2023-2024学年春季学期

计算机体系结构安全

Computer Architecture Security

授课团队：史岗，陈李维

计算机体系结构安全

Computer Architecture Security

[第12次课] 新型代码复用攻击及防御

授课教师：陈李维

授课时间：2024.5.13

内容概要

- 研究背景
- 粗粒度CFI
- 绕过粗粒度CFI的代码复用攻击
- COOP和FOP
- JIT-ROP
- 不可读保护
- CPI
- 总结

内容概要

- 研究背景
- 粗粒度CFI
- 绕过粗粒度CFI的代码复用攻击
- COOP和FOP
- JIT-ROP
- 不可读保护
- CPI
- 总结

代码复用攻击分类

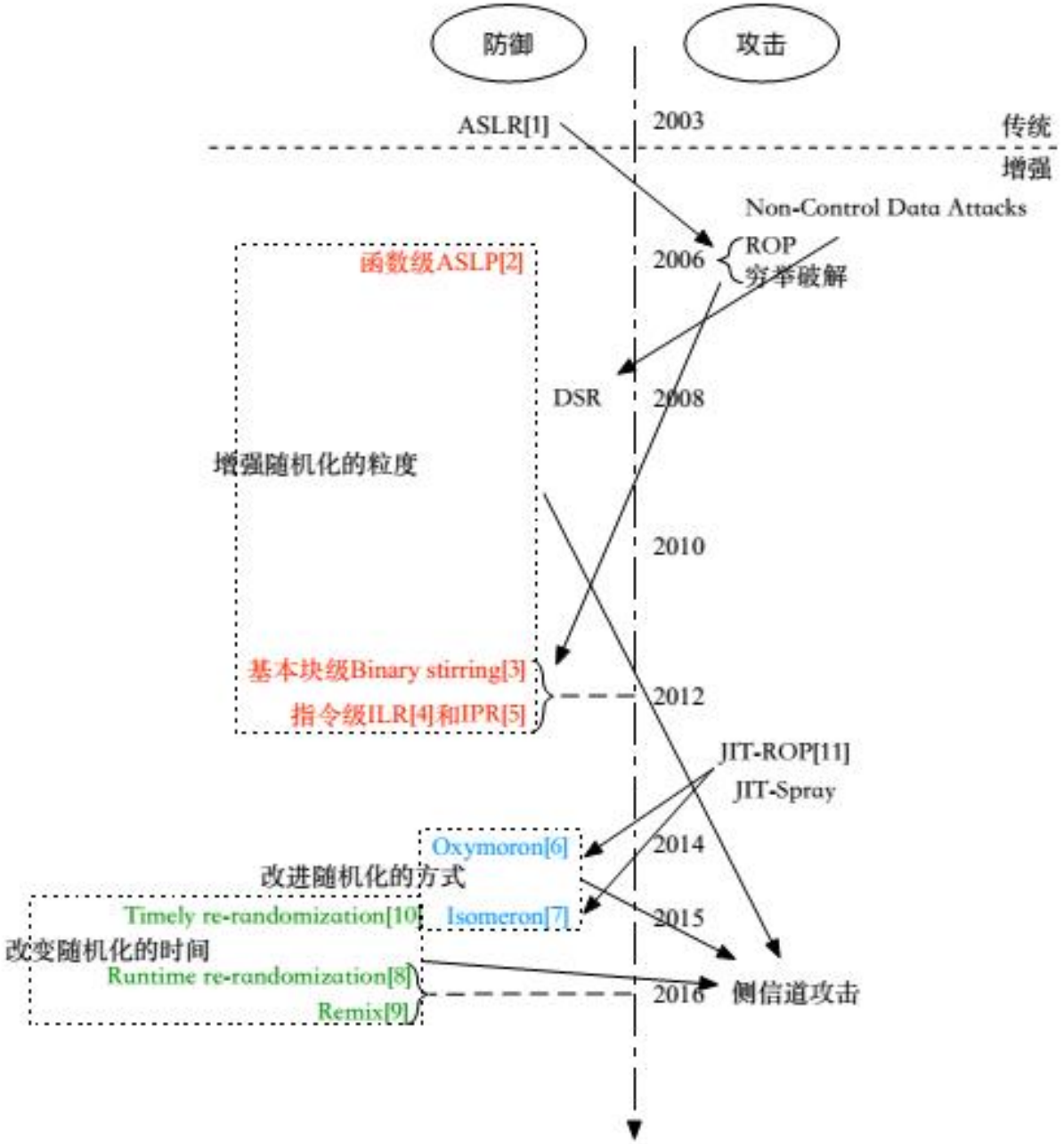
- 根据配件类型、配件之间连接关系、及提出的时间关系等分类，代码复用攻击可以分为两大类：
  - **经典代码复用攻击（上一讲内容）**
    - 都是在2010年以前提出的攻击方法。主要考虑绕过不可执行位保护，实现图灵完备攻击，基本不考虑针对代码复用攻击的防御。
  - **新型代码复用攻击（本讲内容）**
    - 大多是在2010年以后提出的攻击方法。考虑到了对代码复用攻击的防御，思考如何绕过这些针对性的防御方法。

代码复用攻击的防御方法分类

- 代码复用攻击的防御方法也可以分为两大类：
  - **随机化方法：ASLR为代表性方法**
    - 将系统中代码地址随机化，使得攻击者无法找到对应配件的真实地址，从而无法进行攻击。
  - **异常行为检测：CFI为代表性方法**
    - 代码复用攻击的代码执行过程和正常程序过程完全不同。
    - 分析检测程序执行过程，如果发现程序执行过程和正常过程不同，就认为发生了攻击。

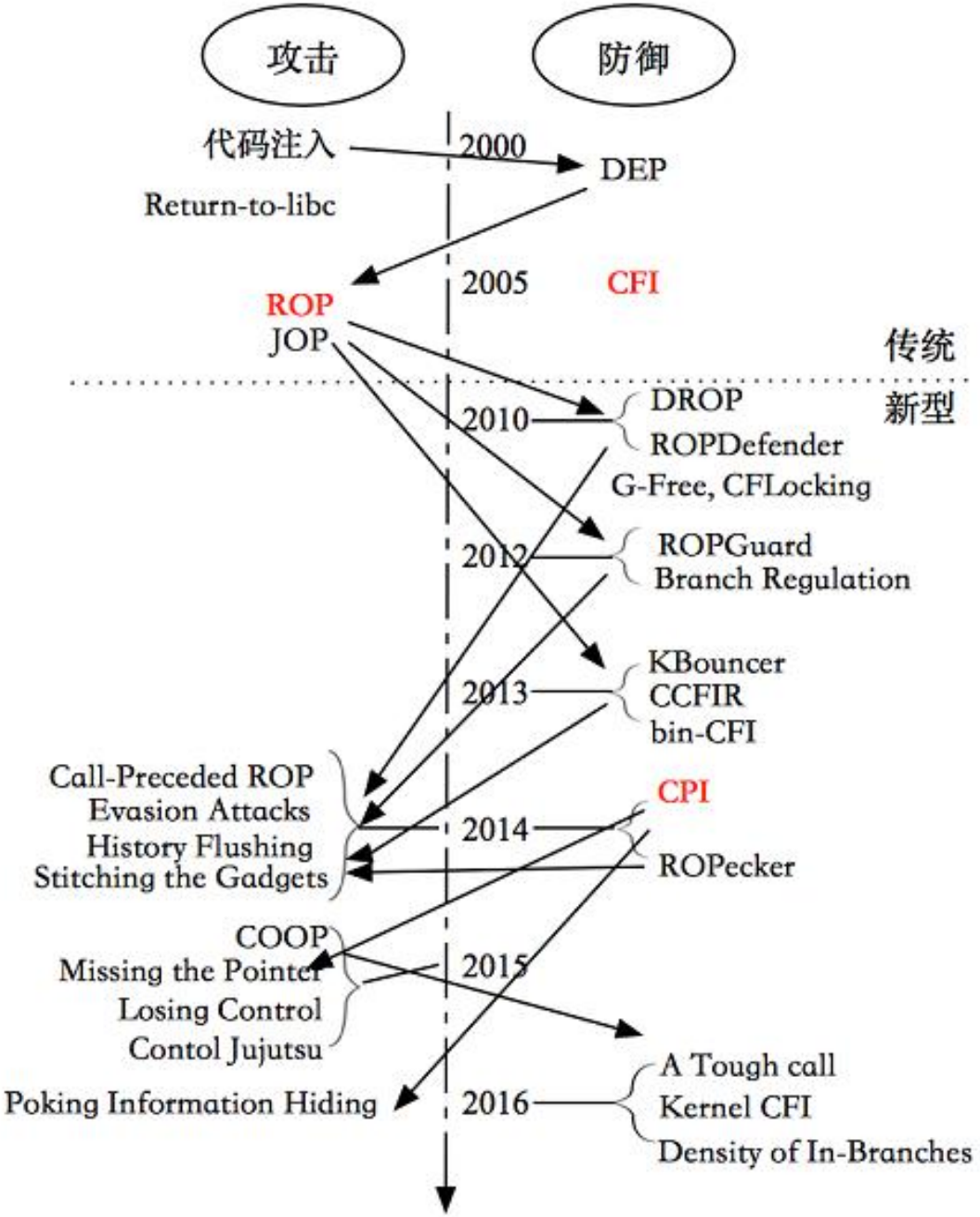
攻防发展脉络

- 从**随机化防御**角度，代码复用攻击及防御之间的相互发展关系
  - ASLR
  - **细粒度**随机化
  - **新的攻击方法：JIT-ROP，代码复用攻击结合内存信息泄露**
  - **新的防御方法：不可读保护**



攻防发展脉络

- 从**异常行为检测**角度，代码复用攻击及防御之间的相互发展关系
  - CFI
  - 粗粒度CFI
  - 绕过粗粒度CFI的攻击
  - 新型代码复用攻击：COOP
  - 新的防御方法：CPI



内容概要

- 研究背景
- **粗粒度CFI**
- 绕过粗粒度CFI的代码复用攻击
- COOP和FOP
- JIT-ROP
- 不可读保护
- CPI
- 总结

粗粒度CFI思想的来源

- CFI是异常行为检测的代表性防御方法，具有很好的防御效果，但是也有以下缺点：
  - **实用性不高，没有被真实系统采用。**
    - 实现复杂
    - 性能损耗过高
  - **依赖CFG（控制流图）**
    - 一个完全精确的CFG是不可能生成的
    - 对于不违反程序CFG的攻击无能为力

粗粒度CFI思想的来源

- 粗粒度CFI的思想来源1：对CFI缺陷的优化和改进
  - 降低复杂性，提高效率
  - 不依赖CFG
- 粗粒度CFI思想来源2：对代码复用攻击和正常程序行为之间特征的分析
  - 对代码复用攻击和正常程序的行为特征进行更深入的分析，从中获得更加通用有效的有区分度的行为特征

粗粒度CFI思想的来源

攻击和正常程序行为的差别

- 代码复用攻击
  - 配件长度很短，也就是说间接跳转指令频率很高，以上两种说法等价
  - 间接跳转指令跳转目标不正常
- 正常程序行为
  - 间接跳转指令频率不高
  - 间接跳转指令跳转目标正常且有规律

粗粒度CFI思想的来源

攻击和正常程序行为的差别

- 代码复用攻击
  - 间接跳转指令（call，ret，jump）的跳转目标：程序代码空间的**任意位置**
- 正常程序行为
  - call的跳转目标：**函数的首地址**
  - ret的跳转目标：**call的下一条指令地址**
    - ret更加准确的目标：**对应call的下一条指令地址**
  - jump的跳转目标：**函数的首地址，本函数内部的任意地址**

粗粒度CFI的原理

- 基本思想：
  - 通过牺牲一部分准确性来降低损耗与实现难度。
- 粗粒度CFI（Coarse-Grained CFI），对代码复用攻击和正常程序行为特征进行深入分析，通过特定的规则来判断程序行为是否正常。
- 细粒度CFI（经典CFI）
  - 使用CFG来判断是否异常
- 粗粒度CFI（启发式CFI）
  - 使用特定的规则来判断是否异常

粗粒度CFI的原理

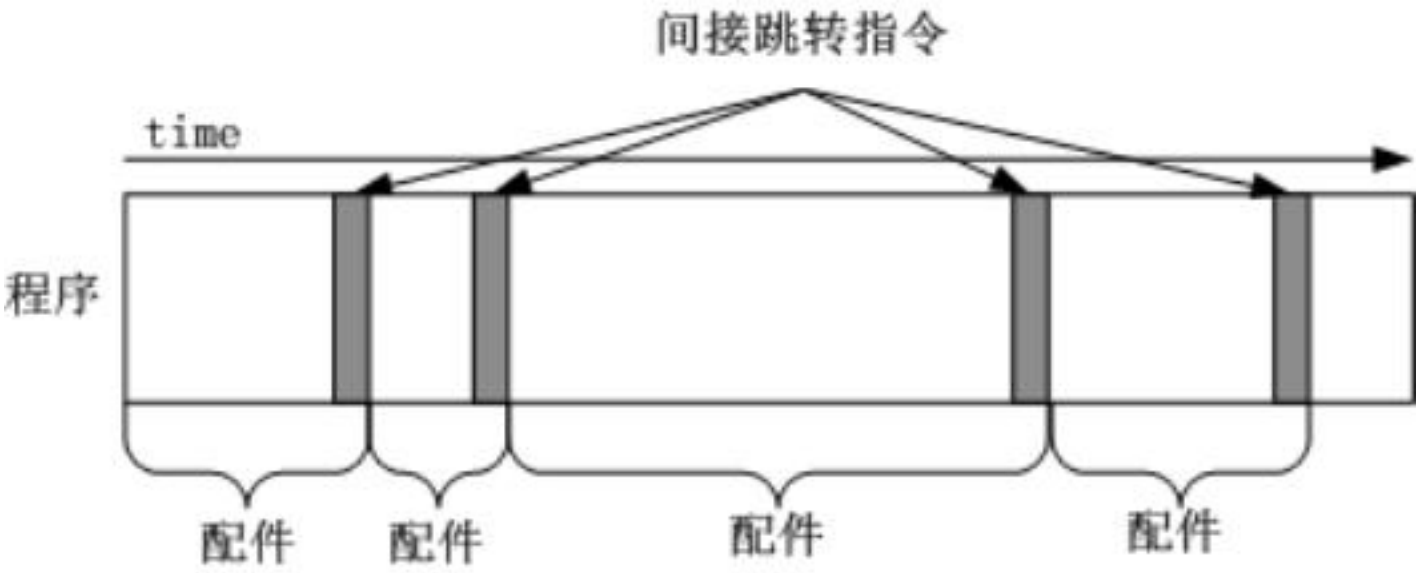
- 粗粒度CFI使用**特定的规则**来判断程序行为是否异常：
  - 1、基于间接跳转指令**跳转目标**的规则
    - 以上规则来源于对CFG的分析和简化，防御效果严格低于细粒度CFI，因此被称为粗粒度CFI
  - 2、基于**配件长度（间接跳转指令频率）**的规则
    - 以上规则来源于对代码复用攻击行为特征的分析
    - 需要设定合适的阈值，存在误判和漏判的可能
    - 也被称为启发式CFI（heuristic-based CFI）

粗粒度CFI的原理

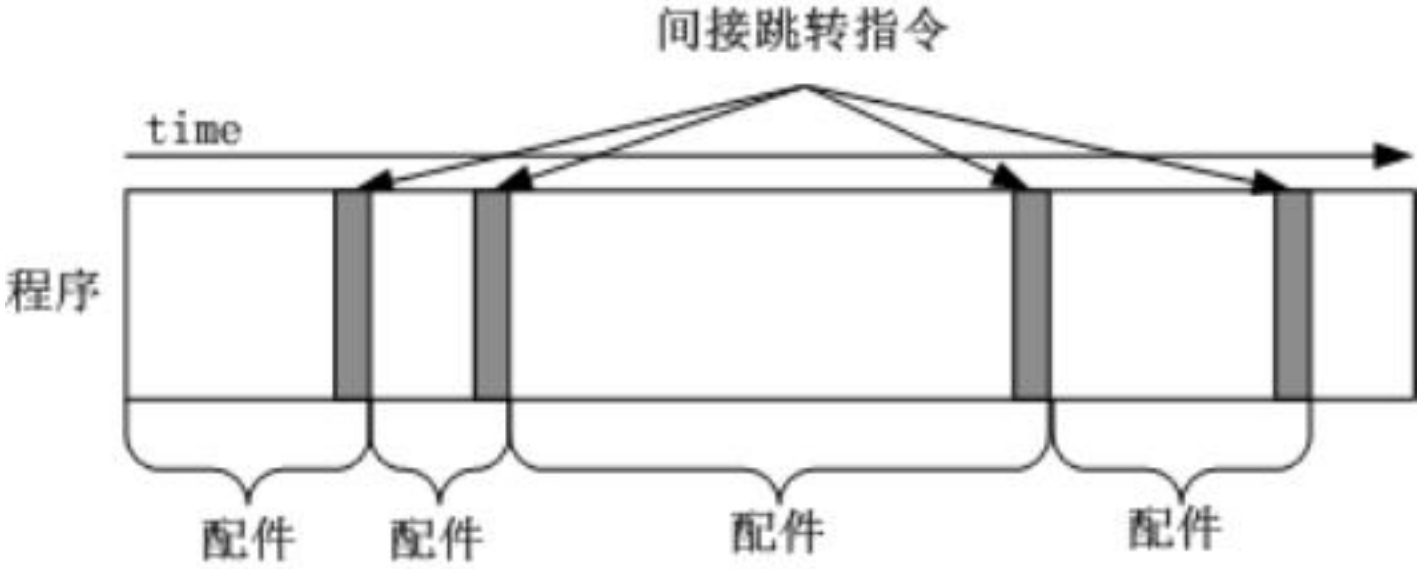
- 1、基于间接跳转指令**跳转目标**的规则：
  - ret指令
    - 1）跳转到**任意**call指令的下一条指令
    - 2）跳转到**对应**call指令的下一条指令
  - call指令
    - 跳转到任意函数的首地址
  - jump指令
    - 跳转到任意函数的首地址，或者当前函数的任意地址

粗粒度CFI的原理

- 2、基于**配件长度**（间接跳转指令频率）的规则
  - 配件是以间接跳转指令为结尾的代码片段
  - 以**间接跳转指令**为边界，将所有处于两个间接跳转指令之间的代码片段都当做是**潜在的配件**



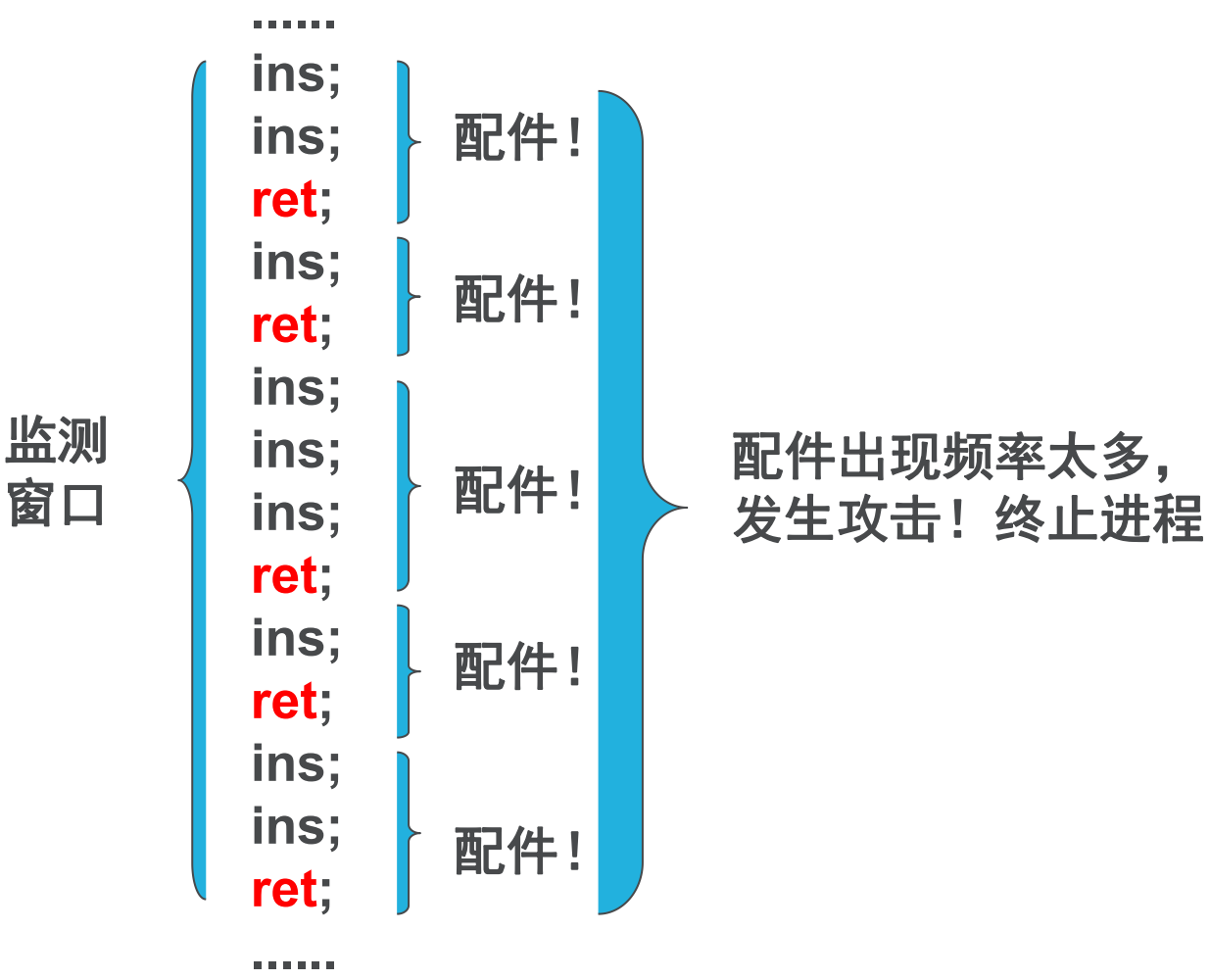
- 2、基于**配件长度**（间接跳转指令频率）的规则
  - 配件长度和间接跳转指令的频率是完全等价的
    - 配件**长度越短**，间接跳转指令出现的**频率越高**
    - 配件**长度越长**，间接跳转指令出现的**频率越低**



粗粒度CFI的原理

- 2、基于**配件长度**（间接跳转指令频率）的规则
  - 为了达成预期攻击目标，避免无用指令的干扰，配件链通常由**多个短配件**组成。
    - **短配件**：配件中的指令数量不大于x
    - **配件链**：连续出现y个短配件
    - 一旦发现连续出现y个长度不大于x的短配件，就认为发生了代码复用攻击。
  - x，y是**可配置的**。
    - x通常设为5-6
    - y通常设为10-20

2、基于配件长度（间接跳转指令频率）的规则

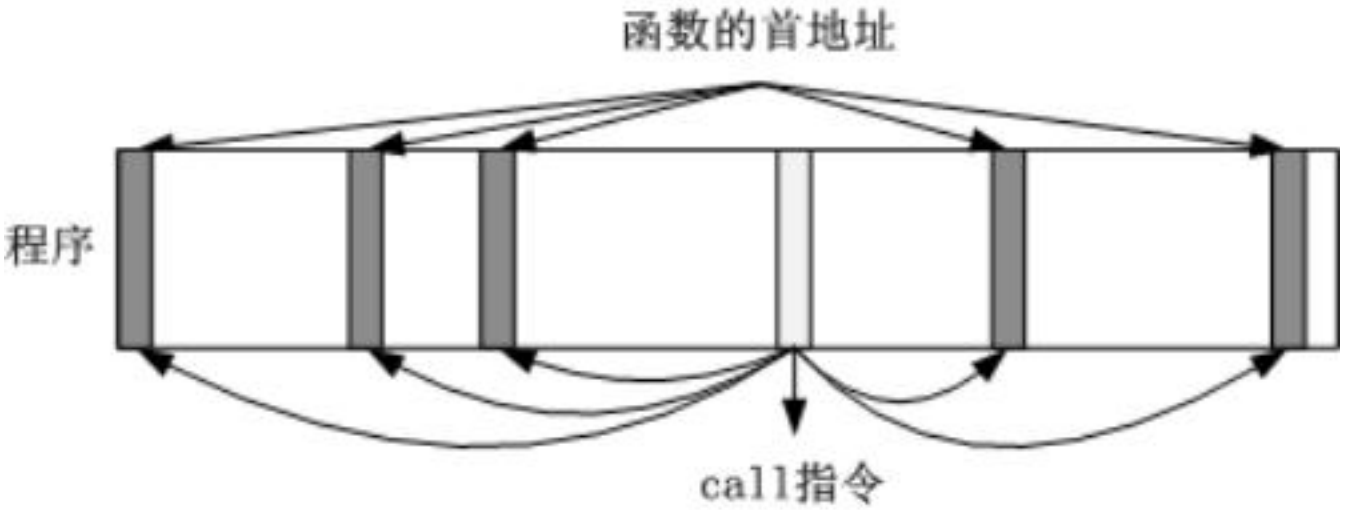


粗粒度CFI的实现

- 粗粒度CFI规则的具体实现：
  - 1、基于间接跳转指令跳转目标规则的实现
    - 对call、jump、ret指令及其跳转目标的实时监控和分析
  - 2、基于配件长度（间接跳转指令频率）规则的实现
    - 对间接跳转指令和普通指令运行数量的实时监控和分析

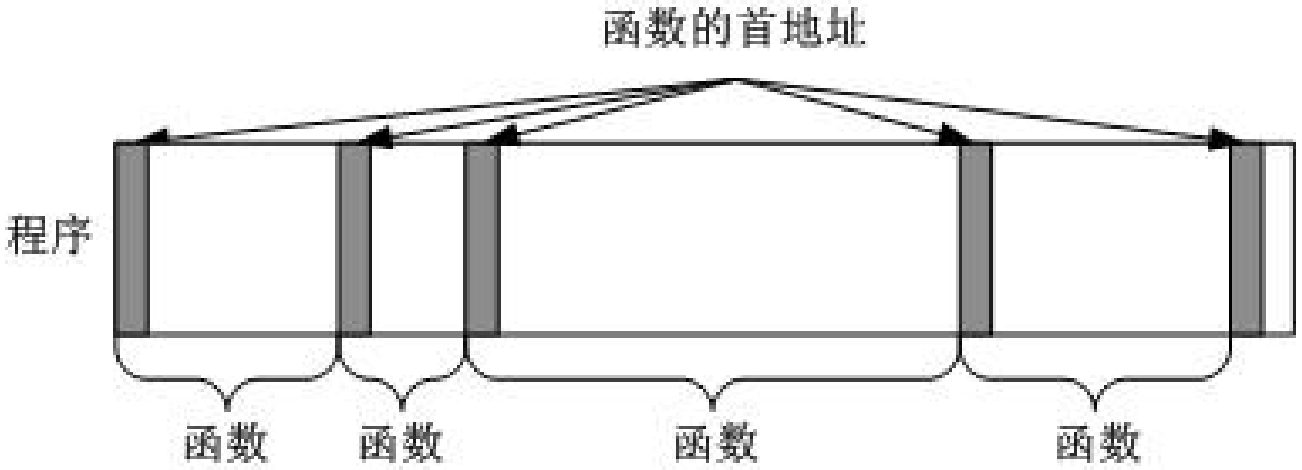
粗粒度CFI的实现

- 1、基于间接跳转指令**跳转目标**规则的实现：
  - 1）call的跳转目标地址应该是函数首地址
    - 修改二进制文件，对所有**函数的首地址**加一个标记。
    - 程序运行时，检测所有call指令的跳转目标地址是否是**被标记的函数首地址**。
    - 如果不是函数首地址，则认为发生了异常。



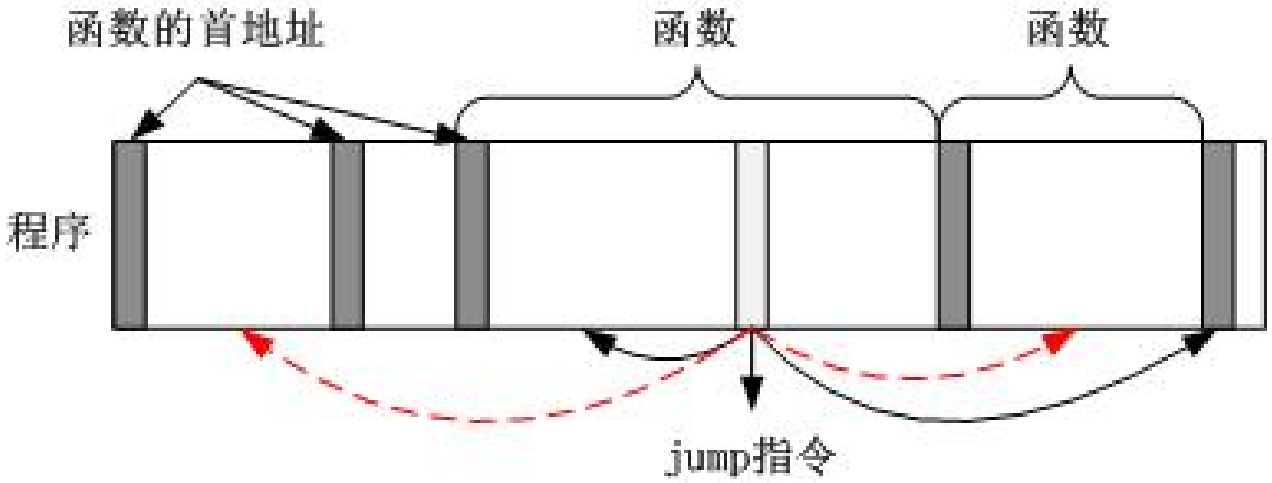
粗粒度CFI的实现

- 1、基于间接跳转指令**跳转目标**规则的实现：
  - 2） jump的跳转目标地址应该是函数首地址或者当前函数内部的任意地址
    - 修改二进制文件，对所有**函数的首地址**加一个标记，并且标记所有**函数的长度**。
    - 程序运行时，根据函数首地址和长度，能够判断当前执行指令处于哪一个函数内部。



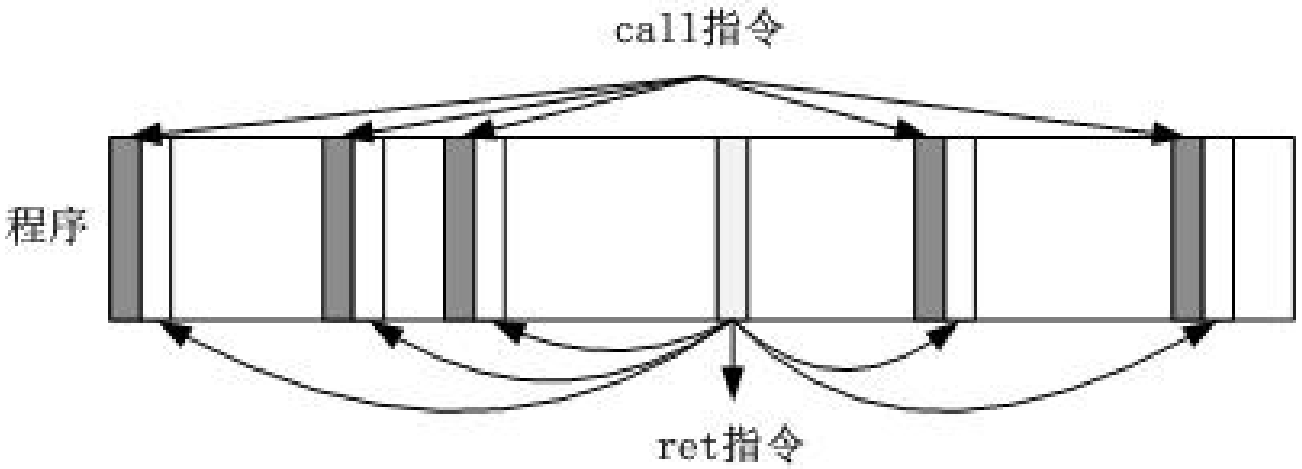
粗粒度CFI的实现

- 1、基于间接跳转指令**跳转目标**规则的实现：
  - 2） jump的跳转目标地址应该是函数首地址或者当前函数内部的任意地址
    - 检测jump指令的跳转目标地址**是否跨越了**当前函数边界。如果没有，继续运行。
    - 如果跨越了当前函数边界，检测jump的跳转目标地址是否为**被标记的函数首地址**。



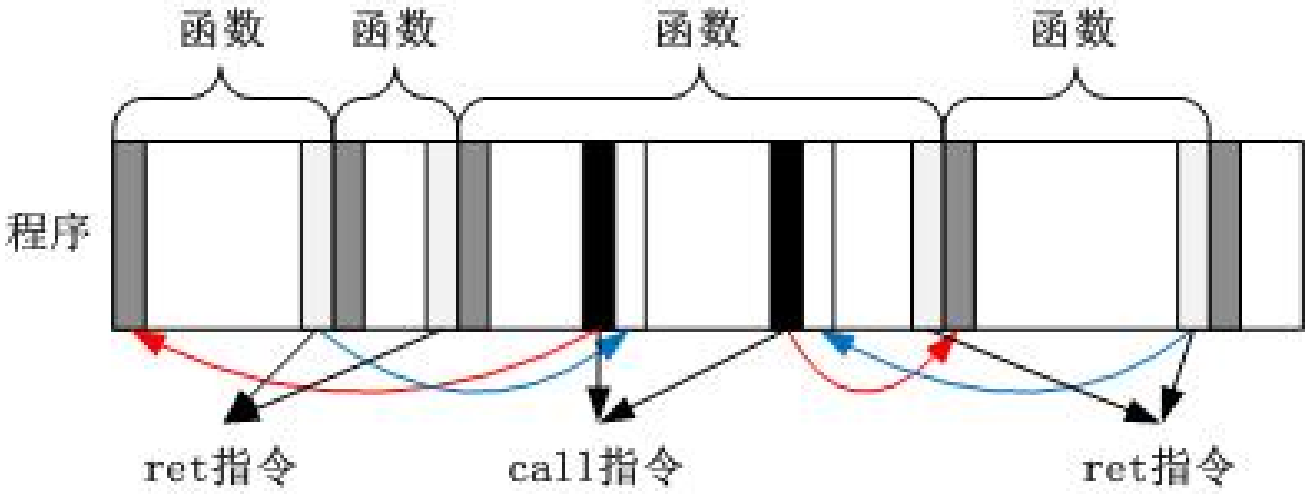
粗粒度CFI的实现

- 1、基于间接跳转指令跳转目标规则的实现：
  - 3）ret的跳转目标地址应该是任意call指令的下一条指令
    - 程序运行时，检测ret指令的跳转目标地址是否是call指令的下一条指令。
    - 如果不是，则认为发生了异常。



粗粒度CFI的实现

- 1、基于间接跳转指令跳转目标规则的实现：
  - 3）ret的跳转目标应该是对应call的下一条指令
    - ret和call是一一对应的，并且ret对应的是最近一次call。
    - 程序运行时，记录call指令的下一条指令地址。
    - 当执行ret指令时，检测ret指令的跳转目标地址是否是最近一次call指令的下一条指令。



粗粒度CFI的实现

- 1、基于间接跳转指令**跳转目标**规则的实现：
  - 3）ret的跳转目标应该是**对应**call的下一条指令
    - call指令的作用就是将**函数返回地址**压栈，然后跳转到对应函数执行。
    - **call指令的下一条指令地址就是函数返回地址。**
  - 粗粒度CFI对ret目标地址的检测行为和正常程序的函数调用行为是**相同的**。

粗粒度CFI的实现

- 影子栈（shadow stack），是独立于用户栈的另外一个栈空间，专门用于保存函数返回地址。
- 影子栈是实现粗粒度CFI对ret目标地址检测的常用方法。
  - 执行call指令时，除了将函数返回地址正常压栈以外，同时将函数返回地址压入影子栈。
  - 执行ret指令时，同时读取栈和影子栈的两个函数返回地址，并进行比较。
  - 如果两者相等，则正常，继续执行。如果不相等，则认为发生了攻击。

粗粒度CFI的实现

- 2、基于**配件长度**规则的实现：
  - x86处理器中有一个特殊的寄存器LBR（Last Branch Record），用于记录最后16项跳转指令的信息。
  - 利用LBR记录并分析间接跳转指令，配件长度等于两个连续间接跳转指令之间指令数量。
  - 一旦发现连续出现y个长度不大于x的短配件，就认为发生了代码复用攻击。

粗粒度CFI的实现

- 2、基于**配件长度**规则的实现：
  - 一旦发现连续出现y个长度不大于x的短配件，就认为发生了代码复用攻击。
  - 一个难点在于x和y的设置，如果设置不好，会导致出现大量的误判或漏判。
    - **误判**：将正常程序当做了异常。
    - **漏判**：将异常程序当做了正常。
  - 如果x设置过小，可能将真正的配件当做了正常程序（漏判）。如果x设置过大，可能将正常程序当做了配件（误判）。
  - y的设置和x类似。

粗粒度CFI的实际应用

- 2016年Intel提出了CET（Control-flow Enforcement Technology，控制流增强技术），是粗粒度CFI的一种具体实现形式。
  - 实现影子栈，完成对ret跳转目标的检测。
  - 实现对call和jump跳转目标的检测。
    - 增加了一条新的指令ENDBR32/ENDBR64，在所有函数头插入ENDBR32/ENDBR64 。
    - call和jump的跳转目标必须是ENDBR32/ENDBR64。
- 此外，ARM公司也提出了类似的防御机制，同学们可以自行搜索学习。

粗粒度CFI分析

- 经典CFI（细粒度CFI）
  - 需要CFG
  - 根据CFG严格限制间接跳转指令的跳转目标，但是精确的CFG难以分析得到
- 粗粒度CFI
  - 使用特定的规则
  - 根据对代码复用攻击和正常程序行为特征分析，总结得到一些特定的规则

粗粒度CFI分析

- 优点：
  - 实现简单，效率高，开始被实际系统逐步的采用
  - 能够防御大部分常见的代码复用攻击，提高代码复用攻击的攻击难度
    - 限制了间接跳转指令的跳转目标，加大了寻找可用配件的难度
    - 限制了配件的长度，加大了构造配件链的难度

粗粒度CFI分析

- 缺点：
  - 防御效果低于经典的CFI
  - 能够被有针对性的绕过
    - 从学术界的角度，防御效果仍然是不够
    - 从工业界的角度，防御有一定的效果，能防御住最主要的、最基本的攻击方法，且实现简单，代价较低

粗粒度CFI小结

- 本小节介绍了**粗粒度CFI**，是异常行为检测领域一种高效实用的防御方法。
- 分析代码复用攻击和正常程序行为特征，总结了几个特定的规则：
  - 基于间接跳转指令跳转目标的规则
  - 基于配件长度（间接跳转指令频率）的规则
- 粗粒度CFI目的**并不在于彻底阻止**代码复用攻击，而是要**增加**攻击者构造配件链的**难度**。

内容概要

- 研究背景
- 粗粒度CFI
- **绕过粗粒度CFI的代码复用攻击**
- COOP和FOP
- JIT-ROP
- 不可读保护
- CPI
- 总结

粗粒度CFI的本质

- 通过一些**特定规则**来判断程序行为是否异常
  - ret指令
    - 1）跳转到**任意**call指令的下一条指令
    - 2）跳转到**对应**call指令的下一条指令
  - call指令
    - 跳转到任意函数的首地址
  - jump指令
    - 跳转到任意函数的首地址，或者当前函数的任意地址
  - 配件长度（间接跳转指令频率）
    - 连续出现多个短配件

绕过粗粒度CFI的方法

- **合法配件**：符合粗粒度CFI规则的特殊的配件。
- 利用合法配件组成配件链，就能够绕过粗粒度CFI的防御。
  - Call-Preceded Gadget，CP配件，符合ret指令跳转目标的规则。
  - Entry-Point Gadget，EP配件，符合call和jump跳转目标的规则。
  - Long-NOP Gadget，LN配件，符合基于配件长度的规则。

绕过粗粒度CFI的方法

call-preceded配件

- ret指令的跳转目标规则：跳转到**任意**call指令的下一条指令。
- **call-preceded指令：call指令的下一条指令**
- **call-preceded配件**：以call-preceded指令为起始的配件。
- call-preceded配件符合上述对于ret指令的规则。
  - **注：call-preceded配件不符合更加精确的ret跳转目标的规则：跳转到对应call指令的下一条指令。**

绕过粗粒度CFI的方法

call-preceded配件

- 程序中call指令的**数量很多**，研究表明能够找到许多call-preceded配件，足够满足攻击者所需要的功能。
- 由call-preceded配件组成的配件链能够绕过对ret返回地址的检测。

Type	Call-Preceded Sequence
Call 1	`lea eax,[ebp-34h]; push eax;`
	`call esi; ret`
Call 2	`call eax`
Call 3	`push eax; call [ebp+0Ch]`

绕过粗粒度CFI的方法

entry-point配件

- call和jump的跳转目标规则：call和jump应该跳转到函数的首地址。
- entry-point配件：以函数首地址为起始的配件。
- entry-point配件符合上述对于call和jump指令的规则。

绕过粗粒度CFI的方法

entry-point配件

- 程序中**函数数量很多**，研究表明能够找到许多entry-point配件，足够满足攻击者所需要的功能。
- 使用entry-point配件组成的配件链就能绕过针对call和jump跳转地址的检测。

绕过粗粒度CFI的方法

Long-NOP配件

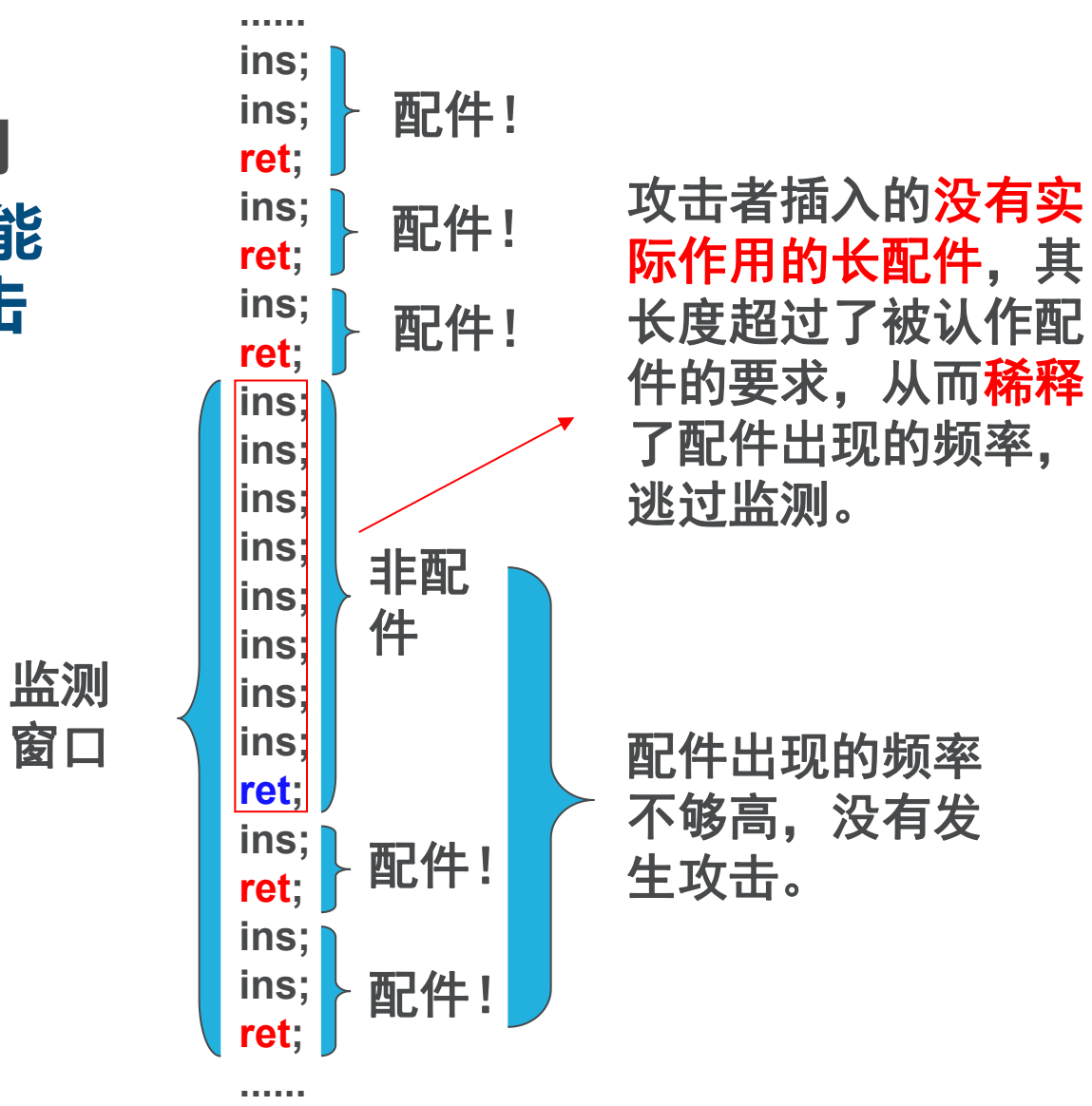
- 基于配件长度的规则：不能连续出现多个短配件。
- 显然，在配件链中插入一些**长配件**就能绕过上述规则。
- 但是，长度越长的配件，指令越多，行为更加复杂，会影响配件链的稳定性和功能。
- **Long-NOP配件**：长度超过阈值x的配件，并且不影响配件链中有效数据的操作与交互。

Long-NOP配件

- **Long-NOP配件**：不能对正常配件之间的数据交互与操作产生影响。配件之间的数据交互是通过寄存器完成的。
- 因此，LN配件应该包含以下指令：
  - 1）包含一些如NOP指令等不修改寄存器的指令。
  - 2）包含一些写内存指令。可以控制这些指令的目标地址，让这些写内存指令去修改无关紧要的内存内容，而不会影响配件链的正常功能。

绕过粗粒度CFI的方法

- Long-NOP配件
  - Long-NOP配件的使用
    - Long-NOP配件不能影响配件链的攻击功能



小结

- 针对粗粒度CFI的规则，采用特殊的**合法配件**，伪装成正常的程序。
  - Call-Preceded Gadget，CP配件，符合ret指令跳转目标的规则。
  - Entry-Point Gadget，EP配件，符合call和jump跳转目标的规则。
  - Long-NOP Gadget，LN配件，符合基于配件长度的规则。
- 以上三种配件可以混合使用，相互配合，共同绕过粗粒度CFI的防御。

内容概要

- 研究背景
- 粗粒度CFI
- 绕过粗粒度CFI的代码复用攻击
- COOP和FOP
- JIT-ROP
- 不可读保护
- CPI
- 总结

粗粒度CFI的分析

- 根据前两节的内容，可以发现粗粒度CFI的规则主要是围绕**函数**开展的。
- 因此，如果以**函数**为配件，**几乎完美符合**所有粗粒度CFI的规则：
  - 符合**精确的**ret跳转目标规则：跳转到**对应**call指令的下一条指令。
  - 符合call和jump的跳转目标规则：跳转到任意函数的首地址。
  - 符合配件长度规则：函数的长度一般要大于5-6条指令。

COOP攻击简介

- COOP（Counterfeit Object-Oriented Programming，面向伪造对象的编程方法）
  - 是2015年提出的一种新的代码复用攻击方法。
  - 针对C++编程语言，COOP攻击通过建立虚假对象，用已经存在的C++虚函数为配件来构造配件链。

FOP攻击简介

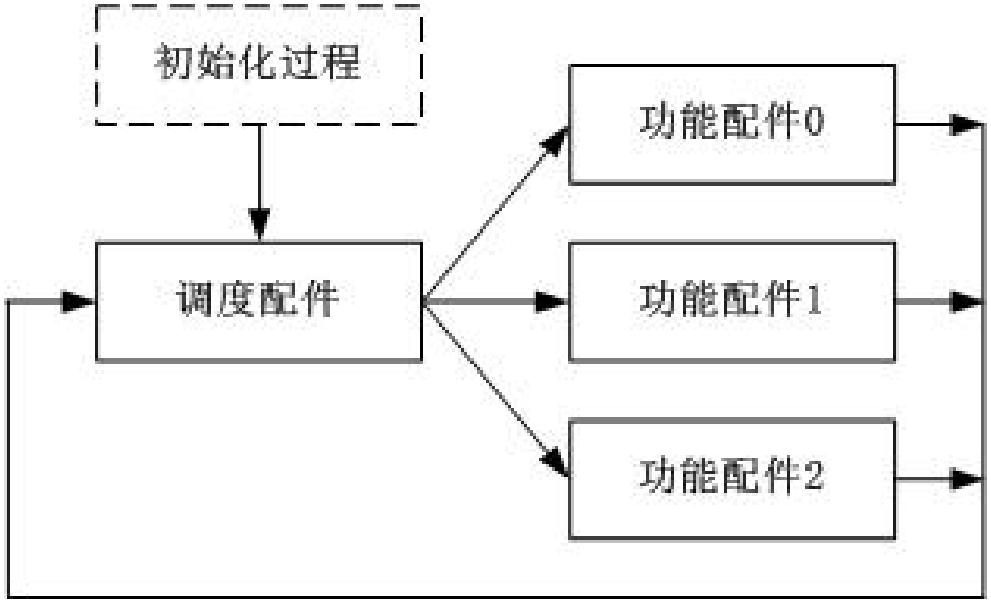
- FOP（Function-oriented programming，面向函数的编程方法）
  - 是我参考COOP的特点，提出的针对C语言的代码复用攻击方法。
  - 简单来说，就是利用程序中已经存在的函数为配件，构造配件链进行攻击。
- COOP攻击：
  - 以虚函数为配件，针对C++语言
- FOP攻击：
  - 以函数为配件，针对C语言

FOP攻击原理

- FOP的实现和JOP类似，将配件分为两种类型：
  - **功能配件**(functional gadget)
    - 完成某种特定功能的函数配件。
  - **调度配件**(dispatcher gadget)
    - 充当程序EIP的作用，实现控制流的转移。
    - 负责组织功能配件的执行。

FOP攻击过程

- 初始化FOP攻击，注入攻击数据。
- 控制调度配件的函数指针，跳转到功能配件。
- 功能配件执行完成，返回到调度配件。
- 调度配件继续，跳转到下一个功能配件。
- 一直循环，完成FOP攻击。



FOP攻击的功能配件

- FOP的功能配件举例：
  - 读写内存的配件

```
void writeMem(char a[],char b[]){
    strcpy(a,b);
}
```

  - 运算配件

```
void arith(){
    int a,b,c;
    c=a+b;
}
```

FOP攻击的功能配件

- FOP的功能配件举例：
  - 条件分支配件

```
void conditionalBranch(){
    if(cond){
        cond=0;
    }
    else{
        cond=1;
    }
}
```

FOP攻击的功能配件

- FOP的功能配件举例：
  - 读寄存器的配件

```
void readArgReg(char a[],char b[],char c[]){

}
```

```
0000000000040065f <readArgReg>:
  40065f:       55                      push   %rbp
  400660:       48 89 e5                mov    %rsp,%rbp
  400663:       48 89 7d f8             mov    %rdi,-0x8(%rbp)
  400667:       48 89 75 f0             mov    %rsi,-0x10(%rbp)
  40066b:       48 89 55 e8             mov    %rdx,-0x18(%rbp)
  40066f:       90                      nop
  400670:       5d                      pop    %rbp
  400671:       c3                      retq
```

FOP攻击的功能配件

- FOP的功能配件举例：
  - 写寄存器的配件

```
void writeArgReg(){
    char a[10],b[10],c[10];
    readArgReg(a,b,c);
}
```

```
lea     -0x20(%rbp),%rdx
lea     -0x30(%rbp),%rcx
lea     -0x40(%rbp),%rax
mov     %rcx,%rsi
mov     %rax,%rdi
mov     $0x0,%eax
callq   40065f <readArgReg>
nop
```

从栈中读取数据

写数据到参数寄存器

FOP攻击的调度配件

- FOP的调度配件
  - 内部包含**函数指针**，并且尽量不要携带参数。
  - 三种常见调度配件：
    - 1）循环，一个函数指针，一个能够修改函数指针的内存漏洞

```
void (*fp1)(void);
void (*fp2)(void);
void (**fp)(void);
char s[10];
int cond,limit;
```

```
void mloop1(){
    for(int i=0;i<limit;i++){
        read(0,s,10);
        fp1();
    }
}
```

FOP攻击的调度配件

- FOP的调度配件
  - 三种常见调度配件：
    - 2）两个不同的函数指针，一个能够修改这两个函数指针的内存漏洞

```
void (*fp1)(void);
void (*fp2)(void);
void (**fp)(void);
char s[10];
int cond,limit;
```

```
void mloop2(){
    read(0,s,10);
    fp1();
    fp2();
}
```

FOP攻击的调度配件

- FOP的调度配件
  - 三种常见调度配件：
    - 3）循环，一个二级函数指针

```
void (*fp1)(void);
void (*fp2)(void);
void (**fp)(void);
char s[10];
int cond,limit;
```

```
void mloop3(){
    for(int i=0;i<limit;i++)
        fp[i]();
}
```

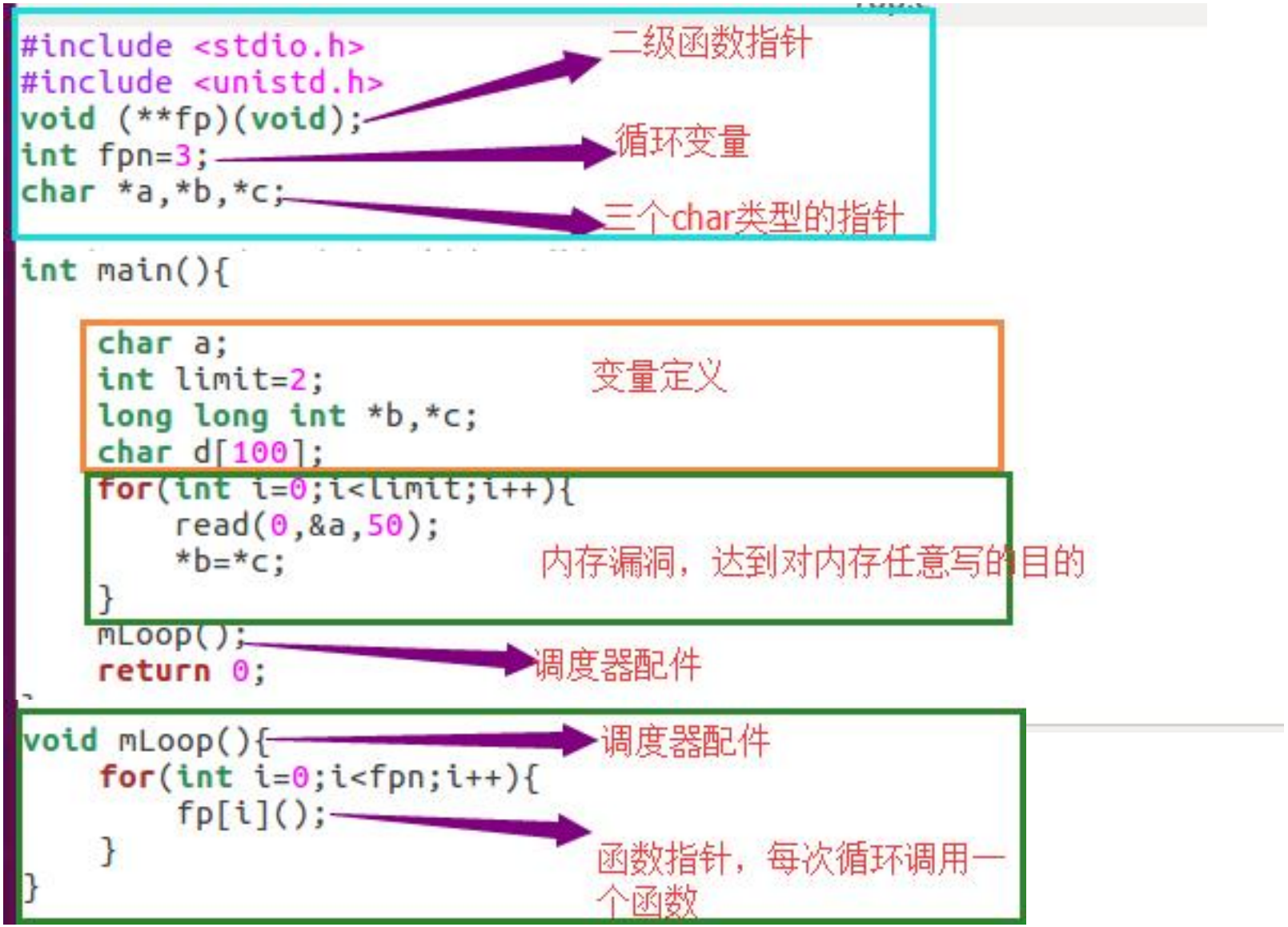
COOP和FOP的分析

- 优点：
  - 函数是程序的基本功能单元，通常具有一个固定的功能，而且函数在程序中数量很多，因此能够很方便的构造出以函数为配件的COOP和FOP。
  - COOP和FOP都是图灵完备的攻击。
  - 函数配件包含较多的指令，长度一般较长，能够绕过以配件长度为判断规则的粗粒度CFI。
  - 以函数为配件，相当于结合了call-preceded和entry-point两种配件的特征，符合粗粒度CFI对ret、call、jump的跳转目标的规则，因此COOP和FOP都能够绕过粗粒度CFI。
  - 函数指针是动态数据，在程序运行时实时赋值和变化，静态分析难以获取函数指针的具体信息，静态分析得到的CFG也难以限制函数指针的跳转，因此COOP和FOP也能绕过不够精确的细粒度CFI。

FOP的攻击示例

- 环境：Ubuntu 16.04， 64位
- 假设条件：
  - 已知程序的内存布局
  - 存在一个可以对内存任意写的漏洞
- 本次攻击目的
  - 执行system(“/bin/sh”)

FOP的攻击示例


#include <stdio.h>
#include <unistd.h>
void (**fp)(void);
int fpn=3;
char *a,*b,*c;
二级函数指针
循环变量
三个char类型的指针
int main(){
char a;
int limit=2;
long long int *b,*c;
char d[100];
变量定义
for(int i=0;i<limit;i++){
read(0,&a,50);
*b=*c;
}
内存漏洞，达到对内存任意写的目的
mLoop();
return 0;
调度器配件
void mLoop(){
调度器配件
for(int i=0;i<fpn;i++){
fp[i]();
}
函数指针，每次循环调用一个函数
}

FOP的攻击示例

```
void callFp(void (**cb)(void)){
     fp=cb;
}

void no_arg_no_return(){
    one_arg_no_return(a);
    two_arg_no_return(a,b);
    three_arg_no_return(a,b,c);
}

void one_arg_no_return(char a[]){

}

void two_arg_no_return(char a[],char b[]){

}

void three_arg_no_return(char a[],char b[],char c[]){

}

int no_arg_one_return(){
    return 1;
}

int one_arg_one_return(int a){
    a=a+2;
    return a;
}
```

writeArgReg配件

readArgReg配件

写寄存器rax

算术运算配件

小结

- 介绍了COOP和FOP，以函数为配件，构造配件链进行攻击。
- COOP和FOP易于构造，并且能够绕过粗粒度CFI，是一种比较优秀的代码复用攻击方法。

内容概要

- 研究背景
- 粗粒度CFI
- 绕过粗粒度CFI的代码复用攻击
- COOP和FOP
- JIT-ROP
- 不可读保护
- CPI
- 总结

随机化防御方法

- ASLR：对堆、栈、共享库映射等线性内存区域布局进行随机化。ASLR随机化的粒度很粗，同一个内存段中**内部相对偏移是固定的**。
  - 如果攻击者知道了一条指令的位置，就能够根据相对偏移，找到其他指令的地址。
- **细粒度**随机化防御：提高内存空间随机化的粒度，以**页面、程序块甚至单条指令为单位**进行内存位置随机化，能够进一步提高攻击者获取配件的难度，提升系统的安全性。

随机化防御方法

- 随机化防御的实现：
  - 在程序的装载阶段，操作系统将程序的每个模块(例如代码段、数据段、栈、堆、代码块、单条指令等)的基地址进行随机化排布。
  - 攻击者无法准确获得某个配件的精确地址，因此无法构造配件链进行攻击。
  - 潜在问题：只在程序加载过程进行随机化，之后的程序运行过程，内存布局保持不变。

内存信息泄露

- **内存信息泄露**是随机化防御面临的最大难题：
  - 直接泄露：读取代码段，找到直接跳转和直接调用的目标地址，收集这些地址进行分析，就可以得出代码页的分布位置。
  - 间接泄露：读取数据段中的函数指针，虚表以及栈中返回地址等包含代码地址的数据，可以达到相同的效果。
- 由于随机化只在加载过程进行，攻击者可以利用内存信息泄露漏洞**在程序运行阶段**获取整个内存空间的信息。

内存信息泄露

- 动态内存泄露与代码复用攻击结合，能够绕过现行大多数商用操作系统的防御机制。
  - 攻击者利用内存信息泄露漏洞，结合对目标程序的预先了解，对运行中的程序内存进行**动态读取**，可以**获取随机化后的内存布局**，从而破解了随机化防御方法。
  - 由于程序运行过程中的内存布局保持不变，攻击者能够找到所需配件的精确内存位置，**动态构造配件链**进行攻击。

JIT-ROP原理

- JIT-ROP（Just-in-time ROP），是于2013年提出的一种代码复用攻击方法。
  - 通过实施一连串的内存信息泄露攻击，绕过内存位置随机化防御机制，迭代搜索内存空间，动态查找构造攻击所需的配件位置。
  - 可以实现自动化的ROP攻击，无需攻击者手工查找构造ROP配件链。

JIT-ROP原理

- JIT-ROP攻击的两个前提条件：
  - 1）需要给攻击框架的接口提供一个**内存泄露漏洞**，以获得某个内存绝对地址中存储的值。
  - 2）需要有一个**可篡改的代码指针**。例如，位于堆上的虚函数指针，或位于栈上的局部变量。

JIT-ROP原理

- JIT-ROP攻击步骤：
  - 1）获取一个代码页：
    - 向JIT-ROP攻击框架接口提供一个内存泄露漏洞和一个初始代码指针。
    - 通过对该代码指针所指位置的反复读取，**获取到整个4KB大小的代码页**。
  - 2）获取多个代码页：
    - 实行动态代码分析技术来确认初始代码页中的间接和直接转移指令（jmp和call），利用这些跳转指令的目标地址来寻找其他代码页。
    - 如果发现某个跳转指令的目标地址不在同一个页内，则说明该目标地址对应的指令存于另外一个新的代码页。
    - 不断进行以上操作，直到**找到足够数量的代码页**。

JIT-ROP原理

- JIT-ROP攻击步骤：
  - 3）查找API函数：
    - 根据API函数调用操作序列的特征在第二步中找到的所有代码页中查找，**寻找能够调用API函数的配件**（如call api_func）。
    - 然后，寻找能够向API函数传递参数的配件。
  - 4）查找配件地址：
    - 根据配件的功能，动态地在第二步中得到的代码页中查找**可用配件的地址**。

JIT-ROP原理

- JIT-ROP攻击步骤：
  - 5）构造配件链：
    - 根据攻击者的需求，利用找到的API调用和传参配件，**动态构造**出可用于代码复用攻击的配件链。
  - 6）攻击：
    - 装载配件链，篡改控制流，发起JIT-ROP攻击。

JIT-ROP分析

- JIT-ROP攻击分析：
  - JIT-ROP针对的是随机化方法，能够绕过当今主流的随机化防御，如ASLR和细粒度随机化。
  - JIT-ROP能够自动化的构造，易于实现，方便攻击者使用。
  - JIT-ROP没有考虑对异常行为检测技术的攻击，如CFI等。

小结

- **内存信息泄露是随机化防御方法的最大难题。**
- 只要攻击者能够利用内存信息泄露漏洞，动态读取内存中的任意数据，那么攻击者就能够找到可用配件的精确位置，构造配件链进行攻击。

内容概要

- 研究背景
- 粗粒度CFI
- 绕过粗粒度CFI的代码复用攻击
- COOP和FOP
- JIT-ROP
- **不可读保护**
- CPI
- 总结

代码信息泄露的本质

- 内存信息泄露+代码复用攻击能够绕过随机化保护。
  - JIT-ROP利用内存信息泄露漏洞，读取程序代码段的内容，获得可用配件的精确内存地址，从而构造配件链进行攻击。
- 代码信息泄露的本质原因是：**代码是可读的。**也就是说，指令被当做普通数据被用户读取。

不可读保护的原理

- 如果将代码段设置为**不可读**，就能够避免程序代码段的信息泄露。
- 攻击者无法读取程序代码段的内容，自然也就无法获得可用配件的精确内存地址。
- 因此，**不可读保护**与内存位置随机化防御机制相结合，可以有效的防御**内存泄露攻击**与代码复用攻击。

不可读保护的原理

- **不可读保护**，于2014年提出。
- 主要思想：将内存的读权限和执行权限剥离。
- 实现方法：对源代码进行重新编译。
  - 将代码段中的指令和数据剥离，彻底消除对代码段的数据读取操作。
  - 生成只具有执行权限的代码段，防止直接代码信息泄露。

内存页的属性

- 在Linux系统中，内存页一共有四种属性：
  - 是否私有：
    - 私有P（private）：属于一个进程的私有页面，只允许一个进程访问。如果另一个进程想要修改该页面的数据，需要将该页面拷贝一份，对拷贝页面进行修改。
    - 共享S（shared）：多个进程共享一个内存页面。
  - 读：用R表示，表示页面是否可读。
  - 写：用W表示，表示页面是否可写。
  - 执行：用X表示，表示页面是否可执行。

内存页的属性

- 在实际系统中，内存页的四个属性**并不是完全解耦**的。
- 其中，字母表示私有或共享，后面的三个数字依次表示xwr。

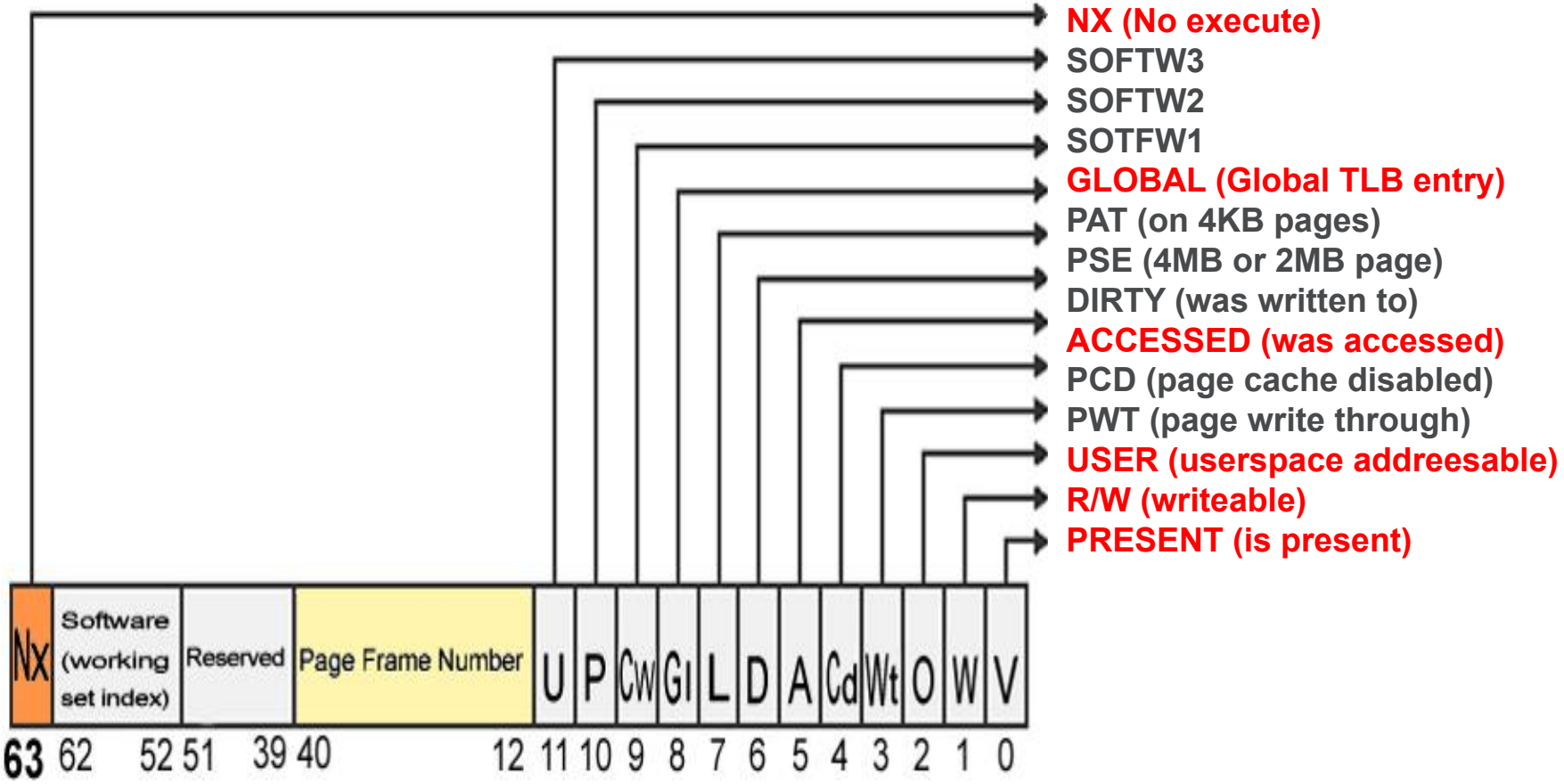
页面权限	权限说明	Prot值
PAGE_NONE	不可访问	P000、S000
PAGE_SHARED	不可执行的共享页面	S010、S011
PAGE_SHARED_EXEC	可执行的共享页面	S110、S111
PAGE_COPY_NOEXEC	不可执行的可拷贝页面	
PAGE_COPY_EXEC	可执行的可拷贝页面	P110、P111
PAGE_COPY	等同于PAGE_COPY_NOEXEC	P010、P011
PAGE_READONLY	只读页面	P001、S001
PAGE_READONLY_EXEC	可执行的只读页面	P100、P101、S100、S101

内存页的属性

- 用户可以通过mmap和mprotect来设置内存页的**读、写、执行**三种属性。
  - prot表示更改后的页面属性常用取值（宏定义），可以利用位或操作给指定内存区域同时赋予多个权限
    - PROT_NONE 指定内存区域不可访问
    - PROT_READ 指定内存区域可读
    - PROT_WRITE 指定内存区域可写
    - PROT_EXEC 指定内存区域可执行

页表的权限位

- 内存页的属性是通过页表中的权限位来表示的。



页表的权限位

- 在目前的x86处理器中，页表有不可执行位（NX）和写权限位（RW），但是**没有读权限位**。
  - **NX：不可执行位，表示是否对应内存页可执行。**
  - **global：全局位，表示对应页表项是否一直保存在TLB中。**
  - **accessed：表示对应内存页是否被访问。**
  - **user：表示对应内存页是否属于用户空间。**
  - **RW：写权限位，表示对应内存页是否可写。**
  - **present：存在标志位，表示对应内存页是否保存在物理内存中。**

页属性和页表权限位的关系

- 可以发现，内存页属性和页表权限位**并不是一一对应**的。
  - 内存页属性，4个：私有或共享、读、写、执行。
  - 页表权限位，6个：NX，global，accessed，user，RW，present。
- 在实际系统中，使用**多个页表权限位的组合**来实现**内存页属性的定义**。

页属性和页表权限位的关系

- 字母PS表示私有或共享，后面的三个数字依次表示xwr。

页面权限	页表项标志位组合	对应的属性
PAGE_NONE	GLOBAL、ACCESSED	P000、S000
PAGE_SHARED	PRESENT、RW、USER、ACCESSED、NX	S010、S011
PAGE_SHARED_EXEC	PRESENT、RW、USER、ACCESSED	P110、S111
PAGE_COPY_NOEXEC	PRESENT、USER、ACCESSED、NX	
PAGE_COPY_EXEC	PRESENT、USER、ACCESSED	P110、P111
PAGE_COPY	等同于PAGE_COPY_NOEXEC	P010、P011
PAGE_READONLY	PRESENT、USER、ACCESSED、NX	P001、S001
PAGE_READONLY_EXEC	PRESENT、USER、ACCESSED	P100、P101、S100、S101

内存页不可读的实现

- 由于页表项中没有读权限位，在大多数情况下，**内存页属性r是没有意义的**。
  - 无论r被设为1或设为0，对内存页属性没有任何影响
- 只有PAGE_NONE时，r发挥了作用，内存页是**真正不可读的**。

页面权限	页表项标志位组合	对应的属性
PAGE_NONE	GLOBAL、ACCESSED	P000、S000
PAGE_READONLY	PRESENT、USER、ACCESSED、NX	P001、S001

内存页不可读的实现

- PAGE_NONE的具体含义：由于PRESENT位置0，表示对应的物理页面实际上是不存在的，即**内存页不可访问**。所以，该内存页**不可读，不可写，也不可执行**。
- 所以，如果一个内存页是不可读的，那么这个内存页肯定是不可访问的，也就不可写、不可执行。

页面权限	页表项标志位组合	对应的属性
PAGE_NONE	GLOBAL、ACCESSED	P000、S000
PAGE_READONLY	PRESENT、USER、ACCESSED、NX	P001、S001

内存页不可读的实现

- 在实际系统中，由于页表项中没有读权限位，内存页面的读属性是通过PRESENT位来表达的。
- 所以，内存页的**读和执行是耦合的**。
  - 如果一个内存页不可读，则这个内存页肯定不可执行。
- 目前，以现有的系统无法实现不可读保护，即内存页的执行权限和读权限的分离。

内存页不可读的实现

- 为了实现**代码段不可读且可执行**，所以需要在页表项中增加一个专门的权限位，来表示内存页面的读权限。
- 可以将该权限位称为**不可读位**。具体的实现方法和不可执行位类似。
  - 对于代码段：可执行，不可读，不可写。
  - 对于数据段：不可执行，可读，可写。

内存页不可读的实现

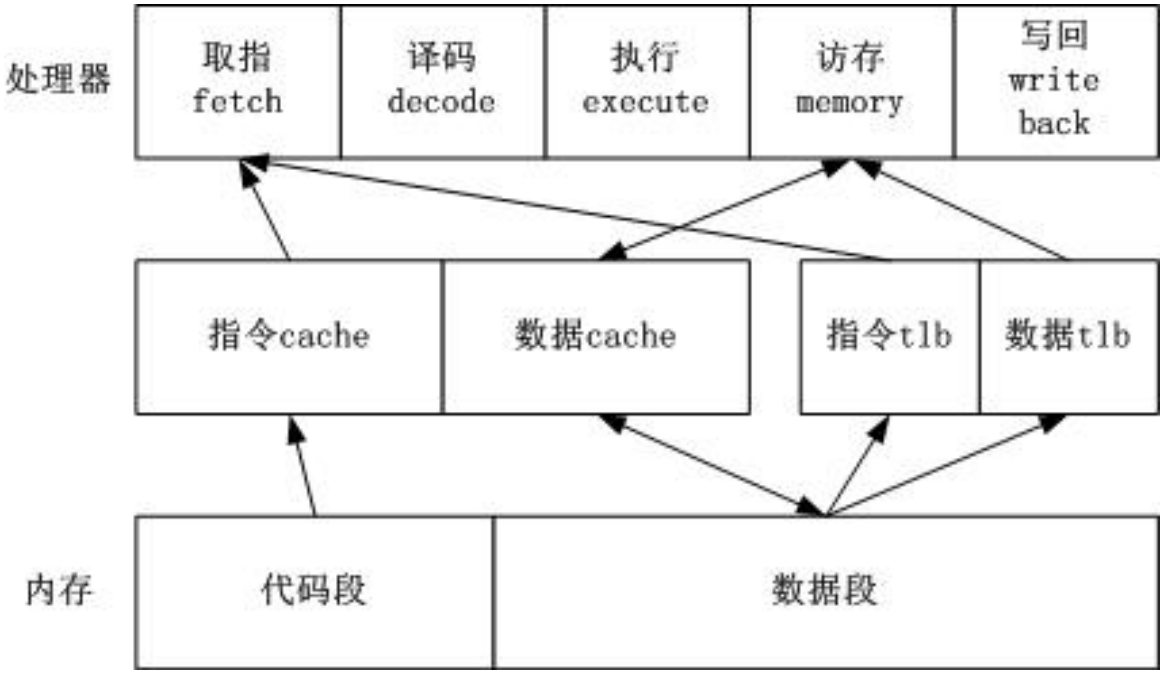
- 和不可执行位保护一样，不可读保护的实现需要计算机系统多个不同层次的支持与配合。
  - 操作系统：需要在页表中增加不可读位，需要在程序运行时管理每个内存页面的不可读位。
  - 硬件：需要在处理器中增加对不可读位的判断逻辑。
  - 可执行文件：需要在文件的代码区标记不可读，在文件的数据区标记可读。
  - 编译器：在编译生成可执行文件时，需要生成不可读的标识。

小结：指令和数据的权限分析

- 在冯诺依曼结构中，指令和数据没有任何区别。
- 在原始的计算机系统中：
  - 指令和数据没有区分，都是**可读、可写、可执行的**。
- 因此，攻击者能够直接修改程序代码，让计算机执行被篡改的恶意程序。

小结：指令和数据的权限分析

- 从处理器运行角度，可以发现处理器读取指令和读取数据的通路是不同的。
- 也就是说，在处理器层面，对数据的读写权限和对指令的执行权限是**可分割的**。



小结：指令和数据的权限分析

- 为了防止指令被攻击者恶意篡改，提出**只读**防御方法，将指令和数据分开，将程序分为代码段和数据段，**将代码段设为不可写**，即：
  - **指令是可读、不可写、可执行的。**
  - **数据是可读、可写、可执行的。**
- **但是，数据依然是可执行的。**
- 因此，攻击者可以采用代码注入攻击，将恶意代码以数据形式注入系统，然后劫持控制流，让系统执行注入的恶意代码。

小结：指令和数据的权限分析

- 为了防御代码注入攻击，提出**不可执行位保护，将数据段设为不可执行的**，即：
  - **指令是可读、不可写、可执行的。**
  - **数据是可读、可写、不可执行的。**
- **但是，代码段依然是可读的。**
- 因此，攻击者可以采用内存信息泄露+代码复用攻击，读取代码段信息，寻找可用配件的精确内存地址，构造配件链进行攻击。

小结：指令和数据的权限分析

- 为了防御内存信息泄露+代码复用攻击，提出**不可读保护，将代码段设为不可读的**，即：
  - **指令是不可读、不可写、可执行的。**
  - **数据是可读、可写、不可执行的。**
- 以上三种防御方法的提出，将**指令和数据的权限彻底分开**。
- 到目前为止，指令不可写和数据不可执行已经在真实系统中实现，而指令不可读还没有在真实系统中实现。

内容概要

- 研究背景
- 粗粒度CFI
- 绕过粗粒度CFI的代码复用攻击
- COOP和FOP
- JIT-ROP
- 不可读保护
- CPI
- 总结

安全隔离思想

- **隔离**是安全防御最常用的技术之一。
  - 用户隔离
    - 多个用户使用计算机，每一个用户都有自己的账号和密码。普通用户无法访问其他用户的数据。
  - 内存空间隔离
    - 内存空间分为内核空间和用户空间。
    - 内核空间存放操作系统源码和系统关键数据，用户空间存放应用程序数据和用户私有数据。
  - 运行权限隔离
    - 处理器有多个不同运行状态，如内核态和用户态。
    - 在用户态时，处理器不能执行特权指令，也不能访问内核空间。

CPI原理

- 安全隔离技术的基本思路是：将数据隔离，分别存放在不同的位置，访问不同位置的数据需要不同的权限，从而**阻止攻击者一次性访问所有数据**。
- **CPI**（Code Pointer Integrity，代码指针完整性）是2014年提出的一种新的防御方法，采用了**隔离**的思想。

CPI原理

- CPI将用户内存空间分为两个区域：**安全区和常规区**。
- CPI通过静态分析，将程序中数据分为**敏感数据和常规数据**。
  - 然后，将敏感数据保存在安全区，将常规数据保存在常规区。
- CPI的敏感数据实际上就是**代码指针（code pointer）**。所以，CPI的基本思想就是将代码指针从普通数据中隔离出来。

敏感数据

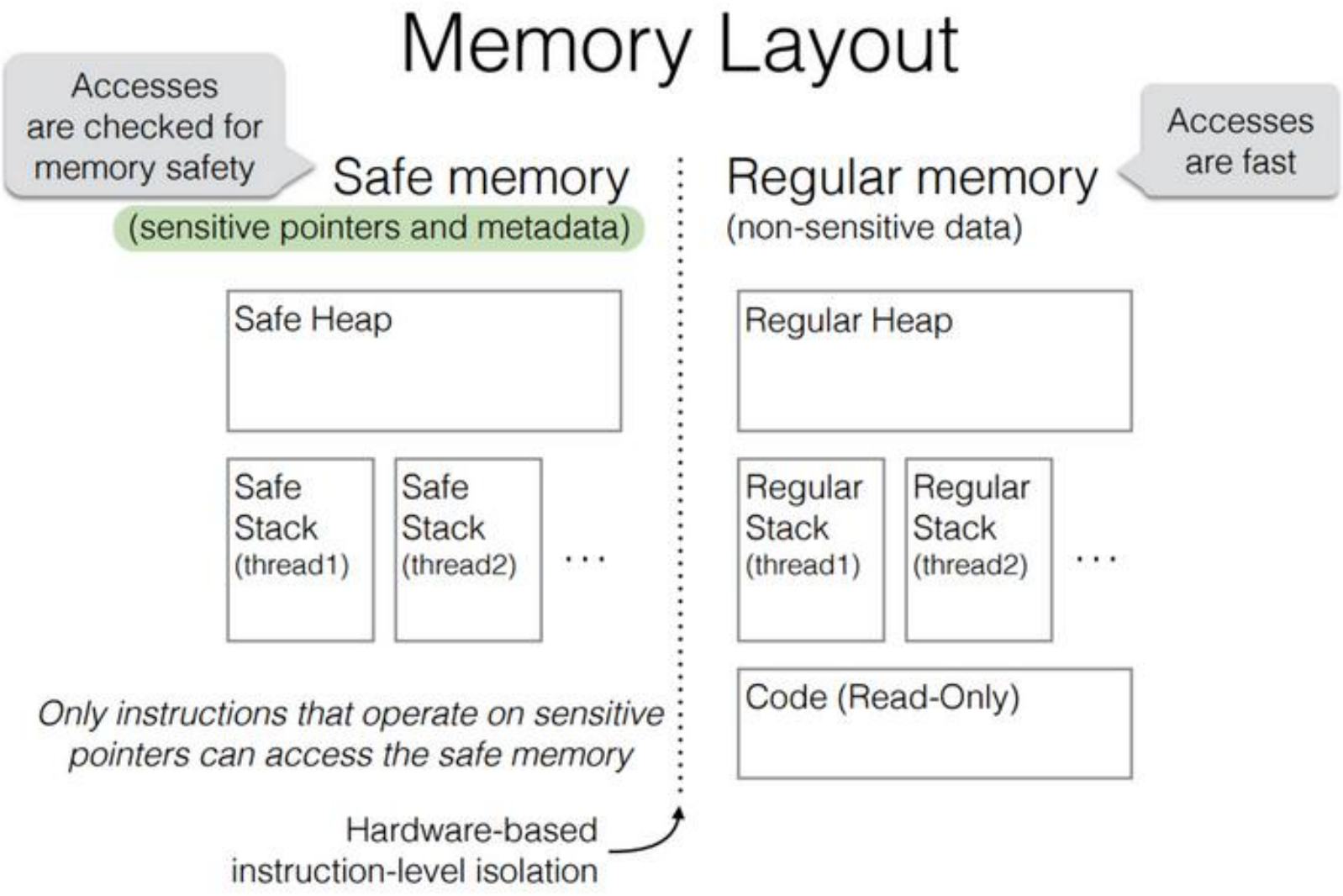
- 如果一个指针是以下类型，那么这个**指针**就是**敏感数据**：
  - 指向函数的指针
  - 指向敏感数据的指针
  - 指向内部包含敏感数据的复合类型（如数组，结构体等）的指针
  - 通用指针（如void*，char*等，或在定义struct或class前就声明的指针）
  - 用户自定义为敏感数据的类型
  - 所有在编译或运行时是隐式生成的代码指针（函数返回地址，C++虚函数表，setjmp缓存等）

敏感数据

- 根据敏感数据的定义，**所有控制流相关的数据全部属于敏感数据**。因此，攻击者想要劫持控制流，就必须要修改敏感数据。
- CPI将内存分为安全区（safe memory）和常规区（regular memory），将所有敏感数据都保存在安全区。
- 所以，**对敏感数据的保护**也就是**对安全区数据的保护**。

安全区保护

- 安全区保存的都是敏感数据，也就是控制流相关的指针。
  - 指针都是固定长度的数据（不会产生溢出），因此安全区不存在溢出漏洞。
  - 对安全区的读写操作进行动态检查，确保对安全区的读写操作都是安全的。
- 因此，攻击者无法利用安全区中内存漏洞来修改安全区中的敏感数据。
  - 想要修改安全区中的敏感数据，就必须要通过常规区中的内存漏洞。所以，**对安全区数据的保护**，实际上就是**对安全区的隔离**。


Memory Layout
Accesses are checked for memory safety
Safe memory
(sensitive pointers and metadata)
Regular memory
(non-sensitive data)
Accesses are fast
Safe Heap
Regular Heap
Safe Stack (thread1)
Safe Stack (thread2)
...
Regular Stack (thread1)
Regular Stack (thread2)
...
Code (Read-Only)
Only instructions that operate on sensitive pointers can access the safe memory
Hardware-based instruction-level isolation

- 安全区和常规区的隔离：
  - **随机化技术**：CPI确保常规区中没有指向安全区的指针，还对安全区采用了内存地址随机化防御技术。因此攻击者无法通过常规区中的数据获得安全区的内存地址，也就无法读取或修改安全区的内容。
  - **指令级隔离**：在运行时，只有读写敏感数据的指令才能访问安全区，其他指令无法访问安全区。因此，攻击者无法通过复用其他指令来修改安全区中的内容。

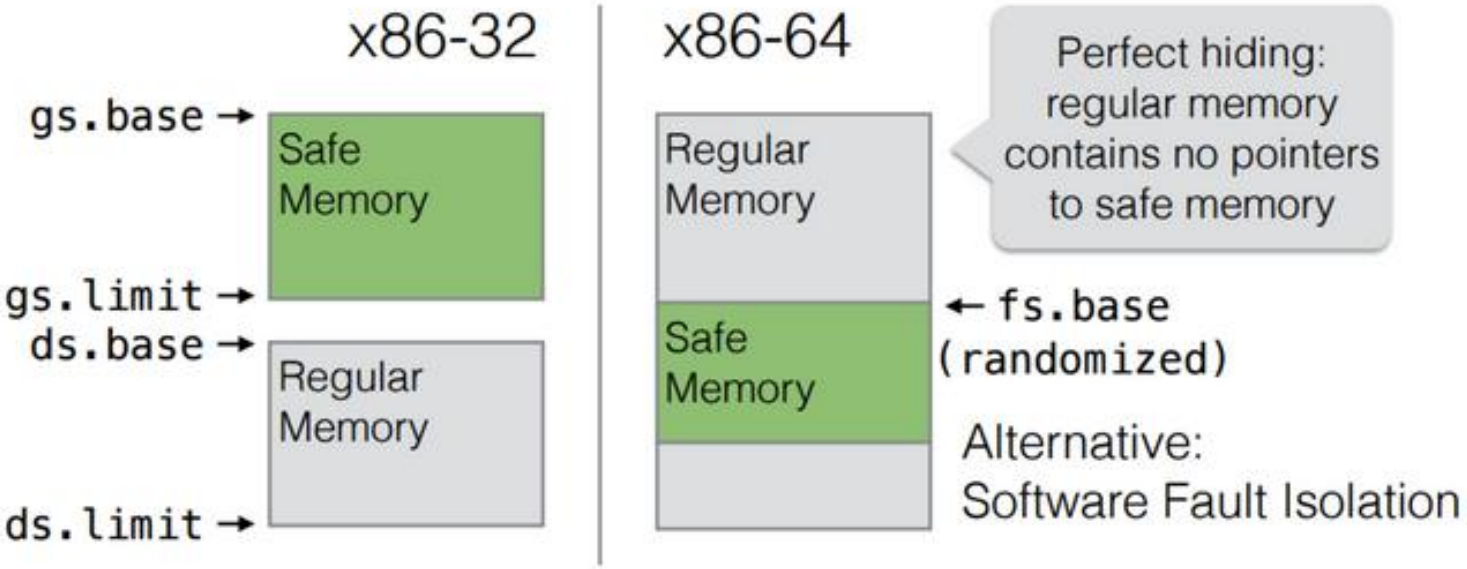
安全区保护

```
int *q = ptr + input;
*q = input2;
…
(*func_ptr)();
```

```
movl input2, q

call *%gs:func_ptr
```

Dedicated segment register, used only to access the safe memory



CPI分析

- CPI将隔离的思想引入对代码复用攻击的防御，将控制流相关的数据保存在安全区，避免攻击者通过修改安全区的数据来劫持控制流。
- 优点：
  - 比较容易实现，效率较高
  - 防御效果很好，能够防御所有控制流劫持攻击
- 缺点：
  - 安全区还是存在被修改的可能
  - 没有考虑对其他数据的保护

对CPI的攻击方法

- 安全区的防御方法是让攻击者找不到安全区的具体位置。
  - CPI中的安全区采用随机化方法，将安全区的内存位置进行随机化。
  - CPI确保常规区中没有任何指向安全区的指针。
- 但是，**安全区本身没有特殊的保护机制**。
- 一旦能够确定安全区的具体位置，攻击者就能够利用内存漏洞来修改安全区的内容。

对CPI的攻击方法

- 内存信息泄露是对付随机化防御的有效方法。
- 对CPI的一种攻击方法：
  - 攻击者利用内存信息泄露漏洞，**搜索整个内存空间**，找到CPI安全区的具体位置，然后篡改其中的内容。
  - 由于安全区的**数据特征**，研究者证明能够**通过较少次数的查询**，就能确定安全区的位置。

CPI小结

- 介绍了CPI防御方法，是一种采用了**隔离思想**的方法。
  - CPI将所有控制流相关数据隔离，保存在安全区中。
  - 然后，阻止攻击者修改安全区中内容，从而避免控制流劫持。
- CPI原理简单，复杂度低，性能损耗也较低，防御效果也很好，是一种可行的安全防御方法。
- 但是，CPI对安全区的保护还不够，可能被内存信息泄露攻击修改安全区数据。

内容概要

- 研究背景
- 粗粒度CFI
- 绕过粗粒度CFI的代码复用攻击
- COOP和FOP
- JIT-ROP
- 不可读保护
- CPI
- **总结**

总结

- 代码复用攻击及其防御是目前研究热点之一。
- 最近几年，研究者不断提出新的攻击和防御方法，攻击和防御之间相互针对性的发展。
  - 最开始，提出经典的代码复用攻击，如ROP，JOP等。
  - 然后，提出针对性的防御方法：ASLR和CFI。
  - 由于CFI的实现难度太高，提出更加简洁可行的粗粒度CFI。
  - 针对粗粒度CFI，研究者提出能够绕过粗粒度CFI的特殊的合法配件，如CP配件、EP配件等。
  - 之后，研究者提出了COOP攻击，以函数为配件，结合了CP配件和EP配件的特点。

总结

- 最近几年，研究者不断提出新的攻击和防御方法，攻击和防御之间相互针对性的发展。
  - 针对ASLR等随机化防御，提出了以JIT-ROP为代表的代码复用攻击，结合了内存信息泄露漏洞，能够破解所有随机化防御。
  - 针对代码信息泄露，提出不可读保护，让代码不可读，防止攻击者获取代码信息。
  - 研究者提出采用隔离思想的CPI，将控制流相关数据隔离，避免控制流劫持。
- 以上攻击和防御都有各自的优缺点，目前对代码复用攻击和防御的研究还在不断进行中。

经典论文推荐

- 新型代码复用攻击
  - **对粗粒度CFI的破解：** Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection, USENIX Security 2014
  - **对随机化的破解：** Just-In-Time Code Reuse:On the Effectiveness of Fine-Grained Address Space Layout Randomization, S&P 2013
  - **COOP：** Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications, S&P 2015
- 对代码复用攻击的防御
  - **CPI：** Code-Pointer Integrity, OSDI 2014


Q&A