

控制流完整性简介

原创 Smilence_Isy 最后发布于2019-01-31 23:33:28 阅读数 779 ☆ 收藏

控制流完整性概述

- 0x00. 基础知识
- 0x01. 控制流完整性发展历程
- 0x02. CFI 机制的比较
- 0x03. CFI 的应用范围、发展前景
- 0x04. 个人想法
- 0x05. 参考文献

0x00. 基础知识

控制流完整性 (Control-Flow Integrity) 是一种针对控制流劫持攻击的防御方法。控制流的转移是以跳转指令为基础的，因此在这一节先介绍跳转指令。在汇编语言中,根据寻址方式的差异可以分为间接和直接两种跳转指令。
直接跳转指令的形式一般如下：

```
CALL 0x1060000F
```

在程序执行到这条语句时，会将指令寄存器的值替换为 0x1060000F。类似于这种在指令中直接给出跳转地址的寻址方式就叫做直接转移。在高级语言中，else、静态函数调用这种跳转目标相对固定的语句就会被编译为直接跳转指令。

间接跳转指令则是使用数据寻址方式间接的指出转移地址，比如：

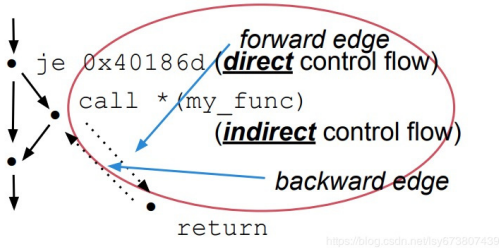
```
JMP EBX
```

执行完这条指令之后，指令寄存器的值就被替换为EBX寄存器的值。与直接转移不同，间接转移的跳转地址是在执行过程中动态决定的。高级语言中可能有函数指针等等。

以上是按照寻址方式进行分类，在控制流完整性中还有一个比较特殊的分类方式——前向和后向转移。

前向转移指的是将控制权定向到程序中一个新位置的转移方式，比如使用 CALL 指令调用函数。

后向转移是将控制权返回到先前位置，最常见的就是RET指令。



对于控制流完整性策略而言，直接跳转指令的地址在编译时就已固定，难以被攻击者更改，无需消耗大量资源对其进行检查。而间接跳转指令的地址有意篡改的可能，因此是检测的重点。间接跳转又分为前向间接跳转（如通过指针的函数调用）和后向间接跳转（如RET指令），几乎所有的控制流完整性策略都对这两者进行检验。

0x01. 控制流完整性发展历程

控制流完整性（以下简称CFI）是一个随着控制流劫持攻击发展而不断演进的策略。

- 20世纪80年代 出现溢出攻击
在20世纪80年代，溢出攻击首次进入大众视野。1988年的Morris蠕虫利用了 Unix 系统的 finger 服务的缓冲区溢出漏洞，使得溢出攻击被广泛知晓。
- 1996年 第一篇具有学术价值的缓冲区溢出论文发表
随着对溢出类漏洞的利用逐渐增多，1996年 Aleph One 在 Phrack 杂志上发表了一篇名为《Smashing the Stack for Fun and Profit》的论文，详细描述了通过覆盖返回地址来劫持控制流的方法。

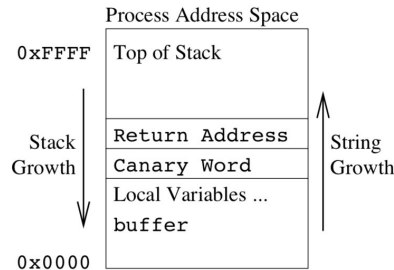
- 1998年 出现StackGuard^[8]

1998年, Cowan、C.Pu等人在 USENIX 上提出了一种检测和防止缓冲区溢出的自适应技术——StackGuard。StackGuard 首次使用 Canary 保护实现, 它于1997年作为GCC的一个扩展发布。

StackGuard简要介绍 (参考文献[8]):

StackGuard是一个编译器扩展, 用于检测并阻止对堆栈的缓冲区溢出攻击。其主要有两种工作模式, 一种是在函数返回之前检测返回地址的, 一种则拒绝写入返回地址来阻止对地址的动态修改 (MemGuard)。

第一种方法 在栈中存放的返回地址附近放置一个"Canary"值, 在函数返回之前检测Canary是否一致再跳转。函数执行过程中的栈结构如下:



下面两段汇编代码展示了 Canary 模式的具体操作。第一段代码被添加在函数执行前, %gs:0x14中存储的是一个随机数, 这个随机数被赋给 EAX 将 EAX 的值压栈。第二段代码是函数返回前的操作, 其将该随机数弹栈, 并与原来的数异或比较, 若相同则跳转, 否则执行异常。(这两段代码属改进版本 SSP)

```
1 | 0x080483a5 <main+17>:  mov    %gs:0x14,%eax
2 | 0x080483ab <main+23>:  mov    %eax,0xffffffff8(%ebp)
3 | 0x080483ae <main+26>:  xor    %eax,%eax

1 | 0x080483c0 <main+44>:  mov    0xffffffff8(%ebp),%edx
2 | 0x080483c3 <main+47>:  xor    %gs:0x14,%edx
3 | 0x080483ca <main+54>:  je     0x80483d1 <main+61>
4 | 0x080483cc <main+56>:  call   0x80482fc <__stack_chk_fail@
```

但是这种方法存在限制, 因为其假设 Canary 不改变的情况下返回地址就不会被改变, 也就是攻击者只会线性、顺序的写入数据, 但实际上由于这个假设不一定成立。文章中还提出了两种可能的攻击, 一种是构造满足对齐要求的数组,使得Canary所在的位为空,这样可以避免覆盖Canary, 另外Canary, 比如猜测和暴力破解的方式。

第二种方法是阻止对函数返回地址的写入。它基于 MemGuard, 一种允许将内存中的特定字设置为只读,只能用特定的API写入的方法保护重要数据。MemGuard 通常将重要数据所在的整个虚页设置为只读, 在对于其他不受保护的数据进行写入时,它采用模拟写入(开辟一块区域把这些写入内存后再一并写入)的方式。这种方式造成的性能开销极大, 尤其当该字位于栈顶这类写入频繁的区域时。文章中提出一种优化方法, 即使用调试寄存器来保护return address, 避免栈顶所在页被设置为只读。

第一种方法更加简洁、效率更高, **第二种方法**安全性更高, 但开销也大。StackGuard 在运行时会自适应的选择当前运行环境下更优的方法。

- 2001年 ASLR (地址空间布局随机化)的提出

2001年, ASLR 作为 Linux 内核的一个补丁提出。它通过对堆、栈、共享库映射等线性区布局的随机化增加攻击者预测目的地址的难度。绕过ASLR喷射,攻击未启用ASLR的模块等等。

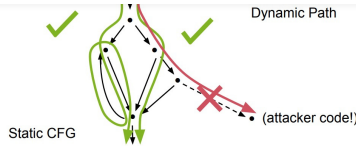
- 2004年 Windows XP Service Pack 2 实现DEP (数据执行保护)

DEP 是 Windows 实现的数据执行保护, 还存在其他比 Windwos 实现更早的系统。它通过将内存页设置为写或者执行, 使攻击者无法向缓冲区注入可执行代码。绕过 DEP 的典型攻击有 ROP 等利用程序中原有的代码段组合进行攻击的方法。

- 2005年 控制流完整性机制的首次提出

在DEP、ASLR、Canary 等技术陆续提出以后, 用于绕过这些防御机制的攻击手段也随之而来。控制流完整性 (CFI) 的提出初衷是为彻底杜绝攻击。2005年 CCS 发表了一篇名为《Control-Flow Integrity》的文章, 正式提出了 CFI 的概念。

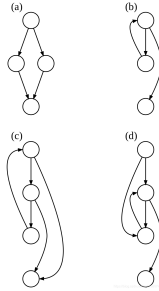
CFI防御机制的核心思想是限制程序运行中的控制流转移, 使其始终处于原有的控制流程图所限定的范围内 (如下图)。主要分为两个阶段, 一是代码程序分析得到控制流图 (CFG), 获取转移指令目标地址的列表; 二是运行时检验转移指令的目标地址是否与列表中的相对应。控制流劫持



- 2010年 CFI机制的发展与改进 (参考文献[4])

原始的 CFI 机制是对所有的间接转移指令进行检查, 确保其只能跳转到预定的目标地址, 但这样开销过大。因此又提出了对 CFI 机制的改进一段 CFI 分析的过程。

首先是 CFG 的构建, 控制流图 (Control-Flow Graph) 是基于静态分析的用图的方式来表达程序的执行路径。下图展示了几个普通的 CFG, 圆圈则表示普通指令。CFI 中的 CFG 构建只考虑将可能受到攻击的间接调用、间接跳转和 RET 指令作为边。



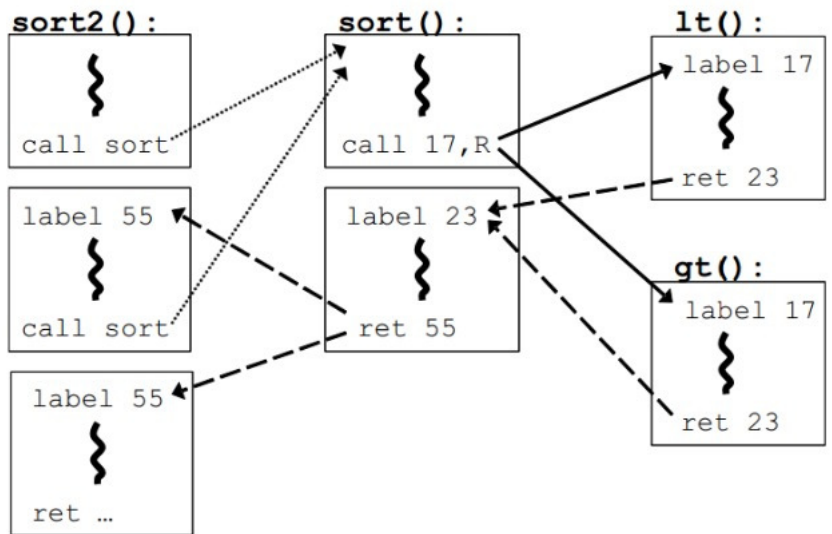
构建完CFG后就是动态检测过程, CFI 通过二进制代码重写技术在间接调用前和返回前插入标识符 ID 和 ID_check, 通过比对两者的值是否一致判断被劫持。

下图左侧展示了一段C语言程序, sort2() 函数调用两次 sort() 函数, sort() 函数又分别以函数指针的方式调用了 lt() 与 gt()。右图以方框代表基本块, 展示了该代码片段的 CFG。其中细虚线表示直接调用, 不需要检查。实线代表间接调用, 粗虚线代表返回指令, 这两者的目标地址需要检查。lt() 和 gt() 两个函数的地址是不同的, 但它们具有相同的标识符17, 这就是为了性能优化而将相似的目标地址放在同一个集合里, 在检查时如果目这个集合即可通过。而这两个函数的 RET 地址相同, 标识符都为23。

```
bool lt(int x, int y) {
    return x < y;
}

bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```

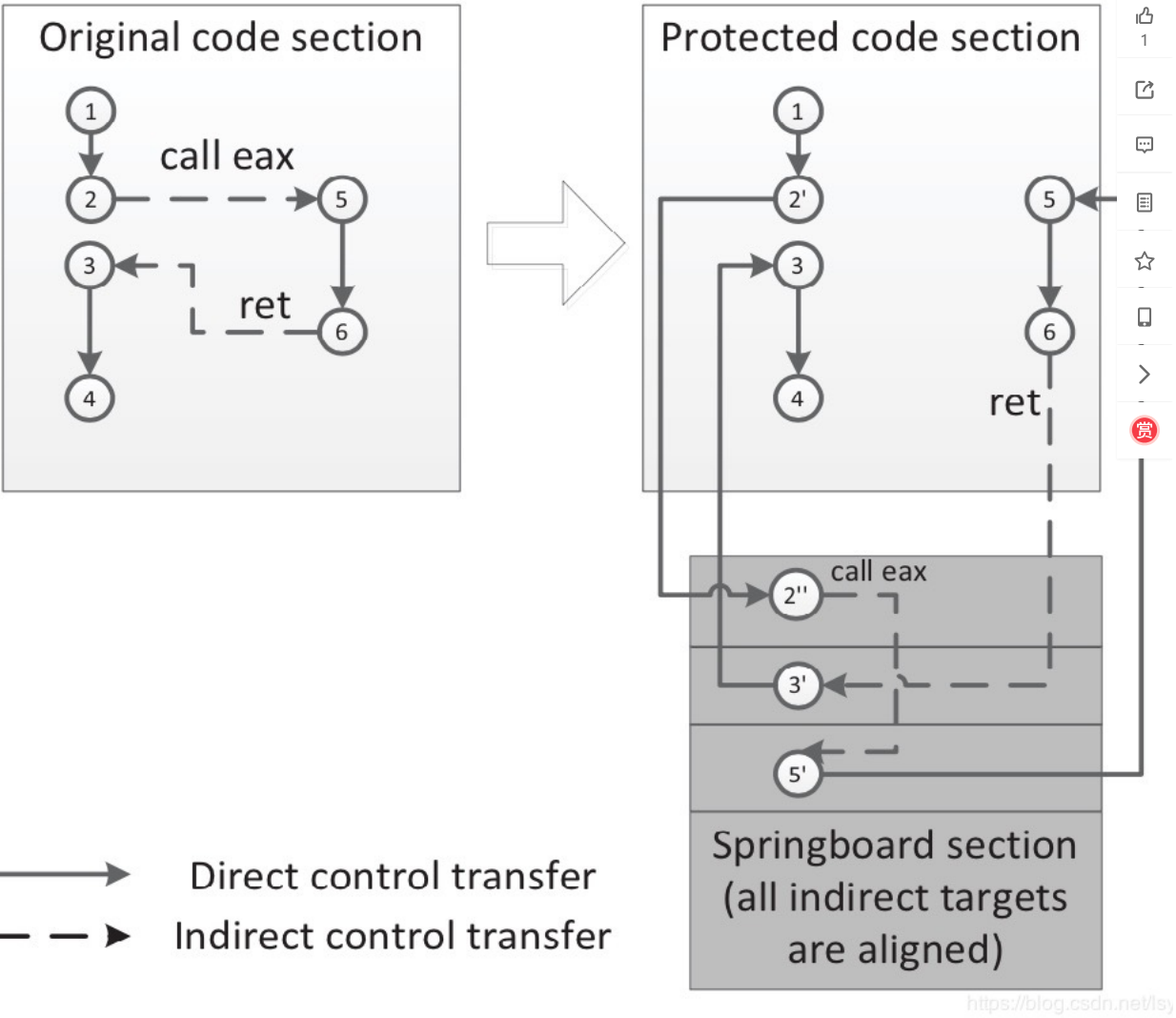


<https://blog.csdn.net/isy673>

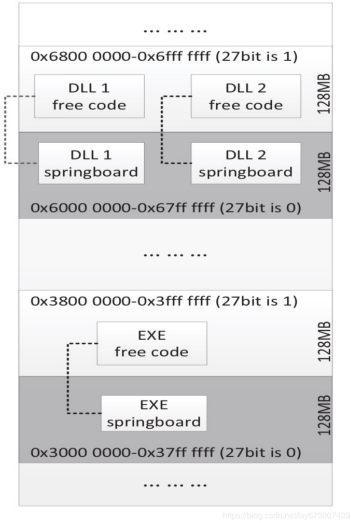
以上就是一种粗粒度的CFI, 它将多个不同的目标地址放在一个集合, 减少需要分配和检查的标签数量。但降低性能开销的同时, 也牺牲了检查的

- 2013年 CCFIR的提出 (参考文献[5])

CFI 被提出后因其开销太大并没有被广泛应用。于是在2013年又提出了CFI 的低精度版本 **CCFIR**, 在同一年提出的还有binCFI,ModularCFI等等目标集合划分为三类, 间接调用的目标地址被归为一类, RET 指令的目标地址被归为两类, 一类是敏感库函数 (比如libc中的system函数), 数。下面以图中的例子来说明CCFIR的具体原理:



图左是原始的控制流，右边是 CCFIR 机制下的控制流。CCFIR 提出了通过 Springboard 段 (特殊内存段 右下方灰色部分) 存放间接转移目标地址。控制流中，5 和 3 节点分别是 CALL EAX 和 RET 这两个间接转移指令的目标地址，都被存储于 Springboard 段中。在程序执行到节点 2' 时，会检查跳转地址是否位于 Springboard 段，是则跳转，否则报错，从 6 跳回 3 时同理。Springboard 段的内存布局如下图所示，通过将某一位设置成 0/1 来区分普通段和 Springboard 段。在检测时检查某一个目标是否在 Springboard 段，测某一位的值即可。



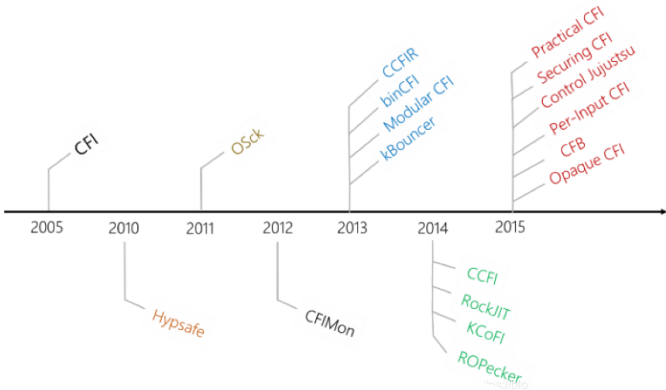
进一步地，CCFIR 将目标地址分为三类，其又通过将地址中的不同位设置为 0/1 来区分。第 27bit 为 0 表示是 Springboard 段，第 3 位为 0 属于 ret 地址，并通过 26 位区分是敏感函数地址还是普通函数地址。

Executable	Bits				Meaning
	27	26	3	2-0	
no	*	*	*	***	Non-executable section
yes	1	*	*	***	Normal code section
yes	0	*	*	!000	Springboard's invalid entry
yes	0	*	1	000	Springboard's function pointer
yes	0	1	0	000	Springboard's sensitive return s
yes	0	0	0	000	Springboard's normal return stu

CCFIR 的主要贡献在于以将 CFI 机制投入生产实际为目的对其进行了改进，效率有了很大的提升。

- 2014年 Google 的 CFI 实践 (参考文献[9])
随着对堆栈的保护越来越完善，出现了很多针对非堆栈的前向转移攻击,尤其是针对 CALL 指令等。例如利用 UAF 漏洞覆盖 vtable 指针等等。这 Google 将 CFI 机制应用到编译器中的实践。
Vtable Verification (VTV) 主要是对 vtable 调用进行检测，VTV在每个调用点验证用于虚拟调用的 vtable 指针的合法性。
Indirect Function Call Checker (IFCC) 基于 LLVM。它通过为间接调用目标生成跳转表并在间接调用点添加代码来转换函数指针来保护间接调用们指向跳转表条目。任何未指向相应表的函数指针都被视为CFI违规。
Indirect Function Call Sanitizer (FSan) 也基于 LLVM，是一个可选的间接调用检查器。
- 2014-2015年 其他CFI
基于前述方案的缺陷，又提出了上下文敏感的 CFI (Context sensitive CFI) 机制。它依赖于上下文敏感的静态分析，将 CFI 不变量和 CFG 中的系到一起，运行时在执行路径上强制执行这些不变量。2014 年的论文《Complete Control-Flow Integrity for Commodity Operating System 操作系统的内核上实现了 CFI，使之免受控制流劫持等攻击，这个系统被称为 KCoFI。他们在基于标签的控制流间接转移保护的基础上，加入一个软件层，负责保护一些关键的操作系统数据结构和监控操作系统进行的所有底层状态操作。（这个系统加入了实时监控系统底层状态操作，如果是下，性能表现比较差）2015 年论文：《CCFI: Cryptographically Enforced Control Flow Integrity》，提出了一种通过对代码指针加密的方法来增护。这个观点出发点是好的，但是在大部分硬件效率跟不上的情况下，很难在现实中运用。
- 针对前面几种粗粒度CFI提出的攻击方式
当然,攻防是相对应的，粗粒度的CFI不够安全一直是公认的。2014 年的论文《Out of Control: Overcoming Control-Flow Integrity》中就针对到了一种攻击手段。他们利用了两种特殊的 Gadget：entry point(EP) gadget 和 call site(CS) gadget，来绕开粗粒度 CFI 机制的防御。
2015 年的论文《Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks》，也提到了对 CFI 保护下的栈的攻击文发表前，通过影子堆栈（Shadow Stack）来检测函数返回目标，再加上 DEP 和 ASLR 的保护，栈应该会变得非常安全，但是事实并非如此。到了三种攻击手段，一是利用堆上的漏洞来破坏栈上的 callee saved 寄存器 保 存 区 域，使得 callee saved 寄存器被劫持；二是利用用户空间和行上下文切换的问题，来劫持 sysenter 指令，使控制跳转到攻击者想跳转的位置；

下图是网络上一张对 CFI 机制发展历史的总结，作为参考。



0x02. CFI 机制的比较

- 2017年 《Control-flow integrity: Precision, security, and performance》对现有CFI机制的安全性和开销作出了系统的评价 (参考文献[6])

制的开销低于 5% 的时候，这项机制才是受到行业从业者欢迎并且真正能投入使用的。从下图中可以看出，大多数 CFI 机制的平均开销符合 Lockdown，CCFI 这种开销过高的机制。

Benchmark Version Options	Measured Performance						Reported Performance					
	VTV 3.7	LLVM-CFI 3.9	VTI LTO	CFI LTO	CFGuard LTO	π CFI ntc	VTV LTO	VTI LTO	π CFI LTO	IFCC LTO	MCFI LTO	PathArmor Lockdown
400.perbench(C)		2.4				8.2						150.0
401.bzip2(C)		-0.7			-0.3	1.2			5.0	1.9	5.0	8.0
403.gecc(C)		CF			6.1	10.5			4.5	4.5	4.5	50.0
429.mcf(C)		3.6			0.5	4.0			4.0	4.0	4.0	2.0
445.gobmk(C)		0.2			-0.2	11.4			7.5	7.0	7.0	43.0
456.hmmcr(C)		0.1			0.7	0.1			0.0	0.0	0.0	3.0
458.sjeng(C)		1.6			3.4	8.4			5.0	5.0	5.0	80.0
464.h264ref(C)		5.3			5.4	7.9			6.0	6.0	6.0	43.0
462.libquantum(C)		-6.9			-3.0	-1.0			-0.3	0.0	0.0	5.0
471.omnibenchpp(+)		5.8	-1.9	CF	3.8	6.7			8.0	1.2	5.0	17.0
473.astar(+)		3.6	-0.3	0.9	1.6	2.0			2.4	0.1	4.0	75.0
483.xalancbmk(+)		24.0	7.1	7.2	3.7	10.3			19.2	1.4	7.0	170.0
410.bwaves(F)												1.0
416.gamess(F)												11.0
433.mile(C)		0.2			2.0	-1.7			2.0	2.0	2.0	8.0
434.zeusmp(F)												0.0
435.gromacs(C,F)												1.0
436.cactusADM(C,F)												0.0
437.leslie3d(F)												1.0
444.namd(+)		-0.1	-0.2	0.1	-0.3	0.1			-0.5	-0.2	-0.5	3.0
447.dealIII(+)		0.7	CF	7.9	CF	-0.1			4.5	-2.2	4.5	12.0
450.soplex(+)		0.5	0.5	-0.3	-0.6	2.3			-0.7	-4.0	-4.0	90.0
453.povray(+)		-0.6	1.5	8.9	2.0	10.8			10.5	0.2	10.0	3.0
454.calculix(C,F)						11.3						7.0
459.gemsFDTD(F)												19.0
465.totolo(F)												2.0
470.lbm(C)		-0.2			4.2	-0.2			1.0	1.0	1.0	8.0
482.sphinx3(C)		-0.8			-0.1	0.7			1.5	1.5	1.5	3.0
Geo Mean	4.6	1.1	4.4	1.3	2.3	4.0	5.8	9.6	0.5	3.2	-0.3	20.0
												45.0
												2.6
												8.5

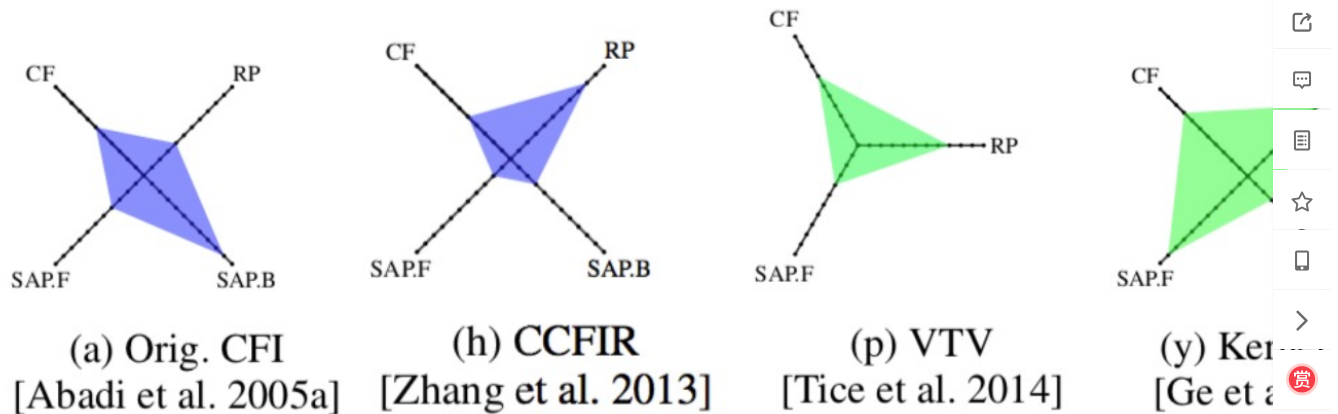
2. 安全性能评估

对 CFI 机制安全性能的评估标准此前都是 AIR (平均间接目标缩减量)，也就是经过CFI机制的保护后减少的可攻击目标数量。但这个机制参 CFI机制 通常都能降低99%以上的目标，无法体现不同CFI机制的区别。

本文提出了一种定性的安全评价方法：它将安全性能的比较分为四个方面，一是支持的控制流传输种类，比如前向后向、间接返回等等，用数值，用1-10来区别，10为最高分，用RP表示。SAP.F是对前向控制流的静态分析精度，SAP.B是对后向控制流的分析精度。文中给出了结果，这里前面提到的来举例：

左一是原始CFI，可以看到它的性能较差但精度较高。左二是CCFIR，它的性能(RP)值很高，但无论是前向还是后向的精度都非常差，因为它只支持

保护。



<https://blog.csdn.net/Isy673807439/article/details/86666684>

0x03. CFI 的应用范围、发展前景

- Clang: <https://en.wikipedia.org/wiki/Clang>
- Microsoft's Control Flow Guard: https://en.wikipedia.org/wiki/Control-flow_integrity
- Return Flow Guard: <https://xlab.tencent.com/en/2016/11/02/return-flow-guard/>
- Google's Indirect Function-Call Checks
- Reuse Attack Protector: https://grsecurity.net/rap_faq.php

以上都是 CFI 机制的一些现有应用，当然 Canary 机制也可以算作 CFI 机制的一种。

0x04. 个人想法

- 粗粒度的CFI安全性不够
- 细粒度的CFI性能开销太大
- 出现很多CFI无法防护的攻击——Data oriented programming等

以上对于 CFI 机制的介绍偏策略。因为随着软硬件的发展，不同的硬件、操作系统、编译器甚至语言都会有不同的实现方式，而且没有哪一种是完美的，能在实际使用中还需要用户自行抉择。

0x05. 参考文献

1. 吴世忠, 郭涛, 董国伟, 张普含, 软件漏洞分析技术, 科学出版社, 2014.
2. Bryant & O'Hallaron, Computer Systems: A Programmer's Perspective (2 ed.), Pearson Education, 2011. 中译本: 深入理解计算机系统, 机械工业出版社, 2011.
3. David Brumley and Vyas Sekar, Introduction to Computer Security (18487/15487), 2015
4. Control-Flow Integrity Principles, Implementations, and Applications[J]. ACM transactions on information and system security, 2010, 13(4): 1-24. (A类 TISSEC)
5. Zhang, Chao, Wei, Tao, Chen, Zhaofeng, et al. Practical Control Flow Integrity and Randomization for Binary Executables[C]. // 2013 IEEE Symposium on Security and Privacy: SP 2013, Berkeley, California, USA, 19-22 May 2013. 2013: 559-573. (A类 S&P)
6. Burow N, Carr S A, Nash J, et al. Control-flow integrity: Precision, security, and performance[J]. ACM Computing Surveys (CSUR), 2017, 50(4): 1-24. (A类 CSUR)
7. Carlini N, Barresi A, Payer M, et al. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity[C]. // USENIX Security Symposium. 2016: 161-176. (A类 USENIX)
8. Cowan C, Pu C, Maier D, et al. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks[C]. // USENIX Security Symposium. 1998, 98: 63-78. (A类 USENIX)
9. Tice C, Roeder T, Collingbourne P, et al. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM[C]. // USENIX Security Symposium. 2014: 941-955. (A类 USENIX)

1

>

賞

国外买点卡

咦！这篇文章不孬，夸两句吧...

၁

举报



CSDN

首页 博客 学院 下载 论坛 问答 活动 专题 招聘 APP VIP会员 搜博主文章

Q

创作

今年，找出C/C++，为了不给人家误导，官网J相关、圈内对反，以及中过C/C++的几组立... 博文 来自： 后脱

超全Python图像处理讲解（多图预警）

阅读数 1万+

文章目录Pillow模块讲解一、Image模块1.1、打开图片和显示图片1.2、创建一个简... 博文 来自： ZackSock...

为什么猝死的都是程序员，基本上不见产品经理猝死呢？

阅读数 14万+

相信大家时不时听到程序员猝死的消息，但是基本上听不到产品经理猝死的消息，这... 博文 来自： 曹银飞的...

毕业5年，我问遍了身边的大佬，总结了他们的学习方法

阅读数 16万+

我问了身边10个大佬，总结了他们的学习方法，原来成功都是有迹可循的。 博文 来自： 敖丙

推荐10个堪称神器的学习网站

阅读数 26万+

每天都会收到很多读者的私信，问我：“二哥，有什么推荐的学习网站吗？最近很浮... 博文 来自： 沉默王二

强烈推荐10本程序员必读的书

阅读数 8万+

很遗憾，这个春节注定是刻骨铭心的，新型冠状病毒让每个人的神经都是紧绷的。那... 博文 来自： 沉默王二

为什么说程序员做外包没前途？

阅读数 10万+

之前做过不到3个月的外包，2020的第一天就被释放了，2019年还剩1天，我从外包... 博文 来自： dotNet全...

柏林艺术家用行为艺术骇了一次谷歌地图

阅读数 2637

用一辆装了99部智能手机的手拉车，一位柏林艺术家在一条空荡荡的街道上，骇了一... 博文 来自： 德国IT那...

B 站上有哪些很好的学习资源？

阅读数 13万+

哇说起B站，在小九眼里就是宝藏般的存在，放年假宅在家时一天刷6、7个小时不在话... 博文 来自： 九章算法...

给新手程序员的一点学习建议

阅读数 4181

我是一个有几年经验的程序员，之前对于自己的发展却是一头雾水，不知道主流技术... 博文 来自： JAVA圈的...

新来个技术总监，禁止我们使用Lombok！

阅读数 3万+

我有个学弟，在一家小型互联网公司做Java后端开发，最近他们公司新来了一个技术... 博文 来自： HollisChu...

字节跳动的技术架构

阅读数 3万+

字节跳动创立于2012年3月，到目前仅4年时间。从十几个工程师开始研发，到上百人... 博文 来自： 作一个独...

在三线城市工作爽吗？

阅读数 8万+

我是一名程序员，从正值青春年华的 24 岁回到三线城市洛阳工作，至今已经 6 年有... 博文 来自： 沉默王二

这些插件太强了，Chrome 必装！尤其程序员！

阅读数 2万+

推荐 10 款我自己珍藏的 Chrome 浏览器插件 博文 来自： 沉默王二

抱歉，我觉得程序员副业赚钱并不靠谱

阅读数 1万+

我最近看到不少关于程序员副业赚钱的文章，其中出的点子有这些：1. 在网上找项目... 博文 来自： 码农翻身

@程序员：GitHub这个项目快薅羊毛

阅读数 1万+

今天下午在朋友圈看到很多人都在发github的羊毛，一时没明白是怎么回事。后来上... 博文 来自： dotNet全...

删库了，我们一定要跑路吗？

阅读数 1万+

在工作中，我们误删数据或者数据库，我们一定需要跑路吗？我看未必，程序员一定... 博文 来自： 平头哥的...

“金三银四”，敢不敢“试”？

阅读数 6849

临近3月份，到了“金三银四”换工作的高峰期，往年可能会3、4月份，今年特殊，多... 博文 来自： 铭毅天下...

Java C语言 Python C++ C# Visual Basic .NET JavaScript PHP SQL Go语言 R语言
Assembly language Swift Ruby MATLAB PL/SQL Perl Visual Basic Objective-C
Delphi/Object Pascal Unity3D

没有更多推荐了，[返回首页](#)


1

赏


举报


https://blog.csdn.net/lisy673807439/article/details/86666684


10/11





1



















举报

