Chapter 16. Client-Side Attacks

In the premodern world of application security (Web 1.0), it was assumed that the client (aka browser) component of a web application was not a common attack vector for hackers. As a result, companies assumed the majority of the application's risk surface area was on the back-end (server side) and invested very little in ensuring that their browser clients were secure.

As Web 2.0 rolled around, more and more functionality that was previously found only on the server would be pushed to the client. Complex computing operations would be rewritten from backend Java or C, into client-side JavaScript. Backend data stores would be replaced with local storage, session storage, or IndexedDB.

Asynchronous JavaScript and XML (AJAX)–type network queries would allow for the development of client-side applications that maintain, update, and store state. Client-side improvements to the JavaScript programming language and browser DOM would allow complex component life cycles with updates, renders, re-renders, deletes, and so on—similar to those of a desktop application.

All in all, the overall architecture of a web application would change from a model in which the server was responsible for all computing operations and the client was only a rendering (view) layer, to a model in which both the server and client were responsible for a variety of complex computing tasks.

The modern web reflects this change: all of the largest websites in the world make use of both complex server- and client-side functionality. The server is no longer the premier mechanism for complex computing operations and as such, security professionals need to rapidly update their skill sets to be able to assist in delivering both secure server-side software AND secure client-side software.

This chapter covers common methods of exploiting a web application by making use of the client as the target rather than the server.

Methods of Attacking a Browser Client

First, when thinking about client-side attacks, start by eliminating the server. A client-side attack is any form of attack (vulnerability that can be exploited) that doesn’t require a vulnerable web server or network calls to a client’s server. Categorically there are two main ways of attacking a browser client: client-targeted attacks and client-specific attacks.

Client-Targeted Attacks

First, there are general forms of vulnerability that can affect either a client or a server. An example of this is the regular expression denial of service (ReDoS) vulnerability we evaluated in a prior chapter.

Regular expressions are a common programming tool that are implemented in almost all major server-side languages (e.g., Java, C#, Python) but also exist in the browser (JavaScript) and other less common clients (e.g., Adobe Air). ReDoS attacks are not always a client-side attack, but occasionally may be a form of client-side attack when the JavaScript code on a browser client is structured so that the client is vulnerable without any network calls or server interaction being required.

Client-Specific Attacks

Next, there are client-side attacks that exist solely on the client and will likely never appear on the server (unless the server is attempting to emulate a web browser). We covered DOM-based cross-site scripting (DOM XSS) in Chapter 10. This is a prime example of a client-side attack that only works against browser clients.

In DOM XSS, unlike stored or reflected XSS, both the sink and the source occur in the browser. For example, a string from `window.location.hash` is rendered into code via `eval()`, resulting in client-side script execution.

For these reasons, DOM XSS is a poster-child for client-side attacks. Like DOM XSS and ReDoS, the additional attacks we discuss in this chapter are capable of occurring entirely within a browser client. However, before digging deep into particular attacks, let's take a look at why client-side attacks are important.

Advantages of Client-Side Attacks

In the case of a DOM XSS attack, both the sink and the source exist solely in the browser DOM. This means that DOM XSS (as well as many other client-side attacks) is capable of being exploited without any web server ever becoming aware that exploitation is occurring.

Payloads can be delivered directly to a browser client, avoiding web servers that may be logging network traffic and attempting to find malformed requests. For these reasons, client-side attacks are often categorically one of the most difficult types of exploit for a mature corporation to detect.

As an ethical hacker, this means it may be possible to exploit users without ever being traced or detected. As a bug bounty hunter, this means that vulnerable surface area a hacker would take advantage of on the client is less likely to have been found and remediated, leading to a higher probability of discovery.

In addition to the lack of required server networking allowing client-side attacks to slip by unnoticed, client-side attacks can also be easier for a malicious user to develop without identification. Consider the case in which a hacker is attempting to develop exploits against a web server. It is unlikely in the case of a complex web application that the first payload developed will successfully infiltrate and bypass all existing security mechanisms. As such, multiple attempts to deliver a payload are often required.

In fact, the majority of successful hackers automate these attempts to save time. They often send dozens of payloads per hour until one sticks. All of these network requests increase the probability of detection, either by a human, firewall, or network scanning tool.

With client-side attacks, it's easy to download the entire HTML/CSS/JS client-side web application from a business and then turn off the network. From that point forward, millions of attempts to attack the local client application could be performed, with none of them sending any data back to their production web servers.

Prototype Pollution Attacks

In recent years, with the rise of npm and other JavaScript package managers, prototype pollution attacks have also been on the rise. Prototype pollution is a form of attack that only works against languages that make use of prototypal inheritance systems (e.g., JavaScript), a form of inheritance that differs from the traditional OOP inheritance found in Java, C#, or other popular languages. Prototype pollution attacks allow you to compromise an *object you do not have access to*, via compromising an object you *do have access to*, that shares a prototypal inheritance relationship with the object you want to attack.

Understanding Prototype Pollution

Consider the following code snippets, written in client-side JavaScript:

```
const Technician = function(name, birthdate, paymentId) {
    this.name = name;
    this.birthdate = birthdate;
    this.paymentId = paymentId; // for paying for jobs
}
```

The preceding function is what is known as a pseudo-class or class-like structure implemented in a prototypal programming language. Using this pseudo-class, you can instance an object that is derived from it as a sort of blueprint, and the object will inherit the state and functions of the parent pseudo-class. Here's an example:

```
const Bob = new Technician("Bob", "12/01/1970", 12345);
console.log(Bob.toString()); // [object Object]
```

The way in which JavaScript and other prototypal inheritance–based languages store this data is inside of what's known as a *prototype chain*. Every object has its own *prototype* in JavaScript, which contains references to all of its ancestor objects that it inherits functions and data from.

We can verify that Bob is, in fact, a technician by comparing the prototype `Bob` with the prototype of `Technician`. Do note that to access Bob's prototype information, we'll use `Bob.__proto__` in this situation and compare it to `Technician.prototype`. This is because `__proto__` points to an actual prototype object, whereas `prototype` points to a blueprint for building more prototypes. Because `Technician` was the constructor function for building the instance `Bob`, when the two are compared using an equality operator, it will return `true`:

```
Bob.__proto__ == Technician.prototype; // true
```

Beyond verifying that Bob is a technician, we can also verify that `Bob` is an object in one of two ways. First, knowing that `Bob` inherits from `Technician`, we can walk our way up the inheritance chain manually since `Technician` inherits from no other custom object:

```
Bob.__proto__.__proto__ == Object.prototype; // true
```

Next, we can use the `instanceof` operator to again confirm that `Bob` is indeed an instance of `Object` via being an instance of `Technician`, which inherits from `Object` as do all JavaScript functions:

```
Bob instanceof Object; // true
```

We could also use the `instanceof` operator to confirm `Bob` is an instance of technician. Consider this operator a short cut rather than having to climb the prototype chain manually:

```
Bob instanceof Technician; // true
```

With this new knowledge in mind, we can conclude that the inheritance hierarchy for this application is as follows:

```
(Object) -> (Technician) -> (Bob)
```

One final thing to note before being able to deploy prototype pollution attacks to compromise applications built on JavaScript prototypes is how information propagates through the prototype chain.

Previously we called a function `toString()` against the object `Bob`. When we did this, it returned a string `[object Object]` despite no function being defined on `Bob` that returns this string. This is because `Object` contains a `toString()` function, and when a function is called but not found on the current object, the interpreter walks up each layer of the prototype chain until it finds a function with the same name.

If no function is called during this prototype walk, an error will be thrown. However, if a parent class contains an appropriately named function, it will be called in lieu of a function on the current class.

In other words, the `toString()` function does not exist on `Bob`. Because the function does not exist on `Bob`, the interpreter will go up the prototype chain stopping at `Technician` and finally `Object`, where it eventually finds an appropriately named `toString()` function. This process is outlined in Table 16-1.

Table 16-1. Prototype chain

Step number	Function called	Class evaluated	Found?
1	`toString()`	`Bob`	False
2	`toString()`	`Technician`	False
3	`toString()`	`Object`	True

In this case, we could perform prototype pollution against the `Bob` class without even having access to `Bob`. If we can find a function that modifies the prototype of either `Technician` or `Object`, we can change the functionality of the `toString()` function in such a way that `Bob` is also impacted.

Consider the following prototype pollution payload, which works against the `Bob` class despite not targeting it directly:

```
// adds functionality to technician class
const addTechnicianFunctionality = function(obj) {
  Technician.prototype[obj.name] = obj.data
}

// user input payload
{
  name: "toString",
  data: `function() { console.log("polluted!"); }`
}

Bob.toString(); // prints "polluted!"
```

In this case, the UI anticipated that `Technician` would be an adjustable class for client-side state management purposes, but it did not anticipate that `Bob` would be adjustable. By changing the `toString` function on `Technician`'s prototype, `Bob`'s `toString` function was also polluted.

Prototype pollution attacks take advantage of the fact that prototypal inheritance systems walk up and down the prototype chain as previously noted. This allows an attacker to pollute a single object, which will than effectively “spread” to nearby related objects that might not be directly targetable.

Attacking with Prototype Pollution

Previously, we discussed how prototypes and prototype chain traversals work. We are now armed with the knowledge that if a function or property does not exist in the current object, it will walk up the chain until it finds a function or property with an identical name.

When full script execution (XSS) is not available on the client, it’s likely there will be many cases where prototypes merge and update. These are our attack surface areas for exploiting prototype pollution attacks.

Consider the npm open source JavaScript package `merge v2.0`. This package is widely noted on the web to be vulnerable to prototype pollution attacks, despite its simple syntax. The *merge* library has a function `merge()` that simply combines two objects together. It’s frequently used while modeling state on either a Node.js server or a JavaScript-based client.

Let’s try to pollute the `Bob` object again, in order to add a new property `isAdmin: true`. Suppose we find a code snippet in the JavaScript client-side code that combines `Object` and `userData` using the `merge()` function. Using a payload of `{ isAdmin: true }` against `Object` will result in `Object` obtaining an `{ isAdmin: true }` property, but it will not be reflected on `Bob` because the `Object` prototype has not yet been updated.

We can see the result of this attack attempt with the following payload:

```
merge(Object, { isAdmin: true });

console.log(Bob.isAdmin); // undefined
```

However, when we attach this payload targeting the prototype we see a different result:

```
merge(Object, { "__proto__.isAdmin": true });

console.log(Bob.isAdmin); // true
```

Here we have successfully polluted the `Object` prototype in a way that makes `Bob` vulnerable as well.

Do note that this particular library is also vulnerable to *constructor pollution*. Rather than polluting the `Object` prototype directly, we can pollute the `constructor` function, which is automatically attached by the JavaScript language to `Object` and used whenever an instance of `Object` is created:

```
merge(Object, { "constructor.prototype.isAdmin": true });

console.log(Bob.isAdmin); // true
```

In this case, every instance that inherits from `Object` will call `constructor`, leading to pollution that has the same net result as directly polluting the `Object` prototype.

Prototype Pollution Archetypes

After obtaining prototype pollution against a web application, there are several things you can do with it in order to obtain information or interfere with intended client-side execution.

Denial of service

Prototype pollution attacks can be used to slow down or interfere with normal client-side script execution. For example, changing a value to be a `float` rather than an `integer`. This will cause bugs later down the line and interfere with the intended use case of the client application.

Property injection

If a script relies on a particular value for a function call, that value can be modified by prototype pollution. This could result in unintended calls against a network or functionality being invoked on the client in a way not expected by the end user.

Remote code execution

Generally speaking, this is the worst-case scenario with prototype pollution. In the client-side realm of prototype pollution, this upgrades the attack to XSS; in the Node.js (server-side) realm, an attack can be upgraded to true server-side code execution. Either of these outcomes results in compromised application state, data, and functionality. Typically, upgrading prototype pollution to code execution requires a script execution sink like `eval()` or a DOM node generation function like `DOMParser.parseFromString()`.

Clickjacking Attacks

Clickjacking attacks are subtle but impactful attacks that occur against an end user within the browser. Clickjacking attacks merge malicious UI elements with nonmalicious UI elements, or transparently trick the browser into sending input to a malicious server or function call rather than an intended function call.

There are many methods of attacking an application using clickjacking, with implementations involving JavaScript, HTML, and CSS, either alone or with another technology. It is feasible to consider a clickjacking attack as a form of user-interface keylogger. When deployed against unsuspecting end users, clickjacking attacks can allow an attacker to siphon up valuable user input that was not intended to be read by a third party.

Camera and Microphone Exploit

One of the most well-known early examples of clickjacking is the Adobe Flash microphone and camera hijack exploit that was disclosed on the web in 2008 after its discovery by security researchers Robert Hansen and Jeremiah Grossman. This clickjacking attack appeared in the form of a set of web links that appeared to be either a game or web page entirely unrelated to Adobe Flash player.

Unbeknownst to the end user, every click within this game or web page corresponded with a click on the iframed Adobe Flash web-based settings page underneath it. The Adobe Flash settings page was loaded in within an iframe and had its opacity set to zero, so it was invisible to the end user.

When interacting with the clickjacking web page, the end user was tricked into passing clicks through to the privileged Adobe Flash privacy settings. The result was that the Adobe Flash browser plug-in would share both camera and microphone control with the hacker. This became one of the most prolific instances of clickjacking in the infosec world, as it was able to use a plug-in to escalate beyond the browser sandbox and obtain privileged access to computer hardware.

Creating Clickjacking Exploits

Modern clickjacking can be done in a number of ways. The most common method is to produce a legitimate-looking website that contains an invisible iframe underneath it pointing to a website that you want to attack.

Consider the following example website:

```
<html>
 <head>
  <title>Clickjacker</title>
 </head>

 <body>
  <div id="clickjacker">
```

```
     <span id="fake_button">click me</span>
    </div>
    <iframe id="target_website" src="target-website.com"></iframe>
  </body>
</html>
```

In this example, we are attacking *target-website.com* by instantiating it in an iframe that appears underneath the div `clickjacker`. We can use the following CSS class to make *target-website.com* invisible to the end user:

```
#target_website {
 opacity: 0;
}
```

The `clickjacker` div contains a button, which can be positioned directly over a legitimate button in the iframe by using CSS positioning:

```
#fake_button {
 position: relative;
 right: 25px;
 top: 25px;
 pointer-events: none;
 background-color: blue;
}
```

By adding the `pointer-events: none` CSS attribute to `#fake_button`, any interaction (clicks) against `#fake_button` will pass through to the element underneath it. In this case, the element underneath it exists within the iframe that the end user is not intending to click on. When `#fake_button` is clicked, the `click` event will pass through to the iframe and trigger functionality in another website.

Unfortunately, because of the browser's security model, the iframe is likely to have access to session cookies for the framed website. This means that the click could initiate privileged requests against a web server, like calling an API to set a profile public or initiating a financial transaction.

Clickjacking attacks against any web application that lacks appropriate framing controls are one of the easiest ways of tricking a user into invoking functionality on behalf of an attacker.

Tabnabbing and Reverse Tabnabbing

Tabnabbing and its sister attack *reverse tabnabbing* are a form of client-side attack that combines elements of *phishing attacks* (attacks that trick the end user into interacting with a malicious web page) and *redirect attacks* (attacks that redirect the current web page to a malicious web page).

In tabnabbing attacks, browser DOM APIs are abused in order to either redirect the current page to a new one—or overwrite the content of the current page with HTML/CSS and JS provided by a hacker.

Traditional Tabnabbing

The traditional implementation of a tabnabbing attack operates via abusing the `window` object that ships with all major web browsers and is defined as a part of the WHATWG DOM specification. When a new tab is opened via the `window.open()` function attached to the `window` object, the function call returns a reference to the `window` object that opened the new tab.

In the traditional tabnabbing approach, the website invoking the new tab is the attacker, as shown in the following code sample:

```
<button onclick="goToLegitWebsite()">click to go to legit website</button>
```

```
const goToLegitWebsite = function() {
 // open new tab pointing to legit-website.com
 const windowObj = window.open("https://website-b.com");

 // after 5 minutes, change the other tab to compromised website
 setTimeout(() => {
  windowObj.location.replace("https://website-c.com");
  }, 1000 * 60 * 5);
};
```

In this example of traditional tabnabbing, a *website A* presents the user with a link to open *website B* in another tab. The new tab opens with *website B* loaded in, but by the nature of it being opened from the `open()` function call, a reference to the `window` object bootstrapping the new tab is stored in the opening tab (*website A*).

At a later point in time, after the end user has already verified that the new tab is legitimate, the opening tab (*website A*) reaches into *website B*'s copy of the DOM API and initiates a redirect using `windowObj.location.replace()`. *Website B* has now been redirected to *website C*. *Website C* is identical in terms of appearances to *website B* (the legitimate website), but when the user tries to log back in, it simply copies their credentials, sends them to a hacker's server, and redirects to an error page on *website B*.

This attack workflow is quite complicated, but the end result is that the tab containing *website B* was temporarily replaced with *website C*—a malicious website maintained by a hacker. Because the user initially verified that the new tab contained a legitimate website, it's unlikely they would note the rapid change of content, which could have happened even while they were browsing another tab. As a result, tricking the user into entering credentials or other sensitive information into the tabnabbed tab becomes much easier than conventional phishing attacks.

This whole workflow is enabled by the fact that the browser DOM function `window.open` returns a reference to the `window` object in the new tab and allows the opening tab to make function calls against the new tab. Exploiting this vulnerability is as simple as developing a website with tabnabbing JavaScript code, developing a phishing website that matches the aesthetic and user experience of a legitimate website, and then tricking an end user into clicking the link to open a new (compromised) tab.

Reverse Tabnabbing

Reverse tabnabbing works in the opposite direction as traditional tabnabbing. Rather than the attacker being the website launching the new tab, the attacker is the website *opened in the new tab* and the attacker targets the initial tab. There are multiple methods of attacking an end user via reverse tabnabbing, each of which relies on mechanisms for a tab to perform DOM API calls against the tab that opened it.

DOM API attack

The easiest method of attacking via reverse tabnabbing is to again set up a malicious website. You then convince a legitimate website to open your malicious website using the `window.open()` function call.

When `window.open()` is invoked to create a new tab, the opener will pass by default a reference to its `window` object to the new tab. This is similar to the way in which a traditional tabnabbing attack works.

Consider the following example:

```html
<!-- UI element for legit website -->
<button onclick="openTab()>click me</button>
```

```javascript
// Script for legit website
window.open("https://malicious-website.com")
```

```javascript
// Script for malicious website
window.opener.location.replace("https://get-hacked.com")
```

As you can see, by referencing the `opener` property on the newly created tab's `window` object, the new tab can begin to control the original tab that spawned it. With control over the original tab, the new tab can now change the location of the original tab from a legitimate website to an illegitimate one—and use that new website to steal credentials and other information from the end user.

Much like traditional tabnabbing, the attack is easy to execute via developing and hosting a malicious website that makes function calls to the opener's `window.opener` property. The main drawback is that other forms of tab spawning and redirect outside of the `window.opener` property do not all produce tabs with a `window.opener` reference, so not all websites will be vulnerable to this form of attack.

HTML link attack

If the `window.opener` property is not available in a website you intend to attack with reverse tabnabbing, it is possible to perform an attack if you are able to spawn an HTML link or trick the website into spawning an HTML link on your behalf as long as that link makes use of the `target=_blank` attribute.

The `target=_blank` HTML link attribute will also force a new tab to spawn in a `window` object that contains an `opener` reference:

```
<!-- legit website, spawned in a user generated link -->
<a href="https://malicious-website.com" target="_blank">click me</a>
```

```
// malicious website script
window.opener.location.replace("https://get-hacked.com")
```

Iframe attack

Finally, if neither of the two aforementioned reverse tabnabbing attack vectors are possible, you can perform reverse tabnabbing inside of an iframe. If the invoked iframe links to your malicious website, and the iframe does not implement a mitigation for reverse tabnabbing (e.g., `sandbox` attribute, or CSP policy), you will be able to access the parent `window` object via the DOM property `window.parent`:

```
<!--- legit website -->
<iframe src="https://malicious-website.com"></iframe>
```

```
// malicious website JavaScript
window.parent.location.replace("https://get-hacked.com");
```

In summary, tabnabbing attacks rely on one tab gaining access to the `window` object that controls another tab via insecure browser DOM API function calls. Once a malicious tab has access to that `window` object—via either direct DOM calls, insecure HTML links, or improperly secured iframe spawns—the legitimate tab is now compromised since the browser DOM provides APIs for control that can be invoked from any location.

Summary

To conclude, client-side attacks are attacks that either solely target the browser client or are capable of being deployed against a browser client without the need to make requests to a web server. Client-side attacks like tabnabbing, clickjacking, and prototype pollution allow an attacker to compromise a user's application state and intercept their keystrokes—often unbeknownst and undetectable by the server that delivered the client-side application code.

For any type of offensive specialist, understanding client-side attacks is an essential component of a well-rounded modern toolkit.