```
                    ,--.                ,----..                                   ,--.
     ,---,.       ,--.'|  ,----..      /   /   \     ,---,       ,---,       ,--.'|  ,----..
   ,'  .' |   ,--,:  : | /   /   \   /   .     :  .'  .' `\   ,`--.' |   ,--,:  : | /   /   \
 ,---.'   |,`--.'`|  ' :|   :     : .   /   ;.  \,---.'     \ |   :  :,`--.'`|  ' :|   :     :
 |   |   .'|   :  :  | |.   |  ;. /.   ;   /  ` ;|   |  .`\  |:   |  '|   :  :  | |.   |  ;. /
 :   :  |-,:   |   \ | :.   ; /--` ;   |  ; \ ; |:   : |  '  ||   :  |:   |   \ | :.   ; /--`
 :   |  ;/||   : '  '; |;   | ;    |   :  | ; | '|   ' '  ;  :'   '  ;|   : '  '; |;   | ;  __
 |   :   .''   ' ;.    ;|   : |    .   |   ' ' ' :'   | ;  .  ||   |  |'   ' ;.    ;|   : |.' .'
 |   |  |-,|   | | \   |.   | '___ '   ;  \; /  ||   | :  |  ''   :  ;|   | | \   |.   | '_.' :
 '   :  ;/|'   : |  ; .''   ; : .'| \   \  ',  / '   : | /  ; |   |  ''   : |  ; .''   ; : \  |
 |   |    \|   | '`--'  '   | '/  :  ;   :    /  |   | '` ,/  '   :  ||   | '`--'  '   | '/  .'
  |   :   .''   : |      |   :    /    \   \ .'   ;   :  .'    ;   |.' '   : |      |   :    /
   |   | ,'  ;   |.'       \   \ .'      `---`     |   ,.'      '---'   ;   |.'       \   \ .'
    `----'    '---'         `---`                  '---'               '---'          `---`
```

First off

- Computers can’t store letters, numbers, pictures, etc.
- Computers store bits: 1 or 0
- Given that, then how do computers interpret letter and number characters?

Encoding

- An encoding scheme is a set of rules that translates bits to letter and numbers characters
- Encode: convert into a coded form
- In ASCII each character is one byte (8 bits)
- `01100010 01101001 01110100 01110011`
- `b        i        t        s`

ASCII

- All English letters a-zA-Z0-9, punctuation, some things like space, line feed, tab, backspace, etc.
- 128 characters in total = 7 bits
- Read left to right converting bits to characters
- `01001000 01100101 01101100 01101100 01101111 00100000 01010111 01101111 01110010 01101100 01100100`
- `Hello World`

What about other languages?

- 128 characters covers English, but what about Korean, Chinese, Hindi, Arabic, Russian, etc.
- To encompass all of these characters and languages more than a single byte (8 bits) is needed
- BIG-5 is a double-byte encoding that covers Traditional Chinese characters
- GB18030 does the same thing but covers simplified and traditional Chinese

GB18030

bits	character
10000001 01000000	丂
10000001 01000001	丄
10000001 01000010	丅
10000001 01000011	丆
10000001 01000100	丏

Unicode Overview

- An attempt to unify all encoding standards
- A code table of 1,114,112 code points
- Enough to encode all European, Middle-Eastern, Far-Eastern, Southern, Northern, Western, pre-historic and future characters
- Big enough for unofficial private-use sections
- There is an unofficial section for Klingon
- How many bits does Unicode use to encode all of this?

none

Unicode

- Unicode is not an encoding
- Unicode is a table of code points for characters
- Does not concern itself with how to represent those code points as bits
- "65 stands for A, 66 stands for B and 9,731 stands for ☃”
- There are several ways to encode Unicode code points into bits

UTF-X

- UTF-32 uses 32bits to encode all Unicode code points. Simple, but wastes space.
- UTF-16 and UTF-8 are variable-length encodings
  - (UTF-8) If a character can be encoded using a single byte it will
  - If it requires two bytes then it will use that instead
  - UTF-16 uses two bytes by default, up to 4 bytes
  - The encoding uses the highest bits to signal how may bytes the character uses

character	encoding	bits
A	UTF-8	01000001
A	UTF-16	00000000 01000001
A	UTF-32	00000000 00000000 00000000 01000001
あ	UTF-8	11100011 10000001 10000010
あ	UTF-16	00110000 01000010
あ	UTF-32	00000000 00000000 00110000 01000010

Code Points

- Characters are often referred to by their Unicode code point
- Written in hex to keep numbers short
- Starts with a U+
- Ḁ = U+1E00 = 7680th character

Encoding Issues

```
bits          encoding      characters
11000100      01000010      Windows Latin 1  ÄB
11000100      01000010      Mac Roman    ƒB
11000100      01000010      GB18030 腂
```

```
characters    encoding              bits
Føö           Windows Latin 1       01000110 11111000 11110110
Føö           Mac Roman             01000110 10111111 10011010
Føö           UTF-8                 01000110 11000011 10111000 11000011 10110110
```

What encoding is this?

```
10000011 01000111 10000011 10010011 10000011 01010010 10000001 01011011
10000011 01100110 10000011 01000010 10000011 10010011 10000011 01001111
10000010 11001101 10010011 11101111 10000010 10110101 10000010 10101101
10000010 11001000 10000010 10100010
```

Well…

- Most of the bytes start with 1 so not ASCII
- Most is not valid UTF-8
- Mac Roman works but you get
  - ÉGÉìÉRÅ[ÉfÉBÉìÉOÇÕìÔÇµÇ≠Ç»Ç¢
- Its Japanese Shift-JIS
  - エンコーディングは難しくない
- Some document viewers (and browsers) will start to read the bits and guess what the encoding is

����

- There's also the "Unicode replacement character" � (U+FFFD)
- A program may decide to insert for any character it couldn't decode correctly when trying to handle Unicode
- If a document is saved with some characters gone or replaced, then those characters are really gone for good with no way to reverse-engineer them

Security Considerations

- Imagine a scenario where:
  - An input validation filter rejects characters such as <, >, ', and " in a Web-application accepting UTF-8 encoded text.
  - An attacker sends in a U+FF1C FULLWIDTH LESS-THAN SIGN ＜ in place of the ASCII <.
  - The attacker’s input looks like: ＜script>
  - After passing through the XSS filter unchanged, the input moves deeper into the application.
  - Another API, perhaps at the data access layer, is configured to use a different character set such as windows-1252.
  - On receiving the input, a data access layer converts the multi-byte UTF-8 text to the single-byte windows-1252 code page, forcing a best-fit conversion to the dangerous characters the original XSS filter was trying to block. 7.The attacker’s input successfully persists to the database.

Best-fit Mappings

Target char	Target code point	Test code point	Name
o	\u006F	\u2134	SCRIPT SMALL O
o	\u006F	\u014D	LATIN SMALL LETTER O WITH MACRON
s	\u0073	\u017F	LATIN SMALL LETTER LONG S
I	\u0049	\u0131	LATIN SMALL LETTER DOTLESS I
i	\u0069	\u0129	LATIN SMALL LETTER I WITH TILDE
K	\u004B	\u212A	KELVIN SIGN
k	\u006B	\u0137	LATIN SMALL LETTER K WITH CEDILLA
A	\u0041	\uFF21	FULLWIDTH LATIN CAPITAL LETTER A
a	\u0061	\u03B1	GREEK SMALL LETTER ALPHA
"	\u0022	\u02BA	MODIFIER LETTER DOUBLE PRIME
"	\u0022	\u030E	COMBINING DOUBLE VERTICAL LINE ABOVE
"	\u0027	\uFF02	FULLWIDTH QUOTATION MARK
'	\u0027	\u02B9	MODIFIER LETTER PRIME
'	\u0027	\u030D	COMBINING VERTICAL LINE ABOVE
'	\u0027	\uFF07	FULLWIDTH APOSTROPHE
<	\u003C	\uFF1C	FULLWIDTH LESS-THAN SIGN
<	\u003C	\uFE64	SMALL LESS-THAN SIGN
<	\u003C	\u2329	LEFT-POINTING ANGLE BRACKET
<	\u003C	\u3008	LEFT ANGLE BRACKET
<	\u003C	\u00AB	LEFT-POINTING DOUBLE ANGLE QUOTATION MARK
>	\u003E	\u00BB	RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK
>	\u003E	\u3009	RIGHT ANGLE BRACKET
>	\u003E	\u232A	RIGHT-POINTING ANGLE BRACKET
>	\u003E	\uFE65	SMALL GREATER-THAN SIGN
>	\u003E	\uFF1E	FULLWIDTH GREATER-THAN SIGN
:	\u003A	\u2236	RATIO
:	\u003A	\u0589	ARMENIAN FULL STOP
:	\u003A	\uFE13	PRESENTATION FORM FOR VERTICAL COLON
:	\u003A	\uFE55	SMALL COLON
:	\u003A	\uFF1A	FULLWIDTH COLON

Filter Bypass

- Overconsumption bug
  - `<img src="#[0xC2]"> " onerror="alert(1)"</ br>`
  - `<img src="#> " onerror="alert(1)"</ br>`
- Character deletion
  - `Unicode BOM (Byte Order Mark) U+FEFF`
  - `Word Joiner (in Unicode 3.2 and up)  U+2060`
  - `<scr[U+FEFF]ipt>`
- String transformation
  - `toLower("&#x0130") == "i”`
  - `toLower("scr&#x0130pt") == "script”`
  - `Never assume: len(x) != len(toLower(x))`
- Whitespace (assigned the whitespace category and whitespace binary property)
  - Ogham space mark U+1680
  - Mongolian vowel separator U+180E
  - `<a href=#[U+180E]onclick=alert()>`

Other Attacks

- Buffer Overflow
- Subtle Crypto Bugs
- Phishing
  - Who had the | domain?

Resources

- http://kunststube.net/encoding/
- http://www.joelonsoftware.com/articles/Unicode.html
- https://websec.github.io/unicode-security-guide/character-transformations/
- http://www.unicode.org/reports/tr36/
- http://www.unicode.org/reports/tr39/
- https://www.blackhat.com/presentations/bh-usa-09/WEBER/BHUSA09-Weber-UnicodeSecurityPreview-PAPER.pdf