

TTP's used

Clickjacking like in this lab where there's a malicious button over an iframe can exploit several techniques described in MITRE ATT&CK like T1190 to exploit a public facing application. The iframe takes advantage of a web feature allowing contents from other websites to be embedded in another site, where the attacker can perfectly emulate the contents of a legitimate webpage. The malicious button is an example of T1204 User Execution where the user is tricked or socially engineered into clicking a malicious link, which in a click jacking attack could also coerce users to provide sensitive information or do other actions. Another tactic that describes a clickjacking entry point is T1189 Drive-by compromise where there are multiple ways of gaining access to information or a user's system through use of legitimate sites or masquerading as them such as malicious ads through a legitimate service that pretend to be from a real site but bring the victim to an attacker's domain with contents that look like the legitimate site.

What you observed

Clickjacking attacks and prevention steps seem like a cat and mouse game with multiple browser features being used to attack and mitigate clickjacking efforts. The lab explored using iframes to embed the defender's site into the attacker's, then to use javascript to redirect back to the defender's site when embedded, then the attacker's site could use sandbox to get around that, and finally the defender's apache configuration prevented the use of embedding into iframes. From this series of escalation I wouldn't be surprised if there is a way to leverage other features like opening a webpage from the attacker's server and serving up a cached version within an iframe from the same origin to prevent the configuration and scripting countermeasures from working as the cat and mouse games continue.

Thoughts on this type of attack (i.e., usefulness, skill set needed, level of effort required)

Embedding a perfect copy of a target website in an attacker's website is trivially easy for an attacker that can at least set up a domain. However, unless there's some specific need for it, it is also trivially easy to set a few options to prevent embedded versions of a defender's domain from being displayed in an attacker's website. While some sites might want to be embedded, it's at least a very easy step to fully disable embedding your domain to prevent clickjacking, and options for content-security-policy exist to whitelist domains you want to be embedded on.

Lessons learned

Since I used the css option for rgba and set the alpha value to a low amount (0 would be invisible), I was able to see that when the x-frame-options and content-security-policy options are set to

prevent the embedded contents from displaying, the malicious button was still present, and a user absentmindedly going to click the refresh button would still have clicked on the malicious button had it been really invisible. If the webpage made it a little more clear that something within that area of the page was prohibited, a user might be more cautious when clicking on the page. It just seemed like a network error and I wouldn't think twice clicking where the refresh page button is. I also learned just how easy it is to disable embedding my domain, and while I used other security features, I will definitely check these options in any domain I run.

Supporting recommendations on how to prevent the attack (i.e., countermeasure)

Using X-Frame-Options and Content-Security-Policy to block embedding the contents of your domain into another is a very easy configuration step that any domain with login flows should set to prevent clickjacking. Using custom scripts to bring your domain's window to the top of the frame can also be used to bust clickjacking, but since methods like using the sandbox attribute on the iframe exist to prevent scripts from running, it can't be relied on. There are other types of clickjacking prevention for users like browser extensions and advanced configuration within the browser settings like using NoScript to block content outside of the domain from loading and blocking transparent elements in HTML/CSS though these might break websites functionality.

Tasks and answers to lab questions are shown with screenshots, code screenshots appear in the [Appendix](#) at the end of the document.

Setup

I downloaded the Ubuntu 20.04 SEED VM, set up a shared folder and copied in the zipped files, then edited the /etc/hosts file to include the domain names for the apache servers that will run in the containers. I then navigated to the labsetup directory and ran the docker commands to build and bring up the servers, dcbuild, and dcup. I then navigated to the domains and they both showed what was expected, the defender cupcake website and the attacker malicious button page.

```
seed@VM: ~/Labsetup
GNU nano 4.8 /etc/hosts
# For XSS Lab
10.9.0.5 www.xsslabelgg.com
10.9.0.5 www.example32a.com
10.9.0.5 www.example32b.com
10.9.0.5 www.example32c.com
10.9.0.5 www.example60.com
10.9.0.5 www.example70.com

# For CSRF Lab
10.9.0.5 www.csrflabelgg.com
10.9.0.5 www.csrf-lab-defense.com
10.9.0.105 www.csrf-lab-attacker.com

# For Shellshock Lab
10.9.0.80 www.seedlab-shellshock.com

10.9.0.5 www.cjlab.com
10.9.0.105 www.cjlab-attacker.com

^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Paste Text ^T To Spell ^_ Go To Line
```

Figure 1: editing /etc/hosts

```
seed@VM: ~/Labsetup
Successfully tagged seed-image-defender-clickjacking:latest
Building attacker
Step 1/3 : FROM handsonsecurity/seed-server:apache
--> 2e03277733dc
Step 2/3 : COPY apache_attacker.conf server_name.conf /etc/apache2/sites-available/
--> Using cache
--> cddb0c15183c
Step 3/3 : RUN a2ensite server_name.conf && a2ensite apache_attacker.conf
--> Using cache
--> 4e3af6f133d1

Successfully built 4e3af6f133d1
Successfully tagged seed-image-attacker-clickjacking:latest
[04/05/25]seed@VM:~/Labsetup$ dcup
Starting defender-10.9.0.5 ... done
Starting attacker-10.9.0.105 ... done
Attaching to attacker-10.9.0.105, defender-10.9.0.5
defender-10.9.0.5 | * Starting Apache httpd web server apache2
*
attacker-10.9.0.105 | * Starting Apache httpd web server apache2
*
```

Figure 2: Build and run the docker containers

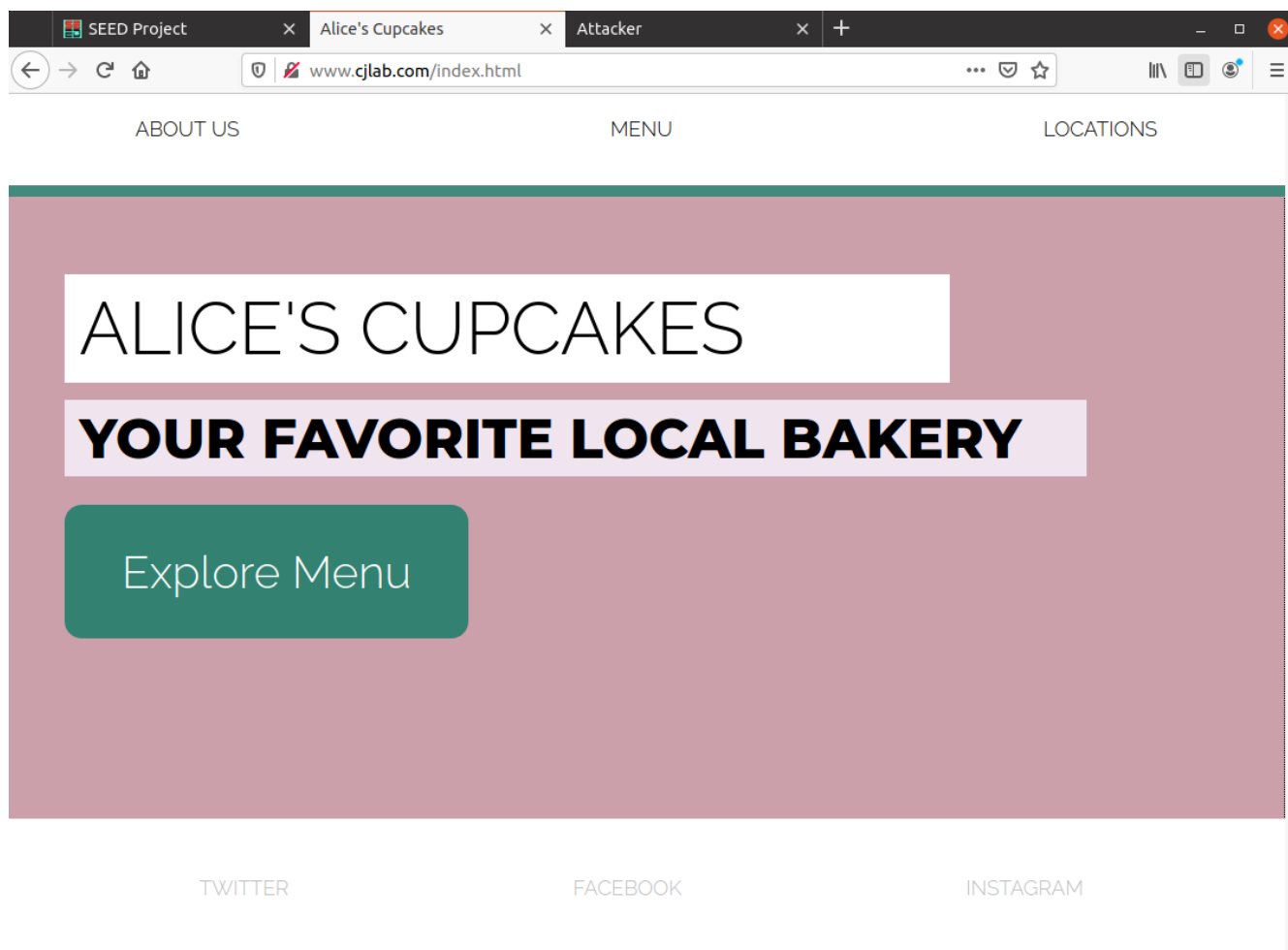


Figure 3: Defender website

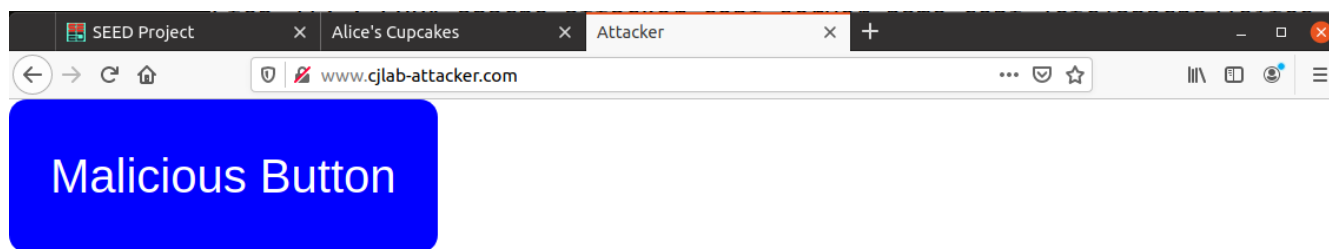


Figure 4: Attacker website before modifications

Task 3.1

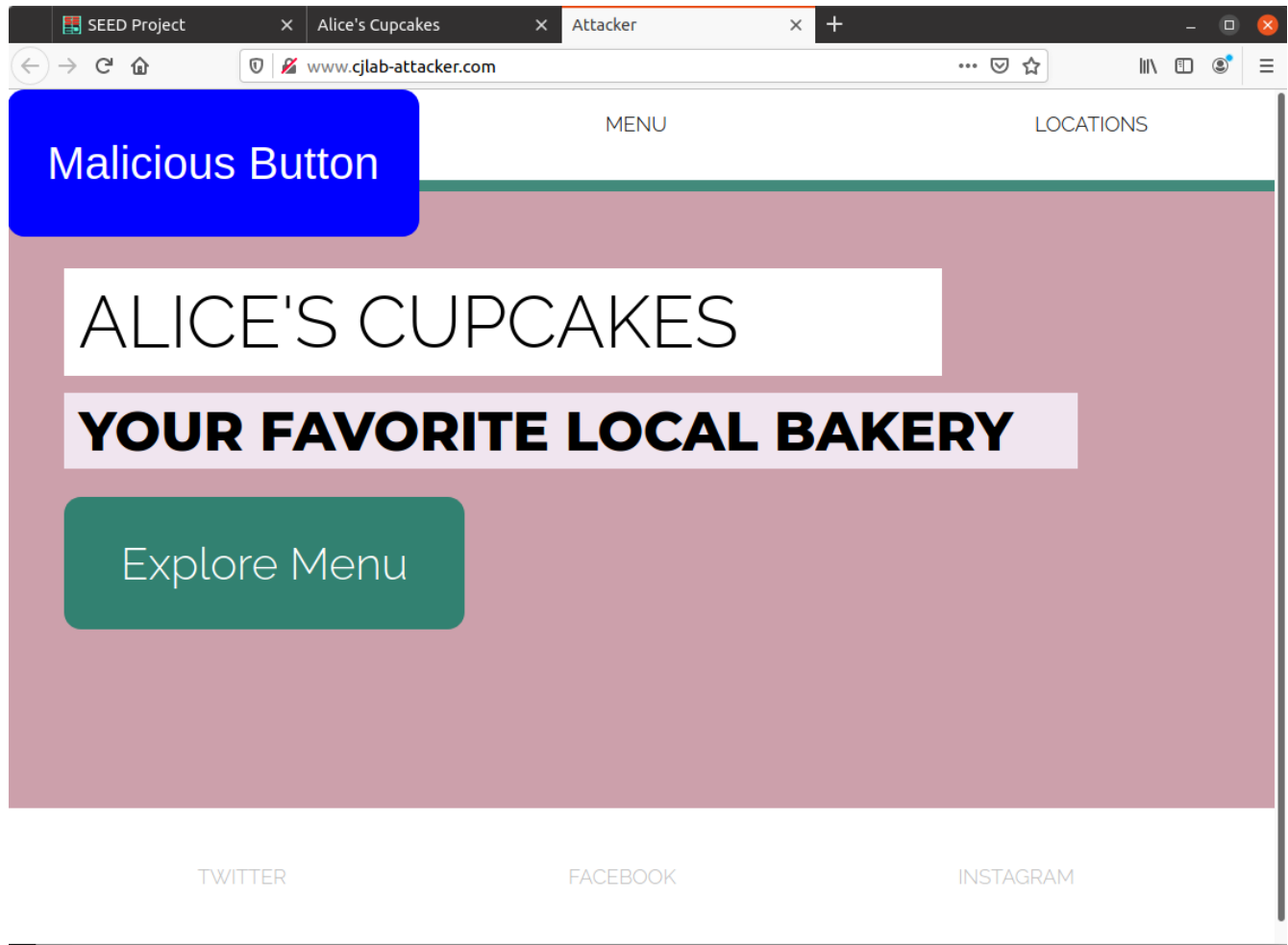


Figure 5: Attacker website after creating an iframe to the defender website

1. *With the iframe inserted, what does the attacker's website look like?*

With the iframe inserted, the attacker's site looks like the defender's webpage, pulling in all sections and text including the headers and footers, as well as the buttons and their destinations.

Task 3.2

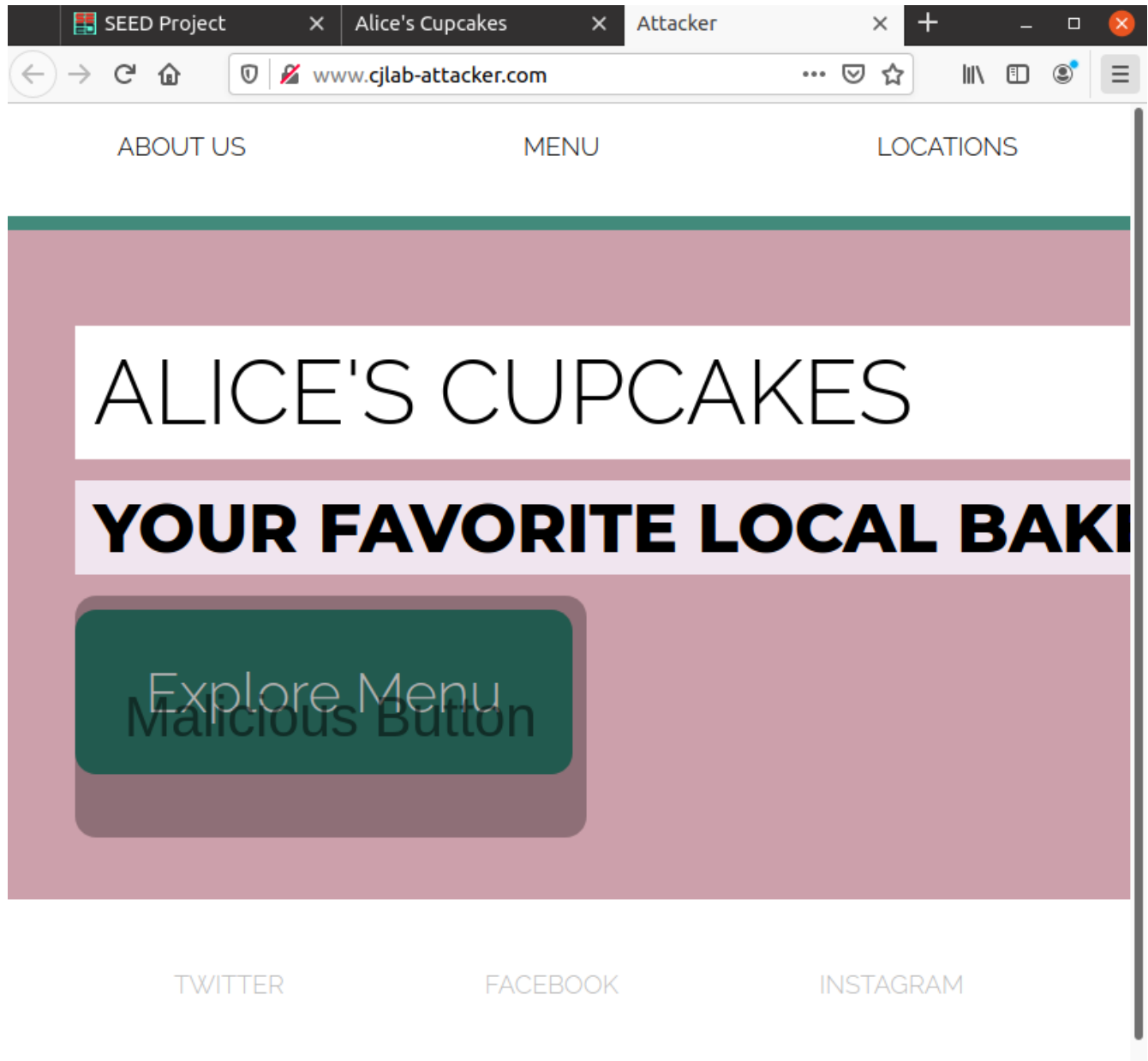


Figure 6: Modified attacker website to make the malicious button invisible (shown semitransparent to show position).

2. *How does the appearance of the attacker's site compare to that of the defender's site?*

With the attacker's webpage pulling in the exact contents of the defender's page, and including a fully transparent button in front of the defender's button location, the webpages are identical, but the functionality when clicking in the area of the "Explore Menu" button is compromised by the clickjacking attack on the attacker's webpage.

3. *What happens when you click on the "Explore Menu" button on the attacker's site?*

When you click the “explore menu” button on the attacker’s site, you are actually clicking on the invisible button that takes you to the attacker webpage that displays you have been hacked.

4. *Describe an attack scenario in which the style of clickjacking implemented for this Task leads to undesirable consequences for a victim user.*

By emulating a legitimate webpage, an attacker can trick a user into navigating to a webpage that the attacker controls. Or they may be able to steal the victims credentials for example navigating them to a login page or using click jacked forms that go to the attacker instead of the legitimate page, or even to steal session or other details by clicking on the attacker’s controlled portions of the page such as clicking something which runs a script.

Task 3.3

5. *What happens when you navigate to the attacker’s site now?*

The iframe contents load, then the malicious button loads, then the iframe contents redirect the webpage to the defender’s website, visible in the URL bar, subverting the attacker’s clickjacking attempt by taking the user to the defender’s actual website unchanged. The same as Figure 3.

6. *What happens when you click the button?*

The malicious button is no longer in front of the real button since we are at the defender’s domain, so clicking “explore menu” brings you to the index of the defender’s website.

Task 3.4

7. *What does the sandbox attribute do? Why does this prevent the frame buster from working?*

The sandbox attribute restricts what the contents of the iframe can load and do, such as scripts, form submission, and other features of the browser and html/js. Since we wrote a script tag function to bring the frame to the top, the sandbox attribute prevents the script from running and the function does not run.

8. *What happens when you navigate to the attacker’s site after updating the iframe to use the sandbox attribute?*

After updating the iframe to use sandbox, the attacker’s site loads like before, with the defender’s webpage in an iframe and the malicious button in front of the “explore menu” button. The same as Figure 6.

9. *What happens when you click the button on the attacker’s site?*

If you click the malicious button in front of “explore more”, the hacked webpage shows up.

Task 3.5

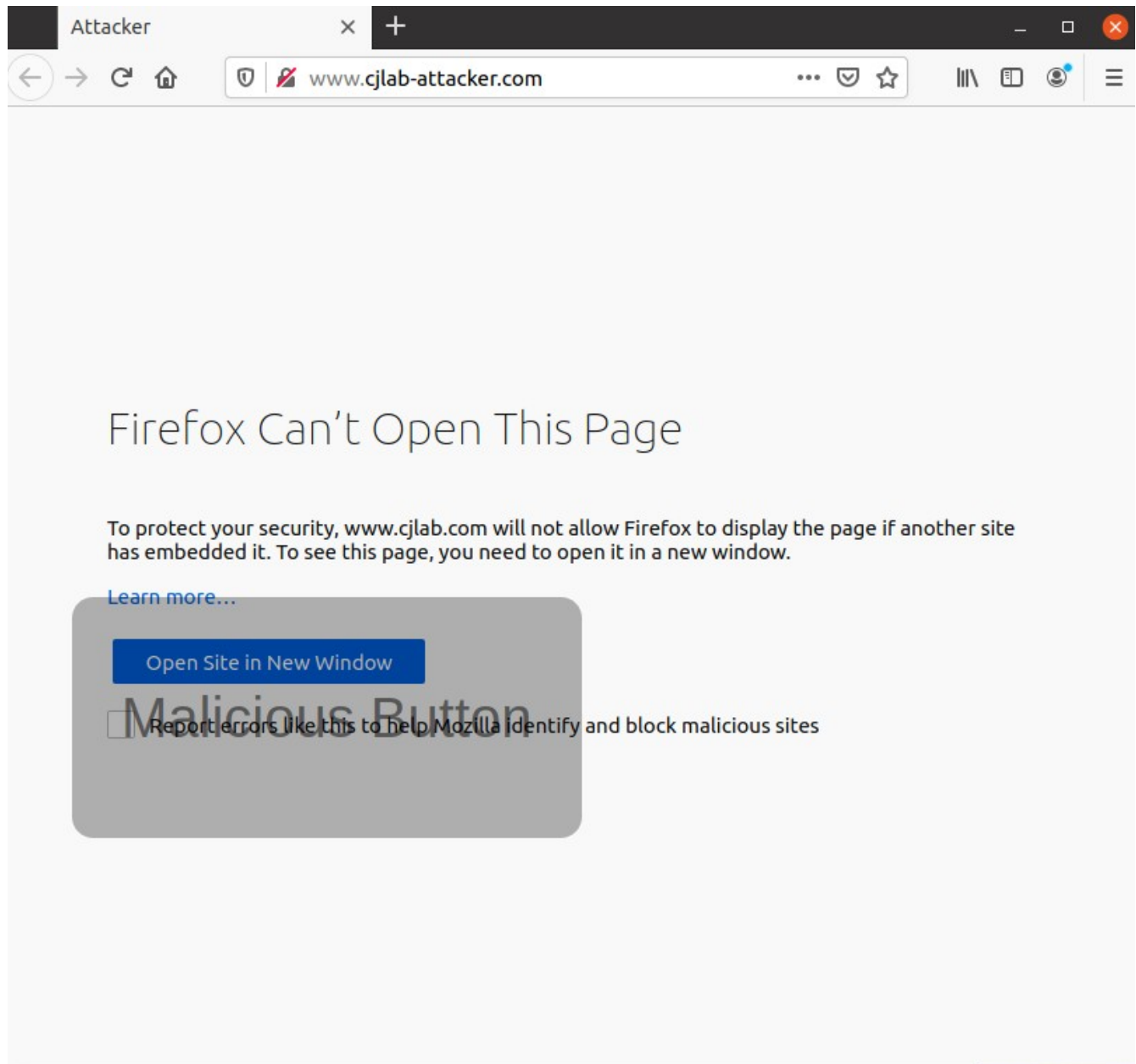


Figure 7: Visiting attacker webpage after setting defender apache options

10. What is the *X-Frame-Options* HTTP header attribute, and why is it set to “DENY” to prevent the attack?

Setting *X-Frame-Options* defines how the webpage is able to be used in frame, iframe, embed, or object tags, so by setting it to “DENY”, the contents of the iframe on the attacker’s site refuse to load.

11. What is the *Content-Security-Policy* header attribute, and why is it set to “frame-ancestors ‘none’ ” to prevent the attack?

Similar to X-Frame-Options, frame-ancestors defines if the page is allowed to be embedded in a nested context like using iframes. Setting this option to 'none' prevents the page from being embedded on another webpage like in the attacker's webpage iframe.

12. *What happens when you navigate to the attacker's site after modifying each response header (one at a time)? What do you see when you click the button?*

Running each option separately, they appear to do the same thing. The iframe that the attacker webpage is trying to use shows a message saying it couldn't load content from defender's domain, and says the defender domain doesn't allow it to be embedded. The malicious button is still present and when I click it, the page redirects to the hacked page on the attacker's site.

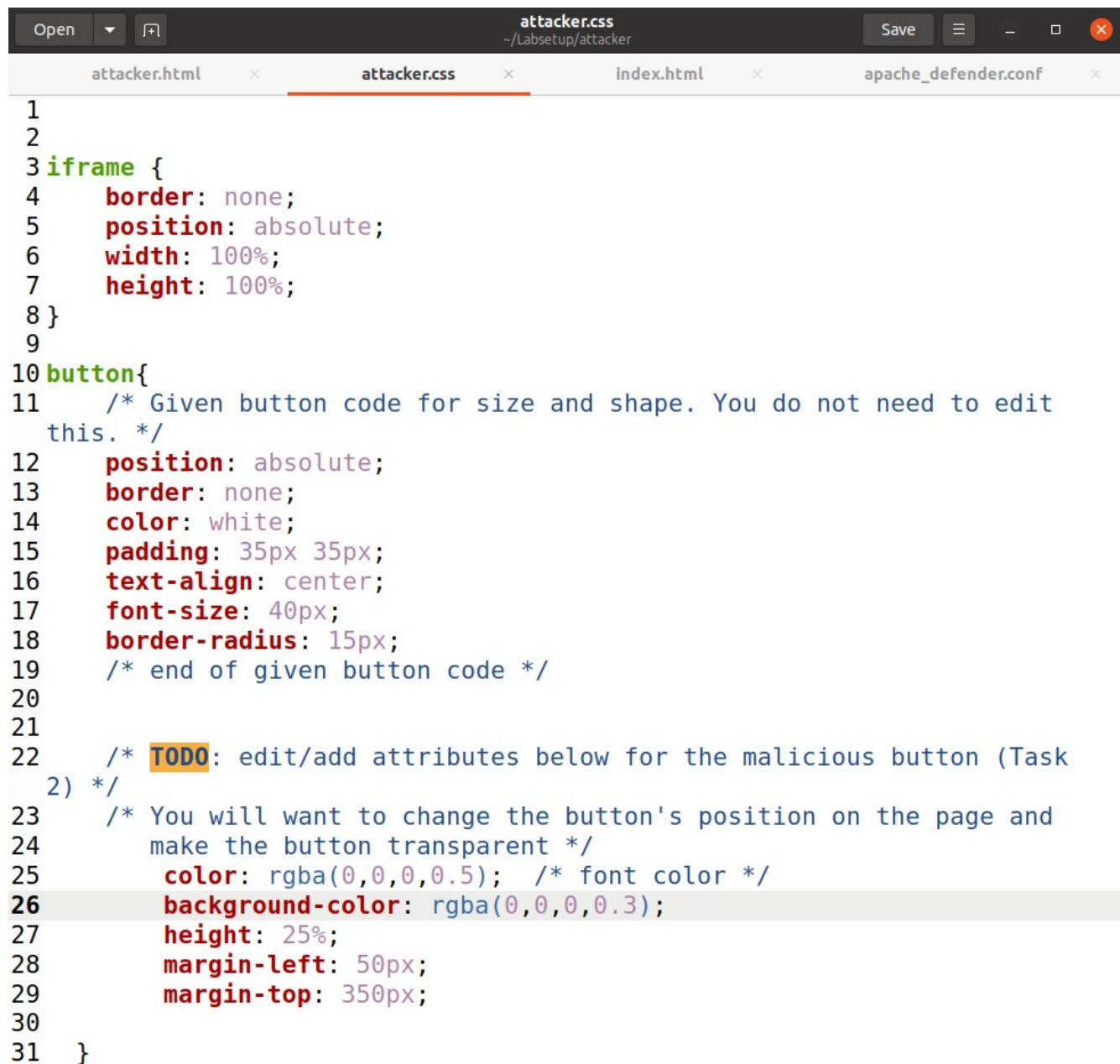
Sources

MozDevNet. (n.d.-a). Content security policy (CSP) - http: MDN. Retrieved from <https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/CSP>

MozDevNet. (n.d.-b). X-Frame-Options - http: MDN. Retrieved from <https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/X-Frame-Options>

Appendix

Code Files



```
1
2
3 iframe {
4     border: none;
5     position: absolute;
6     width: 100%;
7     height: 100%;
8 }
9
10 button{
11     /* Given button code for size and shape. You do not need to edit
12     this. */
13     position: absolute;
14     border: none;
15     color: white;
16     padding: 35px 35px;
17     text-align: center;
18     font-size: 40px;
19     border-radius: 15px;
20     /* end of given button code */
21
22     /* TODO: edit/add attributes below for the malicious button (Task
23     2) */
24     /* You will want to change the button's position on the page and
25     make the button transparent */
26     color: rgba(0,0,0,0.5); /* font color */
27     background-color: rgba(0,0,0,0.3);
28     height: 25%;
29     margin-left: 50px;
30     margin-top: 350px;
31 }
```

Figure 8: Attacker.css transparent button (0,0,0,0 is fully transparent, I left alpha at 0.3 to show the overlapping malicious button in front of the real button)

```
attacker.html
~/Labsetup/attacker

1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <title>Attacker</title>
5     <meta charset="utf-8"/>
6     <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/-
bootstrap/4.4.1/css/bootstrap.min.css" integrity="sha384-
Vkoo8x4CGs03+Hhvx8T/Q5PaXtkKtu6ug5T0eNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
crossorigin="anonymous">
7     <link href="attacker.css" type="text/css" rel="stylesheet"/>
8   </head>
9
10  <body>
11    <!-- TODO: place your iframe HERE (Task 1) -->
12    <iframe src="http://www.cjlab.com"></iframe>
13
14    <!-- The malicious button's html code has already been provided
for you.
15         Note that the button code must come after iframe code-->
16    <button onclick="window.location.href =
'hacked.html';">Malicious Button</button>
17  </body>
18
19
20
21 </html>
```

Figure 9: iframe embedding defender content into attacker website for Task 3.1



```
18
19
20     <div class="Intro">
21         <div class="Intro-container">
22             <div class="name">Alice's Cupcakes</div>
23             <div class="slogan">Your Favorite Local Bakery</div>
24
25             <!-- Here is the benign attacker-targeted button -->
26             <button onclick="window.location.href =
'index.html';">Explore Menu</button>
27
28         </div>
29     </div>
30
31
32     <footer>
33         <div class="footer-contents">
34             <div class="col">Twitter</div>
35             <div class="col">Facebook</div>
36             <div class="col">Instagram</div>
37             <div class="col">Snapchat</div>
38         </div>
39     </footer >
40
41
42 </body>
43
44 <!-- Frame Busting script to prevent clickjacking -->
45 <script>
46     window.onload = function() {
47         makeThisFrameOnTop();
48     };
49
50     function makeThisFrameOnTop() {
51         // TODO: write a frame-busting function according to
52         // instructions (Task 3)
53         if (window.top !== window.self) {
54             //bad
55             window.top.location = window.location
56         } else {
57             //good
58
59         }
60
61     </script>
62
63 </html>
```

Figure 10: Writing the makeThisFrameOnTop function in Task 3.3

```
1 <!DOCTYPE html>
2 <html lang="en">
3   <head>
4     <title>Attacker</title>
5     <meta charset="utf-8"/>
6     <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/-
bootstrap/4.4.1/css/bootstrap.min.css" integrity="sha384-
Vkoo8x4CGs03+Hh xv8T/Q5PaXtkKtu6ug5T0eNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
crossorigin="anonymous">
7     <link href="attacker.css" type="text/css" rel="stylesheet"/>
8   </head>
9
10  <body>
11    <!-- TODO: place your iframe HERE (Task 1) -->
12    <iframe src="http://www.cjlab.com" sandbox></iframe>
13
14    <!-- The malicious button's html code has already been provided
for you.
15         Note that the button code must come after iframe code-->
16    <button onclick="window.location.href =
'hacked.html';">Malicious Button</button>
17  </body>
18
19
20
21 </html>
```

Figure 11: Updated iframe code using the sandbox attribute in Task 3.4