Web Security - Same Origin Policy (SOP)

Stefano Calzavara

Università Ca' Foscari Venezia

Same Origin Policy (SOP)

The Same Origin Policy (SOP) is the baseline defense mechanism of web browsers, which isolates data controlled by `good.com` from read / write accesses by scripts served from `evil.com`.

Key Questions

In this lecture we discuss:

1. What is an isolation domain for SOP?
2. What should be isolated by SOP?
3. How to relax SOP when required for legitimate functionality?
4. What are the main security limitations of SOP?

Web Origins

SOP defines its own isolation domains in terms of origins.

Origin

An origin is a triple including a scheme, a hostname and a port. When the port is omitted, the default port of the scheme is implicitly assumed.

Example

Consider `http://www.flicker.com/galleries`, then:

- `http://www.flicker.com/favorites` has the same origin
- `http://www.flicker.com:80/galleries` has the same origin
- `https://www.flicker.com/galleries` has a different origin
- `http://my.flicker.com/galleries` has a different origin

SOP Restrictions

At a high level, SOP can be summarized as follows: data owned by origin o_1 must be isolated from read / write accesses by any origin $o_2 \neq o_1$.

Example

A script running on a page served by `http://www.foo.com` cannot:

- access cookies of `http://www.bar.com`
- access the DOM of `http://sub.foo.com`
- access the DOM of `https://www.foo.com`

No formal definition!

It is just too complicated to specify all places where SOP must apply, so different browsers might implement SOP slightly differently [3].

Cookies and SOP

Cookies implement a relaxed variant of SOP:

- ✓ cookies set by `good.com` cannot be accessed by `evil.com`
- ✓ `evil.com` is not allowed to set cookies for `good.com`
- ✗ cookies do not provide isolation by scheme and port, i.e., there is no isolation between HTTP and HTTPS
- ✗ cookies can be shared with sibling domains and their children, by means of the `Domain` attribute
- ✗ cookies can be set by sibling domains and their children, by means of the `Domain` attribute

The question whether SOP applies to cookies is largely philosophical!

Web Storage and SOP

Web storage is a simple JavaScript API to store origin-scoped data, introduced in HTML5. It uses the traditional key-value view of cookies.

Example

```
localStorage.setItem("lang", "IT");
v = localStorage.getItem("lang");
```

Web storage is one of the simplest examples where SOP strictly applies: read / write accesses to web storage are separated per origin.

Cookies vs Web Storage

We can use web storage for session management as well, using custom JavaScript to fetch session information and share it with the server.

Cookies	**Web Storage**
- Relaxed SOP (limited security)	+ Traditional SOP (more secure)
- Sent automatically (CSRF)	+ Sent on demand (no CSRF)
+ HttpOnly: shielded from JS	- Always accessible to JS (risky)
+ Sessions are easy to implement	- Sessions require custom JS logic

Both mechanisms have pros and cons, but the simplicity of cookies is preferred by most web developers.

Cookies + WebStorage = Better Sessions

Combine cookies and web storage to get the best of two worlds [1]:

1. set an HttpOnly cookie $c = s$, where s is a session identifier
2. set an item $k = hash(s)$ in the web storage
3. require authenticated requests to include a parameter p, populated by reading the value of k from the web storage
4. authenticate only the requests attaching both a cookie $c = s$ and a parameter $p = hash(s)$

Mark c as Secure on HTTPS applications to prevent network attacks!

Cookies + WebStorage = Better Sessions

The proposed scheme has many advantages:

- session hijacking: security attributes ensure cookie confidentiality and the cookie value cannot be reconstructed from the hash set in the web storage, hence no session hijacking is possible
- session fixation: although cookies provide weak integrity guarantees against network attackers, this is compensated by the web storage, which relies on traditional SOP and offers isolation by scheme
- CSRF: since requests are not authenticated by cookies alone, CSRF is also prevented by construction

The scheme requires the implementation of custom JavaScript logic, which complicates its deployment.

SOP and Content Inclusion

Since the Web is designed to be interconnected, SOP puts very little restrictions on content inclusion. For example, `foo.com` can normally load images and stylesheets from `bar.com`.

Script Inclusion

If `https://foo.com` loads a script from `https://bar.com` using a `<script>` tag, the script will run in the origin `https://foo.com` and acquire its rights to the eyes of SOP. Beware of remote script inclusion!

Example

Assume the attacker was able to take control of the `jquery.com` domain. What could go wrong? How would you safeguard against this threat?

Protecting Content Inclusion: Sub-Resource Integrity

It is possible to enforce integrity checks on included scripts by using the Sub-Resource Integrity mechanism (SRI).

```
<script src="https://code.jquery.com/jquery-3.4.1.min.js"
        integrity="sha384-+/M6kredJcxdsqkczBUjML...">
```

Key idea: browsers fetching the remote script compare the hash in the `integrity` tag with the hash computed from the downloaded script; if the two hashes do not match, the script is not loaded.

Alert!

SRI is far from straightforward to deploy in practice! Do you see why?

XMLHttpRequest (XHR)

XMLHttpRequest is a powerful JavaScript API used to send HTTP requests and process the corresponding HTTP responses.

Example

```
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
       user = JSON.parse(xhttp.responseText);
    }
};
xhttp.open("GET", "https://bar.com/users.php?uid=123");
xhttp.send();
```

XHR and SOP: Request Methods

XHR can be used to send HTTP requests with different methods:

- no restriction on GET, POST and HEAD requests: these requests can be sent from any origin to any origin
- reason: GET and HEAD are safe and idempotent, hence expected to be harmless, while cross-origin POST requests have always been allowed in form submissions and must be accepted
- all the other methods, including PUT and DELETE, are restricted to same-origin requests for security reasons. For example, `foo.com` cannot send a PUT request to `bar.com`

We will discuss how to securely relax this default behavior.

XHR and SOP: Request Headers

XHR also allows the attachment of custom HTTP headers to requests.

Example

```
var xhttp = new XMLHttpRequest();
xhttp.open("GET", "https://bar.com/users.php?uid=123");
xhttp.setRequestHeader("X-Test-Header", "HighSecurity");
xhttp.send();
```

However, SOP restricts this practice to same-origin requests. The reason is that custom headers might have arbitrarily complex semantics, which is specific to individual web applications, hence they are security-sensitive.

XHR and SOP: Cross-Origin Reads

Although SOP allows cross-origin XHR requests with specific methods, it still restricts access to the corresponding HTTP responses. For example, `foo.com` can send a GET request to `bar.com` but can't read its response.

Example

If this was not the case, an attacker at `https://www.evil.com` could sniff the content of the victim's mailbox by sending (authenticated) XHR requests to `https://mail.google.com`.

Alert!

Preventing cross-origin read accesses is a major restriction!

1. Legacy relaxation: JSON with Padding (JSONP)
2. Modern relaxation: Cross Origin Resource Sharing (CORS)

JSON with Padding (JSONP)

Assume `foo.com` wants to access a user database at `bar.com`:

1. `foo.com` implements a callback to process the upcoming response, say a `parseUser` function taking a single JSON parameter
2. `foo.com` loads a script from the following URL: `https://bar.com/users.php?uid=123&cb=parseUser`
3. `bar.com` accesses its database and replies with the following script: `parseUser({"name":"Bob", "sex":"M", "uid":"123"})`
4. since the response is valid JavaScript, the callback is executed with the right input at `foo.com`, thus allowing the embedding page to parse the content before use

This clever pattern is known as JSON with Padding (JSONP).

Insecurity of JSONP

Although useful, JSONP is obviously insecure for at least two reasons:

1. script injection: the caller must place a lot of trust in the callee, who might ignore the callback and just reply with an arbitrary script
2. information leakage: the callee must implement protection against CSRF, unless the content of the response is public data (any page can include the script from the callee!)

Luckily, there is no reason to use JSONP on the modern Web, because CORS provides a much better alternative!

Cross Origin Resource Sharing (CORS)

CORS provides a disciplined way to relax the restrictions of SOP.

Intuition

The key idea of CORS can be summarized as follows:

1. `foo.com` asks for permission to read cross-origin data
2. `bar.com` grants or denies such permission
3. the browser enforces the authorization decision at `foo.com`

However, the protocol details are a bit tricky...

CORS: Simple Requests

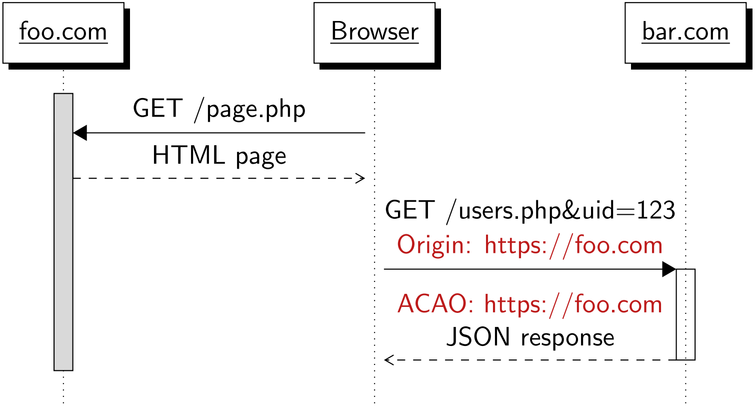
A simple request is essentially a GET, POST or HEAD request without custom HTTP headers attached. Does that ring a bell?

Relevant CORS Headers

- `Origin`: request header containing the origin which is asking for cross-origin read permission
- `Access-Control-Allow-Origin`: response header containing the origin to which such permission is granted ($*$ for any origin)

Access is granted iff the content of the `Access-Control-Allow-Origin` header matches the content of the `Origin` header or is set to `*`.

CORS: Simple Requests



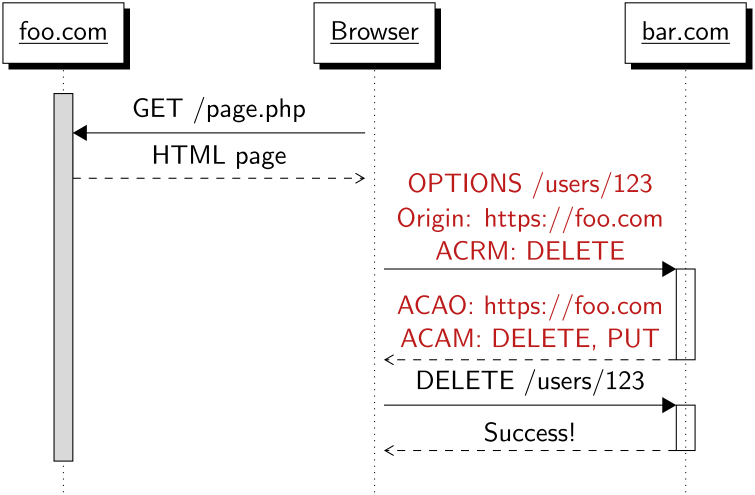
CORS: Non-Simple Requests

HTTP requests which are not simple use a more complicated protocol, involving a preliminary approval step based on a preflight request.

Relevant CORS Headers

- `Access-Control-Request-Method`: preflight request header containing the method of the non-simple request
- `Access-Control-Request-Headers`: preflight request header containing the list of the custom headers of the non-simple request
- `Access-Control-Allow-Methods`: preflight response header containing a list of allowed methods
- `Access-Control-Allow-Headers`: preflight response header containing a list of allowed custom headers
- `Access-Control-Max-Age`: preflight response header for caching

CORS: Non-Simple Requests



CORS: Why Preflight?

Non-simple requests are delicate from a security perspective:

- methods like PUT and DELETE are intended to have a significant, security-sensitive side-effect at the server (unsafe methods)
- custom headers can have an arbitrarily complex semantics for web applications, hence they pose a potential security threat

SOP normally prevents this form of cross-origin requests, so just allowing them directly might introduce new vulnerabilities. The preflight requests of CORS allow relaxation of SOP without breaking the Web.

CORS: Credentialed Requests

Browsers do not attach credentials (cookies) to cross-origin XHRs, unless the `withCredentials` property of the XHR object is activated.

Example

```
var xhttp = new XMLHttpRequest();
xhttp.open("GET", "https://bar.com/users.php?uid=123");
xhttp.withCredentials = true;
xhttp.send();
```

This way, XHR requests are unauthenticated by default: this is useful to prevent unintended side-effects at the callee (least privilege principle).

CORS: Credentialed Requests

Credentialed requests must be explicitly allowed by the callee: this is very useful to prevent unintended information leaks to malicious callers.

Simple Requests

If the response has no `Access-Control-Allow-Credentials: true` header, the response body is left inaccessible.

Non-Simple Requests

If the preflight response has no `Access-Control-Allow-Credentials: true` header, the non-simple request is not sent.

When responding to a credentialed request, the server cannot use the wildcard * in the `Access-Control-Allow-Origin` header! Conscious decisions are forced on sensitive actions enabled by credentialed requests.

CORS: Credentialed Requests

Summary of the discussion:

withCredentials	**ACAC**	**Outcome**	**Notes**
✗	✗	✓	No cookie attached
✗	✓	✓	No cookie attached
✓	✓	✓	Cookies are attached and response body is accessible
✓	✗	✗	Response body is inaccessible / non-simple request is not sent after preflight

Security of CORS

CORS is a standardized, more secure alternative to JSONP:

1. no script injection: though the callee can still respond with arbitrary content, the caller can process the response (not a script) before actually using it. This is normally done in the XHR callback
2. no information leakage: only credentialed requests might disclose confidential information and the callee has control over them, thanks to the `Access-Control-Allow-Origin` header

This is a huge improvement over JSONP, but please note one can still write insecure applications if uncaring enough! Can you see how?

SOP Pitfalls: DNS Rebinding

The security of SOP relies on the security of DNS. If DNS cannot be trusted, then SOP can be bypassed through DNS rebinding [2]:

1. the attacker registers `evil.com` and gets control of its DNS records
2. once the victim accesses `evil.com`, the attacker provides a DNS record with a short TTL pointing to the attacker's server
3. the attacker delivers a malicious script, which waits until the TTL has expired and sends an XHR request to `evil.com/target`
4. since the TTL has expired, the attacker now provides a different DNS record pointing to, e.g., 127.0.0.1 (or some other IP address which is inaccessible from the Internet)
5. from the browser's viewpoint this is same-origin communication, e.g., the malicious script can read the response body and leak it!

SOP Pitfalls: DNS Rebinding

Notice that DNS rebinding could apparently be used also to access responses from `mail.google.com` at `evil.com`, e.g., abusing XHR to access the victim's mailbox on the attacker's website. It would suffice to make `evil.com` resolve to the IP address of `mail.google.com`.

Luckily, this variant of the attack is significantly mitigated in practice:

- the request to `mail.google.com` includes cookies for `evil.com`, which mitigates the dangers of data exfiltration
- the `Host` header of the request is set to `evil.com`, which allows `mail.google.com` to implement protection

More technical details here if you want to read more!

References

[1] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi.
Dr cookie and mr token - web session implementations and how to live with them.
In *ITASEC*, 2018.

[2] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh.
Protecting browsers from dns rebinding attacks.
In *ACM CCS*, pages 421–431, 2007.

[3] Jörg Schwenk, Marcus Niemietz, and Christian Mainka.
Same-origin policy: Evaluation in modern browsers.
In *USENIX Security*, pages 713–727. USENIX Association, 2017.