

8

CLICKJACKING



Clickjacking, or user-interface redressing, is an attack that tricks users into clicking a malicious button that has been made to look legitimate. Attackers achieve this by using HTML page-overlay techniques to hide one web page within another. Let's discuss this fun-to-exploit vulnerability, why it's a problem, and how you can find instances of it.

Note that clickjacking is rarely considered in scope for bug bounty programs, as it usually involves a lot of user interaction on the victim's part. Many programs explicitly list clickjacking as out of scope, so be sure to check the program's policies before you start hunting! However, some programs still accept them if you can demonstrate the impact of the clickjacking vulnerability. We will look at an accepted report later in the chapter.

Mechanisms

Clickjacking relies on an HTML feature called an *iframe*. HTML iframes allow developers to embed one web page within another by placing an `<iframe>` tag on the page, and then specifying the URL to frame in the tag's `src` attribute. For example, save the following page as an HTML file and open it with a browser:

```
<html>
  <h3>This is my web page.</h3>
  <iframe src="https://www.example.com" width="500" height="500"></iframe>
  <p>If this window is not blank, the iframe source URL can be framed!</p>
</html>
```

You should see a web page that looks like Figure 8-1. Notice that a box places *www.example.com* in one area of the larger page.

This is my web page.

Example Domain

This domain is for use in illustrative examples in documents. You may use this domain in literature without prior coordination or asking for permission.

More information...

If this window is not blank, the iframe source URL can be framed!

Figure 8-1: If the *iframe* is not blank, the page specified in the *iframe*'s *src* attribute can be framed!

Some web pages can't be framed. If you place a page that can't be framed within an iframe, you should see a blank iframe, as in Figure 8-2.

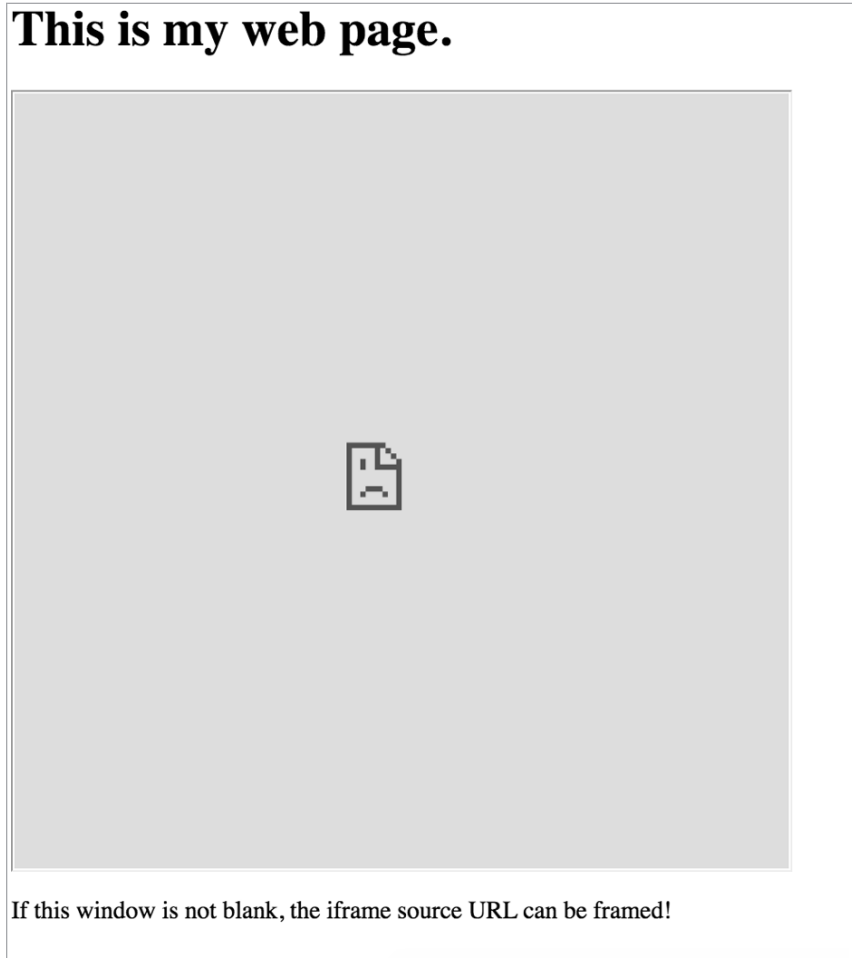


Figure 8-2: If the iframe is blank, the iframe source cannot be framed.

Iframes are useful for many things. The online advertisements you often see at the top or sides of web pages are examples of iframes; companies use these to include a premade ad in your social media or blog. Iframes also allow you to embed other internet resources, like videos and audio, in your web pages. For example, this iframe allows you to embed a YouTube video in an external site:

```
<iframe width="560" height="315"
src="https://www.youtube.com/embed/d1192Sqk" frameborder="0"
allow="accelerometer; autoplay; encrypted-media; gyroscope; picture-in-picture"
allowfullscreen>
</iframe>
```

Iframes have made our internet a more vibrant and interactive place. But they can also be a danger to the framed web page because they introduce the possibilities of a clickjacking attack. Let's say that *example.com* is a banking site that includes a page for transferring your money with a click of a button. You can access the balance transfer page with the URL *https://www.example.com/transfer_money*.

This URL accepts two parameters: the recipient account ID and the transfer amount. If you visit the URL with these parameters present, such as *https://www.example.com/transfer_money?recipient=RECIPIENT_ACCOUNT&amount=AMOUNT_TO_TRANSFER*, the HTML form on the page will appear prefilled (Figure 8-3). All you have to do is to click the Submit button, and the HTML form will initiate the transfer request.

Welcome to example.com bank!

On this page, you can tranfer your money to another account.

Recipient account:

Amount to transfer:

Figure 8-3: The balance transfer page with the HTTP POST parameters prefilled

Now imagine that an attacker embeds this sensitive banking page in an iframe on their own site, like this:

```
<html>
  <h3>Welcome to my site!</h3>
  <iframe src="https://www.example.com/transfer_money?
    recipient=attacker_account_12345&amount=5000"
    width="500" height="500">
  </iframe>
</html>
```

This iframe embeds the URL for the balance transfer page. It also passes in the URL parameters to prefill the transfer recipient and amount. The attacker hides this iframe on a website that appears to be harmless, then tricks the user into clicking a button on the sensitive page. To achieve this, they overlay multiple HTML elements in a way that obscures the banking form. Take a look at this HTML page, for example:

```
<html>
  <style>
    #victim-site {
      width:500px;
```

```

        height:500px;
        ❶ opacity:0.00001;
        ❷ z-index:1;
    }
    #decoy {
        ❸ position:absolute;
        width:500px;
        height:500px;
        ❹ z-index:-1;
    }
</style>
<div id="decoy">
    <h3>Welcome to my site!</h3>
    <h3>This is a cybersecurity newsletter that focuses on bug
bounty news and write-ups!
    Please subscribe to my newsletter below to receive new
cybersecurity articles in your email inbox!</h3>
    <form action="/subscribe" method="post">
        <label for="email">Email:</label>
        ❺ <br>
        <input type="text" id="email" value="Please enter your email!">
        ❻ <br><br>
        <input type="submit" value="Submit">
    </form>
</div>
<iframe id="victim-site"
    src="https://www.example.com/transfer_money?
    recipient=attacker_account_12345&amount=5000"
    width="500" height="500">
</iframe>
</html>

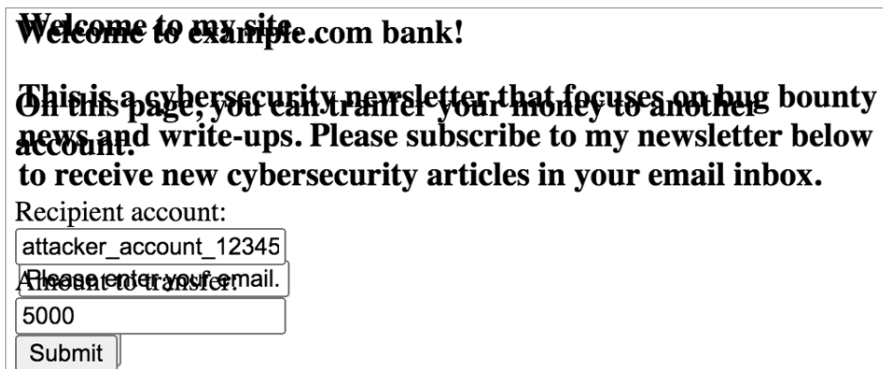
```

You can see that we've added a `<style>` tag at the top of the HTML page. Anything between `<style>` tags is CSS code used to specify the styling of HTML elements, such as font color, element size, and transparency. We can style HTML elements by assigning them IDs and referencing these in our style sheet.

Here, we set the position of our decoy element to `absolute` to make the decoy site overlap with the iframe containing the victim site ❸. Without the `absolute` position directive, HTML would display these elements on separate parts of the screen. The decoy element includes a Subscribe to Newsletter button, and we carefully position the iframe so the Transfer Balance button sits directly on top of this Subscribe button, using new lines created by HTML's line break tag `
` ❺ ❻. We then make the iframe invisible by setting its opacity to a very low value ❶. Finally, we set the z-index of the iframe to a higher value than the decoys ❷ ❹. The *z-index* sets the stack order of different HTML elements. If two HTML elements overlap, the one with the highest z-index will be on top.

By setting these CSS properties for the victim site iframe and decoy form, we get a page that looks like it's for subscribing to a newsletter, but contains an invisible form that transfers the user's money into the attacker's account.

Let's turn the opacity of the iframe back to `opacity:1` to see how the page is actually laid out. You can see that the Transfer Balance button is located directly on top of the Subscribe to Newsletter button (Figure 8-4).



Welcome to my site.

Welcome to example.com bank!

This is a cybersecurity newsletter that focuses on bug bounty news and write-ups. Please subscribe to my newsletter below to receive new cybersecurity articles in your email inbox.

Recipient account:

attacker_account_12345

Please enter your email.

5000

Submit

Figure 8-4: The Transfer Balance button lies directly on top of the Subscribe button. Victims think they're subscribing to a newsletter, but they're actually clicking the button to authorize a balance transfer.

Once we reset the opacity of the iframe to `opacity:0.00001` to make the sensitive form invisible, the site looks like a normal newsletter page (Figure 8-5).



Welcome to my site.

This is a cybersecurity newsletter that focuses on bug bounty news and write-ups. Please subscribe to my newsletter below to receive new cybersecurity articles in your email inbox.

Email:

Please enter your email.

Submit

Figure 8-5: The attacker tricks users into clicking the button by making the sensitive form invisible.

If the user is logged into the banking site, they'll be logged into the iframe too, so the banking site's server will recognize the requests sent by the iframe as legit. When the user clicks the seemingly harmless button, they're executing a balance transfer on *example.com*! They'll have accidentally transferred \$5,000 from their bank account balance to the attacker's account instead of subscribing to a newsletter. This is why we call this attack *user-interface redressing* or *clickjacking*: the attacker redressed the user interface to hijack user clicks, repurposing the clicks meant for their page and using them on a victim site.

This is a simplified example. In reality, payment applications will not be implemented this way, because it would violate data security standards. Another thing to remember is that the presence of an easy-to-prevent vulnerability on a critical functionality, like a clickjacking vulnerability on the balance transfer page, is a symptom that the application does not follow the best practices of secure development. This example application is likely to contain other vulnerabilities, and you should test it extensively.

Prevention

Two conditions must be met for a clickjacking vulnerability to happen. First, the vulnerable page has to have functionality that executes a state-changing action on the user's behalf. A *state-changing action* causes changes to the user's account in some way, such as changing the user's account settings or personal data. Second, the vulnerable page has to allow itself to be framed by an iframe on another site.

The HTTP response header `X-Frame-Options` lets web pages indicate whether the page's contents can be rendered in an iframe. Browsers will follow the directive of the header provided. Otherwise, pages are frameable by default.

This header offers two options: `DENY` and `SAMEORIGIN`. If a page is served with the `DENY` option, it cannot be framed at all. The `SAMEORIGIN` option allows framing from pages of the same origin: pages that share the same protocol, host, and port.

```
X-Frame-Options: DENY
X-Frame-Options: SAMEORIGIN
```

To prevent clickjacking on sensitive actions, the site should serve one of these options on all pages that contain state-changing actions.

The `Content-Security-Policy` response header is another possible defense against clickjacking. This header's `frame-ancestors` directive allows sites to indicate whether a page can be framed. For example, setting the directive to `'none'` will prevent any site from framing the page, whereas setting the directive to `'self'` will allow the current site to frame the page:

```
Content-Security-Policy: frame-ancestors 'none';
Content-Security-Policy: frame-ancestors 'self';
```

Setting `frame-ancestors` to a specific origin will allow that origin to frame the content. This header will allow the current site, as well as any page on the subdomains of *example.com*, to frame its contents:

```
Content-Security-Policy: frame-ancestors 'self' *.example.com;
```

Besides implementing `X-Frame-Options` and the `Content-Security-Policy` to ensure that sensitive pages cannot be framed, another way of protecting against clickjacking is with `SameSite` cookies. A web application instructs

the user's browser to set cookies via a Set-Cookie header. For example, this header will make the client browser set the value of the cookie PHPSESSID to UEhQUoVTUo1E:

Set-Cookie: PHPSESSID=UEhQUoVTUo1E

In addition to the basic `cookie_name=cookie_value` designation, the Set-Cookie header allows several optional flags you can use to protect your users' cookies. One of them is the `SameSite` flag, which helps prevent clickjacking attacks. When the `SameSite` flag on a cookie is set to `Strict` or `Lax`, that cookie won't be sent in requests made within a third-party iframe:

Set-Cookie: PHPSESSID=UEhQUoVTUo1E; Max-Age=86400; Secure; HttpOnly; SameSite=Strict
Set-Cookie: PHPSESSID=UEhQUoVTUo1E; Max-Age=86400; Secure; HttpOnly; SameSite=Lax

This means that any clickjacking attack that requires the victim to be authenticated, like the banking example we mentioned earlier, would not work, even if no HTTP response header restricts framing, because the victim won't be authenticated in the clickjacked request.

Hunting for Clickjacking

Find clickjacking vulnerabilities by looking for pages on the target site that contain sensitive state-changing actions and can be framed.

Step 1: Look for State-Changing Actions

Clickjacking vulnerabilities are valuable only when the target page contains state-changing actions. You should look for pages that allow users to make changes to their accounts, like changing their account details or settings. Otherwise, even if an attacker can hijack user clicks, they can't cause any damage to the website or the user's account. That's why you should start by spotting the state-changing actions on a site.

For example, let's say you're testing a subdomain of *example.com* that handles banking functionalities at *bank.example.com*. Go through all the functionalities of the web application, click all the links, and write down all the state-changing options, along with the URL of the pages they're hosted on:

State-changing requests on *bank.example.com*

- Change password: *bank.example.com/password_change*
- Transfer balance: *bank.example.com/transfer_money*
- Unlink external account: *bank.example.com/unlink*

You should also check that the action can be achieved via clicks alone. Clickjacking allows you to forge only a user's clicks, not their keyboard actions. Attacks that require users to explicitly type in values are possible, but generally not feasible because they require so much social engineering. For example,

on this banking page, if the application requires users to explicitly type the recipient account and transfer amount instead of loading them from a URL parameter, attacking it with clickjacking would not be feasible.

Step 2: Check the Response Headers

Then go through each of the state-changing functionalities you've found and revisit the pages that contain them. Turn on your proxy and intercept the HTTP response that contains that web page. See if the page is being served with the X-Frame-Options or Content-Security-Policy header.

If the page is served without any of these headers, it may be vulnerable to clickjacking. And if the state-changing action requires users to be logged in when it is executed, you should also check if the site uses SameSite cookies. If it does, you won't be able to exploit a clickjacking attack on the site's features that require authentication.

Although setting HTTP response headers is the best way to prevent these attacks, the website might have more obscure safeguards in place. For example, a technique called *frame-busting* uses JavaScript code to check if the page is in an iframe, and if it's framed by a trusted site. Frame-busting is an unreliable way to protect against clickjacking. In fact, frame-busting techniques can often be bypassed, as I will demonstrate later in this chapter.

You can confirm that a page is frameable by creating an HTML page that frames the target page. If the target page shows up in the frame, the page is frameable. This piece of HTML code is a good template:

```
<HTML>
<head>
  <title>Clickjack test page</title>
</head>
<body>
  <p>Web page is vulnerable to clickjacking if the iframe is populated with the target
page!</p>
  <iframe src="URL_OF_TARGET_PAGE" width="500" height="500"></iframe>
</body>
</html>
```

Step 3: Confirm the Vulnerability

Confirm the vulnerability by executing a clickjacking attack on your test account. You should try to execute the state-changing action through the framed page you just constructed and see if the action succeeds. If you can trigger the action via clicks alone through the iframe, the action is vulnerable to clickjacking.

Bypassing Protections

Clickjacking isn't possible when the site implements the proper protections. If a modern browser displays an X-Frame-Options protected page, chances are you can't exploit clickjacking on the page, and you'll have to find another

vulnerability, such as XSS or CSRF, to achieve the same results. Sometimes, however, the page won't show up in your test iframe even though it lacks the headers that prevent clickjacking. If the website itself fails to implement complete clickjacking protections, you might be able to bypass the mitigations.

Here's an example of what you can try if the website uses frame-busting techniques instead of HTTP response headers and SameSite cookies: find a loophole in the frame-busting code. For instance, developers commonly make the mistake of comparing only the top frame to the current frame when trying to detect whether the protected page is framed by a malicious page. If the top frame has the same origin as the framed page, developers may allow it, because they deem the framing site's domain to be safe. Essentially, the protection's code has this structure:

```
if (top.location == self.location){  
    // Allow framing.  
}  
else{  
    // Disallow framing.  
}
```

If that is the case, search for a location on the victim site that allows you to embed custom iframes. For example, many social media sites allow users to share links on their profile. These features often work by embedding the URL in an iframe to display information and a thumbnail of the link. Other common features that require custom iframes are those that allow you to embed videos, audio, images, and custom advertisements and web page builders.

If you find one of these features, you might be able to bypass clickjacking protection by using the *double iframe trick*. This trick works by framing your malicious page within a page in the victim's domain. First, construct a page that frames the victim's targeted functionality. Then place the entire page in an iframe hosted by the victim site (Figure 8-6).

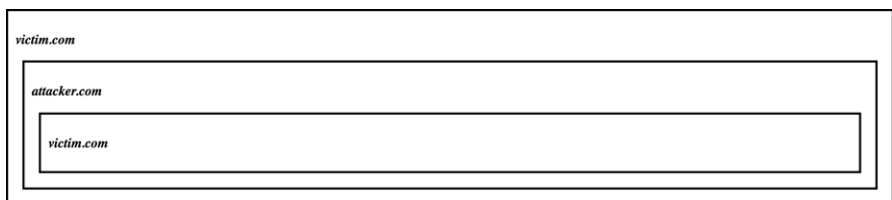


Figure 8-6: You can try to place your site in an iframe hosted by the victim site to bypass improper frame checking.

This way, both `top.location` and `self.location` point to *victim.com*. The frame-busting code would determine that the innermost *victim.com* page is framed by another *victim.com* page within its domain, and therefore deem the framing safe. The intermediary attacker page would go undetected.

Always ask yourself if the developer may have missed any edge cases while implementing protection mechanisms. Can you exploit these edge cases to your advantage?

Let's take a look at an example report. Periscope is a live streaming video application, and on July 10, 2019, it was found to be vulnerable to a clickjacking vulnerability. You can find the disclosed bug report at <https://hackerone.com/reports/591432/>. The site was using the X-Frame-Options ALLOW-FROM directive to prevent clickjacking. This directive lets pages specify the URLs that are allowed to frame it, but it's an obsolete directive that isn't supported by many browsers. This means that all features on the subdomains <https://canary-web.pscp.tv> and <https://canary-web.periscope.tv> were vulnerable to clickjacking if the victim was using a browser that didn't support the directive, such as the latest Chrome, Firefox, and Safari browsers. Since Periscope's account settings page allows users to deactivate their accounts, an attacker could, for example, frame the settings page and trick users into deactivating their accounts.

Escalating the Attack

Websites often serve pages without clickjacking protection. As long as the page doesn't contain exploitable actions, the lack of clickjacking protection isn't considered a vulnerability. On the other hand, if the frameable page contains sensitive actions, the impact of clickjacking would be correspondingly severe.

Focus on the application's most critical functionalities to achieve maximum business impact. For example, let's say a site has two frameable pages. The first page contains a button that performs transfers of the user's bank balance, while the second contains a button that changes the user's theme color on the website. While both of these pages contain clickjacking vulnerabilities, the impact of a clickjacking bug is significantly higher on the first page than on the second.

You can also combine multiple clickjacking vulnerabilities or chain clickjacking with other bugs to pave the way to more severe security issues. For instance, applications often send or disclose information according to user preferences. If you can change these settings via clickjacking, you can often induce sensitive information disclosures. Let's say that *bank.example.com* contains multiple clickjacking vulnerabilities. One of them allows attackers to change an account's billing email, and another one allows attackers to send an account summary to its billing email. The malicious page's HTML looks like this:

```
<html>
  <h3>Welcome to my site!</h3>
  <iframe
    src="https://bank.example.com/change_billing_email?email=attacker@attacker.com"
    width="500" height="500">
  </iframe>
  <iframe src="https://bank.example.com/send_summary" width="500" height="500">
  </iframe>
</html>
```

You could first change the victim's billing email to your own email, then make the victim send an account summary to your email address to leak the information contained in the account summary report. Depending on what the account summary discloses, you might be able to collect data including the street address, phone numbers, and credit card information associated with the account! Note that for this attack to succeed, the victim user would have to click the attacker's site twice.

A Note on Delivering the Clickjacking Payload

Often in bug bounty reports, you'll need to show companies that real attackers could effectively exploit the vulnerability you found. That means you need to understand how attackers can exploit clickjacking bugs in the wild.

Clickjacking vulnerabilities rely on user interaction. For the attack to succeed, the attacker would have to construct a site that is convincing enough for users to click. This usually isn't difficult, since users don't often take precautions before clicking web pages. But if you want your attack to become more convincing, check out the Social-Engineer Toolkit (<https://github.com/trustedsec/social-engineer-toolkit/>). This set of tools can, among other things, help you clone famous websites and use them for malicious purposes. You can then place the iframe on the cloned website.

In my experience, the most effective location in which to place the hidden button is directly on top of a Please Accept That This Site Uses Cookies! pop-up. Users usually click this button to close the window without much thought.

Finding Your First Clickjacking Vulnerability!

Now that you know what clickjacking bugs are, how to exploit them, and how to escalate them, go find your first clickjacking vulnerability! Follow the steps described in this chapter:

1. Spot the state-changing actions on the website and keep a note of their URL locations. Mark the ones that require only mouse clicks to execute for further testing.
2. Check these pages for the X-Frame-Options, Content-Security-Policy header, and a SameSite session cookie. If you can't spot these protective features, the page might be vulnerable!
3. Craft an HTML page that frames the target page, and load that page in a browser to see if the page has been framed.
4. Confirm the vulnerability by executing a simulated clickjacking attack on your own test account.
5. Craft a sneaky way of delivering your payload to end users, and consider the larger impact of the vulnerability.
6. Draft your first clickjacking report!