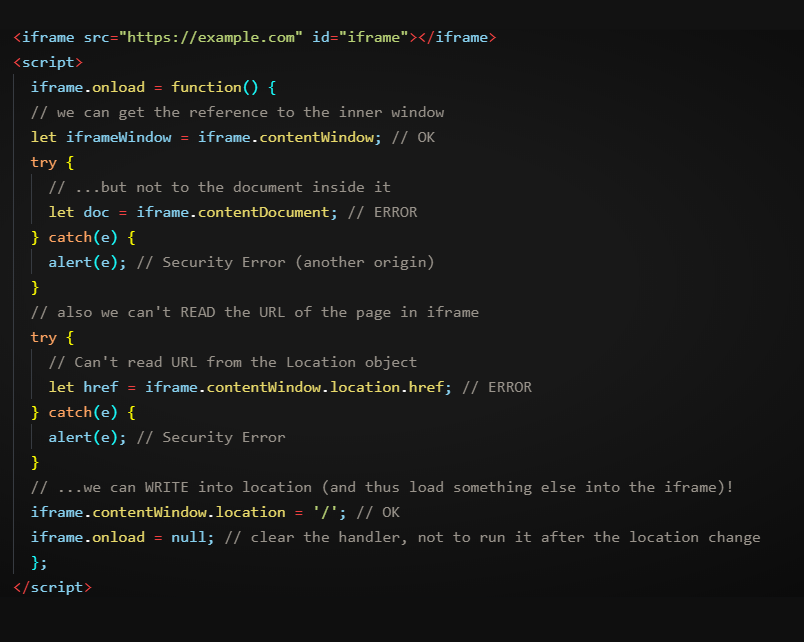
**Iframe**

An <iframe> tag hosts a separate embedded window, with its own separate documents and window objects.

We can access them using properties:

* Iframe.contentWindow to get the window inside the <iframe>
* Iframe.contentDocument to get the document inside the <iframe> iframe.contentWindow.Document

When we access something inside the embedded window, the browser checks if the iframe has the same origin. If that’s not so then the access is denied (writing to location is an exception, its still permitted).



**iframe.onload vsiframe.contentWindow.onload**

The iframe.onload event (on the <iframe>tag) is essentially the same as iframe.contentWindow.onload (on the embedded window object). It triggers when the embedded window fully loads with all resources.

**Wrong Document pitfall**

When an iframe comes from the same origin, and we may access its document, there’s a pitfall. It’s not related to cross-domain things, but important to know.

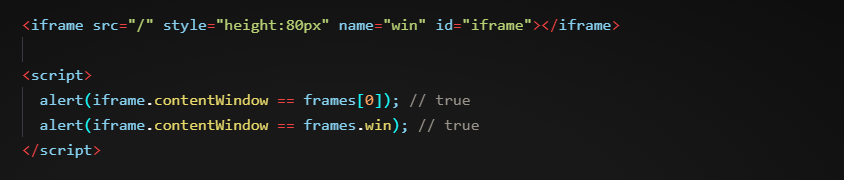
Upon its creation, an iframe immediately has a document. But that document is different from the one that loads into it!

**Window.Frames**

An alternative way to get a window object for <iframe>– is to get it from the named collectionwindow.frames:

* By number: window.frames[0] – the window object for the first frame in the document.
* By name: window.frames.iframeName – the window object for the frame withname="iframeName".

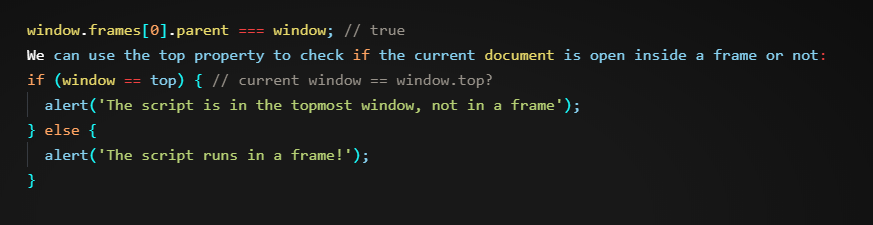
For instance:



An iframe may have other iframes inside. The corresponding window objects form a hierarchy.

Navigation links are:

* + window.frames – the collection of “children” windows (for nested frames).
  + window.parent – the reference to the “parent” (outer) window.
  + window.top – the reference to the topmost parent window.

For instance:  
  
****

**The “SANDBOX” Iframe Attribute**

The sandbox attribute allows for the exclusion of certain actions inside an <iframe> in order to prevent it from executing untrusted code. Its “sandboxes” the iframe by treating it as coming from another origin and/or applying other limitations.

There’s a “default set” of restrictions applied for <iframe sandbox src="...">. But it can be relaxed if we provide a space-separated list of restrictions that should not be applied as a value of the attribute, like this: <iframe sandbox="allow-forms allow-popups">.

In other words, an empty "sandbox" attribute puts the strictest limitations possible, but we can put a space-delimited list of those that we want to lift.

**List of Limitations**

**allow-same-origin**

By default, "sandbox" forces the “different origin” policy for the iframe. In other words, it makes the browser to treat the iframe as coming from another origin, even if its src points to the same site. With all implied restrictions for scripts. This option removes that feature.

**allow-top-navigation**

Allows the iframe to change parent.location.

**allow-forms**

Allows to submit forms from iframe.

**allow-scripts**

Allows to run scripts from the iframe.

**allow-popups**

Allows to window.open popups from the iframe

**Cross Window Messaging**

The postMessage interface allows windows to talk to each other no matter which origin they are from.

So, it’s a way around the “Same Origin” policy. It allows a window from john-smith.com to talk to gmail.comand exchange information, but only if they both agree and call corresponding JavaScript functions. That makes it safe for users.

The interface has two parts.

**postMessage**

The window that wants to send a message calls postMessage method of the receiving window. In other words, if we want to send the message to win, we should call win.postMessage(data, targetOrigin).

**Arguments:**

**data**

The data to send. Can be any object, the data is cloned using the “structured cloning algorithm”. IE supports only strings, so we should JSON.stringify complex objects to support that browser.

**targetOrigin**

Specifies the origin for the target window, so that only a window from the given origin will get the message.

**Clickjacking**

Clickjacking, also known as a "UI redress attack" or "UI redressing," is a type of web security vulnerability where an attacker tricks a user into clicking on something different from what the user perceives, potentially leading to unintended actions. This is typically done by overlaying a malicious webpage element, such as a button or link, on top of a legitimate webpage element, making the user believe they are clicking on one thing when, in reality, they are interacting with something else.

Here's how clickjacking typically works:

1. **Creation of Malicious Page:** The attacker creates a webpage containing content they want the victim to interact with, such as a fake login button or a play button for a video.
2. **Overlay:** The attacker positions this content on top of a legitimate website or web application, which can be transparent or nearly invisible to the user.
3. **Tricking the User:** When the victim visits the legitimate website, they may unknowingly click on the malicious content, thinking they are interacting with the legitimate site.
4. **Unauthorized Actions:** As a result, the victim's actions are redirected to the malicious content, and they might unwittingly perform actions like liking a social media page, sharing sensitive information, or making unintended purchases.

Clickjacking attacks can be used for various malicious purposes, including:

* Stealing sensitive information (e.g., clicking on a hidden "Like" button to post content on a user's social media profile).
* Forcing users to perform actions without their consent (e.g., initiating a money transfer).
* Spreading malware by tricking users into downloading files or running malicious scripts.

To protect against clickjacking attacks, web developers can implement security measures such as:

1. **X-Frame-Options Header:** This HTTP header can be set to deny the website from being displayed in iframes (inline frames), making it more challenging for attackers to overlay content.
2. **Content Security Policy (CSP):** CSP allows web developers to define which resources can be loaded and executed on a webpage, preventing unauthorized scripts from running.
3. **Framebusting Code:** JavaScript can be used to break out of iframes and prevent a webpage from being displayed within a frame. However, this method is not foolproof.
4. **User Education:** Educating users about the risks of interacting with unfamiliar or suspicious elements on webpages can also help prevent clickjacking attacks.

It's essential for both web developers and users to be aware of clickjacking threats and take appropriate precautions to mitigate them.