Web Security

pdf

Overview

- Client-side security :
  - risks, attacks, solutions
- Server-side security :
  - risks, attacks, solutions

Client-Side Security

- Anyone can create a web site, publish it on the Web and have it indexed by search engines :
  - including malicious persons
- Browsers execute arbitrary JavaScript code when you visit a Web site :
  - They don't know a priori if a site is malicious
- Navigating to a malicious site could have more or less dramatic consequences :
  - Browser crash
  - Data leak (passwords, credit card information, email addresses . . .)
  - Identity theft (reuse of stolen information from one site to another site to send emails, transfer money. . .)
  - Data corruption (ransomware)
  - Illegal use of computer resources as part of botnets (spam, DDoS attacks . . .)

Browser strategies

- Browsers limit client-side security issues by :
  - Supporting only JavaScript APIs that have been reviewed for security
  - Not allowing certain APIs (file browsing, network scanning, . . .)
  - Asking permissions to the user for some APIs (e.g. Camera)
  - Running the code of a web page isolated from the other pages, and from the OS :
    - the sand box
  - Fixing bugs/vulnerabilities
- But this is not perfect :
  - Web users/developpers should be careful !
  - Web users/developpers update their browsers/servers regularly !

Same Origin Security

- The architecture of the Internet is based on private sub-networks (local networks, intranets . . .)
  - It is necessary to protect a sub-network even if one computer of this sub-network is compromised
  - –> Goal of the **same origin policy**
- General principle :
  - A page requested from Origin A (e.g. http ://www.example.org) should not be able to retrieve content from Origin B (e.g. http ://mylocalserver.com)

Example of “cross-origin attack”

- Example of a cross-origin attack
  - Computer A is in a Network X (e.g. Telecom Paris)
  - User Alice on computer A receives a link to a (malicious) Web site at http ://malicious.org
  - Alice loads http ://malicious.org (1).
  - http ://malicious.org contains a script S to scan all resources in network X (2).
  - That script forwards all information to X (3)

Same Origin Policy

- Web browsers restrict resource loading based on the resource type and on the origin
- Origin is defined as :
  - protocol + domain + port
  - Examples :
    - http ://example.org is a different origin from http ://example.com
    - http ://www.example.org is different origin from http ://data.example.org
    - http ://example.org is a different origin from http ://example.org :8000
    - http ://example.org is a different origin from https ://example.org
- For historical reasons, markup resources are NOT restricted to the same origin :
  - HTML : an <iframe> can point to a different domain
  - CSS : a <link> element can point to a different domain
  - Images : a <img> element can point to a different domain
    - Basic web behavior : reuse an image from another site

../tp

../logo-IPP-s

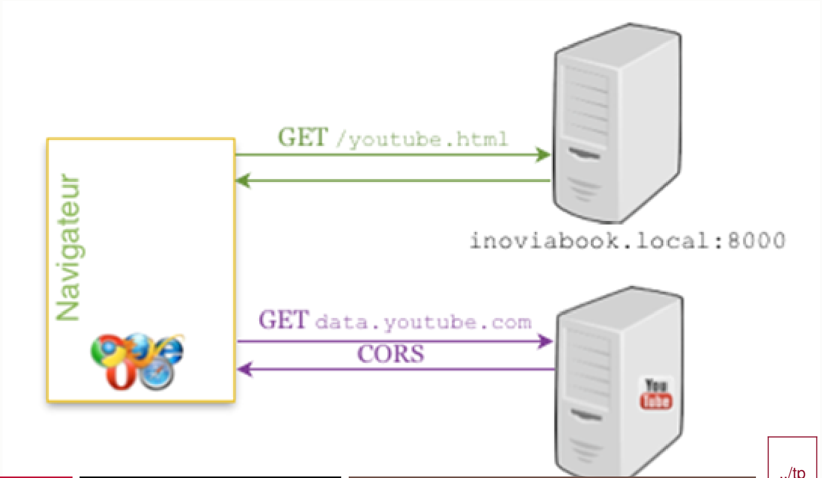
Same Origin and JavaScript

- The main restriction is on JavaScript downloads :
  - XMLHTTPRequests calls in AJAX are restricted to the Same Origin
  - Unless
    - The request is “simple”
    - CORS is used
  - Some work-around exists :
    - create a script tag to make the download (JSONP)
    - Use a proxy so that all requests come from the same origin
- JavaScript Communication between pages of different origins is possible using JavaScript postMessage API

CORS

HTTP Cross-Origin Resource Sharing

- Server A explicitly allows clients to use its data within JavaScript if downloaded from server B



Cookies and Cross-origin

- Reminder : Cookies are small text files, containing server information, stored on the client-side and exchanged in HTTP or HTTPS requests
  - Mainly used today as tracking tools for advertizers (annoying) but server-side tools can do worse. . .
  - Can be used to keep track that the user is logged (cookie is set by the server after login/password have been verified)
  - Cookies can be risky if used to store passwords or long-lived client session information. Modern web sites use short-lived, randomized session ids
  - Cookie can be session-based or permanent (“remember my identifier”)
  - Cookies follow the same origin principle but with a looser concept of Origin
  - Can be created/manipulated in JavaScript using `document.cookie`. Be careful when loading cross origin scripts, they may steal cookies (unless HttpOnly is used)

Some client-side attacks

- Clickjacking
- Phishing
- Data Sniffing

Clickjacking

- Problem :
  - The user is made to be believe that they click on something (e.g. “click here to get a free gift”), while clicking on something else (e.g. allowing camera access, . . .).
  - Usually done using transparent <iframe>
- Solution :
  - Limit the use of <iframe> with `X-Frame-Options` : disallowed or same origin
  - Make sure in JavaScript that the click happend on the most top level window

Phishing

- Problem
  - Alice receives of a link (via email, messaging, . . .) to a site that looks like another site (e-commerce, bank . . .)
  - Alice believes that she is on the official site and enters her personal information (login, password, credit card . . .)
  - Her personal information is stolen and used elsewhere.
- Solution :
  - As a user, make sure the URL of a site is the right one.
  - As a developer, require HTTPS for sensitive sites

Data Sniffing

- Problem :
  - On a wired networks (including possibly within operator's networks) or on a WiFi network not sufficiently protected (e.g. WEP), one can capture IP packets and analyze them.
  - HTTP packets may contain login, password, credit card information that anyone can copy
  - By scanning packets on a network, anyone can steal information
- Solution :
  - As a user : do not transmit sensitive information unless the site uses HTTPS
  - As a developper : do not setup sites requiring sensitive information with HTTP, require HTTPS

Server-side Security

- The Web and the Internet are **hostile** environments !
- Unless the access to the server is restricted (firewalls, NATs …), anyone from the Internet can make request to a web server, including **malicious requests**
- Web security is a **shared** responsibility : web server administrator and web master

Server-side Security Risks

Data leak (e.g. Yahoo !, Ashley Madison . . .)

Data ransom

Illegal use of server resources as part of a botnet (storage, spam or DDoS)

Server-side Security Measures : Why ?

- Some verifications of the data sent to the server can be made at the client-side :
  - Limit text field size, file upload size, . . .
  - use radio buttons or pre-determined choices to avoid arbitrary text upload
  - Use JavaScript client-side checks to verify the validity of data sent
- But a malicious person will probably not use your site to attack the server
  - Necessity to perform input checks at server-side to avoid “code injection”

XSS - Cross Site Scripting : Problem

- A Web Site contains a form to send text content to a server

```
<form action="http://myserver.com/hello">
<input type="text" name="Nom">
</form>
```

- A malicious user uses this site to send HTML content including <script> (not just text) to the server

```
World <script>alert('You have a won the lottery!')</script>
```

- The server receives this content, treats it as simple text and returns an HTML response based on this text content

```
return "<html><body><p>Hello" + Nom + "</p></body></html>"
```

- The browser receives the new HTML page and execute the malicious-user-inserted HTML :
  - A pop-up appears that was not planned by the Web Site author !

../tp ../logo-IPP-s

XSS : Solutions

- Check where the text content will be inserted
- Never insert untrusted text content except in allowed locations (e.g. NOT in scripts, NOT as attribute names, or NOT as element names, or NOT as comments, or NOT in CSS content)
- Use specific escaping mechanisms depending on where the text content is inserted
  - Escape HTML special characters when saving/returning text content in HTML text content using HTML entities (`&...;`)
    - If a malicious user enters `<script>` the server should use `&lt;script&gt;`
  - In many server-side programming languages, there are helper functions. In PHP, you can use the function `htmlspecialchars`, `htmlentities`, or `strip_tags` In NodeJS or Python, use HTML escape modules
  - Use specific escape mechanism when inserting text in attribute content (e.g. quote)
  - Use specific escape mechanism when inserting text in script

XSRF - Cross Site Request Forgery

- Problem :
  - User Alice is already logged onto a site (Forum, Blog, . . .)
  - Malicious user Bob sends a link to a malicious page
  - The malicious page contains a 'hidden' link, such as :

```
<img src="http://www.example.com/forum/delete_user.php?use
```

  - Alice erroneously clicks on the link without realizing, because Alice is still logged in :
    - User Donald is removed from the forum
- Solutions :
  - Use HTTP POST requests to make it more complex to send the parameters and to force a page refresh when the action is done (make the user aware)
    - Can be worked-around with 0-width 0-height iframes
  - Limit the use of `<iframe>` with `X-Frame-Options` : disallowed or same origin
  - Check the referer : where this action comes from
  - Best practice : Use random tokens to avoid the URL of the action to be known in advance

SQL injection

- Problem
  - If the user input text is used to be build SQL requests, well-crafted text can leak database information
  - Example :
    - The HTML form includes a text field called “passwd”
    - The server uses this text to make a request :

  ```
  mysql_query("SELECT * FROM T WHERE passwd='$passwd'")
  ```

    - If the input passwd text is : `' OR 1=1 --` the request becomes :

  ```
  mysql_query("SELECT * FROM T WHERE passwd='' OR 1=1 --'")
  ```

  - Which returns all values !!
- Solution
  - Do not construct SQL requests from user text inputs
  - If not possible, escape text content (e.g. PHP `mysql_real_escape_string`)

Command-line injection

- Problem
  - If the user input text is used to be call executables (e.g. using `exec` in PHP or NodeJS), well-crafted text can execute arbitrary code
  - Example :
    - The HTML form includes a text field called “a”
    - The server uses this text to list the content of a directory named “a” :

    ```
    exec(”ls $a”)
    ```

    - If the input text is : `&& cat /etc/passwd` the system call becomes :

    ```
    exec(”ls && cat /etc/passwd”)
    ```

  - Passwords are leaked !!
- Solution
  - Avoid calling executables using parameters input from a web page
  - If not possible, inspect final command line and escape text content (e.g. PHP `escapeshellcmd` or `escapeshellarg`)

../tp ../logo-IPP-s

Directory Traversal

- Problem
  - If the user input text is used to open files, well-crafted text (i.e. using '..' and '/') can allow reading any file
  - Example :
    - The HTML form includes a text field called "file"
    - The server uses this text to list open a file and return its content :

```
read($file)
```

    - If the input text contains : `../../../../../etc/passwd`, passwords are leaked!!
- Solution
  - Avoid allowing reading of a file whose name is based on an input from a web page
  - If not possible, inspect the final path and disallow specific paths

Summary of the lesson (web data and web security)

- Web data, text, internationalization, text rendering
- Character set, Unicode, encoding, UTF-8
- Structured text, CSV, XML, JSON, JSONP
- REST and web APIs
- Security, browser protection
- Same origin, cross origin, CORS
- Client-side attacks : clickjacking, phishing, data sniffing
- Server attacks : injections