# Format String Vulnerability

Lecture 16

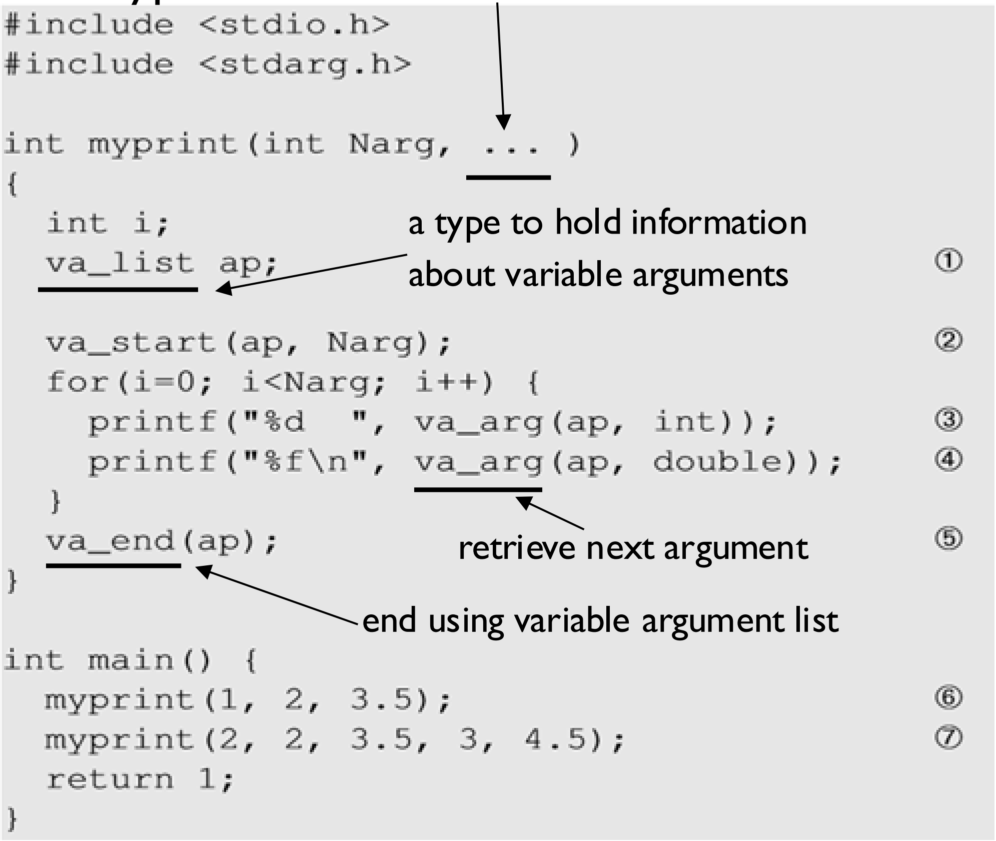
Instructor: Dr. Cong Pu, Ph.D.

*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.*

# 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

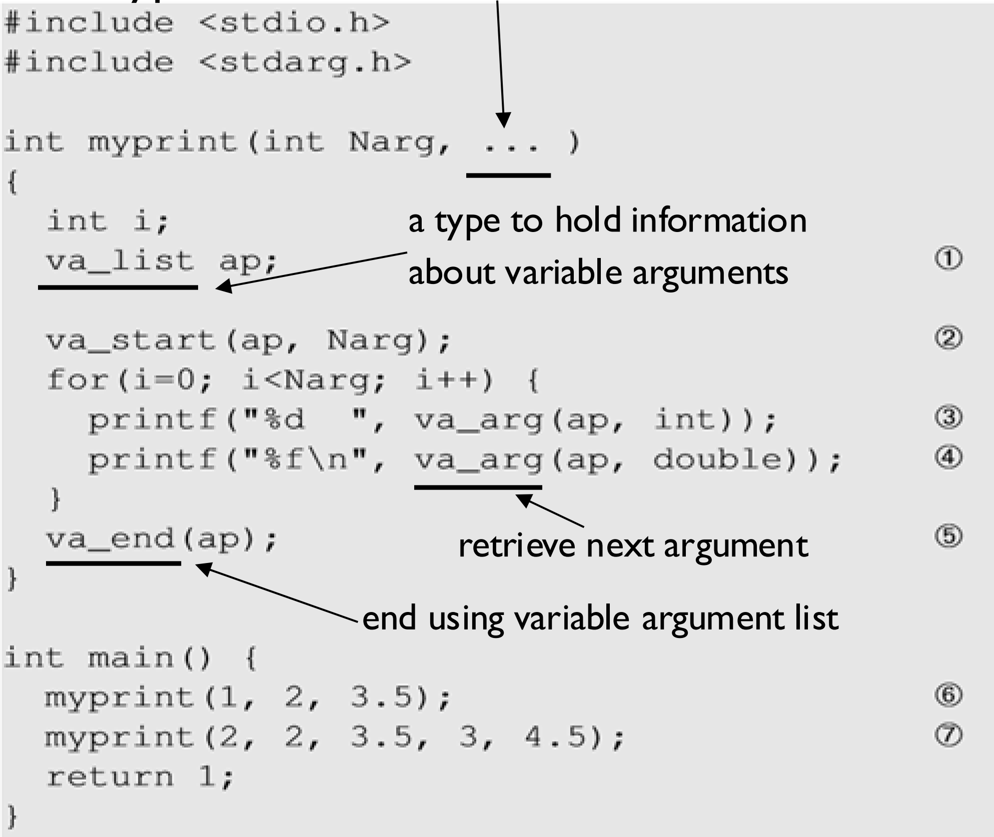
retrieve next argument

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

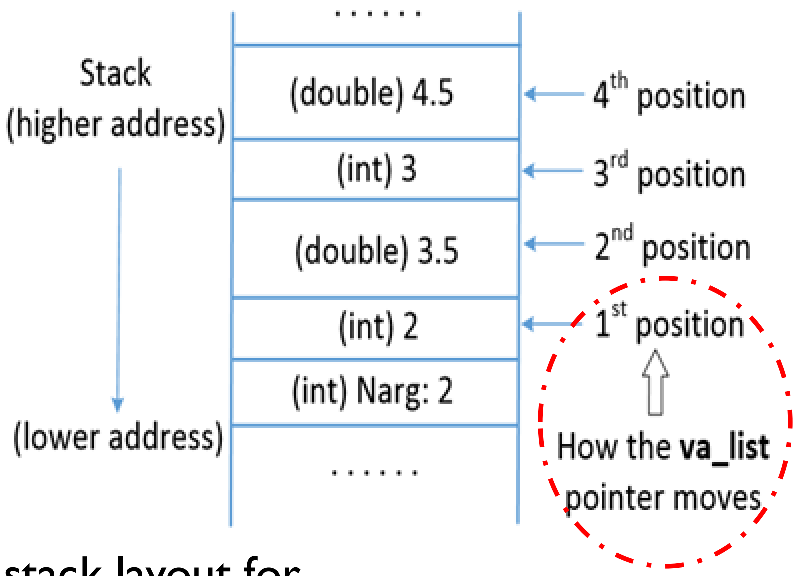
retrieve next argument

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*.

# 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);                        ⑦
```

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

```
va_start(ap, Narg);          ②
```

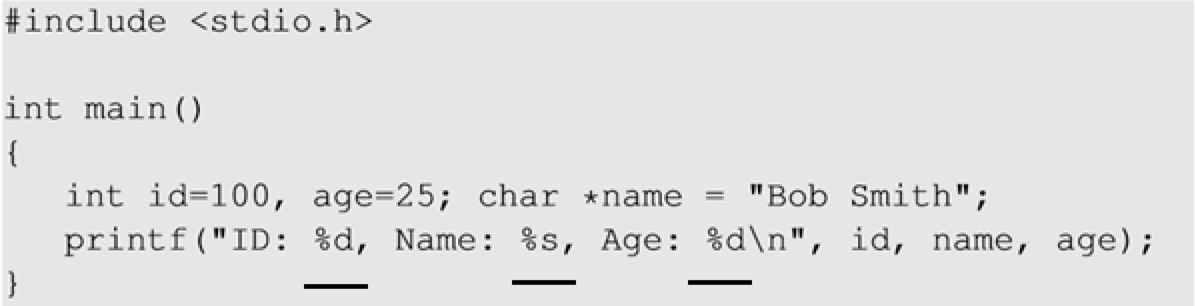
- *va_start()* (line ②) calculates the *initial position* of *va_list* based on the Narg
- to access the optional args pointed by *va_list*, we need to use *va_arg()*

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

*va_list* pointer → ap ← the *<u>type</u>* of optional arg to be accessed

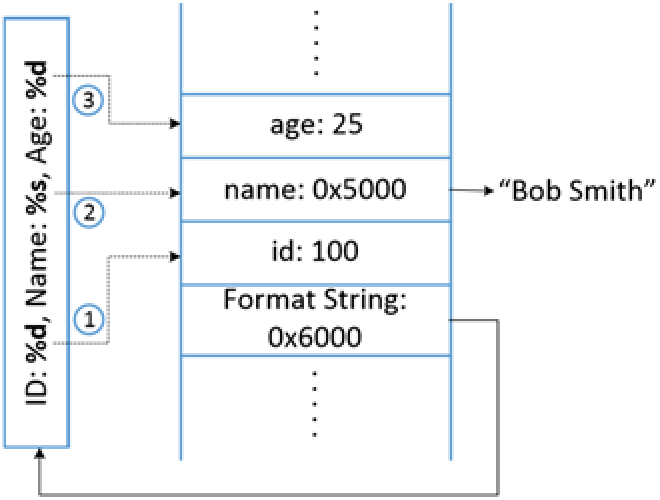
  - *return* the value pointed by the *va_list* pointers
  - *advances (how much)* the pointer to where the next optional arg is stored
- finish up by calling `va_end(ap);`

# 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 (3)* 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 arg pointed by *va_list* and advances it to next arg
  - *type?* -- type field of format specifier

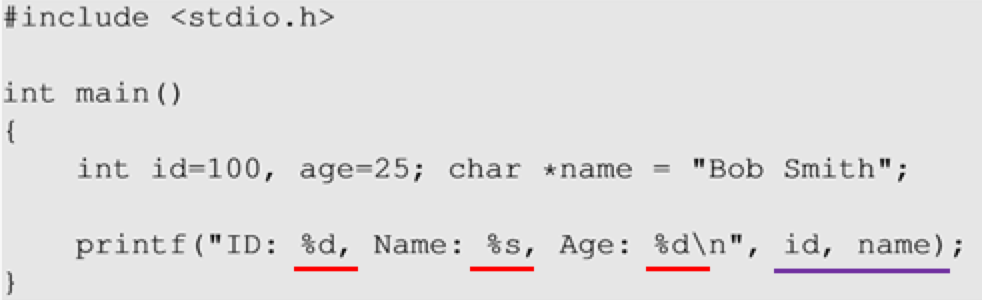


- when *printf()* is called
  - all arguments are pushed into stack
- when scanning and printing
  - replace the 1st format specifier % with the value from the first optional arg
  - the same idea will be applied to other args

# Missing Optional Arguments

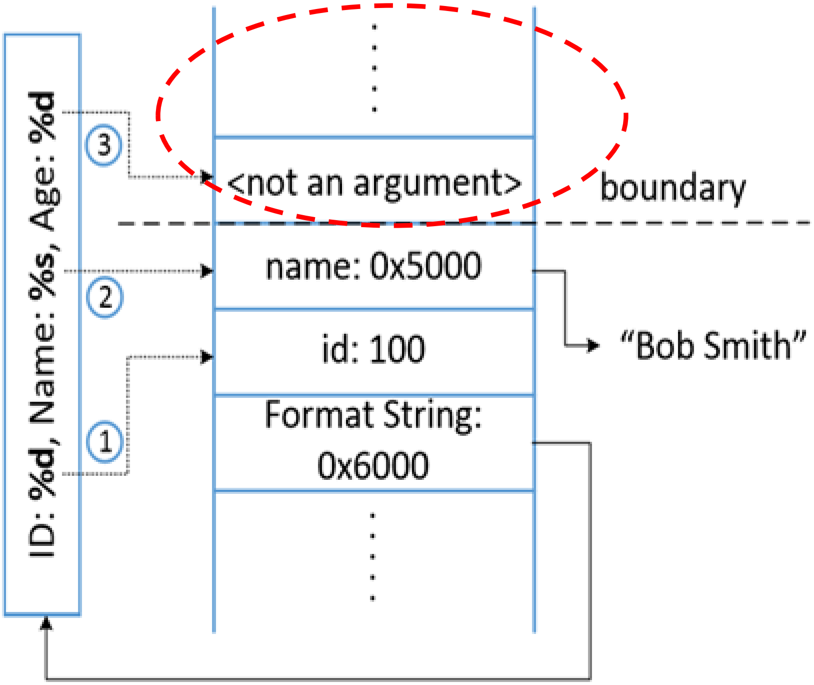
- *printf()* uses the # of format specifiers to determine the # of optional args
- what if a programmer makes a ***mistake***:

*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);
}
```

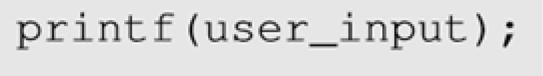
- *three (3)* format specifiers % vs. *two (2)* optional args
  - cannot be caught by compiler
- at runtime, detecting mismatches require boundary marking on the stack
  - detecting when it reaches the last optional arg
  - *Unfortunately, no such marking in the system*



- *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*



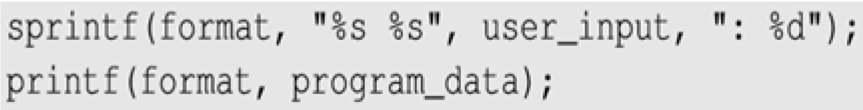
```
printf(user_input);
```

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

*no format specifier*

# 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*

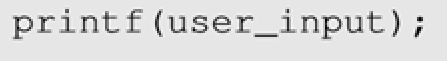


```
sprintf(format, "%s %s", user_input, ": %d");
printf(format, program_data);
```

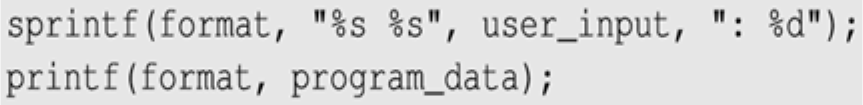
- print out some user-provided information, along with data generated from program
- users may place some format specifiers in their input

# 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);
```

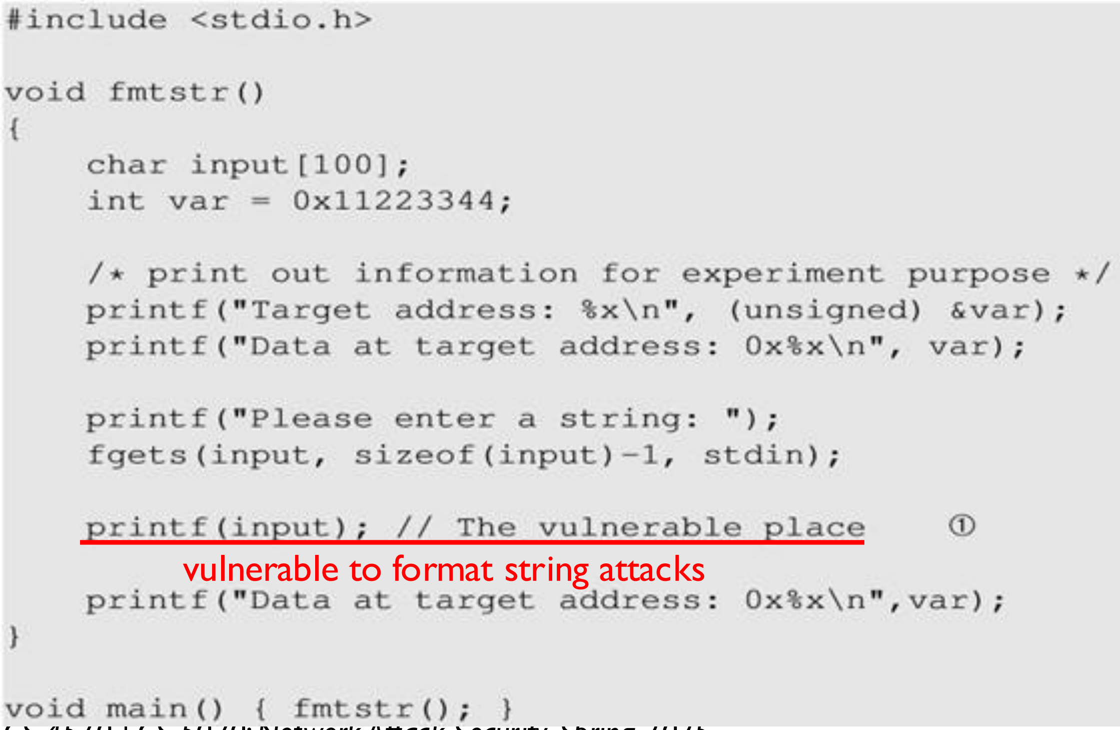


```
sprintf(format, "%s %s", user_input, ": %d");
printf(format, program_data);
```

- 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

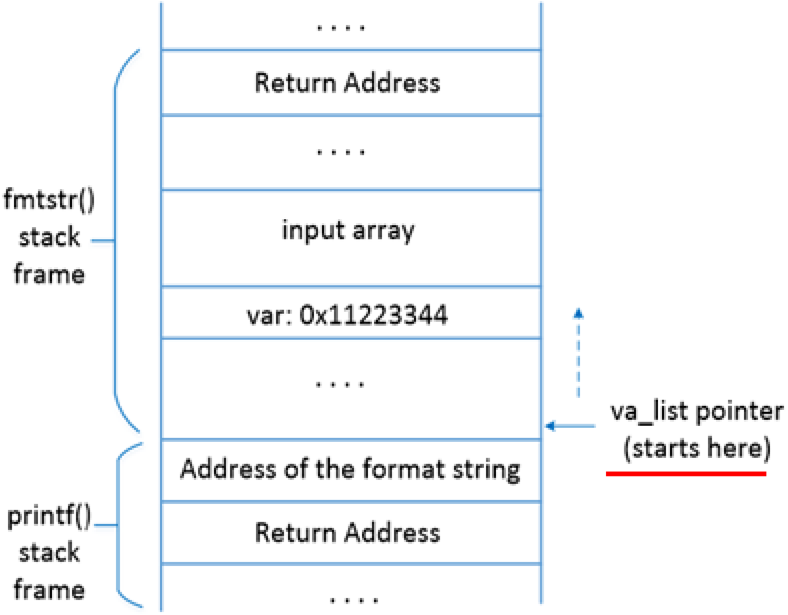
- *vulnerable program*
  - function *fmtstr()*
    - take user input
    - print out the input



```
#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(); }
```

vulnerable to format string attacks

- 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.

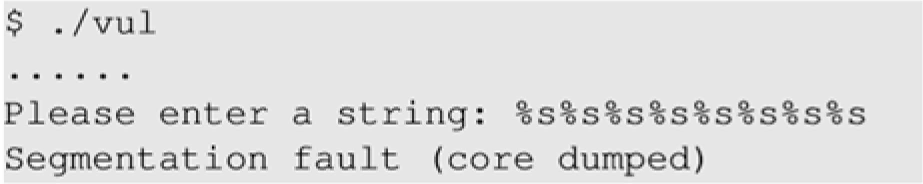


# Exploiting Format String Vulnerability

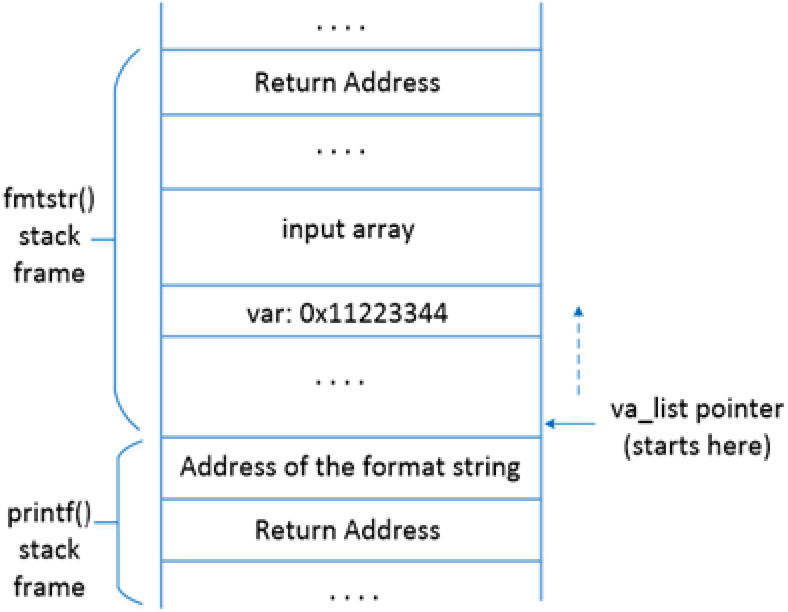
- ***Format string vulnerability*** allows ***attackers*** to do a wide variety of damages
  - crash a program
  - steal secret data from a program
  - modify a program's memory
  - get a program to run attacker's malicious code

```
$ gcc –o vul vul.c
$ sudo chown root vul
$ sudo chmod 4755 vul
$ sudo sysctl -w kernel.randomize_va_space=0
```

# Attack I:
# Crash Program



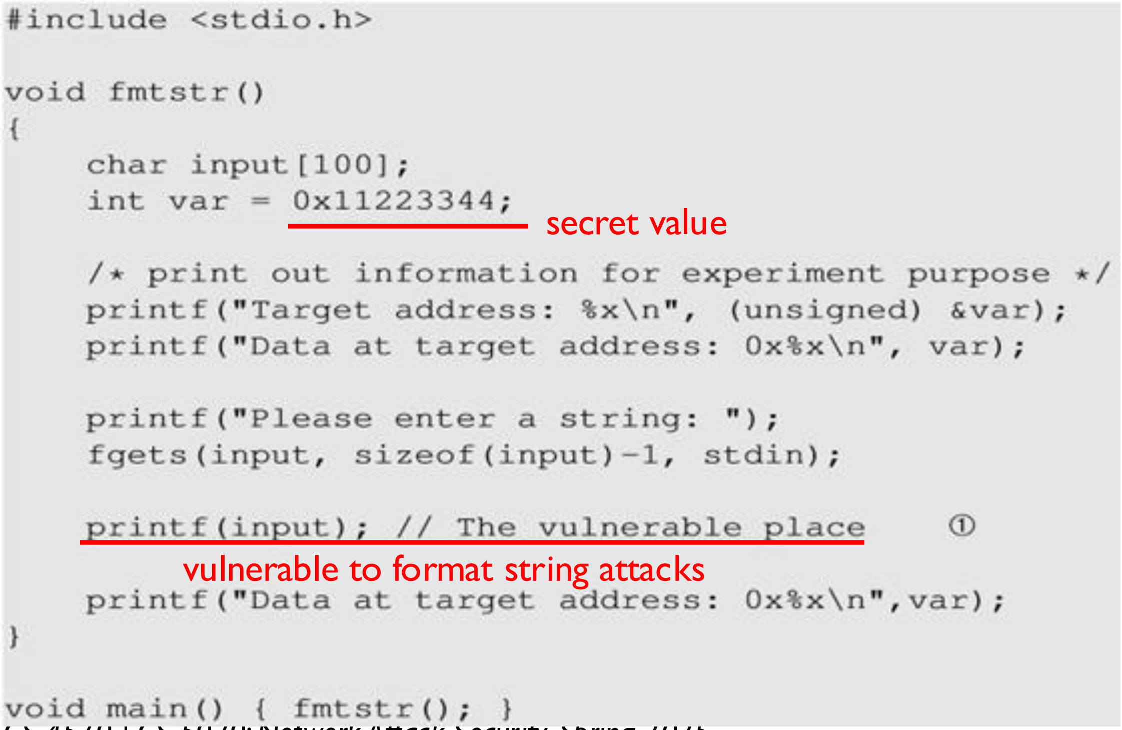
```
$ ./vul
.......
Please enter a string: %s%s%s%s%s%s%s%s
Segmentation fault (core dumped)
```



- *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

# Vulnerable Code

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

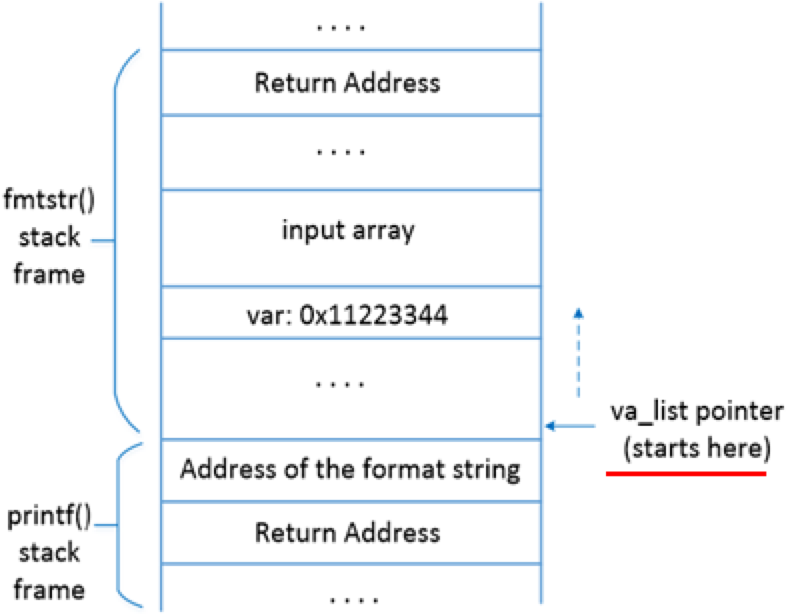


```
#include <stdio.h>

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(); }
```

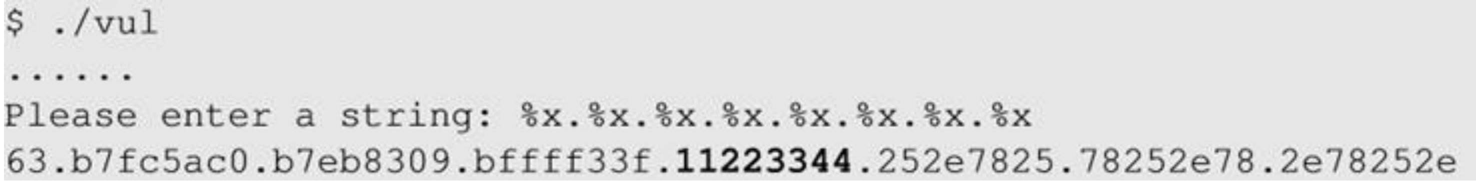
secret value

vulnerable to format string attacks

- 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.



# 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