29. ROP and CFI

SECURITY PRINCIPLES

- **Principle of Least Privilege(P7)**
- **Least Trust(P8)**
- **Users Make Mistakes(P9)**
- **Cost of Security(P10)**

return-oriented programing

- **Stack Buffer Overflow(P13)**
  - **Defense: DEP (Data Execution Prevention)(P15)**
- **Attack: Code Reuse (instead of inject) Attack(P16-18)**

> 回头需要重新看CSAPP

- **Defense（P19）**
  - **ASLR（P20）**
  - **Canary（P21）**

BLIND ROP

- 补充：PLT相关：

- lazy-binding的实现
  - 两个数据结构
    1. GOT (GLOBAL OFFSET TABLE)
    2. PLT (PROCEDURE LINKAGE TABLE)
  - 每个调用了共享库的objfile都有自己的GOT和PLT
    - GOT位于数据段
    - PLT位于代码段
  - GOT和PLT协作, 在运行时解析共享库内函数地址
    - PLT: PLT是一个数组, 每个entry是一个16byte代码;
      - 特殊的PLT[0]将跳转到ld-linux.so中
      - 每个被应用程序调用的库函数都有自己的PLT entry; 每个entry负责调用一个具体的函数
      - PLT[1]调用系统启动函数 (__libc_start_main), 初始化执行环境, 调用main函数并处理器返回值
      - PLT[2]开始调用用户代码调用的函数
    - GOT: GOT是一个数组, 每个entry是8-byte的字节地址:
      - 与PLT联合使用, GOT[0]和GOT[1]包含ld-linux.so在解析函数地址时会使用的信息
      - GOT[2]是ld-linux.so的程序入口
      - 其他的条目各对应一个被调用的函数
      - 当函数地址被解析时, 每个entry对应一个PLT entry
      - 初始时, 每个GOT都指向相同函数对应的PLT的下一条指令

- 实例

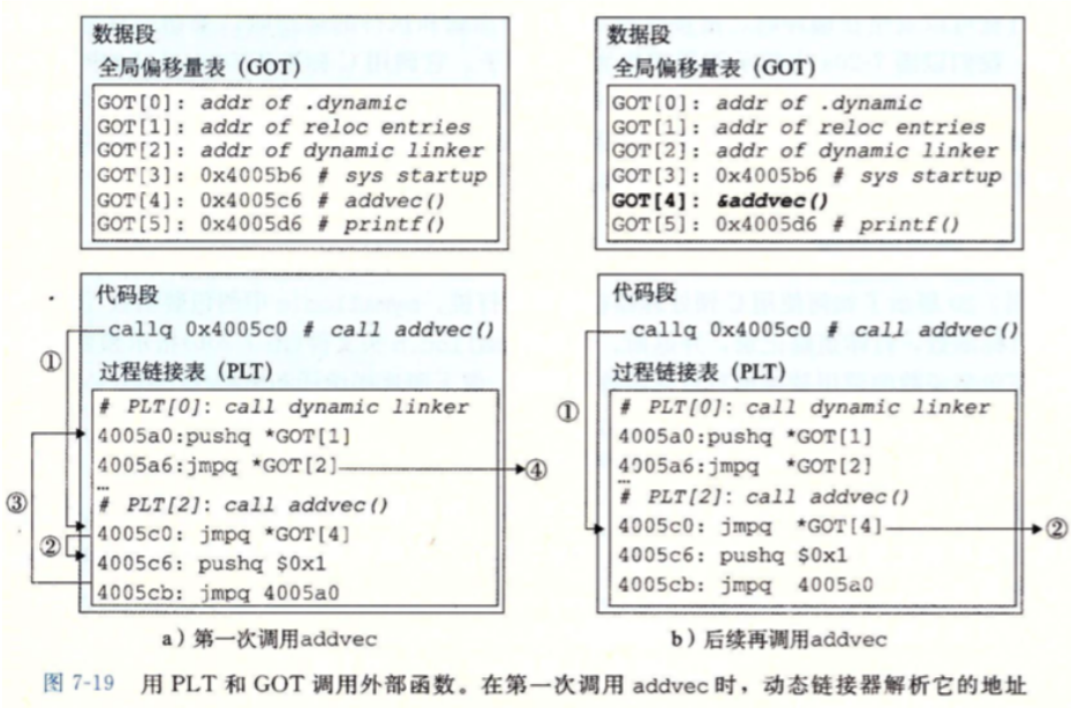

图 7-19 用 PLT 和 GOT 调用外部函数。在第一次调用 addvec 时，动态链接器解析它的地址

- 第一次调用addvec
  1. 程序调用直接进入PLT[2] (addvec的entry)

```
## prog ##
callq 0x4005c0 # call addvec()
## PLT ##
4005c0: jmpq *GOT[4]
```

  2. 通过GOT[4]间接跳转到PLT[2]的下一条指令

```
## GOT ##
GOT[4]: 0x4005c6
```

```
## PLT ##
4005c6: pushq $0x01 # funcID
4005cb: jmpq 4005a0 # to PLT[0]
```

  3. 把addvec的ID (0x01)压栈, 跳转到PLT[0]

```
## PLT ##
4005a0: pushq *GOT[1] # PçLT[0]
4005a6: jmpq *GOT[2]
```

  4. PLT[0]把GOT[1]压栈, 作为ld-linux.so的参数, 跳转到GOT[2]间接跳转到ld-linux.so

```
## GOT ##
GOT[0]: addr of .dynamic
GOT[1]: addr of reloac entry
GOT[2]: addr of dynamic linker
```

  5. ld-linux.so开始执行, 通过当前stack上的ID和GOT[1]作为参数来解析函数运行时位置, 并用这个位置重写GOT[4], 完成GOT填写, 并返回PC到addvec
- 后续调用addvec
  1. 跳转到同样的PLT[2]
  2. PLT[2]跳转到GOT[4] (对应的entry)
  3. GOT[4]直接转移PC到addvec的运行地址

具体操作看回放