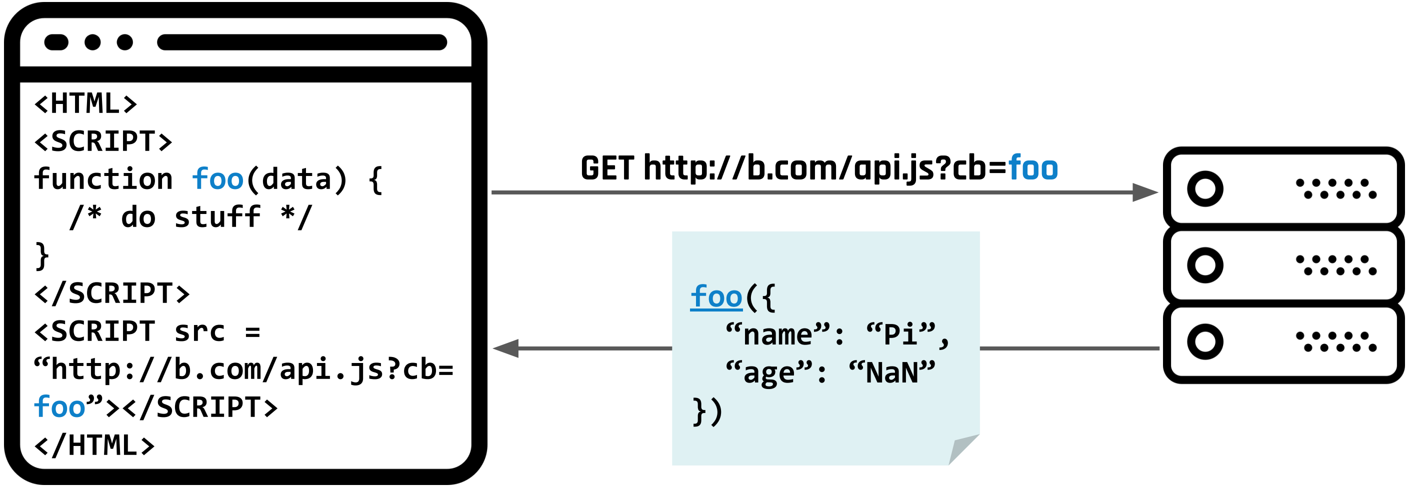
JSON with Padding (JSON-P)

JSON with Padding (JSON-P)

- Sometimes cross-origin read is desired...
- Developers came up with **JSON-P**, a ~~hack~~ technique exploiting the fact that script inclusion is not subject to the SOP

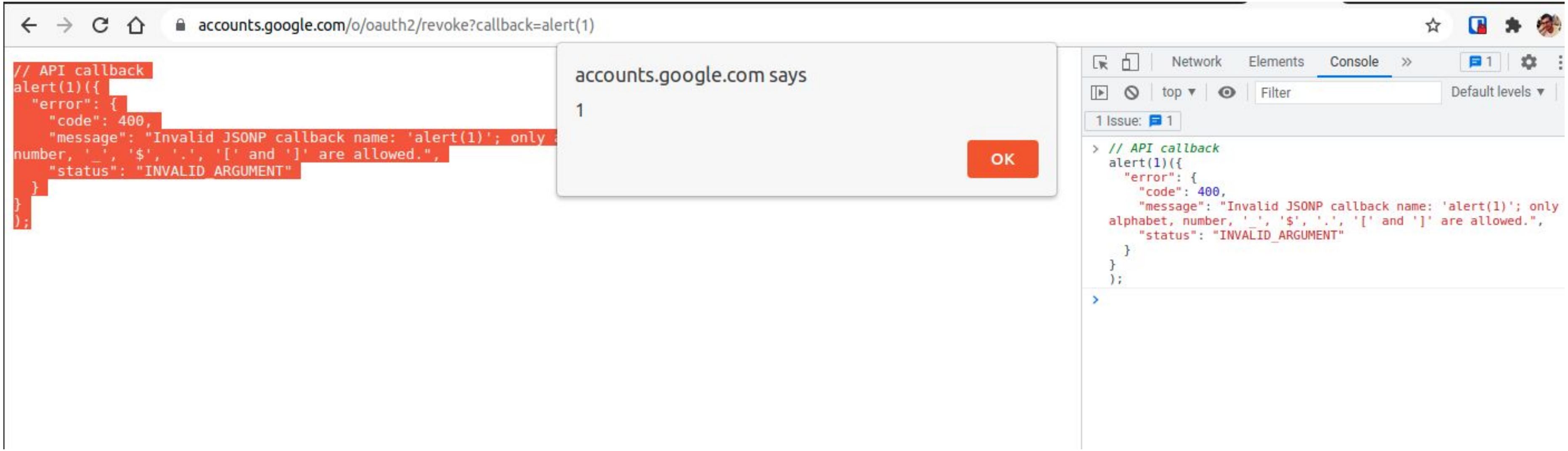


```
<HTML>
<SCRIPT>
function foo(data) {
  /* do stuff */
}
</SCRIPT>
<SCRIPT src =
“http://b.com/api.js?cb=
foo”></SCRIPT>
</HTML>
```

```
foo({
  “name”: “Pi”,
  “age”: “NaN”
})
```

There are several JSON-P endpoints in the wild…

Example: **https://accounts.google.com/o/oauth2/revoke?callback=**



accounts.google.com/o/oauth2/revoke?callback=alert(1)

```
// API callback
alert(1)({
  "error": {
    "code": 400,
    "message": "Invalid JSONP callback name: 'alert(1)'; only a
number, '_', '$', '.', '[' and ']' are allowed.",
    "status": "INVALID_ARGUMENT"
  }
}
);
```

accounts.google.com says

1

```
> // API callback
  alert(1)({
    "error": {
      "code": 400,
      "message": "Invalid JSONP callback name: 'alert(1)'; only
  alphabet, number, '_', '$', '.', '[' and ']' are allowed.",
      "status": "INVALID_ARGUMENT"
    }
  }
  );
>
```

See also: https://dev.to/benregenspan/the-state-of-jsonp-and-jsonp-vulnerabilities-in-2021-52ep

Issues with JSON-P

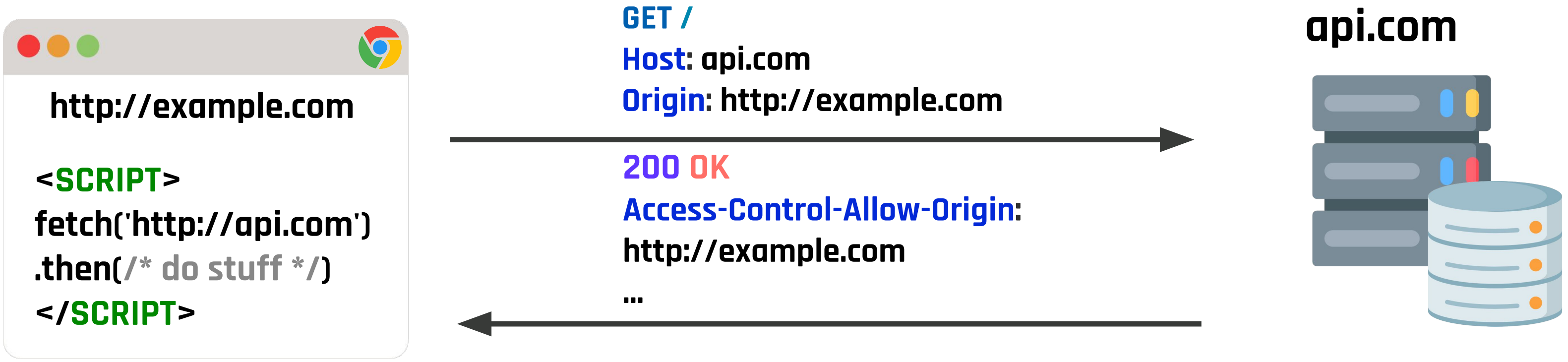
- Only **GET** requests can be performed
- Endpoint could validate Referer but this may be forged or missing
- Requires **complete trust** of the third-party host
  - The third-party is **allowed to execute scripts** within the importing page
  - The importing origin **cannot perform any validation** of the included script
- JSON-P should not be used anymore!

We need a better solution...

Cross-Origin Resource Sharing (CORS)

Relaxing the SOP

- Sometimes it is desirable to allow JavaScript to access the content of cross-site resources
- **Cross-Origin Resource Sharing (CORS)** provides a controlled way to relax the SOP
- JavaScript can access the response content if the Origin header in the request matches the Access-Control-Allow-Origin header in the response (or the latter has value *)



http://example.com

```
<SCRIPT>
fetch('http://api.com')
.then(/* do stuff */)
</SCRIPT>
```

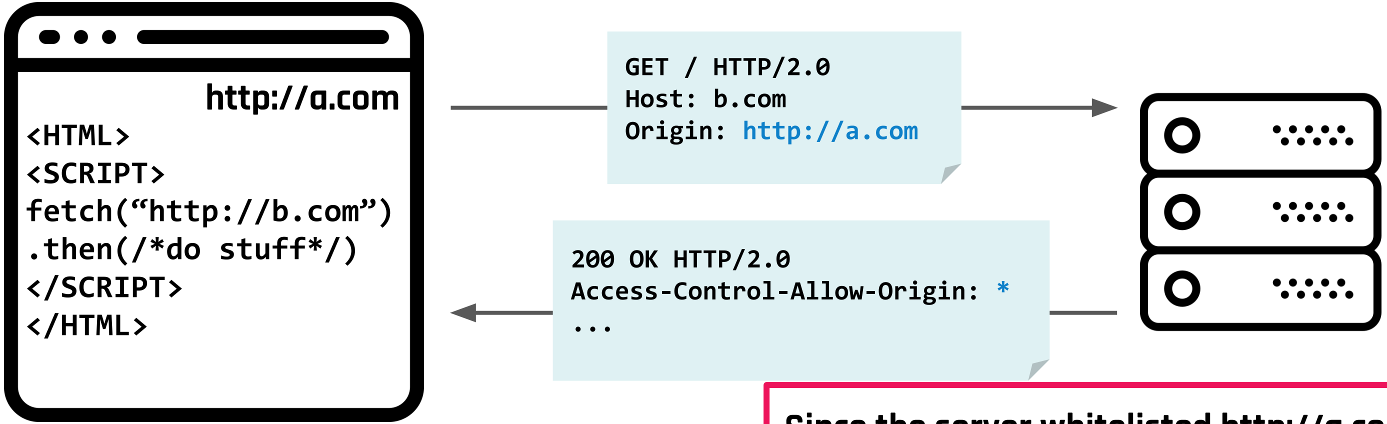
```
GET /
Host: api.com
Origin: http://example.com
```

```
200 OK
Access-Control-Allow-Origin:
http://example.com
...
```

Since the server whitelisted http://example.com, the browser allows the script to access the contents of the response

Reference: https://jakearchibald.com/2021/cors/

CORS with Simple Requests



http://a.com

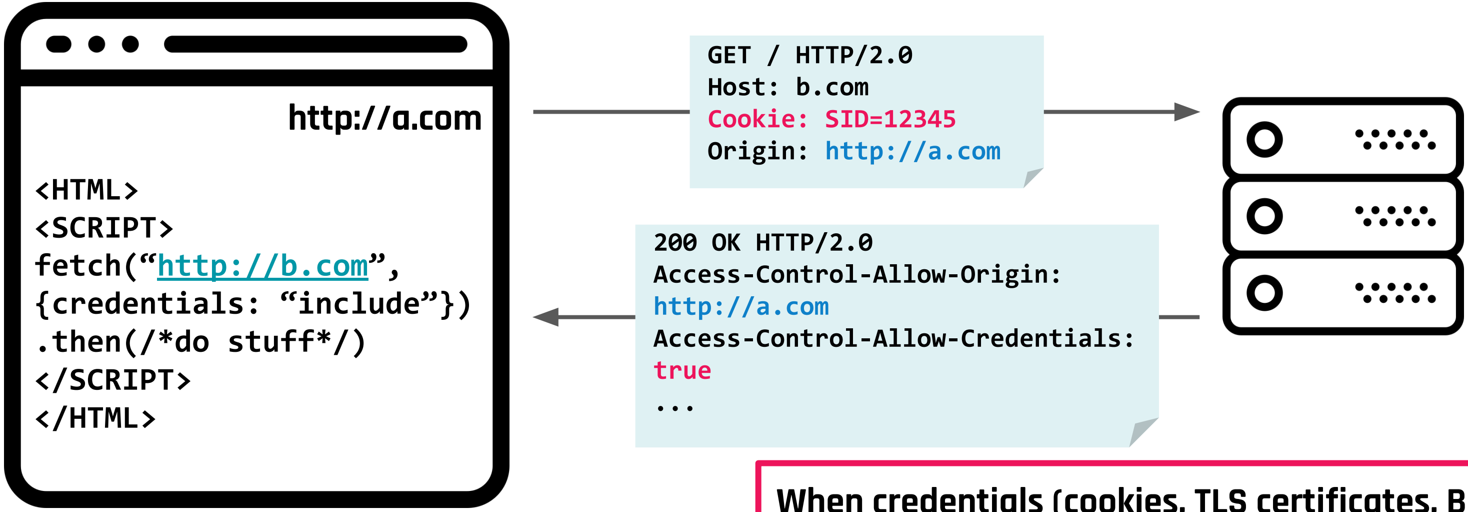
```
<HTML>
<SCRIPT>
fetch(“http://b.com”)
.then(/*do stuff*/)
</SCRIPT>
</HTML>
```

```
GET / HTTP/2.0
Host: b.com
Origin: http://a.com
```

```
200 OK HTTP/2.0
Access-Control-Allow-Origin: *
...
```

Since the server whitelisted http://a.com, the browser allows the script to access the contents of the response

CORS with Credentials



http://a.com

```
<HTML>
<SCRIPT>
fetch("http://b.com",
{credentials: "include"})
.then(/*do stuff*/)
</SCRIPT>
</HTML>
```

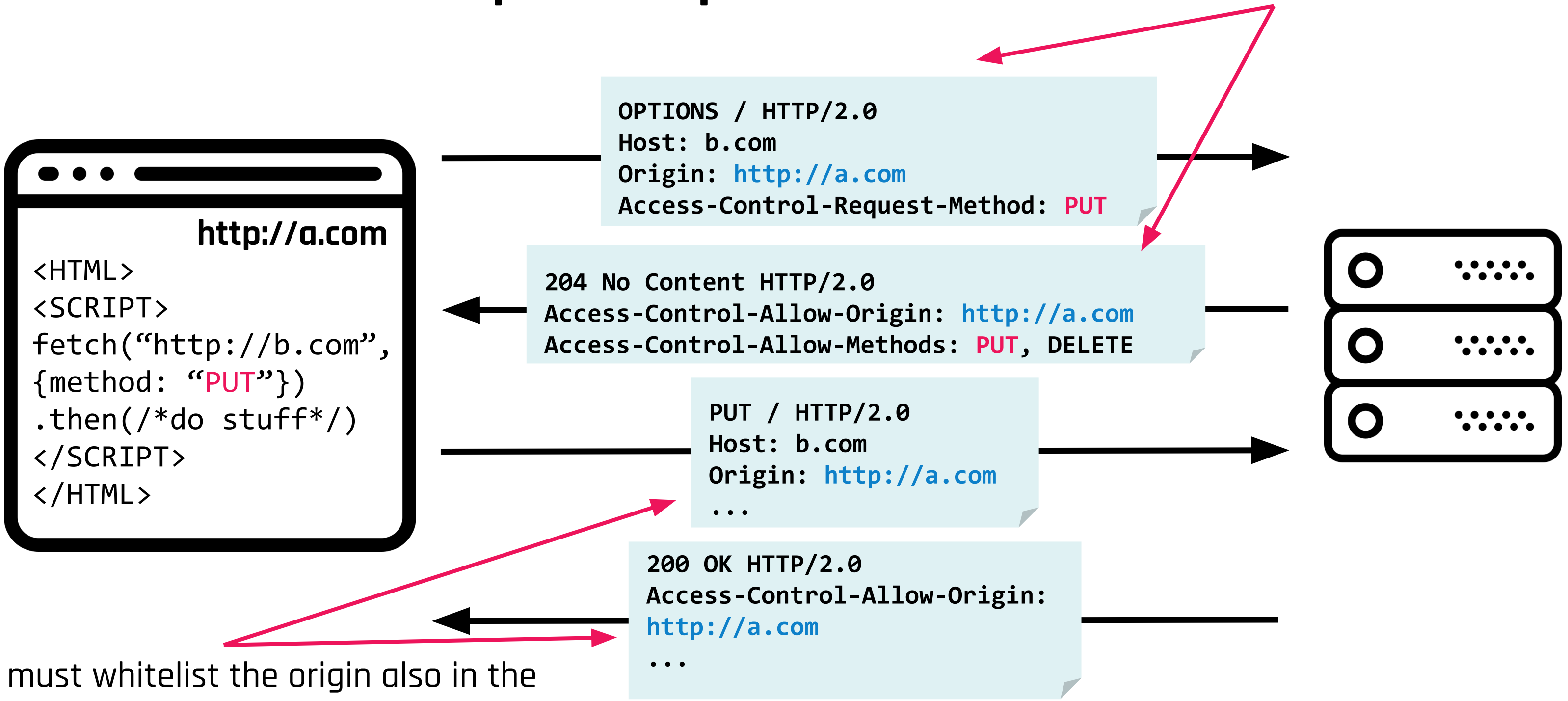
```
GET / HTTP/2.0
Host: b.com
Cookie: SID=12345
Origin: http://a.com
```

```
200 OK HTTP/2.0
Access-Control-Allow-Origin:
http://a.com
Access-Control-Allow-Credentials:
true
...
```

When credentials (cookies, TLS certificates, BasicAuth) are sent, the `Access-Control-Allow-Credentials` header must be provided and set to true

CORS with Non-Simple Requests

The server whitelisted http://a.com and allows the usage of the PUT method: the browser can perform the actual request



```
<HTML>
<SCRIPT>
fetch("http://b.com",
{method: "PUT"})
.then(/*do stuff*/)
</SCRIPT>
</HTML>
```

```
OPTIONS / HTTP/2.0
Host: b.com
Origin: http://a.com
Access-Control-Request-Method: PUT
```

```
204 No Content HTTP/2.0
Access-Control-Allow-Origin: http://a.com
Access-Control-Allow-Methods: PUT, DELETE
```

```
PUT / HTTP/2.0
Host: b.com
Origin: http://a.com
...
```

```
200 OK HTTP/2.0
Access-Control-Allow-Origin:
http://a.com
...
```

The server must whitelist the origin also in the actual request to make the response content available to the script

CORS Headers

- Request headers (used in pre-flight request):
  - `Access-Control-Request-Method`: the HTTP method that will be used in the actual request
  - `Access-Control-Request-Headers`: list of custom HTTP headers that will be sent in the actual request
- Response headers:
  - `Access-Control-Allow-Origin`: used to whitelist origins, allowed values are `null`, * or an origin (value * **cannot** be used if `Access-Control-Allow-Credentials` is specified)
  - `Access-Control-Allow-Methods`: list of allowed HTTP methods
  - `Access-Control-Allow-Headers`: list of custom HTTP headers allowed
  - `Access-Control-Expose-Headers`: list of response HTTP headers that will be available to JS
  - `Access-Control-Allow-Credentials`: used when the request includes client credentials
  - `Access-Control-Max-Age`: used for caching pre-flight requests

Pitfalls in CORS Configurations

- Two different CORS specifications existed until recently:
  - **W3C**: allows a list of origins in `Access-Control-Allow-Origin`
  - **Fetch API**: allows a single origin in `Access-Control-Allow-Origin`
  - Browsers implement CORS from the Fetch API (and the W3C one is now deprecated)

- Browsers implementations complicate CORS configuration:
  - Server-side applications need custom code to validate allowed origins rather than just providing a static header with all the whitelisted origins

Pitfall #1 - Broken Origin Validation

- Snippet of `nginx` configuration setting the CORS header:

```
if ($http_origin ~ "http://(example.com|foo.com)") {
    add_header "Access-Control-Allow-Origin" $http_origin;
}
```

- Allowed origins:
  - `http://example.com`
  - `http://foo.com`
  - `http://example.com.evil.com`

Reference: Jianjun Chen et al. We Still Don't Have Secure Cross-Domain Requests

Pitfall #2 - The `null` origin

- The `Access-Control-Allow-Origin` header may specify the `null` value

- Browsers may send the Origin header with a `null` value in particular conditions:
  - Cross-site redirects
  - Requests using the `file:` protocol
  - Sandboxed cross-origin requests

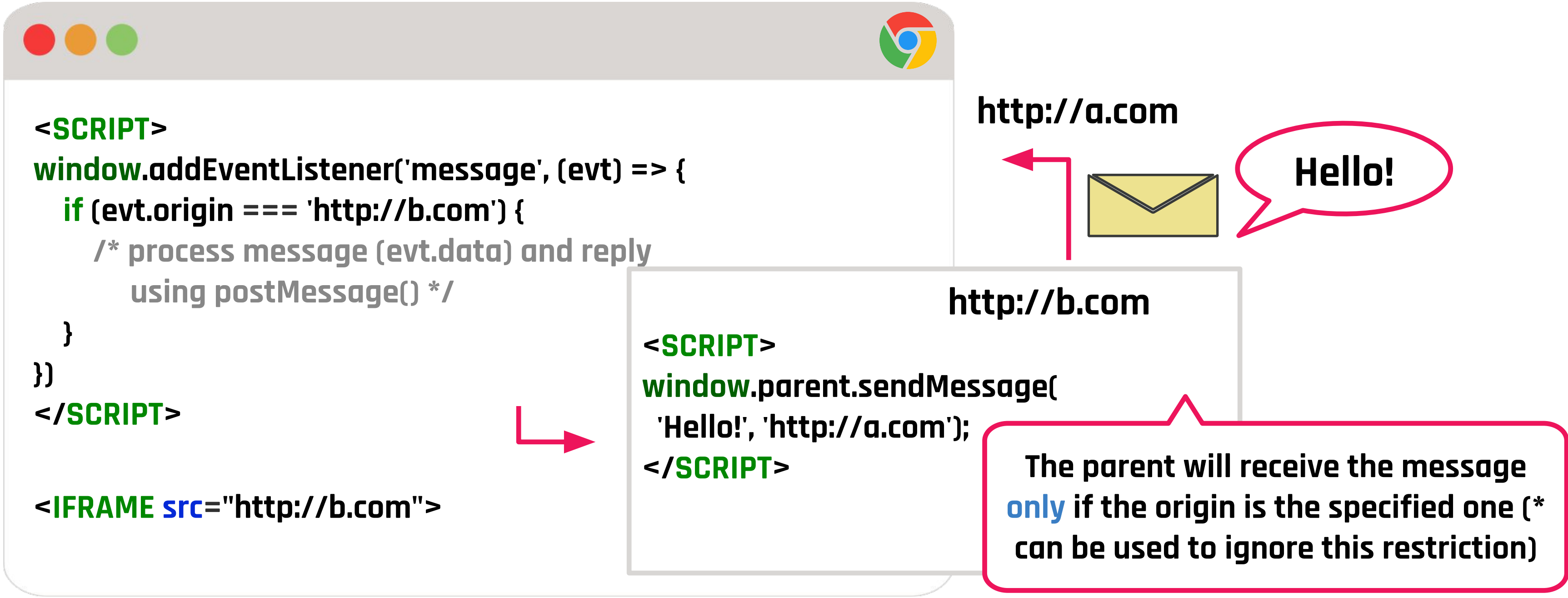
- An attacker can forge requests with the `null` Origin header by performing cross-origin requests from a sandboxed iframe

Reference: Jianjun Chen et al. We Still Don't Have Secure Cross-Domain Requests

Client-side Messaging

Client-Side Messaging via postMessage

- **postMessage** is a web API that enables **cross-origin message exchanges** between windows (e.g., embedded frame with embedder frame)



http://a.com

```
<SCRIPT>
window.addEventListener('message', (evt) => {
   if (evt.origin === 'http://b.com') {
      /* process message (evt.data) and reply
          using postMessage() */
   }
})
</SCRIPT>


<IFRAME src="http://b.com">
```

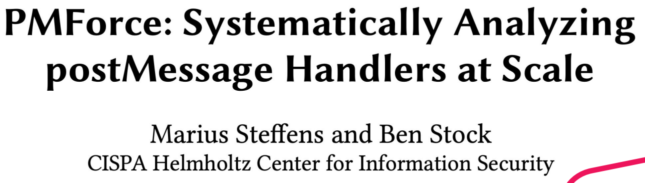
http://b.com

```
<SCRIPT>
window.parent.sendMessage(
  'Hello!', 'http://a.com');
</SCRIPT>
```

The parent will receive the message only if the origin is the specified one (* can be used to ignore this restriction)

Validating Incoming Messages

- Message handlers should validate the origin field of incoming messages in order to communicate only with the desired origins
- Failures to do so may result in security vulnerabilities, e.g., when the received message is evaluated as a script or unsafely embedded into a page
- A recent study found 377 vulnerable message handlers on the top 100k sites
  - Some of these lacked origin checking, others were implementing it in the wrong way (e.g., substring match)



PMForce: Systematically Analyzing postMessage Handlers at Scale

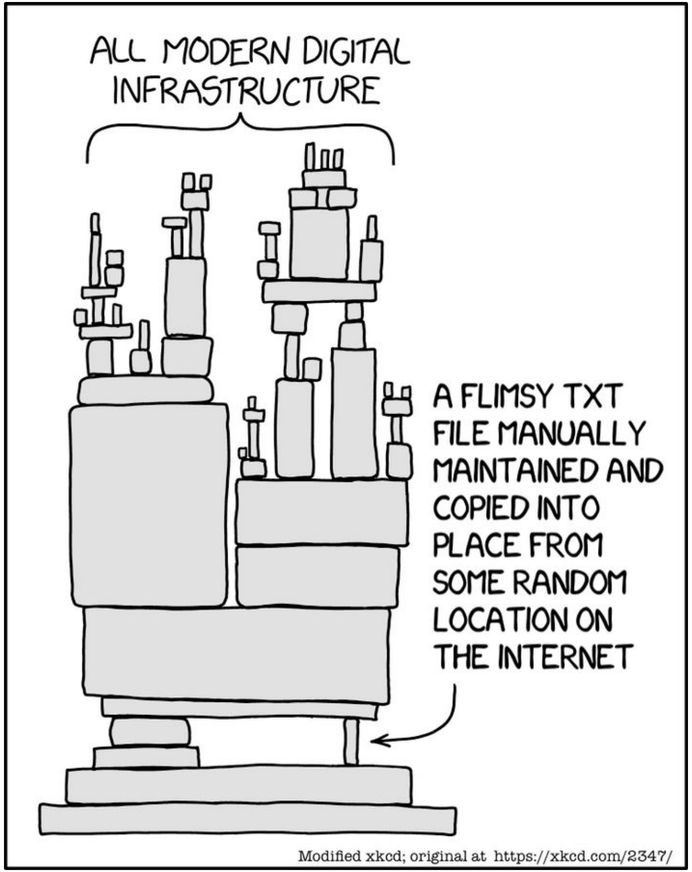
Marius Steffens and Ben Stock
CISPA Helmholtz Center for Information Security

ACM CCS 2020

Cookies

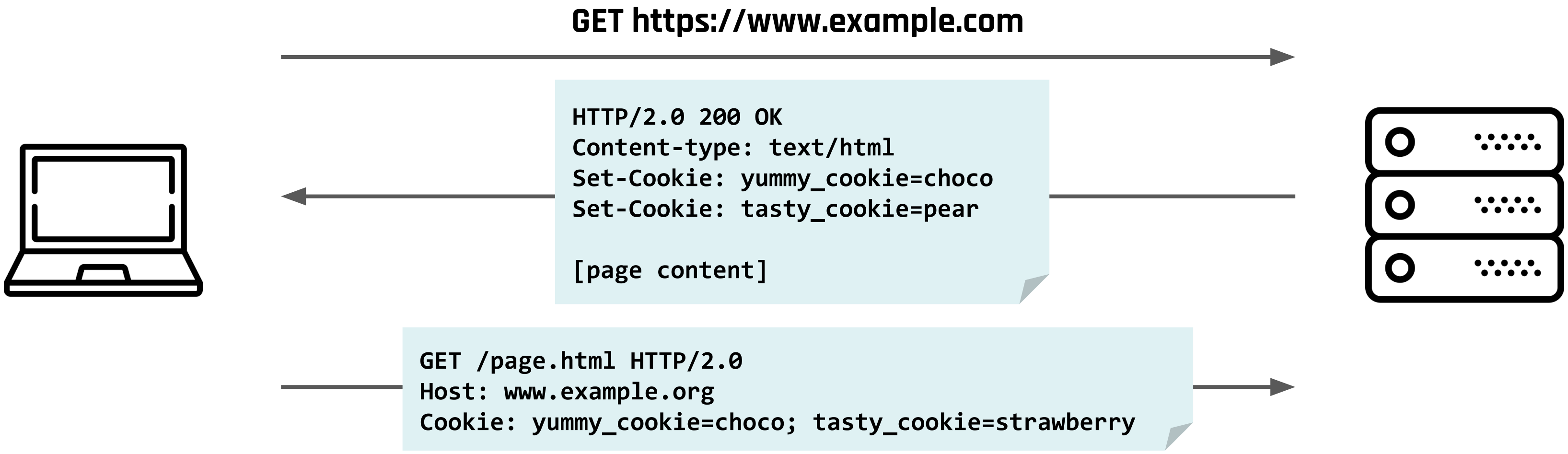
See this document for an example

[Source]


ALL MODERN DIGITAL INFRASTRUCTURE
A FLIMSY TXT FILE MANUALLY MAINTAINED AND COPIED INTO PLACE FROM SOME RANDOM LOCATION ON THE INTERNET

Modified xkcd; original at https://xkcd.com/2347/

Cookies



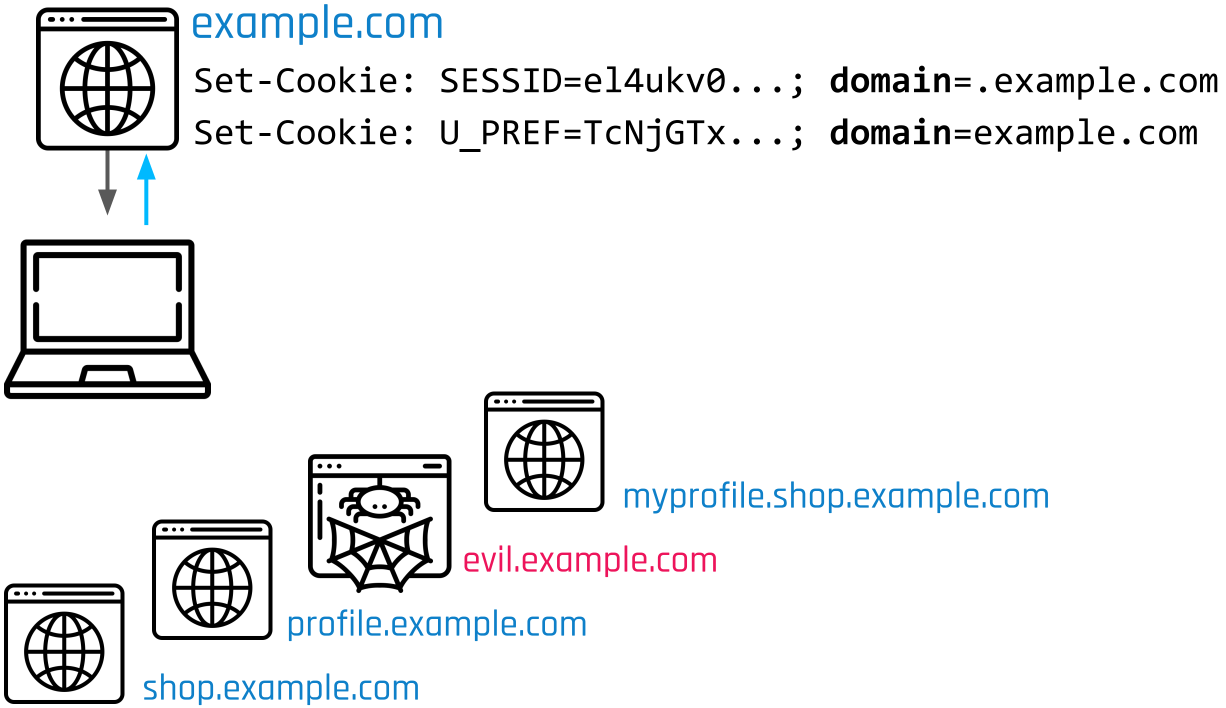
```
HTTP/2.0 200 OK
Content-type: text/html
Set-Cookie: yummy_cookie=choco
Set-Cookie: tasty_cookie=pear

[page content]
```

```
GET /page.html HTTP/2.0
Host: www.example.org
Cookie: yummy_cookie=choco; tasty_cookie=strawberry
```

Attributes					Flags	
Expires	Max-Age	Domain	Path	SameSite	Secure	HttpOnly

Scope of Cookies (1)

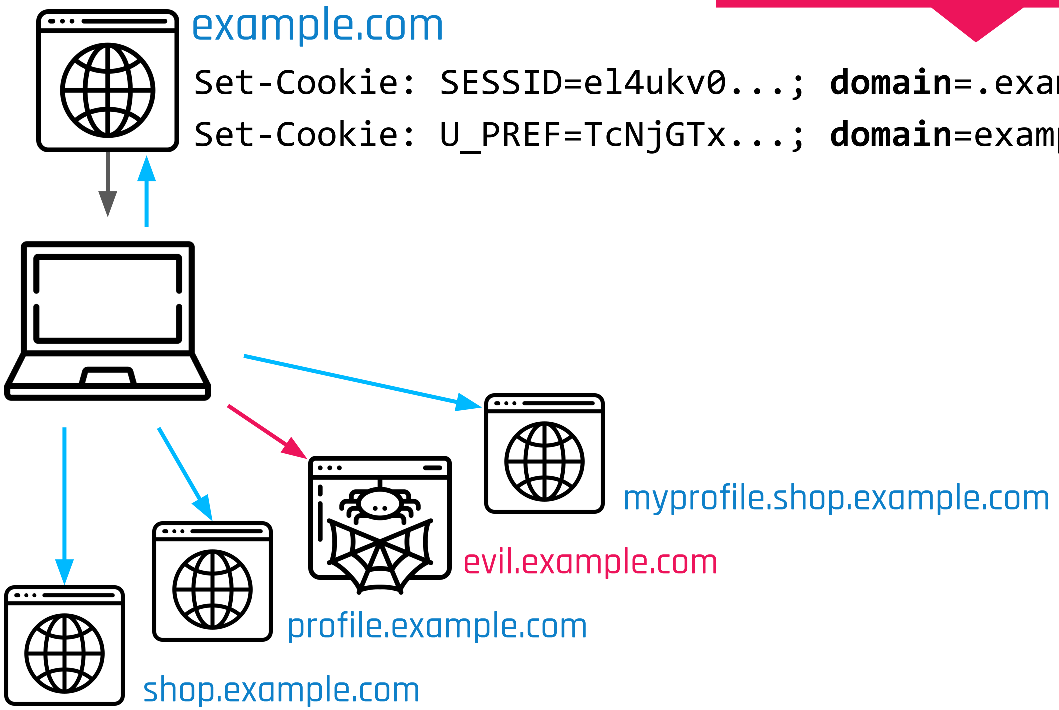


example.com

```
Set-Cookie: SESSID=el4ukv0...; domain=.example.com
Set-Cookie: U_PREF=TcNjGTx...; domain=example.com
```

Scope of Cookies (2)

The “dot” makes no difference



```
Set-Cookie: SESSID=el4ukv0...; domain=.example.com
Set-Cookie: U_PREF=TcNjGTx...; domain=example.com
```

- ➔ The **domain attribute** widens the scope of a cookie to all subdomains
- ➔ If one subdomain is compromised, such cookies will be **leaked to unauthorized parties**
- ➔ **To restrict the scope of a cookie to the domain that set it, the domain attribute must not be specified**

The Domain Attribute

- If the attribute is not set, the cookie is attached only to requests to the domain who set the cookie
- If the attribute is set, the cookie is attached to requests to the specified domain and all its subdomains
  - The value can be any suffix of the domain of the page setting the cookie, up to the registrable domain
  - A related-domain attacker can set cookies that are sent to the target website!

Domain setting the cookie	Value of the Domain attribute	Allowed?	Reason
a.b.example.com	example.com	Yes	the attribute's value is the registrable domain
www.example.ac.at	ac.at	No	ac.at is a public suffix
a.example.com	b.example.com	No	the attribute's value is not a suffix of a.example.com

Cookie Attributes

- The **Path** attribute can be used to restrict the scope of a cookie, i.e., the cookie is attached to a request only if its path is a prefix of the path of the request's URL. Useful, e.g.: example.com/~userA vs example.com/~userB
  - If the attribute is not set, the path is that of the page setting the cookie
  - If the attribute is set, there are no restrictions on its value

- If the **Secure** attribute is set, the cookie will be attached only to HTTPS requests (confidentiality)
  - Since recently, browsers prevent Secure cookies to be set (or overwritten) by HTTP requests (integrity)

Cookie Attributes

- If the **HttpOnly** attribute is set, JavaScript **cannot read the value** of the cookie via **document.cookie**
  - **No integrity** is provided: a script can overflow the cookie jar, so that **older cookies are deleted**, and then set a new cookie with the desired value
  - Prevents the theft of sensitive cookies (e.g., those storing session identifiers) in case of **XSS vulnerabilities**

- **Max-Age** or **Expires** define when the cookie **expires**
  - When both are unset, the cookie is deleted when the browser is closed
  - When **Max-Age** is a negative number or **Expires** is a date in the past, the cookie is **deleted** from the cookie jar
  - If both are specified, **Max-Age** has precedence

The SameSite Attribute

- A request is **cross-site** if the domain of the target URL of the request and that of the page triggering the request **do not share the same registrable domain**
  - A request from **a.example.com** to **b.example.com** is **same-site** (the registrable domain is example.com)
  - A request from **example.com** to **bank.com** is **cross-site**

- The **SameSite** attribute controls whether the cookie should be attached to cross-site requests:
  - **Strict**: the cookie is never attached to cross-site requests
  - **Lax**: the cookie is sent even in case of cross-domain requests, but then there must be a change in top-level navigation (user realizes it!)
  - **None**: the cookie is always attached to all cross-site requests

CSRF Protection

Recent Changes to Cookies (Feb. 2020)

- **SameSite = Lax by default**
  - Cookies that do not explicitly set the SameSite attribute are treated as if they specified SameSite = Lax
  - Before February 2020, these cookies were treated as if they set SameSite = None

- **SameSite = None implies Secure**
  - Cookies with attribute SameSite set to None are discarded by the browser if the Secure attribute is specified as well

SOP for Reading Cookies

- A cookie is attached to a request towards the URL *u* if the following constraints are satisfied:
    - if the Domain attribute is set, it is a domain-suffix of the hostname of *u*, otherwise the hostname of u must be equal to the domain of the page who set the cookie
    - the Path attribute is a prefix of the path of *u*
    - if the Secure attribute is set, the protocol of *u* must be HTTPS
    - if the request is cross-site, take into account the requirements imposed by the SameSite attribute

Example

Name	Value	Domain attribute	Path	Secure	Domain who set the cookie	SameSite
uid	u1	not set	/	Yes	site.com	None
sid	s2	site.com	/admin	Yes	site.com	Strict
lang	en	site.com	/	No	prefs.site.com	Lax

- Which cookies are attached to a cross-site request from https://www.example.com (triggered by the user clicking on a link, changing the top-level navigation context) to:

http://site.com/	→	lang=en
https://site.com/	→	uid=u1;lang=en
https://site.com/admin/	→	uid=u1;lang=en
https://a.site.com/admin/	→	lang=en

sid=s2 is not included because is a cross-site request

Cookies Protocol Issues

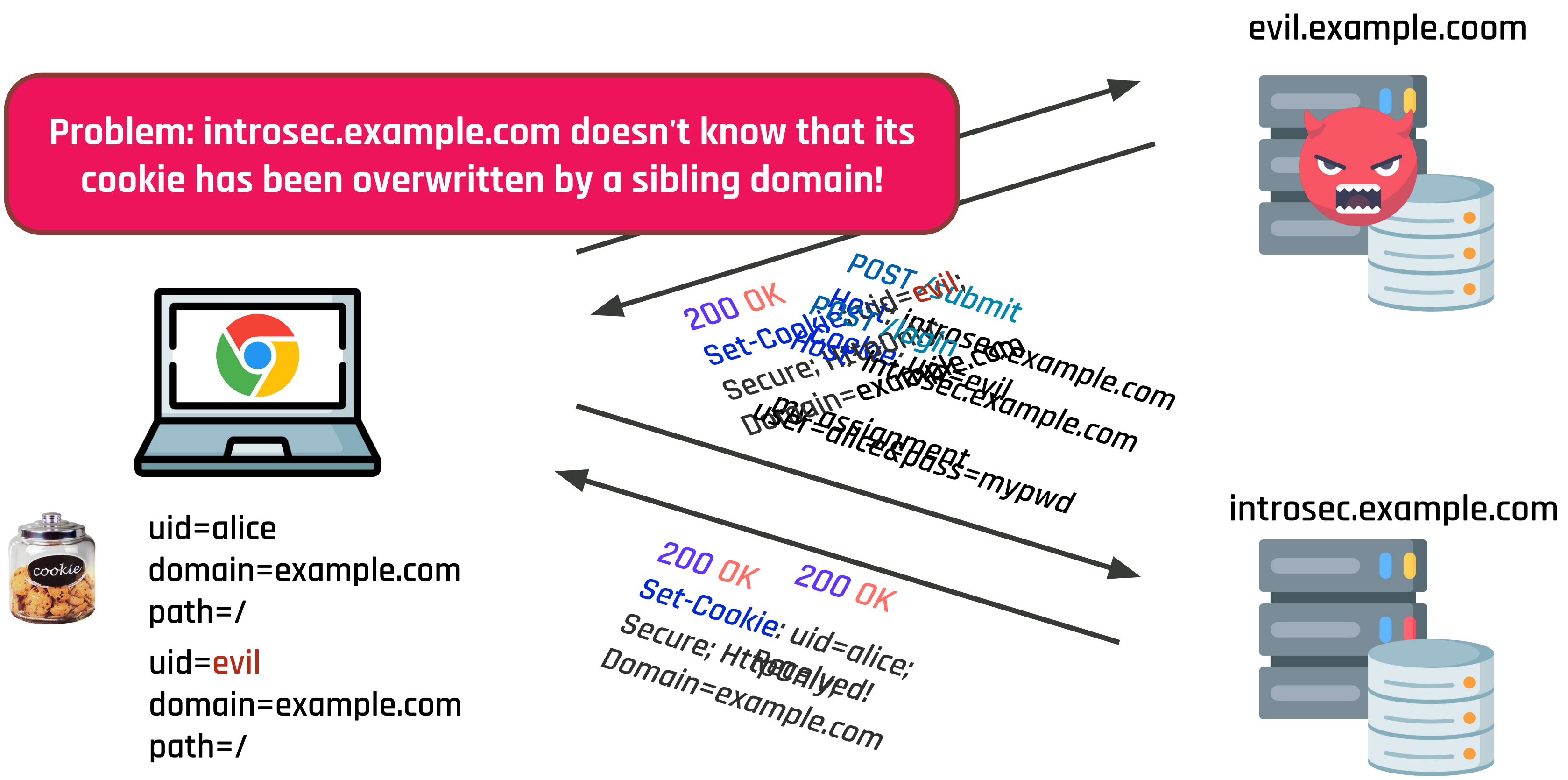
- The Cookie header, which contains the cookies attached by the browser, only contains the **name** and the **value** of the attached cookies
    - the server cannot know if the cookie was set over a secure connection
    - the server does not know which domain has set the received cookie
    - RFC 2109 has an option for including domain, path in the Cookie header, but it is not supported by any browser (and is now deprecated)

Cookie Tossing

- By setting the domain attribute to e.g., `.domain.com`, subdomains can force a cookie to other subdomains, related-domains and even to the apex domain
- The key of the cookie jar is given by the tuple (name, domain, path). When cookies are sent to a given endpoint, attributes are not included (only the name/value pair is sent by the browser)
- Servers have no way to tell which cookie is from which domain/path
- Most servers accept the first occurrence of cookies with the same name
- Most browsers place cookies created earlier first
- Most browsers place cookies with longer paths before cookies with shorter paths
- Impact: Bypass CSRF protections, Login CSRF, Session Fixation, ...

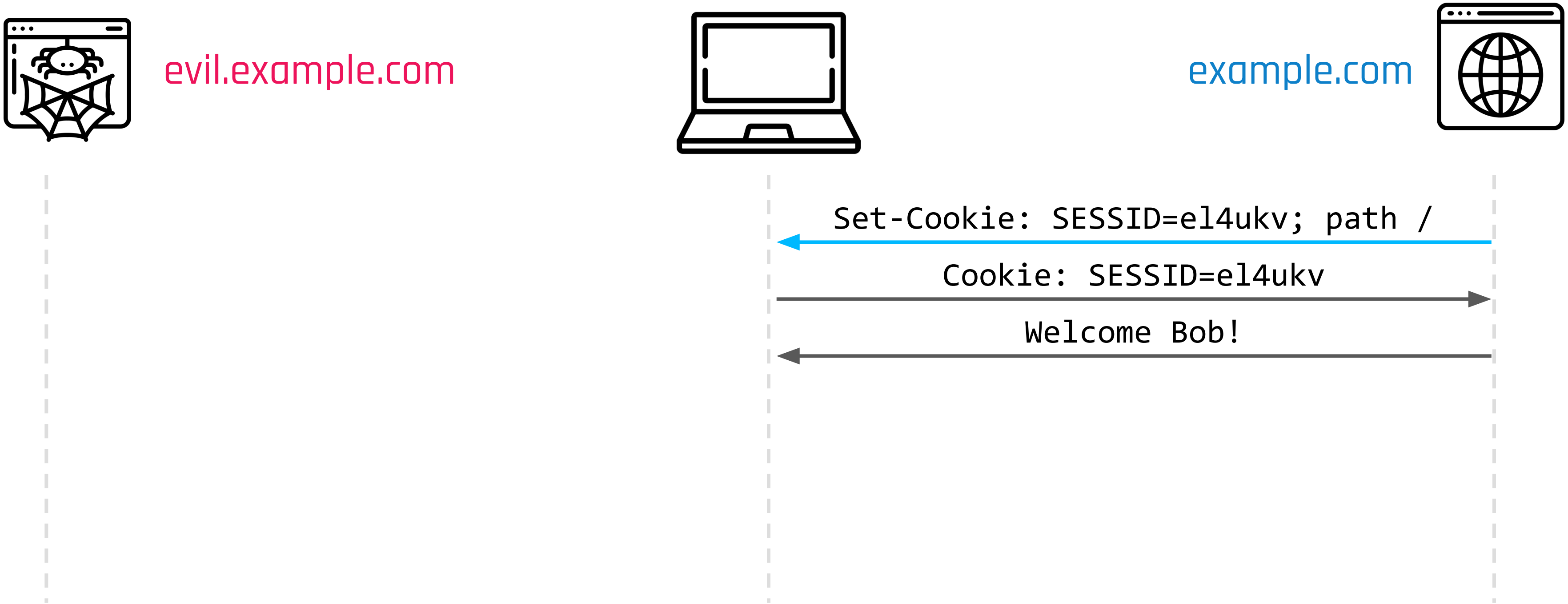
Reference: V. Martí. Yummy Cookies Across Domains

Cookie Overwrite Vulnerabilities



evil.example.coom
Problem: introsec.example.com doesn't know that its cookie has been overwritten by a sibling domain!
200 OK
Set-Cookie: uid=evil; Secure; HttpOnly; Domain=example.com
POST /submit
Host: introsec.example.com
Cookie: uid=evil
assignment
POST /login
Host: introsec.example.com
user=alice&pass=mypwd
introsec.example.com
200 OK
Set-Cookie: uid=alice; Secure; HttpOnly; Domain=example.com
200 OK
Received!
cookie
uid=alice
domain=example.com
path=/
uid=evil
domain=example.com
path=/

Example Login (1)



evil.example.com

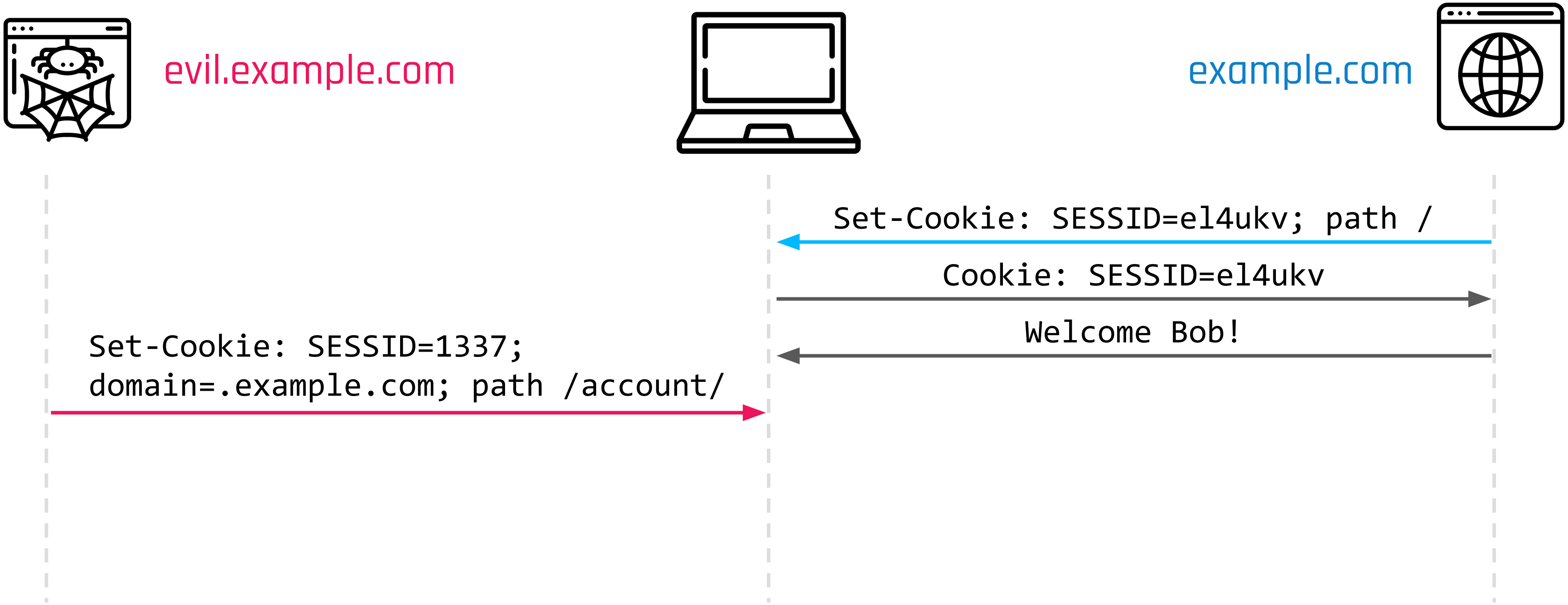
example.com

`Set-Cookie: SESSID=el4ukv; path /`

`Cookie: SESSID=el4ukv`

`Welcome Bob!`

Example Login (2)



evil.example.com

example.com

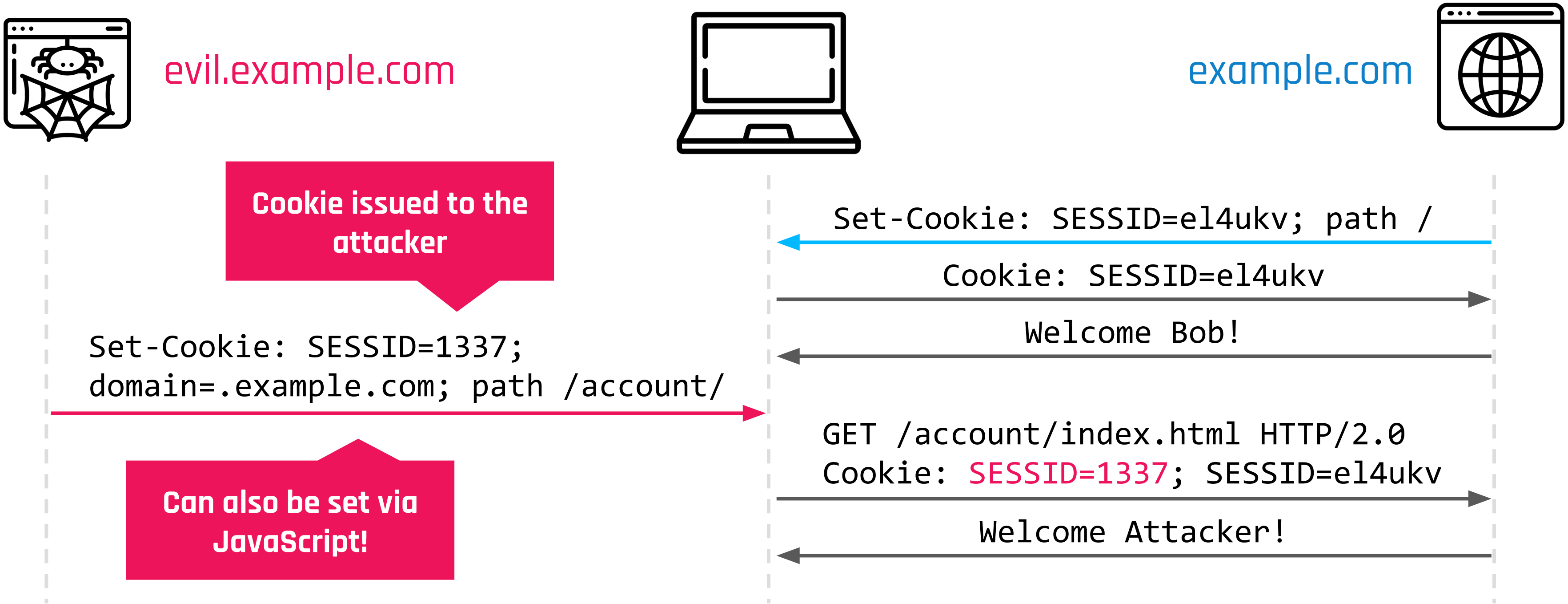
Set-Cookie: SESSID=el4ukv; path /

Cookie: SESSID=el4ukv

Welcome Bob!

Set-Cookie: SESSID=1337;
domain=.example.com; path /account/

Example Login (3)



evil.example.com

example.com

```
Set-Cookie: SESSID=el4ukv; path /
```

```
Cookie: SESSID=el4ukv
```

```
Welcome Bob!
```

Cookie issued to the attacker

```
Set-Cookie: SESSID=1337;
domain=.example.com; path /account/
```

Can also be set via JavaScript!

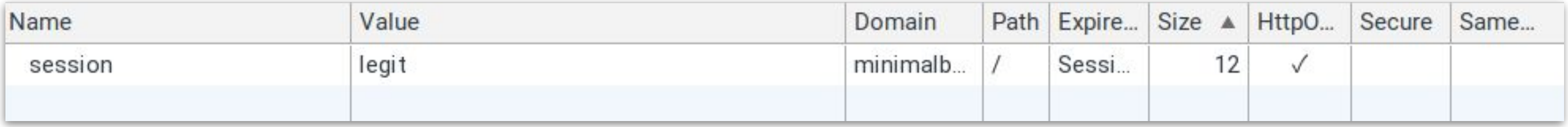
```
GET /account/index.html HTTP/2.0
Cookie: SESSID=1337; SESSID=el4ukv
```

```
Welcome Attacker!
```

Cookie Jar Overflow (1)

- Browsers are limited on the number of cookies an apex domain can have
- When there is no space left, older cookies are deleted
- Attackers can thus overflow the cookie jar to “overwrite” HttpOnly cookies or to bypass cookie tossing protection on servers that block requests with multiple cookies having the same name

Tested on Chrome 79.0.3945.36



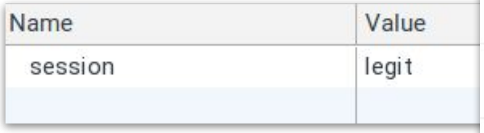
Name	Value	Domain	Path	Expire...	Size ▲	HttpO...	Secure	Same...
session	legit	minimalb...	/	Sessi...	12	✓		

Reference: V. Martí. Yummy Cookies Across Domains

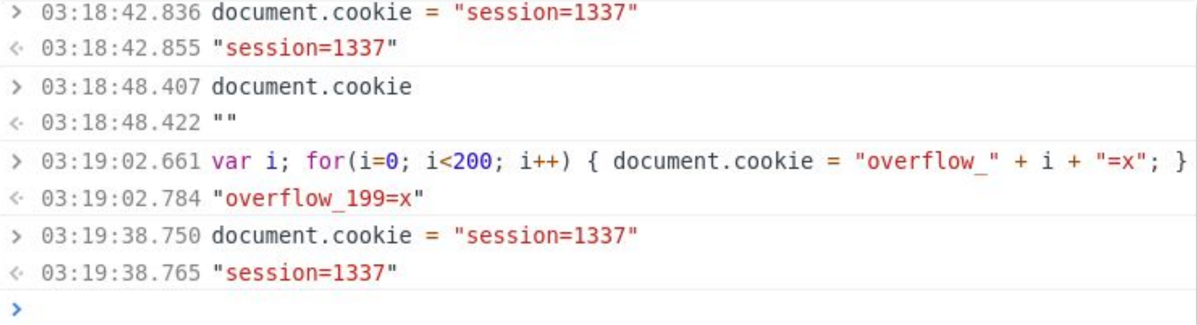
Cookie Jar Overflow (2)

- Browsers are limited on the number of cookies an apex domain can have
- When there is no space left, older cookies are deleted
- Attackers can thus overflow the cookie jar to “overwrite” HttpOnly cookies or to bypass cookie tossing protection on servers that block requests with multiple cookies having the same name

Tested on Chrome 79.0.3945.36



Name	Value
session	legit



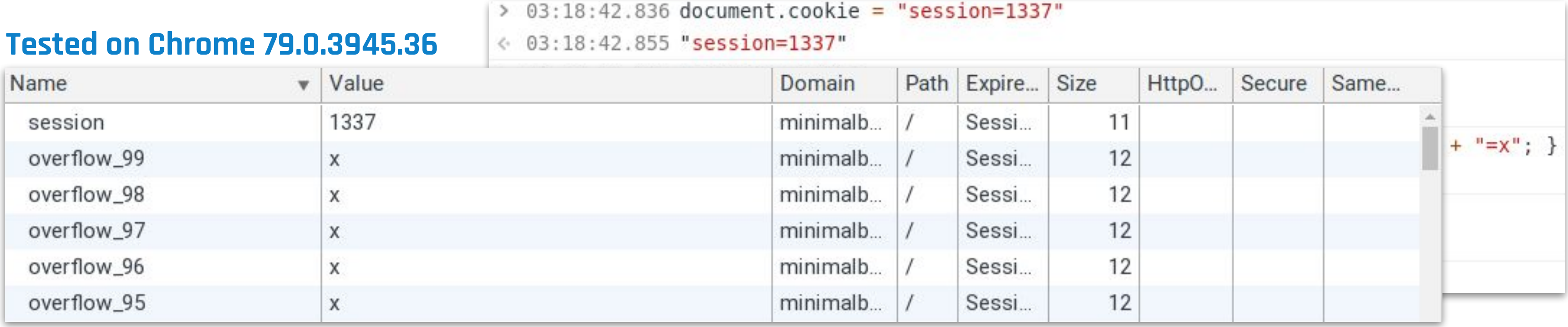
```
> 03:18:42.836 document.cookie = "session=1337"
<· 03:18:42.855 "session=1337"
> 03:18:48.407 document.cookie
<· 03:18:48.422 ""
> 03:19:02.661 var i; for(i=0; i<200; i++) { document.cookie = "overflow_" + i + "=x"; }
<· 03:19:02.784 "overflow_199=x"
> 03:19:38.750 document.cookie = "session=1337"
<· 03:19:38.765 "session=1337"
>
```

Reference: V. Martí. Yummy Cookies Across Domains

Cookie Jar Overflow (3)

- Browsers are limited on the number of cookies an apex domain can have
- When there is no space left, older cookies are deleted
- Attackers can thus overflow the cookie jar to “overwrite” HttpOnly cookies or to bypass cookie tossing protection on servers that block requests with multiple cookies having the same name

Tested on Chrome 79.0.3945.36



```
> 03:18:42.836 document.cookie = "session=1337"
< 03:18:42.855 "session=1337"
+ "=x"; }
```

Name	Value	Domain	Path	Expire...	Size	HttpO...	Secure	Same...
session	1337	minimalb...	/	Sessi...	11			
overflow_99	x	minimalb...	/	Sessi...	12			
overflow_98	x	minimalb...	/	Sessi...	12			
overflow_97	x	minimalb...	/	Sessi...	12			
overflow_96	x	minimalb...	/	Sessi...	12			
overflow_95	x	minimalb...	/	Sessi...	12			

Reference: V. Martí. Yummy Cookies Across Domains

Cookie Prefixes

- Cookie prefixes have been proposed to provide to the server more information on the security guarantees provided by cookies:
  - ○ **__Secure-**: if a cookie name has this prefix, it will only be accepted by the browser if it is marked as **Secure**
  - ○ **__Host-**: If a cookie name has this prefix, it will only be accepted by the browser if it is marked **Secure**, does not include a **Domain** attribute, and has the **Path** attribute set to /

Integrity w.r.t. network attackers

Integrity w.r.t. related-domain attackers

Cookies are still **hard to use securely**, especially in the same site context.
Researchers even proposed disruptive approaches to get rid of cookies

Training challenge #11

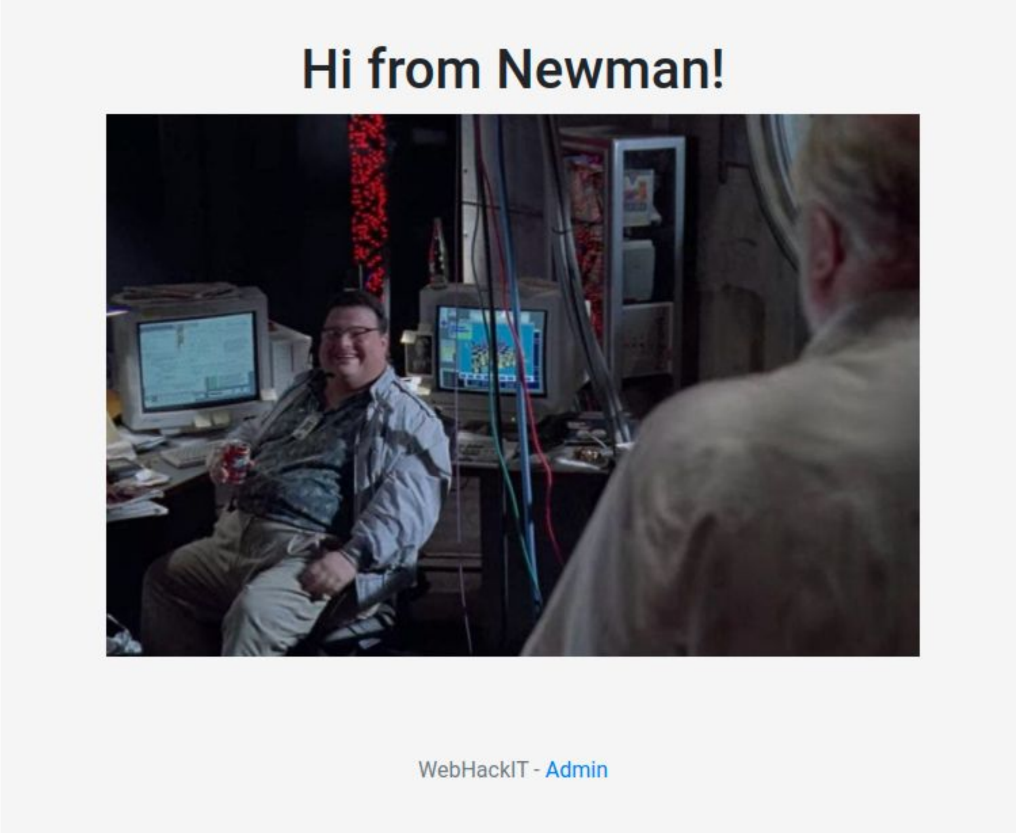
URL: https://training11.webhack.it

NOTE: THE CHALLENGE IS LIVE! TRY IT TO LEARN!

Description:

WebHackIT has deployed the system that was running Jurassic Park. However, security was not always taken into account by Newman…

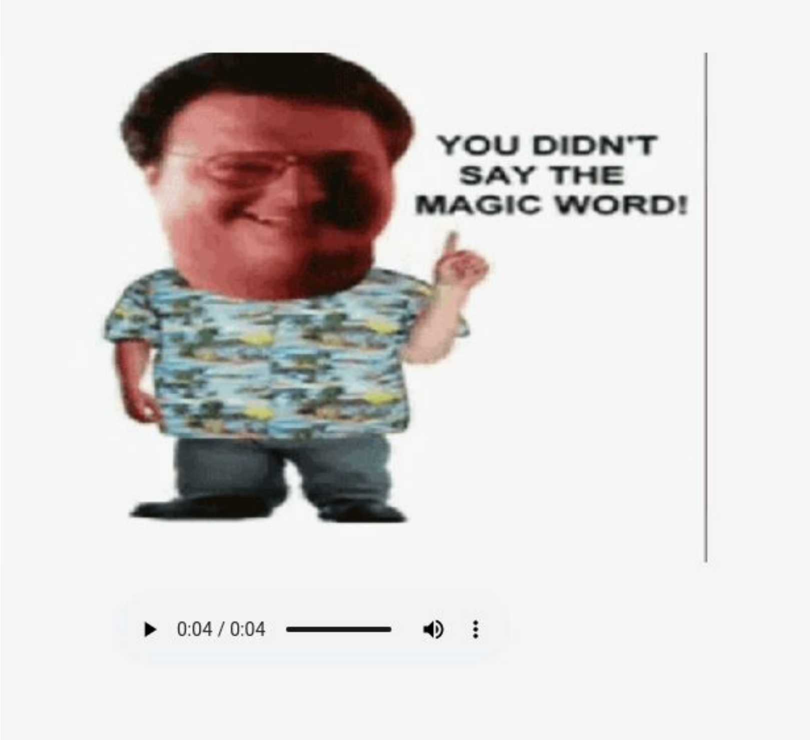
Page: /



Hi from Newman!

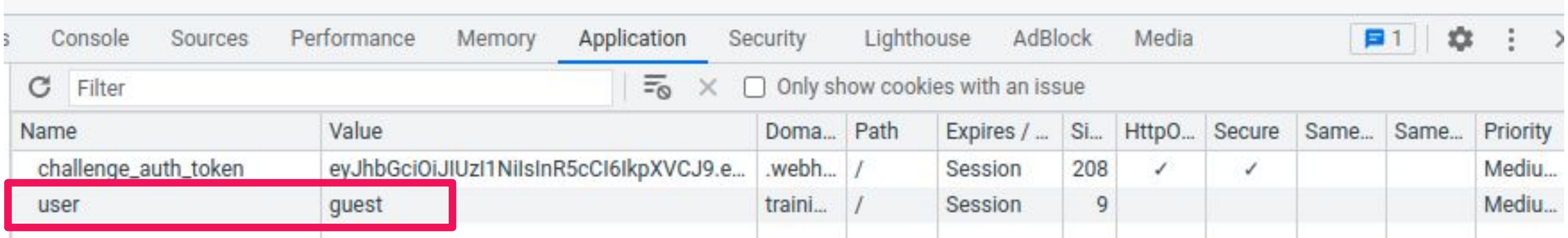
WebHackIT - Admin

Page: /admin



Analysis

- The application has an admin page
- however, no login form is available
- hence, the authentication is performed in some other ways...
- if we look for cookies, we can find something:



□ Only show cookies with an issue

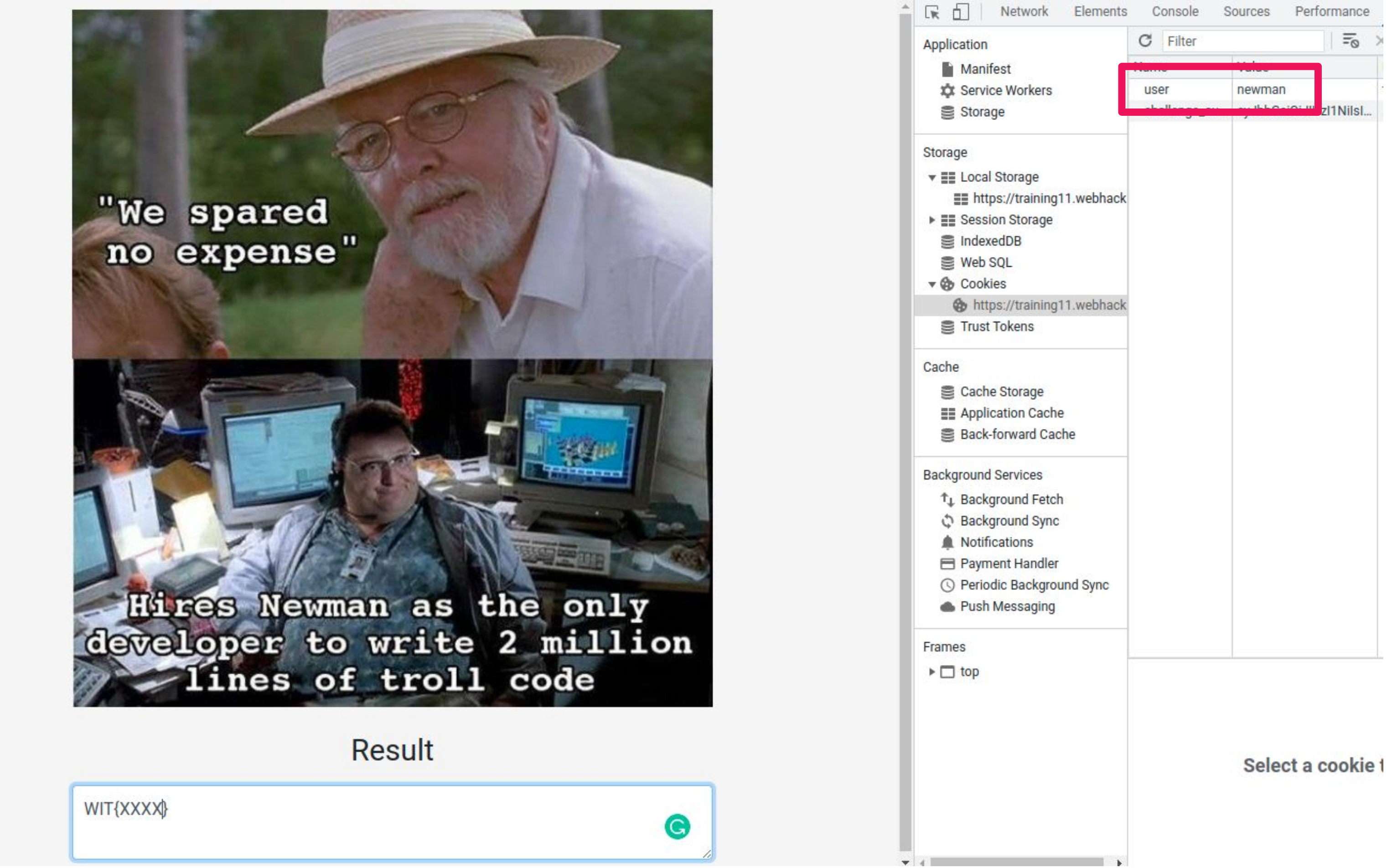
Name	Value	Doma...	Path	Expires / ...	Si...	HttpO...	Secure	Same...	Same...	Priority
challenge_auth_token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.e...	.webh...	/	Session	208	✓	✓			Mediu...
user	guest	traini...	/	Session	9					Mediu...

What can go wrong?

Problems

- We can manipulate cookies as we wish…

Let us try to change the value of the user cookie…



```
user      newman
```

Result

WIT{XXXX}