

Printf-Lab

——by 蔡彦麓

实验原理

什么是Printf漏洞？

```
3 int main()  
4 {  
5     char a[20];  
6     scanf("%s",a);  
7     printf(a);  
8 }
```

```
C:\Program Files (x86)\Vim\vimrun.exe  
C:\WINDOWS\system32\cmd.exe /c ( cs.  
12313  
12313Hit any key to close  
_
```

```
C:\Program Files (x86)\Vim\vimrun.exe  
C:\WINDOWS\system32\cmd.exe /c ( cs.  
%d, %p, %p  
10157804, 009AFEF8, 009AFFCCHit any key  
_
```

Printf函数先读入格式化字符串，然后就会直接从运行栈栈顶，取参数（32位）。

而64位中，前4个参数是存放在寄存器中的（因为寄存器太多了，随便用）。之后才是存放在栈中的。

如何利用printf漏洞呢？

条件：必须可控格式化字符串，即printf的第一个参数，你能控制。如之前的例子就是，直接使用读入作为printf第一个参数。

利用：最初步的利用，就是说使用%d、%p。来让其泄露栈上的地址。但是printf实际上还有更多有趣的格式化字符串符号。

Printf 格式化字符串

- 详细介绍可以参见：
- <https://www.jianshu.com/p/fdb537c40a9d>
- 我们只讲用得到的部分。

%[参数\$][标志][宽度][.精度][长度]类型

- 参数部分：应当形如：K\$。K是一个数字，表示使用第K个参数。
- 参数字段为 POSIX 扩展功能，非 C99 标准定义。请使用linux。

```
whiletruedo@ubuntu: ~  
1 #include<stdio.h>  
2 int main()  
3 {  
4     printf("%2$d %1$d\n",1,2);  
5 }
```

```
PRESS ENTER or  
2 1  
  
Press ENTER or
```

%[参数\$][标志][宽度][.精度][长度]类型

- 宽度部分：直接就是一个整数m，表示至少要占的字符数，超出这个m的部分也会正常显示。

```
#include<stdio.h>
int main()
{
    printf("%32d\n",1);
}
```

```
Press ENTER or type command to continue
1
Press ENTER or type command to continue
```

%[参数\$][标志][宽度][.精度][长度]类型

- 精度部分：以.开头，形如.k形式。无需多讲。
- 实际上printf总共至多出现3次数字。用\$和.即可完全分开（标志部分有一个标志为0，表示用零填充，而宽度部分本不该有前导0，所以也并无歧义）。

%[参数\$][标志][宽度][.精度][长度]类型

- 长度部分（如图）：
- 我们所说的：将char型参数转成int型输出。
- 意思是传的参数实际上在栈上只占1byte。
- 比如同样一个指针：
- p=0x30
- int型的p，就是指0x30~0x33
- char型的p，就单指0x30

长度

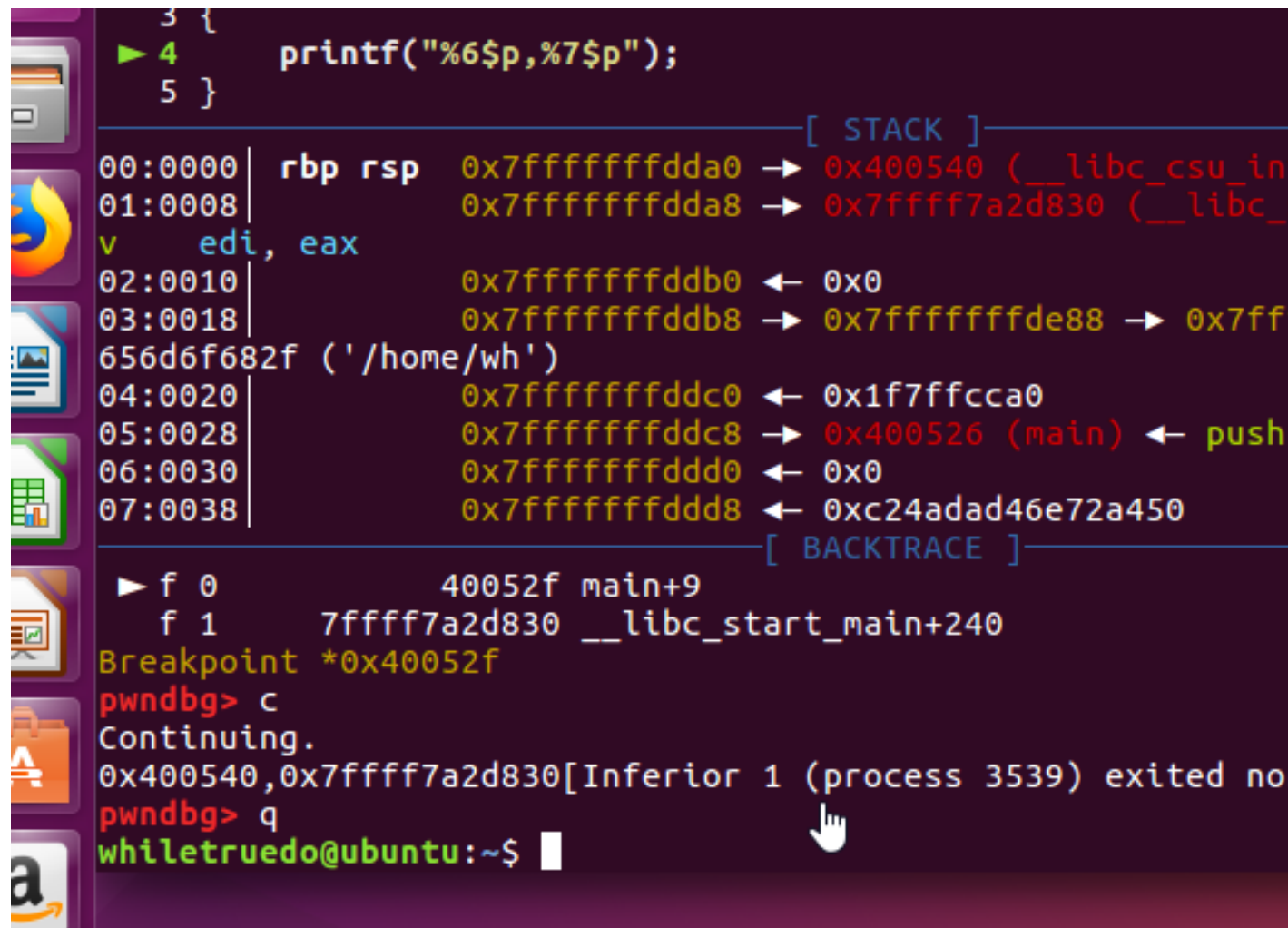
- hh：用于将 char 型参数转换成 int 型输出。
- h：用于将 short 型参数转换成 int 型输出。
- l：用于输出 long 型参数。对于浮点型无效果。
- ll：用于输出 long long 型参数。
- L：用于输出 long double 型参数。
- z：用于输出 size_t 型参数。
- j：用于输出 intmax_t 型参数。
- t：用于输出 ptrdiff_t 型参数。

%[参数\$][标志][宽度][.精度][长度]类型

- %p：将参数当成（void*）输出。其实就当成16进制数值输出。
- 什么是（void*）？其实就是一“坨”无类型的数据（二进制串）。
- %p：一般用于输出指针。64位地址是48（12*4）位的。
- 而我们用它来输出栈上的值：
- 因为用%p的输出与栈上的值毫无区别。
- 当然栈上的值请一定要注意大小端问题。

%[参数\$][标志][宽度][.精度][长度]类型

- 别问为什么是斜的，问就是我也不知道。我真的是直接截的图。
- 结果后面发现原来是我电脑是斜的，所以我才会以为它是斜的。



```
3 {  
4   printf("%6$p,%7$p");  
5 }  
[ STACK ]  
00:0000 | rbp rsp 0x7fffffffdda0 -> 0x400540 (__libc_csu_in  
01:0008 |          0x7fffffffdda8 -> 0x7ffff7a2d830 (__libc_  
v   edi, eax  
02:0010 |          0x7fffffffddb0 <- 0x0  
03:0018 |          0x7fffffffddb8 -> 0x7ffffffffffde88 -> 0x7ff  
656d6f682f ('/home/wh')  
04:0020 |          0x7fffffffddc0 <- 0x1f7ffcca0  
05:0028 |          0x7fffffffddc8 -> 0x400526 (main) <- push  
06:0030 |          0x7fffffffddd0 <- 0x0  
07:0038 |          0x7fffffffddd8 <- 0xc24adad46e72a450  
[ BACKTRACE ]  
f 0          40052f main+9  
f 1          7ffff7a2d830 __libc_start_main+240  
Breakpoint *0x40052f  
pwndbg> c  
Continuing.  
0x400540,0x7ffff7a2d830[Inferior 1 (process 3539) exited no  
pwndbg> q  
whiletruedo@ubuntu:~$
```

%[参数\$][标志][宽度][.精度][长度]类型

- `%n`:终于到重头戏了。这是一个异常神秘的玩意。
- `n`: 将当前的格式化字符串中已成功输出的字符数写入一个整型参数, 字符数不包括 `\n`、`\t` 等转义字符。从输入输出的方向来说, 此类型使用时更像是在 `scanf` 中接受输入, 只不过输入不是来自用户、而是来自系统计数。注意: 此计数仅对当前格式化字符串有效, 重新开始一个格式化字符串时, 计数将重置为 0。例如: `printf("%n", &num)` 将把 `num` 赋值为 0。

%[参数\$][标志][宽度][.精度][长度]类型

- `n`: 将当前的格式化字符串中已成功输出的字符数写入一个整型参数，字符数不包括 `\n`、`\t` 等转义字符。从输入输出的方向来说，此类型使用时更像是在 `scanf` 中接受输入，只不过输入不是来自用户、而是来自系统计数。注意：此计数仅对当前格式化字符串有效，重新开始一个格式化字符串时，计数将重置为 0。例如：`printf("%n", &num)` 将把 `num` 赋值为 0。

```
1 #include<stdio.h>
2 int main()
3 {
4     int a=1,n=0;
5     printf("%9d%n\n",a,&n);
6     puts("-----");
7     printf("%d\n",n);
8 }
```

```
Press ENTER or type command to continue
      1
-----
9
Press ENTER or type command to continue
```

%[参数\$][标志][宽度][.精度][长度]类型

- 简单来说：%n会将对应参数所指向地址的值改为它之前成功输出的字符数。
- 比如printf(“123123%n”,a);
- a的地址是0x400,a的值是0x500
- 我们说0x400的这个变量，实际上是指存放在0x400地址的变量，也就是a。
- 如果0x400存的值是0x500。0x500存的值是0x600。
- 那么执行完语句之后，0x500的值就会变成0x6。

%[参数\$][标志][宽度][.精度][长度]类型

- 而一般而言一个地址对应的是4个byte。（64位是8byte）
- 比如0x500里存的是：0xDDCCBBAA
- 小端法的话：
- 0x500就是AA
- 0x501就是BB
- 0x502就是CC
- 0x503就是DD

	03	02	01	00
0x500:	DD	CC	BB	AA

%[参数\$][标志][宽度][.精度][长度]类型

- 而一般而言一个地址对应的是4个byte。
- 比如0x500里存的是：0xDDCCBBAA
- 如果我们想要把0xDDCCBBAA改成0xDDCCBBEE怎么办呢？
- 当然是只修改0x500这一个byte(0xEE=238)
- printf(“%238c%hhn” ,0x500)
- 当然只是一段写意的代码啊，并不能运行。
- （因为地址的表示什么的）
- 如果一次修改两个byte的话，就用%hn就好了。

小端法的话：
0x500就是AA
0x501就是BB
0x502就是CC
0x503就是DD

	03	02	01	00
0x500:	DD	CC	BB	AA

漏洞利用

- 我们lab中看到的漏洞几乎都是这样的：
- `scanf("%s" ,s)`
- `printf(s)`
- 也就是说，我们可控格式化字符串，但是实际上后面没有参数列表了，（因为原作者就是以为我们的s，是一个普普通通的字符串，当然不该接参数列表）。
- 所以我们的参数实际上就会从栈上取（前四个在寄存器中）。

漏洞利用

- 实现任意写：即可以对任意address写任意value。
- value刚刚讲了怎么控制，现在讲讲怎么控制address
- 如果可控的格式化字符串本身就存在栈上：
- 那么我们就可控栈上的某一块区域的值。
- 我们可以在这块区域中的一个地方利用字符串放上address。

地址	值	备注
0x7fffd80	0x400bd4	ret addr
0x7fffd78	0x7fffe10	(old ebp)
0x7fffd70	0x23	ebp
0x7fffd68	0x7fffd98	stack addr
0x7fffd60	(nil)	value is 0
0x7fffd58	(nil)	value is 0
0x7fffd50	0x7fffd80	(In Dream)
0x7fffd48	0xa7024303125	%10\$p'\n'
0x7fffd40	0x603010	global var
0x7fffd38	0x400d80	esp
0x7fffd30		
0x7fffd28		
0x7fffd20		

漏洞利用

- 实际上字符串在栈上用ascii码储存。
- 想找到0x7fffd80对应的字符串：
- 使用pwntools里的p64(0x7fffd80)
- 之后会讲。。。
- 然而这一步可能会包含不可见字符
- 所以需要非键盘的输入输出方式。
- （也是pwntools）

地址	值	备注
0x7fffd80	0x400bd4	ret addr
0x7fffd78	0x7fffe10	(old ebp)
0x7fffd70	0x23	ebp
0x7fffd68	0x7fffd98	stack addr
0x7fffd60	(nil)	value is 0
0x7fffd58	(nil)	value is 0
0x7fffd50	0x7fffd80	(In Dream)
0x7fffd48	0xa7024303125	%10\$p'\n'
0x7fffd40	0x603010	global var
0x7fffd38	0x400d80	esp
0x7fffd30		
0x7fffd28		
~ ~ ~ ~ ~		

漏洞利用

- `print("%p,%p,%p,%p,%p,%p,%p,%p,%p,%p")`
- 就会一路遍历栈（当然前四个参数是在寄存器里）
- 假如说地址0x7fffd48，是printf的第9个参数。（即第9个%p的值或者说%9\$p的值是0xa7024303125）。
- 那么%10\$p就是0x7fffd80

地址	值	备注
0x7fffd80	0x400bd4	ret addr
0x7fffd78	0x7fffe10	(old ebp)
0x7fffd70	0x23	ebp
0x7fffd68	0x7fffd98	stack addr
0x7fffd60	(nil)	value is 0
0x7fffd58	(nil)	value is 0
0x7fffd50	0x7fffd80	(In Dream)
0x7fffd48	0xa7024303125	%10\$p'\n'
0x7fffd40	0x603010	global var
0x7fffd38	0x400d80	esp
0x7fffd30		
0x7fffd28		
0x7fffd20		

漏洞利用

- 那么我们
printf(“%160c%10\$hhn”)
- 就会将第十个参数：0x7fffd80的值改成0xa0。（hh会让程序以为是一个只有1byte长的类型（char）的指针。）
- 所以0x7fffd80的值就会是：0x400ba0。
- 当ret 时就会跳转到你希望的位置。
- 如果你希望改变某变量的值，也将是一样的做法。

地址	值	备注
0x7fffd80	0x400bd4	ret addr
0x7fffd78	0x7fffe10	(old ebp)
0x7fffd70	0x23	ebp
0x7fffd68	0x7fffd98	stack addr
0x7fffd60	(nil)	value is 0
0x7fffd58	(nil)	value is 0
0x7fffd50	0x7fffd80	(In Dream)
0x7fffd48	0xa7024303125	%10\$p'\n'
0x7fffd40	0x603010	global var
0x7fffd38	0x400d80	esp
0x7fffd30		
0x7fffd28		
0x7fffd20		

漏洞利用

可是栈上的地址是变化的，我们怎么知道用字符串把它填成什么值呢？

答案是，栈上的值是变化的，但是相对位置是不变的。（而且第十个参数的位置也每次都指相对意义下的同一个地方。）而栈上总会有一些地方的值本身就是栈的地址。（比如old ebp）。

那么你%p出来的值-0x90就是ret addr的储存位置。

地址	值	备注
0x7fffd80	0x400bd4	ret addr
0x7fffd78	0x7fffe10	(old ebp)
0x7fffd70	0x23	ebp
0x7fffd68	0x7fffd98	stack addr
0x7fffd60	(nil)	value is 0
0x7fffd58	(nil)	value is 0
0x7fffd50	0x7fffd80	(In Dream)
0x7fffd48	0xa7024303125	%10\$p'\n'
0x7fffd40	0x603010	global var
0x7fffd38	0x400d80	esp
0x7fffd30		
0x7fffd28		
~ ~ ~ ~ ~		

漏洞利用

当然你就需要两次printf，一次用来暴露栈地址，一次用来改ret addr。（也确实至少需要两次）

不过我们实验中有时为了降低难度，会提前暴露给你ret addr的存放地址。

地址	值	备注
0x7fffd80	0x400bd4	ret addr
0x7fffd78	0x7fffe10	(old ebp)
0x7fffd70	0x23	ebp
0x7fffd68	0x7fffd98	stack addr
0x7fffd60	(nil)	value is 0
0x7fffd58	(nil)	value is 0
0x7fffd50	0x7fffd80	(In Dream)
0x7fffd48	0xa7024303125	%10\$p'\n'
0x7fffd40	0x603010	global var
0x7fffd38	0x400d80	esp
0x7fffd30		
0x7fffd28		
0x7fffd20		

提示或注意

- 地址必须align (8) 。请注意用占位符对齐8的倍数。
- 地址只会有12*4位。高位（小端法会放在后面）将会是0x00（对应字符' \0' ）
- 它会截断字符串，建议放成单独的一个字符串，或者是放在字符串末尾。
- 常用的payload
 - “%K\$pxx……x” +p64(addr)
 - “%Nc%K\$nx…x……x” +p64(addr)

实验框架

框架缘由

- 我们考虑有这样一种程序，任何地方都有它的身影。
- 而你突然发现某个程序有这个bug，然后你发现XX服务器上也有这个程序，于是你希望使用这个程序来hack服务器，以此获得一些有价值的信息。
- 我们的实验框架也是基于此。
- 我会下发的包里有4个程序及其源代码。（从lvl-0到lvl-3）
- 同时在服务器上挂载有（几乎）同样的程序。可能在某些变量的初值上并不相同（这些变量的初值就代表着你想知道的有价值的信息——下一个程序挂载在哪里的）。

框架结构

- 一开始你打开lwl-0的程序，在本地尝试执行出包含port的代码
- 其中sprintf(r,s);printf("%s" ,r);
- 你可以理解为就是printf(s);
- 只不过这样的话：
- 我可以拿到一份输出的副本

```
len=read(0,&s,30);
if(len<=20)
{
    puts("The essence of human beings is REPEATER:");
    sprintf(r,s); //instead of printf, I use sprintf to
    printf("%s",r); //these two are equal to: printf(s)
    if(strcmp(r,s)!=0)
    {
        if(strcmp(r,"0x521\n")!=0)
            puts("\nWhy make me an inferior repeater?\nDo
        else
        {
            printf("The port is %s",r);
            puts("You have got the port.Just go to the nex
            return 0;
        }
    }
}
else
{
    puts("len error(max length:20)");
    return 0;
}
```

框架结构

- 如果你成功，你就会看到：

```
The port is 0x521  
You have got the port. Just go to the next problem.
```

- 无需遮掩，因为代码都下发给你们了。（ $0x521=1313$ ）
- 然后你发现你连接到服务器端：`nc 212.64.2.70 1313`
- 服务器端就在运行着这个程序。
- 如果你输入同样的东西，就会同样返回port的值。
- 不过这个值就是下一个程序挂载在服务器的端口号了：
- 即需要你`nc 212.64.2.70 PORT`

框架结构

- 自此之后，所有本地端的程序中你想要的值都是0，而服务器端的值则是下一个程序的端口号。
- 最后一个程序，服务器端存有一个文件，里面的值形如：
- `flag{XXXX}`
- 请尽量去找到这个值吧，但是找不到也没关系。。每个level都是有分的，请在实验报告中阐明自己做到了哪一步、每一步的端口号（甚至最后的flag值）、每一步的详细思路（包括错误的）以及最后的python代码（之后会讲到）。
- 之后还有两道可以想想看的思考题（第二题挺Trick的）。

思考题

- 1.一直在说，前4个参数在寄存器上，那么分别在哪个寄存器里呢？
 - 提示：你可以去看看printf怎么写的，也可以尝试着在gdb中去实际用用。
- 2.如果格式化字符串不在栈上怎么办呢？是否还能实现相同的攻击？要实现这样的攻击，是否需要满足什么额外的特征条件？
 - 提示，想一想%n的实际攻击原理，看看栈上是否有可能有可以利用的东西，因此，要实现这样的攻击，需要满足什么额外的条件？
- 思考题2，建议在完成实验之后再思考。（1就无所谓了）
- 如果对思考题有想法的同学，可以附加在实验报告中。

工具推荐

工具推荐：

- Pwndbg：gdb的插件，用来优化gdb的使用体验
- Pwntools：
 - python2.7的包（对，你没有看错，都8102年了，还有不支持python3的包）。
 - 在本次实验中主要用于代替键盘进行不可见字符的输入输出，直接与远端服务器进行数据传输，以及用来将任意的数据构造成对应的字符串（p64）。

pwndbg的安装

- `git clone https://github.com/pwndbg/pwndbg`
- `cd pwndbg`
- `./setup.sh`

pwndbg的使用

gdb的使用不再赘述, lab 2 应该用到了
disassemble main # 查看某部分的汇编
stack n # 查看栈的最低n个
c # continue
在程序运行中ctrl-c, 直接break回来。

84 ../sysdeps/unix/syscall-template.S: No such file or directory.

LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA

[REGISTERS]

```
RAX 0xfffffffffffffe00
RBX 0x0
RCX 0x7ffff7b04260 (__read_nocancel+7) ← cmp    rax, -0xfff
RDX 0x1e
RDI 0x0
RSI 0x7ffff7dd50 ← 0x0
R8  0x7ffff7dd3780 (_IO_stdfile_1_lock) ← 0x0
R9  0x7ffff7fdb700 ← 0x7ffff7fdb700
R10 0x2d2d2d2d2d2d2d2d ('-----')
R11 0x246
R12 0x400600 (_start) ← xor    ebp, ebp
R13 0x7ffff7ffde60 ← 0x1
R14 0x0
R15 0x0
RBP 0x7ffff7dd80 → 0x400830 (__libc_csu_init) ← push  r15
RSP 0x7ffff7dd48 → 0x40076e (main+120) ← cdqe
RIP 0x7ffff7b04260 (__read_nocancel+7) ← cmp    rax, -0xfff
```

[DISASM]

```
► 0x7ffff7b04260 <__read_nocancel+7>    cmp    rax, -0xfff
0x7ffff7b04266 <__read_nocancel+13>    jae    read+73 <0x7ffff7b04299>
↓
0x7ffff7b04299 <read+73>                mov    rcx, qword ptr [rip + 0x2ccbd8]
0x7ffff7b042a0 <read+80>                neg    eax
0x7ffff7b042a2 <read+82>                mov    dword ptr fs:[rcx], eax
0x7ffff7b042a5 <read+85>                or     rax, 0xffffffffffffffff
0x7ffff7b042a9 <read+89>                ret

0x7ffff7b042aa                                nop    word ptr [rax + rax]
0x7ffff7b042b0 <write>                  cmp    dword ptr [rip + 0x2d2489], 0 <0x7ffff7dd6740>
0x7ffff7b042b7 <write+7>              jne    write+25 <0x7ffff7b042c9>
↓
0x7ffff7b042c9 <write+25>              sub    rsp, 8
```

[STACK]

```
00:0000 | rsp 0x7ffff7dd48 → 0x40076e (main+120) ← cdqe
01:0008 | rsi 0x7ffff7dd50 ← 0x0
... ↓
05:0028 |      0x7ffff7ffdd70 → 0x7ffff7ffde60 ← 0x1
06:0030 |      0x7ffff7ffdd78 ← 0x521
07:0038 | rbp 0x7ffff7dd80 → 0x400830 (__libc_csu_init) ← push  r15
```

[BACKTRACE]

```
► f 0      7ffff7b04260 __read_nocancel+7
  f 1      40076e main+120
  f 2      7ffff7a2d830 __libc_start_main+240
```

Program received signal SIGINT

pwndbg> █

```

f 2      7ffff7a2d830 __libc_start_main+240
Program received signal SIGINT
pwndbg> stack 30
00:0000 | rsp 0x7fffffffdd48 → 0x40076e (main+120) ← cdqe
01:0008 | rsi 0x7fffffffdd50 ← 0x0
... ↓
05:0028 |      0x7fffffffdd70 → 0x7fffffffde60 ← 0x1
06:0030 |      0x7fffffffdd78 ← 0x521
07:0038 | rbp 0x7fffffffdd80 → 0x400830 (__libc_csu_init) ← push r15
08:0040 |      0x7fffffffdd88 → 0x7ffff7a2d830 (__libc_start_main+240) ← mov edi, eax
09:0048 |      0x7fffffffdd90 ← 0x1
0a:0050 |      0x7fffffffdd98 → 0x7fffffffde68 → 0x7fffffffef1e5 ← 0x68772f656d6f682f ('/home/wh')
0b:0058 |      0x7fffffffdda0 ← 0x1f7ffcca0
0c:0060 |      0x7fffffffdda8 → 0x4006f6 (main) ← push rbp
0d:0068 |      0x7fffffffddb0 ← 0x0
0e:0070 |      0x7fffffffddb8 ← 0x59ab0f1df9f22cb7
0f:0078 |      0x7fffffffddc0 → 0x400600 (_start) ← xor ebp, ebp
10:0080 |      0x7fffffffddc8 → 0x7fffffffde60 ← 0x1
11:0088 |      0x7fffffffddd0 ← 0x0
... ↓
13:0098 |      0x7fffffffdde0 ← 0xa654f06252b22cb7
14:00a0 |      0x7fffffffdde8 ← 0xa654e0d846422cb7
15:00a8 |      0x7fffffffddf0 ← 0x0
... ↓
18:00c0 |      0x7fffffffde08 → 0x7fffffffde78 → 0x7fffffffef207 ← 'XDG_VTNR=7'
19:00c8 |      0x7fffffffde10 → 0x7ffff7ffe168 ← 0x0
1a:00d0 |      0x7fffffffde18 → 0x7ffff7de77cb (_dl_init+139) ← jmp 0x7ffff7de77a0
1b:00d8 |      0x7fffffffde20 ← 0x0
... ↓
1d:00e8 |      0x7fffffffde30 → 0x400600 (_start) ← xor ebp, ebp
pwndbg>

```

Pwntools的安装

- Linux自带python2.7, 但请确定其安装了pip。pip的安装方法, 请学会使用搜索引擎。(如果你安装了pwndbg, 应该会自动更新pip)。
- `pip install pwntools`

Pwntools的使用

- `from pwn import *`
- `p=process("./client")` #本地
- `p=remote("IP Address" , " PORT")` #服务器
- `p=remote("212.64.2.70" , " 1313")` #等价于：`nc 212.64.2.70 1313`
- `context(arch= "i686" ,os= "linux" ,log_level= "debug")` #告知环境
- `p.recvline()` #接受一行，相当于从程序input
- `p.sendline()` #发送一行，相当于output到程序
- `p.recvlines(numlines=n)` #接受n行，常用于忽略掉某些提示行。

Pwntools的使用

- `p.interactive()` # 从机器交互中退出来，返还给人类。此时键盘和屏幕重新变成 `stdin` 和 `stdout`
- `int(s,16)` # built-in 的函数，用于将字符串（16进制表示）转化为数。如：“0x10” 变成 16。
- `hex(a)` # built-in 的函数，用于将数变成 16 进制字符串。
- `str(a)` # built-in 的函数，用于将数变成字符串。
- `int(s)` # built-in 的函数，用于将字符串变成数。
- `p64(a)` # 将构造等同于数 `a` 的字符串。如 0x70243125 转化为 “%1\$p\x00\x00\x00\x00”。如果只需要 32 位，则用 `p32`，同理 `p16`、`p8`
- `u64(s)` # 将 `s` 反构造成数，是 `p64` 的反函数。

论重度拖延症患者的死因

- 4点钟才写完这个ppt，我快要猝死了。
- 真的不要养成拖延的坏习惯。
- 我要死了。。
- 爆炸吧。。
- 我OS的lab还没开始写，周五就要交。。。。
- 爆炸吧。。
- 炸吧。。
- 吧。。

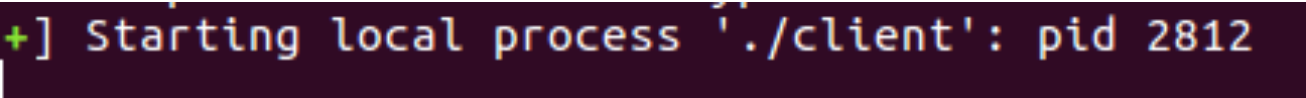
4:29

2018/10/18

2018/10/18

补充部分

gdb和pwntools的联动

- 当你使用pwntools的：p=process(“./client”)之后。
- 可以在后面添加一个input(), 将python卡在那里等待你的输入：
- 你会看到  `[+] Starting local process './client': pid 2812`
- 在gdb中：attach pid #这里是attach 2812
- 就可以正常使用gdb调试./client了。
- （在连接远程服务器的时候，和本地最大的区别就是没有debug环境）