....|W..(T[_.....{..:fNu.{o.....2....R.nd2..}mS.....nmcN-
```

Hidden Fields

Setup

- Form to send POST request to server
- Hidden input field (not visible to user – only in source code)

```
<form action="admin_customers.php" method="post" enctype="application/x-www-form-urlencoded">
    <input type="hidden" name="s" value="e2a60422bfce50f7791eea89b612bce3" />
    <input type="hidden" name="action" value="add" />
    <input type="hidden" name="send" value="send" />

    <table class="full"><tr><td>
        <input type="text" name="new_loginname" id="new_loginname" value="" />
```

→ URL does not contain any session info anymore

→ Contained in body of POST request

→ Session ID / user has to be inserted dynamically into form

Cookies

Different Types

- **Session** cookies: No expiration date, valid until browser closed
- **Persistent** cookies: Valid until expiration date
- **Third-party** cookies: Page sets cookie for another domain
- **Supercookies**: Set for entire TLD (e.g. .at) → sent to app.at and attacker.at. Potential security flaw → often blocked!
- **Zombie cookies**: Recreated after deletion from another storage, e.g. Flash or HTML5 storage

Sent within HTTP header

```
GET / HTTP/1.1
Host: orf.at
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64; rv:46.0) Gecko/20100101 Firefox/46.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Cookie: vote-2293=cast; vote-2563=cast; HopSession=127.0.0.129.27.152.ej3ly0nnpu92; Vietnam2Session=127.0.0.129.27.152.p4k2cp9f64ib
Connection: keep-alive
If-None-Match: "wGAhvKXOC9SUJlTqyVtolw=="

HTTP/1.1 200 OK
```

Cookies

Workflow

1. Set by server via HTTP header „Set-Cookie“
2. Browser stores cookie and sends it back when revisiting same domain / path
3. Data within name/value pairs

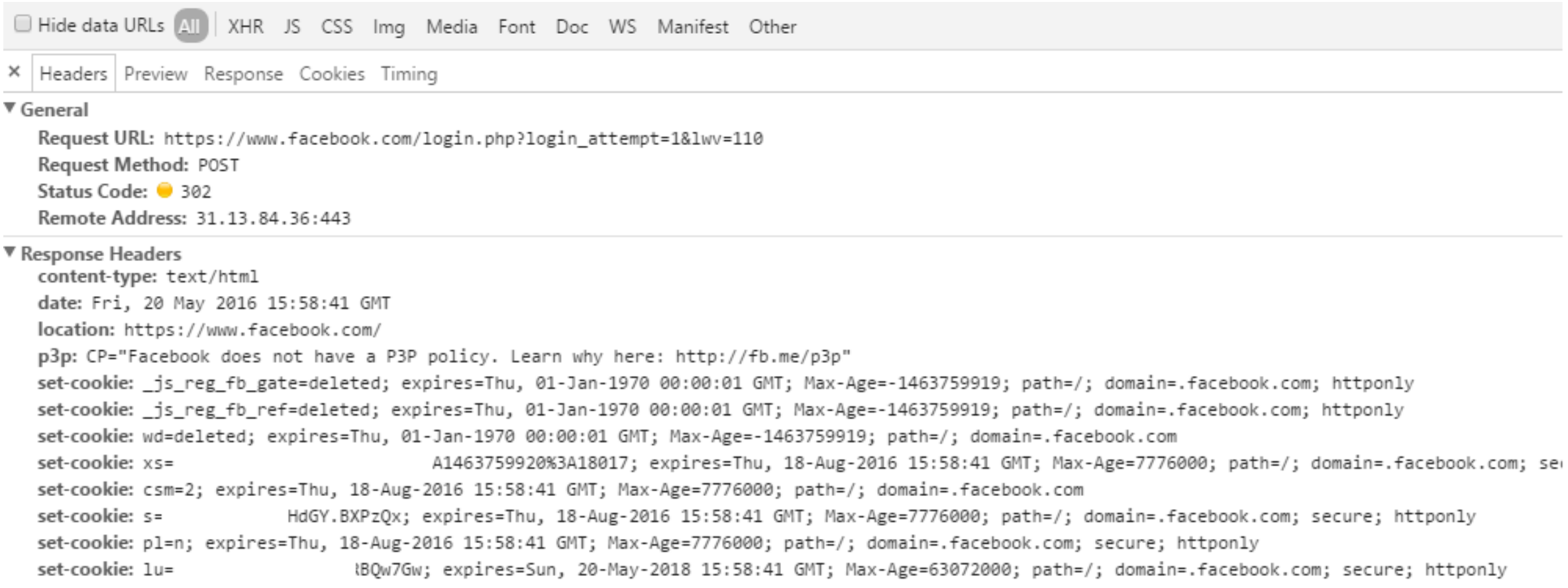
Cookie Structure

- **Domain:** iaik.tugraz.at
- **Path:** /
- **Expiration:** Deletion date – if not set: session cookie, valid until browser closed
- **Secure:** If flag set → Cookie only to be used within HTTPS connections
- **Httponly:** If flag set → Do not allow scripts to access the cookie, e.g. JavaScript `alert(document.cookie)` would fail! (Prevents XSS attacks!)

Cookies Example

Facebook Login

1. Send POST request with credentials to server
2. After successful login, receive response with „Set-Cookie“ header



Cookies Example

Requesting another page

Browser sets cookie in „Cookie“ header field

Cookies Pros / Cons

Advantages

- Do not appear in server logs
- User cannot interfere, e.g. copy cookie accidentially

Problem: Tracking & Privacy

1. On first visit of page, server sets cookie with unique identifier
2. On subsequent visits, same cookie sent

→ Profiling which pages were visited, in what sequence, for how long?

Technical issues / attacks

- **Man-in-the-middle**: If traffic unencrypted → cookie could be sniffed
- **Cross-site scripting (XSS):** Attacker injects code into website and steals cookie
- **Cross-site request forgery (CSRF)**

Bearer Tokens

= Access token sent in HTTP header

Workflow

1. User authenticates using credentials
2. Server returns bearer token
3. Client saves it locally, e.g. HTML5 localStorage
4. User requests protected resource → web app inserts token in HTTP header

```
Authorization: Bearer <token>
```

→ *Popular examples*

JSON Web Tokens (JWT), Simple Web Tokens (SWT), Security Assertion Markup Language (SAML)

Bearer Tokens

Why to use tokens instead of cookies?

- Easier for Single Sign-On (SSO) scenarios
  - Pass identity of authenticated users between identity provider and service provider
  - No 3rd-party cookie needed

- Tokens contain „claims“ = statements about user + additional metadata
  - Useful to allow/deny access to resources, services, routes

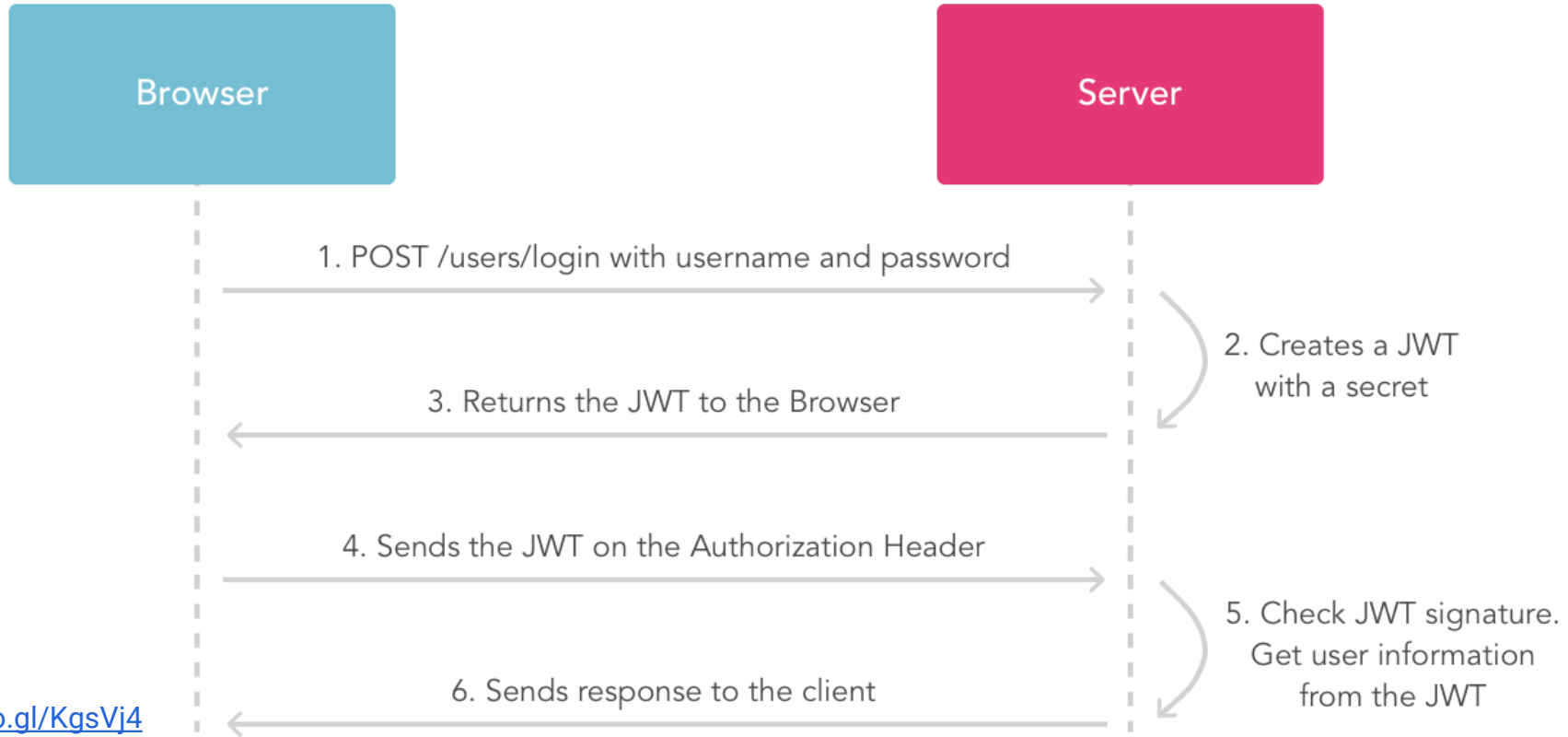
- Trusted information exchange
  - Tokens can be signed → Ensures authenticity of sender („I know who you are“)
  - Signature calculated over header + payload → Ensures integrity („no modification“)

"

JSON Web Tokens

RFC 7519

- Information exchange using JSON object
- Digital signature makes it verifiable
  - Relies on JSON Web Signatures (JWS, RFC 7515)
  - Using a secret + HMAC algorithm or by private / public RSA key pair



Source: https://goo.gl/KgsVj4

JSON Web Tokens

Try it yourself: https://jwt.io

Header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

Base64 →

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

Payload

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

Base64 →

eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9

Signature

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  secret)
```

Base64 →

TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

Token format: *<header>.<payload>.<signature>*

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIi…
```

IAIK TU Graz

Same Origin Policy

Introduction

Browser Security

- Scripts run in separated „sandboxes“
  - Isolated environment
  - No direct file access, restricted network access
  - Is this always enough protection?

- What can we do to assure that data is only exchanged with web application and **not** any other domain?
  - E.g. web application on https://iaik.tugraz.at
  - Do not allow content inclusion from https://www.evil.com

→ *Security Mechanism: Same Origin Policy (SOP)*

Same Origin Policy

> The same-origin policy restricts how a document or script loaded from one **origin** can interact with a resource from another **origin**. It is a critical security mechanism for isolating potentially malicious documents.

Source: https://goo.gl/SsXHYa

Features

- Provides further degree of isolation

- Scripts shall only access properties of documents & windows of *same* origin
  - Eliminate requests to other domain than origin
  - Not usable in reality since cross-origin requests required for many scenarios
    - External scripts, resources, using existing APIs (e.g. Maps, Dropbox, Facebook, …)

→ *What is an „origin“?*

Same Origin Policy

RFC 6454

URL structure: *scheme://domain:port/path?params*

Origin A: *http://www.example.com/dir/page.html*

- Origin A can access origin B's **DOM** if match on *(scheme, host, port)*

Compared URL	Outcome	Reason
http://www.example.com/dir/page2.html	Success	Same protocol, host and port
http://www.example.com/dir2/other.html	Success	Same protocol, host and port
http://username:password@**www.example.com**/dir2/other.html	Success	Same protocol, host and port
http://www.example.com:**81**/dir/other.html	Failure	Same protocol and host but different port
https://www.example.com/dir/other.html	Failure	Different protocol
http://**en.example.com**/dir/other.html	Failure	Different host
http://**example.com**/dir/other.html	Failure	Different host (exact match required)
http://**v2.www.example.com**/dir/other.html	Failure	Different host (exact match required)
http://www.example.com:**80**/dir/other.html	Depends	Port explicit. Depends on implementation in browser.

Source: https://goo.gl/p2miio

Path and params are not considered!

Secure flag!

- SOP for cookies granted if match on *([scheme], domain, path)*

IAIK TU Graz

Same Origin Policy

In a world without SOP…

Scenario 1

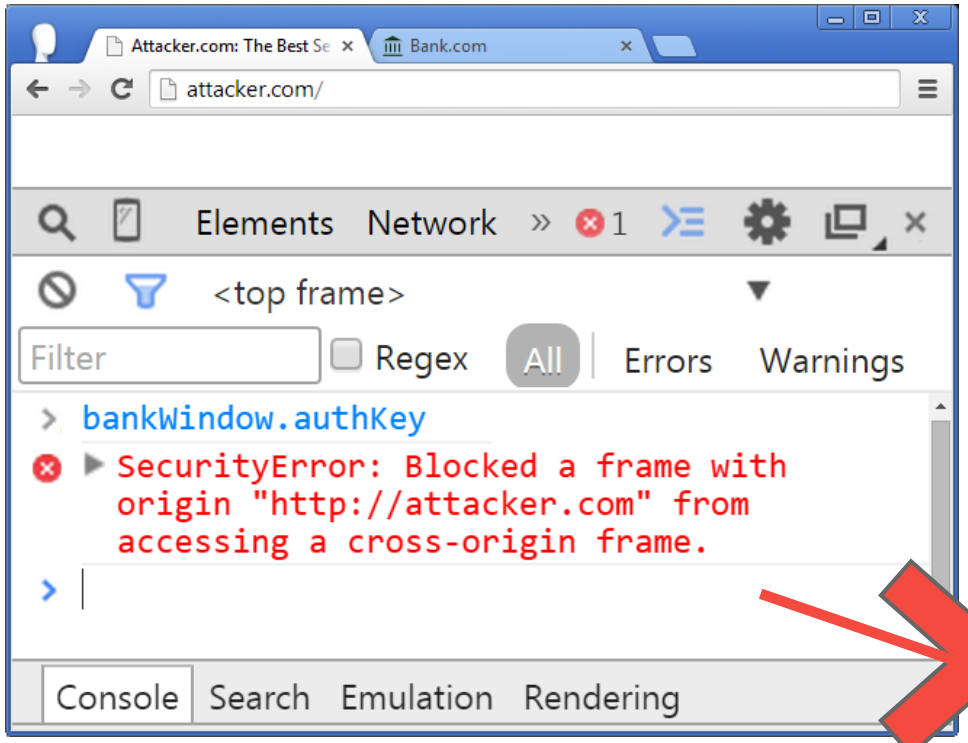
1. User is tricked into visiting login.mybank.al e.g., using Phishing e-mails
2. Attacker‘s page includes login.mybank.at in frame
3. User enters login credentials
   → Attacker has access to resources from login.mybank.at

Scenario 2 (XSS)

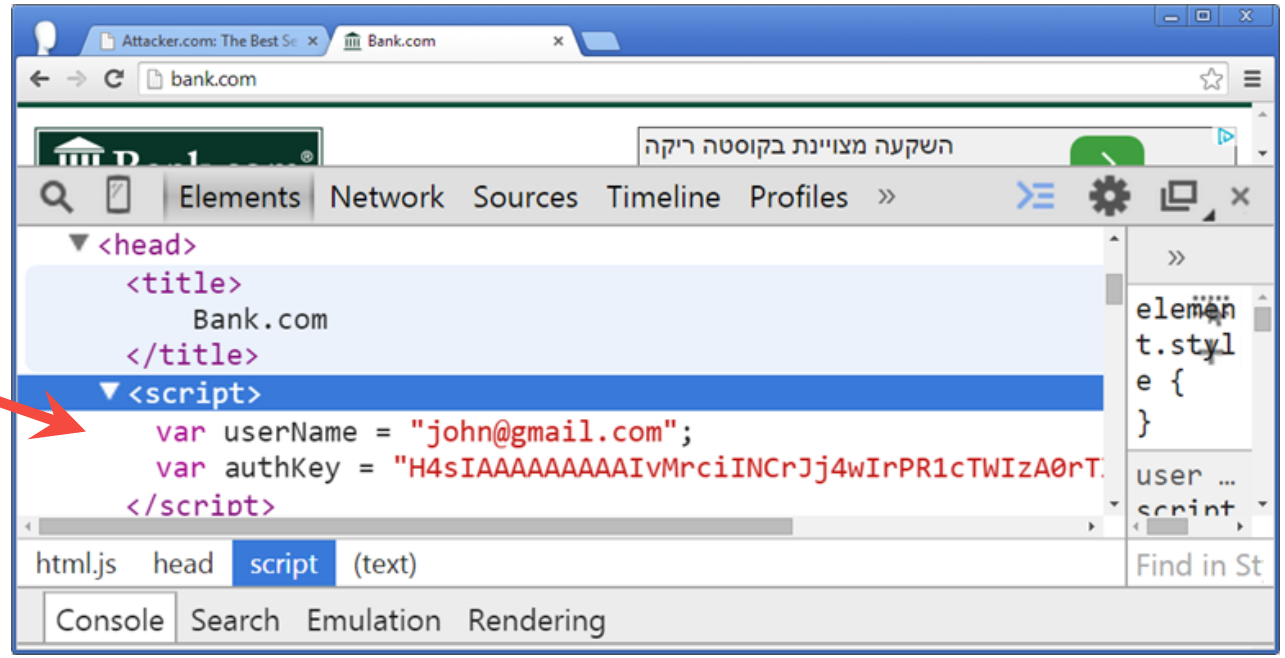
1. Good application on login.mybank.at
2. Attacker injects JavaScript into login.mybank.at
3. Malicious script now runs within login.mybank.at origin
   Can now access resources → send them to www.evil.com

Same Origin Policy

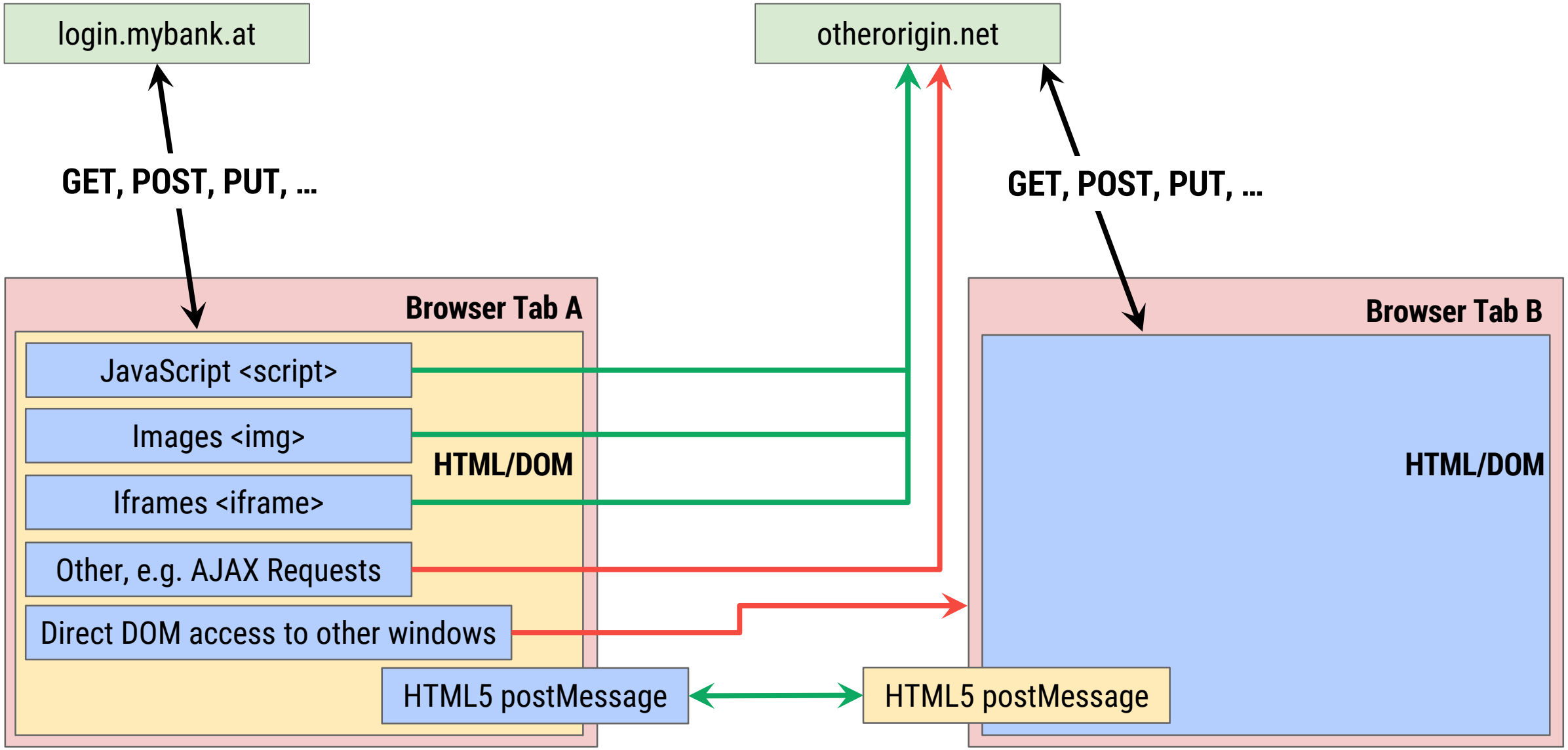
With SOP



Source: https://goo.gl/8qgeQv



Same Origin Policy



Same Origin Policy

By default…

Forbidden

- Direct access to DOM, cookies, window from other origins
- Direct HTTP(S) requests other origins (e.g. `XMLHttpRequest`)

Allowed requests to other origins

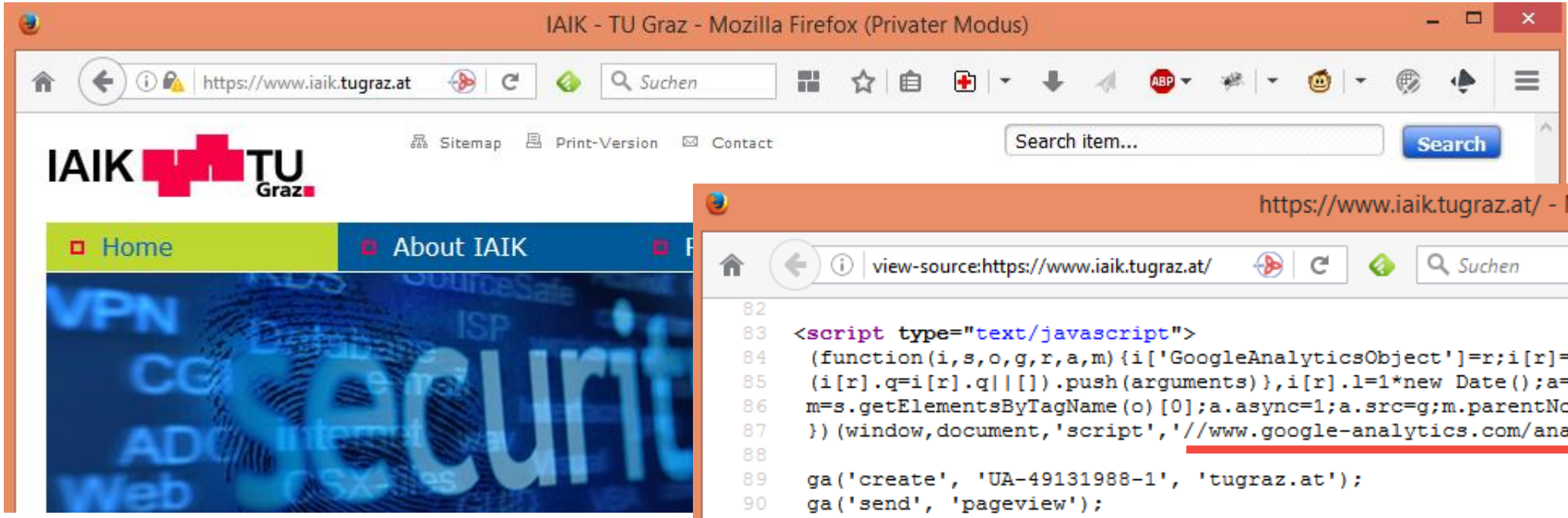
- <img> Including remote images
- <script> JavaScript libraries from other domains
- <iframe> Other page included in iframe
- HTML5 postMessage to other windows / frames
- Remaining HTML tags

Same Origin Policy

Script example

```
<script type="text/javascript" src="https://otherorigin.at/demo.js"></script>
```

→ SOP does **not** apply to scripts loaded in enclosing frame from other origin. JavaScript will be loaded as if provided on that page!



Bypassing SOP

Overview

SOP bypassing important for developers and security!

Status quo – we can…

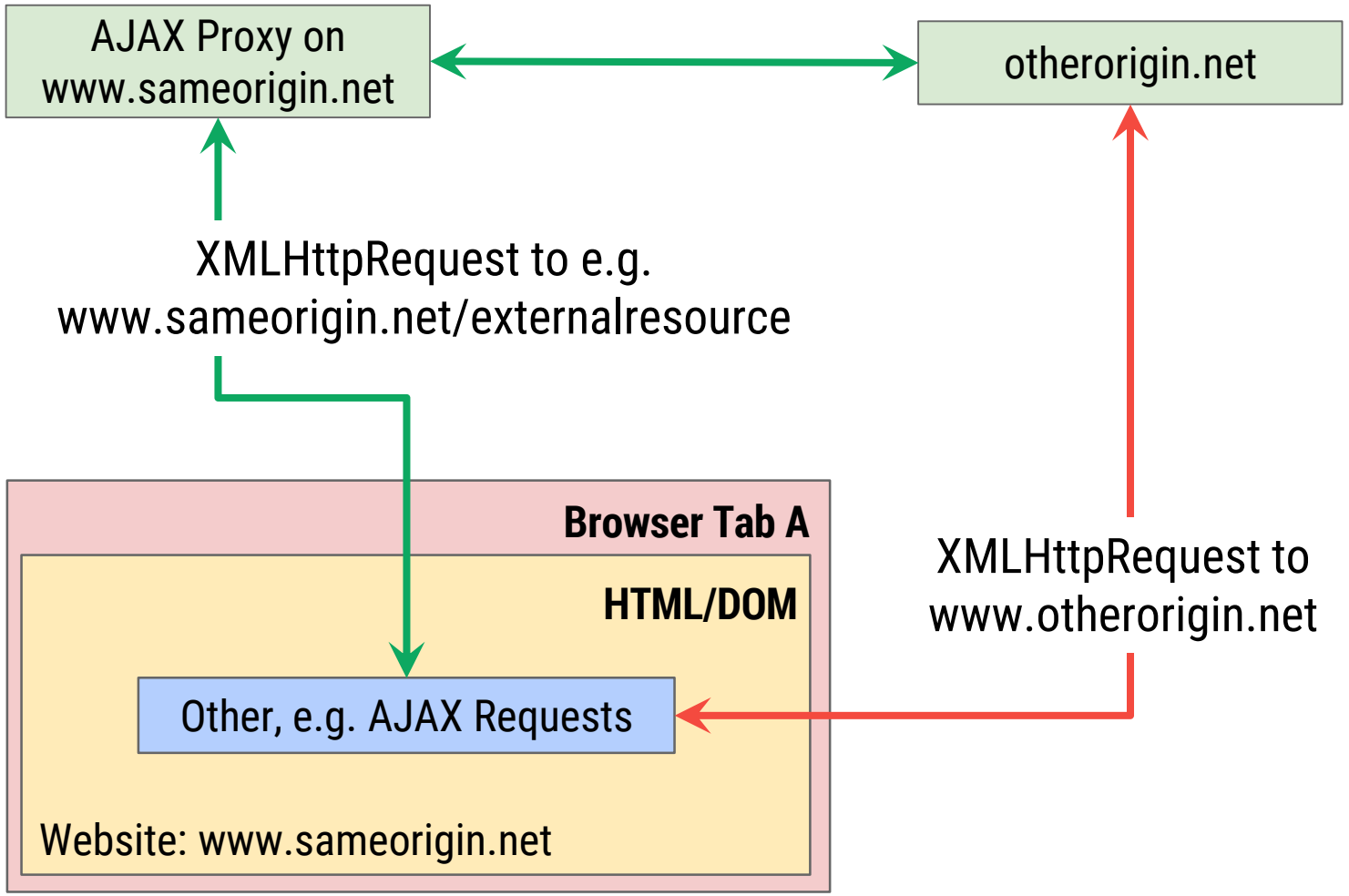
- Load resources from other servers: <img>, <iframe>, <script>, etc.
- Load resources from other browser windows: HTML5 postMessage

Q: But how can we perform HTTP requests (e.g. AJAX) to other domains?

A: Some ways to bypass SOP benevolently:

- AJAX Proxy
- JSONP
- CORS
- Content Security Policy (CSP)

AJAX Proxy



Workflow

- Website implements full HTTP client on same origin
- Website sends XmlHttpRequest to AJAX proxy
- Proxy crafts regular request to other domain
- Result passed back to original caller

→ *Indirect SOP bypassing because browser does not load content from another origin!*

JSONP

Idea

- We want to include code from other domains
  - <script> element only allows us to request a (valid) script on foreign origin, e.g. Javascript library but no **interaction with other page**

- How can we work with **data** resources from other websites?
  - Objects need a name / identifier / reference in order to address them somehow
  - Any JSON object without reference → garbage collector would simply delete it

→ JSON with Padding

Like JSON but data is wrapped in (= „padded with“) JavaScript function call

JSONP

Example - Problem

Web app includes script tag

```
<script type="text/javascript" src="http://domain.com/Users/1234"></script>
```

JSON

```
{
  "Name": "Foo",
  "Id": 1234,
  "Rank": 7
}
```

- Response would include JSON data as string
  - Browser evaluates file → syntax error because pure object literals are inaccessible
  - Need some variable assignment around to make it executable!

JSONP

Example - Solution

We need to process data we fetched
→ pass it to an existing function call (also known as „callback“)

JSON

```
{
  "Name": "Foo",
  "Id": 1234,
  "Rank": 7
}
```

JSONP

```
parseResponse(
  {
    "Name": "Foo",
    "Id": 1234,
    "Rank": 7
  }
)
```

Callback function typically specified by calling website, e.g.

```
<script type="text/javascript"
src="http://domain.com/Users/1234?jsonp=parseResponse"></script>
```

JSONP

Real-world example

```
<script type="text/javascript">
  function ourcallback(jsonData) {
    document.write("Geolocation info for IP address" + jsonData.query);
    document.write("Coordinates: " + jsonData.lat + ", " + jsonData.lon);
  }
</script>
<script type="text/javascript" src="http://ip-api.com/json/?callback=ourcallback">
</script>
```

JSONP Response

```
ourcallback({"as":"AS1113 Technische Universitaet Graz,", "city":"Graz",
"country":"Austria", "countryCode":"AT", "isp":"Technische Universitaet
Graz", "lat":47.1157, "lon":15.5901, "org":"Technische Universitaet Graz",
"query":"129.27.142.148", "region":"6", "regionName":"Styria",
"status":"success", "timezone":"Europe/Vienna", "zip":"8010"});
```

JSONP

Problems

- With JSONP, any content can be injected into the page
- You cannot control who (which *origins*) access your JSONP API

Attack scenario

- You are logged in on account.example.com → cookie set in your browser
- account.example.com exposes JSONP API, e.g. `/userinfo`

- Attacker tricks user into visiting evil.com which requests foreign JSONP API: account.example.com/userinfo?f=rkncallback
  - Upon request, browser includes cookie of authenticated user

→ `rkncallback` function on evil.com learns sensitive data of user!

CORS

https://goo.gl/NtqA3G

= Cross-Origin Resource Sharing

→ CORS is a mechanism to limit which origin can access resources

Features

- Perform cross-origin AJAX requests
  - Request page on remote origin
  - Specify type of request (GET, POST, etc.)
    - JSONP allowed only GET requests
  - Allow to send credentials (cookies)

- Permissions defined by server in HTTP headers

CORS Example – Request

- Website loaded in browser: http://example.com
- Resource to be requested via AJAX: http://thirdparty.com/resource.js

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/resource.js');
xhr.onload = function() { ... };
xhr.send();
```

(Same-Origin AJAX Request)

```
var xhr = new XMLHttpRequest();
xhr.open('GET',
'http://thirdparty.com/resource.js');
xhr.onload = function() { ... };
xhr.send();
```

Cross-Origin AJAX Request

- GET request from client to thirdparty.com includes origin header

```
GET /resource.js HTTP/1.1
Host: thirdparty.com
Origin: http://example.com
```

CORS Example – Response

Server

- thirdparty.com knows whether origin is trusted
- Server responds with allowed origin domains in HTTP response header or with * if any domain is fine

```
HTTP/1.1 200 OK
Content-Type: text/html
Access-Control-Allow-Origin: http://example.com
…
```

Browser

- Checks if current domain matches allowed origin
  - Yes → pass through the response
  - No → Block the response due to SOP

CORS Headers

Server provides permissions in HTTP headers

Header	Purpose
`Access-Control-Allow-Origin`	`Lists allowed domains, * to allow any`
`Access-Control-Allow-Credentials`	`True | False → Indicates whether client is allowed to send cookies`
`Access-Control-Allow-Methods`	`Defines allowed methods (PUT, DELETE, etc.) for requests other than GET, POST, HEAD`
`Access-Control-Allow-Headers`	`Defines which HTTP headers can be used in requests`
`Access-Control-Max-Age`	`How long can information be cached until next Preflight request is necessary`

- Technical overview: https://goo.gl/hXBxzW
- Tutorial: http://goo.gl/jNd8p8
- Try it yourself: http://goo.gl/FsRnl3

CSP

Problem

SOP enables us only to restrict certain outgoing communication

→ Cross-Site Scripting (XSS), local JavaScript injection still possible!

Solution: Content Security Policy

- Idea is to define policy for web application
- Browser shall enforce policy, received via special HTTP header field

```
HTTP/1.1 200 OK
Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self';
img-src 'self'; style-src 'self';
…
```

Important: Used CSP tags have to be supported by browser!

→ https://content-security-policy.com/browser-test/

CSP

Idea

Create a sandbox for the web application

- Allow resources (frames, scripts, stylesheets) only from specific sources
- Forbid plain HTTP communication
- Restrict cross-origin AJAX requests to certain domains
- Prevent insecure code execution, e.g. inline code or JavaScript's `eval()` function

Effect

- By defining a strict policy → XSS can be prevented
- Injected code cannot talk to targets that are not defined in policy
  - Black-/Whitelisting approach

→ Like permissions on Android / iOS but with different granularity!

CSP Examples

Scenario 1: *Only load resources (images, scripts, frames) from local origin & http://example.com*

```
Content-Security-Policy: default-src 'self' http://example.com
```

Scenario 2: *Load resources that can modify the page only via HTTPS*

```
Content-Security-Policy:
img-src https: data:; font-src https: data:; media-src https:
```

Scenario 3: *Load resources only from current origin (self), block mixed content, enable reflected XSS protections, ensure no referer headers are sent on downgrade (HTTPS → HTTP)*

```
Content-Security-Policy:
default-src 'none'; script-src 'self'; style-src 'self'; img-src 'self'; font-src
'self'; upgrade-insecure-requests; block-all-mixed-content; reflected-xss block;
referrer no-referrer-when-downgrade
```

→ More directives at http://content-security-policy.com. CSP Generator: http://goo.gl/N4RkTY

SOP Attack Defense

Scenario 1

How can you protect an API on your server against misuse by attacker‘s page?

Scenario 2

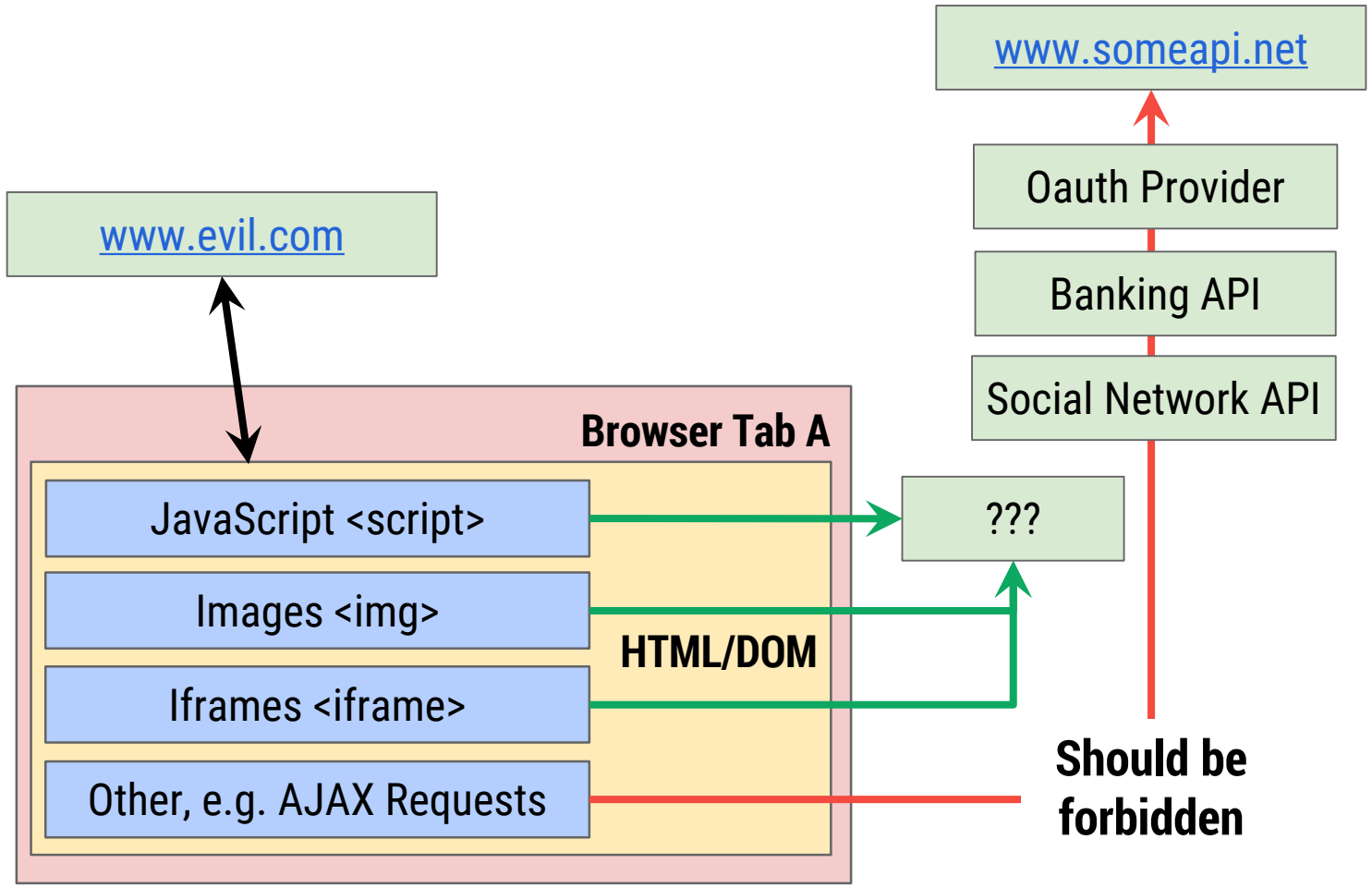
How can you protect your web application from injected code (XSS)?

1. Remove possibility to inject code
2. If everything fails → make sure that private information is not sent to attacker, e.g. session IDs, (probably internal) data of web applications

SOP Attack Scenario 1

Phishing Mail:

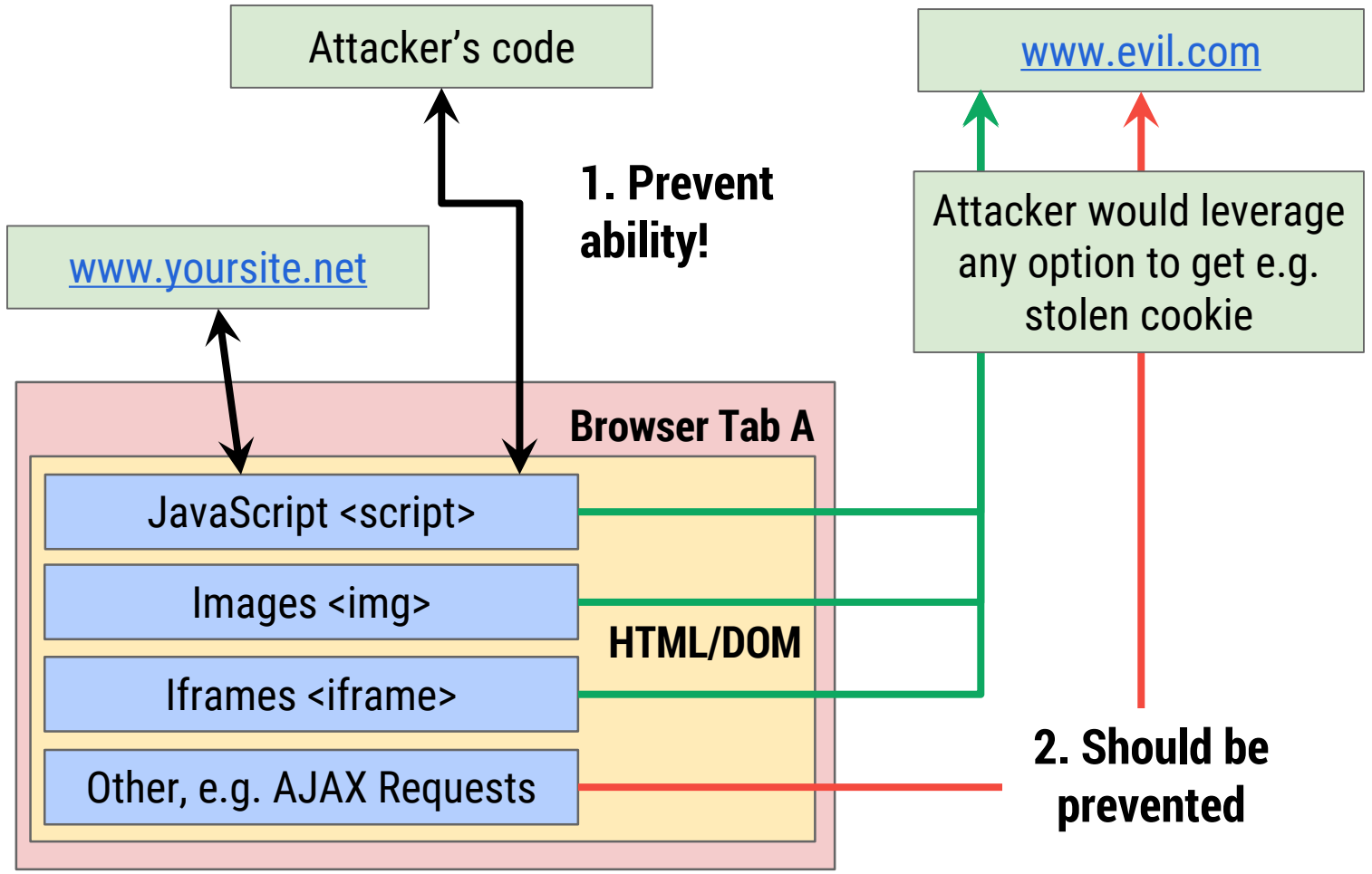
```
Dear Mr. X! Please visit http://www.evil.com, and you will get free
beer for life! Best regards, Dr. Hopmalt
```



Protection mechanism against access from arbitrary origins?

- AJAX Proxy ✗
- JSONP ✗
- **CORS** ✓
- CSP ✗

SOP Attack Scenario 2



Protection mechanism against XSS code injection?

- AJAX Proxy ✗
- JSONP ✗
- CORS ✗
- **CSP** ✓

Protection Tips

- Be cautious when embedding <script> elements pointing to 3rd party sites into your web application
  - If attacker gains access to these scripts → can compromise your website and your user‘s personal data

- JSONP is not „secure by design“
  - Essentially the same thing as using <script> elements
  - Do not use it to send sensitive data → not protected by SOP

- Using CORS,
  - do not set Access-Control-Allow-Origin header to *
    → does not protect access to your API

Client-Side Attacks

Stealing Sessions

Remember:

Whoever knows the session ID has access (even without credentials)!

Attack Scenarios

- Session Fixation
  - Attacker injects own session ID which is then used by user (and known by attacker)
- Session Hijacking
  - Prediction
  - Brute-Force
  - Sniffing (XSS)

Session Fixation

Idea

Trick victim into using an attacker‘s session ID

Workflow

1. Attacker signs in on https://vulnerable.iaikshop.at
2. Server returns session ID: https://vulnerable.iaikshop.at/?sid=fa392522a05d0
3. Attacker sends this link to victim
4. Victim clicks link and signs in using this session ID

→ **If** server does not regenerate ID upon login, attacker already knows ID

Problem typically arises when using home-brew session management scripts
→ Can be easily mitigated by checks on server side!

Session Hijacking – Sniffing

Intercepting transmission

- Easy if website uses HTTP
- Man-in-the-middle (MITM) attack on TLS connection if using HTTPS

→ Capture session ID from recording (URLs, cookies from HTTP headers, etc.)

Leverage flaws in cookie processing

- **Secure flag**
  - Cookie only to be used within HTTPS connections
  - If not set → Cookie is also sent if connection is downgraded (HTTPS → HTTP)

- **HttpOnly**
  - Do not allow scripts to access the cookie
  - If not set → Readable from JavaScript, e.g. `alert(document.cookie)`

Cross-Site Scripting (XSS)

Idea

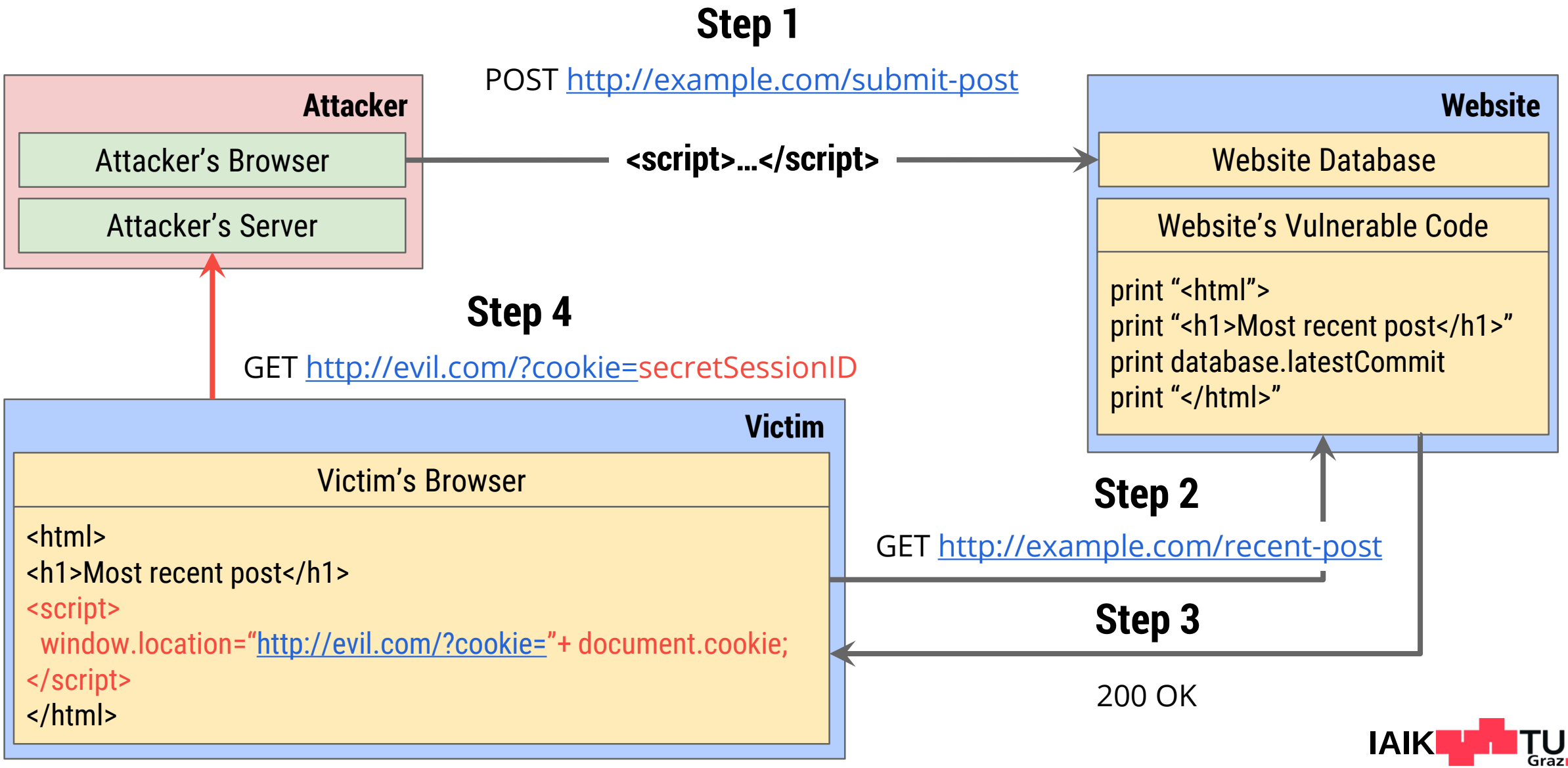
Code injection attack to execute malicious JavaScript in another user‘s browser

→ Bypasses SOP because browsers trust local (same) origins!

Consequences

- Cookie Theft
  - Attacker may access victim‘s cookie, e.g. using document.cookie
- Keylogging
  - Attacker can register keyboard event listener using addEventListener
  - Send all keystrokes (passwords, credit card number) to own server
- Phishing
  - Attacker can manipulate DOM, e.g. insert fake login form

XSS – Workflow



XSS Types

3 categories of Cross-Site Scripting

- Stored („Persistent“) XSS
  - Attacker manages to store malicious payload in target database, e.g. comment field, blog or forum post
  - Every victim calling the page will be served (and execute) the XSS payload

- Reflected („Non-Persistent“) XSS
  - XSS payload is part of the request URI → „reflected“ back in HTTP response
  - Often used in Phishing mails, social engineering attempts

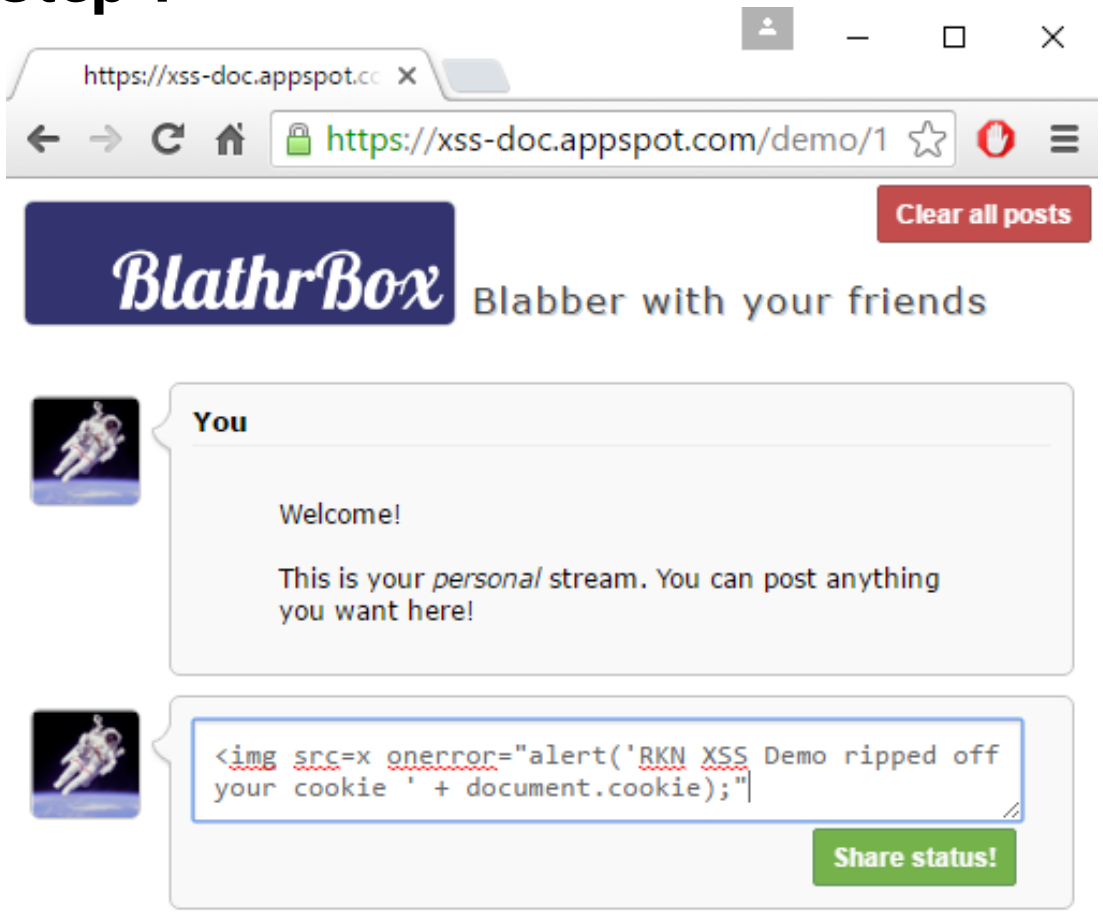
- DOM-based XSS
  - DOM injection on client-side → server is not involved in any way
  - Example: Script writes user-provided data to DOM

Stored XSS Example

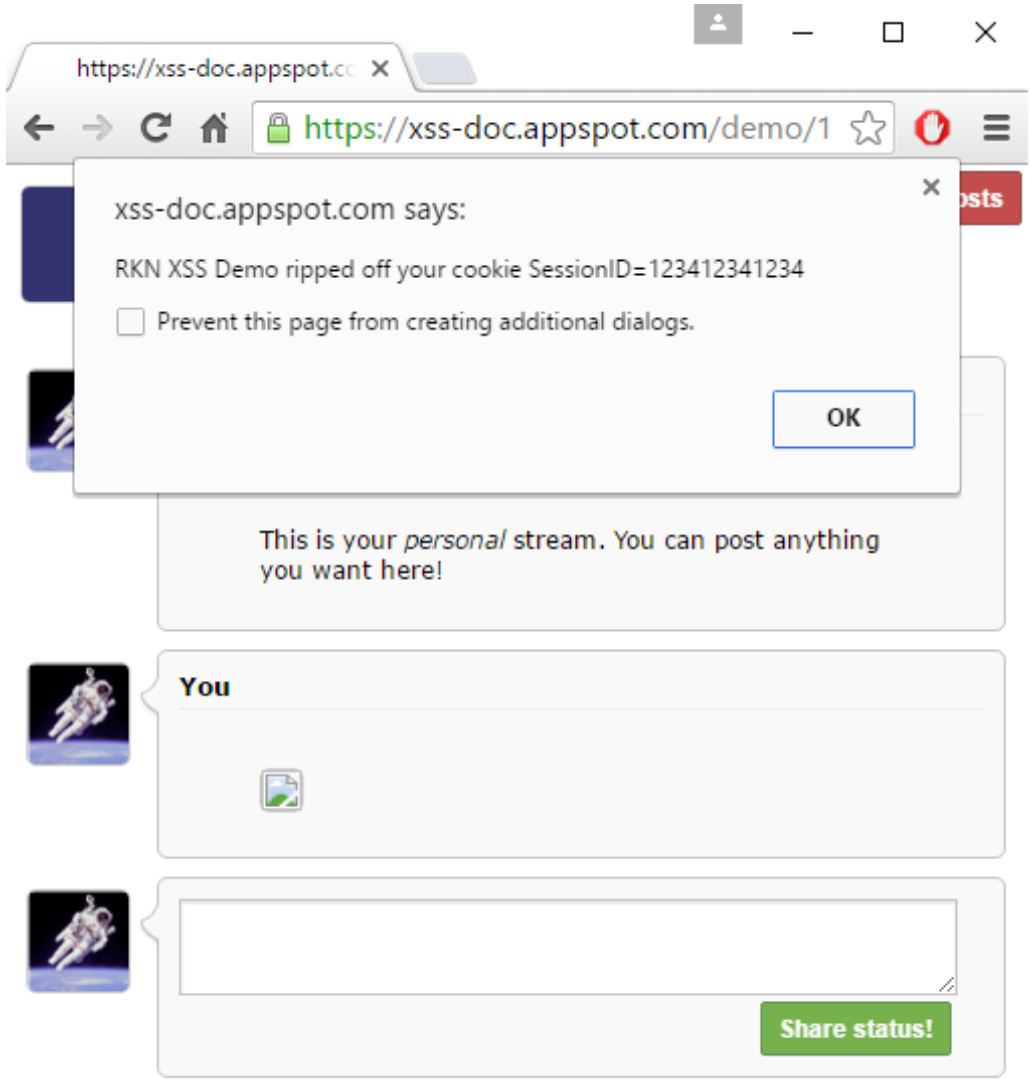
https://goo.gl/qYi7lv

Webserver stores and echoes back XSS payload

Step 1



Step 2

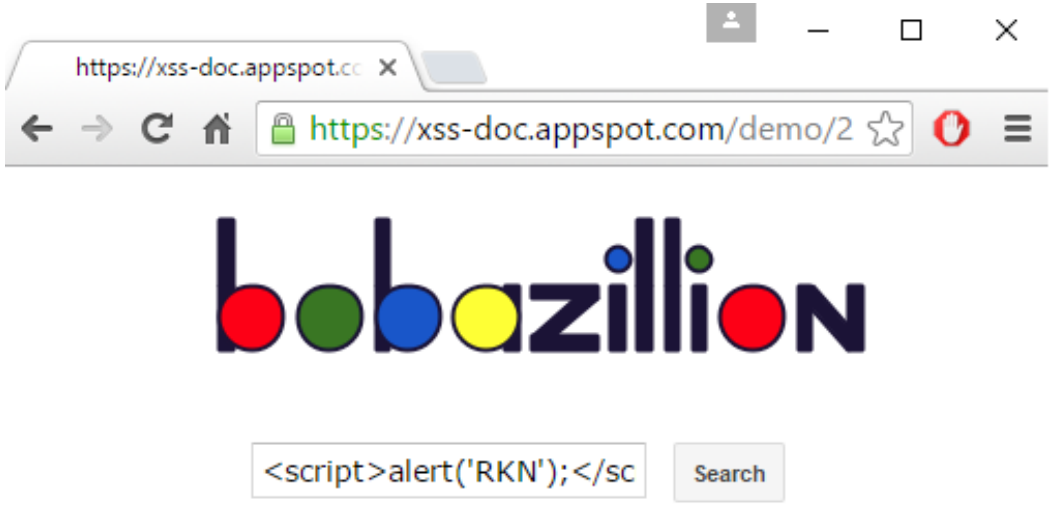


Reflected XSS Example

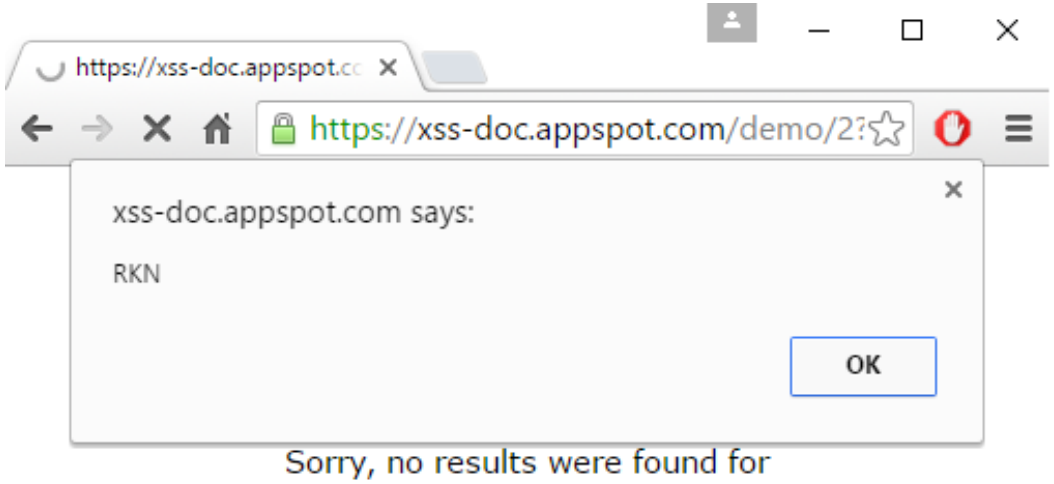
https://goo.gl/mq927p

Webserver replies user input without escaping or validation

Step 1



Step 2



- Might seem harmless because who would click on such a link?
  - https://xss-doc.appspot.com/demo/2?query=<script>alert('RKN')</script>
- But what if it is hidden behind a URL shortener?

DOM-based XSS Example

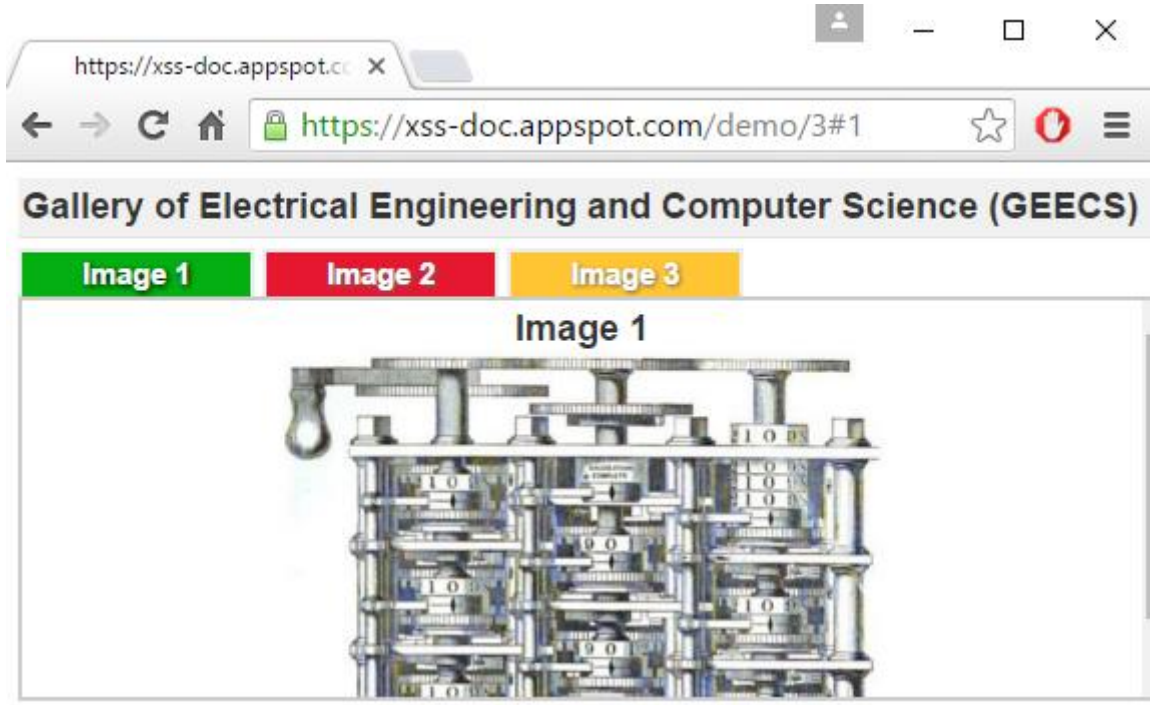
https://goo.gl/WI9dGL

DOM element is filled with user-provided data

Here: `window.location` is set to `innerHtml`

```
<script>
function chooseTab(name) {
 var html = "Image " + parseInt(name) + <br>";
 html += "<img src='/static/demos/GEECS" + name +
   ".jpg' />";
 document.getElementById('tabContent').
   innerHTML = html;
 ...
}

chooseTab(self.location.hash.substr(1));
</script>
```



Attack: *https://xss-doc.appspot.com/demo/3#'><img src=x onerror=alert(/RKN XSS/)>*

Rendered in DOM:

```
<img src='/static/demos/GEECS'><img src=x
onerror=alert(/RKN XSS/)>.jpg' />
```

XSS Attack Vectors

Attacks often also work without <script>…</script> and thereby aim to bypass detection!

Examples

`<body>` tag

An XSS payload can be delivered inside `<body>` tag by using the `onload` attribute or other more obscure attributes such as the `background` attribute.

```
<!-- onload attribute -->
<body onload=alert("XSS")>
<!-- background attribute -->
<body background="javascript:alert("XSS")">
```

`<img>` tag

Some browsers will execute JavaScript when found in the <img>.

```
<!-- <img> tag XSS -->
<img src="javascript:alert("XSS");">
<!--  tag XSS using lesser-known attributes -->
<img dynsrc="javascript:alert('XSS')">
<img lowsrc="javascript:alert('XSS')">
```

Source: http://goo.gl/A1XoXe

XSS Prevention

***Important:** Vulnerabilities **only** exist if the payload ultimately gets rendered in the victim's browser!*

- Site parameters need to be filtered / „escaped“
  - `<script>alert(‘Sec’);</script>` → `<script>alert(‘Sec')</script>`
- In practice hard to manually consider every input
  - Prefer templating system or framework with context-aware auto escaping
- Always use httpOnly flag with cookies
  - Very effectively blocks XSS attacks!
- Test application for XSS using tools, e.g. Burp Proxy

See: https://goo.gl/dhoUUV

XSS Resources

- Tutorials on Cross-Site Scripting
  - https://www.google.com/about/appsecurity/learning/xss/
  - https://excess-xss.com

- Attack vectors
  - https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet
  - https://n0p.net/penguicon/php_app_sec/mirror/xss.html
  - http://www.xenuser.org/xss-cheat-sheet/

- Test XSS safely: https://xss-game.appspot.com

CSRF / XSRF

= Cross-Site Request Forgery

Note: This is not XSS!

Problem

> CSRF vulnerabilities occur when a website allows an **authenticated user** to perform a sensitive action but does not verify that the **user himself** is invoking that action. The key to understanding CSRF attacks is to recognize that websites typically don't verify that a request came from an authorized user. Instead they verify only that the request came from the browser of an authorized user.

Source: https://goo.gl/QPQoDn

Consequences

- Attacker speculates that users are authenticated at some website
- Provides victim with crafted URL (Malware, email, XSS injection)
  - Tries to perform action on some website on user's behalf

CSRF / XSRF

Example

- Victim authenticated (= valid cookie) as admin at blog service
- Some component of blog is vulnerable to XSS

1. Attacker manages to inject XSS payload into blog post that calls this URL

   ```
   http://www.example.com/admin.php?action=new_user&name=rkn&password=badboy
   ```

2. Victim visits blog → XSS payload is called by user‘s browser
   - Request includes valid cookie and action is executed on victim‘s behalf

***Note**: XSS is just one helper here, attacker could also supply direct URL, e.g.*

- User is authenticated at crypto coin website: https://mymonero.com
- Attacker sends user link: https://mymonero.com/send?amount=1000&acct=attacker
  - Action would be executed within user‘s authenticated browser

CSRF / XSRF

Real-world example: CVE-2015-7984

```
Multiple cross-site request forgery (CSRF) vulnerabilities in Horde before
5.2.8, Horde Groupware before 5.2.11, and Horde Groupware Webmail Edition before
5.2.11 allow remote attackers to hijack the authentication of administrators for
requests that execute arbitrary (1) commands via the cmd parameter to
admin/cmdshell.php, …
```

Source: http://goo.gl/YihLbe

Exploit code, e.g. for injection via XSS or triggered by Malware:

```
<form action="http://[host]/admin/cmdshell.php" method="post" name="main">
<input type="hidden" name="cmd" value="ls">
<input value="submit" id="btn" type="submit" />
</form>
<script>
document.getElementById('btn').click();
</script>
```

Source: https://goo.gl/Hioa4d

CSRF / XSRF Prevention

- Synchronized tokens
  - Should be generated randomly (unpredicted, unique)
  - Token transmitted with every form field

```
<input type="hidden" name="csrftoken" value="KbyUpYj7CDP3qmLlkPt" />
```

→ Attacker will not manage to place correct token into forged request

- Cookie-to-header token
  - Server sets CSRF token into cookie `Set-Cookie: Csrf-token=…`
  - Sent by with every request to server
  - Server validates presence and integrity of token

Other methods, e.g. „checking HTTP referer header“ or „Only allow POST“ ***are not reliable!***

See https://goo.gl/5vtFih for more tips!

Outlook

- 17.01.2020
  - TLS Hanshake
  - TLS Security Features

- 24.01.2020
  - TLS Vulnerabilities & Attacks
  - DNS Security

IAIK TU Graz