Security II - Same Origin Policy

Stefano Calzavara

Università Ca' Foscari Venezia

February 13, 2020

Same Origin Policy (SOP)

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

Key Questions

1. What is an isolation domain for SOP?
2. What should be isolated by SOP?
3. What are the limitations of SOP?

Web Origins

SOP defines 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 different origin
- `http://my.flicker.com/galleries` has 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 attempts 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 `https://www.bar.com`
- access the DOM of `https://www.bar.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 often implement SOP differently!

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
- ✗ cookies can be shared across sibling domains (different origins)
- ✗ cookies do not provide integrity for sibling domains and their children

The question whether SOP applies to cookies is essentially philosophical!

Web Storage and SOP

Web storage is a simple JS API to store origin-scoped data, introduced in HTML 5. It proposes 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

Wait a minute! Why can't we use web storage for session management?

Cookies	Web Storage
Relaxed SOP	Traditional SOP
Sent automatically	Sent on demand
HttpOnly = shielded from JS	Always accessible to JS
Sessions are easy to implement	Sessions require 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 entry $k = h(s)$ in the web storage, where h is a hash function
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 $p = h(s)$

Bonus: mark c as Secure on HTTPS applications.

Cookies + WebStorage = Better Sessions

The proposed scheme has many advantages:

- the use of security attributes ensures cookie confidentiality and the cookie value cannot be reconstructed from its hash, hence session hijacking is not possible
- since requests are not authenticated by cookies alone, CSRF is also prevented by construction
- although cookies provide weak integrity guarantees against network attackers, this is compensated by the use of the web storage

The scheme still requires the implementation of custom JS logic.

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 for 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 content by using the recently introduced Sub-Resource Integrity mechanism.

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

Browsers fetching the resource compare the hash in the `integrity` tag with the hash computed from the resource: if the hashes do not match, the resource is discarded.

Protecting Content Inclusion: Mixed Content

The Mixed Content policy implements restrictions on the inclusion of HTTP resources in HTTPS pages:

- passive content like images, audios and videos might be allowed at the discretion of browser vendors
- everything which does not fall in this list of exceptions is considered active content and must be blocked

All modern browsers implement the Mixed Content policy in some form, which puts a light flavour of SOP into content inclusion.

XMLHttpRequest (XHR)

XMLHttpRequest is a powerful JS 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: such requests can be sent to any origin
- GET and HEAD are safe and idempotent, while cross-origin POST requests are already allowed by form submissions
- all the other methods, including PUT and DELETE, are restricted to same-origin requests for security reasons

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.

XHR and SOP: Cross-Origin Reads

Though SOP does not prevent a cross-origin XHR request, it restricts access to the corresponding HTTP response for security reasons.

Example

If this was not the case, an attacker at `https://www.evil.com` could try to get read access to 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 solution: JSON with Padding (JSONP)
2. Modern solution: 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` responds with the following script:
   `parseUser({"name": "Bob", "sex": "M", "ID": "123"})`
4. the callback is invoked with the right content at `foo.com`

This “fancy” pattern is known as JSON with Padding (JSONP).

Insecurity of JSONP

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 reply with an arbitrary script
2. information leakage: the callee must implement protection against CSRF, unless the content of the response is public data

Luckily, there is no reason to use JSONP on the modern Web!

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 the requested permission
3. the browser enforces the final choice of `bar.com`

However, the details are a bit tricky...

CORS: Simple Requests

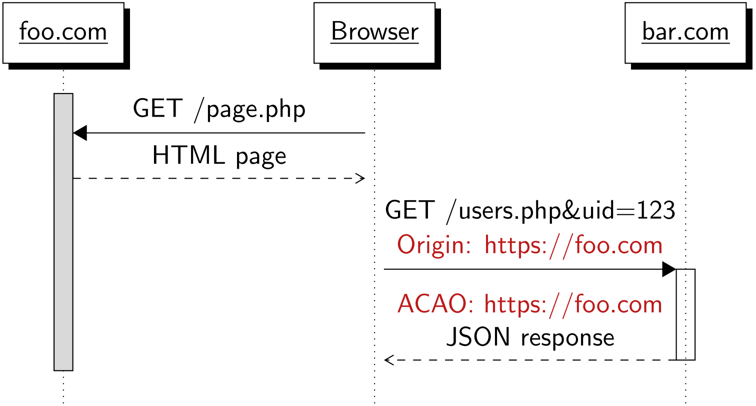
A simple request is essentially a GET, POST or HEAD request without custom HTTP headers attached.

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.

CORS: Simple Requests



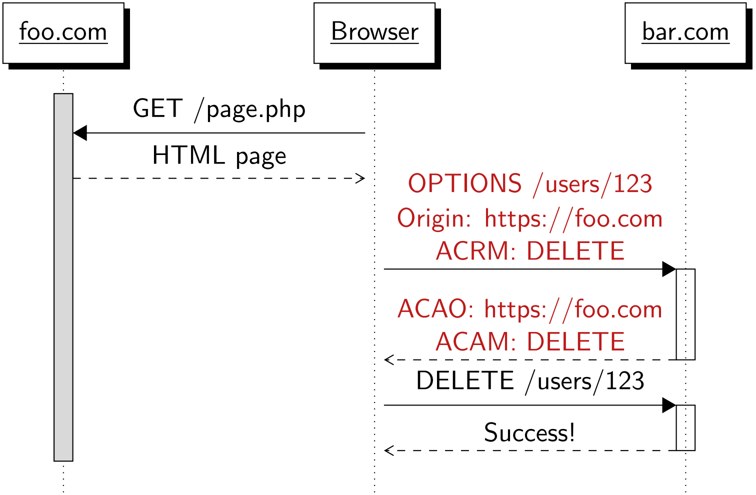
CORS: Non-Simple Requests

HTTP requests which are not simple use a more complicated protocol, involving a preliminary approval 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, sensitive side-effect at the server (unsafe methods)
- custom headers can have an arbitrarily complex semantics for web applications, hence pose a potential security threat

SOP normally prevents this form of cross-origin requests. The preflight requests in CORS allow a relaxation 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();
```

CORS: Credentialed Requests

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

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 even sent.

When responding to a credentialed request, the server cannot use the wildcard * in the `Access-Control-Allow-Origin` header!

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 before using it
2. no information leakage: only credentialed requests might disclose confidential information and the callee has control over them

This is a huge improvement over JSONP, but please note one can still write insecure applications!

SOP Pitfalls: DNS Rebinding

The security of SOP relies on the security of DNS. If the DNS cannot be trusted, then SOP can be bypassed. A well-known attack against SOP is called DNS rebinding [2]:

1. the attacker registers `evil.com` and gets access to 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 a request to `evil.com/password.txt`
4. since the TTL has expired, the attacker now provides a different DNS record pointing to a target server on the victim's intranet
5. since the browser sees a same-origin response, the malicious script can read the password file and leak it to the attacker!

References

Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi.
Dr cookie and mr token - web session implementations and how to live with them.
In *Proceedings of the Second Italian Conference on Cyber Security, Milan, Italy, February 6th - to - 9th, 2018*, 2018.

Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh.
Protecting browsers from dns rebinding attacks.
In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, pages 421–431, 2007.