Control Flow Integrity for COTS Binaries

APR 13TH, 2016

论文下载

Abstract

CFI作为一种重要的底层安全属性，能够抵御大多数注入和已有的攻击，包括ROP。以前的实现通常需要编译器支持或者二进制程序中的重定位或调试信息。本文提出了的技术能将CFI运用到去除了信息（stripped）的x86/Linux 二进制程序中。本文中的实验是第一个能够将CFI技术运用到复杂的共享库(glibc)中的工作。实验结果显示，文中的CFI实现能够有效对抗控制流劫持攻击，并且消除了大部分ROP gadget。为达到这个目标，作者开发了许多robust的技术，包括反汇编，静态分析，大型binary的转换。实验中成功在超过300MB的binary上测试通过。

Introduction

CFI是重要的底层安全属性，不易受到信息泄露和猜测攻击，可以抵御ROP和JOP攻击，同时CFI也为其他抵御底层代码攻击的安全机制提供了理论基础，且已被应用在软件错误隔离和不可信代码的沙盒机制中。

CFI一个重要的特点是能够直接被应用在binary上，目前很多CFI的实现都是作为一部分被集成在编译工具链中，因为需要

一些汇编代码或更高层的信息，然而这些信息在COTS(Commercial-off-the-shelf)软件中式无法获得的。也有实现依赖于程序中的重定位信息，尽管这些在Windows库中的信息是支持ASLR的，但在*NIX系统中通常依赖于PIC的随机化，故在COTS binary中并不包含重定位信息。本文的新方法则无需重定位和其他的高层信息。
本文设计的核心特点主要有：

1 模块化
   - 施加CFI时，每个共享库和可执行文件都是独立插桩的，且保证可执行文件在装载和运行时，CFI的性质是全局的，保证可执行文件和所有共享库都可以共享。

2 透明性
   - 一旦插桩对程序的内存分布（堆，栈，静态数据）产生了小的影响，都有可能导致程序崩溃或者功能异常，改变返回地址将导致在下列3种情况中崩溃。故需要保证本文中的CFI对程序保证完全透明。
     - PIC：PIC从返回地址计算静态变量的位置
     - C++异常处理：使用返回地址来定位对应异常的处理函数（更详细的，函数中的try模块）
     - 程序可能使用返回地址（或其他code pointer)来读取存在代码中的数据。

3 编译器无关 && 支持手写汇编
   - 不依赖于编译器生成代码的具体特征（如跳转表）
   - 支持手写汇编，因为在很多底层库中（如glibc）存在这种情况。

Quality of Protection

理想的CFI实现将程序的执行精确地限制在一个可执行的程序路径集合内。但是在实现中，由于间接跳转（Indirect Control-Flow）的存在，实现的CFI往往只是保守近似于理想CFI。

文中提出了一个简单的标准，average indirect target reduction(AIR),定义通过CFI技术消除的可能间接跳转目标的比例。

CFI type	Description	AIR (%)
null	no CFI protection	0.00
instr	Restrict ICFs to valid instruction boundaries	79.27
bundle	Instructions grouped into 32-byte bundles [46]. All ICFs must target the start of a bundle.	96.04
reloc	CFI based on relocation information. Indirect calls/jumps to target any location present in relocation table, returns to target a location immediately following a call.	99.13
strict	Enforces property closely matching reloc-CFI but does not require relocation info.	99.08
bin	Generalizes strict-CFI to avoid special treatment of threads and exceptions	98.86

Figure 1: CFI flavors and strengths on SPEC CPU2006.

- null: 代码段中每个字节地址都是可能的ICF目标，故AIR为0%
- instrCFI:将ICF目标限制在指令边界，可抵御跳转到指令中间的攻击
- BundleCFI:将ICF目标地址限制到16或32的倍数的地址
- relocCFI:即为CFI提出者实现的CFI技术，依赖于Binary中的重定位信息

大型和复杂的binary中包含很多异常的CALL, RET和间接跳转：

1. Return used as jumps：有时会把return指令当成跳转到某个函数来使用，只需要将地址放到栈上，然后return即可。典型的例子有线程上下文切换、信号处理
2. Return to caller function， but not a return address: 有时，return直接返回到调用者，但不是返回到返回地址处。比如 C++异常处理
3. Jumps to return addr: 如longjmp这样的函数，使用间接跳转到返回地址
4. Runtime generation of new ICF targets: 有些应用在运行时动态生成ICF，使用dlopen将附加的库添加到任何点。
5. Indirect jumps using arithmetic operations: ICF的目标地址可能是通过多个算术操作计算出来的

CFI避免对一般的ICF目标做任何假设，而是依赖于静态分析和一个保守的假设集合，因此适用于大的可执行程序和库。
由于binCFI的AIR相对于relocCFI略低，所以在BinCFI基础上实现了一个strict-CFI,但不同于BinCFI的解决异常和多线程的一般做法，它针对一小部分系统库中进行线程切换或异常回溯的特殊指令，提供了一个更宽松的策略。

Disassembly

Background

反汇编有两种基本的技术：线性反汇编和递归反汇编。
线性反汇编从一个段的第一条指令开始反汇编，一旦地址为l的一条指令被反汇编，并且其长度为k，那么下一条指令开始于l+k，重复这个过程直到这个段的末尾。

- 但是线性技术会被代码中的“缝隙（gaps）”混淆，这些缝隙可能由数据或相关的用于对齐的padding。使用线性反汇编时，这个gaps会被解码成指令，从而导致错误的反汇编结果。在像x86这样的变长指令集中，对于一条指令的错误反汇编可能导致错误地定位出下一条指令的起始地址，因此这个错误可能会一直级联下去，甚至越过gaps的边界，继续下去。

递归反汇编则类似于通过深度优先的方式构建一个程序的CFG。它起始于binary中声明的代码入口点集合。对于一个可执行文件，可能只有一个这样的特定入口点；但对于共享库文件而言，每个导出函数的起始地址都需要显式给出。这个技术起始于对入口点的指令的反汇编，接下来的方式类似于线性技术。但在遇到控制流跳转指令时，它的处理方式与线性反汇编不同。具体来说，可分为以下两步：

1 每个可由直接跳转指令确定的目标地址都加入到入口点列表中

2 反汇编停止于无条件跳转。

递归反汇编不会被代码中的gaps混淆，因此不会产生错误的反汇编结果（这还依赖于下面的若干假设，这些假设只在很少的情况下不成立（混淆代码）），但是递归技术无法反汇编出只能通过ICF跳转到达的代码

1 . call的返回地址是call后指令的地址

2 . 所有条件跳转后的代码都是有效代码

3 . 所有条件和无条件直接跳转的目标地址上的代码是有效代码

修正递归反汇编错误结果的一个方法就是，显式给出对所有ICF可达的目标地址列表，这些信息可由程序中的重定位信息获得。但在stripped binary中，由于不包含重定位信息，则无法修正这个错误。

Our Disassembly Technique

首先使用线性反汇编技术反汇编出整个binary，然后再做错误检查（错误检查主要依赖于递归反汇编），最后一步进行错误修正，识别和标记出表现为gaps的反汇编代码区域。

错误检查步骤依赖于以下几个检查点：

- 非法opcode:有的字节不对应于任何指令，故对这些字节进行反汇编将导致错误。由于x86机器指令相当紧凑，这种情况很少发生，但一旦发生，则是反汇编错误的一个明显标志。
- 直接跳转跳出了当前module：跨模块跳转需要使用PLT和GOT这样特殊的结构，同时需要ICF指令。因此任何直接跳转跳出了当前模块都标志着反汇编错误的发生。
- 直接跳转跳到了指令中间：发生这种情况的原因是：
  - 错误地反汇编了target
  - 错误地反汇编了跳转指令
    - 若错误点的source或target附近还有其他的错误，则这些其他错误的类型能够辅助我们确定错误点的错误类型是前述两种原因的哪一种。
    - 若错误点附近没有错误发生，则两种原因都有可能发生线性反汇编的错误是由gaps引起的，故错误的反汇编结果预示着这里存在gap，我们需要找到这个gap的begin和end。
- 找begin: 从错误的反汇编指令开始，反向回溯，寻找最近的无条件跳转指令。若还有另一个错误在gap前若干字节处（也即对当前gap找到的这个无条件跳转可能也是一个错误反汇编的结果？），则继续往前找下一个无条件跳转。（从前面的递归反汇编的假设可以看到，gap只可能存在于无条件跳转之后）。

- 找end：这里依赖于之前静态分析的结果，大于错误反汇编处的最小的ICF的target地址被视为这个gap的end，如果在该target后若干字节内仍有反汇编错误，则将gap延伸到下一个ICF target。

在错误修正这步结束后，所有识别出的反汇编错误都在gap中。这时，再重新反汇编，这次则略过gap，若这次反汇编没有识别出新的错误，则反汇编结束。否则，重复前述过程。尽管这很低效，但也很简单。。。在实现中也很少出现需要重复的情况。

Indirect Control Flow Analysis

使用静态分析技术来识别可能的ICF target。这里将ICF target分别若干类，分别设计不同的分析方法来计算他们：

- Code pointer constants(CK， 代码指针常量):由编译时计算出来的代码地址组成
- Computed code address(CC, 需计算的代码地址）：包括运行时计算出来的代码地址
- Exception handling addresses(EH，异常处理地址）：包括用户处理异常的代码地址
- Exported symbol addresses(ES，导出符号地址）：包括导出表中的函数地址
- Return addresses（RA，返回地址）：包括call指令的后面的（下一条？）代码地址

Indentifying Code Pointer Constants(CK)

很难将代码指针与其他代码中的数据区分开，所以这里用了个保守的策略：任何长得像代码指针的常数，只要通过后面两个测试，就加入CK中：

1 这个常数落在当前模块的代码地址段内
2 它指向反汇编代码的一个指令边界

一个模块是不知道其他模块编译时的地址信息的，因此我们可以检查常数是否落在当前模块的代码地址段内。对于共享库文件，绝对地址是不知道的，所以只能检查常数是否代表一个从代码段基址开始的合法的偏移值，这个偏移也可能与共享库的GOT关联，所以这里的合法性检查也需要考虑这个。

扫描所有代码段和数据段，找出可能的CK值。由于x86中32-bit值不需要按地址的4字节边界对齐，故在代码段和数据段上使用一个4字节的滑动窗口来查找CK。

Identifying Computed Code Pointers(CC)

这里CC分析没有CK那么保守，因为大多数代码都是由高级语言转换而来，对代码指针的任意指针计算不是有意义的。及时对于手写汇编，考虑到可维护性，依赖性和可一致性，程序员都不会随便对代码指针做奇怪的计算。所以这里只需要支持在时间中更有可能出现的对代码指针进行计算的代码上下文。实际上，文中观察到的唯一一种对代码指针进行计算的上下文就是跳转表。

跳转表在C和C++程序中的switch语句处最常见。若这是CC仅有的来源，那只需根据编译器典型的翻译switch语句的惯例来开发一个简单的方法即可，但文中还希望处理底层库函数中手写汇编的情况。这里给出可用于识别跳转表的通用属性：

- 跳转表target目标是个内部过程（intra-procedural）:ICF跳转指令和ICF的target在同一个函数中（不需要函数边界，只需要保守估算）
- 目标地址使用简单的运算计算而来，比如加法和乘法
- 除了一个值是下标（index），计算过程中所有其他的值都是代码段或数据段中的常量

- 所有计算都在一个固定的指令集窗口内完成，实现中用的是50

将这个计算可能CC target的静态分析技术分为三步：

1. 识别函数边界，构造CFG。在缺少完整符号表信息的前提下，很难识别出所有的函数边界。这里使用导出函数符号表的信息，将两个连续的导出函数符号之间的代码区域近似地视为一个函数（这个近似是保守的，因为可能没有导出函数），对每个这样的区域生成一个CFG。
2. 识别出间接跳转指令，并从这些间接跳转指令起，沿CFG反向遍历。每条反向路径都要跟随下去，并且对于每条路径，都追踪生成一个数据以来的链，通过这个链计算出一个间接跳转target的表达式。这个表达式的形式是*(CE1 + Ind) + CE2,*其中*CE1*和*CE2*一定是常量,*ind*是下标，表示地址解引用。某些情况下能识别出Ind的范围，但文中并未实现，因为index可能从其他函数过来。这里假设Ind从0开始.
3. 枚举index可能的值，计算由每个可能index算出来的target，检查这个target是否在当前region中（即第一步识别出来的）。具体来说，首先检查CE1+Ind是否在当前模块的数据段或代码段中，若在，从这个地址中提取出相应值，再拿这个值于CE2想家，检查结果是否落在当前region内，若在，则target加入CC集合，若这两步检查中任何一步失败了，Ind都是不合法的。
    - Ind从1开始，分别向（增加和减少）两侧枚举Ind的值，知道遇到一个非法的target时停止。