Reading: [PF] 3.1, 3.2, 3.3

Lecture 8: Software Security

8.1 Overview of software security

8.2 Computer architecture background

- 8.2.1 Code vs data, program counter
- 8.2.2 Stack (aka execution stack, call stack)
- 8.2.3 Control flow integrity

8.3 Attacks on software

- 8.3.1 Integer overflow
- 8.3.2 Data/string representation & security
- 8.3.3 Buffer overflow
- 8.3.4 Code/script injection
- 8.3.5 Undocumented access points

8.4 Defense and preventive measures

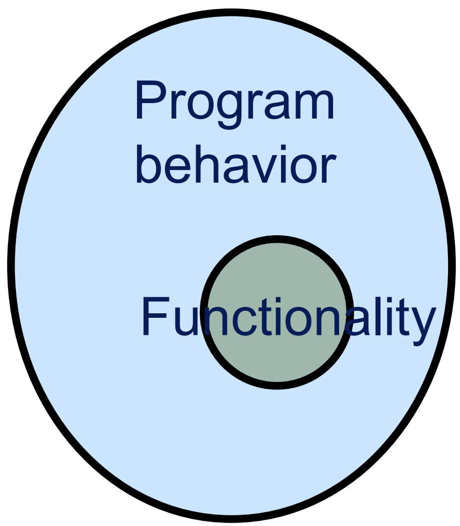
8.1 Overview of Software Security

Program: Requirements and Possible Behavior

Requirements of a program:

- A program has to be **correct**
- A program has to be **efficient**
- A program also has to be ***secure***

Targeted/intended program functionality vs possible behavior:



A program *may* behave **beyond** its intended functionality!

Possible Security Problems

- **Insecure implementation**:
  Many programs are ***not*** implemented properly,
  allowing attacker (i.e. the person who invokes the process)
  to **deviate from the programmer's intents**

- **Unanticipated input**:
  The attacker may supply **input** in a form that is ***not*** anticipated by the programmer, which can unintendedly cause the process to:
  - Access sensitive resources;
  - Execute some "injected" codes; or
  - Deviate from the original intended execution path

  → Either way, the attacker manages to **elevate its privilege**

- In this lecture, we will look at several **classes of insecure programs** and also **the reasons** behind their insecurity!

Some Sample Cases

Buffer Overflow:

- **Morris worm** (1988): exploited a Unix **finger** service to propagate itself over the Internet
- **Code Red worm** (2001): exploited **Microsoft's IIS 5.0**
- **SQL Slammer worm** (2003): compromised machines running **Microsoft SQL Server 2000**
- Various attacks on **game consoles** so that unlicensed software can run without the need for hardware modifications: **Xbox**, **PlayStation 2** (PS2 Independence Exploit), **Wii** (Twilight hack)
- …

Some Sample Cases

SQL Injection:

- **Yahoo!** (2012): a hacker group was reported to have stolen **450,000** login credentials from Yahoo! by using a "*union-based* SQL injection technique“
- **British telco company TalkTalk** (2015): an attack exploiting a vulnerability in a legacy web portal was used to steal the personal details of **156,959** customers

Integer Overflow:

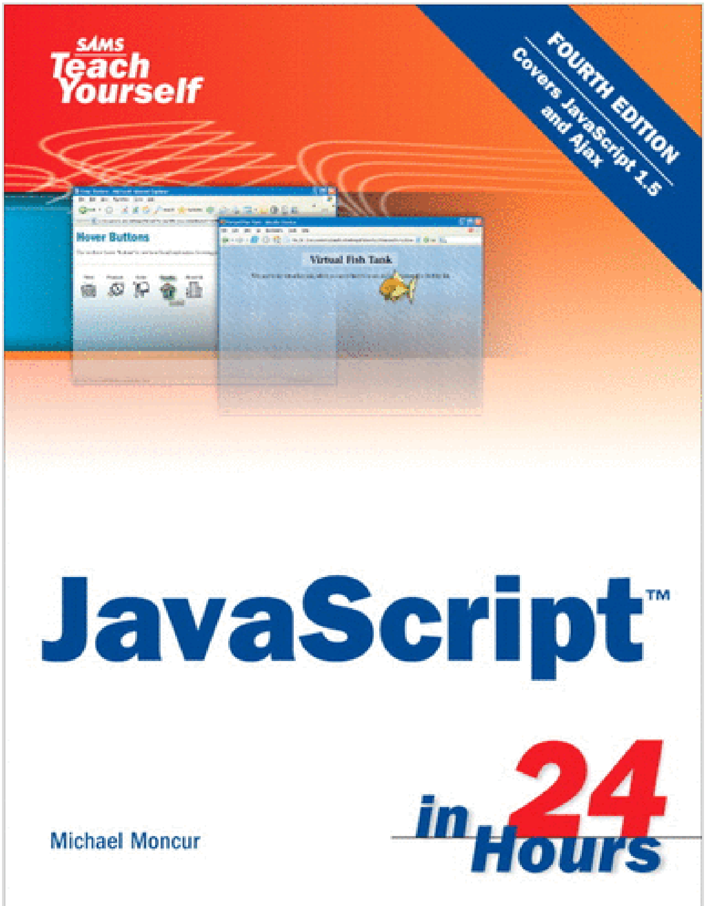
- **European Space Agency’s Ariane 5 rocket** (1996): an unhandled **arithmetic overflow** in the engine steering software caused its crash, costing $7 billion
- **Resorts World Casino** (2016): a casino machine printed a prize ticket of $42,949,672.76

Root Causes of Security Problems

Why is there a **big security** challenge?

- **Functionality**: still the **primary concern** during design and implementation
  - Security is a **secondary goal**
  - Features pay the bills (typically)
- Unavoidable **human mistakes**:
  - (Lack of) awareness of security problems
  - Careless programmers
- **Complex** modern computing systems:
  - Many of the “bugs” are very **simple** and seem **easy to prevent**, but programs for complex systems are **large**,
    e.g. Window XP has 45 millions SLOC (source line of codes)
    http://en.wikipedia.org/wiki/Source_lines_of_code.
  - Large **attack surface** as well

Programming can be Easy, but Good Programming Isn't So



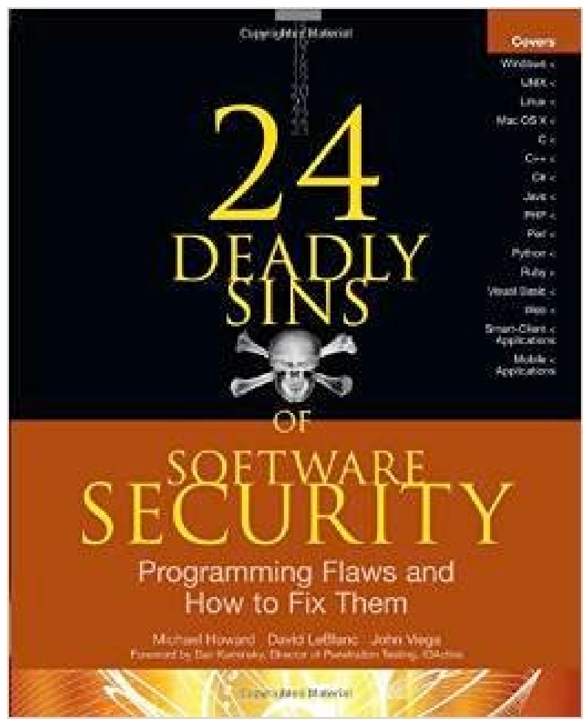
- Maybe enough for learning basic functionality
- **Never enough** for learning *subtle implications* of functionalities
- Result: programs can **do more** than you expect!

Recommended References for Secure Programming

Optional

Some well-known references:

- Michael Howard and David LeBlanc, *Writing Secure Code*, 2^{nd} ed, Microsoft Press, 2002
- Michael Howard, David LeBlanc, and John Viega, *24 Dealy Sins of Software Security: Programming Flaws and How to Fix Them,* McGraw-Hill, 2010



8.2 Computer Architecture Background

8.2.1 Code vs data, program counter

8.2.2 Stack (a.k.a. execution stack, call stack)

8.2.3 Control flow integrity

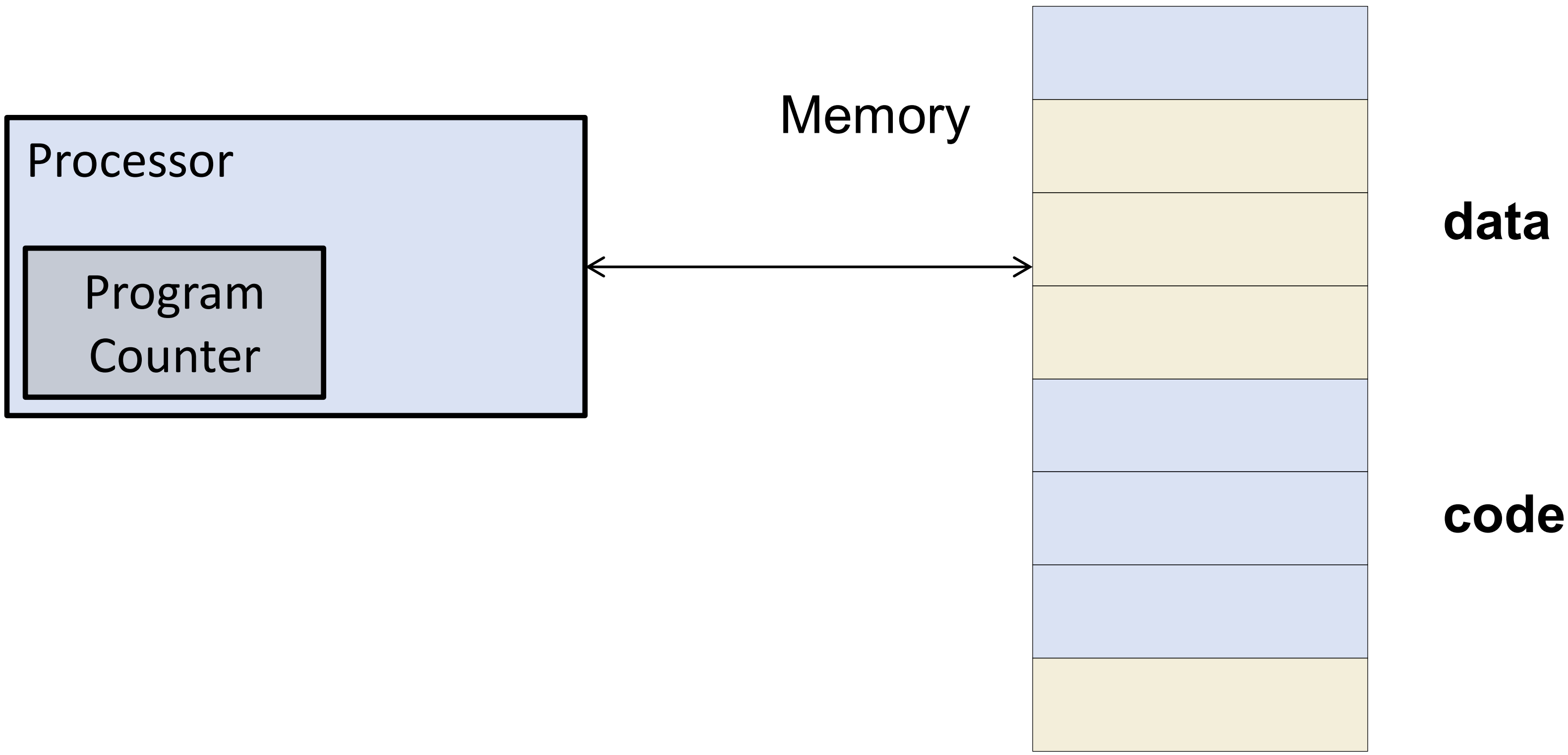
8.2.1 Code vs Data, Program Counter

Code vs Data in Modern Computers

Modern computers are based on the
***Von Neumann* computer architecture**:

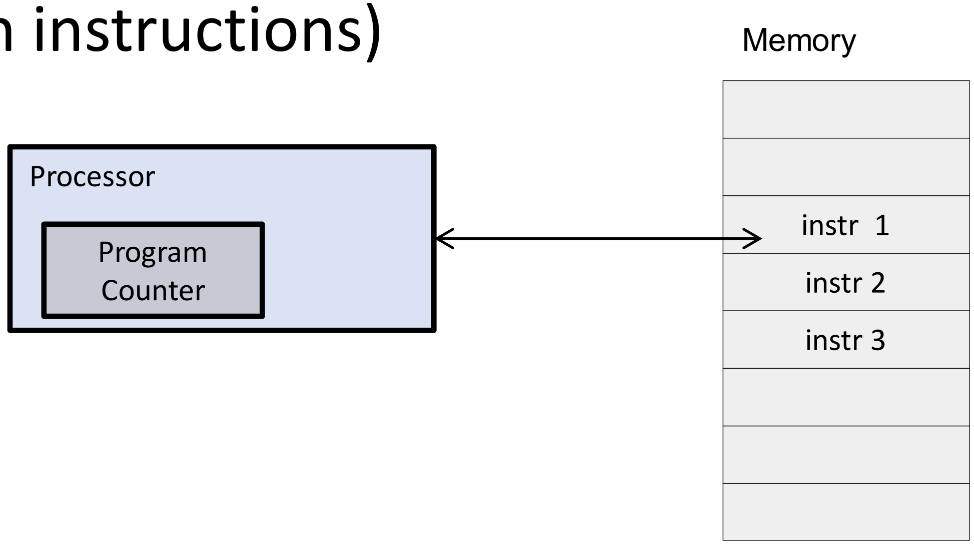
- The code and data are stored **together** in the memory
- There is **no clear distinction** of code and data
- This is in contrast to the ***Harvard architecture,***
  which has hardware components that separately store code and data
- **Serious implication**:
  programs may be tricked into **treating input data as code**: basis for all ***code-injection attacks***!

Code vs Data in Modern Computers



Control Flow

- The ***program counter*** (aka ***Instruction Pointer***):
  a register (i.e. small & fast memory within the processor)
  that stores the address of the next instruction
- After an instruction is completed, the processor **fetches** the next instruction from the address stored in the program counter
- After the next instruction is fetched, the program counter automatically **increases** by 1 (assuming a system with fixed-length instructions)

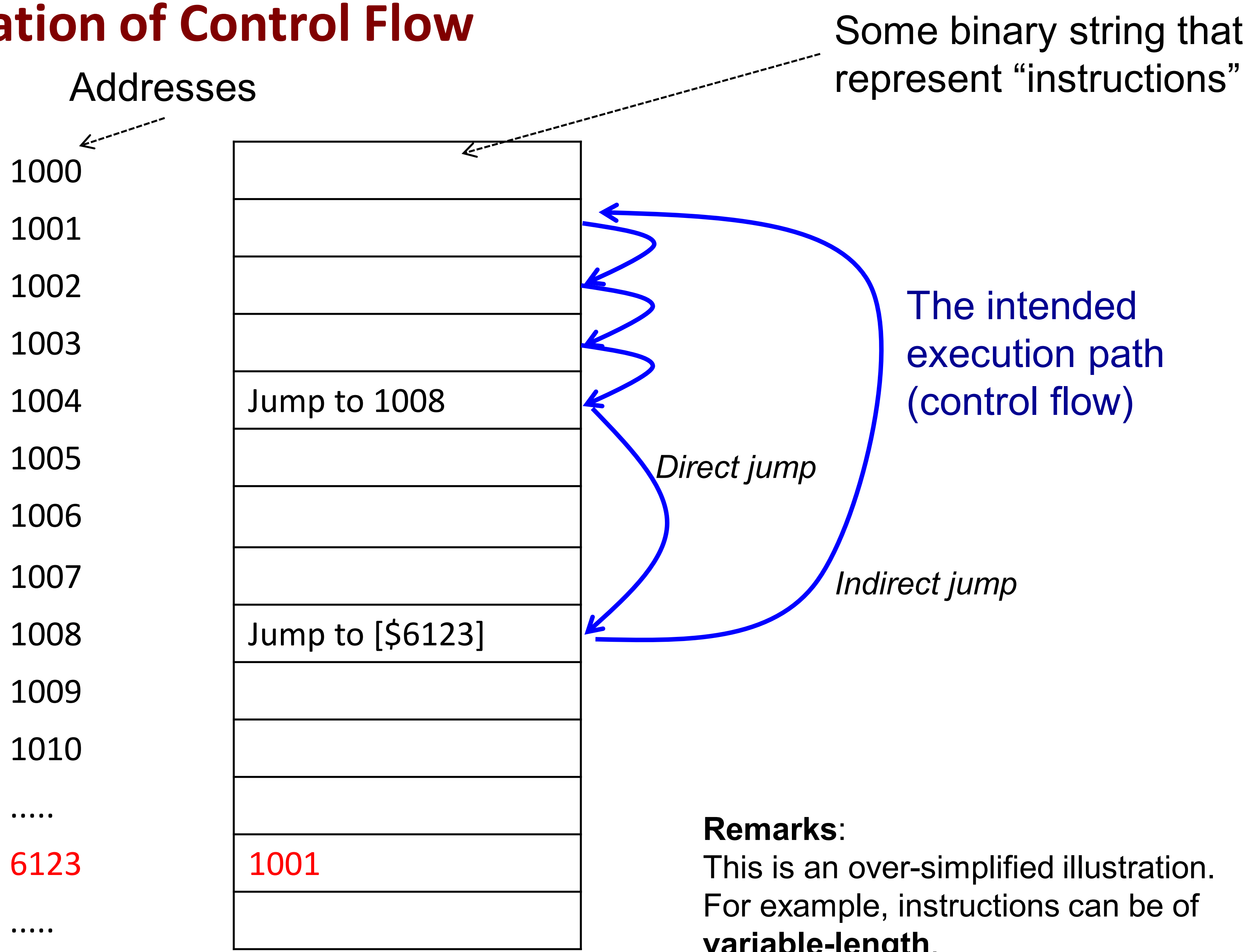


Control Flow

- During execution, besides getting **incremented**, the *program counter* (PC) can also be **changed**, for examples*, by:
  1. **Direct jump**:
     The PC is replaced with a **constant value** specified in the instruction
  2. **Indirect jump**:
     The PC is replaced with a **value** fetched from **the memory** or stored in a general-purpose **register**
     (Note that there are many different forms of indirect jump)

*: For simplicity in this module, we **omit** *conditional branch* as well as *call/return* here

Illustration of Control Flow



Remarks:
This is an over-simplified illustration.
For example, instructions can be of **variable-length**.

8.2.2 Stack
(a.k.a. Execution Stack, Call Stack)

See: https://en.wikipedia.org/wiki/Call_stack

Functions and Their Executions

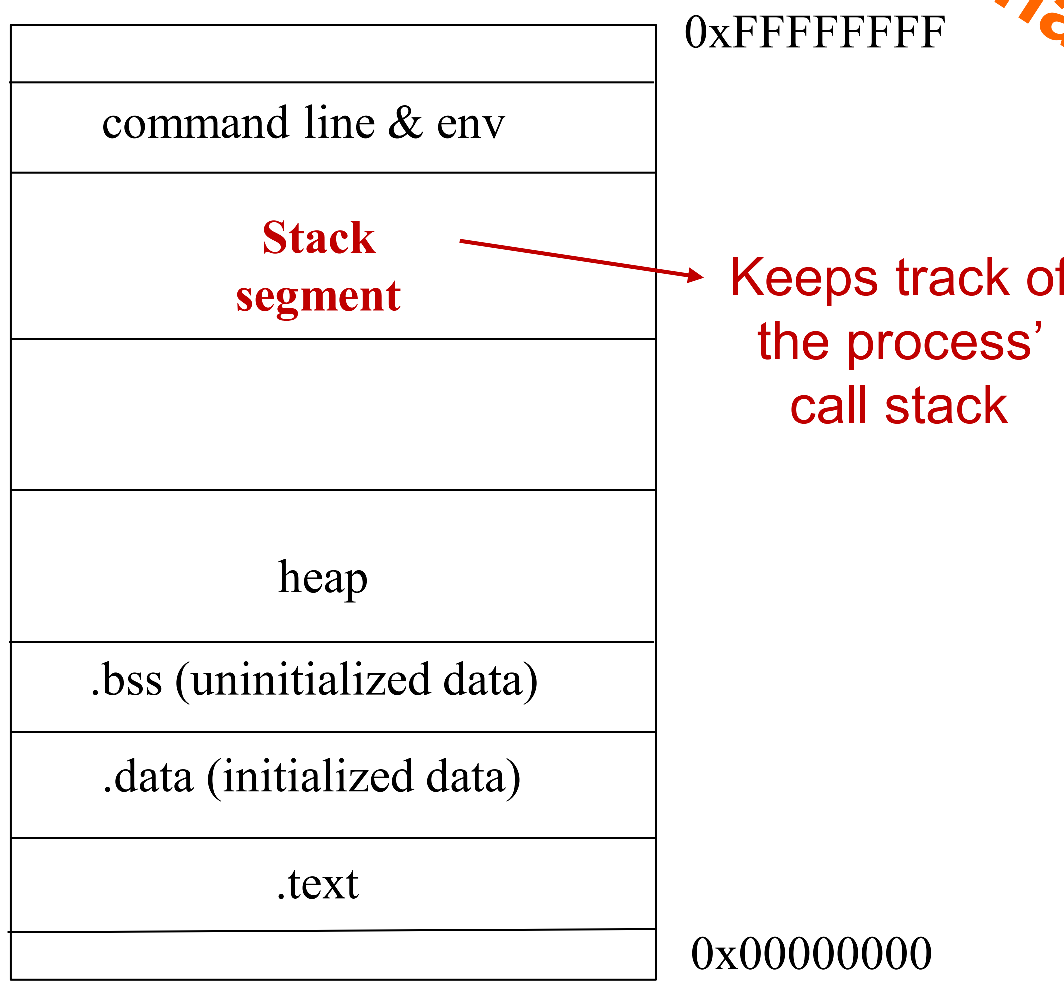
- **Functions** break code into smaller pieces:
  - Facilitate modular design and code reuse
- A function can be called in **many** program locations, e.g. 2, 10, 100, … times (e.g. recursive function)
- ***Question 1***: How does the program know where it should continue after it finishes?
- ***Question 2***: Where are the function's *arguments* and *local variables* stored?

```
void sample_function(void)
    {
        char buffer[10];
        printf("Hello!\n");
        return;
    }
main()
    {
        sample_function();
        printf("Loc 1\n");
        sample_function();
        printf("Loc 2\n");
    }
```

Process Memory Layout

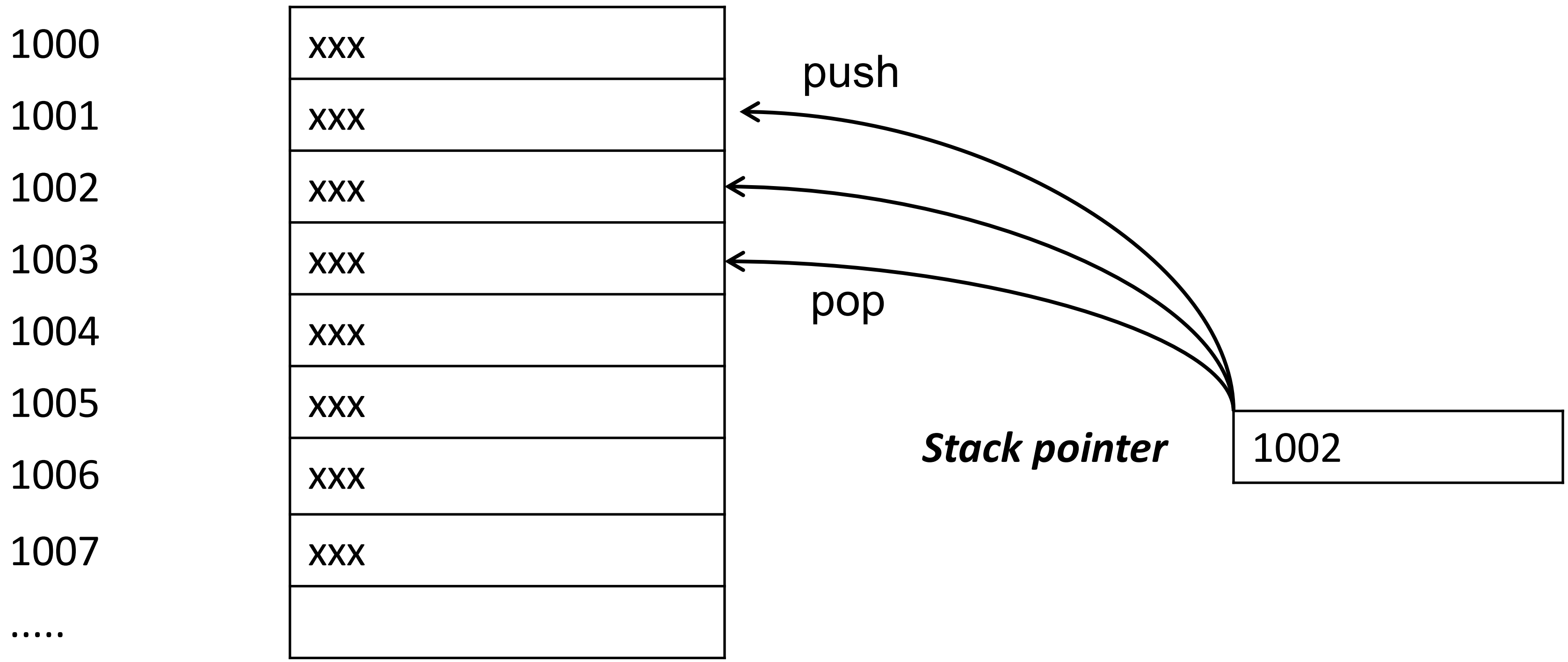
Optional

- Simplified **Linux process memory** showing various *segments*:
- (Optional: http://dirac.org/linux/gdb/02a-Memory_Layout_And_The_Stack.php)



Call Stack

- **Call stack**: a ***data structure*** **in the memory** (*not* in a separate hardware) that stores important information of a running process
- **Last in, first out (LIFO)**: with push(), pop(), top() operations
- The location of the top element is referred to by the ***stack pointer***



In this example, the stack grows "***upward***", ***but*** from **high addresses** to **low addresses**. It is possible to have stack that grows *downward*.

Call Stack

- During a program execution, a stack is used to keep tracks of:
  - **Control flow information**: i.e. return addresses
  - **Parameters** passed to functions
  - **Local variables** of functions
- Each call of a function pushes an ***activation record*** (***stack frame***) to the stack, which contains:
  passed parameters, return address, pointer to the previous stack frame, and function local variables
- **Note**: this stack is known as ***call stack.***
  In the context of system security, very often it is simply called the "***stack***".
  Sample expression: "smashing the *stack* for fun and profit".

Illustration: A Function Call

When a function is called, the parameters, return address, and local variables are “**pushed**" into the stack

Consider the following C program segment:

```
int test(int a) {
    int b = 1;
      ...
}
int main() {
      test(5);
      ...
}
```

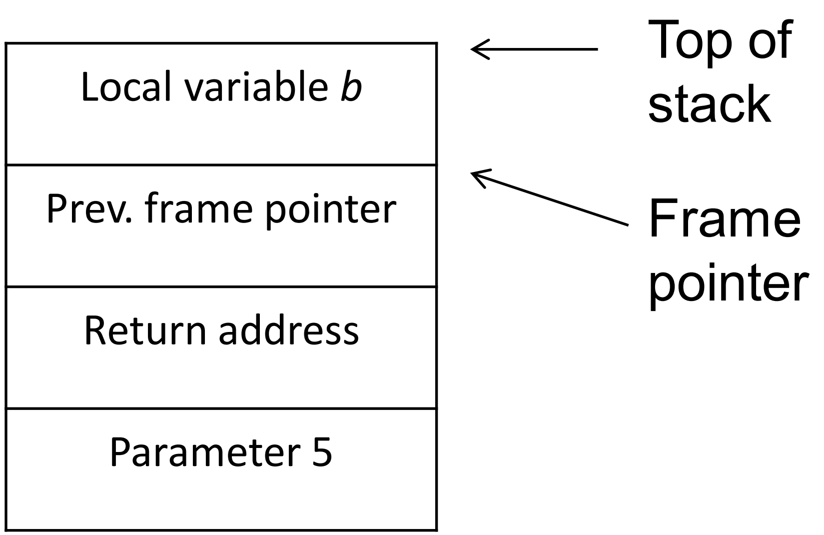


Illustration: A Function Call

When the function `test(5)` is invoked, the following are carried out:

(1) Some values are pushed into the stack:
the parameter (i.e. 5), the return address, the previous frame pointer, and the value of the local variable *b* (i.e. 1)

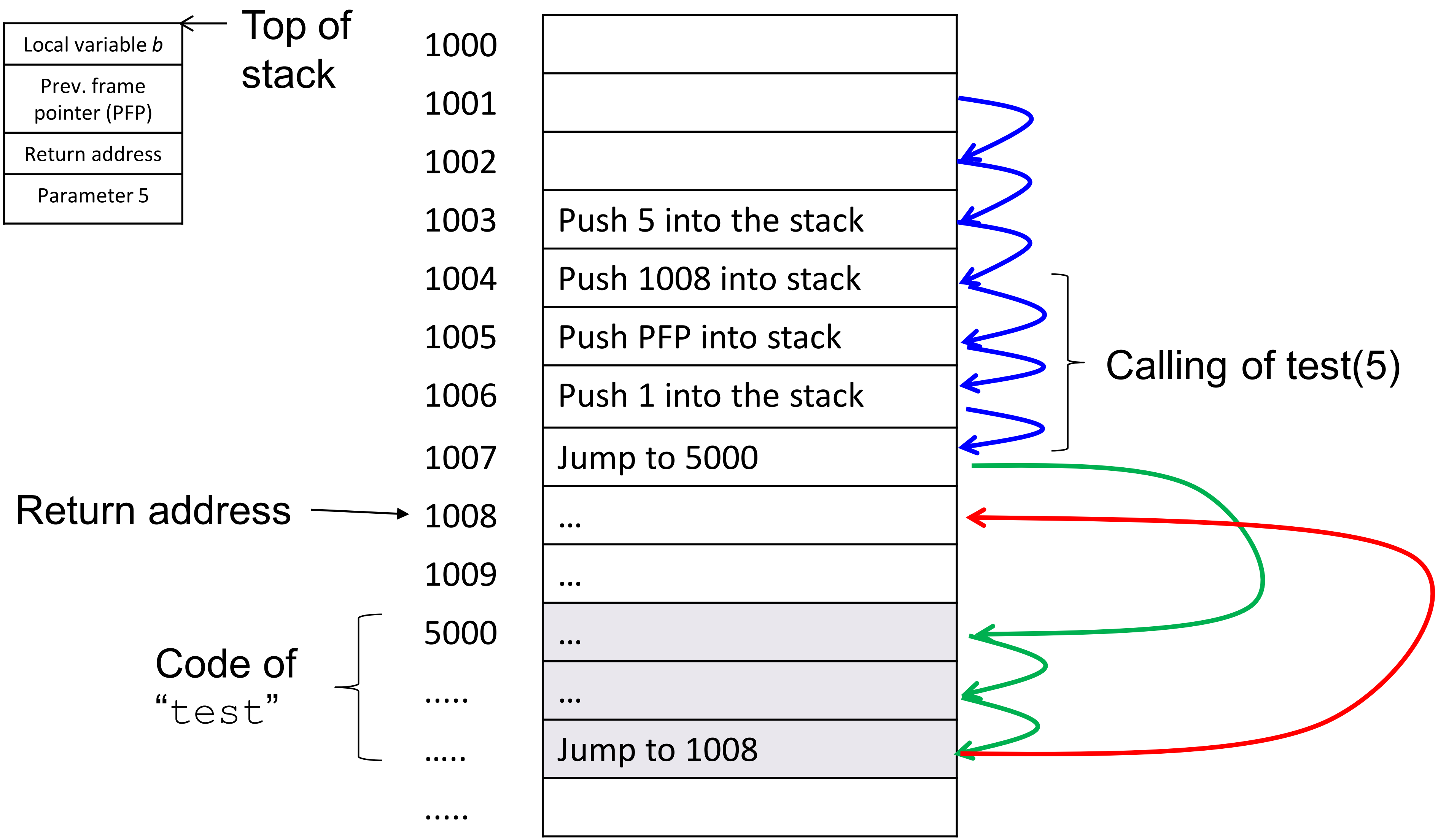
(2) The control flow jumps into the code of "`test`"

(3) Execute "`test`"

(4) After "`test`" is completed, the stack frame is **popped from the stack**

(5) The control flow jumps into the restored return address

(Simplified) Illustration*



*: This slide gives a **simplified** view. Actual implementation also includes "function return value".
For more details, see: http://www.tenouk.com/Bufferoverflowc/Bufferoverflow2a.html or https://en.wikipedia.org/wiki/Stack_buffer_overflow.

8.2.3 Control Flow Integrity

Treating Code as Data: Security Implications

- You have seen how the call stack stores a ***return address*** *(i.e. location of a to-be-executed instruction*) as **data** in the memory
- In fact, the **instruction** itself is stored as data in the memory
- The flexibility of treating code as data is useful, but it leads to **many security issues**
- Attacker could compromise a **process'** ***execution integrity*** by either modifying the process' **code** or the **control flow**
- It is difficult for the system to distinguish those malicious pieces of code from benign data

Notes on Compromising Memory Integrity

- In general, it is ***not* so easy** for an attacker to compromise memory integrity
- For **example**, consider an attacker who can only remotely communicate with the targeted Web server via HTTP. How *can* he maliciously write to the web-server's memory?
- One way for the attacker to gain that capability is by: **exploiting some vulnerabilities** so as to "trick" the victim process to **write to** some of its memory locations, e.g. via a "**buffer overflow**" attack
- The above mechanisms typically have **some restrictions**: for example, the attacker can only write to a small number of memory, or can only write a sequence of consecutive bytes. Hence, the attack has to be extremely "**surgical**".

8.3 Attacks on Software

8.3.1 Integer overflow
8.3.2 Data/string representation & security
8.3.3 Buffer overflow
8.3.4 Code/script injection
8.3.5 Undocumented access points

8.3.1 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 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)

8.3.2 Data/String 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

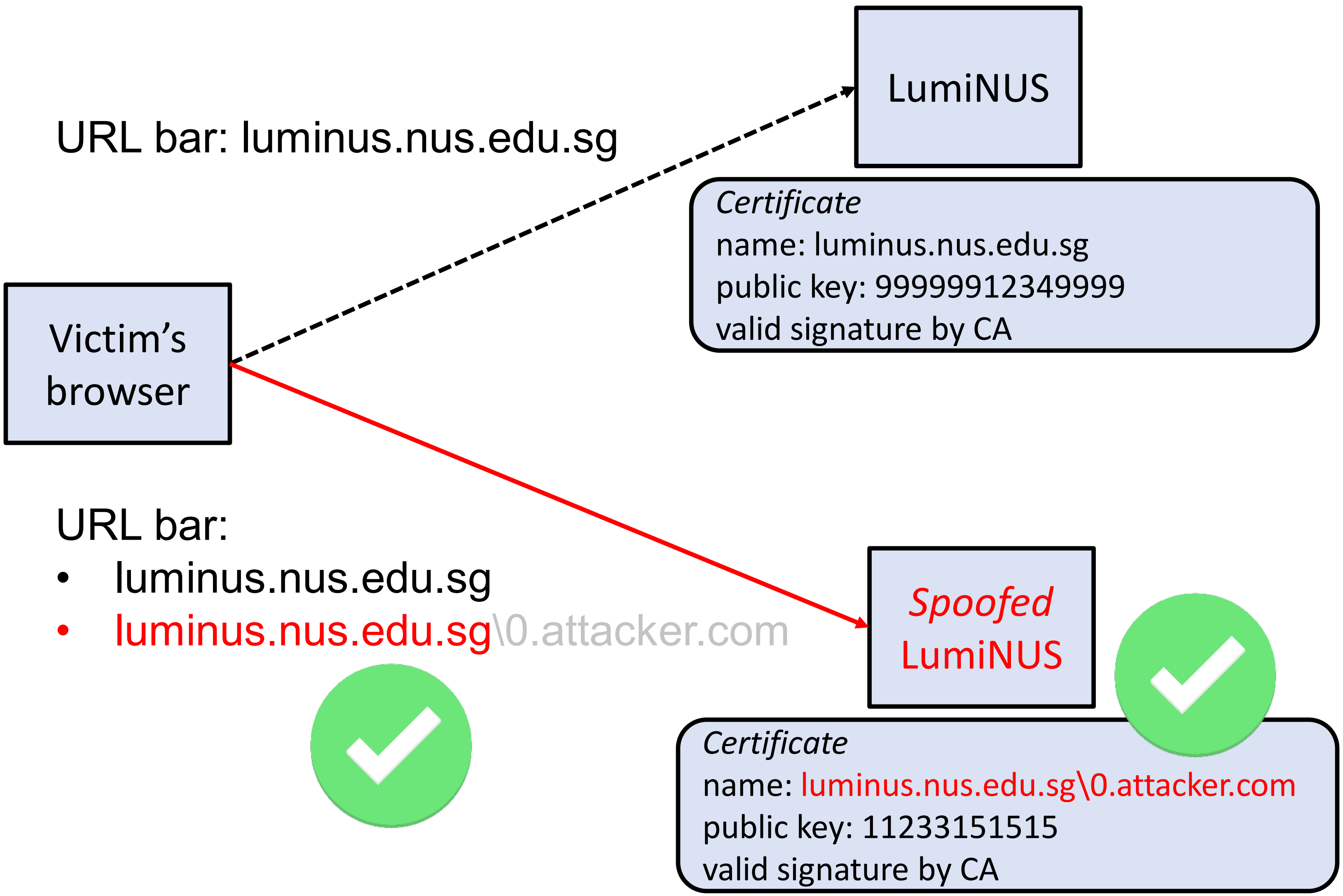
2. The attacker set up a **spoofed LumiNUS** website on another web server

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

.

A Sample Attack (on LumiNUS): Illustration



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

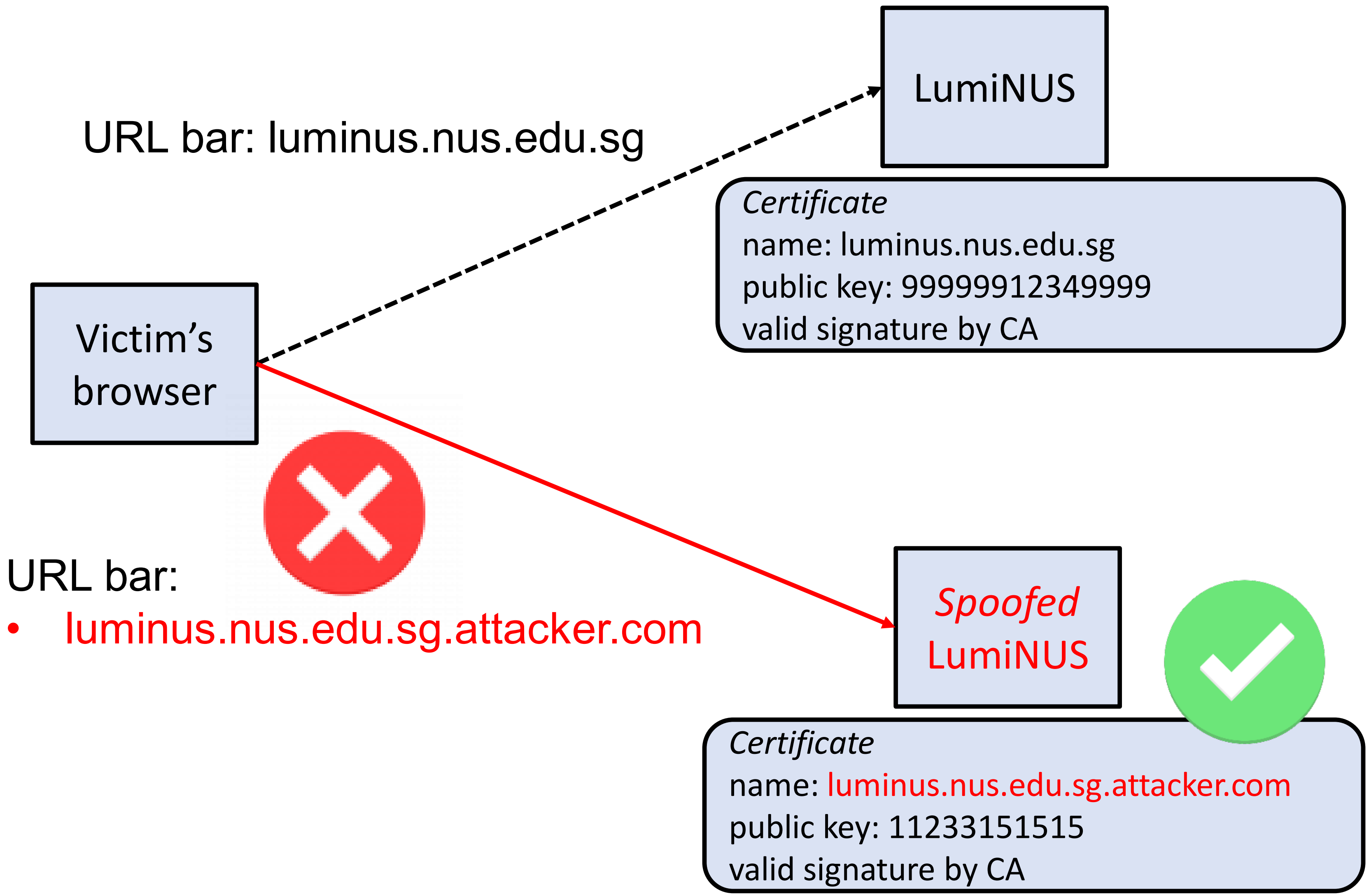
What if it is just a **normal web-spoofing** attack scenario?

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

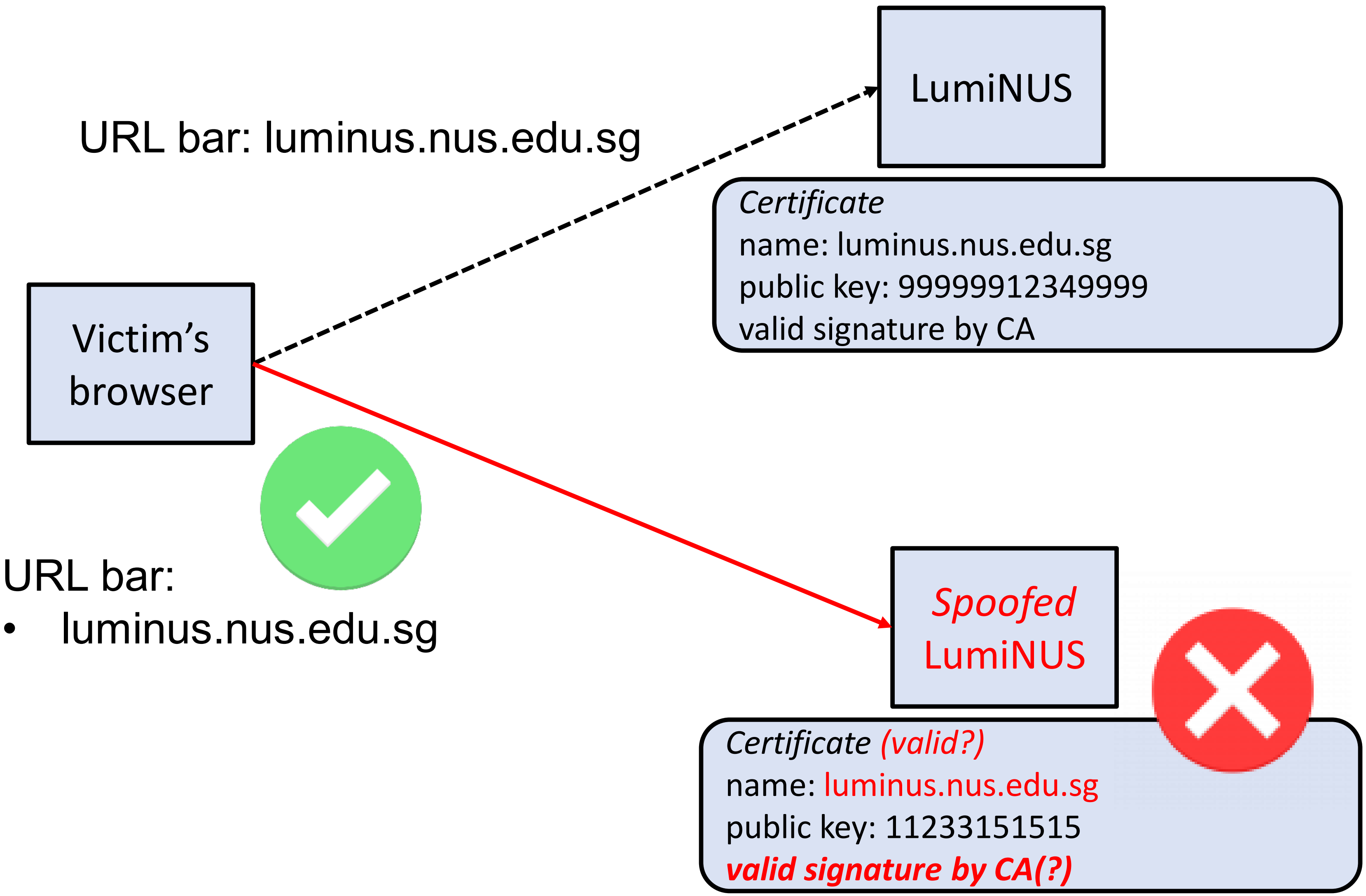
- The address displayed in the browser's address bar is ***not*** LumiNUS; or
- The address bar displays luminus.nus.edu.sg, but the TLS/SSL authentication protocol **rejects** the connection (i.e. "certificate is ***not*** trusted")

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

A Normal Web-Spoofing Attack (on LumiNUS): Case 1



A Normal Web-Spoofing Attack (on LumiNUS): Case 2



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	~

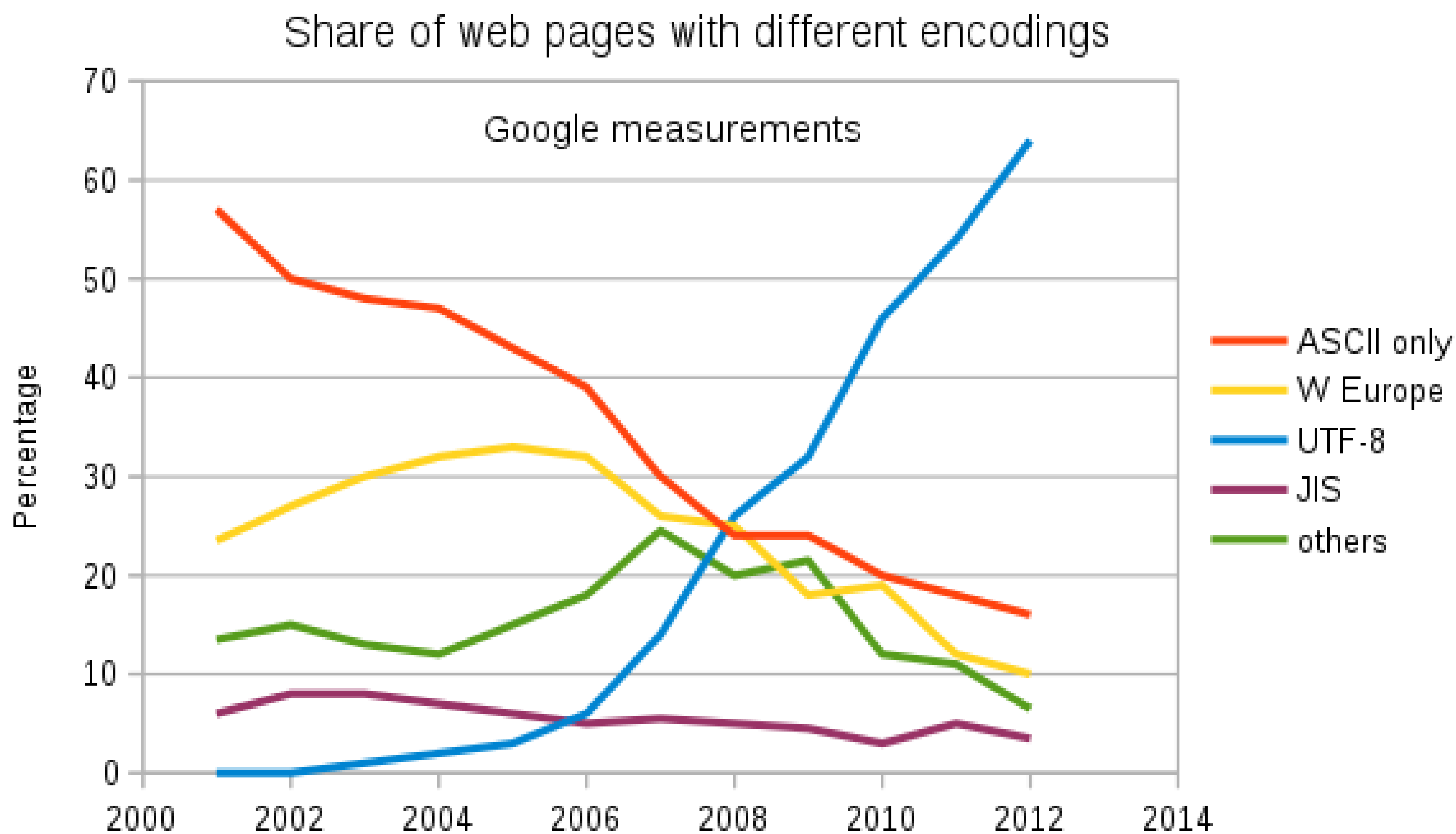
Background: UTF-8 (Unicode Transformation Format 8-bit)

- **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                              7 bits
U000080-U0007FF:    110xxxxx 10xxxxxx                     11 bits
U000800-U00FFFF:    1110xxxx 10xxxxxx 10xxxxxx            16 bits
U010800-U10FFFF:    11110xxx 10xxxxxx 10xxxxxx 10xxxxxx   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**:

- 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`
- 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

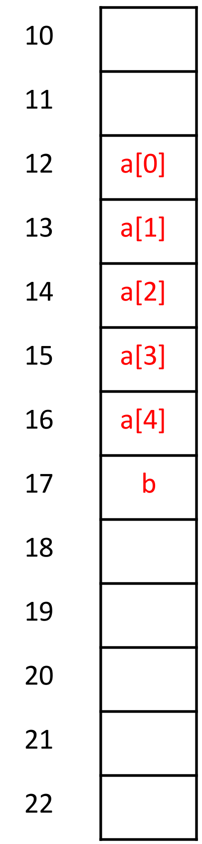
- Below are the important **lesson** and suggested **measures**
- ***Never*** trust the input from user
- Always convert them to a **standard** (i.e. ***canonical***) **representation** immediately
- Preferably, *do not* rely on the verification check done in the **application**;
  i.e. *do not* rely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

8.3.3 Buffer Overflow

C/C++ and Memory Access

- C and C++ allows the programmers to **manage** the memory: pointer arithmetic, no *array-bounds 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 later

Stack Smashing (Stack Overflow): Example

- Consider the following vulnerable segment of C program:

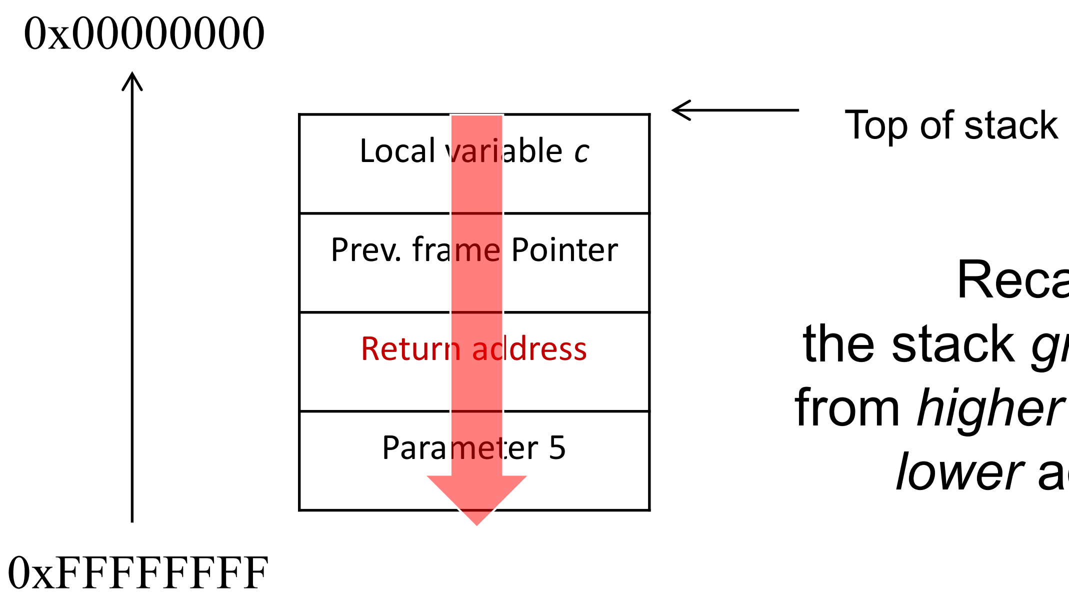
```
int foo(int a)
{
   char c[12];
   ......
   strcpy(c, bar); /* bar is a string input by user */
}

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

Stack Smashing (Stack Overflow): Example

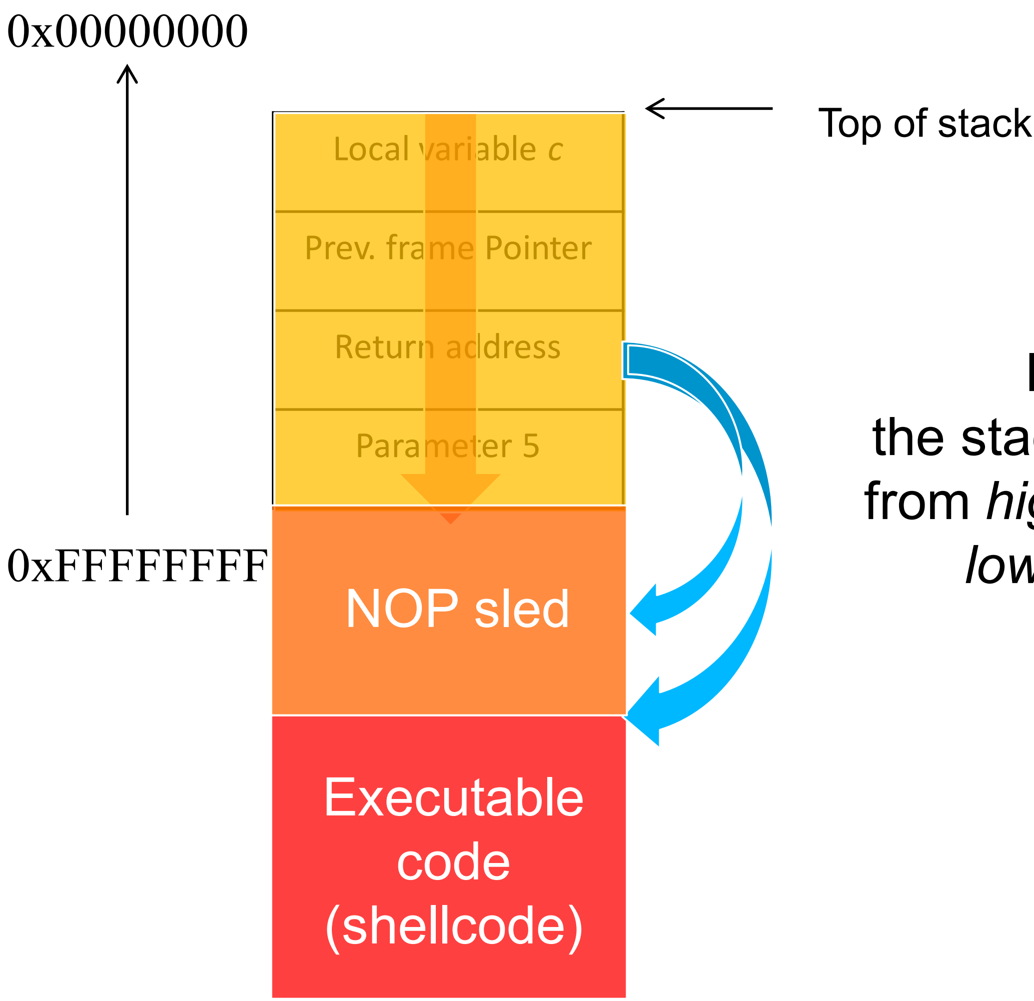
- After the `foo(5)` is invoked, a few values are pushed into the stack
- Important observation: the buffer `c` grows **toward return address**!
- If an attacker manages to modify the **return address**,
  the control flow **will jump** to the address indicated by the attacker

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

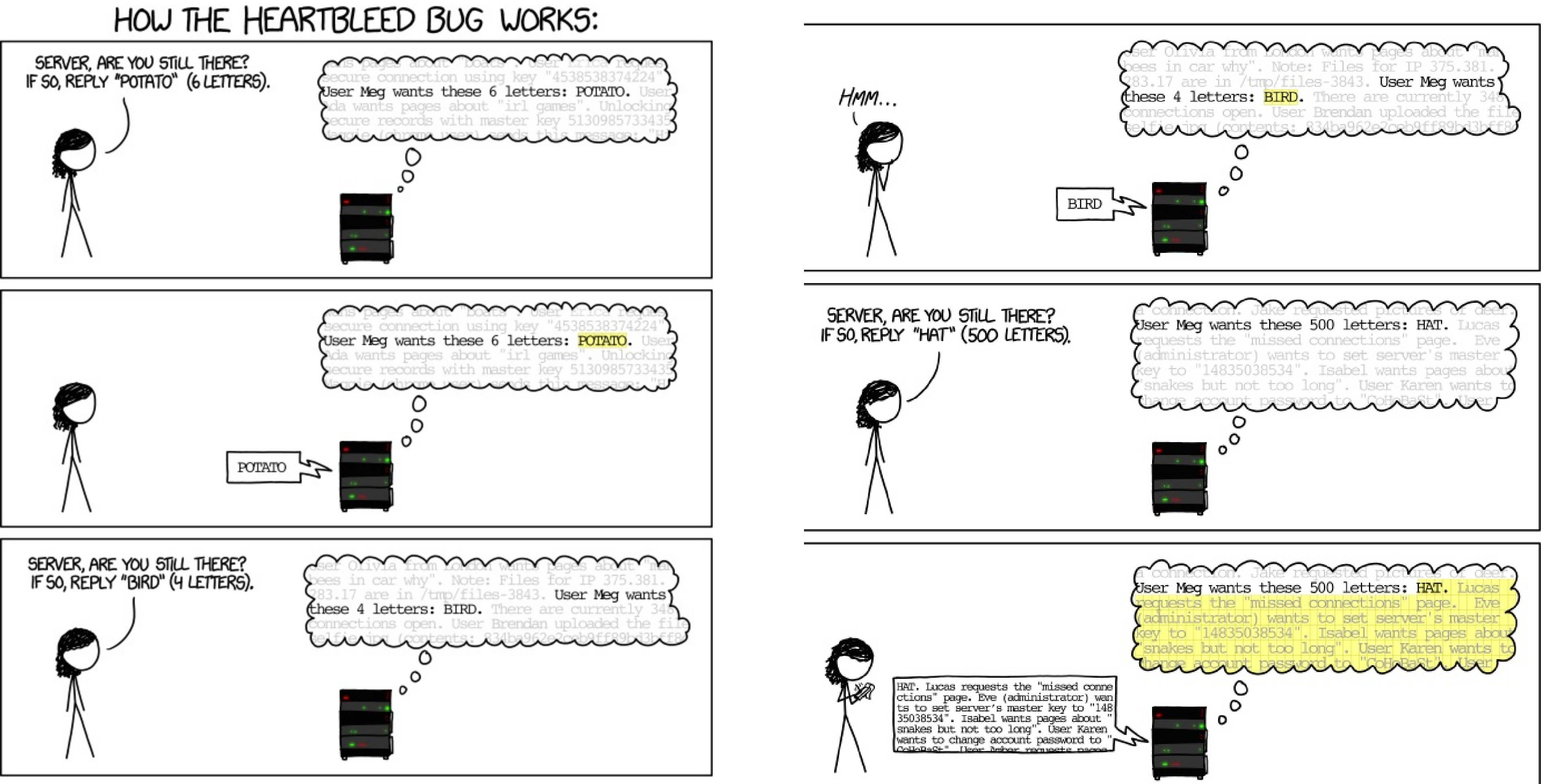
Stack Smashing (Stack Overflow): Shellcode Illustration



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

Heartbleed Bug: Illustration of Buffer *Over-Read* Request

Source: http://xkcd.com/1354/



8.3.4 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

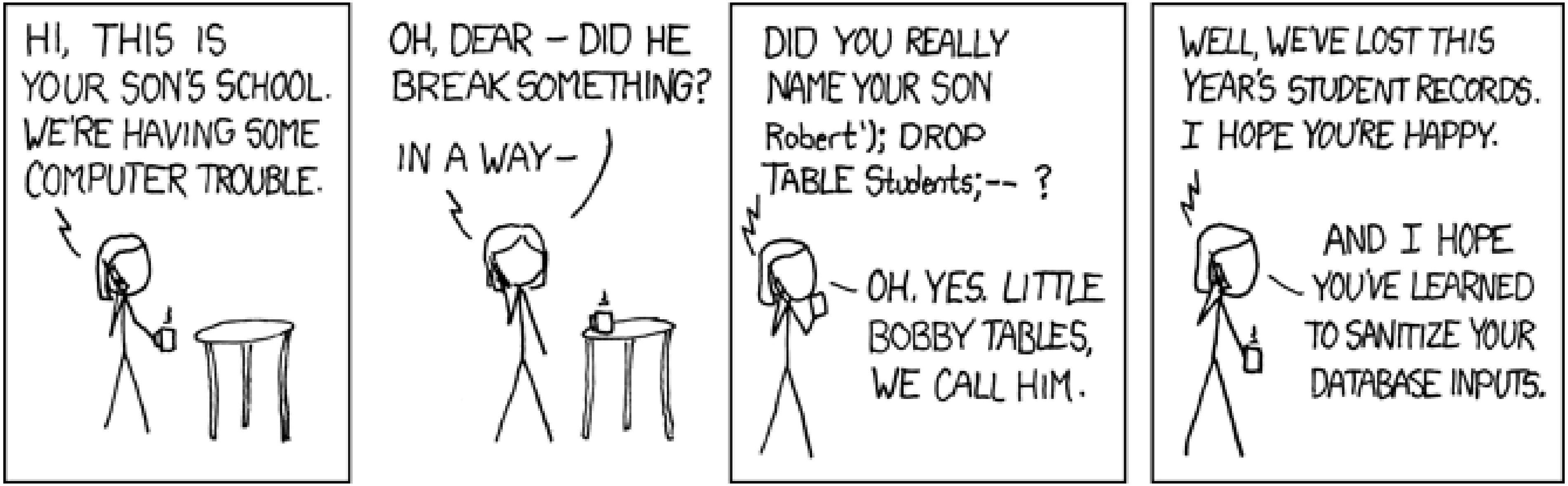
  `Bob’ OR 1=1 --`
- The interpreter, after seeing this script

  ```
  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/

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

8.3.5 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

8.4 Defense and Preventive Measures

Read http://en.wikipedia.org/wiki/Bounds_checking

General Comments

- As illustrated in previous examples, **many bugs** and **vulnerabilities** are due to **programmer's ignorance**
- In general, it is **difficult** to analysis a program to ensure that it is bug-free (recall the "halting-problem")
- There is no "fool-proof" method
- However, various useful **counter measures** are available as discussed next

8.4.1 Input Validation using Filtering

Filtering

- In almost all examples we have seen, the attack is carried out by feeding **carefully-crafted input** to the system
- Those inputs **do not** follow the **“expected” format**:
  e.g. the input is too long, contains control/meta characters, contains negative number, etc.
- Hence, a preventive measure is to perform an **input validation/filtering** whenever an input is obtained from the user: if the input is not of the expected format, reject it

Problems with Filtering

- It is **difficult** to ensure that the filtering is **“complete”** (i.e. it doesn’t miss out any malicious strings), as illustrated in the example on UTF
- In that example on UTF, the input validation intend to detect the substring “ . . / ”
- Unfortunately, there are **multiple representations** of “ . . / ” that the programmer is not aware of
- A filter that completely **blocks all bad inputs** and **accepts all legitimate inputs** is *very difficult* to design

Filtering

- There are generally **two approaches** of filtering:
   1. ***White list***: A list of items that are known to be **benign** and allowed to **pass**, which could be expressed using regular expression
      However, some legitimate inputs may be blocked.
   2. ***Black list***: A list of items that are known to be **bad** and to be **rejected**.
      For example, to prevent SQL injection,
      if the input contains meta characters, reject it.
      However, some malicious input may be passed.
- *Which type of filtering is then more secure?*

8.4.2 Using “Safer” Functions

Safer Function Alternatives

- Completely **avoid** functions that are known to create problems
- Use the **"safer" versions** of the functions
- C/C++ have many of those:

  `strcpy()` ←→ `strncpy()`

- Again, even if they are avoided, there ***could*** still be vulnerability: recall the example that uses a combination of `strlen()` and `strncpy()` in your **tutorial**

8.4.3 Bounds Checking and Type Safety

Bounds Checking

- Some programming languages (e.g. Java, Pascal) perform ***bounds checking*** at runtime
- That is, when an array is declared, its upper and lower bounds have to be declared
- At runtime, whenever a reference to an array location is made, the index/subscript is **checked** against the upper and lower bounds
- Hence, a simple assignment like:

```
a[i] = 5;
```

would **consists of** these steps:

1. Checks that `i` ≥ the lower bound;
2. Checks that `i` ≤ the upper bound; and
3. Assigns 5 to the memory location

Bounds Checking

- If the checks **fail**, the process will be **halted** (or an exception is to be thrown as in Java)
- The added first 2 steps reduce efficiency, but will **prevent** buffer overflow
- Many of the known vulnerabilities is due to buffer overflow that can be prevented by this **simple bounds checking**: visit http://cve.mitre.org/cve/cve.html to see how many entries contains “buffer overflow” as keywords

- The infamous C and C++ **do not** perform bounds checking!
- Yet, **many** pieces of software are written in C/C++!

Some Words of Wisdom

C. A. R. Hoare (1980 Turing Award winner) on his experience in the design of ALGOL 60, a language that **included** bounds checking:

> "A consequence of this principle is that every occurrence of every subscript of every subscripted variable was on every occasion checked at run time against both the upper and the lower declared bounds of the array. Many years later we asked our customers whether they wished us to provide an option to switch off these checks in the interest of efficiency on production runs. Unanimously, they urged us not to—they already knew how frequently subscript errors occur on production runs where failure to detect them could be disastrous. I note with fear and horror that even in 1980, language designers and users have not learned this lesson. **In any respectable branch of engineering, failure to observe such elementary precautions *would have long been against the law*.**"

Type Safety

- Some programming languages carry out ***"type" checking*** to ensure that the arguments an operation get during execution are always correct
- For example: **`a = b;`**
  if `a` is a 8-bit integer, `b` is a 64-bit integer, then the type is **wrong**
- The checking could be done at **runtime** (i.e. **dynamic type checking**), or when the program is being **compiled** (i.e. **static type checking**)
- Bounds checking can also be **considered** as one mechanism that ensures "type safety"
- For example in Pascal:

```
Type    indexrange = 1..10;
Var     A: array [indexrange] of integer;
Begin
    A[0] = 5;
```

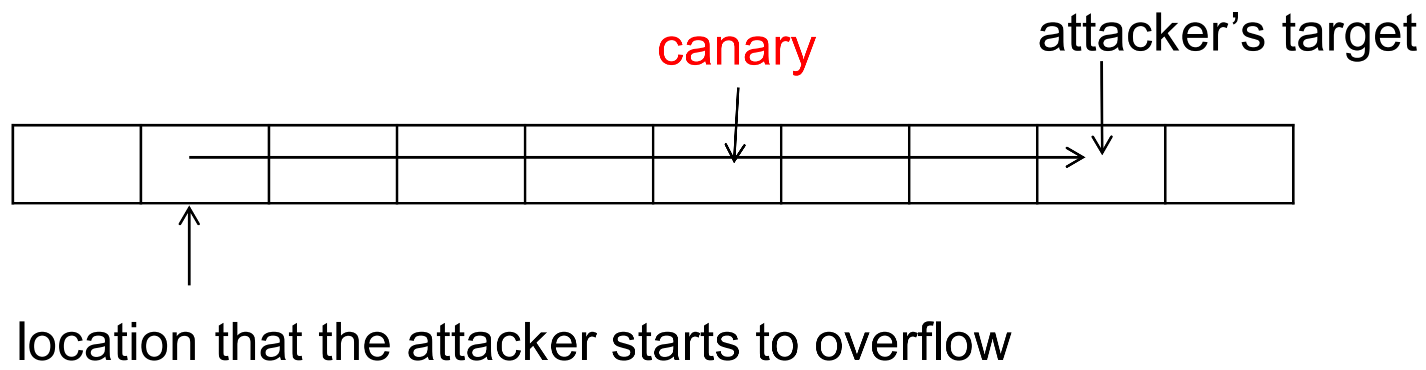
wrong type

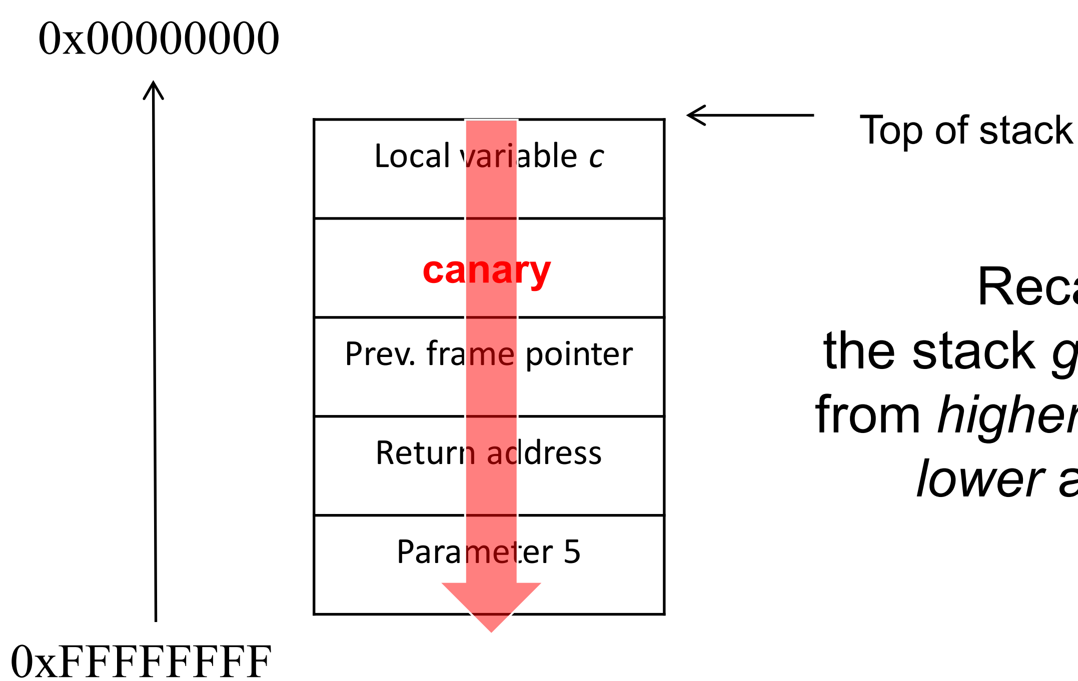
8.4.4 Memory Protection (Randomization, Canaries)

Canaries

- **Canaries** are "secret" values inserted at carefully selected memory locations at runtime
- Checks are carried out at **runtime** to make sure that the values are **not** being modified: if so, halts
- Canaries can help ***detect*** overflow, especially stack overflow:
  - In a typical buffer overflow, **consecutive** memory locations have to be over-ran: the canaries would be **modified**
- It is important to keep the values "**secret**":
  if the attacker knows the value, it may able to write the secret value to the canary while over-running it!
- (***Optional***) In Linux, you can **turn off** gcc canary-based stack protection by supplying this flag when invoking gcc:
  `-fno-stack-protector`

Canaries





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

Stack Smashing Detected: Program 1

```
#include <stdio.h>
#include <string.h>

void copy_this(char *arg1)
{
   char text[15];

   strcpy(text, arg1);
   printf("Your supplied argument is :%s\n", text);
   printf("Function is exiting\n");
}

int main(int argc, char *argv[])
{
    copy_this(argv[1]);

    printf("main() is exiting\n");
    return 0;
}
```

The buffer to overflow (inside this function)

Stack Smashing Detected: Program 1

- If compiled with stack protector:

```
./program-too-wsp
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your supplied argument is
:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Function is exiting
*** stack smashing detected ***: ./program-too-wsp
terminated
Aborted (core dumped)
```

- If compiled *without* stack protector:

```
./program-too-wosp
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your supplied argument is
:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Function is exiting
Segmentation fault (core dumped)
```

Stack Smashing Detected: Program 2

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char text[15];

    strcpy(text, argv[1]);
    printf("Your supplied argument is :%s\n", text);

    printf("main() is exiting\n");
    return 0;
}
```

The buffer to overflow (inside main)

Stack Smashing Detected: Program 2

- If compiled **with** stack protector:

```
./program-wsp aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your supplied argument is
:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
main() is exiting
*** stack smashing detected ***: ./program-wsp
terminated
Aborted (core dumped)
```

- If compiled **without** stack protector:

```
./program-wosp aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Your supplied argument is
:aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
main() is exiting
Segmentation fault (core dumped)
```

Memory Randomization

- It is to the attacker's advantage when the data and codes are always stored in the same locations in memory
- ***Address space layout randomization (ASLR)*** is a **prevention** technique that can help decrease the attacker's chance
- ASLR: **randomly** arranges the address space positions of **key data areas** of a process, including:
  the base of the executable and the positions of the stack, heap and libraries
- *(Details are omitted in this module)*
- ***Optional***: in Linux, you can turn off (disable) address randomization using:
  `sudo sysctl -w kernel.randomize_va_space=0`

8.4.5 Code Inspection

Code Inspection

- **Manual checking**: manually checks the program, is tedious
- **Automated checking**: some automations and tools are possible
- An example is ***taint analysis***:
  - Variables that contain input from the (potential malicious) users are labeled as ***sources***
  - Critical functions are labeled as ***sinks***
  - Taint analysis **checks** whether any of the sink's arguments could potentially be affected (i.e. tainted) by a source
  - Example: sources = user input
    sink: opening of system files, function evaluating a SQL command
  - If so, special check (e.g. manual inspection) would be carried out
  - Taint analysis can be **static** (i.e. checking the code without running/ tracing it), or **dynamic** (i.e. running the code with some inputs)

8.4.6 Testing

Testing

- Vulnerability can be discovered via **testing**
- Types of testing:
  - ***White-box*** testing:
    the tester has an access to the application's **source code**
  - ***Black-box*** testing:
    the tester does not have an access to the source code
  - ***Grey-box*** testing:
    A combination of the above, reverse-engineered binary/executable

Testing

- Security testing attempts to discover **intentional attack**, and hence would test for inputs that are **rarely occurred** under normal circumstances
- Examples: very long names, names containing numeric values, string containing meta characters, etc.
- ***Fuzzing*** is a technique that sends **malformed inputs** to discover vulnerabilities:
  - There are techniques that are more effective than sending in random inputs
  - Fuzzing can be automated or semi-automated: *(the details are not required)*
- **Terminology**: white list vs black list, white-box testing vs black-box testing, white hat vs black hat

8.4.7 The Principle of Least Privilege

The Principle of Least Privilege

Apply the "***Principle of Least Privilege***":

- When writing a program, be **conservative** in elevating the privilege
- When deploying software system, do **not** give the users more access rights than necessary, and do **not** activate unnecessary options

The Principle of Least Privilege

- Example:

  Software contain many **features**: e.g. a web-cam software could provide many features so that the user can remotely control it.
  A user can choose to set which features to be on/off.

  Suppose you are the developer of the software.
  Should all features to be **switched on by default** when the software is shipped to your clients?

  If so, it is **the clients' responsibility** to "***harden***" the system by selectively **switch off the unnecessary features**.
  Your clients might not aware of the implications and thus at a higher risk.

- Terminology: What does "*hardening*" mean?

8.4.8 Patching
(Keeping up to Date)

Vulnerability Lifecycle

- **Life-cycle of a vulnerability**:
  (1) a **vulnerability** is discovered → (2) affected code is fixed →
  (3) the revised version is tested → (4) a **patch** is made public →
  (5) patch is applied
- In some cases, the vulnerability could be announced (1?) **without the technical details** before a patch is released:
  the vulnerability is likely to be known to only a small number of attackers (even none) before it is announced
- When **a patch** is released (4), the patch can be **useful to attackers** too: they can inspect the patch and derive the vulnerability
- Hence, interestingly, the number of successful attacks can **go up** after the vulnerability/patch is announced:
  since more attackers would be aware of the exploit
  (see the next slide)

Vulnerability Lifecycle

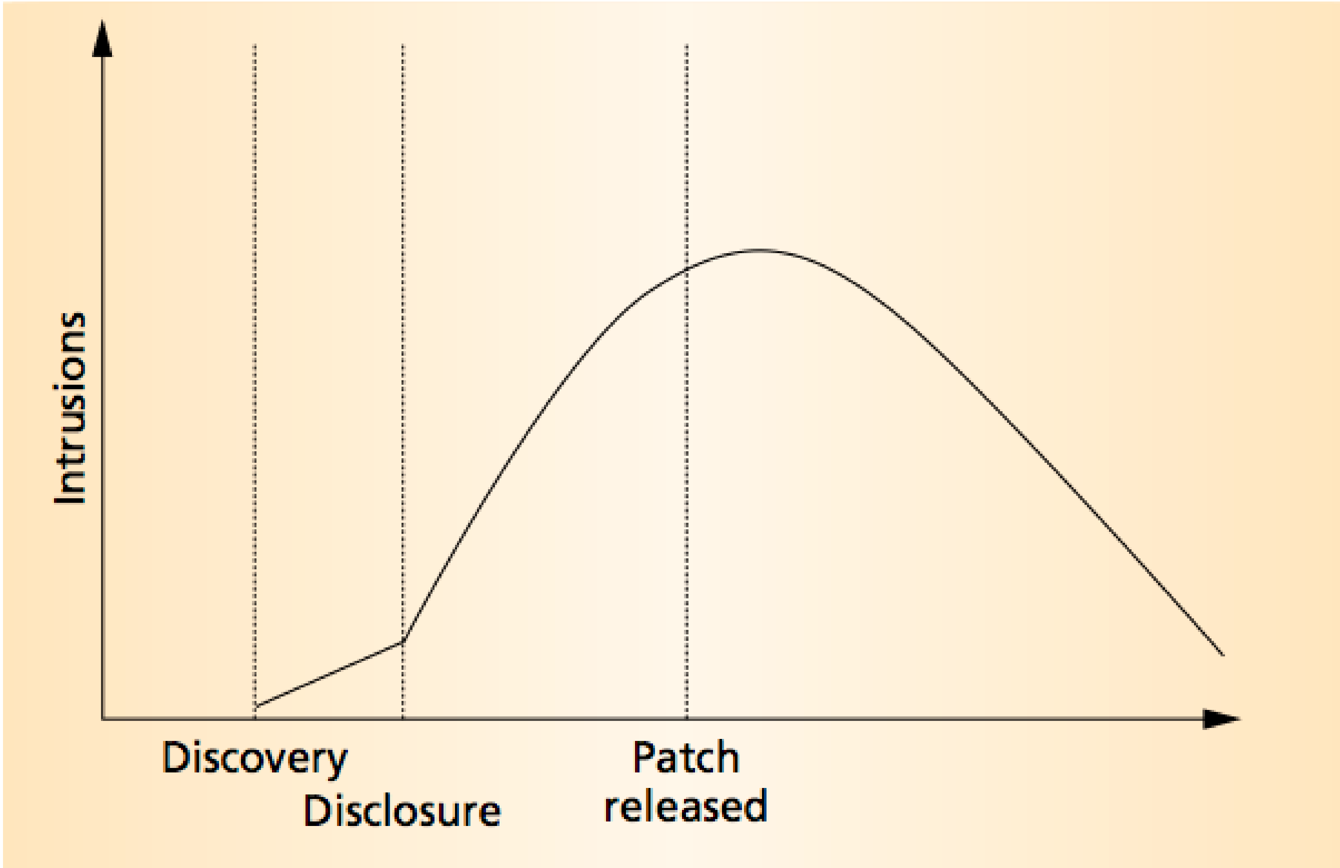


Figure 1. Intuitive life cycle of a system-security vulnerability. Intrusions increase once users discover a vulnerability, and the rate continues to increase until the system administrator releases a patch or workaround.

image obtained from
William A. Arbaugh et al. Windows of vulnerability: A case study analysis. IEEE Computer, 2000.

http://www.cs.umd.edu/~waa/pubs/Windows_of_Vulnerability.pdf

Patching

- It is crucial to apply the patch **timely**
- Although it seems easy, applying patches is **not** that straightforward:
  - For **critical systems**, it is not wise to apply the patch immediately before rigorous testing:
    E.g. after a patch is applied, the train scheduling software crashes
  - Patches might **affect** the applications, and thus affect an organization **operation**:
    E.g. after a student applied a patch on Adobe Flash, he couldn't upload a report to LumiNUS and thus missed a project deadline
- ***Patch management*** is a field of study
- See the guide on patch management issued by NIST:
  "*Guide to Enterprise Patch Management Technologies*", 2013,
  http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-40r3.pdf

Summary: Defense Mechanisms and Stages of SDLC

Adopt various counter measures at **different *stages of SDLC***:

- **Development** stage:
  - Using safer functions
  - Bounds checking and type safety
  - Filtering (input validation)
  - Code inspection (taint analysis)
  - The principle of least privilege*
  - Executable generation with enabled memory protection*
- **Testing** stage:
  - Testing: fuzzing, penetration testing
- **Deployment** (including software execution) stage:
  - Runtime memory protection*: randomization
  - The principle of least privilege*: disable unnecessary features
  - Patching

* Applicable to *multiple* stages