BINARY IN CTF

Liuyikun@nsfocus.com

单击此处编辑母版标题样式

- WeaponX

- 安全研究员
- 威胁情报与网络安全实验室

TOOLS

WINDOWS

- IDA with Hex-Ray Decompile
- OllyDebug
- Windbg
- Processexp
- Metasploit
- Mona(python)
- dnspy

LINUX

- Gdb
- Pwntools(python)
- Peda
- Checksec
- Metasploit
- Roputils(python)
- Rp++(python)

调试工具

- 静态分析工具
  - IDA with Hex-Ray Decompile
- 动态调试工具
  - OllyDebug
  - windbg
  - gdb
  - dnspy(.NET 程序解包调试工具)

辅助程序

- processexp
  - 微软出品的进程信息读取程序
- peda
  - gdb的增强插件
- pwntools
  - 快速开发exploit的python模块
- Roputils
  - 快速开发rop chain的python模块
- checksec
  - 检查ELF文件的安全措施

辅助程序

- Rp++
  - 搜索ROP gadget(ELF)
- Mona
  - 搜索ROP gadget(PE)

BASICS

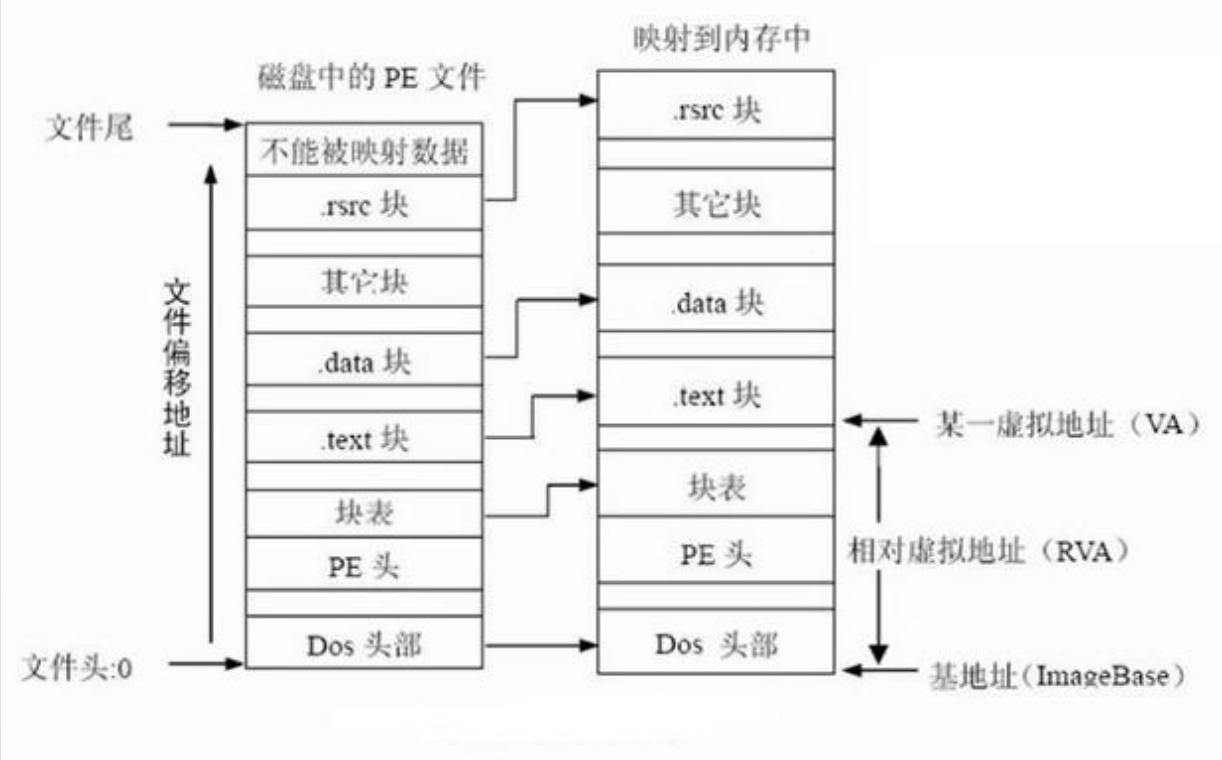
WINDOWS

- PE
  - Portable Executable
- Suffix
  - EXE, DLL, SYS

LINUX

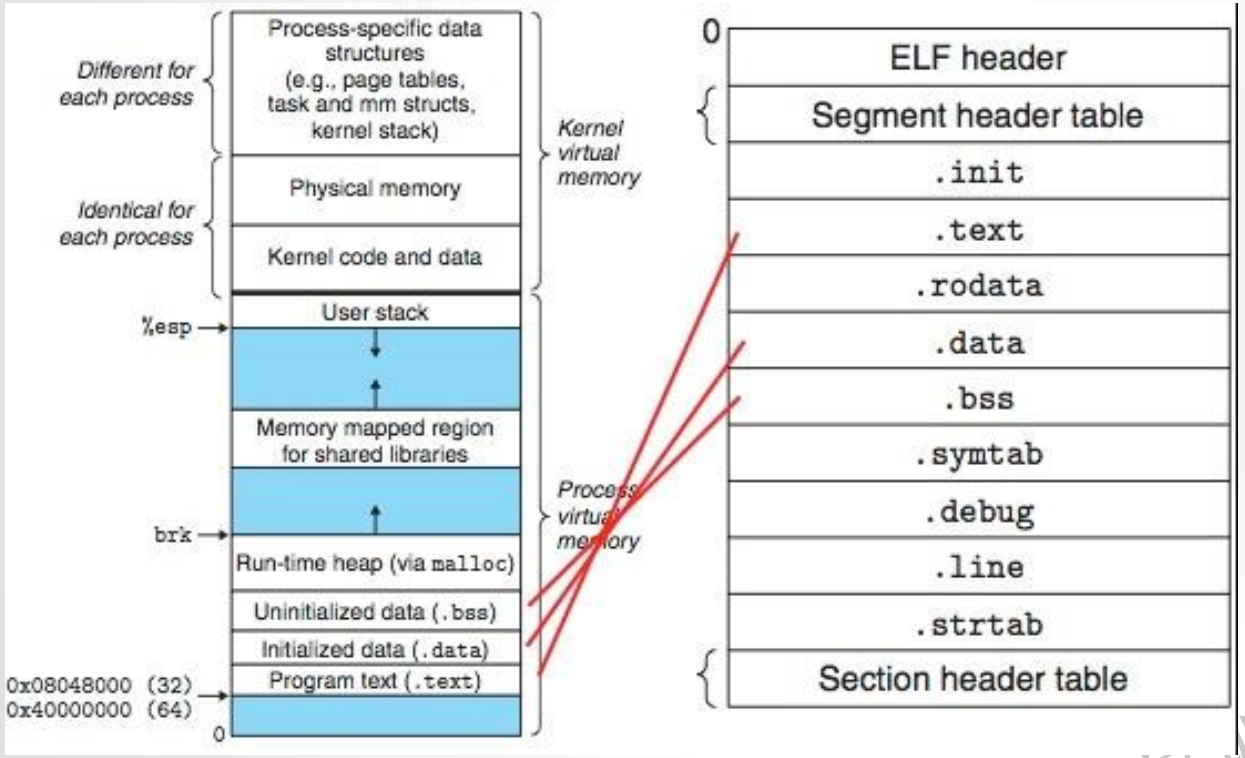
- ELF
  - Executable and Linkable Format
- Suffix
  - 0, so, elf

PE文件结构



- .rsrc
  - 模块的资源信息
- .data
  - 变量信息
- .text
  - 可执行代码段

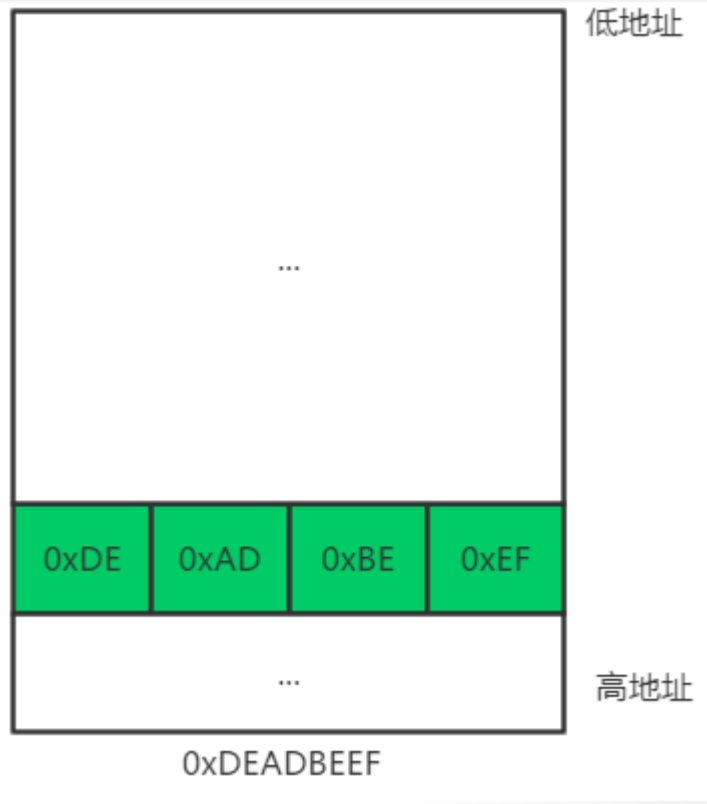
ELF文件结构



- .text
  - 已编译程序的机器代码
- .symtab
  - 符号表它存放在程序中被定义和引用的函数和全局变量的信息
- .strtab
  - 一个字符串表，其内容包括.symtab和.debug节中的符号表
- .bss
  - 存放程序中未初始化的全局变量和静态变量的一块内存区域。特点是可读写的

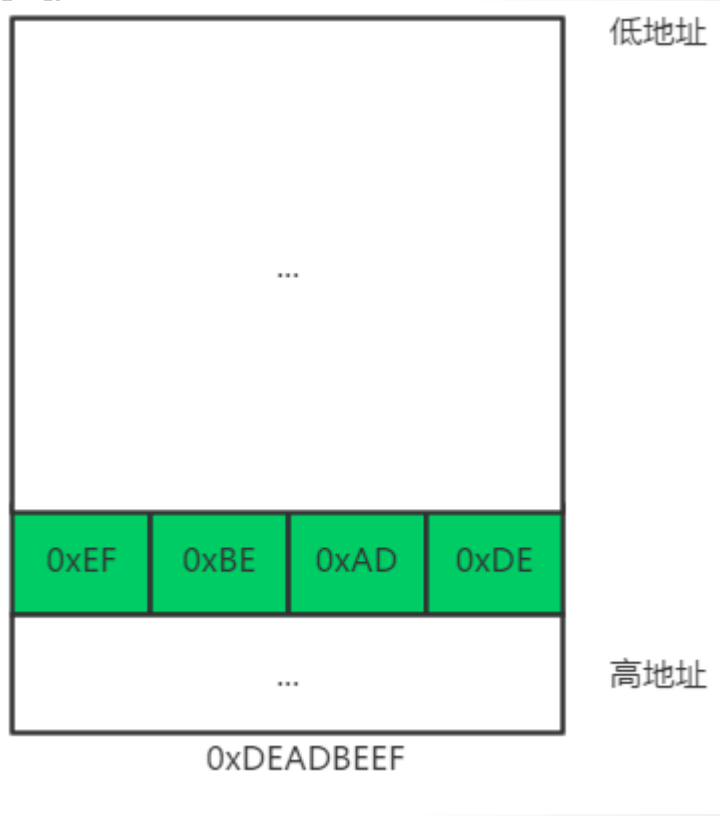
大端模式

- Big-Endian
- Arch
  - MIPS
  - ARM
- 高字节保存在内存的低地址中，低字节保存在内存的高地址中



小端模式

- Little-Endian
- Arch
  - 80x86
- 高字节保存在内存的高地址中，低字节保存在内存的低地址中



- 数据传输指令
- 算术运算指令
- 逻辑运算指令
- 程序转移指令

数据传输指令

- MOV
  - MOV eax, [ESP]
- PUSH
- POP
- LEA
  - LEA eax, [0xABCD]

算术运算指令

- ADD
- INC
- SUB
- DEC
- MUL/IMUL
- DIV/IDIV

逻辑运算指令

- AND
- OR
- XOR
- NOT
- SHL
- SHR

程序转移指令

- 无条件转移
  - JMP
  - CALL
  - RET/RETN
- 有条件转移
  - J[Condition]
    - JG/JNLE 大于转移.
      JGE/JNL 大于或等于转移.
      JL/JNGE 小于转移.
      JLE/JNG 小于或等于转移

返回指令

- RET
  - POP eip
- LEAVE
  - MOV esp, ebp
  - POP ebp

- 4个数据寄存器(EAX、EBX、ECX和EDX)
- 2个变址和指针寄存器(ESI和EDI) 2个指针寄存器(ESP和EBP)
- 6个段寄存器(ES、CS、SS、DS、FS和GS)
- 1个指令指针寄存器(EIP) 1个标志寄存器(EFlags)

- 12个数据寄存器(RAX、RBX、RCX、RDX、R8-R15)
- 2个变址和指针寄存器(RSI和RDI) 2个指针寄存器(RSP和RBP)
- 6个段寄存器(ES、CS、SS、DS、FS和GS)
- 1个指令指针寄存器(RIP) 1个标志寄存器(RFlags)

汇编书写格式

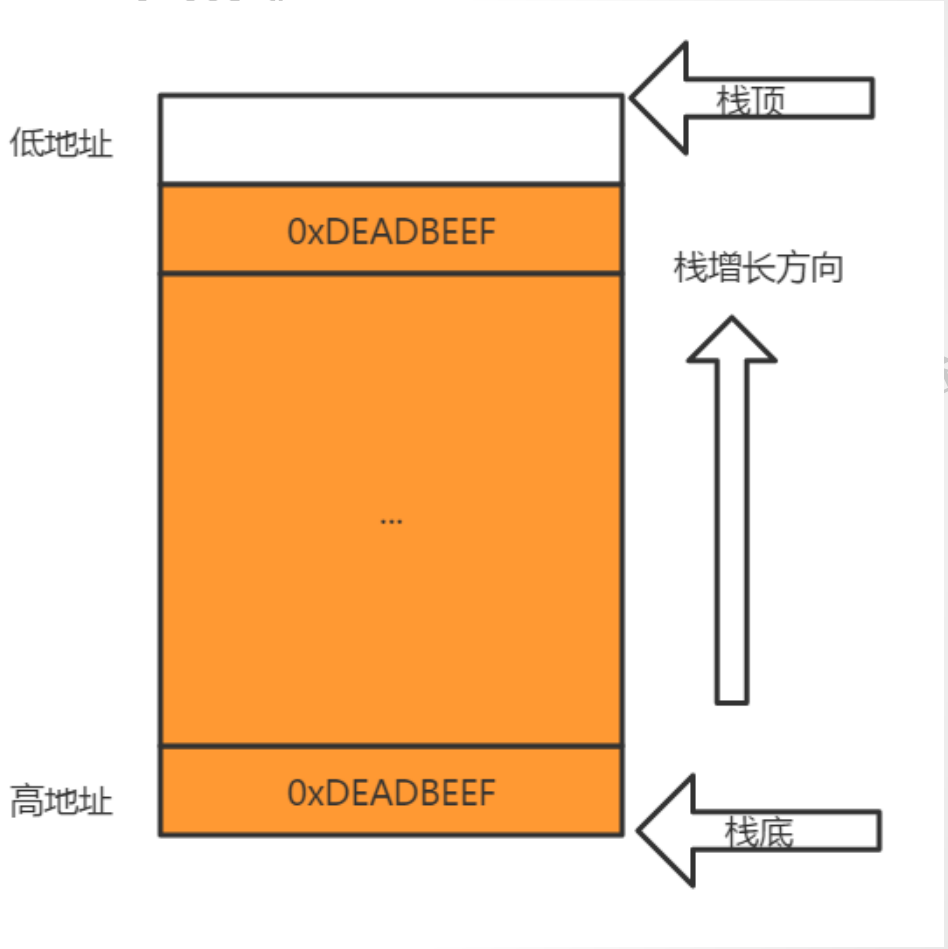
INTEL

```
push   ebp
mov    ebp,esp
sub    esp,0x10
mov    DWORD PTR [ebp-0x4],0x0
mov    DWORD PTR [ebp-0x4],0x3
mov    eax,0x0
leave
ret
```

AT&T

```
push   %ebp
mov    %esp,%ebp
sub    $0x10,%esp
movl   $0x0,-0x4(%ebp)
movl   $0x3,-0x4(%ebp)
mov    $0x0,%eax
leave
ret
```

- 数据结构
  - (LIFO, Last In First Out, 后进先出)
- 存储内容
  - 变量， 函数调用信息
- 栈增长方向
  - 高地址 -> 低地址



栈操作相关寄存器

- ESP(RSP, x86_64)
  - Extended Stack Pointer
  - 当前栈顶指针
- EBP(RBP, x86_64)
  - Extended Base Pointer
  - 系统栈最上层的栈的底部

常用栈操作

- POP eax
  - SUB esp, 4
  - MOV eax, [esp]

- PUSH eax
  - MOV [esp], eax
  - ADD esp, 4

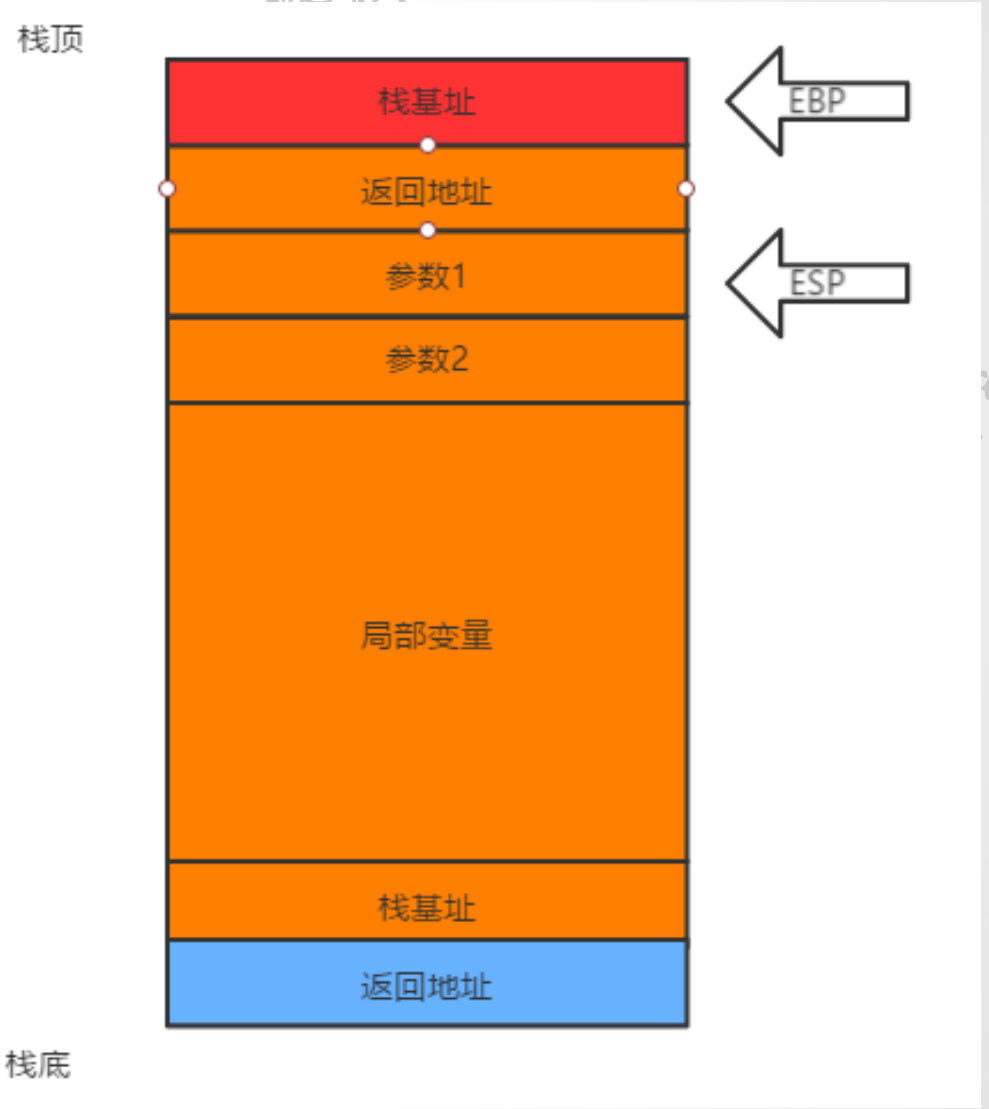
- PUSH eax
  - SUB esp, 4
  - MOV [esp], eax

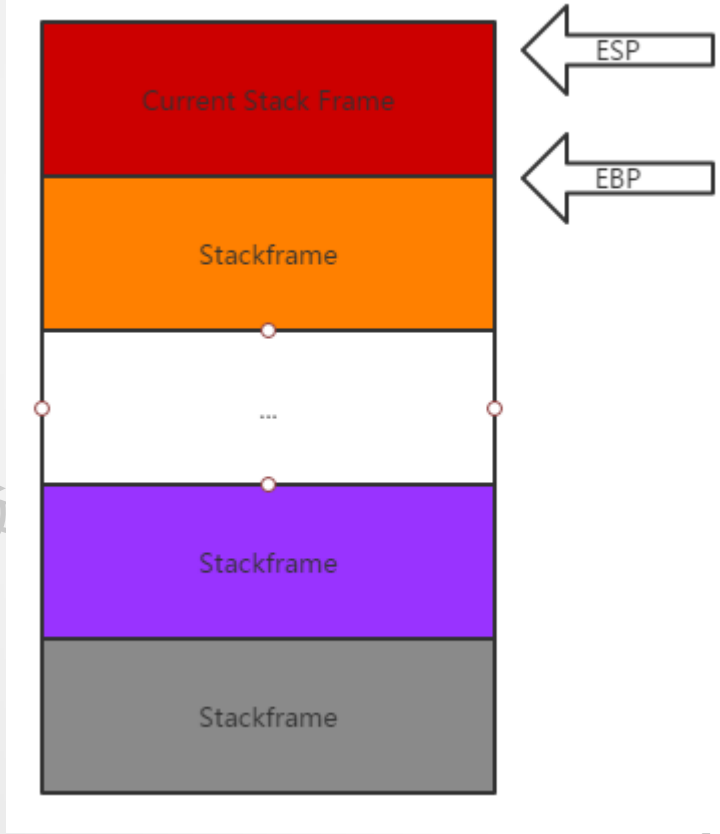
```
int main()
{
        __asm {
                sub eax, eax;
                mov eax, 0xdeadbeef;
                push eax;
                sub eax, eax;
                mov eax, 0xdeadbeef;
                sub esp, 4;
                mov[esp], eax;
        }
    return 0;
}
```

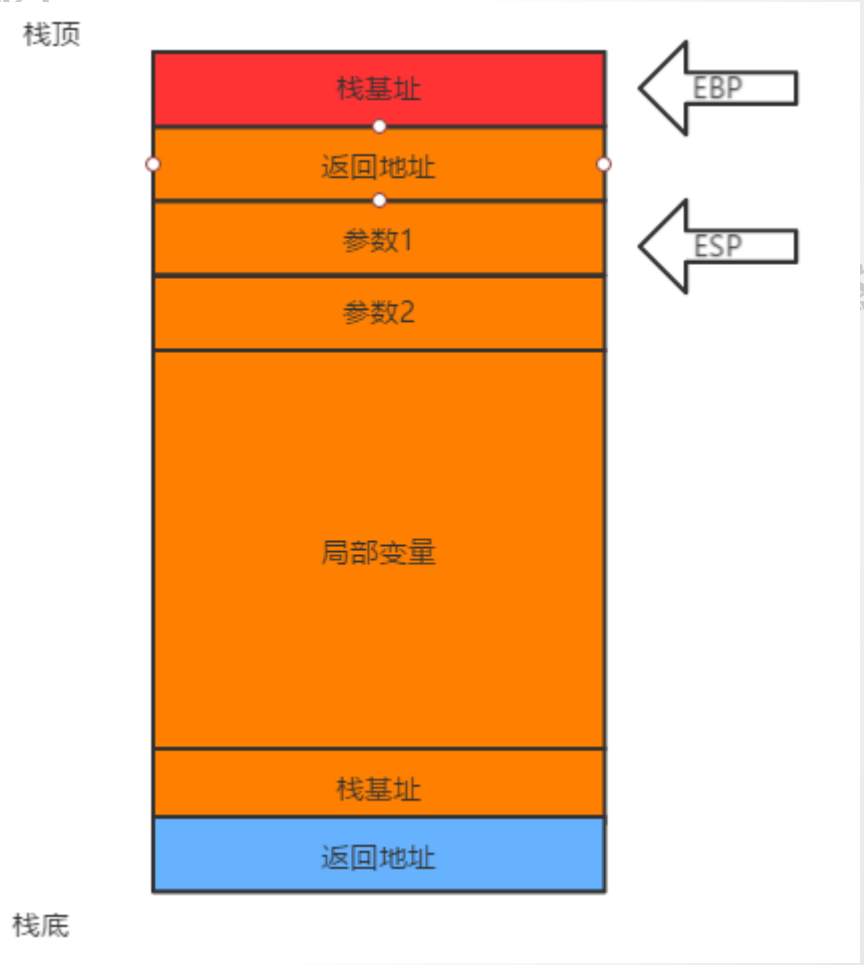
- POP eax
  - MOV eax, [esp]
  - ADD esp, 4

```
int main()
{
        __asm {
                sub eax, eax;
                pop eax;
                sub eax, eax;
                mov eax, [esp];
                add esp, 4;
        }
    return 0;
}
```

- 栈帧也叫过程活动记录，是编译器用来实现过程/函数调用的一种数据结构。







函数传参规则

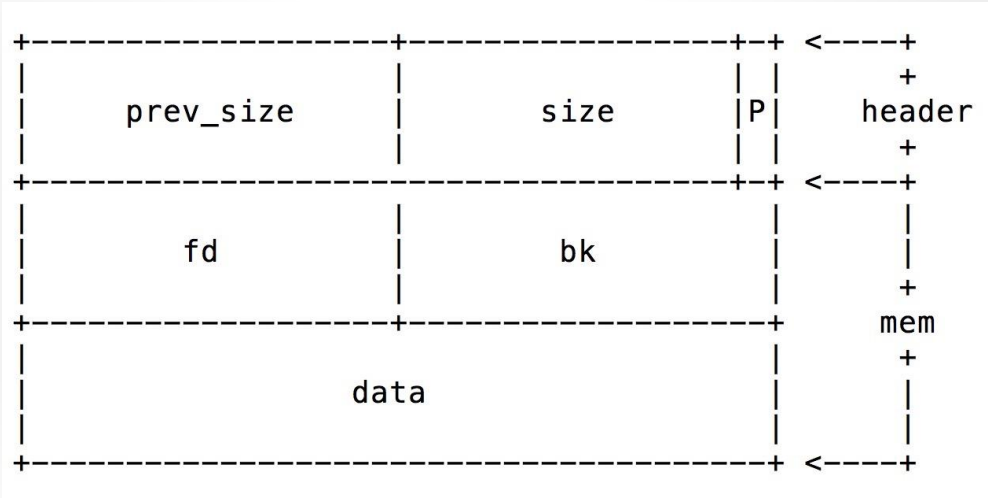
- x86
  - 所有参数都在栈上
  - 函数返回值在eax寄存器中
- x86_64
  - 函数参数依次存在rdi、rsi、rdx、rcx、r8、r9中
  - 若参数超过6个的话，之后的参数存在栈上
  - 函数返回值在rax寄存器中

调用模式演示

```
int function(int var1,int var2)
{
        int sum = 0;
        sum = var1 + var2;
        return sum;
}
int main()
{
        int sum = 0;
        sum = function(1, 2);
        return 0;
}
```

```
.text:00401010 ; =============== S U B R O U T I N E =======================================
.text:00401010
.text:00401010
.text:00401010 sub_401010      proc near               ; CODE XREF: _main+10↓p
.text:00401010
.text:00401010 arg_0           = dword ptr  4
.text:00401010 arg_4           = dword ptr  8
.text:00401010
.text:00401010                 mov     eax, [esp+arg_4]
.text:00401014                 mov     ecx, [esp+arg_0]
.text:00401018                 add     eax, ecx
.text:0040101A                 retn    8
.text:0040101A sub_401010      endp
.text:0040101A
.text:0040101A ; ---------------------------------------------------------------------------
.text:0040101D                 align 10h
.text:00401020
.text:00401020 ; =============== S U B R O U T I N E =======================================
.text:00401020
.text:00401020
.text:00401020 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401020 _main           proc near               ; CODE XREF: start+AF↓p
.text:00401020                 push    2
.text:00401022                 push    1
.text:00401024                 call    sub_401000
.text:00401029                 add     esp, 8
.text:0040102C                 push    2
.text:0040102E                 push    1
.text:00401030                 call    sub_401010
.text:00401035                 xor     eax, eax
.text:00401037                 retn
.text:00401037 _main           endp
.text:00401037
```

- 数据结构
  - (FIFO, First In First Out, 先进先出)
- 存储内容
  - 任意数据



堆和栈的对比

HEAP

- 运行时动态分配
- 程序员申请
- 程序员回收
- 地址空间内“杂乱的”

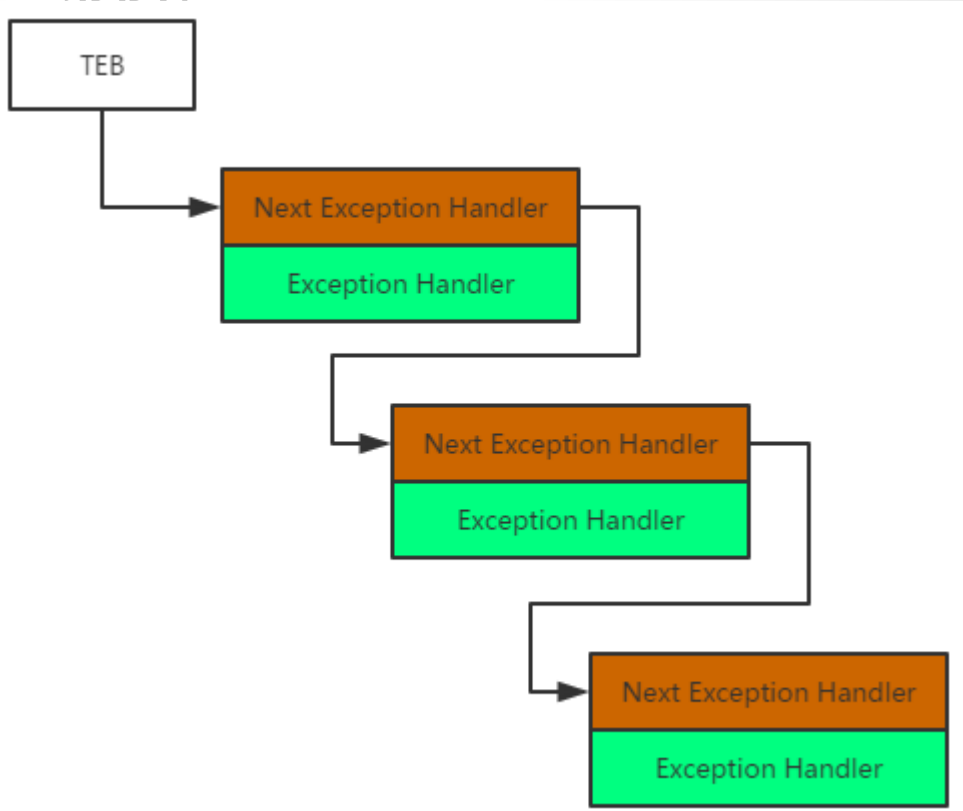
STACK

- 预先分配
- 系统自动分配
- 系统自动回收
- 地址空间内“整齐的”

堆管理机制

- 堆表
  - 空表
    - 将空闲堆块组织成双向链表
    - 空闲堆块大小 = 索引index x 8 字节
  - 块表
    - 堆组织为单向链表
    - 不发生堆块合并，加快分配速度
- 堆块合并
  - 系统发现物理地址相连的堆块都未使用，则合并这两个堆块为一个堆块

- Structure Exception Handler
- 异常处理结构体
- 特点：
  - 链表
  - 存放在栈中
  - 顺序判断
  - 都无法处理时，调用系统默认处理函数



常见函数调用方式

- _stdcall
  - Windows API默认的函数调用规则
- _cdecl
  - C/C++默认的函数调用规则
- _fastcall
  - 适用于对性能要求较高的场合

调用方式对比

_STDCALL

- 函数参数由右向左入栈
- 函数调用结束后由被调用函数清除栈内数据

_CDECL

- 函数参数由右向左入栈
- 函数调用结束后由调用者函数清除栈内数据

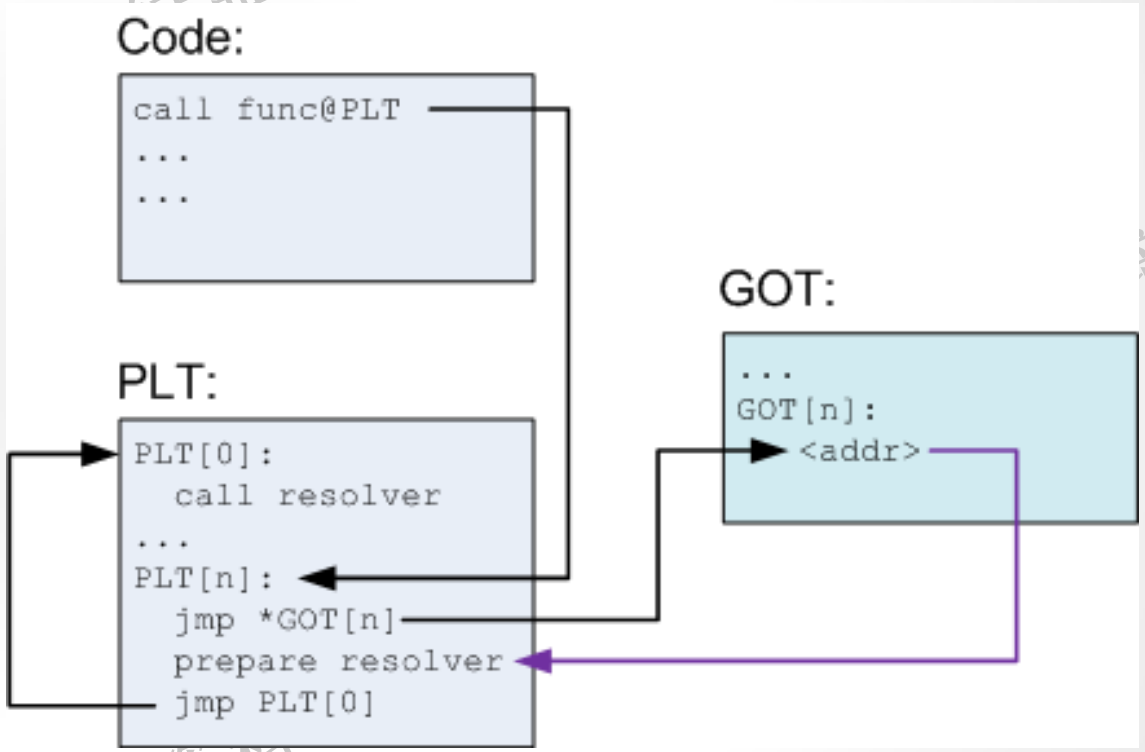
- GOT(Global Offset Table)
- PLT(Procedure Linkage Table)
- Lazy Binding(延迟绑定)

- Global Offset Table
- 全局偏移表
- 作用：
  - 存储解析出来的函数地址

- Procedure Linkage Table
- 过程链接表
- 作用：
  - 从GOT中找到函数地址
  - 若GOT中没有，则解析地址填入GOT

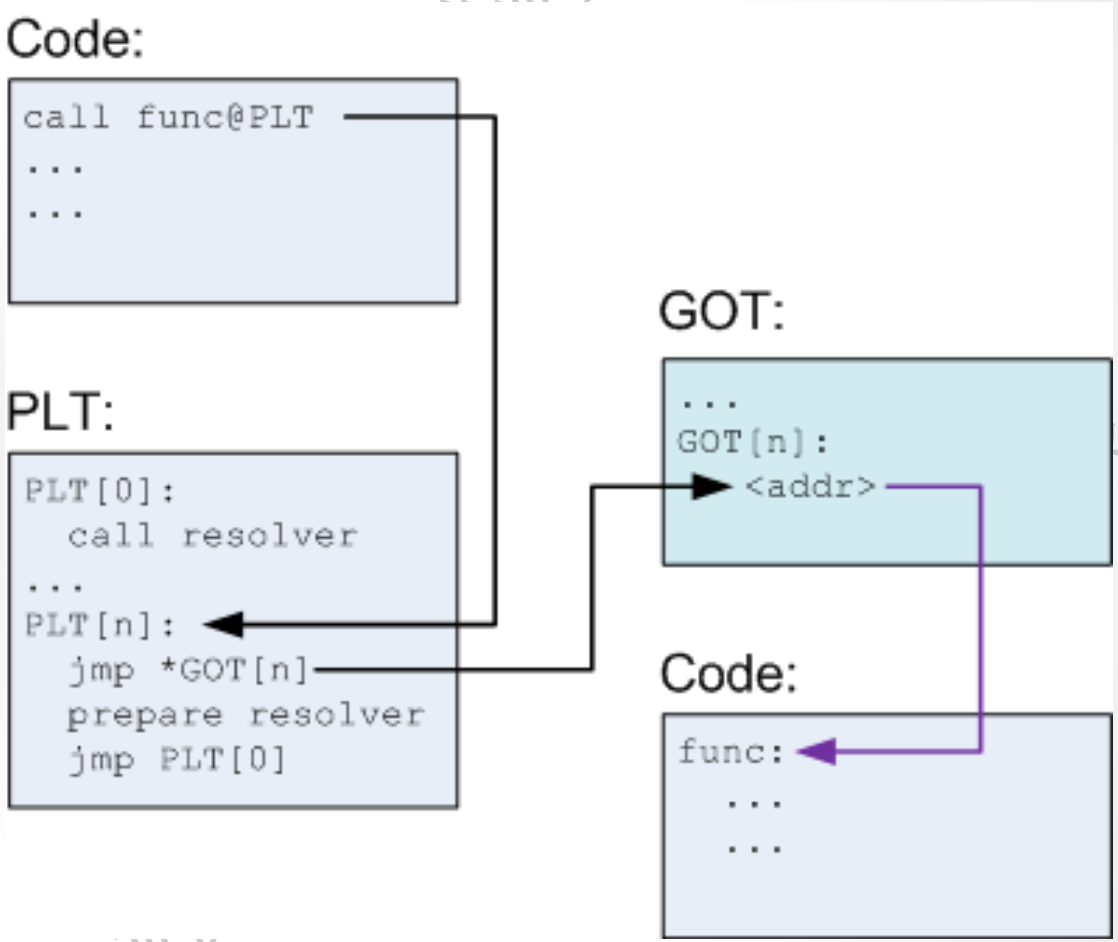
延时绑定

- First time:
  - func@plt -> PLT -> GOT -> PLT(resolve address)
  - _dl_runtime_resolve
  - _dl_fixup



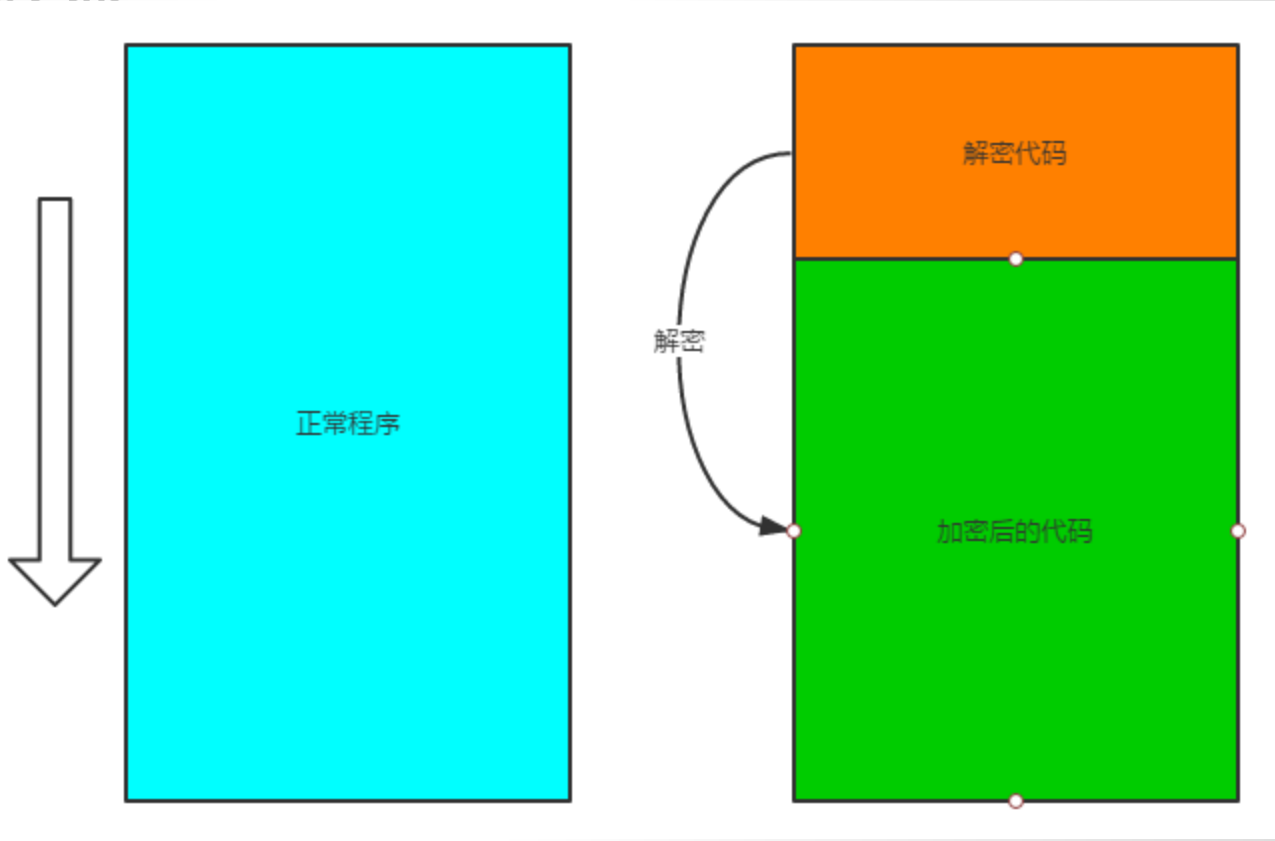
延时绑定

- Second time:
  - func@plt -> PLT -> GOT



REVERSE

- 压缩壳
  - 作用：压缩资源，减少体积
  - UPX、ASPack
- 加密壳
  - 作用：提供保护，提供额外功能
  - ASProtect、Armadillofan
- 虚拟机壳
  - 作用：提供增强保护
  - VMProtect、Themida



- ESP定律
  - 试用范围：压缩壳，部分加密壳
- 原理
  - 如果我们要返回父程序，则当我们在堆栈中进行堆栈的操作的时候，一定要保证在RET这条指令之前，ESP指向的是我们压入栈中的地址。这也就是著名的“堆栈平衡”原理

无法脱壳的情况

- 带壳调试
- 关键API下断点分析
  - Windows 上的MFC程序
  - 弹窗
    - MessageBox(A/W)
  - 读写文件
    - ReadFile/WriteFile/CreateFile
  - 反调试退出
    - ExitProcess

压缩壳脱壳

实验

花指令

- 作用
  - 干扰逆向分析人员分析程序
- 类别
  - 可执行的
  - 不执行的
- 解决方法
  - 使用OD插件去除花指令

- 查找通用调试器
- 检测专用调试器
- 检测断点
- 检测跟踪
- 检测补丁

常见的反调试函数

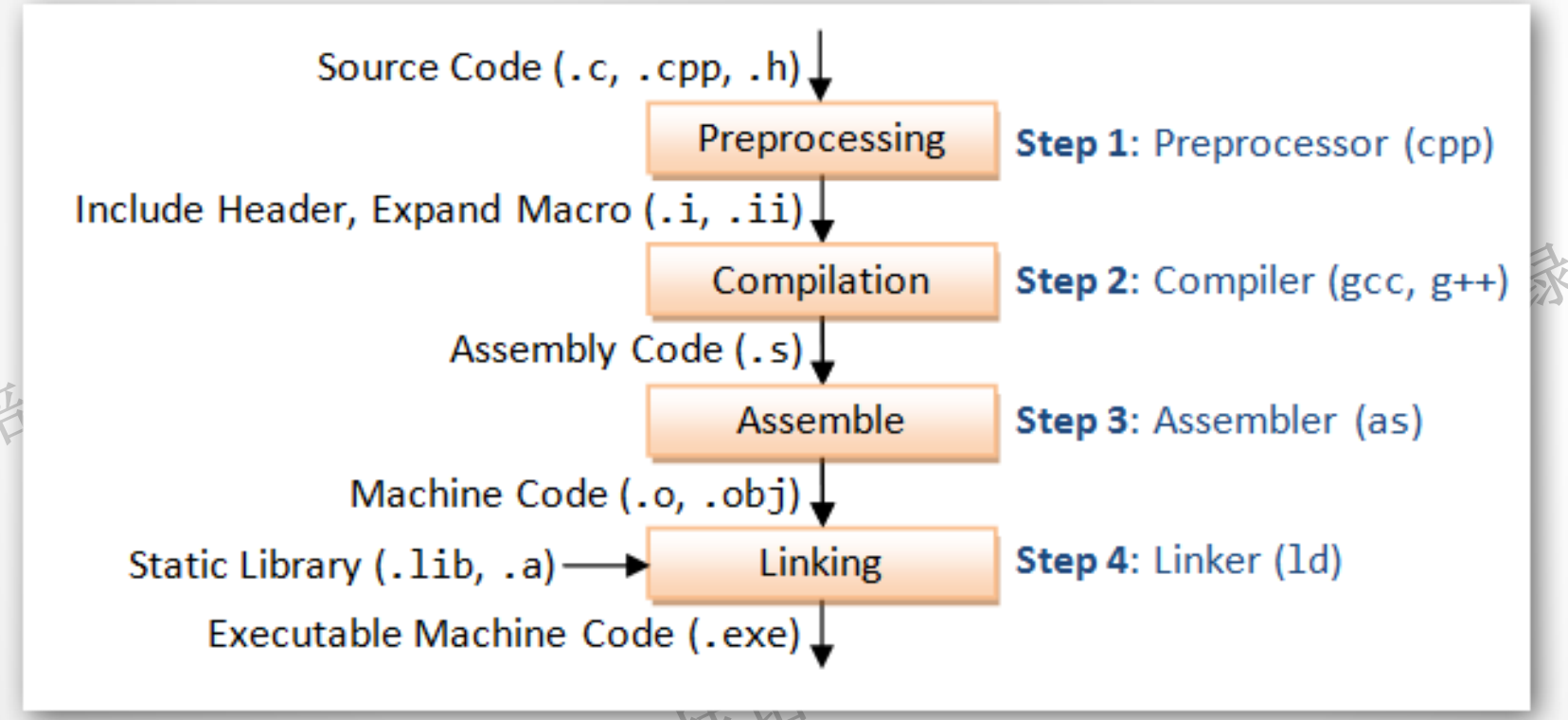
- IsDebuggerPresent
- TLS_callback
- 解决方法：
  - 找到函数调用者
  - 删除反调试函数

- 快速定位key比对函数
  - 使用OD字符串搜索插件
- 使用F5插件
  - 快速理解加密/解密函数
- 使用脱壳机
  - 尽量使用脱壳机节省时间

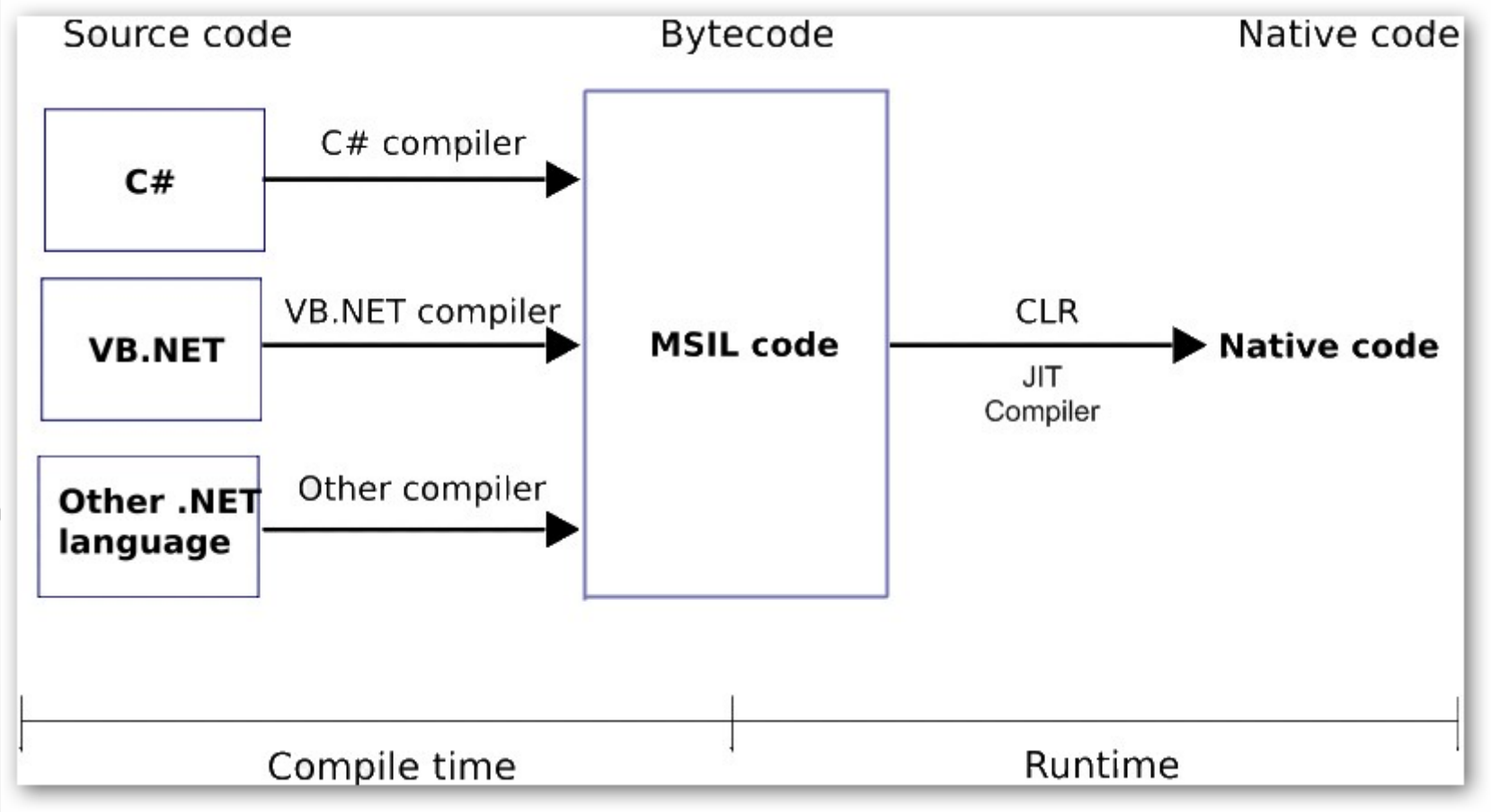
实验

- C#
  - ILSPY
  - .Net Reflector & reflexil
- Java
  - JD-GUI
- Python
  - Uncompyle2
- 方法
  - 直接源码审计

二进制程序编译流程



.NET 程序编译流程



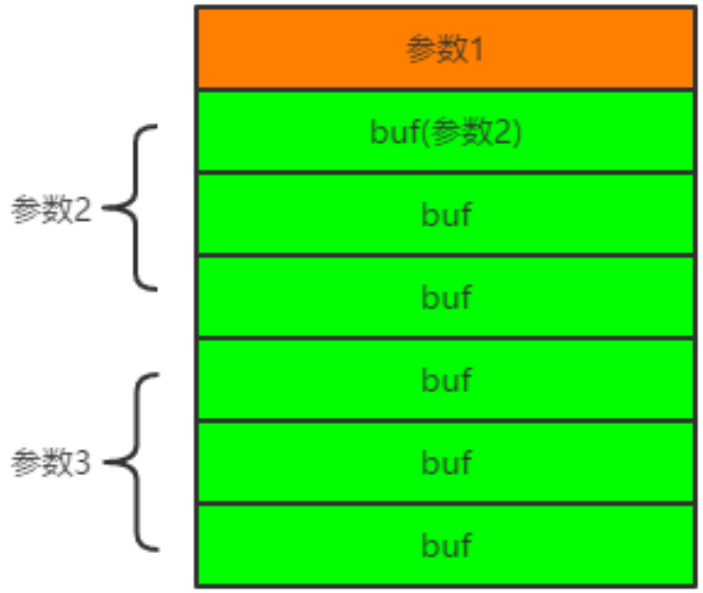
- 混淆
  - 名称混淆
    - 将代码里的常量，变量，函数名变为随机字符串，增加阅读难度
  - 流程混淆
    - 在代码中插入不影响程序的虚假代码
- 加壳

实验

PWN

- Stack Overflow(栈溢出)
- Heap Overflow(堆溢出)
- Integer Overflow(整形溢出)
- Format String Vulnerabilities(格式化字符串)
- Use After Free(释放后重用)

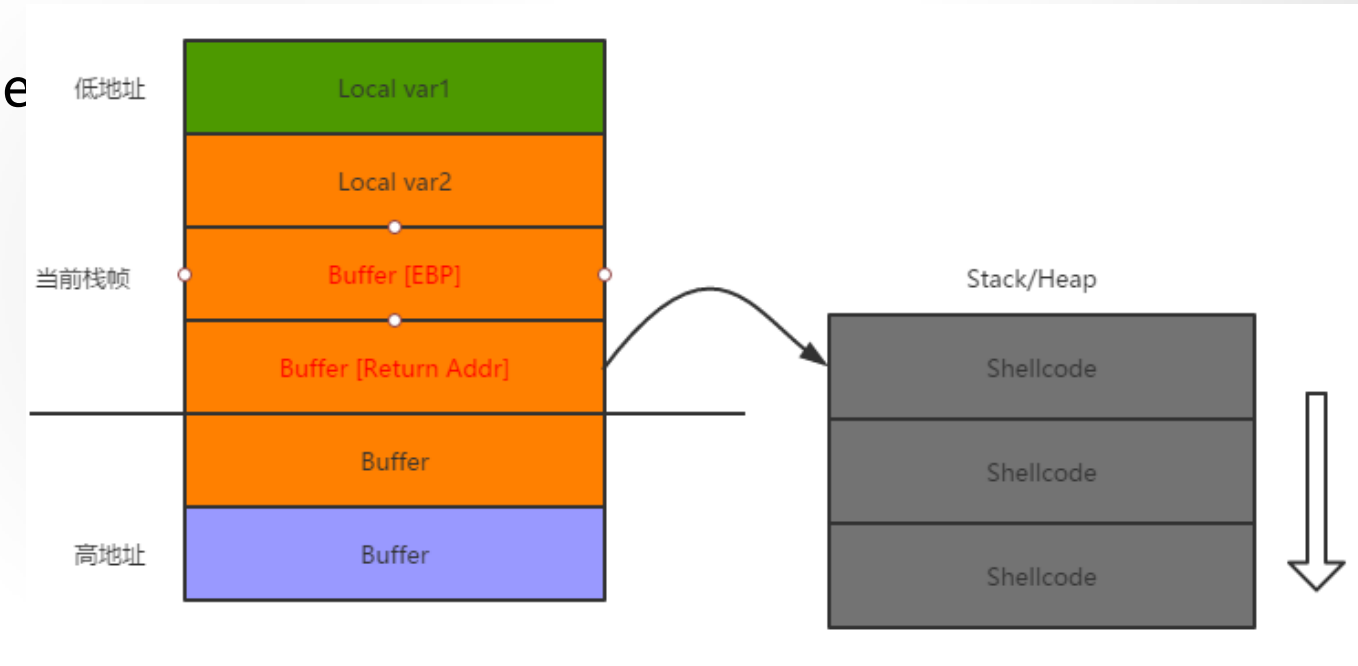
- 保证程序需要，程序会预先分配一定大小的内存空间
- 向分配的内存空间写入超过分配的大小
- 溢出的数据会改写分配空间以外的内存单元



STACK OVERFLOW EXAMPLE

```
void vulnerable_function() {
        char buf[128];
        read(STDIN_FILENO, buf, 256);
}
int main(int argc, char** argv) {
        vulnerable_function();
        write(STDOUT_FILENO, "Hello, World\n", 13);
}
```

- 利用方式
  - 输入超长字符串
  - 造成程序存储越界
  - 覆盖返回地址
  - 将返回地址指向预先填充She
    的内存空间



实验

STACK OVERFLOW PROTECTION

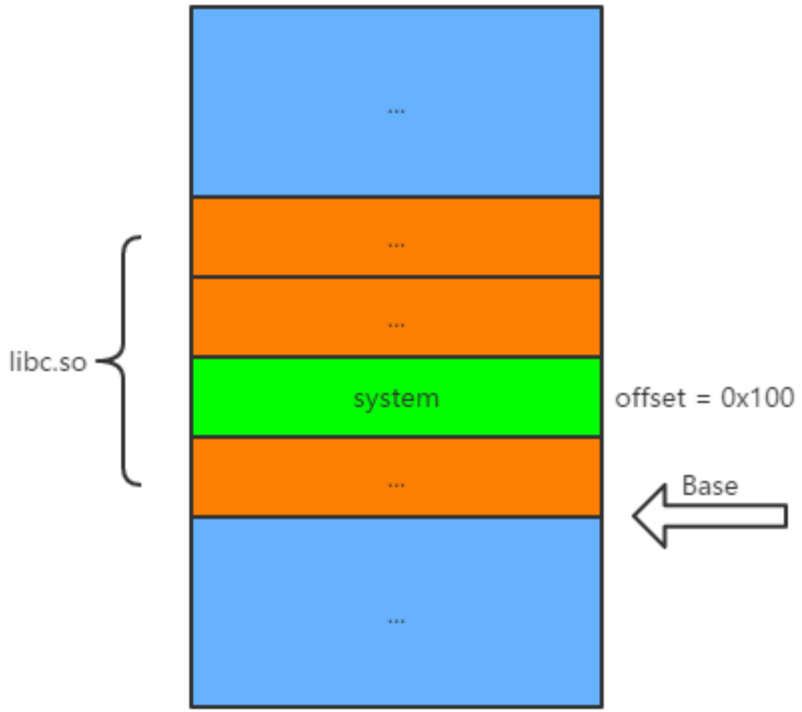
WINDOWS

- GS
- DEP
- ASLR
- SafeSEH
- SEHOP

LINUX

- RELRO
- Canary
- NX
- PIE
- PIC

- Address Space Layout Randomization
- 地址空间分布随机化
- Addr = base + offset
  - Base -> random
- Bypass
  - Leak memory(泄漏内存)
  - Attack no-ASLR module
  - Heap Spray(堆喷射)



给LIBC.SO

- 查找system地址
- 查找write地址
- 查找’ /bin/sh’ 地址
- 计算地址偏移
- 通过got得到write地址
- 计算出system地址

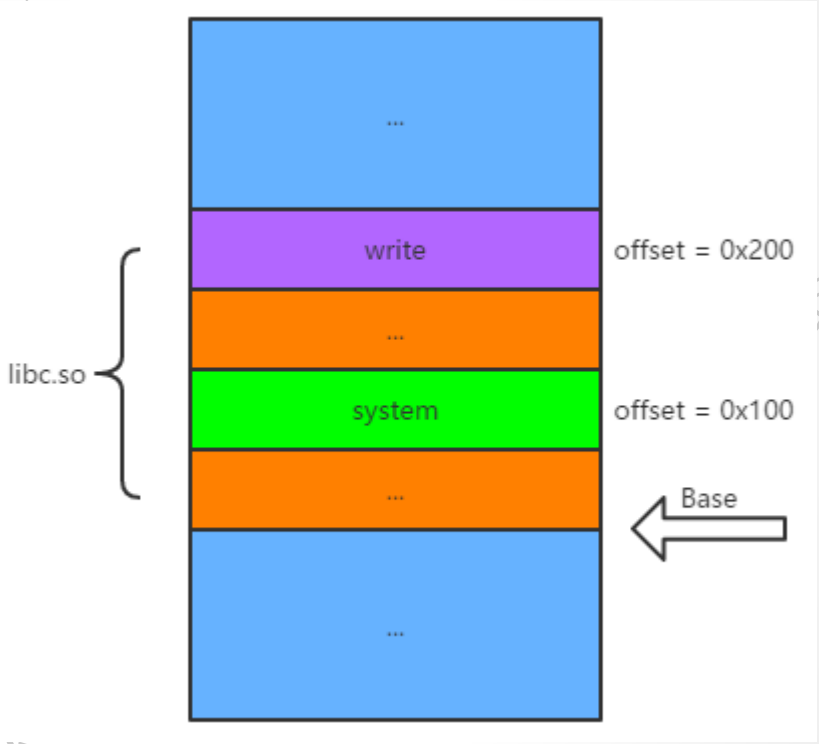
不给LIBC.SO

- 泄漏system地址
- 泄漏write地址
- 计算偏移
- 通过got得到write地址
- 计算出system地址

```
➜  pwn readelf -a libc.so.6| grep system
   243: 00116930    73 FUNC    GLOBAL DEFAULT   12 svcerr_systemerr@@GLIBC_2.0
   620: 0003fe70    56 FUNC    GLOBAL DEFAULT   12 __libc_system@@GLIBC_PRIVATE
  1443: 0003fe70    56 FUNC    WEAK   DEFAULT   12 system@@GLIBC_2.0
➜  pwn readelf -a libc.so.6| grep write
  W (write), A (alloc), X (execute), M (merge), S (strings)
   108: 0006b950   323 FUNC    GLOBAL DEFAULT   12 _IO_wdo_write@@GLIBC_2.2
   182: 000d9900   119 FUNC    WEAK   DEFAULT   12 __write@@GLIBC_2.0
   306: 001256a0    45 FUNC    GLOBAL DEFAULT   12 _IO_do_write@GLIBC_2.0
   307: 0006f1b0    45 FUNC    GLOBAL DEFAULT   12 _IO_do_write@@GLIBC_2.1
   521: 000eb3a0    78 FUNC    GLOBAL DEFAULT   12 process_vm_writev@@GLIBC_2.15
   523: 000c1eb0   184 FUNC    WEAK   DEFAULT   12 __pwrite64@@GLIBC_2.1
   908: 000e1570   160 FUNC    WEAK   DEFAULT   12 writev@@GLIBC_2.0
  1330: 000c1d30   190 FUNC    GLOBAL DEFAULT   12 __libc_pwrite@@GLIBC_PRIVATE
  1621: 000e1c60   251 FUNC    GLOBAL DEFAULT   12 pwritev@@GLIBC_2.10
  1677: 000ea440    62 FUNC    GLOBAL DEFAULT   12 eventfd_write@@GLIBC_2.7
  1692: 00063de0   372 FUNC    WEAK   DEFAULT   12 fwrite@@GLIBC_2.0
  1981: 000e1ec0   229 FUNC    GLOBAL DEFAULT   12 pwritev64@@GLIBC_2.10
  2140: 00125100   108 FUNC    GLOBAL DEFAULT   12 _IO_file_write@GLIBC_2.0
  2144: 0006e210   131 FUNC    GLOBAL DEFAULT   12 _IO_file_write@@GLIBC_2.1
  2165: 00063de0   372 FUNC    GLOBAL DEFAULT   12 _IO_fwrite@@GLIBC_2.0
  2186: 000c1d30   190 FUNC    WEAK   DEFAULT   12 pwrite@@GLIBC_2.1
  2247: 00068cc0   157 FUNC    GLOBAL DEFAULT   12 fwrite_unlocked@@GLIBC_2.1
  2256: 000c1eb0   184 FUNC    WEAK   DEFAULT   12 pwrite64@@GLIBC_2.1
  2303: 000d9900   119 FUNC    WEAK   DEFAULT   12 write@@GLIBC_2.0
➜  pwn python -c "print hex(0xd9900-0x3fe70)"
0x99a90
```

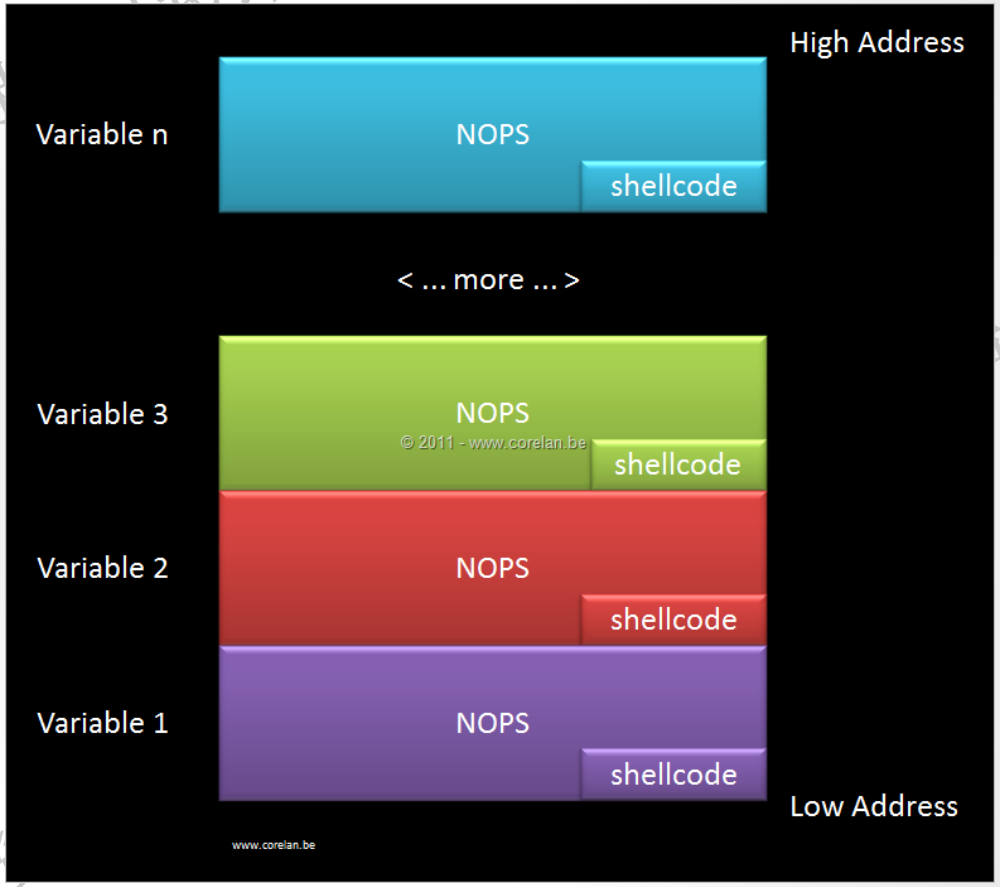
```
Breakpoint 1, 0x08048479 in main ()
gdb-peda$ find "/bin/sh"
Searching for '/bin/sh' in: None ranges
Found 1 results, display max 1 items:
libc : 0xf7f75a8c ("/bin/sh")
gdb-peda$ x/s 0xf7f75a8c
0xf7f75a8c:     "/bin/sh"
gdb-peda$
```

- 原理
  - 模块中不同函数的相对位置不变
- 步骤
  - 找一个程序中的导入函数
  - 计算这个函数和需要函数的位置差
  - 下次加载时利用导入函数地址和位置差计算需要的函数地址
  - 调用需要的函数



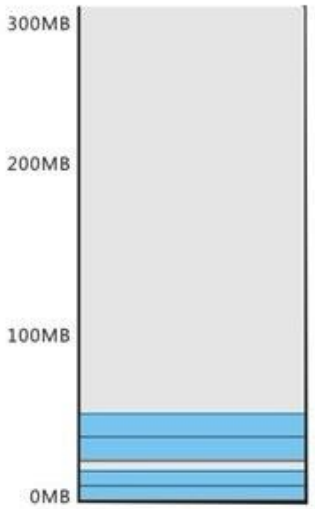
实验

- 控制EIP稳定落入Shellcode的技术
- 方法：
  - 分配大量内存块
  - 在内存块中部署超长滑行区slide
  - 在内存块末尾部署shellcode



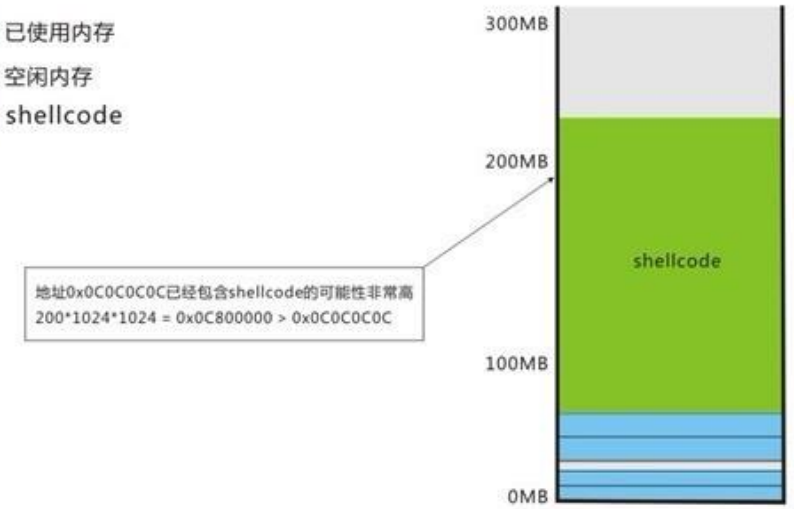
常规情况下堆布局

：已使用内存
：空闲内存



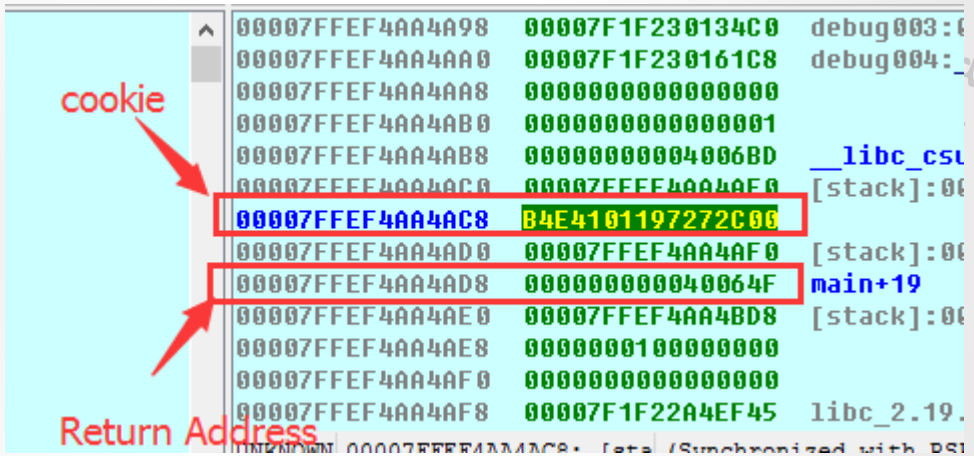
堆喷（heap spraying)之后

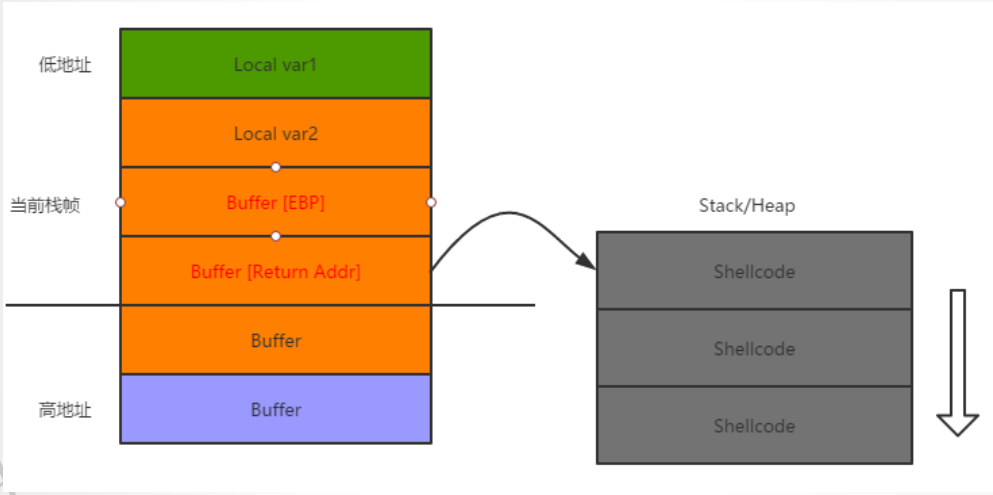
：已使用内存
：空闲内存
：shellcode

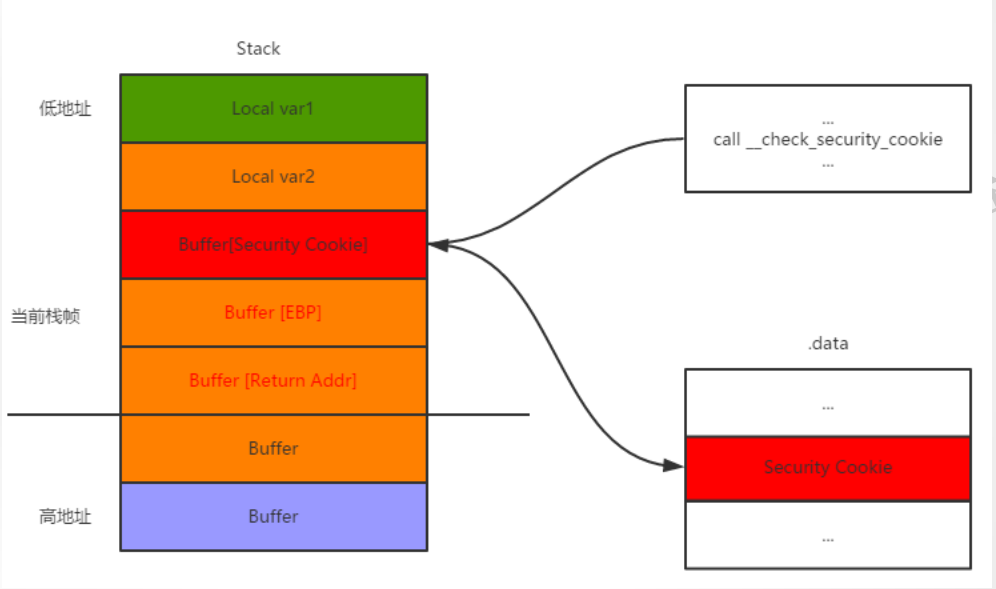


- 栈溢出检测
  - 在栈帧中存放一个随机值cookie，在函数返回时检测这个cookie是否被修改
- Bypass
  - Leak memory(泄漏cookie)
  - 覆盖 SEH (Windows)

```
[-------------------------------------code-------------------------------------]
   0x40061b <vulnerable_function+46>:    call   0x4004d0 <read@plt>
   0x400620 <vulnerable_function+51>:    mov    rax,QWORD PTR [rbp-0x8]
   0x400624 <vulnerable_function+55>:    xor    rax,QWORD PTR fs:0x28
=> 0x40062d <vulnerable_function+64>:    je     0x400634 <vulnerable_function+71>
 | 0x40062f <vulnerable_function+66>:    call   0x4004c0 <__stack_chk_fail@plt>
 | 0x400634 <vulnerable_function+71>:    leave
 | 0x400635 <vulnerable_function+72>:    ret
 | 0x400636 <main>:     push   rbp
 |->   0x400634 <vulnerable_function+71>:       leave
       0x400635 <vulnerable_function+72>:       ret
       0x400636 <main>: push   rbp
       0x400637 <main+1>:       mov    rbp,rsp
                                                                  JUMP is taken
[-------------------------------------stack------------------------------------]
```





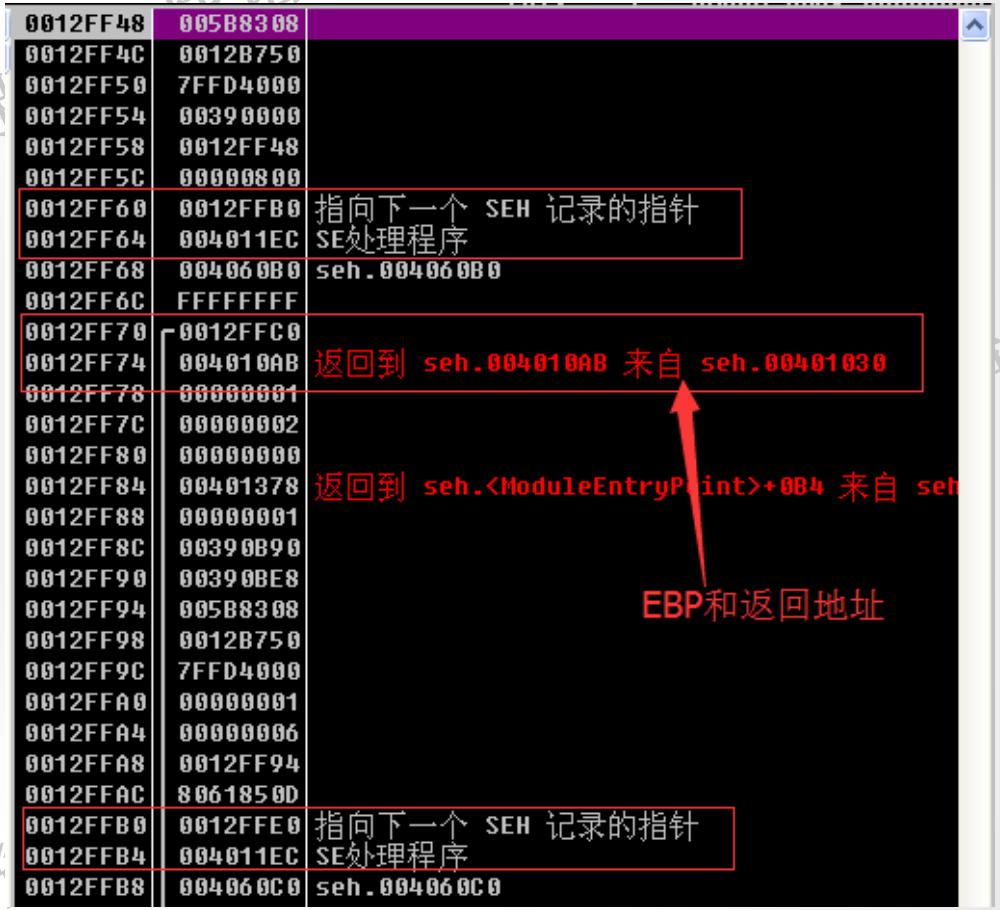


- 内存泄漏
  - Fork()后程序的security cookie不会变
- 覆盖SEH
  - SEH在栈上
  - 触发程序异常
  - 跳入篡改后的SEH执行shellcode

SEH 覆盖

```
void testOwSEH()
{
    printf("test");
}
void MyExceptionHandler()
{
    printf("Found Exception");
    ExitProcess(0);
}
int foo(int a, int b, int c)
{
    int sum = 0;
    sum = a + b;
    __try{
        sum = sum / c;
    }
    __except(MyExceptionHandler()){}

    return sum;
}
int main()
{
    foo(1,2,0);
    testOwSEH();
    return 0;
}
```



seh - Microsoft Visual C++ - [seh.cpp]

File Edit View Insert Project Build Tools Window Help

seh | Win32 Release

(Globals) | (All global members) | main

seh classes

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

void testOwSEH()
{
    printf("test");
}

void MyExceptionHandler()
{
    printf("Found Exception");
    ExitProcess(0);
}

int foo(int a, int b, int c)
{
    int sum = 0;
    sum = a + b;
    __try{
        sum = sum / c;
    }
    __except(MyExceptionHandler()){}

    return sum;
}


int main()
{
    foo(1,2,0);
    testOwSEH();
    return 0;
}
```

ClassV... FileView

seh.exe - 0 error(s), 0 warning(s)

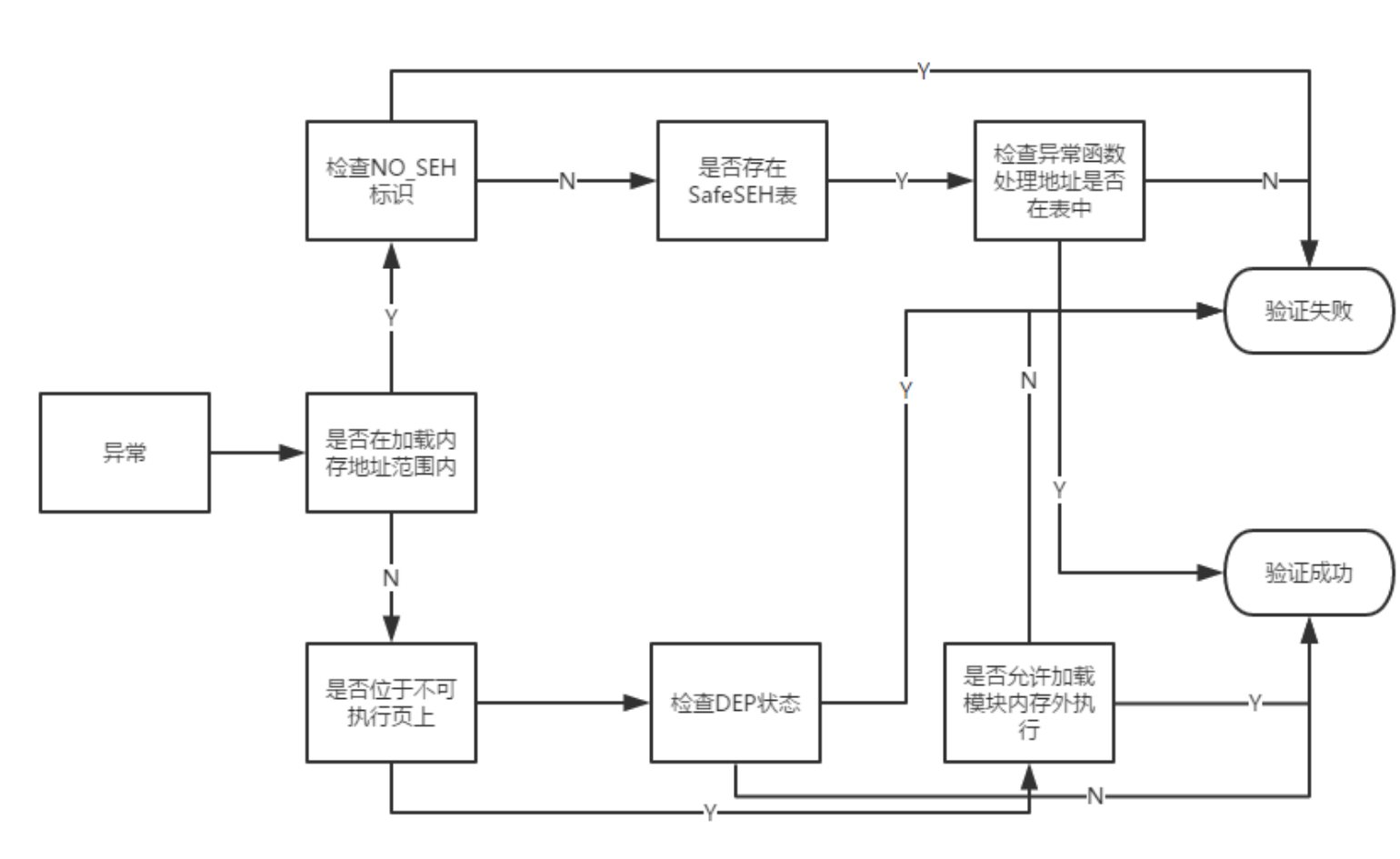
Build Debug Find in Files 1 Find in Files 2 Results SQL Debugging

Ln 31, Col 14 REC COL OVR READ

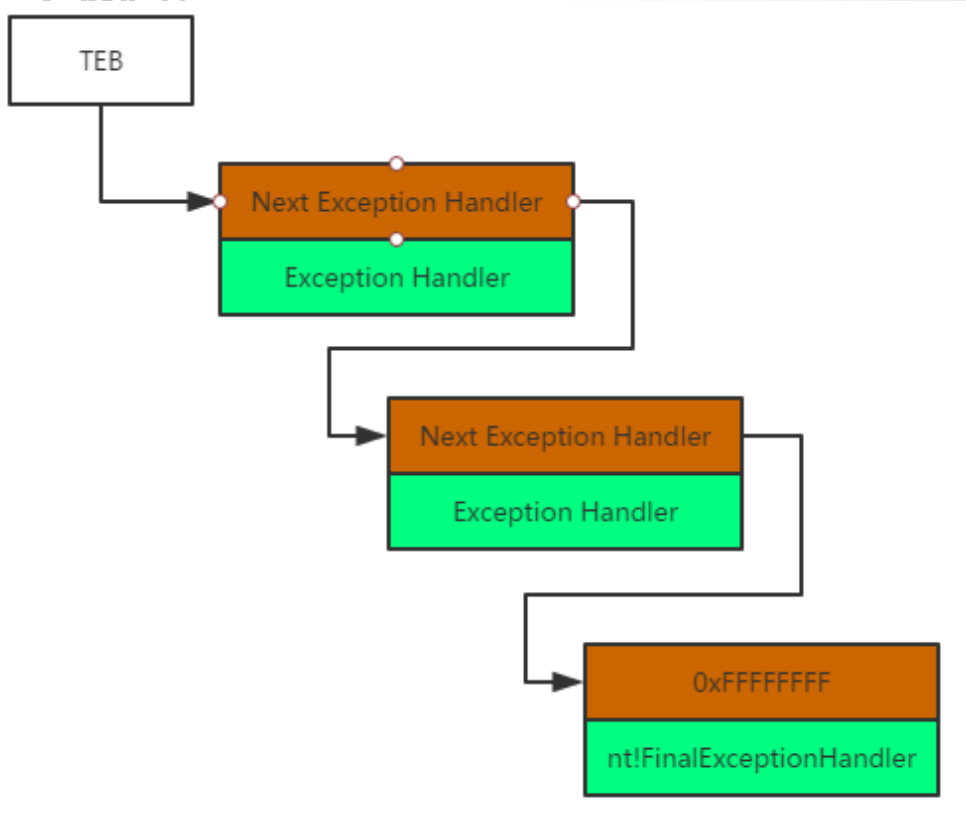
开始 seh - Microsoft ... 吾爱破解专用版01... 11:48

实验

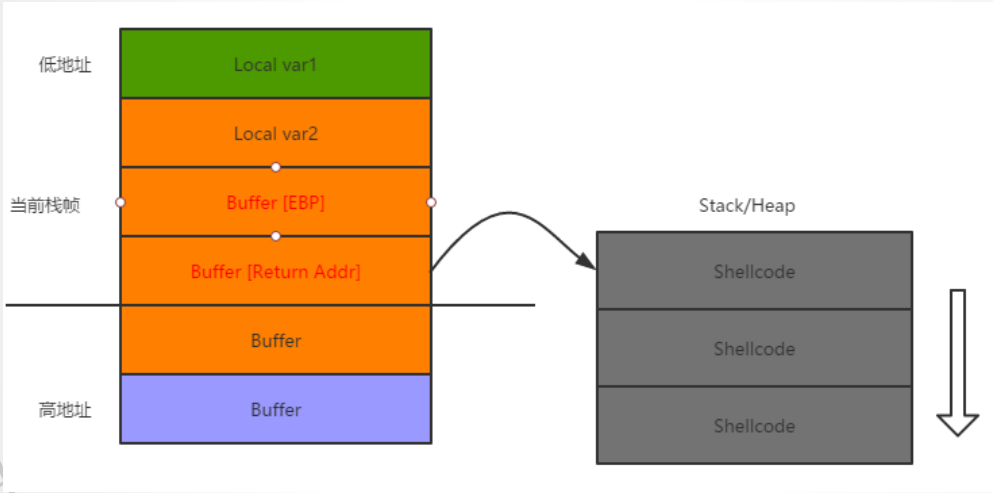
- nt!RtlIsValidHandler

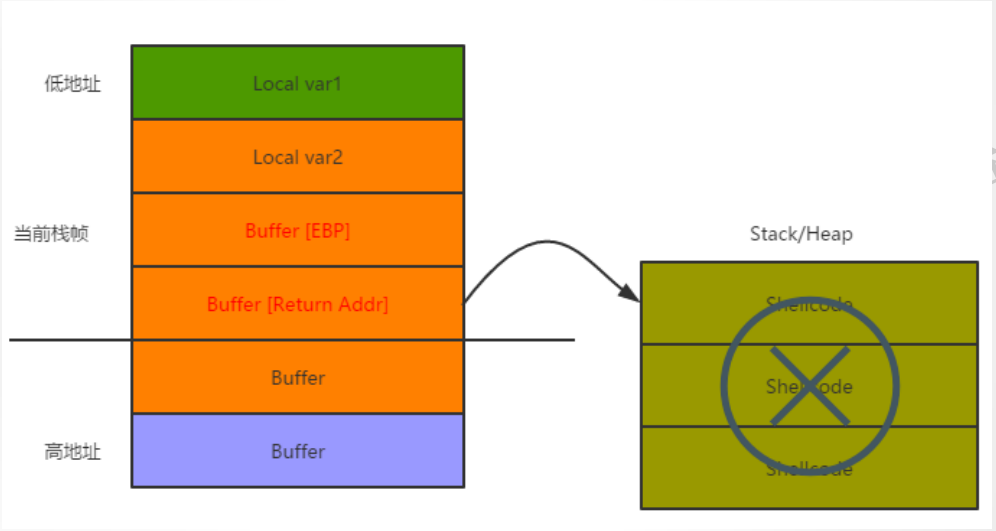


- SHE Overwrite Protect
- 原理
  - 遍历SEH链表
  - 检测链尾是否是默认异常处理函数

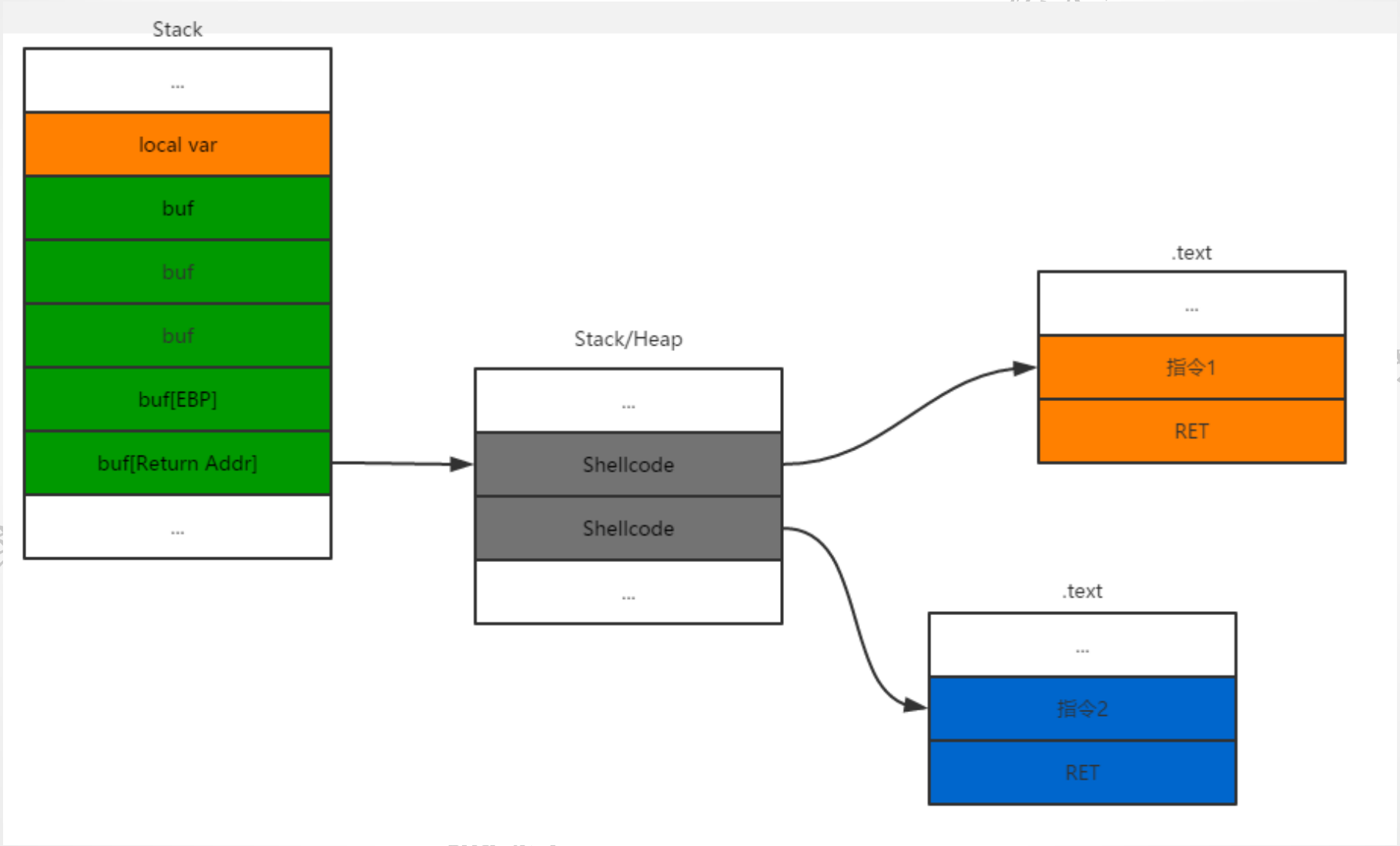


- DEP
  - Data Execution Protection
- NX
  - No eXecute
- Bypass
  - ROP(Return Orientated Programming)
  - Ret2libc(Return to libc)
  - Return to dl resolve





- ROP/Ret2libc
  - 返回导向编程
  - 绕过DEP/NX 数据执行保护的技术
- 原理
  - 将exploit的等效指令组装成一个执行链，将执行链中每一块的地址部署在栈上(栈上不存放指令，只存放指令地址)，使得执行结果等效于shellcode


Stack
...
local var
buf
buf
buf
buf[EBP]
buf[Return Addr]
...
Stack/Heap
...
Shellcode
Shellcode
...
.text
...
指令1
RET
.text
...
指令2
RET

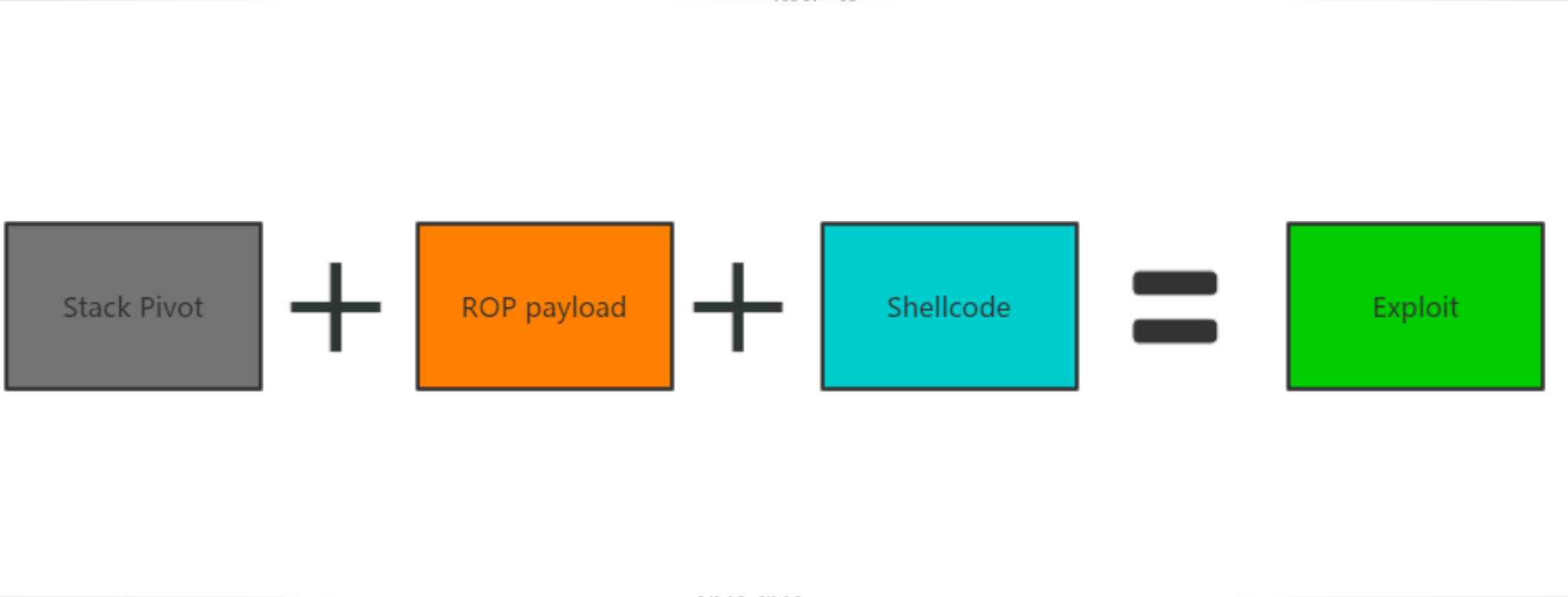
WINDOWS

- 伪造函数调用栈
- 调用windows API
- 执行ROP chain修改shellcode地址执行权限
- 跳转到shellcode的地址并执行

LINUX

- 伪造函数调用栈
- 调用write泄漏libc中system地址
- 执行system(/bin/sh)
- 返回交互式shell

```
BOOL VirtualProtect(
        LPVOID lpAddress,       // 目标地址起始位置
        DWORD dwSize,           // 大小
        DWORD flNewProtect,     // 请求的保护方式
        PDWORD lpflOldProtect // 保存老的保护方式
);
flNewProtect = 0x40  // Executable
```



如何构造ROP CHAIN

- Stack Pivot(栈反转)
  - 无法操作ESP(当前栈)
- 寻找ROP Gadgets
  - 搜索程序中的指令
- 根据ROP Gadgets拼接ROP payload
  - 从所有的ROP Gadgets中过滤需要的
  - 拼接出ROP payload
- 编写shellcode(可选)
  - 用metasploit生成
  - Exploit-DB上下载

- 以RET结尾的指令序列
  - POP eax; RET
  - POP ebx; POP eax; RET
- 伪造栈帧
  - X86
    - PUSHAD
    - 入栈顺序：EAX, ECX, EDX, EBX, ESI, EDI
    - ESP - 32
  - X86_64
    - 直接放在寄存器

RETN

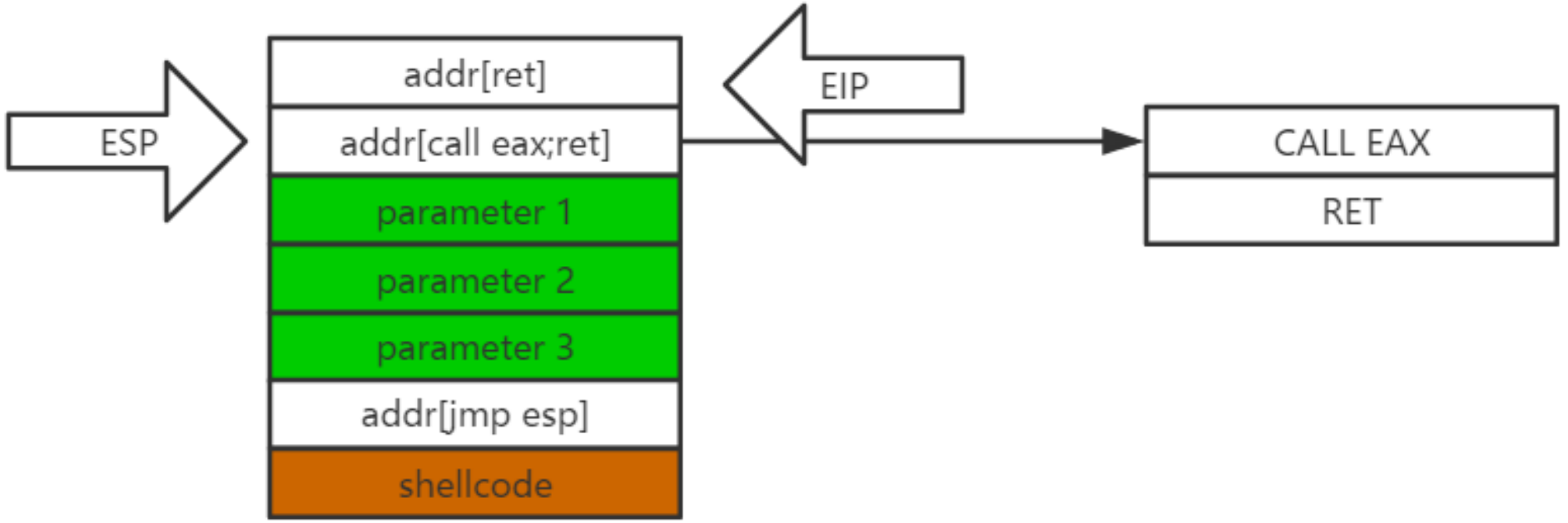
POP EIP

CALL

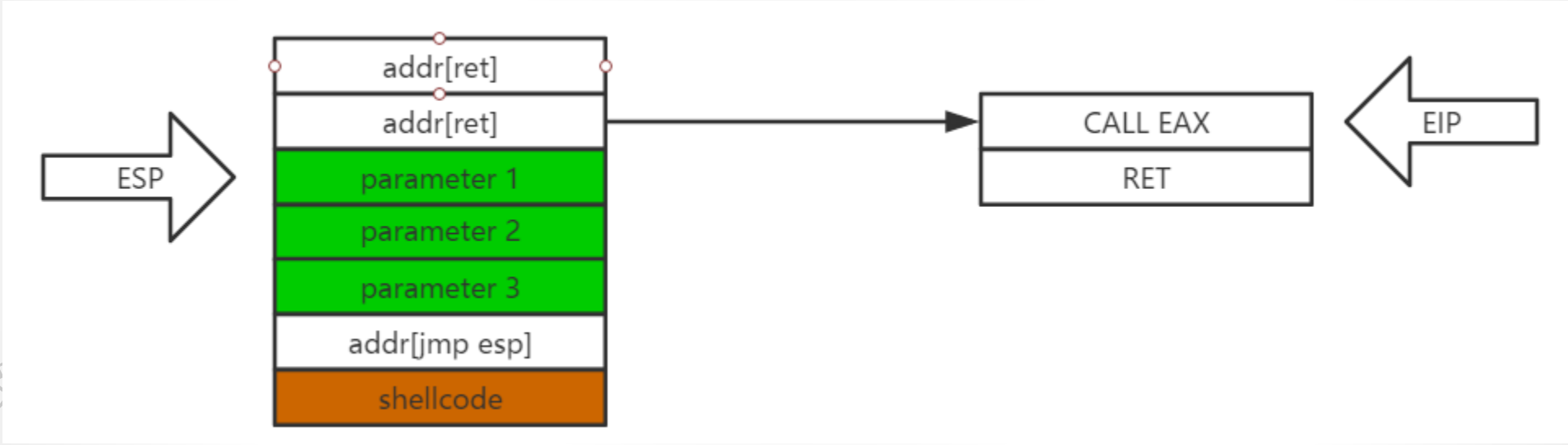
PUSH EIP

JMP

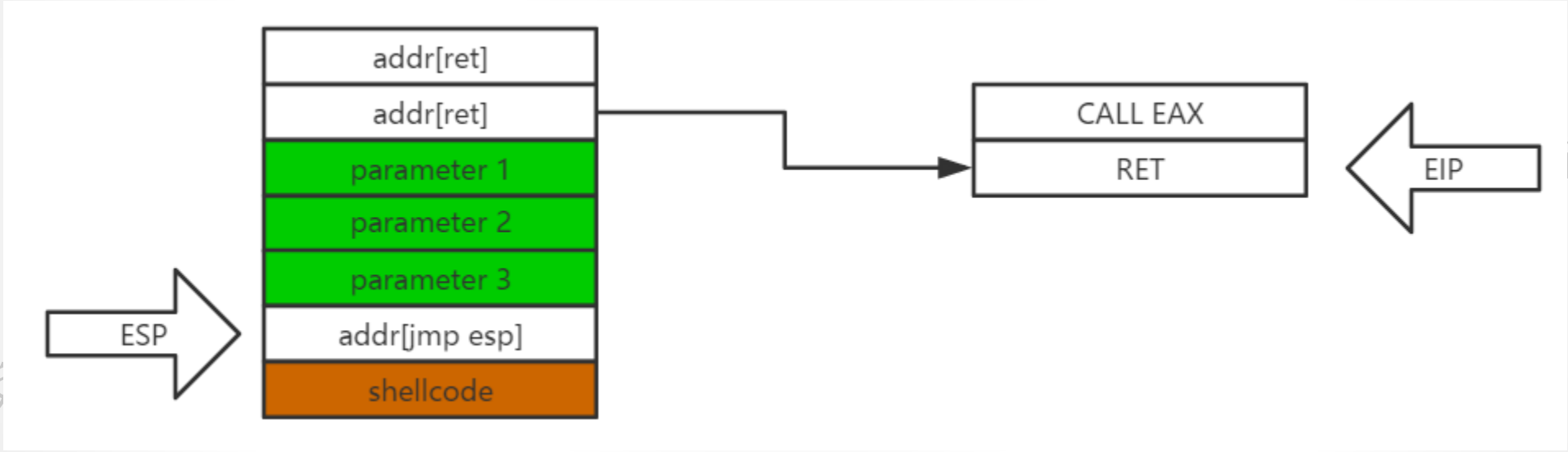
ROP CHAIN X86(EXAMPLE1)



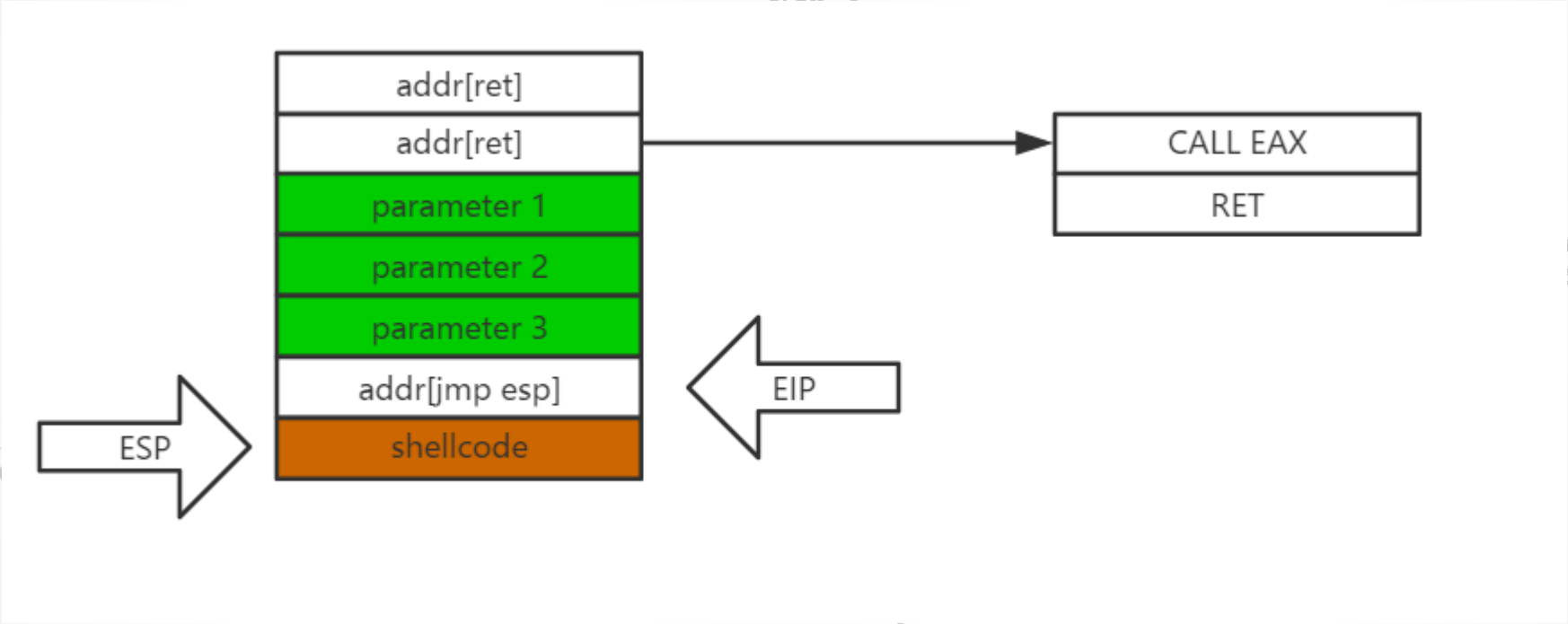
ROP CHAIN X86(EXAMPLE1)



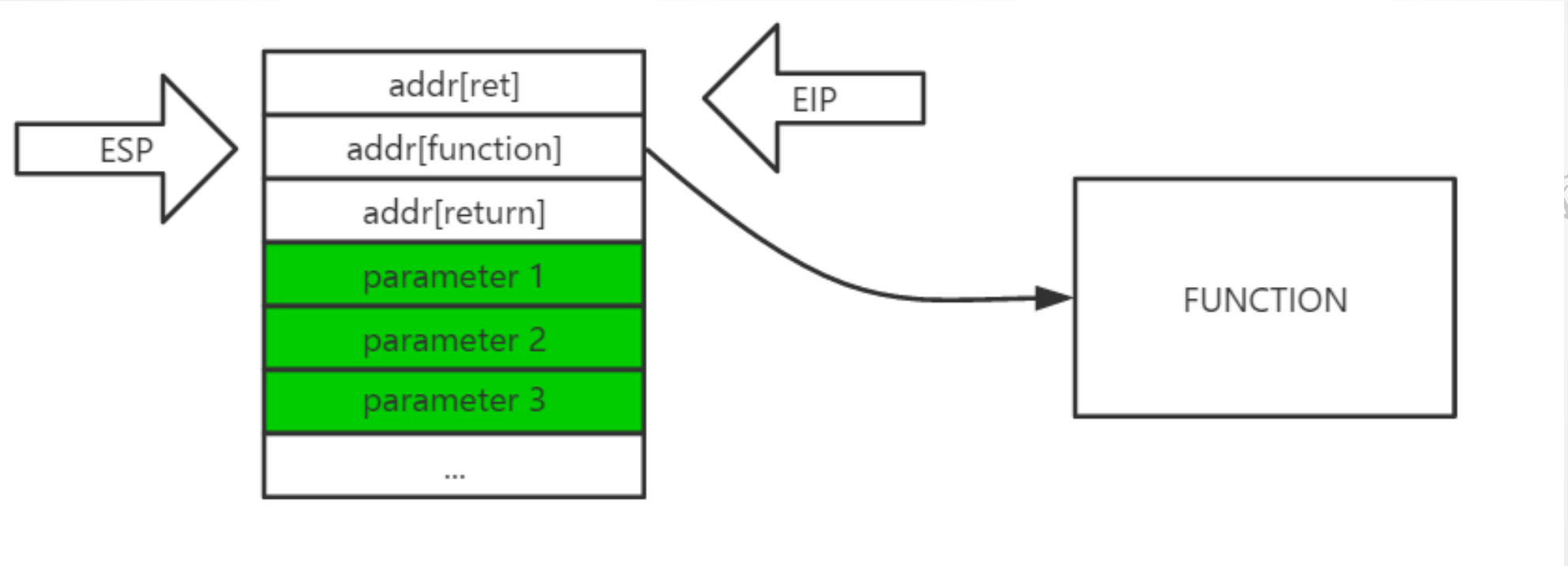
ROP CHAIN X86(EXAMPLE1)



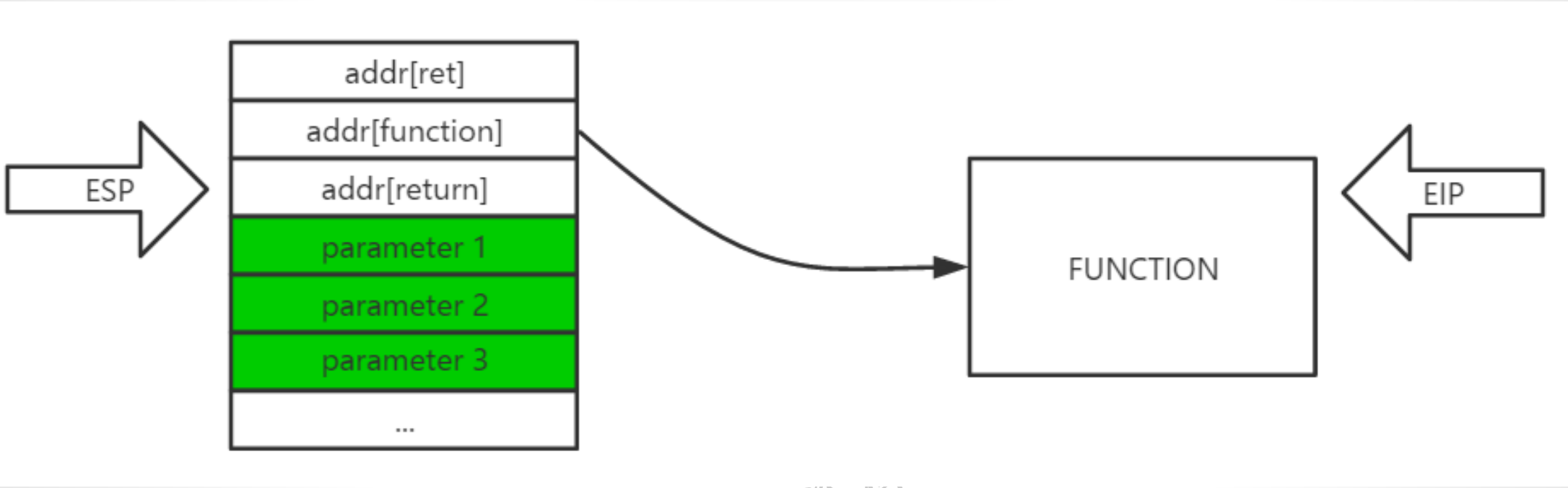
ROP CHAIN X86(EXAMPLE1)



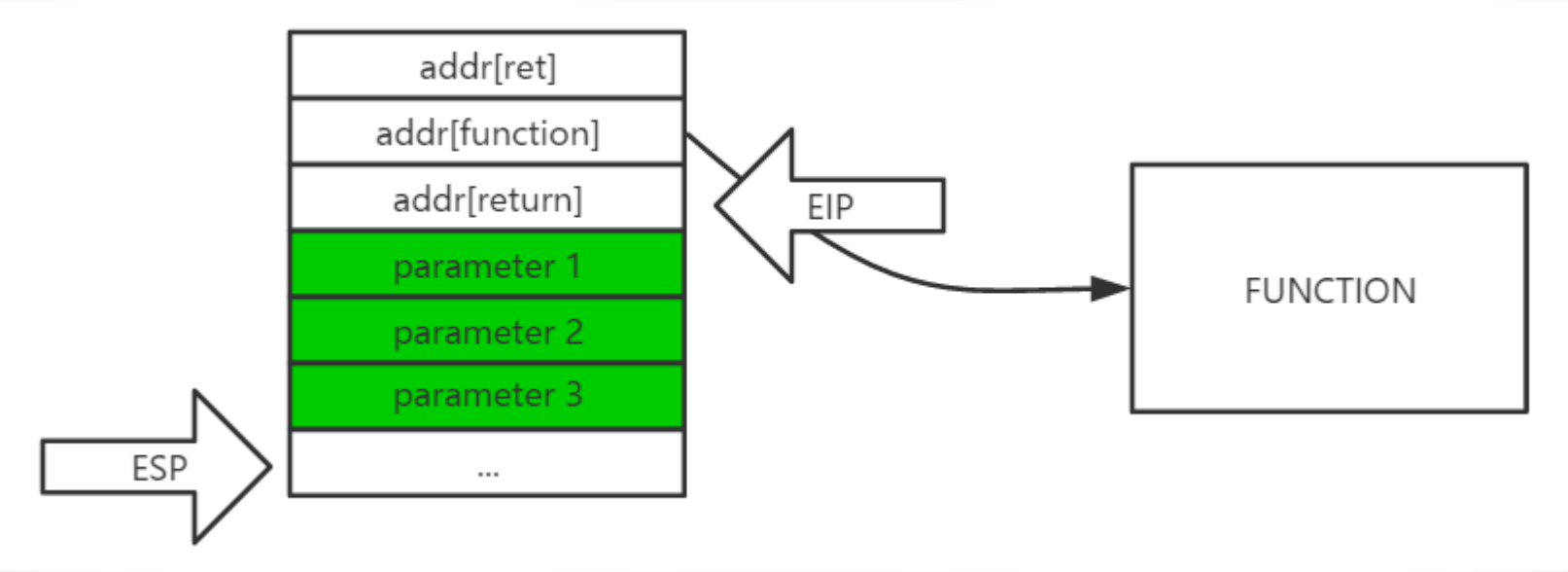
ROP CHAIN X86(EXAMPLE2)

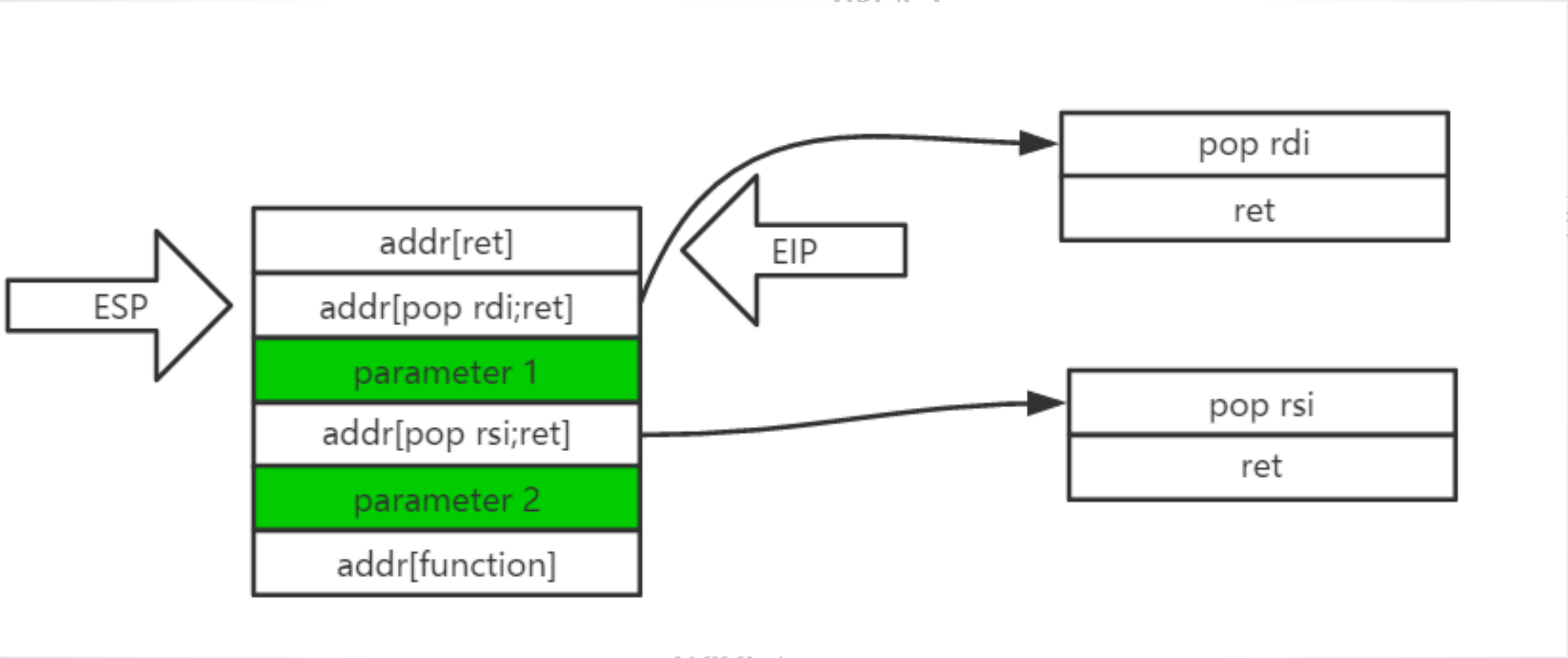


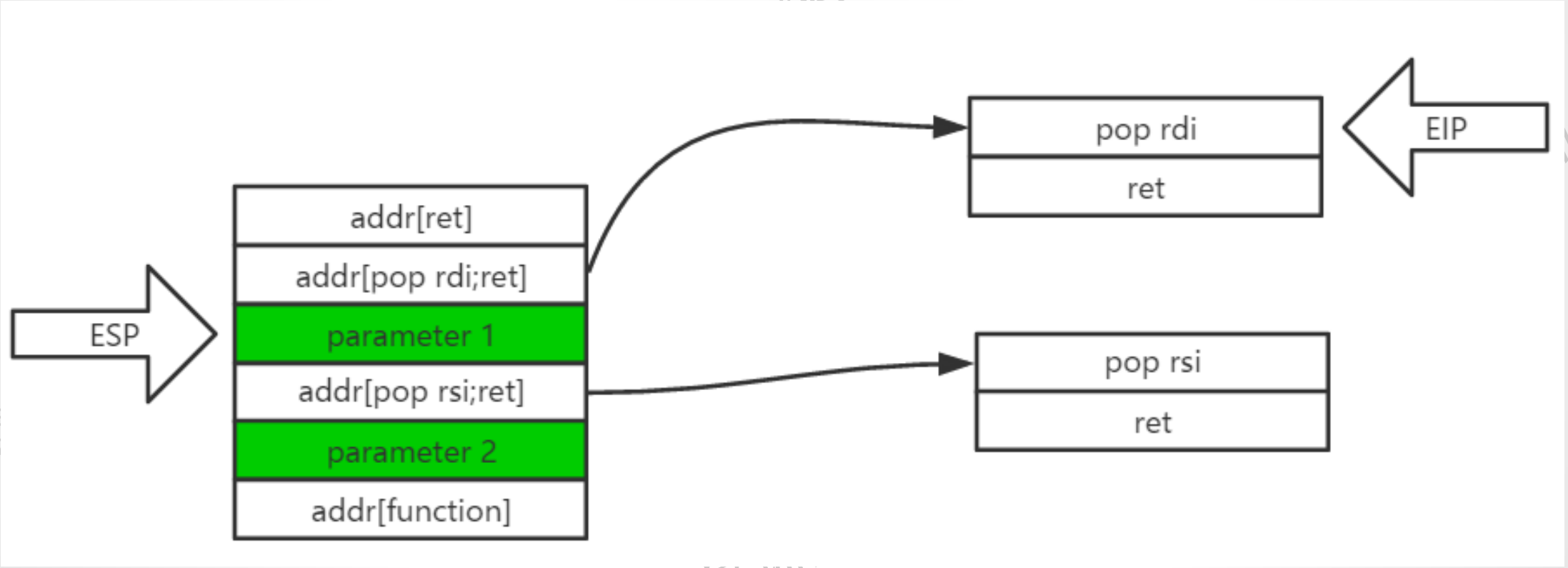
ROP CHAIN X86(EXAMPLE2)

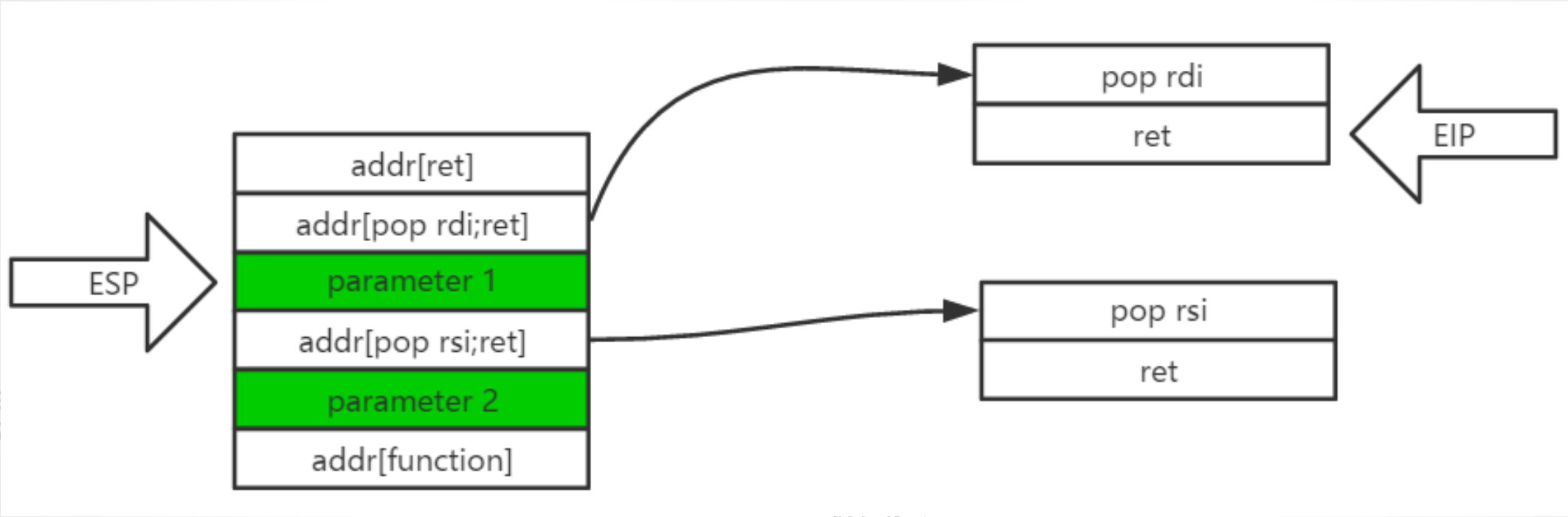


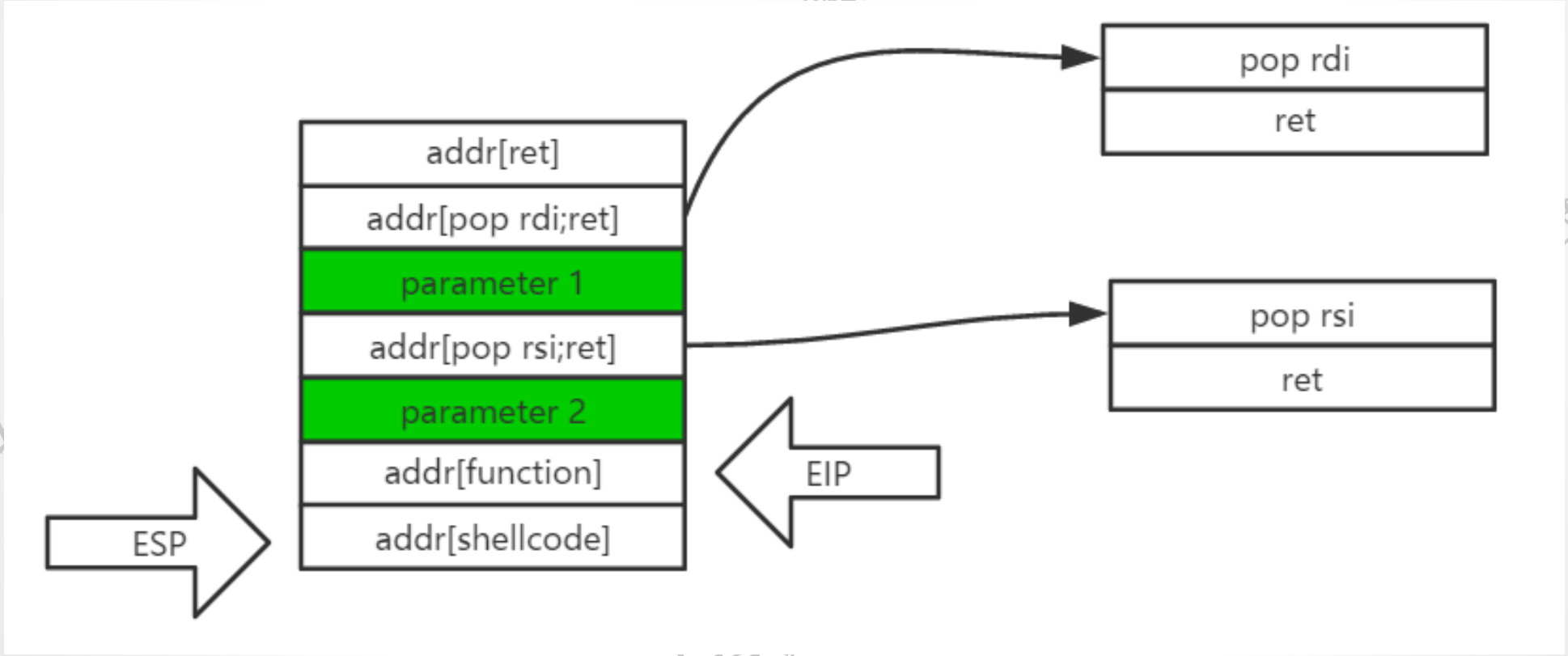
ROP CHAIN X86(EXAMPLE2)




ESP
addr[ret]
addr[pop rdi;ret]
parameter 1
addr[pop rsi;ret]
parameter 2
addr[function]
EIP
pop rdi
ret
pop rsi
ret

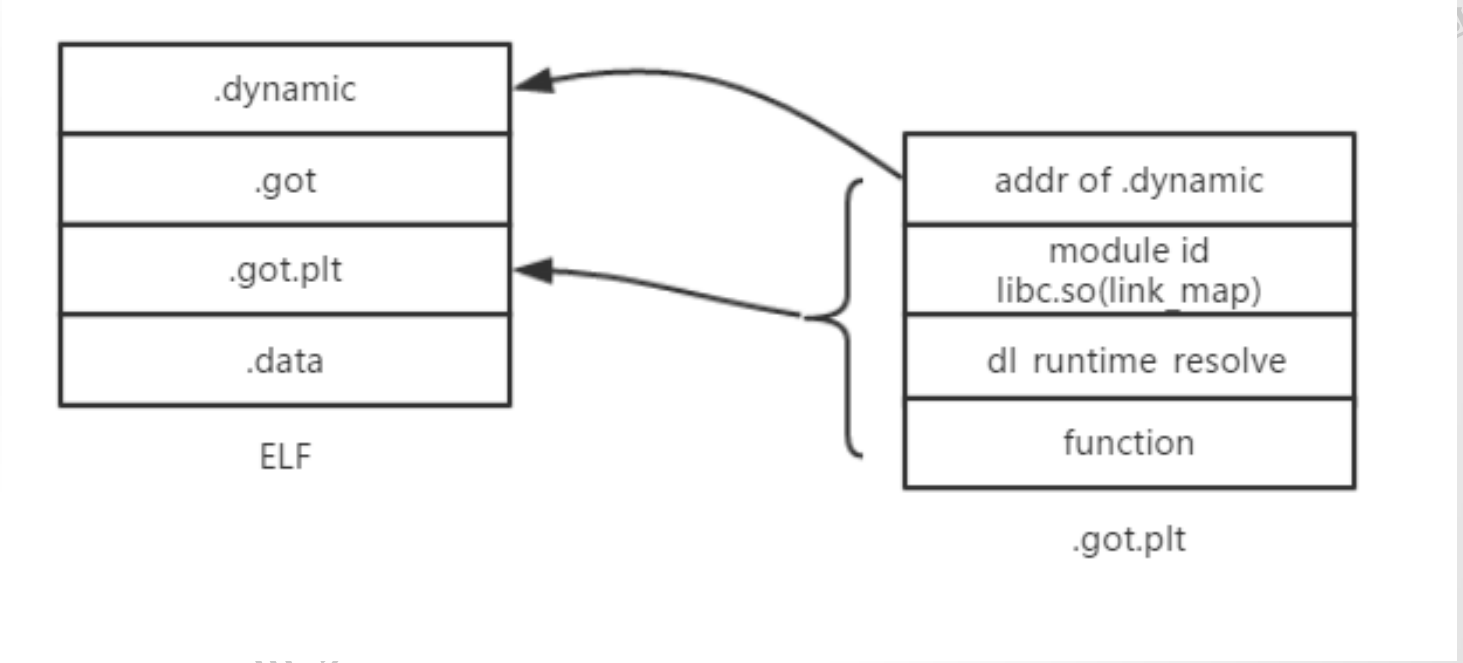

pop rdi
ret
EIP
addr[ret]
addr[pop rdi;ret]
parameter 1
addr[pop rsi;ret]
parameter 2
addr[function]
ESP
pop rsi
ret


pop rdi
ret
EIP
addr[ret]
addr[pop rdi;ret]
ESP
parameter 1
addr[pop rsi;ret]
parameter 2
addr[function]
pop rsi
ret

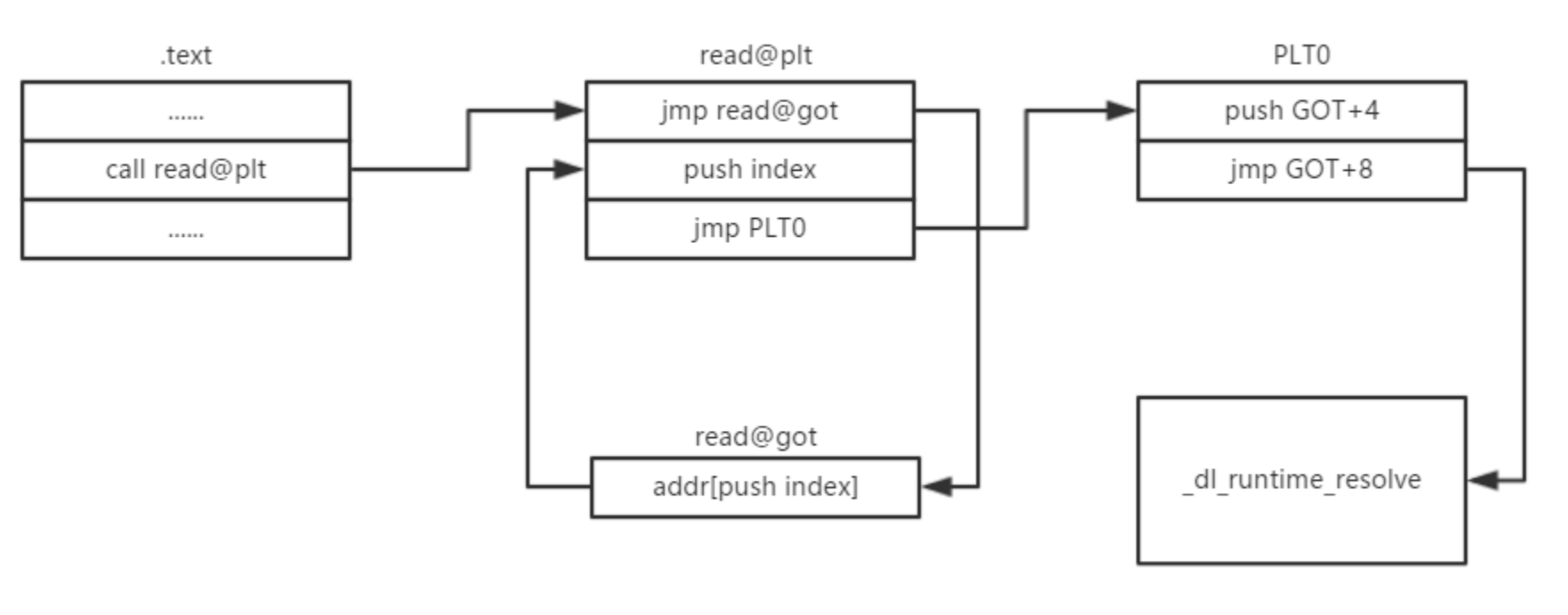

pop rdi
ret
addr[ret]
addr[pop rdi;ret]
parameter 1
addr[pop rsi;ret]
pop rsi
ret
parameter 2
addr[function]
EIP
ESP
addr[shellcode]

实验

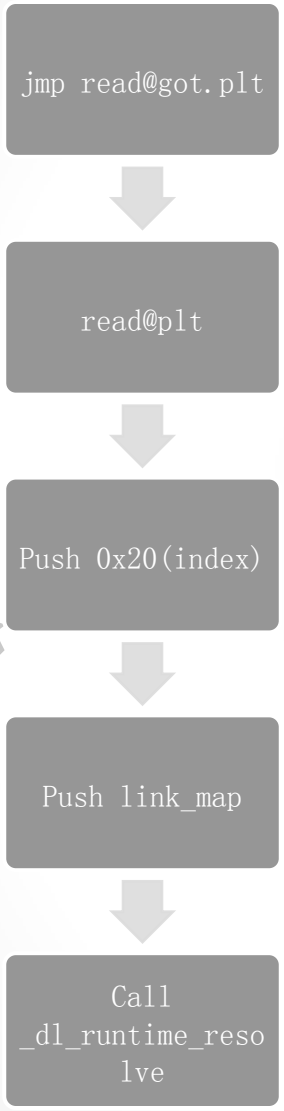
- Link_map
  - 加载的库文件列表
- _dl_runtime_resolve
  - 解析符号地址的函数



再看LAZY BINDING



RETURN TO DL RESOLVE




```
[-----------------------------------code-----------------------------------]
   0x80483d0 <__libc_start_main@plt>:   jmp    DWORD PTR ds:0x804a00c
   0x80483d6 <__libc_start_main@plt+6>: push   0x18
   0x80483db <__libc_start_main@plt+11>:        jmp    0x8048390
=> 0x80483e0 <read@plt>:        jmp    DWORD PTR ds:0x804a010
 | 0x80483e6 <read@plt+6>:      push   0x20
 | 0x80483eb <read@plt+11>:     jmp    0x8048390
 | 0x80483f0 <strlen@plt>:      jmp    DWORD PTR ds:0x804a014
 | 0x80483f6 <strlen@plt+6>:    push   0x28
 |->   0x80483e6 <read@plt+6>:  push   0x20
       0x80483eb <read@plt+11>: jmp    0x8048390
       0x80483f0 <strlen@plt>:  jmp    DWORD PTR ds:0x804a014
       0x80483f6 <strlen@plt+6>:        push   0x28
                                                        JUMP is taken
[-----------------------------------stack-----------------------------------]
[-----------------------------------code-----------------------------------]
   0x80483db <__libc_start_main@plt+11>:        jmp    0x8048390
   0x80483e0 <read@plt>:        jmp    DWORD PTR ds:0x804a010
   0x80483e6 <read@plt+6>:      push   0x20
=> 0x80483eb <read@plt+11>:     jmp    0x8048390
 | 0x80483f0 <strlen@plt>:      jmp    DWORD PTR ds:0x804a014
 | 0x80483f6 <strlen@plt+6>:    push   0x28
 | 0x80483fb <strlen@plt+11>:   jmp    0x8048390
 | 0x8048400 <_start>:  xor    ebp,ebp
 |->   0x8048390:       push   DWORD PTR ds:0x8049ff8
       0x8048396:       jmp    DWORD PTR ds:0x8049ffc
       0x804839c:       add    BYTE PTR [eax],al
       0x804839e:       add    BYTE PTR [eax],al
                                                        JUMP is taken
[-----------------------------------stack-----------------------------------]
```

再看LAZY BINDING

```
[----------------------------------code-----------------------------------]
   0x8048490 <main+35>: lea    eax,[esp+0x1c]
   0x8048494 <main+39>: mov    DWORD PTR [esp+0x4],eax
   0x8048498 <main+43>: mov    DWORD PTR [esp],0x1
=> 0x804849f <main+50>: call   0x8048330 <read@plt>
   0x80484a4 <main+55>: mov    DWORD PTR [esp+0x8],0x80
   0x80484ac <main+63>: lea    eax,[esp+0x1c]
   0x80484b0 <main+67>: mov    DWORD PTR [esp+0x4],eax
   0x80484b4 <main+71>: mov    DWORD PTR [esp],0x1
Guessed arguments:
arg[0]: 0x1
arg[1]: 0xffffd36c --> 0x0
arg[2]: 0x80
[-------------------------------------stack-------------------------------------]
```

再看LAZY BINDING

```
[-------------------------------------code-------------------------------------]
   0x8048326:    jmp    DWORD PTR ds:0x804a008
   0x804832c:    add    BYTE PTR [eax],al
   0x804832e:    add    BYTE PTR [eax],al
=> 0x8048330 <read@plt>:         jmp    DWORD PTR ds:0x804a00c
 | 0x8048336 <read@plt+6>:       push   0x0
 | 0x804833b <read@plt+11>:      jmp    0x8048320
 | 0x8048340 <__stack_chk_fail@plt>:     jmp    DWORD PTR ds:0x804a010
 | 0x8048346 <__stack_chk_fail@plt+6>:   push   0x8
 |->    0x8048336 <read@plt+6>:   push   0x0
        0x804833b <read@plt+11>: jmp    0x8048320
        0x8048340 <__stack_chk_fail@plt>:         jmp    DWORD PTR ds:0x804a010
        0x8048346 <__stack_chk_fail@plt+6>:       push   0x8
                                                            JUMP is taken
[-------------------------------------stack------------------------------------]
```

再看LAZY BINDING

```
gdb-peda$ x/wx 0x804a00c
0x804a00c <read@got.plt>:       0x08048336
gdb-peda$ x/2i 0x08048336
   0x8048336 <read@plt+6>:      push   0x0
   0x804833b <read@plt+11>:     jmp    0x8048320
gdb-peda$ x/3i 0x8048320
   0x8048320:   push   DWORD PTR ds:0x804a004
   0x8048326:   jmp    DWORD PTR ds:0x804a008
   0x804832c:   add    BYTE PTR [eax],al
gdb-peda$
```

再看LAZY BINDING

再看LAZY BINDING

```
gdb-peda$ x/wx 0x804a00c
0x804a00c <read@got.plt>:       0xf7ef1880
gdb-peda$ x/3i 0xf7ef1880
   0xf7ef1880 <read>:   cmp    DWORD PTR gs:0xc,0x0
   0xf7ef1888 <read+8>: jne    0xf7ef18ac <read+44>
   0xf7ef188a <read+10>:        push   ebx
gdb-peda$
```

- JMPREL
  - .rel.plt
- STRTAB
  - .dynstr
- SYMTAB
  - .got.plt

```
[redacted] ~/roputils/examples $ readelf -d bof32

Dynamic section at offset 0xf20 contains 21 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x0000000c (INIT)                       0x8048378
 0x0000000d (FINI)                       0x804862c
 0x00000004 (HASH)                       0x8048188
 0x6ffffef5 (GNU_HASH)                   0x80481c4
 0x00000005 (STRTAB)                     0x8048290
 0x00000006 (SYMTAB)                     0x80481f0
 0x0000000a (STRSZ)                      107 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000015 (DEBUG)                      0x0
 0x00000003 (PLTGOT)                     0x8049ff4
 0x00000002 (PLTRELSZ)                   48 (bytes)
 0x00000014 (PLTREL)                     REL
 0x00000017 (JMPREL)                     0x8048348
 0x00000011 (REL)                        0x8048330
 0x00000012 (RELSZ)                      24 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x6ffffffe (VERNEED)                    0x8048310
 0x6fffffff (VERNEEDNUM)                 1
 0x6ffffff0 (VERSYM)                     0x80482fc
 0x00000000 (NULL)                       0x0
```

- JMPREL Segment
  - 每个条目8bytes
  - 高4bytes为虚拟地址
  - 低4bytes为类型和序号

```
gdb-peda$ x/2x 0x8048348
0x8048348:      0x0804a000      0x00000107
gdb-peda$ x/2x 0x8048348 + 8
0x8048350:      0x0804a004      0x00000207
gdb-peda$ x/2x 0x8048348 + 8 * 2
0x8048358:      0x0804a008      0x00000307
gdb-peda$ x/2x 0x8048348 + 8 * 3
0x8048360:      0x0804a00c      0x00000407
```

```
0804a040  00000705 R_386_COPY        0804a040   stdout

Relocation section '.rel.plt' at offset 0x348 contains 6 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
0804a000  00000107 R_386_JUMP_SLOT   00000000   setbuf
0804a004  00000207 R_386_JUMP_SLOT   00000000   __gmon_start__
0804a008  00000307 R_386_JUMP_SLOT   00000000   write
0804a00c  00000407 R_386_JUMP_SLOT   00000000   __libc_start_main
0804a010  00000507 R_386_JUMP_SLOT   00000000   read
0804a014  00000607 R_386_JUMP_SLOT   00000000   strlen
```

```
typedef uint32_t Elf32_Addr;
typedef uint32_t Elf32_Word;
typedef struct
{
  Elf32_Addr    r_offset;               /* Address */
  Elf32_Word    r_info;                 /* Relocation type and symbol index */
} Elf32_Rel;
#define ELF32_R_SYM(val)                ((val) >> 8)
#define ELF32_R_TYPE(val)               ((val) & 0xff)
```

- SYMTAB Segment
  - 每个条目16bytes
  - 前4bytes为字符串表的序号

- 例如Read
  - 0x507 >> 8 = 5

```
gdb-peda$ x/4x 0x80481f0 + 16 * 5
0x8048240:      0x00000036      0x00000000      0x00000000      0x00000012
gdb-peda$ x/s 0x8048290 + 0x36
0x80482c6:      "read"
```

```
typedef struct
{
  Elf32_Word    st_name;   /* Symbol name (string tbl index) */
  Elf32_Addr    st_value;  /* Symbol value */
  Elf32_Word    st_size;   /* Symbol size */
  unsigned char st_info;   /* Symbol type and binding */
  unsigned char st_other;  /* Symbol visibility under glibc>=2.2 */
  Elf32_Section st_shndx;  /* Section index */
} Elf32_Sym;
```

- 首先得到rel_entry的地址
  - Elf32_Rel * rel_entry = JMPREL + rel_offset;
- 根据rel_entry中的符号表条目编号，得到对应的符号信息
  - Elf32_Sym *sym_entry = SYMTAB[ELF32_R_SYM(rel_entry->r_info)];
- 再找到符号信息中的符号名称
  - char *sym_name = STRTAB + sym_entry->st_name;
- 由此名称，搜索动态库
- 找到地址后，填充至.got对应位置
  - _dl_fixup
- 最后调整栈，调用这一解析得到的函数

- 传入很大的rel_offset值，使得rel_entry落在能控制的内存里
- 伪造SYMTAB序号，使得sym_entry落在能控制的内存里
- 伪造STRTAB序号，使得name落在我们能控制的内存里
- 写入需要的函数名
- _dl_runtime_resolve得到这个函数的地址，如system

- 与x86的区别
  - JMPREL 长度不同
  - SYMTAB 格式长度不同
  - _dl_runtime_resolve的offset参数(变为index)
  - link_map + 0x1c8 = NULL(取消offset检查)

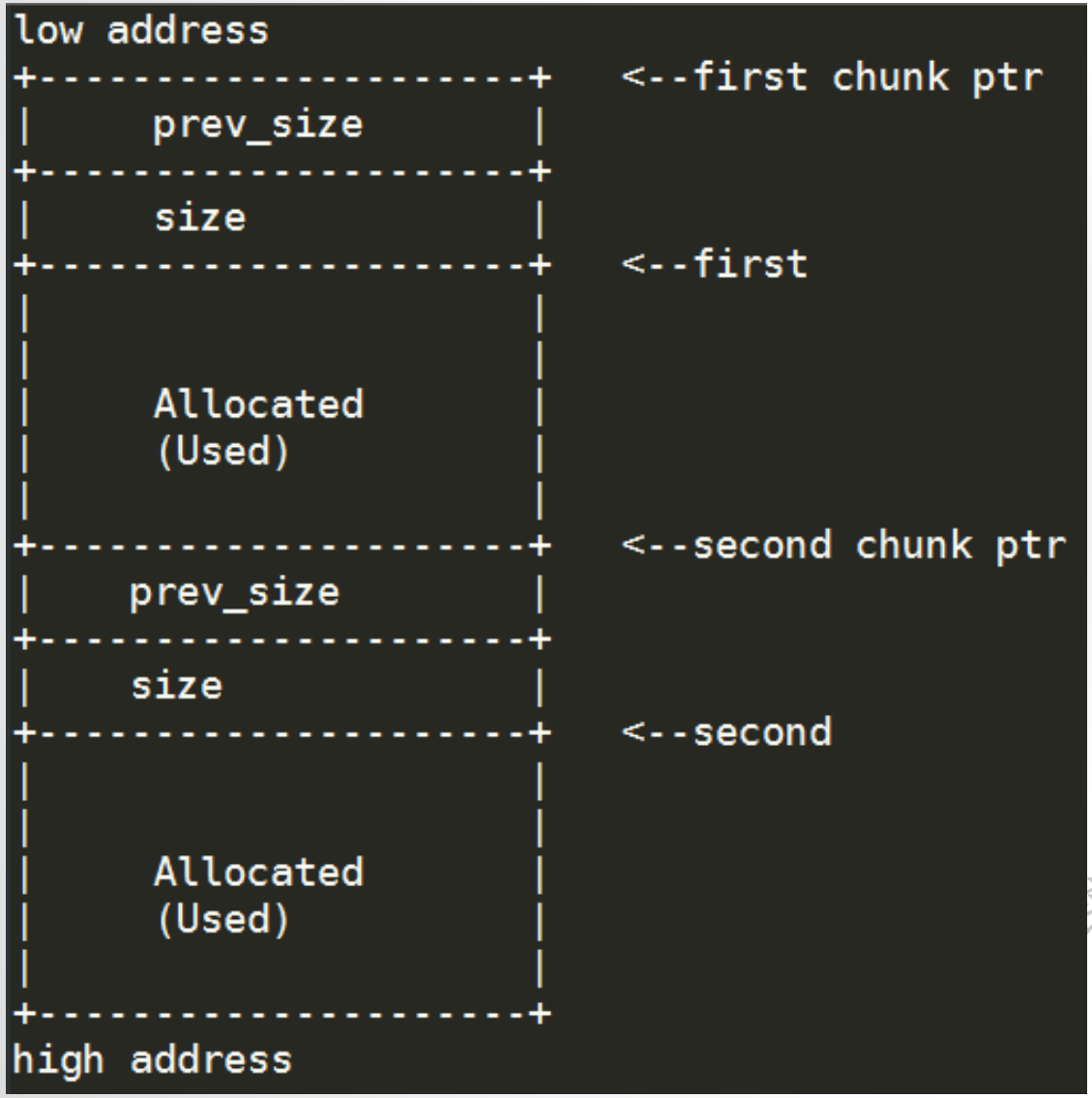
ROP/RET2LIBC

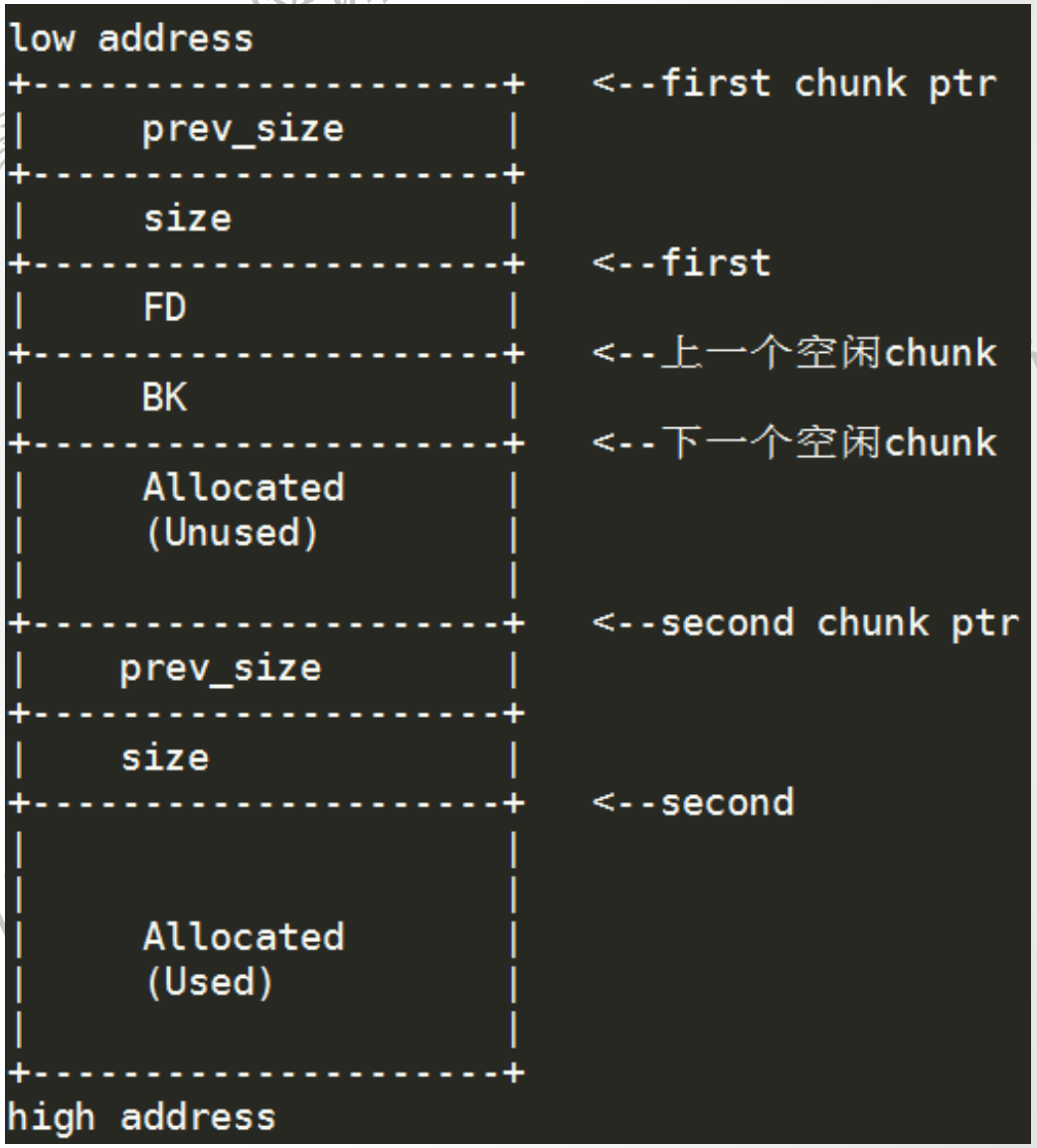
- 需要write泄漏addr
- 需要read写入数据
- 理解简单
- 构造exp复杂

RETURN TO DL RESOLVE

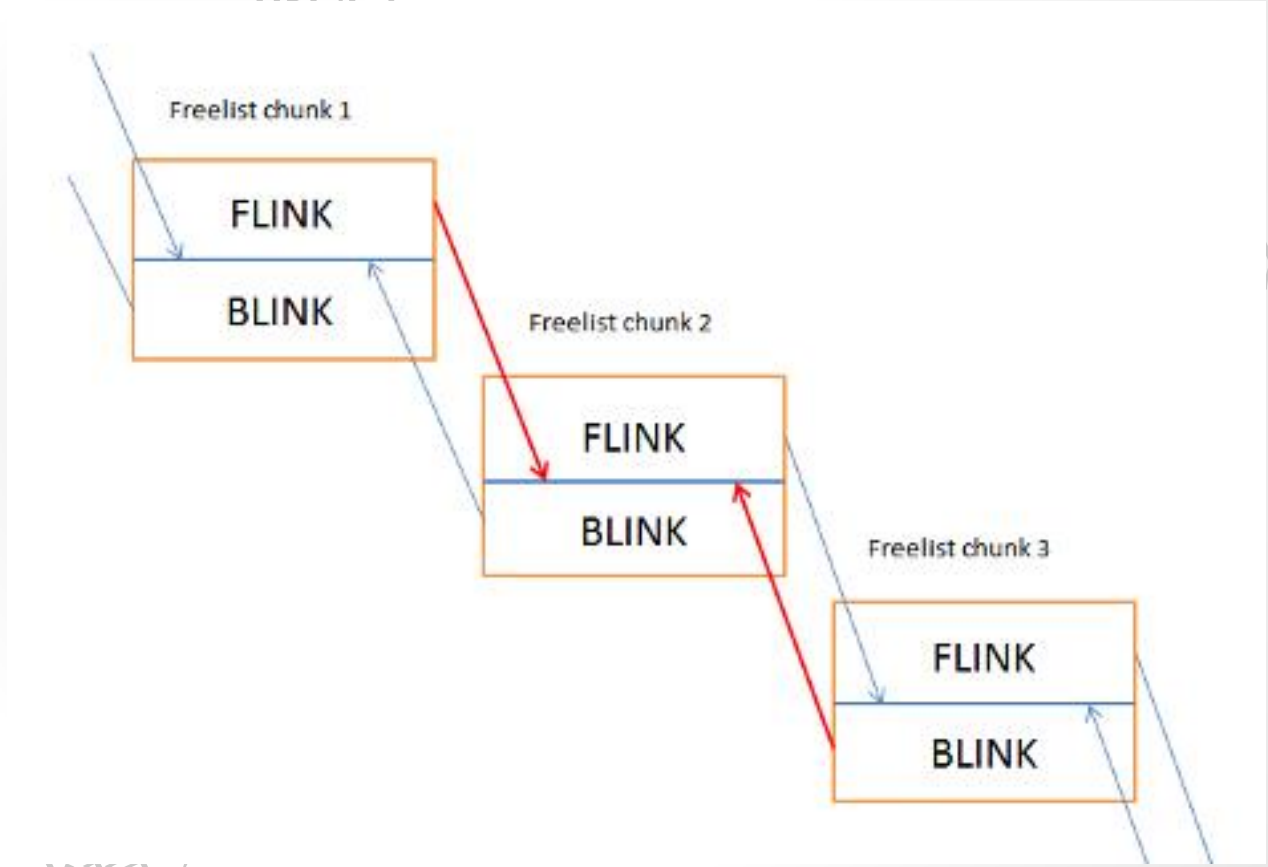
- 需要write泄漏addr
- 需要read写入数据
- 理解复杂
- 构造exp简单

- 利用堆块的溢出
- 重写下一个堆块的块首
- 而下一个堆块中可能包含其它的堆的入口地址或者已经释放的堆的节块（每个都包含前面所述的块首）。之后，当堆管理器操作堆块时（比如说申请一个新的堆块），使的操作的空间指向了重写的堆块，这个错位的块首是可以被头管理函数访问的，但这将产生一个异常,在合适的条件下可以被利用。

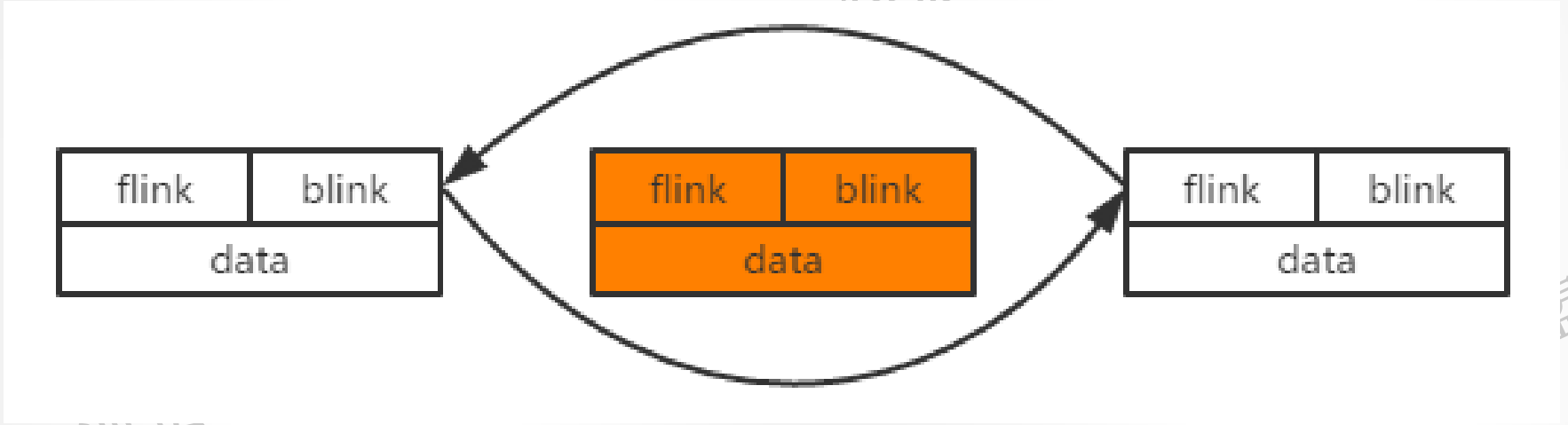

low address
<--first chunk ptr
prev_size
size
<--first
Allocated
(Used)
<--second chunk ptr
prev_size
size
<--second
Allocated
(Used)
high address

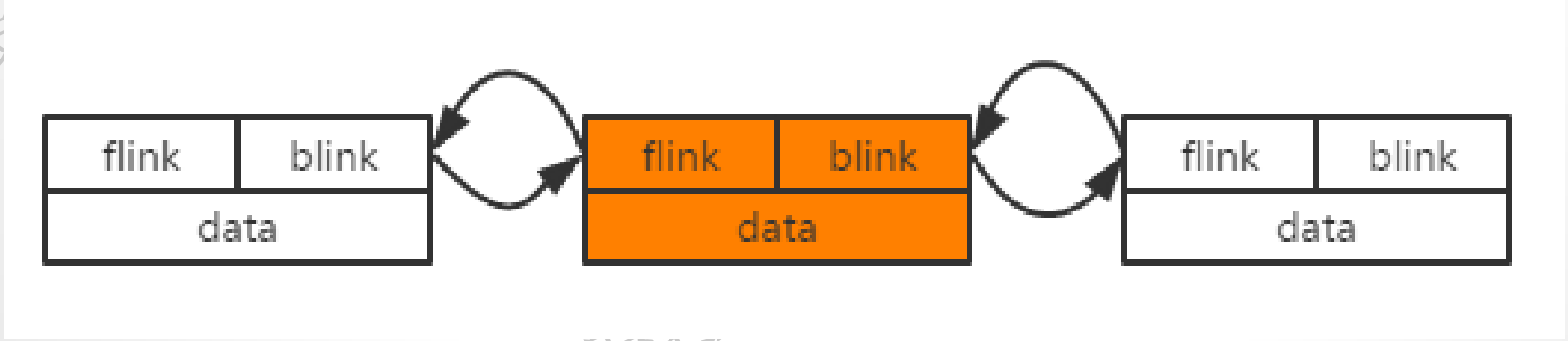

low address
<--first chunk ptr
prev_size
size
<--first
FD
<--上一个空闲chunk
BK
<--下一个空闲chunk
Allocated
(Unused)
<--second chunk ptr
prev_size
size
<--second
Allocated
(Used)
high address

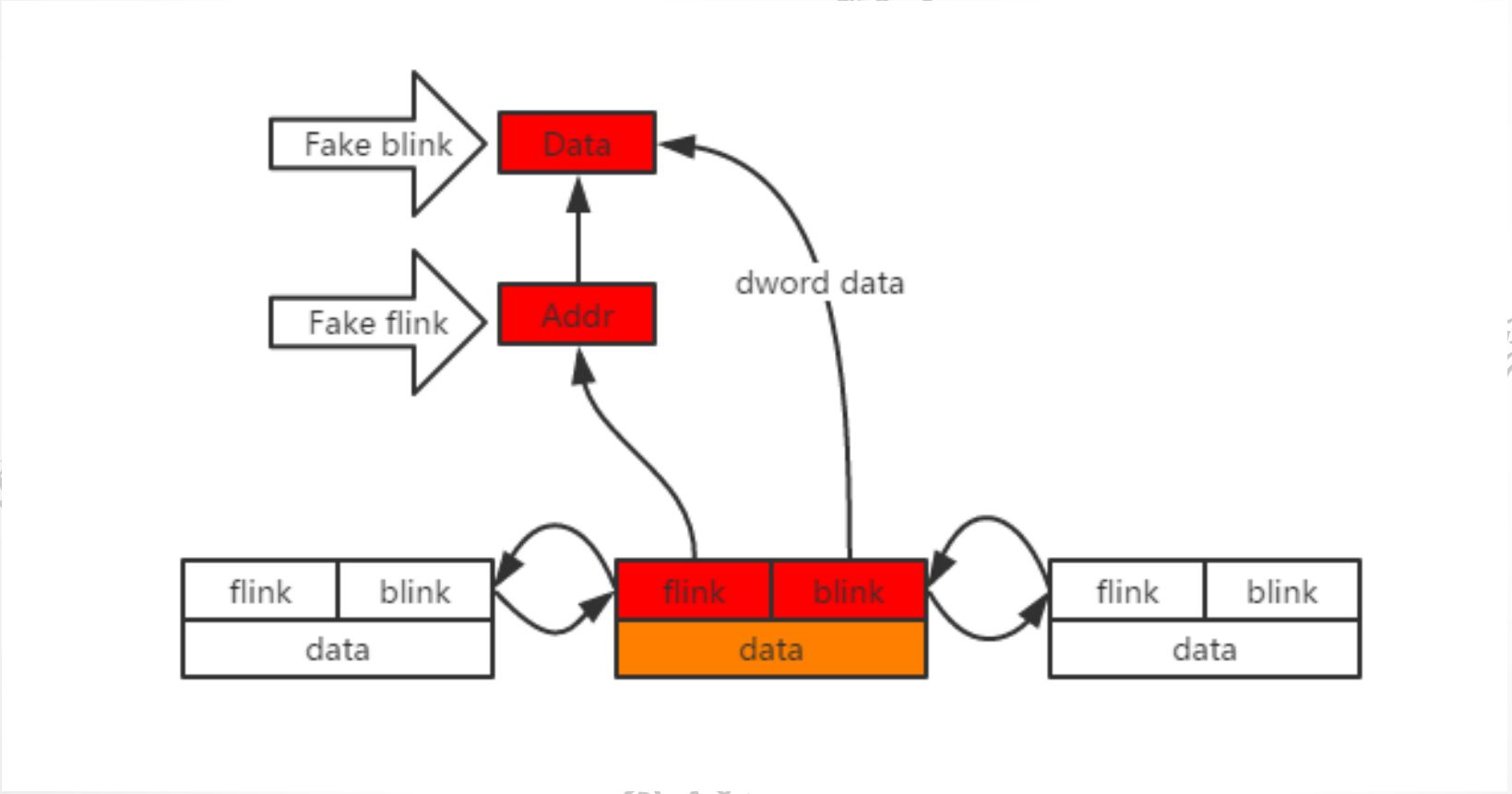
```
#define unlink(P, BK, FD)
{
        FD = P->fd;
        BK = P->bk;
        FD->bk = BK;
        BK->fd = FD;
}
```



堆块合并操作



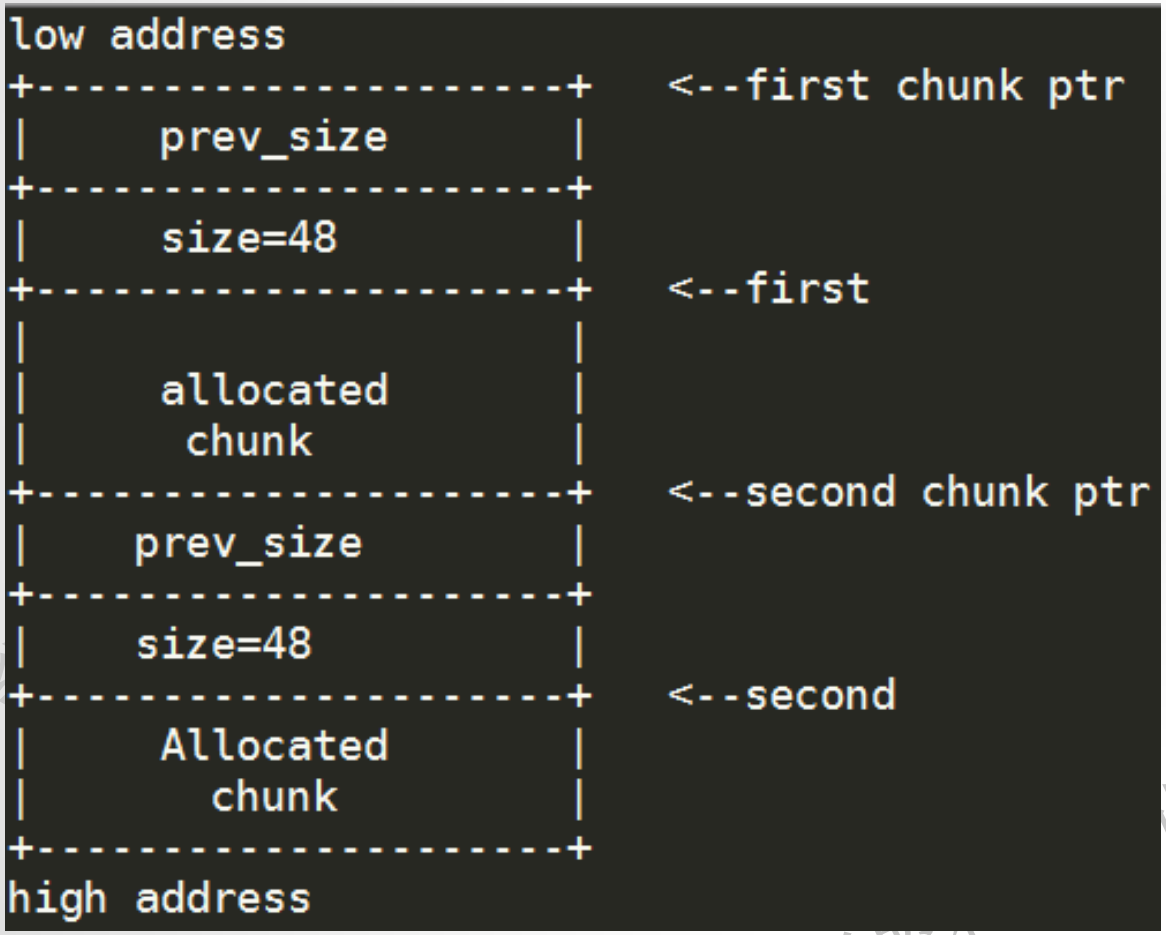


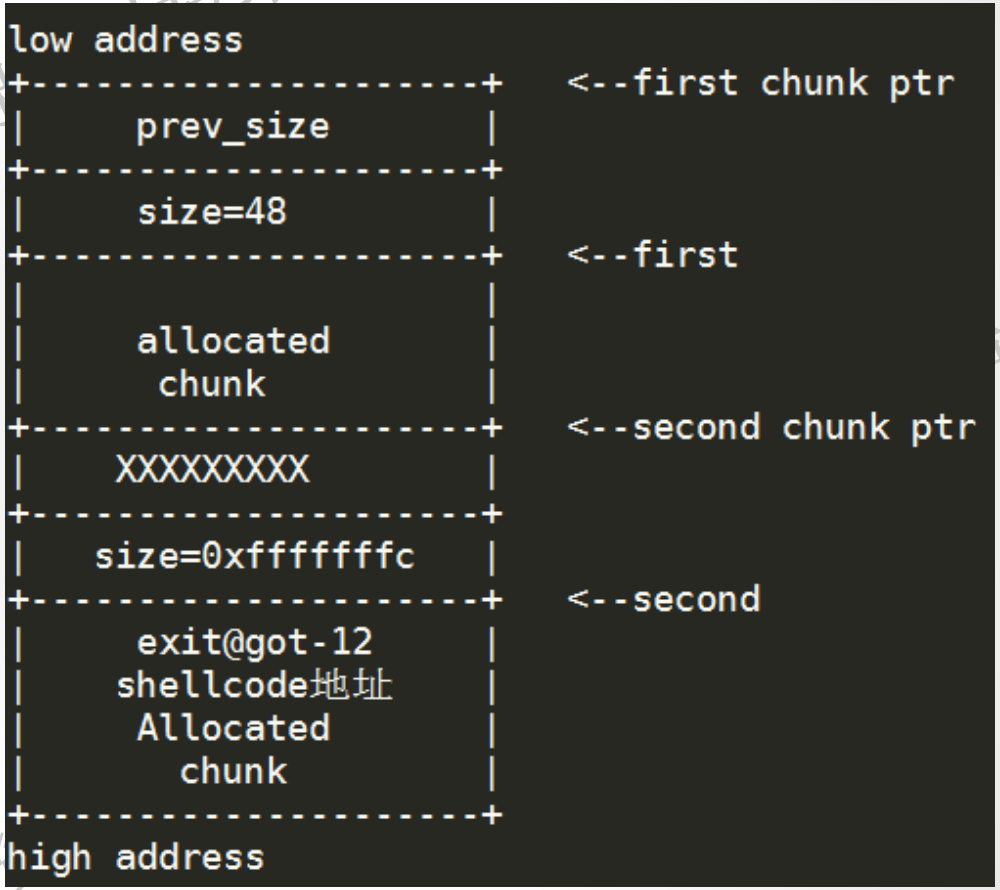

Fake blink
Data
Fake flink
Addr
dword data
flink
blink
data
flink
blink
data
flink
blink
data

HEAP OVERFLOW EXAMPLE

```
int main(void)
{
        char *buff1, *buff2;
        buff1 = malloc(40);
        buff2 = malloc(40);
        gets(buff1);
        free(buff1);
        exit(0);
}
```

HEAP OVERFLOW EXAMPLE





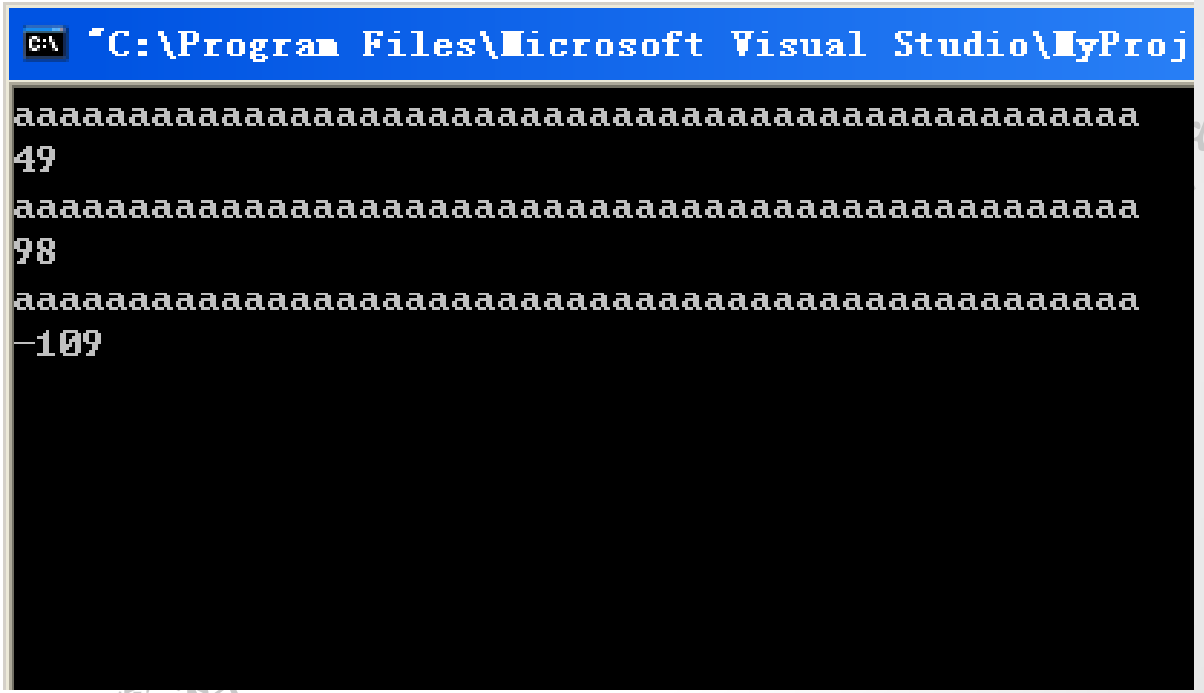
- Safe Unlink
  - 拆卸堆之前，检查链表是否完整
- Security cookie
  - 在链表前加一个cookie，判断是否被覆盖

- 有符号型和无符号型
- 有符号型(signed char)
  - 127 + 1 = -128
  - -128 -1 = 127
- 无符号型(unsigned char)
  - 255 + 1 = 0
  - 0 – 1 = 255

INTEGER OVERFLOW EXAMPLE

```
#include<stdio.h>
#include<string.h>
#define MAX 100

int main()
{
    int n = 0;
    char str[100];
    signed char a = 0;
    while((int)a < MAX)
    {
        scanf("%s", str);
        n = strlen(str);
        a += n;
        printf("%d\n", a);
    }
    return 0;
}
```



实验

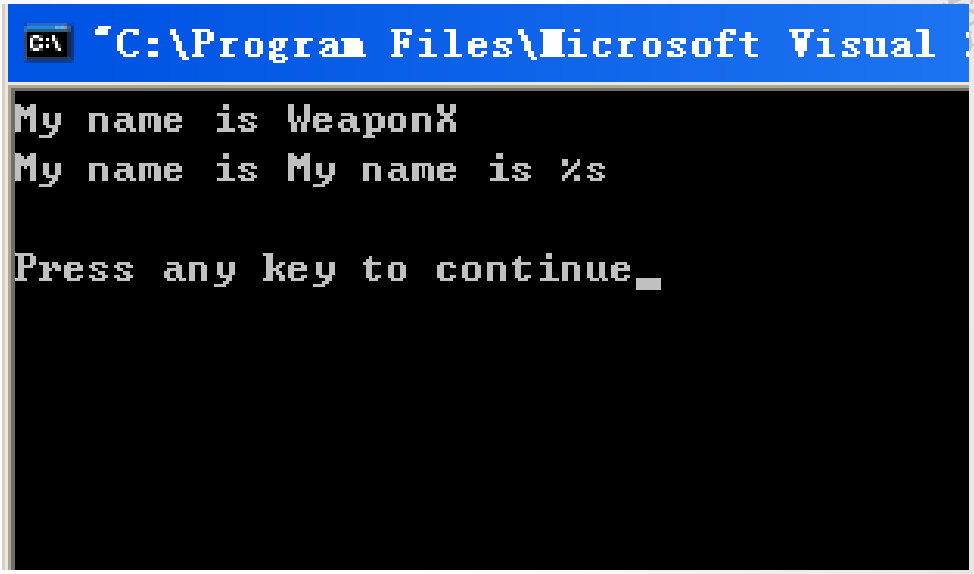
格式化字符串漏洞

- 格式化字符串
  - 包括文本和格式参数的字符串
  - Printf(“my name is %s”, “WeaponX”)
- 常见的格式
  - %d - 十进制 - 输出十进制整数
  - %s - 字符串 - 从内存中读取字符串
  - %x - 十六进制 - 输出十六进制数
  - %c - 字符 - 输出字符
  - %p - 指针 - 指针地址
  - %n - 到目前为止所写的字符数

格式化字符串漏洞

```
#include<stdio.h>


int main()
{
        char *name = "WeaponX";
        printf("My name is %s\n", name);
        printf("My name is %s\n");
        return 0;
}
```

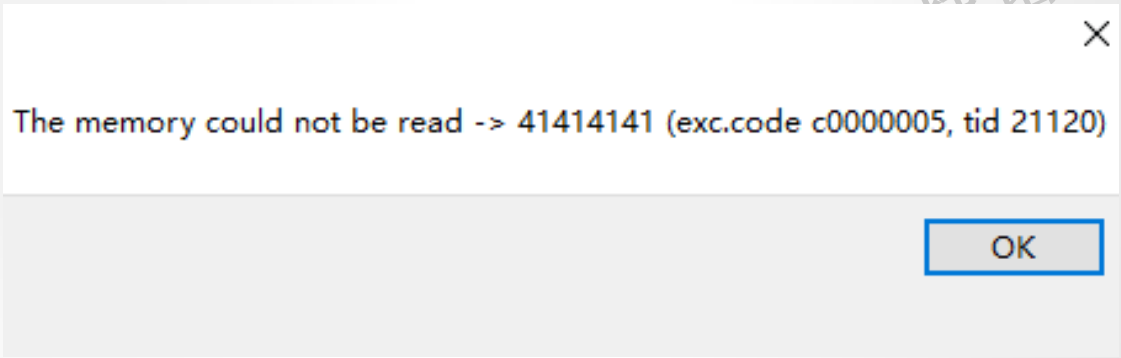


任意地址读取

```
#include<stdio.h>


int main()
{
        char str[200] = {0};
        scanf("%s", str);
        printf(str);
        return 0;
}
```

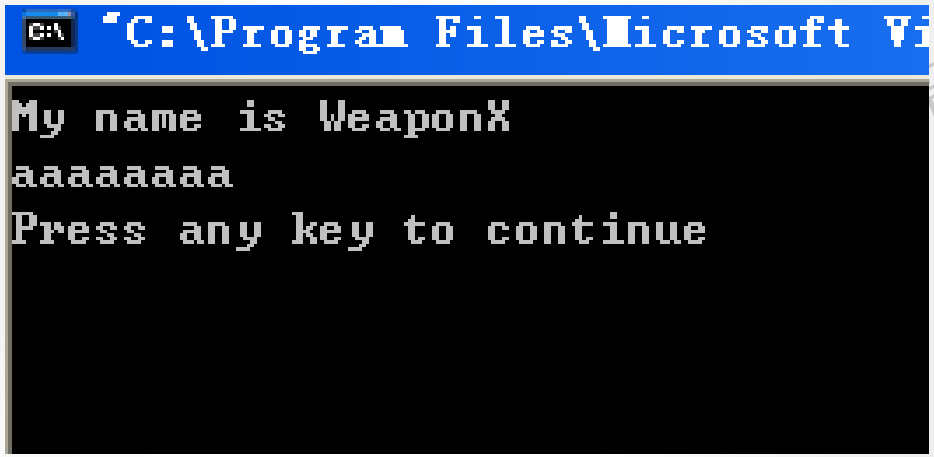
- Payload
  - AAAA%x%x%x%s



格式化字符串漏洞

```
#include<stdio.h>


int main()
{
        char *name = "WeaponX";
        printf("My name is %s\n", name);
        printf("aaaaaaaa%n\n");
        printf("My name is %s\n");
        return 0;
}
```

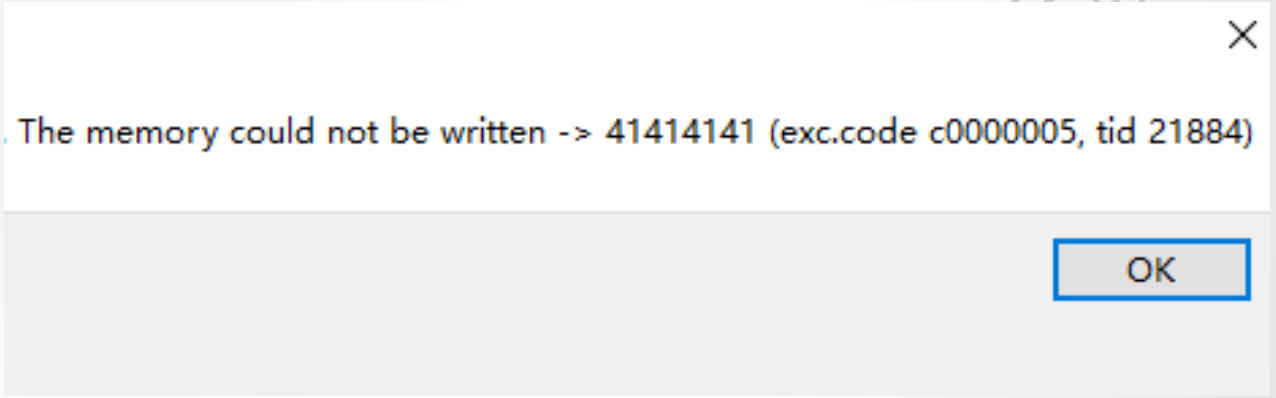


任意地址写

```
#include<stdio.h>


int main()
{
        char str[200] = {0};
        scanf("%s", str);
        printf(str);
        return 0;
}
```

- Payload
  - AAAA%x%x%x%n

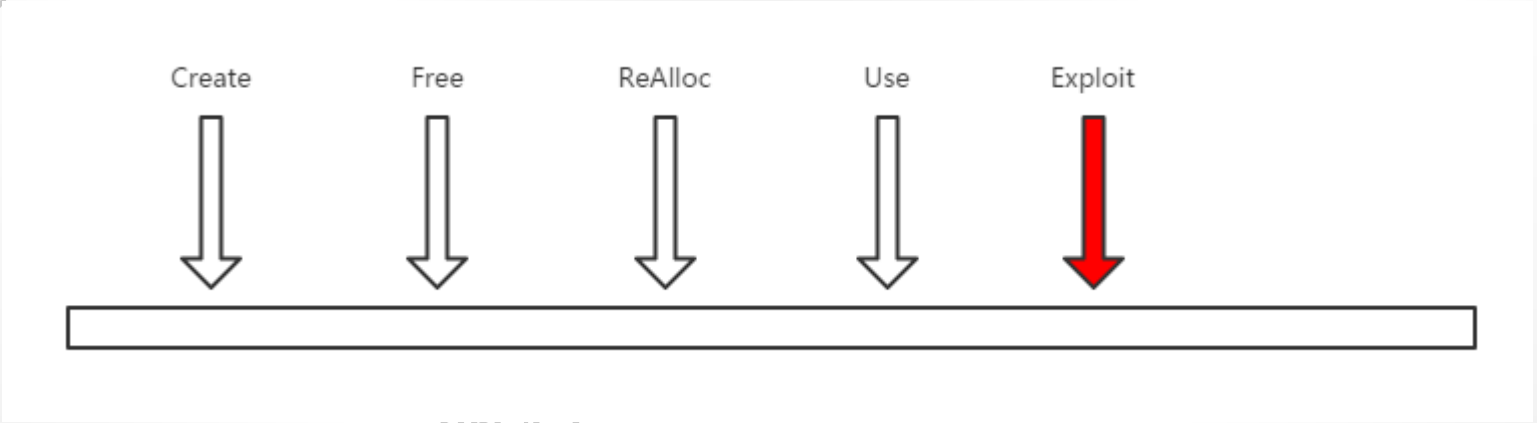


格式化字符串漏洞

- 如何写一个大的值?
  - 0xdeadbeef = 3735928559(Dec)
- %08x
  - 左边补0 的等宽格式，不够8位左边补8，超过8位则全部打印
- Notice:
  - Printf(“%08s”，“aa”) -> 000000aa
  - Printf(“%02s”，“aaaa”) -> aaaa

实验

- 原理
  - 在指针释放后再申请相同大小的内存，系统会将释放的地址进行分配，以提高系统运行速度。因此可以修改到被释放的内存数据，如果被释放的指针继续被使用，则会造成UAF漏洞。

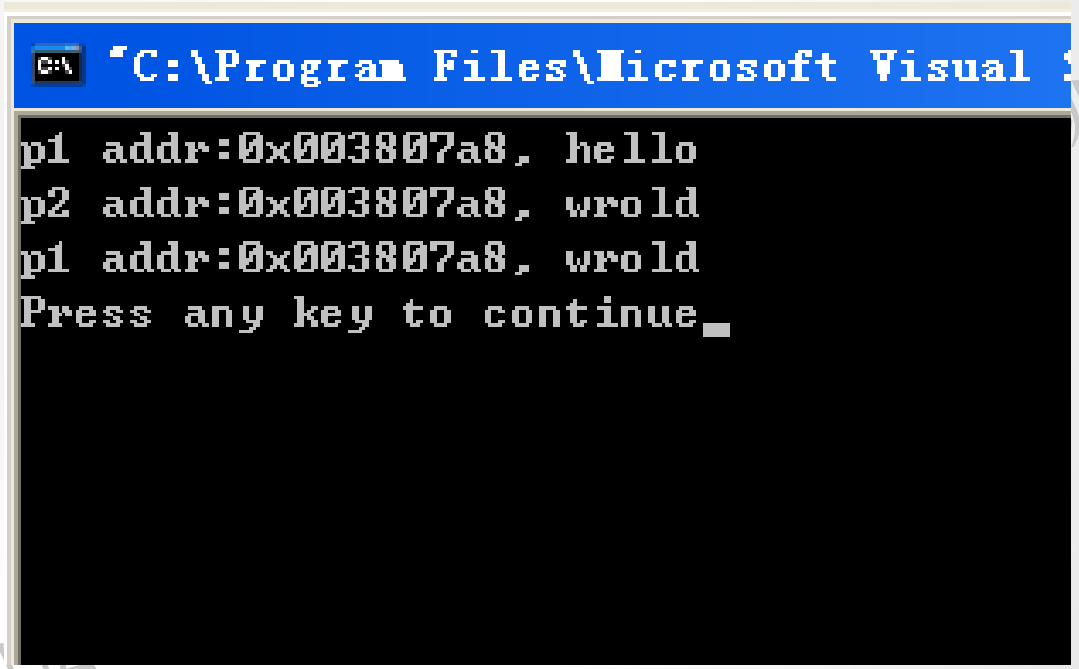


```c
int main()
{
    char *p1;
    p1 = (char *)malloc(sizeof(char)*10);
    memcpy(p1, "hello", 10);
    printf("p1 addr:0x%08x, %s\n",p1,p1);

    free(p1);

    char *p2;
    p2 = (char *)malloc(sizeof(char)*10);
    memcpy(p2, "wrold", 10);
    printf("p2 addr:0x%08x, %s\n",p2,p2);
    printf("p1 addr:0x%08x, %s\n",p1,p1);

    return 0;
}
```



常用工具下载

- Pwntools
  - https://github.com/Gallopsled/pwntools
- Ctf-tools
  - https://github.com/zardus/ctf-tools
- Mona
  - https://github.com/corelan/mona
- Roputils
  - https://github.com/inaz2/roputils
- dnspy
  - https://github.com/0xd4d/dnSpy

常用工具下载

- Windbg
  - https://msdn.microsoft.com/zh-cn/library/ff551063(v=vs.85).aspx
- Processexp
  - https://technet.microsoft.com/en-us/sysinternals/bb896653/
- Metaploit
  - https://www.metasploit.com/

THANK YOU