10: XSS, Cross-Site Request Forgery, Clickjacking

10-04-2022
Web Security Trivia!
How severe are Cross-Site Scripting (XSS) and Cross-Site Request Forgery Attacks (CSRF)?

A. XSS and CSRF: Top 5 most severe attacks
B. XSS and CSRF: Top 10 most severe attacks
C. XSS and CSRF: Top 100 most severe attacks
D. These are old school – we now know how to protect against such attacks
# How severe are Cross-Site Scripting and Cross-Site Request Forgery Attacks?

<table>
<thead>
<tr>
<th>Rank</th>
<th>ID</th>
<th>Name</th>
<th>Score</th>
<th>KEV Count (CVEs)</th>
<th>Rank Change vs. 2021</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CWE-287</td>
<td>Out-of-bounds Write</td>
<td>64.20</td>
<td>62</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>CWE-79</td>
<td>Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
<td>45.97</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>CWE-89</td>
<td>Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
<td>22.11</td>
<td>7</td>
<td>+3 ▲</td>
</tr>
<tr>
<td>4</td>
<td>CWE-20</td>
<td>Improper Input Validation</td>
<td>20.63</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>CWE-125</td>
<td>Out-of-bounds Read</td>
<td>17.67</td>
<td>1</td>
<td>-2 ▼</td>
</tr>
<tr>
<td>6</td>
<td>CWE-78</td>
<td>Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')</td>
<td>17.53</td>
<td>32</td>
<td>-1 ▼</td>
</tr>
<tr>
<td>7</td>
<td>CWE-416</td>
<td>Use After Free</td>
<td>15.50</td>
<td>28</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>CWE-22</td>
<td>Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')</td>
<td>14.08</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>CWE-352</td>
<td>Cross-Site Request Forgery (CSRF)</td>
<td>11.53</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>CWE-434</td>
<td>Unrestricted Upload of File with Dangerous Type</td>
<td>9.56</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>CWE-476</td>
<td>NULL Pointer Dereference</td>
<td>7.15</td>
<td>0</td>
<td>+4 ▲</td>
</tr>
<tr>
<td>12</td>
<td>CWE-502</td>
<td>Deserialization of Untrusted Data</td>
<td>6.68</td>
<td>7</td>
<td>+1 ▲</td>
</tr>
<tr>
<td>13</td>
<td>CWE-190</td>
<td>Integer Overflow or Wraparound</td>
<td>6.53</td>
<td>2</td>
<td>-1 ▼</td>
</tr>
<tr>
<td>14</td>
<td>CWE-287</td>
<td>Improper Authentication</td>
<td>6.35</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>CWE-798</td>
<td>Use of Hard-coded Credentials</td>
<td>5.66</td>
<td>0</td>
<td>+1 ▲</td>
</tr>
<tr>
<td>16</td>
<td>CWE-862</td>
<td>Missing Authorization</td>
<td>5.53</td>
<td>1</td>
<td>+2 ▲</td>
</tr>
<tr>
<td>17</td>
<td>CWE-77</td>
<td>Improper Neutralization of Special Elements used in a Command ('Command Injection')</td>
<td>5.42</td>
<td>5</td>
<td>+8 ▲</td>
</tr>
<tr>
<td>18</td>
<td>CWE-306</td>
<td>Missing Authentication for Critical Function</td>
<td>5.15</td>
<td>6</td>
<td>-7 ▼</td>
</tr>
<tr>
<td>19</td>
<td>CWE-119</td>
<td>Improper Restriction of Operations within the Bounds of a Memory Buffer</td>
<td>4.85</td>
<td>6</td>
<td>-2 ▼</td>
</tr>
<tr>
<td>20</td>
<td>CWE-276</td>
<td>Incorrect Default Permissions</td>
<td>4.84</td>
<td>0</td>
<td>-1 ▼</td>
</tr>
<tr>
<td>21</td>
<td>CWE-918</td>
<td>Server-Side Request Forgery (SSRF)</td>
<td>4.27</td>
<td>8</td>
<td>+3 ▲</td>
</tr>
<tr>
<td>22</td>
<td>CWE-362</td>
<td>Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')</td>
<td>3.57</td>
<td>6</td>
<td>+11 ▲</td>
</tr>
<tr>
<td>23</td>
<td>CWE-400</td>
<td>Uncontrolled Resource Consumption</td>
<td>3.56</td>
<td>2</td>
<td>+4 ▲</td>
</tr>
<tr>
<td>24</td>
<td>CWE-611</td>
<td>Improper Restriction of XML External Entity Reference</td>
<td>3.38</td>
<td>0</td>
<td>-1 ▼</td>
</tr>
<tr>
<td>25</td>
<td>CWE-94</td>
<td>Improper Control of Generation of Code ('Code Injection')</td>
<td>3.32</td>
<td>4</td>
<td>+3 ▲</td>
</tr>
</tbody>
</table>
If you use Do Not Track in your browser, will that ensure that no third-party cookies are set?

A. Yes  
B. No
Sending Data Over HTTP

Four ways to send data to the server

1. Embedded in the URL (typically URL encoded, but not always)
2. In cookies (cookie encoded)
3. Inside a custom HTTP request header
4. In the HTTP request body

Examples

a. GET /purchase.html?user=alice&item=iPad&price=400 HTTP/1.1
b. Cookie: user=alice; item=iPad; price=400;
c. BODY of HTTP POST
   user=alice&item=iPad&price=400
d. My-Custom-Header: alice/iPad/400

Clicker Options
A. 1->a, 2->b, 3->c, 4->d
B. 1->d, 2->c, 3->b, 4->a
C. 1->a, 2->b, 4->c, 3->d
D. 1->a, 3->b, 2->c, 4->d
Each browser window or frame:
  ○ The browser receives HTML, CSS, and JavaScript from the server
  ○ HTML and CSS are parsed into a DOM (Document Object Model)
  ○ JavaScript is interpreted and executed, possibly modifying the DOM
    ○ Responds to events

Events
  ○ User actions: OnClick, OnMouseover
  ○ Rendering: OnLoad, OnUnload
  ○ Timing: setTimeout(), clearTimeout()
Last Time HTTP:

http://www.cs.swarthmore.edu/~chaganti/index.html

Protocol: ftp, https, tor

Hostname/server • translated to an IP address by DNS

Path to the resource

index.html is static content i.e., a fixed file returned by the server
Last Time: HTTP

## Anatomy of Request

### HTTP Request

<table>
<thead>
<tr>
<th>method</th>
<th>path</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/index.html</td>
<td>HTTP/1.1</td>
</tr>
</tbody>
</table>

- **headers**
  - Accept: image/gif, image/x-bitmap, image/jpeg, */*
  - Accept-Language: en
  - Connection: Keep-Alive
  - User-Agent: Mozilla/1.22 (compatible; MSIE 2.0; Windows 95)
  - Host: www.example.com
  - Referer: http://www.google.com?q=dingbats

- **body**
  - (empty)
HTTP Response

HTTP/1.0 200 OK

Date: Sun, 21 Apr 1996 02:20:42 GMT
Server: Microsoft-Internet-Information-Server/5.0
Content-Type: text/html
Last-Modified: Thu, 18 Apr 1996 17:39:05 GMT
Content-Length: 2543
Set-Cookie: aldkfj2314

<html>Some data... announcement! ... </html>
Goals of Web Security: Safely Browse the Web

• Safe to visit an evil website
  • sandboxing Javascript
  • privilege separation

• Safe to visit two pages at the same time,
  • same-origin policy

• Safe delegation
Same Origin Policy

• rule that prevents one website from tampering with other unrelated websites.
  • enforced by browser
Every webpage has an **origin** defined by its URL with three parts:

- **Protocol**: The protocol in the URL
- **Domain**: The domain in the URL’s location
- **Port**: The port in the URL’s location
  - If no port is specified, the default is 80 for HTTP and 443 for HTTPS

**https://cs.swarthmore.edu:443/assets/lock.PNG**

**http://cs.swarthmore.edu/assets/images/404.png**

80 (default port)
Bounding Origins — Windows

Every Window and Frame has an origin.
Origins are blocked from accessing other origin’s objects.

attacker.com cannot...

- read or write content from bank.com tab
- read or write bank.com’s cookies
- detect that the other tab has bank.com loaded
Bounding Origins — Frames

Every Window and Frame has an origin
Origins are blocked from accessing other origin’s objects

attacker.com cannot…
- read content from bank.com frame
- access bank.com's cookies
- detect that has bank.com loaded
**Same-Origin Policy**

- Two webpages have the same origin *if and only if* the protocol, domain, and port of the URL all match exactly: string matching:
  - The *protocol*, *domain*, and *port* strings must be equal

<table>
<thead>
<tr>
<th>First domain</th>
<th>Second domain</th>
<th>Same origin?</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://cs88.swat.org">http://cs88.swat.org</a></td>
<td><a href="https://cs88.swat.org">https://cs88.swat.org</a></td>
<td></td>
</tr>
<tr>
<td><a href="http://cs88.swat.org">http://cs88.swat.org</a></td>
<td><a href="http://swat.org">http://swat.org</a></td>
<td></td>
</tr>
</tbody>
</table>
Same-Origin Policy

- Two webpages have the same origin if and only if the protocol, domain, and port of the URL all match exactly: string matching:
  - The **protocol**, **domain**, and **port** strings must be equal

<table>
<thead>
<tr>
<th>First domain</th>
<th>Second domain</th>
<th>Same origin?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>http://cs88.swat.org</code></td>
<td><code>https://cs88.swat.org</code></td>
<td>Same origin?</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>http ≠ https</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>swat.cs88.org ≠ cs88.org</code></td>
</tr>
</tbody>
</table>
Same-Origin Policy: Two websites with different origins can’t interact with each other.

Example: If cs88.org embeds google.com, the inner frame cannot interact with the outer frame, and the outer frame cannot interact with the inner-frame

- So what happens when...
  - JavaScript runs with the origin of the page that loads it?
    - E.g., cs88.org fetches Javascript from Google analytics.
  - Websites fetch and display images from other origins?
    - E.g. if we include `<img src="http://google.com/logo.jpg" on http://cs.swarthmore.edu, the image has origin http://google.com.
  - We load frames such as `<iframe src="http://google.com"></iframe>` on http://cs.swarthmore.edu?
Same-Origin Policy

- Two websites with different origins cannot interact with each other
  - Example: If cs88.org embeds google.com, the inner frame cannot interact with the outer frame, and the outer frame cannot interact with the inner-frame
- Exception: JavaScript runs with the origin of the page that loads it
  - Example: If cs88.org fetches JavaScript from google.com, the JavaScript has the origin of cs88.org
  - Intuition: cs88.org has “copy-pasted” JavaScript onto its webpage
- Exception: Websites can fetch and display images from other origins
  - However, the website only knows about the image’s size and dimensions (cannot actually manipulate the image)
- Exception: Websites can agree to allow some limited sharing
  - Cross-origin resource sharing (CORS)
  - The postMessage function in JavaScript
Same-Origin Policy: Summary

- Rule enforced by the browser: **Two websites with different origins cannot interact with each other**
- Two webpages have the same origin *if and only if* the protocol, domain, and port of the URL all match exactly (string matching)
- Exceptions
  - JavaScript runs with the origin of the page that loads it
  - Websites can fetch and display images from other origins
  - Websites can agree to allow some limited sharing
Cookie Policy
Cookie Policy

- **Cookie policy**: A set of rules enforced by the browser
  - When the browser receives a cookie from a server, should the cookie be accepted?
  - When the browser makes a request to a server, should the cookie be attached?

- **Cookie policy is not the same as same-origin policy**
Login Session

GET /loginform HTTP/1.1
cookies: []
Login Session

GET /loginform HTTP/1.1
cookies: []

HTTP/1.1 200 OK
cookies: []
<html><form>...</form></html>
Login Session

GET /loginform HTTP/1.1
cookies: []

POST /login HTTP/1.1
cookies: []
username: chaganti
password: swarthmore

HTTP/1.1 200 OK
cookies: []
<html><form>...</form></html>
Login Session

GET /loginform HTTP/1.1
cookies: []

POST /login HTTP/1.1
cookies: []
username: chaganti
password: swarthmore

GET /account HTTP/1.1
cookies: [session: e82a7b92]
Login Session

GET /loginform HTTP/1.1
cookies: []

POST /login HTTP/1.1
cookies: []
username: chaganti
password: swarthmore

GET /account HTTP/1.1
cookies: [session: e82a7b92]

GET /img/user.jpg HTTP/1.1
cookies: [session: e82a7b92]
Can the following attack succeed?

If we have a google analytics Javascript running on bank.com’s login page. Assume that the site has no frames, and everything on this page has the same origin. Can google analytics see Alice’s session cookie on bank.com?

A. Yes  B. No  C. Maybe  D. Something Else
Can the following attack succeed?

If we have a google analytics Javascript running on bank.com’s login page. Assume that the site has no frames, and everything on this page has the same origin. Can google analytics see Alice’s session cookie on bank.com?

A. Yes!     B. No     C. Maybe     D. Something Else
Cookies

“In scope” cookies are sent based on origin regardless of requester

POST /login

GET /img/user.jpg

GET /img/user.jpg
Aside: Domain Hierarchy

- Domains
  - Located after the double slashes, but before the next single slash
  - Written as several phrases separated by dots
- Domains can be sorted into a hierarchy
  - The hierarchy is separated by dots

```
. (root)
  .edu
  .org
  .com
  swarthmore.edu
  mit.edu
  cs88.org
  piazza.com
  google.com
```
Aside: Domain Hierarchy

. (root)

.edu

swarthmore.edu

.cs.swarthmore.edu

.edu is a top-level domain (TLD), because it is directly below the root of the tree.

swarthmore.edu is a subdomain of edu

.cs.swarthmore.edu is a subdomain of swarthmore.edu
Cookie Policy: Setting Cookies

- When the browser receives a cookie from a server, should the cookie be accepted?
- Server with domain $X$ can set a cookie with domain attribute $Y$ if
  - The domain attribute is a domain suffix of the server’s domain
    - $X$ ends in $Y$
    - $X$ is below or equal to $Y$ on the hierarchy
    - $X$ is more specific or equal to $Y$
  - The domain attribute $Y$ is not a top-level domain (TLD)
  - No restrictions for the Path attribute (the browser will accept any path)
- Examples:
  - `mail.google.com` can set cookies for Domain=`google.com`
  - `google.com` can set cookies for Domain=`google.com`
  - `google.com` cannot set cookies for Domain=`com`, because com is a top-level domain
Cookie Policy: Sending Cookies

- When the browser makes a request to a server, should the cookie be attached?
- The browser sends the cookie if both of these are true:
  - The **domain attribute** is a *domain suffix* of the *server’s domain*
  - The **path attribute** is a **prefix** of the *server’s path*
Quick method to check cookie sending:
Concatenate the cookie domain and path. Line it up below the requested URL at the first single slash.

If the domains and paths all match, then the cookie is sent.
Quick method to check cookie sending:
Concatenate the cookie domain and path. Line it up below the requested URL at the first single slash.

If the domain or path doesn’t match, then the cookie is not sent.
**Scoping Example**

<table>
<thead>
<tr>
<th>Cookie 1</th>
<th>Cookie 2</th>
<th>Cookie 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>name = cookie1</td>
<td>name = cookie2</td>
<td>name = cookie3</td>
</tr>
<tr>
<td>value = a</td>
<td>value = b</td>
<td>value = c</td>
</tr>
<tr>
<td>domain = login.site.com</td>
<td>domain = site.com</td>
<td>domain = site.com</td>
</tr>
<tr>
<td>path = /</td>
<td>path = /</td>
<td>path = /my/home</td>
</tr>
</tbody>
</table>

Concatenate the cookie domain and path. Line it up below the requested URL at the first single slash.
### Scoping Example

<table>
<thead>
<tr>
<th></th>
<th>Cookie 1</th>
<th>Cookie 2</th>
<th>Cookie 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkout.site.com</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>login.site.com</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>login.site.com/my/home</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>site.com/account</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Concatenate the cookie domain and path. Line it up below the requested URL at the first single slash.

- **Cookie 1**: name = cookie1, value = a, domain = login.site.com, path = /
- **Cookie 2**: name = cookie2, value = b, domain = site.com, path = /
- **Cookie 3**: name = cookie3, value = c, domain = site.com, path = /my/home
Can the following attack succeed?

If we have a google analytics Javascript running on bank.com’s login page. Assume that the site has no frames, and everything on this page has the same origin. Can google analytics see Alice’s session cookie on bank.com?

No. Cookie Policy: Domain and Path not the same!
Session Tokens: Security

- If an attacker steals your session token, they can log in as you!
  - The attacker can make requests and attach your session token
  - The browser will think the attacker’s requests come from you
- Servers need to generate session tokens *randomly and securely*
- Browsers need to make sure *malicious websites cannot steal session tokens*
  - Enforce isolation with *cookie policy and same-origin policy*
- Browsers should not send session tokens to the wrong websites
  - Enforced by *cookie policy*
Session Token Cookie Attributes

What attributes should the server set for the session token?

- **Domain and Path:** Set so that the cookie is only sent on requests that require authentication
- **Secure:** Can set to True so the cookie is only sent over secure HTTPS connections
- **HttpOnly:** Can set to True so JavaScript can’t access session tokens
- **Expires:** Set so that the cookie expires when the session times out

<table>
<thead>
<tr>
<th>Name</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>{random value}</td>
</tr>
<tr>
<td>Domain</td>
<td>mail.google.com</td>
</tr>
<tr>
<td>Path</td>
<td>/</td>
</tr>
<tr>
<td>Secure</td>
<td>True</td>
</tr>
<tr>
<td>HttpOnly</td>
<td>True</td>
</tr>
<tr>
<td>Expires</td>
<td>{15 minutes later}</td>
</tr>
</tbody>
</table>

(Other fields omitted)
Cross-Site Request Forgery
Review: Cookies and Session Tokens

- Session token cookies are used to associate a request with a user
- The browser automatically attaches relevant cookies in every request
What if the attacker tricks the victim into making an unintended request to a legitimate website?

A. The victim’s browser will automatically attach relevant cookies
B. The victim’s browser will block sending the cookies because of the same-origin policy
C. The victim’s browser will block sending the cookies because of the cookie policy
D. Something else
Cross-Site Request Forgery (CSRF)

- Idea: What if the attacker tricks the victim into making an unintended request?
  - The victim’s browser will automatically attach relevant cookies
  - The server will think the request came from the victim!
- Cross-site request forgery (CSRF or XSRF): An attack that exploits cookie-based authentication to perform an action as the victim
Steps of a CSRF Attack

1. User authenticates to the server
   ○ User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious request to the server
3. The server accepts the malicious request from the victim
   ○ Recall: The cookie is automatically attached in the request
Steps of a CSRF Attack

1. User authenticates to the server
   ○ User receives a cookie with a valid session token
2. Attacker tricks the victim into making a malicious request to the server: how?

   https://www.bank.com/transfer?amount=100&to=Mallory

A. GET Request
B. POST Request
C. Put some JavaScript on a website the victim will visit
D. Some combination of the above
Executing a CSRF Attack

- How might we trick the victim into making a POST request?
  - Example POST request: Submitting a form
- Strategy #1: Trick the victim into clicking a link
  - Note: Clicking a link in your browser makes a GET request, not a POST request, so the link cannot directly make the malicious POST request
  - The link can open an attacker’s website, which contains some JavaScript that makes the actual malicious POST request
- Strategy #2: Put some JavaScript on a website the victim will visit
  - Example: Pay for an advertisement on the website, and put JavaScript in the ad
  - Recall: JavaScript can make a POST request
Defense: CSRF Tokens

- **Idea:** Add a secret value in the request that the attacker doesn’t know
  - The server **only accepts requests if it has a valid secret**
  - Attacker can’t create a malicious request without knowing the secret

- **CSRF token:** A secret value provided by the server to the user. The user must attach the same value in the request for the server to accept the request.
  - CSRF tokens cannot be sent to the server in a cookie!
    - The token must be sent somewhere else (e.g. a header, GET parameter, or POST content)
  - CSRF tokens are usually valid for only one or two requests
CSRF Tokens: Usage

1. Login
2. Get token
3. Make this request
4. Make request

The request in step 4 will fail, because the attacker doesn’t know the token!
Defense: Referer Header

- Idea: In a CSRF attack, the victim usually makes the malicious request from a different website
- Referer header: A header in an HTTP request that indicates which webpage made the request.
Referer Header

- Checking the Referer header
  - Allow same-site requests: The Referer header matches an expected URL
    - Example: For a login request, expect it to come from https://bank.com/login
  - Disallow cross-site requests: The Referer header does not match an expected URL
- If the server sees a cross-site request, reject it

Issues?
SameSite Cookie Attribute

- Idea: Implement a flag on a cookie that makes it unexploitable by CSRF attacks
  - This flag must specify that cross-site requests will not contain the cookie

- **SameSite flag**: A flag on a cookie that specifies it should be sent only when the domain of the cookie **exactly** matches the domain of the origin
  - SameSite=None: No effect
  - SameSite=Strict: The cookie will not be sent if the cookie domain does not match the origin domain
  - Example: If `https://evil.com/` causes your browser to make a request to `https://bank.com/transfer?to=mallory`, cookies for bank.com will not be sent if SameSite=Strict, because the origin domain (evil.com) and cookie domain (bank.com) are different

- Issue: Not yet implemented on all browsers
**What is Cross-Site Scripting**

**Cross Site Scripting**: Attack occurs when application takes untrusted data and sends it to a web browser without proper validation or sanitization.

**Command/SQL Injection**

attacker's malicious code is executed on app's server

**Cross Site Scripting**

attacker's malicious code is executed on victim’s browser
Search Example

https://google.com/search?q=<search term>

```
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for <script>$_GET['q']</script></h1>
  </body>
</html>
```
Normal Request

https://google.com/search?q=apple

```html
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for apple</h1>
  </body>
</html>
```

Sent to Browser

```html
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for apple</h1>
  </body>
</html>
```
Embedded Script

https://google.com/search?q=\<script>alert("hello")\</script>

```html
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for \<?php echo $_GET["q"] ?\\</h1>
  </body>
</html>
```

Sent to Browser

```html
<html>
  <title>Search Results</title>
  <body>
    <h1>Results for \=\<script>alert("hello")\</script> \</h1>
  </body>
</html>
```
Cookie Theft!

https://google.com/search?q=\(<\text{script}>...\)</script>
Types of XSS

An XSS vulnerability is present when an attacker can inject scripting code into pages generated by a web application.

**Two Types:**

Reflected XSS. The attack script is reflected back to the user as part of a page from the victim site.

Stored XSS. The attacker stores the malicious code in a resource managed by the web application, such as a database.
Reflected Example

Attackers contacted PayPal users via email and fooled them into accessing a 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.
Stored XSS

The attacker stores the malicious code in a resource managed by the web application, such as a database.
Samy Worm

XSS-based worm that spread on MySpace. It would display the string "but most of all, samy is my hero" on a victim's MySpace profile page as well as send Samy a friend request.

In 20 hours, it spread to one million users.
MySpace Bug

MySpace allowed users to post HTML to their pages. Filtered out

   <script>, <body>, onclick, <a href=javascript://>

Missed one. You can run Javascript inside of CSS tags.

   <div style="background:url('javascript:alert(1)')">
Filtering Malicious Tags

For a long time, the only way to prevent XSS attacks was to try to filter out malicious content.

Validate all headers, cookies, query strings, form fields, and hidden fields (i.e., all parameters) against a rigorous specification of what is allowed.

‘Negative’ or attack signature based policies are difficult to maintain and are likely to be incomplete.
Filtering is Really Hard

Large number of ways to call Javascript and to escape content

URI Scheme: `<img src="javascript:alert(document.cookie);">`

On{event} Handers: onSubmit, OnError, onSyncRestored, ... (there’s ~105)

Samy Worm: CSS

Tremendous number of ways of encoding content

`<IMG SRC=##0000106##0000097##0000118##0000097##0000115##0000099##0000114##0000105##0000101##00000115##0000058##0000097##0000108##0000101##0000114##0000116##0000040##0000039##0000088##0000083##0000083##0000039##0000041>`

Google XSS Filter Evasion!
Filters that Change Content

Filter Action: filter out `<script`

Attempt 1: `<script src="...">`  
```html
src="..."
</script>`

Attempt 2: `<script src="..."></script>`