



# Foundations of Cybersecurity

## C and C++ Secure Coding

Gianluca Dini  
Dept. of Information Engineering  
University of Pisa

Email: [gianluca.dini@unipi.it](mailto:gianluca.dini@unipi.it)

Version: 2021-03-03

1

# Credits




- These slides come from a version originally produced by Dr. Pericle Perazzo

C and C++ Secure Coding

**STRINGS**

3

# Strings



UNIVERSITÀ DI PISA

H	e	l	l	o	,		W	o	r	l	d	!	\0	?	?
---	---	---	---	---	---	--	---	---	---	---	---	---	----	---	---


← string length →

← array capacity →

4

In C, there is not a native string type. Strings are implemented by arrays of characters, with a special character (*string terminator*, '\0') to indicate the end of the string. The array capacity must always be  $\geq$  of the string length + 1, to accomodate the terminator at the end. No terminator character is allowed inside the string. The absence of a native string type and the intrinsic insecurity of the C string library functions can lead to many vulnerabilities, some of which are hard to correct. The C++ standard introduces the `std::string` type which is far less error-prone.


# gets()



UNIVERSITÀ DI PISA

```
void func() {  
    char buf[1024];  
    if (gets(buf) == NULL) {  
        /* Handle error */  
    }  
}
```

\* all code snippets taken  
or elaborated from:  
Robert C. Seacord,  
«Secure Coding in C  
and C++ (2° ed.)»,  
Addison-Wesley



is this vulnerability exploitable?

in most cases, NO

5

This function takes a string inserted by the user with `gets()` and do some processing on it. If an error occurred in the input, it handles it.

The `gets()` function, which was deprecated in the C99 Technical Corrigendum 3 and removed from C11, is inherently unsafe and should never be used because it provides no way to control how much data is read into a buffer from standard input. It is interesting to study the exploitability of the `gets()` vulnerability. Is this obvious vulnerability exploitable by a real adversary? The answer is, in the majority of cases, no.


## gets()



- User attacking process is the same that launched it  
→ Attacker is attacking herself
- Dangerous cases:
  - Standard input redirected to untrusted source (e.g., socket)
  - Program used by anonymous users (e.g., public computers)
- Should we leave it unfixed?
  - A bug that seems not to be exploitable could be exploitable
  - It could become exploitable in future (program changes, system reconfigurations, etc.)
- You cannot leave an armed bomb in your system!

6

The reason is that, in the majority of cases, the user that attacks the process by entering a long input is the same user that launched such a process. In other words, the attacker is attacking herself. Therefore, this vulnerability is exploitable only in those cases in which the input is provided by an entity different from the one that launched the program. For example, when the standard input is redirected to an untrusted source, for example a network socket, or when the program is used by the anonymous users, like it happens in public computers. In all the other cases, the bug does not lead to an exploitable vulnerability. So, should we ignore it and leave it unfixed? The answer is no, because it is very hard to be sure that a vulnerability that seems not to be exploitable is actually not exploitable, by leveraging ways that you did not expect. Moreover, it could become exploitable in the future, when the program will be modified by other programmers, or re-used in other systems or with other configurations. In other words, we cannot leave an armed bomb inside a system, hoping that no one will find a way to explode it.

  
UNIVERSITÀ DI PISA

# gets()

- User types:  
Hello, World!<enter>
- gets():  


H	e	l	l	o	,		W	o	r	l	d	!	\0	?	?
---	---	---	---	---	---	--	---	---	---	---	---	---	----	---	---
- fgets():  

H	e	l	l	o	,		W	o	r	l	d	!	\n	\0	?
---	---	---	---	---	---	--	---	---	---	---	---	---	----	----	---

!!!

A solution could be using fgets(char\* s, int size, stdin) instead of gets(). The function fgets() stops reading characters from standard input either if a '\n' (*new line*) character has been read, or if BUFFER\_SIZE-1 characters have been read, so it cannot cause a buffer overflow. However, fgets() is not an exact replacer of gets(), since it does not remove the new line character from the returned string, so additional code is needed to remove it. Moreover, if the user inserts an input line which is more than BUFFER\_SIZE-1 characters long, fgets() leaves the exceeding characters in the internal buffer of the standard input. In this way, successive calls to fgets() will read these exceeding characters instead of letting the user insert a new input line.

# gets()




UNIVERSITÀ DI PISA

```
void func() {  
    char buf[1024];  
    if (fgets(buf, 1024, stdin) == NULL) {  
        /* Handle error */  
    }  
    char* p = strchr(buf, '\n');  
    if (p) { *p = '\0'; }  
}
```

must be >0

```
void func() {  
    char buf[1024];  
    if (gets_s(buf, 1024) == NULL) {  
        /* Handle error */  
    }  
}
```

NOT AVAILABLE IN SOME COMPILERS!



8

Correcting the vulnerability with fgets() thus requires additional code to replace the (possible) new line characters with a string terminator. The function strchr() can be used to do this. Note that fgets() wants a buffer length parameter greater than 0, to accommodate at least the string terminator.

Another solution is to use gets\_s(), which is available from C11 and makes part of the *bounds-checked version* of some string functions. Other bounds-checked string functions are fopen\_s(), printf\_s(), strcpy\_s(), wcsncpy\_s(), mbstowcs\_s(), qsort\_s(), getenv\_s(). These functions have been first defined by Microsoft, and then accepted (with some changes) in the C11 standard as an optional extension of the standard library. Since gets\_s() is optional in the C11 standard, not all compilers implement it. In particular, it is available only if \_\_STDC\_LIB\_EXT1\_\_ is defined. To use gets\_s(), you have to define \_\_STDC\_WANT\_LIB\_EXT1\_\_ to the value 1.



# gets()



```
void func() {  
    std::string buf;  
    std::getline(std::cin, buf);  
    if (!std::cin) {  
        /* Handle error */  
    }  
}
```



9


If you can use standard C++ library, maybe the best solution is to use `std::string` and `std::getline(std::cin, ...)` method.


`std::string` uses dynamic approach to strings in that memory is allocated as required. This means that in all cases `size() <= capacity()`. The class implements the “callee allocates, callee frees” memory management strategy. This is the most secure approach, but it is supported only in C++. Because `std::string` manages memory, the caller does not need to worry about the details of memory management.

For example, string concatenation is handled simply as follows:

```
string str1 = "Hello ";  
string str2 = "World!";  
string str3 = str1 + str2;
```


# gets()

  
UNIVERSITÀ DI PISA



```
#include <iostream>
#include <string>

int main(void) {
    std::string buf;
    std::cin >> buf;
    if (!std::cin) {
        // Handle error
    }
}
```



10

This program is elegant, handles buffer overflows and string truncation, and behaves in a predictable fashion.


Std::string is less prone to security vulnerability than null-terminated byte strings, although coding errors leading to security vulnerabilities are still possible. One area of concern is *iterators*.

C and C++ secure coding

# FORMATTED OUTPUT AND VARIADIC FUNCTIONS

11

# Variadic Functions



UNIVERSITÀ DI PISA

- Number/type of args determined at runtime from a format string
- Output:

```
int a = 10, b = 20;  
char s[] = "Hello, world!";  
printf("a is: %d, b is: %d, s is: %s\n", a, b, s);
```

format string

%d = signed int      %s = C-string

```
a is: 10, b is: 20, s is: Hello, world!
```

12

The standard way to perform i/o in C programs is through `printf()` and `scanf()` functions, which are *variadic functions*. A variadic function is a function with a flexible number of arguments (typically, one or more). The number and type of the arguments of each call are determined at runtime from the *format specifiers* (`%d`, `%x`, `%p`, `%s`, etc.) in the first argument, which is the *format string*. For example, with `printf()` and a format string `"A = %d, B = %d, STR = %s"` it is possible to print two signed integers (`%d`) and a string (`%s`) onto standard output (screen).

# Variadic Functions



- Some common format specifiers:

Format specifier	Meaning	Example
%i	Integer in decimal digits	12
%x	Unsigned integer in hexadecimal digits	3f
%f	Floating-point number	3.14
%C	Single character	a
%s	C-string	Hello
%%	Literal '%'	%



# Variadic Functions

- Input:

```
int a, b;  
float c;  
int ret;  
if (scanf("%d %d %f", &a, &b, &c) != 3) {  
    /* Handle error */  
}
```

%d = signed int      %f = float

30

40

3.14

a

b

c

14

With `scanf()` and a format string `"%d %d %f"` it is possible to read two signed integers (`%d`) and a float number (`%f`) from standard input (keyboard) separated by spaces.  
With `scanf()`, arguments must be pointers to variables.

## scanf() + String Argument



UNIVERSITÀ DI PISA


```
void func() {  
    char buf[1024];  
    if (scanf("%s", buf) != 1) {  
        /* Handle error */  
    }  
}
```



15

The standard function `scanf()` gets a series of values (of possibly different types) separated by spaces from the standard input. With a `"%s"` parameter, `scanf()` gets a *word string* (i.e., a string with no spaces inside) and puts it in the associated argument (in this case `"buf"`). The function `scanf()` returns the number of items effectively read (in this case 1, or 0 on error). However, it fails to check if the buffer has enough space to accommodate the string. The same problem affects similar standard functions like `fscanf()` and `sscanf()`.

# scanf() + String Argument





UNIVERSITÀ DI PISA

```
void func() {  
    char buf[1024];  
    if (scanf("%1023s", buf) != 1) {  
        /* Handle error */  
    }  
}
```

ERROR-PRONE!

```
void func() {  
    std::string buf;  
    std::cin >> buf;  
    if (!std::cin) {  
        /* Handle error */  
    }  
}
```





A solution is to specify the max string length (i.e., the buffer capacity minus 1) inside the format string, for example: "%1023s". This solution is error-prone too, because if the buffer capacity gets changed, then the programmer must remember to change also the format string accordingly. Moreover, it is difficult to apply in case that the buffer capacity is unknown at compile time.

If you can use standard C++ library, maybe the best solution is to use std::string and the >> operator of std::cin.



# sprintf()



```
void func(const char *name) {  
    char filename[128];  
    sprintf(filename, "%s.txt", name);  
}
```



17

The standard variadic function `sprintf()` formats a series of values (possibly of different types) following a format string and stores the result in a C string. For example, this function adds a ".txt" extension to the file name "name" and stores the result in "filename". From this slide on, the input arguments in red font are to be considered tainted. We consider a C-string tainted when its pointer points to a valid, allocated, and NULL-terminated string, but the length and content of such a string is tainted. Also, here there is no bound checking on the "filename" buffer.

## sprintf()



```
void func(const char *name) {  
    char filename[128];  
    sprintf(filename, "%.123s.txt", name);  
}
```



```
void func(const char *name) {  
    std::string filename = (std::string)name + ".txt";  
}
```



A solution is to specify the max number of characters to read from "name" in the format string: "%.123s.txt". This solution is highly error-prone, since the number in the format string must not be greater than the buffer capacity minus the other characters written in the string (i.e., ".txt") minus 1. Moreover, in the presence of other specifiers whose length is decided at runtime (e.g., an integer with %d), counting the other characters written in the string could be difficult. In this case, the max "name" length should be computed basing on the worst case, corresponding to the max possible number of other characters written in the string.

If you can use standard C++ library, maybe the best solution is to use std::string and the + operator to concatenate, and possibly convert the std::string to (constant) C string with c\_str() method.

# strcpy()



```
void func(const char* str) {  
    char str2[128];  
    strcpy(str2, str);  
    /* ... */  
}
```



19

The standard function `strcpy()` copies a series of characters from a source string to a destination string, until a terminator is found (and copied). This program copies the C-string argument “str” to the C-string “str2”. The length of “str” could go beyond the capacity of “str2”, causing a buffer overflow.

# strcpy()



```
void func(const char* str) {  
    char str2[128];  
    strncpy(str2, str, 128);  
    str2[127] = '\0';  
    /* ... */  
}
```

MANUALLY PUT TERMINATOR!



A solution is to use the strncpy() standard function in place of strcpy(), which lets the programmer specify the maximum number of characters to copy. The strncpy() function has the peculiarity that, if the maximum number of copied characters is copied, the destination string is left without terminator. This could lead to further vulnerabilities. To avoid non-terminated strings it is always needed to “manually” put a terminator at the end of the destination buffer.

C and C++ Secure Coding

# FORMAT STRINGS

21

## Catch the bug!

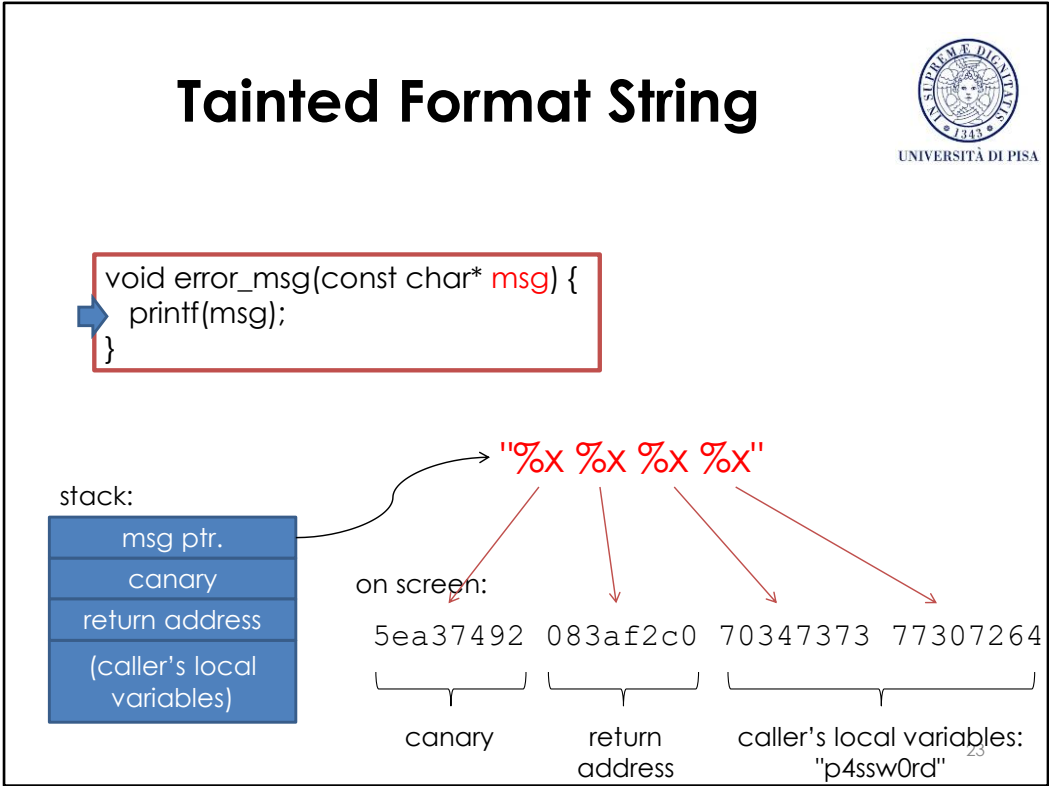


```
void error_msg(const char* msg) {  
    printf(msg);  
}
```



22

This function prints the string “msg” on screen using printf(). The programmer is supposed to pass to the printf() function the same number of optional arguments as the number of specifiers in the format string. However, if the format string is tainted this could not be true, because an attacker could inject an arbitrary number of specifiers.



In this example code, if an attacker gives a message “msg” containing some “%x” specifiers, printf() will try to retrieve and print (in hexadecimal digits) some non-existent integer arguments, leading to an undefined behavior. Usually, the actual arguments of a variadic function are stored in the stack, so printf() will write on the screen the current content of the stack, which constitutes a serious information leakage. Indeed, the stack contains the return address of the printf() function, which in turn can reveal the ASLR relocation address of the program. The stack possibly contains the value of the stack canary, too. Moreover, the stack contains also the caller function’s local variables, which could be sensitive information (e.g., passwords).

## Tainted Format String



- Other advanced techniques allow an attacker to write on arbitrary memory locations → Integrity vulnerability!



# Tainted Format String



```
void error_msg(const char* msg) {  
    printf("%s", msg);  
}
```




25

Tainted data should never be used as format string. This example code shows how to use printf() correctly.


## Tainted Format String

```
void func(const char* str) {  
    if (strlen(str) > 20) { /* Handle error */ }  
    char buf[100];  
    sprintf(buf, str);  
}
```

← BUFFER OVERFLOW!



```
void func(const char* str) {  
    if (strlen(str) > 20) { /* Handle error */ }  
    char buf[100];  
    sprintf(buf, "%s", msg);  
}
```



This example code takes a C-string “str” as argument, checking that it is at most 20 characters long, and then it copies it to a larger buffer “buf” with the variadic function `sprintf()`. However, due to the possible presence of wildchars, the string actually copied in “buf” could be much longer than 20 characters. For example, if the attacker injects the format specifier “%0200x”, it will be expanded in a 200-digits hexadecimal number (padded with zeros), thus causing a buffer overflow on “buf”. Also here, tainted data should never be used as format string.