CH 3: Network Protocol Structures

Sunday, December 23, 2018 3:04 AM

Binary Protocol Structures: Smallest unit of data single binary digit: **octet:** 8-bit units/bytes: unit of network protocols

Bit fmt:

0 (Bit 7/MSB)	1	0	0	0	0	0	**1 (Bit 0/LSB)**

= 0x41/65: Octet: 0x41

MSB: Most Significant Bit || LSB: Least Significant Bit

Numeric Data: Data values represented: Core of binary protocol: Ints/dec values: Length of data/ID tags

Unsigned ints: Based on position: Values added together to represent the int

Bit	Dec	Hex
0	1	0x01
1	2	0x02
2	4	0x04
3	8	0x08
4	16	0x10
5	32	0x20
6	64	0x40
7	128	0x80

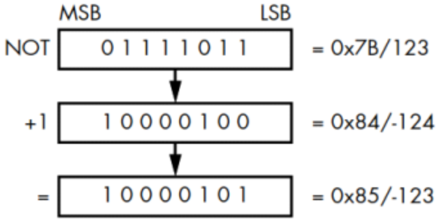
Signed ints: Not all ints positive: Neg ints req: Only signed ints can hold neg values

- CPU can only work w/same set of bits
- Reqs way of interpreting unsigned int value as signed: Two's complement

Two's complement: Way in which signed int represented w/in native int value in CPU

- Conversion bet unsigned/signed values in 2's complement done by taking bitwise NOT
  - 0 bit converted to 1 vice versa: Then adding 1

Example:



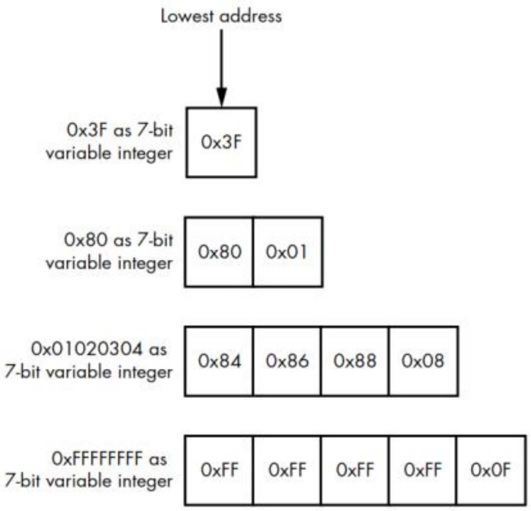
2's complement representation sec consequence:

- 8-bit signed int range: -128 to 127: Magnitude of min larger than max
- **If min value negated result is itself -(-128) is -128**
- Calcs incorrect parsed fmt leading to vuln

Note: Above img is confusing: NOT is auto flipped then +1 added for clarity

Var-Length Ints:

Length fields: When sending blocks of data bet 0-127 bytes in size: **Could use a 7-bit var int representation**


Lowest address
0x3F as 7-bit variable integer
0x3F
0x80 as 7-bit variable integer
0x80
0x01
0x01020304 as 7-bit variable integer
0x84
0x86
0x88
0x08
0xFFFFFFFF as 7-bit variable integer
0xFF
0xFF
0xFF
0xFF
0x0F

Parse more than 5 octets? Resulting int from parsing op will depend on parsing program

- Some programs will drop any bits beyond given range
- Other envs will generate int overflow: Possible BoF

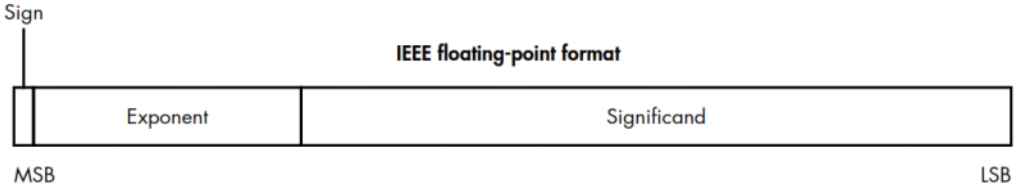
Floating-Point Data: Sometimes ints not enough to represent range of dec values needed for a protocol

- Could run against limited range of 32-/64-bit fixed-point value

IEEE Standard for Floating-Point Arithmetic [IEEE 754]

Standard specifies num of diff bin/dec fmts for floating-point values: Likely to encounter 2

1. **32-bit**
2. **64-bit: Double-precision**
    - Each specifies position/bit size of significand/exponent
    - **Sign bit also specified**: Indicates whether value positive/negative



Bit size	**Exponent bits**	**Significant bits**	**Value range**
32	8	23	+/-3.402823 x 10^38
64	11	52	+/-1.79769313486232 x 10^308

Booleans: Protocols: How to represent **true[1]/false[0]**

Bit Flags: 1 way to represent specific Booleans in protocol

Example: TCP: Bit flags to determine current state of connection

- Client: Sends packet SYN: Indicates connection should sync timers
- Server: Respond ACK to indicate client req as SYN: Establishes sync
- Handshake: Single enumerated values: Dual state impossible w/out SYN/ACK state

Binary Endian: How computers store data in mem:

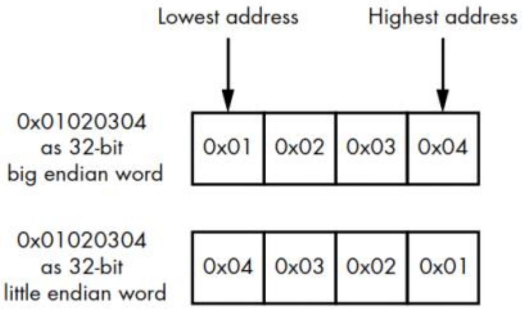
- Octets transmitted sequentially: Possible to send most significant octet of value as 1st part of transmission
- Least significant octet: Also possible to send as value of 1st part of transmission

Order in which octets sent determines endianness of data:

- Failure to handle endian fmt: Bugs in parsing protocols

Main Endian fmts:

1. **Big endian:** Stores **most** significant byte at lowest addr
2. **Little endian:** Stores **least** significant byte in lowest addr

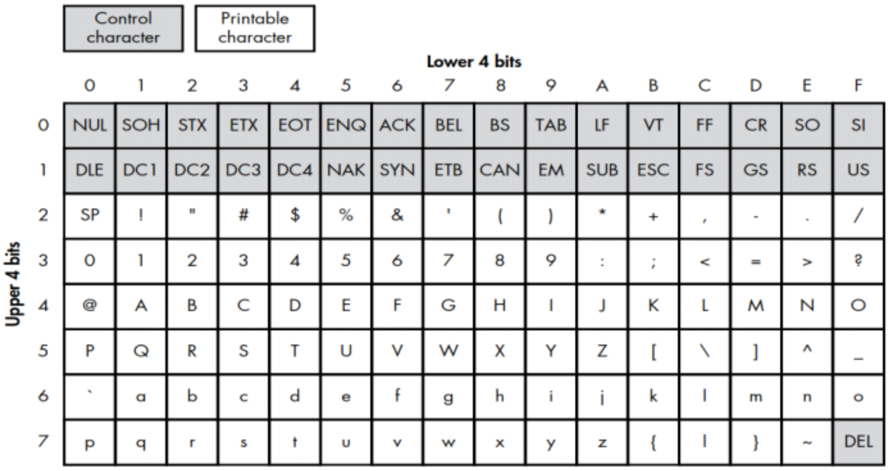

Lowest address
Highest address
0x01020304 as 32-bit big endian word
0x01
0x02
0x03
0x04
0x01020304 as 32-bit little endian word
0x04
0x03
0x02
0x01

Network/Host order: Endianness of value: Internet RFC's typically use big endian as preferred type for network protocols

- Big endian referred to as network order
- Computer could be big/little
- Proc arch: x86: Little endian: SPARC: Big endian

Text/Human-Readable Data: English chars: Encoded using ASCII

- Original ASCII standard defined 7-bit char set from 0 to 0x7F: Most to represent English


Control character
Printable character
Lower 4 bits
Upper 4 bits

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

<u>Originally for txt terms:</u> Control chars used to send msgs to terminal to move printing head to sync serial comms bet computer/term

ASCII char set 2 types of chars:

Control	
Printable	**Ones seen:** Familiar symbols/alphanumeric chars: Not useful to represent intl chars ▪ Can't represent fraction of possible chars in all world languages w/7-bit num **<u>Strategies to counter limit:</u>** **1. Code pages** **2. Multibyte character sets** **3. Unicode** Protocols: Req 1-of-3 ways to represent txt: Offers option app can select

More on counter limitations:

Code pages	**<u>Code pages/char encodings:</u>** Which chars mapped to which values codified in specifications **Simplest:** Extend ASCII char set: Recognized if all data stored in octets: ▪ **128 unused values 128-255:** Can be repurposed for storing extra chars ▪ **256 values:** Not enough to store all chars in all lang: Diff ways to use unused range
Multibyte char sets	**<u>Multibyte char sets:</u>** Allow use 2/more octets in seq to encode desired char Languages: **CJK:** AKA: Chinese/Japanese/Korean: ▪ Uses multibyte char sets combined w/ASCII to encode languages **<u>Common encodings:</u>** ▪ **Shift-JIS:** Japanese ▪ **GB2312:** Chinese
Unicode	**Standard: 1991:** Aim: Represent all languages w/in Unified Char Set: Multibyte char set ▪ Tries to encode all written languages: Archaic/constructed **Unicode defines 2 related concepts**

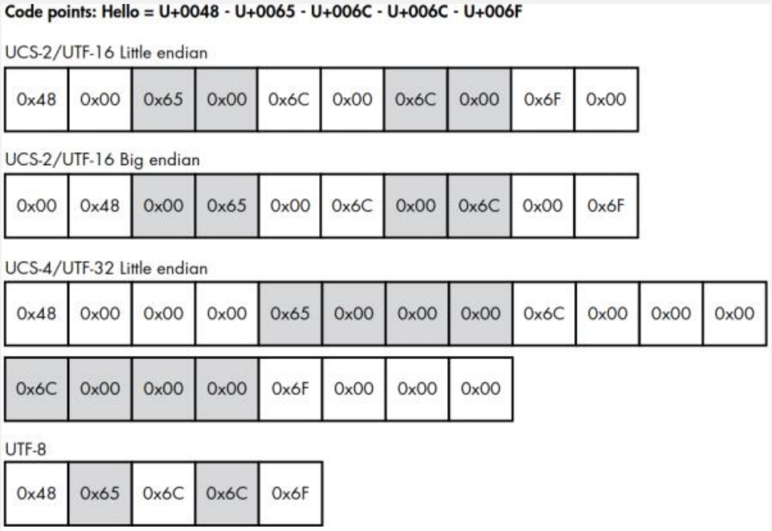
1. **Character mapping**
    - Mappings bet num value/char: Rules/reg on how chars used/combined
2. **Character encoding**
    - Define way num values encoded in underlying file/network protocol

Code point: Each char in Unicode assigned code point: Represents unique char

- Code points commonly written in fmt **U+ABCD**
- **ABCD: Code point's hex value**
- **1st 128 code points:** What's specified in ASCII
- **2nd 128 code points:** From **ISO/IEC 8859-1**

UCS: Universal Char Set || UTF: Unicode Transformation Fmt

<u>Encodings:</u> Resulting value encoded in one of said schemes



Code points: Hello = U+0048 - U+0065 - U+006C - U+006C - U+006F

UCS-2/UTF-16 Little endian

0x48	0x00	0x65	0x00	0x6C	0x00	0x6C	0x00	0x6F	0x00

UCS-2/UTF-16 Big endian

0x00	0x48	0x00	0x65	0x00	0x6C	0x00	0x6C	0x00	0x6F

UCS-4/UTF-32 Little endian

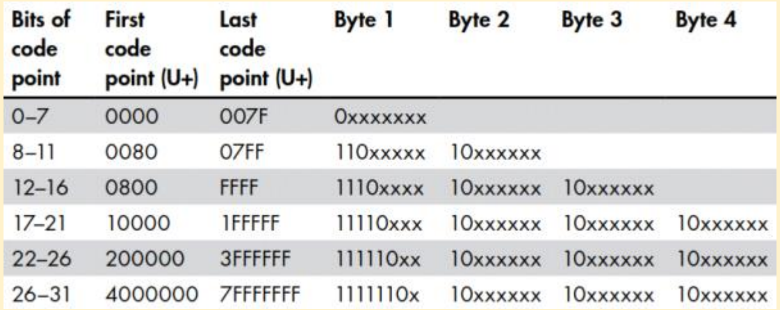
0x48	0x00	0x00	0x00	0x65	0x00	0x00	0x00	0x6C	0x00	0x00	0x00
0x6C	0x00	0x00	0x00	0x6F	0x00	0x00	0x00				

UTF-8

0x48	0x65	0x6C	0x6C	0x6F

3 Common Unicode encodings:

UCS-2/UTF-16	Native of MS Win/Java/.NET VM's when running code ▪ Code points: Seq of 16-bit ints ▪ Little/big endian variants
UCS-4/UTF-32	UNIX apps: Default wide-char fmt many C/C++ compilers ▪ Code points: Seq of 32-bit ints ▪ Diff endian variants
UTF-8	Most common UNIX: Default input/output fmt for platforms like XML ▪ Int size using simple var-length value: No fixed int size



Bits of code point	First code point (U+)	Last code point (U+)	Byte 1	Byte 2	Byte 3	Byte 4
0–7	0000	007F	0xxxxxxx			
8–11	0080	07FF	110xxxxx	10xxxxxx		
12–16	0800	FFFF	1110xxxx	10xxxxxx	10xxxxxx	
17–21	10000	1FFFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx
22–26	200000	3FFFFFF	111110xx	10xxxxxx	10xxxxxx	10xxxxxx
26–31	4000000	7FFFFFFF	1111110x	10xxxxxx	10xxxxxx	10xxxxxx

Incorrect/naïve char encoding: Source of subtle sec issues:

- Range: Bypassing filtering (req resource path)/BoF

Variable Binary Length Data:

- Dev knows exactly what data to be transmitted? Can ensure all values w/in protocol fixed length

Protocols use a few strategies to produce variable-length data values

1. **Terminated data**
2. **Length-prefixed data**
3. **Implicit-length data**
4. **Padded data**

Terminated

Variable-length int value terminated when octet's MSB 0

- Can extend concept of terminating values further to elements like strings/data arrays

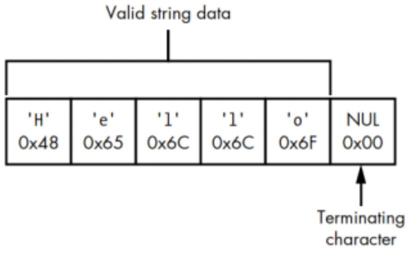
Terminated data value: Symbol defined: Tells parser end of data reached

- Unlikely present in typical data: Ensures value isn't terminated prematurely
- String data: Terminating value can be NUL value/1 of other control chars in ASCII set
- If term symbol occurs during normal data xfer: Need escape symbols

With strings:

- Common to see terminating char prefixed with \ or repeated 2x to prevent ID as term symbol
- Useful when protocol doesn't know ahead of time how long value is

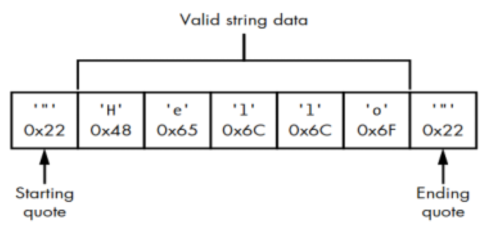
  Example: Generated dynamically



Bounded data often terminated by symbol that matches 1st char in var-length sequence

Example: String data w/quoted string in bet " "

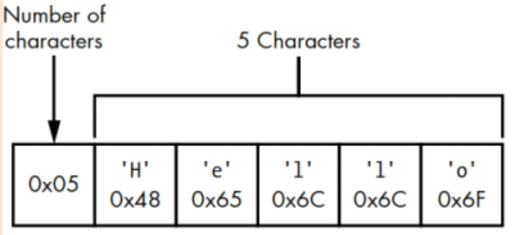
- Initial " " tells parser to look for matching char to end data



Length-Prefixed

If data value known: Possible to insert length into protocol directly

- Parser can read value/appropriate # of units (chars/octets) to extract original value: Common
- Actual length prefix/size not impt: Representative of data type transmitted
- Most protocols don't need to specify full range of 32-bit int
  - Size often a length field: Fits w/most processor arch/platforms



Implicit-Length

Sometime length implicit in values around it

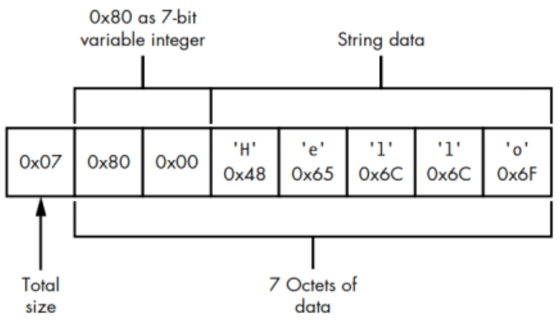
Example: TCP

- Protocol sending data back to client using connection-oriented protocol
- Instead of specifying data size: Server could close TCP connection
  - Implicitly signifies end of data: How data returned in HTTP version 1.0 response

Example II: Higher-lvl protocol/struct that already specified length of set of values

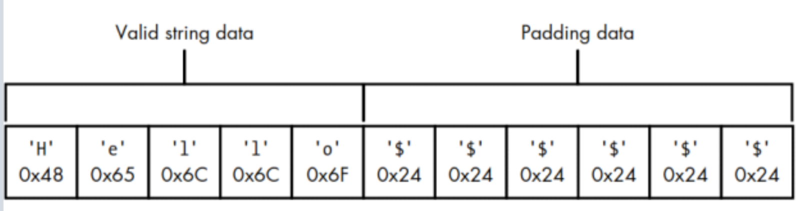
- Parser might extract higher-lvl struct 1st then read the values contained w/in
- Protocol could use struct w/finite length to implicitly calc length similar to closing connection

7-bit var int/str contained w/in single block:



Padded Data **Used when max upper bound on length of value like 32-octet limit**

- Instead of prefixing value w/length/explicit terminating value
- Protocol could send entire fixed-length str but terminate value by padding unused data w/known value



Dates/Times: Impt for protocols: Metadata: File mod timestamps: Determine expiration of auth credentials

- Failure: Serious sec issues: Depends on usage reqs platform/protocol space reqs

POSIX/UNIX Time **Stored as 32-bit signed int:** Represents num of sec elapsed since UNIX epoch **00:00:00 (UTC), 1 January 1980**

- **Value limited to 03:14:07 (UTC), 19 January 2038**
  - Representation will overflow
  - Some OS's use 64-bit representation to address issue

Windows FILETIME **MS filesystem timestamps:** Only fmt on Win w/simple bin representation

- In a few protocols: **Stored as 64-bit unsigned int**
- One unit of int: 100 ns interval

Epoch: **00:00:00 (UTC), 1 January 1601**

- Larger range than POSIX/UNIX

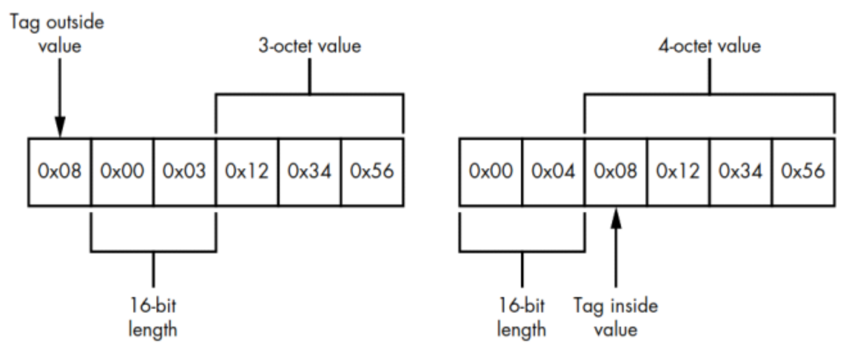
TLV: Tag, Length Value Pattern: Protocol can send diff types of structures must have way to represent bounds of struct/type

Tag value **Type of data being sent by protocol: Commonly num:** Can be anything that provides data structs w/unique pattern

- Can be used to determine how to further process data

Example: 2 types of Tags: 1 auth credentials to app: Other msg transmitted to parser

- Allows us to extend protocol w/out breaking apps not updated to support it
- Protocol parser can ignore structs that it doesn't understand



Length Variable-length value

Value Pattern Variable-length value

Multiplexing/Fragmentation: Multiple tasks happening at once

Multiplexing

Allows multiple connections to share same underlying network connection:

- Multiple types of traffic by fragmenting large transmissions to smaller chunks
- Combines chunks into single connection

Protocol analysis? Demultiplex chan to get original data out

Some protocols restrict type of data transmitted: How large each packet can be

- IP: Max traffic frames: 1500 octets: Packets 65,535

Fragmentation

Mech that allows network stack to convert large packets into smaller fragments

- OS knows entire packet can't be handled by next layer

Structured Bin Fmts:

ASN.1

Abstract Syntax Notation 1:

- Basis for protocols like SNMP: Simple Network Management Protocol
- Encoding mechanism for cryptographic values: X.509 certificates
- Standard: ISO/IEC/ITU: X.680 series

Defines abstract syntax to represent structured data

- Data represented depending on encoding rules
- **DER: Distinguished Encoding Rules**
  - Designed to represent ASN.1 structures that can't be misinterpreted
  - Property for cryptographic protocols
  - Representation of TLV protocol

Text Protocol Structures: Good choice when purpose to xfer txt: Mail/Msg/News

- Must have structures similar to bin protocols

Common text protocol structures:

Numeric Data

Integers

Simple representation: Size limitations no concern: Num larger than bin word/can add digits

- Hope protocol parser can handle the extra digits or sec issues:

<u>Make signed num:</u>

- Add - char to front of num
- Add + char for positive num

Dec Num

Defined using human-readable forms: Bin representations [floating points] can't represent all dec

- Can make some values diff to represent in txt fmt: Can cause sec issues

Txt Booleans

True/False: Some may require words be capitalized exactly to be valid

Dates/Times

Not everyone can agree on standard fmt: Many competing representations: Issue w/mail clients

Var-Length

When txt field separated out of original protocol: Token

- Some protocols specify fixed length for tokens: More common to req type of var-length data

Delimited txt

Separating tokens/field w/delimiting chars very common: Any char can be used as delimiter

- Whitespace usually encountered: Doesn't have to be
- **FIX: Financial Info Exchange** protocol delimits tokens using **ASCII SOH:**
  - **Start of Header** char w/value 1

Terminated txt

If separate individual tokens: Must also have way to define End of Command condition

- If protocol broken into separate lines: Must be terminated in some way
- HTTP/IRC: Line terminated protocols
- Typically delimit entire structs such as end of a cmd

OS dev: Usually define EOL char as:

- **LF: Line Feed: ASCII: Value 10**
- **CR: Carriage Return: Value 13**
- **Combo CR LF: EOL: End of Line combo**

Structured Txt Fmts:

MIME

Multipurpose Internet Mail Extensions: Dev for multipart email msgs: HTTP: RFC's 2045/46/47

- Separates body parts by defining common separator line prefixed w/2 --
- Msg terminated by following separator w/same 2 --
- Common uses: Content-Type values: MIME types

MIME type: Widely used w/HTTP content in OS to map app to particular content type
Each type consists of form of data: Txt/app

JSON

JavaScript Object Notation: Simple representation for struct based on object fmt

- Originally used to xfer data bet web page/backend service such as AJAX
- **AJAX: Asynchronous JavaScript/XML**
- Commonly used for web service data xfer/all manner of other protocols

JSON fmt: JSON object enclosed using {}

- W/in braces 0/more member entries
- Each consists of key/value

Example:

```
{
        "index"  : 0,
        "str"  : "Hello World!",
        "arr" : [ "A", "B" ]
}
```

Also designed for JS processing: Can be parsed using "eval" function

- Sec risk: Possible to insert arbitrary script code during object creation
- Lead to XSS

XML

Extensible Markup Language: Describing struct doc fmt

- Dev by W3C: Derived from **SGML: Standard Generalized Markup Lang**
- Similarities to HTML: Aims to be stricter in def to simplify parsers/create fewer sec issues

Consists of elements/attributes/txt

Elements: Main structural values

- Have name/can contain child elements/txt content
- Only 1 root element allowed in single doc

Attributes: Addl name-value pairs: Can be assigned to element

- Take form of **name="Value"**
- Txt is child of an element/value component of an attribute

Example:

```
<value index="0">    <str>Hello World!</str>
   <arr><value>A</value><value>B</value></arr>
</value>
```

All XML data txt: No type info provided: Parser must know what values represent

- Used in many ways: RSS: Rich Site Summary
- XMPP

Encoding Bin Data: Early comms: 8-bit bytes not norm: Most comm txt based: 7 bits per byte req by ASCII

- Allowed other bits to provide control for serial link protocols for perf
- SMTP/NNTP: Network News Transfer Protocol: Assume 7-but comm chans

7-but limitations: Problems w/pics/non-English char set

- Dev devised ways to encode bin data as txt
- Still has advantages: Ex. Sending bin data in structured txt fmt: JSON/XML: Delimiters properly escaped
- Can choose encoding fmt like Base64 to send bin data

Hex Encoding

Each octet split into 2 4-bit values converted to 2 txt chars denoting hex representation

- Not space efficient: Bin data auto becomes 100% larger
- Advantage: Encoding/decoding ops fast/simple

HTTP: Similar encoding for URL's/txt protocols

Percent encoding: Only nonprintable data converted to hex

- Values signified by prefixing w/% char

Base64

Counters inefficiencies w/Hex encoding: Dev as part of MIME spec:

- 64: Num of chars used to encode data
- Input bin separated into individual 6-bit values: 0-63
- Used to look up corresponding char in encoding table

Problems: 8-bits divided by 6: 2 bits remain

Counter?

- Input taken in units of 3 octets: Dividing 24 bits by 7 bits = 4 value
- Encodes 3 bytes into 4: Increase of 33%
- Better than hex

What if 1/2 octets to encode? Placeholder char =

- If no valid bits avail: Encoder will encode value as placeholder