CS3235 Part2 Lec1& Lec2

$_/|/|U_Ch@NgRu!$

December 16, 2021

1 Software and security

1.1 Two key problems

1. If code is right under expected scenario. Can we guarantee whether it will work as expected under adversary setting?
2. If code is right and works correctly in all setting, can we make sure make sure the code does not leak any sensitive information through tis trace of action

1.1.1 General Attack model

1.1.2 Attacker's goal

1. change/control the flow of the program execution?
2. extreact sensitive information from through the trace of action from the program
3. exploit the bug of the program to execute arbitary code?

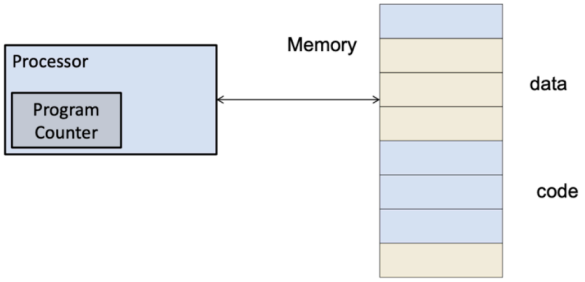
1.2 Attacker's capability

(RTBD: expand how each factor affect the attack model)

1. Understand C functions, stack and heap
2. May apply system call (e.g. exec())
3. OS information(Unix/Windows)
4. Stack frame structure
5. CPU information

1.3 Computer architecture and control flow

Modern computers are generally Von Neumann computer, which stores code and data together in the memory(in contrast of Harvard architecture, which stores the code and memory seperatedly), which means **Programs may be tricked into treating input data as code**



- Processor: electronic circuitry within computer that carries out the instructions of a computer program by performing basic arithmetic, logic, controlling, I/O operations specified by the instruction
- Program counter: also called program/instruction pointer, stores the address of the next instruction. After an instruction is executed, the process fetches the next instruction from the address stored in the program counter. Once this fetching is done, the program counter automatically change

  Change to program counter

  * increment by length of one instruction(increase 1 if the instruction length is fixed)
  * Direct jump: replace with a constant value specified in the instruction
  * Indirect jump:
    - Replace with a value fetched from memory
    - There are many forms of indirect jump (RTBD)

1.3.1 Processes

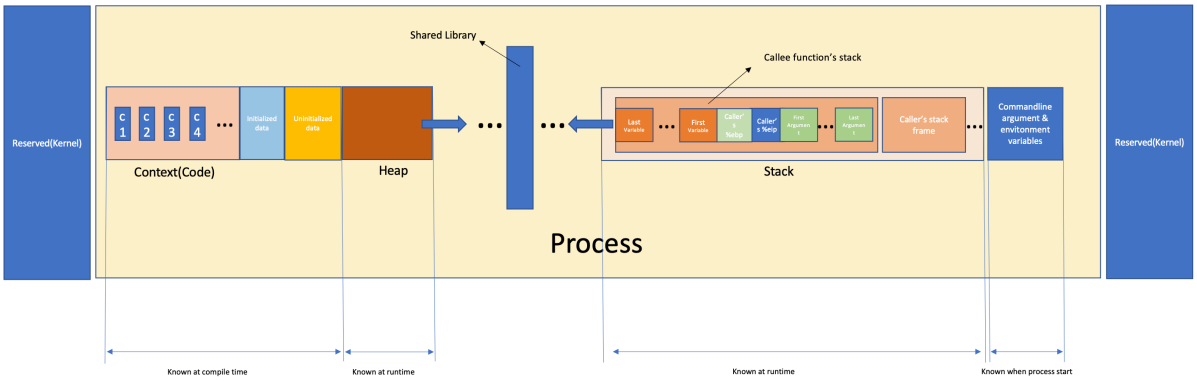
A process is the instance of a computer program that is being executed by one or many threads. It contains the program code and its activity(data, memory, and stack structure etc.).
In the research of software security, we abstract the architecture difference and understand the process's memory in virtual memory level, which means the process has unlimited memory in a certain OS dependent structure

1.3.2 Stack

Stack, also called execution stack or call stack, is the memory area used by the process to store local

2 Process in OS

The attacks on processes are based on the structure of processes



In the image above, relative dynamic parts are marked in warm color, while cold color is used to mark relative static parts

2.1 Reference of pointer in assembly

Register	Usage	Preserved across function calls
%rax	temporary register; with variable arguments passes information about the number of vector registers used; 1st return register	No
%rbx	callee-saved register; optionally used as base pointer	Yes
%rcx	used to pass 4th integer argument to functions	No
%rdx	used to pass 3rd argument to functions; 2nd return register	No
%rsp	stack pointer	Yes
%rbp	callee-saved register; optionally used as frame pointer	Yes
%rsi	used to pass 2nd argument to functions	No
%rdi	used to pass 1st argument to functions	No
%r8	used to pass 5th argument to functions	No
%r9	used to pass 6th argument to functions	No
%r10	temporary register, used for passing a function's static chain pointer	No
%r11	temporary register	No
%r12–r15	callee-saved registers	Yes
%xmm0–%xmm1	used to pass and return floating point arguments	No
%xmm2–%xmm7	used to pass floating point arguments	No
%xmm8–%xmm15	temporary registers	No
%mmx0–%mmx7	temporary registers	No
%k0–%k7	temporary registers	No
%st0,%st1	temporary registers; used to return long double arguments	No
%st2–%st7	temporary registers	No
%fs	Reserved for system (as thread specific data register)	No
mxcsr	SSE2 control and status word	partial
x87 SW	x87 status word	No
x87 CW	x87 control word	Yes
%bnd0–%bnd3	used to pass/return bounds of pointer arguments/return values	No

2.2 Analysis of the vulnerability of each part of process

1. Commandline argument & environment variables: this part is determined when the process start; the C-I-A on this part is generally determined by the C-I-A of Kernel; Generally the C-I-A of processes should be based on the C-I-A of kernel; Whether there is vulnerability in this part? (RTBD)
2. Arguments in stack:
   (a) Vulnerability may come from use the user's input as the argument of certain function directly without checking or restricting
3. %eip: store the address of caller function's code; this part usually cannot be modified by users legitimately, but the C-I-A of the variable may be compromised from both Stack and Heap base
4. %ebp: store the address of caller function's stack frame; this part usually cannot be modified by users legitimately, but the C-I-A of the variable may be compromised from both Stack and Heap base
5. Variable
   - pointer: may access to some un-intended data
   - when data is applied as the judgement conditions, overflow(integer overflow or ASCII overflow etc.) may make the confition not complete enough, which gives attacker to access to some un-authenticated flow
6. shared library: Attacker can exploit vulnerability to call functions in shared library(e.g: "/bin/sh") to achieve higher goal
7. Heap:
   - String is a pointer to certain address of heap
     – String format attack to compromise confidentiality
     – C-style String end with "\0", which can be obmitted or put at certain position to compromise C-I-A
8. Initialized data(e.g. static const int x=100;):further searching is needed: RTBD
9. Uninitialized data(e.g. static int x=100;):further searching is needed: RTBD

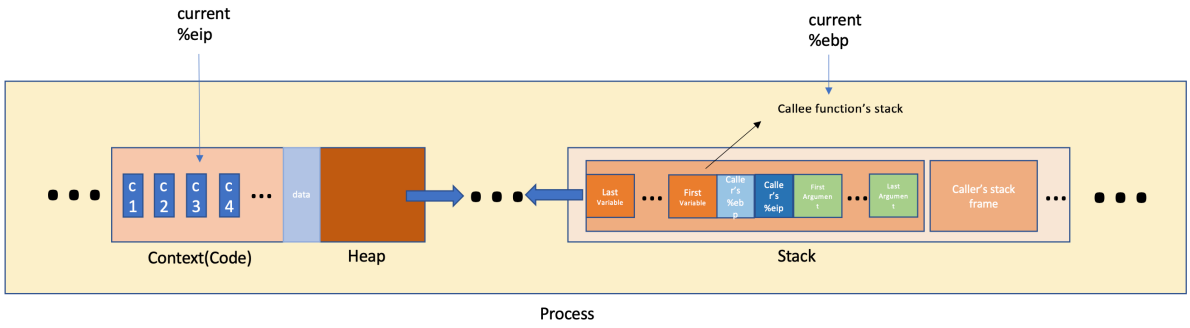
2.3 Process of calling function

2.3.1 Function

In process, function break code into smaller pieces and facilitate modular design and code reuse. A function code can be called in many different locations(even at the same time).Here to make it clear, we say

- function code: the code for a function
- function: the code of a function(static) and stack of the function(dynamic)

How does the program know the control flow of the process(i.e. the create, elimination and cohesion of function)



Here given that: function in c4 calls function in c3

Call function

1. Push arguments on the stack(in reverse order) (why? RTBD)
2. Push the return address(i.e. stores c4 in "caller's %eip") (The address that containing the code of the caller function: i.e. caller function's %eip)(at this stage, the process's %eip still point to c4)
3. Jump to the function's address(i.e. %eip points to c3)
4. Push the caller's frame pointer(%ebp)(after
5. Set the processes frame pointer at the end of the stack, which informs the starting point of the data of callee function
6. Push local variables

(Here allocation of dynamic memory is obmitted)

Return function

1. Reset the previous stack frame(set the process's frame pointer to caller function's frame pointer, pop the part of stack that belongs to callee)
2. Jump back to return address(i.e. c4)

2.4 Security requirements

By the key software security problem and security requirements, we should consider

1. Confidentiality
   (a) Confidentiality of arguments/variables when the process start
   (b) Confidentiality of data during compiling
   (c) Confidentiality of variable and data during runtime
      i. Based on variable
      ii. Based on pointer
      iii. Based on string format function
      iv. Based on Heap
   (d) Confidentiality of data/file that the process having access to during runtime
2. Integrity
   (a) Flow integrity during compiling
   (b) Flow integrity during runtime
      i. Based on Stack
      ii. Based on Heap
      iii. Based on Variable
   (c) Data integrity during runtime
3. Availability
4. Authentication(Here process's)

3 Attack %eip

%eip will point to the address of code that will be executed after the function is returned, therefore, the confidentiality of %eip will decide the confidentiality of the process flow and the integrity of %eip will influence the integrity of flow

3.1 Attacks confidentiality of %eip

1. From heap:
   - Use un-terminated string(string with no \0)
   - Use buffer overflow(pointer to the head of a array + certain offset)
2. Dangling pointer: i.e. the pointer that points to a free variable(even if the variable is freed, the address may be applied later for other use, which can be illegally accessed by the dangling pointer)

3.2 Attack Integrity of %eip

1. From heap:
   - Apply buffer overflow or Dangling pointer to modify %eip in order to change the flow of the process, the modified address can be
     - Pre-inserted code in buffer
     - Code in shared library
     - Code in context part of the process
     - Pre-inserted code in heap(?)
     - A random place to compromise the availability of the process

3.2.1 Return Oriented Programming(ROP)

Thesis: In any sufficiently large X86 executable code, there will exist sufficiently many useful code sequence available for attackers who control the stack to exploit program to undertake arbitrary computation

Idea: instead of calling a single function, compose shellcode by putting together pieces of existing code, called gadgets
(usually all gadgets end with ret)

Challenge of ROP

Challenge of finding gadgets

Challenge of sequencing the gadgets to make the attack

Blind ROP: in case where the attackers do not known exactly teh address of code in memory, but the ROP can still be conducded, ROP works if:

1. If server starts adter crash, but does not rerandomize
2. the confidentiality of stack is compromised
3. Initiate a write() system call to send a dump of the binary to the attack(RTBD)

3.3 Defense related to %eip

Generally compromising the confidentiality and integrity of %eip is in order to achieve higher goal(jump to some executable), therefore we can

1. make it hard to find the executable
   - Add many nop(no operation) instruction prior to the shell code
   - Address space layout randomization(ASLR)
2. make it hard to create executable: Non-executation memory(e.g. Data execution prevention/DEP in Windows)
3. detect the modification to stack(stack canaries)

3.3.1 NOP

The NOP sled or NOP ramp is a sequence of NOP (no-operation) instructions meant to "slide" the CPU' s instruction execution flow to its final, desired destination whenever the program branches to a memory address anywhere on the slide.
This is because sometimes attackers may not be able to have the return address point to the exact location or when the target address that the control flow is looking for is not known precisely. It creates a greater area for attackers to strike and still ensure that their shellcode will run.
While a NOP slide will function if it consists of a list of canonical NOP instructions, the presence of such code is suspicious and easy to automatically detect.
For this reason, practical NOP slides are often composed of non-canonical NOP instructions such as a moving a program counter/register to itself or adding zero i.e. instructions that affect program state inconsequentially, which makes them a lot more difficult to identify.

3.3.2 ASLR

RTBD

3.3.3 DEP

RTBD

3.3.4 Stack Canaries

RTBD

4 Attack %ebp

4.1 Attack confidentiality of %ebp

Basically the same as attack confidentiality of %eip

4.2 Attack integrity of %ebp

Way:

- from heap: buffer overflow
- dangling pointer

Compromising the %ebp can also change the flow of process, (usually cooperation with the compromization of %eip);
e.g.

F1 calls F2, F2 calls F3

- Here we modify the %ebp in F3's stack pointing to F1 (pointing to F2's stack frame before modification), and may also modify the %eip to code of F1,
- This will make the F3 return to F1 directly(skip F2), and compromise the flow integrity of the process

5 Attack "Variable"

5.1 Integer overflow

- In C++/C
  - int: 4 bytes (-2147483648 to 2147483647 $[-2^{31}$ *to* $2^{31}-1]$)

- unsigned int: 4 bytes(0 to 4294967295)

5.1.1 How integer overflow can be applied

1. If an integer variable is applied as the judgement factor of if/while condition, and the value of integer variable can be influenced by users, attacker may apply integer overflow to execute un-authoritative function or flow
2. If an integer variable is applied as index for a array/vector, integer overflow can be applied to make the index accessible to any location even if only one math operator is available(e.g. Only have "+","-" or "*")

6 Attack & Shared Library

Unsecure and high-privileged code may be stored in the shared library, which may be applied by attacker in ROP or just %eip jump e.g.

- strcpy(char * dest, const char * src)
- strcat(char * dest, const char * src)
- get(char *)
- scanf(const char * format,...)

6.1 "secure" lib function

- strncpy()
- strncat()

Note that the "secure" lib function may still have vulnerability if applied wrongly

6.2 e.g.

strncpy may make string unterminated overflow until the first \0 (p.s. Windows C run (CRT) applies strcpy_s(*dest, DestSize, *src) ensures proper termination(will auto-append \0))

7 Attack Heap

7.1 Buffer overflow

Buffer overflow means putting more into the buffer than it is intended to hold Generally related to string and array

- Array access: pointer to the head of array+ offset; It's a vulnerability if the compile has no check of the range of the array
- String:

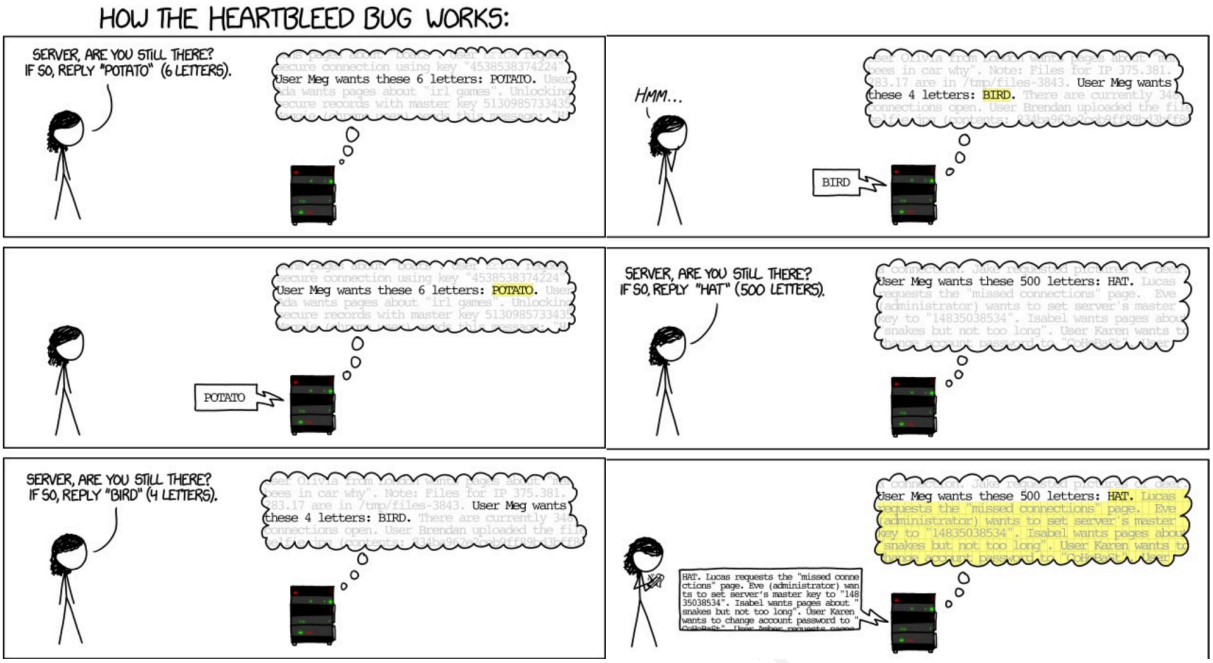
    e.g. strcpy

        overwrite other value(data integrity, %eip & %ebp integrity, stack integrity etc.)

7.2 String

In C, String is a special form of array, in generally a array of char(in simplest problem, a array of ASCII chars, but can also be array of utf-8 chars where one char may take 3 or 4 bytes in Chinese, for example); End of string is detected by \, and no range check in C
Vulnerability related to string in C includes

- The string may not end with "\0", and hence print the contends following the string unitl the first "\0"
- buffer overflow
  - (e.g. strcpy)
  - HeartBleed attack(the server does not verify the string length given by user, which makes user able to get information more than the string by adjusting the string length)
    1. Client sends a heartbeat message to the SSL server to echo back the message if it is alive
    2. The heartbeat message contains the length of the echo message
    3. The SSL software did not check if the specified length match the length of the message
    4. The attacker can specify a long enough length to read information stored in buffer beyond the message



- prinf() and String format attack printf() is a C function for formatting output. It is special in that it can take in **any** number of arguments
  - Format specifiers(%s,%d,...) indicates the type of argument, and their position in string indicate which stack argument to print

%%	a percent sign
%c	a character with the given number
%s	a string
%d	a signed integer, in decimal
%u	an unsigned integer, in decimal
%o	an unsigned integer, in octal
%x	an unsigned integer, in hexadecimal
%e	a floating-point number, in scientific notation
%f	a floating-point number, in fixed decimal notation
%g	a floating-point number, in %e or %f notation
%n	Nothing printed. The argument is usually a pointer to a signed int, where the number of characters written so far is stored.

Note that %s and %n are passed as a reference.

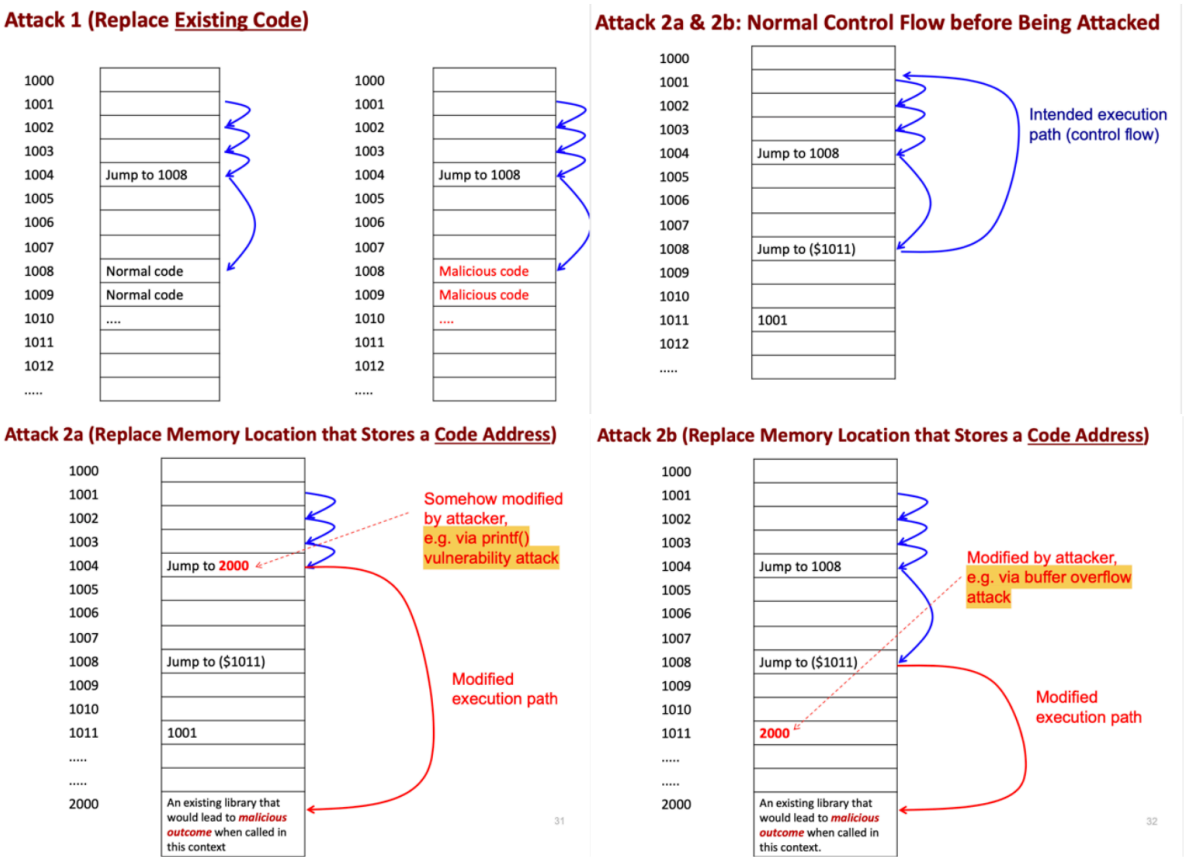
```
printf("\%d"); //print stack entry 4 bytes above saved \%ebp
//
printf("\%s" ) ;// “%s” will fetch a number from the stack and treat this number
    as an address.
//if the number is a valid address, The printf() function will try to print out
    the memory contents pointed by this address as a string, until a null
    character is read.
//randomly fetched number is not an address,thus the memory pointed at by this
    number does not exist.The program will thus crash
//
printf("\%d\%d\%d\%d\%d\%d\%d\%d...");  // print a series of stack entries as
    integer from top of the stack
//
printf("write14tostack%n"); //there are 14 characters printed out before %n, so
    write the number 14 address pointed by the stack frame
```

7.3 Code injection

7.4 Conclusion of the process integrity attack related to buffer overflow

Assuming that the attacker can now write to some memory locations, the attacker could:

1. Overwrite existing execution code portion with malicious code
2. Overwrite a piece of control flow information
    a. Replace a memory location that storing a code address that is used by direct jump
    b. Replace a memory localtion that storing a code address that is used by indirect jump

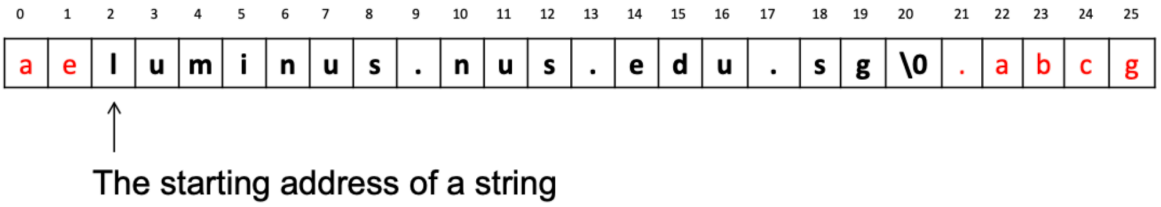


8 System Difference and Security

Different parts of a program or system may adopt different data representations. Such inconsistencies could lead to vulnerabilities.

8.1 Inconsistency of treating string

For example, CVE-2013-4073: "Ruby' s SSL client implements hostname identity check, but it does not properly handle hostnames in the certificate that contain null bytes."
In C, printf() adopts a efficient representation, where the length is not explicitly stored, and the first occurrence of the null character i.e. byte with value 0, indicates the end of the string, thus implicitly providing the length.



However, not all systems adopt this convention. There are two types:

- NULL-termination representation
- Non-NULL-termination representation

Exploitable Vulnerability 1: NULL-Byte Injection
A Certificate Authority may accept a hostname containing a null character, e.g.

luminus.nus.edu.sg\0.attacker.com

A verifier who uses both of the above representation conventions to verify the certificate could be vulnerable. Consider a browser that does this:

- It verifies a certificate based on a non-NULL-termination representation
- It compares the name in the certificate and the name entered by user based on the NULL-termination representation

We can thus have this scenario:

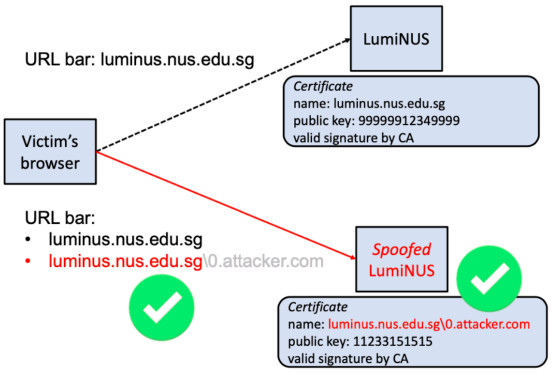
1. Let' s say an attacker registers the following domain name and purchases a valid certificate with this domain name from some CA: luminus.nus.edu.sg\0.attacker.com
2. The attacker then sets up a spoofed LumiNUS website on another web server. The attacker directs the victim to the spoofed web server by controlling the physical later or via social engineering.
3. When visiting the spoofed site, the victim' s browser would then:
   (a) Find that the web server in the certificate is valid based on the non-NULL representation i.e. luminus.nus.edu.sg\0.attacker.com
   (b) Compares and displays the address as luminus.nus.edu.sg based on the NULL-termination representation

   This is more effective than the normal web-spoofing attack, as a careful user would notice that the address displayed in the address bar is not LumiNUS, or that the address bar

displays luminus.nus.edu.sg but the TLS/SSL authentication protocol rejects the connection i.e. certificate is not trusted.

Thus this attack is much more dangerous.

Below is a slide showing a summary:



8.2 Encoding: ASCII and UTF-8

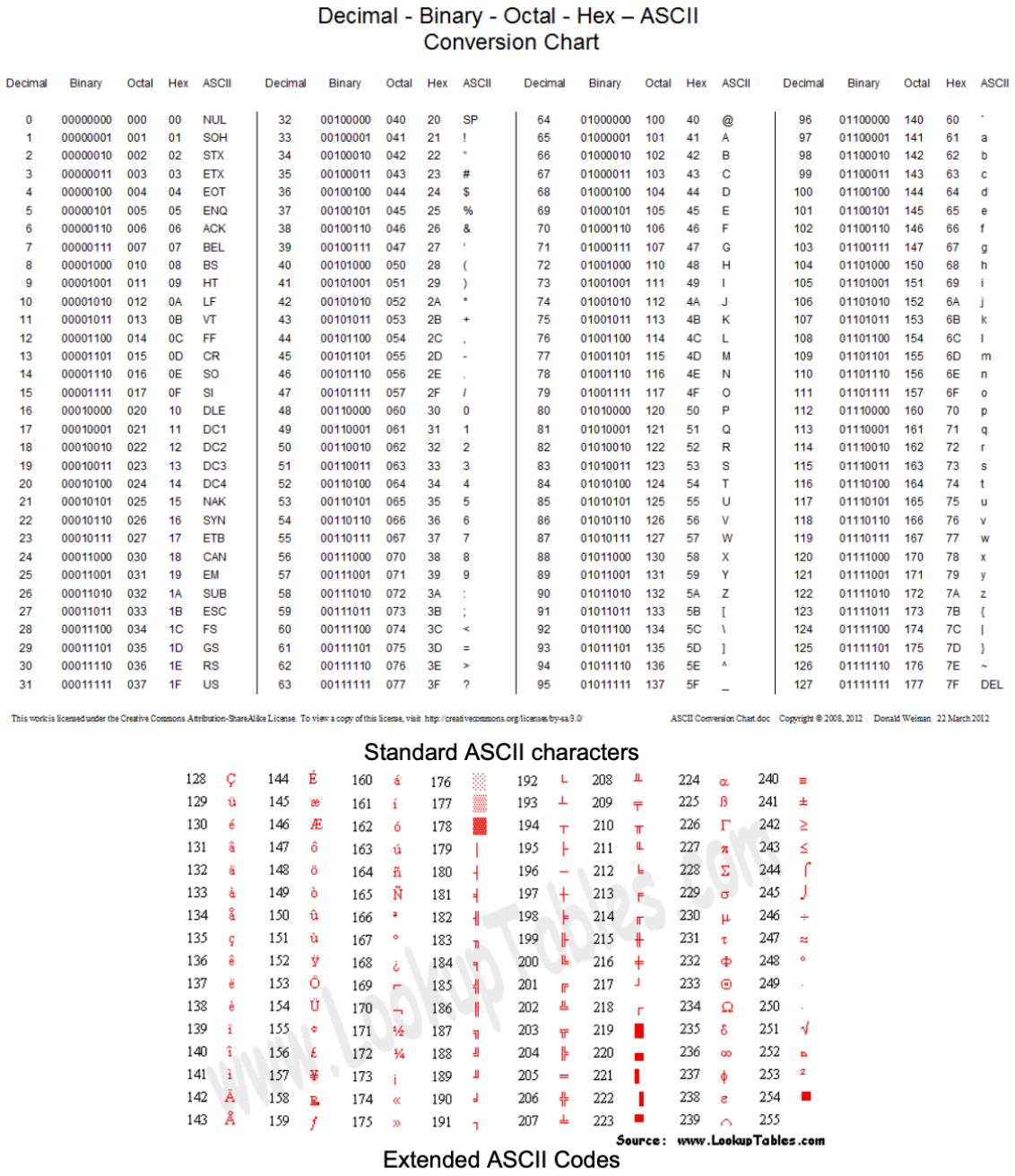
8.2.1 ASCII

American Standard Code for Information Interchange (ASCII)

ASCII character encoding is a standard for electronic communication. It encodes 128 characters into 7-bit integers, with 95 printable characters (digits, letters, punctuation symbols) and 33 non-printing (control) characters.

There is also an Extended ASCII, EASCII or high ASCII character encoding, which comprises of:

- The standard 7-bit ASCII characters
- Additional characters



Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Standard ASCII characters

128	Ç	144	É	160	á	176	▒	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	|	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	n
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Extended ASCII Codes

8.2.2 Unicode Transformation Format 8-bit (UTF-8)

UTF-8 is a character encoding that is capable of encoding all 1,112,064 valid code points in Unicode using one to four 8-bit bytes. It is a variable-length encoding, where code points with higher

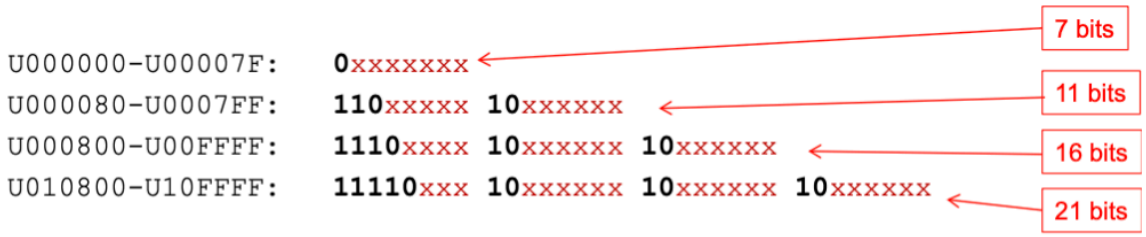
frequency of occurring are encoded with lower numerical values, thus using fewer byes.

Code point <-> UTF-8 conversion

First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
U+0000	U+007F	0xxxxxxx			
U+0080	U+07FF	110xxxxx	10xxxxxx		
U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
U+10000	[nb 2]U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

The first 128 characters (US-ASCII) need one byte.A ”character” can actually take more than 4 bytes, e.g. an emoji flag character takes 8 bytes since it’s ”constructed from a pair of Unicode scalar values”. The first 128 characters of Unicode correspond 1-to-1 with ASCII, and is encoded using a single octet with the same binary value as ASCII. Each byte thus starts with the bit 0 followed by the 7 bits of the original ASCII bits. Hence ASCII characters remain unchanged in UTF-8. There is backward compatibility with ASCII, as UTF-8 encoding was defined for Unicode on systems that were designed for ASCII.
The next 1,920 characters need two bytes to encode, which covers the remainder of almost all Latin-script alphabets, and also IPA extensions, Greek, Cyrillic, Coptic, Armenian, Hebrew, Arabic, Syriac, Thaana and N’Ko alphabets, as well as Combining Diacritical Marks. Three bytes are needed for characters in the rest of the Basic Multilingual Plane, which contains virtually all characters in common use,[14] including most Chinese, Japanese and Korean characters. Four bytes are needed for characters in the other planes of Unicode, which include less common CJK characters, various historic scripts, mathematical symbols, and emoji (pictographic symbols).

Exploitable Vulnerability : UTF-8 “Variant” Encoding Issues A Unicode character is referred to by “U+” and its hexadecimal digits. The following are byte representations of Unicode characters, with the left-hand side being the Unicode representation, and the right-hand side being the byte representation:



Notice that the prefix bits in the first byte changes based on the overall length, and that there are prefix bits as well in the continuation bytes or following bytes. The xxx bits are replaced by the significant bits of the code point of the respective Unicode character. By the rules above, the byte representation of a UTF-8 character is unique.

However, many implementations also accept multiple and longer "variants" of a character! In other words, there is more than one way to represent a single character using UTF-8. The reason is that different interpreters of UTF-8 interpret differently, and there is some room to accommodate the differences.

8.2.3 Vulnerability Example 1

Example 1: '/' (Representation using UTF-8 encoding)
Consider the ASCII character '/', whose ASCII code is: 0010 1111 = 0x2F

Under the UTF-8 definition, a 1-byte 2F is a unique representation. However, in many implementations, the following longer variants are also decoded to be '/':

- (2-byte) 11000000 10101111
- (3-byte) 11100000 10000000 10101111
- (4-byte) 11110000 10000000 10000000 10101111

e.g. English and Chinese input model, both has "/", the "/" inputed by both typewriting achieves the same result, but one UTF-8 Chinese character takes 4 bytes while one English ASCII-code takes 1 byte only
There is potential inconsistency when doing character verification before any operations that use this character.

Scenario: In a typical file system, files are organised inside a directory. Suppose there is a server-side program that receives a string <file-name> from a client and carries out the following steps:

Step 1: Append $< file-name >$ to the prefix (directory) string /home/student/alice/public_html/ and take the concatenated string as string F

Step 2: Invoke a system call to open the file F and send the file content to the client

In the above example, the client can be any remote public user, i.e. similar to a HTTP client. The original intention is the limit the files that the client can retrieve to only those under the directory public_html. This is called **file-access containment**.
However, an attacker may send in this string: ../cs2107report.pdf
The server would then try to read /home/student/alice/public_html/../cs2107report.pdf, which violates the intended file-access containment. To prevent this, the server may add an input validation step, making sure that the substring "../" does not appear within the input string. In other words, there is now a:

Step 1.5: Check that the <file-name> does not contain the substring "../" , else quit.

Vulnerability caused by utf-8 Let us assume that the system call in Step 2 above uses a convention that can process "%" followed by two hexadecimal digits as a single byte, similar to URL encoding, e.g. "/home/student/%61lice/" will have the %61 replaced by a to give "/home/student/alice/" .
We also assume that the system call uses UTF-8.
Then the following strings will actually all be equivalent to the string "../cs2107report.pdf" :

- "%2Fcs2107report.pdf"
- "%C0%AFcs2107report.pdf"
- "%E0%80%AFcs2107report.pdf"
- "%F0%E0%80%AFcs2107report.pdf"

All these inputs, when decoded, will give the same system call as before. In general, a blacklisting-based filtering system can be incomplete due to the "flexibility" of character encoding.

8.2.4 Vulnerability Example 2

Example 2: IP Address (Representation as Strings and Integers)
The 4-byte IP address is typically written as a string, e.g. 132.127.8.16. Consider a blacklist that contains a list of banned IP addresses, where each IP address is represented as 4 bytes.
Assume that there is a function BL() (blacklist) that takes in 4 integers of type int (i.e. 32-bits) and checks whether the IP address represented by these 4 integers is in the blacklist: int BL(int a, int b, int c, int d)
There are thus 4 arrays of integers, named A, B, C and D, and it simply tries to find i such that

A[i]==a, B[i]==b, C[i]==c, and D[i]==d.

The overall program thus does the following:

1. Get string s from user
2. Check if the s is of the correct format i.e. 4 integers separated by "." . If not, quit, else extract a, b, c and d.
3. Call BL() to check, if in blacklist, quit.
4. Let $ip = a * 2^{24} + b * 2^{16} + c * 2^8 + d$, where ip is a 32-bit integer
5. Continue the remaining processes with filtered address ip.

What can happen now is that this process can still be exploited, as integers can go negative. Unexpected and undesired results may occur.

How to deal with issues with Data Representation: Use Canonical Representation
The important lesson is that we cannot trust input from the user, and we can never directly use input from them. Always convert the input to a standard i.e. canonical representation immediately.
Preferably, do not rely on the verification check done in the application i.e. do not rely on the application developers to write the verification. Rather, try to make use of the underlying **system access control mechanism.**

8.3 Big-Endian and Little-Endian

CPU	Endianess
X86	Small/little-endian
Motorola	Big-endian
PowerPC	Big-endian

8.3.1 Difference bettween big-endian and little-endian

Definition:

1. Big-endian: the most significant byte of the integer is stored in the smallest address
2. Little-endian: the most significant byte of the integer is stored in the smallest address

e.g.

```
addr.|
200  | char str[30]
.    |
.    |
.    |
230  | int a
```

Here, a is an int, which takes 4 bytes
If a = 1147483647 = 0x446535FF

Big endian

```
addr.|
200  | char str[30]
.    |
.    |
.    |
230  | 0x44
231  | 0x65
232  | 0x35
233  | 0xFF
```

If the program mistakenly execute "str[30]=0"(note that the legitimate range of str[] if from str[0] to str[29])

then the most significant byte of a will be modified to 0x0 from 0x44

Then "a" will be modified to 0x006535FF=6632959_{10}

Little endian

```
addr.|
200  | char str[30]
.    |
.    |
.    |
230  | 0xFF
231  | 0x35
232  | 0x65
233  | 0x44
```

If the program mistakenly execute "str[30]=0"(note that the legitimate range of str[] if from str[0] to str[29])

then the least significant byte of a will be modified to 0x0 from 0xFF

Then ”a” will be modified to 0x446535FF=1147483392_{10}

9 SQL injection

RTBD

Apart from compromising the C-I-A directly