

1. Module - Encoding and Filtering - part 1

You need to understand what kind of type encoding is used.

4 main types of data encoding

- URL encoding
- HTML encoding
- Base(32/64) encoding
- Unicode encoding

1. URL encoding

-> called URL-encoding or percent-encoding

-> URL over internet must be in range of US-ASCII, if unsafe character are presented, we use url encoding

-> unsafe character is replaced with a % followed by 2 hexadecimal digits

characters

CLASSIFICATION	INCLUDED CHARACTERS	ENCODING REQ
Safe characters	Alphanumeric [0-9a-zA-Z], special characters \$-_.+!*'(), and reserved characters used for their reserved purposes (e.g., question mark used to denote a query string)	NO
ASCII Control characters	Includes the ISO-8859-1 (ISO-Latin) character ranges 00-1F hex (0-31 decimal) and 7F (127 decimal.)	YES
Non-ASCII characters	Includes the entire "top half" of the ISO-Latin set 80-FF hex (128-255 decimal.)	YES
Reserved characters	\$ & + , / : ; = ? @ (not including blank space)	YES*
Unsafe characters	Includes the blank/empty space and " < > # % { } \ ^ ~ [] `	YES

commonly encoded characters

CHARACTER	PURPOSE IN URI	ENCODING
#	Separate anchors	%23
?	Separate query string	%3F
&	Separate query elements	%26
%	Indicates an encoded character	%25
/	Separate domain and directories	%2F
+	Indicates a space	%2B
<space>	Not recommended	%20 or +

2. HTML encoding

-> Content-Type header specify the character encoding of the sent document.

like this : `Content-Type: text/html; charset=utf-8`

The header define character encoding using HTTP

If the header is not set, RFC define `ISO/IEC 8859-1 / Latin-1` as a standard

Programming to set the rep. header

- PHP -> uses **header()** function to send a raw HTTP header

```
header('Content-type: text/html; charset=utf-8');
```

- ASP.Net -> uses the **response object**

```
<%Response.charset="utf-8"%>
```

- JSP (JavaServer Pages) -> Uses the page directive

```
<%@ page contentType="text/html; charset=UTF-8" %>
```

-> 7 < 9 to represent it as a plain text, many ways to do that:

We want to encode the character **<** (*less-than sign*):

Character Reference	Rule	Encoded character
Named entity	& + <u>named character references</u> + ;	<
Numeric Decimal	& + # + D + ; D = a decimal number	<
Numeric Hexadecimal	& + #x + H + ; H = an hexadecimal number (case-insensitive)	< <

Here we can see some interesting variations:

Character Reference	Variation	Encoded character
Numeric Decimal	No terminator (;)	&#60
	One or more zeroes before code	&#060 &#00000060
Numeric Hexadecimal	No terminator (;)	&#x3c
	One or more zeroes before code	&#0x3c &#00000x3c

3. Base(32/64) encoding

-> binary = Base 2 [0-1]

-> octal = Base 8 [0-7]

-> decimal = Base 10 [0-9]

-> hexadecimal = Base 16 [0-9A-F]

Base 36 Encoding schema [0-9A-Z]

Firstly why 36? -> because it is used in many real-world scenarios. like `Reddit`, `TinyURL`

- most compact ???
- case-insensitive -> XSS = xss = XsS
- alphanumeric numeral system using ASCII characters -> [0-9A-Z]

Table to clarify

binary	dec	hex	36	binary	dec	hex	36	binary	dec	hex	36
0000	0	0	0	1101	13	d	d	11010	26	1a	q
0001	1	1	1	1110	14	e	e	11011	27	1b	r
0010	2	2	2	1111	15	f	f	11100	28	1c	s
0011	3	3	3	10000	16	10	g	11101	29	1d	t
0100	4	4	4	10001	17	11	h	11110	30	1e	u
0101	5	5	5	10010	18	12	i	11111	31	1f	v
0110	6	6	6	10011	19	13	j	100000	32	20	w
0111	7	7	7	10100	20	14	k	100001	33	21	x
1000	8	8	8	10101	21	15	l	100010	34	22	y
1001	9	9	9	10110	22	16	m	100011	35	23	z
1010	10	a	a	10111	23	17	n	100100	36	24	10
1011	11	b	b	11000	24	18	o				
1100	12	c	c	11001	25	19	p				

Programming to Convert from x to y in base36

- PHP ->

```
base_convert(number,frombase,tobase);

<?=base_convert("Text",36,10);?>
```

- JS ->

```
(1142690).toString(36)

1142690..toString(36) // encode

parseInt("ohpe",36) // decode
```

Base 64 Encoding Scheme

why ? -> most widespread

- The alphabet of the Base64 -> [0-1a-zA-Z] / +

How it works ?

to encode, the algorithm divide the msg into groups of 6 bits (That's why the allowed characters are 64 (2 squar 6 = 64))

Binary (dec)	Base 64	Binary (dec)	Base 64	Binary (dec)	Base 64	Binary (dec)	Base 64
000000 (0)	A	010000 (16)	Q	100000 (32)	g	110000 (48)	w
000001 (1)	B	010001 (17)	R	100001 (33)	h	110001 (49)	x
000010 (2)	C	010010 (18)	S	100010 (34)	i	110010 (50)	y
000011 (3)	D	010011 (19)	T	100011 (35)	j	110011 (51)	z
000100 (4)	E	010100 (20)	U	100100 (36)	k	110100 (52)	0
000101 (5)	F	010101 (21)	V	100101 (37)	l	110101 (53)	1
000110 (6)	G	010110 (22)	W	100110 (38)	m	110110 (54)	2
000111 (7)	H	010111 (23)	X	100111 (39)	n	110111 (55)	3
001000 (8)	I	011000 (24)	Y	101000 (40)	o	111000 (56)	4
001001 (9)	J	011001 (25)	Z	101001 (41)	p	111001 (57)	5
001010 (10)	K	011010 (26)	a	101010 (42)	q	111010 (58)	6
001011 (11)	L	011011 (27)	b	101011 (43)	r	111011 (59)	7
001100 (12)	M	011100 (28)	c	101100 (44)	s	111100 (60)	8
001101 (13)	N	011101 (29)	d	101101 (45)	t	111101 (61)	9
001110 (14)	O	011110 (30)	e	101110 (46)	u	111110 (62)	+
001111 (15)	P	011111 (31)	f	101111 (47)	v	111111 (63)	/

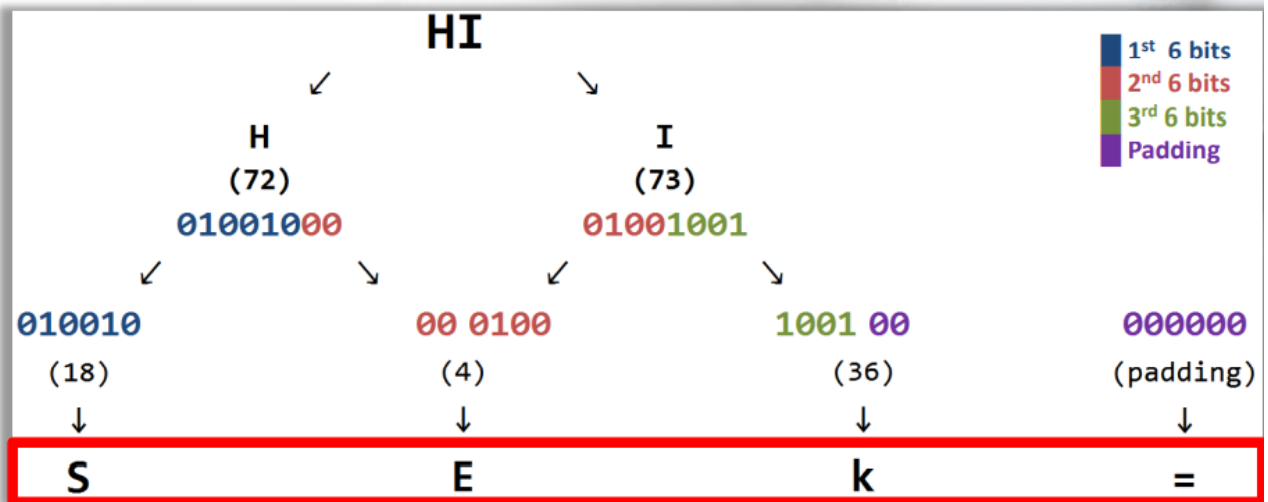
If the total number of bits is not a multiple of 6, then null bits need to be added until the total is both a multiple of 6 and the result length a multiple of 4.

666666 666666 666666 666666 and must a group contain 4 subgroups of 6 digits

if the latest group is 'null' (000000), encoding value is =
 if the trailing "null groups" are two they will be encoded as ==

Example

To encode the term "HI" we have:



Programming to Convert from x to y in base64

- PHP ->

```
<?=base64_encode('encode this string')?>
```

```
<?=base64_decode('ZW5jb2RlIHRobXMgc3RyaW5n')?>
```

- JS ->

```
window.btoa('encode this string');
```

```
window.atob('ZW5jb2RlIHRobXMgc3RyaW5n');
```

Important to notice

How we handle/encode Unicode String ? (using the usual way to encode will trigger an error..)

<https://developer.mozilla.org/en-US/docs/Web/API/btoa>

4. Unicode

website: <https://home.unicode.org/>

It supports all the world's writing systems.

-> U+0639 called code point. The U+ means "Unicode" and the numbers are hexadecimal.

Example: Hello -> U+0048 U+0065 U+006C U+006C U+006F

3 Way to map unicode character

- UTF-8 -> A character takes 1-4 bytes
- UTF-16 -> A character takes either 2 bytes or 4 bytes
- UTF-32 -> A character takes 4 bytes

-> difference: How many bytes it requires to represent a character in memory.

Unicode is a character set of various symbols, while UTF-8, UTF-16, and UTF-32 are different ways to represent them in byte format. Both UTF-8 and UTF-16 are variable-length encoding, where the number of bytes used depends upon Unicode code points and UTF-32 is fixed 4 bytes.

UTF -> Unicode Transformation Format

Table show msg encoded in 3 different UTF Format

CHARACTER	REPRESENTATION			
	Unicode	UTF-8 code point	UTF-16 code point	UTF-32 code point
I	U+0049	49	0049	00000049
♥	U+2665	E2 99 A5	2665	00002665
📄	U+1F37B	F0 9F 8D BB	D83C DF7B	0001F37B

different implementations like URLs, HTML, JavaScript

CHARACTER	REPRESENTATION					
	Unicode Code Point	URL-Encoding (UTF-8)	HTML Entity Named	HTML Entity Decimal	HTML Entity Hexadecimal	JavaScript, JSON, Java (UTF-16)
I	U+0049 (hex 49, dec 73)	%49	-	I	I	\u0049
♥	U+2665 (hex 2665, dec 9829)	%E2%99%A5	♥	♥	♥	\u2665
📄	U+1F37B (hex 1F37B, dec 127867)	%F0%9F%8D%BB	-	🍻	🍻	\uD83C\uDF7B

Homoglyph

-> a word with same mean but different shape

-> Ex: google == xn--goole-tmc

-> <https://www.irongeek.com/homoglyph-attack-generator.php>

-> https://github.com/codebox/homoglyph/blob/master/raw_data/chars.txt

An additional classification



- HOMOGRAPH > a word that looks the same as another word
- HOMOGLIPH > a look-alike character but not the same

Visual Spoofing attack

Visual Spoofing

U+006F
LATIN SMALL
LETTER O

U+03BF
GREEK SMALL
LETTER OMICRON

If we analyze the characters code points of the string, the differences between the  and the  are evident, but for a human this is not so obvious.

EX: AA A A Δ α A -> for human are same, for computer has different meaning.

Spoofing Domain Name / Extra add by me

<http://websec.github.io/unicode-security-guide/visual-spoofing/>

The links below explains a very dangerous attack trick!!!!

<https://www.youtube.com/watch?v=ynx-NxN5Axs>

<https://unicode-explorer.com/c/202E>

www.google.com is not www.google.com

Latin
U+0069

Latin
U+0261

identical domain names, however, the second contains the U+0261 LATIN SMALL LETTER SCRIPT G.

http://www.google.com/path/file.nottrusted.org

Katakana No
U+FF89

Computer Interpretation

➤ A user sends the following message:

Evil intent, as usual!

➤ The filter checks for evil strings, but without success.

U+0130 (İ)
LATIN CAPITAL LETTER I
WITH DOT ABOVE

Evil != evil



This may allow attack to bypass filters

There are other ways in which characters and strings can be transformed by software processes, such as normalization, canonicalization, best fit mapping, etc.

check: <http://websec.github.io/unicode-security-guide/character-transformations/>

Computer Interpretations: Mixed Examples

Normalization:

(d)(r)(o)(p) (t)(a)(b)(l)(e) becomes drop table

Canonicalization:

< (U+2039)
< (U+FE64)
< (U+ff1c) } **becomes** **< (U+003C)**

online tools:

<https://qaz.wtf/u/convert.cgi>

<https://xssor.io/>

reference to read:

<http://www.irongeek.com/i.php?page=security/out-of-character-use-of-punycode-and-homoglyph-attacks-to-obfuscate-urls-for-phishing>

<http://websec.github.io/unicode-security-guide/visual-spoofing/>

<https://engineering.atspotify.com/2013/06/18/creative-usernames/>

<http://websec.github.io/unicode-security-guide/character-transformations/>