Hunting for reflected XSS vulnerabilities: A complete guide

BY BLACKBIRD-EU · OCTOBER 20, 2025 · LAST UPDATED ON OCTOBER 26, 2025

Cross-site scripting vulnerabilities are, by no doubt, one of the vulnerability types that'll keep haunting applications for a long time. This seamless injection bug can often be further escalated to allow attackers to perform malicious actions on behalf of the victim, or even worse, on behalf of a vulnerable server-side component, from reading and changing account information, such as passwords or emails, to reaching internal-only resources and even reading local files.

In this article, we'll look at a proven methodology to identify reflective XSS vulnerabilities while also diving deeper into some advanced exploitation methods.

Let's dive in!

What is Cross-site scripting (XSS)

Cross-site scripting (XSS) is an injection vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. It works by exploiting insufficient input validation and a lack of encoding of any reflection of this user input, enabling attackers to insert HTML or JavaScript code that runs in the victims' browsers upon visiting the compromised page.

Depending on the vulnerable component, XSS can occur on both the client-side and server-side. Client-side XSS happens solely in the victim's web browser, and it allows attackers to take over the victim's session.

Server-side XSS (or also referred to as blind XSS) occurs when the vulnerable component uses your unsanitized input and evaluates it within the DOM on the server-side, for instance, via a headless web browser. This enables attackers to use arbitrary JavaScript code to initiate outgoing requests on behalf of the server, and in severe cases, even read local files (depending on the headless web browser's deployment configuration).

Now let's take a look at the different types of XSS vulnerabilities.

Reflected XSS

Reflected XSS (or sometimes referred to as reflective XSS) occurs when malicious user input is injected through a request property (such as the URL path, fragment, query/body parameter, or HTTP header) and immediately reflected back to the user without proper sanitization.

The server processes the user input and includes it in the HTTP response without any encoding, causing the victim's browser to execute the attacker's script when they click a specially crafted link.

Stored XSS

Stored XSS (or sometimes referred to as persistent XSS) happens when an attacker's malicious script gets saved in the target's database, file system, or any other type of storage service (such as AWS S3). When other users later retrieve this stored data (for instance, viewing a comment section, public profile, or forum post), the malicious script executes in their browser.

Stored XSS is particularly dangerous as it can affect multiple victims without requiring them to click a specific link, unlike reflected XSS.

DOM-based XSS

DOM-based XSS occurs when unsafe JavaScript code processes user-controllable data (from a DOM source) and passes it to a DOM sink. This allows attackers to craft JavaScript payloads that would be evaluated by the vulnerable application.

DOM-based XSS is harder to spot as malicious user input is not immediately reflected into the HTTP response but rather passed to a DOM sink. The identification and exploitation of DOM-based XSS vulnerabilities will be discussed in an upcoming article.

Throughout this article, we will solely cover reflected (or reflective) and stored (persistent) XSS, as both of these types share the same characteristics.

What is self-XSS?

Self-XSS occurs when an attacker tricks a victim into executing malicious JavaScript in their own browser, typically by convincing them to paste code into the browser's developer console or a text field on a legitimate website that's only accessible to them (for instance, the address field on your profile).

While this technically causes script execution, most bug bounty programs and security researchers don't consider self-XSS to pose a security risk because it requires the victim to actively perform the attack against themselves, which breaks the fundamental security principle that vulnerabilities should be exploitable without extensive social engineering.

Do note that there are cases where self-XSS vulnerabilities can be chained and further escalated, but this is a topic that we will cover more extensively in an upcoming article.

Methodology

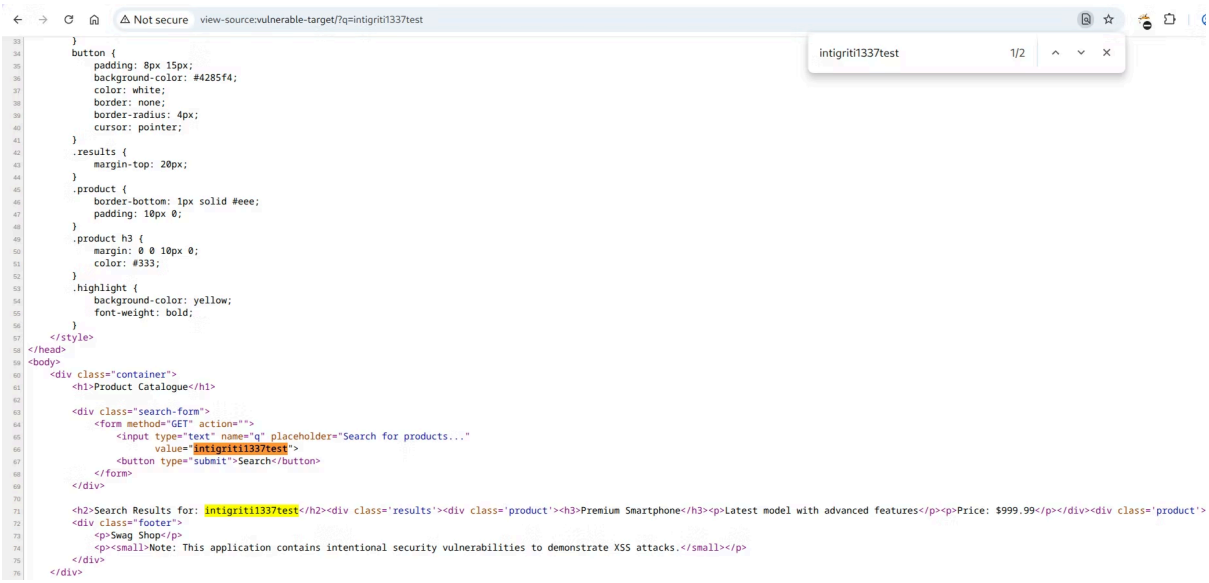
If you're a beginner, this part is essential. It can help you save hours in determining whether you've found an XSS vulnerability or are dealing with a simple content injection. Let's go through this 3-step methodology to help us identify a reflected or stored XSS vulnerability.

Hunting for reflected XSS vulnerabilities (methodology)

Step 1: Reflection

The first step is identifying where your input gets reflected in the application's response. Insert a unique string (such as **intigriti1337test**) into various input fields, URL parameters, headers, or any other user-controllable data points.

Next, search for this string in the HTTP response. This helps you map out all reflection points and understand where your input ends up, which is crucial for determining whether exploitation is possible and what type of injection you're dealing with.



Searching for cross-site scripting (XSS) reflection point

Finding hidden input parameters

Enumerating (hidden) parameters can help you discover all types of injection vulnerabilities, including XSS! In our detailed guide, we outlined 5 ways to discover hidden parameters. Read the article today.

https://www.intigriti.com/researchers/blog/hacking-tools/finding-hidden-input-parameters

Step 2: Injection

Once you've found a reflection point, test whether you can break out of the current context by injecting a simple HTML tag like `<s>intigriti1337test` and observe how the application handles it.

For reflections in JavaScript context (anywhere inside the `<script>` tag, the injection string will look slightly different. We will discuss this case more in-depth shortly.

Input reflected with no injection

Your input likely got HTML-encoded. In this instance, XSS is often not possible since your arbitrary input got sanitized. However, keep in mind that there are caveats whereby applications can show different behavior based on your user-agent (mobile vs desktop) or whenever you inject special characters, such as null bytes, CR/LF characters, etc.

Example of a non-vulnerable cross-site scripting (XSS) case

Input reflected with injection

When your input is reflected and injected, meaning no special characters are encoded, you have a strong indication that XSS is possible. All we have to do right now is inject a special HTML tag or event handler to transform our simple HTML injection into an XSS vulnerability.

Searching for cross-site scripting (XSS) injection point

Step 3: Payload (proof of concept)

Now it's time to craft a working proof of concept that executes JavaScript in the victim's browser. To do so, your payload depends on 2 factors: 1) the context in which your input is reflected, and 2) any existing filters preventing you from injecting malicious XSS payloads.

Let's take a look at several contexts in which your unsanitized input can appear, and also go through a few payloads that can help us break out of it and achieve code execution.

Generic (HTML context)

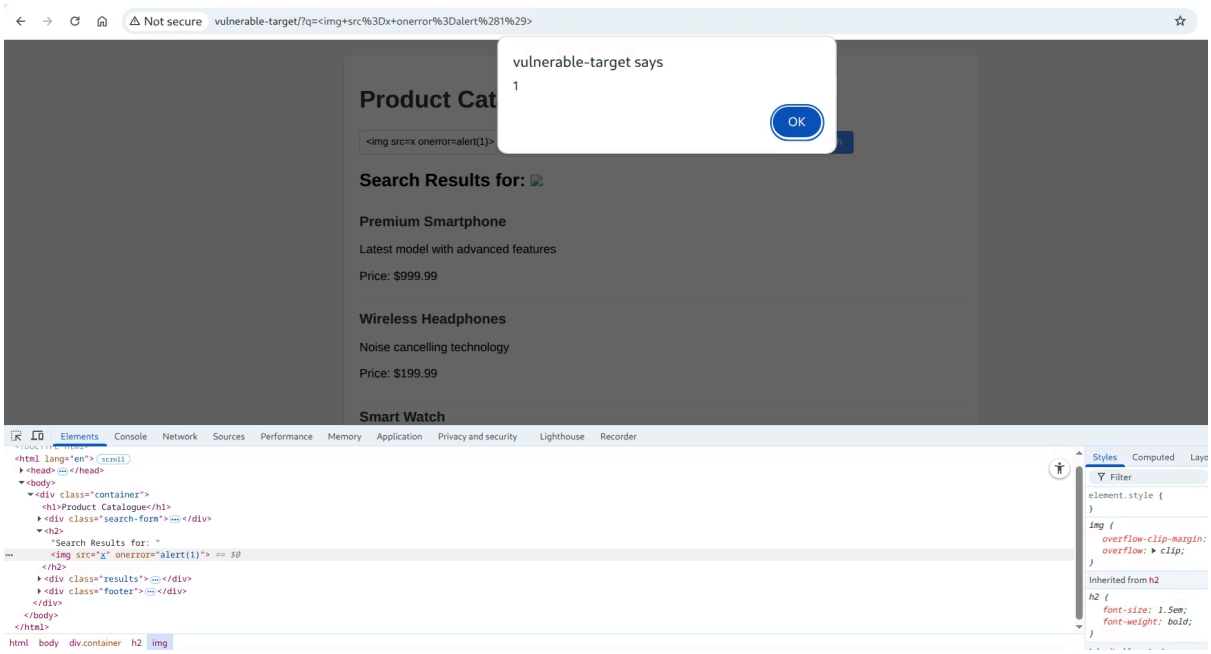
When your input is directly reflected in the HTML body without being wrapped in any specific tags or attributes, you have the most flexibility for exploitation. Start simple with payloads like:

```
<script>alert(1)</script>
```

Or:

```
<img src=x onerror=alert(1)>
```

Both aforementioned payloads will work without needing to break out of any tag. This is the easiest scenario for beginners, as you can use virtually any HTML tag that supports JavaScript execution, including `<svg>`, `<iframe>`, `<object>`, or event handler attributes on self-closing tags.



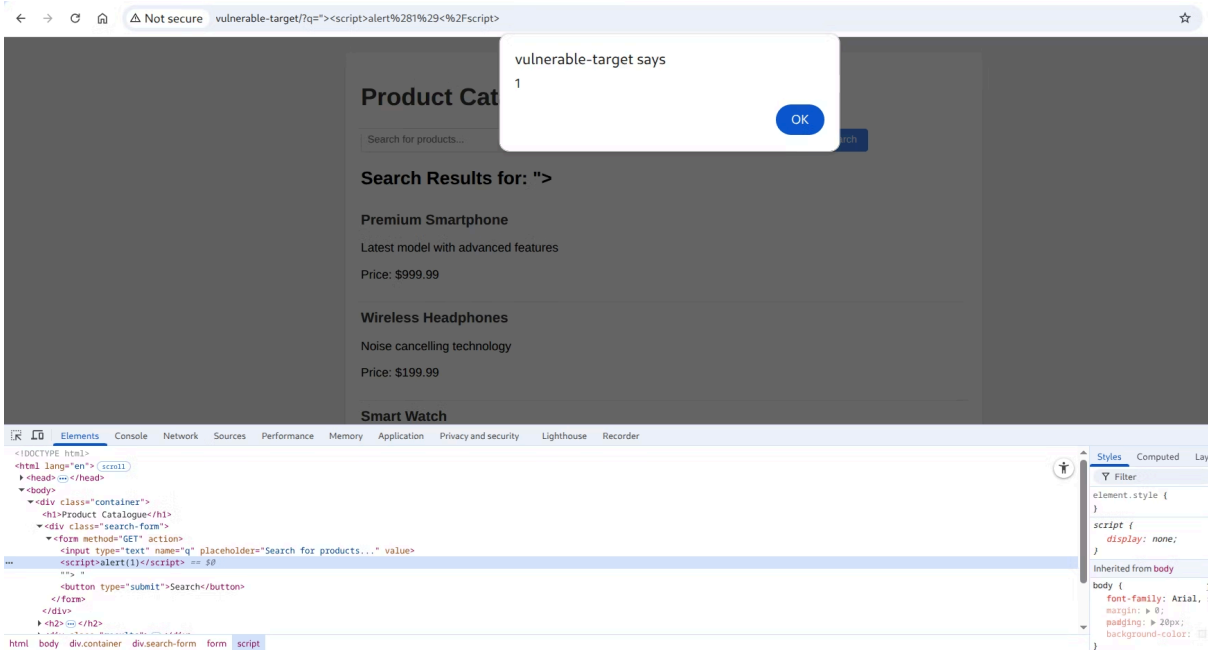
Cross-site scripting (XSS) in HTML page (generic)

HTML attribute (inline HTML)

When your reflection appears inside an HTML attribute (like **<input value="REFLECTION">** or **<form action="/path/to/submit?param1=REFLECTION">**), you'll need to break out of the attribute context first before injecting your payload.

You can either close the attribute with a quote, close the entire tag with **>**, and then inject a new malicious tag like **"><script>alert(1)</script>**, or stay within the same tag by adding an event handler like **" onload=alert(1) x="**.

The key is understanding which quote type (single or double) is used to wrap the attribute and whether the application filters any other characters that could block us from escaping the context.



Cross-site scripting (XSS) in an HTML attribute (inline HTML)

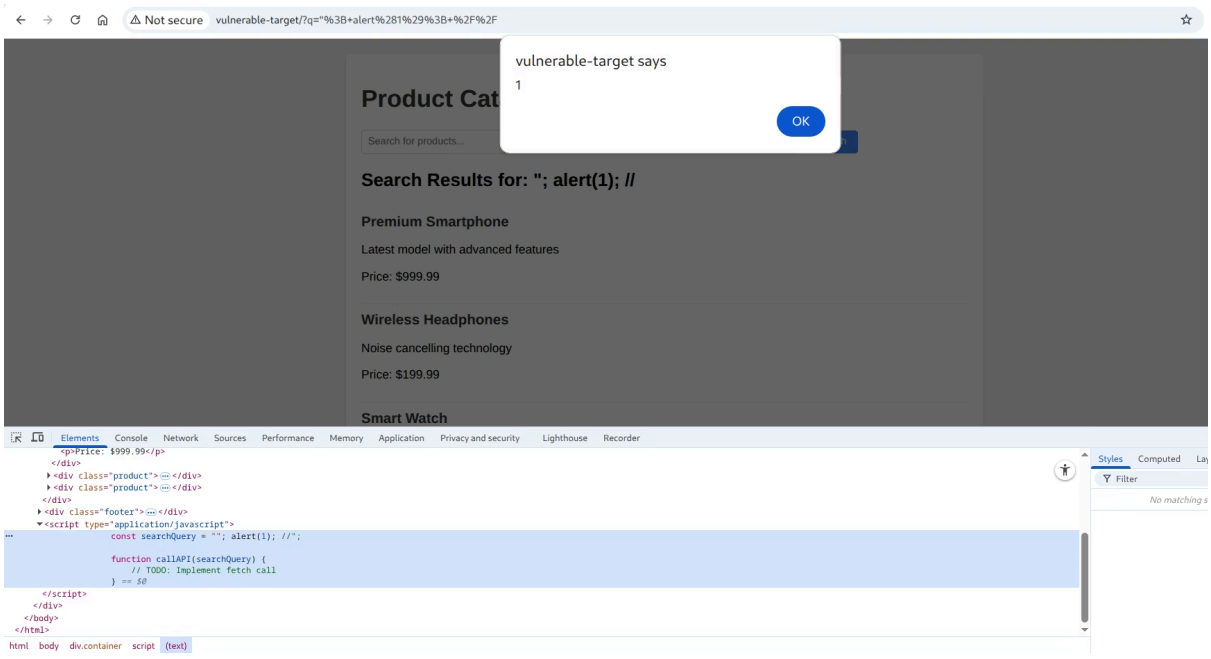
JavaScript block

When your input gets reflected inside `<script>` tags, typically as part of a JavaScript variable assignment like `var data = "REFLECTION";` or `var config = {"key": "REFLECTION"};`, you need to break out of the JavaScript syntax to execute your code.

For regular strings, escape with `"; alert(document.domain); //` to close the string, execute your code, and comment out the rest.

For template literals (backticks), you can use `${alert(document.domain)}` to execute code directly within the template without breaking the syntax.

The tricky part here is ensuring the JavaScript remains syntactically valid after your injection, so pay attention to unclosed brackets, quotes, or parentheses that might cause errors and prevent execution.



Cross-site scripting (XSS) in JavaScript block

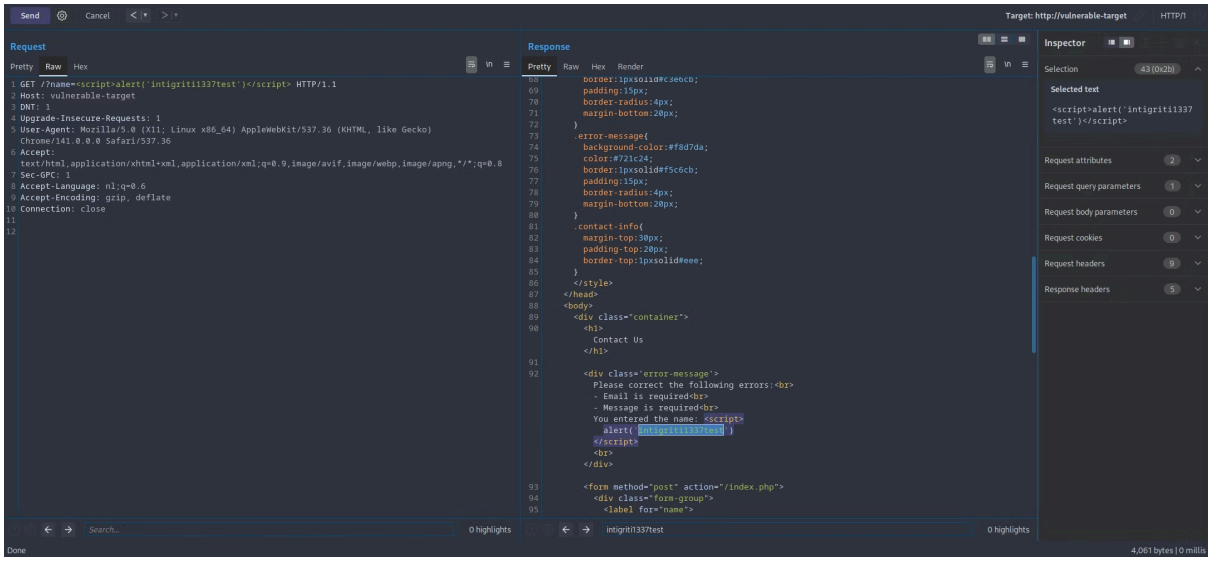
XSS exploitation

We've covered what XSS vulnerabilities are and how we can methodically probe and identify possible injection points. Now it's time to put our newly acquired skills to the test. Let's take a look at a few real-world examples to better help us craft our payloads.

XSS with no filtering

In rare cases, you'll come across scenarios where no filtering or validation is done to prevent XSS vulnerabilities. These are usually introduced by oversights from developers forgetting to sanitize user input.

In these instances, we can use any payload we like to prove the existence of XSS:



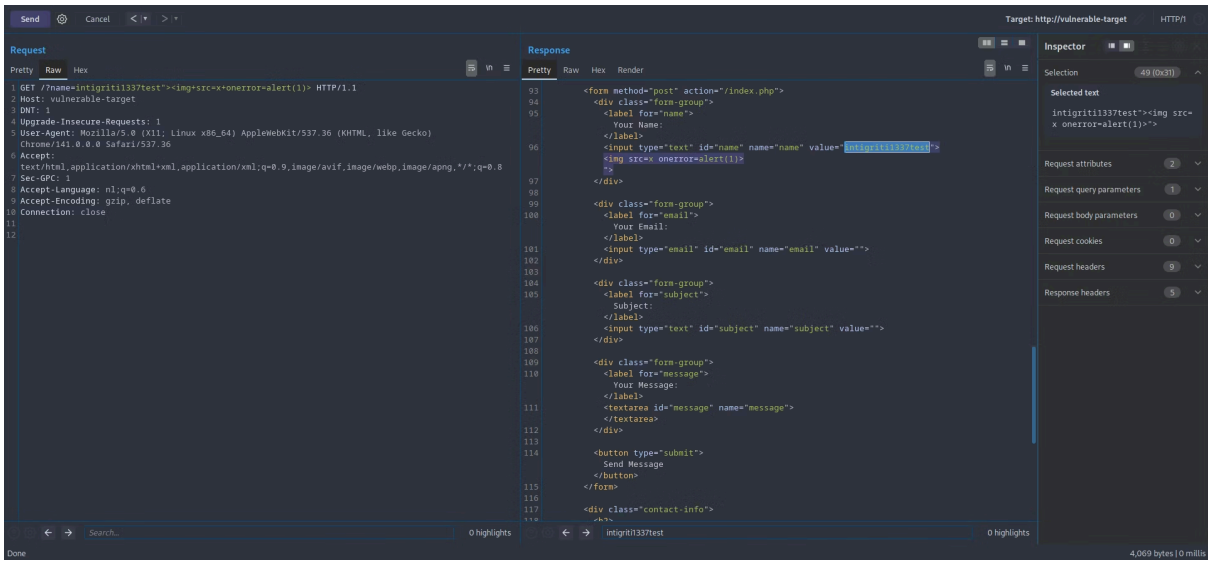
Example 1: Basic cross-site scripting (XSS) with no input sanitization

XSS in inline-HTML (attributes)

Another trivial scenario is where your input is reflected in the value of an HTML attribute. In this instance, you can either:

1. Break out of the HTML attribute and inject an event handler to execute arbitrary JS code
2. Or, break out of the HTML attribute, close the tag, and open a new tag to execute arbitrary JS code

Depending on the context, it's usually far easier to inject a new event handler. While in other cases, it might be the only possible solution, as characters used to open and close HTML tags are explicitly filtered.



Example 2: Basic cross-site scripting (XSS) with an inline HTML payload

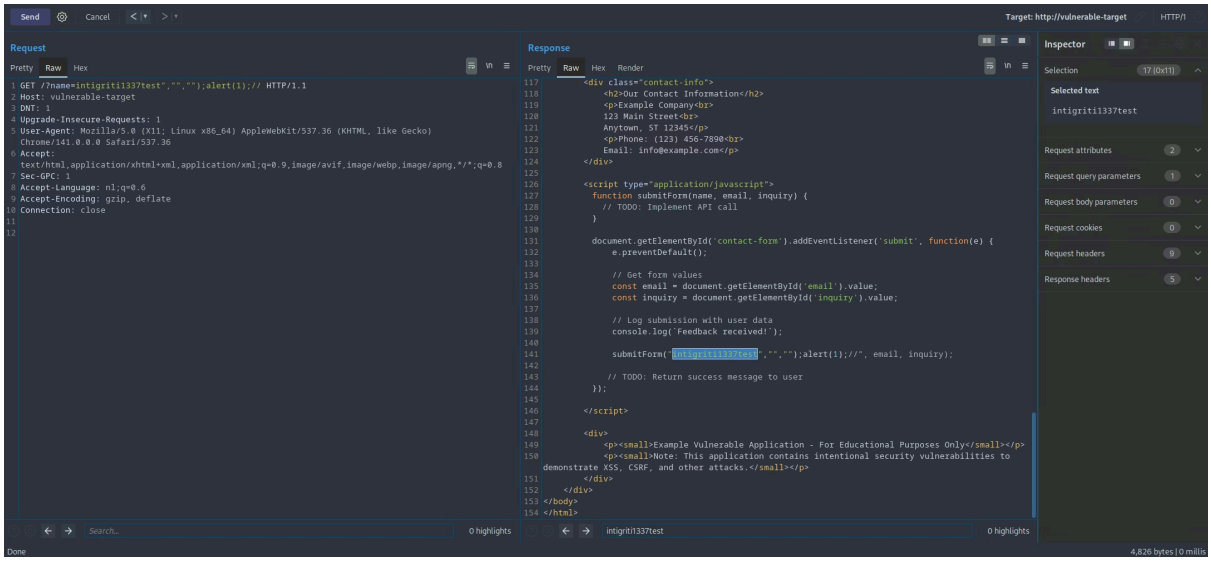
XSS in JavaScript context

As applications become more complex, developers tend to introduce more security weaknesses, including XSS vulnerabilities. While this context occurs a lot less, sometimes developers reflect unsanitized user input right inside the JavaScript context. Often unaware of the consequences, we can

use this opportunity to escape the context and inject our arbitrary JavaScript code without making use of any HTML.

Depending on the reflection point, you'll usually need to:

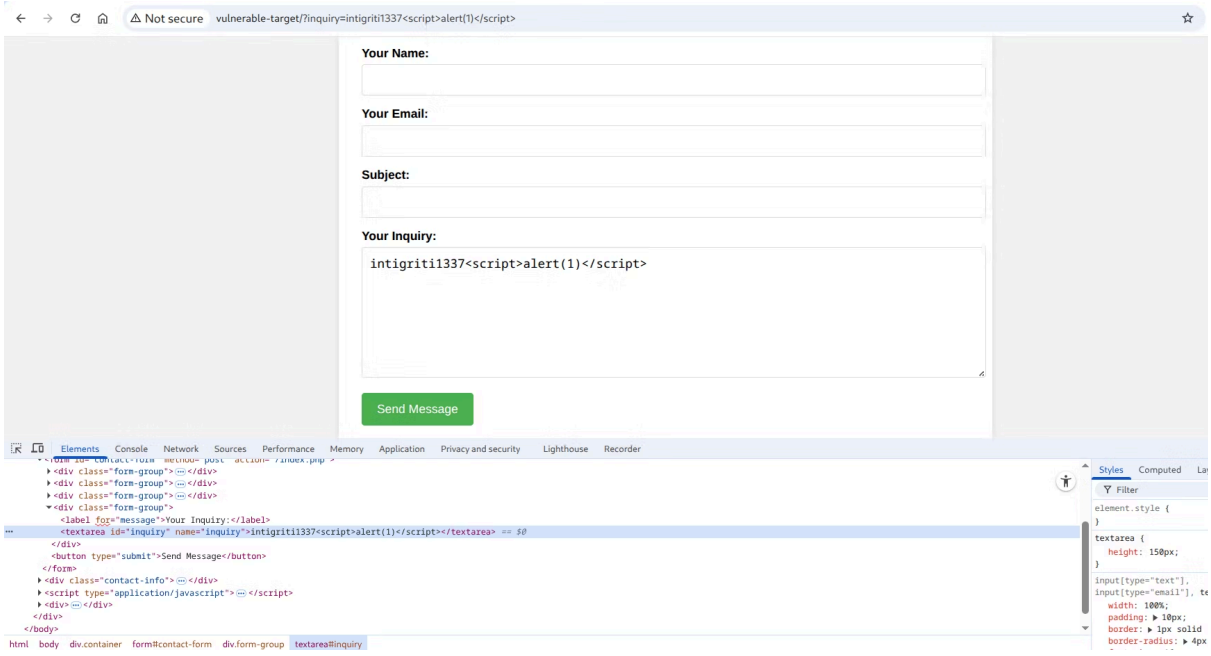
1. Escape out of the current context (often a variable value, or function parameter)
2. Inject your payload
3. Close your payload to make the syntax match



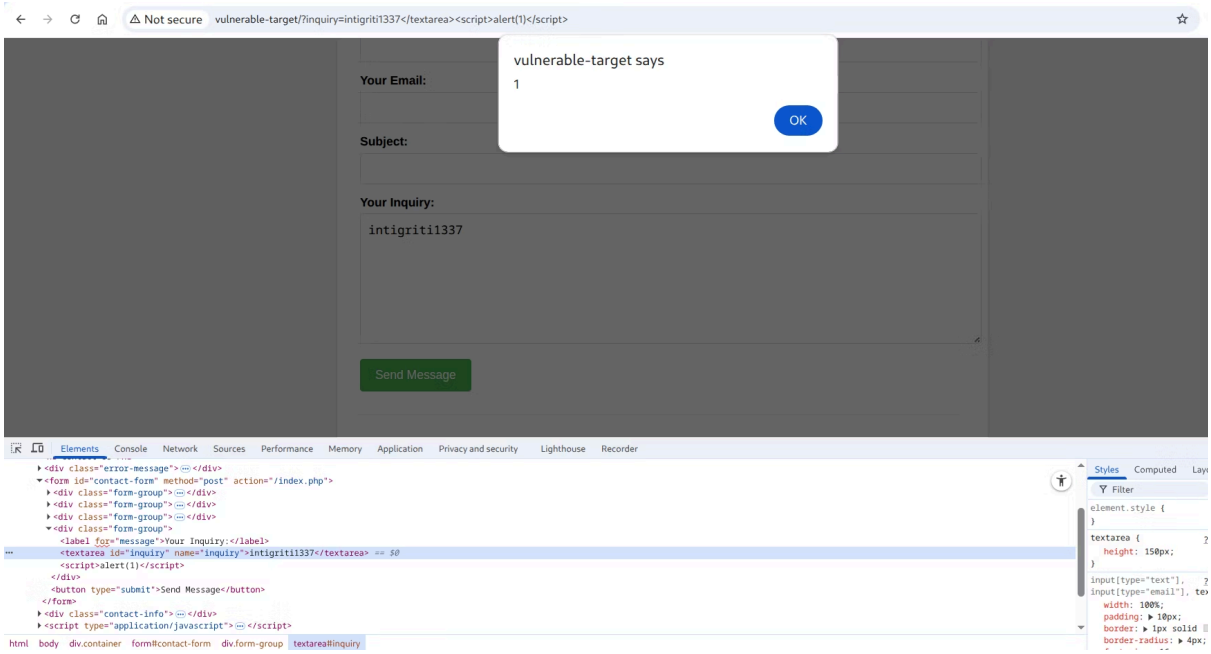
Example 3: Basic cross-site scripting (XSS) within a JavaScript code block

XSS inside the textarea field

Browsers will, with some HTML tags, such as the **<textarea>** HTML tag, refuse to render their value to preserve the contents. If you're a beginner, you might at first deem this scenario to be unexploitable. However, if your input ever gets reflected within any of these tags, we'll be forced to close the tag before injecting our payload.



Example 4: Basic cross-site scripting (XSS) within a Textarea context



Example 4: Basic cross-site scripting (XSS) within a Textarea context

XSS with limited HTML & attributes allowed

A common scenario you will come across is one where input is limited, filtered, or blocked altogether when a specific pattern is matched. This is by far one of the most used filtering methods that developers tend to rely upon.

Fortunately for us, we can, in most cases, simply fuzz for any allowed HTML tags, event handlers, and characters to help us evade the filter or web application firewall (WAF) block.

```php
<?php
function sanitizer($input) {
    $allowed_tags = ['div', 'span', 'img', 'input', 'form', 'a', 'style', 'button'];
    $allowed_attributes = ['name', 'src', 'class', 'id', 'type'];

    preg_match_all('/<(\/?)(.*?)(\s|>)/i', $input, $tags);
    foreach ($tags[2] as $tag) {
        if (!in_array(strtolower($tag), $allowed_tags)) {
            return "BLOCKED";
        }
    }

    preg_match_all('/[\s\n\r\t]+([\w-]+)=\s*(["\'\`]|[^\s>]*)/i', urldecode($input), $attrs);
    foreach ($attrs[1] as $attr) {
        if (!in_array(strtolower($attr), $allowed_attributes)) {
            return "BLOCKED";
        }
    }

    return $input;
}

echo "<div class='container'>Welcome back," . sanitizer($_GET["name"]) . "!</div>";
?>
```

Example 5: Basic cross-site scripting (XSS) with limited HTML tags and attributes

Fuzzing for XSS

Have you tried everything in your defense to make the target execute your XSS payload, but couldn't manage to evade the filter? No worries, this is a strong indication to fuzz for XSS!

PortSwigger Research Academy provides a cheat sheet with all available HTML tags and attributes. We can use this list to practically enumerate all accepted tags and event handlers by probing our injection point. Afterward, all we have to do is match our puzzle pieces by injecting an allowed HTML tag and an attribute to help us craft our payload!

https://portswigger.net/web-security/cross-site-scripting/cheat-sheet

Conclusion

The key takeaway from this article is identifying the context of your injection point and understanding the way your target behaves to malicious characters. Whenever testing targets that deploy strict filter rules, it's recommended to take the time to manually test your reflection points. Some tools can easily fail to spot these and can therefore not always be relied upon.

So, you've just learned something new about hunting for reflected XSS vulnerabilities... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs and CTFs or... browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe earn a bounty on your next submission!