

## **Client-side Testing**

### **Aim :**

To perform client side testing using burpsuite

### **Description:**

There exist a variety of testing tools that can make it easy to implement client-side A/B testing. Many of them include a WYSIWYG editor that lets you easily change components in a visual editor without needing to reach into the code at all. This type of testing framework makes running tests on the client-side extremely easy and intuitive.

It also makes it possible for marketing teams to run experiments without needing to employ a front-end developer. Not a single line of code needs to be written, not a single actual deployment needs to happen until the experiment is complete. Once that happens, the developers only need to be brought in if the winning variation was one of the alternatives: otherwise, the alternate variations can simply be scrapped and a new experiment can begin.

Another benefit of client side testing is the additional user data available. Because the variation hasn't been decided until the page loads in the visitor's browser, more data can be gathered about the user to determine which variation to serve. On the other hand, server side testing has less user data to work on, so it's less able to segment users.

There are some drawbacks to client side testing, though. The most common is that, since the test is implemented using client side JavaScript, the user experience can suffer. Depending on the specific implementation, the page load time can get higher as it takes a second to determine what variation the user should see, or the user could see a "flickering" effect on the webpage as the original version is displayed before the test variation displays in its place. While load time issues are harder to fix, flickering can be tactically improved by only using client-side tests for elements below the fold.

A client side A/B test, like any other A/B test, begins with a hypothesis. "We think changing the color of this CTA button will improve conversion rate" is a classic example. Once the hypothesis is determined, the variations can be created using the visual editor and displayed to users using the testing tool.

After the test is complete, significance is calculated, and the winning variation is determined, it's time to implement the winner. This is a key difference between client-side and server-side testing: when an alternate version wins in an experiment, the actual deployment process is slower than in client-side testing because the variations have not yet been built. With a server-side test, the variations have to be built in order to be tested, so the rollout process is extremely fast. However, on the converse, if a test fails to produce significant results, the variations that cost developer effort for a server-side test will have to simply be scrapped, whereas no developer effort went into creating variations in a client-side test. There is less cost to doing more experiments if they're done on the client-side.

## **Procedure:**

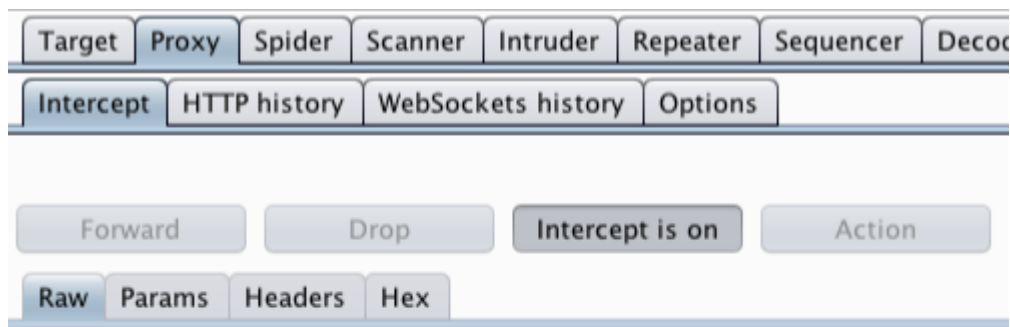
### **1. Testing for Cross-Site Scripting:**

Reflected cross-site scripting vulnerabilities arise when data is copied from a request and echoed in to the application's immediate response in an unsafe way. An attacker can use the vulnerability to construct a request which, if issued by another application user, will cause JavaScript code supplied by the attacker to execute within the user's browser in the context of that user's session with the application. The attacker-supplied code can perform a wide variety of actions, such as stealing the victim's session token or login credentials, performing arbitrary actions on the victim's behalf, and logging their keystrokes.

First, ensure that Burp is correctly [configured with your browser](#). With intercept turned off in the [Proxy](#) "Intercept" tab, visit the web application you are testing in your browser.

The screenshot shows a web application interface for performing a DNS lookup. At the top, a pink rounded rectangle contains the text "Who would you like to do a DNS lookup on?" and "Enter IP or hostname". Below this, the label "Hostname/IP" is positioned to the left of a text input field. Underneath the input field is a blue button with the text "Lookup DNS". At the bottom of the interface, there is a grey bar with the text "Results for" followed by a horizontal line.

Visit the page of the website you wish to test for XSS vulnerabilities.



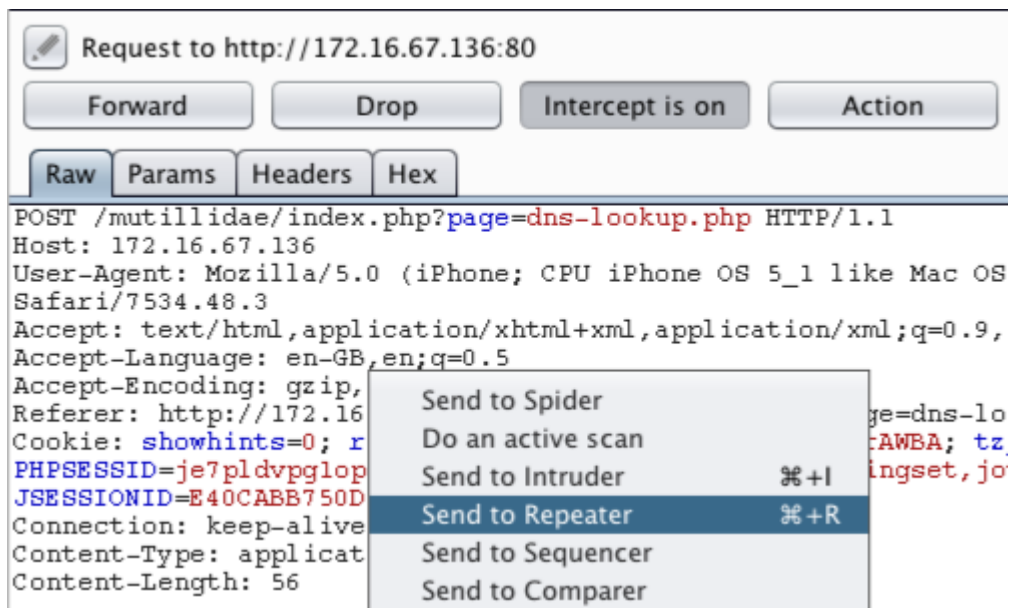
Return to Burp. In the [Proxy](#) "Intercept" tab, ensure "Intercept is on".

**Who would you like to do a DNS lookup on?**  
**Enter IP or hostname**

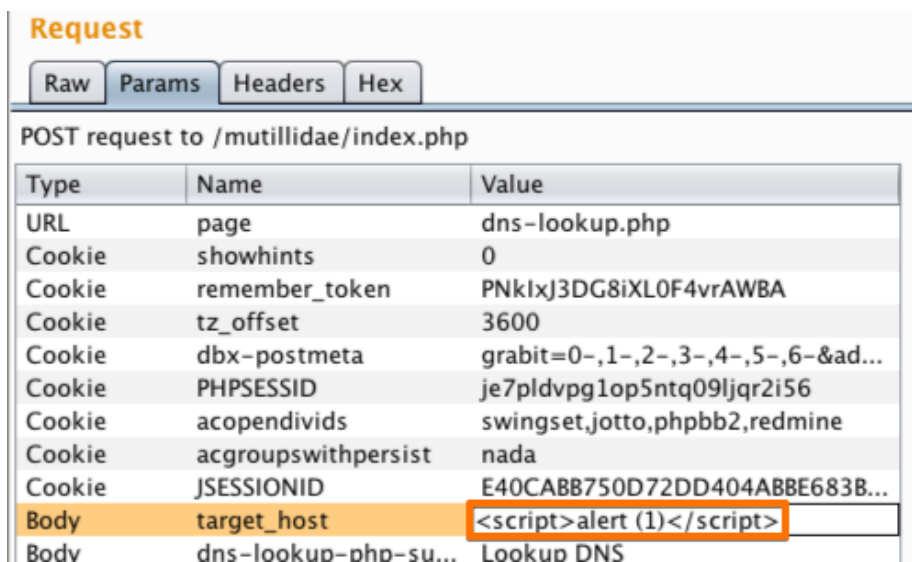
**Hostname/IP**

**Results for**

Enter some appropriate input in to the web application and submit the request.



The request will be captured by Burp. You can view the HTTP request in the [Proxy](#) "Intercept" tab. You can also locate the relevant request in various Burp tabs without having to use the intercept function, e.g. requests are logged and detailed in the "HTTP history" tab within the "Proxy" tab. Right click anywhere on the request to bring up the context menu. Click "Send to Repeater"



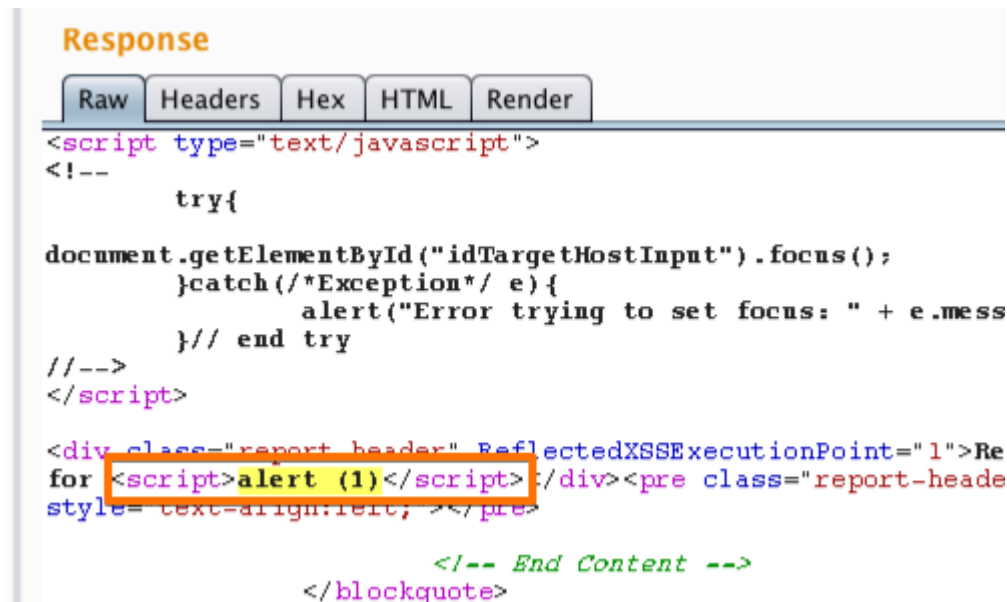
Go to the [Repeater](#) tab.

Here we can input various XSS payloads into the input field. We can test various inputs by editing the "Value" of the appropriate parameter in the "Raw" or "Params" tabs.

A simple payload such as <s> can often be used to check for issues.

In this example we have used a payload that attempts to perform a proof of concept pop up in our browser.

Click "Go".



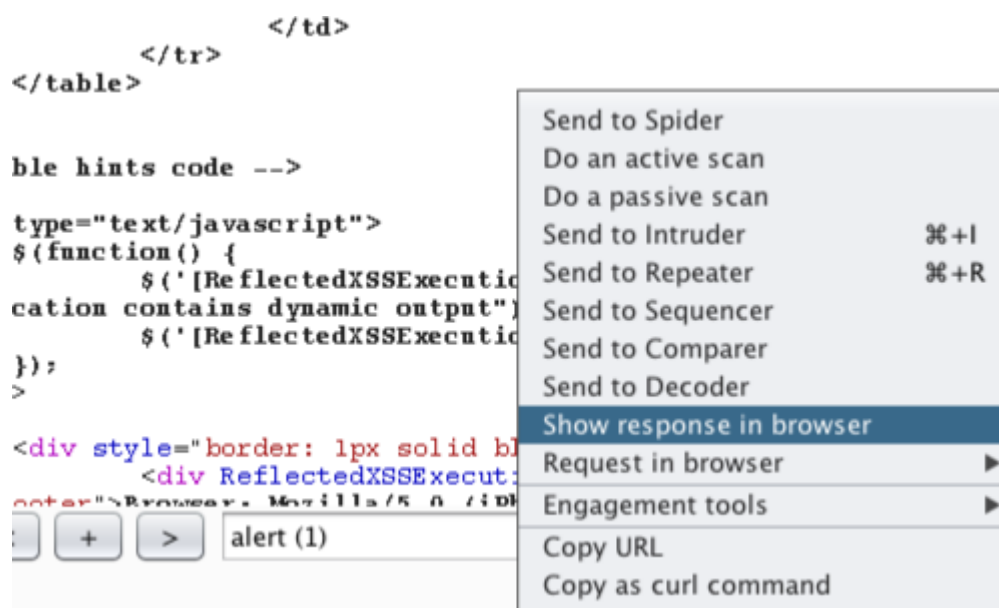
The screenshot shows a web application response in a tool like Burp Suite. The 'Response' tab is active, and the 'Render' sub-tab is selected. The rendered HTML shows a JavaScript payload that attempts to focus an element with the ID 'idTargetHostInput'. If this fails, it triggers an alert with the message '1'. The payload is highlighted with a red box. Below the payload, there is a comment indicating the end of the content.

```
<script type="text/javascript">
<!--
    try{
document.getElementById("idTargetHostInput").focus();
    }catch(*Exception*/ e){
        alert("Error trying to set focus: " + e.message);
    }// end try
//-->
</script>

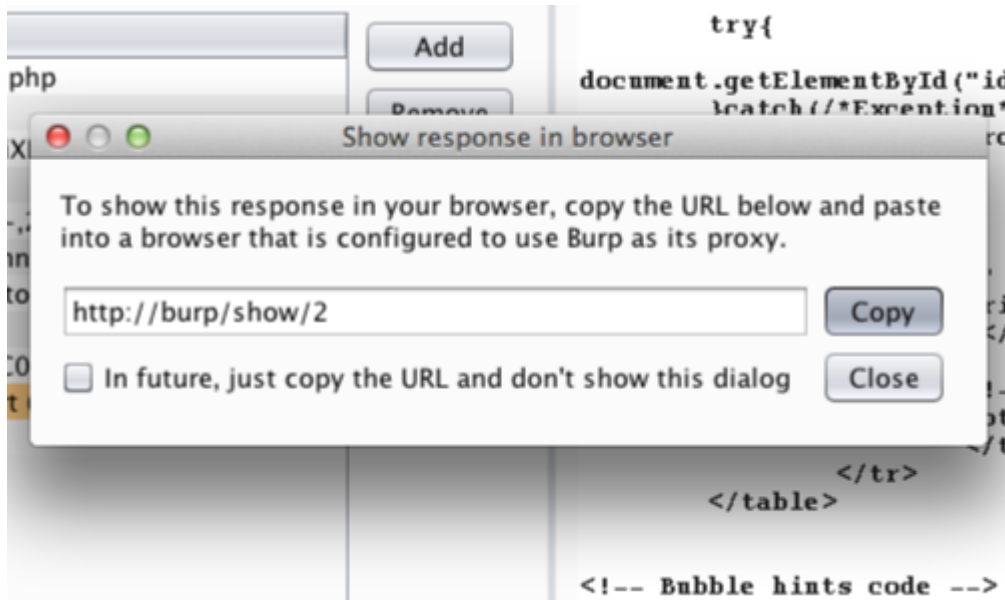
<div class="report_header" ReflectedXSSExecutionPoint="1">Re
for <script>alert (1)</script>:/div<pre class="report-heade
style="text-align:left;"></pre>

<!-- End Content -->
</blockquote>
```

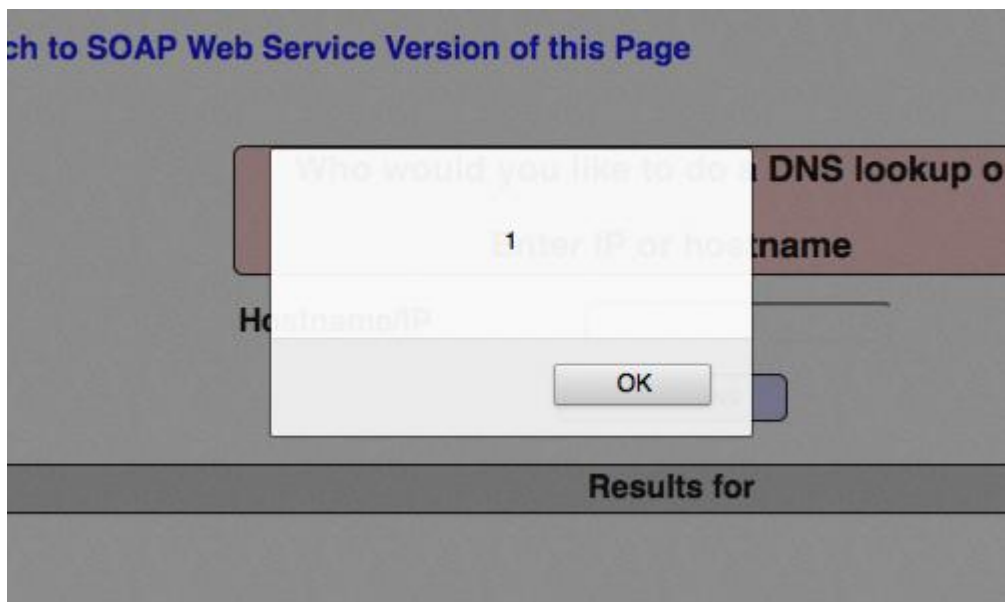
We can assess whether the attack payload appears unmodified in the response. If so, the application is almost certainly vulnerable to XSS. You can find the response quickly using the search bar at the bottom of the response panel. The highlighted text is the result of our search.



Right click on the response to bring up the context menu. Click "Show response in browser" to copy the URL. You can also use "Copy URL" or "Request in browser".



In the pop up window, click "Copy".

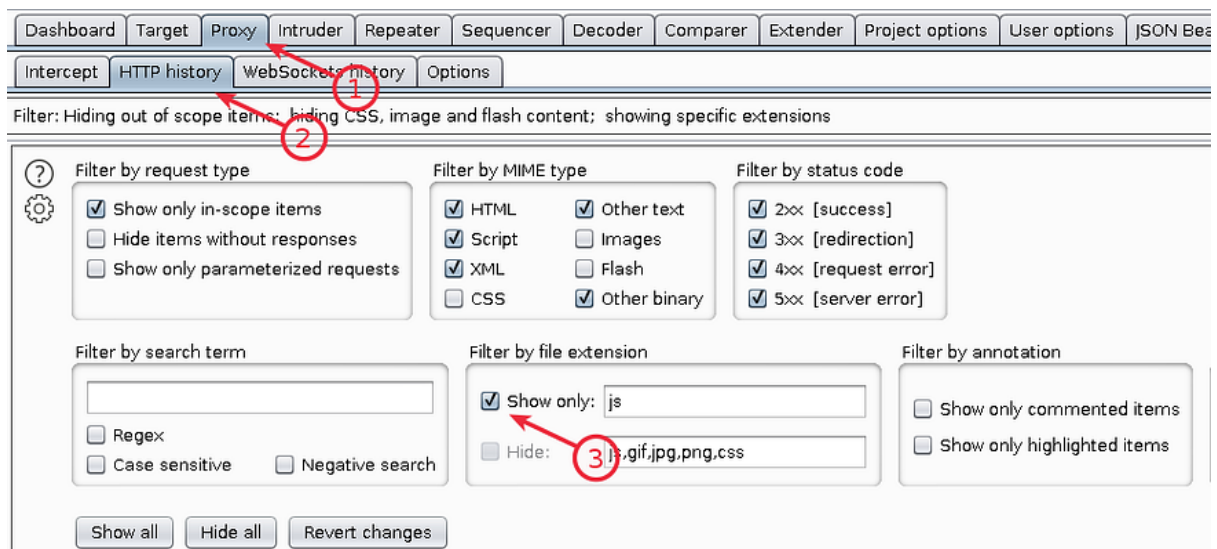


Copy the URL in to your browser's address bar.

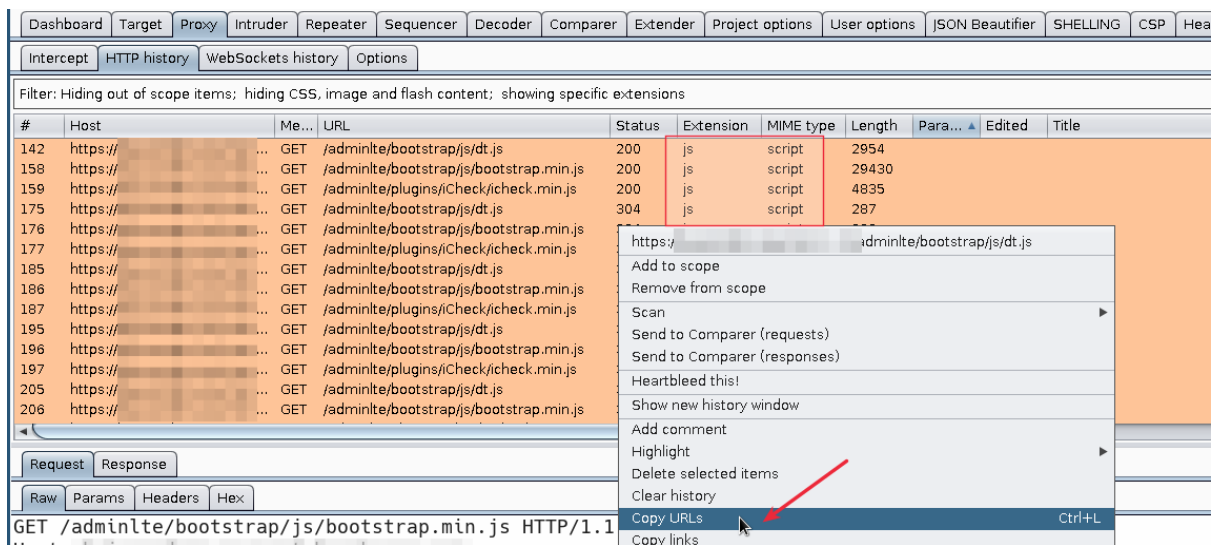
In this example we were able to produce a proof of concept for the vulnerability.

## 2. Testing for JavaScript Execution

If you use Burp Suite for testing applications then there are multiple ways to gather all the JavaScript files in an application. Navigate through an application while the traffic is being sent through Burp proxy. Once you are done with the navigation, you can use Burp's tool-set to extract all the JavaScript files. If you are using Burp Suite Community Edition, you can navigate to proxy > HTTP history and use the display filters to only display the JavaScript files used by the application. You can also copy the URLs for all the JavaScript files displayed.

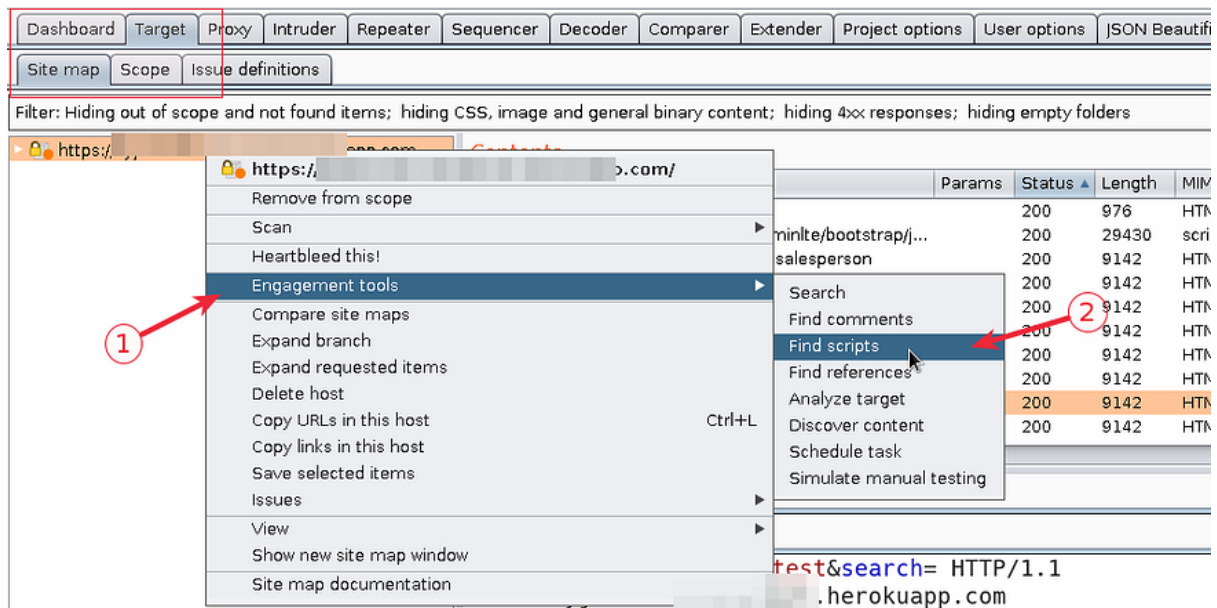


Burp display filters to display only JavaScript files for a given application

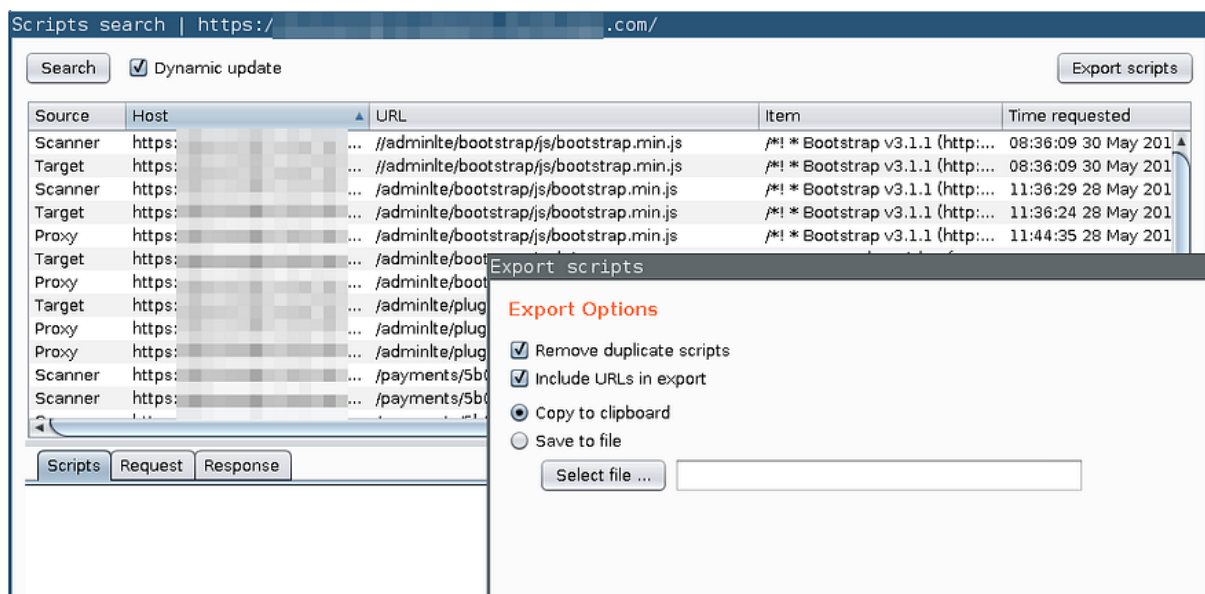


Copy the URLs for all the JavaScript file displayed after filtering

If you are using Burp Suite Professional, You can not only copy the URLs for all the JavaScript files in an application but also export all the scripts. Under Target > Site map right click on the site of interest and select Engagement tools > Find scripts . Using this feature you can export all the scripts in that application and also copy URLs.



Burp “Find Scripts” to identify all the JS files on an application



Burp “Find scripts” can export all the scripts, not just URLs



### **3. Testing for HTML Injection**

This vulnerability occurs when the user input is not correctly sanitized and the output is not encoded. An injection allows the attacker to send a malicious HTML page to a victim. The targeted browser will not be able to distinguish (trust) the legit from the malicious parts and consequently will parse and execute all as legit in the victim context.

There is a wide range of methods and attributes that could be used to render HTML content. If these methods are provided with an untrusted input, then there is an high risk of XSS, specifically an HTML injection one. Malicious HTML code could be injected for example via `innerHTML`, that is used to render user inserted HTML code. If strings are not correctly sanitized the problem could lead to XSS based HTML injection. Another method could be `document.write()`

When trying to exploit this kind of issues, consider that some characters are treated differently by different browsers. For reference see the DOM XSS Wiki. The `innerHTML` property sets or returns the inner HTML of an element. An improper usage of this property, that means lack of sanitization from untrusted input and missing output encoding, could allow an attacker to inject malicious HTML code.