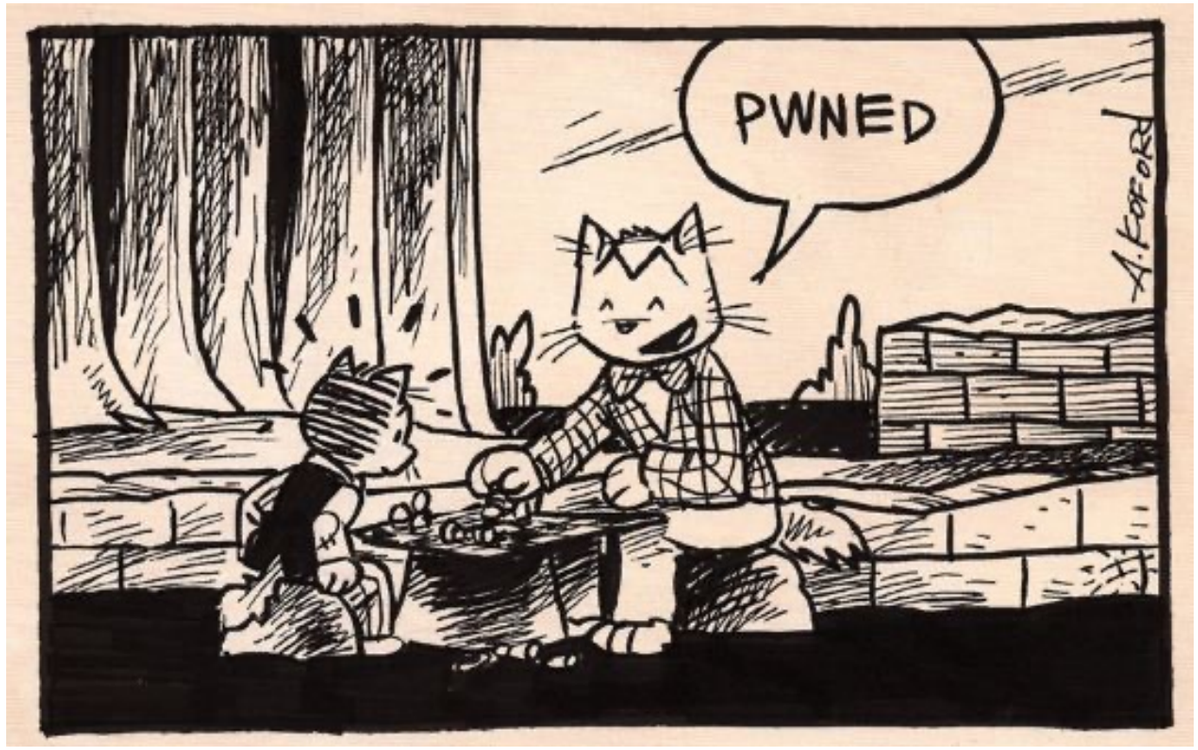
pwn 专题 01 – format string bug



Acknowledgement: Some slides are based on Zhi Wang (FSU)'s slides.

关于 pwn 专题课程的【目前】计划

- 可恶的 pwn 基础没有讲保护机制
- FSB (Format-String Bug)
- ROP (Return-Oriented Programming)
- Heap Basics

蓝色表示会和后续课程《软件安全》重合
红色表示基本后续课程不会涉及

pwntools自带工具 pwn checksec

使用方式:

`pwn checksec / checksec + 目标 binary`

```
lin@ubuntu:~/pwnlab/codes$ checksec helloworld
[*] '/home/lin/pwnlab/codes/helloworld'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

几个保护手段先认识认识

综合型保护手段 ASLR

/proc/sys/kernel/randomize_va_space

everything is a file (again)

- 栈地址随机
  - 演示一下每次 printf 一个栈上地址
  - 对比 gdb
- 堆基地址随机
  - 演示一下每次输出堆上地址
  - 对比 gdb

特别的

- （对于开启 pie 的程序）各个 mapped 的段加载基地址随机

这是为什么 stack overflow example3 的时候助教会特别加上 -pie

format string bug

~~printf~~

- printf 原理
- fsb 原理
- fsb 利用

printf 原理

- 最熟悉莫不过 hello world
  - 但你知道的 hello world 实际上是 puts 完成的

- printf 函数家族 (man 3 printf)

```
SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int dprintf(int fd, const char *format, ...);
       int sprintf(char *str, const char *format, ...);
       int snprintf(char *str, size_t size, const char *format, ...);

       #include <stdarg.h>

       int vprintf(const char *format, va_list ap);
       int vfprintf(FILE *stream, const char *format, va_list ap);
       int vdprintf(int fd, const char *format, va_list ap);
       int vsprintf(char *str, const char *format, va_list ap);
       int vsnprintf(char *str, size_t size, const char *format, va_list ap);
```

缺省参数在 C 语言中的绝佳实践

```
int printf(const char *format, ...);
```

结构

```
char s[] = “AAA”;
int number = 0xdead;
printf(“string is %s, int is %d\n”, s, number);
```

format string 格式化字符串

理论上应该与格式化串对应的参数

变参的实现是通过 va_list 完成的，细节以后软件安全课程会介绍，也可以自行了解

printf参数的行为准备 (32位) 演示

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

int main(void)
{
    int a = 1, b = 2;
    char c = 'A', d = 'B';
    char e[10] = "hello";
    char f[10] = "world";

    printf("%d %d %c %c %s %s\n", a, b, c, d, e, f);

    return 0;
}
```

printf参数的行为准备 (64位) 演示

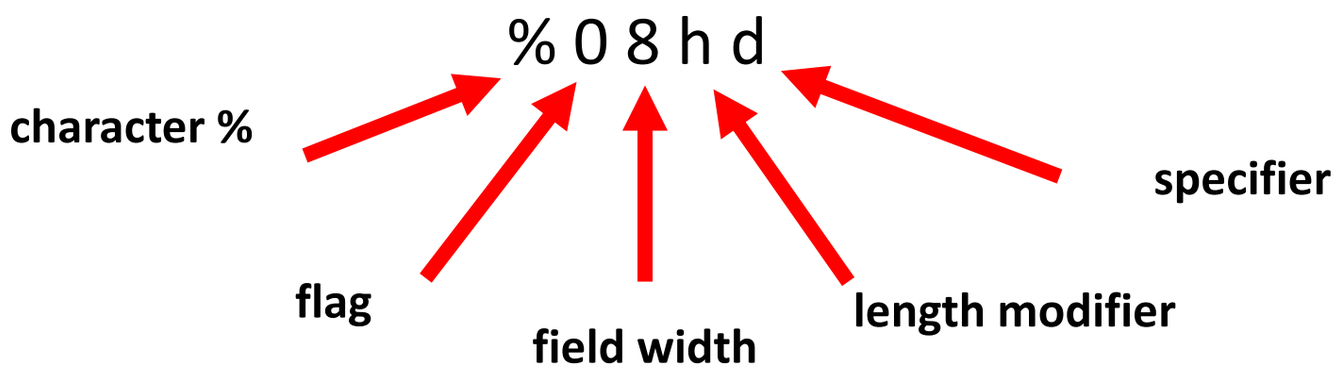
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void)
{
    int a = 1, b = 2;
    char c = 'A', d = 'B';
    char e[10] = "hello";
    char f[10] = "world";

    printf("%d %d %c %c %s %s\n", a, b, c, d, e, f);

    return 0;
}
```

格式化字符串核心: **conversion specification**

```
printf(“string is %s, int is %d\n”, s, number);
```

Refer to manual:

Each **conversion specification** is introduced by **the character %**, and ends with a **conversion specifier**. In between there may be (in this order) zero or more **flags**, an optional **minimum field width**, an **optional precision** and an **optional length modifier**.



多认识一点 specifier

- d, i (decimal int)
- o, u, x, X
  unsigned octal, unsigned decimal, unsigned hexadecimal
- e (科学计数法)
- f, F (单浮点数)
- c (单字符)
- s (字符串)
- p (当 void* pointer 按 %#x, %#lx 输出)

demo: 打印一个超长的 %c

有什么用？一会就知道了

so can we just printf format string

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

int main(void)
{
    char string[] = "Hello world";

    printf(string);

    return 0;
}
```

程序同样打印出了"Hello world"的内容，这与 `printf("%s", string)` 行为一致，且写法上似乎更加简单。

这也是 fsb 的历史成因

what if

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

int main(void)
{
    char string[20];

    scanf("%20s", string);

    printf(string);

    return 0;
}
```

当 programmer 期待 user 仅仅是提供字符串，但是 user (attacker) 却可以使用格式化黑魔法时，FSB 就出现了

sum up

当 printf 的 format string 中的 conversion specification 和后续的变参没有**正确对应**时，format string bug 就会发生

- `printf(“%d %d”, a);` // 少对应了个整数
- `printf(“%s”);` // 字符串指针没有对应
- `printf(“%d”, a, b);` // 勉强也算吧 emm

这个 fsb 的效果应该很直接，即可能第二个 %d 会打印出奇奇怪怪的，程序员无法预料的东西

这漏洞有啥用的?

虽然漏洞原理很简单，但 格式化字符串漏洞 具有非常强的影响力，基本上都可以为攻击者构造 **任意地址读写** 攻击原语

fsb 漏洞利用

- 栈上 format string bug 利用
- 非栈上 format string bug 利用

攻击输入会影响栈上数据，所以攻击者伪造 conversion specifier 时候还可以伪造对应的变参，可控性极强

攻击者的输入无法直接影响栈上内容，即只能布局 conversion specifier 而不能布局变参

栈上 format string bug 利用

目标：

- 任意读
  - 泄露敏感信息；泄露栈地址；泄露堆地址；泄露程序段地址；泄露 libc 地址 blabla
- 任意写
  - 劫持和控制流有关的对象；如 GOT 表等

作业相关，从简到难依次介绍

栈上 fsb – 1: 泄露栈上已有数据 (demo1.c)

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

int main(int argc, char* argv[])
{
    char key_in_stack[32] = "sensitive key";
    char buffer[512] = {0};

    scanf("%s", buffer);
    getchar();
    printf(buffer);

    return 0;
}
```

大力出奇迹

栈上 fsb – 2: $泄露栈上特定数据 (demo1.c)

演示计算偏移过程 （联系 printf 原理的栈距离）

- 特别注意32位和64位在计算上的不同

还可以用 pwntools 将 hex 转化成字符串

栈上 fsb – 3: 通过 %s 泄露非栈上内容 (demo2.c)

- 首先我们知道 `printf(“%s”, ptr);`

解析时将 `ptr (char* pointer)` 作为地址，访问指向内容上的字符串

所以，如果布置格式化串上能复用/伪造 `ptr` 即可获取更强的读能力

复用栈上地址

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

char key_in_global[32] = "verysecure";

int main(int argc, char* argv[])
{
    char *keyptr = key_in_global;
    char buffer[512] = {0};

    scanf("%s", buffer);
    getchar();
    printf(buffer);

    return 0;
}
~
```

demo

Q: 大力出奇迹直接布置一堆 %s 为什么会 crash

noob 流程

1. 首先 readelf 获取 data 段字符串地址
2. 大力 %p 找到地址
3. 配合 $ + %s 完成 leak

完全自行构造地址

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

char key_in_global[32] = "verysecure";

int main(int argc, char* argv[])
{
    // char *keyptr = key_in_global;
    char buffer[512] = {0};

    scanf("%s", buffer);
    getchar();
    printf(buffer);

    return 0;
}
```

demo

noob 流程

1. 首先 readelf 获取 data 段字符串地址
2. 可以用特定字符串 AAAA 等找到 call printf 时栈与存放格式化字符串的距离，用 %p 来完成定位
   当然熟练了后直接计算即可，不用尝试
3. 将特定串的内容换成目标地址，确保 %p 可以输出该地址
4. 将 %p 换成 %s 即可完成 leak

For now you can leak everything !

（当然，你可能需要先知道地址才能 leak，但至少

- 栈上的随机化可以用 fsb leak
- text 段（整个程序的加载地址）可以用 fsb leak

在程序加载地址已知，能够算到 GOT 的地址时

- 可以 leak GOT 的内容从而
  得知 libc 等其他外部程序的加载地址

moreover

- 根据具体的题目的情况，栈上还可能有各种各样有用的信息，
  甚至堆内存地址

do some demos

leak is not enough …….

fsb 最激动人心的事是：不仅有“读”能力，还有“写”能力

%n specifier

The number of characters written so far is stored into the integer pointed to by the corresponding argument.

%n 的正确用法 (demo4.c)

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

int main(int argc, char* argv[])
{
    int print_character_count = 0;
    char some_str[] = "totally agree";

    printf("What a nice day, we are so happy"
           " here to learn advanced pwn class!!"
           " %s ...%n\n", some_str, &print_character_count);

    printf("last printf output %d chars\n", print_character_count);

    return 0;
}
```

但是

- 从 `%s leak` 任意字符串时我们已知，可以在栈上布置一个地址，让 `%s` 将这个地址作为字符串指针并去读出来
- 那么，给定 `%n`，我们也可以在栈上布置任意地址，让 `%n` 将这个地址作为一个 `int*` 指针并去更改内容
- `%hn`，`%hhn` —— 以更小的粒度进行修改

先试试随便改改 (demo5.c)

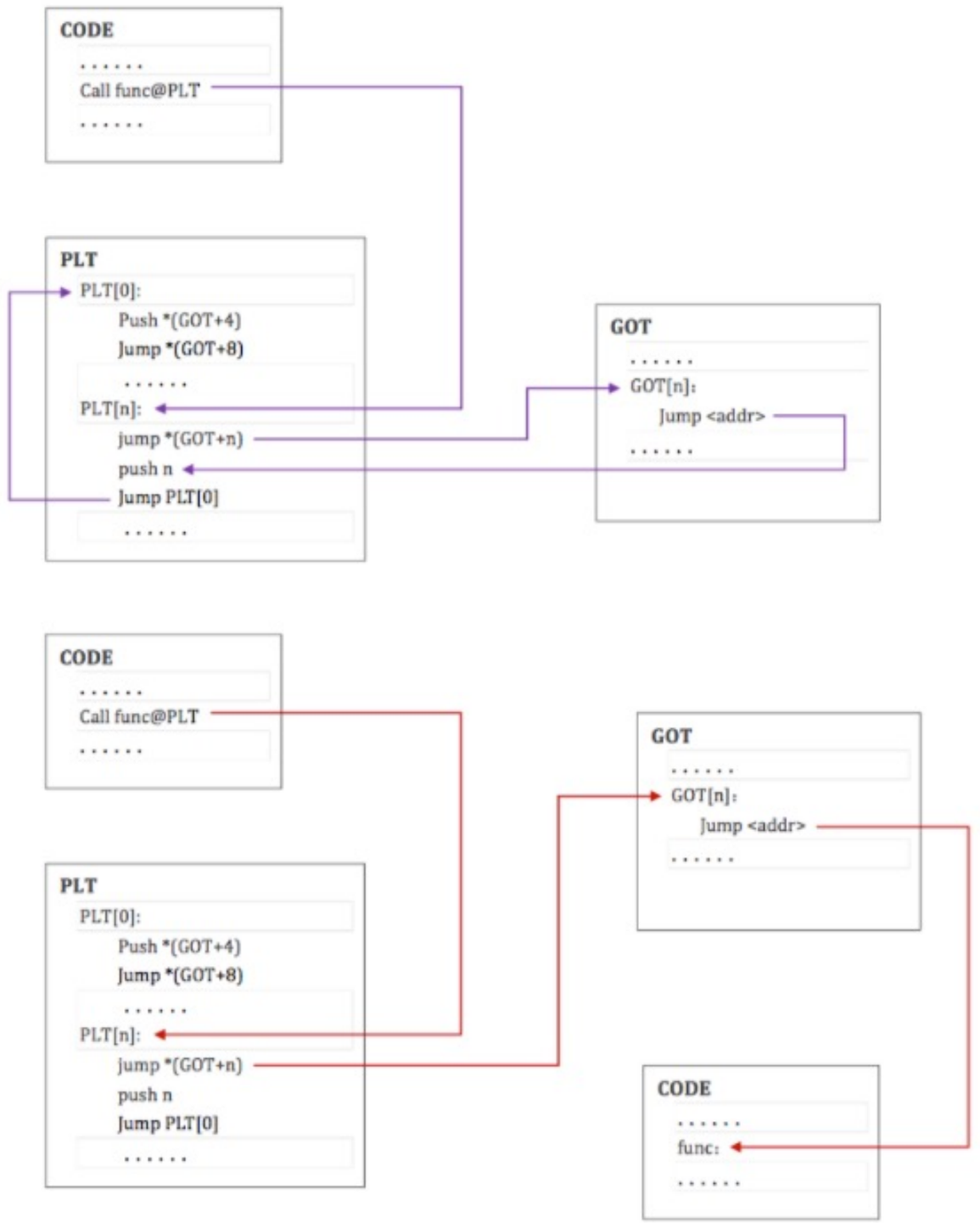
```c
#include <stdio.h>
#include <stdlib.h>

char key_in_global[32] = "verysecure";

int main(int argc, char* argv[])
{
    char buffer[512] = {0};

    printf("before fsb, key: %s\n", key_in_global);

    scanf("%s", buffer);
    getchar();
    printf(buffer);

    printf("after fsb, key: %s\n", key_in_global);

    return 0;
}
```

通过任意写去覆盖控制流相关变量

- GOT 表
- 其他
  - libc中一些有趣的 hook
  - 和程序逻辑本身有关的变量
  - .....

栈上 fsb – 6: 通过 %n 劫持 exit GOT 表到后门

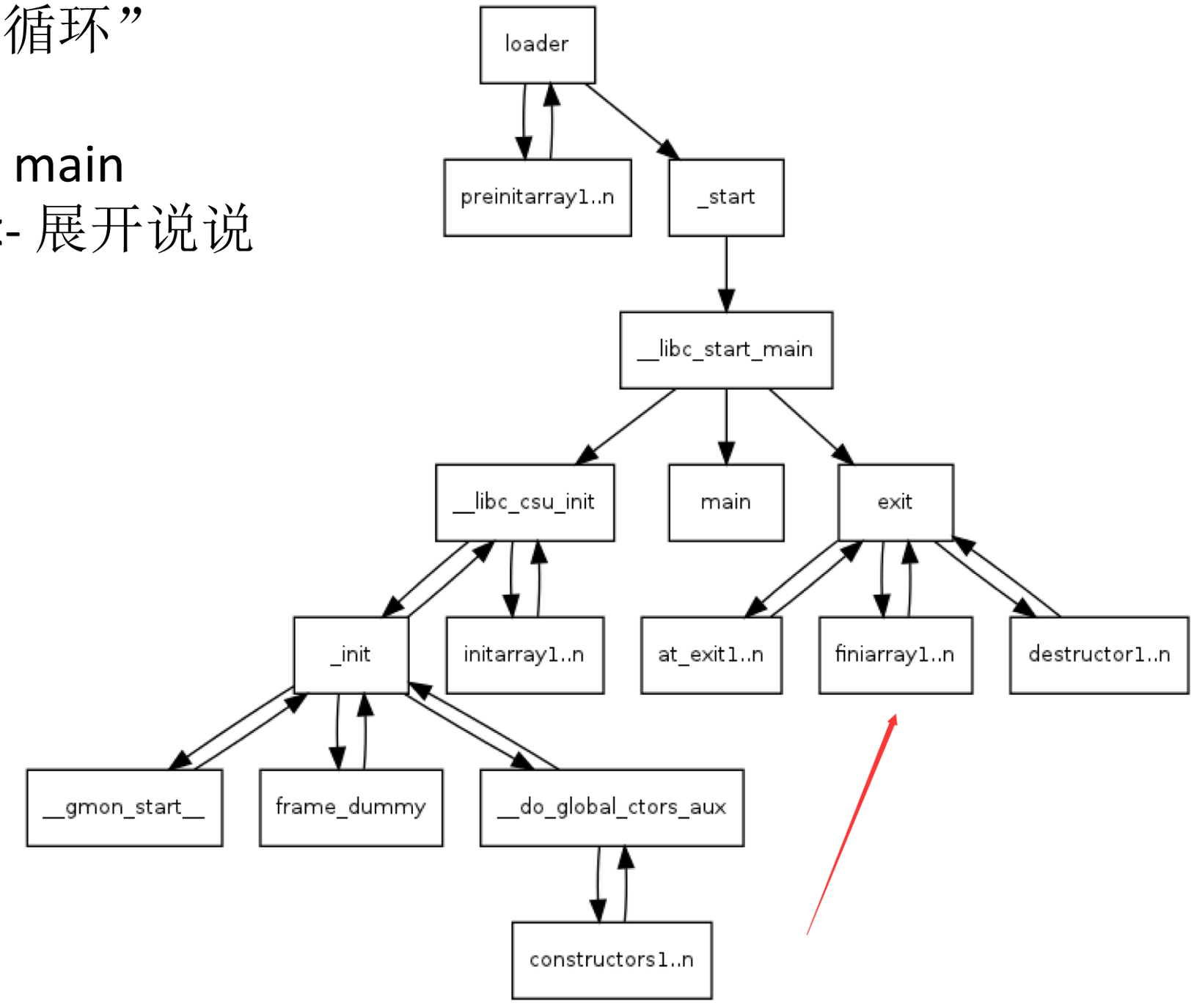
demo6.c



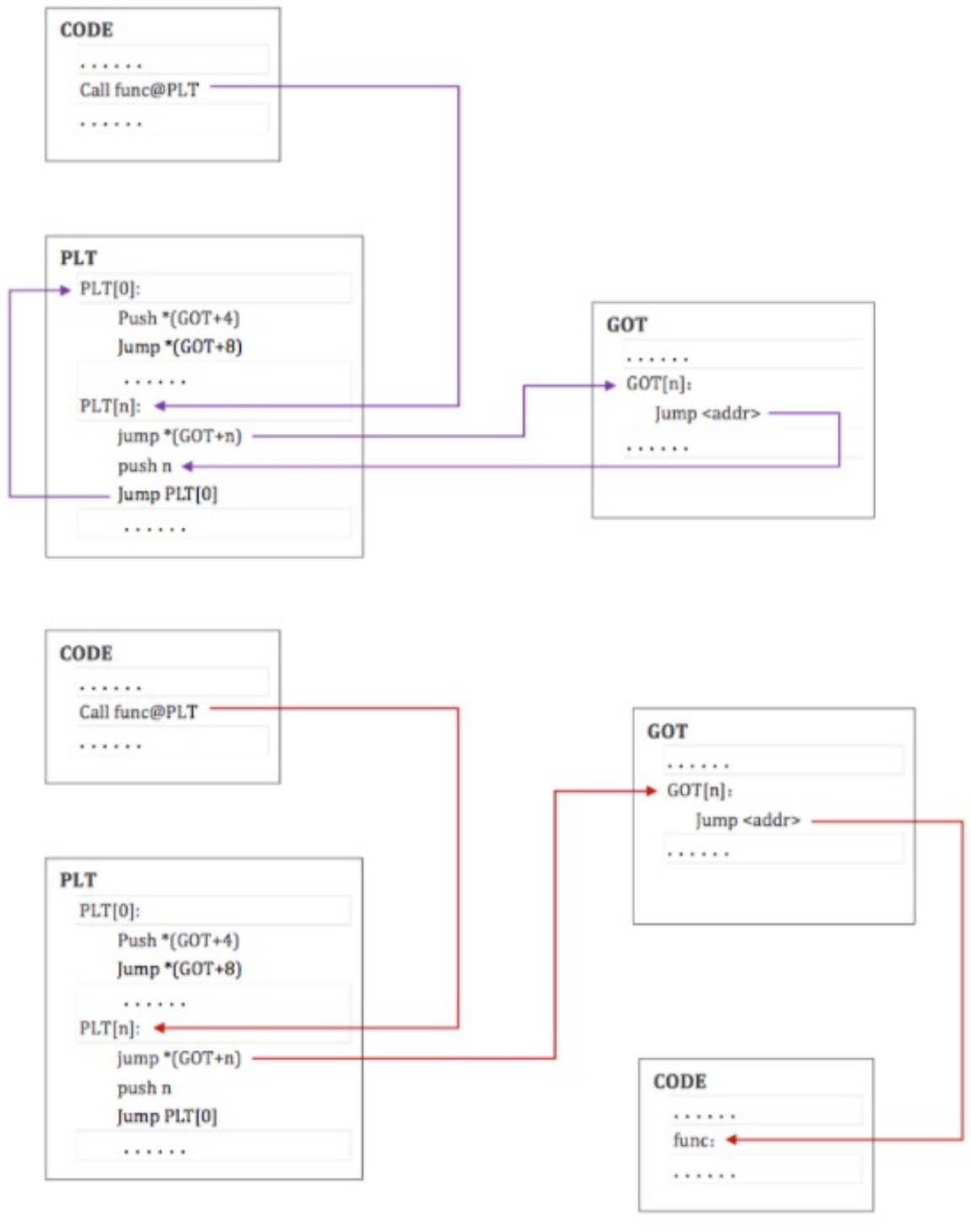
栈上 fsb – 7: 感觉一次 printf 不够怎么办

可以构造程序的 “无限循环”

- 劫持控制流重新回到 main
- 写 __fini_array 对象 <- 展开说说
- 写栈上的返回地址



关于 GOT 保护 Full Relro



非栈上 format string bug 利用

如果用来 printf 的 buffer 不在栈上，事情就变得复杂起来

demo3.c 改

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

char key_in_global[32] = "verysecure";
char buffer[512] = {0};

int main(int argc, char* argv[])
{
    scanf("%s", buffer);
    getchar();
    printf(buffer);

    return 0;
}
```

注意

- 当泄露栈上的内容时 (demo1)
- 或者栈上本来已经有了想要 leak / 覆盖对象的指针时 (demo2)

buffer 在不在栈上没啥影响 ☺

最核心的差异就是：

buffer 不在栈上时，难以自行直接构造出地址，来对应 %s 和 %n

所以至少

- 泄露栈上的内容是依旧可以的
  - 仍然可以 leak 出栈随机
  - 仍然可以通过 leak 出返回地址来知道程序加载基地址
  - 仍然可以根据程序逻辑 leak 出栈上其他有价值的数据

但是还有机会任意读任意写么???

=> not a problem

非栈上 format string bug 利用核心：复用栈上指针

%s, %n 都是对于指针进行读、写操作；所以只要栈上有可控的指针，
就可以实现更强的原语

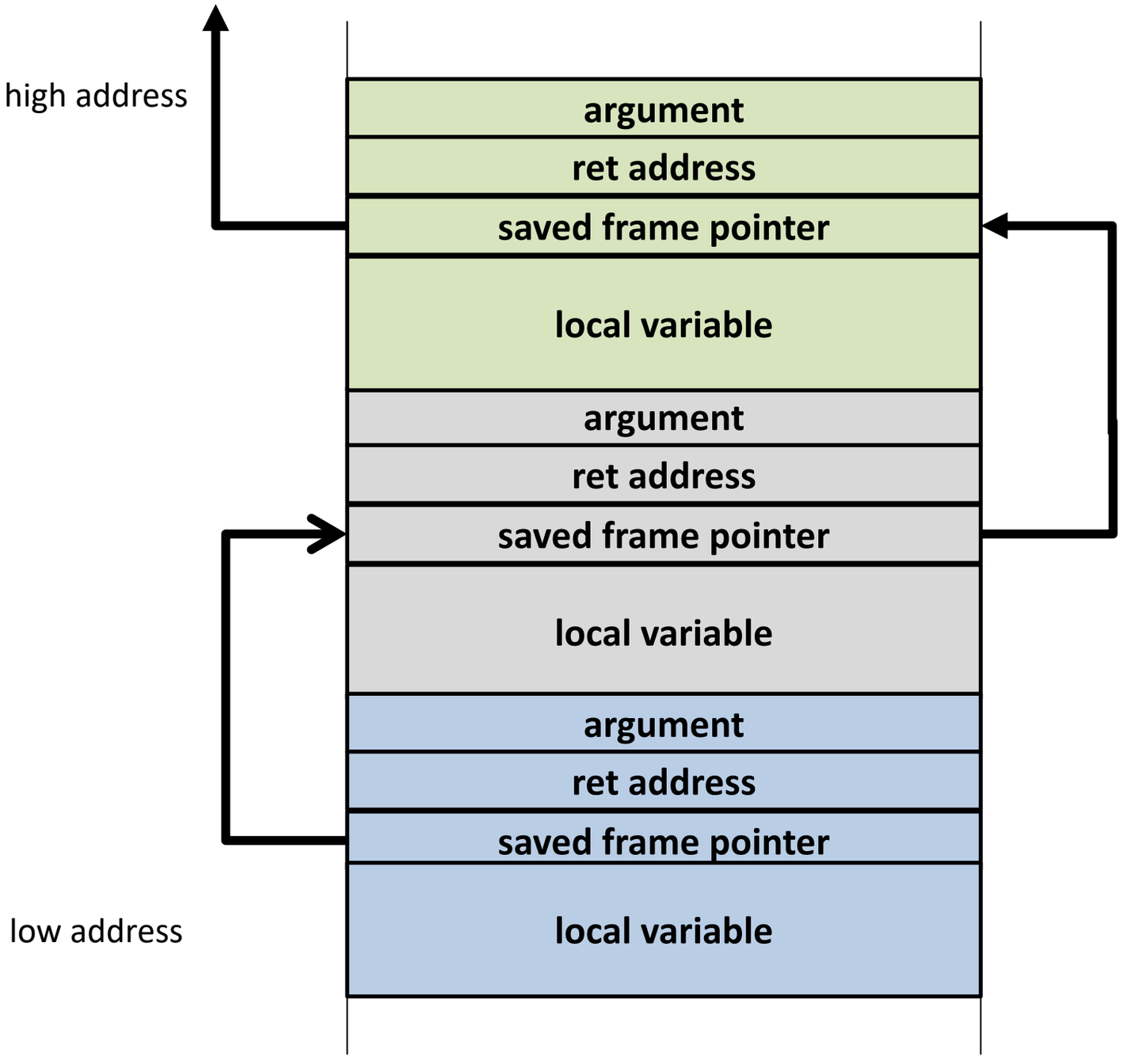
算法与场景：

- 栈上有指针 ptrA，指向另外一个已知位置的 ptrB
- 那么可以借助 %n + ptrA 把 ptrB 覆盖成一个理想指针 ptrC，
- 然后再借助 %s/%n + ptrB(ptrC) 实现任意读写

emmm 栈上真的有这么好的 ptrA, ptrB 么

没有的话不是都是白搭 ☹

frame pointer chain




```
push rbp
mov rbp, rsp

…

push rbp
mov rbp, rsp

…

push rbp
mov rbp, rsp
```

fsb heap demo – demo7.c

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

char string[256];

void backdoor()
{
    printf("Backdoor!\n");
    system("/bin/sh");
}

void vul()
{
    int i;
    for(i = 0; i < 100; i++)
    {
        scanf("%256s", string);
        printf(string);
        fflush(stdout);
    }
}

void wrapper()
{
    printf("This is a wrapper\n");
    vul();
}

int main(void)
{
    wrapper();

    return 0;
}
```

值得一提的轮子：

pwntools: pwnlib.fmtstr

https://docs.pwntools.com/en/stable/fmtstr.html

总结

- printf 结合调用约定的栈结构
- 格式化字符串细节
- 格式化字符串漏洞原理
- 栈上格式化字符串漏洞利用
- 非栈上格式化字符串漏洞利用

作业部分简介