Information Retrieval
WS 2016 / 2017

Lecture 7, Tuesday December 6th, 2016
(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

- Contents
  - Web applications, second part
    - JavaScript: Continuation from last lecture
    - Vulnerabilities: privacy, code injection, cross origin
    - Cookies: store information across web sessions
    - Unicode: ISO-8859-1, UTF-8, URL encoding

**Exercise Sheet 7:** complete your web app + make it nice & secure + use cookies + deal with Unicode properly
Experiences with ES6  1/4

Experiences + Results

- Many of you liked this exercise sheet **a lot**
- Time consumption ok, because split over two sheets
  
  Some of you did the JavaScript part already now
- No errors in TIP file this time ... but one in the lecture!
  
  Anyway, we brought some cookies for all of you
- Some of you have a nice and working web app already
  
  We will show a selection next week !
Which objective do life forms optimize?

- From the perspective of the individual consciousness:
  
  Maximize happiness, satisfaction, etc.

- From the perspective of the genome:
  
  Spread DNA as much as possible (the whole body and being is just a tool for that)

  Richard Dawkins "The Selfish Gene"

- From the perspective of the universe / physics:
  
  Why and how did life emerge in the first place?

  See next two slides for some interesting thoughts ...
Experiences with ES6  3/4

Why did life emerge in the first place?

- Abiogenesis: how life arises from non-living matter

  Early theories: life must be something "spiritual", with "spontaneous generation" (maggots arise from dead flesh)

- Miller-Urey experiment: basic elements → 23 amino acids

  Earth's early atmosphere simulated: \( \text{H}_2\text{O}, \text{CH}_4, \text{NH}_3, \text{H}_2 \) and heat and sparks and evaporation/condensation

- Next steps from there:

  From monomers (amino acids) to polymers (proteins)

  From polymers to cells

  From single cells to multicellular organisms
Which objective function is optimized?

- By the second law of thermodynamics, closed systems tend to dissipate energy until the entropy is maximized

  Intuitively: a state of least structure / highest disorder

- However, with an external energy source far away (think of earth and sun), something else happens

- Hypothesis: life forms are the best configuration to dissipate energy from the external source... that's it

  "You start with a random clump of atoms, and if you shine light on it for long enough, it should not be so surprising that you get a plant"

  This is far from being proven... but certainly interesting
Motivation

- Web Apps are particularly vulnerable to privacy breaches

  Because data + code is sent back forth between multiple computers (foreign to each other), 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

- Top-10 web app vulnerabilities ... google: OWASP Top Ten

  OWASP = Open Web Application Security 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://etna.informatik.privat:8888/etc/passwd](http://etna.informatik.privat:8888/etc/passwd)
- 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

`<a href="javascript:alert(document.cookie)">Click me!</a>`

**Example 2:** send someone a mail with a link

`...index.html?user=guest<script>alert("Ha!")</script>`

Note: the `<script>...</script>` part can be made more unsuspicious by URL-decoding (see slide 27):

`...index.html?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

  **ES7:** if you don't pay attention, strange things might happen
The Same-Origin-Policy (SOP)

- Domain + port of client and server URL must be identical
  
  http://etna.informatik.privat:8888/search.html
  
  http://etna.informatik.privat: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
Vulnerabilities  6/7

- CORS = Cross-Origin Resource Sharing
  - When JavaScript requests a resource from a different host+port (than the website on which the script is executed), the following header is added to the request:
    \[
    \text{Origin: http://<host name>:<port>}
    \]
  - The result (think: JSON) then must be augmented by the following header:
    \[
    \text{Access-Control-Allow-Origin: http://<host name>:<port>}
    \]
  - The website can access the result if and only if both host name and port match exactly
  - For a public service, the result can also be returned with
    \[
    \text{Access-Control-Allow-Origin: *}
    \]
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">`

- Seemed reasonable at the time, because in HTML, objects like images could also be loaded from anywhere

- However, this allows security hacks like **JSONP**, which dynamically adds `<script>myFct("...")</script>` to the HTML tree, which lets `myFct` do arbitrary things with "..."

  A hack to circumvent SOP ... which became a standard
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:

    \texttt{user=cookie-monster; prefers=kekse}

  - Implementation in JavaScript is very simple, just read and write this string via the variable \texttt{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";
  ```

View in browser: F12 → Application → Storage → Cookies
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[i].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
Using **js-cookie** … [https://github.com/js-cookie/js-cookie](https://github.com/js-cookie/js-cookie)

- Setting a cookie
  ```javascript
  Cookies.set("user", "cookie-monster");
  ```

- Value of a cookie
  ```javascript
  var user = Cookies.get("user");
  ```

- Removing a cookie
  ```javascript
  Cookies.remove("user");
  ```

- Cookie with expiry date (10 days from now)
  ```javascript
  Cookies.set("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 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)
  - α : 945 (hex = 03B1)
  - € : 8364 (hex = 20AC)
  - 😊 : 128512 (hex = 1F600)

- 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
    
    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 \([0, 127]\) = xxxxxxx
  
  UTF-8 code: 0xxxxxxx 7 Bits

- **2 Bytes**: Code point in \([128, 2047]\) = yyyyyyyyyy
  
  UTF-8 code: 110yyyyx 10xxxxxx 11 Bits

- **3 Bytes**: Unicode in \([2048, 65535]\) = yyyyyyyyyyyyyy
  
  UTF-8 code: 1110yyyy 10yyyyxx 10xxxxxx 16 Bits

- **4 Bytes**: Unicode in \([65536, 2^{21} - 1]\) = zzzzzyyyyyyyyyyyyyyyy
  
  UTF-8 code: 11110zzz 10zyyyyy 10yyyyxx 10xxxxxx 21 Bits

In principle, this could continue with 5 bytes and 6 bytes, but \(2^{21} \approx 2M\) is enough for the 1.1M Unicode code points.
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: $0xxxxxxx$.
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, á, 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
Encoding in files

- All modern editors let you choose the encoding.
- 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 you type or print something on the terminal, the encoding used by the terminal is relevant.

  This can usually be changed easily in the menu.
Encoding in C++

- In C++, there is `std::string` and `std::wstring`

```cpp
// std::string = array of char (char = 1 byte)
std::string s = "\xc3\xa4"; std::cout << s; // prints ä

// std::wstring = array of wchar_t (Unicodes)
std::wcout.sync_with_stdio(false);
std::imbue(std::locale(""));
std::wstring w = L"ä"; std::wcout << w; // prints ä

// Convert between std::string and std::wstring
std::wstring_convert<std::codecvt_utf8<wchar_t>> conv;
std::string utf8string = conv.to_bytes(L"ä"); \{ c3, a4 \}
std::wstring wstring = conv.from_bytes("ä"); \{ U+00e4 \}
```
Encoding in Java

- In Java, there is String and byte[]

```java
// String = array of char (char = 2 bytes)
"ä".length(); 1   (U+00E4)

// Unicodes ≥ 2^{16} are UTF-16 encoded
"😊".length(); 2   (U+1F600)
"😊".charAt(0); U+0001
"😊".charAt(1); U+F600

// Convert between String and byte array
byte[] b = "ä".getBytes("UTF-8");  { 0xc3, 0xa4 }
new String(b, "UTF-8").charAt(0);   ä   (U+00E4)
```
Encoding in Python3

- Python has both "byte array" strings and Unicode strings

  // Byte array strings = b"...
  print(b"\xc3\xa4") ä on UTF-8 terminal
  print(b"\xc3\xa4") Å on ISO terminal
  print(b"ä") not allowed in Python3

  // Unicode strings = u"...
  print(len(u"ä")) prints 1

  // Convert between the two
  b"\xc3\xa4".decode("UTF-8") u"ä"
  b"\xc3\xa4".decode("LATIN1") u"Ã"
  u"ä".encode("UTF-8") b"\xc3\xa4"
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