Information Security
CS3002

Lecture 13
10th October 2023

Dr. Rana Asif Rehman
Email: r.asif@lhr.nu.edu.pk

Control Hijacking

Source: web.nvd.nist.gov

Control Hijacking

- Attacker's goal
  - Take over target machine (e.g. web server)
    - Execute arbitrary code on target by hijacking application control flow

- Examples:
  - Integer Overflows
  - Format String vulnerabilitie
  - Buffer Overflows

    - Mostly in C/C++ programs
      - Can you think of a reason?

Control Hijacking – Our Assumptions

- We write our code in languages that offer several layers of abstraction over machine code; even C
  - High level statement: “=” (assign), “;” (seq), i**f, while..etc.**
  - Procedures/ functions
- Naturally, our execution model assumes
  - Basic statements are atomic.
  - Only one of the branches of an if statement can be taken.
  - Functions start at the beginning.
  - Functions execute from beginning to end.
  - And, when done, they return to their call site.
  - Only the code in the program can be executed.

Control Hijacking – Realities, Machine Code Level

- Each basic statement compiled down to many instructions.
- There is no restriction on the target of the jump.
- Can start executing in the middle of functions.
- Returns can go to any program instruction.
- Dead code(e.g. unused library functions) can be executed.
- On the x86, can start executing not only in the middle of functions, but in the middle of instructions.

Control Hijacking – Why so much C?

- Systems software is often written in C (operating systems, file systems, databases, compilers, network servers, command shells and console utilities)
- Why?
  - C is essentially high level assembly
  - Programmers obsessed with speed

Control Hijacking – Why so much C?
Disadvantages

- As C, is high level assembly
  - Exposes raw pointers to memory
  - Does not perform bound checking on arrays
    - Hardware does not do that
    - People want maximum amount of speed possible.
    - C wants to get you as close to the hardware as possible
    - Difficult to determine what it means to have a pointer that is in bound.
- Does C# or Java have pointers?
- Pointers are not bad, there arithmetic raises security issues.

Control Hijacking – Integer Overflows

- Integer overflow: an arithmetic operation attempts to create a numeric value that is larger than can be represented within the available storage space.

- Example:

Test 1:
```
    short x = 30000;
    short y = 30000;
    printf("%d\n", x+y);
```

Test 2:
```
    short x = 30000;
    short y = 30000;
    short z = x + y;
    printf("%d\n", z);
```

Will two programs output the same?
Assuming short uses 16 bits.
What will they output?

C Data Types

• short int	16bits	[-32,768; 32,767]
• unsigned short int	16bits	[0; 65,535]
• unsigned int	16bits	[0; 4,294,967,295]
• Int	32bits	[-2,147,483,648; 2,147,483,647]
• long int	32 bits	[-2,147,483,648; 2,147,483,647]
• char	8 bits	[0; 255]

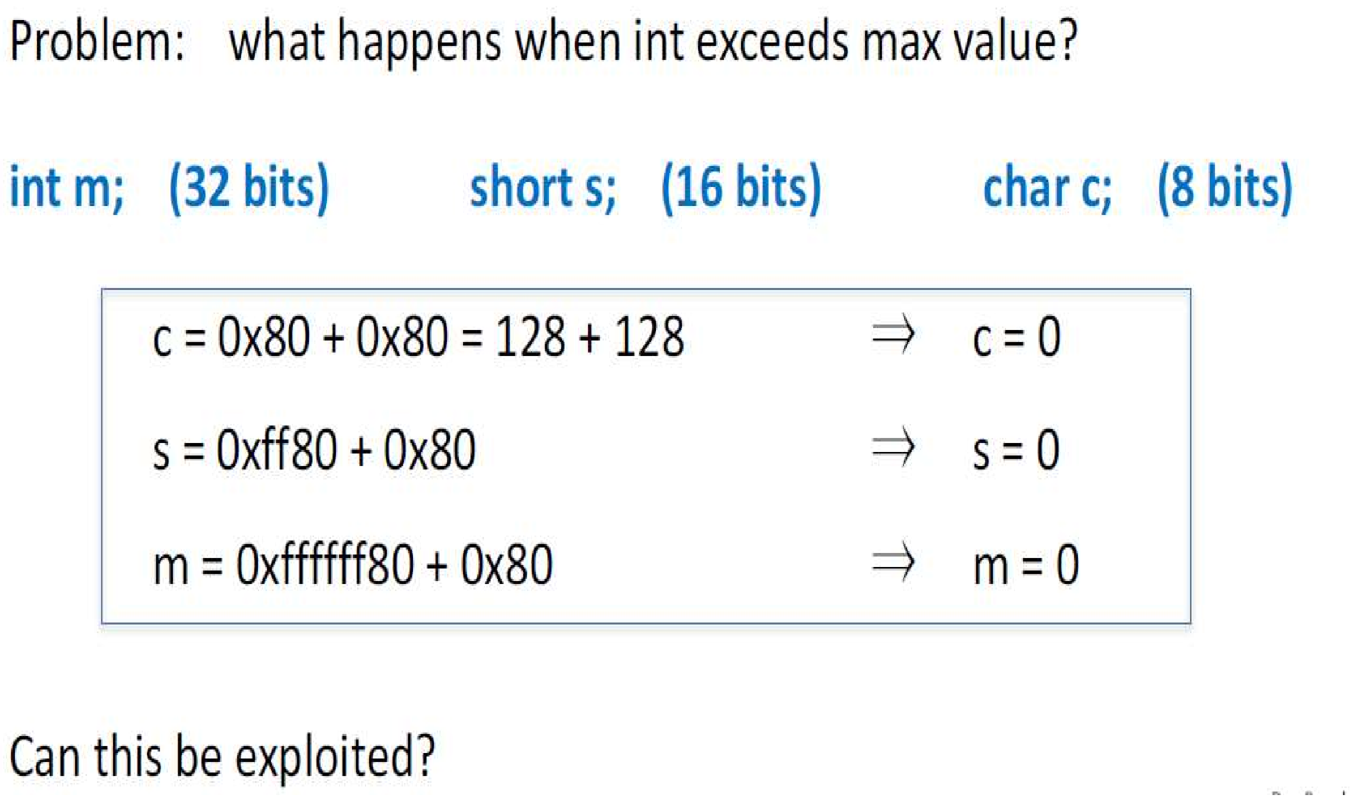
Where Does Integer Overflow Matter?

- Allocating spaces using calculation.
- Calculating indexes into arrays
- Checking whether an overflow could occur

- Direct causes:
  - Truncation; Integer casting

Control Hijacking – Integer Overflows

What is the size of int in 64 bits machines?



Problem: what happens when int exceeds max value?

int m; (32 bits) short s; (16 bits) char c; (8 bits)

c = 0x80 + 0x80 = 128 + 128	$\Rightarrow$	c = 0
s = 0xff80 + 0x80	$\Rightarrow$	s = 0
m = 0xffffff80 + 0x80	$\Rightarrow$	m = 0

Can this be exploited?

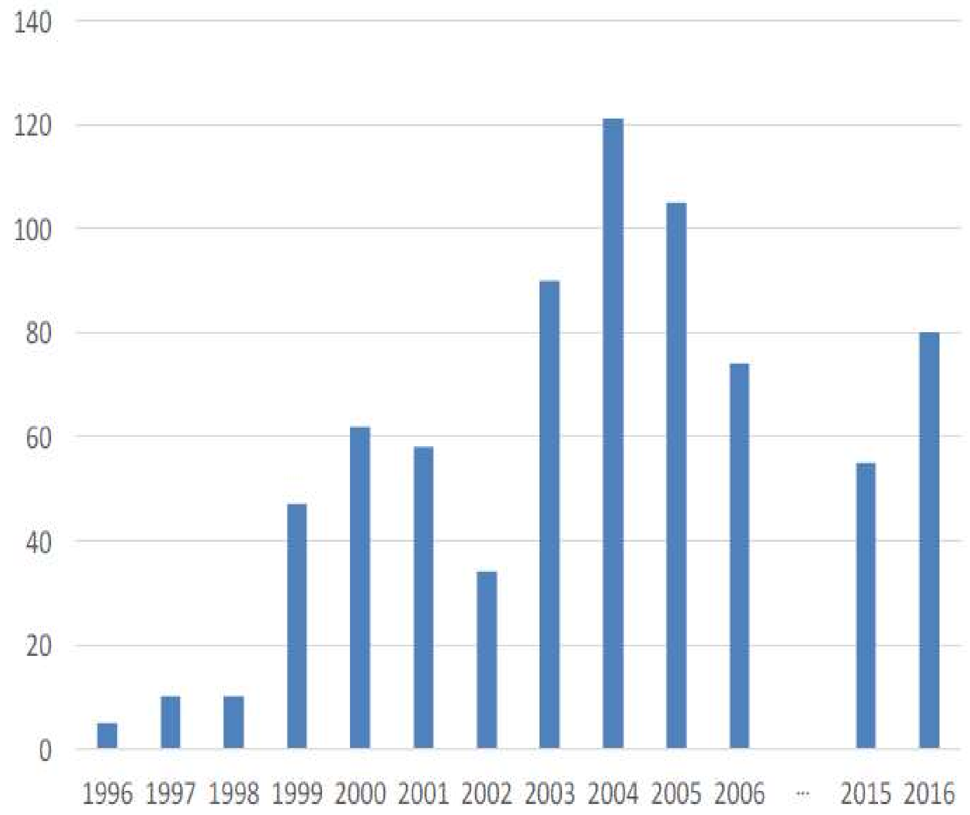
Some Real Life Examples

There is a Facebook group called “If this group reaches 4,294,967,296 it might cause an integer overflow. “ This value is the largest number that can fit in a 32 bit unsigned integer. If the number of members of the group exceeded this number, it might cause an overflow. Whether it will cause an overflow or not depends upon how Facebook is implemented and which language is used – they might use data types that can hold larger numbers. In any case, the chances of an overflow seem remote, as roughly 2/3 of the people on earth would be required to reach the goal of more than 4 billion members.

Some Real Life Examples (cont.)

- On December 25, 2004, Comair airlines was forced to ground 1,100 flights after its flight crew scheduling software crashed. The software used a 16-bit integer (max 32,768) to store the number of crew changes. That number was exceeded due to bad weather that month which led to numerous crew reassignments.
- Many Unix operating systems store time values in 32-bit signed (positive or negative) integers, counting the number of seconds since midnight on January 1, 1970. On Tuesday, January 19, 2038, this value will overflow, becoming a negative number. Although the impact of this problem in 2038 is not yet known, there are concerns that software that projects out to future dates – including tools for mortgage payment and retirement fund distribution – might face problems long before then. Source: Year 2038 Problem" http://en.wikipedia.org/wiki/Year_2038_problem

Control Hijacking – Integer Overflows (cont.)



Source: NVD/CVE

Countermeasures

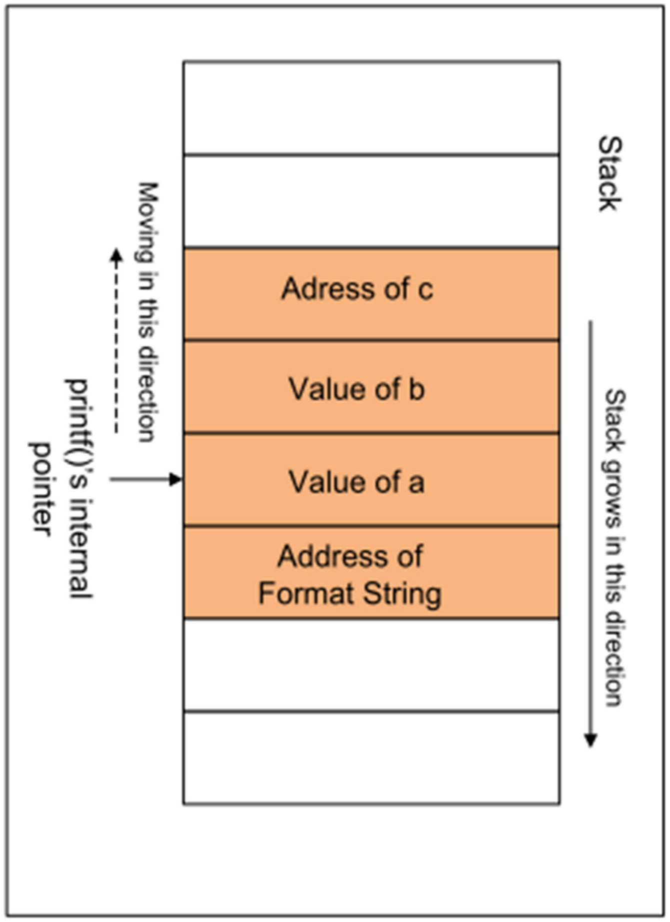
- Sound Range Checking
  - 1) Calculate sum
  - 2) If both numbers are positive and sum is negative then return -1 Else If both numbers are negative and sum is positive then return -1 Else return 0

```
int addOvf(int* result, int a, int b)
{
    *result = a + b;
    if(a > 0 && b > 0 && *result < 0)
        return -1;
    if(a < 0 && b < 0 && *result > 0)
        return -1;
    return 0;
}
```

- Use more save native types like long

Control Hijacking – String Format Vulnerabilities

- A format string is an ASCII string that contains text and format parameters.
- printf ("a has value %d, b has value %d, c is at address: %08x\n",a, b, &c);



Vulnerable functions

Any function using a format string.

Printing:

- printf, fprintf, sprintf, …
- vprintf, vfprintf, vsprintf, …

Logging:

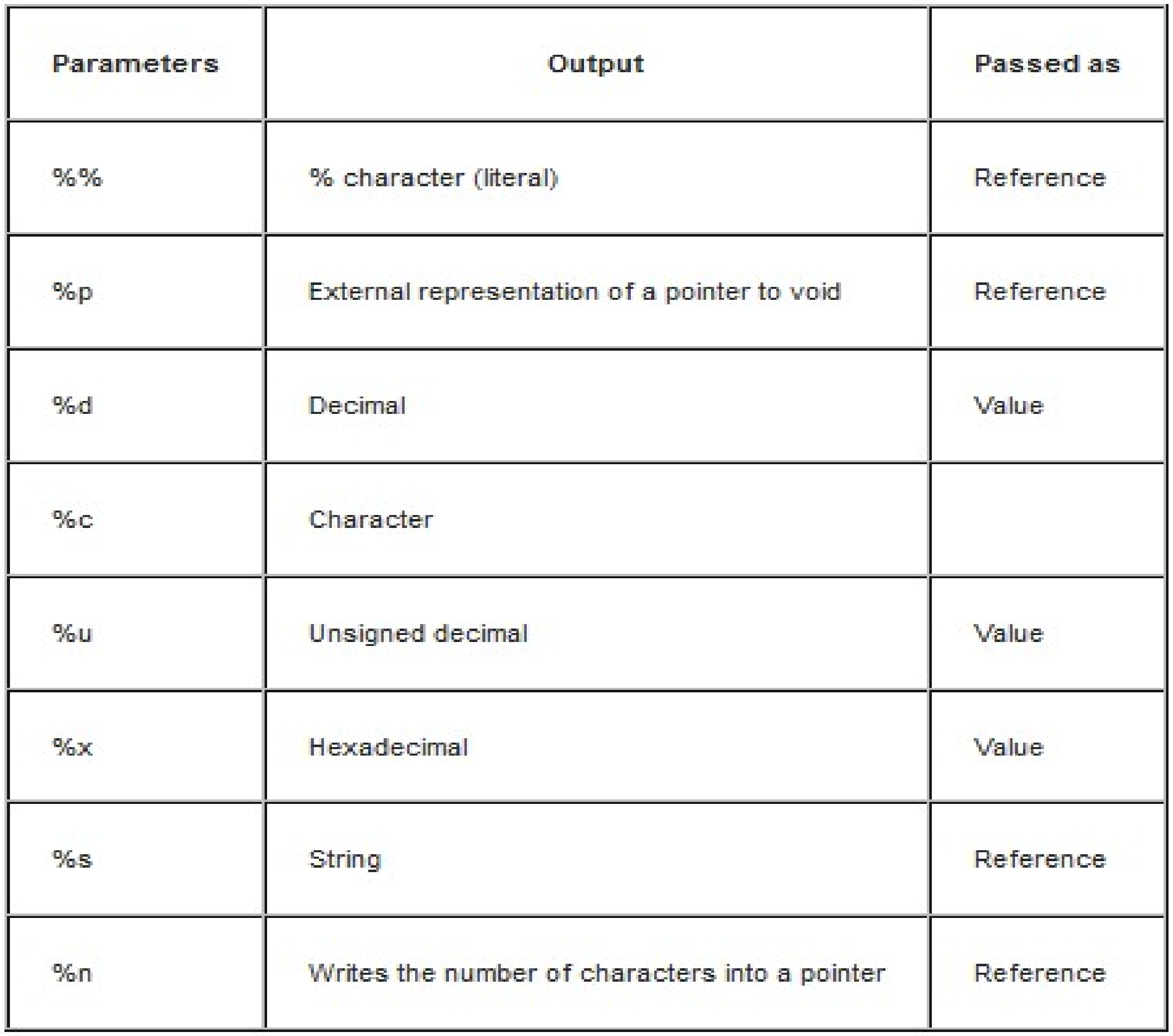
- syslog, err, warn

Control Hijacking – String Format Vulnerabilities (cont.)

- Char *name = “whatever”;
- printf("my name is : %s\n", name);   output ?
- printf(“my name is : %s\n”); output ?

Control Hijacking – String Format Vulnerabilities (cont.)

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



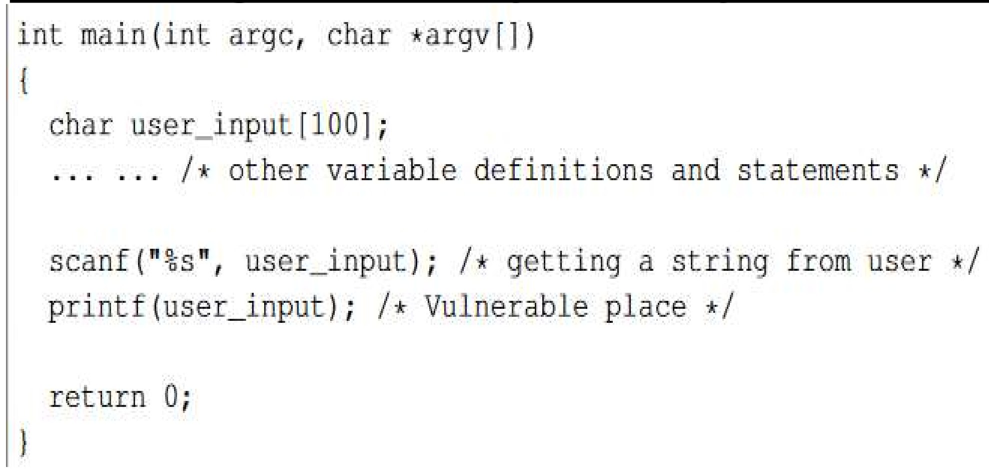
Parameters	Output	Passed as
%%	% character (literal)	Reference
%p	External representation of a pointer to void	Reference
%d	Decimal	Value
%c	Character	
%u	Unsigned decimal	Value
%x	Hexadecimal	Value
%s	String	Reference
%n	Writes the number of characters into a pointer	Reference

Malicious format strings

- %x%x%x%x%x%x%x%x
  will print bytes from the top of the stack
- %s
  will interpret the top bytes of the stack as an address X, and then prints the string starting at that address A in memory, ie. it dumps all memory from A up to the next null terminator
- %n
  will interpret the top bytes of the stack as an address X, and then writes the number of characters output so far to that address

Control Hijacking – String Format Vulnerabilities (cont.)

- What can be done?
  - **Viewing the process memory**
    - printf ("%08x.%08x.%08x.%08x.%08x\n");
      - 16d2e5f8.00000000.e9a7ee80.e9cb2560.00000000
  - **Viewing memory at any location**

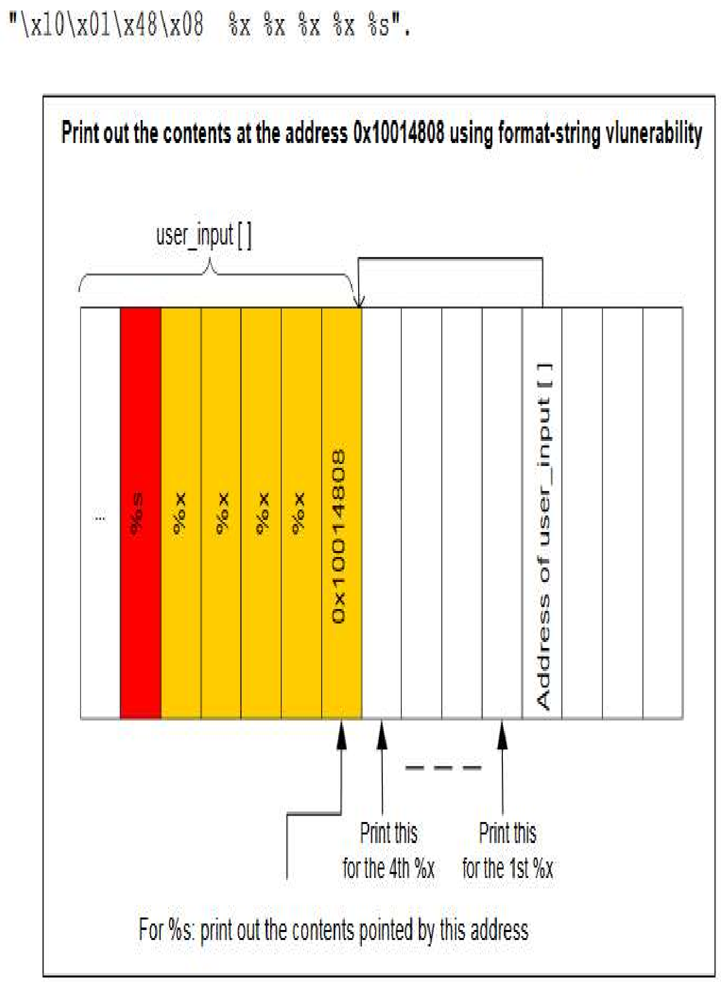


```
int main(int argc, char *argv[])
{
  char user_input[100];
  ... ... /* other variable definitions and statements */

  scanf("%s", user_input); /* getting a string from user */
  printf(user_input); /* Vulnerable place */

  return 0;
}
```

Control Hijacking – String Format Vulnerabilities (cont.)

- What can be done?
  - **<u>Viewing memory at any location</u>**
- The key challenge in this attack is to figure out the distance between the ‘s’ and the address passed to the printf() function. This distance decides how many %x you need to insert into the format string, before giving %s

  - main () { char *s = "\x10\x01\x48\x08 %x %x %x %x %s"; printf (s); }
  - f154df88 0 33d83e80 33fb7560 0 339e4e c5 (null)



"\x10\x01\x48\x08 %x %x %x %x %s".
Print out the contents at the address 0x10014808 using format-string vlunerability
user_input [ ]
...
%s
%x
%x
%x
%x
0x10014808
Address of user_input [ ]
Print this for the 4th %x
Print this for the 1st %x
For %s: print out the contents pointed by this address

Control Hijacking – String Format Vulnerabilities (cont.)

- What can be done?
  - **Overwriting arbitrary memory**
    - int i;
    - printf ("12345%n", &i); How can you write a particular no. at any next location?
  - **Overwriting any memory**
    - Any idea?

- Using this attack, attackers can do the following:
  - Overwrite important program flags that control access privileges
  - Overwrite return addresses on the stack, function pointers, etc.

Malicious format strings

```
int j;
char* msg; ...
printf( ”how long is this? %n”, &j);
```

- %n causes the number of characters printed to be written to j.
- Here it will give j the value 18

- Any guess what printf(”how long is this? %n”, msg”); will do?

- It interprets the top of the stack as an address, and writes the value 18 to it

Control Hijacking – String Format Vulnerabilities (cont.)

- What can be done?
  - **Crash of the program**
    - printf ("%s%s%s%s%s%s%s%s%s%s%s%s");
    - How?

Because '%s' displays memory from an address that is supplied on the stack, where a lot of other data is stored, too, our chances are high to read from an illegal address, which is not mapped.

Countermeasures (cont.)

- In the GNU Compiler Collection, the relevant compiler flags are, -Wall,-Wformat, -Wno-format-extra-args, -Wformat-security, -Wformat-nonliteral, and -Wformat=2.
- Most of these are only useful for detecting bad format strings that are known at compile-time. If the format string may come from the user or from a source external to the application, the application must validate the format string before using it. Care must also be taken if the application generates or selects format strings on the fly. The -Wformat-nonliteral check is more stringent.
- Src: WikiPedia