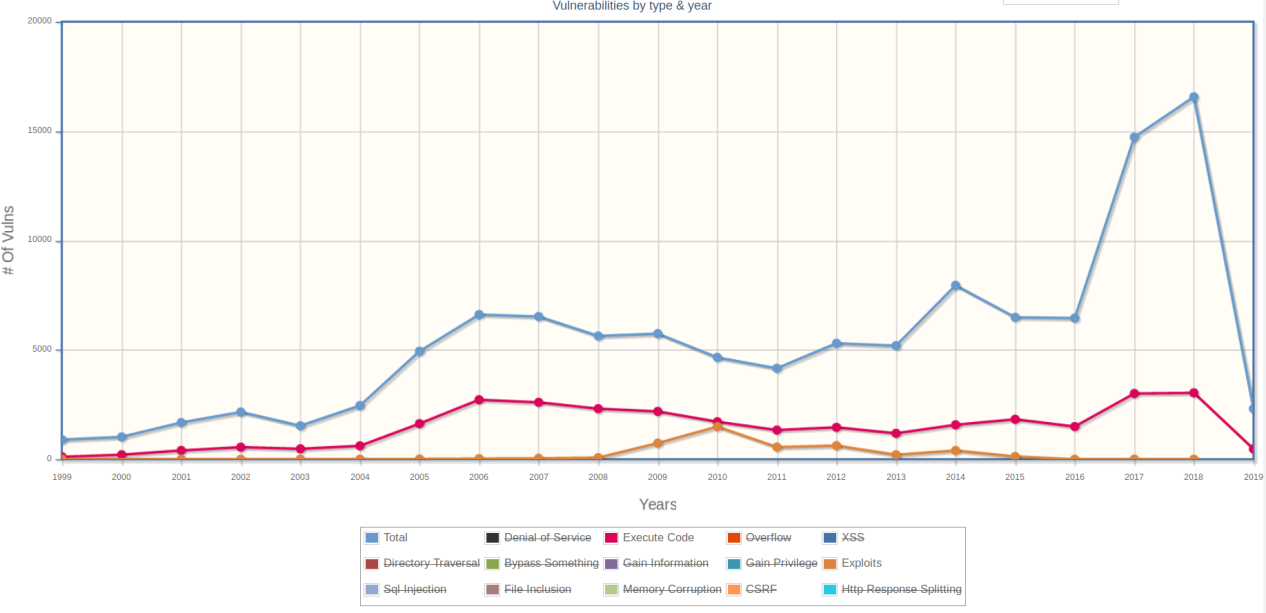
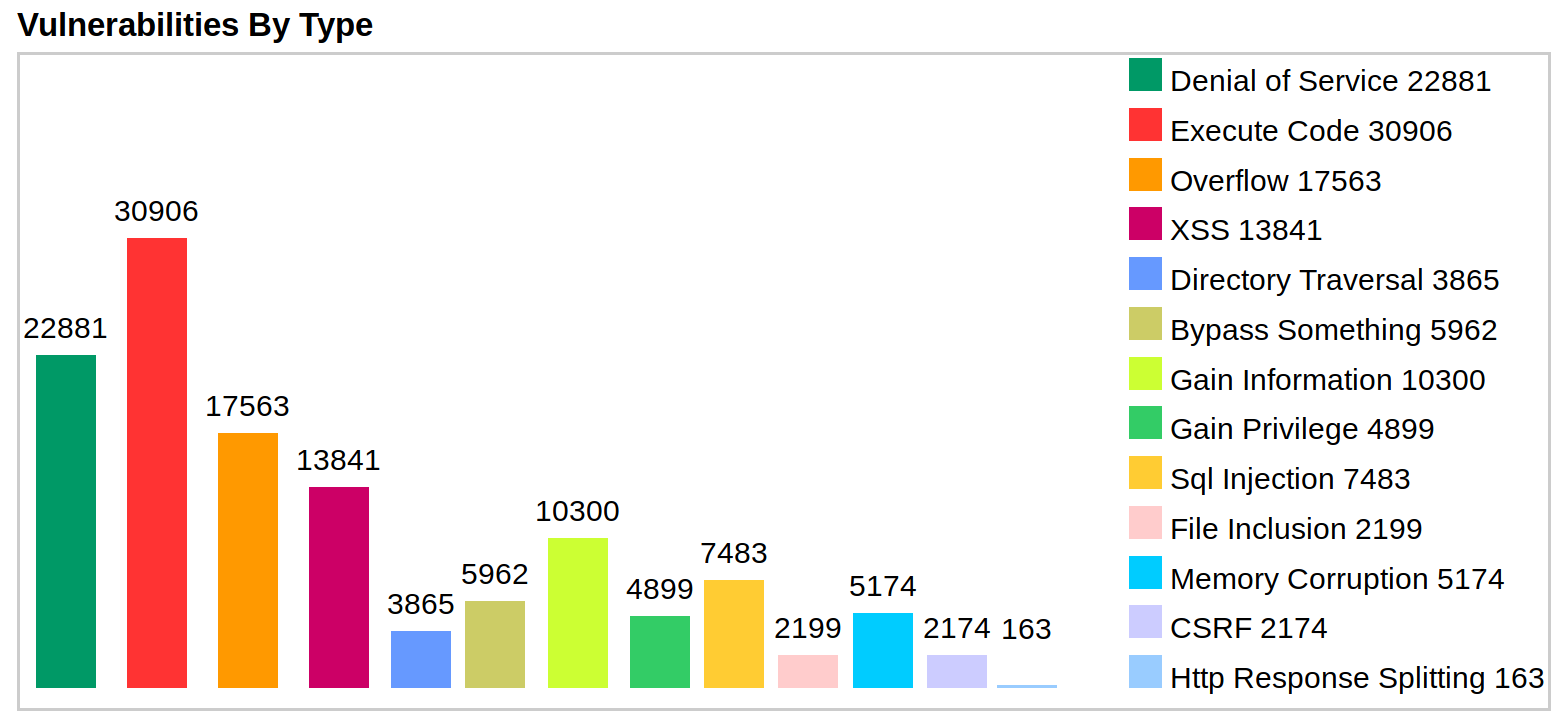
1. 引言
   1. 研究背景

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。



在众多的安全漏洞中，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，随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。



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（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。

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。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]。



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)

为了绕过现有的保护机制，攻击者更喜欢使用组合型gadget。图2展示了一个仅由4个gadget构成的非常简单的混合式ROP攻击，它是由传统shellcode[3]派生出的，在x86架构下，用于关闭正在运行的进程。为方便起见，我们使用系统调用*exit(n)*（n表示非零整数）代替*exit(0)*。其中，寄存器*eax*中存储系统调用号，*ebx*中存储参数。由于DROP[17]和DynIMA [20]只检测基于*ret*的连续的gadget，攻击者可以利用上述简单的ROP恶意代码来绕过这两种防御机制。

* 1. 本文主要工作和贡献
  2. 本文组织结构

1. ROP攻击与防御

2.1 ROP攻击

原理

攻击流程

2.2 ROP防御

2.3 本章小结

1. 基于PIN的ROP攻击检测方法

3.1 假设

3.2 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。

3.3 技术及工具

3.4 框架概述

框架

3.5 本章小结

1. 实现

4.1 pin

4.2 实现框架

4.3 实验与评估

1. 总结展望

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)