Security II - Frames

Stefano Calzavara

Università Ca' Foscari Venezia

March 9, 2020

Università
Ca'Foscari
Venezia

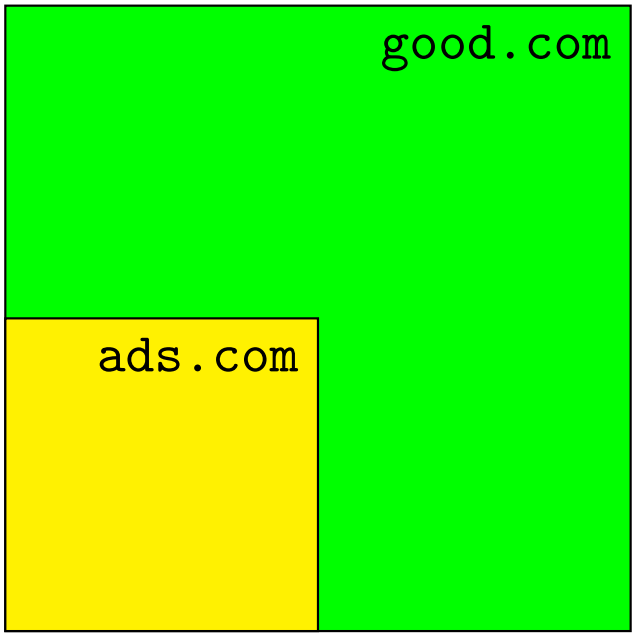
Frames

A frame is a part of a web page which renders content independently of its container. Typically used for:

- advertisement: content served by an advertisement network is placed in a separate area to generate revenue
- authentication: login form of a single sign-on provider is placed in a separate area, which looks the same on all including sites
- gadgets: "like" buttons and such

Frames are quite interesting from a security perspective!

Frames and SOP



The parent frame and its children keep living in their own origins:

- if a page at origin o_1 opens a frame towards a page at origin o_2, the two pages can access the DOM of each other only when $o_1 = o_2$
- this way, `good.com` can load ads from `ads.com` without exposing cookies and other secrets

Frames and SOP

SOP does not constrain framing in anyway: normally, every page on the Web can open frames towards any other page on the Web.

Alert!

The operators of `good.com` might want to forbid framing content from `evil.com`: they can do this by using the `frame-src` directive of CSP.

Alert!

The operators of `good.com` might want to forbid being framed from `evil.com`: we will discuss this later in the lecture.

Opening Frames

It is possible to load a frame by using the `<iframe>` tag, setting its `src` attribute to the page hosting the desired content.

```
<iframe src="https://foo.com" height="200" width="300"/>
```

The parent and the child can then get a reference to each other:

- the parent stores the opened frames in the `window.frames` array
- the child stores its opener in the `window.parent` variable

This is useful to implement frame communication between the parent and the child.

Frame Communication

Frames on the same origin can communicate by reading and writing over their DOMs, since SOP does not isolate them.

```
child = window.frames[0];
secret = child.document.getElementById("secret");
got_secret = 1;
```

```
got_secret = 0;
while (got_secret != 1) {
    parent = window.parent;
    got_secret = parent.document.got_secret;
    sleep(1000);
}
```

Domain Relaxation

Two pages on domains sharing a sufficiently long common suffix[1] can relax their `document.domain` property to get the same origin and enable frame communication.

p_1 at `https://www.foo.com` and p_2 at `https://mail.foo.com`:

1. p_1 loads p_2 inside a frame
2. p_1 sets `document.domain` to `foo.com`
3. p_2 sets `document.domain` to `foo.com`
4. p_1 and p_2 now have the same origin and can communicate

Step 2 would be required even if p_1 was already sitting at `foo.com`!

[1] `https://publicsuffix.org/`

Domain Relaxation

Domain relaxation might be useful to play around SOP, but might introduce security risks:

1. assume that `www.foo.com` implements domain relaxation, e.g., because it wants to communicate with `mail.foo.com`
2. the attacker finds an XSS vulnerability at `vuln.foo.com`
3. the attacker exploits the XSS to open a frame to `www.foo.com`
4. the XSS then sets `document.domain` to `foo.com`

Now the XSS at `vuln.foo.com` escalated to `www.foo.com`, since the attacker can access the DOM of the latter.

postMessage

Frame communication between different origins is better implemented by means of the `postMessage` API.

Message sending can be performed using:

```
targetWindow.postMessage(message, targetOrigin)
```

with:

- `targetWindow`: a reference to the receiver, e.g, parent or child
- `message`: any serializable data to be sent
- `targetOrigin`: the origin of the intended recipient (can be `"*"`)

postMessage

Message reception is handled by setting a listener as follows:

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

where `receiveMessage` is a callback waiting for an event object with the following properties:

- `data`: the deserialized received data
- `origin`: the origin of the sender (when `postMessage` was called!)
- `source`: a reference to the sender, e.g., parent or child

Notice that the origin of `source` might be different from `origin`!

postMessage: Security Considerations

`postMessage` is better than domain relaxation for frame communication, because it is more general and exchanges serialized data without granting scripting access to the frame.

However, please recall the following [2]:

- if you are sending confidential data, always specify the origin of the intended recipient in the `postMessage` invocation
- if you are willing to receive data, check the origin of the sender whenever possible and always sanitize the content of the message
- if you want to communicate back to the original sender, do not blindly trust the reference to its window and specify its origin

postMessage: Example

Sender:

```
rcv = window.frames[0];
msg = document.cookie;
rcv.postMessage(msg, "https://www.trusted.com");
```

Receiver:

```
window.addEventListener("message", getCookie);

function getCookie(ev) {
    if (ev.origin != "https://www.example.com")
       return;
    msg = sanitize(ev.data);
    ack = "Got cookie: " + msg;
    ev.source.postMessage(ack, "https://www.example.com");
}
```

Frame Sandboxing

It is possible to restrict the privileges of frames by setting the `sandbox` attribute to the empty string. The most important restrictions are:

1. the content of the frame is treated as being from a unique origin
2. form submission is blocked
3. all forms of script execution are blocked
4. plugin execution is blocked
5. popup creation is blocked
6. top-level navigation via `window.top.location` is blocked

Frame Sandboxing

It is possible to relax individual security restrictions by setting a list of special keywords in the `sandbox` attribute.

Default restriction	**Relaxed with**
enforce unique origin	`allow-same-origin`
form submission is blocked	`allow-forms`
script execution is blocked	`allow-scripts`
popup creation is blocked	`allow-popups`
top-level navigation is blocked	`allow-top-navigation`

The sandboxing flags applied to a frame also apply to any windows or frames created in the sandbox.

Frame Sandboxing: Example

The following snippet of code shows how to embed a "posting" widget from Twitter while enforcing least privilege:

```
<iframe src="https://platform.twitter.com/tweet_button.html"
        sandbox="allow-same-origin allow-scripts
                 allow-popups allow-forms"/>
```

- `allow-scripts`: required to let JS deal with user interaction
- `allow-popups`: required as the button pops up a tweeting form
- `allow-forms`: required to submit the tweet
- `allow-same-origin`: required to support login to Twitter

Frame Sandboxing and Privilege Separation

Frame sandboxing is great not just to safely load content from untrusted web origins, but also to enforce privilege separation in web apps

1. break your application up into logical pieces
2. sandbox each piece with the minimal privilege possible
3. use `postMessage` to let pieces communicate

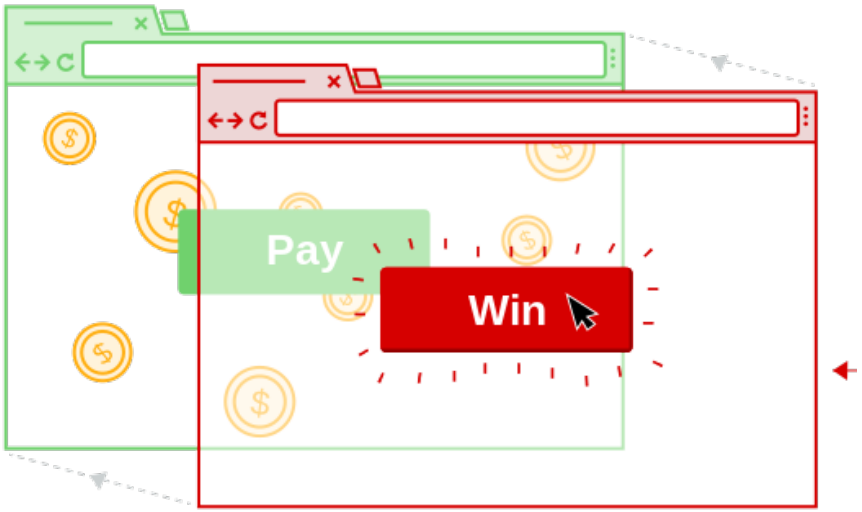
Demo: Evalbox from HTML5 Rocks.

Alert!

When framing same-origin content, be aware that the combination of `allow-same-origin` and `allow-scripts` might lead to the removal of the `sandbox` attribute!

Clickjacking

Clickjacking is a UI-based attack in which a user is tricked into clicking on actionable content on a target website by clicking on some other content in a decoy website.



The attack uses the `opacity` and `z-index` attributes of CSS to place an invisible frame pointing to the target website on top of content of the decoy website.

Framebusting

Back in the days, websites used to rely on Javascript-based framebusting techniques to defend against clickjacking.

```
<script type="text/javascript">
if (top != self)
   top.location = self.location;
</script>
```

Unfortunately, framebusting is only deceptively simple, even more so in modern browsers... for example, setting `sandbox="allow-forms"` is a great way to disable framebusting!

X-Frame-Options

A better solution against clickjacking is based on the `X-Frame-Options` header (XFO for short). It can take three possible values:

- `DENY`: page framing is denied
- `SAMEORIGIN`: page framing is only allowed on the same origin
- `ALLOW-FROM` u: page framing is only allowed at u

Note that `ALLOW-FROM` is not supported by Chrome and derivates! It has also been recently removed from Firefox, hence it is better avoided.

Problems with XFO

Limited Expressiveness

XFO does not allow to express useful policies like:

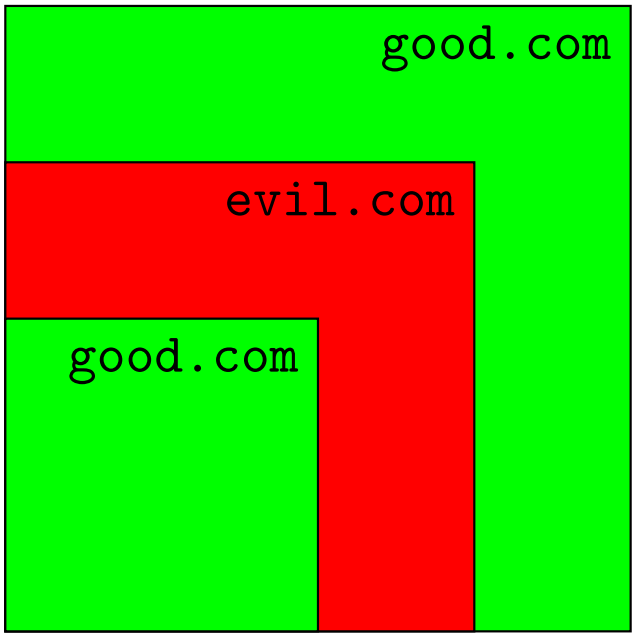
- framing is allowed on any origin from a whitelist
- framing is allowed on any subdomain of a given site

Incomplete Specification

Since XFO was implemented in browsers before being standardized, different browsers might give different interpretations of the same header.

Double Framing

The XFO specification does not mandate whether origin checks have to be performed on the top-level browsing context or on the full chain of ancestors, hence some browsers are subject to double framing attacks.



An attacker who exploits a markup injection at `good.com` can perform clickjacking by abusing nested frames, unless the browser checks the full chain of frame ancestors.

CSP frame-ancestors

The `frame-ancestors` directive introduced in CSP Level 2 is the best protection mechanism against clickjacking on modern browsers

- it leverages the full expressive power of the source expressions of CSP
- it solves the issue of double framing, since all ancestors are checked

What about users of legacy browsers?

- it is possible to send both CSP and XFO: the former is enforced by modern browsers, the latter is enforced by legacy browsers
- wait a minute, XFO is less expressive than CSP...

CSP vs XFO: Consistency

Recent research studied the problem of combining CSP and XFO [1].

Definition

A policy p is consistent for the set of browsers B if and only if it enforces the same security restrictions on all $b \in B$.

Example

```
CSP: frame-ancestors 'self'
XFO: SAMEORIGIN
```

Unfortunately, coming up with consistent policies is not always so easy...

CSP vs XFO: Relaxed Consistency

Definition

A policy p is security-oriented if and only if:

1. p is consistent for the set of legacy browsers B_l
2. p is consistent for the set of modern browsers B_m
3. for all $b_l \in B_l$ and $b_m \in B_m$, the security restrictions on b_l are no weaker than the security restrictions on b_m

Example (at `https://www.example.com`)

```
CSP: frame-ancestors https://*.example.com
XFO: SAMEORIGIN
```

CSP vs XFO: Relaxed Consistency

Definition

A policy p is compatibility-oriented if and only if:

1. p is consistent for the set of legacy browsers B_l
2. p is consistent for the set of modern browsers B_m
3. for all $b_l \in B_l$ and $b_m \in B_m$, the security restrictions on b_m are no weaker than the security restrictions on b_l

Example

```
CSP: frame-ancestors 'none'
XFO: SAMEORIGIN
```

CSP vs XFO: Summary

It is important to use both CSP and XFO, yet their combination is hard:

- consistency is the desired property, yet hard to achieve in practice
- inconsistent policies which are just either security-oriented or compatibility-oriented might be acceptable in practice
- most of the other inconsistent policies are bad!

A recent paper [1] estimated that 10% of the existing framing control policies are inconsistent and most of them can be completely bypassed in at least one browser!

CSP vs XFO: Examples

Which consistency properties are satisfied by the following policies?

```
XFO: ALLOW-FROM https://www.foo.com
```

CSP vs XFO: Examples

Which consistency properties are satisfied by the following policies?

```
XFO: ALLOW-FROM https://www.foo.com
```

```
XFO: ALLOW-FROM https://www.foo.com
CSP: frame-ancestors https://www.foo.com
```

CSP vs XFO: Examples

Which consistency properties are satisfied by the following policies?

```
XFO: ALLOW-FROM https://www.foo.com
```

```
XFO: ALLOW-FROM https://www.foo.com
CSP: frame-ancestors https://www.foo.com
```

```
XFO: DENY
CSP: frame-ancestors https://www.foo.com
```

CSP vs XFO: Examples

Which consistency properties are satisfied by the following policies?

```
XFO: ALLOW-FROM https://www.foo.com
```

```
XFO: ALLOW-FROM https://www.foo.com
CSP: frame-ancestors https://www.foo.com
```

```
XFO: DENY
CSP: frame-ancestors https://www.foo.com
```

```
XFO: SAMEORIGIN, DENY
```

Web Tracking

A last interesting aspect of frames is related to privacy. In particular, frames are a key component of web tracking:

1. u visits `a.com`, which loads in a frame a tracking script s from `t.com`
2. s sets a cookie c with a unique identifier: this is called a third-party cookie, since s runs at `t.com` (who owns the cookie)
3. u visits `b.com`, which also loads s: since `t.com` receives both c and the `Referer` header, it learns that the user who got c visited `b.com`
4. this practice allows tracking the navigation profile of u, though it does not necessarily disclose her identity

Different browsers have different approaches to this form of tracking.

References

Stefano Calzavara, Sebastian Roth, Alvise Rabitti, Michael Backes, and Ben Stock.
A tale of two headers: a formal analysis of inconsistent click-jacking protection on the Web.
In *USENIX Security*. USENIX Association, 2020.

Sooel Son and Vitaly Shmatikov.
The postman always rings twice: Attacking and defending postmessage in HTML5 websites.
In *NDSS*. The Internet Society, 2013.