Information Retrieval
WS 2015 / 2016

Lecture 7, Tuesday December 1st, 2015
(Web App Vulnerabilities, Cookies, Unicode)

Prof. Dr. Hannah Bast
Chair of Algorithms and Data Structures
Department of Computer Science
University of Freiburg
Overview of this lecture

- Organizational
  - Your experiences with ES6 web application
  - Questions on the forum guidelines again

- Contents
  - More practically relevant web app stuff:
    - Vulnerabilities injection flaws, cross-site scripting
    - Cookies store information across web sessions
    - Unicode ISO-8859-1, UTF-8, URL encoding

**Exercise Sheet 7:** add a "UTF-8 repair" feature to your web app from ES6 + cookies for a better user experience
Experiences with ES6

Experiences + Results

- Very interesting / cool / fun / nice / fascinating exercise
- Time-consuming for some ... especially for those new to the whole web stuff, in particular JavaScript
- Lots of Googling and Stackoverflow
- URL encoding not discussed in lecture ... will be today!
- Pictures from Freebase: "User Rate Limit Exceeded"
- Many of you made some really nice web apps
  
  Let's have a look at some of them

  Sorry to those with awesome web apps not shown today
Questions on the Forum

Guidelines (again)

– Please feel absolutely free to ask questions

    Especially if you spend hours on minor problems otherwise

– Before that consider the Zen of Self-Help for this course:

    Look at the exercise sheet it is there for a reason
    Look at the slides they are there for a reason
    Look at the lecture code it is there for a reason

– If your code produces a surprising error message, try
  pasting it into Google (the error message not your code)

    This leads to a page with the solution surprisingly often
Motivation

- Web Apps are particularly vulnerable to privacy breaches
  
  Because stuff is constantly sent back forth between your computer and a foreign computer, with so many different layers of software and hardware in-between

- We will briefly look at three kinds of vulnerabilities today:
  
  Access to private data
  
  Execution of code injected by an attacker
  
  Communication of trusted information to an untrusted site

- For a list of the top-10 web app vulnerabilities:
  
  google: OWASP Top Ten Project
Access to private data

- When writing or configuring a web server, take care to serve only those files/data you want to serve.
- We saw a simple problem + exploit in the last lecture:
  http://stromboli.cs.uni-freiburg.de:8081//proc/cpuinfo
- This is easily fixed by carefully restricting access.
  For example, only serve files in a certain directory subtree.
  Even safer: a "whitelist" of files are served ... for all other files, return a 404 (Not Found) or a 403 (Forbidden).
Vulnerabilities 3/7

- Code Injection
  - Exploit: make a web site execute malicious code

  **Example 1:** enter JavaScript into search box
  ```html
  <a href="javascript:alert(document.cookie)">Click me!</a>
  ```

  **Example 2:** send someone a mail with a link
  ```html
  ...index.php?user=guest<script>alert("Ha!")</script>
  ```

  Note: the `<script>...</script>` part can be made more unsuspicious by URL-decoding (see slide 24) it:
  ```html
  ...index.php?user=guest%3C%73%63%72%69%70...
  ```
Vulnerabilities 4/7

- Code Injection
  - Exploit: make a web site execute malicious code

  **Example 3:** post to forum with some script in it

  I have a question<script>... JavaScript code that sends user info by mail to evil person ...
</script>

  Note: The <script>...</script> will not show on the website, but code will be executed by any client viewing the post

  JS code could also open Gmail Tab and inspect private mail

  - This can be fixed by carefully checking the content that is dynamically added to a webpage
The Same-Origin-Policy (SOP)

- Domain + port of client and server URL must be **identical**
  
  http://etna.cs.uni-freiburg.de:8888/search.html
  http://etna.cs.uni-freiburg.de:8888/?q=zurich

- To understand why, consider the following scenario:

  You somehow get redirected to an evil site that looks just like your banking website, e.g. http://www.postbamk.de

  Without the same-origin-policy, the evil site could now communicate with the bank server like the real site

  Worse: with stolen session cookie, evil person could do anything in your name without you even participating
Exceptions to the Same-Origin-Policy (SOP)

- JavaScript can be loaded from anywhere

  That way we could use jQuery without downloading it

  `<script src="http://code.jquery.com/jquery1.10.2.js">`

- There are applications where it is actually desirable that everybody (or many people) can access them

  For example, our backend for query suggestions

  Or an API to a public database

  Historical note: JSON uses `<script>...</script>` to circumvent SOP and became a standard → weird!
CORS = Cross-Origin Resource Sharing

- Principle: the server explicitly specifies which web sites may use the results it returns as follows:

  If the JavaScript wants to communicate with a machine (or port) other than the one it was loaded from, then the following additional request header is sent:

  **Origin**: http://<host name>:<port>

  Depending on that header, or independent of it, the server can then send a response header like this:

  **Access-Control-Allow-Origin**: http://<host name>:<port>

  Browser then uses the result **only when the two agree**.
Cookies  1/5

- Basic mechanism
  - A cookie is simply a string associated with a web page that is stored on the client's computer
    
    Each client has its own cookie
    
    Typically used for user data and preferences
  
  - A cookie can contain any contents, but the convention is that it contains a sequence of key-value pairs, separated by semicolons, for example:
    
    user=cookie-monster; prefers=kekse
  
  - Implementation in JavaScript is very simple, just read and write this string via the variable `document.cookie`
Adding key-value pairs to a Cookie

- To add a key-value pair, just write
  ```javascript
document.cookie = "user=cookie-monster";
  ```
- Multiple assignments **add** to the string ... *weird but true*
  ```javascript
document.cookie = "user=cookie-monster";
document.cookie = "prefers=kekse";
  ```
- To overwrite the value for a key, just write again
  ```javascript
document.cookie = "prefers=kekse";
document.cookie = "prefers=kruemel";
  ```

Inspect in browser with F12 → Resources → Cookies
Cookies  3/5

- Getting the value for a particular key
  - In raw JavaScript, need some string processing:

```javascript
var cookies = document.cookie.split(";");
for (var i = 0; i < cookies.length; i++) {
    var args = cookies.replace(/\s/g, "").split("=");
    if (args[0] == "user")
        alert("Hi " + args[1] + " !!!");
}
```
Different kinds of cookies

- **Chocolate chip cookie**
  Accidentally developed by Ruth Wakefield in 1930

- **Session cookie** ... lasts as long as your browser is open
  
  user=cookie-monster

- **Persistent cookie** ... lasts until the specified date
  
  user=cookie-monster; expires=Wed 04 Dec 2013 17:45

- **Third-party cookies** ... from JavaScript from other domains
  Beware: these often give access to sensitive information
In jQuery … using https://plugins.jquery.com/cookie/

- Setting a cookie
  
  \$.cookie("user", "cookie-monster");

- Value of a cookie
  
  var user = \$.cookie("user");

- Removing a cookie
  
  \$.removeCookie("user");

- Cookie with expiry date (10 days from now)
  
  \$.cookie("user", "cookie-monster", { expires: 10});
Motivation

- To represent text in binary, we need a standard for how to represent the characters of the alphabet, numbers, etc.
- For a very long time, this standard was ASCII:
  1 Byte per symbol = can represent 256 different symbols
- Obviously there are more than 256 symbols in the world
  Chinese alone has (tens of) thousands of different symbols
Solution before Unicode

- Use the ASCII codes 0 – 127 for common symbols, which (almost) everybody needs

  a-z  A-Z 0-9 ( ) [ ] { } , . : ; " ' ...

  ASCII codes 0 – 31 used for control characters

- For the ASCII codes 128 – 255, have (many) different variants, depending on the context

  For example, ISO-8859-1: use the codes to encode all the funny characters from most European languages

    à á â ã ä å ç è é ê ë ì í î ï ð ñ ò ó ô õ ö ø …

- Problem: if you need more than one variant, you need to switch the encoding in the middle of the document
The Unicode solution

- Simply assign a unique number, called code point, to (almost) every character / symbol in the world, e.g.

  a : 97 (hex = 61)
  A : 65 (hex = 41)
  ä : 228 (hex = E4)
  ä : 228 (hex = E4)
  α : 945 (hex = 03B1)
  € : 8364 (hex = 20AC)
  ☛ : 128584 (hex = 1F648)

- Unicode knows 1,114,112 code points (hex: 0 .. 10FFFF)

Note: 1 Byte not enough, and 2 Bytes also not enough
UTF = Unicode Transformation Standard

- There are different schemes for how to actually represent these code points in binary

**UTF-32**: always use 4 bytes per code point obviously enough for all $1,114,112$ known code points

**UTF-16**: use 2 bytes for the common code points, and 4 bytes for the others ... used for **String** in Java

**UTF-8**: use 1 byte for the very common code points, and 2 or 3 or 4 bytes for the others ... see next 2 slides

UTF-16 and UTF-8 are **variable-byte** encodings
Details of UTF-8

- **1 Byte:** Code point in $[0, 127] = \text{xxxxxxx}$
  
  UTF-8 code: $0\text{xxxxxxx}$ 7 Bits

- **2 Bytes:** Code point in $[128, 2047] = \text{yyyyyyyyy}$
  
  UTF-8 code: $110\text{yyyyx} 10\text{xxxxx}$ 11 Bits

- **3 Bytes:** Unicode in $[2048, 65535] = \text{yyyyyyyyyyyyyyyy}$
  
  UTF-8 code: $1110\text{yyyyy} 10\text{yyyyy} 10\text{xxxxx}$ 16 Bits

- **4 Bytes:** Unicode in $[65536, 2^{21} - 1] = \text{zzzzzzzzzzzzzzzzzzzzzzzzzz}$
  
  UTF-8 code: $11110\text{zzz} 10\text{zzyyyy} 10\text{yyyyy} 10\text{xxxxx}$ 21 Bits

In principle, could continue with 5-byte and 6-byte sequences, but UTF-8 stops here, since $2^{21} \approx 2M$ is enough  

[RFC 3629]
UTF-8 has the following nice properties

- **ASCII compatible** = a string of characters with ASCII codes < 128 is the same in ASCII as in UTF-8
  
  So old C / C++ code only fails on the special characters

- **ISO-8859-1 compatible** = characters with code 1xyyyyyy have the 2-byte UTF-8 encoding 1100001x 10yyyyyy

- Only rarely used characters need more than 2 bytes

- Easy to decode: codes start and end at byte boundaries

- Can decode starting from anywhere within a string
  
  Just move left to the next byte not starting with 10
Some more properties of UTF-8

- In a multi-byte UTF-8 character all bytes are $\geq 128$, and vice versa such bytes occur only for multi-byte characters.
- The number of leading 1s in the first byte of a multi-byte character is equal to the number of bytes of its code.
- For every Unicode in $[0, 2^{21} - 1]$ there is **exactly one** valid UTF-8 multi-byte sequence.
- But vice versa not all multi-byte sequences are valid UTF-8.

For example $1100000x$ $10xxxxxx$ is **not** valid.

Should be encoded with 1 byte: $0xxxxxx$
URL decoding and encoding, motivation

- In a URL, only a restricted character set is allowed:
  
a-z A-Z 0-9 $ % / - _ . + ! * ... and a few more

  In particular, not allowed: space, ä, ã, â, ...

- Arguments of GET request are part of the URL

  In particular, the ?q=... part of your web app for ES6

  For ES7 (city search), this part can contain arbitrary characters, in particular umlauts as in München
URL decoding and encoding, realization

- Special characters are encoded by a % followed by the code in hex-decimal ... for example:

  If encoding of web page is UTF-8
  
  ä : UTF-8 code C3A4 → URL-encoded as %C3%A4

  If encoding of web page is ISO-8859-1:
  
  ä : ISO-8859-1 code E4 → URL-encoded as %E4
Implementation Advice for ES7

- To view the **byte-wise** contents of a file, independent of its encoding use the Linux tool `xxd` or `xxd -b`

  Inside an IDE, Text Editor, or Console what you see is already an interpretation of the contents of the file, assuming a certain encoding, e.g. UTF-8 or ISO-8859-1

- Beware when reading the file into a string

  **Java:** read into byte[] to avoid implicit conversion
  
  **Python:** use decode
  
  **C++:** convert to std::wstring (= std::string<wchar_t>)
References

- CORS

- Cookies
  - http://www.w3schools.com/js/js_cookies.asp

- UTF-8, URL-encoding and -decoding
  - http://www.utf8-chartable.de
  - http://www.w3schools.com/tags/ref_urlencode.asp