**ROP-Hunt：在应用程序中检测返回导向编程(ROP)攻击**

摘要:

返回导向编程(ROP) 是一种新型漏洞利用技术，该技术通过重复使用已有的小代码序列(gadget)构造出一条gadget链，从而执行任意的非法操作。尽管很多防御机制已经被提出，但是一些新的ROP攻击的变种能够轻易绕过这些机制防御。

在本文中，我们将介绍一个新工具ROP-Hunt，它能依据正常程序与恶意ROP代码间的差异来防御ROP攻击。ROP-Hunt利用**插桩检测**(instrumentation)技术并在程序运行时检测ROP攻击。在我们的实验中，ROP-Hunt可以在众多实例程序中检测出所有类型的ROP攻击。我们使用了几个原版的SPEC2006标准来测试ROP-Hunt的性能，结果表明它具有零误报率和可接受的系统开销。

关键词：返回导向编程、缓冲区溢出、检测、代码复用攻击、二进制插桩检测

一、引言

由于数据执行保护（DEP）[1]的广泛采用，确保了内存中的所有可写页面都是不可执行的，因此攻击者很难将被劫持的控制流重定向到他们自行注入的恶意代码。为了绕过DEP机制，代码复用攻击（CRA）被提出并成为了攻击者们的利器。攻击者们不再注入代码，而是通过复用被攻击的漏洞进程中的现有指令来构造恶意行为。Return-into-libc技术[37]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，将位于栈中的返回地址覆写为攻击者挑选出的将要被执行的库函数地址。传统的return-into-libc攻击利用libc函数，并不支持在受害计算机上的执行任意操作。

返回导向编程（ROP）是另一种代码复用攻击技术，它执行称为gadgets的短指令序列，而不是执行一整个函数。它最初由Shacham[35]提出并应用于x86平台，随后被拓展到其他体系结构[13,16,23,26]。ROP已被证明可实现图灵完备计算[36]。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[22,24,33,34]。

在过去几年中，许多用于减缓基于ROP攻击的软硬件防御方法已被提出。例如：如果连续执行以ret指令为结尾的小指令序列，DROP [17]和DynIMA [20]将触发警报。ROPdefender [21]则是维护一个影子栈，并验证所有返回地址。李等人[27]提出了一个用于x86平台的编译器，它能避免生成可用作恶意返回指令的“0xc3”字节。此外，它使用间接调用机制替换了预期的调用和返回指令。但是，这些机制只关注了以返回指令为结尾的gadget，并不能防御其他类型，即不以返回指令为结尾的gadget的类ROP攻击。CFLocking [11]和G-Free [30]旨在防御所有类型的ROP攻击，但它们需要源代码，对于实际的终端用户而言，这些源代码通常难以取得的。KBouncer [32]涵盖了所有ROP攻击类型，且不需要辅助信息，并实现了不错的运行时效率。然而，它只是监视目标关键路径上应用程序的执行流，例如系统API，如此，它不可避免地漏掉了那些不使用这些路径的ROP攻击。

ROP攻击组合gadget成链，以执行复杂的操作，它具有如下特征：gadget长度很短；连续的gadget并不在同一个程序中，并且它们都在某个地方执行系统调用。基于这些特征，我们设计并实现了一个名为ROP-Hunt的工具，该工具通过检查程序执行的行为是否与这些特征匹配，从而对所有类型的ROP攻击进行动态检测。在ROP-Hunt中，基于危害程度，我们将ROP报告分为两类：*警告*和*攻击*。

总的来说，我们工作的主要贡献是：

- 统计分析了大量正常的应用程序和最新的ROP恶意代码，并提取了ROP攻击的特征。

- 提出了一种新方法，可以在不访问源代码的情况下保护传统应用程序免受所有类型的ROP攻击。

- 在x86框架的Linux平台上，设计并实现一台样机，即ROP-Hunt，并评估了其安全有效性和性能开销。

本文的其余部分安排如下：在第二、三部分中，我们描述了ROP攻击并分析了它们的特征。ROP-Hunt的设计和实现在第四部分介绍。第五部分和第六部分，分别讨论了参数选取和特殊的延迟gadget。第七部分介绍了ROP-Hunt的安全性和性能评估。第八部分研究了它的局限性。最后，我们在第九部分总结全文并讨论有待完成的工作。

二、ROP攻击

在不向程序地址空间注入新代码的情况下，ROP攻击由称为gadget的短指令序列组成。每个gadget执行一些小的计算操作，例如将两个寄存器相加或将某个值加载到内存，并以返回指令为结尾。我们可以将这些gadget链接在一起，并通过在堆栈上写入适当的值，使控制流从一个gadget转至另一个gadget。



图1 一般的ROP攻击

图1说明了一般的ROP攻击流程。第一步，攻击者利用漏洞程序的如缓冲区溢出这类的与内存相关的漏洞，将堆栈指针（ESP）移至第一个返回地址所在位置。例如，Aleph在[31]这篇文章中，通过栈溢出覆写了函数返回地址。由于返回地址1覆盖了原返回地址所在位置，ESP的值将自动改为此点。第二步，通过pop栈中的返回地址1，程序执行流被重定向至第一个gadget。该gadget以另一个返回指令为终止，该指令pop栈中的返回地址2（第三歩）并将程序执行流重定向至下一个gadget（第四歩）。每个gadget通过这种方式逐个执行，直到攻击者达到目标。

最近，一些不使用ret指令的ROP攻击新变种被提出。Checkoway等人[15]发现可以通过搜寻尾随有间接跳转的pop指令（例如*pop edx; jmp [edx]*）来进行返回导向编程。这种指令序列的行为类似于返回指令，亦可用作gadget的链接。

跳转导向编程（Jump-Oriented Programming, JOP）[12]是ROP攻击的另一种变体，它使用寄存器间接跳转代替了返回指令。JOP使用调度程序表来保存gadget的地址。每个gadget对应一个调度程序，调度程序是一段可以控制程序控制流的指令序列。调度程序用作虚拟程序计数器(PC)，将程序控制流转换为调度表中存储的地址条目，这些地址是特殊的、具有跳转导向功能的gadget的地址。在这些gadget的结尾，攻击者通过间接跳使程序控制流回归调度程序。随后，调度程序将指针指向下一个gadget。一个简单的调度程序如下：add edx,4; jmp [edx]。

调用导向编程（Call Oriented Programming, COP）[14]由Nicholas Carlini和David Wagner于2014年推出。攻击者用以间接调用指令为结尾的gadget代替以返回指令为结尾的gadget。COP攻击不需要调度程序，它通过依次地将内存间接位置指向下一个gadget的方法，来将gadget链接在一起。



(a) 传统shellcode (b)ROP恶意代码

图2 简单混合式ROP攻击

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]只检测连续的基于ret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。

三、ROP攻击特征

ROP攻击检测的关键是找出ROP恶意代码和普通程序代码间的差异。ROP中的一个重要因素是gadget的长度。 在[20]这篇文章中，研究者们发现ROP攻击中使用的指令序列长度为2到5个指令。DROP [17]也指出gadget中的指令数不超过5条。Kayaalp等人[25]从libc标准库中提取了所有gadget，并对平均gadget长度进行了研究。结果表明，随着gadget长度的增加，副作用的数量呈线性增长，使得它们越来越难以被利用。

在目前的检测机制中还需要考虑了一些其他因素。如果三个小指令序列一个接一个地被执行，DynIMA [20]将报告一次ROP攻击。Fan Yao等人[38]发现很少有两地址相离较近的gadget存在。

基于编写ROP恶意代码的经验，我们发现了另外两个特性。其一，无论是以跳转还是调用指令为结尾，连续的gadget都不会位于同一个例程中。其二，shellcode总是利用系统调用将控制流从用户态转移至内核态。

在计算机编程中，**例程**是一串代码序列，被用于在程序执行期间重复调用和使用。在高级语言中，许多常用的例程被打包为函数。在传统的ROP攻击中，每个gadget都以返回指令为结尾，除了递归返回，在大多数情况下，它们不在同一个例程中。ROPGadget [8]是一个开源的gadget搜索工具，我们用它从glibc中提取gadget，并使用文章[12]中提出的算法构造几段JOP恶意代码。我们发现相同例程中的连续的gadget极难被利用。

ROP恶意代码由shellcode派生而出，基于传统的shellcode来构建gadget。我们分析了文章[5]中全部247个shellcode，发现其中212个至少调用了一次系统调用。其他的shellcode为了绕过特征检测机制，使用了加密或自修改payload的方式，于是“int 0x80”不被直接使用，避免敏感数据（如cd 80）的出现。但无论如何，为了获得更高的权限，他们终将在runtime调用系统调用。 文章[2]中调用了内核vsyscall函数，该函数使用sysenter指令，将控制流从在第3特权级运行的用户态转移至操作系统。但是，sysenter指令提供了对内核的快速访问方式，也可以被视为另一种系统调用。

我们认为（1）gadget的大小；（2）系统调用的执行；（3）连续的候选gadget不在同一程序中，这三点可以作为ROP攻击最具代表性的特征。我们基于ROP恶意代码和普通程序之间存在的这三种差异，开发了一个名为ROP-Hunt的工具，它通过检查程序运行轨迹是否偏离正常运行路径来动态检测ROP攻击。我们将在下一部分中展示ROP-Hunt的设计。

四、ROP-Hunt设计与实现

基于ROP攻击的特征，我们提出了能够有效地检测ROP攻击的方法。由于我们设想的是无法访问源代码的情况，因此我们使用了允许向程序添加额外代码以观察、调试其行为[29]的插桩检测技术。

4.1假设及定义

在本文中，我们将gadget中的指令数定义为G\_size。候选gadget是指G\_size大于阈值T0的gadget。连续的候选gadget序列的长度定义为S\_length，Max(S\_length)代表S\_length的最大值。

为了模拟真实环境，我们做出如下假设：

1.我们假设底层系统支持DEP[1]模型，该模型禁止了可执行内存的写权限。在这种情况下，基于代码注入的攻击是不可行的。现代处理器和操作系统默认启用DEP。

2.我们假设攻击者能够通过缓冲区溢出[19,31,39]、字符串格式化攻击或非本地跳转缓冲区（使用*setjmp*和*longjmp* [4]）来发起ROP攻击。

3.我们假设攻击者在用户态下进行操作，并且利用漏洞发起的攻击不会导致权限提升。

4.我们假设我们无法访问源代码。

4.2系统概述

图3展示了ROP-Hunt的流程图。根据我们在第三部分中分析的ROP的特征，ROP-Hunt动态地监视程序，并截取系统调用指令和三个控制流敏感指令：call，jmp和ret。ROP报告分为两类：*警告*和*攻击*。*警告*表示该进程存在严重的被ROP攻击风险，但是由于它没有调用系统调用来访问底层系统源，我们认为攻击者无法利用其进行任何有意义的攻击。如果**统计值越过阈值**且**有调用系统被调用**，ROP-Hunt将终止进程并反馈攻击报告。

* *报告警告*：当ROP-Hunt识别到这三种指令（调用、间接跳转和返回指令）时，它会检查指令序列的长度是否大于阈值T0。若非，则提取目标地址和当前指令地址，尤应重视ret指令，因为ret指令会将目标地址pop出栈。随后，ROP-Hunt会检查这两个地址是否位于同一例程中。若非，我们便将该指令序列记录为候选gadget。接下来，我们统计连续候选gadget的最大长度S\_length。如果S\_length小于或等于阈值T1，我们便将**潜在攻击**标志置为*True*并发出*警告*。
* *报告攻击*：系统调用是将控制流从用户空间转移至内核空间的唯一途径。故当识别出系统调用指令时，ROP-Hunt检查**潜在攻击**标志是否为True。若是，ROP-Hunt将报告*攻击*并终止该进程。

4.3实现细则

为了证明我们的方法的有效性并对其进行性能评估，我们开发了一个内核版本3.19的x86 32位版的Ubuntu 14.04样机。我们在样机ROP-Hunt中，使用了二进制插桩检测框架Pin [28]（版本2.14）。

我们将ROP-Hunt直接整合到Pin框架中。 Pin是程序插桩检测的工具，能检测所有实际执行的指令。Pin有两种工作模式：探针模式和即时（JIT）模式。在JIT模式下，Pin能够在处理器执行每条指令前将其拦截，包括那些程序员“意想不到”的指令。



图3 ROP-Hunt的工作流程

要想在runtime检测二进制文件，我们必须确定代码被插入在什么位置以及在插入点处执行了什么代码。 Pin提供了名为Pintools的插桩检测工具。Pintools以C/C++语言编写，在使用Pin提供的丰富API的同时，用户还可以自定义插桩检测代码。我们设计并实现了自己的Pintool来在Pin框架下检测ROP攻击。

runtime系统的总体架构如图4所示。我们的架构由Pin框架和Pintool，即ROP-Hunt组成。Pin是用于即时(JIT)调试及检测二进制程序的引擎。Pin框架包含一台虚拟机（VM）、代码缓存和供Pintools调用的API。其中，**虚拟机**包含JIT编译器、模拟器和调度程序。当程序开始运行时，各条指令先经JIT**编译**并检测，再交由**调度程序**激活并执行。经过编译的指令存储在**代码缓存**中，以便在多次调用代码段时降低性能开销。**模拟器**用于解释无法被直接执行的指令。

我们的Pintool，即ROP-Hunt，由记录单元和检测单元组成，检测单元包含**插桩检测例程**和**分析例程**。检测单元利用各插桩检测API与Pin进行通信，记录单元仅用于存储runtime中的一些统计值。

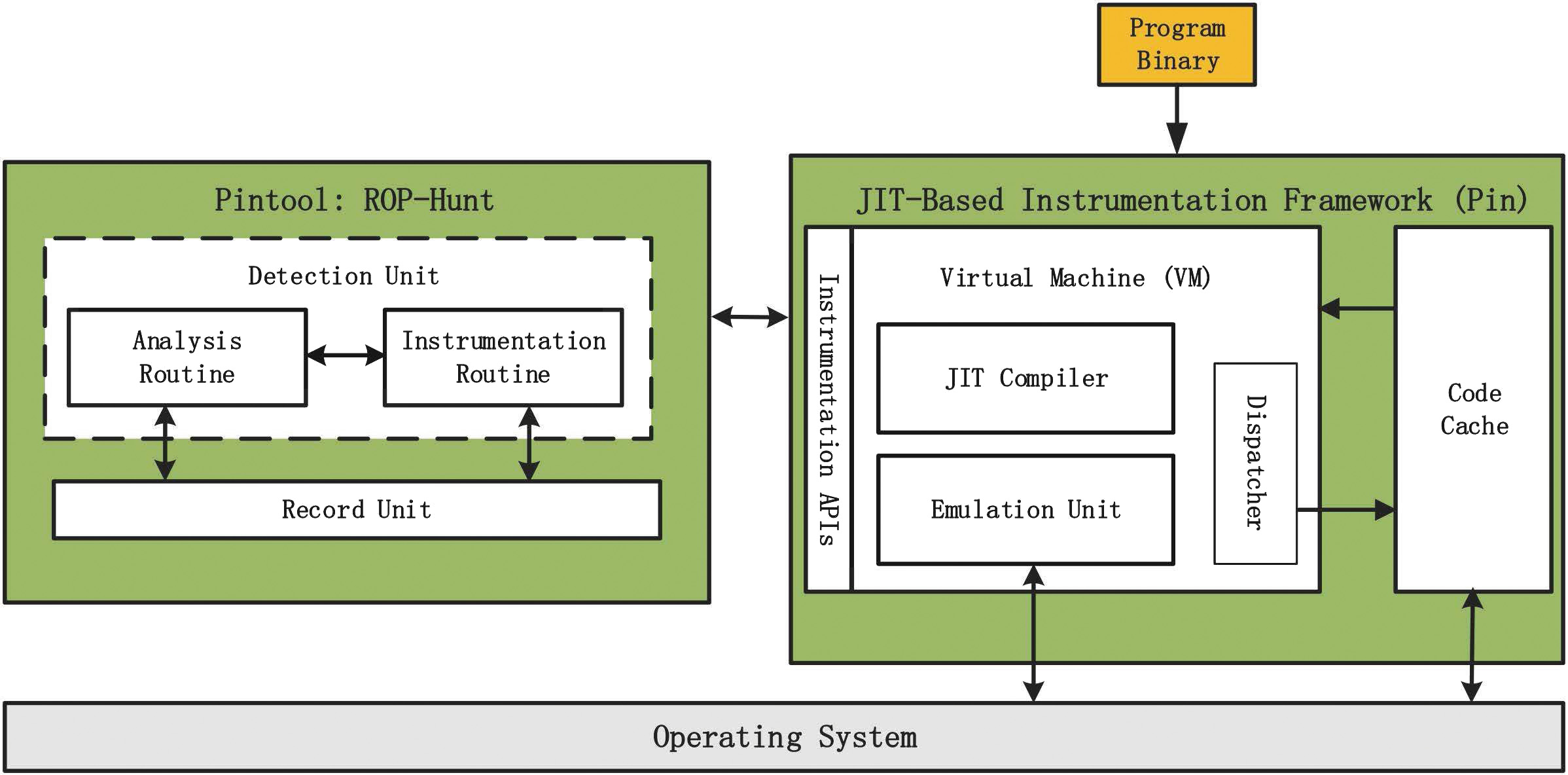


图4 在Pin框架下实现的ROP-Hunt

4.4插桩检测和分析例程

据4.1小节中所述内容，识别指令类型是关键点之一。ROP-Hunt的插桩检测例程通过Pin API提供的检查函数INS\_IsSyscall(INS ins)和INS\_IsSysenter(INS ins)来确定当前指令是否为*syscall*或*sysenter*，并通过INS\_IsIndirectBranchOrCall(INS ins)来确定当前指令是否为跳转指令。如果当前指令是间接跳转、调用或返回指令，我们将调用分析函数来提取当前指令地址以及目标地址。

ROP-Hunt为每个例程分配一个ID，该ID全局唯一，即同一个ID不会出现在两个内存镜像中。如果有同名例程存在于两个不同的内存镜像中（即它们在不同的地址中），那么每个例程将被分配不同的ID。如果内存镜像被卸载然后重载，那么其中的例程ID极有可能与以前ID不同。ROP-Hunt使用PIN\_InitSymbols()来函数初始化符号表并从二进制文件中读取符号。由此，我们可以通过地址来获取例程ID。

记录单元分别为每个线程分配数据空间。我们使用Pin API中的**线程本地存储**（TLS）来避免一个线程访问另一个线程记录的情况。