Format String Vulnerability

Lecture 8

Instructor: C. Pu (Ph.D., Assistant Professor)

puc@marshall.edu

Introduction

- ***printf***(): print out a string according to a format

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

  - 1st arg: *format string* (defines how string should be formatted)
  - format string uses placeholders marked by % character
    - replacing placeholder with data during the printing
- format strings in other functions:
  - *sprintf*(), *fprintf*(), and *scanf*()
- user can provide the entire or part of the contents in a format string
  - *format string vulnerability*: if contents are not sanitized, adversary can get program to run arbitrary code

Introduction

- ***printf***() accepts any number of args
  - (unlike other functions taking a fixed # of args)
  - ref: https://www.cplusplus.com/reference/cstdio/printf/

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

  - writes the string pointed by *format* to the standard output (stdout)
  - if *format* includes format specifiers (subsequences beginning with %), the additional arguments following format are formatted and inserted in the resulting string replacing their respective specifiers
- e.g.,

```
#include <stdio.h>
void main() {
   int i = 1, j = 2, k = 3;
   printf(“hello world \n”);
   printf(“print 1 number: %d\n”, i);
   printf(“print 2 numbers: %d, %d\n”, i, j);
   printf(“print 3 numbers: %d, %d, %d\n”, i, j, k);
}
```

Introduction

- ***printf***() accepts any # of args
- how can ***printf***() achieve that?
  - if a function requiring three args, but two args are provided, no error?
  - compiler never complain about ***printf***(), regardless of how many args are provided

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

  - one concrete arg, *format*
  - 3 dots (…)
    - indicating ***zero*** or ***more*** optional args

How to Access Optional Args

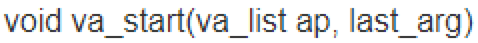
- when a function is defined with a fixed # of arguments
  - each of its arguments is represented by a variable
  - access arguments using their names
- optional arguments do not have names. how ***printf***() access arguments?
  - in C, most functions with *a variable # of args* access optional arguments using the *stdarg* macros defined in the *stdarg.h* header file

a macro is a fragment of code that is given a name.

  - ref: https://www.tutorialspoint.com/c_standard_library/stdarg_h.htm

stdarg.h

- ***stdarg.h*** header defines a variable type ***va_list*** and three macros which can be used to get the args in a function when the # of args are not known (variable # of args).
- ***va_list***
  - a type suitable for holding info. needed by three macros ***va_start***(), ***va_arg***(), and ***va_end***()
- ***va_start***()

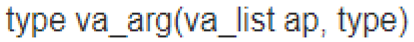


```
void va_start(va_list ap, last_arg)
```

  - initializes *ap* variable to be used with the ***va_arg*** and ***va_end*** macros
  - the *last_arg* is the last known fixed argument being passed to the function i.e. the argument before the ellipsis

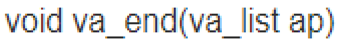
stdarg.h

- ***stdarg.h*** header defines a variable type ***va_list*** and three macros which can be used to get the args in a function when the # of args are not known (variable # of args).
- ***va_arg***()



```
type va_arg(va_list ap, type)
```

  - retrieves the next argument in the parameter list of the function with *type* type
- ***va_end***()

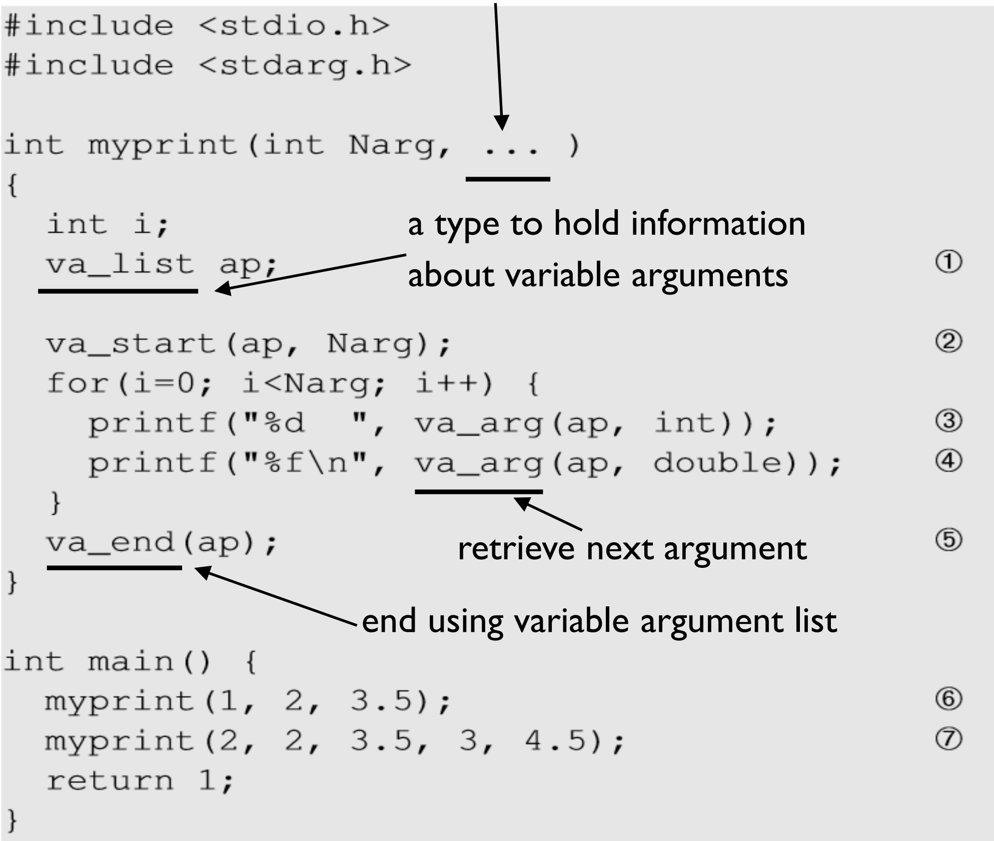


```
void va_end(va_list ap)
```

  - allows a function with variable arguments which used the va_start macro to return
  - if va_end is not called before returning from the function, the result is undefined

Access Optional Arguments

a list of unnamed arguments whose number and types are not known to the called function.



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

int myprint(int Narg, ... )
{
  int i;
  va_list ap;                                  ①

  va_start(ap, Narg);                          ②
  for(i=0; i<Narg; i++) {
    printf("%d  ", va_arg(ap, int));           ③
    printf("%f\n", va_arg(ap, double));        ④
  }
  va_end(ap);                                  ⑤
}

int main() {
  myprint(1, 2, 3.5);                          ⑥
  myprint(2, 2, 3.5, 3, 4.5);                  ⑦
  return 1;
}
```

a type to hold information about variable arguments

retrieve next argument

end using variable argument list

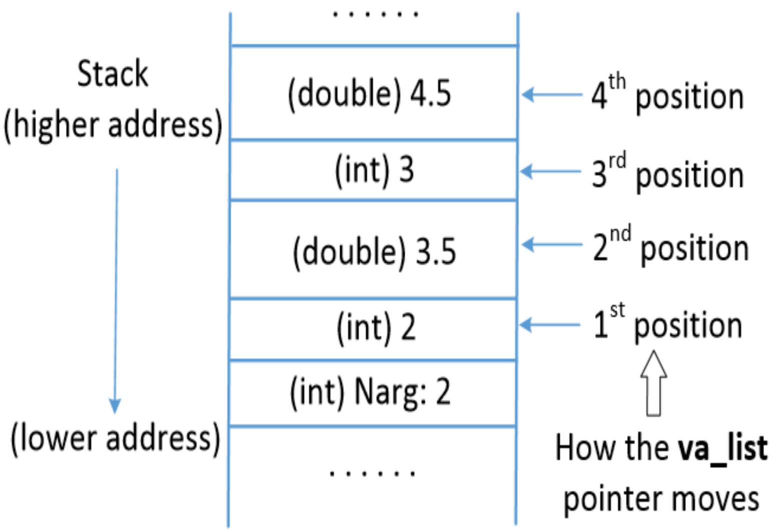
- va_list pointer (line 1) accesses the optional arguments.
- va_start() macro (line 2) calculates the initial position of va_list based on the second argument Narg (last argument before the optional arguments begin)

- void va_start (va_list *ap*, *paramN*)
  - initializes *ap* to retrieve the additional arguments after parameter *paramN*.

- type va_arg (va_list *ap*, *type*)
  - retrieve the value of the current argument in the variable arguments list identified by *ap*.
  - advance to the next argument in the the variable arguments list identified by *ap*.

Access Optional Arguments

```
myprint(1, 2, 3.5);                ⑥
myprint(2, 2, 3.5, 3, 4.5);        ⑦
```



stack layout for myprint(2, 2, 3.5, 3, 4.5);

- when myprint() is invoked (line ⑥ and ⑦)
  - all arguments are pushed into the stack
  - va_list is used to access the optional args

```
va_start(ap, Narg);        ②
```

- va_start() (line ②) calculates the initial position of va_list based on the Narg
- to access the optional args pointed by va_list, we need to use va_arg()

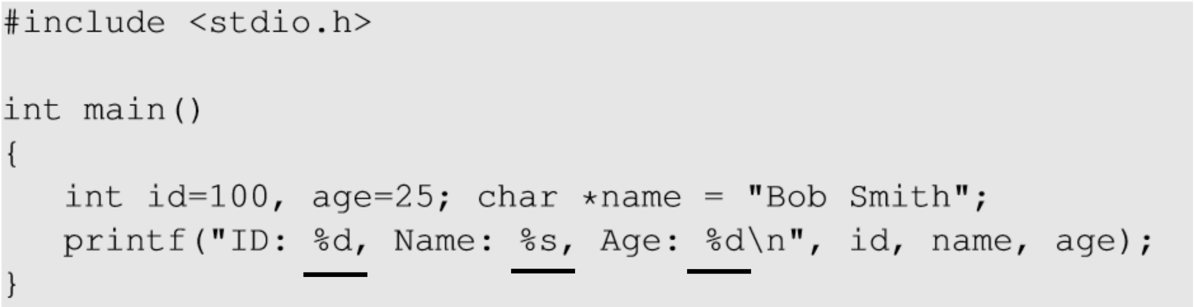
```
va_arg(ap, int)
va_arg(ap, double)
```

va_list pointer

the type of optional arg to be accessed

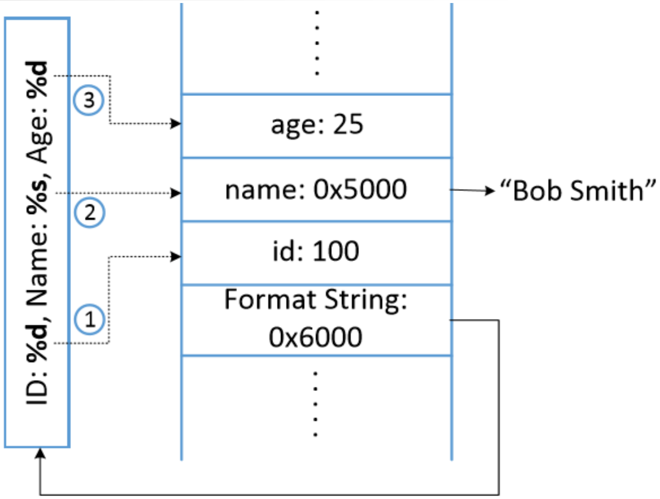
  - return the value pointed by the va_list pointers
  - advances the pointer to where the next optional arg is stored
- finish up by calling `va_end(ap);`

How printf() Access Optional Arguments



```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

- printf() also uses the stdarg macros
- Q: how it know the type of arg?
- Q: how it know the end of arg list?
- here, printf() has three optional arguments.
  - elements starting with "%" are called *format specifiers*.
- printf() scans the format string and prints out each character until "%" is encountered.
- printf() calls va_arg(), which returns the optional argument pointed by va_list and advances it to the next argument.



- when printf() is called
  - all arguments are pushed into stack
- when scanning and printing
  - replace the 1st format specifier with the value from the first optional arg.
  - the same idea will be applied to other args

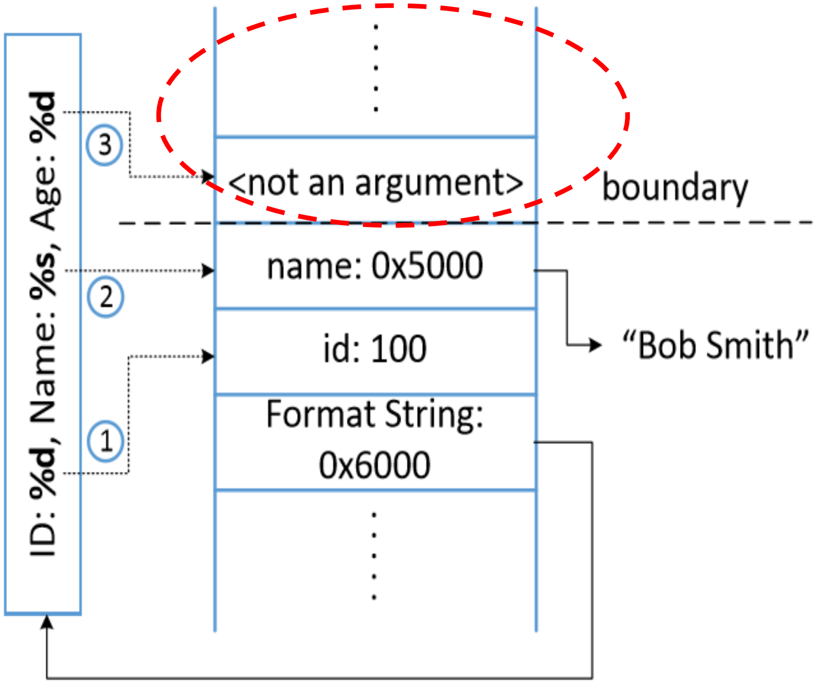
Missing Optional Arguments

- printf() uses the # of format specifiers to determine the # of optional args.
- what if a programmer makes a mistake:

the # of optional args ≠ the # of format specifiers

```
#include <stdio.h>

int main()
{
    int id=100, age=25; char *name = "Bob Smith";

    printf("ID: %d, Name: %s, Age: %d\n", id, name);
}
```

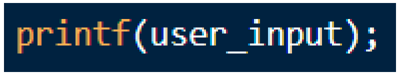
- three format specifiers vs. two optional args
  - cannot be caught by compiler
- at runtime, detecting mismatches require boundary marking on the stack
  - detecting when it reaches the last optional arg



- printf() relies on va_arg() to fetch optional args from stack
  - when va_arg() is called
    - the value of arg is fetched
    - advance to next arg
  - va_arg() doesn't know whether it has reached the end of optional args list
    - if called again, va_arg() continues fetching data from stack (even though the data is not optional arg)

Format String Vulnerability

- if there is a *mismatch* in a format string
  - the # of optional args ≠ the # of format specifiers
  - print out incorrect information and cause some problems
  - does not pose any severe threat
    - it might be true *if the mismatch comes from programmer*
- *if a format string comes from malicious users*
  - the damage can be far worse than what we can expect
  - *format string vulnerability*

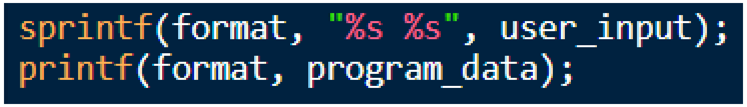


```
printf(user_input);
```

- print out some data provided by users, *user_input*
- what if *user_input* has *format specifiers*
- correct way: printf("%s", *user_input*);

Format String Vulnerability

- if there is a *mismatch* in a format string
  - the # of optional args ≠ the # of format specifiers
  - print out incorrect information and cause some problems
  - does not pose any severe threat
    - it might be true *if the mismatch comes from programmer*
- *if a format string comes from malicious users*
  - the damage can be far worse than what we can expect
  - *format string vulnerability*

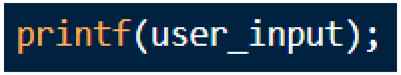


```
sprintf(format, "%s %s", user_input);
printf(format, program_data);
```

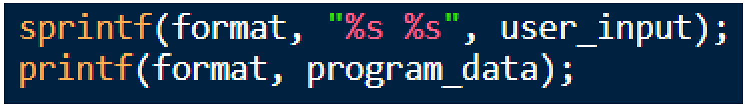
- print out some user-provided information, along with data generated from program
- users may place some format specifiers in their input

Format String Vulnerability

- if there is a *mismatch* in a format string
    - the # of optional args ≠ the # of format specifiers
    - print out incorrect information and cause some problems
    - does not pose any severe threat
        - it might be true *if the mismatch comes from programmer*
- *if a format string comes from malicious users*
    - the damage can be far worse than what we can expect
    - *format string vulnerability*



```
printf(user_input);
```

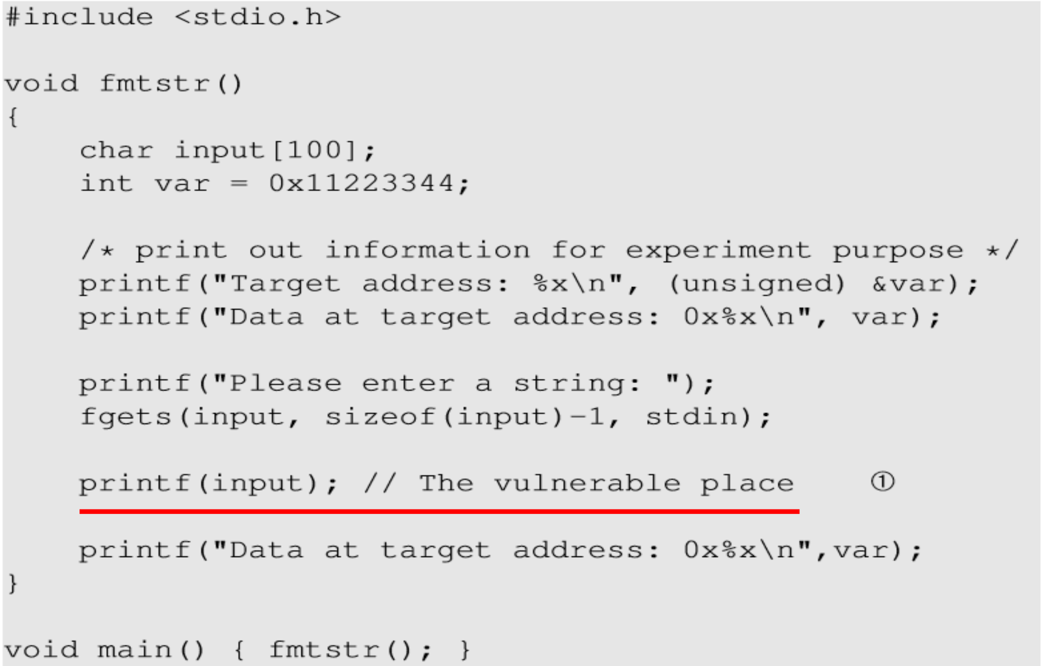


```
sprintf(format, "%s %s", user_input);
printf(format, program_data);
```

- in these two examples, user's input (user_input) becomes part of a format string.
- what will happen if user_input contains format specifiers?

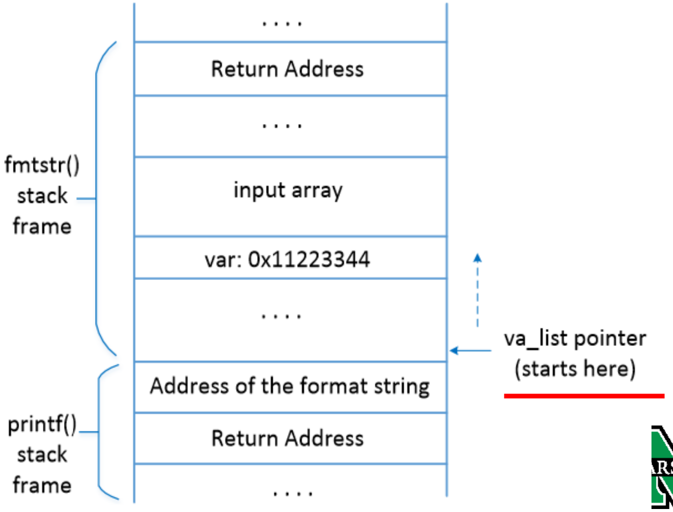
Vulnerable Code

- vulnerable program
  - function fmtstr()
    - take user input
    - print out the input



```
#include <stdio.h>

void fmtstr()
{
    char input[100];
    int var = 0x11223344;

    /* print out information for experiment purpose */
    printf("Target address: %x\n", (unsigned) &var);
    printf("Data at target address: 0x%x\n", var);

    printf("Please enter a string: ");
    fgets(input, sizeof(input)-1, stdin);

    printf(input); // The vulnerable place     ①

    printf("Data at target address: 0x%x\n",var);
}

void main() { fmtstr(); }
```

- char *fgets(char **str*, int *n*, FILE **stream*)
  - *str*: this is the pointer to an array of chars where the string read is stored.
  - *n*: this is the maximum number of characters to be read (including the final null-character). usually, the length of the array passed as str is used.
  - *stream*: this is the pointer to a FILE object that identifies the stream where characters are read from.

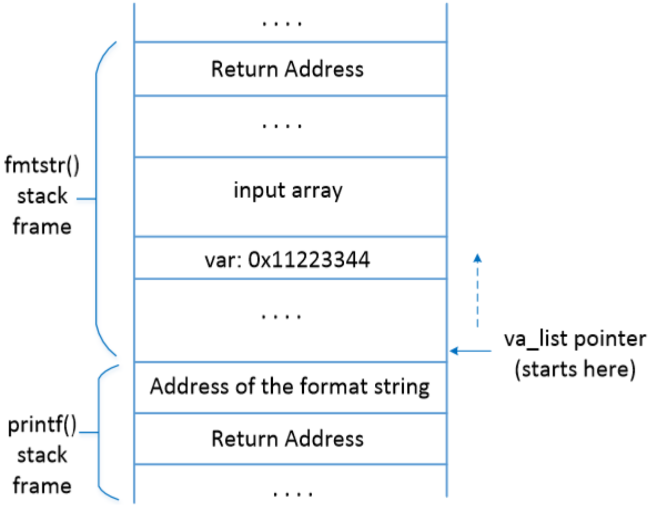


Exploiting Format String Vulnerability

- format string vulnerability allows attackers to do a wide variety of damages
  - crash a program
  - steal secret data from a program
  - modify a program’s memory
  - get a program to run attacker’s malicious code

Attack I: Crash Program

```
$ ./vul
.......
Please enter a string: %s%s%s%s%s%s%s%s
Segmentation fault (core dumped)
```



....

Return Address

....

input array

var: 0x11223344

....

Address of the format string

Return Address

....

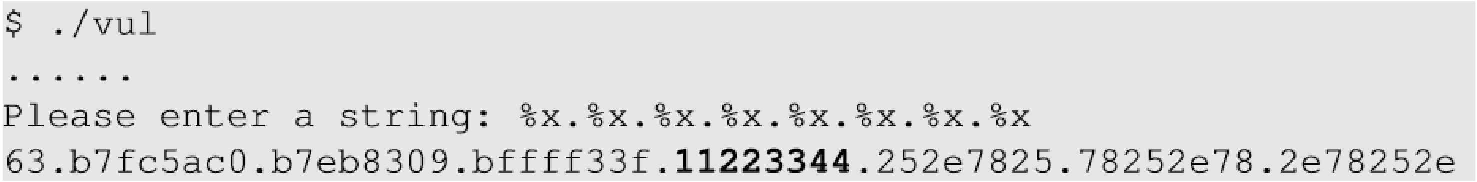
fmtstr() stack frame

printf() stack frame

va_list pointer (starts here)

- printf() does not include any optional argument, `printf(input);`
- if we put several format specifiers in the input, we can get printf() to advance its va_list pointer to the places beyond the printf() function's stack frame
- use input: %s%s%s%s%s%s%s%s
- printf() parses the format string
  - for each %s, it fetches a value where va_list points to and advances va_list to the next position
  - as we give %s, printf() treats the value as address and fetches data from that address
    - if the value is not a valid address, the program crashes

Attack 2:
Print Out Data on the Stack

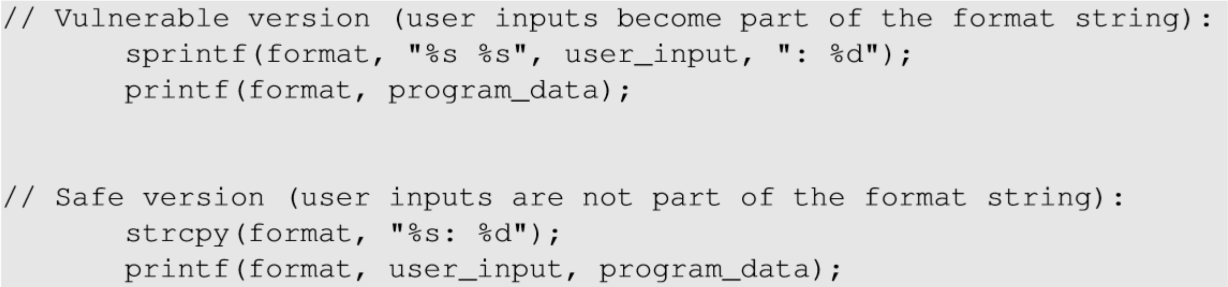


```
$ ./vul
......
Please enter a string: %x.%x.%x.%x.%x.%x.%x.%x
63.b7fc5ac0.b7eb8309.bffff33f.11223344.252e7825.78252e78.2e78252e
```

- suppose a variable on the stack contains a secret (constant) and we need to print it out
  - assume that the *var* variable contains a secret (dynamically generated)
- use user input: %x.%x.%x.%x.%x.%x.%x.%x
  - printf() prints out the integer value pointed by va_list pointer and advances it by 4 bytes
  - the number of %x is decided by the distance between the starting point of the va_list pointer and the variable
    - it can be achieved by trial and error

Countermeasures: Developer

- format string are used by many other functions
  - e.g., fprintf(), springf(), snprintf(), vprintf(), vfprintf(), vsprintf(), and vsnprintf()
  - those are C functions; other languages have similar functions that use format strings
- good program habit: avoid using untrusted user inputs for format strings in functions like printf, sprintf, fprintf, vprintf, scanf, vfscanf

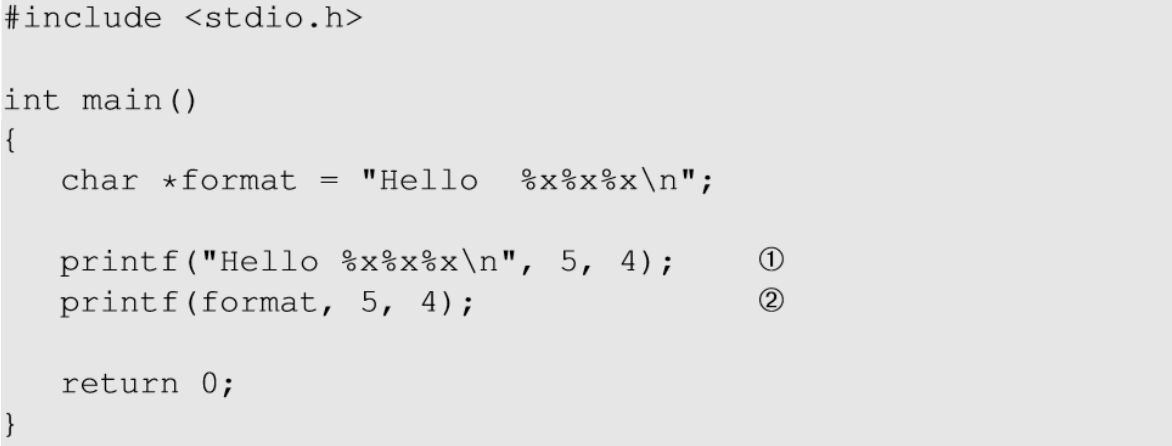


```
// Vulnerable version (user inputs become part of the format string):
       sprintf(format, "%s %s", user_input, ": %d");
       printf(format, program_data);


// Safe version (user inputs are not part of the format string):
       strcpy(format, "%s: %d");
       printf(format, user_input, program_data);
```

  - ask users for data input, but not for code

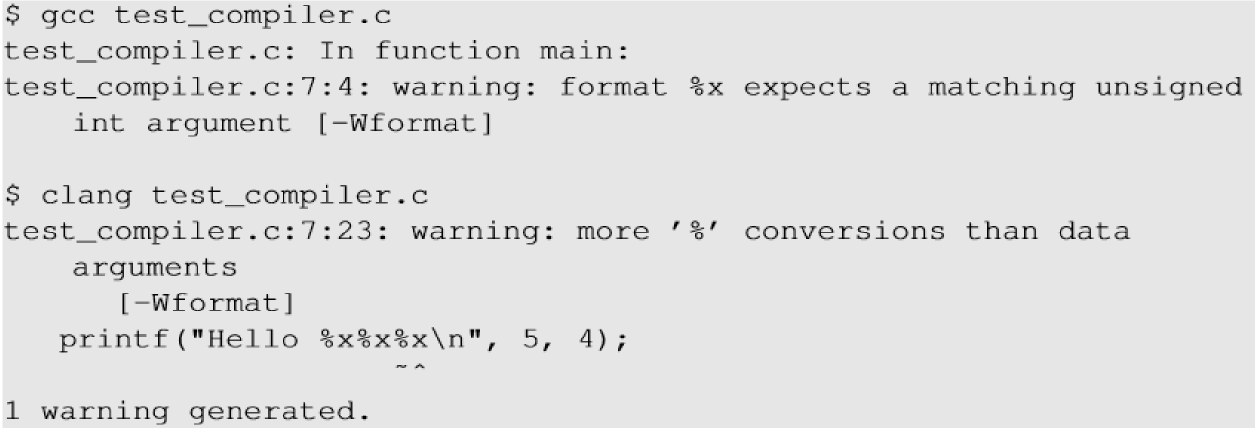
Countermeasures: Compiler

- compilers can detect potential format string vulnerabilities



```
#include <stdio.h>

int main()
{
   char *format = "Hello  %x%x%x\n";

   printf("Hello %x%x%x\n", 5, 4);     ①
   printf(format, 5, 4);               ②

   return 0;
}
```

- use two compilers to compile the program: gcc and clang
- we can see that there is a mismatch in the format string (line ①)
- none of them report line ②

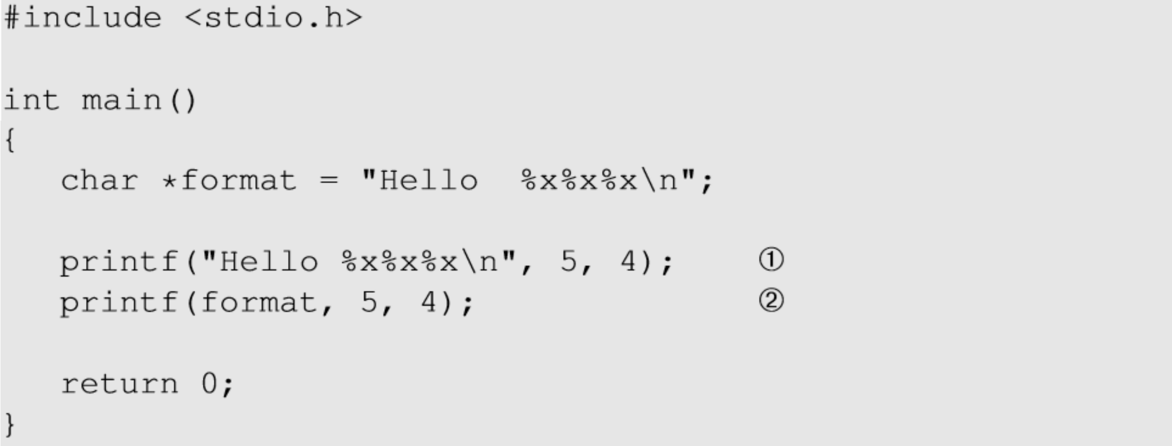


```
$ gcc test_compiler.c
test_compiler.c: In function main:
test_compiler.c:7:4: warning: format %x expects a matching unsigned
    int argument [-Wformat]

$ clang test_compiler.c
test_compiler.c:7:23: warning: more '%' conversions than data
    arguments
      [-Wformat]
   printf("Hello %x%x%x\n", 5, 4);
                     ~^

1 warning generated.
```

- with default settings, both compilers gave warning for the first printf()
- no warning was given out for the second one

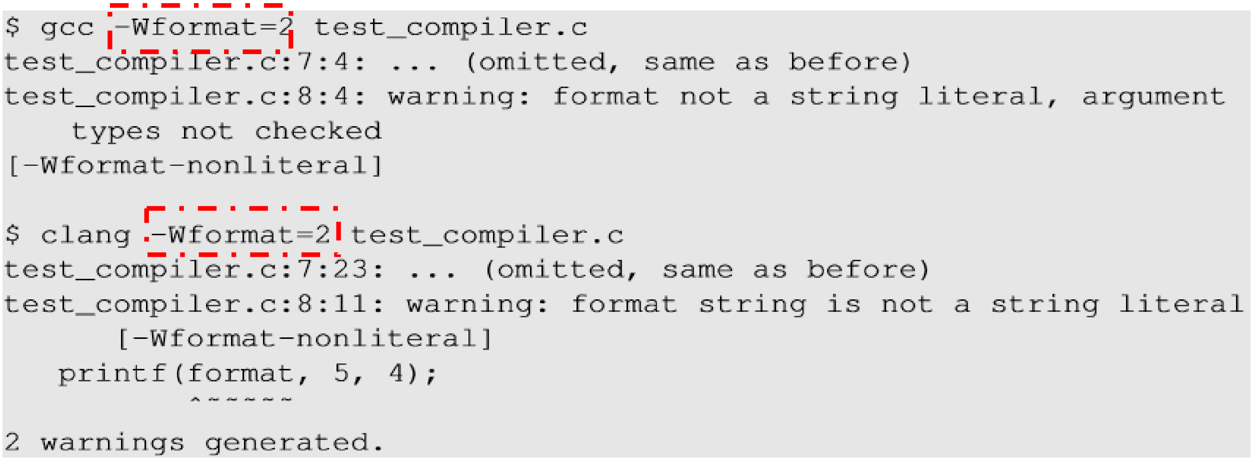
Countermeasures: Compiler

- compilers can detect potential format string vulnerabilities



```
#include <stdio.h>

int main()
{
   char *format = "Hello  %x%x%x\n";

   printf("Hello %x%x%x\n", 5, 4);     ①
   printf(format, 5, 4);               ②

   return 0;
}
```

- use two compilers to compile the program: gcc and clang
- we can see that there is a mismatch in the format string (line ①)



```
$ gcc -Wformat=2 test_compiler.c
test_compiler.c:7:4: ... (omitted, same as before)
test_compiler.c:8:4: warning: format not a string literal, argument
    types not checked
[-Wformat-nonliteral]

$ clang -Wformat=2 test_compiler.c
test_compiler.c:7:23: ... (omitted, same as before)
test_compiler.c:8:11: warning: format string is not a string literal
       [-Wformat-nonliteral]
   printf(format, 5, 4);
          ^~~~~~

2 warnings generated.
```

- if we attach –Wformat=2 option in compiler command, both of them warm the developer
  - format string vulnerability