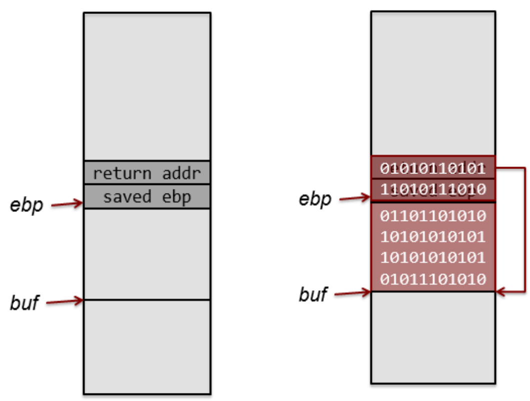
28 ROP & CFI

复习：Stack Buffer Overflow



A Typical Buffer Overflow Attack

- Inject malicious code in buffer
  - E.g., shellcode
- Overwrite return address to buffer
- Once return, the malicious code runs

```
void function(char *str) {
    char buf[16];
    strcpy(buf,str);
}
```

Shellcode Sample to Open "/bin/sh" (21 bytes)

```
char sc[] =   "\x6a\x0b"                  // push byte 0xb
              "\x58"                      // pop eax (now eax=0xb)
              "\x99"                      // cdq
              "\x52"                      // push edx (now edx=0)
              "\x68\x2f\x2f\x73\x68"      // push dword 0x68732f2f
              "\x68\x2f\x62\x69\x6e"      // push dword 0x6e69622f
              "\x89\xe3"                  // mov ebx, esp
              "\x31\xc9"                  // xor ecx, ecx (now ecx=0)
              "\xcd\x80";                 // int 0x80
```

int 0x80：int是interpret缩写，这里其实是exception，表示触发系统调用，进入kernel

举例：命令passwd保存用户的密码，记录在电脑的/etc/password和/etc/shadow里，这两个文件夹需要root权限，但是普通用户也可以使用passwd命令，这说明执行这条命令有一段时间会有root权限，可以在这一小段的时间里execv("/bin/sh")进行攻击

ROP : Return-oriented Programming

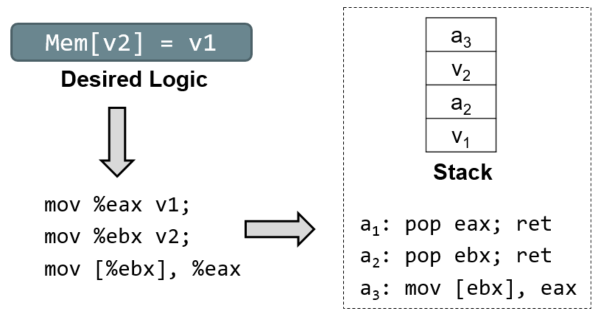
A1：部分数据可写不可执行

如 stack/heap 就是不可执行的

B1：到可执行序曲找到相同语义的代码，重定向到这些代码上(ROP)

ROP: Return-oriented Programming

- Find code gadgets in existed code base
  - Usually 1-3 instructions, ends with 'ret'
  - In libc and application, intended and unintended
- Push address of gadgets on the stack
- Leverage 'ret' at the end of gadget to connect each code gadgets
- **No code injection**

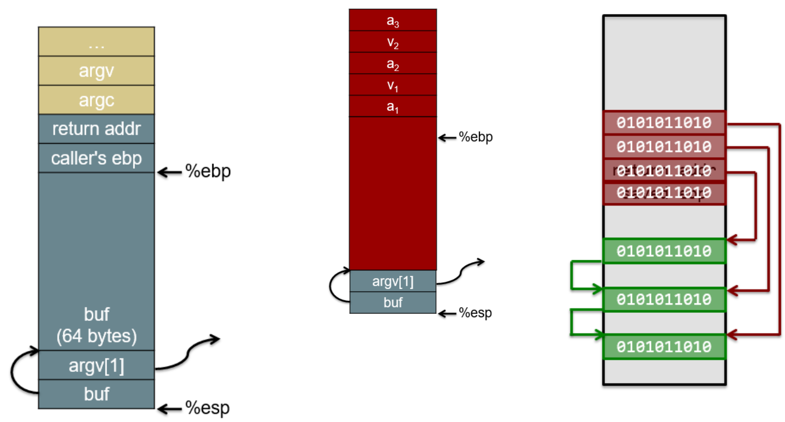


Code Execution – ROP Way

`Mem[v2] = v1`

Desired *Shellcode*

```
a1: pop eax; ret
a2: pop ebx; ret
a3: mov [ebx], eax
```



A2：三种防御方法

Hide the binary file

- No way to get any gadget

ASLR to randomize the code position

- Short for "Address Space Layout Randomization"
- Harder to find gadgets

Canary to protect the stack

- Try to detect stack overflow (e.g., overflow return address)

A2-1 藏二进制文件

A2-2 **ASLR**加一个扰动，使得每次生成的汇编都有一点差别

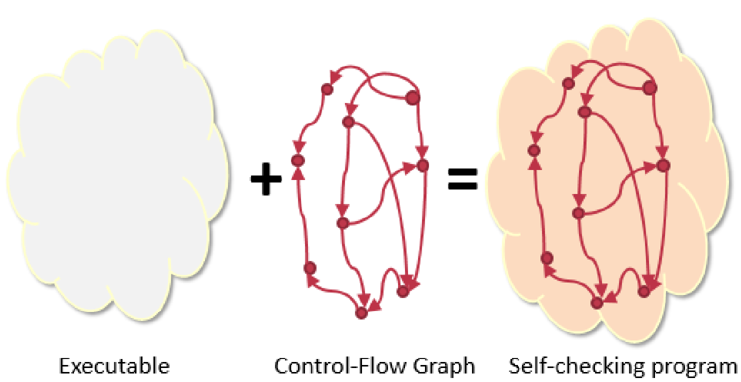
A2-3 **Canary**（金丝雀）对stack overflow敏感

B1-2 ASLR如果是fork出来的生成的汇编还是一样的，只有在刚启动的时候才会随机在汇编代码里插入东西

CFI : Control-Flow Integrity

主要思想：预先确定控制流图control flow graph(CFG)，执行时要按照控制流图（跳到应该跳到的地方）

- Static analysis of source code 静态分析源代码
- Static binary analysis 静态分析二进制 ← **CFI**
- Execution profiling 执行分析
- Explicit specification of security policy 安全规范

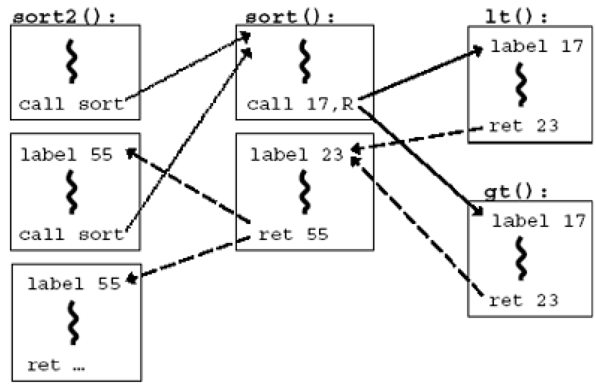


Example:

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



Branch Types

- Direct Branches
  - Direct call
  - Direct jump
- Indirect Branches
  - Return
  - Indirect call
  - Indirect jump

In Apache and its libraries

Types	In Binary	Run-time
Direct call	16.8%	14.5%
Direct jump	74.3%	0.8%
Return	6.3%	16.3%
Indirect call	2.1%	0.2%
Indirect jump	0.5%	68.3%

- has 1 target: 94.7%
- <= 2 targets: 99.3%
- >10 targets: 0.1%

直接跳转在源代码里会比较多，但是实际运行中因为有循环等过程，间接跳转就多了很多，这也和编译器的选择有关

存在问题一：后溯label相同导致逻辑错误

Suppose a call from A goes to C, and a call from B goes to either C, or D. CFI will use the same tag for C and D, but this allows an "invalid" call from A to D.

法一: duplicate code or inline

法二: multiple tags

存在问题二：前溯有多个合法的

CFI will use the same tag for both call sites, but this allows F to return to B after being called from A

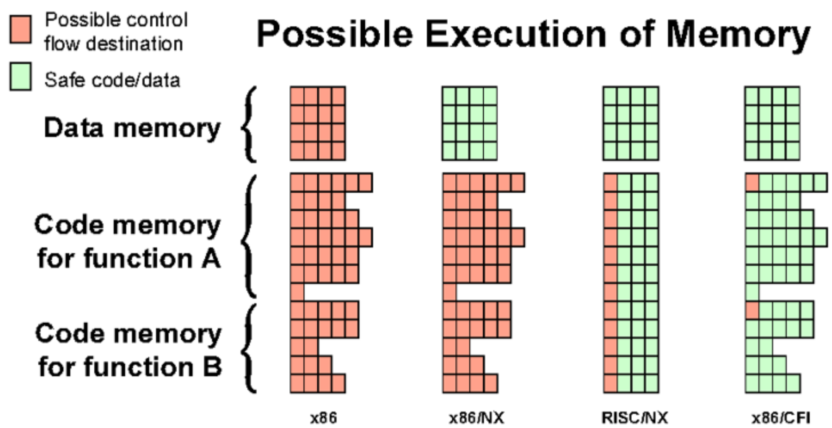
根本问题：ret地址和数据在stack里是混着存的

解决办法：shadow call stack

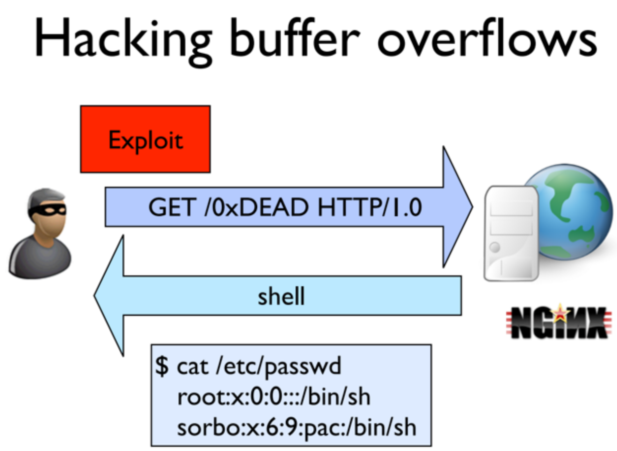
- Maintain another stack, just for return address

- Intel CET to the rescue (not available yet)

结果：允许跳的位置范围变小了很多

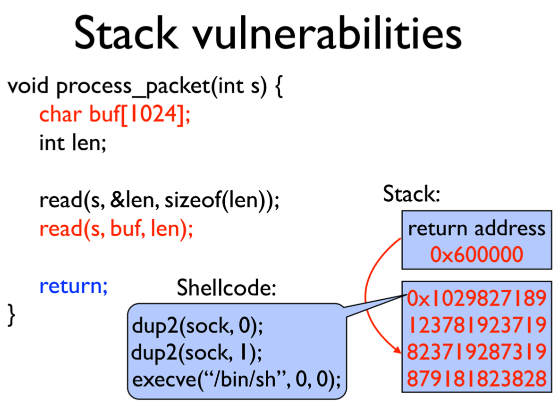


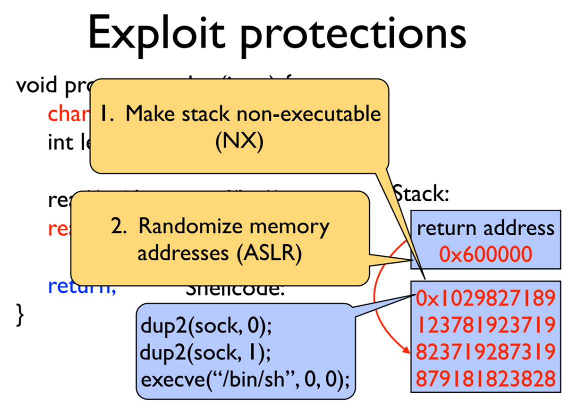
Blind ROP（BROP）



攻击者通过服务器返回是否crash就能把服务器变成shell了，甚至不用知道服务器在跑什么应用

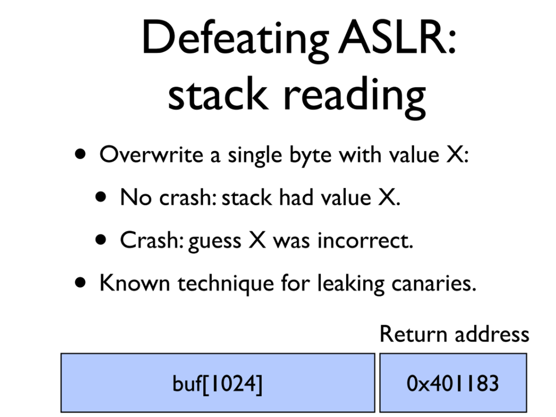
条件：服务器会有buffer overflow，在crash之后会respawn (如nginx / MySQL / Apache / OpenSSH / Samba)

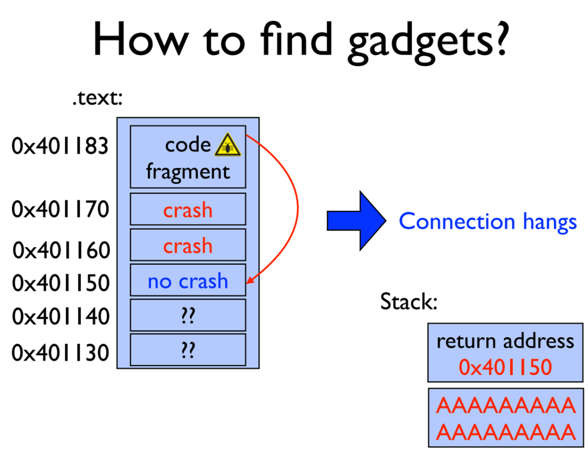


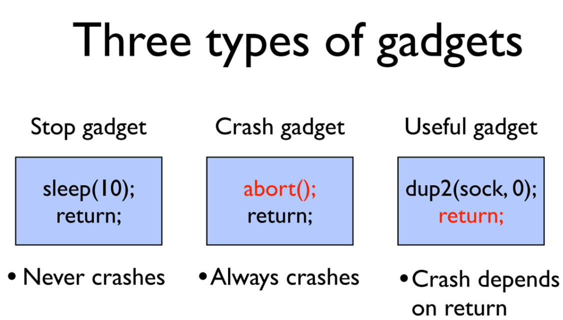


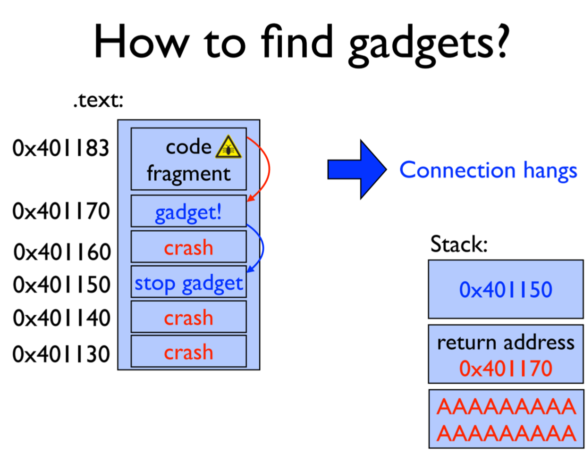
Step1：破解ASLR

一个一个bytes试，找到大概位置；再在大概位置附近找connection hangs，说明前面有return

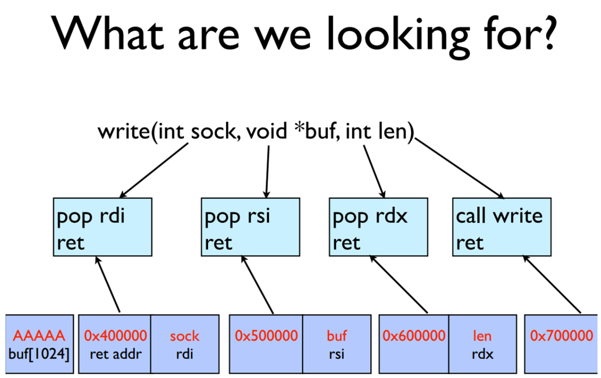


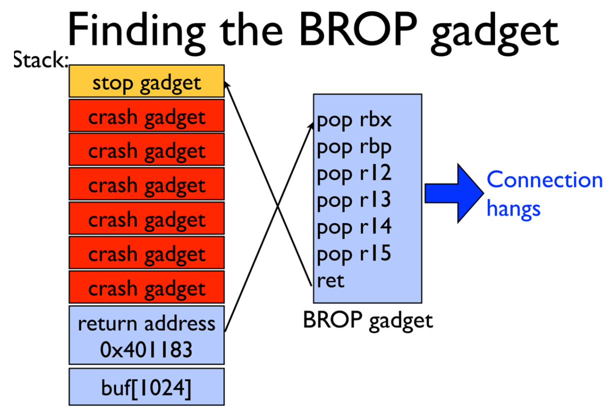






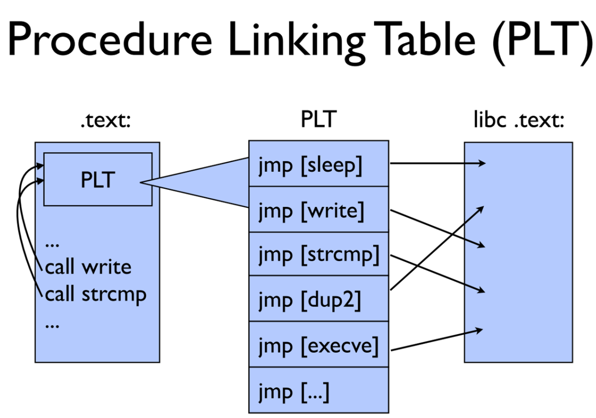
Step2：获得二进制文件的副本（ROP实现write()）





1、前两个在Callee Save中很常见→找Callee Save代码在的位置→Callee Save特点：6个连着的pop

2、pop rdx在strcmp里有（strcmp函数会用rdx存string长度）→ 找strcmp代码在的位置→ 通过PLT找（PLT存着所有jmp命令）→ PLT特点：位于栈中，jmp命令长度为4，+4+4都可以执行 → 找到PLT之后再在里面找strcmp → strcmp特点：arg1和arg2都可读的情况下才能nocrash



Fingerprinting strcmp

arg1	arg2	result
readable	0x0	crash
0x0	readable	crash
readable	readable	nocrash

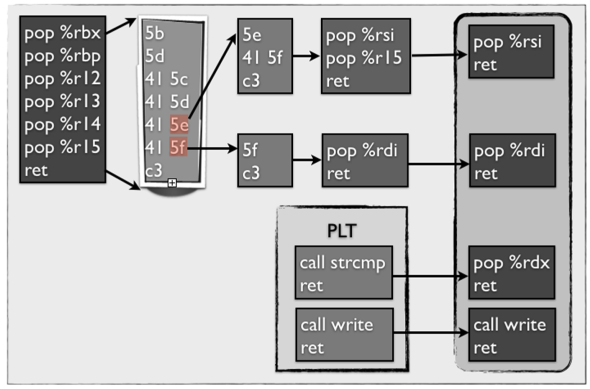
Can now control three arguments:
strcmp sets RDX to length of string

3、找call write：前面dup2(socket, 1)是把标准输出重定向到socket，这里就可以一个一个试PLT里的函数跳转，找到一个会向socket输内容的就可能是write函数

4、最后拼接成Write函数

Finding write

- Try sending data to socket by calling candidate PLT function.
- check if data received on socket.
- chain writes with different FD numbers to find socket. Use multiple connections.

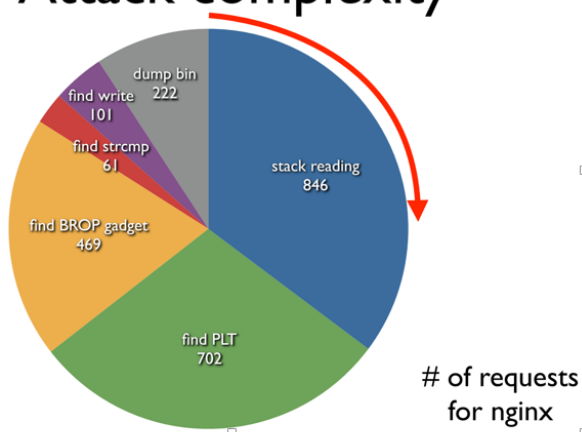


总结：基本步骤+每步复杂度

Launching a shell

1. dump binary from memory to network. Not blind anymore!
2. dump symbol table to find PLT calls.
3. redirect stdin/out to socket:
   - dup2(sock, 0); dup2(sock, 1);
4. read() "/bin/sh" from socket to memory
5. execve("/bin/sh", 0, 0)

Attack complexity



try it 😏

http://www.scs.stanford.edu/brop/