# Introduction to JavaScript Security

This course will cover a range of topics in JavaScript and web security. Over the course of these lessons, we will discuss:

* Fundamental web languages and protocols: HTML, JavaScript, IP, DNS, HTTP
* Core web security constructs: Same Origin Policy, cookies, sessions
* Common web attacks: Cross Site Request Forgery, Cross Site Scripting, Server-Side Request Forgery, ReDoS
* JavaScript supply chain attacks

## Security can be fun

Cybersecurity is often portrayed as an advanced dark art, but when I started learning about it, I was surprised at how accessible it is if you simply bring a creative mindset and a willingness to think outside the box. You don’t need advanced programming knowledge or specialized tools; you just need to think critically about how systems are built, and think creatively about how they might break.

Here are two stories from my early days of security exploration that show how fun and accessible security research can be:

### Bypassing the localStorage limit

While learning web development, I encountered the localStorage API, which allows websites to store up to 5 MB of data in a user’s browser. The API is straightforward: it lets a site save small amounts of data in the browser for fast access. However, being in a security class at that time, I started to wonder: Is there a way to circumvent that 5 MB limit?

The localStorage API works on the concept of an **origin**. In web security, an origin is defined as the combination of three components:

1. The scheme (e.g., http or https).
2. The hostname (e.g., example.com).
3. The port (e.g., 80 for http or 443 for https).

An origin is a way for browsers to determine whether two pieces of content come from the same source. For example:

* https://example.com and http://example.com are **different origins** because their schemes differ.
* https://example.com and https://sub.example.com are **different origins** because their hostnames differ.
* https://example.com:8080 and https://example.com:443 are **different origins** because their ports differ.

The browser enforces security policies like the **same-origin policy**, which restricts how data can be accessed across origins. Similarly, the localStorage API applies its 5 MB limit per origin.

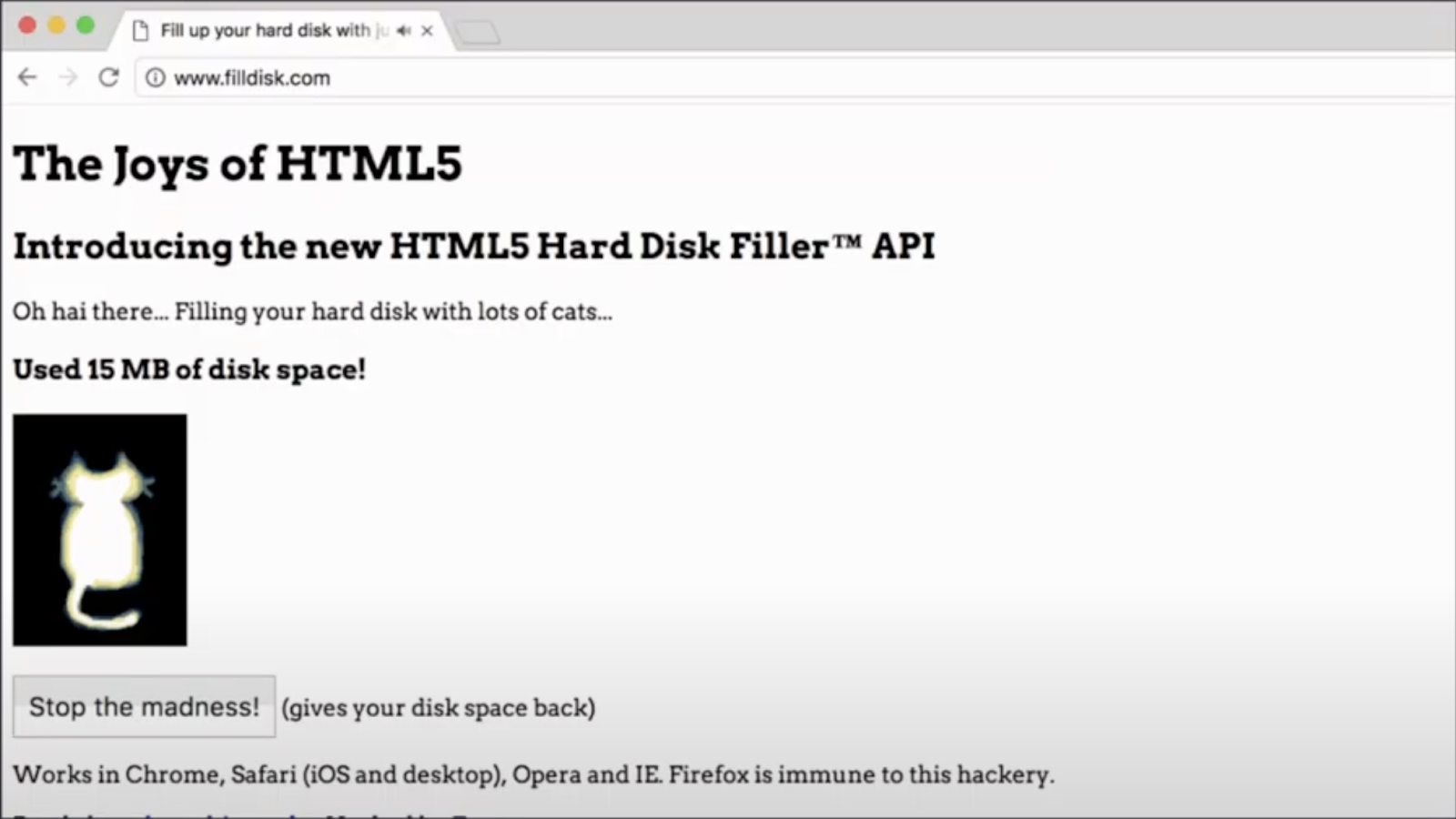
I was curious: Would it be possible to set up a site that uses subdomains to create an infinite number of origins, each utilizing its full quota, to fill up a visitor’s entire hard drive?

The web storage specification includes a safeguard against such abuse. It states:

User agents should guard against sites storing data under the origins of affiliated sites, e.g., storing up to the limit in a1.example.com, a2.example.com, a3.example.com, etc., circumventing the main example.com storage limit.

However, at the time, **Chrome, Safari, and Internet Explorer did not implement this safeguard**. Each subdomain was treated as a completely separate origin with separate quota, creating an opportunity to bypass the intended limits.

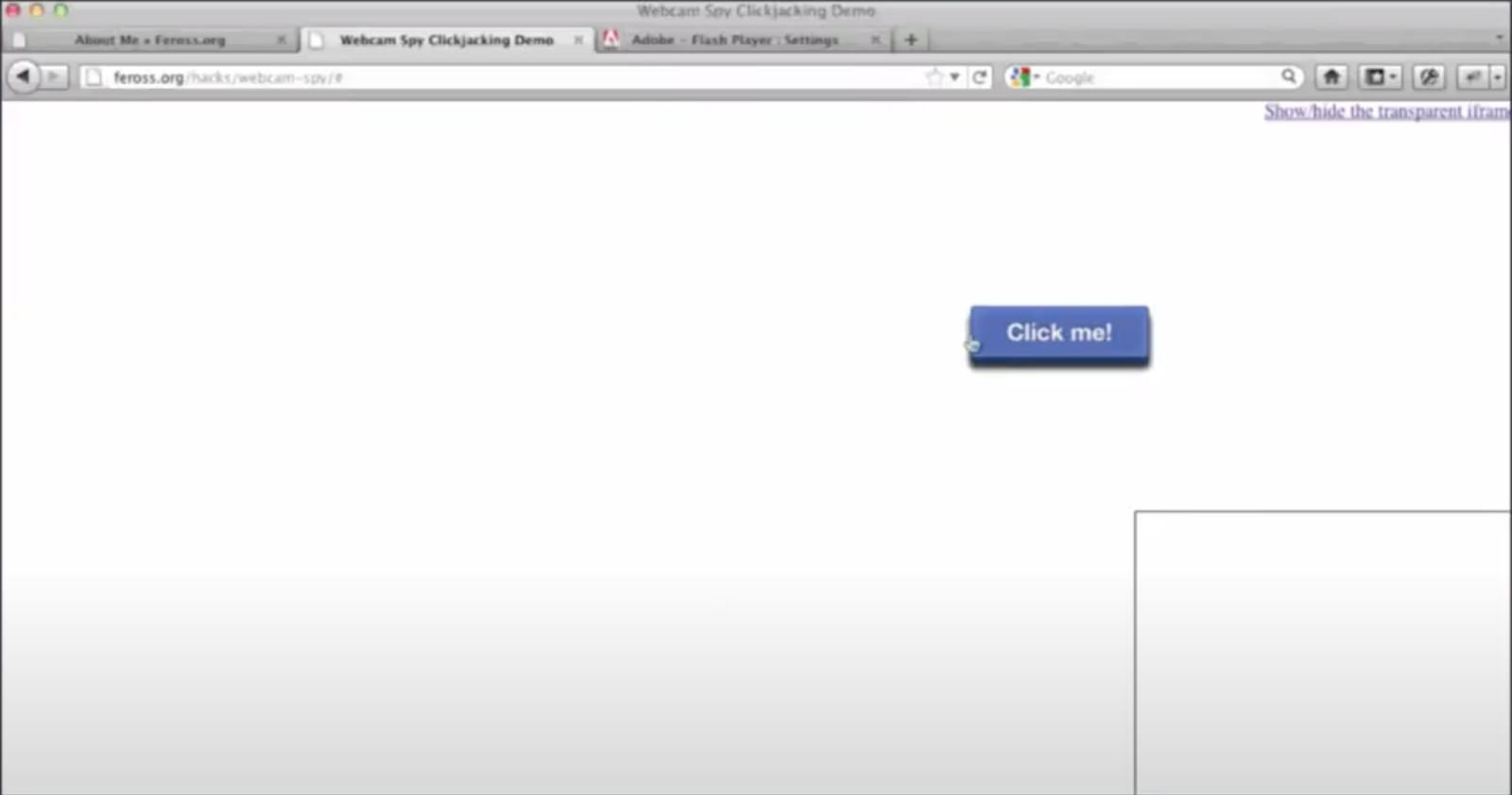
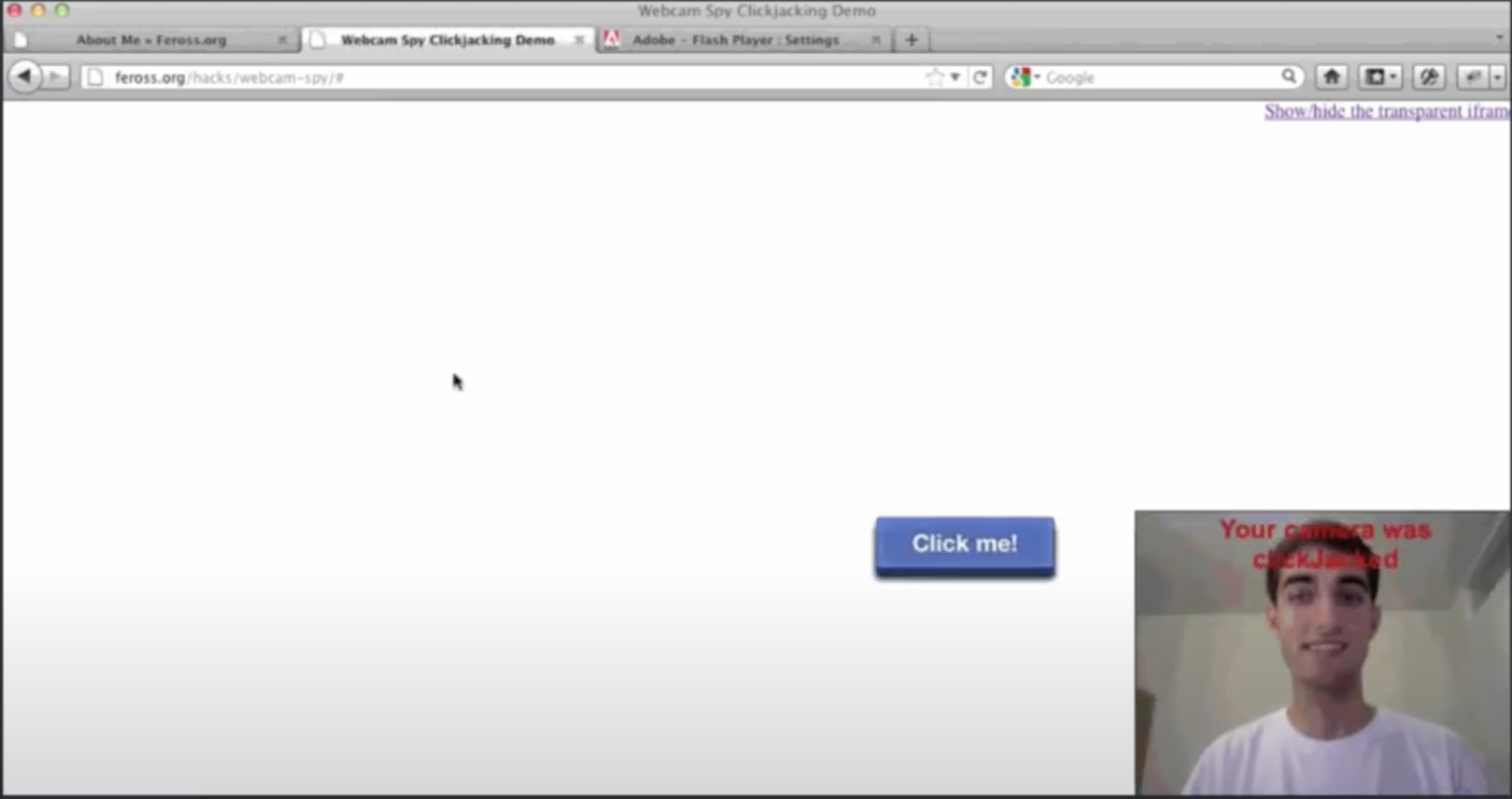
By programmatically generating subdomains (e.g., 1.example.com, 2.example.com, 3.example.com, etc.) and using hidden iframes to load them, a malicious script could quickly fill the user’s hard drive. You can explore this concept at [filldisk.com](http://www.filldisk.com/).



This attack was simple yet fun to execute, and it highlighted the impact of small implementation details. While this issue has since been fixed in all major browsers, it was a perfect example of how a security mindset can lead to significant findings.

### Clickjacking with invisible frames

Another interesting vulnerability I explored was a **clickjacking attack**. In one experiment, I created a game where users thought they were clicking a moving button on their screen. In reality, an invisible frame was positioned underneath the button, and the user’s clicks were being redirected to perform actions in the hidden frame, such as enabling their webcam without their knowledge.

This attack was surprising to execute – by manipulating the visual elements of the web page, it was possible to trick users into actions they didn’t intend to perform.

## Security can be scary

The previous examples show how it can be fun to explore security even without much experience. However, security deficiencies can be scary if not taken seriously. Here are two cases of JavaScript security gone badly at scale.

### British Airways data theft

In 2018, a [flaw in the British Airways ticket purchase platform](https://www.techrepublic.com/article/british-airways-data-theft-demonstrates-need-for-cross-site-scripting-restrictions/) allowed attackers to inject malicious JavaScript code into the website. This compromised the personal and financial information of approximately 380,000 customers between August and September 2018. Attackers stole names, addresses, emails, and full credit card information in the attack.

Following the attack, the UK Information Commissioner’s Office (ICO) issued a notice of intent to fine British Airways £183 million for infringements under the General Data Protection Regulation (GDPR). However, following appeals and discussions, the final penalty was £20 million – still one of the largest GDPR-related fines issued in the UK at the time.

### LottiePlayer crypto drainer

In October 2024, a [supply chain attack targeted](https://socket.dev/blog/supply-chain-attack-on-lottie-files-player) the [LottieFiles Player](https://lottiefiles.com/web-player) npm package, which is used for embedding and playing animations on websites, with 52k weekly downloads at time of writing. The attacker leveraged a compromised npm token from a LottieFiles engineer, publishing several malicious versions of the package. This malicious code was then bundled into various legitimate websites using the library to display animations. When users visited these compromised websites, the library would display a prompt to connect a Web3 wallet, and would subsequently drain users’ funds.

lottie-player crypto drainer prompt

lottie-player crypto drainer prompt

LottieFiles responded by revoking the compromised credentials and releasing an updated version (2.0.8) that removed the malicious payload. Although the malicious versions were quickly taken down from npm, we estimate that >$1M USD was stolen from unsuspecting users by this attack.

## Why is web security hard?

Web browsers have extremely ambitious goals to accomplish:

* **Running untrusted code securely**: The core challenge of web security lies in allowing users to run untrusted code safely in their browsers. This is no small feat; browsers blindly execute code from any website without exposing users to attack.
* **Mediating complex interactions between sites**: Modern websites often combine content and functionality from multiple sources, known as “mashups.” For example, ads, analytics, and embedded widgets often run alongside a website’s code.
* **Increasingly low-level APIs**: Users and developers demand increasingly powerful browser capabilities, pushing for APIs that interact directly with hardware or low-level system features, such as APIs for the camera, microphone, and GPU. While these features enable rich web applications, they expand the attack surface, making secure implementation more challenging.
* **Performance demands**: Web applications demand high performance. Delivering performance often involves increased complexity.
* **APIs not designed from first principles**: The web was never designed with modern security needs in mind. It started as a platform for sharing simple documents but evolved into a complex application platform. As described in *Tangled Web*:
* “Modern web applications are built on a tangle of technologies that have been developed over time and then haphazardly pieced together. Every piece of the web application stack, from HTTP requests to browser-side scripts, comes with important yet subtle security consequences. To keep users safe, it is essential for developers to confidently navigate this landscape.”
* **Strict backwards compatibility requirements**: The web operates under a “don’t break the web” philosophy, ensuring old websites remain functional indefinitely. Browsers cannot easily deprecate or remove features without risking compatibility issues. This commitment to compatibility makes improving security a slower, more difficult process.

To reiterate, browsers aim to allow any site – even malicious ones – to:

* Download content from anywhere.
* Spawn worker processes.
* Open network connections to servers or other users’ browsers.
* Render media in numerous formats.
* Run code on the GPU.
* Save and read data from the local filesystem.

This flexibility, while empowering for developers, is a significant challenge for secure implementation.

Despite these challenges, the web has proven to be fairly robust. As noted by Ilya Grigorik, a former Google web performance engineer:

“It’s all too easy to criticize, lament, and create paranoid scenarios about the ‘unsound security foundations’ of the web. Truth is, all of that criticism is true, and yet the web has proven to be an incredibly robust platform.”

Although the web is one of the most attacked platforms, its iterative security improvements make it robust. However, building secure systems on this platform requires deliberate effort and a security-focused mindset, which we will practice developing through this course.

## Preview of the class

This course will systematically build your understanding of web security from the ground up:

* **Fundamentals (Lessons 1-3)**: We’ll start by understanding the core technologies that power the web. You’ll learn about:
  + IP addresses and DNS: how computers talk to each other on the internet
  + HTTP: the protocol that powers web communication
  + HTML and JavaScript: the languages that create web experiences
* **Core Security Mechanisms (Lessons 4-5)**: We’ll explore the fundamental security boundaries in web browsers:
  + The Same-Origin Policy: how browsers isolate different websites from each other
  + Cookies and Sessions: how websites maintain state and authenticate users
* **Common Attack Patterns (Lessons 6-9)**: We’ll examine major classes of web vulnerabilities:
  + Cross-Site Request Forgery (CSRF): tricking browsers into making unauthorized requests
  + Cross-Site Scripting (XSS): injecting malicious code into trusted websites
  + SQL Injection and Command Injection: exploiting server-side code execution
  + Server-Side Request Forgery (SSRF): manipulating server-side network requests
  + Regular Expression Denial of Service (ReDoS): triggering exponential runtime in regular expression matching
* **Supply Chain Security (Lesson 10)**: Finally, we’ll look at how attackers target the dependencies that modern web applications rely on.

By the end of this course, you’ll have both a theoretical foundation in web security and practical skills for building secure applications. More importantly, you’ll develop the security mindset needed to think critically about potential vulnerabilities and creative ways systems might be exploited.