Software Security

Overview of Software Security

Requirements of a program:

- Correct
- Efficient
- Secure

But oftentimes, a program behaves beyond its intended functionality!

Security Problems, Examples and Root Causes

Security problems faced:

- Insecure Implementation
    - Many programs are not implemented properly, allowing the attacker i.e. the person who is invoking 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
        - Deviate from the original intended execution path
    - Regardless, the attacker is using the program to **elevate their privilege.**

Examples:

- 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 such that unlicensed software can run without the need for hardware modifications
        - Xbox
        - PlayStation 2 (PS2 Independence Exploit)
        - Wii (Twilight Hack)
- SQL Injection
    - Yahoo! (2012)
        - 450,000 login credentials claimed to be stolen using a "union-based SQL injection technique"
    - British Telco TalkTalk (2015)
        - A vulnerability in a legacy web portal was exploited to steal the personal details of 156959 customers
- Integer Overflow
    - European Space Agency's Ariane 5 Rocket (1996)
        - 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:

- Functionality still remains the primary concern during design and implementation
    - Security is the secondary goal
    - Features pay the bills

- Unavoidable human mistakes
  - Lack of awareness of security problems
  - Careless programmers
- Complex modern computing systems
  - Many of the bugs are simple and seem easy to prevent
  - But programs for complex systems are large
    - Windows XP has 45 million source line of codes
  - This results in a large attack surface
- Learn Programming Fast! guide books
  - Guide books may be enough for learning basic functionality, but never enough for learning the subtle implications of the various functionalities
  - Result: Programs can do more than you expect!

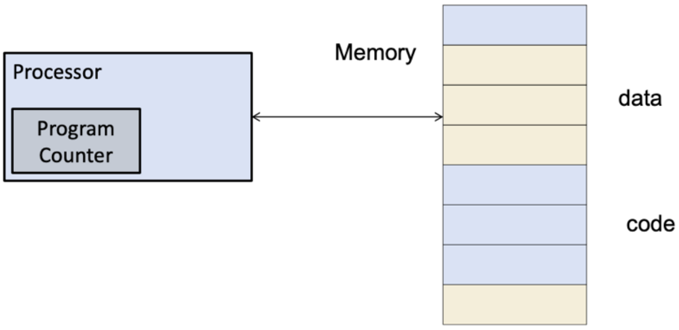
Computer Architecture

Code vs Data

Modern computers are based on the Von Neumann computer architecture. This means that:

- Code and data are stored together in the memory
- There is no clear distinction between code and data
- This is in contrast to the Harvard architecture, which has hardware components that separately store code and data
- **Implication: Programs may be tricked into treating input data as code!**
  - Basis for all code-injection attacks

Control Flow and Program Counter



The processor, as the name suggests, is the electronic circuitry within a computer that carries out the instructions of a computer program by performing the basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions.

These instructions are read with the help of a program counter, or instruction pointer/counter. The program counter is a processor register that indicates where a computer is in its program sequence.

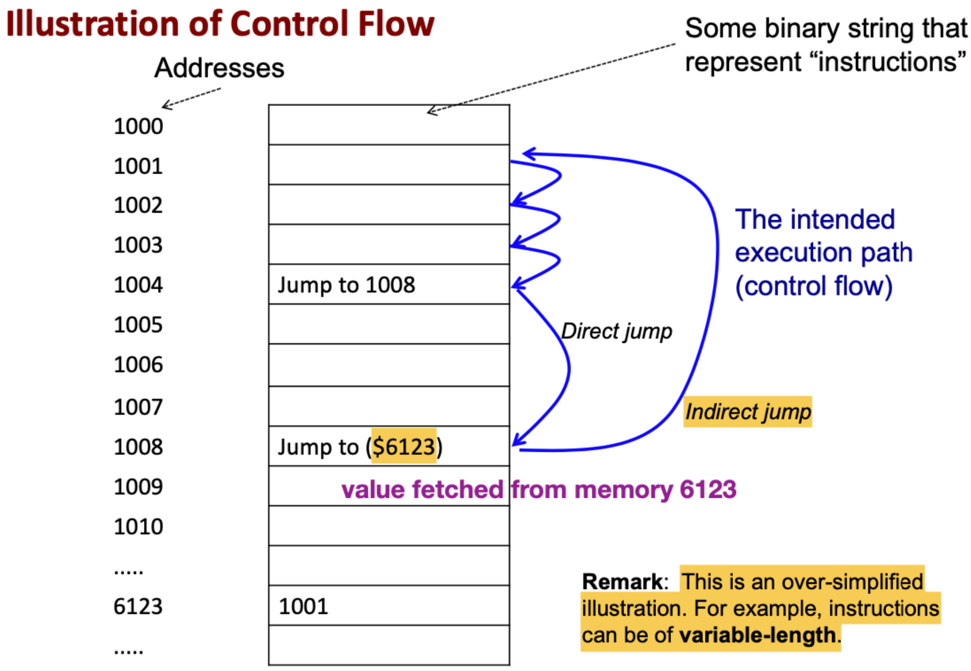
Usually, it holds the memory address of (or “points to”) the next instruction that would be executed. After an instruction is completed, the processor fetches the next instruction from the address stored in the program counter. Once this fetching is done, the program counter automatically increases by 1 (assuming a system with fixed length-instructions).

Changes to Program Counter

Besides getting incremented, the program counter can also be changed, for example by:

- Direct Jump
  - Replaced with a constant value specified in the instruction
- Indirect Jump
  - Replaced with a value fetched from memory
  - There are many different forms of indirect jump

Illustration of Control Flow



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

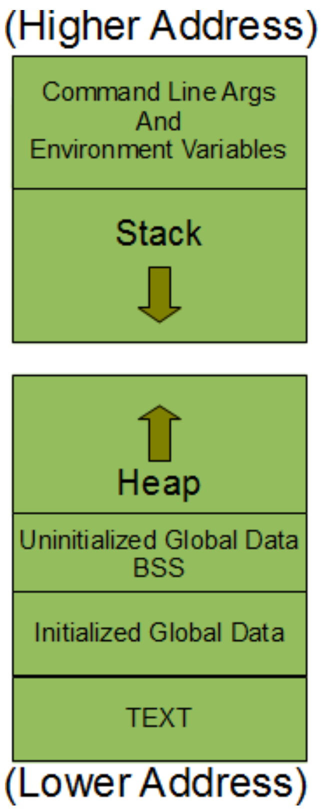
Stack
Also called Execution Stack or Call Stack

Functions
Functions break code into smaller pieces and facilitate modular design and code reuse. A function can be called in many different locations, and can be called many times e.g. recursive function. How does the program know where it should continue from after the function finishes, and where are the function's arguments and local variables stored?

Process Memory Layout

As we can see in the image to the right, the Linux process memory contains the following:

- Command Line Arguments and Environment Variables
  - Stored at the top of the process memory layout at the higher addresses
- Stack
  - Memory area used by the process to store local variables of functions and other information that is saved every time a function is called. More about this later.
  - Grows downwards
- Heap
  - Used for dynamic memory allocation
  - Shared amongst all threads of a single process, but not between different processes running in the system
  - Memory from this segment should be used cautiously and should be deallocated as soon as the process is done using that memory
  - Grows upwards
- Uninitialized Global Variables
  - They are initialized with the value zero


(Higher Address)
Command Line Args And Environment Variables
Stack
Heap
Uninitialized Global Data BSS
Initialized Global Data
TEXT
(Lower Address)

    - BSS stands for “Block Started by Symbol”.
- Initialized Global Variables
- Text
    - Memory area that contains the machine instructions that the CPU executes
    - Shared across different instances of the same program being executed
    - Since there is no point in changing the CPU instructions, this segment has read-only privileges.

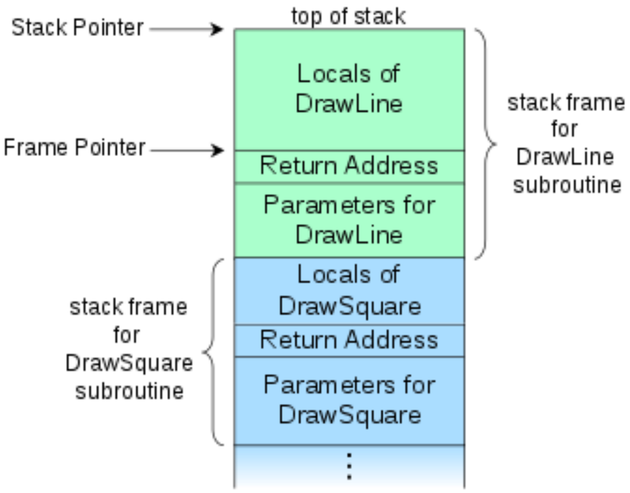
Call Stack
The Call Stack is a data structure in the memory that stores important information of a running process. As with a stack, it is LIFO or FILO, with push(), pop() and top() operations.

The location of the top element is referred to by the stack pointer. Generally, in this mod, we talk about stacks growing upwards, but from higher addresses to lower addresses. Some diagrams have the higher addresses above, and have the stack growing “downwards”, but it’s still the same.

Architectures do differ as to whether call stacks grow towards higher addresses or towards lower addresses. What is important in drawing a stack diagram is that the placement of the top, and so the direction of stack growth, can be easily understood. The logic of the diagram is independent of the addressing choice.

The stack helps to keep track of:
- Control flow information, such as return addresses
- Parameters passed to functions
- Local variables of functions

Each call of a function pushes an activation record, or **stack frame**, to the stack.



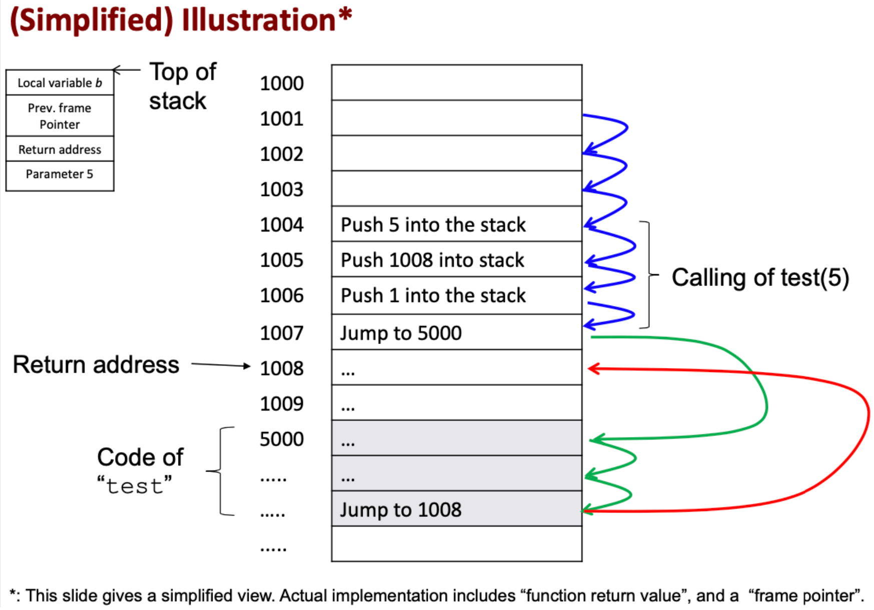
The stack frame at the top of the stack is for the currently executing routine. The stack frame usually includes at least the following items, in push order:
- Passed parameters
- Return address back to the routine’s caller e.g. in the DrawLine stack frame, an address into DrawSquare’s code
- Previous frame pointer or pointer to the previous stack frame
- Space for the local variables of the routine, if any

What is the frame pointer? Frame pointer is a pointer within the stack frame that is unchanging. Unlike the stack pointer and the top of the stack that changes as local values are pushed on to and popped off of the stack, the frame pointer can allow for easier access to variables via an offset. This is also called the base address or ebp. The frame pointer will need to be pushed into the stack (as the pointer to the previous stack frame) and restored when the current function calls other functions as well.

This is very confusing stuff, but for now, this understanding is sufficient.

Example:
Consider a function test that takes in parameter int a and declares an int b = 1 within. When test(5) is invoked, the following happens:

- Stack pushing:
  - Parameter 5 is pushed into the stack
  - Return address is pushed into the stack
  - Previous frame pointer is pushed into the stack
  - Local variable b and its value 1 is pushed into the stack
- Control flow jumps into the code of test
- Execute test
- After test is completed, the stack frame is popped from the stack
- Control flow jumps into the restored return address.



(Simplified) Illustration*

1000	
1001	
1002	
1003	
1004	Push 5 into the stack
1005	Push 1008 into stack
1006	Push 1 into the stack
1007	Jump to 5000
1008	...
1009	...
5000	...
.....	...
.....	Jump to 1008
.....	

*: This slide gives a simplified view. Actual implementation includes "function return value", and a "frame pointer".

Control Flow Integrity
The call stack stores a return address, the location of a to-be-executed instruction, as data in the memory. In fact, the instruction itself is also stored as data in the memory. This allows for greater flexibility, but also many security issues.

An attacker could compromise a process' execution integrity by either modifying the code or the control flow. It is difficult for the system to distinguish these malicious pieces of code from benign data.

Memory Integrity
In general, it is not easy for an attacker to compromise memory integrity, i.e. modify data in the memory. One way an attacker can do that is to exploit 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:

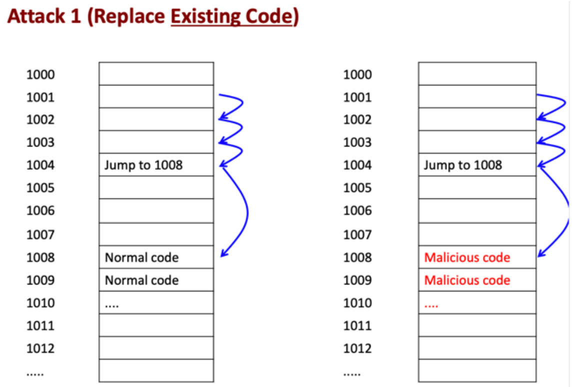
- The attacker can only write to a small amount of memory
- Or they can only write a sequence of consecutive bytes

Hence the attack needs to be very precise and surgical.

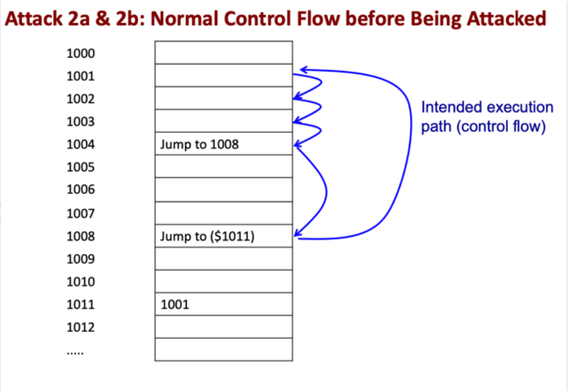
Possible Attack Mechanisms

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

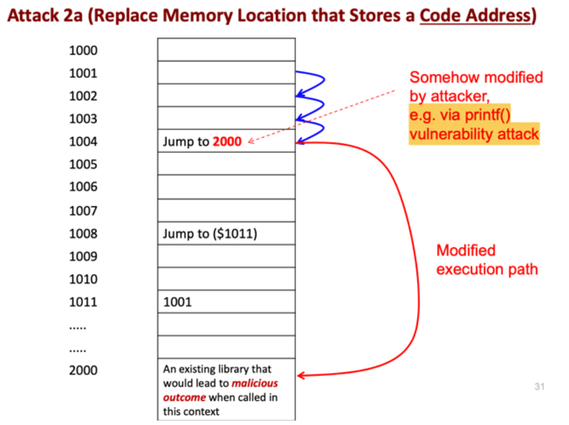
1. Overwrite existing execution code portion with malicious code
2. Overwrite a piece of control-flow information:
    a. Replace a memory location storing a code address that is used by a **direct jump**
    b. Replace a memory location storing a code address that is used by an **indirect jump**



Attack 1 (Replace Existing Code)

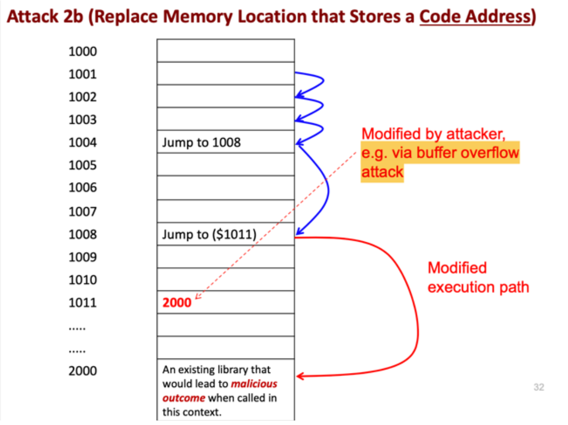


Attack 2a & 2b: Normal Control Flow before Being Attacked



Attack 2a (Replace Memory Location that Stores a Code Address)

1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
.....
.....
2000
Jump to 2000
Jump to ($1011)
1001
An existing library that would lead to malicious outcome when called in this context
Somehow modified by attacker, e.g. via printf() vulnerability attack
Modified execution path
31



Attack 2b (Replace Memory Location that Stores a Code Address)

1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
.....
.....
2000
Jump to 1008
Jump to ($1011)
2000
An existing library that would lead to malicious outcome when called in this context.
Modified by attacker, e.g. via buffer overflow attack
Modified execution path
32

Attacks and Vulnerabilities

Printf() and Format String Vulnerability

printf() is the C function for formatting output. It is special in that it can take in **any** number of arguments.

The general format is as such:

```
int printf(const char* format, …)
```

where … is any amount of arguments that would be printed based on how the format string is written.

Format Specifiers

In the format string, there would be format specifiers to indicate the type of the variable to be printed. For example,

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

would cause printf() to print "1st string is ", look up for the second parameter in the call stack (which is s1) and print it, then "2nd string is ", and look up for the third parameter in the call stack and print it.

Note that the first parameter is the format string itself.

Some common format specifiers:

- %d: decimal (int)
- %u: unsigned decimal (unsigned int)
- %x: hexadecimal (unsigned int)
- %s: string ((const) (unsigned) char *)
- %n: number of bytes written so far, (* int)

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

Missing Arguments

When only one parameter is supplied, only that parameter will be displayed.

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

That is fine if the parameter is a string by itself, but if it has a format specifier within, printf() will actually search for the second parameter in the stack to be displayed.

This is especially the case when the format string variable to be printed is supplied by the user. The attacker can actually get information by carefully designing the string to be printed.

For example, the attacker can:

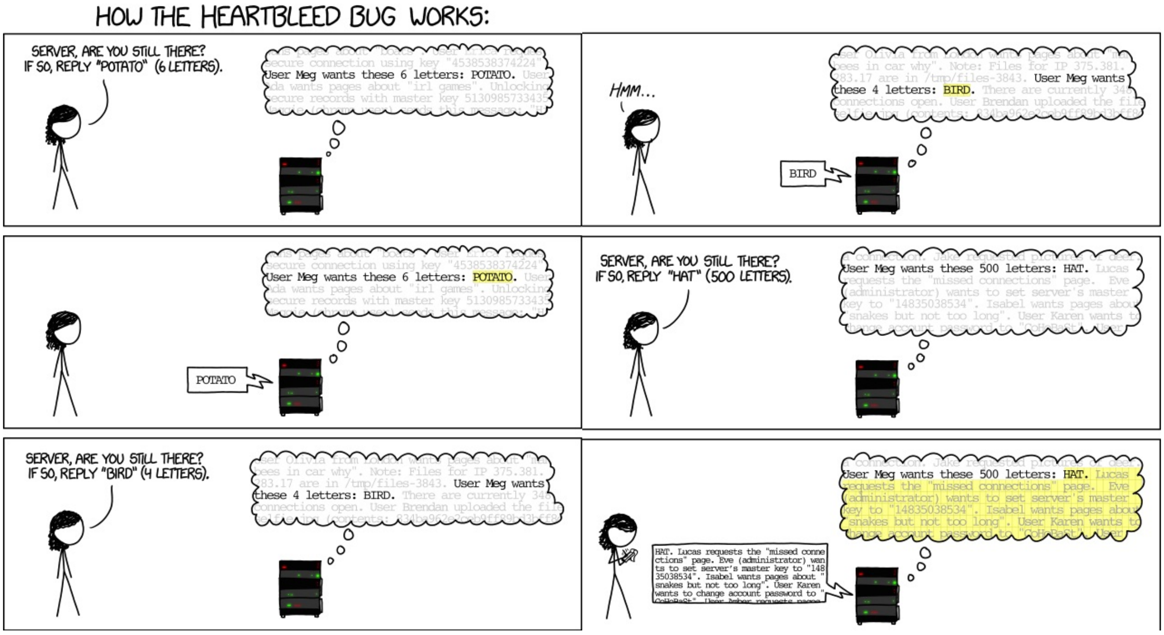
- Obtain more information of the program's call stack
  - Using "%d.%d.%d" to fetch values from the top of the stack and print them out as decimal values
    - Sample output: 73896.0.269401708
  - Using "%08x.%08x.%08x" to fetch values from the top of the stack and print them out as lower-case hexadecimal, with 8 digits shown (from the 8), and with padding/prefixes of 0s (from the 0)
    - Sample output: 000120a8.00000000.08309e6c
    - The 000120a8 and 00000000 actually corresponds to the first two decimals printed before
- Cause the program to crash
  - "%s" will fetch a number from the stack and treat this number as an **address**.
  - The printf() function will try to print out the memory contents pointed by this address as a string, until a null character is read.

    - Typically, a randomly fetched number is not an address, thus the memory pointed at by this number does not exist.
    - The program will thus crash
- Modify the program's memory content, such as by using %n. This is not covered in this module.

Big Picture Exploitation

This vulnerability can be exploited:

- In a multi-user setting
    - If the program has an elevated privilege i.e. via set-UID, an attacker may be able to obtain **system-level information**
- In a client-server setting
    - If the server program is vulnerable, the attacker may be able to submit a request and obtain sensitive information e.g. a secret key
    - This is how the **Heartbleed** bug works, where an Over-Read Request allows the client to receive information that's on the server


HOW THE HEARTBLEED BUG WORKS:
SERVER, ARE YOU STILL THERE? IF SO, REPLY "POTATO" (6 LETTERS).
User Meg wants these 6 letters: POTATO.
User Meg wants these 6 letters: POTATO.
POTATO
SERVER, ARE YOU STILL THERE? IF SO, REPLY "BIRD" (4 LETTERS).
User Meg wants these 4 letters: BIRD.
HMM...
User Meg wants these 4 letters: BIRD.
BIRD
SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHoBaSt". User Amber requests pages

Preventive Measures

Avoid taking a user input as a format string. We can read the input into a separate variable, then print out that variable via our own format string. For example:

- printf(t)
    - Where t is supplied by the user, thus it is potentially insecure
- printf(f, t)
    - Where f is not supplied by the user, thus it is generally okay

Many modern compilers also statically check format strings and produce warnings for dangerous or suspect formats.

In GNU Compiler Collection, the relevant compiler flags are:

- `-Wall`
    - This enables all the warnings about constructions that some users consider questionable, and that are easy to avoid (or modify to prevent the warning).
- `-Wformat`

    - Check calls to printf and scanf, etc., to make sure that the arguments supplied have types appropriate to the format string specified, and that the conversions specified in the format string make sense.
- `-Wno-format-extra-args`
    - If -Wformat is specified, do not warn about excess arguments to a printf or scanf format function. The C standard specifies that such arguments are ignored.
- `-Wformat-security`
    - If -Wformat is specified, also warn about uses of format functions that represent possible security problems. At present, this warns about calls to printf and scanf functions where the format string is not a string literal and there are no format arguments, as in `printf(foo);`.
- `-Wformat-nonliteral`
    - If -Wformat is specified, also warn if the format string is not a string literal and so cannot be checked, unless the format function takes its format arguments as a va_list.
- `-Wformat=2`
    - Enable -Wformat plus additional format checks. Currently equivalent to -Wformat -Wformat-nonliteral -Wformat-security -Wformat-y2k.

Data Representation & Security

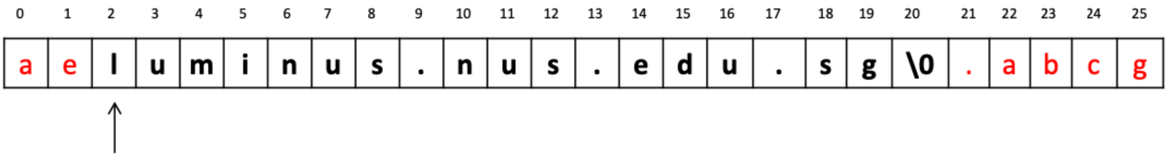
Different parts of a program or system may adopt different data representations. Such inconsistencies could lead to vulnerabilities.

For example, CVE-2013-4073: “Ruby’s SSL client implements hostname identity check, but it does not properly handle hostnames in the certificate that contain null bytes.”

One important data representation type is string.

String

In C, printf() adopts a efficient representation, where the length is not explicitly stored, and the first occurrence of the null character i.e. byte with value 0, indicates the end of the string, thus implicitly providing the length.



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

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

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

Exploitable Vulnerability 1: NULL-Byte Injection

A Certificate Authority **may accept** a hostname containing a null character, e.g. luminus.nus.edu.sg\0.attacker.com

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

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

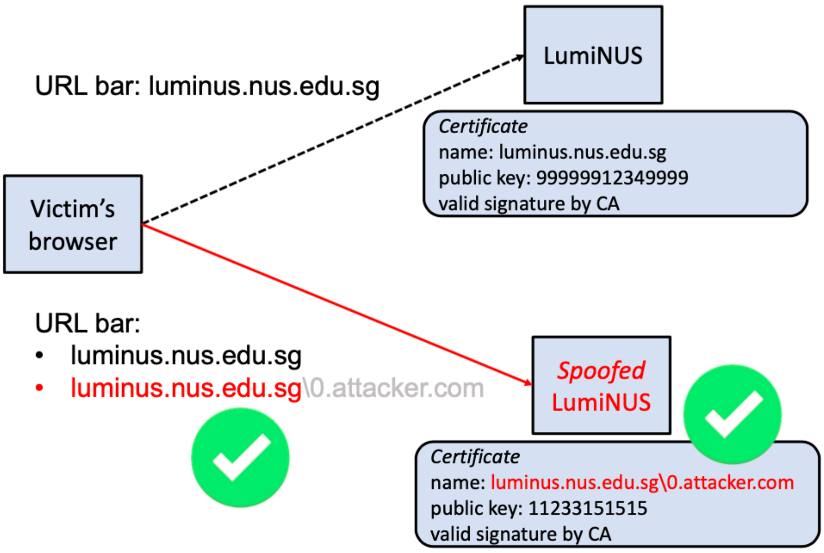
We can thus have this scenario:

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

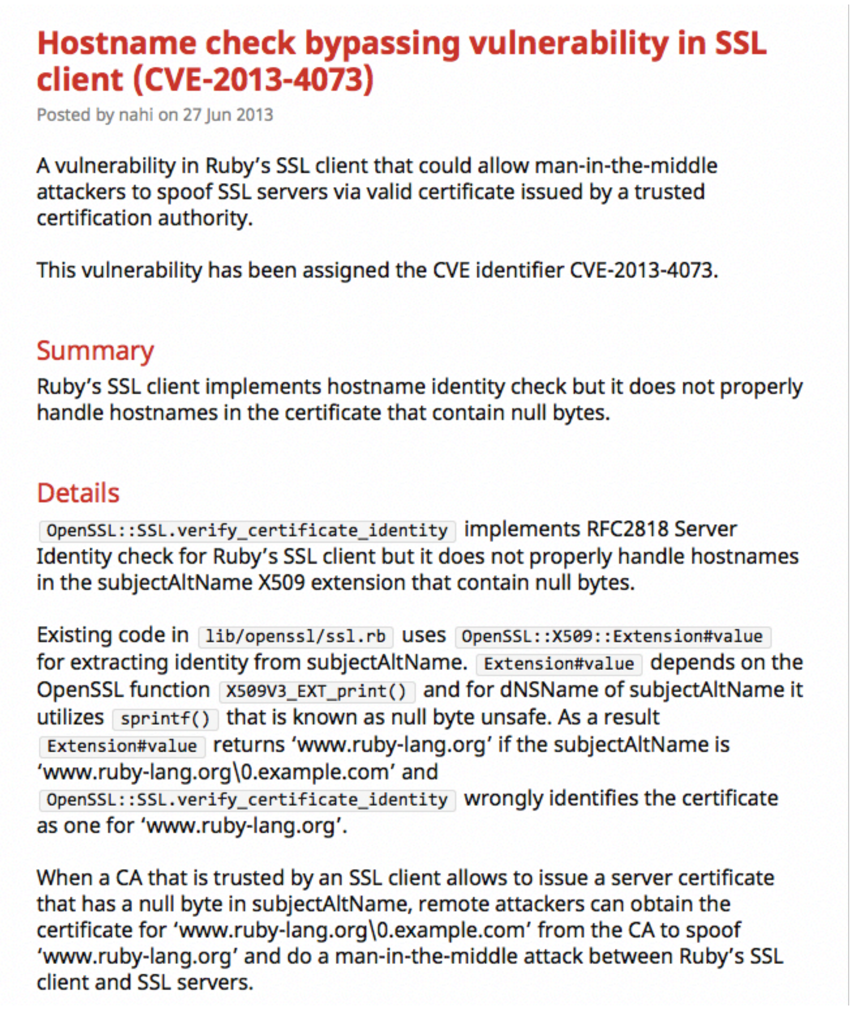
This is more effective than the normal web-spoofing attack, as a careful user would notice that the address displayed in the address bar is not LumiNUS, or that the address bar displays luminus.nus.edu.sg but the TLS/SSL authentication protocol rejects the connection i.e. certificate is not trusted.

Thus this attack is much more dangerous.

Below is a slide showing a summary:



Below are some details on the CVE-2013-4073. CVE stands for Common Vulnerabilities and 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.

Encoding: ASCII and UTF-8

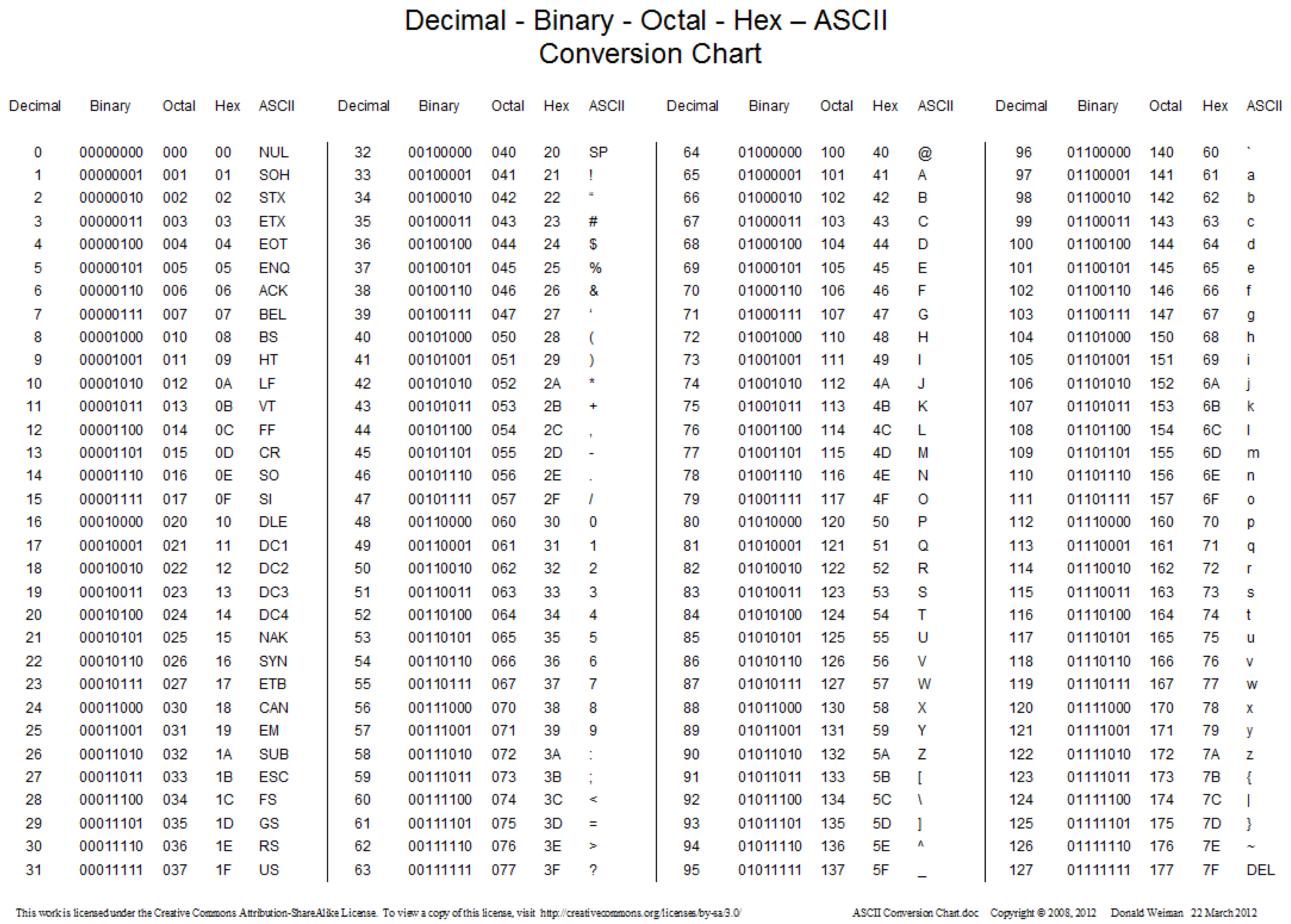
American Standard Code for Information Interchange (ASCII)

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

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

- The standard 7-bit ASCII characters

- Additional characters

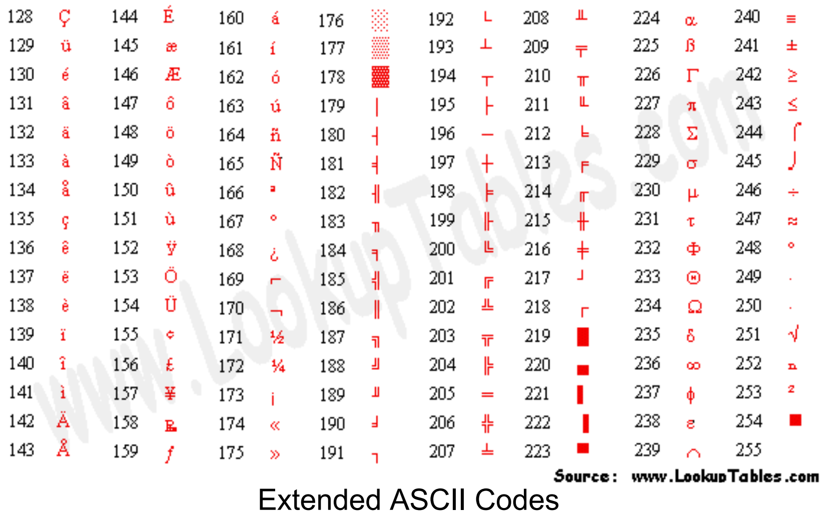


Decimal - Binary - Octal - Hex – ASCII
Conversion Chart

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

ASCII Conversion Chart.doc Copyright © 2008, 2012 Donald Weiman 22 March 2012

Standard ASCII characters



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

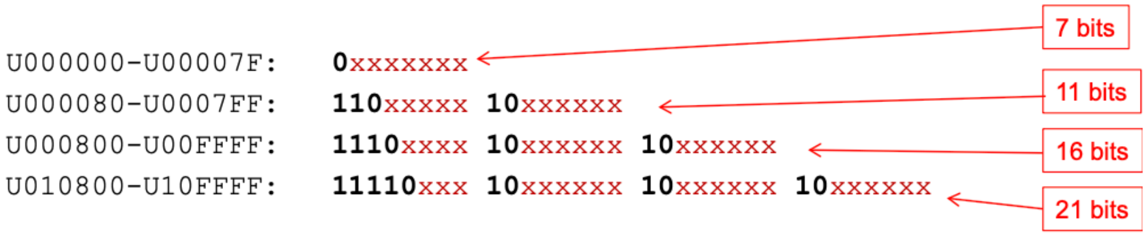
Source : www.LookupTables.com

Extended ASCII Codes

Unicode Transformation Format 8-bit (UTF-8)
UTF-8 is a character encoding that is capable of encoding all 1,112,064 valid code points in Unicode using one to four 8-bit bytes. It is a variable-length encoding, where code points with higher frequency of occurring are encoded with lower numerical values, thus using fewer byes.

The first 128 characters of Unicode correspond 1-to-1 with ASCII, and is encoded using a single octet with the same binary value as ASCII. Each byte thus starts with the bit 0 followed by the 7 bits of the original ASCII bits. Hence ASCII characters remain unchanged in UTF-8. There is backward compatibility with ASCII, as UTF-8 encoding was defined for Unicode on systems that were designed for ASCII.

Exploitable Vulnerability 2: UTF-8 "Variant" Encoding Issues
A Unicode character is referred to by "U+" and its hexadecimal digits. The following are byte representations of Unicode characters, with the left-hand side being the Unicode representation, and the right-hand side being the byte representation:



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

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

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

This leads to certain issues. Consider the following example.

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

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

- (2-byte) 110**00000** 10**101111**
- (3-byte) 1110**0000** 10**000000** 10**101111**
- (4-byte) 11110**000** 10**000000** 10**000000** 10**101111**

There is potential inconsistency when doing character verification before any operations that use this character.

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

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

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

In the above example, the client can be any remote public user, i.e. similar to a HTTP client. The original intention is the limit the files that the client can retrieve to only those under the directory public_html. This is called **file-access containment.**

However, an attacker may send in this string: ../cs2107report.pdf

The server would then try to read /home/student/alice/public_html/../cs2107report.pdf, which violates the intended file-access containment. To prevent this, the server may add an input validation step, making sure that the substring "../" does not appear within the input string.

In other words, there is now a:

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

Does this work?
Let us assume that the system call in Step 2 above uses a convention that can process "%" followed by two hexadecimal digits as a single byte, similar to URL encoding, e.g. "/home/student/%61lice/" will have the %61 replaced by a to give "/home/student/alice/".

We also assume that the system call uses UTF-8.

Then the following strings will actually all be equivalent to the string "../cs2107report.pdf":

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

All these inputs, when decoded, will give the same system call as before.

In general, a blacklisting-based filtering system can be incomplete due to the "flexibility" of character encoding.

Example 2: IP Address (Representation as Strings and Integers)
The 4-byte IP address is typically written as a string, e.g. 132.127.8.16. Consider a blacklist that contains a list of banned IP addresses, where each IP address is represented as 4 bytes.

Assume that there is a function BL() (blacklist) that takes in 4 integers of type int (i.e. 32-bits) and checks whether the IP address represented by these 4 integers is in the blacklist:
int BL(int a, int b, int c, int d)

There are thus 4 arrays of integers, named A, B, C and D, and it simply tries to find i such that A[i]==a, B[i]==b, C[i]==c, and D[i]==d.

The overall program thus does the following:

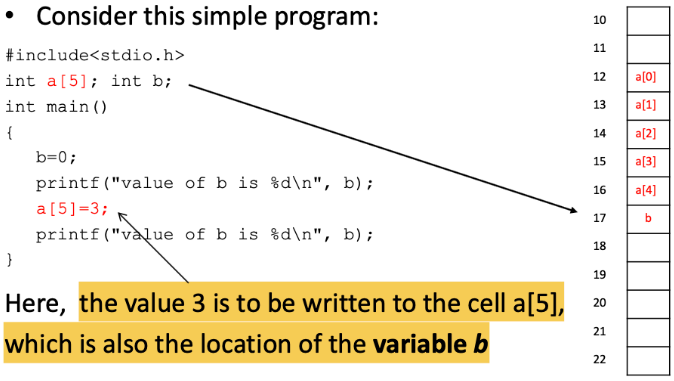
1. Get string s from user
2. Check if the s is of the correct format i.e. 4 integers separated by ".". If not, quit, else extract a, b, c and d.
3. Call BL() to check, if in blacklist, quit.
4. Let ip = a*2^{24} + b*2^{16} + c*2^{8} + d, where ip is a 32-bit **integer**
5. Continue the remaining processes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

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

How to deal with issues with Data Representation: Use Canonical Representation
The important lesson is that we cannot trust input from the user, and we can never directly use input from them. Always convert the input to a standard i.e. canonical representation immediately.

Preferably, do not rely on the verification check done in the application i.e. do not rely on the application developers to write the verification. Rather, try to make use of the underlying **system access control mechanism.**

Buffer Overflow
In languages such as C and C++, programmers themselves are able to manage the memory via pointer arithmetic. There is no array-bound checking. Such flexibility is useful but very prone to bugs, leading to vulnerabilities.



- 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***

The image above illustrates buffer overflow, or buffer overrun.

What is a buffer?
A data buffer, or just buffer, is 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 of size 5, and the location a[5] is beyond its boundary. Hence, writing to it causes a buffer overflow.

A well-known function in C that is prone to buffer overflow is a string copying function: strcpy().

strcpy()
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 be potentially 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 the buffer size of s1 is only 10, the extra values will be **overflowed** and written to other parts of the memory.

If s2 is supplied by a **malicious** user, a well-crafted input can overwrite important memory and modify the computation!

As such, avoid using strcpy(), and use strncpy() instead. The function is as such:

`strncpy(s1, s2, n)`

where at most n characters are copied. There are still vulnerabilities if used improperly, it is merely **more secure.**

Stack Smashing

Stack smashing is a special case of buffer overflow that targets a process' call stack. As we've covered before, when a function is invoked, the return address will be pushed into the stack. If the stack is being overflowed such that the return address is modified, then the execution's control flow is changed.

A well-designed overflow could also inject the attacker's **shellcode** into the process' memory, then execute the shellcode. If the target executable is set-UID-root, then the injected shellcode runs with root privilege.

Some defences such as canary are available, which will be covered later.

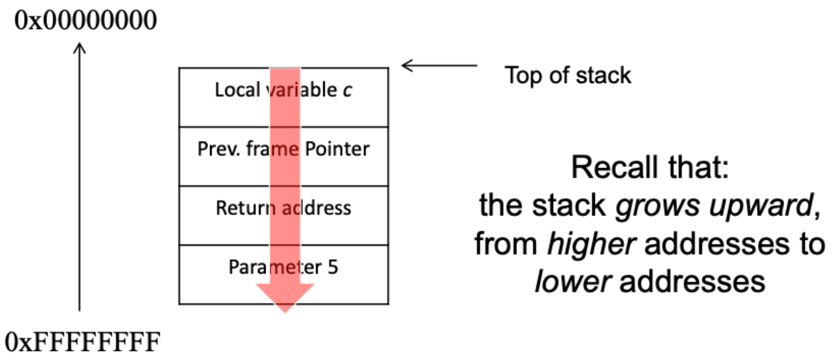
Example of Stack Smashing

Consider the following segment of a C program:

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

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

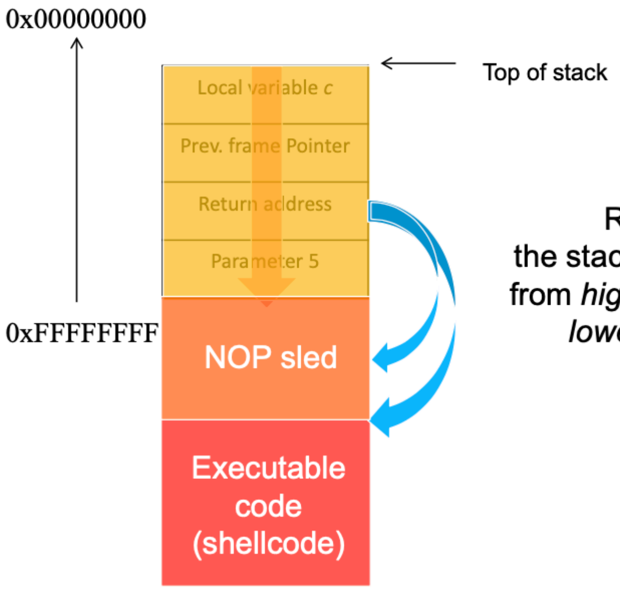
After foo(5) is invoked, a few values are pushed into the stack.



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

It is important to observe that the buffer c actually **grows towards** the return address! If an attacker manages to modify the return address, the control flow will jump to the address indicated by the attacker.

More can be seen from the diagram below:



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

What is the NOP Sled?
The NOP sled or NOP ramp is a sequence of NOP (no-operation) instructions meant to “slide” the CPU’s instruction execution flow to its final, desired destination whenever the program branches to a memory address anywhere on the slide.

This is because sometimes attackers may not be able to have the return address point to the exact location or when the target address that the control flow is looking for is not known precisely. It creates a greater area for attackers to strike and still ensure that their shellcode will run.

While a NOP slide will function if it consists of a list of canonical NOP instructions, the presence of such code is suspicious and easy to automatically detect. For this reason, practical NOP slides are often composed of non-canonical NOP instructions such as a moving a program counter/register to itself or adding zero i.e. instructions that affect program state inconsequentially, which makes them a lot more difficult to identify.

Integer Overflow
The integer arithmetic in many programming languages is actually **modulo arithmetic**. Suppose a is a single byte (i.e. 8-bit) unsigned integer.

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

Its value would now be 0, since the addition is done in the form of modulo 256. Hence, the predicate (a < a+1) is not always true! Many programmers do not realise this, leading to possible vulnerabilities.

Code/Script Injection

As mentioned earlier, a key concept in computer architecture is how code is treated as data. This is actually unsafe, since many attacks inject malicious code as data, which then gets executed by the target system.

Scripting languages are programming languages that can be interpreted by another program during runtime, instead of during compilation. Some well-known examples are JavaScript, Perl, PHP and SQL. Many scripting languages also allow the script to be modified while being interpreted, opening up the possibility of malicious code being injected into the script.

We will consider a well-known attack: Structured Query Language (SQL) Injection (SQLI) Attacks.

Structured Query Language (SQL)

SQL is a database query language. Consider a database, which can be viewed as a table, with each column being associated with an attribute.

name	account	weight
bob12367	12333	56
alice153315	4314	75
eve3141451	111	45
peter341614	312341	86

Table name: client

This query script

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

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

Sometimes, the script may also include variables. 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

In the above example, the database is designed such that the username is a secret, hence only the authentic entity who knows the name can get the record. However, if the attacker passes the following as the input:

```
Bob’ OR 1=1 --
```

The interpreter will execute the following:

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

Since 1=1 for all, the interpreter actually returns all the records! The -- causes the interpreter to ignore everything behind, including the trailing single colon.

Code Injection + Buffer Overflow

Code injection is not limited to SQL injection. It is possible to exploit buffer overflow by injecting malicious code and then transferring the process execution to the malicious code.

Undocumented Access Points

For debugging purposes, many programmers insert undocumented access points to allow them to better inspect states. For example:

- Pressing a certain combination of keys will allow the values of certain variables to be displayed
- Certain input strings would cause the program to branch to some debugging mode

Many of these access points are left in the final production system, providing backdoors to attackers. A backdoor is a covert method of bypassing normal authentication.

Such access points are also known as Easter eggs. Some of these are benign and are intentionally planted by the developer for fun or publicity. However, there are also known cases where unhappy and disgruntled programmers planted the backdoors on purpose.

The backdoor can be accessed by the programmer and any other user who knows or discovers them.

Time-of-Check Time-of-Use (TOCTOU) Race Condition

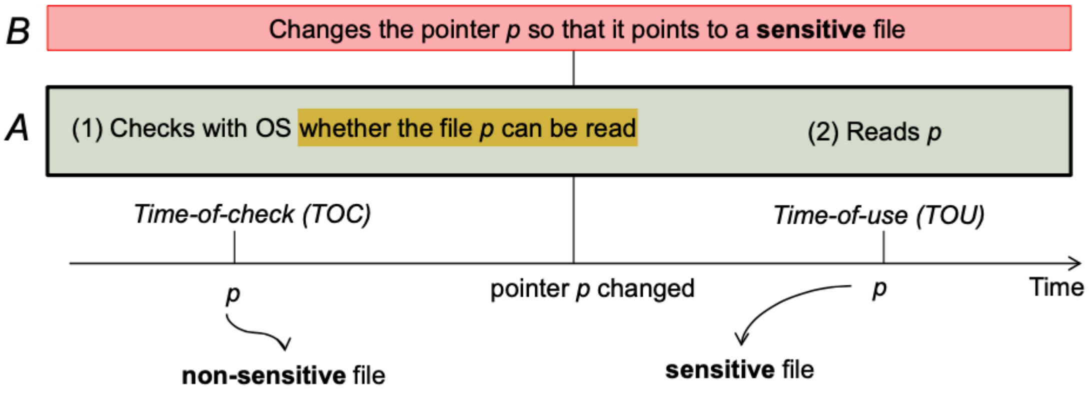
In general, a race condition occurs when multiple processes access a piece of shared data in a way that the final outcome depends on the **sequence of the accesses.**

In security, the multiple processes usually refer to:

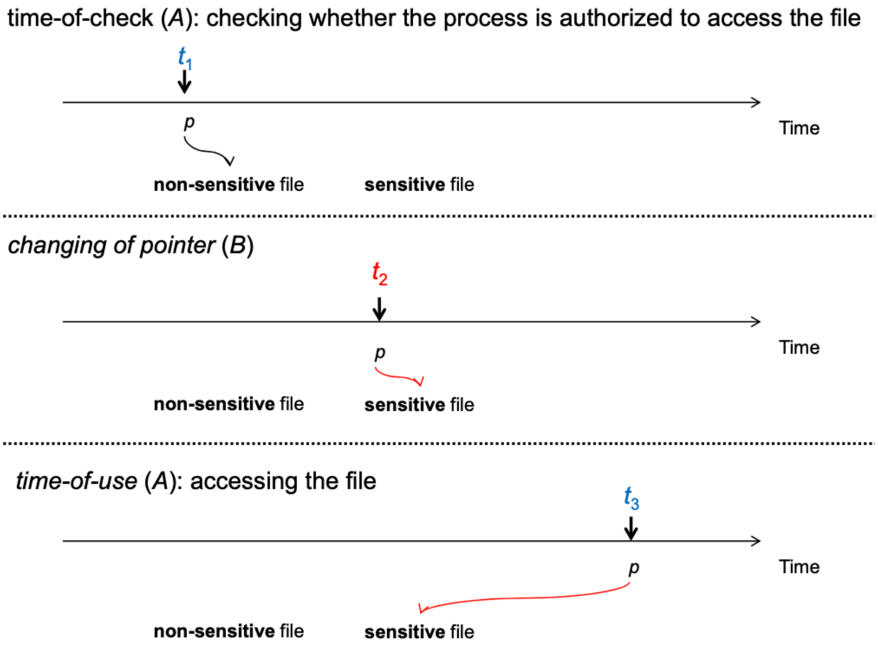
- A vulnerable process A that checks for permission to access a shared data, and subsequently accesses the data
- A malicious process B that swaps the data

These two processes can be run by a single malicious user in the system. There is thus a race between processes A and B. If B is able to complete the swapping after A checks for permission but before A accesses the data, then the attack as succeeded.

Refer to the diagram below:


B
Changes the pointer p so that it points to a sensitive file
A
(1) Checks with OS whether the file p can be read
(2) Reads p
Time-of-check (TOC)
Time-of-use (TOU)
p
pointer p changed
p
Time
non-sensitive file
sensitive file

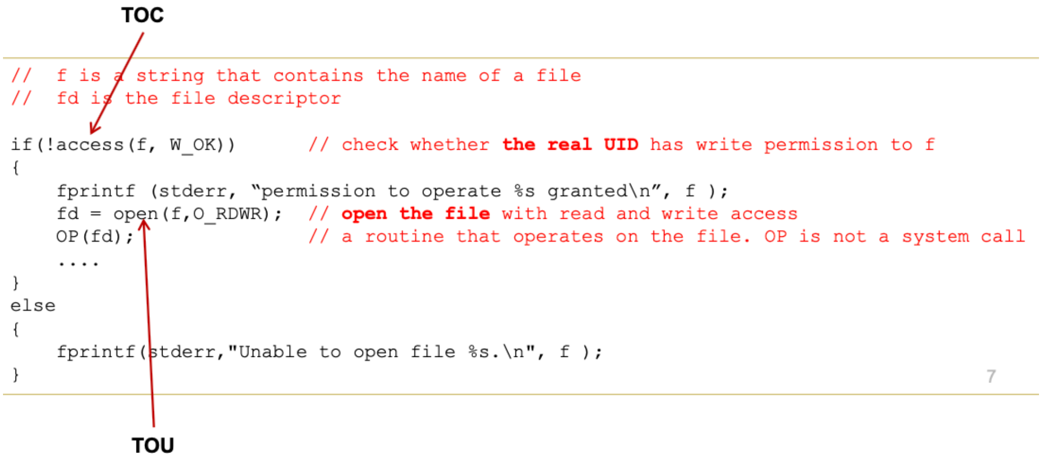
Overview of TOCTOU



Step by step breakdown

Example CWE-367 (Common Weakness Enumeration)
This program needs to be run with elevated privilege, i.e. set-UID-root. Inside, there is a invocation of function access(f, W_OK).

This function checks whether the user executing the program has the permission to write to the specified filename f based on the process' **real UID and GID**, and returns 0 if the process **has** the permission. Since it checks real UID, a malicious user needs to find a way around it in order to get a sensitive target file.



```
//  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 on the file. OP is not a system call
    ....
}
else
{
    fprintf(stderr,"Unable to open file %s.\n", f );
}
```

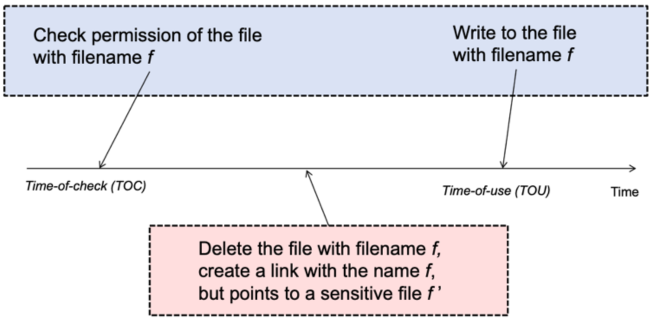
One key note about access() is that access() checks whether the calling process can access the file pathname. If pathname is a symbolic link, it is dereferenced, i.e. the contents that is linked is accessed. R_OK, W_OK, and X_OK test whether the file exists and grants read, write, and execute permissions, respectively.

This thus allows a malicious user Bob (real UID: bob, effective UID: root) to do the following:

1. Let us suppose that the filename f is `/usr/year1/bob/list.txt`, which is a file owned by Bob. Bob has the permission to write to it.
2. After Bob starts the process, he immediately replaces the file `/usr/year1/bob/list.txt` with a **symbolic link** that points to a sensitive system file.

    a. He does this by running a script that does the following:
        i. Delete `/usr/year1/bob/list.txt`
        ii. Create a symbolic name with the filename f and point it to a system file via this UNIX command:
            1. `ln -s /usr/course/cs2107/grade.txt /usr/year1/bob/list.txt`

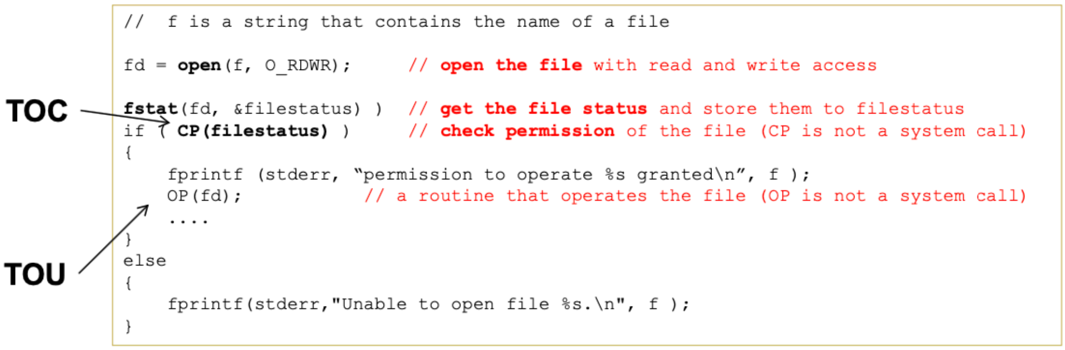
3. With some probability, Bob wins the race, and the process operates on the system file `/usr/course/cs2107/grade.txt`
4. Bob himself does not have write permission to the sensitive system file, but he is able to change the file due to his effective UID.


Check permission of the file with filename *f*
Write to the file with filename *f*
Time-of-check (TOC)
Time-of-use (TOU)
Time
Delete the file with filename *f*, create a link with the name *f*, but points to a sensitive file *f'*

Defence Approach 1 for TOCTOU

In C programming, avoid using separate system calls that take the same filename as input. Instead, try to open the file only once, effectively locking it to block any further changes by other processes, and use the file handle/descriptor.

In terms of code, avoid using the access() system call; rather, open the file once using open() and use fstat() system call to check the permission.



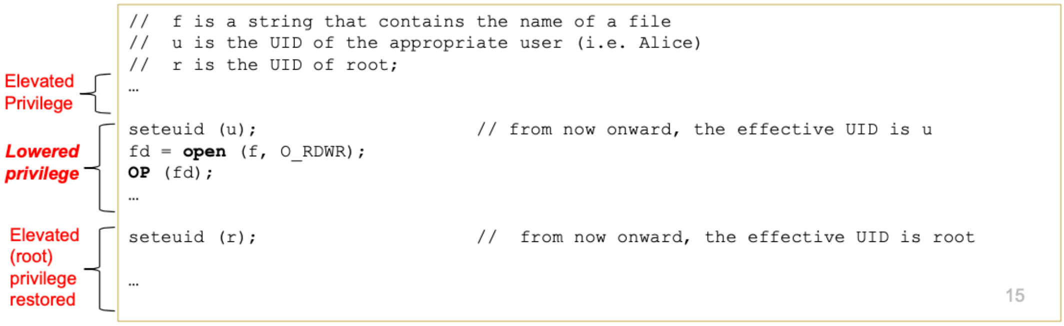
```
//  f is a string that contains the name of a file

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 );
}
```

Defence Approach 2 for TOCTOU

A better practice would be to avoid writing your own access-control on files, and leave the checking to the **underlying OS** after appropriately setting your process credentials. For instance, we can set the process' effective UID to the user real UID (normal/non-root) before accessing the file. This way, the OS checks the permission, and decides whether to grant or deny the access. Afterwards, reset the effective UID back to root if required.



```
//  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

…
```

Defence and Preventive Measures

Now that we've covered all the various attacks, it's time for us to look at what we can do against them. Many bugs and vulnerabilities are due to a programmer's ignorance. It is difficult to ensure that a program is bug-free. However, there are various useful countermeasures available.

Input Validation using Filtering

In almost all examples we have seen (except TOCTOU), attacks are carried out by feeding carefully-crafted input to the system. Those inputs do not follow the expected format, e.g. input is too long, contains control or meta characters, contains negative numbers etc.

Hence, one preventive measure is to perform an input validation or filtering whenever an input is required from the user. If it is not of the expected format, reject it.

Difficulties

It is difficult to ensure that the filtering is perfect and complete, as seen in the case of UTF-8. There may be representations that a programmer is not aware of, and a filter that completely blocks all bad input while still accepting all legitimate inputs is **very difficult to design**.

Approaches

There are generally two approaches:

1. White List
    a. A list of items that are known to be benign and are allowed to pass
    b. Could be expressed using regular expression
    c. However, some legitimate inputs may be blocked
2. Black List
    a. A list of items that are known to be bad and to be rejected
    b. For example, to prevent SQL injection, if the input contains meta characters, reject it
    c. However, some malicious input may be passed

The White List approach is more secure, but may not be more desirable. It is a trade-off.

Using "Safer" Functions

We can completely avoid functions that are known to be insecure, and use the **safer** versions of these functions.

For example, we can use strncpy() instead of strcpy(). However, even then, there could still be vulnerabilities. These functions are merely safer, it does not mean that they cannot be exploited still.

Bounds Checking and Type Safety

Bounds Checking

Some programming languages such as Java and Pascal perform bounds checking at runtime. In other words, 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 or subscript is checked against the upper and lower bounds. In other words, when doing a[i] = 5;, the following happen:

- Check if i >= lower bound
- Check if i <= upper bound
- Then assign 5 to the memory location

If the checks fail, the process will be halted, and an exception may be thrown. The added 2 steps reduce efficiency, but it prevents buffer overflow.

Many of the known vulnerabilities are actually due to buffer overflow, which can be prevented by this simple bounds checking.

The infamous C and C++ do not perform bounds checking, yet so many pieces of software are written in C or C++! Here's a sharing by a Turing Award winner:

> *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, trying to do a = b; when a is an 8-bit integer and b is a 64-bit integer would throw an error.

This can 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
```

Memory Protection

Randomization

It is to the attacker's advantage when data and codes are always stored in the same locations in memory. As such, we have **address space layout randomization (ASLR)** as a prevention technique that can help decrease the attacker's chances.

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.

We can turn disable ASLR in Linux via:

```
sudo sysctl -w kernel.randomize_va_space=0
```

Canary

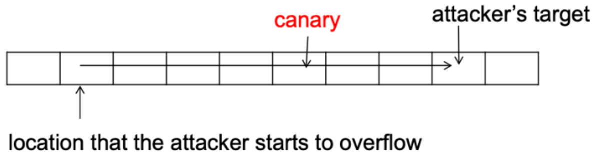
Canaries are secret values inserted at carefully selected memory locations during runtime. These secret values are not to be modified; checks are carried out at runtime to ensure that the values remain the same. If they have been changed, the program will halt.

Canaries can help to **detect (but not prevent)** buffer overflow, especially stack overflow. In a typical buffer overflow situation, consecutive memory locations have to be ran over. The canaries would naturally be modified.

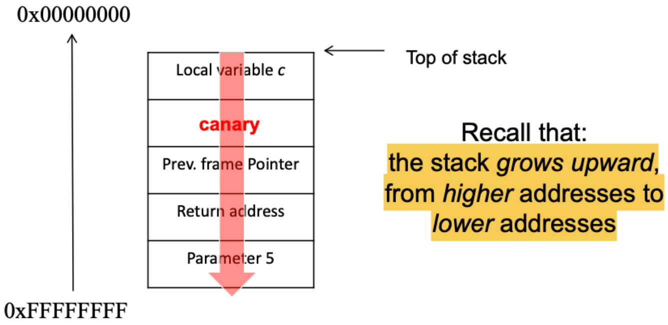
It is also important to keep the values secret. If the attacker knows the secret value, they can simply write the value to the canary while overrunning it, making it impossible to detect.

In Linux, we can turn off gcc canary-based stack protection by supplying this flag when compiling: `-fno-stack-protector`

This is the model of the memory:



Rough representation



Canary is located below the local variables

Let us look at some examples of how canaries can help detect stack smashing. Let us look at the following program:

```
#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;
}
```

When compiled with stack protector:

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

Canary is checked only when the function exits, hence the program completes.

When compiled without stack protector:

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

The segmentation fault is due to the return address pointing to an invalid instruction. If the attacker is smart, the return address would actually still point to a valid address, hence stack smashing would not be detected.

Similar events would occur if the program calls another function with stack smashing. Once the function completes, canary check would terminate the program, while segmentation fault may occur depending on the return address.

Code Inspection

One common sense solution is to check our code. We can do it manually, but this is tedious and often not very reliable.

The other way is to do automated checking using some automations and tools. An example would be taint analysis, or taint propagation analysis.

In taint analysis:

- Variables that contain input from users are labelled as **sources**.
- Critical functions are labelled as **sinks**.
- Taint analysis checks if any of the sinks' arguments could potentially be affected or tainted by a source
- For example
    - Source is user input
    - Sink is the opening of a system file or the evaluation of a SQL command
- If so, a special check i.e. manual inspection would be carried out

Taint analysis can be both static, i.e. checking the code without running or tracing it, or dynamic, i.e. running the code with some inputs. However, code coverage is an issue for dynamic analysis, since you do not know what are all the possible inputs that can result in issues.

Testing

Vulnerabilities can be discovered via testing. There are three types of testing:

- White-box testing
    - The tester has access to the application's source code
- Black-box testing
    - The tester does not have access to the source code
- Grey-box testing
    - A combination of the above, where the tester has some reverse-engineered binary or executable

Security testing attempts to discover areas susceptible to intentional attack, and hence would test for inputs that rarely occur under normal circumstances, e.g. very long names, names with numeric values, strings containing meta characters, etc.

Fuzzing is a technique that sends **malformed inputs** to discover vulnerabilities. Fuzzing can be both automated or semi-automated, and is more effective than simply sending in random inputs.

Principle of Least Privilege

We have covered this principle before. Basically, when writing a program, be conservative in elevating privileges. When deploying a software system, do not give users more access rights than necessary, and do not activate unnecessary options.

For example, 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 or off.

Should all features 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 switching off all unnecessary features. Your clients might not be aware of the implications and thus are at a higher risk.

Hardening refers to the process of securing a system by reducing its surface of vulnerability. This is done by reducing the number of functions in a system.

Patching

This is the lifecycle of a vulnerability:

1. Vulnerability is discovered
2. Affected code is fixed
3. Revised version is tested
4. Patch is made public
5. Patch is applied

In some cases, the vulnerability may be announced without any technical details before a patch is released. This vulnerability is likely to be known to only a small number of attackers, or even none, before it is announced.

When a patch is released, the patch may actually help attackers to derive the vulnerability, as now they can inspect the patch. Hence, interestingly, the number of successful attacks can go up after the vulnerability or patch is announced, since more attackers would be aware of the exploit.

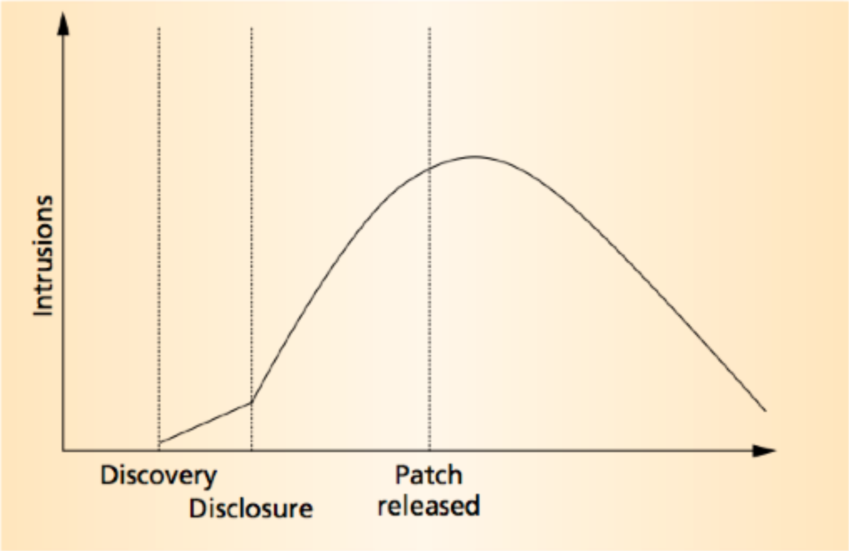


Figure 1. Intuitive life cycle of a system-security vulnerability. Intrusions increase once users discover a vulnerability, and the rate continues to increase until the system administrator releases a patch or workaround.

It is thus crucial to apply a patch timely. However, it is much more complex than it seems:

- For critical systems, it is not wise to apply the patch immediately before rigorous testing
  - For example, a patch may cause the train scheduling software to crash
- Patches may affect the applications, thus affecting the operations of organisations
  - For example, after a student applied a patch on his Adobe Flash, he could not upload a report to LumiNUS and thus missed a project deadline.

It is so complex that Patch Management is an actual field of study!

Summary

We thus have the following defence mechanisms and countermeasures, to be adopted at different stages of the Software Development Lifecycle (SDLC):

- Development Stage
    - Using Safer Functions
    - Bounds Checking and Type Safety
    - Filtering (Input validation)
    - Code Inspection (Taint analysis)
    - Principle of Least Privilege*
    - Executable Generation with Enabled Memory Protection*
- Testing Stage
    - Testing: Fuzzing and Penetration Testing
- Deployment (including Software Execution) Stage
    - Runtime Memory Protection*: Randomization
    - Principle of Least Privilege*: Disable Unnecessary Features
    - Patching

* applies to multiple stages

Tutorial Learnings

strcat(dest, source);
Unsafe. The buffer dest can get overflowed since there is no limit to the number of characters concatenated into it.

strncat(dest, source, n);
Safe if n < the remaining characters (bytes) available on the dest buffer. Note that the dest buffer must be large enough to contain the concatenated resulting string and also an additional null character.

memcpy(dest, source, n);
Safe if n ≤ the sizes of the dest and src buffers. Note that memcpy performs a binary-copying operation.

strncpy(dest, source, strlen(source))
Unsafe. The buffer dest can get overflowed since strlen(source) can be greater than dest's length.

printf(f, str)
Potentially unsafe if f comes from a user input.

printf("hello my name is %s", str);
Safe.

sprint(str, f)
Unsafe. The buffer str can get overflowed since the formatted string output can be longer than str's length.

printf("Please key in your name: "); gets(str);
Unsafe. The buffer str can get overflowed since there is no limit to the number of characters read and stored into it.

scanf("%s", str);
Unsafe. The buffer str can get overflowed since there is no limit to the number of characters read and stored into it.

scanf("%20s", str);
Safe if the size of str > 20 since at most 20 characters are stored by scanf() into str. Note that a terminating null character is added at the end of str. The specified maximum input length (i.e. 20) does not include this additional terminator character. As such, str must be at least one character longer than the specified maximum input length.

Accessing Uninitialized Variables
In C, the value of an uninitialized local variable is indeterminate/undefined, which basically can be anything. Accessing such an uninitialized variable leads to an "undefined behaviour". A program which prints out an uninitialized array poses a security risk since the printed data could be sensitive. This is the case since memory chunks in a running process' memory are basically "recycled", both between different process executions and during the process execution. If the array happens to contain a sensitive piece of information, the information will then get leaked out to the external user.