基于二进制动态翻译的ROP攻击检测方法研究与实现

1. 绪论
   1. 研究背景

如今无论计算机技术发展到何种程度，计算机软件安全永远是人们最为关心的话题，相关的研究总在不断地进展和延续。随着操作系统的更新换代，软件自身的安全性不断提升，针对各种攻击类型，大量防御策略被提出并应用，对软件进行攻击变得越发困难。但是由于操作系统代码量日益增大、复杂度逐步提高，攻击者总能找出系统漏洞，并利用漏洞进行攻击，如图1-1所示，CVE[1]漏洞数量呈现逐年提升的趋势。此外，程序员编程的不规范以及软件安全更新的不及时更是导致软件漏洞被广泛利用。软件漏洞的必然存在，就像一颗定时炸弹，带来了极大的安全隐患。例如勒索病毒WannaCry利用美国国家安全局泄露的危险漏洞“EternalBlue”（永恒之蓝）进行传播，从2018年初到9月中旬，总计对超过200万台终端发起过攻击，攻击次数高达1700万余次，该病毒通过互联网在全球爆发，国内大量高校及企事业单位被攻击。

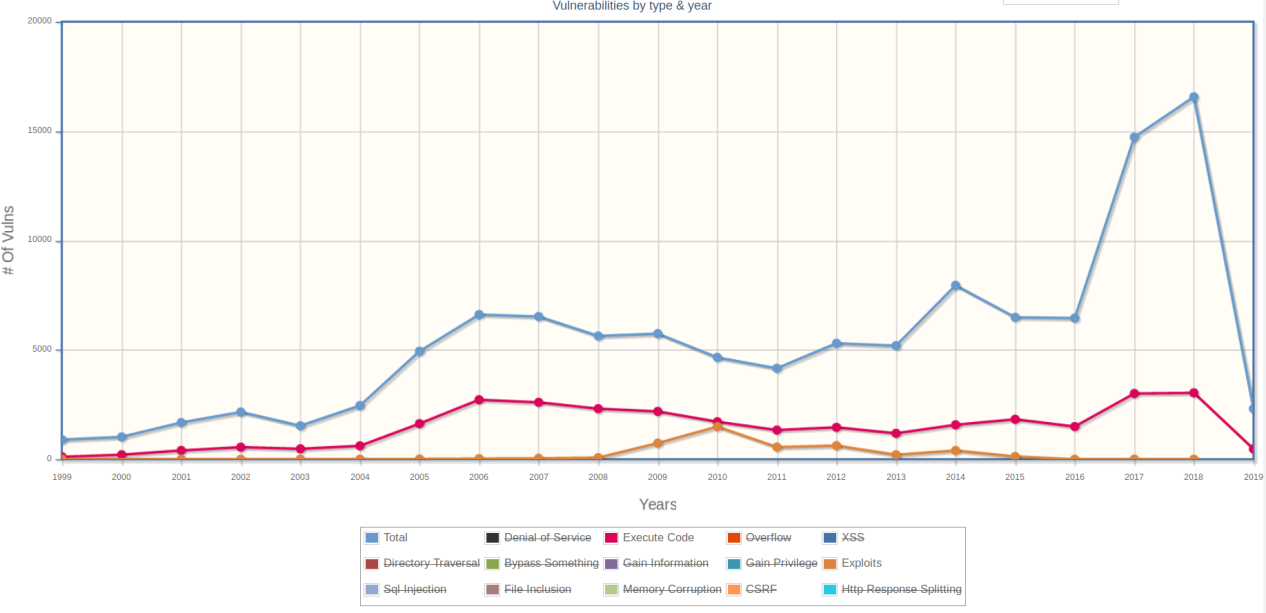


图1-1 近20年CVE漏洞数量

在众多的安全漏洞中，如图1-2，二进制安全占据了半壁江山，其中缓冲区溢出(buffer overflow)是一种常见的漏洞。由于c语言对程序缓冲区边界不进行检测，当攻击者向缓冲区写入过多数据后，缓冲区将溢出。若缓冲区在栈中发生溢出，栈中的函数返回地址将被覆盖，当程序返回时，程序控制流将被攻击者劫持。此外整型溢出、浮点型溢出、格式化字符串、UAF等常见漏洞，均可使攻击者劫持程序控制流。劫持程序控制流，然后执行攻击者构建的攻击代码，是进行攻击的基本流程。早先攻击者将恶意代码注入内存空间，并将控制流劫持至恶意代码，从而达到攻击目的。这些被注入的代码称做shellcode，他们通常是可执行的代码，通过系统调用实现打开shell、更改系统权限、执行程序等恶意行为。

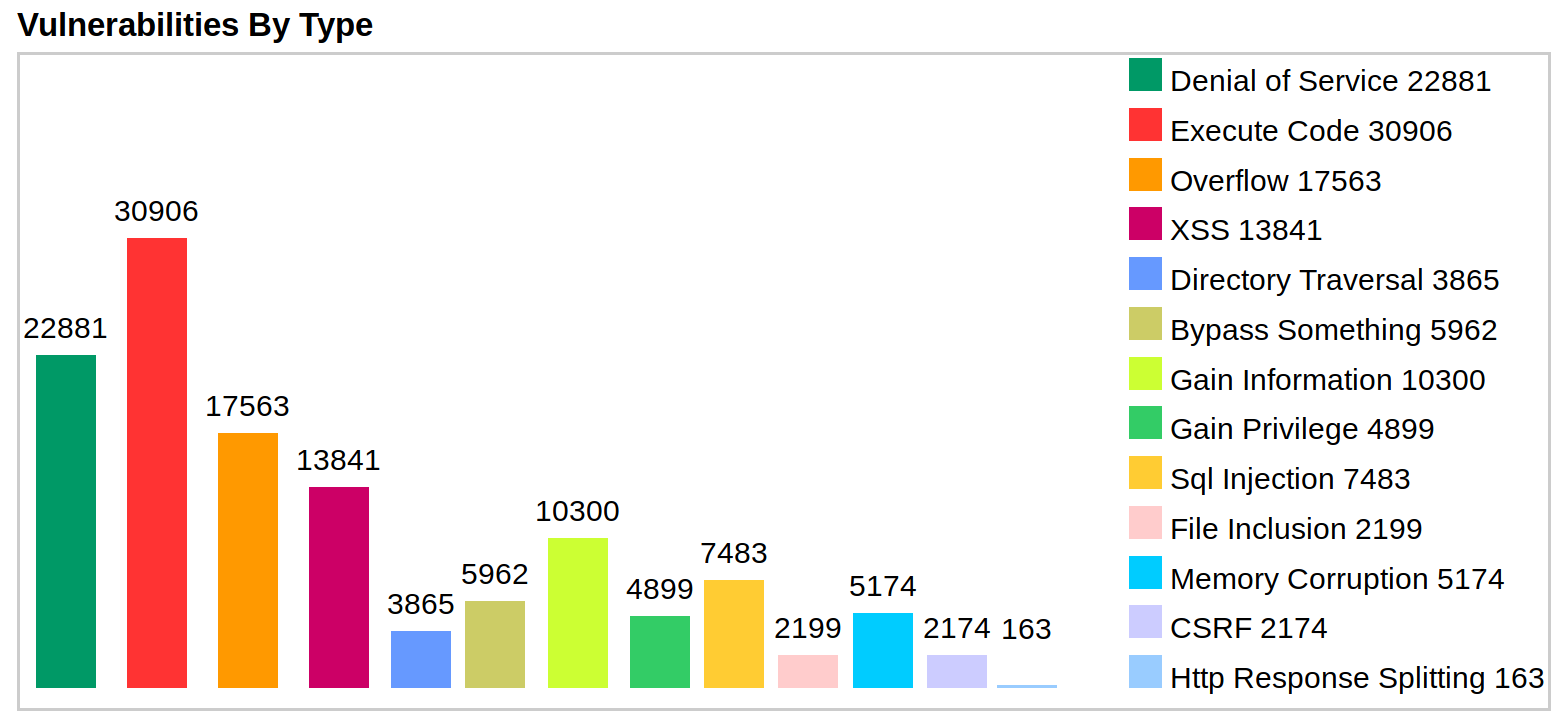


图1-2 近20年各类CVE漏洞数量统计

但是在数据执行保护（DEP）[2]广泛采用后，内存中的所有可写页面均不具有可执行权限。因此，即使攻击者将程序控制流劫持至他们注入的恶意代码，这些代码也无法执行。为了绕过DEP机制实现攻击，攻击者不再注入代码，而是通过利用漏洞进程中的现有的可执行指令来构造恶意行为，即代码复用攻击（CRA）。Return-into-libc[3]是代码复用攻击的一种简单应用，攻击者利用缓冲区溢出漏洞，将栈中的返回地址覆写为某些敏感的libc函数地址（如system、execve），再通过修改栈或寄存器中的数据，构造函数参数，从而实现攻击行为。

* 1. ROP攻击及防御现状

返回导向编程[4]（ROP）是一种常用的代码复用攻击技术，不同于Return-into-libc，攻击者在返回导向编程时，不执行整个函数，而是执行源自各函数片段中的指令序列，这些指令序列称作gadget，具有如下两个基本特点：1.具有一定的功能，如：寄存器相加、加载某值到内存；2.以ret指令为结尾。攻击者首先搜索可用gadget，然后将各个gadget链接在一起，从而实现一次完整的攻击（详见2.2节）。ROP最初由Shacham[4]提出并应用于x86平台，随后被拓展到其他体系结构[5,6,7,8]。ROP已被证明可实现图灵完备计算[9]。此外，一些允许攻击者使用ROP自动构造任意恶意程序的工具已被开发出[10,11,12,13]。

一些防御机制通过检测连续gadget末尾的ret指令特征识别ROP攻击，为了绕过该检测机制，将ret指令替换为call指令的Call Oriented Programming(COP)[14]和将ret指令替换为jmp指令的Jump Oriented Programming(JOP)[15]被相继提出。Snow[]还提出了实时ROP，攻击者在程序运行时完成gadget的搜索与链接。Bittau提出了BROP[]，他指出即使不清楚任何目标服务器的信息，也能够根据服务器返回的内容，搜索gadget并构造攻击。

针对现有的各种ROP攻击，研究者提出了几类防御方案。第一类方案是基于内存地址随机化，ASLR是被广泛应用的一种，ASLR在程序代码段、共享库加载到内存的过程中，为其基址随机增加一个偏移量，从而使攻击者无法准确获取gadget。ASLR的变种防御相继被提出，随机化粒度也在不断优化。第二类方案是基于程序二进制动态检测技术[]，通过监测程序运行行为，判断其是否符合ROP攻击特征，Lu[]等人通过二进制动态插桩检测框架PIN并基于统计学方法，识别ROP攻击。第三类方案是检测程序控制流的完整性[]，通过判断程序执行流程是否符合控制流图CFG，或是通过影子栈[]检测程序返回地址是否合法，来判断程序是否被攻击。

根据上述的ROP防御思想，本文将使用二进制动态插桩检测框架PIN，提出一种综合方案，应用于ROP攻击的动态防御与检测。

* 1. 本文主要研究内容
     1. ROP攻击的动态特征提取

本文分析多个ROP攻击实例在攻击过程中的泄漏内存地址、修改程序返回地址、连续调用gadget、篡改GOT表、调用敏感libc函数等攻击行为。当程序遭受攻击时，其运行时环境出现异常，本文通过PIN框架提供的各种实用API，构造攻击特征提取模块，提取并分析了被攻击程序的运行时动态特征。

1.3.2 ROP攻击防御方案与实现

本文分析现有的各种防御机制的优劣，并基于现有的各种防御机制，提出了影子栈、调用返回平衡、GOT表保护三种策略，这些策略根据攻击特征提取模块提供的程序运行时异常信息，对程序遭受的ROP攻击进行综合的实时防御。

* 1. 本文组织结构

第一章 引言。介绍本文研究背景以及ROP攻击与防御的发展和现状，最后阐述了本文的研究内容。

第二章 ROP攻击与防御分析。详细介绍ROP攻击原理、攻击流程，介绍并分析现有的ROP防御机制的优势与不足。

第三章 常见ROP攻击方法。通过实例介绍常见的ROP攻击方法。

第四章 基于PIN的ROP攻击检测方法。介绍检测工具的总体框架，分析其各部分功能。利用实验验证检测方案的有效性。

第五章 总结和展望。总结本文的工作，分析其中的不足，展望基于二进制插桩检测方法值得改进的地方。

1. ROP攻击与防御分析

2.1 ROP攻击

2.2.1 原理

在现代操作系统中，栈被用作函数调用返回的场所。当函数被调用时，操作系统将在栈中分配一块新的内存空间，称作栈帧，栈帧中存储上一个栈帧的栈基址、函数返回地址、局部变量、函数参数等信息。当函数调用发生时，程序控制流会发生转移，即从原函数转移至被调函数。函数调用返回的流程如下：call指令执行后，操作系统将返回地址（call指令的下一条指令地址）压入栈中，然后跳转到call指令的目标地址，执行被调函数，当函数执行结束后，其末尾的ret指令，将栈中的返回地址赋值给程序计数指针ip，程序控制流于是回到原函数。由于函数调用返回的信息存储在栈中，函数调用的过程也伴随着栈帧的切换。栈帧的切换流程如下：call指令执行后，程序控制流转移至被调函数，被调函数首先将旧的栈基址压入栈中1，然后设置新的栈基址2，并移动栈指针，开辟新的栈空间3，ret指令执行前，将栈指针指向栈基址4，并恢复保存的栈基址5。其中1-3为栈帧建立过程，4-5为栈帧的销毁过程。

push ebp

mov ebp, esp

sub esp, 0x8

...

mov esp, ebp

pop ebp

ret

2.2.2 攻击流程

一般情况下，攻击者首先在程序或者程序所链接的共享库的代码段中搜索可用的gadget，随后利用程序存在的漏洞（如：栈溢出），将搜集的gadget地址写入内存空间，并将程序的返回地址覆盖为gadget的地址。如图1x，攻击者将程序的原返回地址覆盖为返回地址1，并将一些数据以及返回地址2，3写入栈中，返回地址1，2，3分别指向三个不同的gadget。当程序返回时，程序控制流首先被劫持至第一个gadget中，其结尾的ret指令对应返回地址2，当第一个gadget完成一定操作后，程序控制流将转移至下一个gadget。由此，攻击者可以将搜集到的gadget链接为gadget链，进而实现一次完整的攻击。



2.2.3 变种攻击

COP

JOP

混合式ROP

如：检测以ret指令为结尾的的连续短代码序列，jmp将破坏之，



Xor eax,eax

Pop ebx

Inc eax

Pop ecx

stc

jmp ecx

Int 0x80 (再说 expliotdb随便找一个)

2.2 ROP防御

2.2.1 基于随机化

随机化是针对代码复用攻击最直接的防御方式之一，随机化使代码在每次运行时被加载到随机的地址空间，因此攻击者无法获取被复用代码的准确地址。Pax最早提出了随机化ASLR[]，程序开启ASLR后，各各镜像(image)被加载到随机的地址空间。但是ASLR存在明显漏洞，攻击者可以通过暴力搜索[]或者内存泄漏[]等方法，获取被随机化镜像的基地址，从而绕过ASLR防御机制。考虑到ASLR存在的种种问题，细粒度的随机化相继被提出，这些方案将随机化粒度缩小到函数级别[],基本代码块级别[]，甚至是指令级别[],但是这些随机化方案或多或少都存在限制，需要提供额外信息，并且会带来额外的系统开销，因此没有得到广泛的应用。

2.2.2 基于运行时状态监控

2.2.3 基于控制流

2.2.4 ...

2.3 本章小结

1. 常见ROP攻击方法
2. 基于PIN的ROP攻击检测方法

4.1 假设

4.2 总体设计

基本上，当程序执行必须转移到子程序时，call指令本身将返回地址压入堆栈。然后，被调用的子例程完成其任务并通常返回其原始调用者。但是，有一些例外打破了传统的调用约定，函数返回到其他位置。

我们将在3.4节讨论这些例外。目前，我们假设函数总是返回到最初由堆栈顶部的调用指令（TOS）推送的返回地址。然而，我们的xxx原型实现也处理异常（参见第4节）。

ROP攻击滥用堆栈指针的任务。在普通程序中，在返回函数时，堆栈指针将指向返回地址，以便控制转移回调用函数。如第2节所述，攻击者错误地使用每个指令序列末尾的返回指令，以便将控制转移到后续指令序列。返回指令可以是最初由libc的程序放置的预期指令，或者是非预期的指令，其中返回指令的字节值只是另一个有效指令的后缀。

出于这个原因，我们特别关注如图2所示的调用和返回指令。为了评估在程序执行期间发出的每个返回指令，我们将返回地址的副本存储到单独的影子堆栈中（类似于[12,43,21]）一旦程序发出调用指令。我们检测程序执行期间发出的所有返回指令，并执行返回地址检查，如下所述：

1.在处理器执行指令之前，我们的解决方案拦截指令并评估指令的类型和目标。在实践中，这可以通过二进制检测框架来完成，我们将在3.3节和3.4节中解释。

2.首先，我们检查指令是否是一个调用指令。如果是这种情况，我们将推送的返回地址的副本存储到我们的影子堆栈中（图2中的过渡2a）。否则，我们检查截获的指令是否是返回指令（图2中的转换2b）。

3.如果指令是返回指令，我们的解决方案将检查影子堆栈上的顶部返回地址是否等于程序堆栈顶部的返回地址。如果两个返回地址不同，我们得出结论，发生了返回地址损坏，程序执行被重定向到程序员不想要的指令。

程序堆栈将保存第二个指令序列的起始地址。但是，由于我们的影子堆栈只保存由调用指令本身推送的返回地址，因此它不能包含从函数中间某处开始的该指令序列的起始地址。因此，我们的解决方案可以检测到任何返回地址违规。

显然，在我们的方法中，我们假设在ROP攻击期间发出的指令序列以返回指令结束。实际上，这是Shacham [38]提出的原始ROP攻击。绕过我们的解决方案的一个想法是使用以间接跳转/分支指令结束的指令序列。实际上，Shacham简要提到了使用间接分支指令的能力，最近发表了一篇论文[10]，描述了针对Intel x86的攻击。这些攻击超出了本文的范围。但是，我们将在第7节中讨论如何检测此类攻击。

在3.4节中，我们将描述如何将我们的解决方案实现到像Pin这样的二进制检测框架中。

4.3 技术及工具

4.4 框架概述

框架

4.5 实验与评估

4.6 本章小结

1. 总结和展望

5.1 总结

5.2 展望

1. Vulnerabilities By Type,https://www.cvedetails.com/vulnerabilities-by-types.php
2. Data execution prevention. <http://support.microsoft.com/kb/875352/EN-US>
3. Wojtczuk, R.: The advanced return-into-lib(c) exploits: PaX case study. Phrack Mag. 0x0b(0x3a), Phile# 0x04 of 0x0e (2001)
4. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: Proceedings of the 14th ACM Conference on Computer and Communications Security, pp. 552-561. ACM (2007)
5. Kornau, T.: Return oriented programming for the ARM architecture. Ph.D. thesis, Masters thesis, Ruhr-Universität Bochum (2010)
6. Buchanan, E., Roemer, R., Shacham, H., Savage, S.: When good instructions go bad: generalizing return-oriented programming to risc. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 27-38. ACM (2008)
7. Checkoway, S., Feldman, A.J., Kantor, B., Halderman, J.A., Felten, E.W., Shacham, H.: Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage. In: EVT/WOTE 2009 (2009)
8. Francillon, A., Castelluccia, C.: Code injection attacks on Harvard-architecture devices. In: Proceedings of the 15th ACM Conference on Computer and Communications Security, pp. 15-26. ACM (2008)
9. Tran, M., Etheridge, M., Bletsch, T., Jiang, X., Freeh, V., Ning, P.: On the expressiveness of return-into-libc attacks. In: Sommer, R., Balzarotti, D., Maier, G. (eds.) RAID 2011. LNCS, vol. 6961, pp. 121-141. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23644-0 7
10. Dullien, T., Kornau, T., Weinmann, R.P.: A framework for automated architecture-independent gadget search. In: WOOT (2010)
11. Hund, R., Holz, T., Freiling, F.C.: Return-oriented rootkits: bypassing kernel code integrity protection mechanisms. In: USENIX Security Symposium, pp. 383-398 (2009)
12. Roemer, R.G.: Finding the bad in good code: automated return-oriented programming exploit discovery (2009)
13. Schwartz, E.J., Avgerinos, T., Brumley, D.: Q: Exploit hardening made easy. In: USENIX Security Symposium, pp. 25-41 (2011)
14. Carlini, N., Wagner, D.: ROP is still dangerous: breaking modern defenses. In: 23rd USENIX Security Symposium (USENIX Security 2014), pp. 385-399 (2014)
15. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, pp. 30-40. ACM (2011)