Format String Vulnerability

Lecture 15

Instructor: Dr. Cong Pu, Ph.D.

cong.pu@okstate.edu

Acknowledgment: Adapted partially from course materials from Dr. Wenliang Du at Syracuse University, Dr. Fengwei Zhang at Southern University of Science and Technology, and Dr. Steven M. Bellovin at Columbia University.

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 *placeholder* % character
    - replacing *placeholder* % with data during printing
      - data are from …
- format strings in other functions:
  - *sprintf*(), *fprintf*(), and *scanf*()

    int ***sprintf*** (char *str, const char *format, ...); ⟹ write to buffer

    int ***fprintf*** (FILE *stream, const char *format, ...); ⟹ write to file

    int ***scanf*** (const char *format, ...); ⟹ read from input

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 *placeholder* % character
    - replacing placeholder with data during printing
      - data are from …
- format strings in other functions:
  - *sprintf*(), *fprintf*(), and *scanf*()
- users 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 # of args* (unlike other functions)

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

ref.: https://www.cplusplus.com/reference/cstdio/printf/

  - writes the string pointed by *format* to the standard output (stdout)
    - typically, the terminal or console where the program is being executed
  - if *format* includes format specifiers or placeholders (%), the additional arguments following format are *formatted and inserted in the resulting string replacing their respective specifiers or placeholders*

Introduction

- ***printf***() accepts *any # of args* (unlike other functions)

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

ref.: https://www.cplusplus.com/reference/cstdio/printf/

- e.g., #include <stdio.h> ⟵ Standard input/output library for performing input and output operations

```
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);
}
```

⟵ no additional arg

⟵ 1 additional arg

⟵ 2 additional args

3 additional args

Introduction

- ***printf***() accepts *any # of args* (unlike other functions)

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

ref.: https://www.cplusplus.com/reference/cstdio/printf/

- How can ***printf***() achieve that, accepting *any # of args*?
  - if a function requiring three (3) args, but two (2) args are provided, *no error*?
  - compiler never complain about ***printf***(), regardless of how many args are provided
  - 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 args

```
// Function definition
int addTwoNumbers(int a, int b) {
    return a + b; // Adds two fixed arguments
}
```

  - *each of its args* is represented by a *variable*
  - access args using their names
- Optional args do not have names, how ***printf***() access arguments? `int printf(const char *format, ...);`
  - in C, most functions with *a variable # of args* access optional args using the ***stdarg*** macros defined in *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 in C provides a way to work with *variadic functions* that *accept a variable number of args*.
  - key components:
    - ***va_list***: a ***type*** used to declare an *argument pointer*
    - ***va_start()***: initializes the *argument pointer* to the *first variable argument* in the function
    - ***va_arg()***: retrieves the *next argument* in the list
    - ***va_end()***: *cleans up* after the argument pointer
  - use cases:
    - functions like *printf()* and *scanf()* use <stdarg.h> to handle variable arguments
    - custom utility functions, such as logging or dynamic argument processing

stdarg.h

- Key components:
  - *va_list*: a ***type*** used to declare an *argument pointer*
    - acts as *container* for the args passed to variadic function
    - represents the list of args provided after the fixed parameters in function
    - it's initialized with *va_start* and used with *va_arg* to retrieve each argument
    - once done, *va_end* is called to clean up
  - va_start(): initializes the argument pointer to the first variable argument in the function
  - va_arg(): retrieves the next argument in the list
  - va_end(): cleans up after the argument pointer

stdarg.h

- Key components:
  - *va_list*: a ***type*** used to declare an *argument pointer*
  - va_start(): initializes the *argument pointer* to the *first variable argument* in the function
    - initializes *va_list* variable to process a variable number of args in function
    - must be called before using *va_arg()* to retrieve args

```
void va_start(va_list ap, last_fixed_arg);
```

      - ap: the *va_list* variable that will be used to access the args
      - last_fixed_arg: the last named (fixed) argument before the variable args start
  - va_arg(): retrieves the next argument in the list
  - va_end(): cleans up after the argument pointer

stdarg.h

- Key components:
  - *va_list*: a ***type*** used to declare an *argument pointer*
  - va_start(): initializes the *argument pointer* to the *first variable argument* in the function
  - va_arg(): retrieves the *next argument* in the list
    - retrieve the next arg from *va_list*
    - each call to *va_arg()* advances the list to the next arg

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

      - ap: the *va_list* variable that was initialized using *va_start()*
      - type: the ***expected data type*** of the arg (e.g., int, double, char *)
  - va_end(): cleans up after the argument pointer

stdarg.h

- Key components:
  - *va_list*: a **type** used to declare an *argument pointer*
  - va_start(): initializes the *argument pointer* to the *first variable argument* in the function
  - va_arg(): retrieves the *next argument* in the list
  - va_end(): cleans up after the arg pointer
    - clean up *va_list* after processing a variable number of args in function
    - ensures proper resource management and should always be called after *va_start()* and *va_arg()*

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

      - ap: the *va_list* variable that was previously initialized with *va_start()*

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*.

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

- 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*.

stdarg.h

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

// Variadic function to calculate the sum of arguments
int sum(int count, ...) {
    va_list args;
    int total = 0;
    va_start(args, count);  // Initialize args to start at the first variable argument
    for (int i = 0; i < count; i++) {
        total += va_arg(args, int);  // Retrieve each argument as an int
    }
    va_end(args);  // Clean up
    return total;
}

int main() {
    printf("Sum of 2, 4, and 6: %d\n", sum(3, 2, 4, 6));
    return 0;
}
```