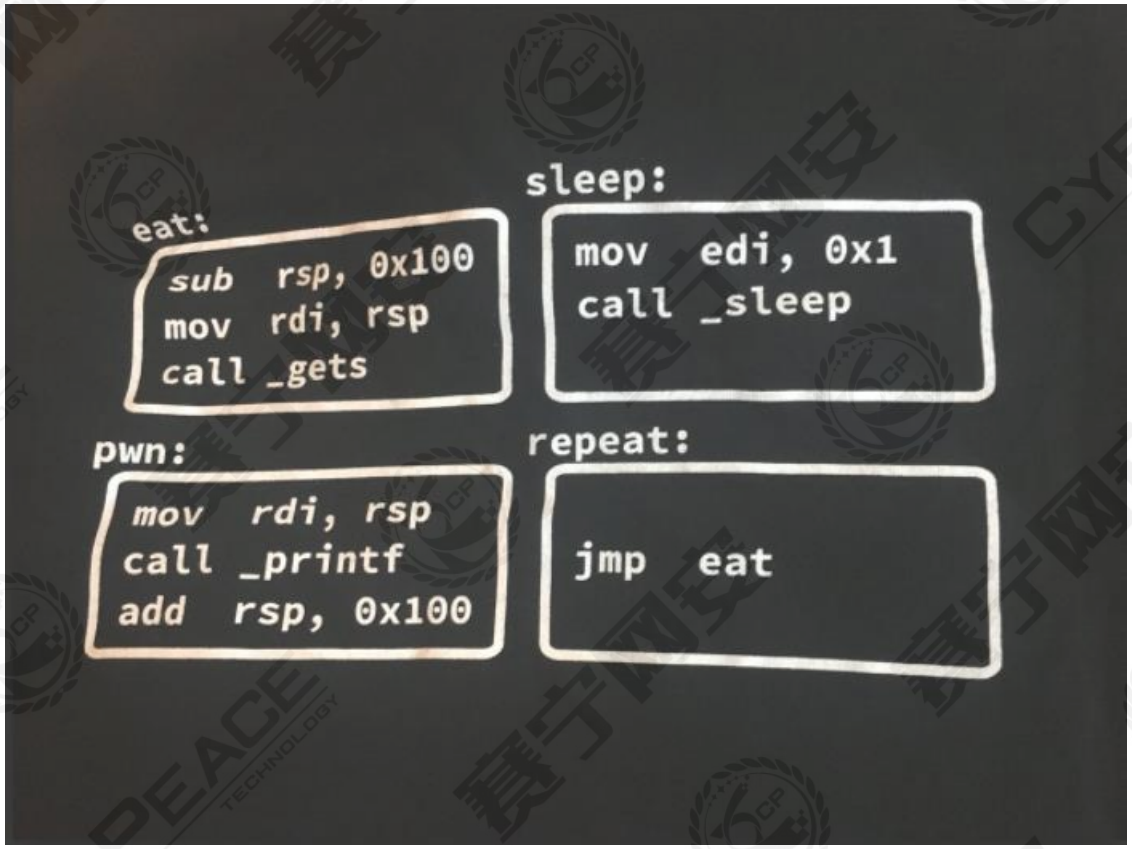
Pwn Lfie的开始



0 二进制基础知识

1 ELF文件介绍

2 Shellcode介绍

3 堆栈寄存器介绍

4 系统保护机制介绍

5 GDB 一些操作

抛砖引玉

什么是C语言代码、汇编指令、机器码？

32位程序、64位程序的区别？

抛砖引玉

```c
#include <stdio.h>

int main()
{
    printf("Hello World\n");
    return 0;
}
```

```
00000590 <main>:
 590:   8d 4c 24 04             lea    ecx,[esp+0x4]
 594:   83 e4 f0                and    esp,0xfffffff0
 597:   ff 71 fc                push   DWORD PTR [ecx-0x4]
 59a:   55                      push   ebp
 59b:   89 e5                   mov    ebp,esp
 59d:   53                      push   ebx
 59e:   51                      push   ecx
 59f:   e8 28 00 00 00          call   5cc <__x86.get_pc_thunk.ax>
 5a4:   05 5c 1a 00 00          add    eax,0x1a5c
 5a9:   83 ec 0c                sub    esp,0xc
 5ac:   8d 90 50 e6 ff ff       lea    edx,[eax-0x19b0]
 5b2:   52                      push   edx
 5b3:   89 c3                   mov    ebx,eax
 5b5:   e8 36 fe ff ff          call   3f0 <puts@plt>
 5ba:   83 c4 10                add    esp,0x10
 5bd:   b8 00 00 00 00          mov    eax,0x0
 5c2:   8d 65 f8                lea    esp,[ebp-0x8]
 5c5:   59                      pop    ecx
 5c6:   5b                      pop    ebx
 5c7:   5d                      pop    ebp
 5c8:   8d 61 fc                lea    esp,[ecx-0x4]
 5cb:   c3                      ret
```

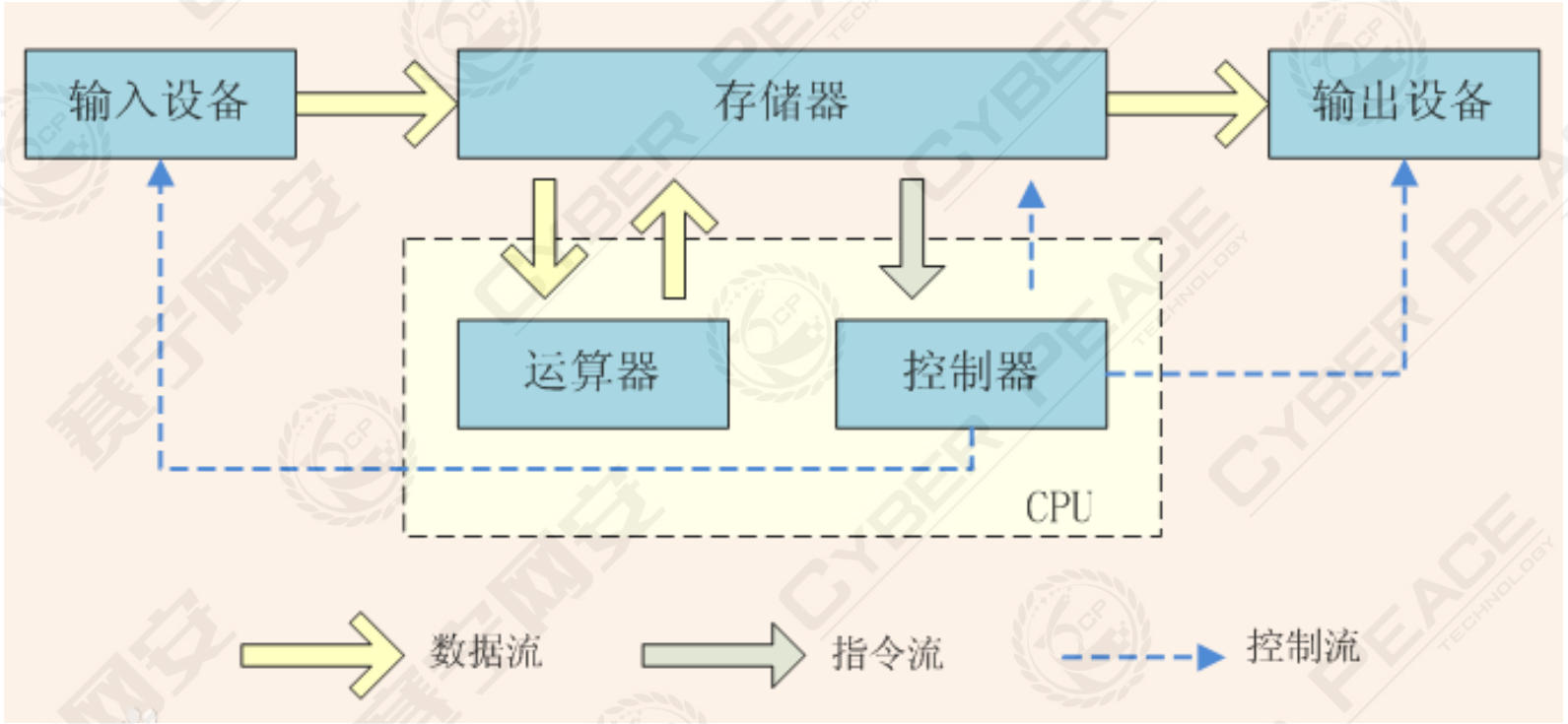
抛砖引玉

0x00804800:0x41

0x00804801:0x42

0x00804802:0x43

计算机硬件由运算器、控制器、存储器、输入设备和输出设备五大部分组成



抛砖引玉

计算机处理的数据和指令一律用**二进制**数表示

CTF中的二进制

Reverse：软件脱壳、算法破解

Pwn：漏洞利用，利用程序漏洞在对方机器执行任意代码

Pwn与我们生活

远程代码执行

Pwn与我们生活

本地提权

实验演示

通过HTTP服务进行远程代码执行

ELF文件结构介绍

bss段：

通常是指用来存放程序中未初始化的全局变量的一块内存区域，可读可写

```c
#include <stdio.h>

char global_var[100];

int main()
{
    scanf("%s", global_var);
    printf("Hello World %s\n", global_var);
    return 0;
}
```

ELF文件结构介绍

data段：

数据段通常是指用来存放程序中已初始化的全局变量的一块内存区域。数据段属于静态内存分配，可读不可写

rodata段：

存放C中的字符串和#define定义的常量，可读不可写

```
.rodata:080486FD                db    0
.rodata:080486FE                db    2
.rodata:080486FF                db    0
.rodata:08048700 ; char s[]
.rodata:08048700 s              db 'Welcome~',0       ; DATA XREF: main+19↑o
.rodata:08048700 _rodata        ends
.rodata:08048700
```

ELF文件结构介绍

text段：

代码段通常是指用来存放程序执行代码的一块内存区域。这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于可读可执行不可写

```
; int __cdecl main(int argc, const char **argv, const char
public main
main proc near

nptr= byte ptr -40h
buf= byte ptr -30h
var_8= dword ptr -8
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 40h
mov     eax, 0
call    Init
mov     edx, 19h          ; n
lea     rsi, aWelcomeToRecho ; "Welcome to Recho server!\n"
mov     edi, 1            ; fd
call    _write
jmp     short loc_400813
```

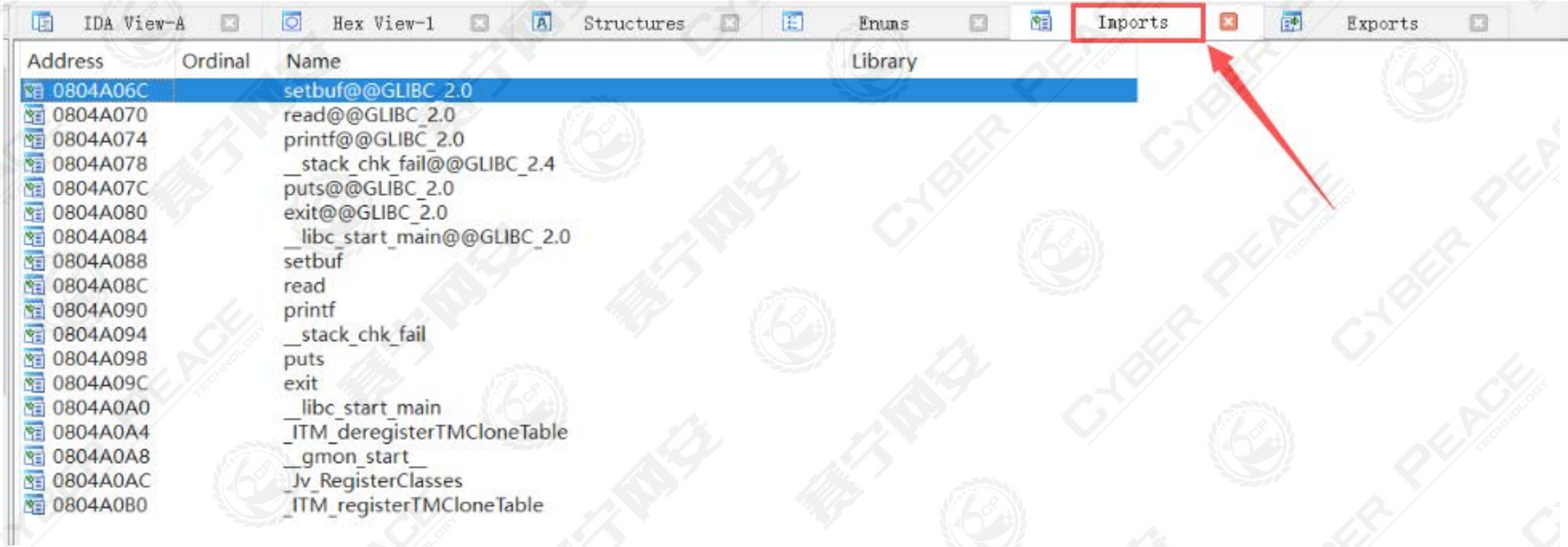
IDA对ELF的解析

程序中的各个段：

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD
.init	080483E8	0804840B	R	.	X	.	L	dword	0001	public	CODE	32
.plt	08048410	080484A0	R	.	X	.	L	para	0002	public	CODE	32
.text	080484A0	080486E2	R	.	X	.	L	para	0003	public	CODE	32
.fini	080486E4	080486F8	R	.	X	.	L	dword	0004	public	CODE	32
.rodata	080486F8	08048709	R	.	.	.	L	dword	0005	public	CONST	32
.eh_frame_hdr	0804870C	08048748	R	.	.	.	L	dword	0006	public	CONST	32
.eh_frame	08048748	08048850	R	.	.	.	L	dword	0007	public	CONST	32
.init_array	08049F08	08049F0C	R	W	.	.	L	dword	0008	public	DATA	32
.fini_array	08049F0C	08049F10	R	W	.	.	L	dword	0009	public	DATA	32
.jcr	08049F10	08049F14	R	W	.	.	L	dword	000A	public	DATA	32
.got	08049FFC	0804A000	R	W	.	.	L	dword	000B	public	DATA	32
.got.plt	0804A000	0804A02C	R	W	.	.	L	dword	000C	public	DATA	32
.data	0804A02C	0804A034	R	W	.	.	L	dword	000D	public	DATA	32
.bss	0804A040	0804A06C	R	W	.	.	L	32byte	000E	public	BSS	32

IDA对ELF的解析

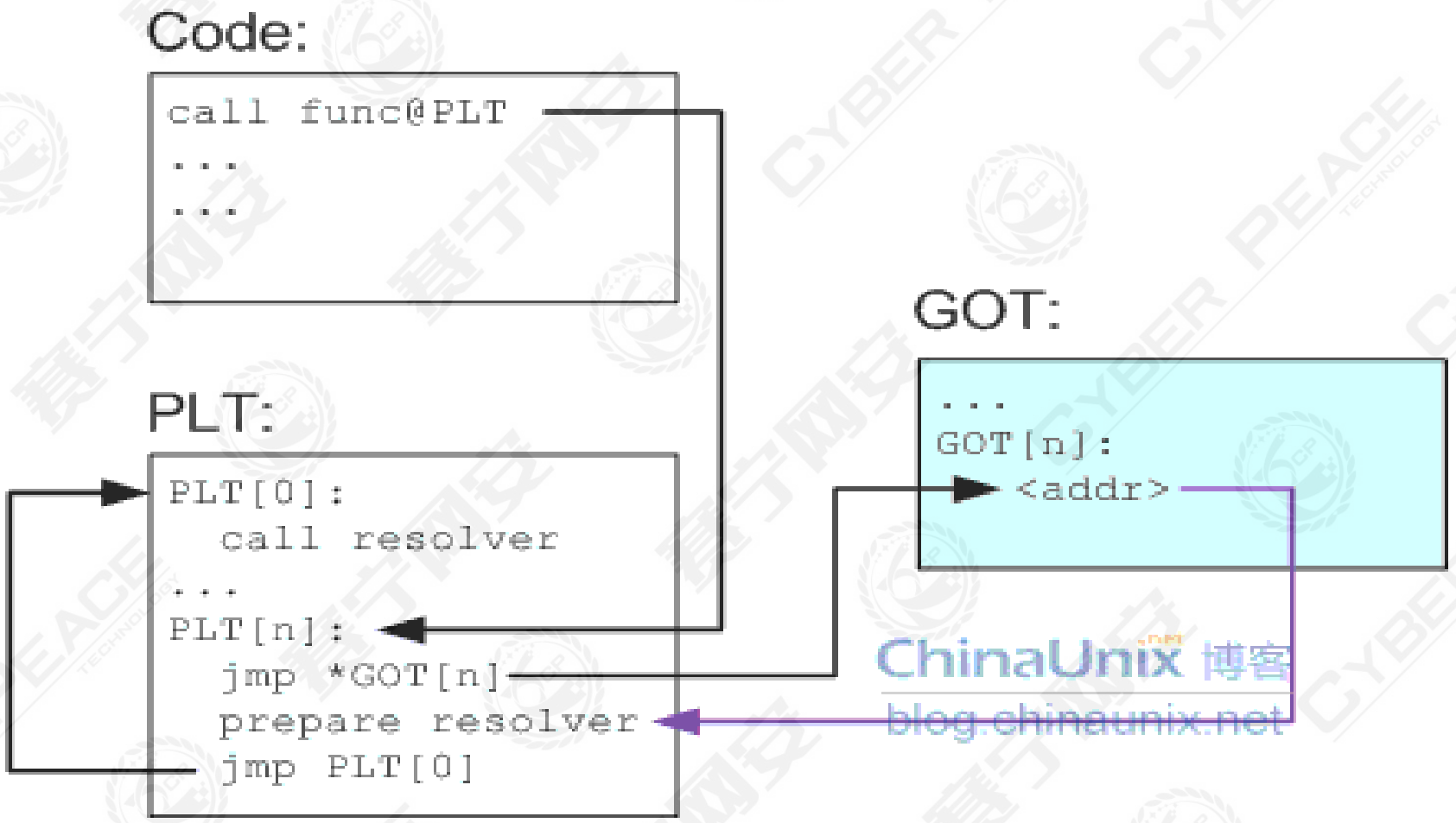
静态编译的函数：



PLT表和GOT表

在ELF文件的动态连接机制中，每一个外部定义的符号在全局偏移表(Global Offset Table，GOT)中有相应的条目，如果符号是函数则在过程连接表(Procedure Linkage Table，PLT)中也有相应的条目，且一个PLT条目对应一个GOT条目。

PLT表和GOT表



PLT表和GOT表

1. 执行“call func”指令后，首先都会跳转到func函数的plt表

2. plt表第一行都是 jmp *got[n]，在该函数第一次调用时，got[n]中存放的是plt表下一条地址，继续往下执行会调用到函数_dl_runtime_resolve函数，其功能就是找到要调用函数func在内存中的真正地址

3. 查询该函数的地址后，该函数将真实地址写入到got[n]中来

4. 当第二次调用func函数时，got[n]中存的就是该函数代码的地址，所以jmp *got[n]就能准确的进入到该函数进行执行

Lazy binding——延迟绑定，动态确定库函数真实地址

PLT表和GOT表

实验演示

Shellcode

1. Shellcode是一段代码

2. Shellcode一般是作为数据发送给受攻击服务器的

3. Shellcode是溢出程序和蠕虫病毒的核心

Shellcode常见功能

1. 获取shell型
- 开放端口等待正向连接，获得shell
- 反向连接，得到shell
- 直接得到shell（linux）

2. 本地操作型：
- 用户权限操作
- 修改文件执行权限
- Download & Execute

3. 验证型：
- 弹出计算器

程序执行单位

```
EIP ▸ 0x565555a4 <main+20>    add    eax, 0x1a5c
      0x565555a9 <main+25>    sub    esp, 0xc
      0x565555ac <main+28>    lea    edx, [eax - 0x19b0]
      0x565555b2 <main+34>    push   edx
      0x565555b3 <main+35>    mov    ebx, eax
      0x565555b5 <main+37>    call   puts@plt
```

寄存器

EAX：累加(Accumulator)寄存器，常用于函数返回值
EBX：基址(Base)寄存器，以它为基址访问内存
ECX：计数器(Counter)寄存器，常用作字符串和循环操作中的计数器
EDX：数据(Data)寄存器，常用于乘除法和I/O指针
ESI：源变址寄存器
EDI：目的变址寄存器

ESP：堆栈(Stack)指针寄存器，指向堆栈顶部
EBP：基址指针寄存器，指向当前堆栈底部

EIP：指令寄存器，指向下一条指令的地址

EIP是第一生产力　　**不惜一切代价，控制EIP**

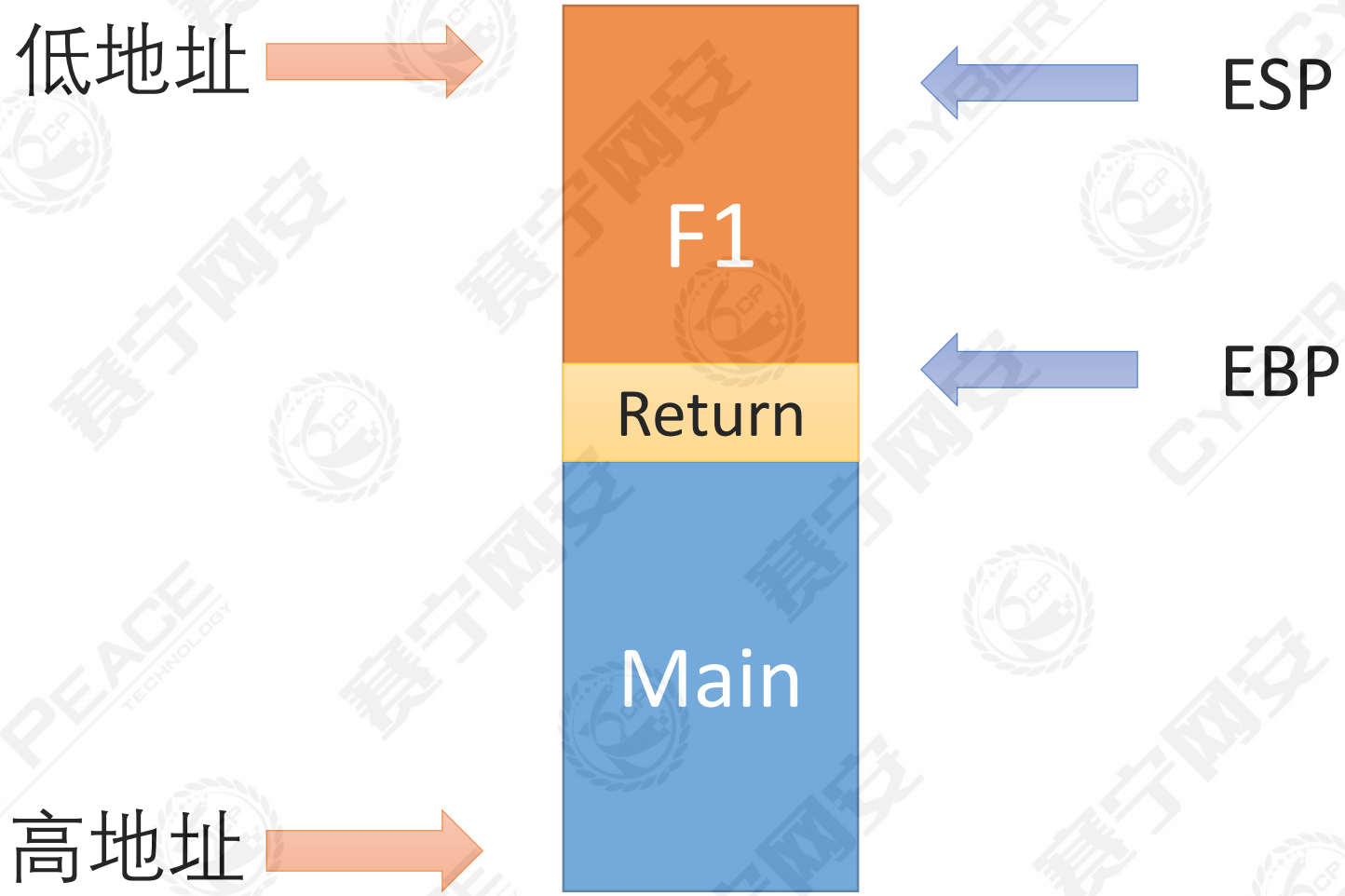
程序运行时

- 代码段（.text）
- 动态库
- GOT表和PLT表
- 栈（stack）
- 堆（动态分配内存）

栈

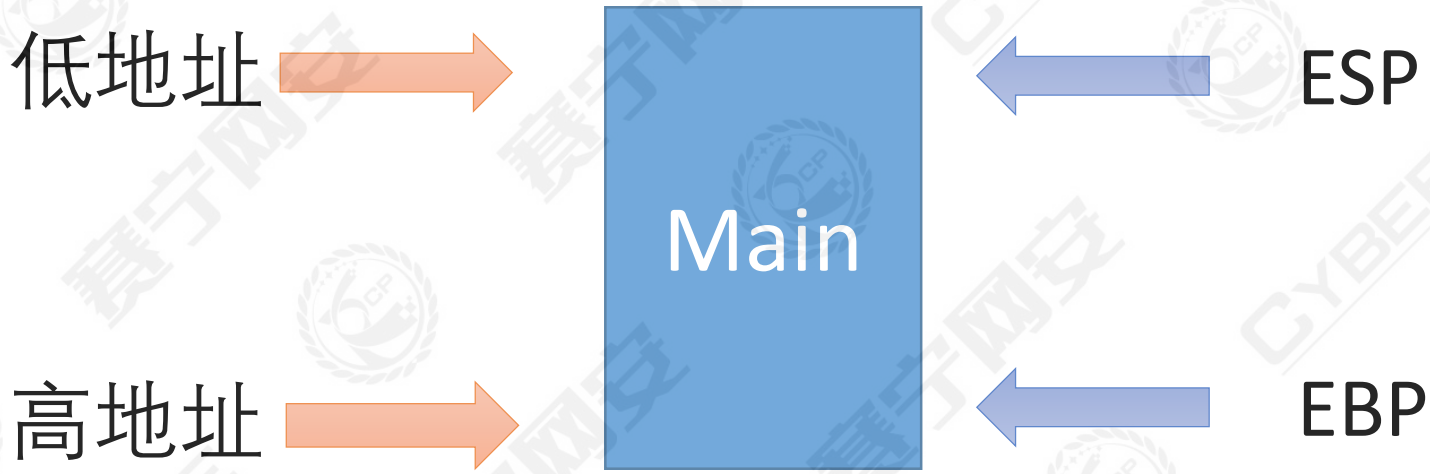

0x56555000 0x56556000 r-xp 1000 0 /root/桌面/te
0x56556000 0x56557000 r--p 1000 0 /root/桌面/te
0x56557000 0x56558000 rw-p 1000 1000 /root/桌面/te
0xf7dfa000 0xf7dfc000 rw-p 2000 0
0xf7dfc000 0xf7fad000 r-xp 1b1000 0 /lib32/libc-2
0xf7fad000 0xf7faf000 r--p 2000 1b0000 /lib32/libc-2
0xf7faf000 0xf7fb0000 rw-p 1000 1b2000 /lib32/libc-2
0xf7fb0000 0xf7fb3000 rw-p 3000 0
0xf7fd2000 0xf7fd4000 rw-p 2000 0
0xf7fd4000 0xf7fd7000 r--p 3000 0 [vvar]
0xf7fd7000 0xf7fd9000 r-xp 2000 0 [vdso]
0xf7fd9000 0xf7ffb000 r-xp 22000 0 /lib32/ld-2.2
0xf7ffc000 0xf7ffd000 r--p 1000 22000 /lib32/ld-2.2
0xf7ffd000 0xf7ffe000 rw-p 1000 23000 /lib32/ld-2.2
0xfffdd000 0xffffe000 rw-p 21000 0 [stack]

栈



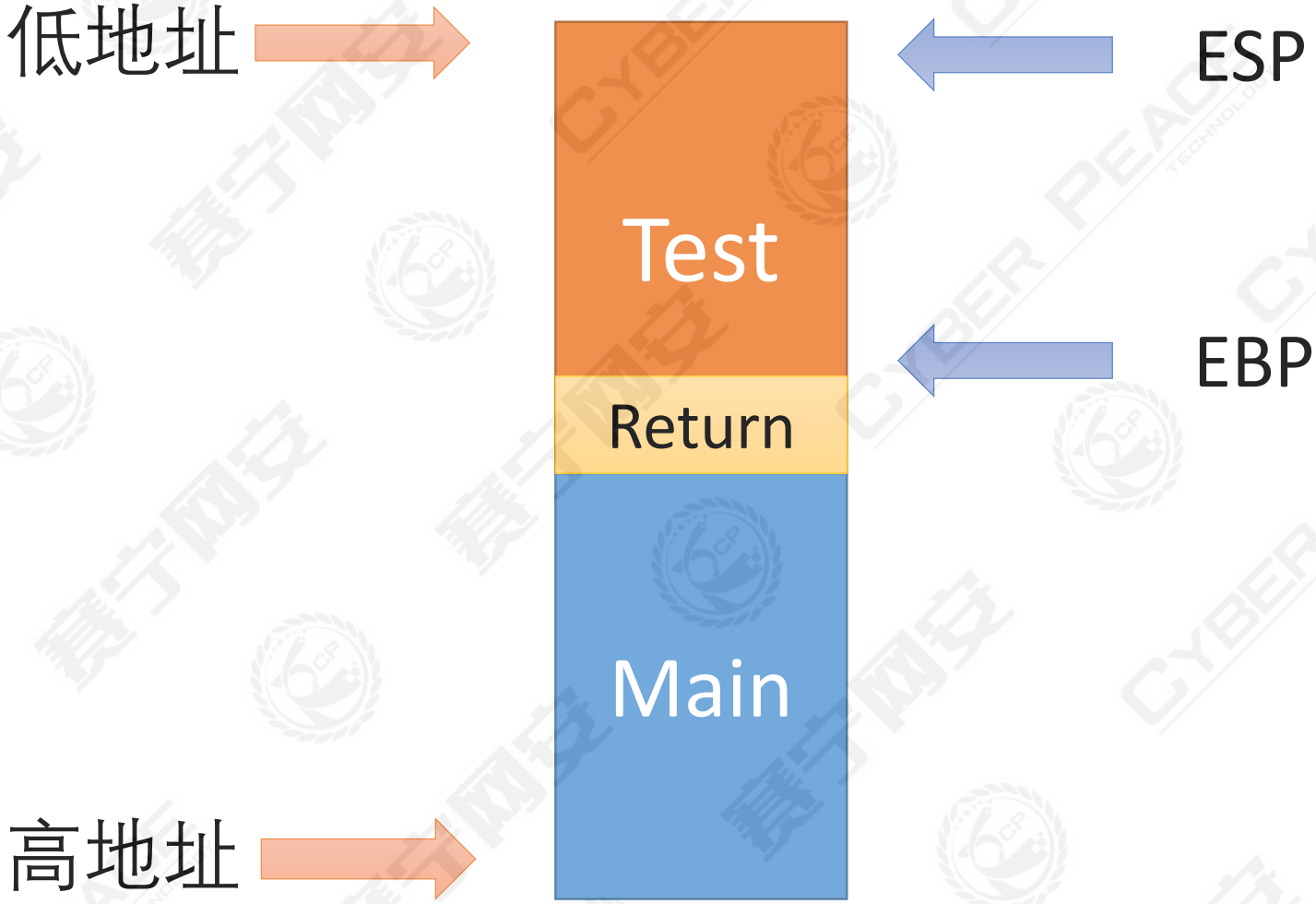
栈

调用Test函数之前

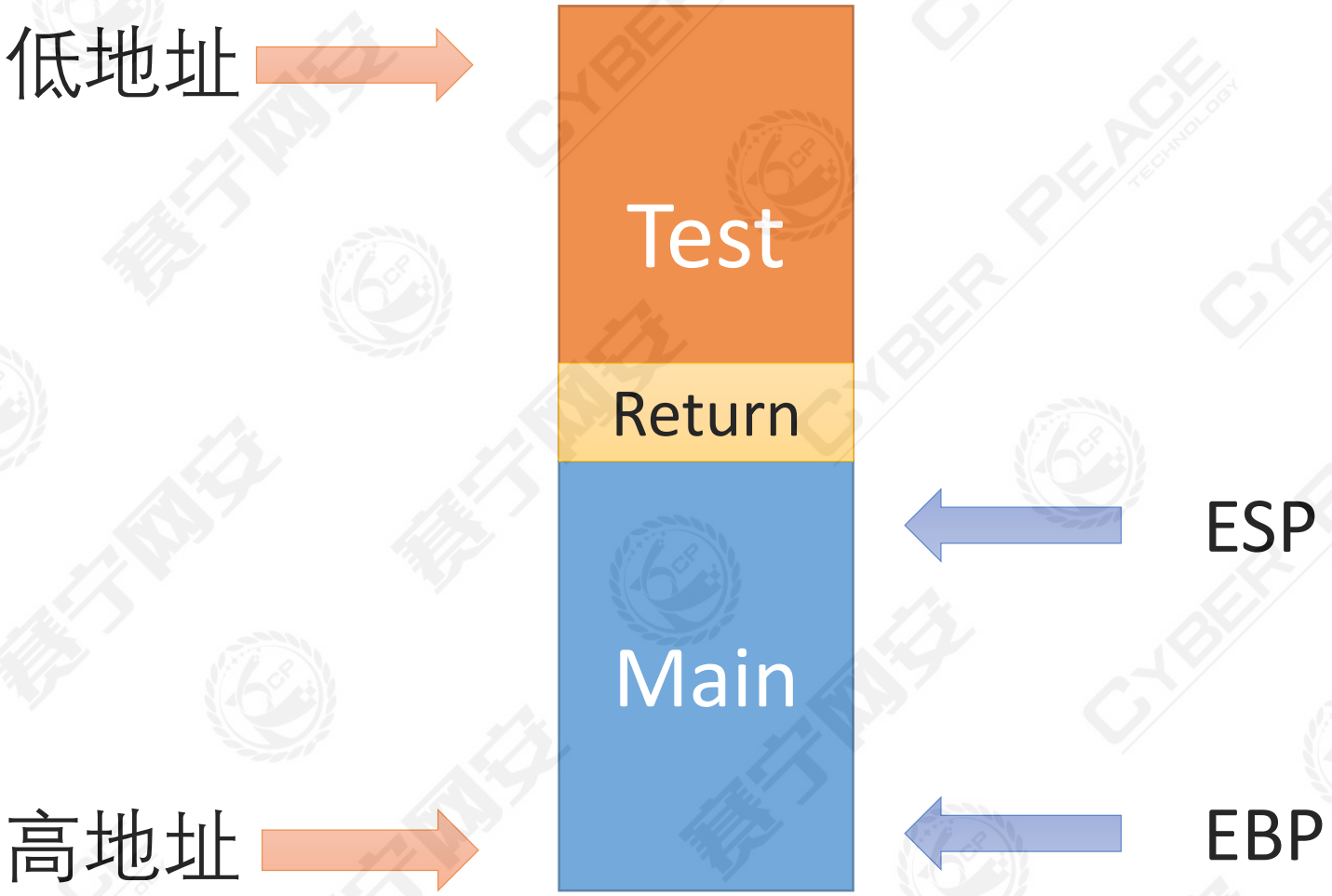


栈

调用Test函数



Test函数执行结束后

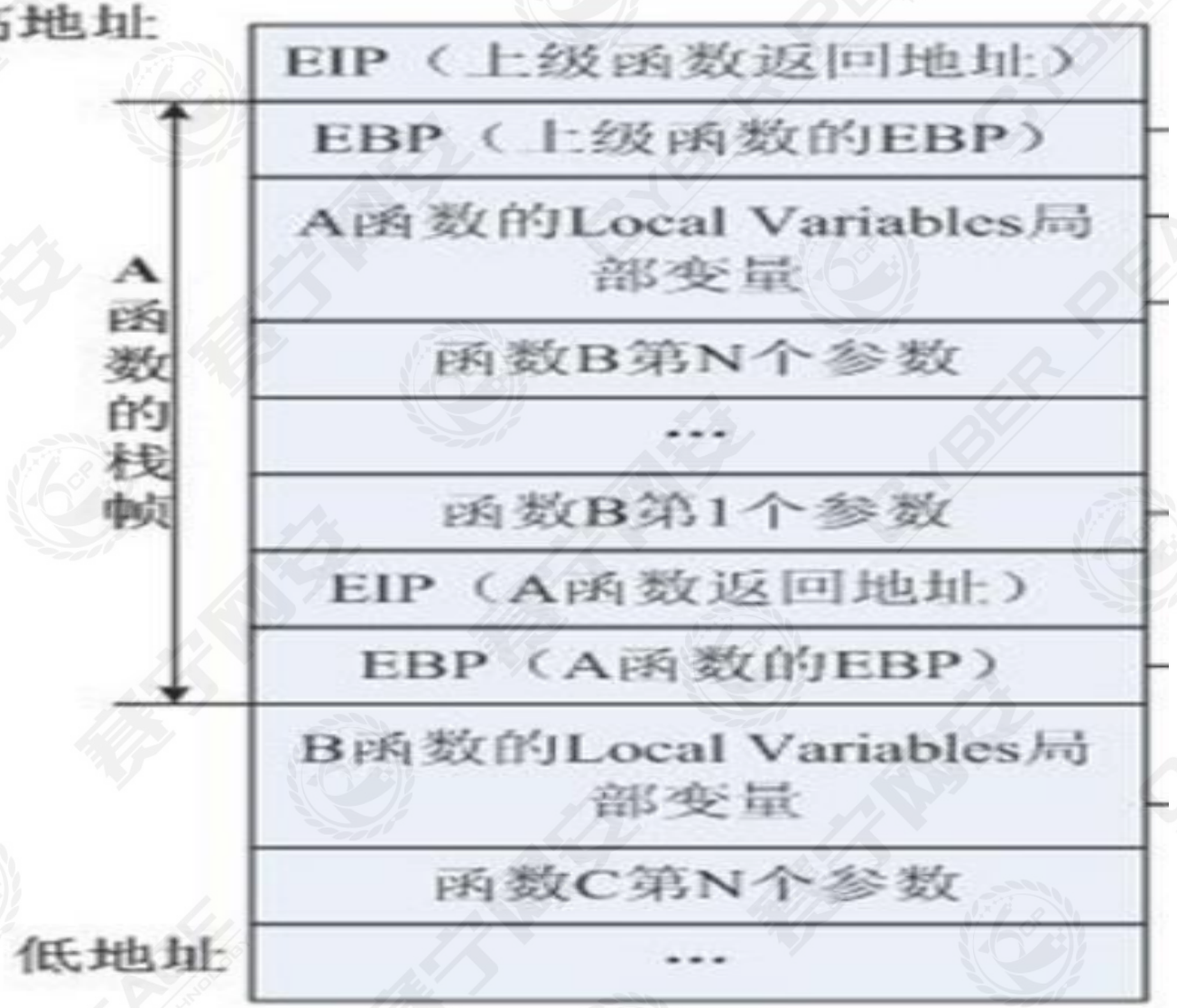


栈在函数调用中的作用

调用函数：Call func

1. 将函数参数依次压栈
2. 将call指令下一条指令地址压栈
3. 保存前函数栈帧，并抬高栈帧为新函数使用

栈在函数调用中的作用


高地址
EIP（上级函数返回地址）
EBP（上级函数的EBP）
A函数的Local Variables局部变量
函数B第N个参数
...
函数B第1个参数
EIP（A函数返回地址）
EBP（A函数的EBP）
B函数的Local Variables局部变量
函数C第N个参数
...
低地址
A函数的栈帧

栈在函数调用中的作用

保存前函数栈帧，并抬高栈帧为新函数使用：

1. push ebp：保存原来ebp位置
2. mov ebp,esp：保存原来esp位置
3. sub esp,0x90：抬高栈帧，让新函数使用

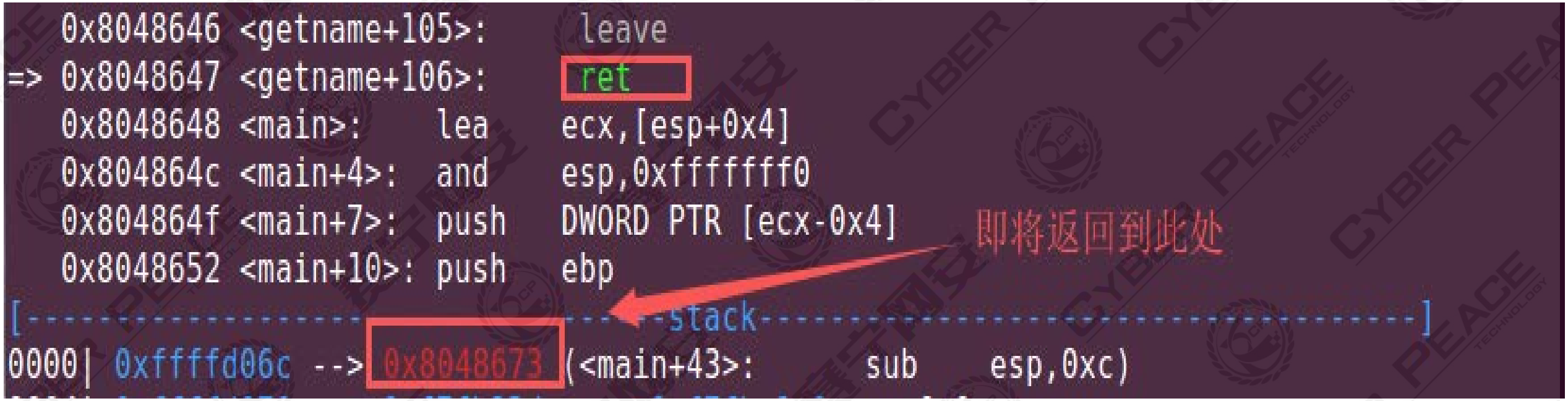
函数返回时 leave指令等效于：

1. esp = ebp + 4：恢复原来esp位置
2. ebp = *ebp：恢复原来ebp位置

栈在函数调用中的作用

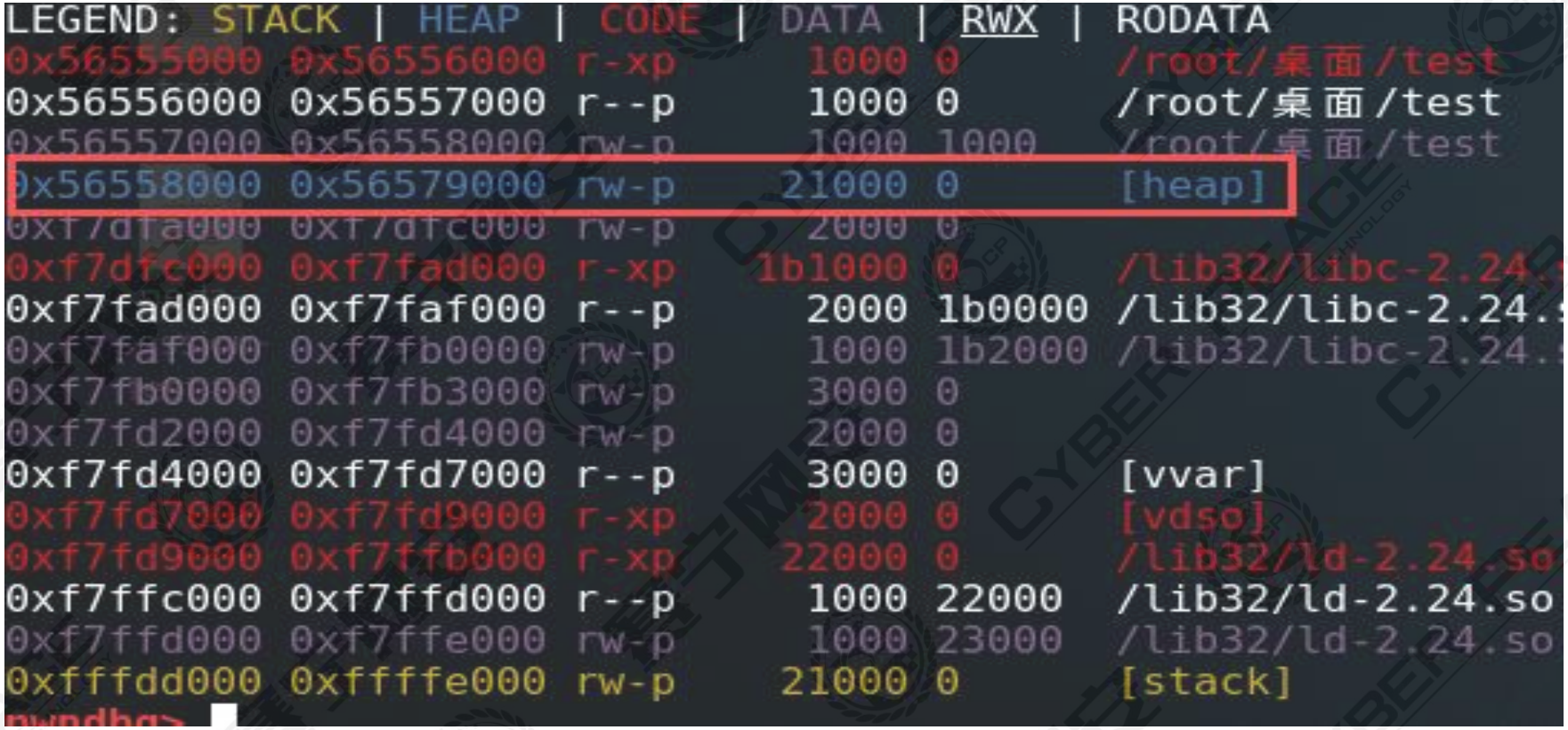
函数返回：ret

1. 弹出栈顶元素给EIP
2. 程序跳转到此位置



堆

- malloc、cmalloc、remalloc函数分配
- free等函数释放
- 可读可写权限

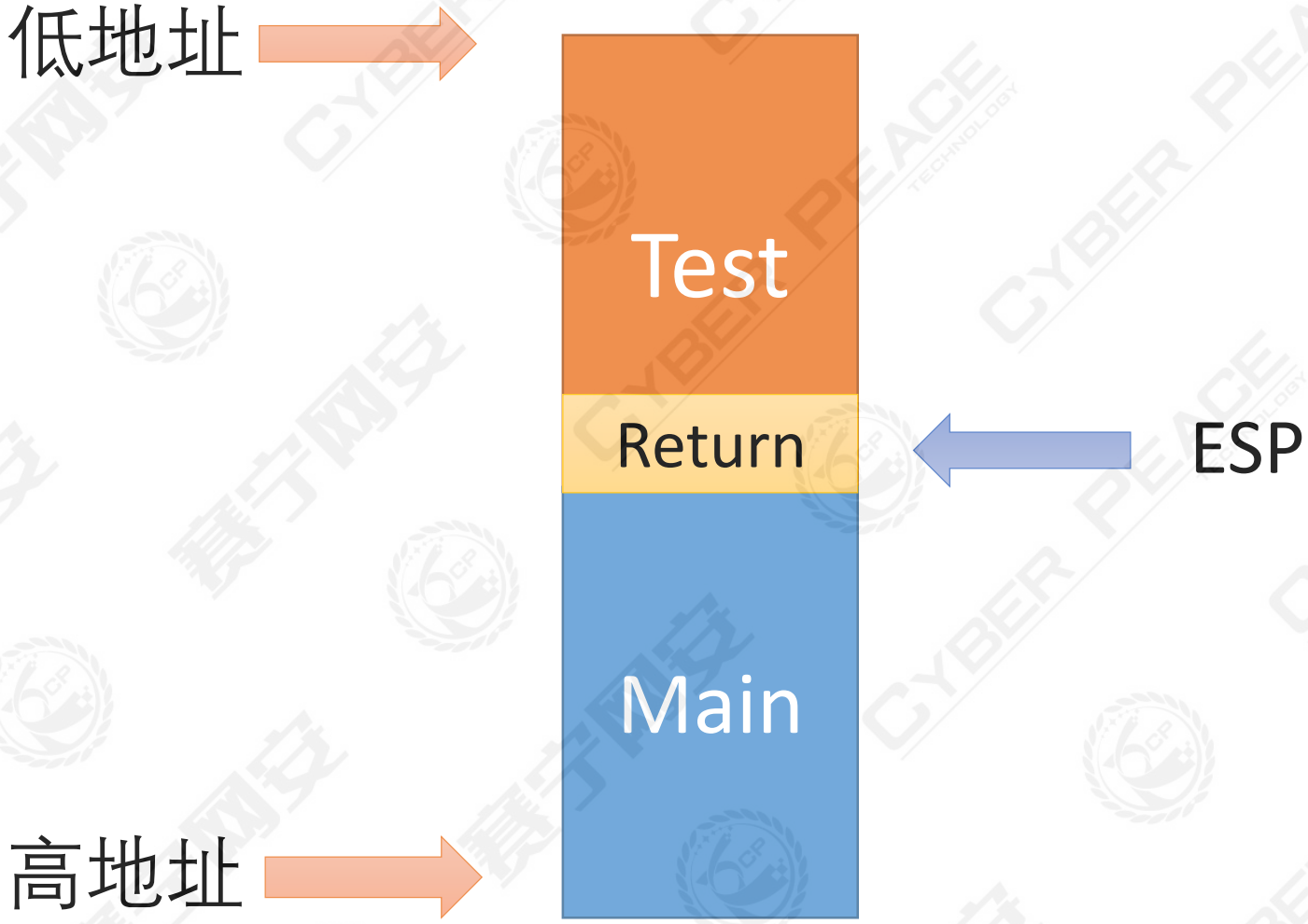


如何控制EIP

- 修改返回地址
- 修改函数指针（C++虚表、自定义结构体）
- 修改got表内容

如何控制EIP

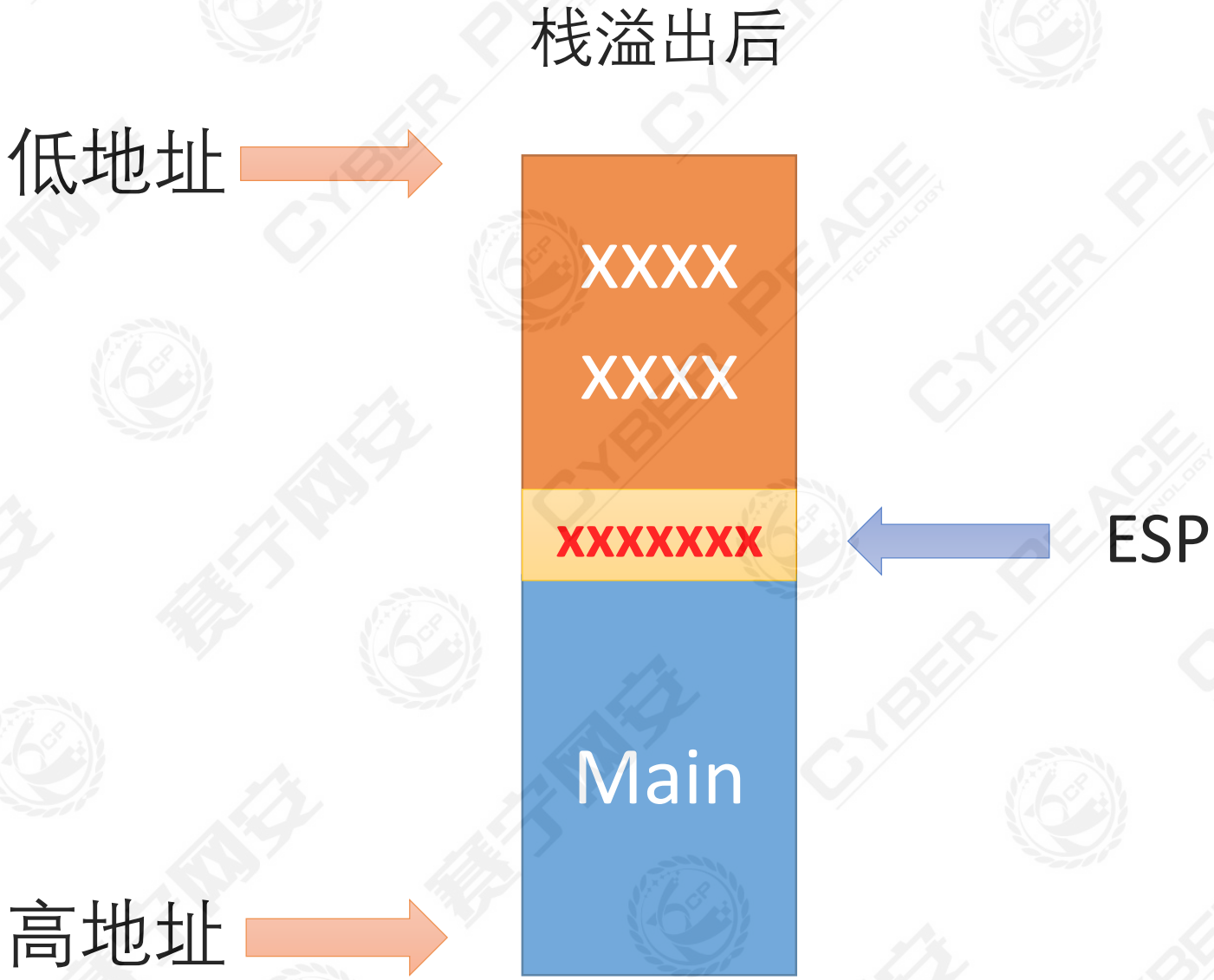
覆盖RET：



如何控制EIP

覆盖RET：

RET == POP EIP



如何控制EIP

在JMP或CALL处控制EIP：

示例一：

```
mov eax, cs:test_function
jmp eax
```

示例二

```
mov eax, cs:test_function
call eax
```

call eax == push eip && jmp eax

如何控制EIP

修改函数指针：

```
struct stuff
{
    char job[20];
    int age;
    int **get_age;
};
```

Linux系统保护机制

- **NX(DEP)：数据执行防护**
- **Canary(FS)：栈溢出保护**
- **RELRO(ASLR)：（地址随机化）**
- **PIE（代码地址随机化）**

```
gdb-peda$ checksec
CANARY    : ENABLED
FORTIFY   : disabled
NX        : ENABLED
PIE       : disabled
RELRO     : Partial
```

Linux系统保护机制

NX：

栈上的数据没有执行权限

防止攻击手段：栈溢出 + 跳到栈上执行shellcode

Linux系统保护机制

CANARY：

在函数开始时就随机产生一个值，将这个值CANARY放到栈上紧挨ebp的上一个位置，当攻击者想通过缓冲区溢出覆盖ebp或者ebp下方的返回地址时，一定会覆盖掉CANARY的值；当程序结束时，程序会检查CANARY这个值和之前的是否一致，如果不一致，则不会往下运行，从而避免了缓冲区溢出攻击。

防止攻击手段：所有单纯的栈溢出

Linux系统保护机制

ASLR：

堆栈地址随机化

防止攻击手段：所有需要用到堆栈精确地址的攻击，要想成功，必须用提前泄露地址

Linux系统保护机制

PIE：

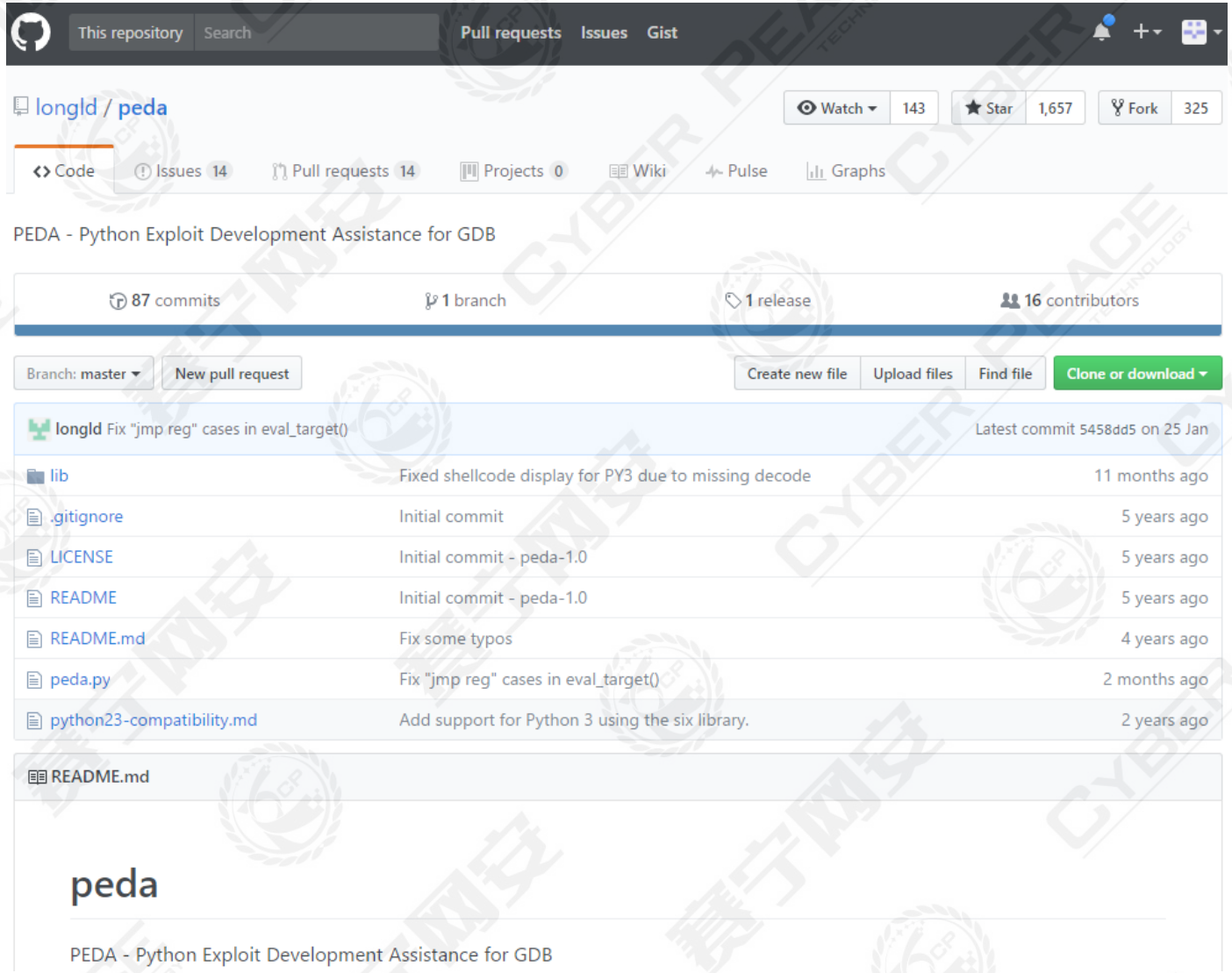
代码部分地址无关

防止攻击手段：构造ROP链攻击

GDB常用命令

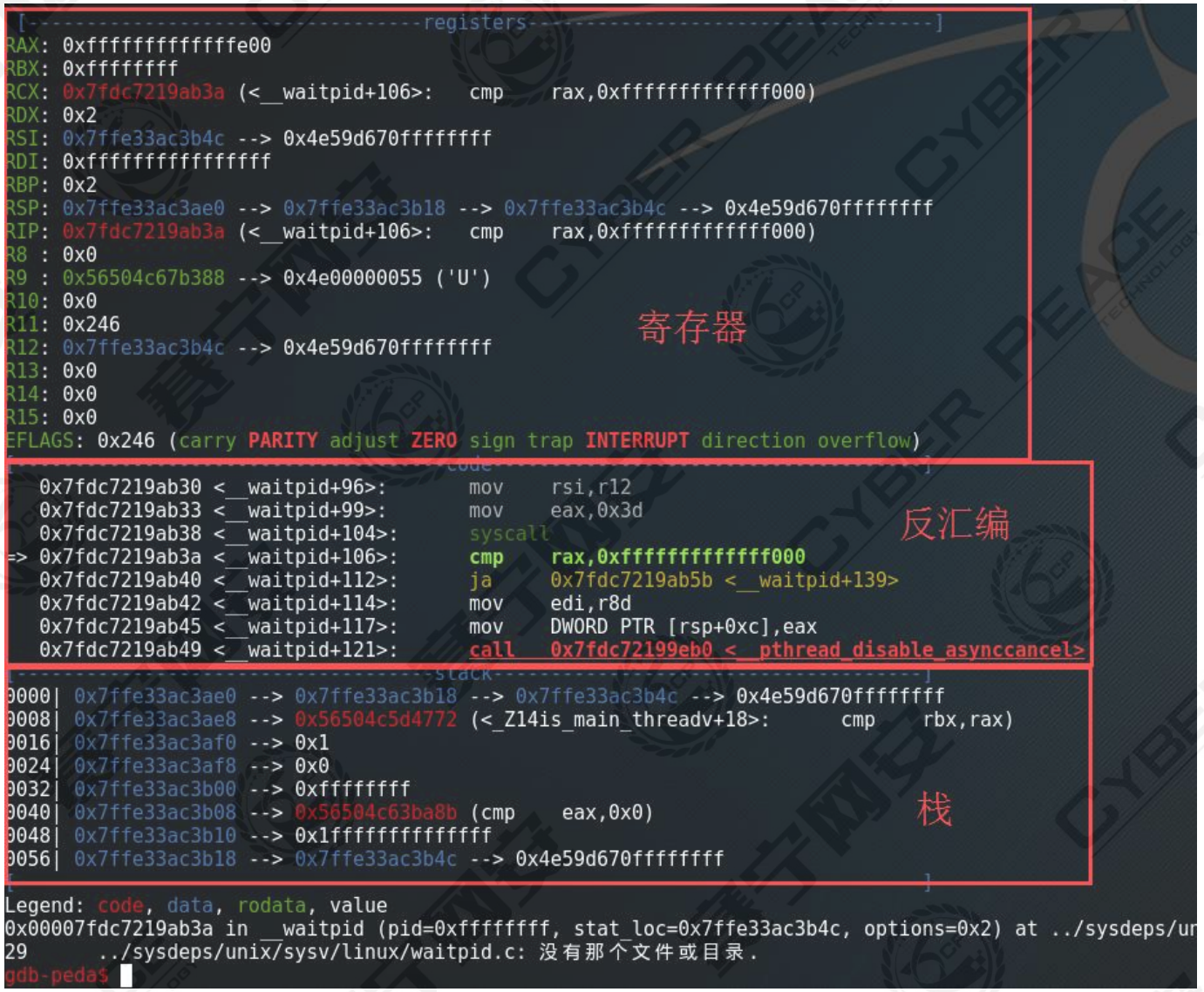
命令	功能
continue	恢复程序运行
finish	执行到函数退出
x	打印内存数据
p	打印表达式内容
command	断点触发时命令
step	单步步入
reverse-step	反向单步步入
next	单步步过
reverse-next	反向单步步过

GDB常用插件peda



https://github.com/longld/peda

GDB常用插件peda



GDB常用插件peda

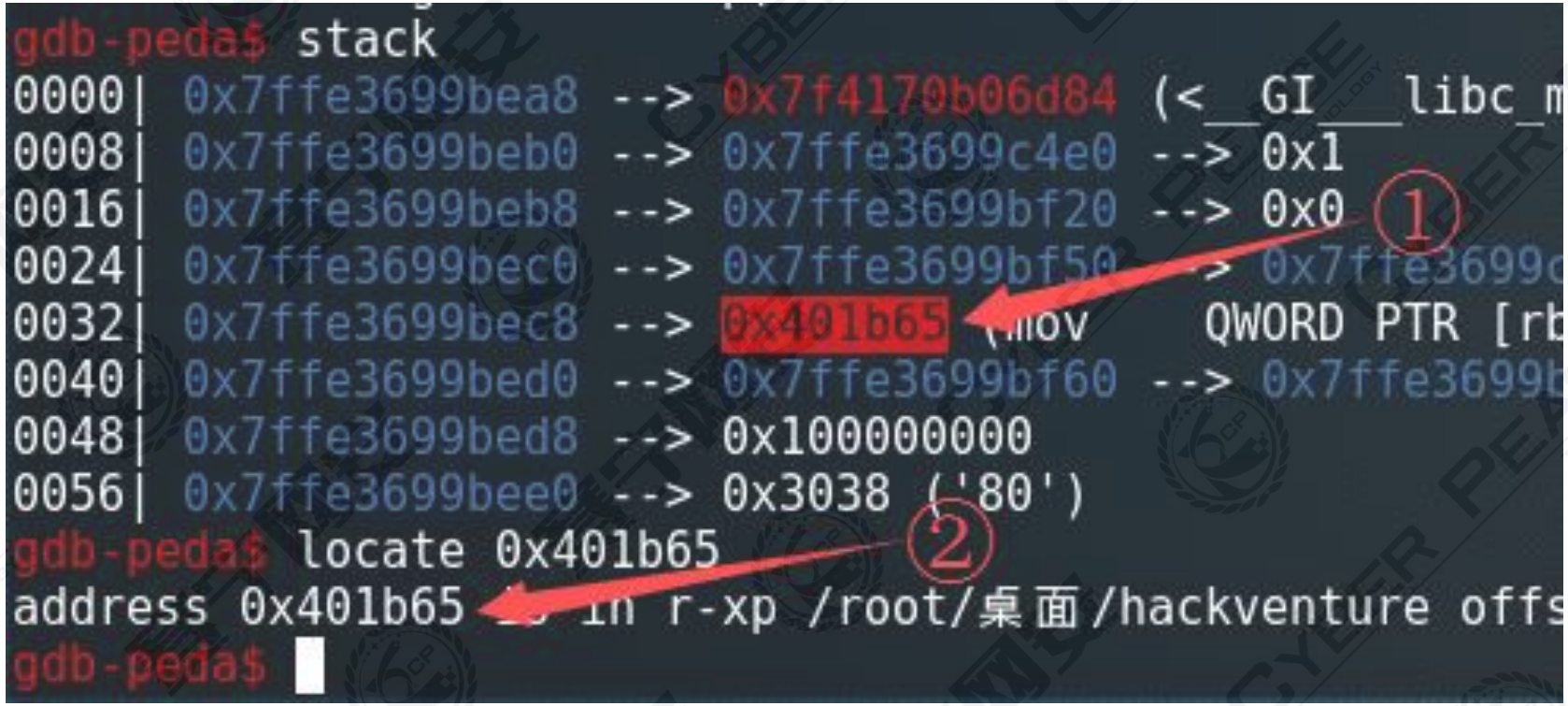
命令	功能
pattern_create	构造非重复字符串
pattern_offset	计算偏移值
pdisass	反汇编
checksec	查看保护措施
vmmap	查看段地址信息
ropgadget	寻找简单的ropgadget
goto	执行到指定地址
find	查找字符串、地址等
deactive	Bypass function

Vmmap查看内存布局

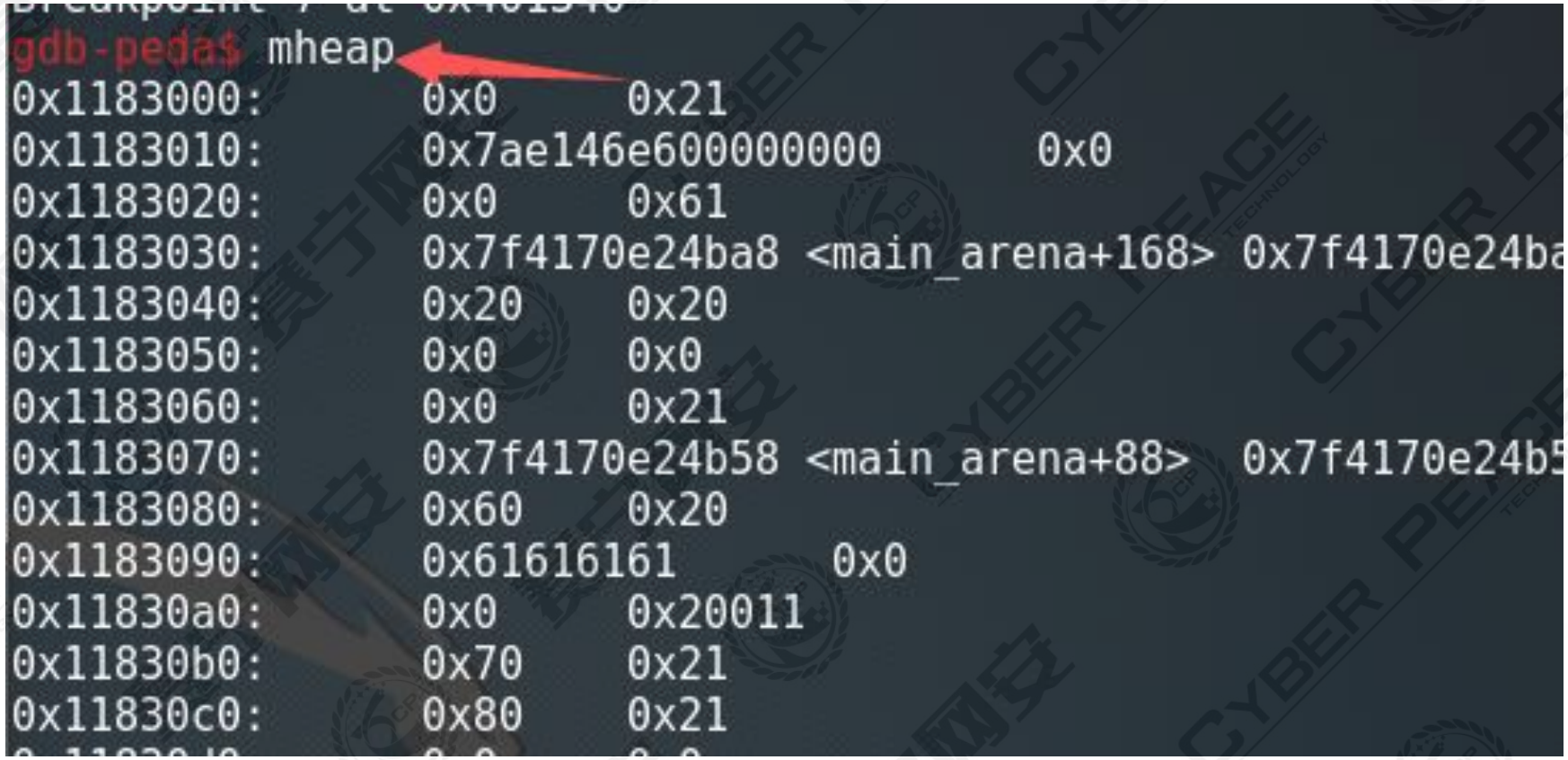
```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00402000         r-xp      /root/桌面/book
0x00601000         0x00602000         rw-p      /root/桌面/book
0x00007ffff7a3b000 0x00007ffff7bd0000 r-xp      /lib/x86_64-linux-gnu/libc-2.24.so
0x00007ffff7bd0000 0x00007ffff7dcf000 ---p      /lib/x86_64-linux-gnu/libc-2.24.so
0x00007ffff7dcf000 0x00007ffff7dd3000 r--p      /lib/x86_64-linux-gnu/libc-2.24.so
0x00007ffff7dd3000 0x00007ffff7dd5000 rw-p      /lib/x86_64-linux-gnu/libc-2.24.so
0x00007ffff7dd5000 0x00007ffff7dd9000 rw-p      mapped
0x00007ffff7dd9000 0x00007ffff7dfc000 r-xp      /lib/x86_64-linux-gnu/ld-2.24.so
0x00007ffff7fd3000 0x00007ffff7fd5000 rw-p      mapped
0x00007ffff7ff4000 0x00007ffff7ff7000 rw-p      mapped
0x00007ffff7ff7000 0x00007ffff7ffa000 r--p      [vvar]
0x00007ffff7ffa000 0x00007ffff7ffc000 r-xp      [vdso]
0x00007ffff7ffc000 0x00007ffff7ffd000 r--p      /lib/x86_64-linux-gnu/ld-2.24.so
0x00007ffff7ffd000 0x00007ffff7ffe000 rw-p      /lib/x86_64-linux-gnu/ld-2.24.so
0x00007ffff7ffe000 0x00007ffff7fff000 rw-p      mapped
0x00007ffffffde000 0x00007ffffffff000 rw-p      [stack]
0xffffffffff600000 0xffffffffff601000 r-xp      [vsyscall]
```

GDB功能扩充——基于peda

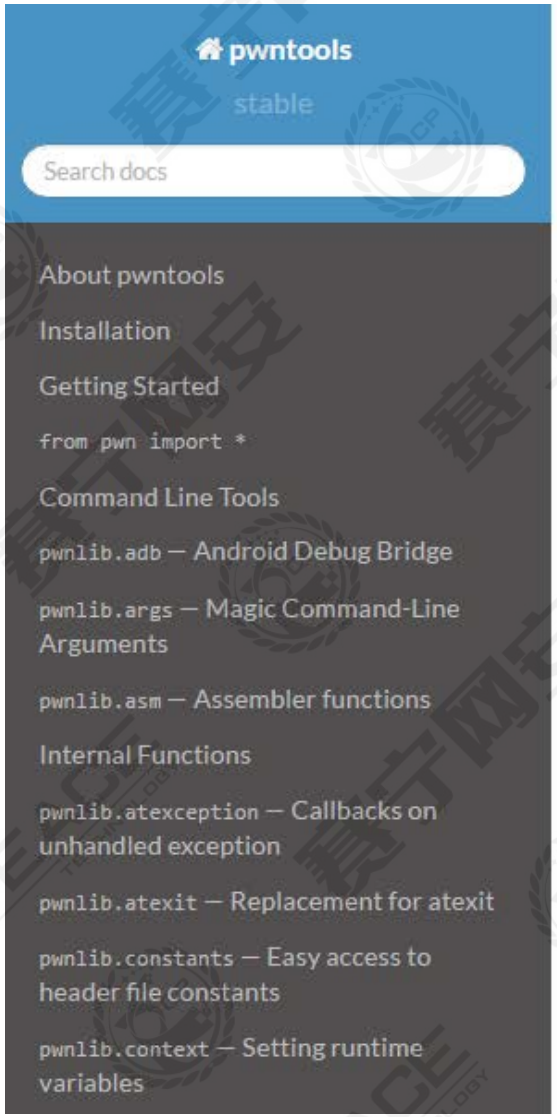
Peda.py--class PEDACmd



GDB功能扩充——基于GDB脚本



EXP脚本开发——pwntools



Docs » pwntools

Edit on GitHub

pwntools

`pwntools` is a CTF framework and exploit development library. Written in Python, it is designed for rapid prototyping and development, and intended to make exploit writing as simple as possible.

The primary location for this documentation is at docs.pwntools.com, which uses readthedocs. It comes in three primary flavors:

- Stable
- Beta
- Dev

Getting Started

- About pwntools
  - `pwn` — Toolbox optimized for CTFs
  - `pwnlib` — Normal python library
- Installation
  - Prerequisites
    - Binutils

pwntools常用模块

```
In [14]: p32(0x8048414)
Out[14]: '\x14\x84\x04\x08'

In [15]: hex(u32('\x14\x84\x04\x08'))
Out[15]: '0x8048414'

In [16]: p64(0x7fdeadbeaf)
Out[16]: '\xaf\xbe\xad\xde\x7f\x00\x00\x00'

In [17]: hex(u64('\xaf\xbe\xad\xde\x7f\x00\x00\x00'))
Out[17]: '0x7fdeadbeaf'
```

pwntools常用模块

```
In [19]: print shellcraft.execve('/bin/sh')
    /* execve(path='/bin/sh', argv=0, envp=0) */
    /* push '/bin/sh\x00' */
    push 0x1010101
    xor dword ptr [esp], 0x169722e
    push 0x6e69622f
    mov ebx, esp
    xor ecx, ecx
    xor edx, edx
    /* call execve() */
    push SYS_execve /* 0xb */
    pop eax
    int 0x80

In [20]: asm(shellcraft.execve('/bin/sh'))
Out[20]: 'h\x01\x01\x01\x01\x814$.ri\x01h/bin\x89\xe31
```

快速编写EXP脚本

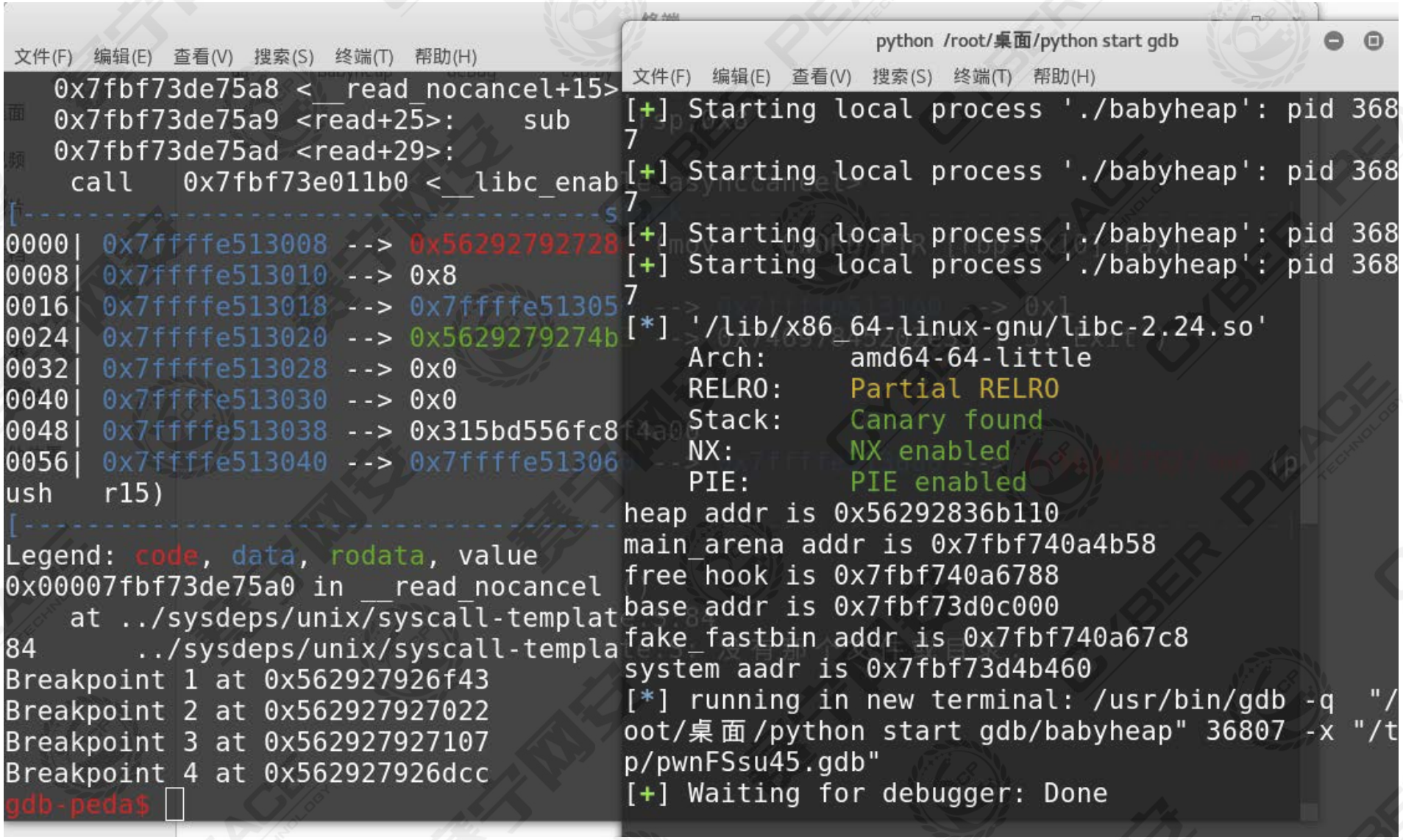
```python
from pwn import *

slog = 1
local = 1
debug = 1

if slog: context.log_level = True
if local:
    p = process('./pwnme')
else:
    p = remote('127.0.0.1', 8888)
if local and debug:
    gdb.attach(p, open('debug'))
//交互数据处理
//拿到shell
p.interactive()

```

快速编写EXP脚本


文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
0x7fbf73de75a8 <__read_nocancel+15>
0x7fbf73de75a9 <read+25>: sub
0x7fbf73de75ad <read+29>:
call 0x7fbf73e011b0 <__libc_enab
0000| 0x7ffffe513008 --> 0x56292792728
0008| 0x7ffffe513010 --> 0x8
0016| 0x7ffffe513018 --> 0x7ffffe51305
0024| 0x7ffffe513020 --> 0x5629279274b
0032| 0x7ffffe513028 --> 0x0
0040| 0x7ffffe513030 --> 0x0
0048| 0x7ffffe513038 --> 0x315bd556fc8
0056| 0x7ffffe513040 --> 0x7ffffe51306
ush r15)
Legend: code, data, rodata, value
0x00007fbf73de75a0 in __read_nocancel
at ../sysdeps/unix/syscall-templat
84 ../sysdeps/unix/syscall-templa
Breakpoint 1 at 0x562927926f43
Breakpoint 2 at 0x562927927022
Breakpoint 3 at 0x562927927107
Breakpoint 4 at 0x562927926dcc
gdb-peda$
python /root/桌面/python start gdb
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
[+] Starting local process './babyheap': pid 368
7
[+] Starting local process './babyheap': pid 368
7
[+] Starting local process './babyheap': pid 368
[+] Starting local process './babyheap': pid 368
7
[*] '/lib/x86_64-linux-gnu/libc-2.24.so'
Arch: amd64-64-little
RELRO: Partial RELRO
Stack: Canary found
NX: NX enabled
PIE: PIE enabled
heap addr is 0x56292836b110
main_arena addr is 0x7fbf740a4b58
free_hook is 0x7fbf740a6788
base_addr is 0x7fbf73d0c000
fake_fastbin addr is 0x7fbf740a67c8
system aadr is 0x7fbf73d4b460
[*] running in new terminal: /usr/bin/gdb -q "/
oot/桌面/python start gdb/babyheap" 36807 -x "/t
p/pwnFSsu45.gdb"
[+] Waiting for debugger: Done

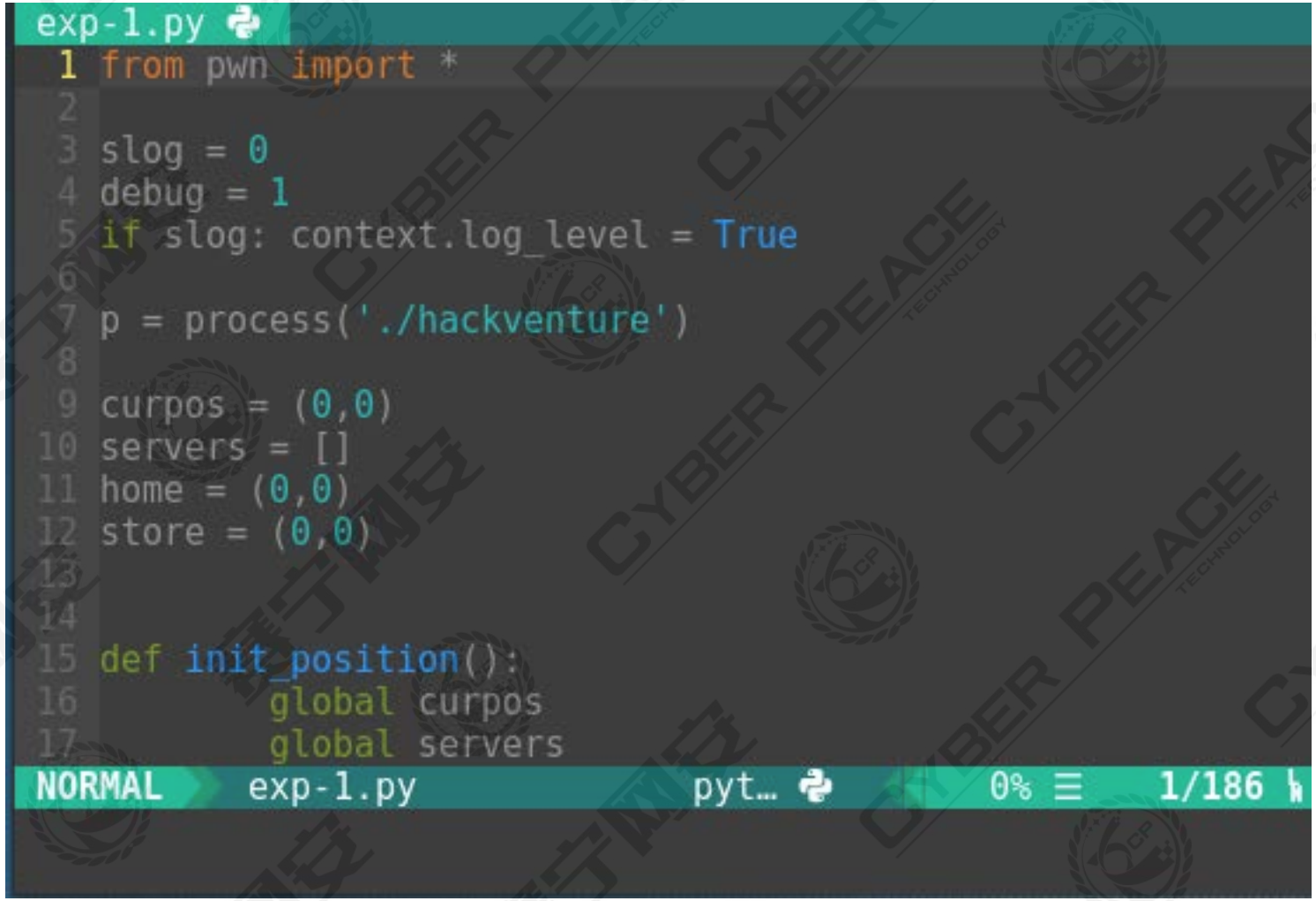
脚本编辑器——Vimplus

优势

功能强大
纯命令行操作
调试方便

缺点

学习难度大
配置复杂



https://github.com/chxuan/vimplus

Libc源码跟踪调试

准备工作

sudo apt-get install libc6-dbg
sudo apt-get source libc6-dev

加载源码

directory ~/desktop/glibc-2.24/malloc/

```
Breakpoint 8, _int_malloc (av=av@entry=0x7f4170e24b00 <main_arena>,
    bytes=bytes@entry=0x51) at malloc.c:3354
3354    {
gdb-peda$ list
3349      ------------------------------ malloc ------------------------------
3350       */
3351
3352    static void *
3353    _int_malloc (mstate av, size_t bytes)
3354    {
3355      INTERNAL_SIZE_T nb;               /* normalized request size */
3356      unsigned int idx;                 /* associated bin index */
3357      mbinptr bin;                      /* associated bin */
3358
```