XSS- Stored XSS

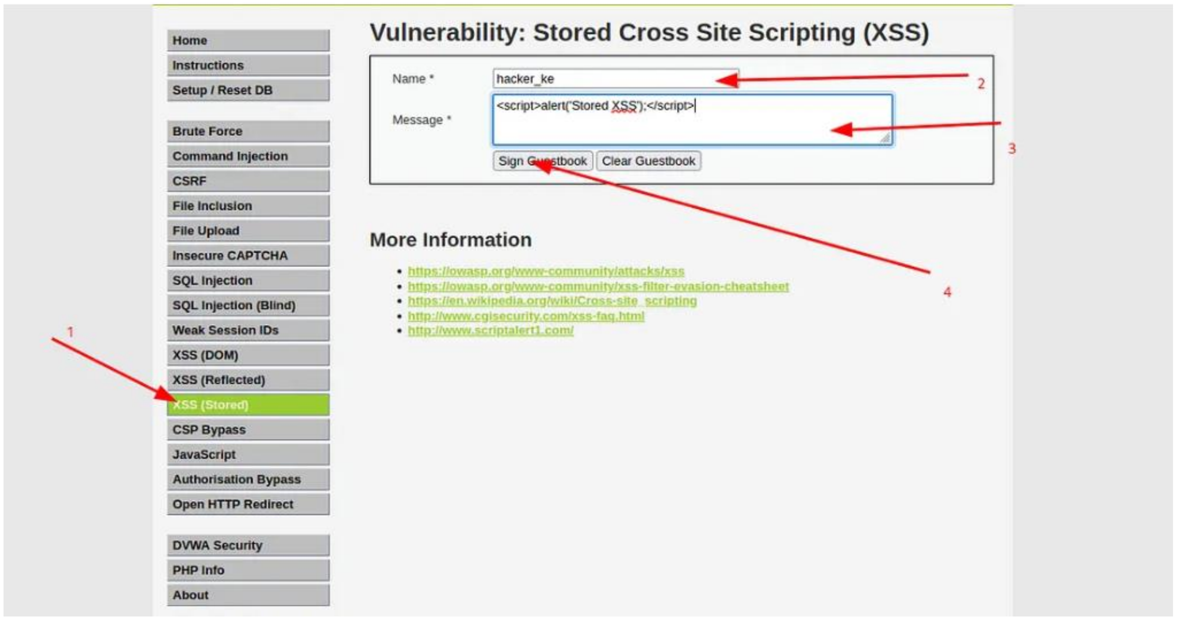
A Cross-Site Scripting (XSS) attack is a type of security vulnerability commonly found in web applications. It occurs when an attacker injects malicious scripts into content from otherwise trusted websites. The attack is made possible when a web application accepts untrusted inputs and sends them to a web browser without proper validation or escaping. The injected scripts can then be executed in the context of the user's browser, allowing the attacker to steal session cookies, deface websites, or redirect the user to malicious sites. XSS attacks exploit the trust that users have in a website and can lead to significant security breaches if not properly mitigated.

Stored Cross-Site Scripting (Stored XSS) is a type of XSS attack where the malicious script is permanently stored on the target server, such as in a database, message forum, visitor log, or comment field. Unlike Reflected XSS, where the payload is reflected off a web server, Stored XSS payloads are directly injected into a website's stored data. When a user requests the infected page, the malicious script is served as part of the page and executed in the user's browser. This allows attackers to steal sensitive information, hijack user sessions, or deface websites, exploiting the trust users have in the content served by the website. Stored XSS can be particularly damaging because the payload persists on the server, affecting any user who views the compromised content.

Examples: -

➢ Low Security

1. Firstly, input a name of your choice into the designated field. For the message field, insert the payload below before clicking the "Sign Guestbook" option.

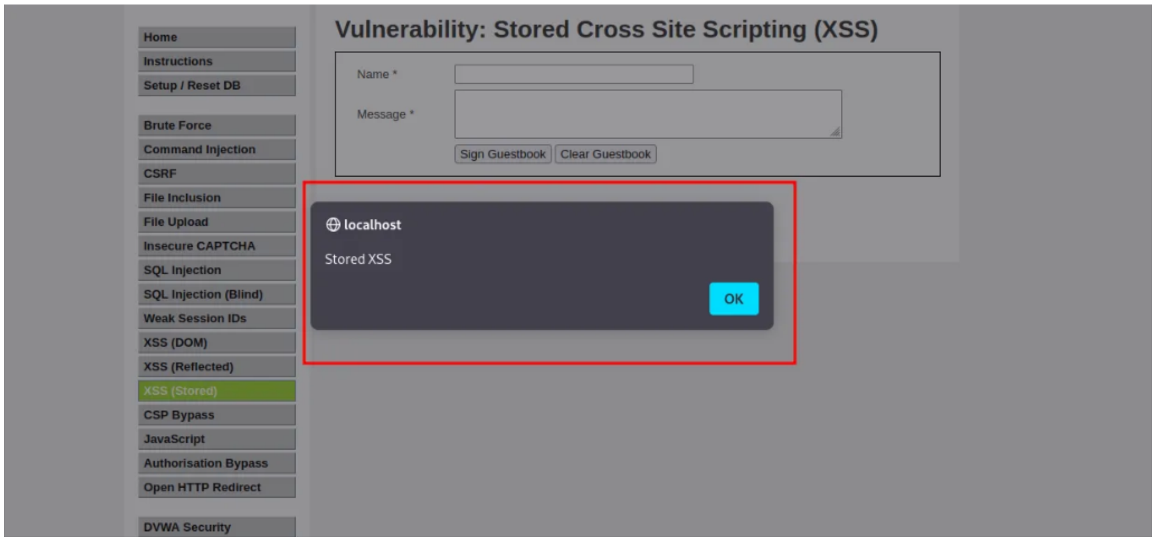


Vulnerability: Stored Cross Site Scripting (XSS)

Name * hacker_ke

Message * `<script>alert('Stored XSS');</script>`

2. After this, an alert box appeared, confirming the vulnerability of the site to stored XSS attacks.



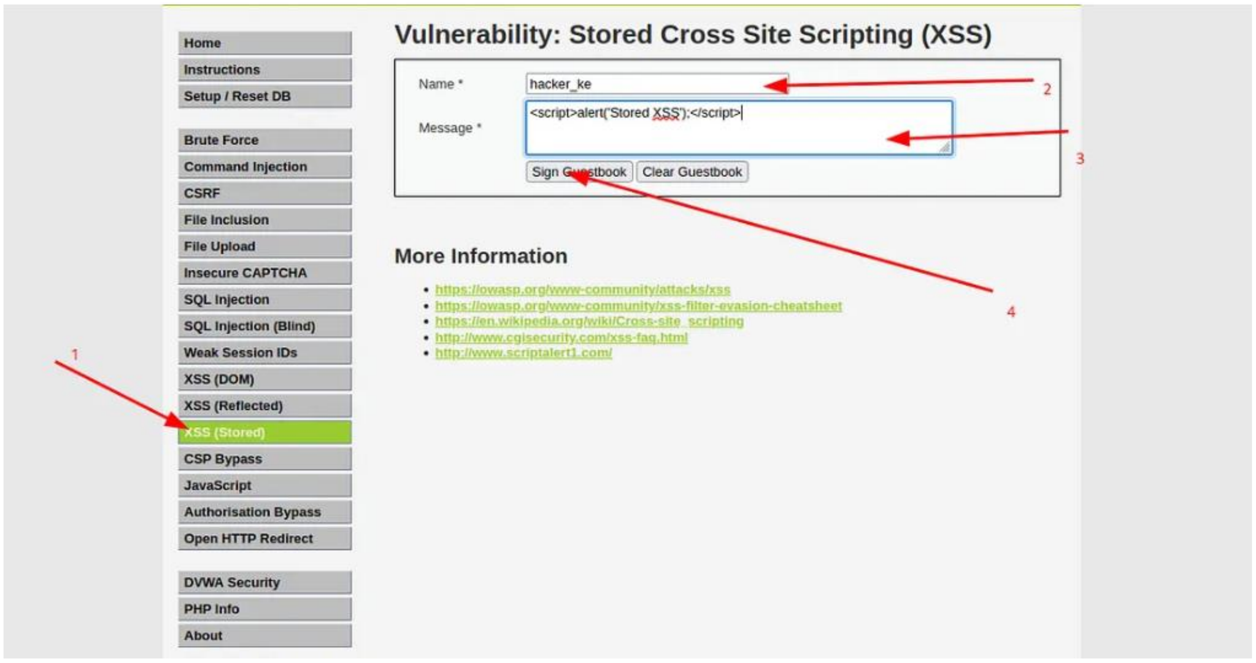
Vulnerability: Stored Cross Site Scripting (XSS)

localhost

Stored XSS

➢ Medium Security

1. Upon re-entering the previous inputs, it becomes evident that no pop-up is triggered, even though the XSS payload is correct. This could be attributed to various reasons, such as input sanitization being implemented in the source code, which may include processes like removing special characters or employing other security measures. It's important to note that without access to the source code, we can only speculate on the specific mechanisms in place.



Vulnerability: Stored Cross Site Scripting (XSS)

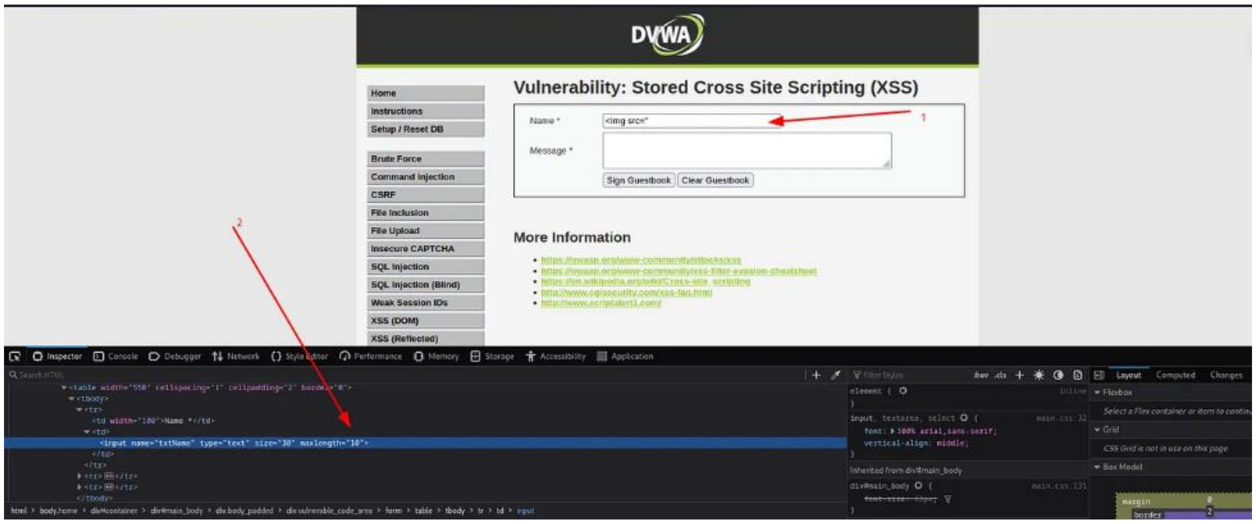
Name * hacker_ke

Message * <script>alert('Stored XSS');</script>

More Information

- https://owasp.org/www-community/attacks/xss
- https://owasp.org/www-community/xss-filter-evasion-cheatsheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

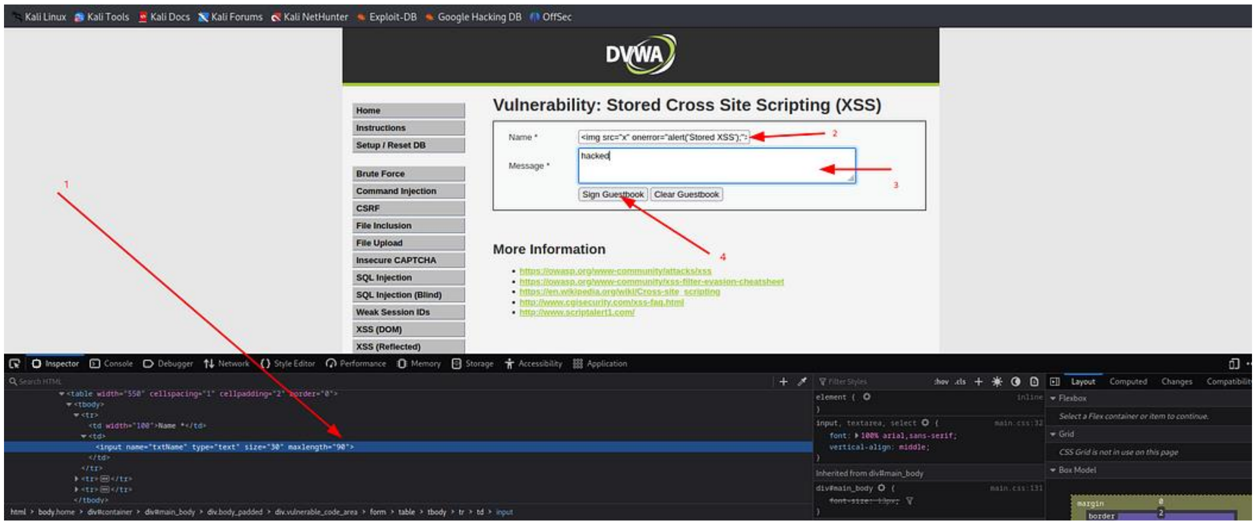
2. I attempted to input the payload into the name field, but encountered client-side restrictions that limit users to entering a maximum of 10 characters.



Name field: `<img src="`

```
<table width="550" cellspacing="1" cellpadding="2" border="0">
  <tbody>
    <tr>
      <td width="100">Name *</td>
      <td>
        <input name="txtName" type="text" size="30" maxlength="10">
      </td>
    </tr>
    <tr>…</tr>
    <tr>…</tr>
  </tbody>
html > body.home > div#container > div#main_body > div.body_padded > div.vulnerable_code_area > form > table > tbody > tr > td > input
```

3. To bypass this limitation, right-click on the name input field, select "Inspect," and then modify the `maxlength="10"` attribute to a different value, such as `"90"`. After making this adjustment, press Enter to apply the changes.

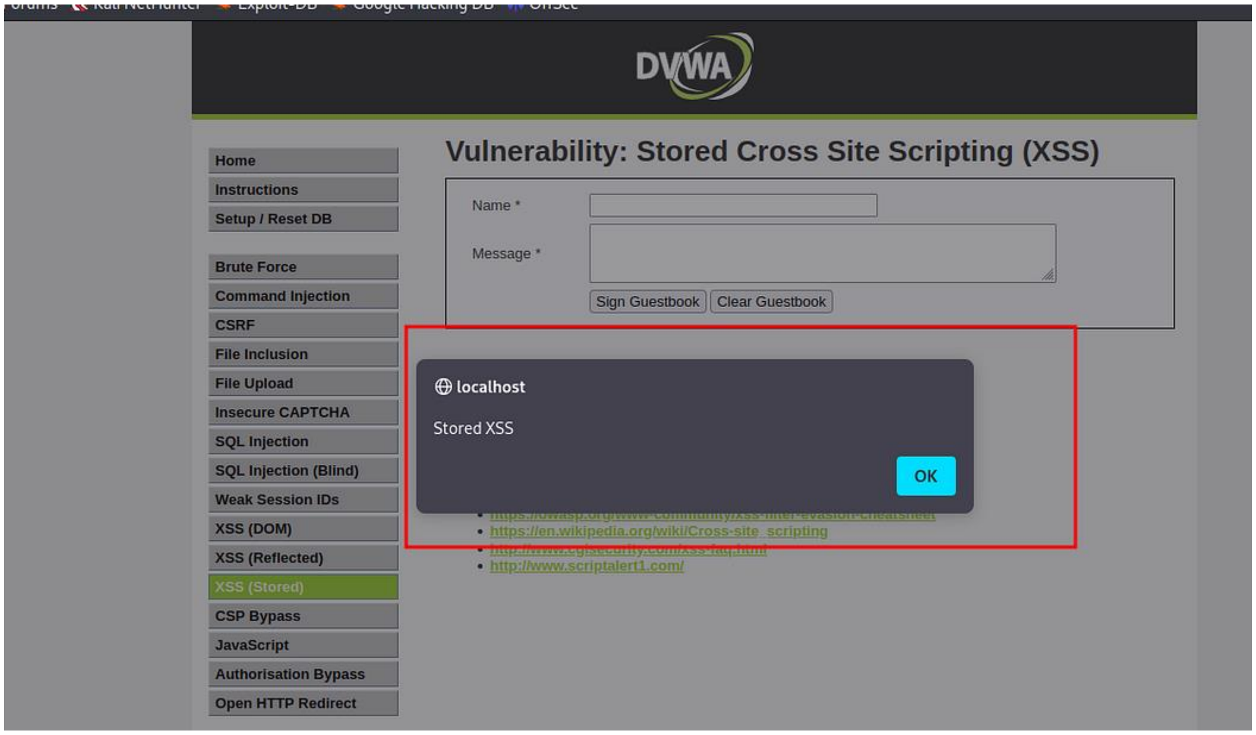


Name field: `<img src="x" onerror="alert('Stored XSS');">`

Message field: `hacked`

```
<table width="550" cellspacing="1" cellpadding="2" border="0">
  <tbody>
    <tr>
      <td width="100">Name *</td>
      <td>
        <input name="txtName" type="text" size="30" maxlength="90">
      </td>
    </tr>
    <tr>…</tr>
    <tr>…</tr>
  </tbody>
html > body.home > div#container > div#main_body > div.body_padded > div.vulnerable_code_area > form > table > tbody > tr > td > input
```

4. Now, input the payload into the name field and enter any text in the message field. Afterward, click on the submit button.



Vulnerability: Stored Cross Site Scripting (XSS)

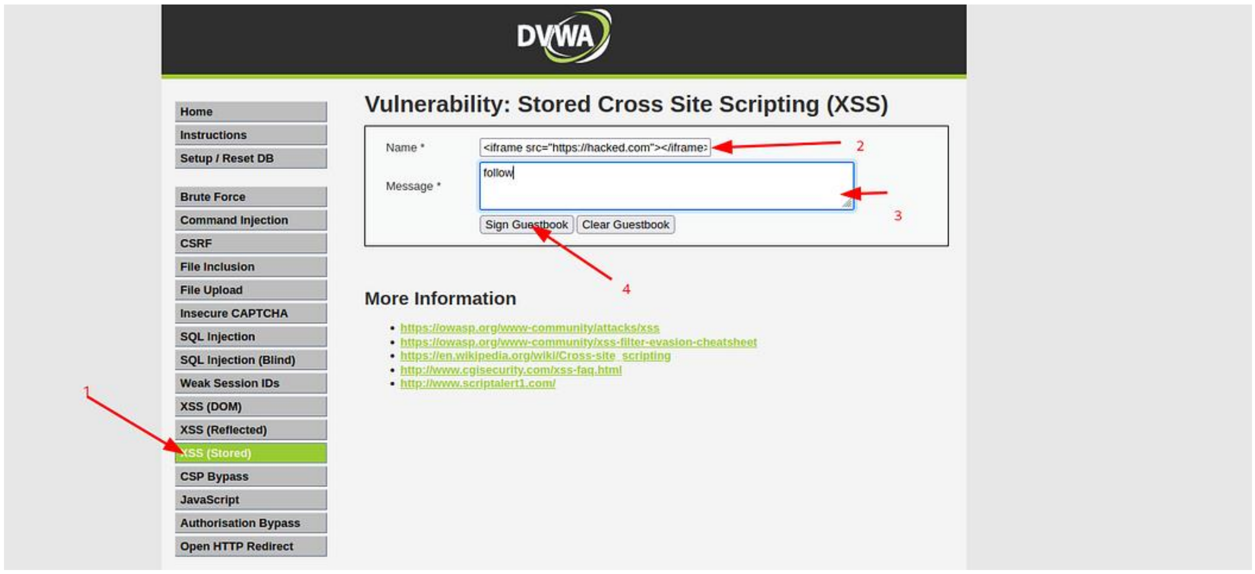
localhost

Stored XSS

- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

➢ High Security

1. Since the field has been sanitized to exclude special characters, our payload, which does not include the <*script*> tag, can still be effective. Let’s proceed by injecting the payload into the name field and entering arbitrary text into the message field. Afterward, submit the form to observe the outcome.



Vulnerability: Stored Cross Site Scripting (XSS)

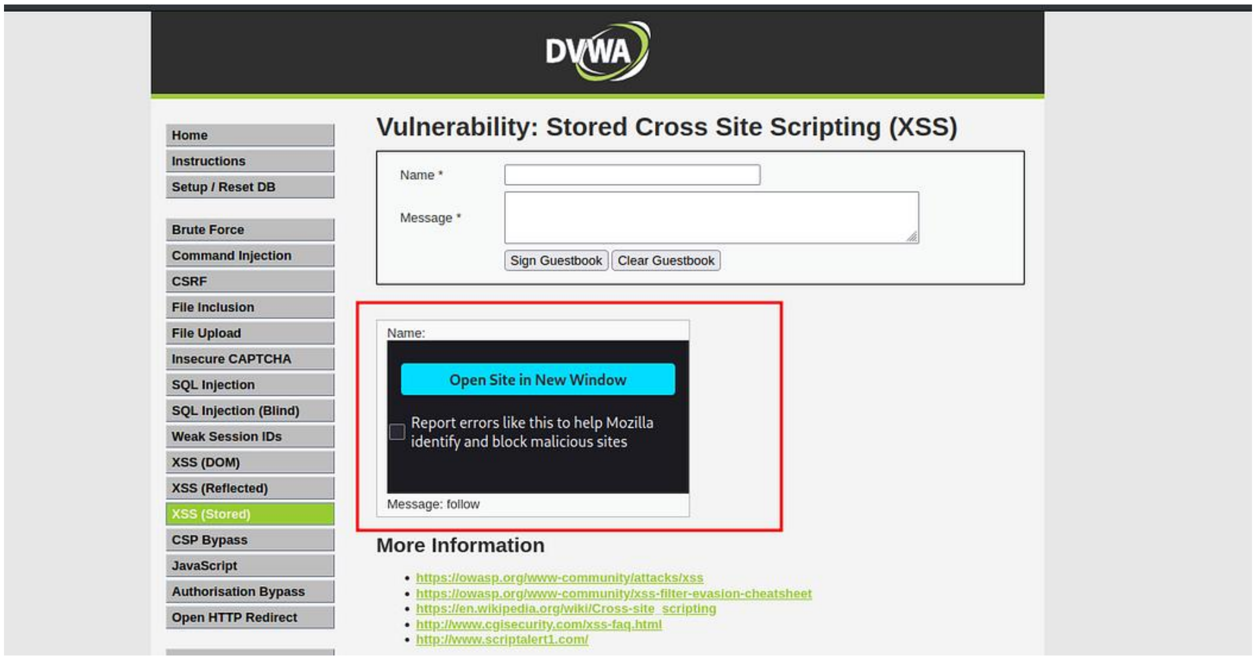
Name * `<iframe src="https://hacked.com"></iframe>`

Message * follow

More Information

- https://owasp.org/www-community/attacks/xss
- https://owasp.org/www-community/xss-filter-evasion-cheatsheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/

2. I utilized the <*iframe*> tag, which is employed to embed one web page within another web page. Based on the obtained results, it is evident that the webpage has indeed been successfully embedded. If a user clicks on the link, they will be directed to the webpage specified in the script. This outcome unequivocally confirms the successful exploitation of this vulnerability at high security level.



Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

Name:

Report errors like this to help Mozilla identify and block malicious sites

Message: follow

More Information

- https://owasp.org/www-community/attacks/xss
- https://owasp.org/www-community/xss-filter-evasion-cheatsheet
- https://en.wikipedia.org/wiki/Cross-site_scripting
- http://www.cgisecurity.com/xss-faq.html
- http://www.scriptalert1.com/