Get Exam Results earlier via SMS on 24 Dec 2019!

For more information ➔

https://myportal.nus.edu.sg/studentportal/academics/all/scheduleofresultsrelease.html

Subscribe via NUS EduRec System by 10 Dec 2019.

Reading:
See [PF] pg. 131-166, or
See [Gollmann] 10.1, 10.2, 10.3

Lecture 8: Software Security (Part II)

8.1 Data Representation & Security

8.2 Buffer Overflow

8.3 Integer Overflow

8.4 Code/Script Injection

8.5 Undocumented Access Points

8.1 Data Representation & Security

Data Representation Problem

- **Different** parts of a program/system adopts **different** data representations
- Such **inconsistencies** could lead to vulnerability
- A sample vulnerability is 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***."

  (Read https://www.ruby-lang.org/en/news/2013/06/27/hostname-check-bypassing-vulnerability-in-openssl-client-cve-2013-4073/.)

- **String** is a very important data representation type:
  - It has a **variable** length
  - How can we **represent** a string?

String Representations

- In C, `printf()` adopts an **efficient** representation:
  - The length is ***not*** explicitly stored
  - **The first occurrence** of the **null character** (i.e. byte with value 0) indicates the **end of the string,** thus *implicitly* giving the length

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
a	e	**l**	**u**	**m**	**i**	**n**	**u**	**s**	**.**	**n**	**u**	**s**	**.**	**e**	**d**	**u**	**.**	**s**	**g**	**\0**	.	a	b	c	g

The starting address of a string

- Note that ***not*** all systems adopt this convention: **NULL-termination** vs **non NULL-termination representation**

Exploitable Vulnerability 1: NULL-Byte Injection

- A CA ***may*** **accept** a host name containing null character
- For example: luminus.nus.edu.sg\0.attacker.com
- A verifier who uses **both** string-representation conventions to verify the certificate ***could be*** vulnerable
- Consider a browser implementation that does the following:
  1. Verify a certificate: based on **non NULL-termination** representation
  2. Compare the name in the certificate and the name enter by user: based on the **NULL-termination** representation
- Now, there could be an **attack** as described on the next slide!

A Sample Attack (on LumiNUS)

1. The attacker registered the following **domain name**, and purchased a **valid certificate** with the domain name from some CA:

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

2. The attacker set up a **spoofed LumiNUS** website on another web server

3. The attacker **directed** a victim to the **spoofed web server** (e.g. by controlling the physical layer or social engineering)

4. When visiting the spoofed web server, the victim's browser:
   - Finds that the Web server in the certificate is **valid**: based on the ***non*** NULL-termination representation
   - Compares and displays the address as **luminus.nus.edu.sg**: based on NULL-termination representation

.

Comparison: A Normal Web-Spoofing Attack (on LumiNUS)

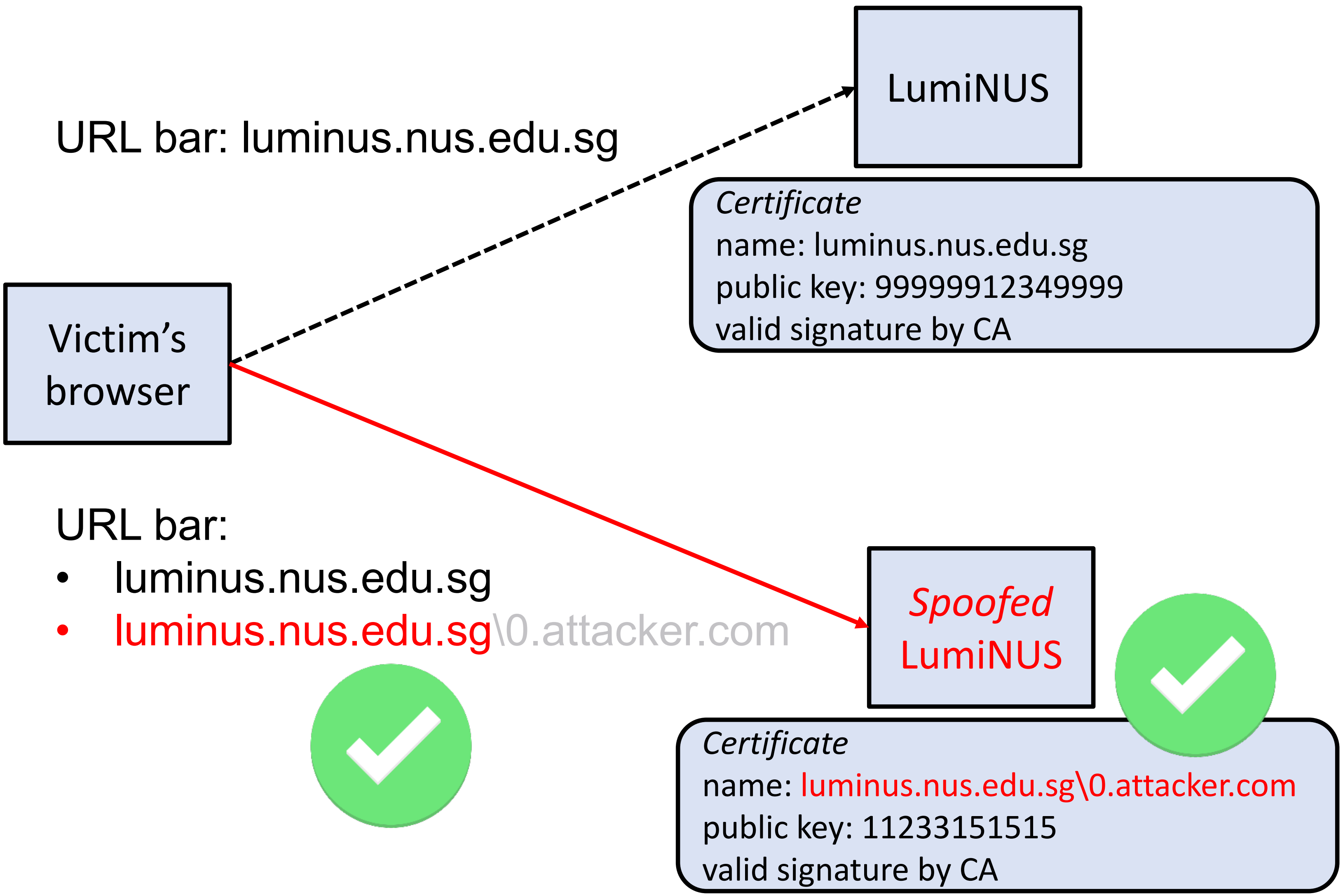
What if it is just a **normal web-spoofing** attack scenario?

Even if the attacker manages to redirect the victim to the spoofed web server (Step 3), a **careful** user would notice that *either*:

- The address displayed in the browser's address bar is ***not*** LumiNUS; or
- The address bar displays luminus.nus.edu.sg, but the TLS/SSL authentication protocol **rejects** the connection (i.e. "certificate is ***not*** trusted")

Hence, the attack on the previous slide is **much more dangerous**: it can **trick** all browser users!

A Sample Attack (on LumiNUS): Illustration



CVE-2013-4073:

What is **CVE**?

What is **zero-day vulnerability**?

What is an **exploit**?

Hostname check bypassing vulnerability in SSL client (CVE-2013-4073)

Posted by nahi on 27 Jun 2013

A vulnerability in Ruby's SSL client that could allow man-in-the-middle attackers to spoof SSL servers via valid certificate issued by a trusted certification authority.

This vulnerability has been assigned the CVE identifier CVE-2013-4073.

Summary

Ruby's SSL client implements hostname identity check but it does not properly handle hostnames in the certificate that contain null bytes.

Details

`OpenSSL::SSL.verify_certificate_identity` implements RFC2818 Server Identity check for Ruby's SSL client but it does not properly handle hostnames in the subjectAltName X509 extension that contain null bytes.

Existing code in `lib/openssl/ssl.rb` uses `OpenSSL::X509::Extension#value` for extracting identity from subjectAltName. `Extension#value` depends on the OpenSSL function `X509V3_EXT_print()` and for dNSName of subjectAltName it utilizes `sprintf()` that is known as null byte unsafe. As a result `Extension#value` returns 'www.ruby-lang.org' if the subjectAltName is 'www.ruby-lang.org\0.example.com' and `OpenSSL::SSL.verify_certificate_identity` wrongly identifies the certificate as one for 'www.ruby-lang.org'.

When a CA that is trusted by an SSL client allows to issue a server certificate that has a null byte in subjectAltName, remote attackers can obtain the certificate for 'www.ruby-lang.org\0.example.com' from the CA to spoof 'www.ruby-lang.org' and do a man-in-the-middle attack between Ruby's SSL client and SSL servers.

Background: ASCII Character Encoding

- **ASCII** (American Standard Code for Information Interchange) **character encoding**: a character-encoding standard for electronic communication
- Encodes **128 characters** into **7-bit integers** (see the ASCII chart on the next slide):
  - 95 printable characters: digits, letters, punctuation symbols
  - 33 non-printing (control) characters
- **Extended ASCII** (EASCII or high ASCII) character encodings, which comprises:
  - The standard 7-bit ASCII characters
  - Plus ***additional characters***
  - See: https://en.wikipedia.org/wiki/Extended_ASCII

ASCII Chart

ASCII printable code chart [edit]

Binary	Oct	Dec	Hex	Glyph
010 0000	040	32	20	(space)
010 0001	041	33	21	!
010 0010	042	34	22	"
010 0011	043	35	23	#
010 0100	044	36	24	$
010 0101	045	37	25	%
010 0110	046	38	26	&
010 0111	047	39	27	'
010 1000	050	40	28	(
010 1001	051	41	29	)
010 1010	052	42	2A	*
010 1011	053	43	2B	+
010 1100	054	44	2C	,
010 1101	055	45	2D	-
010 1110	056	46	2E	.
010 1111	057	47	2F	/
011 0000	060	48	30	0
011 0001	061	49	31	1
011 0010	062	50	32	2
011 0011	063	51	33	3
011 0100	064	52	34	4
011 0101	065	53	35	5
011 0110	066	54	36	6
011 0111	067	55	37	7
011 1000	070	56	38	8
011 1001	071	57	39	9
011 1010	072	58	3A	:
011 1011	073	59	3B	;
011 1100	074	60	3C	<
011 1101	075	61	3D	=
011 1110	076	62	3E	>
011 1111	077	63	3F	?

Binary	Oct	Dec	Hex	Glyph
100 0000	100	64	40	@
100 0001	101	65	41	A
100 0010	102	66	42	B
100 0011	103	67	43	C
100 0100	104	68	44	D
100 0101	105	69	45	E
100 0110	106	70	46	F
100 0111	107	71	47	G
100 1000	110	72	48	H
100 1001	111	73	49	I
100 1010	112	74	4A	J
100 1011	113	75	4B	K
100 1100	114	76	4C	L
100 1101	115	77	4D	M
100 1110	116	78	4E	N
100 1111	117	79	4F	O
101 0000	120	80	50	P
101 0001	121	81	51	Q
101 0010	122	82	52	R
101 0011	123	83	53	S
101 0100	124	84	54	T
101 0101	125	85	55	U
101 0110	126	86	56	V
101 0111	127	87	57	W
101 1000	130	88	58	X
101 1001	131	89	59	Y
101 1010	132	90	5A	Z
101 1011	133	91	5B	[
101 1100	134	92	5C	\
101 1101	135	93	5D	]
101 1110	136	94	5E	^
101 1111	137	95	5F	_

Binary	Oct	Dec	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
110 0111	147	103	67	g
110 1000	150	104	68	h
110 1001	151	105	69	i
110 1010	152	106	6A	j
110 1011	153	107	6B	k
110 1100	154	108	6C	l
110 1101	155	109	6D	m
110 1110	156	110	6E	n
110 1111	157	111	6F	o
111 0000	160	112	70	p
111 0001	161	113	71	q
111 0010	162	114	72	r
111 0011	163	115	73	s
111 0100	164	116	74	t
111 0101	165	117	75	u
111 0110	166	118	76	v
111 0111	167	119	77	w
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z
111 1011	173	123	7B	{
111 1100	174	124	7C	|
111 1101	175	125	7D	}
111 1110	176	126	7E	~

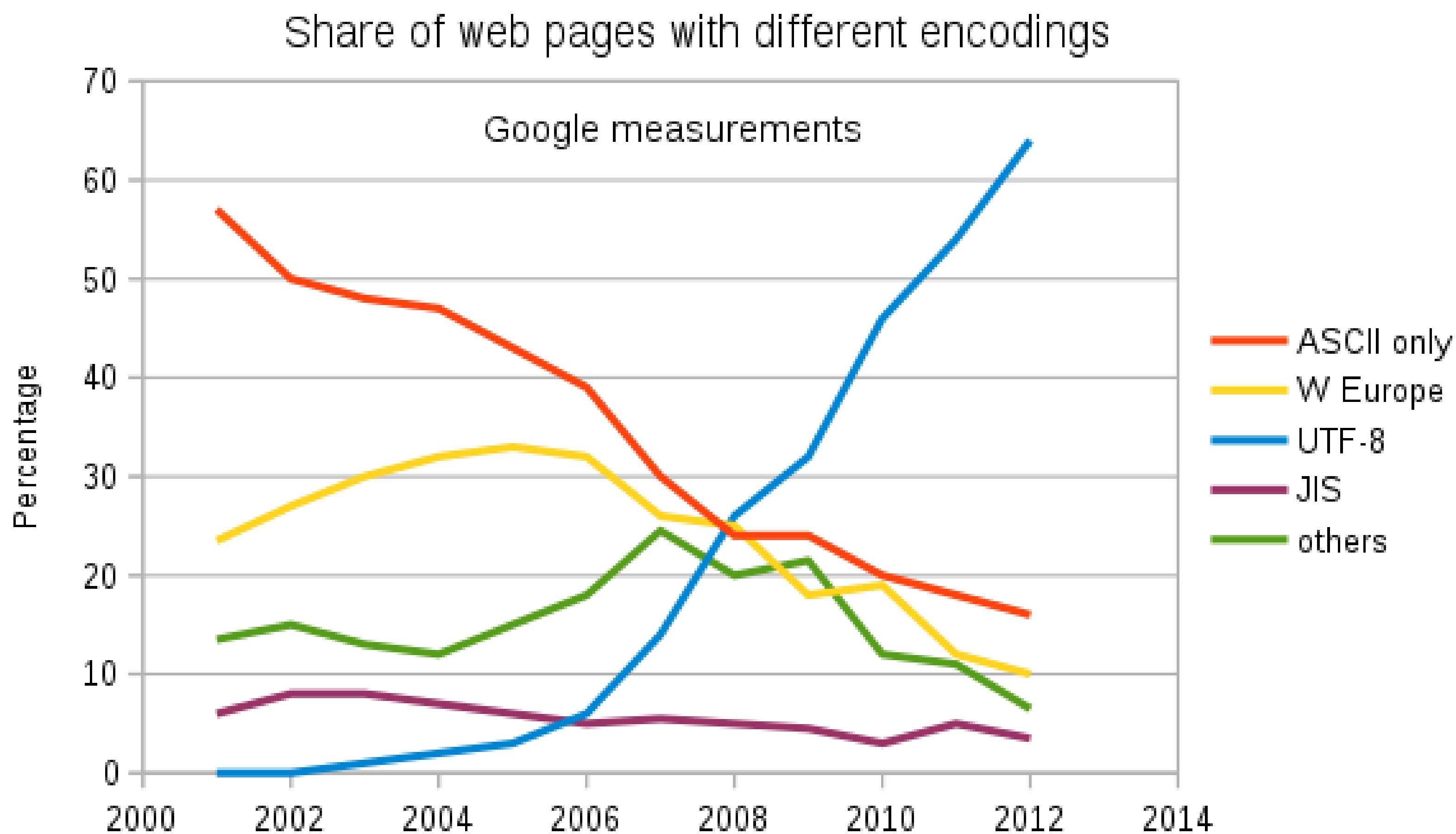
Background: UTF-8 (Unicode Transformation Format 8-bit)

- **UTF-8**: a character encoding capable of encoding all **1,112,064** valid code points in **Unicode** using **one to four** 8-bit bytes
- **A *variable-length* encoding**: code points that tend to occur **more frequently** are encoded with **lower** numerical values, thus fewer bytes are used
- The **first 128 characters** of Unicode:
  - Correspond 1-to-1 with **ASCII**
  - Encoded using a **single octet** with the same binary value as ASCII: Recall that there are 128 ASCII characters, and each starts with the bit 0 in a single byte
- Hence, ASCII characters **remain *unchanged*** in UTF-8
- **Backward compatibility** with ASCII: UTF-8 encoding was defined for "Unicode" *on* systems that were designed for ASCII
- See: https://en.wikipedia.org/wiki/UTF-8 for details

Background: UTF-8 Popularity

Optional

The **dominant** character encoding for the Web since 2009, as of October 2019 accounts for **94.1%** of all Web pages



(Source: Wikipedia)

Exploitable Vulnerability 2: UTF-8 “Variant” Encoding Issue

- A **Unicode** character: referred to by "**U+**" & its hexadecimal digits
- The following are **byte representations** of Unicode characters: the left-hand side is the **Unicode** representation, the right-hand side is the **byte** representation

```
U000000-U00007F:    0xxxxxxx
U000080-U0007FF:    110xxxxx 10xxxxxx
U000800-U00FFFF:    1110xxxx 10xxxxxx 10xxxxxx
U010800-U10FFFF:    11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

7 bits

11 bits

16 bits

21 bits

- Notice the **prefix bits** in the **first/leading byte** and **continuation byte(s)**
- The **xxx bits** are replaced by the **significant bits** of the code point of the respective Unicode character
- By the rules above, byte representation of a UTF-8 character is ***unique***
- *However*, many implementations also accepts **multiple** and **longer** “***variants***” of a character! *Why is that so?*

Different Representations of the Same UTF-8 Character

- Consider the ASCII character '**/**', whose ASCII code is:

  0010 1111 = 0x2F

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

  (2-byte version) `110`**`00000`** `10`**`101111`**

  (3-byte version) `1110`**`0000`** `10`**`000000`** `10`**`101111`**

  (4-byte version) `11110`**`000`** `10`**`000000`** `10`**`000000`** `10`**`101111`**

- That is, all the above would be decoded to '/'
- Now, there could be an inconsistency between:
  1. The **character verification** process; and
  2. The **character usage(s)**: operations using the character

Potential Problem with UTF-8: A Sample Scenario

- In a typical file system, **files** are organized inside a **directory**
- Example: the **full path name** of a **file name** "`index.html`" is: `/home/student/alice/public_html/index.html`
- Suppose a server-side program, upon receiving a string *<file-name>* from a client, carry 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 then **send** the file content to the client

Potential Problem with UTF-8: A Sample Scenario

- In the above example, the client can be any remote **public user** (similar to HTTP client)
- The original **intention**: the client can retrieve only files under the directory `public_html` → ***file-access containment***
- However, an attacker (the client) may send in this string:

  `../cs2107report.pdf`

  Which file would be read and sent by the server?
- This is the file:
  `/home/student/alice/public_html/../cs2107report.pdf`
- This access **violates** the intended file-access containment
- To prevent this, the server may add an "**input validation**" step, making sure that "`../`" never appear as a substring in the input string: *is this check complete?*

Added Input-Validation Step

Is this check "complete"?

Step 1: Append *<file-name>* to the prefix (directory) string:

`/home/student/alice/public_html/`

and take the concatenated string as string *F*

Step 1a: Checks that *<file-name>* does not contain the substring "`../`";
Otherwise, quit

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

Now, further suppose that the **system call in Step 2**:

1. Uses a convention that '%' followed by two hexadecimal digits indicates **a single byte** (like ***URL encoding***)
   E.g.: In "`/home/student/%61lice/`", `%61` is to be replaced by `a`
2. Uses **UTF-8**

The Security Problem

- Then, the check carried out by Step 1a is ***incomplete***: it misses some cases!
- Any of the following string will **pass** the check in Step 1a, since it literally does not contain the substring "`../`":

```
(1)          ..%2Fcs2107report.pdf
(2)          ..%C0%AFcs2107report.pdf
(3)          ..%E0%80%AFcs2107report.pdf
(4)          ..%F0%E0%80%AFcs2107report.pdf
```

- However, eventually, the filename will be **decoded** to: `/home/student/alice/public_html/../cs2107report.pdf`
- In general: a ***blacklisting-based*** **filtering** could be **incomplete** due to the "flexible use" of character encoding

Yet Another Example: IP Address

- Recall that the **4-byte IP address** is typically written as a string, e.g. "132.127.8.16"
- Consider a **blacklist** containing a lists of banned IP addresses, where each IP address is represented as 4 bytes
- A programmer wrote a **function `BL()`** :
  - Takes in 4 integers, where each integer is of the type "int" represented using **32 bits**
  - Checks whether the IP address represented by these 4 integers is in the black list
- In **C** language: `int BL(int a, int b, int c, int d)`
- `BL()` stores the blacklist as **4 arrays of integers** `A`, `B`, `C`, `D`: Given the 4 input parameters `a`, `b`, `c`, `d`, `BL()` simply searches for the existence of index *i* such that: `A[i]==a, B[i]==b, C[i]==c,` and `D[i]==d`

Potential Problem

Now, a program that performs the **following checks** is vulnerable:

(1) Get a string ***s*** from user

(2) Extract 4 integers (each integer is of type **`int`**, i.e. 32-bits) from the string ***s***, and let them be ***a***, ***b***, ***c***, ***d***:
If *s* does not follow the correct input format
(the correct format is 4 integers separated by "."), then quit

(3) Call `BL()` to check that that (***a***, ***b***, ***c***, ***d***) is not in the black list;
Otherwise, quit

(4) Let ***ip*** = $a*2^{24} + b*2^{16} + c*2^{8} + d$, where *ip* is a 32-bit **integer**

(5) Continue the rest of processing with the filtered address *ip*

Why is it vulnerable? Can you exploit it?

Security Guideline: Use Canonical Representation

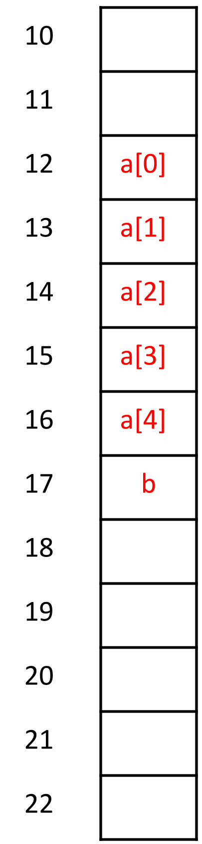
- Below are the important **lesson** and suggested **measures**
- ***Never*** trust the input from user
- Always convert them to a **standard** (i.e. ***canonical***) **representation** immediately
- Preferably, *do not* rely on the verification check done in the **application**;
  i.e. *do not* rely on the **application developers** to write the verification
- Rather, try to make use of the **underlying system access control** mechanism

8.2 Buffer Overflow

C/C++ and Memory Access

- C and C++ allows the programmers to **manage** the memory: pointer arithmetic, no *array-bound checking*
- Such **flexibility** is useful, but **prone to bugs**, which in turn leads to **vulnerability**
- Consider this simple program:

```
#include<stdio.h>
int a[5]; int b;
int main()
{
   b=0;
   printf("value of b is %d\n", b);
   a[5]=3;
   printf("value of b is %d\n", b);
}
```



Here, the value 3 is to be written to the cell a[5], which is also the location of the **variable *b***

Buffer Overflow/Overrun

- The previous example illustrates ***buffer overflow*** (a.k.a. **buffer overrun**)
- A ***data buffer*** (or just ***buffer***): "a contiguous region of memory used to temporarily store data, while it is being moved from one place to another"
- In general, a **buffer overflow** refers to a situation where data is written ***beyond*** a buffer's boundary
- In the previous example, the array *a* is a buffer of size **5**, and the location **a[5]** is beyond its boundary: hence, writing on it causes a "buffer overflow"
- A well-known function in C that is prone to buffer overflow is a string copying function: **`strcpy()`**

Strcpy() Function

- Consider this code segment:

```
{
  char s1[10];
    // ..  get some input from user and store it in a string s2
  strcpy(s1, s2);
}
```

- In the above, the length of `s2` can potentially be **more than 10**, since the length is determined by the first occurrence of null
- The `strcpy()` may copy the whole string of `s2` to `s1`, ***even*** *if* the length of `s2` is more than 10
- Since that the buffer size of `s1` is only 10, the extra values will be **overflowed** and written to **other part** of the memory
- If `s2` is ***supplied* by a malicious user**, a well-crafted input can overwrite important memory and modify the computation!

Secure Programming Defense/Practice

- Avoid using `strcpy()` !
- In secure programming practice,
  use **`strncpy()`** instead
- The function `stcncpy()` takes in **3** parameters:

  `strncpy (s1, s2, `**`n`**`)`

- At **most** `n` characters are copied
- Note that improper usage of `strncpy()` could still lead to vulnerability: *to be discussed in tutorial*

Stack Smashing (Stack Overflow)

- ***Stack smashing*:** a special case of buffer overflow that targets a process' **call stack**
- Recall that when a function is invoked, information like parameters, *return address* will be pushed into the stack
- If the stack is being overflowed such that the **return address** is modified, the execution's control flow will *be changed*
- A **well-designed overflow** could also "inject" the attacker's ***shellcode*** into the process' memory, and then execute the shellcode
- What will happen if the target executable is **setUID-root**?
- Some defenses/counter-measures are available, such as: ***canary***, which will be discussed in the next lecture

Stack Smashing (Stack Overflow): Example

- Consider the following vulnerable segment of C program:

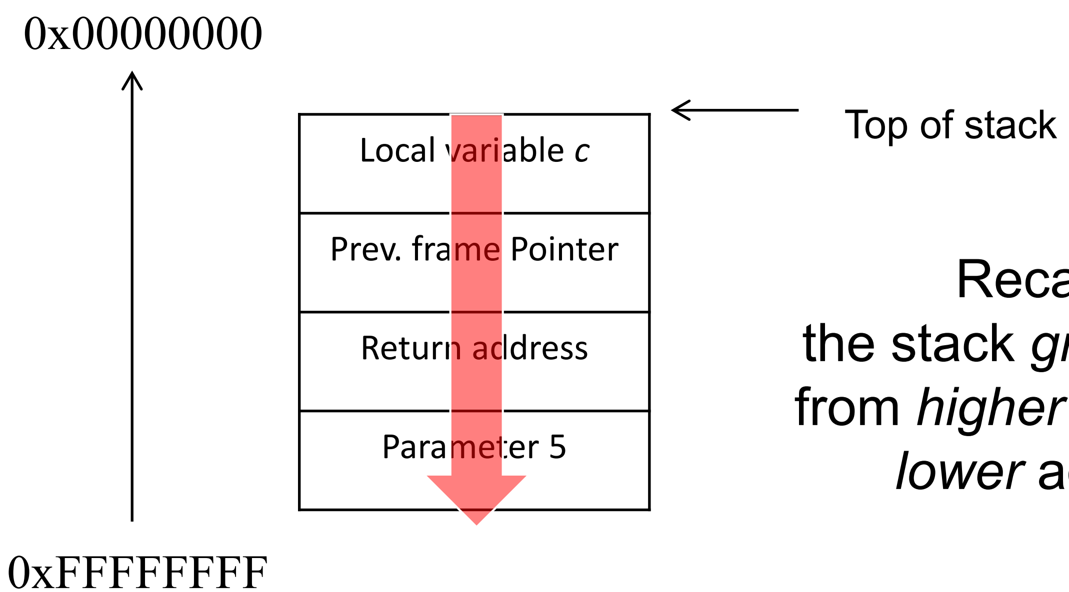
```
int foo(int a)
{
   char c[12];
   ......
   strcpy(c, bar); /* bar is a string input by user */
}

int main()
{
    foo(5);
}
```

Stack Smashing (Stack Overflow): Example

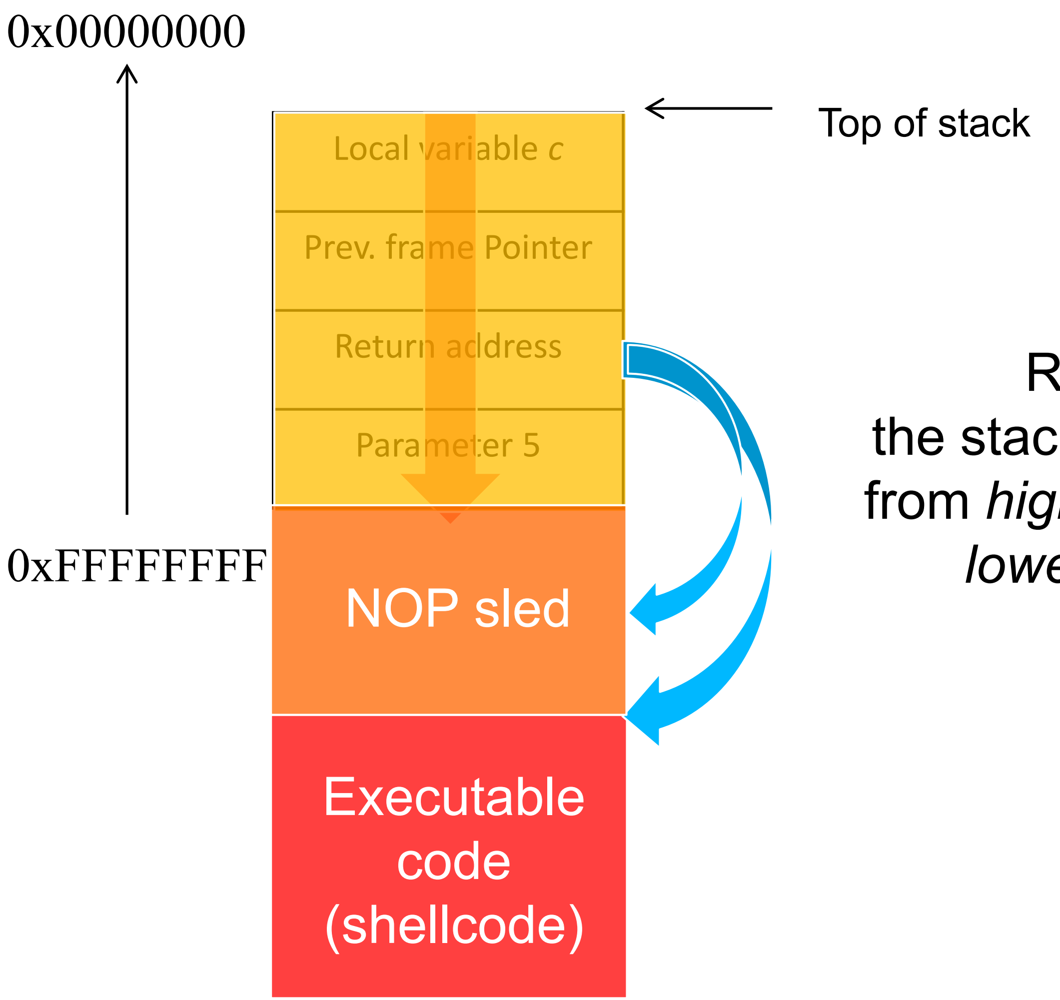
- After the `foo(5)` is invoked, a few values are pushed into the stack
- Important observation: the buffer `c` grows **toward return address**!
- If an attacker manages to modify the **return address**,
  the control flow ***will jump*** to the address indicated by the attacker

Read the *first* section: "**Exploiting stack buffer overflows**" of https://en.wikipedia.org/wiki/Stack_buffer_overflow**,** *other sections* 2-4 are optional**)**



Recall that:
the stack *grows upward*,
from *higher* addresses to
lower addresses

Stack Smashing (Stack Overflow): Shellcode Illustration



Recall that:
the stack *grows upward*, from *higher* addresses to *lower* addresses

8.3 Integer Overflow

(Note: This is *not* to be confused with “buffer overflow”)

Integer Arithmetic and Overflow

- The **integer arithmetic** in many programming language are actually "***modulo arithmetic***"
- Suppose `a` is a single byte (i.e**. 8-bit**) **unsigned integer**.
  In the following C or Java statements,
  what would be the final value of `a`?

```
a = 254;
a = a+2;
```

- Its value is **0**, since the addition is done w.r.t./in **modulo 256**
- Hence, the following predicate is ***not*** necessarily always true!

```
(a < a+1)
```

- Yet, many programmers ***do not*** realize this,
  leading to possible vulnerability (see Tutorial 8)

8.4 Code/Script Injection

Scripting Language and Security

- A key concept in computer architecture is the **treatment of "code" (i.e. program) as "data"**
- In security, mixing "code" and "data" is potentially **unsafe**: many attacks inject malicious code as data, which then gets executed by the target system!
- We will consider a well-known ***SQL injection (SQLI) attack***
- ***"Scripting" languages*****:** programming languages that can be "***interpreted***" by another program during runtime, instead of being compiled
- Well-known examples: JavaScript, Perl, PHP, **SQL**
- Many scripting languages allow the "script" to be **modified while being interpreted**: this opens up the possibility of injecting malicious code into the script!

SQL and Query

- **SQL** is a *database query language*
- Consider a database (which can be viewed as a **table**): each ***column/field*** is associated with an *attribute*, e.g. “name”

name	account	weight
bob12367	12333	56
alice153315	4314	75
eve3141451	111	45
petter341614	312341	86

- This query script

```
SELECT * FROM client WHERE name = ‘bob’
```

searches and returns the **rows** where the name matches ‘bob’

- The scripting language also allows **variable**: e.g. a script may first get the user’s input and stores it in the variable `$userinput`, and subsequently runs:

```
SELECT * FROM client WHERE name = ‘$userinput’
```

SQL Injection: Example

- In this example, the database is designed such that the **user name is a secret**: hence, only the authentic entity who knows the name can get the record
- Now, an attacker can pass the following as the input:

  ```
  Bob' OR 1=1 --
  ```

  That is, the variable `$userinput` becomes

  ```
  Bob' OR 1=1 --
  ```

- The interpreter, after seeing this script

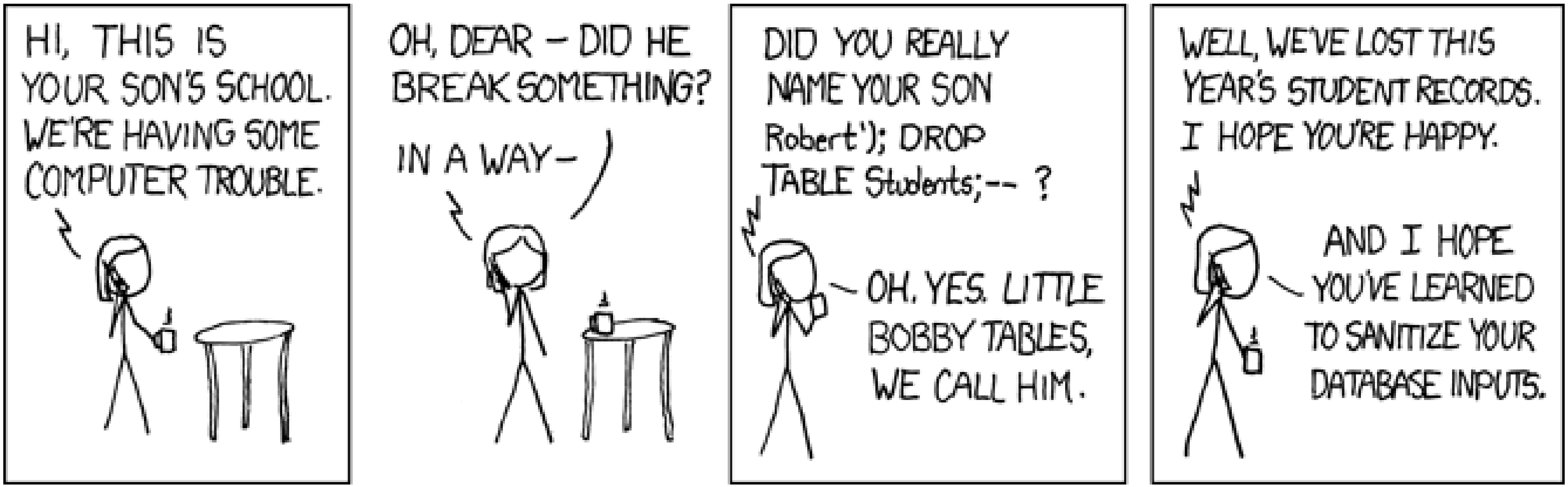
  ```
  SELECT * FROM client WHERE name = '$userinput'
  ```

  simply substitutes the above to get and execute:

  ```
  SELECT * FROM client WHERE name = 'Bob' OR 1=1 --'
  ```

- Note: "`--`" is interpreted as **the start of a comment**
- The interpreter runs the above and return ***all*** the records!

SQL Injection a.k.a. “Bobby Tables”



Source:
https://xkcd.com/327/

Code Injection & Buffer Overflow

- Code injection does *not* limit to SQL injection
- It is possible to **exploit buffer overflow** by:
  injecting malicious code, and then
  transferring the process execution to the malicious code
- Details are omitted for this module
- For more details, see:

  http://www.cis.syr.edu/~wedu/Teaching/IntrCompSec/LectureNotes_New/Buffer_Overflow.pdf

See fun and non-security related easter eggs:
www.pcadvisor.co.uk/feature/social-networks/11-best-easter-eggs-on-web-in-apps-3530683/

8.5 Undocumented Access Points (Easter Eggs)

Undocumented Access Points

- For debugging purposes, many programmers insert "***undocumented access point***" to inspect the states
- Examples:
  - By pressing **certain combination of keys**, the values of certain variables would be displayed
  - For certain **input strings**, the program would branch to some debugging mode
- These access points may mistakenly **remain** in the final production system, providing "***backdoors***" to the attackers
- A ***backdoor***: a covert method of bypassing normal authentication
- Such access points are also known as ***Easter eggs***

Undocumented Access Points

- Some Easter eggs are **benign** and intentionally planted by the developer for **fun** or **publicity**
- But, there are also known cases where unhappy/disgruntled programmer purposely **planted** the backdoors
- The backdoors can be accessed by the programmer, and also by **any** other users who knows/discovers them!
- ***Terminology***: Logic bombs, Easter eggs, backdoors