Pragmatic Web Security

Security training for developers

The security model of the web

Philippe De Ryck

@PhilippeDeRyck – philippe@PragmaticWebSecurity.com

@PhilippeDeRyck

Dr. Philippe De Ryck

- Ph.D-level understanding of the web security landscape
- Google Developer Expert (not employed by Google)
- Author of the *primer on client-side web security*
- Course curator of the

course (https://secappdev.org)

@PhilippeDeRyck
https://PragmaticWebSecurity.com

Pragmatic Web Security

Custom training courses on web/API/JS frontend security

Technical writing, architectural security assessments and brief consultancy

THE SECURITY MODEL OF THE WEB

ORIGINS AND BROWSING CONTEXTS

SCRIPT EXECUTION CONTEXTS

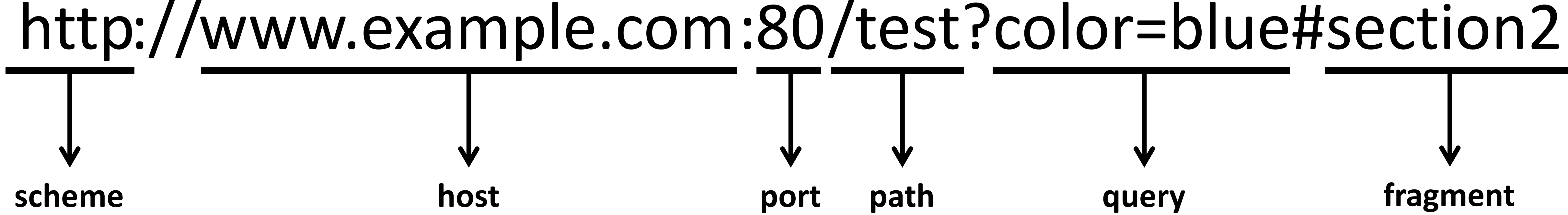
THE EVOLUTION OF CLIENT-SIDE SECURITY

MODERN COOKIE SECURITY

CONCLUSION

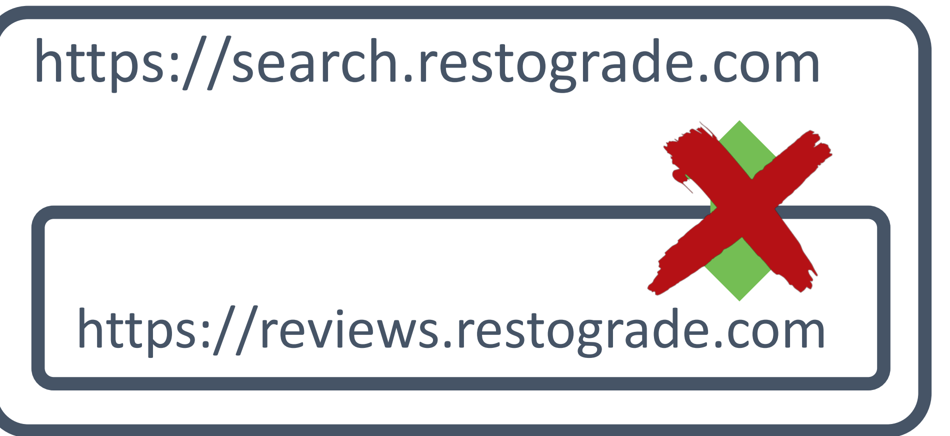
WHAT IS THE DEFINITION OF AN ORIGIN?

The concept of an origin



The Same-Origin Policy (SOP)

Content retrieved from one origin can freely interact with other content from that origin, but interactions with content from other origins are restricted



The origin as a principal in the browser

- Origins are used as a principal for making security decisions
  - The Same-Origin Policy governs interaction between contexts
  - The SOP affects the DOM and all its contents

- Other origin-protected resources in a modern browser
  - Permissions for sensitive features are also granted per origin
  - Client-side storage areas (Web Storage, IndexedDB, virtual file systems, ...)
  - Ability to send JavaScript-based XHR requests without CORS restrictions
    - Includes the capability to load resources and inspect their contents (e.g. JS source code)

- One of the most important aspects of web security is controlling your origin
  - Once an attacker runs code within your origin, it will be hard to provide any security

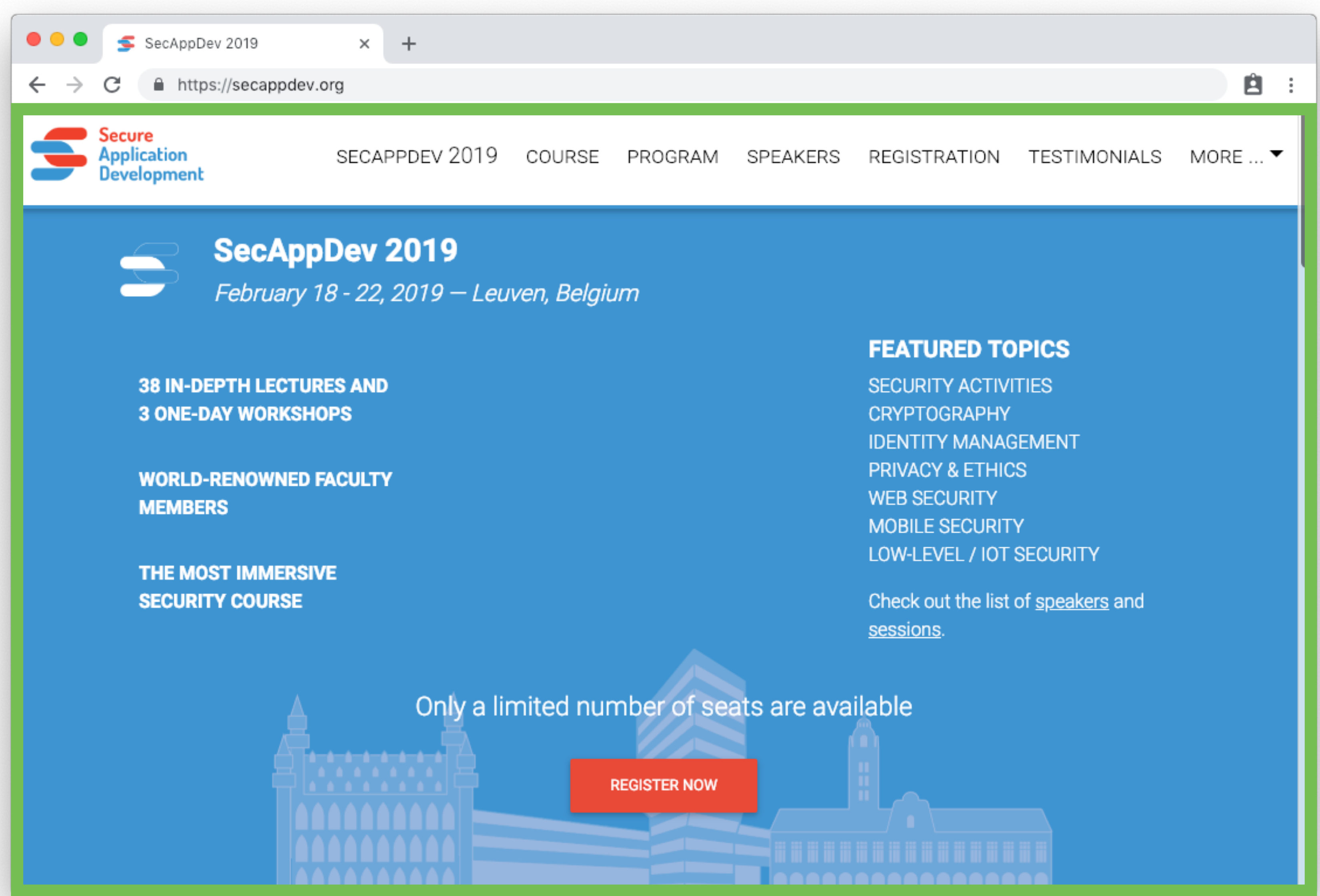
Security assumptions about the browser

- Web security builds on top of an extensive ***Trusted Computing Base***
  - If the server machine is compromised, anything is possible
  - If the user's machine or operating system is compromised, all bets are off
  - If the user is running a malicious browser, there are no security guarantees at all

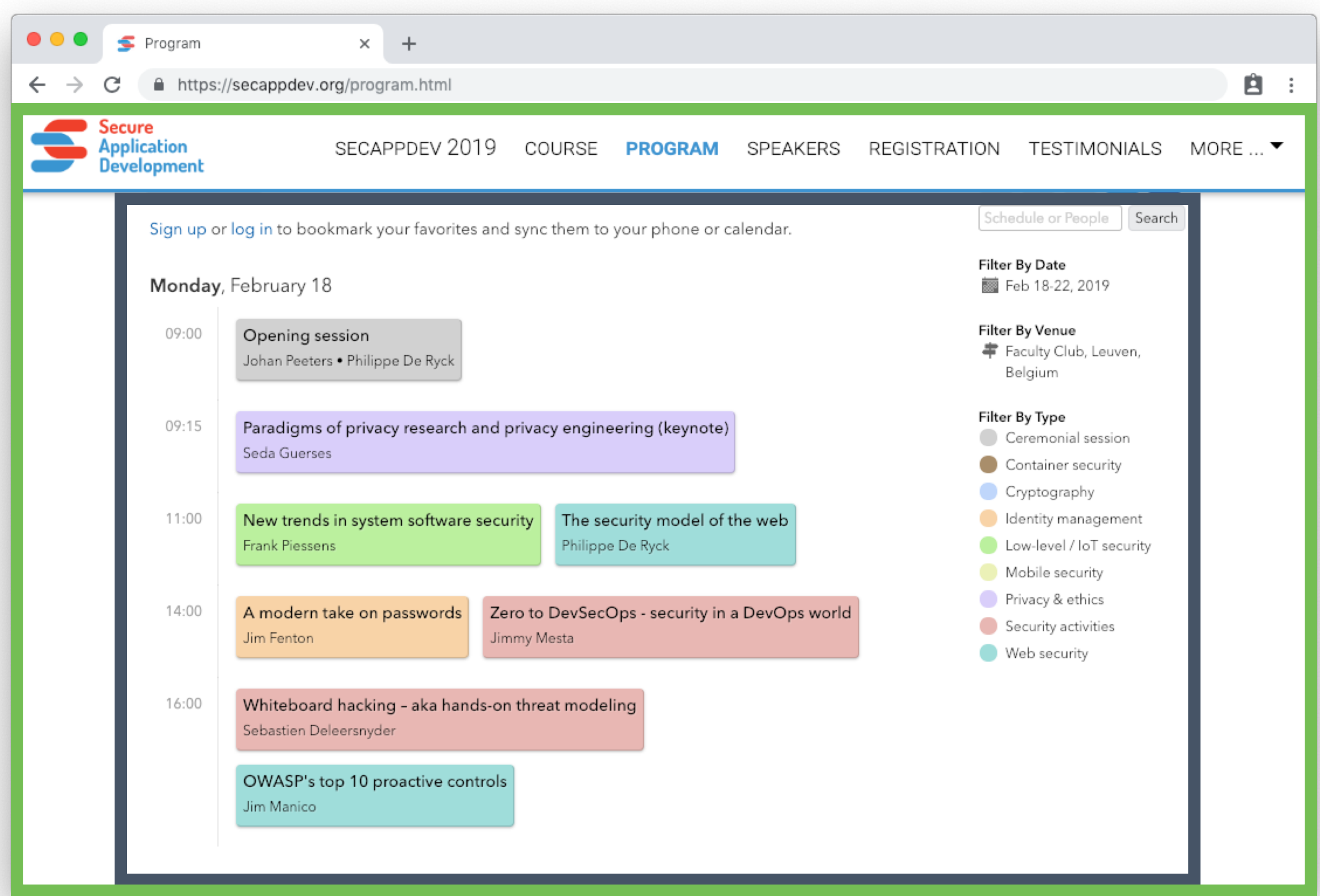
- Web security mechanisms and policies depend on a few security assumptions
  - The server-side execution environment of the application is secure (e.g., OS, JVM, …)
  - The client-side execution environment of the application is secure (e.g., OS, browser, …)
  - Anything else can potentially be compromised (e.g., network, third-party code, …)

```
chromium-browser --disable-web-security
```

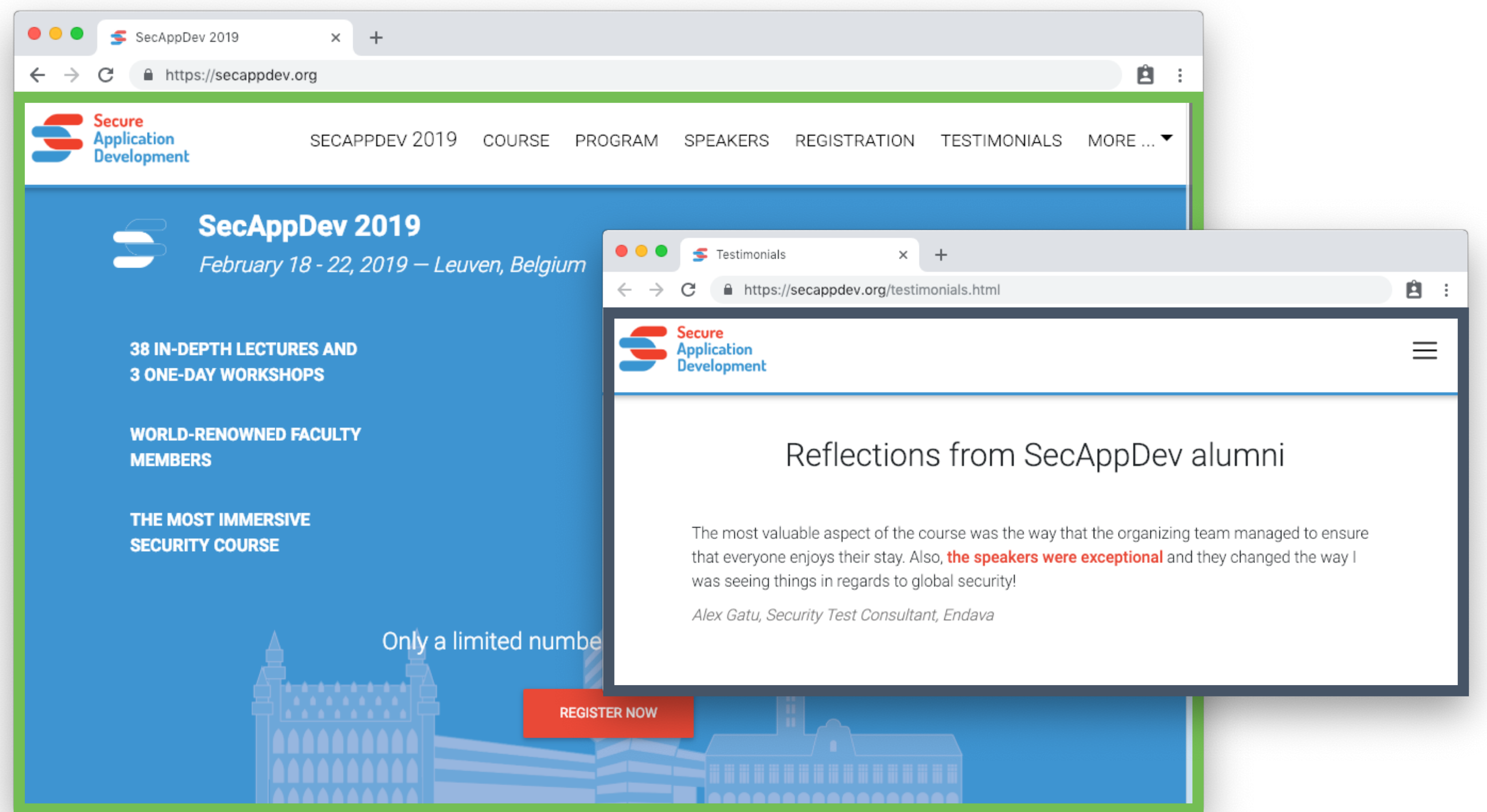
TOP-LEVEL BROWSING CONTEXTS



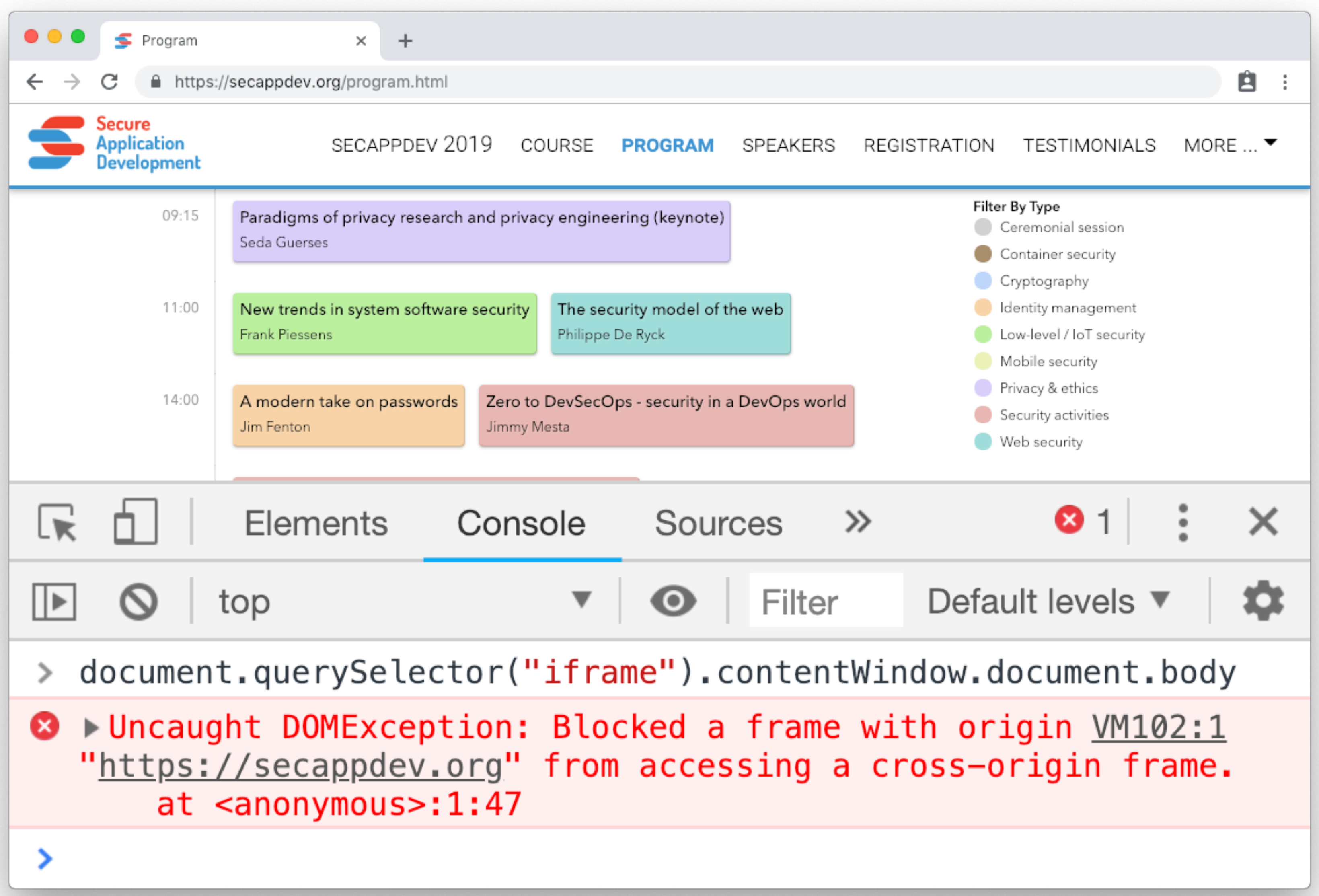
NESTED BROWSING CONTEXTS



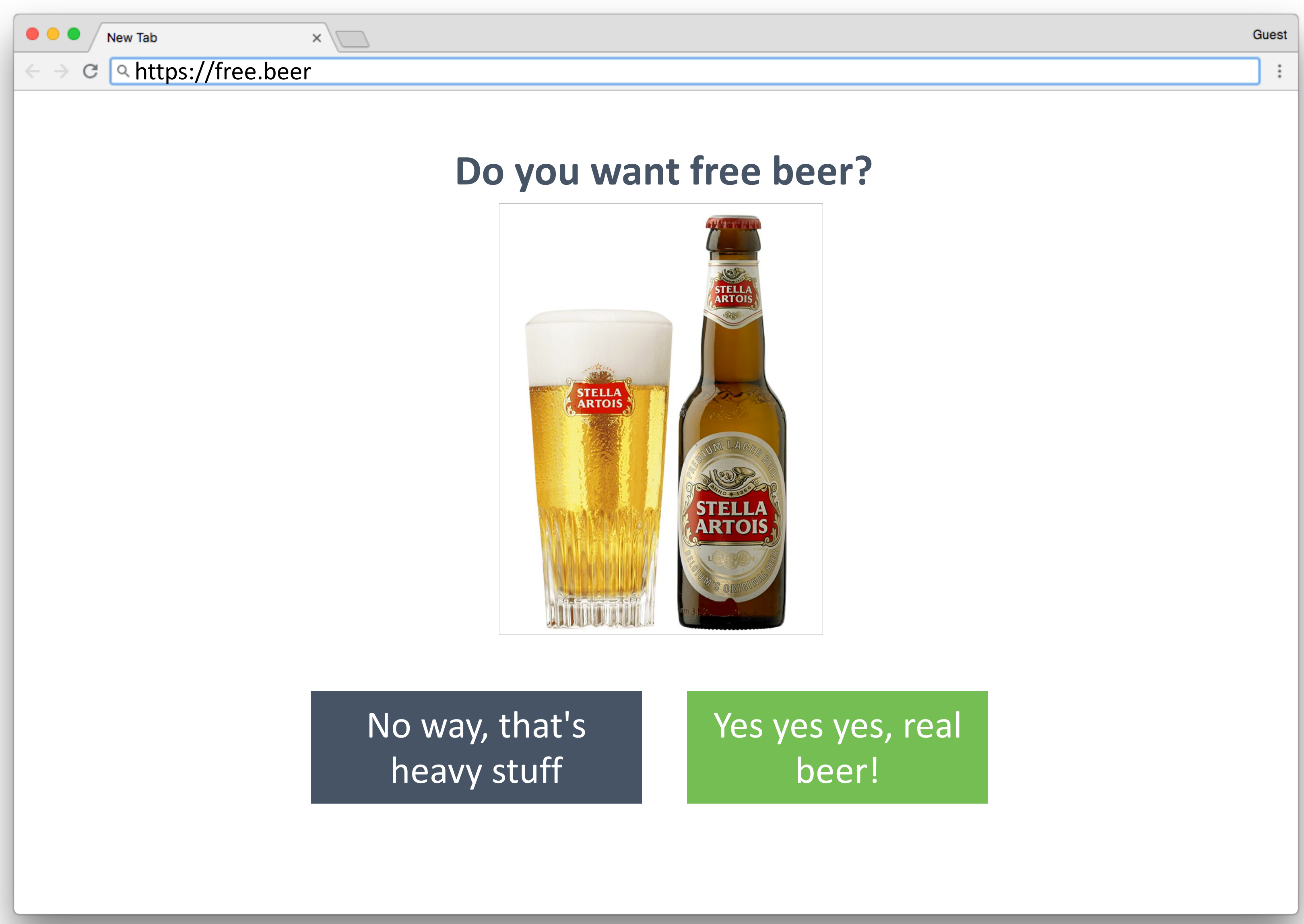
AUXILIARY BROWSING CONTEXTS



INTERACTIONS ARE GOVERNED BY THE SAME-ORIGIN POLICY



A CLICKJACKING SCENARIO



A CLICKJACKING SCENARIO

UI REDRESSING ATTACKS

- Clickjacking is one example of a UI redressing attack
  - The UI around a real application is redressed to confuse the user
- Another example reduces the viewport of the framed site
  - Only one UI element remains visible, and everything else around it is changed
  - The surrounding context tricks the user into clicking on the element
- UI redressing attacks are possible, even within the protections of the SOP
  - The malicious page does not get access to the framed victim page
  - Instead, it misdirects the user's input into the victim page
  - The enabler for UI redressing attacks is the ability to frame the victim page

Restricting framing to prevent UI redressing attacks

- The victim application can tell the browser to prevent framing
  - The browser will enforce the framing policy, and prevent framing by the malicious site
  - Two mechanisms to convey a framing policy to the browser

- ***X-Frame-Options*** header is the oldest mechanism
  - Supports ***SAMEORIGIN***, ***DENY*** or ***ALLOW-FROM*** with an origin
  - ***ALLOW-FROM*** not supported by all browsers

```
X-Frame-Options: DENY
```

```
X-Frame-Options: ALLOW-FROM https://restograde.com
```

A CLICKJACKI ARIO
New Tab
https://free.beer
<ifram
Do you want free beer?
No way, that's heavy stuff
Yes yes yes, real beer!
La Table du Chef
It was superb!
Meh, I've had better
Malicious page
Response contains X-Frame-Options header

X-Frame-Options HTTP header - OTHER

Usage % of all users

Global 9.42% + 83.25% = 92.67%

An HTTP header which indicates whether the browser should allow the webpage to be displayed in a frame within another webpage. Used as a defense against clickjacking attacks.

Current aligned | Usage relative | Date relative | Apply filters | Show all | ?

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Blackberry Browser
		2 - 3.6							
6 - 7		4 - 17	4 - 25	3.1 - 5	10 - 11.5	3.2 - 6.1		2.1 - 3	
8 - 10	12 - 17	18 - 64	26 - 71	5.1 - 11.1	12.1 - 56	7 - 11.4		4 - 4.4.4	7
11	18	65	72	12	57	12.1	all	67	10
		66 - 67	73 - 75	12.1 - TP		12.2			

Restricting framing to prevent UI redressing attacks

- *X-Frame-Options* header is the oldest mechanism
  - Supports *SAMEORIGIN*, *DENY* or *ALLOW-FROM* with an origin
  - *ALLOW-FROM* not supported by all browsers

```
X-Frame-Options: DENY
```

```
X-Frame-Options: ALLOW-FROM https://restograde.com
```

- *Content-Security-Policy* has a *frame-ancestors* directive
  - Supports *'self'*, *'none'* or a list of allowed origins
  - Supported by all browsers except IE11

```
Content-Security-Policy: frame-ancestors 'none'
```

```
Content-Security-Policy: frame-ancestors https://restograde.com
```

Content Security Policy Level 2 - CR

Usage % of all users

Global 77.3% + 6.45% = 83.75%

Mitigate cross-site scripting attacks by whitelisting allowed sources of script, style, and other resources. CSP 2 adds hash-source, nonce-source, and five new directives

Current aligned | Usage relative | Date relative | Apply filters | Show all | ?

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Blackberry Browser
		2-30							
		[1] 31-34	4-35		10-22				
		[2] 35	[4] 36-38		[4] 23-25				
	12-14	[3] 36-44	[5] 39	3.1-9.1	[5] 26	3.2-9.3			
6-10	[9] 15-17	[7] 45-64	40-71	10-11.1	27-56	10-11.4		2.1-4.4.4	7
11	[9] 18	[7] 65	72	12	57	12.1	all	67	10
		[7] 66-67	73-75	12.1-TP		12.2			

RESTRICTING FRAMING IN PRACTICE

- Denying all framing or restricting framing to the same origin is straightforward
  - Both ***X-Frame-Options*** and ***Content-Security-Policy*** support these features
  - Deploy either ***X-Frame-Options*** or both
    - Deploying ***X-Frame-Options*** ensures that users with legacy browsers are also protected

- Selective framing works well for a single origin
  - Use ***Content-Security-Policy*** for one set of browsers (Chrome, Safari, …)
  - Use ***X-Frame-Options*** to enable selective framing in IE11 with ***ALLOW-FROM***

- Selective framing for multiple origins requires a dynamic whitelist for IE11
  - This is messy to implement, and hard to get right
  - I would not recommend this, unless this is a critical defense

Tabnabbing is even sneakier than UI redressing

philippederyck (Philippe De Ry
Guest
https://github.com/philippederyck/allaboutbeer

Features Business Explore Marketplace Pricing Search GitHub Sign in or Sign up

Beer Information

Here, you can find some information on Belgian brews.

Learn more

A browser tab with an innocent application

TABNABBING IS EVEN SNEAKIER THAN UI REDRESSING

philippederyck (Philippe De Ry
New Tab
Guest
https://info.beer

A delicious but malicious page

TABNABBING IS EVEN SNEAKIER THAN UI REDRESSING

Sign in to GitHub · GitHub | New Tab | Guest

https://info.beer

STELLA ARTOIS

TABNABBING IS EVEN SNEAKIER THAN UI REDRESSING

Sign in to GitHub · GitHub
New Tab
Guest
https://pages.github.io/hackforbeer

Sign in to GitHub

Username or email address

Password
Forgot password?

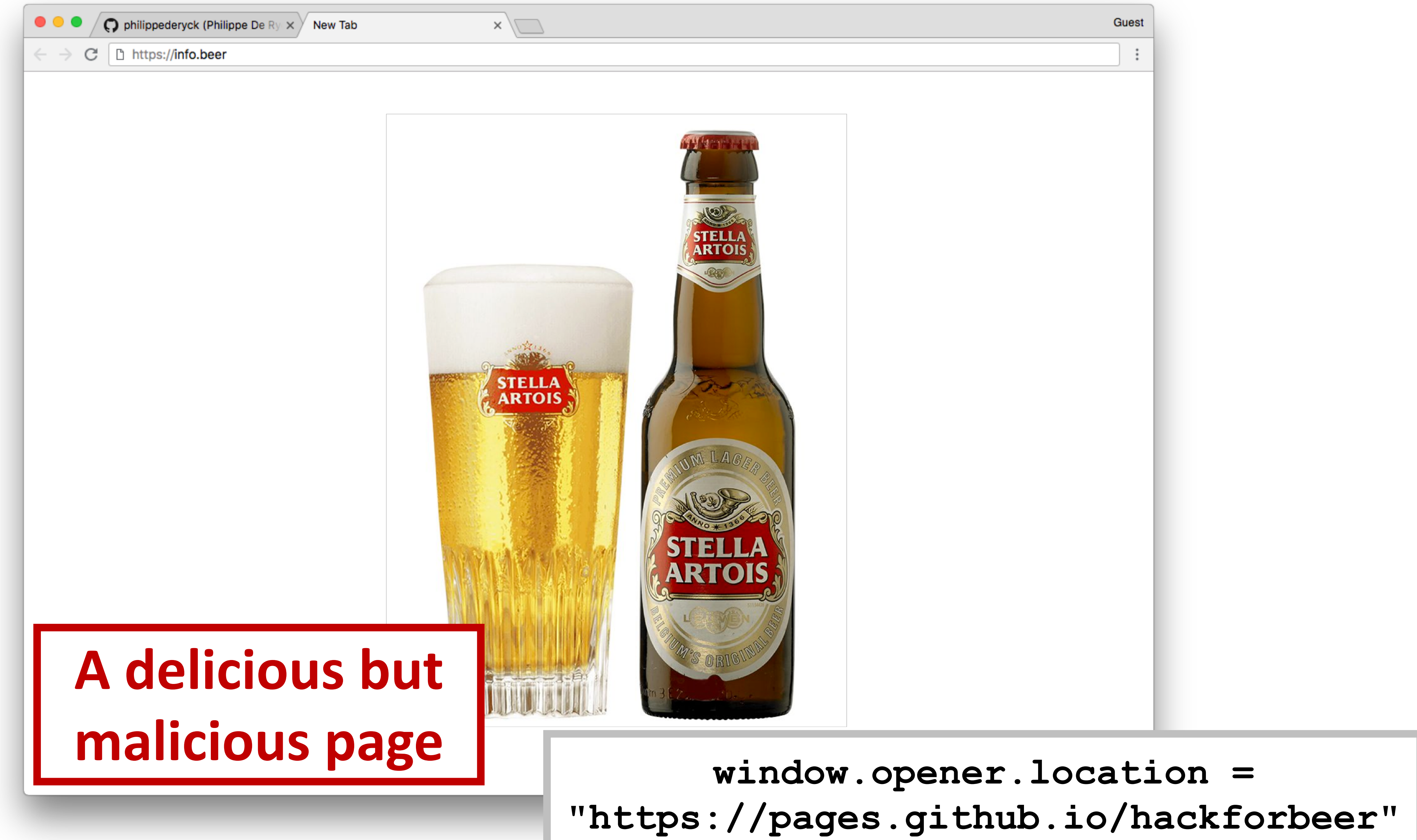
Sign in

New to GitHub? Create an account.

Security Contact GitHub

A phishing page, loaded by the malicious page

TABNABBING IS EVEN SNEAKIER THAN UI REDRESSING



A delicious but malicious page

```
window.opener.location =
"https://pages.github.io/hackforbeer"
```

Preventing tabnabbing

- A tabnabbing attack changes the content of a page in the background
  - A user returning to a page will be less vigilant than when visiting a new page
  - The malicious tab disguises itself as a legitimate site
  - When done well, it will trick the user into entering sensitive information
- These kind of tabnabbing attacks abuse the *window.opener* property
  - This property refers to the context that opened the new context
  - The SOP prevents access to DOM, but allows navigation of the opener context
- A browsing context can instruct the browser to remove the *window.opener*
  - An attribute on an anchor tag

```
<a href="https://info.beer" rel="noopener" />
```

rel=noopener - LS

Usage % of all users

Global 84.22%

Ensure new browsing contexts are opened without a useful `window.opener`

Current aligned | Usage relative | Date relative | Apply filters | Show all | ?

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Blackberry Browser
		2-51	4-48	3.1-10	10-35	3.2-10.2			
6-10	12-17	52-64	49-71	10.1-11.1	36-56	10.3-11.4		2.1-4.4.4	7
11	18	65	72	12	57	12.1	all	67	10
		66-67	73-75	12.1-TP		12.2			

SECURING BROWSING CONTEXTS

- Every browsing context is associated with an origin
  - The origin is used as a primary principal to make security decisions
- By default, cross-origin contexts are separated by the Same-Origin Policy
  - Only a very limited amount of interaction is allowed
- UI redressing attacks sidestep the SOP by tricking the user
  - The user interaction is sent to the victim page, causing unintended actions
  - The mitigation technique against UI redressing is to restrict framing
- Tabnabbing attacks abuse access to the ***window.opener*** property
  - The mitigation technique is to clear the opener when the user follows links

The security model of the web

Origins and browsing contexts

Script execution contexts

The evolution of client-side security

Modern cookie security

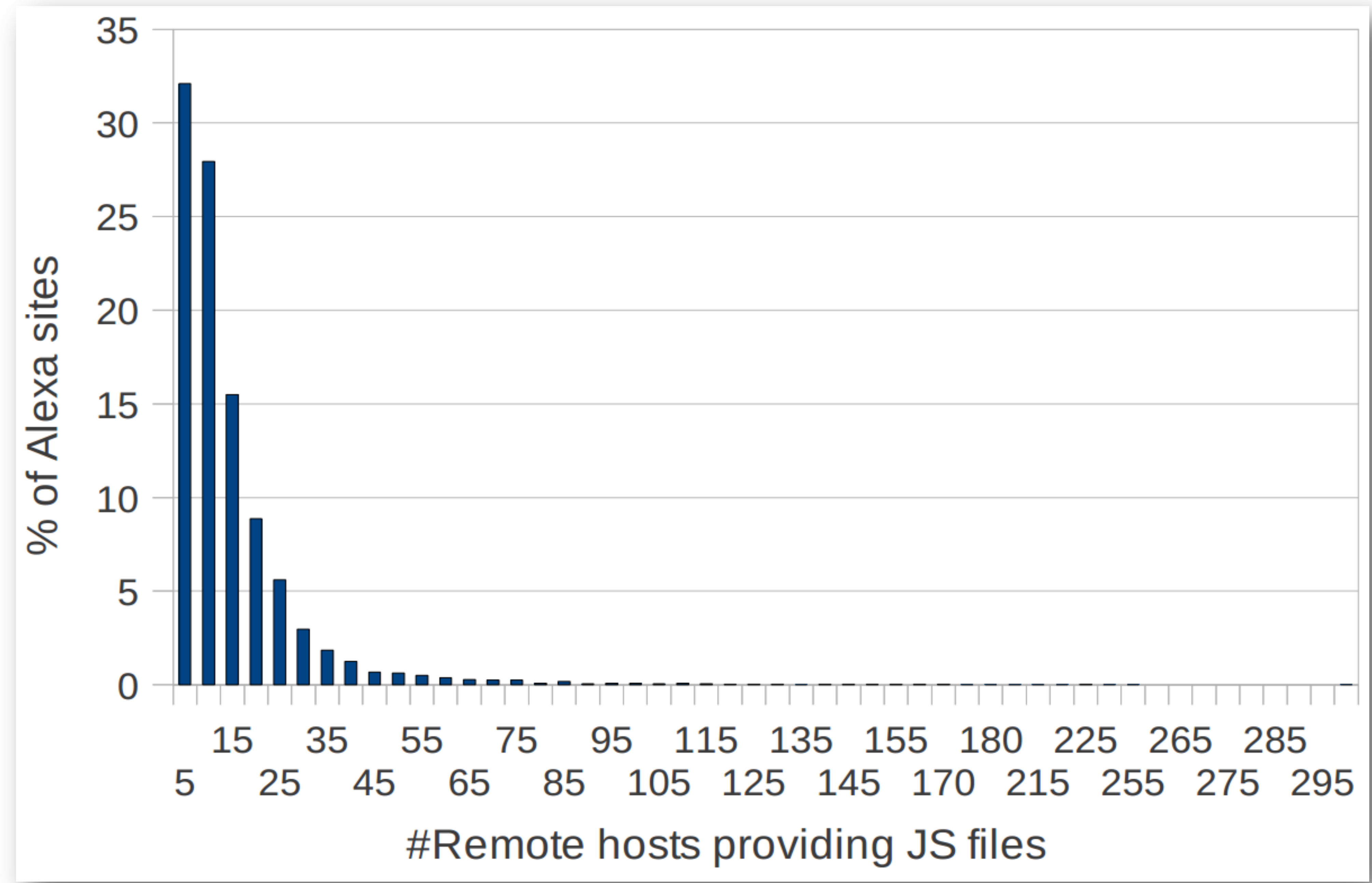
Conclusion

You Are What You Include: Large-scale Evaluation of Remote JavaScript Inclusions

Nick Nikiforakis[1], Luca Invernizzi[2], Alexandros Kapravelos[2], Steven Van Acker[1], Wouter Joosen[1], Christopher Kruegel[2], Frank Piessens[1], and Giovanni Vigna[2]

[1]IBBT-DistriNet, KU Leuven, 3001 Leuven, Belgium
firstname.lastname@cs.kuleuven.be
[2]University of California, Santa Barbara, CA, USA
{invernizzi,kapravel,chris,vigna}@cs.ucsb.edu

“88.45% of the Alexa top 10,000 web sites included at least one remote JavaScript library”


% of Alexa sites
35
30
25
20
15
10
5
0
15 35 55 75 95 115 135 155 180 225 265 285
5 25 45 65 85 105 125 145 170 215 255 275 295
#Remote hosts providing JS files

Why is third-party JavaScript code so dangerous?

The Ticketmaster breach – what happened and what to do

28 JUN 2018

> **The issue was caused by malware, spotted on 23 June 2018, that had infected a customer support system managed by Ticketmaster partner Inbenta Technologies**

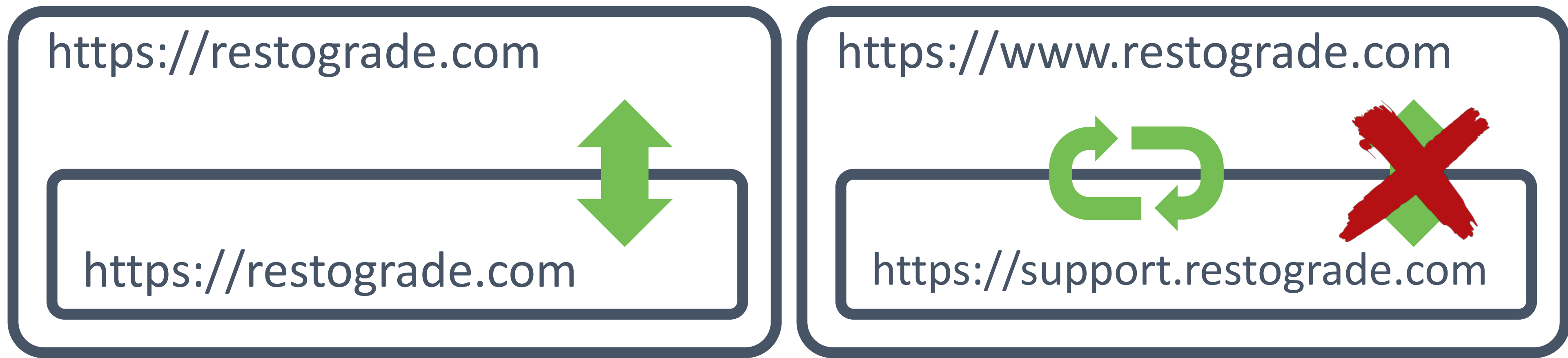
Script contexts vs browsing contexts

- Every browsing context has one script execution context
  - Every script file loaded in that context runs within the same script execution context
  - There is no isolation between code, and no separation between namespaces
- Unfortunately, the web evolved without seeing the need for code isolation
  - Advertisements are embedded directly into the page, without restrictions
  - Third-party components are embedded without isolation
  - User-provided data can even result in script execution within the main context (XSS)
- Isolating script code into a separate context is not as easy as it should be
  - Requires the creation of a separate browsing context

Leveraging browsing contexts for code isolation

- Each browsing context has its own script context
  - Cross-origin browsing contexts effectively isolate script contexts from each other
  - This enables a mechanism to support compartmentalization

- To make this practical, we need interaction between contexts
  - This allows the main context to offer or request data to the isolated
  - E.g., showing a chat dialog, requesting user data, …



Communication between browsing contexts

- Communication between two cross-origin contexts used to be hard
  - Sidechannels that enabled transferring data, but no by-design communication channel
  - ***Web Messaging*** (HTML5) offers a proper opt-in communication channel

```
frame.postMessage("Start Chat",
  "https://support.restograde.com")
```

```
window.addEventListener("message", receiveMessage, false);

function receiveMessage(event)  {
  if (event.origin !== "https://www.restograde.com")
    return;

  // Handle data
}
```

Communication between browsing contexts

- Communication between two cross-origin contexts used to be hard
  - Sidechannels that enabled transferring data, but no by-design communication channel
  - ***Web Messaging*** (HTML5) offers a proper opt-in communication channel

- You need to specify an origin when sending a message
  - The browser will check it to the origin of the context before delivering the message
  - A mismatch happens when the framed document changes without the parent realizing
  - A wildcard can be used when sending ***public*** messages anyone can see

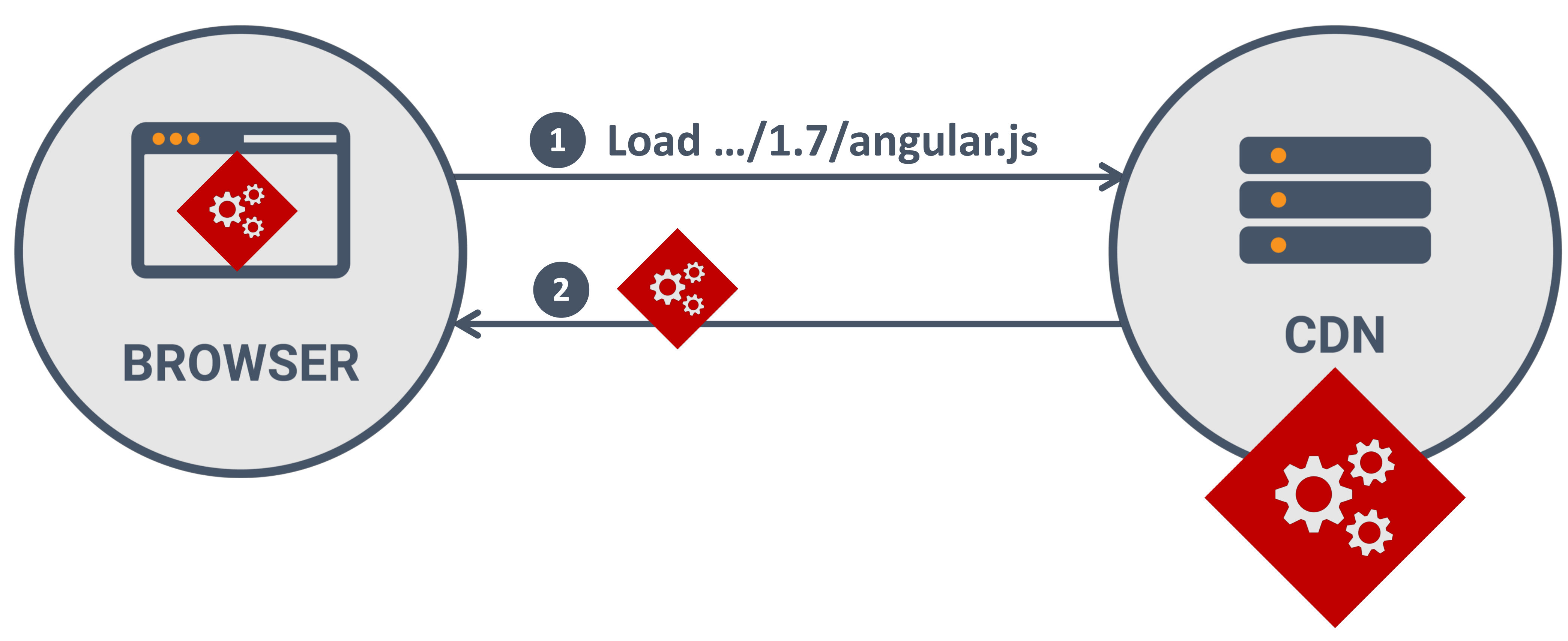
- Receiving messages requires an explicit origin check in your code
  - In theory, anyone can send a frame messages, so check if it matches your expectations
  - Handle the data securely to avoid script injection problems

LILY HAY NEWMAN SECURITY 09.11.18 03:00 AM

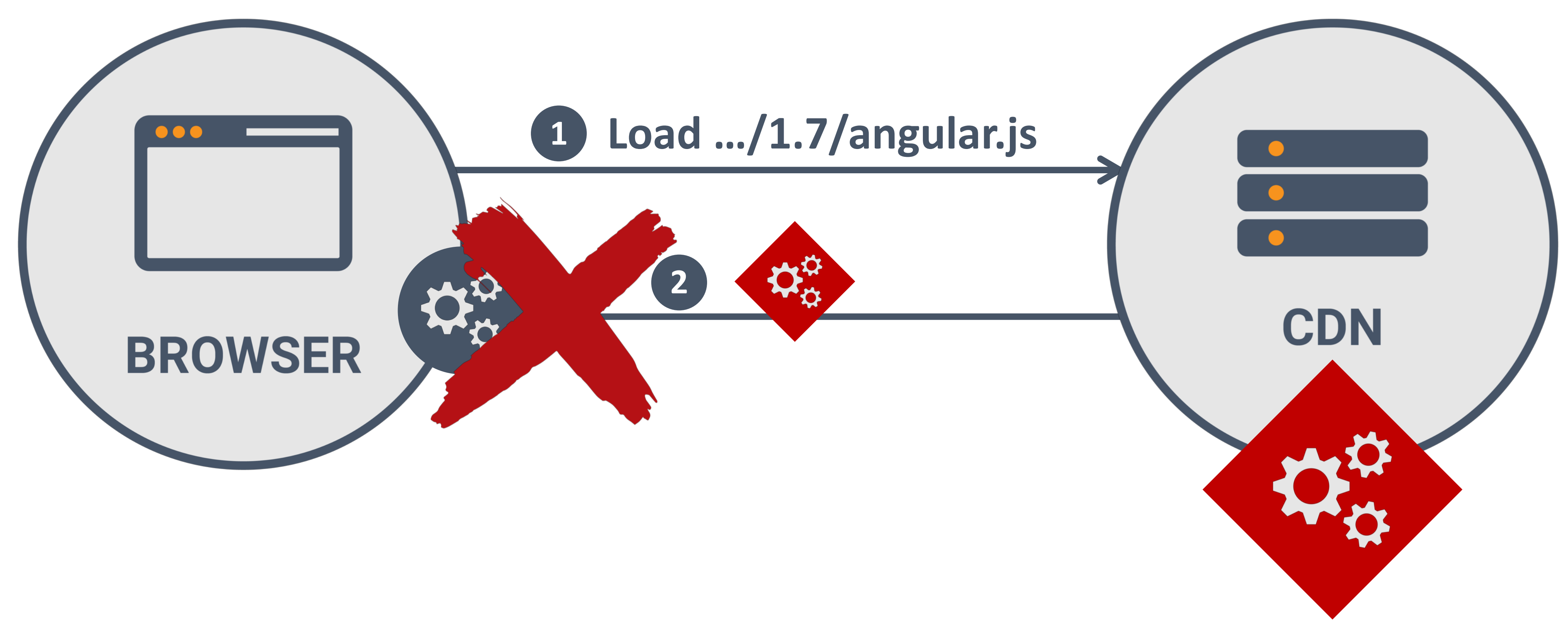
HOW HACKERS SLIPPED BY BRITISH AIRWAYS' DEFENSES

"The script is connected to the British Airways baggage claim information page; the last time it had been modified prior to the breach was December 2012. Attackers revised the component to include code, just 22 lines of it."

HOW DO YOU KNOW WHAT RESOURCE YOU LOAD FROM A CDN?




```
<script src="https://.../1.7/angular.js"></script>
```




```
<script src="https://.../1.7/angular.js" integrity="sha384-ChfqqxuZUCnJ...41rYiqJxyMiZ6OW/JmZQ5stwEULTy" crossorigin="anonymous"></script>
```

Subresource Integrity

- An ***integrity*** attribute can be added to scripts and stylesheets
  - The value of the attribute is a hash of the contents of the file
  - The hash and the contents are uniquely linked

- When loading the resource, the browser first verifies the hash
  - It calculates the hash of the file it just retrieved
  - It compares this hash to the hash in the ***integrity*** attribute
  - If they match, the file is the one the developer intended to load
  - A mismatch means that the file is not the same, and results in an error

- The properties of the hash function ensure the security of this mechanism
  - Only secure hash functions are supported by browsers

Subresource Integrity - REC

Usage % of all users

Global 85.3%

Subresource Integrity enables browsers to verify that file is delivered without unexpected manipulation.

Current aligned | Usage relative | Date relative | Apply filters | Show all | ?

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Blackberry Browser
						3.2 - 10.3			
	12 - 16	2 - 42	4 - 44	3.1 - 10.1	10 - 31	[1] 11.2			
6 - 10	17	43 - 64	45 - 71	11 - 11.1	32 - 56	11.4		2.1 - 4.4.4	7
11	18	65	72	12	57	12.1	all	67	10
		66 - 67	73 - 75	12.1 - TP		12.2			

Complete JavaScript

https://stackpath.bootstrapcdn.com/bootstrap/4

Click to copy

HTML

ap.min.js" integrity="sha384-ChfqqxuZUCnJSK3+MXmPNIyE6

Copied text to clipboard

SRI Hash Generator

Enter the URL of the resource you wish to use:

https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/js/bootstrap.min.js

Hash!

```
<script src="https://stackpath.bootstrapcdn.com/bootstrap/4.3.
1/js/bootstrap.min.js" integrity="sha384-JjSmVgyd0p3pXB1rRibZUA
```

How can I generate Integrity hashes?

Use the generator above or the following shell command:

```
openssl dgst -sha384 -binary FILENAME.js | openssl base64 -A
```

SRI DEPENDS ON THE USE OF CORS

- The ***integrity*** attribute also requires the ***cross-origin*** attribute
  - This tells to enable CORS checking on the response when loading the resource
  - SRI only works if the server also sends the proper CORS headers

- Many public hosts are configured with the CORS wildcard
  - They allow every origin to access the resource
  - As a result, everyone can apply SRI on the files served

```
<script
src="https://.../bootstrap/4.1.3/js/bootstrap.min.js"
integrity="sha384-ChfqqxuZUCnJSK3+MXm…iZ6OW/JmZQ5stwEULTy"
crossorigin="anonymous"></script>
```

The security model of the web

Origins and browsing contexts

Script execution contexts

The evolution of client-side security

Modern cookie security

Conclusion

BROWSERS ARE BECOMING APPLICATION PLATFORMS

- Modern browsers offer a platform to run independent web applications
  - Single Page Applications with offline capabilities are a modern example
  - More extreme are Chromebooks, where everything runs in a browser environment
- As an application platform, a browser supports a variety of new features
  - Applications have access to persistent data storage in the browser
  - Modern browsers enable cross-origin access to APIs
- Security is one of the cornerstones of an application platform
  - In the browser, the Same-Origin Policy is the most important security mechanism
  - Over time, additional client-side security features have been added

Storage as an origin-protected resource

- Storage areas in the browser are containerized per origin
  - ***Web Storage*** and ***IndexedDB*** are two widely available examples

- The ***Web Storage*** specification offers a ***localStorage*** area
  - A key/value-based storage mechanism available in modern browsers
  - Each origin has a separate storage container
  - Within the origin, every browsing context can access the same storage area

```
localStorage.setItem(key, value);
localStorage.getItem(key);
```

Origin-based protection only works if your origin is secure

- The moment malicious code runs within your origin, you are doomed
  - Malicious code can come from third-party dependencies
  - Malicious code can be injected by the user

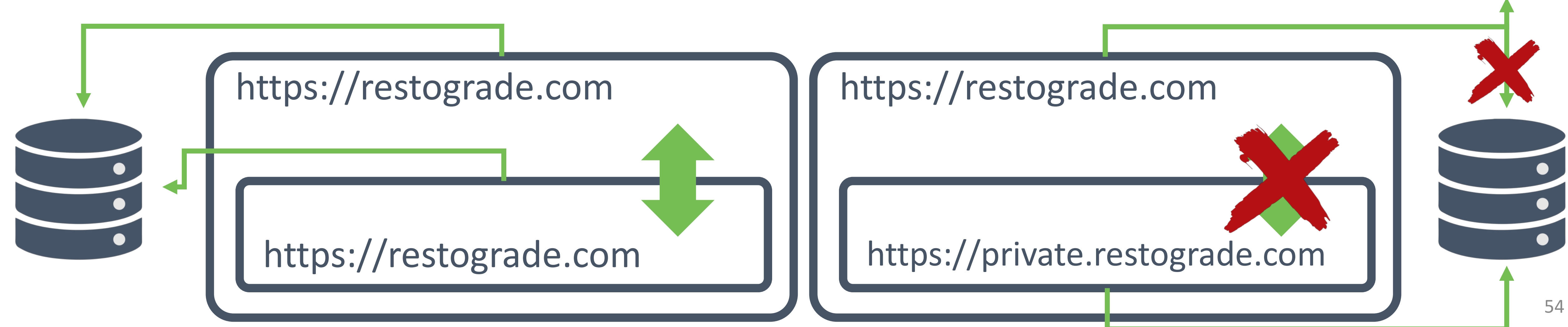
- Code running within the origin has all the same privileges as legitimate code
  - It can perform requests in the name of the application
  - It can access origin-based storage, such as ***localStorage***

- A common payload for XSS attacks is to steal ***localStorage*** data
  - Generic payload that yields interesting results
  - Especially fruitful if the application stores session data in the ***localStorage*** container

LEVERAGING THE SOP FOR DATA ISOLATION

- Sensitive data can be stored in a ***private*** origin
  - The main application asks the origin to perform operations using ***Web Messaging***
  - The private context can independently decide to fulfil this request

- Origin-based isolation helps protect the sensitive data
  - Only the messaging interface is exposed to the main application
  - A breach of the main application context does not automatically compromise all data



What if we turn the tables on trusted content?

```
<iframe src="..." sandbox> </iframe>
```

THE HTML5 *SANDBOX* ATTRIBUTE

- By default, the browser will enforce the following restrictions
  - The context is assigned a separate, unique origin
  - JavaScript code is not executed
  - Forms cannot be submitted
  - External navigation is not permitted, and popups are not allowed
  - Plugin content cannot be run (Flash, Java, ...)
  - Fullscreen capabilities are not available
  - Autoplay is not available
  - ...

```
<iframe src="..." sandbox> </iframe>
```

ENFORCING CONTENT RESTRICTIONS

- The Same-Origin Policy governs interactions within the browser
  - It can be used to isolate content, but not to restrict its capabilities
  - The content loaded within the origin can still perform arbitrary actions

- Content Security Policy gives you control about what is loaded in a context
  - It prevents the loading of potentially untrusted content
  - It does not impose behavioral restrictions on the content that is allowed to be loaded

- The ***HTML5 sandbox*** brings behavioral control over an execution context
  - The ***sandbox*** attribute can be specified on an ***iframe*** tag
  - When the ***sandbox*** attribute is present, the browser will enforce a set of restrictions

```
<iframe src="..." sandbox> </iframe>
```

SELECTIVELY LIFTING *SANDBOX* RESTRICTIONS

- The value of the attribute can be used to specify a sandboxing policy
  - ***Allow-**** keywords can be used to re-enable a set of features
  - A few possible values are:
    - allow-scripts
    - allow-same-origin
    - allow-forms
    - allow-top-navigation
    - allow-popups
    - …
- Some restrictions cannot be lifted
  - Plugin content is never allowed to run
  - Arbitrary context navigation is not possible, only top-level or popup navigation

```
<iframe src="..." sandbox="allow-scripts allow-forms"> </iframe>
```

WHICH TWO SANDBOX KEYWORDS SHOULD NEVER BE SPECIFIED TOGETHER?

Escaping the sandbox

- Code running in a sandbox is by definition untrusted
  - If it is allowed to execute, it needs to remain contained in a separate origin
  - The moment the code can reach out of the sandbox, the isolation is broken
- The ***sandbox*** attribute supports lifting restrictions on scripts and isolation
  - ***allow-scripts*** supports lifting the restriction on JavaScript execution
  - ***allow-same-origin*** supports lifting the restriction on context isolation
- When these restrictions are both lifted, the sandbox is lifted
  - Now, only the browser's default Same-Origin Policy applies
  - In a same-origin context, the code can reach out and escape the sandbox
  - In a cross-origin context, the code can impersonate legitimate application contexts

MODERN BROWSERS SUPPORT A VARIETY OF SECURITY FEATURES

- Modern browsers offer much more than origin-based isolation
  - Architectural building blocks offer control over resources being loaded
  - Server-controlled security policies offer various security features

- Architectural building blocks are inherent part of the application
  - The application is constructed to use these features on the relevant elements
    - E.g., adding a ***sandbox*** attribute to an ***iframe***

- Policy-based mechanisms are complementary to the application's architecture
  - They enable additional restrictions on the client-side application
    - E.g., forcing the use of HTTPS, controlling the loading of external resources
  - Commonly, these features can be controlled through HTTP response headers
  - The server can set a customized policy for each application
    - In some cases, policies can even be set on a per-page basis

The security model of the web

Origins and browsing contexts

Script execution contexts

The evolution of client-side security

Modern cookie security

Conclusion

WHICH OF THESE IS THE BEST PRACTICE FOR ISOLATED APPLICATIONS?

A. **Session=...; Secure; HttpOnly**

B. **__Secure-Session=...; Secure; HttpOnly**

C. **__Host-Session=...; Secure; HttpOnly**

D. **__Host-Session=...; Secure; HttpOnly; SameSite**

E. **__Host-Session=...; Secure; HttpOnly; SameSite; LockOrigin**

THE PROPERTIES OF COOKIES

- Cookies are set by the server, and stored and sent by the browser
  - Cookies are associated with the domain of the server that sets them
  - Cookies require additional security flags and prefixes to ensure secure handling

- Security considerations for using cookies securely
  - Set the ***Secure*** flag on cookies that are served over HTTPS
    - That should be all cookies, since your application should fully deployed over HTTPS
  - Set the ***HttpOnly*** flag on cookies that do not need to be accessed from JavaScript
  - Avoid the use of the the ***Domain*** attribute

- Cookie-based systems also suffer from Cross-Site Request Forgery (CSRF)
  - Contrary to popular belief, CSRF also matters for API-based backends

Session hijacking countermeasures

- Session cookies should be configured with the ***Secure*** flag
  - Prevents eavesdropping attacks on the network
  - Applies to all cookies in your application
- Session cookies should be configured with the ***HttpOnly*** flag
  - Raises the bar for XSS attacks, as well as Spectre/Meltdown attacks
  - Applies to all cookies not needed in JavaScript
- Session cookies should not be configured with the ***Domain*** attribute
  - Exposes the cookie to all subdomains, without any limitations
  - An alternative to domain-wide cookies are per-application sessions with single-sign on

Do you see a problem here?

```
Set-Cookie: session=...; Secure; HttpOnly
```

```
Cookie: session=...
```

Cookie security is very weak

- The server has no guarantees about the security of incoming cookies
  - The ***Secure*** and ***HttpOnly*** instruct the browser to limit the use of the cookie
  - This information is not reflected back to the server
  - Even within these restrictions, the cookies can be attacked

- Example attacks on cookies in the browser
  - An attacker can set malicious domain-wide cookies from a subdomain
  - An attacker can set malicious ***Secure*** cookies over an insecure HTTP channel
  - An attacker can overflow the cookie jar and set malicious ***HttpOnly*** cookies from JS

- **Cookie prefixes** further restrict how the browser handles cookies
  - Supported in every modern browser, except MS

The __Secure- cookie prefix

- The name of the cookie can be prefixed with ***__Secure-***
  - The cookie can only be set over a secure connection
  - The cookie can only be set with the ***Secure*** flag enabled

- Since the ***__Secure-*** prefix is part of the name, it is sent to the server
  - The server now knows that the cookie has been set over HTTPS
  - Whoever set the cookie was able to set up a valid HTTPS connection

- Attackers able to set such prefixed cookies can do a lot worse

SERVER

```
Set-Cookie: __Secure-session=...; Secure; HttpOnly
```

```
Cookie: __Secure-session=...
```

THE __HOST- COOKIE PREFIX

- The name of the cookie can also be prefixed with ***__Host-***
  - Everything from the ***__Secure-*** prefix applies
  - The cookie can only be set for the root path (/)
  - The cookie will only be sent to that host, never for sibling or child domains

- Since the ***__Host-*** prefix is part of the name, it is sent to the server
  - Whoever set the cookie was able to set up a valid HTTPS connection for the domain
  - In principle, that is only that application / server

- An attacker able to set a ***__Host-*** prefixed cookie has full control of the application

```
Set-Cookie: __Host-session=...; Secure; HttpOnly
```

```
Cookie: __Host-session=...
```

Recommendations for secure cookies

- Since everything runs over HTTPS, cookies can be locked down
  - Set the ***Secure*** flag on all cookies
  - Add the ***__Secure-*** prefix to all cookies
- Most cookies do not need to be accessed from JavaScript
  - Set the ***HttpOnly*** flag on those cookies
- Most cookies are set and used by one application only
  - Do not set the ***Domain*** attribute on cookies
  - Replace the ***__Secure-*** prefix with the ***__Host-*** prefix

Cross-Site Request Forgery (CSRF)

ID: 1234

2

1 Open application and login

3 Post review

4 Done!

Post review 7

8 Done

5 Request unrelated page

6 Unrelated page

SERVER

THE ESSENCE OF CSRF

- CSRF exists because the browser handles cookies very liberally
  - They are automatically attached to any outgoing request, regardless of the source
  - The browser prevents direct access to the cookies, but not their use on requests

- Many applications are unaware that any browsing context can send requests
  - The session cookies will be attached automatically by the browser
  - The web depends on this behavior, for better or for worse

- None of the cookie security measures covered so far helps here
  - The only difference between CSRF and legitimate scenarios is **intent**
  - CSRF requires additional defenses and explicit action by the developer

DEFENDING AGAINST CSRF ATTACKS

- To defend against CSRF, the application must identify forged requests
  - By design, there is no way to identify if a request came from a malicious context
  - The ***Referer*** header may help, but is not always present

- Common CSRF defenses add a secret token to legitimate requests
  - Only legitimate contexts have the token
  - Attackers can still make requests with cookies, but not with the secret token

- Recently, additional client-side security mechanisms have been introduced
  - The ***Origin*** header tells the server where a request is coming from
  - The ***SameSite*** cookie flag prevents the use of cookies on forged requests

First-party cookies as a CSRF defense

1 `Set-Cookie: session=...; SameSite`

'SameSite' cookie attribute - OTHER

Usage % of all users

Global 78.69% + 2.37% = 81.06%

Same-site cookies ("First-Party-Only" or "First-Party") allow servers to mitigate the risk of CSRF and information leakage attacks by asserting that a particular cookie should only be sent with requests initiated from the same registrable domain.

Current aligned | Usage relative | Date relative | Apply filters | Show all | ?

IE	Edge *	Firefox	Chrome	Safari	Opera	iOS Safari *	Opera Mini *	Android Browser *	Blackberry Browser
	12 - 15	2 - 59	4 - 50		10 - 38				
6 - 10	[1] 16 - 17	60 - 64	51 - 71	3.1 - 11.1	39 - 56	3.2 - 11.4		2.1 - 4.4.4	7
[1] [2] 11	18	65	72	12	57	12.1	all	67	10
		66 - 67	73 - 75	12.1 - TP		12.2			

First-party cookies

- The ***SameSite*** attribute actually supports a ***strict*** and ***lax*** mode
  - In ***strict*** mode, the browser will never attach the cookie to a cross-site request
    - This is determined based on the registered domain, not the origin
  - In ***lax*** mode, the cookie will be present on safe top-level navigations
    - e.g. a GET request that results in a navigation of the context

- The default setting for the ***SameSite*** attribute is ***strict*** mode
  - This is the mode you get when you simply add ***SameSite*** to the cookie
  - This will stop all CSRF attacks

- Adding the ***SameSite*** attribute in lax mode will stop most CSRF attacks
  - Unless the attack can be launched with a GET request

The security model of the web

K

Origins and browsing contexts

Script execution contexts

The evolution of client-side security

Modern cookie security

Conclusion

RECAP

- The origin is a primary principal for security decisions within the browser
  - Every browsing context has an origin
  - The Same-Origin Policy (SOP) isolates contexts based on the origin
  - Resources within the browser typically belong to an origin
- Each browsing context has exactly one script context
  - All scripts are executed within the same context
  - There is no concept of origin-isolation for scripts within the browser (unfortunately)
- Cookies require attributes and prefixes to guarantee secure handling
  - Attributes alone are not strict enough
  - Almost all modern browsers support the new prefixes, so start using them

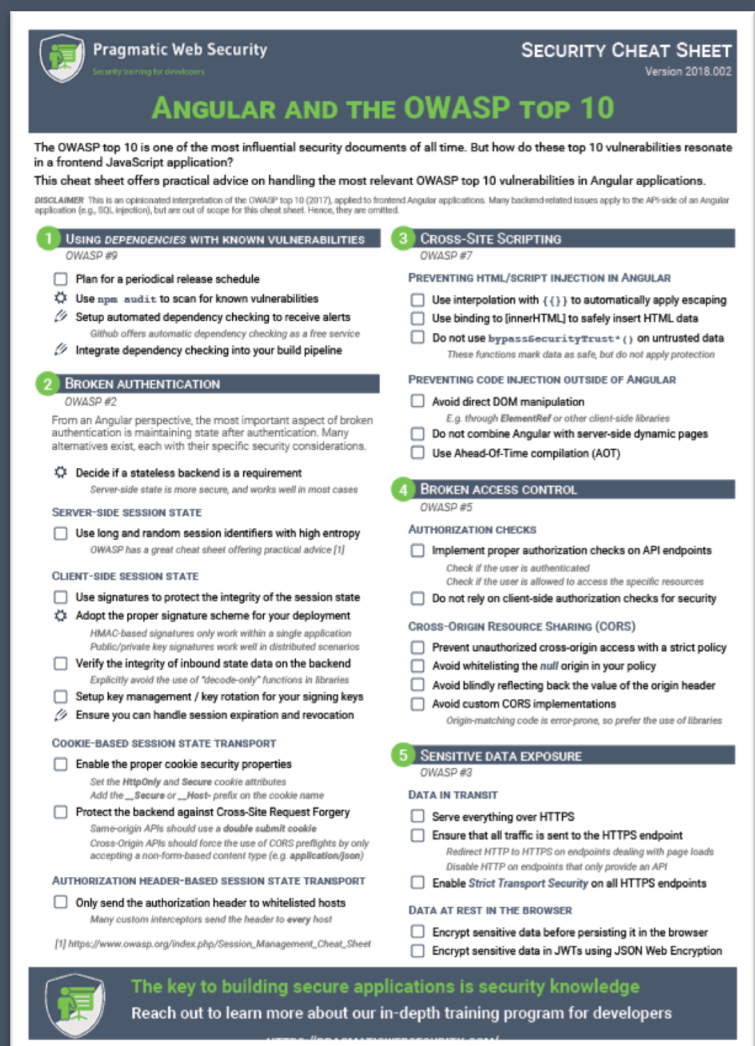
BEST PRACTICES

- Compartmentalize your applications
  - The SOP is present in all browsers, leverage it!
  - Different applications should be deployed in different origins
  - Sensitive components within an application can also be isolated in a separate origin
  - Communication between contexts is possible through ***Web Messaging***

- Be careful with including third-party scripts
  - Each script has full access to your origin
  - When embedding less-trusted components, leverage compartmentalization for security

- Protect your cookies using the following best practices
  - The ***__Secure-*** prefix and ***Secure*** flag should be present on all cookies
  - Cookies should have the ***HttpOnly*** flag, unless they need to be accessed from JavaScript
  - Deploy the ***SameSite*** cookie flag as a complementary CSRF defense

Free security cheat sheets for modern applications



Pragmatic Web Security
Security training for developers

Security Cheat Sheet
Version 2018.002

Angular and the OWASP top 10

The OWASP top 10 is one of the most influential security documents of all time. But how do these top 10 vulnerabilities resonate in a frontend JavaScript application?

This cheat sheet offers practical advice on handling the most relevant OWASP top 10 vulnerabilities in Angular applications.

DISCLAIMER This is an opinionated interpretation of the OWASP top 10 (2017), applied to frontend Angular applications. Many backend-related issues apply to the API-side of an Angular application (e.g., SQL injection), but are out of scope for this cheat sheet. Hence, they are omitted.

1 Using dependencies with known vulnerabilities
OWASP #9

- Plan for a periodical release schedule
- Use `npm audit` to scan for known vulnerabilities
- Setup automated dependency checking to receive alerts
  Github offers automatic dependency checking as a free service
- Integrate dependency checking into your build pipeline

2 Broken authentication
OWASP #2

From an Angular perspective, the most important aspect of broken authentication is maintaining state after authentication. Many alternatives exist, each with their specific security considerations.

- Decide if a stateless backend is a requirement
  Server-side state is more secure, and works well in most cases

Server-side session state
- Use long and random session identifiers with high entropy
  OWASP has a great cheat sheet offering practical advice [1]

Client-side session state
- Use signatures to protect the integrity of the session state
- Adopt the proper signature scheme for your deployment
  HMAC-based signatures only work within a single application
  Public/private key signatures work well in distributed scenarios
- Verify the integrity of inbound state data on the backend
  Explicitly avoid the use of "decode-only" functions in libraries
- Setup key management / key rotation for your signing keys
- Ensure you can handle session expiration and revocation

Cookie-based session state transport
- Enable the proper cookie security properties
  *Set the **HttpOnly** and **Secure** cookie attributes*
  *Add the **__Secure** or **__Host-** prefix on the cookie name*
- Protect the backend against Cross-Site Request Forgery
  *Same-origin APIs should use a **double submit cookie***
  *Cross-Origin APIs should force the use of CORS preflights by only accepting a non-form-based content type (e.g. **application/json**)*

Authorization header-based session state transport
- Only send the authorization header to whitelisted hosts
  *Many custom interceptors send the header to **every** host*

[1] https://www.owasp.org/index.php/Session_Management_Cheat_Sheet

3 Cross-Site Scripting
OWASP #7

Preventing HTML/script injection in Angular
- Use interpolation with `{{}}` to automatically apply escaping
- Use binding to [innerHTML] to safely insert HTML data
- Do not use `bypassSecurityTrust*()` on untrusted data
  These functions mark data as safe, but do not apply protection

Preventing code injection outside of Angular
- Avoid direct DOM manipulation
  *E.g. through **ElementRef** or other client-side libraries*
- Do not combine Angular with server-side dynamic pages
- Use Ahead-Of-Time compilation (AOT)

4 Broken access control
OWASP #5

Authorization checks
- Implement proper authorization checks on API endpoints
  Check if the user is authenticated
  Check if the user is allowed to access the specific resources
- Do not rely on client-side authorization checks for security

Cross-Origin Resource Sharing (CORS)
- Prevent unauthorized cross-origin access with a strict policy
- Avoid whitelisting the *null* origin in your policy
- Avoid blindly reflecting back the value of the origin header
- Avoid custom CORS implementations
  Origin-matching code is error-prone, so prefer the use of libraries

5 Sensitive data exposure
OWASP #3

Data in transit
- Serve everything over HTTPS
- Ensure that all traffic is sent to the HTTPS endpoint
  Redirect HTTP to HTTPS on endpoints dealing with page loads
  Disable HTTP on endpoints that only provide an API
- Enable *Strict Transport Security* on all HTTPS endpoints

Data at rest in the browser
- Encrypt sensitive data before persisting it in the browser
- Encrypt sensitive data in JWTs using JSON Web Encryption

The key to building secure applications is security knowledge
Reach out to learn more about our in-depth training program for developers

Pragmatic Web Security
Security training for developers

Security Cheat Sheet
Version 2018.001

JSON Web Tokens (JWT)

JSON Web Tokens (JWTs) have become extremely popular. JWTs seem deceivingly simple. However, to ensure their security properties, they depend on complex and often misunderstood concepts. This cheat sheet focuses on the underlying concepts. The cheat sheet covers essential knowledge for every developer producing or consuming JWTs.

Introduction

A JWT is a convenient way to represent claims securely. A claim is nothing more than a key/value pair. One common use case is a set of claims representing the user's identity. The claims are the payload of a JWT. Two other parts are the header and the signature.

- JWTs should always use the appropriate signature scheme
- If a JWT contains sensitive data, it should be encrypted
- JWTs require proper cryptographic key management
- Using JWTs for sessions introduces certain risks

JWT integrity verification

Claims in a JWT are often used for security-sensitive operations. Preventing tampering with previously generated claims is essential. The issuer of a JWT signs the token, allowing the receiver to verify its integrity. These signatures are crucial for security.

Symmetric signatures

Symmetric signatures use an HMAC function. They are easy to setup, but rely on the same secret for generating and verifying signatures. Symmetric signatures only work well within a single application.

Asymmetric signatures

Asymmetric signatures rely on a public/private key pair. The private key is used for signing, and is kept secret. The public key is used for verification, and can be widely known. Asymmetric signatures are ideal for distributed scenarios

Best practices
- Always verify the signature of JWT tokens
- Avoid library functions that do not verify signatures
  Example: The decode function of the auth0 Java JWT library
- Check that the secret of symmetric signatures is not shared
- A distributed setup should only use asymmetric signatures

JWT Encryption is a complex topic. It is out of scope for this cheat sheet.

Validating JWTs

Apart from the signature, a JWT contains other security properties. These properties help enforce a lifetime on a JWT. They also identify the issuer and the intended target audience. The receiver of a JWT should always check these properties before using any of the claims.

- Check the `exp` claim to ensure the JWT is not expired
- Check the `nbf` claim to ensure the JWT can already be used
- Check the `iss` claim against your list of trusted issuers
- Check the `aud` claim to see if the JWT is meant for you

Some libraries offer support for checking these properties. Verify which properties are covered, and complement these checks with your own.

Cryptographic key management

The use of keys for signatures and encryption requires careful management. Keys should be stored in a secure location. Keys also need to be rotated frequently. As a result, multiple keys can be in use simultaneously. The application has to foresee a way to manage the JWT key material.

- Store key material in a dedicated key vault service
  Keys should be fetched dynamically, instead of being hardcoded
- Use the `kid` claim in the header to identify a specific key
  Keys should be fetched dynamically, instead of being hardcoded
- Public keys can be embedded in the header of a JWT
  The `jwk` claim can hold a JSON Web Key-formatted public key
  The `x5c` claim can hold a public key and X509-certificate
- Validate an embedded public key against a whitelist
  Failure to whitelist will cause an attacker's JWT to be accepted
- The header can also contain a URL pointing to public keys
  The `jku` claim can point to a file containing JSON Web Keys
  The `x5u` claim can point to a certificate containing a public key
- Validate a key URL against a whitelist of URLs / domains
  Failure to whitelist will cause an attacker's JWT to be accepted

Using JWTs for authorization state

Many modern applications use JWTs to push authorization state to the client. Such an architecture benefits from a stateless backend, often at the cost of security. These JWTs are typically bearer tokens, which can be used or abused by whoever obtains them.

- It is hard to revoke a self-contained JWT before it expires
- JWTs with authorization data should have a short lifetime
- Combine short-lived JWTs with a long-lived session

The key to building secure applications is security knowledge
Reach out to learn more about our in-depth training program for developers

https://cheatsheets.pragmaticwebsecurity.com/

Web Security Essentials

2-day training course

Modern-day best practices

Hands-on labs on a custom training application

April 25th – 26th, 2019

Leuven, Belgium

https://essentials.pragmaticwebsecurity.com

Pragmatic Web Security

Security training for developers

philippe@pragmaticwebsecurity.com