Chapter 34. Defense Against Client-Side Attacks

In Chapter 16, we covered three forms of client-side attack (prototype pollution, clickjacking, and tabnabbing) extensively. In each of these attacks, we were able to exploit users via their *client* browsers rather than having to go through a server in order to attack the end user. Client-side attacks are on the rise as browser clients become more sophisticated each and every year.

Because of the limitations that come when attempting to detect client-side attacks, it is imperative to understand how to set up your application in a way that makes client-side attacks difficult for an attacker to pull off successfully. In this chapter, we will cover those techniques so your web application can be as secure as possible when targeted with client-side attacks.

Defending Against Prototype Pollution

Prototype pollution attacks rely on JavaScript's prototypal inheritance system in order to function (see Chapter 16 for detailed attacks and payload development). Because of the way a JavaScript interpreter walks up the prototype chain looking for functions and data, it's very possible for an attacker to *pollute* one object without direct access (by polluting a related object in the inheritance hierarchy).

At first, prototype pollution attacks are difficult to find and mitigate due to their reliance on largely JavaScript-specific language features, but once you get the hang of it, mitigations become significantly easier to understand. Let's look at several of the most effective mitigations for stopping prototype pollution attacks.

Key Sanitization

One of the easiest ways to prevent prototype pollution attacks is to ensure your application's JavaScript code sanitizes all user-provided keys prior to merging or manipulating JavaScript objects with those keys.

Consider the following code snippet. This snippet of JavaScript code represents an input that would be generated in an application on behalf of a malicious user's input:

```
// submitted via address collection form
const obj = {
  "__proto__": { "role": "admin" }
}
```

In this case, the key `__proto__` is the part of the object that exploits the prototype chain when merged into another object.

If we provide an allowlist of available keys, we can easily iterate through user inputs and prevent attacks via simple sanitization:

```
const allowedKeys = ["street", "city", "state", "firstName", "lastName"];

const isKeyValid = function(key) {
  if (!allowedKeys.includes(key) {
    return false;
  } else {
    return true;
  }
};

const updateUserData(data) {
 let isValid = true;

 for (const [key, value] of Object.entries(data)) {
  if (!isKeyValid(key)) {
   isValid = false;
  }
 }
```

```
  if (!isValid) {
   console.error("an invalid key was found!");
  } else {
   updateUserData(data);
  }
 };
```

Allowlisting and validating object keys is not always an acceptable mitigation, as some applications need to accept flexible user input and can't rely on a static set of allowed keys. However, where it is possible, this is one of the best first-line solutions for defending against prototype pollution attacks.

Prototype Freezing

Another solution for mitigating prototype pollution attacks is that of JavaScript's built-in `Object.freeze()` function. `Object.freeze()`, aka the "freeze function", modifies the default nature of JavaScript objects, making them immutable rather than mutable.

A frozen object cannot have any keys, values, or other properties updated for the remainder of the browsing session. Once the tab is closed and reopened, the object becomes *mutable* once again until `Object.freeze()` is called again. This means that calling the freeze function against an object named `userData` with the function call `Object.freeze(userData)` will prevent `userData` from ever being updated until the page is refreshed.

Freezing objects is an excellent mitigation against prototype pollution attacks, but the major limitation is that a frozen object cannot be modified for either malicious or nonmalicious purposes. This means that the level of functionality an object has when interacting with legitimate first-party code is significantly diminished, which can make developing complex applications much harder for engineers.

One other caveat to be aware of is that bulk freezing JavaScript or DOM APIs, while sounding like a good idea in theory, has significant consequences. Dozens of DOM APIs and JavaScript APIs are written depending on the mutability of other objects and functions. Because of this, entire

swaths of functionality will break if you attempt to perform a bulk freeze or accidentally freeze built-in APIs while performing an iterative (looped) freeze call down a prototype chain.

For these reasons, while the freeze function is powerful when it comes to stopping prototype pollution attacks, it should be used sparingly in order to prevent unintended application functionality loss.

Null Prototypes

As a final mitigation against prototype pollution attacks, you can cut off an object's prototype chain during instantiation if you fear it may be subject to prototype pollution attacks later in the application life cycle.

By utilizing the *object constructor* directly, versus creating an object manually, you can specify a prototype to inherit from as a parameter. Consider the following code snippet:

```
// traditional object creation
const myObj1 = { username: "testUser1" }

// manual object constructor invocation inheriting from null
const myObj2 = Object.create(null);
myObj2["username"] = "testUser2";
Object.getPrototypeOf(myObj2); // null
```

Although any object can be passed through the object constructor in order to perform initial prototype setup, passing `null` creates the object with no prototype. Because the object is created with no prototype, it cannot walk up the prototype chain and cannot be polluted via parent objects because no parent objects exist.

Defending Against Clickjacking

There are a multitude of defenses against clickjacking but only two are highly effective. One is implemented at the CSP level, and the other is implemented in JavaScript.

You may see legacy defenses if you read through web documentation on defending against clickjacking attacks—specifically defenses making use of headers like `X-Frame-Options`. Note that these headers are considered obsolete by most major browsers and should not be considered an effective line of defense.

Frame Ancestors

The easiest and most effective way to stop the majority of clickjacking attacks is by implementing a simple change in a website's CSP policy. (See Chapter 22, "Secure Application Configuration" for more details on implementing CSP policies.)

The *frame ancestors* directive can be implemented in either a CSP header or meta tag, similarly to all other major CSP directives. This directive takes the following structure regardless of where it is implemented:

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

In this case, a web application sets `frame-ancestors` CSP directive to `none`, which tells the browser to prevent *any* web page from loading it inside of an iframe. Because over 95% of modern web browsers support this directive, implementing this simple solution will prevent your website from being loaded in an iframe—stopping almost all clickjacking attempts.

If your web application has a use case for being framed elsewhere, you can loosen the policy and specify the locations it can be framed in an allowlist fashion:

```
Content-Security-Policy: frame-ancestors subdomain.my-website.com
```

The option `self` can also be used in the rare case your web application needs to contain a copy of *itself* within an iframe.

By implementing either of these solutions, you prevent malicious websites from framing your website within an iframe, which will stop the ma-

jority of clickjacking attacks.

Framebusting

In the case that your web application is supported on one of the few browsers that does not support CSP `frame-ancestors`—or if you have a rare use case for framing that prevents CSP from being an acceptable solution—there is an alternative mitigation. Framebuster scripts (sometimes also called *framekillers*) are JavaScript functions you can include in your page to detect if your website is being framed within an iframe somewhere you would not permit.

Framebuster scripts can stop your application code from loading in the browser or unload it rapidly if an iframe is detected. A simple framebuster script looks like the following:

```
html {
  display:none;
}
```

```
if (self == top) {
    document.documentElement.style.display = 'block';
}
```

Using this framebuster script, your website is set to `display: none` in its CSS code. This CSS code should be contained in the `head` block to ensure it loads prior to the `body` of the page. The JavaScript code should be imported into the `body` of the page so it loads after the CSS.

When an application with both of these snippets loads into a web browser, its content will have `display: none` set, which both renders the content invisible and prevents user interaction. By preventing user interaction, clickjacked clicks will not be able to interact with the page.

When the JavaScript framebuster loads, it will check if it is the top-most window by comparing the built-in objects `self` and `top`. If it is the top-most window, then it will convert its display mode to `block`, which will

cause HTML content to become visible and interactive. If it is not the topmost content, its scripts will continue to execute but without rendering any interactive or visible content on the page.

You will probably see references to older forms of framebuster code, which allow the page to load *before* initiating a framebuster script. These legacy solutions are not considered ideal; there will still be a window of time where clicks can be clickjacked, and smart attackers can halt or delay script execution in order to prevent the legacy framebuster code from executing as intended.

Defending Against Tabnabbing

Tabnabbing attacks are easy to prevent with proper planning as you build your web application, due to the many built-in browser mitigations that exist. Because these solutions are relatively simple to put in place, they should be implemented by default on any new application. Unfortunately, many applications forgo these steps and are therefore vulnerable to tabnabbing.

Cross-Origin-Opener Policy

As noted in Chapter 22, CSP now supports a policy called Cross-Origin-Opener Policy (COOP). This policy determines which websites are given access to a *window* object reference when opened in a hyperlink.

By default, COOP should always be set to the following, which will prevent any opened links from being able to reference the website that they originated from:

```
Cross-Origin-Opener-Policy: same-origin
```

For the majority of tabnabbing cases, this should be the first-line solution. CSP COOP only becomes an issue in the case of large websites that make use of multiple domains. In which case, a more relaxed CSP COOP in combination with one of the following techniques should be used.

Link Blockers

Because hyperlinks by default allow references to the opening web page (for the *opened* web page to access), permitting users to generate links is risky if proper precautions are not taken. Fortunately, a simple HTML attribute that is now supported on all major browsers can switch the reference to `window.opener` provided to the target website to `null`:

```
<a href="malicious-website.com rel="noopener">click me</a>
```

The `noopener` attribute seen in the previous code snippet blocks the window reference of the newly opened page by setting it equal to null.

While using `noopener` on dynamically generated links, you should also include its sibling attribute `noreferrer`, which will block the target website from accessing *referrer* information in order to determine where its traffic came from.

Both of these policies are implemented in the following snippet:

```
<a href="malicious-website.com rel="noopener noreferrer">click me</a>
```

By putting both of these policies in place, both your application's security and privacy are greatly improved, and tabnabbing via dynamic links becomes impossible.

Isolation Policies

A multitude of client-side attacks can be mitigated with a brand-new browser feature called *fetch metadata*, which supports flexible *isolation policies*. Isolation policies are a very new and powerful browser security feature that is currently only available on the most recent builds of Firefox and Chrome web browsers. Isolation policies are not yet available on the desktop or iOS versions of Safari, nor on the majority of international browsers. As such, isolation policies should currently be consid-

ered *defense in depth*, that is, a defensive mechanism that is always implemented alongside other mitigations.

As part of the fetch metadata feature, Chrome and Firefox (and in the future, all web browsers) send back headers with every request called `Sec-Fetch-Site`, `Sec-Fetch-Mode`, `Sec-Fetch-Dest`, and `Sec-Fetch-User`, which provide the server valuable data regarding how an application is being requested and where it will be loading.

The header `Sec-Fetch-Site` indicates to the server where the website is being requested. The following are valid options:

`same-origin`

: The request is being made from its own origin.

`same-site`

: The request is being made from a subdomain of the application.

`cross-site`

: The request is being made from a different site.

`none`

: The request is not being made from a website (e.g., bookmark or plug-in).

The header `Sec-Fetch-Mode` provides information on the mode by which the website is being requested. The following options are possible:

`same-origin`

: The browser is making a request to the same origin.

`no-cors`

: The browser is making a request to another origin but doesn't expect to read the response.

`cors`

The browser is making a request that utilizes CORS (see Chapter 22 for more info on CORS).

`navigate`

The request has been initiated by a link click, bookmark, or redirect.

The `Sec-Fetch-Dest` header tells the server where the content will be loaded. The following values are permitted:

- `audio`
- `audioworklet`
- `embed`
- `font`
- `frame`
- `manifest`
- `object`
- `paintworklet`
- `report`
- `script`
- `serviceoworker`
- `sharedworker`
- `style`
- `track`
- `video`
- `xslt`

And finally, the `Sec-Fetch-User` header has either the value of `null` or `1`, with `1` indicating if the browser believes the user initiated the request. `null` indicates that the browser believes a script or plug-in initiated the request.

By combining the results of these headers on the server side, you can create powerful mitigations for a variety of client-side attacks. As an example, the following server-side mitigation is implemented to prevent a website from being framed, hence blocking most clickjacking attacks:

```
app.get('/index.html', function(req, res, next) {
  if (req.headers["Sec-Fetch-Dest"]) {
    const dest = req.headers["Sec-Fetch-Dest"];
    if (dest === "frame") {
      return res.sendStatus(400);
    } else {
      return res.sendFile("/index.html"));
    }
  }
});
```

Summary

Although client-side attacks are becoming more and more common, mitigations against these attacks are also becoming more advanced at a rapid pace. Fortunately, most client-side attacks are stopped at least in part by the browser, using mitigations that can be implemented with simple configuration changes.

Learning about all forms of client-side attacks and ensuring the appropriate mitigations are enabled will dramatically improve the security posture of your web applications with only a minimal amount of time and effort. Because of this, learning client-side attack mitigations is one of the most time-saving and cost-effective methods for improving your application's security posture.