See [PF] pg 131-166
or
see [Gollmann] 10.1, 10.2, 10.3

Lecture 8: Secure Programming

8.1 Overview

8.2 Example of unsafe function: printf()

8.3 Data Representation

8.4 Buffer Overflow

8.5 Integer Overflow

8.6 Code Injection

8.7 Undocumented Access Point

8.8 TOCTOU

8.9 Defense

- Program has to be “correct”.
- Program has to be “efficient”.
- *Program has to be “secure”*.

Privilege Escalation Attack

- Many programs are not implemented properly, different from the original intents of the programmers.

- Under typical execution environment, the program may function normally, since the conditions that trigger failure rarely occur. However, a malicious attacker may search for the conditions and intentionally trigger them.

- E.g. the attacker may supply input in a form that is not anticipated by the programmer, which causes the process to access resources (which the programmer originally not intended to), to execute some "injected" codes, or crash. By feeding in carefully crafted input, the attack might able to compromise "execution integrity".

- Many bugs are very simple and easy to correct when found. But program for complex system are large, making detecting such bug challenging. E.g. Window XP has 45 millions SLOC (source line of codes) http://en.wikipedia.org/wiki/Source_lines_of_code

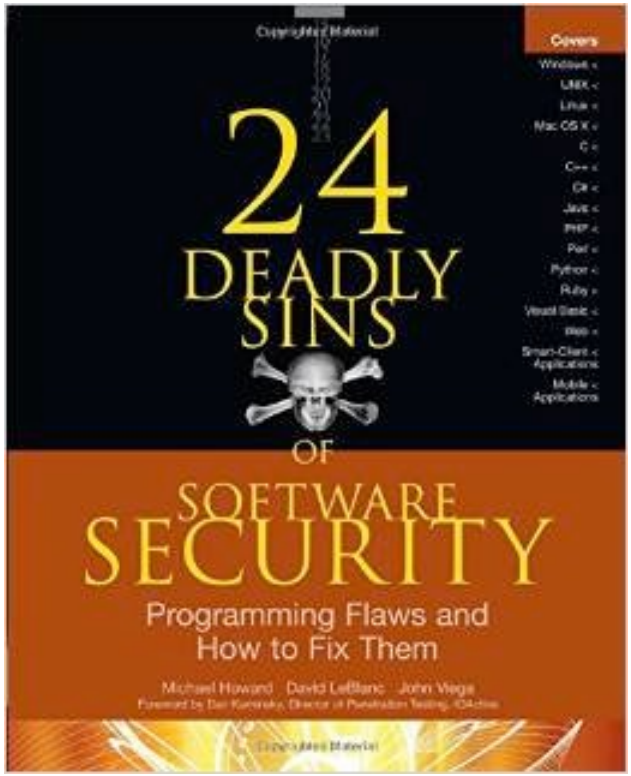
- Many bugs are specific to applications. The book in the next slide classifies them into 24 sins.

References

well known references:

Michael Howard and David LeBlanc, *Writing Secure Code*, 2nd ed, Microsoft Press, 2002.

Michael Howard, David LeBlanc, and John Viega, *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill, 2010.



8.2 Unsafe function. Printf()

read wiki http://en.wikipedia.org/wiki/Uncontrolled_format_string

read https://www.owasp.org/index.php/Format_string_attack

For more detail, see
http://www.cis.syr.edu/~wedu/Teaching/cis643/LectureNotes_New/Format_String.pdf

- `printf()` is a function in C for formatting output.
- It can take in one, or two or more than 2 parameters.

- Common usage is

```
printf ( format,  s)
```

where `format` specifies the format, and `s` is the variable to be displayed. E.g.

```
printf (”the value in temp is %d\n", temp)
```

would display the following if `temp` contains the value 100

```
the value in temp is 100
```

- The special symbol “%d” indicates the type of the variable.

E.g.

```
printf ( “1st string is %s   2nd string is %s”,s1, s2);
```

Hence, printf() would

1. first displays “1st string is ”;
2. next, lookups for the 2nd parameter and displays its value;
3. displays “2nd string is ”;
4. Finally, lookups for the 3rd parameter and displays its value.

- When only one parameter is supplied

```
printf ( "hello world" )
```

only `"hello world"` will be displayed.

If there is "%d" in the string, during execution, printf() will still fetch value of the 2nd parameter, from the supposing location of the 2nd parameter, and display it. This is done even if the "printf" in the program does not have the 2nd parameter, e.g.

```
printf ( "hello world %d" )
```

If the corresponding value in pickup location happened to be 15, then what being displayed will be

```
hello world 15
```

You may wonder why can't printf() double-check that the program has only one parameter. Doing so would incur additional check during runtime and thus reduce efficiency. We omit the details here.

Example

In the following eg. if the value of `t` is supplied by a user (attacker), this would allow the attacker to get more information by careful design of the string t.

declare a string of 100 characters.

```
#include <stdio.h>

int main()
{
char t[100];

scanf ("%s",t );
printf (t);
}
```

What if send in 'hello %s'

read in a string and store it in *t*.

display the string *t*.

Simple preventive measure.

Avoid using the following form where the variable `t` is a variable:

- `printf (t)`
- `printf (t, a1, a2)`

Safe version

- `printf ("hello" );`
- `printf ("The value of %s is %d", a1, a2)`

How such printf vulnerability can be exploited.

- If a program is vulnerable, the attacker might able to

  (1) obtain more information (*confidentiality*)

  (2) cause the program to crash, e.g. using %s. (*execution integrity*)

  (3) modifying the memory content using "%n". *(memory integrity which might lead to execution integrity)* Can craft the previous few characters to return a specific return address pointer

- If the program that invokes *printf* has elevated privilege (set UID enabled), a user (the attacker) might able to obtain information that was previously inaccessible. E.g
  - Suppose the program for a web-server has the unsafe printf. Under normal usage, the server would obtain a request from the client, and then display (via the printf) it for confirmation. Now, a client (the attacker) might able to submit a web request to obtain sensitive information (e.g. the secret key), or to cause the web-server to crash.

8.3 Data Representation

- Different parts of a program (or system) adopts different data representations. Such inconsistencies could lead to vulnerability.

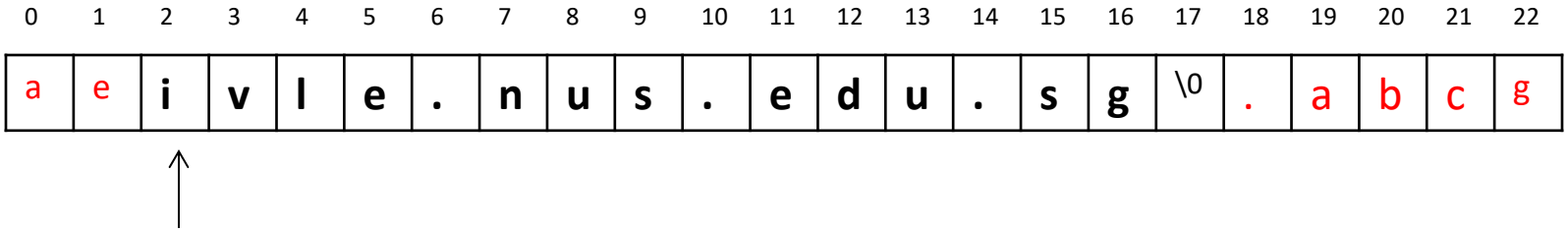
Example: bug in a SSL implementation that is exploitable.

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

read https://tools.cisco.com/security/center/viewAlert.x?alertId=19157

See https://security.stackexchange.com/questions/31760/what-are-those-nul-bytes-doing-in-certificate-subject-cn

String has variable length. How to represent a string?



The *printf()* in C adopts an efficient representation. The length is not stored explicitly. The first occurrence of the "null" character (i.e. byte with value 0) indicates end of the string, and thus implicitly gives the string length.

Not all systems adopt the above convention. Let's call the above NULL termination, and other non-NULL termination.

A system which uses both null and non-null definitions to verify the certificate may get “confused”.

E.g.

Consider a browser implementation that:

(1) verifies a certificate based on non-null termination;

(2) displays the name based on null termination.

Now, there could be an attack as described in the next slide.

*: the X509 standard adopt non-null termination to represent Common Name. See https://stackoverflow.com/questions/5136198/what-strings-are-allowed-in-the-common-name-attribute-in-an-x-509-certificate/5142550#5142550

1. The attacker registered the following domain name, and purchased a valid certificate C_1 with the following name from a CA.

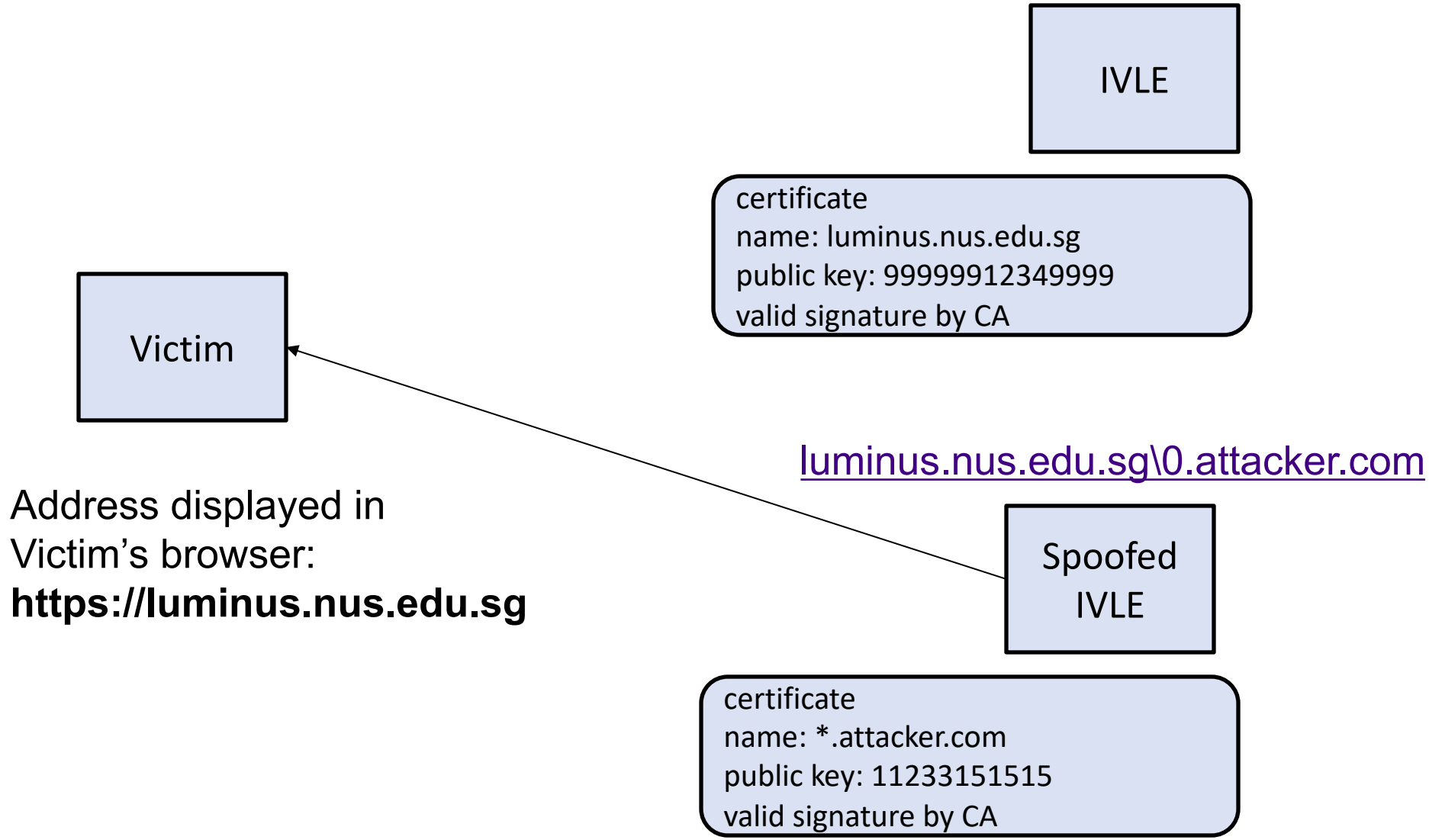
   *.attacker.com

2. The attacker setup a spoofed website of Luminus with the domain name.

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

3. The attacker directed a victim to the spoofed webserver (For e.g., attacker controlled the physical layer. Alternatively, using phishing email, the attacker tricked the victim to visit the attacker's webserver).

4. The attacker's webserver presented the certificate C_1.
   - The browser displayed the address (based on null-termination) on the browser's address bar.
   - The browser verified the certificate (based on non null-termination). Since it was valid, the browser displayed the pad-lock in the address bar.


IVLE
certificate
name: luminus.nus.edu.sg
public key: 99999912349999
valid signature by CA
Victim
luminus.nus.edu.sg\0.attacker.com
Address displayed in
Victim's browser:
https://luminus.nus.edu.sg
Spoofed
IVLE
certificate
name: *.attacker.com
public key: 11233151515
valid signature by CA

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.

Another e.g. on data representation: UTF-8 encoding

Optional. Skip.

- UTF-8 encoding was defined for "Unicode" on systems that were designed for ASCII. (see https://en.wikipedia.org/wiki/UTF-8 for details)

- ASCII characters remain unchanged in UTF-8. (Recall that there are 128 ASCII characters and each starts with the bit 0 in a single byte)

- The following are byte representations of Unicode. Left-hand-side is the Unicode representation. Right-hand-side is the bytes representation.

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

7bits

11bits

16bits

21bits

the xxx bits are the least significant bits of the respective Unicode character.

By the above rules, byte representation of an UTF-8 character is unique.

However, many implementations accepts multiple and longer "variants" of a character.

e.g. different representations of a same UTF-8 character

Optional. Skip.

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

  0010 1111 i.e. 2F

under UTF-8 definition, the one-byte "2F" is the unique representation.

However, in many implementations, the following longer variants are also treated to be '/'.

(2 bytes version) `110`**`00000`** `10`**`101111`**

(3 bytes version) `1110`**`0000`** `10`**`000000`** `10`**`101111`**

(4 bytes version) `11110`**`000`** `10`**`000000`** `10`**`000000`** `10`**`101111`**

That is, all the above would be decoded to '/'.

Now, there could be inconsistency between (1) the verification process (2) the actual usages.

E.g on UTF-8

Optional. Skip.

- In a typical file system, files are organized as directory. E.g. the full path name of a file is

  `/home/student/alice/public_html/index.html`

Suppose a server-side program, on receiving a string ***f*** from a client, carry out the followings:

Step1: Appends *f* to the following string, and take the concatenated string as F

`/home/student/alice/public_html/`

Step2: Makes system call to read the file F, and sends the content to the client.

In the above example, the client can be any public user. (similar to http). The original intention is that, the client can only retrieve files in the directory `public_html`.

However, an attacker (which is the client) may send in this string

`../cs2107report.pdf`

Which file would be read and sent? `/home/student/alice/cs2107report.pdf`

In order to prevent this, one method is to add an "input validation" step, making sure that `../` does not appear as a substring in ***f***.

is this check "complete"?

Optional. Skip.

(With input validation)

Step 1a: Checks that the string *f* does not contain the following. If so, quit.

`../`

Step1: Appends *f* with the following string to get F

`/home/student/alice/public_html/`

Step2: Makes system call to read the file F, and sends the content to the client.

Now, further suppose that

(1) the system call in Step 2 uses a convention that the special character '%' indicates that the corresponding byte is represented by its hexadecimal value (same as http). E.g. `/home/student/%61lice/`

where `%61` is to be replaced by `a`

(2) the system call uses UTF-8.

Optional. Skip.

Then, the check carried out by Step (1)a is incomplete. It misses some cases.

Any one of the following string will pass the check by Step (1)a, since it literally does not contain the substring

```
../
```

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

However, eventually, the filename will be decoded to

```
/home/student/alice/public_html/../cs2107report.pdf
```

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	~

Example: IP address

Recall that the 4 bytes ip address is typically written as the string "132.127.8.16". An ip address can be represented as

1. A string, e.g "132.127.8.16"
2. 4 integers
3. A single 32-bit inegere.

Consider a black list containing a lists of ip-addresses. Each ip-address is represented as 4 bytes. A programmer wrote a sub-routine `BL` that, takes in 4 integers (each integer is of the type "int", i.e. represented using 32 bits), and check whether the ip-address represented by these 4 integers is in the black list. (in C, `int BL ( int a, int b, int c, int d )`)

In the routine `BL`, the black list is stored as 4 arrays of integers `A`, `B`, `C`, `D`. Given the 4 input parameters `a`, `b`, `c`, `d`, the routine `BL` simply searches for the index i such that `A[i] ==a, B[i]==b, C[i]==c, and D[i]==d.`

"12.12.0.1025" ← input

↳ change to 4 int = 12 . 12 . 0 . 1025 (not in BL)

boundary = 2^8

but exceeded so ip change

↳ change to 32 bit int. = $12 \times 2^{24} + 12 \times 2^{16} + 0 + 2^{10} + 1$

= | 12 | 12 | 3 | 1 |

Now, the following program is vulnerable. Why?

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

(2) Extract four integers (each integer is of type `int`, i.e. 32-bits) from the string *s*. 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 (a, b, c, d) is not in the black list. If so, quit.

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

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

Guideline: Use Canonical representation

Do not trust the input from user. Always convert them to a standard (i.e. canonical) representation immediately.

7.4 Buffer Overflow

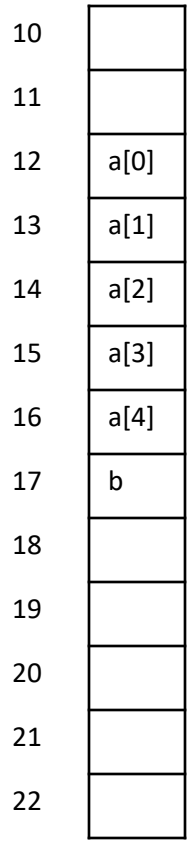
memory

C and C++ do not employ “bound check”. This achieves efficiency but prone to bugs.

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/Overruns

The previous example illustrates ***Buffer Overflow*** (aka buffer overrun). In general, buffer overflow refers to a situation where data are written beyond the (buffer's) boundary.

In the previous example, the array is a buffer of size 5. The location a[5] is beyond its boundary. Hence, writing on it causes "buffer overflow".

Well-known function in C that prone to buffer overflow is the string copying function: `strcpy`

> man strcpy

NAME

stpcpy, stpncpy, strcpy, strncpy -- copy strings

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <string.h>

char *
stpcpy(char * dst, const char * src);

char *
stpncpy(char * dst, const char * src, size_t len);

char *
strcpy(char * dst, const char * src);

char *
strncpy(char * dst, const char * src, size_t len);
```

DESCRIPTION

The **stpcpy**() and **strcpy**() functions copy the string src to dst (including the terminating `\0' character.)

The **stpncpy**() and **strncpy**() functions copy at most len characters from src into dst. If src is less than len characters long, the remainder of dst is filled with `\0' characters. Otherwise, dst is not terminated.

The source and destination strings should not overlap, as the behavior is undefined.

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

In the above, potentially the length of s2 can be more than 10 (length of s2 is determined by the first occurrence of null). The strcpy function copies the whole string of s2 to s1, even if the length of s2 is more than 10. Note that the "buffer" of s1 is only 10. Thus the extra values will be overflown and written to other part of the memory. If s2 is supplied by the (malicious) user, a well-crafted input could overwrite important memory and compromise execution integrity.

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 also lead to vulnerability.

Stack Smashing

Stack smash is a special case of buffer overflow that targets stack. Buffer overflow on stack is called stack overflow, stack overrun, or stack smashing.

If the return address (which is stored in stack) is modified, the execution control flow will be changed. A well-designed overflow could "inject" attacker's shell-code into the memory, and then run the shell-code. *(this is an example illustrating how compromise of memory integrity could lead to compromise of execution integrity)*

There are effective (but not foolproof) mechanisms (canary) to detect stack overflow. (to be covered later)

See a "classic" tutorial:
Aleph One, *Smashing The Stack for Fun and Profit*, 1996, https://www.cs.umd.edu/class/spring2016/cmsc414/papers/stack-smashing.pdf

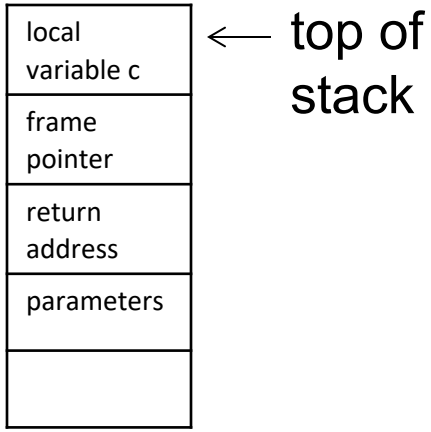
Stack Smashing.

When a function is called, the parameters, return address, local variables are pushed in a stack.

Consider the following segment of C program:

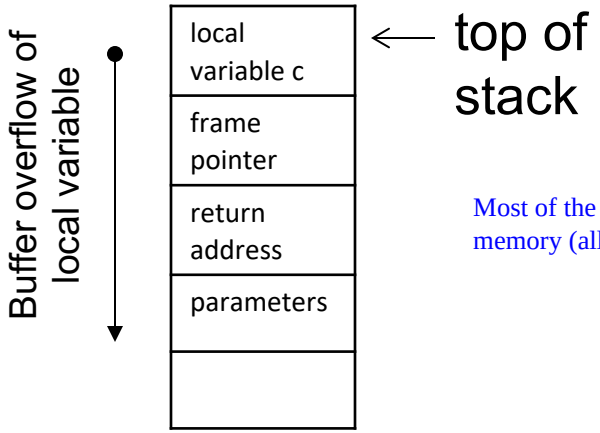
```
int foo( int a)
{  char c[12];
   ......
  strcpy (c,  bar );    /* bar is a string input by user */
}

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



After "foo(5)" is invoked, a few values are pushed into the stack.

- If an attacker manages to modify the return address, the control flow will jump to the address indicated by the attacker.



Most of the time, the attack on the stack will remain within the allocated process memory (allocated by the OS) for this program

read https://en.wikipedia.org/wiki/Stack_buffer_overflow

(Read the first section: **Exploiting stack buffer overflows.** Others, sections 2,3 & 4, are optional.**)**

7.5 Integer Overflow

not to be confused with “buffer 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 and java statements what would be the final value of `a`?

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

actually (a+2) % n-bits = 256

Its value is 0. The addition is with respect to module 256.

Hence, the following predicate is not necessary always true.

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

Many programmers don't realize that, leading to vulnerability (see tutorial).

7.6 Code (Script) injection

(optional) Tool: sqlmap

"sqlmap is an open source software that is used to detect and exploit database vulnerabilities and provides options for injecting malicious codes into them. It is a penetration testing tool that automates the process of detecting and exploiting SQL injection flaws providing its user interface in the terminal"

Scripting language

- A key concept in computer architecture is the treatment of "code"(i.e. program) as "data". In the context of security, mixing "code" and "data" is potentially unsafe. Many attacks inject malicious codes as data.

- *Scripting languages* are particularly vulnerable to such attack. ***Scripts*** are programs that automates the execution of tasks that could alternatively be executed line-by-line by a human operator. Scripting language are typically "interpreted", instead of being compiled. Well-known examples are PHP, Perl, JavaSript, SQL.

- Many scripting languages allow the "script" to modify itself before being interpreted. This opens up the possibility of injecting malicious code into the script.

- Let's consider the well-known ***SQL injection attack***.

SQL injection

SQL is a database query language.

Consider a database (can be viewed as a table). Each column is associated with an attribute, e.g. "name".

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

The query script

SELECT * FROM client WHERE name = 'bob'

searches and returns the rows where the name matches 'bob'. The scripting language supports variables. For e.g. a script first gets the user's input and stores it in the variable **$userinput**. Next, it runs the following:

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

name = 'test'

In this example, let's consider an application where the names are secrets. Hence, only the authentic entity who knows a name can get the record. Furthermore, the attacker can decide the content in the variable **$userinput**.

Now, suppose the attacker can pass the following as the input.

anything' OR 1=1 --

That is, the variable **$userinput** becomes

anything' OR 1=1 --

The interpreter, after seeing the next script

SELECT * FROM client WHERE name = '$userinput'

simply substitutes the above to get

SELECT * FROM client WHERE name = 'anything' OR 1=1 --'

(remark: - - is interpreted as start of comment.)

The interpreter runs the above and return all the records.

Code injection + buffer overflow

- Note that the term “Code injection” is not limited to SQL injection. It is possible to exploit buffer overflow to inject *malicious code* and transfer the process execution to the malicious code. Details omitted. See http://www.cis.syr.edu/~wedu/Teaching/IntrCompSec/LectureNotes_New/Buffer_Overflow.pdf

Overflowing the buffer using specified code, until it reaches the return pointer to go all the way back to the start of the allocated buffer for input

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

8.7 Undocumented access point (Easter eggs)

- For debugging, many programmers insert “undocumented access point” to inspect the states. For example, by pressing certain combination of keys, value of certain variables would be displayed, or for certain input string, the program would branch to some debugging mode.

- These access points may mistakenly remain in the final production system, providing a “back door” for the attackers.

- Also known as Easter Eggs. Some Easter eggs are benign and intentionally planted by the developer for fun or publicity.

- There are known cases where unhappy programmer planted the backdoors.

Terminologies: Logic Bombs, Easter Eggs, Backdoors.

8.8 Race Condition (TOCTOU)

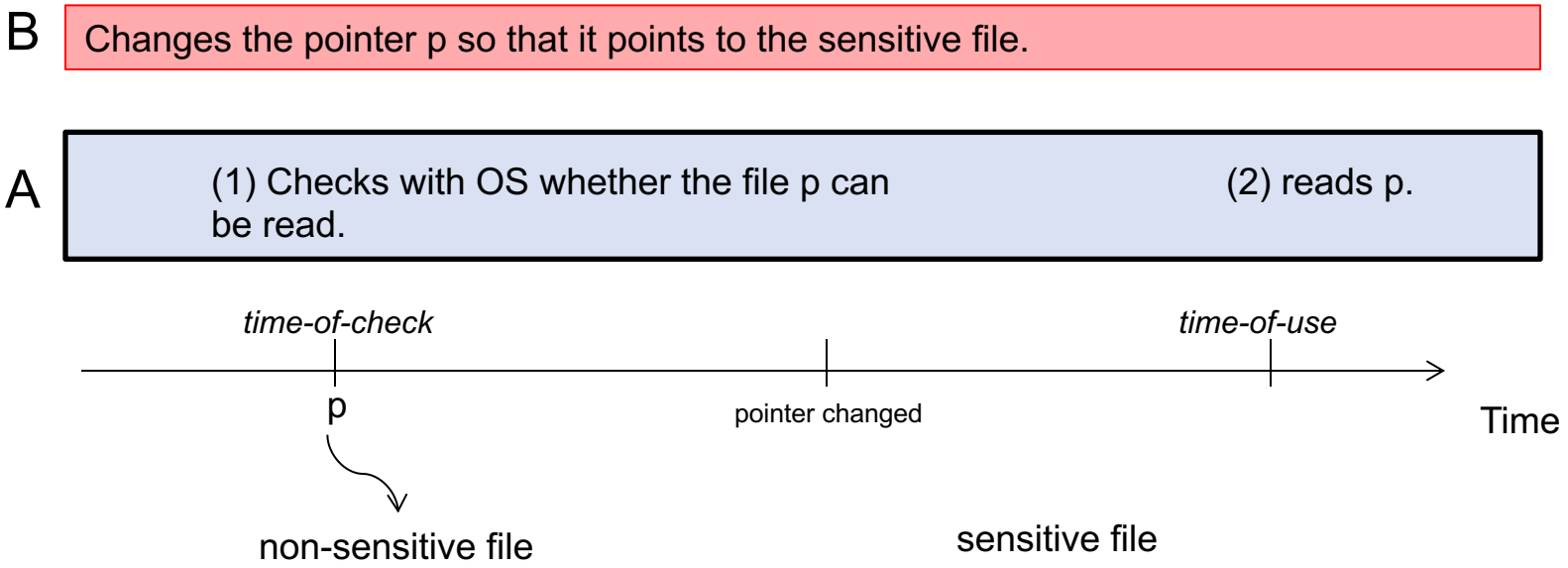
see

- https://cwe.mitre.org/data/definitions/367.html

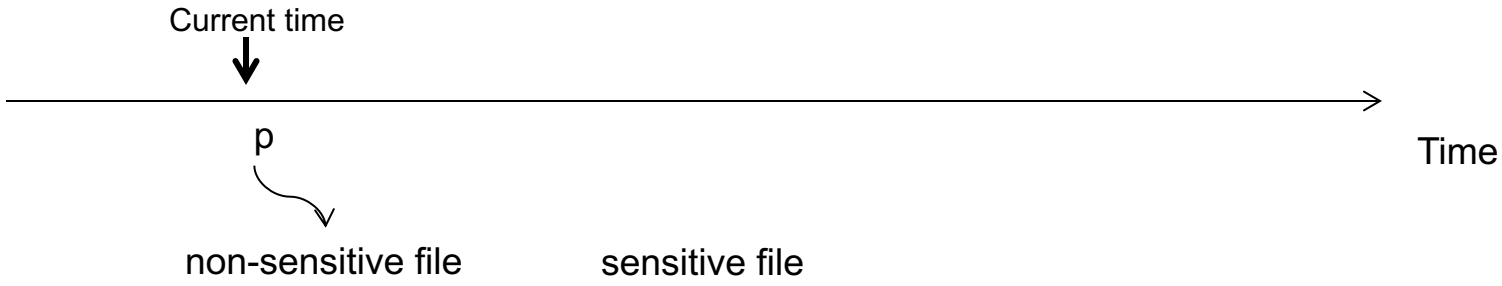
Race condition and TOCTOU

- Generally, race condition occurs when multiple processes access a piece of shared data in such a way that the final outcome depend on the sequence of accesses.
- In the context of security, the "multiple processes" typically refer to
  - (1) a process **A** that checks the permission to access the data, follow by accessing the data, and
  - (2) another process **B** (the malicious one) that "swaps" the data.
- There is a "race" among **A** and **B**. If **B** manages to be completed in between the time-of-check and time-of-use in **A,** the attack succeed. Thus, this type of race condition is also known as ***time-of-check-time-of-use*** **(TOCTOU)**.

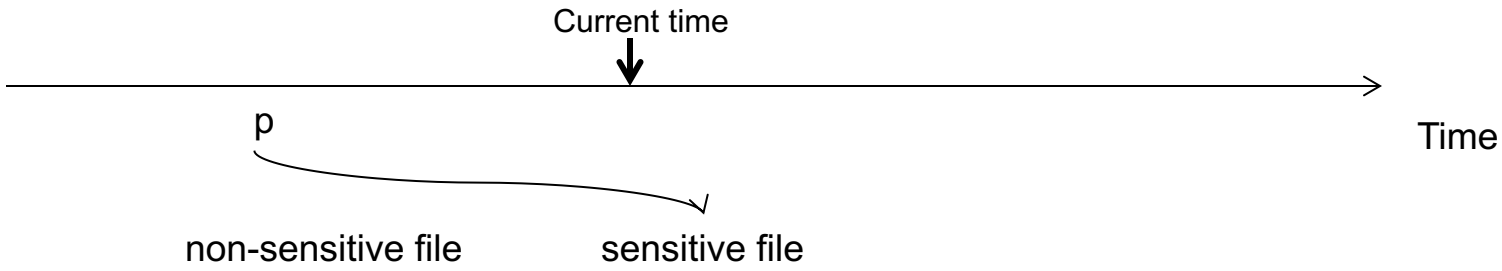
In the following e.g. B doesn't have permission to access the sensitive file. But B has permission to modify a file pointer ***p.***



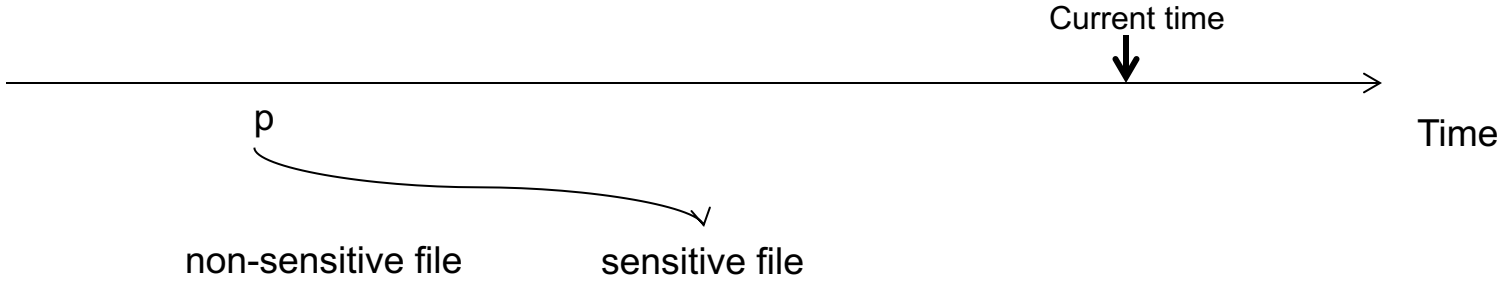
time-of-check : checking whether the process is authorized.



changing pointer



time-of-use: Accessing the file.



Example 1

See https://cwe.mitre.org/data/definitions/367.html for another example

- The system call `access(f,W_OK)` checks whether the user executing the program has the permission to write to the specified file `f`. It returns 0 if the process has the permission. The check is done based on the process ***real*** UID.
- This program is to be ran under elevated privilege (i.e setuid enabled, and the owner is root).

TOC →
TOU →

```
//  f is a string that contains the name of a file
//  fd is the file descriptor

if(!access(f, W_OK))              // check whether the real UID has write permission to f
{
    fprintf (stderr, "permission to operate %s granted\n", f );
    fd = open(f,O_RDWR);          // open the file with read and write access
    OP(fd);                        // a routine that operates the file. OP is not a system call
    ....
}
else
{
    fprintf(stderr,"Unable to open file %s.\n", f );
}
```

- The program has the "setuid enabled" and its owner is "root". Suppose `bob` invokes the program. Thus the real UID is `bob`, but effective UID is `root`. Furthermore suppose that the filename `f` is

  `/usr/year1/bob/list.txt`

  which is a file owned by `bob`.

- After `bob` starts the process, he immediately replaces the file

  `/usr/year1/bob/list.txt`

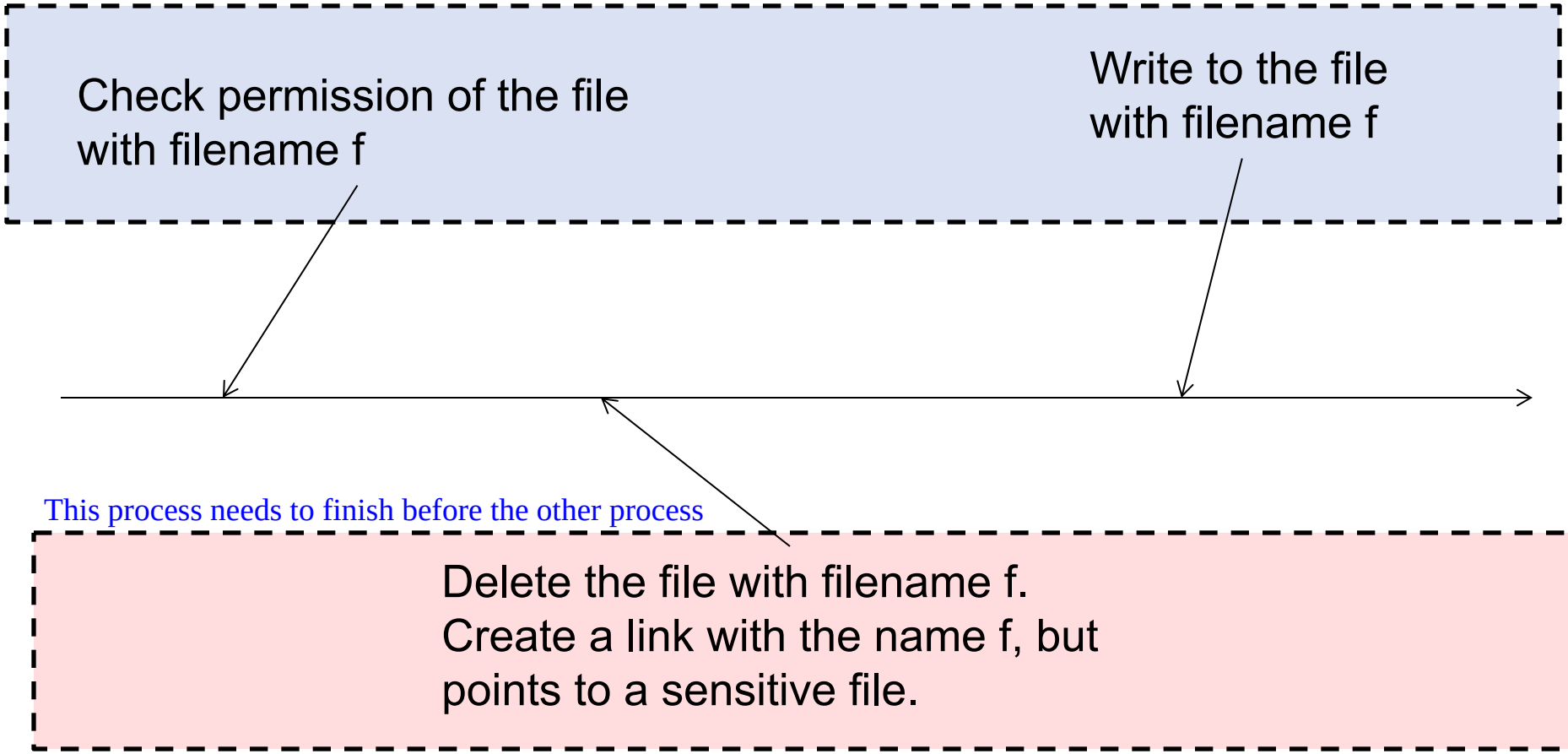
  by a symbolic link that points to a system file. `bob` does not has write permission of that system file, but has permission to change `list.txt`

- This can be done by running a script that carries out the following 2 steps:
  1. Delete `/usr/year1/bob/list.txt`
  2. Create a symbolic link with the filename f, and points to a system file. That is, issuing this Unix command

```
ln -s /usr/course/cz2017/grade.txt /usr/year1/bob/list.txt
```

- With some chances, `bob` wins the race. Hence the process operates on the system file `/usr/course/cz2017/grade.txt` instead of `/usr/year1/bob/list.txt`

Footnote: In this example, Bob is malicious.



`/usr/year1/bob/list.txt`

`/usr/course/cz2017/grade.txt`

Avoiding TOCTOU on file-checking (C-programming)

Optional.
This example is quite involved.

- In your program, avoid using separate system calls that takes the same filename as input. Instead, try to open the file only once (*and thus lock it to block further changes by other process*), and use the file handle or file descriptor. (in general, avoid using the system call `access()`. Open the file once using `open()` and then use `fsat()` to check the permission)

- A better practice would be this: avoid writing your own access-control on files. Leave the checking to the underlying OS. For instance, write the following in your program:
    - set the effective UID (of the process) to the appropriate user before accessing the file. In this way, the OS checks the permission and decide whether to grant or deny. After it is done, reset the effective UID back to root when required.

Safe version

Optional. This example is quite involved.

TOC

TOU

```
//  f is a string that contains the name of a file

fd = open(f, O_RDWR); //  open the file with read and write access

fstat(fd, &filestatus) )    //  get the file status and store them to filestatus
if ( CP (filestatus)  )    //  check permission of the file(CP is not a system call)
{
    fprintf (stderr, "permission to operate %s granted\n", f );
    OP(fd);              // a routine that operates the file (OP is not a system call)
    ....
}
else
{
    fprintf(stderr,"Unable to open file %s.\n", f );
}
```

Another Safe version

Elevated Privilege

Privilege lowered

Privilege restored to root

```
//  f is a string that contains the name of a file
//  u is the UID of the appropriate user (i.e. Alice)
//  r is the UID of root;
…

seteuid ( u );                      // from now onward, the effective UID is u
fd = open (f, O_RDWR);
OP (fd);
…

seteuid ( r );                      //  from now onward, the effective UID is root

…
```

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

8.9 Defense and preventive measures

8.9.1 Filtering (input validation)
8.9.2 Use safer functions
8.9.3 Bound check and “Type” Safety
8.9.4 Protecting the memory (randomization, canaries)
8.9.5 Code Inspection (taint analysis)
8.9.6 Testing
8.9.7 The principle of Least Privilege
8.9.8 Keeping up to date (Patching)

- As illustrated in previous examples, many are bugs due to programmer ignorance.

- In general, it is difficult to analyze a program to ensure that it is bug-free (the “halting-problem”). There isn’t a “fool-proof” method.

8.9.1 Input Validation, Filtering, Parameterized Queries (SQL)

- In almost all examples (except TOCTOU) we have seen, the attack is carried out by feeding carefully crafted input to the system. Those malicious input does not follow the "expected" format. For example, the input is too long, contains control characters, contains negative number, etc.

- Hence, a preventive measure is to perform *input validation* whenever an input is obtained from the user. If the input is not of the expected format, reject it. There are generally two ways of filtering:

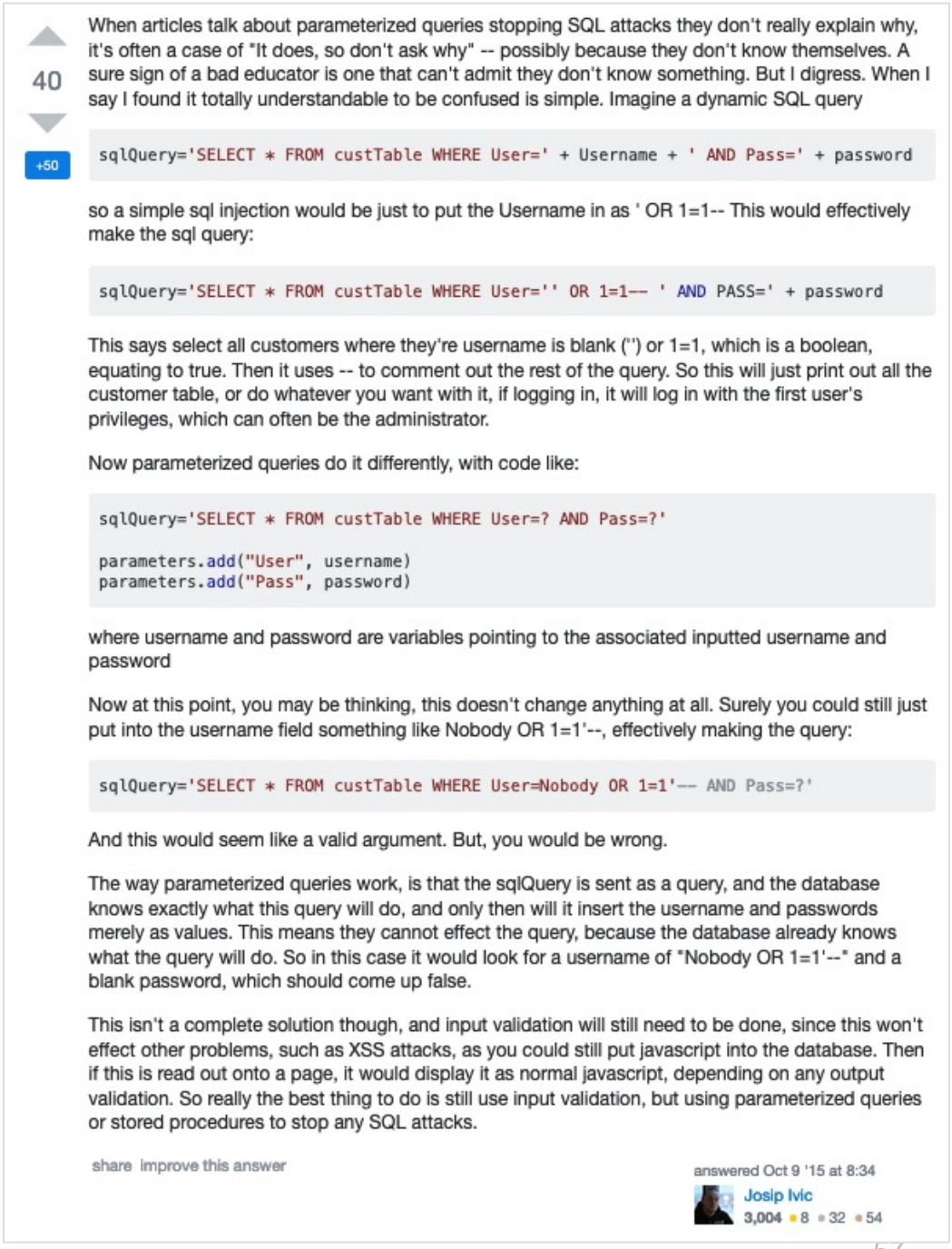
  1. White list: A list of items that are known to be benign and allowed to pass. The white list could be expressed using regular expression. However, some legitimate input may be blocked. Also, there is still no assurance that all malicious input would be blocked.
  2. Black list: A list of items that 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.

- It is difficult to ensure that the filtering is "***complete***" (i.e. doesn't miss out any malicious string), as illustrated in the example on UTF. In that example on UTF, the input validation intend to detect the substring "..%". Unfortunately, there are multiple representation of that "..%" that the programmer is not aware of.

- It is difficult to design a filter that is
  - *complete;* and
  - *accepts all legitimate* inputs.

Parameterized Queries

- Parameterized queries are mechanisms introduced in some SQL servers to protect against SQL injection. Here, queries sent to the SQL are explicitly divided to two types: the queries, and the parameters.
- The SQL parser will know that the parameters are “data” and are not “script”. The SQL parser is designed in such a way that it would never execute any script in the parameters. This check will prevent most injection attack.
- The stack overflow post gives a good explanation of parameterized queries.
- *(optional) Will this stop all scripting attacks? No. E.g. it can’t stop XSS.*



40

+50

When articles talk about parameterized queries stopping SQL attacks they don't really explain why, it's often a case of "It does, so don't ask why" -- possibly because they don't know themselves. A sure sign of a bad educator is one that can't admit they don't know something. But I digress. When I say I found it totally understandable to be confused is simple. Imagine a dynamic SQL query

```
sqlQuery='SELECT * FROM custTable WHERE User=' + Username + ' AND Pass=' + password
```

so a simple sql injection would be just to put the Username in as ' OR 1=1-- This would effectively make the sql query:

```
sqlQuery='SELECT * FROM custTable WHERE User='' OR 1=1-- ' AND PASS=' + password
```

This says select all customers where they're username is blank ('') or 1=1, which is a boolean, equating to true. Then it uses -- to comment out the rest of the query. So this will just print out all the customer table, or do whatever you want with it, if logging in, it will log in with the first user's privileges, which can often be the administrator.

Now parameterized queries do it differently, with code like:

```
sqlQuery='SELECT * FROM custTable WHERE User=? AND Pass=?'

parameters.add("User", username)
parameters.add("Pass", password)
```

where username and password are variables pointing to the associated inputted username and password

Now at this point, you may be thinking, this doesn't change anything at all. Surely you could still just put into the username field something like Nobody OR 1=1'--, effectively making the query:

```
sqlQuery='SELECT * FROM custTable WHERE User=Nobody OR 1=1'-- AND Pass=?'
```

And this would seem like a valid argument. But, you would be wrong.

The way parameterized queries work, is that the sqlQuery is sent as a query, and the database knows exactly what this query will do, and only then will it insert the username and passwords merely as values. This means they cannot effect the query, because the database already knows what the query will do. So in this case it would look for a username of "Nobody OR 1=1'--" and a blank password, which should come up false.

This isn't a complete solution though, and input validation will still need to be done, since this won't effect other problems, such as XSS attacks, as you could still put javascript into the database. Then if this is read out onto a page, it would display it as normal javascript, depending on any output validation. So really the best thing to do is still use input validation, but using parameterized queries or stored procedures to stop any SQL attacks.

share improve this answer

answered Oct 9 '15 at 8:34

Josip Ivic

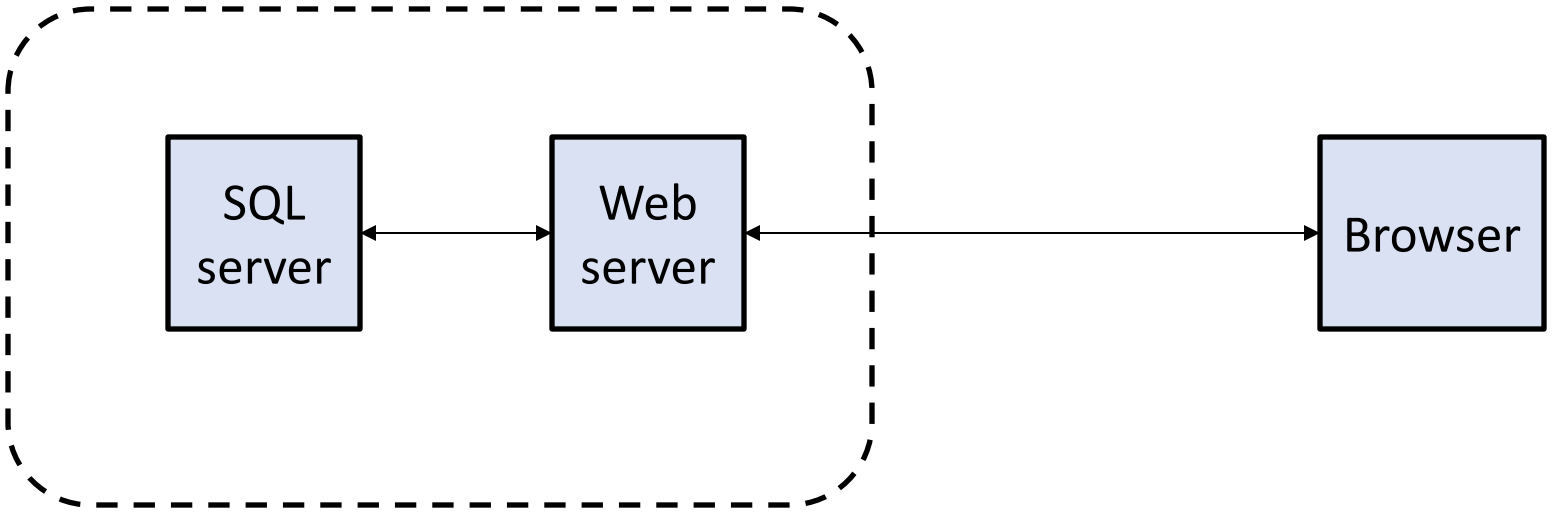
3,004 ● 8 ● 32 ● 54

E.g. Where should input validation apply?

- At Browser, Web server or SQL server?

Definitely not browser. Normally In web server instead of SQL server, because SQL server is not aware which sub-strings of the script are the "input" and thus unable to check.

(note: parameterized queries in some sense tags substrings given by user as "input". Hence the SQL server can differentiate "data" from "script" and ensure that data are not parsed as script)



8.9.2 Use “safe” function

- Completely avoid functions that are known to create problems. Use the “safe” version of the functions.

C and C++ have many of those:

strcpy ←→ strncpy

printf(f)

access()

Again, even if they are avoided, there could still be vulnerability.
(e.g. the example that uses a combination of `strlen()` and `strncpy()`)

.9.3 Bound checks and type safety

Bound checks

Some programming languages (e.g. Java, Pascal) perform bound checking during runtime (i.e. while the program is executing). During runtime, when an array is instantiated, its upper and lower bounds will be stored in some memory, and whenever a reference to an array location is made, the index (subscript) is checked against the upper and lower bound. Hence, for a simple assignment like

a[i] = 5;

what actually being carried out are:

(1) if i < lower bound, then halts;

(2) if i > upper bound, then halts;

(3) assigns 5 to the location.

If the check fails, the process will be halted (or exception to be thrown, as in Java).

The first 2 steps reduce efficiency, but will prevent buffer overflow.

The infamous C, C++ do not perform bound check.

Many of the known vulnerabilities are due to buffer overflow that can be prevented by the simple bound check.

(goto http://cve.mitre.org/cve/ to see how many entries contains "buffer overflow" as keywords). (http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=buffer+overflow)

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.

  e.g. 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 during runtime (i.e. *dynamic* type check), or when the program is being compiled (i.e. *static* type check).

- Bound checking can also be considered as one mechanism that ensures “type safety”.

e.g. in Pascal,

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

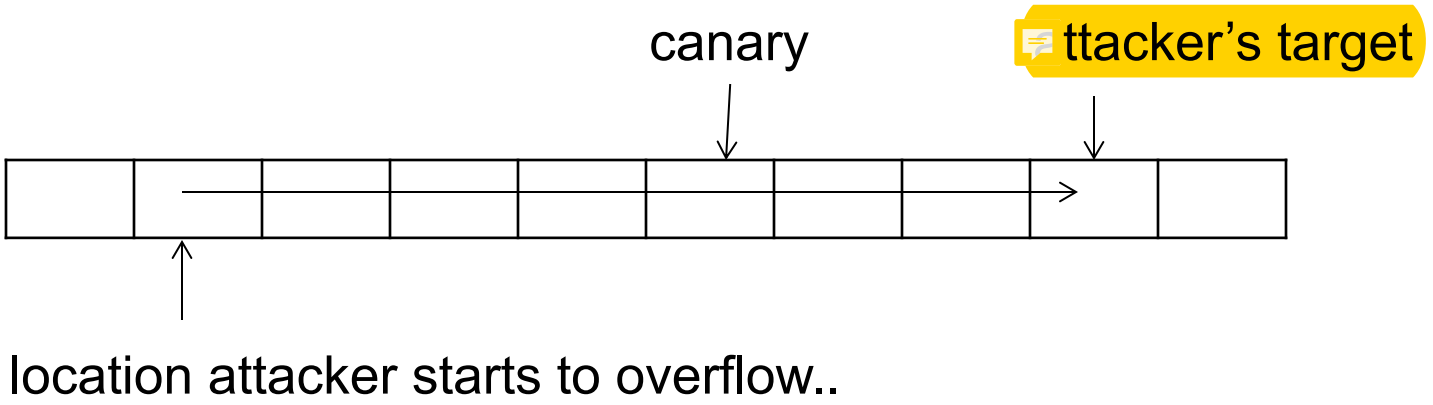
wrong type.

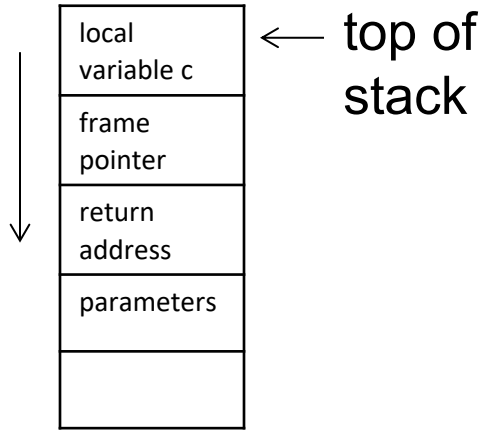
7.9.4 Canaries and Memory protection

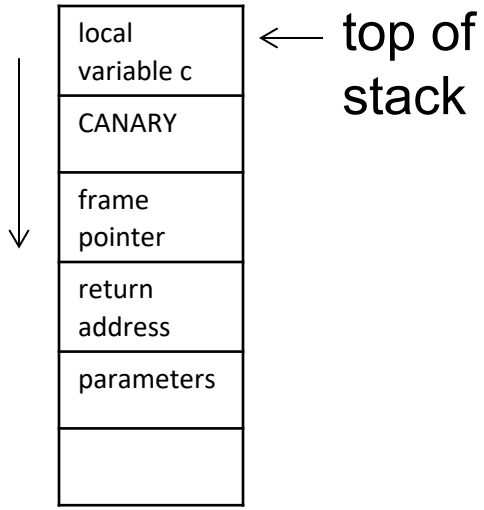
Canaries

- Canaries are secrets inserted at carefully selected memory locations during runtime. Checks are carried out during runtime to make sure that the values are not being modified. If so, halts.

- Canaries can help to detect overflow, especially stack overflow. This is because in a typical buffer overflow, consecutive memory locations have to be over-ran. If the attacker wants to write to a particular memory location via buffer overflow, the canaries would be modified. *(It is important to keep the value as "secret". If the attacker happens to know the value, it may able to write the secret value to the canary while over-running it).*







This command turns off canary. (by default it is on)

```
>  gcc myprogram.c  -fno-stack-protector
```

Question: What is the disadvantage of having the canary protection?

Question: Consider a C program `t.c` compiled using gcc and to be run in a particular OS, say ubuntu. Who should implement the canary protection?

programmer of `t.c`, person who write gcc compiler, or the person who wrote OS?

Memory randomization

- It is to the attacker's advantage when the data and codes are always stored in the same locations. *Address space layout randomization (ASLR)* (details omitted) can help to decrease the attackers chance of success.

8.9.5 Code Inspection

- Manual Checking: Manually checking the program. Certainly tedious.

- Automated Checking: Some automations are possible. For example, *taint analysis:*

  Variables that contain input from the (potential malicious) users are labeled as *source.* Critical functions are labeled as *sink.* Taint analysis checks whether the sink's arguments could potentially be affected (i.e. tainted) by the source. If so, special check(for e.g. manual inspection) would be carried out. The taint analysis can be static (i.e. checking the code without "tracing it"), or dynamic (i.e. run the code with some input).

  E.g.

  Sources: user input

  Sink: opening of system files, function that evaluates a SQL command, etc

8.9.6 Testing

Vulnerability can be discovered via testing.

White-box testing: The tester has access to the source code.

Black-box testing: The tester does not has access to the source code.

Grey-box testing: A combination of the above.

Security testing attempts to discover intentional attack, and hence would test for inputs that are rarely occurred under normal circumstances. (for e.g. very long names, or names that contain numeric values.)

Fuzzing is a technique that sends malformed inputs to discover vulnerability. There are techniques that are more effective than sending in random input. (detail not required). Fuzzing can be automated or semi-automated.

Terminology: White list vs Black list, White-box testing vs Black-box testing
White hat vs Black hat.

8.9.7 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 options unnecessary.

E.g.: Software contain many features. (For 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 "harden" mean?

8.9.8 Patching

Life cycle of vulnerability:

vulnerability is discovered → affected code is fixed → the revised version is tested → a patch is made public → patch is applied.

In some cases, the vulnerability could be announced without the technical details before a patch is released. The vulnerability likely to be known to only a small number of attackers (even none) before it is announced.

When a patch is released, the patch can be useful to the attackers. The attackers can inspect the patch and derive the vulnerability.

Hence, interestingly, the number of successful attacks goes up after the vulnerability/patch is announced, since more attackers would be aware of the exploit. (see next slide)

Terminology: zero-day vulnerability

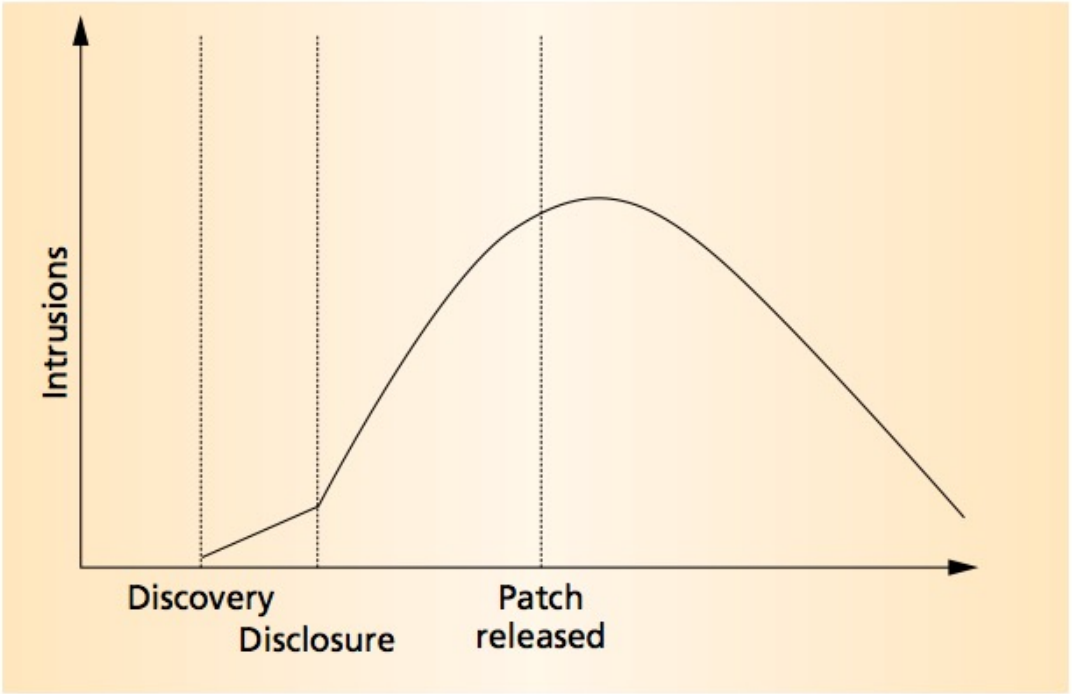


Figure 1. Intuitive life cycle of a system-security vulnerability. Intrusions increase once users discover a vulnerability, and the rate continues to increase until the system administrator releases a patch or workaround.

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

It is crucial to apply the patch timely. Although seems easy, applying patches is not that straightforward. For critical system, it is not wise to apply the patch immediately before rigorous testing. Patches might affect the applications, and thus affect an organization operation. (imagine a scenario where after a patch is applied, a software that manages critical infrastructure crash. Or a scenario where after a student applied a patch on browser, the student can't upload report to IVLE workbin and thus missed the project deadline).

"Patch Management" is a field of study. See the guide by NIST on patch management.

see Guide to Enterprise Patch Management Technologies, 2013.

http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-40r3.pdf