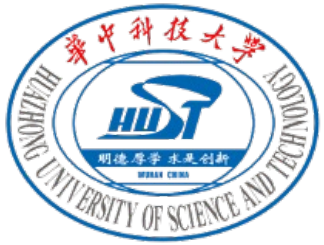


华中科技大学
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

网络空间安全学院

漏洞利用

网络空间安全学院 慕冬亮

Email : dzm91@hust.edu.cn

本讲提纲

- 1 漏洞的利用与Exploit
- 2 Shellcode开发
- 3 软件漏洞利用平台与框架
- 4 软件漏洞挖掘技术与工具

漏洞利用

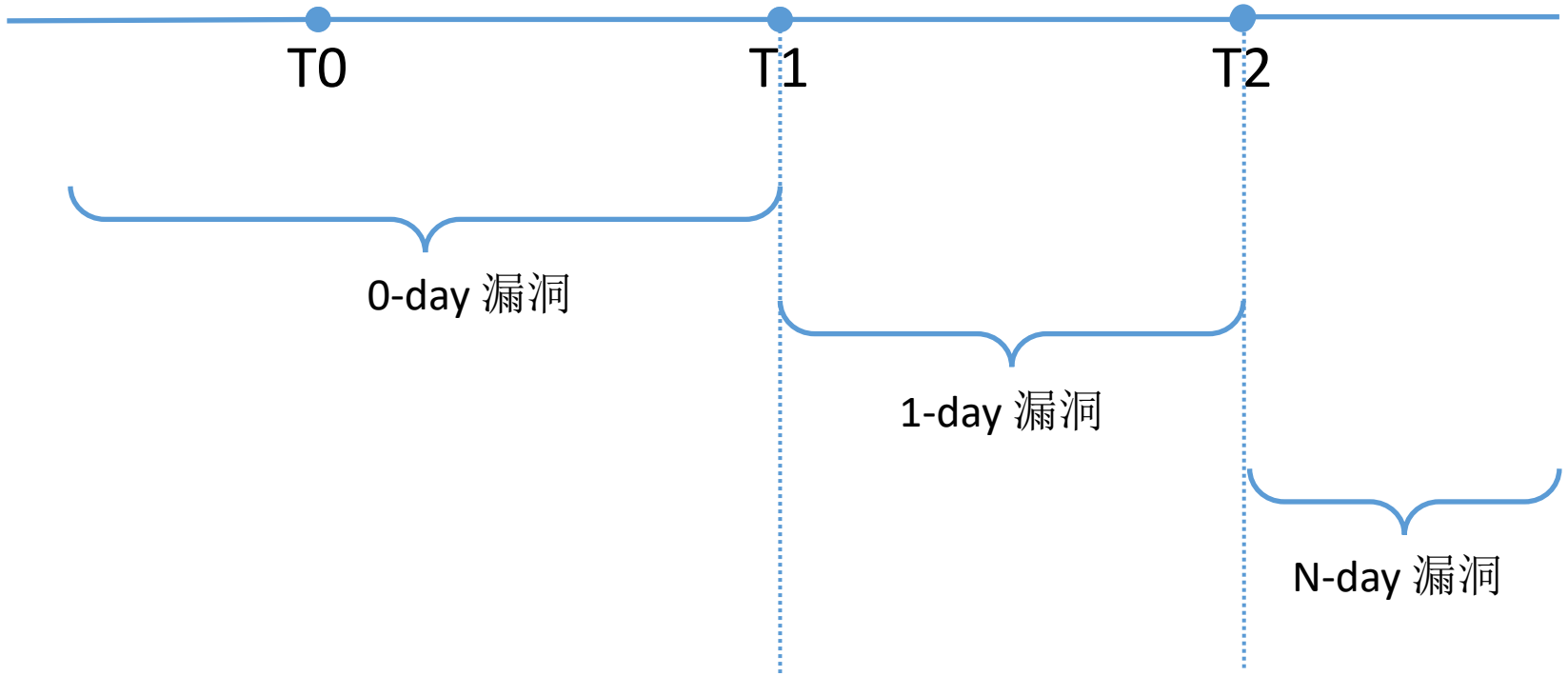
- 漏洞研究
  - 漏洞挖掘
    - 人工代码审计、工具分析挖掘
  - 漏洞分析
    - 漏洞机理、触发条件、漏洞危害
  - 漏洞利用
    - 编制触发漏洞的PoC，或者攻击者实施攻击目的的程序（Exploit）
  - 漏洞防御
    - 软件本身修补、热补丁、防御机制

漏洞利用

- 漏洞来源
  - 黑客自己挖掘的漏洞
    - 0-Day VUL
  - 从公开发布的POC或者黑客交换得到的漏洞
    - 0-Day VUL or 1-Day VUL
  - 从已发布的漏洞公告和漏洞补丁获得的漏洞
    - N-Day VUL
- 漏洞利用的条件
  - 用户没有打补丁或者更新安全工具
  - 管理员没有打补丁或者更新安全工具
  - 存在脏数据渗透路径

0-day vs 1-day vs N-days

- 漏洞被发现 - T0
- 漏洞信息公布 - T1
- 漏洞修复生成 - T2



Exploit结构

- **Exploit: 利用漏洞实现 Shellcode 的植入和触发的过程**
- **Exploit ≈ Payload+Shellcode**
- **Payload： 部署基本的数据（用于漏洞的触发）， 携带Shellcode**

结论： **Payload与漏洞关联，Shellcode独立于漏洞**

漏洞利用的思路

利用目标：

- 修改内存变量 （邻接变量）
- 修改代码逻辑（代码的任意跳转）
- 修改函数的返回地址
- 修改异常处理函数指针（SEH等）
- 修改线程同步的函数指针

漏洞利用的思路

利用过程：

- 定位漏洞点：利用静态分析和动态调试确定漏洞机理，如堆溢出、栈溢出、整数溢出的数据结构，影响范围
- 按照利用要求，编写Shellcode
- 溢出，覆盖代码指针，使得Shellcode获得可执行权

Shellcode设计

什么是**Shellcode**？

- 缘起：1996年，出现在Aleph One的论文“Smashing the Stack for fun and profit”
- 原义：the code to spawn a shell
- 延伸：攻击者植入目标内存中的代码片段

ShellCode

- Shellcode一般是作为数据形式发送给服务器制造溢出得以执行代码并获取控制权的。不同的漏洞利用方式，对于数据包的格式都会有特殊的要求，Shellcode必须首先满足被攻击程序对于数据报格式的特殊要求。

- 从总体上来讲，Shellcode具有如下一些特点：
  - 长度受限
  - 不能使用特定字符，例如\x00等
  - API函数自搜索和重定位能力。由于shellcode没有PE头，因此shellcode中使用的API和数据必须由shellcode自己进行搜索和重定位
  - 一定的兼容性。为了支持更多的操作系统平台，shellcode需要具有一定的兼容性 (难)。

ShellCode的功能

- 常见功能
  - 下载程序并运行
  - 添加管理员账号
  - 开启Shell(正向、反向)
  - …

Shellcode不通用

- Shellcode为什么不通用
  - 不同硬件平台
    - IBM PC、Alpha，PowerPC
  - 不同系统平台
    - Unix、Windows
  - 不同内核与补丁版本
  - 不同漏洞对字符串限制不同

Shellcode设计

设计流程

- 编写shellcode（高级语言）
- 反汇编该shellcode（调试）
- 从汇编级分析程序执行流程
- 生成完成的Shellcode（机器代码）
- 优化Shellcode（编码、长度）
- 适配漏洞

Shellcode编写语言

- 汇编语言（不建议）
  - 代码短小，可控性强
  - 但耗时或对语言精通

- C 语言
  - 效率高，便于入手
  - 但需要调整

- Python pwntools shellcraft 模块

地址重定位

正常情况下，在调用的时候编译器已经算好了偏移，所以不存在重定位的问题，但是，向createthread这种将传入函数地址作为参数的函数就会出现这种问题.

```
        call  GetEIP
GetEIP:
        pop   eax
        sub   eax, offset GetEIP
```

pFuns->lpThreadProc = (DWORD)dwGetEip+((DWORD)ThreadProc-(DWORD)GetEip)

API函数地址自搜索

Shellcode外部的函数重定位

- 获得kernel32.dll的基地址
  - LoadlibraryA
  - GetProcAddress
- 函数名的压缩表示
  - fValue=Hash(fName)

```
push    2
pop     ecx
mov     eax, fs:[ecx+2Eh]
; fs:[30] -> PEB
mov     eax, [eax+0Ch]
; Ldr : _PEB_LDR_DATA
mov     eax, [eax+1Ch]  ; InInitializationOrderModuleList
mov     eax, [eax]      ; 下一个节点
mov     ebx, [eax+8]    ; kernel32.dll 基地址
lea     esi, [edi+0A1h] ; 000000A6 -> 768AA260
```

Shellcode编码问题

- 需求：
  - 不能含有0x00(字符串的结束符)
  - 只能为可打印字符
  - 字符覆盖（恢复）
  - 对抗IDS的特征码检测
- 办法：
  - 借代码（jmp ESP）
  - XOR编码（分段编码）

```
fDecode:   ; CODE XREF:
xor     byte ptr [ecx], 0C4h ; xor 解密ShellCode
inc     ecx
cmp     word ptr [ecx], 534Dh ; 结束符 MS
jnz     short fDecode   ; ShellCode解码Stub
cld                     ; ShellCode开始
```

Shellcode典型功能

- 正向连接（目标主机开一个服务端口）
- 反向连接（目标主机主动连接攻击者的控制端）
- 下载并执行程序
- 动态生成可执行程序并执行
- 执行一个程序
- 打开Shell
- 消息弹框
- ……

实践-如何编写Shellcode(1)

- 一般先用C语言写出功能代码

```
int main0()
{
	LoadLibrary("msvcrt.dll");
	system("command.com");
	//1.how to get string
	//2.how to get API address
	return 0;
}
```

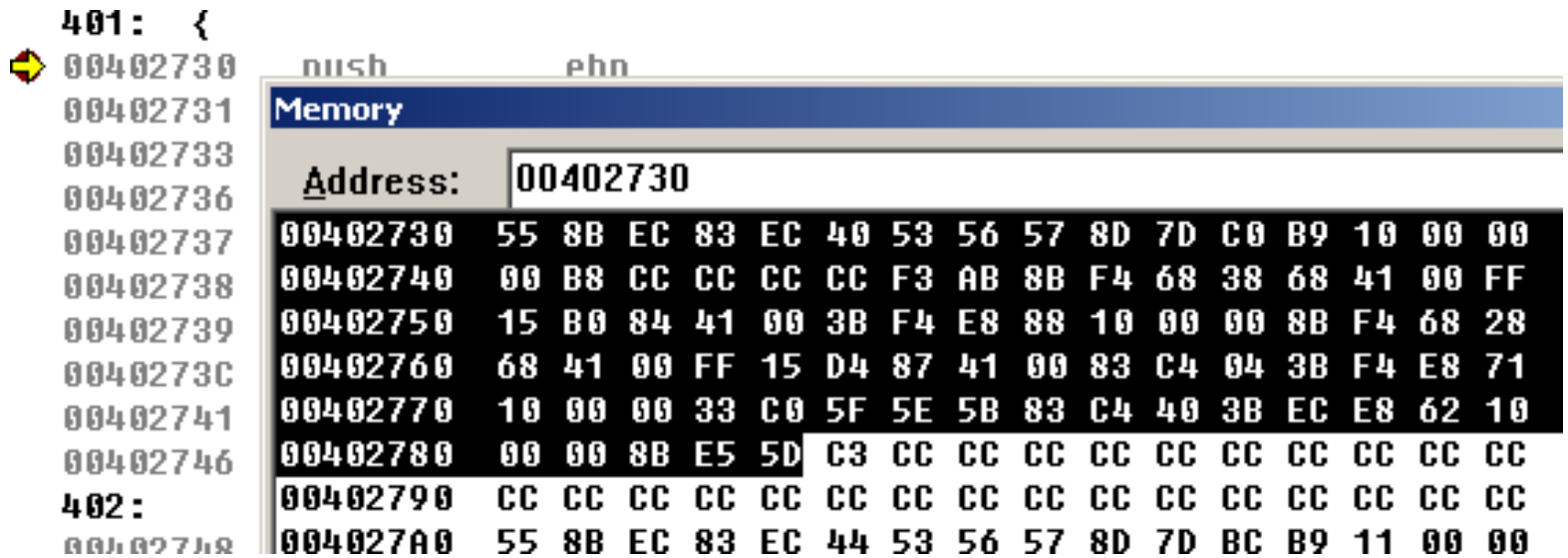
实践-如何编写Shellcode(2)

- 反汇编得到二进制代码

```
400:  int main0()
401:  {
00402730   push        ebp
00402731   mov         ebp,esp
00402733   sub         esp,40h
00402736   push        ebx
00402737   push        esi
00402738   push        edi
00402739   lea         edi,[ebp-40h]
0040273C   mov         ecx,10h
00402741   mov         eax,0CCCCCCCCh
00402746   rep stos    dword ptr [edi]
402:       LoadLibrary("msvcrt.dll");
00402748   mov         esi,esp
0040274A   push        offset string "msvcrt.dll" (00416838)
0040274F   call        dword ptr [__imp__LoadLibraryA@4 (004184b0)]
00402755   cmp         esi,esp
00402757   call        _chkesp (004037e4)
403:       system("command.com");
0040275C   mov         esi,esp
0040275E   push        offset string "command.com" (00416828)
00402763   call        dword ptr [__imp__system (004187d4)]
00402769   add         esp,4
0040276C   cmp         esi,esp
0040276E   call        _chkesp (004037e4)
404:       //1.how to get string
405:       //2.how to get API address
406:       return 0;
00402773   xor         eax,eax
407:  }
00402775   pop         edi
00402776   pop         esi
00402777   pop         ebx
00402778   add         esp,40h
0040277B   cmp         ebp,esp
0040277D   call        _chkesp (004037e4)
00402782   mov         esp,ebp
00402784   pop         ebp
00402785   ret
```

实践-如何编写Shellcode(3)

- 找到代码内存地址，列拷贝



实践-如何编写Shellcode(4)

- 作为二进制数组变量，初步ShellCode完成

```
char shellcodeOwn1[] =
"\x55\x8B\xEC\x33\xC0\x50\x83\xEC\x08\xC7\x45\xF4\x6D\x73\x76\x63"
"\xC7\x45\xF8\x72\x74\x2E\x64\xC6\x45\xFC\x6C\xC6\x45\xFD\x6C\x8B"
"\xDD\x83\xEB\x0C\x53\xBA\x77\x1D\x80\x7C\xFF\xD2\x8B\xE5\x33\xC0"
"\x50\x83\xEC\x08\xC7\x45\xF4\x63\x6F\x6D\x6D\xC7\x45\xF8\x61\x6E"
"\x64\x2E\xC6\x45\xFC\x63\xC6\x45\xFD\x6F\xC6\x45\xFE\x6D\x8B\xC5"
"\x83\xE8\x0C\x50\xB8\xC7\x93\xBF\x77\xFF\xD0";
```

实践-如何编写Shellcode(5)

- 初步ShellCode的问题

ShellCode作为恶意代码，精简高效，无上下文环境的充分准备，无正确堆栈…

```
401:  {
00402730   push        ebp
00402731   mov         ebp,esp
00402733   sub         esp,40h
00402736   push        ebx
00402737   push        esi
00402738   push        edi
00402739   lea         edi,[ebp-40h]
0040273C   mov         ecx,10h
00402741   mov         eax,0CCCCCCCCh
00402746   rep stos    dword ptr [edi]
402:       LoadLibrary("msvcrt.dll");
00402748   mov         esi,esp
0040274A   push        offset string "msvcrt.dll" (00416838)
0040274F   call        dword ptr [__imp__LoadLibraryA@4 (004184b0)]
```

1. 怎么得到字符串?

2. 怎么得到API地址?

实践-如何编写Shellcode(6)

- 完善/优化ShellCode

1. 运行时直接构造变量

2.API地址硬编码

```
__asm{
    push ebp ;//保存ebp, esp－4
    mov ebp,esp ;//把ebp的内容赋值给esp
    sub esp,0dh
    push edx
    push eax
    push ebx
    //调用LoadLibrary("msvcrt.dll");
    mov byte ptr[ebp-0Ch],6dh
    mov byte ptr[ebp-0Bh],73h
    mov byte ptr[ebp-0Ah],76h
    mov byte ptr[ebp-09h],63h
    mov byte ptr[ebp-08h],72h
    mov byte ptr[ebp-07h],74h
    mov byte ptr[ebp-06h],2eh
    mov byte ptr[ebp-05h],64h
    mov byte ptr[ebp-04h],6ch
    mov byte ptr[ebp-03h],6ch
    mov byte ptr[ebp-02h],0h
    mov ebx,ebp
    sub ebx,0ch
    push ebx
    mov edx,0x7c801d77
    call edx
```

实践-如何编写Shellcode(6)

- 完善/优化ShellCode

ShellCode在实际利用中困难更多… …

实践-使用pwntools编写shellcode

- cat 这个 shellcode 的具体的实现

- from pwnlib.shellcraft.amd64 import syscall, pushstr
- from pwnlib.shellcraft import common

- pushstr(filename)
- syscall('SYS_open', 'rsp', 'O_RDONLY', 'rdx')
- syscall('SYS_sendfile', fd, 'rax', 0, 0x7fffffff)

https://github.com/Gallopsled/pwntools/blob/dev/pwnlib/shellcraft/templates/amd64/linux/cat.asm

然而....

- 简单的 ShellCode 几乎都会失效
- 比如：DEP，ASLR
- ……

数据执行保护-DEP

- Shellcode一般位于堆或者栈中，堆栈中的Shellcode能执行源于冯洛伊曼体系结构中的代码和数据混合存储。

- 代码和数据的分离
  - Data Execution Protection(DEP) / No Execute (NX)
  - 禁用 Stack/Heap 中的代码执行
  - 带来兼容性、灵活性问题

- INTEL，AMD，ARM 等 CPU 支持 DEP
  - 硬件特性

数据执行保护-NX(Linux)

- Checksec 命令，安装pwntools后自动安装checksec命令，可以用来查看可执行文件开启保护情况，包括RELRO，Stack，NX，PIE。
- 也可以使用 sudo apt-get install checksec 安装

```
# liber @ liber-MS-7D42 in ~/Downloads/software-security-dojo/integer-overflow/level-1-1 on git:main x [22:26:17]
$ checksec --file=foritytest
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable
Full RELRO      Canary found      NX enabled    PIE enabled     No RPATH   No RUNPATH   38) Symbols       No    0              1
```

数据执行保护-NX(Linux)

- -z execstack : GCC默认开启NX保护，添加参数后，编译选项开启，则NX disabled
- 在 Linux 中，当装载器将程序装载进内存空间后，将程序的 .text 段标记为可执行，而其余的数据段（.data、.bss 等）以及栈为不可执行。因此，传统利用方式中通过执行 shellcode 的方式不再可行

```
root@f953676d993d:/mnt/software-securi [*] '/mnt/software-security-dojo/integer-ec integer-overflow-level1.1
[*] '/mnt/software-security-dojo/integ     Arch:     amd64-64-little           el1.1'
    Arch:     amd64-64-little              RELRO:    Partial RELRO
    RELRO:    Partial RELRO                Stack:    No canary found
    Stack:    No canary found              NX:       NX disabled
    NX:       NX enabled                   PIE:      No PIE (0x400000)
    PIE:      No PIE (0x400000)            RWX:      Has RWX segments
root@f953676d993d:/mnt/software-securi root@f953676d993d:/mnt/software-security-
```

数据执行保护-NX(Linux)

- 只有stack 栈段在 NX 保护不开启的时候拥有可执行权限，反之则没有

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset File
           0x400000           0x401000 r--p     1000      0 /mnt/software-security-dojo/integer-overflow/level-1-1/nx_test
           0x401000           0x402000 r-xp     1000   1000 /mnt/software-security-dojo/integer-overflow/level-1-1/nx_test
           0x402000           0x403000 r--p     1000   2000 /mnt/software-security-dojo/integer-overflow/level-1-1/nx_test
           0x403000           0x404000 rw-p     1000   2000 /mnt/software-security-dojo/integer-overflow/level-1-1/nx_test
      0x7f390b49f000     0x7f390b4c1000 r--p    22000      0 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7f390b4c1000     0x7f390b639000 r-xp   178000  22000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7f390b639000     0x7f390b687000 r--p    4e000 19a000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7f390b687000     0x7f390b68b000 r--p     4000 1e7000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7f390b68b000     0x7f390b68d000 rw-p     2000 1eb000 /usr/lib/x86_64-linux-gnu/libc-2.31.so
      0x7f390b68d000     0x7f390b693000 rw-p     6000      0 [anon_7f390b68d]
      0x7f390b69b000     0x7f390b69c000 r--p     1000      0 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7f390b69c000     0x7f390b6bf000 r-xp    23000   1000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7f390b6bf000     0x7f390b6c7000 r--p     8000  24000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7f390b6c8000     0x7f390b6c9000 r--p     1000  2c000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7f390b6c9000     0x7f390b6ca000 rw-p     1000  2d000 /usr/lib/x86_64-linux-gnu/ld-2.31.so
      0x7ffc60cef000     0x7ffc60d10000 rwxp    21000      0 [stack]
      0x7ffc60df8000     0x7ffc60dfa000 r-xp     2000      0 [vdso]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
pwndbg>
```

```
      0x7f02e4e5d000     0x7f02e4e5e000 rw-p     1000      0 [anon_7f02e4e5d]
      0x7ffc7718d000     0x7ffc771ae000 rw-p    21000      0 [stack]
      0x7ffc771b5000     0x7ffc771b9000 r--p     4000      0 [vvar]
      0x7ffc771b9000     0x7ffc771bb000 r-xp     2000      0 [vdso]
0xffffffffff600000 0xffffffffff601000 --xp     1000      0 [vsyscall]
pwndbg>
```

数据执行保护-NX(Linux)

- NX 标记位在内核中的实现（32位）

```
static void mark_nxdata_nx(void)
{
        /*
         * When this called, init has already been executed and released,
         * so everything past _etext should be NX.
         */
        unsigned long start = PFN_ALIGN(_etext);
        /*
         * This comes from is_x86_32_kernel_text upper limit. Also HPAGE where used:
         */
        unsigned long size = (((unsigned long)__init_end + HPAGE_SIZE) & HPAGE_MASK) - start;

        if (__supported_pte_mask & _PAGE_NX)
                printk(KERN_INFO "NX-protecting the kernel data: %luk\n", size >> 10);
        set_memory_nx(start, size >> PAGE_SHIFT);
}
```

江湖诞生 Ret2Libc & ROP

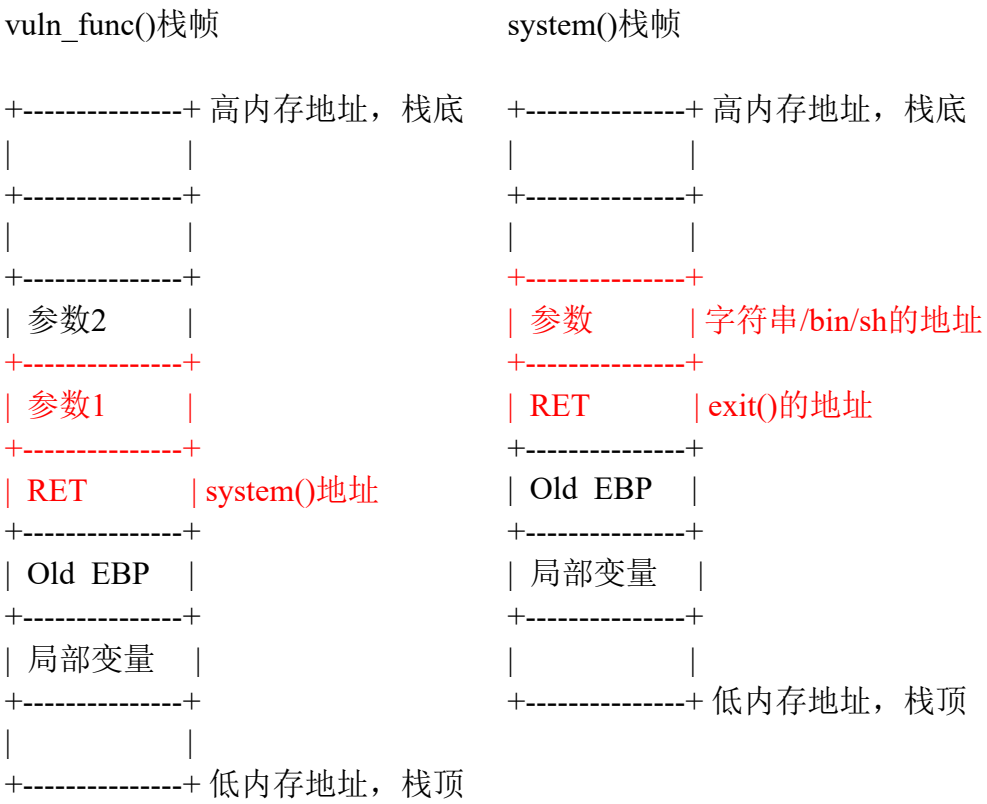
- 缘起：
  - 减小 Shellcode 的长度
  - 绕过 DEP
- 办法：从软件及其依赖库借代码
  - 借函数（Ret2Libc）
  - 借代码片段（ROP）

Ret2Libc

- Ret2Libc
  - 在栈中准备好函数的参数以及返回地址
  - 溢出修改函数的返回地址

- 实例
  - system(“/bin/sh”)＋exit()
  - system函数的地址填充到eip的位置，然后再把”/bin/bash”地址填充到RET + 4的位置

此处代码仅针对32位情况



Ret2Libc

- 怎么防止 Ret2Libc?

ASCII armoring

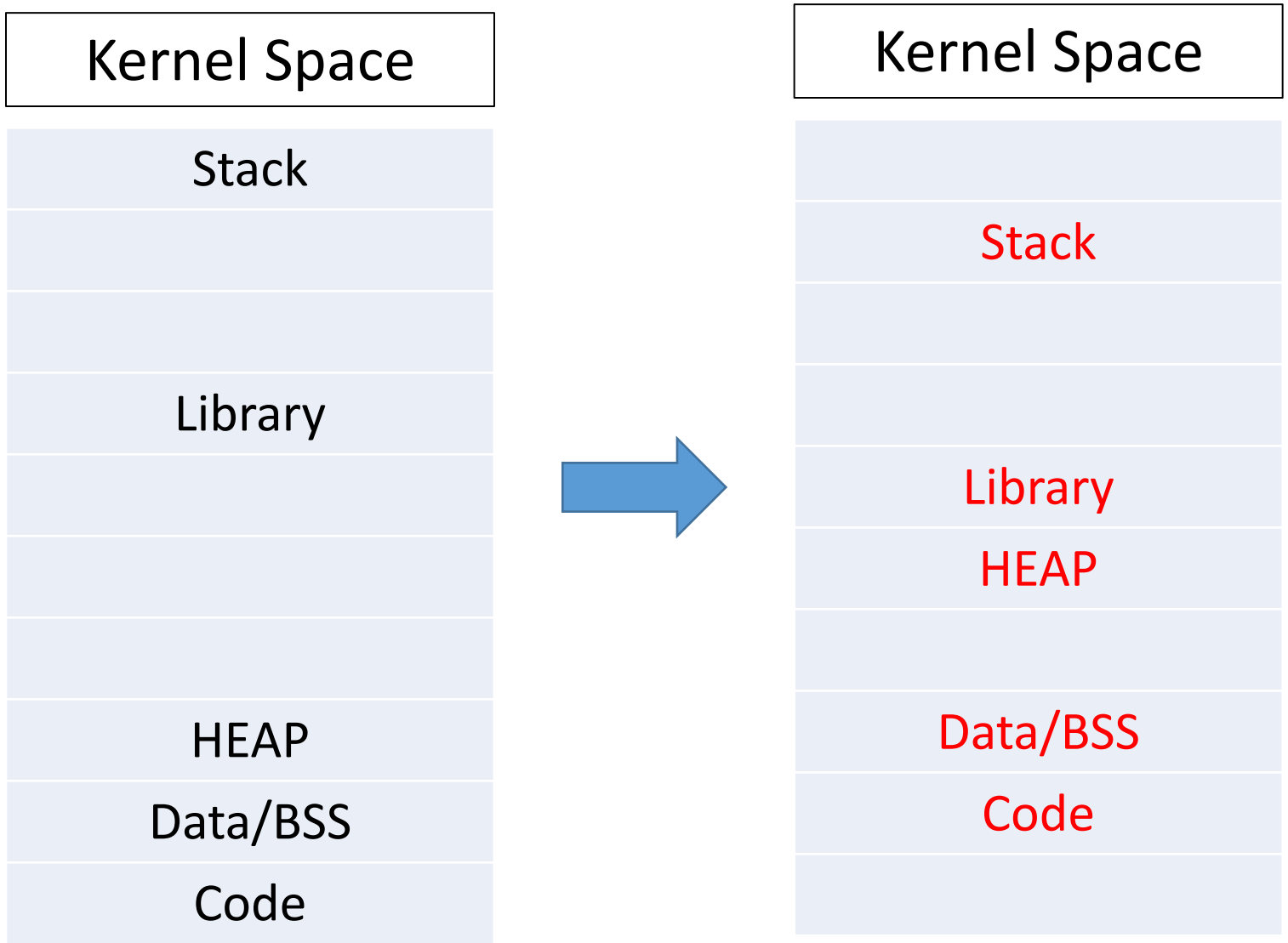
ASCII armoring机制想办法让libc所有函数的地址都包含一个零字节，让strcpy拷贝函数在遇到零地址时结束拷贝，攻击失败!

- Ret2PLT

找到4个地址空间，它的首字节分别是system地址的第一个byte，第二个byte，第三个byte和第四个byte，然后一个个byte拷贝，将这4个byte拼凑到函数调用表里面。从而绕过直接拷贝system地址造成失败。

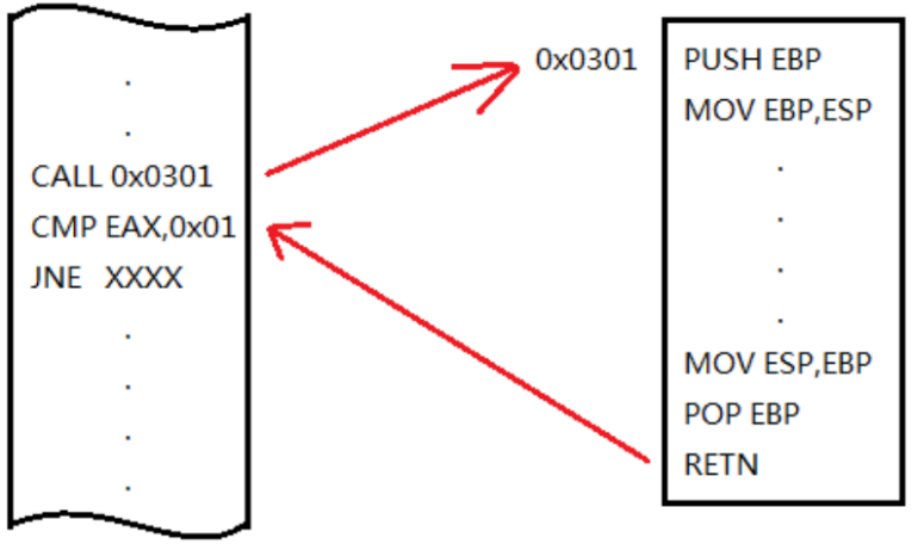
思考：该Ret2PLT攻击成功的前提是什么？

ASLR


Kernel Space
Stack
Library
HEAP
Data/BSS
Code
Kernel Space
Stack
Library
HEAP
Data/BSS
Code

ROP

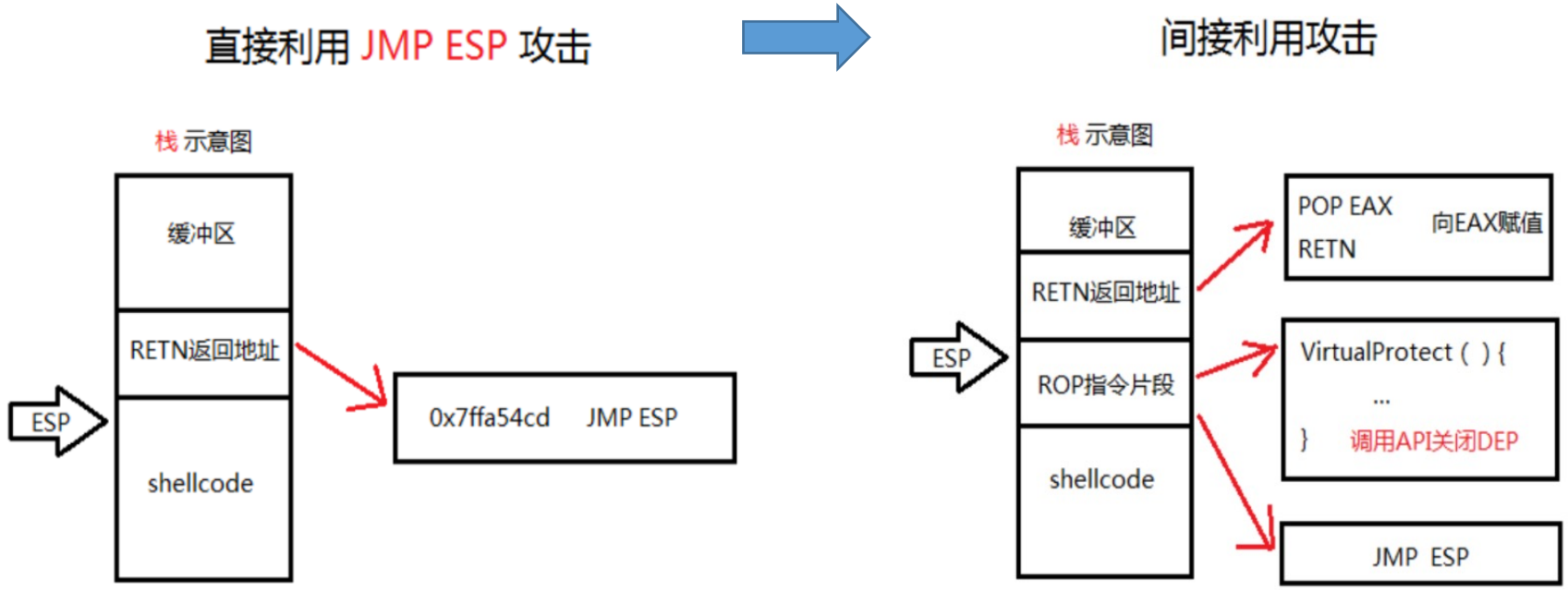
- DEP 保护是拦路虎！ 黑客的思维是无法预料的...
- 汇编语言中有一系列非常有用的指令，叫“RETN系列指令”，这些指令的原始功能是当函数调用完成时，回退到上一层调用函数，并继续下面的执行，示意图如下：



DEP的保护机制，虽然安全，但操作系统在做某些操作时受到限制，所以操作系统中又提供了一些解除DEP保护的API供软件开发人员调用，当攻击者在内存中定位到这些API并调用时，DEP保护便失去作用了。这些API一直散落于内存的某些角落，当攻击者触发它们时，就好像触发了某个密室的暗门一样，豁然开朗。

ROP

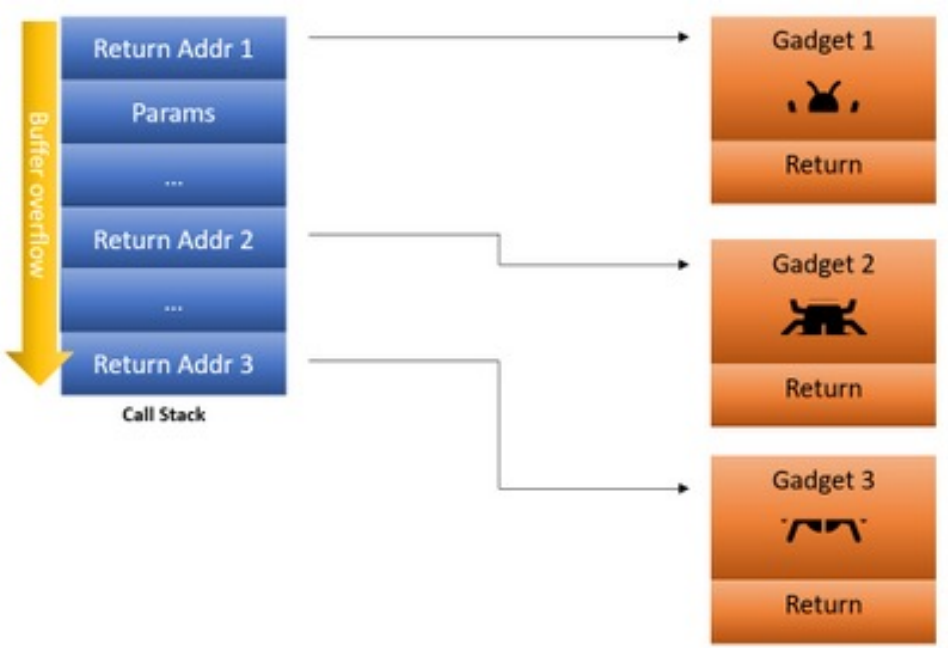
- 当RETN指令同这些 API 结合时，就会产生一些奇妙的....
- 借用系统现有指令完成攻击，示意图如下



在第二种攻击下，**EIP**的控制是通过栈中的地址，以及代码中的**RETN**指令共同控制的，此时栈中的数据仍然是“数据”，而执行位置却转移到了内存的代码空间，如此下来便巧妙的绕过了**DEP**保护。这种绕过技术其实是基于一个特定条件的，那就是到某个地址一定能找到对应的包含**RETN**的代码片段，可以说这是当前漏洞利用方式的薄弱环节。

ROP

- ROP：栈中所有的代码地址不是函数的开始地址，而是位于函数体中的地址。
  - 在栈中准备好数据以及返回地址(Prepare var &retAddr)
  - 溢出修改函数的返回地址（return to code in Function）
  - 栈中只有数据和代码指针



二进制代码重用示例

（以代码重用实现了两个值的加法运算,并将结果写入指定内存地址 0x400000. ）

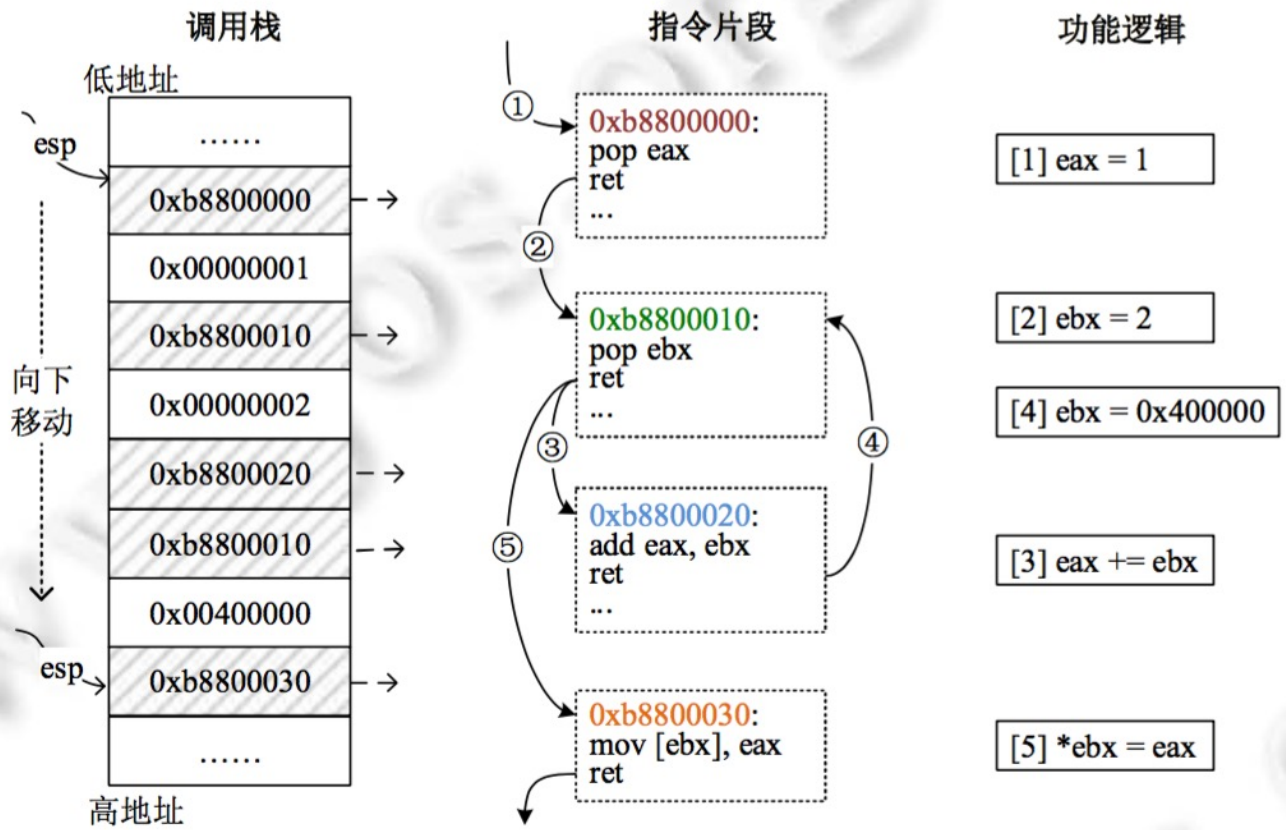


Fig.1 An example of binary code reuse and connected code blocks

图 1 二进制代码重用示例及指令片段连续调用

ROPgadget

- 用于搜索二进制文件中的 gadgets

- 安装
  - sudo apt install python3-pip
  - sudo -H python3 -m pip install ROPgadget

- 使用方法
  - ROPgadget --help：查看帮助信息
  - --binary：指定要分析的二进制文件，并展示其包含的 gadgets
  - --string：用于搜索字符串
  - --only：只显示包含指定指令的gadgets
  - --ropchain：试图生成 ROP 链，不保证成功

ROP 常用利用链

- X86_64系统调用与寄存器
  - 系统调用号由 RAX 控制
  - 系统调用参数传递顺序： RDI, RSI, RDX, R10, R8, R9
  - ROP 控制寄存器RDI，例如 POP RDI; RET

◆ROP 常见利用思路

- ORW -- 打开、读取并输出flag文件

函数调用	系统调用
fd = open(pathname, flags)	$rax=syscall($rax=__NR_open, $rdi, $rsi)
read(fd, buffer, count)	syscall($rax=__NR_read, $rdi, $rsi, $rdx)
write(fd, buffer, count)	syscall($rax=__NR_write, $rdi, $rsi, $rdx)

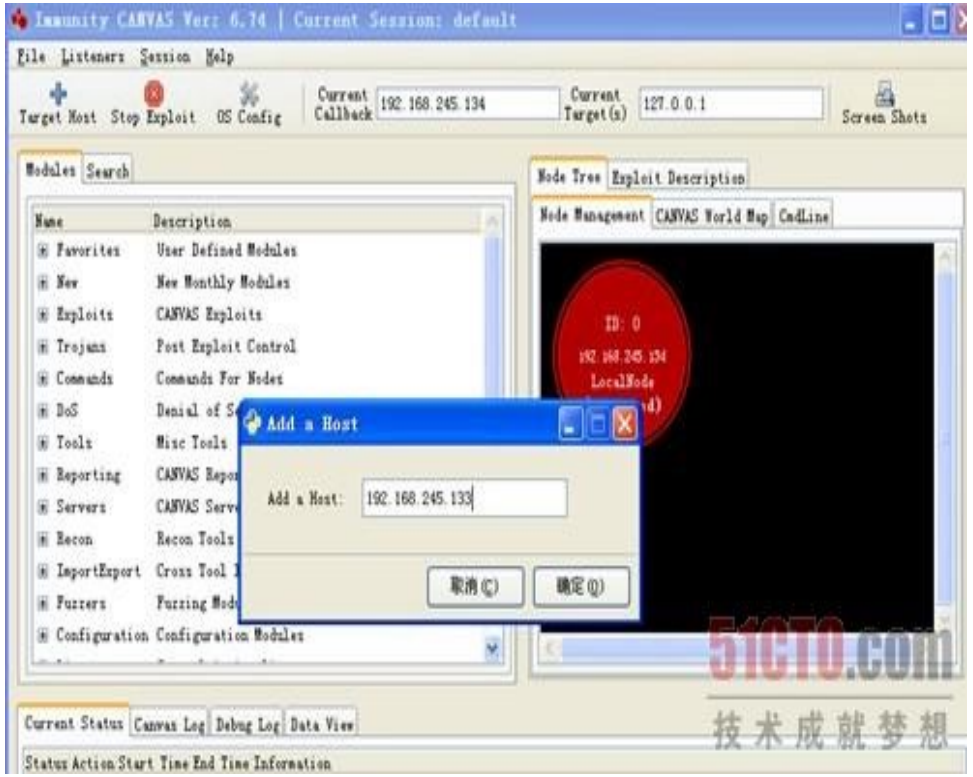
JOP/COP

- Jump-oriented programming（跳转导向编程）
- JOP 攻击利用的是程序间接接跳转和间接调用指令（间接 call 指令）来改变程序的控制流
- 程序将从指定寄存器中获得其跳转的目的地址
- 攻击者又能通过修改栈中的内容来修改寄存器内容，这使得程序中间接跳转和间接调用的目的地址能被攻击者篡改
- 当攻击者篡改这些寄存器当中的内容时，攻击者就能够使程序跳转到攻击者所构建的 gadget 地址处，进而实施 JOP 攻击

攻击手段无止境…

软件漏洞利用平台

- Metaspoit
- Immunity Canvas
- ……



Metasploit Framework (MSF)

- 2004年8月，在拉斯维加斯如开了一次世界黑客交流会---黑帽简报（Black Hat Briefings). 在这个会议上，一款叫Metasploit 的攻击和渗透工具备受众黑客关注，出尽了风头。
  - 吸引了来自“美国国防部”和“国家安全局”等政府机构的众多安全顾问和个人
  - Metasploit 很简单，只需要求“找到目标，单击和控制”即可。

- Metasploit 是HD Moore 和 Spoonm 等4名年轻人开发的，这款免费软件可以帮助黑客攻击和控制计算机，安全人员也可以利用 Metasploit 来加强系统对此类工具的攻击。

Metasploit Framework

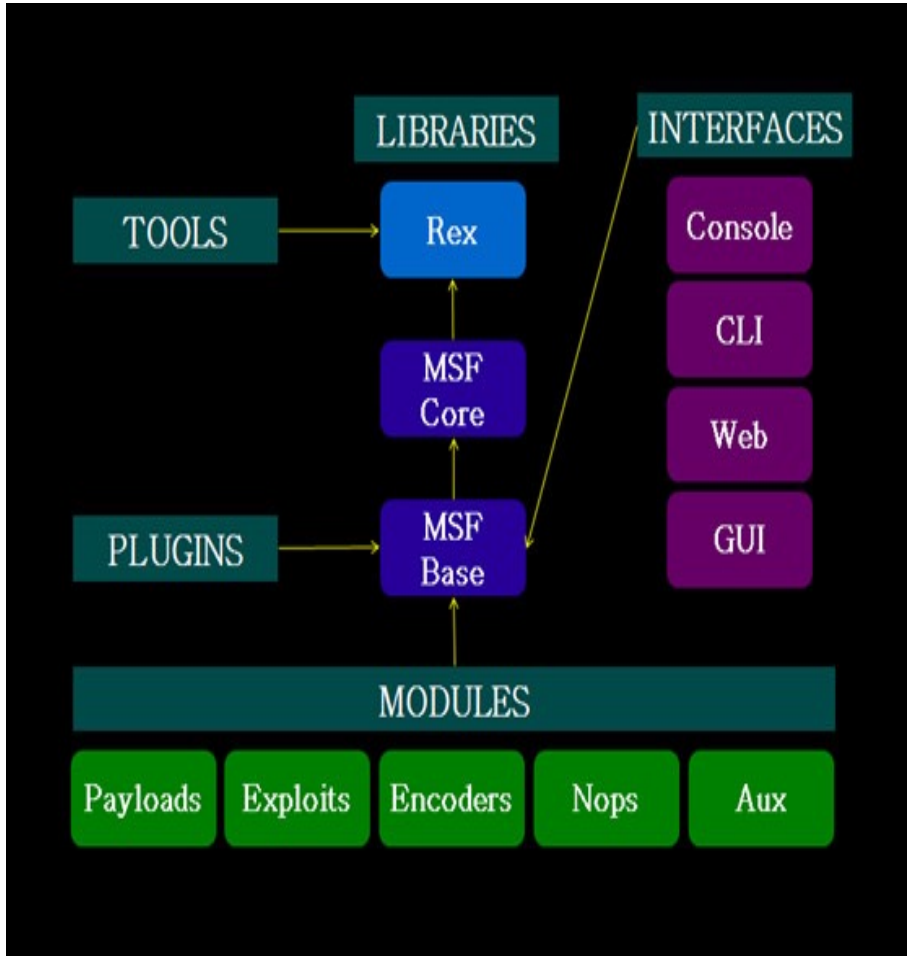
Metasploit Framework (MSF) 在2003年以开放源码方式发布，是可以自由获取的开发框架。它是一个强大的开源平台，供开发，测试和使用恶意代码，这个环境为渗透测试、shellcode 编写和漏洞研究提供了一个可靠平台。

这种可以扩展的模型将负载控制（payload），编码器（encoder），无操作指令（nops）生成器和漏洞整合在一起，使 Metasploit Framework 成为一种研究高危漏洞的途径。

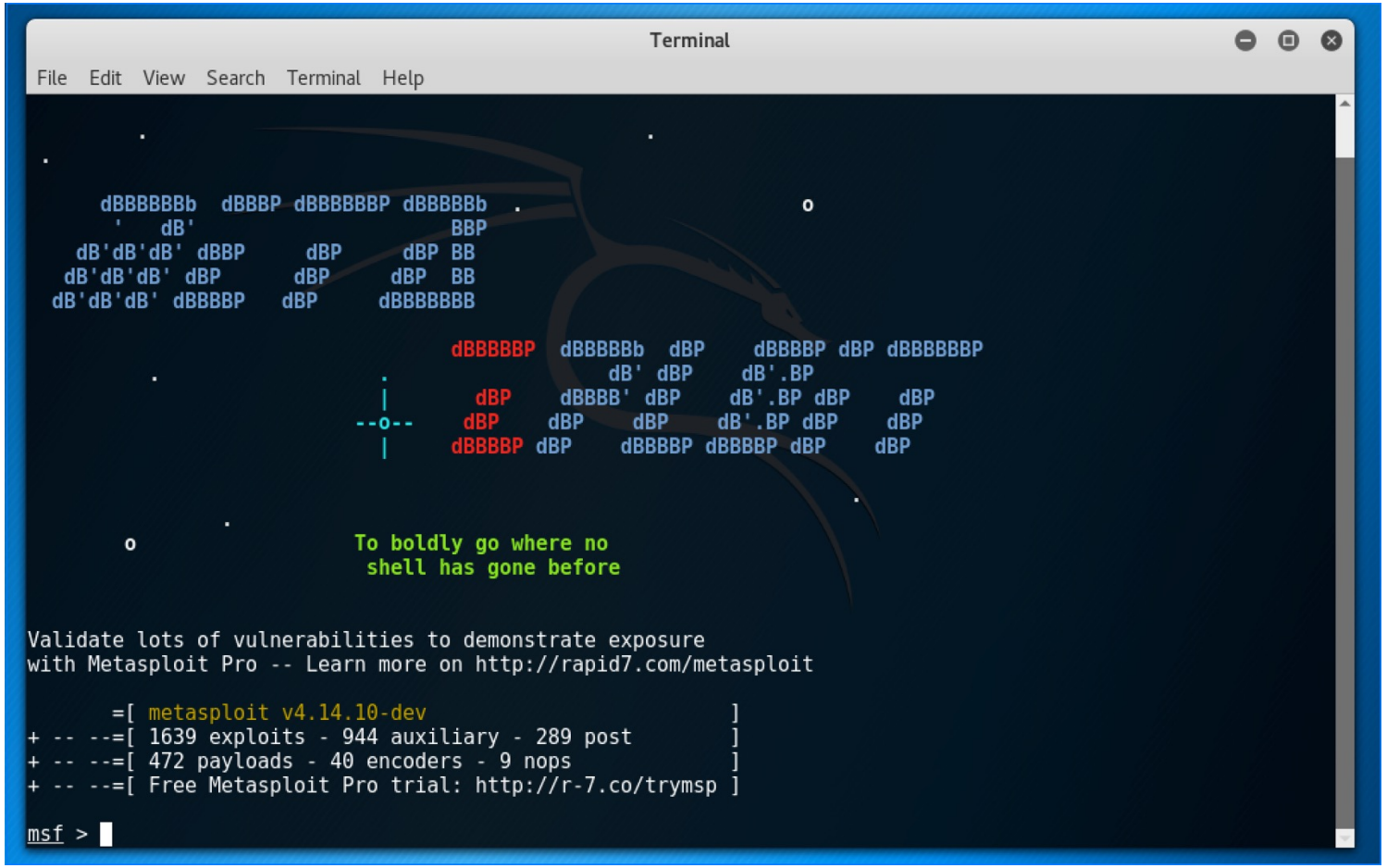
- ✓ 它集成了各平台上常见的溢出漏洞和流行的 shellcode，并且不断更新。
- ✓ MSF 包含了上千种流行的操作系统的应用软件的漏洞，以及数百个 payload。
- ✓ 作为安全工具，它在安全检测中用着不容忽视的作用，并为漏洞自动化探测和及时检测系统漏洞提供了有力保障。

软件漏洞利用平台

TOOLS	集成了各种实用工具，多数为收集的其它软件
PLUGINS	各种插件，多数为收集的其它软件。直接调用其API，但只能在console工作。
MODULES	目前的Metasploit Framework 的各个模块
MSF core	表示Metasploit Framework core 提供基本的API，并且定义了MSF的框架。并将各个子系统集成在一起。组织比较散乱，不建议更改。
MSF Base	提供了一些扩展的、易用的API以供调用，允许更改
Rex LIBRARIES	Metasploit Framework中所包含的各种库，是类、方法和模块的集合
CLI	表示命令行界面
GUI	图形用户界面
Console	控制台用户界面
Web	网页界面，目前已不再支持
Exploits	定义实现了一些溢出模块，不含payload的话是一个Aux
Payload	由一些可动态运行在远程主机上的代码组成
Nops	用以产生缓冲区填充的非操作性指令
Aux	一些辅助模块，用以实现辅助攻击，如端口扫描工具
Encoders	重新进行编码，用以实现反检测功能等



Metasploit Framework (MSF)



推荐书籍

- 诸葛建伟等，Metasploit渗透测试指南，电子工业出版，2012年