

2018-2019学年春季学期

计算机体系结构安全
Computer Architecture Security

授课团队：侯锐、宋威、陈李维
助 教：李沛南

计算机体系结构安全

Computer Architecture Security

[第9次课] 代码复用攻击及防御

授课教师：宋威

授课时间：4月23日

内容概要

○代码复用攻击

- 产生的背景

- 攻击的整体思路

- 攻击的细节：ROP, COP, JOP

○防御方法

- 地址随机化ASLR

- 代码隐藏/加密

- 控制流完整性CFI

- 代码指针完整性CPI

内容概要

○代码复用攻击

- 产生的背景
- 攻击的整体思路
- 攻击的细节：ROP, COP, JOP

○防御方法

- 地址随机化ASLR
- 代码隐藏/加密
- 控制流完整性CFI
- 代码指针完整性CPI

○直接代码注入攻击

攻击者通过一定的方法将恶意代码(shellcode)注入到用户进程，并通过溢出等手段改变程序的正常的控制流，使程序执行恶意代码从而实现某种攻击目的。

○必要条件

- 构造shellcode
- 将shellcode注入可写内存空间
- 劫持控制流执行shellcode

○防御方式

- 可写不可执行

○攻击新思路

既然不能注入代码，可否使用已经载入内存的代码

○攻击者的资源

○可执行文件可以被反汇编

○已知的运行时内存结构（代码、堆、栈、等等）

○大量的内存错误（任意内存位置读写）

○大量的已加载代码（标准库、第三方库、软件本身）

○代码页可读

○攻击者还缺乏的攻击要素

○绑架内存中的现有代码完成shellcode的功能！

○代码复用攻击

- 复用计算机系统内部已有的代码，将已有的代码重新组合，形成具有一定功能的恶意代码，从而对计算机系统攻击。

○攻击的要素

○功能代码

- 功能代码的发现
- 功能代码的完备性

○重新组合

- 功能代码执行的可控性

Firefox的库依赖

```
~$ ldd /usr/lib/firefox/firefox
linux-vdso.so.1 => (0x00007ffeb5bca000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0
(0x00007faeb58de000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007faeb56da000)
libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6
(0x00007faeb5358000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007faeb504f000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007faeb4e39000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007faeb4a6f000)
/lib64/ld-linux-x86-64.so.2 (0x00007faeb5d31000)
```

```
~$ objdump -d /lib/x86_64-linux-gnu/libc.so.6 | less
/lib/x86_64-linux-gnu/libc.so.6:      file format elf64-x86-64
```

Disassembly of section .plt:

```
000000000001f7c0 <*ABS*+0x8f14b@plt-0x10>:
  1f7c0:      ff 35 42 48 3a 00      pushq  0x3a4842(%rip)      # 3c4008
<_IO_file_jumps@@GLIBC_2.2.5+0x928>
  1f7c6:      ff 25 44 48 3a 00      jmpq   *0x3a4844(%rip)    # 3c4010
<_IO_file_jumps@@GLIBC_2.2.5+0x930>
  1f7cc:      0f 1f 40 00           nopl   0x0(%rax)

000000000001f7d0 <*ABS*+0x8f14b@plt>:
  1f7d0:      ff 25 42 48 3a 00      jmpq   *0x3a4842(%rip)    # 3c4018
<_IO_file_jumps@@GLIBC_2.2.5+0x938>
  1f7d6:      68 0c 00 00 00         pushq  $0xc
  1f7db:      e9 e0 ff ff ff        jmpq   1f7c0
<_h_errno@@GLIBC_PRIVATE+0x1f75c>
```

寻找希望执行的指令

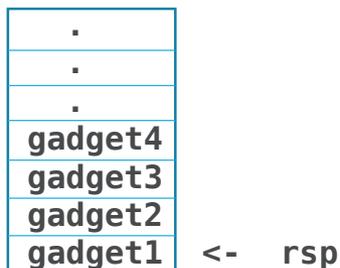
```
2 208a0:    31 ff          xor    %edi,%edi
3 208a2:    89 d0          mov    %edx,%eax
5 208a4:    0f 05          syscall
208a6:    eb f8          jmp   208a0 <__libc_start_main@@GLIBC_2.2.5+0x160>
208a8:    48 8b 44 24 08  mov   0x8(%rsp),%rax
208ad:    48 8b 15 bc 35 3a 00  mov   0x3a35bc(%rip),%rdi
208b4:    48 8d 3d 94 bd 16 00  lea   0x16bd94(%rip),%rdi
208bb:    48 8b 30          mov   (%rax),%rsi
208be:    31 c0          xor   %eax,%eax
208c0:    ff 92 18 01 00 00  callq *0x118(%rdx)
208c6:    e9 13 ff ff ff    jmpq  207de <__libc_start_main@@GLIBC_2.2.5+0x9e>
208cb:    4c 8b a8 68 01 00 00  mov   0x168(%rax),%r13
208d2:    48 8b 05 27 35 3a 00  mov   0x3a3527(%rip),%rax
208d9:    45 31 e4          xor   %r12d,%r12d
1 208dc:    48 8b 28          mov   (%rax),%rbp
208df:    49 8b 45 18          mov   0x18(%r13),%rax
208e3:    48 85 c0          test  %rax,%rax
208e6:    74 10          je    208f8 <__libc_start_main@@GLIBC_2.2.5+0x1b8>
208e8:    44 89 e7          mov   %r12d,%edi
208eb:    48 83 c7 47          add   $0x47,%rdi
208ef:    48 c1 e7 04          shl   $0x4,%rdi
208f3:    48 01 ef          add   %rbp,%rdi
4 208f6:    ff d0          callq %rax
208f8:    41 83 c4 01          add   $0x1,%r12d
208fc:    4d 8b 6d 40          mov   0x40(%r13),%r13
20900:    45 39 e6          cmp   %r12d,%r14d
20903:    75 da          jne  208df <__libc_start_main@@GLIBC_2.2.5+0x19f>
20905:    e9 cc fe ff ff    jmpq  207d6 <__libc_start_main@@GLIBC_2.2.5+0x96>
2090a:    66 0f 1f 44 00 00  nopw  0x0(%rax,%rax,1)
20910:    48 8d 35 c9 33 17 00  lea   0x1733c9(%rip),%rsi
20917:    ba 82 02 00 00    mov   $0x282,%edx
2091c:    bf 01 00 00 00    mov   $0x1,%edi
```

构造指令执行序列-配件 (Gadget)

```
mov    (%rax),%rbp
xor    %edi,%edi
mov    %edx,%eax
callq  *%rax
syscall
```

```
retq:
rip = (rsp)
rsp = rsp + 8
```

On stack:



```
gadget2:
xor    %edi,%edi
mov    %edx,%eax
retq
```

```
gadget1:
mov    (%rax),%rbp
retq
```

```
gadget4:
syscall
retq
```

```
gadget3:
callq  *%rax
retq
```

Return Oriented Programming (ROP)

配件 (Gadget)

○寻找可用的配件 (Gadget)

- 具有特定功能的代码片段：攻击功能
- 以间接跳转结尾：跳转目标劫持

○配件的类型：按跳转类型分

○Return (ROP)

- 以RET为结尾，通过控制栈来控制跳转目标

○Call (COP)

- 以函数指针调用为结尾，通过控制函数指针控制跳转目标
- `pop %rax; call *%rax;`

○Jump (JOP)

- 以间接跳转为结尾，直接控制间接跳转目标
- `pop %rax; jump *%rax;`

配件执行和正常程序的类比

○正常程序

○资源

Reg, Mem

○计算

Reg \leftarrow Reg op Reg

○PC

PC \leftarrow PC + 4

○跳转

PC \leftarrow PC + imm

○LD/ST

Reg \leftarrow [mem]

[mem] \leftarrow Reg

○配件执行

○资源

Data, Mem

○计算

Data \leftarrow *Data* op *Data*

○PC

Data \leftarrow *Data* + 4

○跳转

Data \leftarrow *Data* + imm

○LD/ST

Data \leftarrow [mem]

[mem] \leftarrow *Data*

Data:

攻击者控制的数据区，
栈或者其他。

○配件

任何对内存数据做操作同时控制流跳转受内存影响的代码片段都可以成为配件

○函数的一部分

○函数

○多个函数

○只读数据（非代码）

○配件的编织：在内存中构造配件链条

通过控制一块内存，预先配置好配件的操作数和配件间跳转的控制流劫持数据。

○栈

○表（JOP的dispatcher table）

○GOT/PLT

○VTable

- 配件的寻找
 - 静态分析
 - 动态分析
- 在内存中构造配件链条
 - 直接构造
 - 任意内存写
- 执行第一个配件
 - 劫持控制流的内存错误
 - 溢出
 - 修改GOT/PLT/Vtable

- ret2libc: 复用C标准库
 - 几乎所有的程序都依赖于C标准库
 - 直接利用C标准库的代码
 - x86时代, 函数调用都是用栈传递参数, 容易攻击, 大量配件
- ROP: Return Oriented Programming
 - 利用栈和结尾于ret指令配件攻击
 - 栈比较好改写
 - 大量存在的以ret结尾的指令片段
- COP/JOP/COOP
 - 利用更广泛的数据区和所有的间接跳转指令

代码复用攻击的特点

- 非直接代码注入
 - 绕过可写不可执行
- 配件寻找
 - 事先二进制代码分析：静态/动态
- 控制可写数据区
 - 直接控制或内存错误
- 初始控制流劫持
 - 内存错误
- 运行时控制流不断劫持
 - 函数返回不遵守函数调用规则
 - 函数调用不遵守函数调用规则
 - 跳转频繁跨越函数边界

内容概要

○代码复用攻击

- 产生的背景

- 攻击的整体思路

- 攻击的细节：ROP, COP, JOP

○防御方法

- 地址随机化ASLR

- 代码隐藏/加密

- 控制流完整性CFI

- 代码指针完整性CPI

代码复用攻击的特点

- 非直接代码注入

- 绕过可写不可执行

- 配件寻找

- 事先二进制代码分析：静态/动态

地址随机化ASLR



- 控制可写数据区

- 直接控制或内存错误

- 初始控制流劫持

- 内存错误

- 运行时控制流不断劫持

- 函数返回不遵守函数调用规则
 - 函数调用不遵守函数调用规则
 - 跳转频繁跨越函数边界

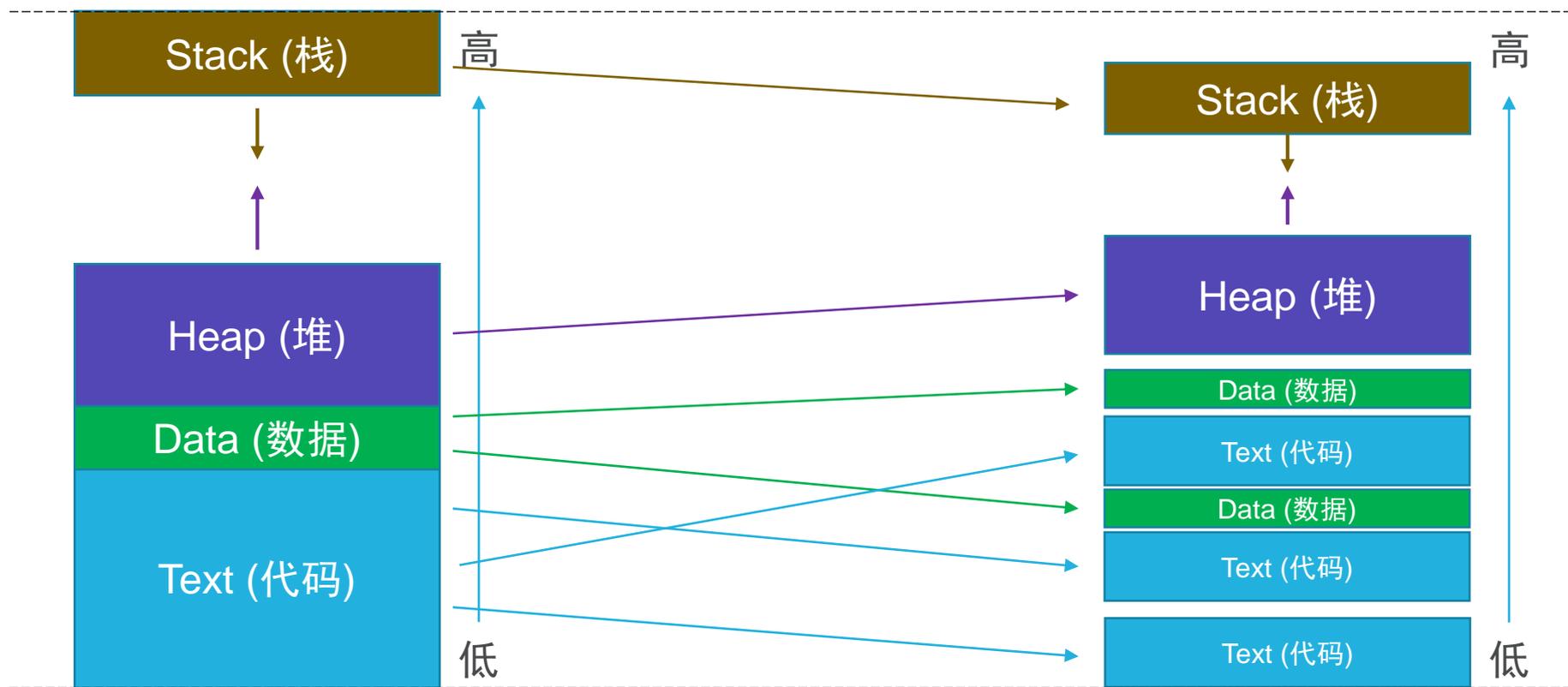
地址随机化 (ASLR)

○地址随机化

○Address Space Layout Randomization (ASLR)

○Position Independent Execution (PIE)

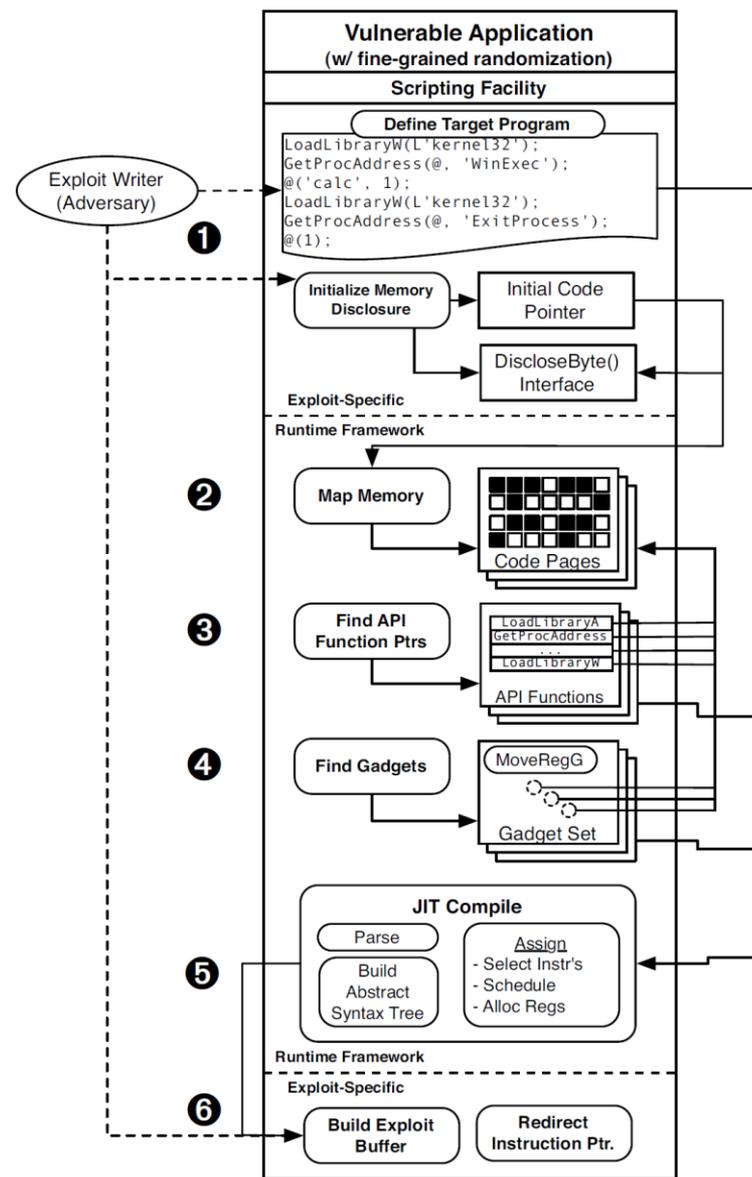
○绝对地址寻址 -> PC-related 寻址



JIT-ROP 绕过ASLR

1. 找到一个代码指针
2. 通过偏移量推测找下一个代码页
3. 通过代码反汇编找函数指针
4. 通过代码反汇编找配件
5. 运行时配件编译
6. 装载配件, 执行配件

只要代码页可读, 运行时也可以分析。



○ 防御

- 静态分析时的地址不可直接用于攻击
- 对代码执行效率基本没有损伤
- 大量部署于所有的常见操作系统

○ 攻击

- 引入的随机量（熵）不够
- 静态/动态分析仍然可行
 - 寻找不变量
 - 地址基址变化但是偏移不变
 - 程序地址随机化在执行过程中不变
 - 共享库的地址随机化持续时间更久
 - 直接运行时分析

○ 加强版：细粒度随机化

- 页内随机化和运行时动态随机化
- 执行代价增大，防御力仍然不够

○ 防御机理

- 动态二级制分析需要读取代码页（可读可执行）
- 攻击者通过分析代码绕过随机化
- 但是代码只需可执行
- 剥离代码和只读数据，然后设定代码不可执行
- 攻击者不能直接读取代码

○ 实际情况

- 理论上可行，实际复杂，需要大量修改编译器。
- 攻击者是否可以通过其他不变量绕过？
- 仍然是一个好防御方案。

○ 防御机理

- 动态二级制分析需要读取代码页（可读可执行）
- 攻击者通过分析代码绕过随机化
- 加密代码页
- 当代码页被读入一级代码缓存时解密
- 直接读取代码页为加密后的乱码

○ 实际情况

- 理论上可行，实际复杂，需要大量修改编译器。
- 加密和解密的复杂度。
- 密钥的分发和保护。
- 仍然是一个好防御方案。

代码复用攻击的特点

- 非直接代码注入
 - 绕过可写不可执行
- 配件寻找
 - 事先二进制代码分析：静态/动态
- 控制可写数据区
 - 直接控制或内存错误
- 初始控制流劫持
 - 内存错误
- 运行时控制流不断劫持
 - 函数返回不遵守函数调用规则
 - 函数调用不遵守函数调用规则
 - 跳转频繁跨越函数边界

控制流完整性



○ 防御机理

○ 假设：正常执行的代码和配件执行存在区别

- 函数返回不遵守函数调用规则
- 函数调用不遵守函数调用规则
- 跳转频繁跨越函数边界

○ 通过硬件或者软件检查代码执行

○ 控制流的分类

○ 后向控制流

- 函数返回

○ 前向控制流

- 函数调用
- 其他间接跳转：switch?

○ 具体防御

○ 粗粒度或细粒度

○ 硬件或软件

○粗粒度的控制流完整性

- 函数调用跳转到函数入口点
- 函数返回跳到call的下一条指令
- 间接跳转的目标代码块不能太短

○实现

○利用现有硬件

- LBR (Last Branch Register)
- BTB (Branch Target Buffer)
- 在关键位置 (syscall) 检查完整性

○使用特殊指令和插桩

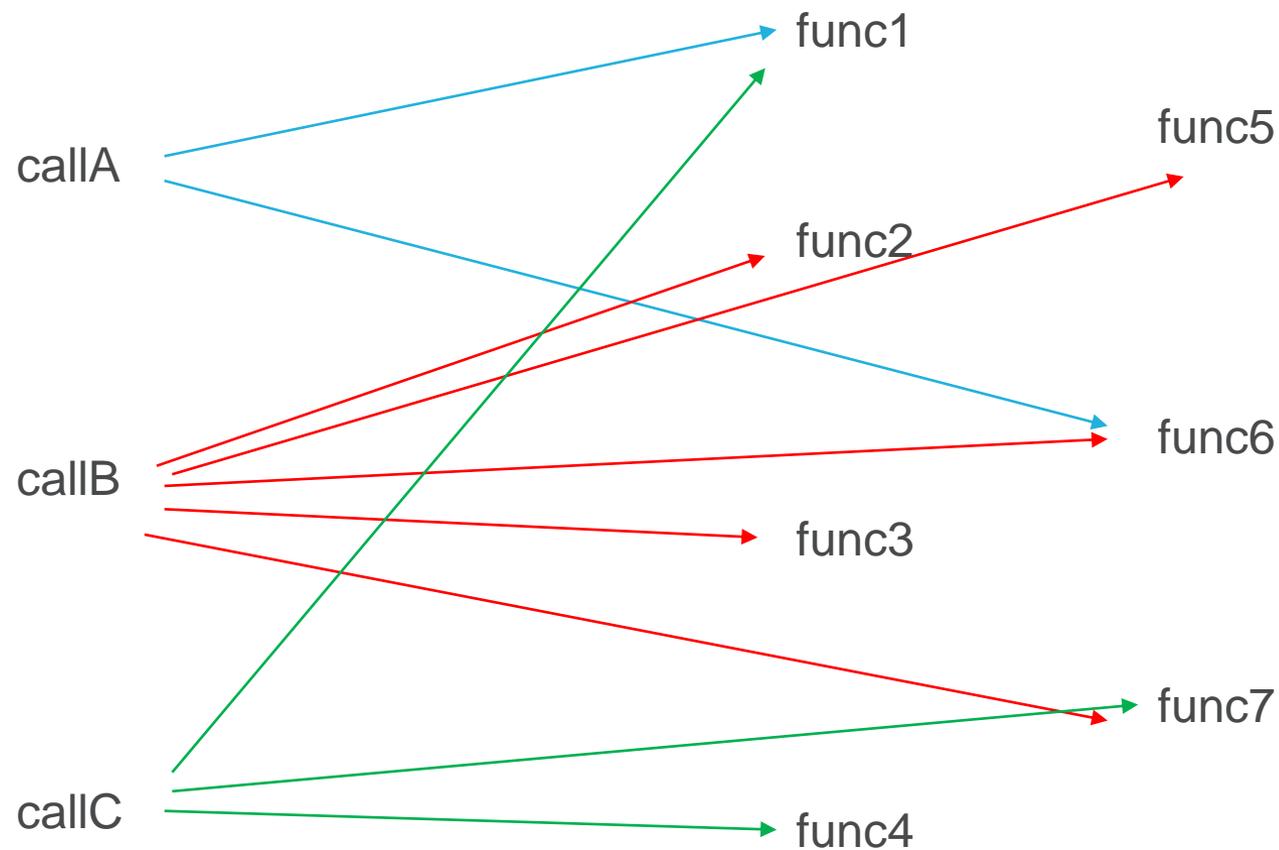
- Intel CET

○可以被绕过!

- 使用长插件
- 使用函数当插件

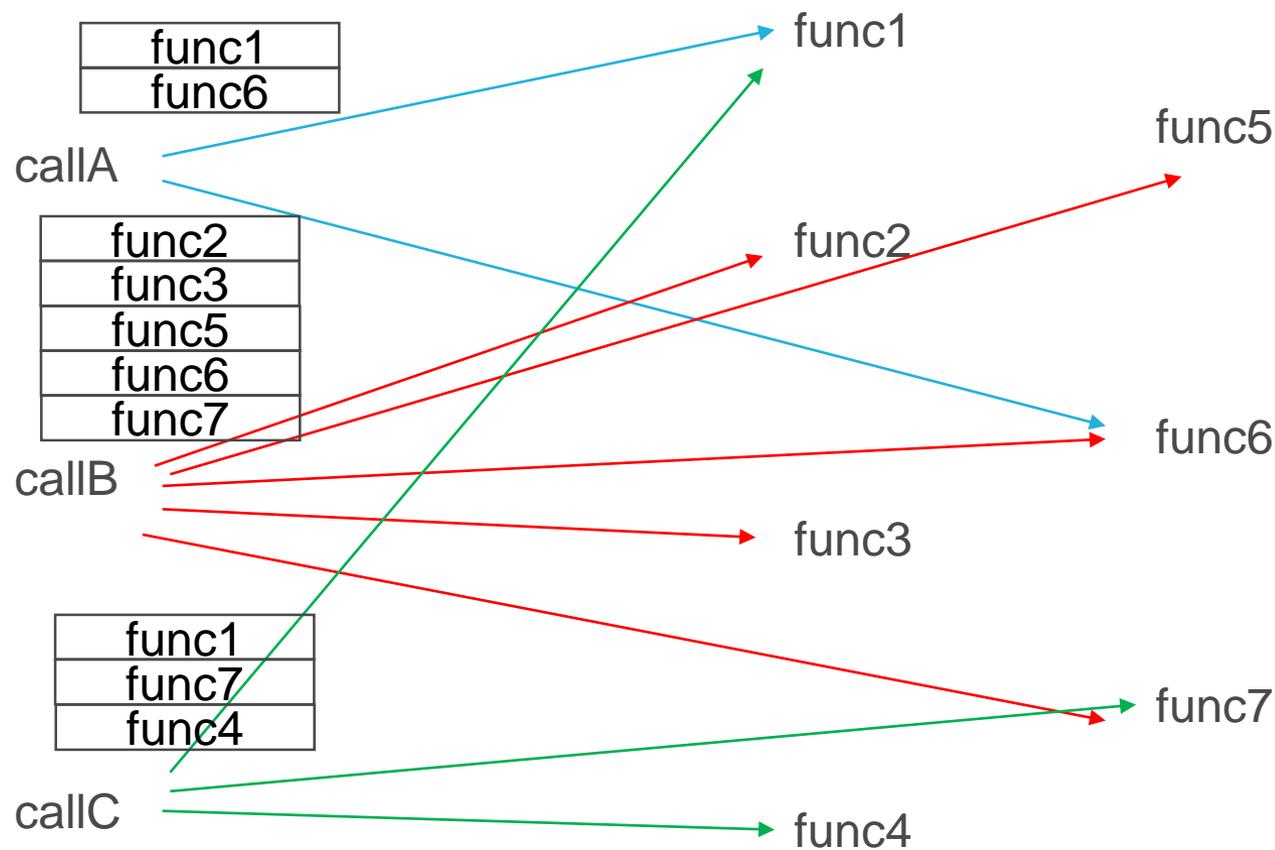
- **细粒度的反向控制流完整性**
 - 函数调用一定会返回，严格配对
 - 利用影子栈 (shadow stack) 记录所有的函数调用
 - 在函数返回时检查是否匹配影子栈数据
- **实现**
 - **软件插桩**
 - 在函数调用处记录函数调用
 - 在函数返回时检查函数配对
 - **系统分配并隐藏影子栈**
 - 影子栈不可以被攻击者改写
 - 相信操作系统
- **可以被绕过!**
 - 只针对ROP攻击
 - 对JOP, COP, COOP无效

正向控制流完整性-粗粒度-软件静态分析



静态分析、函数类型、虚函数

正向控制流完整性-粗粒度-软件静态分析



静态分析、函数类型、虚函数

- **基于静态分析的正向控制流完整性**
 - 利用编译器的CFG分析，建立函数调用关系表
 - 将表存于代码的只读区域内，防止被修改
 - 利用插桩，在函数调用时检查跳转目标是否在表内
 - LLVM已经部署

- **可以被绕过!**
 - 仅利用表内的目标仍然可以完成攻击
 - 表内目标的范围仍然过大
 - 静态分析不精确
 - 万能指针 (void **)

- **基于静态分析的粒度降低**
 - 函数调用参数的个数
 - 函数调用参数的类型
 - 函数返回值的类型
- **基于运行时的粒度降低**
 - 运行时代码分析和目标表的修改
 - 将temporal信息引入控制流检查
 - 还没什么好方案
- **仍然可以被绕过!**
 - 函数类型丢失
 - C++函数名丢失
 - 即使加入调试信息, 数据类型丢失, 通用类型
 - 默认参数
 - 变长参数列表

代码复用攻击的特点

- 非直接代码注入
 - 绕过可写不可执行
 - 配件寻找
 - 事先二进制代码分析：静态/动态
 - 控制可写数据区
 - 直接控制或内存错误
 - **初始控制流劫持**
 - **内存错误**
 - **运行时控制流不断劫持**
 - 函数返回不遵守函数调用规则
 - 函数调用不遵守函数调用规则
 - 跳转频繁跨越函数边界
- 代码指针完整性
- 

○基于加密的代码指针完整性

- CCFI: Cryptographically Enforced Control Flow Integrity

- 假设对代码指针的改写都发生在可写内存

- 当代码指针被保存于可写内存时

 - 生成一个密钥

 - $(p, *p, pid, \sim\text{version/time}) \rightarrow \text{MAC}$

 - 将MAC保存在固定位置

- 当代码指针被读回时

 - 检查MAC, 如果不符, 报错!

○分析

- 是一种很强的保护, 可用于正向和反向

- 加密的复杂度

- 重演攻击

 - 构造相同 $(p, *p, pid, \sim\text{version/time}) \rightarrow \text{MAC}$

- 破解加密算法

- **基于安全存储的代码指针完整性**
 - **CPI: Code Pointer Integrity**
 - **加密比较慢**
 - **直接通过安全存储区将指针信息保存**
- **分析**
 - **是一种很强的保护，可用于正向和反向**
 - **比CCFI的性能好一些，但是内存带宽变大**
 - **重演攻击**
 - **构造相同 (p, *p, pid, ~version/time)**
 - **攻击安全区**

Capability

○ Capability

- **每一个指针（数据/代码）都有自己的权限（Capability）**
- **宽指针（fat pointer） / 指针标签**
- **利用宽指针保存指针的capability信息**
 - 内容 content
 - 范围 spatial boundary
 - 时间 version (validity/temporal boundary)
 - 权限 更细节的capability

○ 分析

- **同意数据完整性和代码指针完整性**
- **基于规则的完整性检查**
- **实现代价？**
 - 软件方案：Cambridge Cherry
 - 硬件方案：MIT PUMP

- 代码复用攻击
 - 不直接代码注入
 - 复用已装载入内存的代码
 - 事前分析
 - 配件装载
 - 利用配件完成基本功能
- 特点
 - 破坏控制流完整性
- 防御
 - 内存随机化/加密/隐藏
 - 控制流完整性
 - 代码指针完整性

- 代码复用攻击的根源?
- 控制流劫持的根源?
- 去除指针，使用安全语言?