**Client-Side Prototype Pollution**

So far, we have explored and exploited server-side prototype pollution vulnerabilities. However, web browsers also commonly execute JavaScript on the client-side, which can also be vulnerable to prototype pollution. In this section, we will discuss how we can exploit client-side prototype pollution vulnerabilities.

Since client-side prototype pollution is a client-side vulnerability, a common exploit is [DOM-based Cross-Site Scripting (XSS)](https://owasp.org/www-community/attacks/DOM_Based_XSS) or bypassing HTML sanitizers to enable other XSS vulnerabilities, as demonstrated in [this](https://research.securitum.com/prototype-pollution-and-bypassing-client-side-html-sanitizers/) blog post.

**Code Review - Identifying the Vulnerability**

This section will examine a client-side prototype pollution vulnerability without access to the web application's source code, focusing solely on the frontend source code. After starting the lab's target, we can immediately notice that it is a PHP web application due to the .php extension in /index.php. Therefore, server-side prototype pollution is impossible. However, let us analyze the frontend source code for any vulnerabilities.

After logging in to the sample web application, we see a form to report profiles to the admin, secured with a Google reCaptcha:

A screenshot of a computer

Description automatically generated

The server response consists of the following code:

Code: html

<!DOCTYPE html>

<html lang="en">

<head>

<SNIP>

<script src="/jquery-deparam.js"></script>

<script src="/purify.min.js"></script>

<script src="https://www.google.com/recaptcha/api.js" async defer></script>

</head>

<body>

<SNIP>

<script>

let params = deparam(location.search.slice(1))

let color = DOMPurify.sanitize(params.color);

document.getElementById("form").style.backgroundColor = color;

</script>

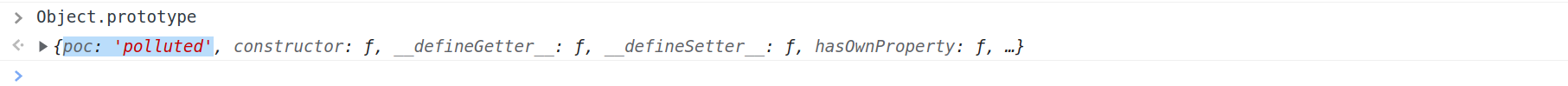
</div>

</body>

</html>

The response contains three JavaScript libraries: jQuery-deparam, DOMPurify, and Google ReCaptcha. We can set the submission form's background color using the GET parameter color. However, the parameter is correctly sanitized using DOMPurify, thus preventing an XSS vulnerability. However, if we search for prototype pollution vulnerabilities in these client-side libraries, we find an issue. Let us look at the overview provided [here](https://github.com/BlackFan/client-side-prototype-pollution#prototype-pollution). We can see that jQuery-deparam is vulnerable to prototype pollution. For more details, check out [this](https://github.com/BlackFan/client-side-prototype-pollution/blob/master/pp/jquery-deparam.md) page.

Let us use the PoC in the GitHub page to trigger the prototype pollution vulnerability by navigating to the following URL: /profile.php?\_\_proto\_\_[poc]=polluted. Subsequently, we can open the JavaScript console in the browser by pressing F12 and confirm the prototype pollution vulnerability by inspecting the Object.prototype object and finding our polluted property:



Now that we have successfully confirmed prototype pollution, let us investigate how to exploit it to obtain DOM-based XSS.

Since we are analyzing a client-side prototype pollution vulnerability that does not result in permanent changes in the web application, we do not need to test our exploit on a local copy of the source code first. In this particular case, we are unable to do so anyway because we do not have access to the backend source code. If we refresh the web page, we can start over again without worrying about breaking the web application or harming other users.

**Exploitation**

Looking at the JavaScript code in the response, the params.color property looks like a good target for prototype pollution since we can use it if we do not specify a color GET parameter. We identified previously that the params.color property is sanitized by DOMPurify, so we cannot use it to achieve XSS. However, we can look for script gadgets that we can exploit in combination with prototype pollution to achieve XSS in external libraries. Script gadgets are legitimate and benign JavaScript code that can be used in combination with a different attack vector to achieve JavaScript code execution (XSS). In particular, we are interested in script gadgets that lead to XSS if the prototype object is manipulated. Looking at the overview [here](https://github.com/BlackFan/client-side-prototype-pollution#script-gadgets), we see that Google reCaptcha contains a script gadget. For more details, check out [this](https://github.com/BlackFan/client-side-prototype-pollution/blob/master/gadgets/recaptcha.md) page.

We can confirm the XSS vulnerability using the payload provided on [this](https://github.com/BlackFan/client-side-prototype-pollution/blob/master/gadgets/recaptcha.md) page by navigating to /profile.php?\_\_proto\_\_[srcdoc][]=<script>alert(1)</script>. With proper URL encoding of special characters, we can then inject any XSS payload, for instance, the following:

Code: html

/profile.php?\_\_proto\_\_[srcdoc][]=<script>window.location%3d"/poc.php";</script>

**Tools**

Now that we have discussed how to identify and exploit client-side prototype pollution vulnerabilities, let us discuss tools we can use to help us in the process. In particular, we will focus on [DOM Invader](https://portswigger.net/burp/documentation/desktop/tools/dom-invader), a browser-based tool in Burp. In order to use it, we need to start the Chromium browser integrated into Burp Suite. The DOM Invader extension is automatically installed.

We can find it by clicking on the Extensions icon next to the URL bar and pinning the Burp Suite extension:

A screenshot of a computer

Description automatically generated

Afterward, we can click on the Burp Suite Extension logo, navigate to the DOM Invader tab, and toggle the DOM Invader option:

A screenshot of a computer

Description automatically generated

Next, click on Attack types and enable Prototype Pollution:

A screenshot of a computer

Description automatically generated

Finally, click Reload at the bottom and navigate to the vulnerable page, in our case, /profile.php in the sample web application. DOM Invader now checks the page for prototype pollution vulnerabilities. We can find the results in the DOM Invader tab after opening the devtools by pressing F12:

A screenshot of a computer

Description automatically generated

In our sample web application, DOM Invader finds the prototype pollution we have discussed above. However, it displays two possible exploitation vectors using \_\_proto\_\_ and constructor[prototype]. We can click Test for any of the vulnerabilities to confirm them. The vulnerable URL is opened in a new tab, where we can display the Object.prototype object in the JavaScript console to confirm the prototype pollution vulnerability:

A screenshot of a computer

Description automatically generated

Lastly, DOM Invader can also find script gadgets that enable us to escalate the prototype pollution to an XSS vulnerability. We can click Scan for gadgets to let DOM Invader search for script gadgets. After the scan has finished, we can find the result in the DOM Invader tab in the devtools. However, in this case, DOM Invader does not find the XSS vector we exploited above.

For more details on how to use DOM Invader to identify and exploit client-side prototype pollution vulnerabilities, check out the [documentation](https://portswigger.net/burp/documentation/desktop/tools/dom-invader/prototype-pollution).