ELF如何摧毁圣诞 ——通过ELF动态装载机制进行漏洞利用

By Wei Liu | 一月 4, 2016 | SecComm

作者：Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel and Giovanni Vigna(UCSB,CA, USA; Politecnico di Milano, Milan, Italy)

原文出自：USENIX Security 2015

翻译：裴中煜，清华大学网络研究院（本文是清华大学本科毕设要求翻译的外文文献）全部译文

==========

摘要

近几十年来，计算机软件经历着一场利用技术、发现与保护技术之间的军备竞赛。一些有效的保护措施（例如，ASLR地址空间布局随机化），显著地增加了成功利用一个漏洞的难度。一个现代的漏洞利用一般分为两个阶段：第一步需要进行信息泄漏以取得程序的内存分布，接着第二步则进行实际的利用。然而，由于内存破坏后的具体情况各不相同，从程序中得到内存布局的方法并不总是可行。

在这篇文章中，我们展示了一种不需要进行信息泄漏，而是使用动态装载器来直接标识关键(critical)函数的位置并调用它们的技术。我们在ELF文件标准和动态装载器的实现中找到了几个弱点，这些弱点能够用来解析、执行任意库函数。因此，我们能够绕过特定的安全缓解措施，包括专门为保护ELF数据结构不被攻击者破坏而设计的partial RELRO与full RELRO。我们实现了一个名为Leakless的原型工具，并针对动态装载器的实现、之前的攻击技术和真实案例进行评估以确定我们的发现的影响。另外，Leakless也可以进行更可靠、更不具侵略性的攻击，以减少被入侵检测系统发现的几率。

==========

1．**简介**

从1998年Morris worm发表的第一个被广泛应用的栈溢出文章[27]以来，内存破坏漏洞的保护、利用和减缓技术研究占据着安全研究人员和网络罪犯相同的时间。尽管近年来内存破坏漏洞的盛行趋势有所减缓，经典的栈溢出仍然雄踞最常见软件漏洞的第3位，而另外4种内存破坏漏洞已经跌出了前25名。

而能够刹住内存破坏之风的原因，是在内存保护与缓解措施方面的巨大投入。这些缓解措施主要应用于2个方面：系统级加固(例如CGroups [23], Apparmor [4], Capsicum [41], 和 GRSecurity [18]) 和应用级加固(如 stack canaries [3], Address Space Layout Randomization (ASLR), 和 No-eXecute (NX) bit [8]).

尤其是地址空间布局随机化(ASLR)，通过将动态库加载到内存中随机的一块区域（对于攻击者来说未知），使得攻击者需要将利用过程拆分为2个阶段。在第一个阶段，攻击者必须使用一个信息泄漏漏洞将程序以及动态库的内存布局泄漏出来，这样就可以标识出安全关键(security-critical)函数（例如system()库函数）代码的地址。在第二个阶段，攻击者使用一个控制流重定向漏洞，将程序的控制流重定向到这个函数。

然而，由于内存破坏后的具体情况各不相同，从程序中得到内存布局信息的方法并不总是可行。例如，大多数解析代码（例如解码图像或者视频）经常不会与攻击者有直接的交互，这就排除了信息泄漏的可能性。没有这些信息，再对ASLR保护下的二进制文件使用现在的技术进行利用通常是不可行或不可靠的。

就像 [36] 中写的那样，除了加固应用和系统的竞赛，对于二进制的格式以及系统组件的一些鲜为人知的角落，则缺少仔细的检查。特别地，我们着眼于操作系统中的一个用户态组件——动态装载器，负责装载二进制文件以及它们依赖的库文件到内存中。二进制文件使用动态装载器来支持导入符号的解析功能。有趣的是，这恰好就是一个面对加固应用的攻击者通过泄漏库地址与内容尝试“重塑”一个符号的表现。

我们的技术的亮点，在于可以通过活用一个动态装载器的功能，完全省去对信息泄漏漏洞的需要。我们的技术利用动态装载器与ELF格式的弱点，解析并执行任意库函数，允许我们在没有信息泄漏的情况下成功地攻击加固后的应用。任何库函数都可以被执行，只要它所在库被加载进程序。既然所有的二进制程序都依赖于C语言库，这就表示我们的技术能执行system()和execve()这类安全关键(security-critical)函数，从而允许执行任意命令。我们还会展示一些通过重用特定应用程序库中的函数来进行复杂又隐秘的攻击。这项技术非常可靠且架构无关，攻击者不需要知道版本、布局、内容或者其他关于库函数不可知的信息。

我们实现了自己的想法，写成了一个称为Leakless的原型工具。要使用Leakless，攻击者必须拥有目标应用的副本，且能够利用漏洞（即劫持控制流）。之后，Leakless可以在没有信息泄漏的情况下自动地创建利用过程，并且调用攻击者感兴趣的关键库函数。

为了评估我们技术的影响，我们对几个不同的Linux(以及FreeBSD)发行版进行了调研，发现其中大部分的二进制程序容易被Leakless的攻击所影响（如果目标程序有内存破坏漏洞的话）。我们还审查了多种C语言库的动态装载器实现，发现其中大多数也是容易被Leakless的技术影响的。除此之外，我们展示了一种流行的缓解技术，RELocation Read-Only (RELRO) 重定位只读，它能够保护库函数的调用不受攻击者重定向的影响。然而它也被Leakless完全地绕过了。最后，我们比较了Leakless与类似的ROP编译器产生的ROP链的长度。Leakless产生的ROP链的长度显著地短于现有技术产生的ROP链。就像我们展示的那样，与传统

ROP编译器相比Leakless能够实现更加广泛的利用。

总的来说，我们作出了如下贡献：

– 我们开发了一个新的、架构与平台无关的攻击，使用基于ELF、支持动态装载的系统的固有功能，使得攻击者能够在不做信息泄漏的情况下，执行任意库函数。

– 我们详述了实现自己的系统的过程中，面对不同动态装载器实现和多种缓解措施(包括RELRO)的挑战，并最终克服了它们。

– 最后，我们进行了一次深入的评估，包括以前复杂的利用因使用了我们的技术变得容易的案例分析，对几种不同动态装载器实现安全性的评定，我们的技术在不同操作系统配置下适用性的调研，以及Leakless在ROP链长度改善方面的测量。

==========

2. **相关工作：内存破坏的军备竞赛**

内存破坏的军备竞赛(即防御者针对现有的利用技术开发对抗措施，接着攻击者想出新的利用技术来绕过这些措施的过程)已经持续了几十年。这场竞赛的历史已经被记录在别处 [37]，这一节着眼于那些使得现代利用技术被拆分成2个阶段的事件，就是说，需要攻击者在执行任意代码前进行信息泄漏这一步。

早期的栈溢出利用依赖于向缓冲区中注入二进制代码(称为shellcode)的能力，并需要覆盖在栈上的一个返回地址使其指向这个缓冲区。随后，当程序从当前函数返回时，执行流就会被重定向到攻击者的shellcode，接着攻击者就能取得程序的控制权。

结果，安全研究者提出了另一项减缓技术：不可执行位(the NX bit)。不可执行位具有防止那些不该有代码的区域(栈就是典型的这类区域)被执行的效果。

不可执行位逼迫攻击者们开始采用“代码重用”的理念：使用程序中已经存在的代码(例如系统调用或安全关键(security-critical)库函数)来达到他们的目的。在return-to-libc的利用中 [30,39] ，一个攻击者将控制流重定向到一个敏感的libc函数(例如system())，并给予其恰当的参数来实行恶意的行为，而不是注入shellcode。

为了对抗这项技术，一项系统级的加固措施，称为地址空间布局随机化ASLR被开发出来。一旦ASLR起作用，攻击者将无法知道库的位置。实际上，程序的内存布局(库被加载的位置，栈的位置以及堆的位置)每次执行都是随机的。因此，攻击者不知道将控制流重定向到哪里才能执行特定函数。更糟糕的是，即使是攻击者能够确定这些信息，他仍然不知道特定函数在库中的位置，除非他取得库文件的一份副本。结果，攻击者常常需要泄漏库本身的内容并解析代码来确定关键函数的位置。为了泄漏库，攻击者需要重用一些程序代码段里的小块代码(称为gadgets)来泄漏内存位置。这些gadgets可以通过将其地址写在栈上并连续地执行返回(ret)指令被组合使用。所以，这项技术被称作“面向返回编程”(Return Oriented Programming (ROP)).

ROP是攻击者的一个强有力的工具。实际上，在许多二进制程序中发现的ROP Gadgets是“图灵完全”集合，借助ROP编译器能够完成利用的任务。然而，由于其普适性的需要，ROP编译器倾向于生成依赖于具体漏洞细节的长ROP链，它们“太长而无用” [22].在这之后，我们将展示Leakless生成的相对短的ROP链，并且依赖于存在的缓解措施，只需要很少的gadgets。此外，Leakless在没有图灵完全的Gadgets的集合的情况下也能发挥作用。

在真实世界的利用中，攻击者往往使用一个信息泄漏攻击来泄露库的地址或内容，然后使用这些信息来计算安全关键(security-critical)函数(例如system())的地址，最后发送第二段攻击载荷(payload)到漏洞应用来重定向控制流到想要的函数。

实际上，我们观察到寻找特定库函数的目标已经被动态装载器实现了，这是一个能够进行符号解析(即确定库函数地址)的操作系统组件。所以，我们意识到可以使用动态装载器来略过信息泄漏这一步并巧妙地进行利用。因为我们的攻击不需要信息泄漏的步骤，所以我们称它为Leakless.

使用动态连接器的观念已经在一些return-to-libc的攻击中作为利用过程的一部分被简要地探究过 [15,21,30] 。然而，现有的技术非常依赖现有状况，依赖平台，需要2个阶段，或者易受到现有的缓解技术(例如RELRO)的影响，这一影响我们将在后面讨论。而Leakless，作为只有1个阶段，平台无关，具有普适性的技术，在这样的缓解技术面前仍然能够发挥作用。

在下一节中，我们将阐述动态装载器是如何工作的，然后将会展示如何活用它的功能来进行我们的攻击。

==========

3. **动态装载器**

动态装载器是一个用户执行环境的组件，它能够帮助在开始时加载应用需要的库并解析库导出的动态符号(函数和全局变量)供应用程序使用。在这一节中，我们将会阐述动态符号解析的过程在基于ELF的系统上是如何工作的 [33].

ELF是Unix类平台(包括GNU/Linux与FreeBSD)上比较普遍的一种标准格式，其定义独立于任何一种特定动态装载器实现。因为Leakless主要依赖与ELF标准的特性，它也很容易应用于很多系统。

3.1 ELF对象

一个应用由一个主要ELF二进制文件(可执行文件)和数个动态库构成，它们都是ELF格式。每个ELF对象由多个segments组成，每个segment则含有一个或多个sections(译注: 以下称sections为段).

每个段都有约定的含义。举个例子，.text段包含着程序的代码，.data段包含着它可写的数据(例如全局变量)，而.rodata段则包含着只读的数据(例如常量和字符串)。段的列表以数组的形式存于ELF文件的Elf_Shdr结构体中。

注意这里有两种ELF结构体的版本: 一种是32位的(例如Elf32_Rel)，一种是64位的(例如Elf64_Rel).为了简化起见，除了在相关讨论的特定案例中，一般情况我们将忽略这些细节。

3.2 动态符号与重定位

在这一节中，我们将对ELF符号解析过程相关的数据结构进行一个总结。图1总体展示了这些数据结构以及它们之间的关系。

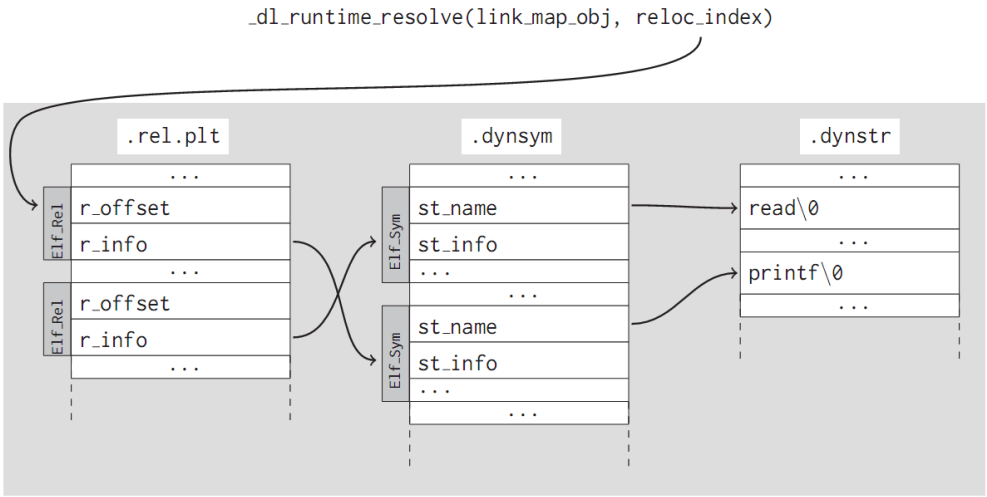

figure 1: 在符号解析过程中相关数据结构的关系(除去符号版本)。阴影背景部分表示只读内存。

一个ELF对象可以向其它ELF对象导出符号或导入符号。一个符号表示一个具有名称标识的函数或者全局变量。

每个符号都使用Elf_Sym结构体来描述。这个结构体的实例是ELF中.dynsym段的组成部分，它包含以下相关的域:

[st_name] 相对.dynstr段开始的偏移值，那里有这个符号名字的字符串。

[st_value] 如果这个符号被导出，则存有这个导出函数的虚拟地址，否则为NULL.

这些结构被用来解析导入的符号。导入符号的解析需要重定位的支持，重定位项以Elf_Rel结构体来描述。这个结构的实例存在于.rel.plt段(用于导入函数)和.rel.dyn段中(用于导入全局变量)。在这里我们感兴趣的是前者。Elf_Rel结构体有以下域:

[r_info] 此域的高位3个字节作为一个无符号的下标，表示这个符号在.dynsym段中的位置。

[r_offset] 解析后的符号地址被写入内存中的位置(绝对地址)。

当程序导入一个正常函数时，链接器会在.dynstr段中包含一个函数名称的字符串，在.dynsym段中包含一个指向它的符号(Elf_Sym)，在.rel.plt段中包含一个指向这个符号的重定位项(Elf_Rel).

重定位的目标(即Elf_Rel结构中的r_offset域)将会是全局偏移量表(Global Offset Table, GOT)中的一个条目。GOT表保存于.got.plt段，由能够解析.rel.plt段中的重定位的动态链接器来填写。

3.3 惰性符号解析

因为在程序开始时就解析所有导入符号并应用重定位是一项开销较大的操作，符号的解析将是惰性的。在惰性符号解析中，每个函数地址(相当于GOT中的条目)只在需要的时候才进行解析(即这个函数第一次被调用的时候)。

当一个程序需要调用导入函数时，他将会调用过程链接表(Procedure Linkage Table, .plt段)中的一段特定代码。就像列表1展示的那样，每个导入函数在PLT中有一段特定代码，其执行无条件跳转到相关的GOT条目。

在符号解析结束后，GOT条目已经包含了实际函数的地址，所以执行能够无缝地进入导入的库中。当函数返回时，控制流返回到PLT中特定代码的调用者位置，故剩下的PLT代码不会被执行。不过，当程序刚启动时，GOT条目被初始化为一个指向相关PLT代码第2条指令的地址。这部分代码将会将导入函数的标识(以一个Elf_Rel实例在.rel.plt段中偏移的形式)压栈，然后跳到.plt段开头PLT0的代码处。这回，PLT0的代码，将GOT[1]的值压栈并间接跳转至GOT[2].这两个GOT表的条目有着特殊的含义，动态装载器在开始时给他们填充了特殊的内容:

– GOT[1]. 一个指向内部数据结构的指针，类型是link_map，在动态装载器内部使用，包含了进行符号解析需要的当前ELF对象的信息。

– GOT[2]. 一个指向动态装载器中_dl_runtime_resolve函数的指针。

总的来说，PLT条目只是进行了以下函数调用:

_dl_runtime_resolve(link_map_obj, reloc_index)

这个函数使用link_map_obj参数来取得解析导入函数(使用reloc_index参数标识)需要的信息，并将结果写到正确的GOT条目中。在_dl_runtime_resolve解析完成导入符号中，控制流就交到了那个函数手中，使得解析的过程对调用者来说完全透明。下次PLT代码调用时则会直接进入目标函数执行。

```
100 PLT0:                   196 ; .plt.got start
100     push *0x200         196 ; Empty entry
106     jmp *0x204          196 0
110 printf@plt:             200 ; link_map object
110     jmp *0x208          200 &link_map_obj
116     push #0             204 ; Resolver function
11B     jmp PLT0            204 &_dl_runtime_resolve
120 read@plt:               208 ; printf entry
120     jmp *0x20C          208 0x116
126     push #1             20C ; read entry
12B     jmp PLT0            20C 0x126
```

Listing 1: PLT与GOT的例子

link_map结构体包含了动态装载器加载ELF对象需要的全部信息。每个link_map实例都是一条双向链表的一个节点，而这个链表保存了所有加载的ELF对象的信息。

3.4 符号版本

ELF标准提供了一个可以导入一个特定版本符号的机能。这个特性被用于从一个特定的库中导入函数。例如，使用版本标识GLIBC_2.2.5，就可以从2.2.5版本的GNU C标准库中导入fopen这个C标准库函数。.gnu.version_r段保存了版本的定义，形式是Elf_Verdef结构体。

一个动态符号与指向它的Elf_Verdef的关联保存在.gnu.version段中，其中有一个Elf_Verneed结构体组成的数组，每个元素对应动态符号表中的一项。这个结构体只有一个域: 一个16位的整数，表示.gnu.version_r段中的下标。

得益于这样的布局，动态连接器使用Elf_Rel结构体成员r_info中的下标同时作为.dynsym段和.gnu.version段的下标。理解这一过程非常重要，因为Leakless之后将被它所扰。

3.5 .dynamic段和RELRO

动态装载器从.dynamic段收集所有它需要的关于ELF对象的信息。.dynamic段由Elf_Dyn结构组成，一个Elf_Dyn是一个键值对，其中存储了不同类型的信息。相关的条目已经在表1中展示，它们保存着特定段的绝对地址。有一个例外是DT_DEBUG条目，它保存的动态装载器内部数据结构的指针。这个条目是为了调试的需要由动态装载器初始化的。

d_tag	d_value
DT_SYMTAB	.dynsym
DT_STRTAB	.dynstr
DT_JMPREL	.rel.plt

d_tag	d_value
DT_PLTGOT	.got.plt
DT_VERNEED	.gnu.version
DT_VERSYM	.gnu.version_r

Table 1: .dynamic段的条目，d_tag是键，d_value是值。

一个攻击者如果能干扰这些值，那将会造成安全威胁。为此，一种称作RELRO(重定位只读 RELocation Read Only)保护机制被引入了动态装载器。RELRO有2种形式：部分和完全。

[部分RELRO] 在这种模式下，一些段(包括.dynamic)在初始化后将会被标识为只读。

[完全RELRO] 除了部分RELRO，惰性解析会被禁用: 所有的导入符号将在开始时被解析，.got.plt段会被完全初始化为目标函数的最终地址，并被标记为只读。此外，既然惰性解析被禁用，GOT[1]与GOT[2]条目将不会被初始化为之前在3.3节中提到的值。

可以看到，RELRO显著地增加了复杂性，Leakless为了能在这些对抗措施下工作，必须解决这个问题(它也做到了)。

值得注意的是之前提到的link_map结构出于内部用途考虑，在l_info域中保存了.dynamic段中大多数条目的指针构成的一个数组。既然动态装载器完全地信任这个域的内容，那么Leakless将有能力巧用这个结构达成自己的目的。

==========

4. **攻击**

Leakless使攻击者只用名字就能够调用任意库函数，不需要关于内存布局以及漏洞程序库的信息。为了达到这个目标，Leakless活用了动态装载器，强迫其解析请求的函数。由于它和内存破坏漏洞的破坏性有着同样的根源: 可控数据和不可控数据的混杂，所以这样的攻击同样可能。在栈溢出的攻击案例中，可控数据的问题就出在保存的返回地址上。对于动态装载器来说，可控数据就是众多用于符号解析的数据结构。特别地，函数的名字，保存在.dynstr段中，与返回地址相似: 当函数被调用，它也指定了一个特定的执行目标。

动态装载器认为它接收到的参数都是值得信任的，因为它假设这些都是直接由ELF文件提供的或者是它自己在开始时初始化的。然而，当一个攻击者能够修改这些数据时，这个假设就不成立了。一些动态装载器(FreeBSD)会验证自己接收到的输入。然而，他们还是完全地信任控制结构，但这些也会被Leakless轻易地破坏。

Leakless被设计用于利用一个存在的漏洞(指缓冲区溢出等)。Leakless的输入包括可执行ELF文件，一组ROP Gadgets的集合，和攻击者希望调用的库函数名称(典型的例子就是execve())。有了这些信息，Leakless输出一段ROP的攻击载荷(payload)能够执行需要的库函数，且能够绕过多种应用在二进制文件上的加固技术。这段ROP链通常来说非常短: 依赖于二进制文件中的减缓技术，需要3到12次不等的写操作。一些Leakless产生的输出的例子能够在Leakless代码库的文档里找到 [17].

Leakless不需要任何关于库地址和内容的信息; 我们假设ASLR在所有的动态库上启用且不能获得关于它们的任何知识。然而，我们需要假设可执行文件不是“位置无关的”，所以它们会被加载到内存中的特定位置。我们在7.2节中讨论了这个限制，并且在6.2节中展示了位置无关可执行(Position Independent Executables, PIE)文件在现代操作系统中的分布情况。

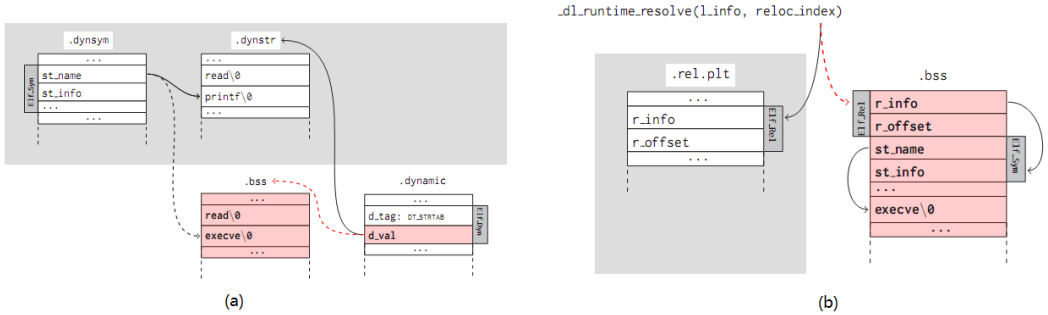

Figure 2: 攻击图示。阴影背景表示只读的内存，白色背景表示可写的内存，红色或加粗的部分表示攻击者伪造的数据

a) 4.1节中攻击的例子。攻击者能够改写DT_STRTAB条目的内容，欺骗动态装载器使其认为.dynstr段在.bss段中，并在那里伪造一个假的字符串表。当动态装载器尝试解析printf时将会使用不同的基地址来寻找函数的名称，最终实际会解析并执行execve.

b) 4.2节中攻击的例子。传递给_dl_runtime_resolve的参数reloc_index超出了.rel.plt段并最终落在.bss段，在那里攻击者伪造了Elf_Rel结构。这个重定位项指向一个就位于其后的Elf_Sym结构，而Elf_Sym结构中的index同样超出了.dynsym段。这样这个符号就会包含一个相对.dynstr地址足够大的偏移使其能够达到这个符号之后的一段内存，那里保存着这个将要调用的函数的名称。

在多数情况下，Leakless并不依赖目标系统上运行的动态装载器的实现和版本，不过有些攻击需要一些小的改动以适应不同的动态装载器实现。

值得注意的是Leakless的目标，即获得一个库函数的地址并执行它，与libdl库中的dlsym函数十分相似。但在实际当中这个函数很少被应用程序使用，所以，它的地址一般攻击者也不知道。

4.1 基础情形

就像第3节与图1中展示的那样，动态装载器从.rel.plt中的Elf_Rel结构开始工作，顺着其中的下标找到.dynsym段中对应Elf_Sym结构的位置，并最终使用它确定待解析符号的名称(在.dynstr段中的一段字符串)。最简单的调用任意函数的办法就是使用希望的函数的名称覆盖字符串表中的条目，然后再调用动态装载器，但这是不可能的，因为保存着动态符号字符串表的段，即.dynstr，是不可写的。

然而，动态装载器是从.dynamic段的DT_STRTAB条目中获得.dynstr段的地址的，而且DT_STRTAB条目的位置是已知的，默认情况下也可写。这样，就像图2a中展示的那样，我们可以将这个条目的d_val域覆盖为一个指向攻击者控制内存区域的指针(典型的就是.bss或.data段)。这块内存区域上将会包含一段字符串，比如execve。到了这一步，攻击者需要选择一个已经存在的符号，它的偏移在伪造的字符串表中正好指向execve的位置，接着调用其对应的符号解析重定位过程。可以通过将其重定位项的偏移压栈并跳转到PLT0实现。

这种方式非常简单，但仅当二进制程序的.dynamic段可写时有效。对于使用部分或完全RELRO编译的二进制程序，需要使用更复杂的攻击。

4.2 绕过部分RELRO

就像我们在3.3节中解释的那样，_dl_runtime_resolve函数的第二个参数是Elf_Rel条目在重定位表(.rel.plt段)中对应当前请求函数的的偏移。动态装载器将这个值加上.rel.plt的基地址来得到目标Elf_Rel结构的绝对地址。然而多数动态装载器实现不去检查重定位表的边界。这就表明如果一个大于.rel.plt的值传到_dl_runtime_resolve中，装载器将会认为特定的地址上的数据是一个Elf_Rel结构并使用它，即使那里已经超出了.rel.plt段的范围。

就像图2b显示的那样，Leakless计算一个能够将_dl_runtime_resolve导向到攻击者控制的内存空间的下标值。然后它制造一个Elf_Rel结构，并填写r_offset的值为一个可写的内存地址来将解析后的函数地址写在那里。同理，r_info的值将会是一个将动态装载器导向到攻击者控制内存的下标。Leakless会将一个伪造的Elf_Sym对象放在那个下标对应的位置，其中的st_name域，这个值也大到足以达到攻击者控制的内存。在这段内存最后，Leakless会放置将要执行的函数的名称。

总之，Leakless将这一条符号解析过程中需要使用的结构链全部都伪造了出来，完全控制了对于攻击者控制内存中内容的“函数调用”过程。在这之后，Leakless将计算好的假Elf_Rel结构的偏移压栈，并调用PLT0代码。

然而，这个方法会受到几个限制。首先，Elf_Rel的下标需要是正数，因为r_info域在ELF标准中规定是一个无符号整数。这就意味着在实际中这块可写的内存空间(例如.bss段)必须是位于.dynsym段之后。在我们的评估中，情况总是满足的。

另一个限制是ELF会使用在3.4节中提到的符号版本系统。在这种情况下，Elf_Rel.r_info域不仅用作动态符号表中的下标，也用作符号版本表(.gnu.version段)中的下标。通常来说，Leakless能够自动的满足这些限制，除了x86-64中使用huge pages的小型二进制程序 [32]. 我们在附录A中详述了关于符号版本的额外限制。当这些限制不能被满足时，必须使用一个替代的方法。这就需要活用动态装载器，通过破坏其内部数据结构来改变动态解析的过程。

4.3 破坏动态装载数据

我们回想起_dl_runtime_resolve的第一个参数是一个指向link_map数据类型的指针。这个结构体，包含了ELF可执行文件的信息，而且这些内容会被动态装载器完全地信任。此外，Leakless可以获得有漏洞程序的GOT表的第二个条目，它的位置是确定已知的。

回想3.5节中link_map的结构，在l_info域中，包含着.dynamic段所有条目指针构成的一个数组。这些指针就是动态链接器拿来定位符号解析过程中使用的对象的。就像图3中显示的那样，通过覆盖这个数据结构的一部分，Leakless能够将l_info域中的DT_STRTAB条目指向一个特意制造的动态条目，那里指向一个假的动态字符串表。结果，攻击者就可以将攻击简化为4.1节中的基础情形了。

这个技术较上一节中的技术而言有着更加广泛的适用性，因为它没有特定的限制。特别的，它对于使用huge pages的小型64位ELF同样适用。然而，相比于之前只依赖于标准ELF的特性的攻击，在这种情况(和下一节要叙述的情况)下，我们需要假设特定glibc的结构(link_map)布局是已知的。每个动态装载器有它自己的结构实现，故当面对不同的动态装载器时就需要做一些小的改动。需要注意的是link_map的布局在同一种动态装载器的不同版本之间也可能不同。然而，他们显得非常稳定，尤其是glibc中的相关结构从2004年起就没变过。

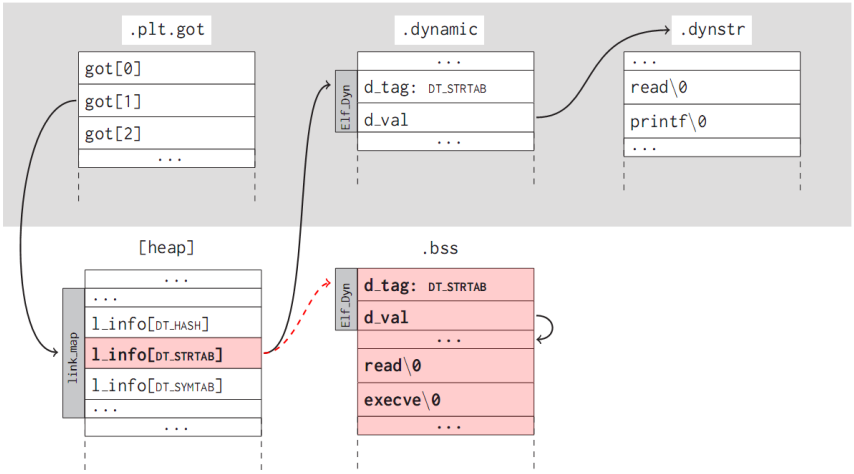

Figure 3: 4.3节中攻击的例子。攻击者通过解引用GOT的第二项来取得link_map结构。在这个结构中破坏保存DT_STRTAB指针的l_info域。它的值被设成一个伪造的动态条目的地址，那里指向了一个位于.bss段中的假的动态字符串表。

4.4 完全RELRO的情形

Leakless可以绕过完全RELRO的保护。

当完全RELRO应用时，所有的重定位将在加载时完成，不会有惰性解析的过程，并且link_map结构的地址和在GOT中的_dl_runtime_resolve也不会被初始化。所以，像普通技术绕过部分RELRO那样直接获得它们的地址是不可能的。然而，从动态表的DT_DEBUG条目中间接恢复这两个值仍然是可能的。DT_DEBUG条目的值是动态装载器在加载时设置好的，它指向一个r_debug类型的数据结构。这个数据结构保存着调试器用来标识动态装载器的基地址并拦截相应事件需要的信息。此外，这个结构的r_map域保存着一个指向link_map链表头部的指针。

Leakless破坏了这个链表中描述ELF可执行文件的第一个节点，使得用于保存DT_STRTAB的l_info条目指向一个假的动态字符串表的指针。具体情形如图4所示。

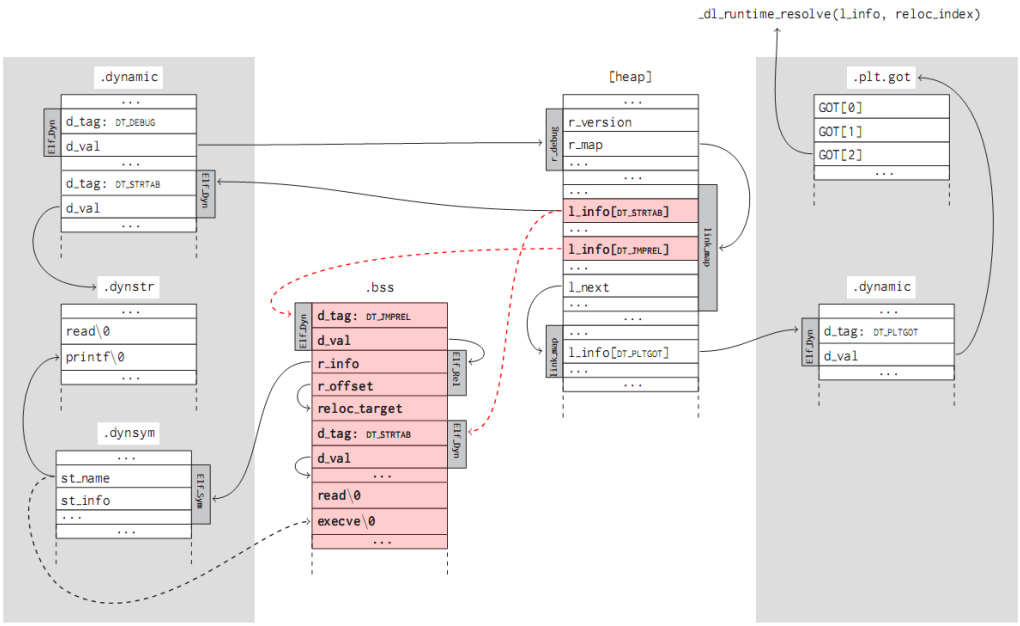

Figure 4: 4.4节中的攻击图示。阴影背景表示只读的内存，白色的背景表示可写的内存，红色与加粗的部分表示攻击者伪造的数据。攻击者使用DT_DEBUG这个动态条目来获取r_debug结构，接着，解引用r_map域从而得到主程序的link_map结构，然后像第3节中展示的那样破坏l_info[DT_STRTAB]。

因为完全RELRO的缘故.got.plt是只读的，攻击者需要伪造一个重定位项。为此，他破坏l_info[DT_JMPREL]使其指向一个假的动态条目，而这个动态条目则指向一个重定位项。这个重定位项引用了已经存在的printf符号，但r_offset则指向一块可写的内存区域。

接着攻击者同样需要恢复_dl_runtime_resolve函数的指针，因为完全RELRO现在它在主程序的GOT中已经不存在了，所以他解引用l_info域中的第一个link_map结构取得描述第一个共享库的link_map，而这个共享库是不被完全RELRO保护的。攻击者通过l_info[DT_PLTGOT]域来得到对应的动态条目(右侧的.dynamic)，接着是.plt.got段(总是在右侧)，其中的第二个条目里就有_dl_runtime_resolve的地址。

在这之后，Leakless必须调用_dl_runtime_resolve，将刚刚破坏的link_map结构作为第一个参数传过去，并将一个新的.dynsym偏移作为第二个参数传过去。然而，就像之前提到的那样，_dl_runtime_resolve因为完全RELRO的缘故在GOT中已经不存在了。所以，Leakless必须在另一个ELF对象的GOT表中找到它的地址，换句话说，就是一个被程序使用而没有完全RELRO保护的库。在大多数情况下，只有ELF可执行文件本身是被完全RELRO保护的，但库并不会。这是因为RELRO是在牺牲性能的前提下，用来加固一些被认为比较"有风险"的特定应用程序的。在共享库上应用完全RELRO将会影响所有使用这个库应用程序的性能，所以库文件一般是不受保护的。因为各个库在链表上的顺序是确定的，Leakless可以解引用link_map中的l_next项来得到不被完全RELRO保护的库文件的link_map，解引用它的l_info项得到对应的DT_PLTGOT动态条目的值，再解引用它的值(即这个库GOT的基地址)，就可以从GOT中获得_dl_runtime_resolve的地址了。

Leakless接下来必须要克服以下问题: _dl_runtime_resolve不仅仅会调用目标函数，还会尝试将它的地址写到正确的GOT项中。如果这件事发生，程序就会崩溃，因为完全RELRO保护下GOT是不可写的。我们可以通过伪造link_map中的DT_JMPREL动态条目来绕过这个问题。原本DT_JMPREL指向.rel.dyn段，Leakless将其指向攻击者控制的一块内存区域，那里写有一个Elf_Rel结构，且其r_offset域指向一块可写的内存区域，其r_info指向我们的目标符号。所以，当一个库被解析的时候，它的地址将会被写到一个可写的位置，程序就不会崩溃了，而且请求的函数也将会被执行。

==========

5. **实现**

Leakless将会分析二进制文件以确定它的技术是否适用，接着制造必要的数据，然后生成一段ROP链来实现所选技术。至于发现最初始的漏洞以及自动提取有用的gadgets那并非我们的工作，它们已经在很多著作中被很好地研究和实现过了 [6,16,19,20,34,38].我们将Leakless设计成与多种gadget发现技术兼容，并实现了一个手动的后端(用户可以给程序提供gadgets)，另外还有一个使用ROPC [22] 的后端。ROPC是一个以Q [34] 提出的方法为基础实现的一个自动化ROP编译器的原型。

我们还开发了一个小型的测试套件，由一个具有基于栈的缓冲区溢出的C程序组成，同时提供无保护，部分RELRO和完全RELRO的版本。这个测试套件可以在

x86, x86-64, arm架构的GNU/Linux系统和x86-64架构的FreeBSD系统上运行。

5.1 需要的Gadgets

Leakless包含了4种用于不同加固措施的利用技术。应用这些不同的技术需要提供不同的gadgets。表2是对这些gadgets类型的一个总结。write_memory gadget主要用于在已知地址伪造数据结构，deref_write gadget用于遍历和破坏数据结构(尤其是link_map).deref_save和copy_to_stack gadgets是用在完全RELRO的情况中的。前者的目的是将link_map和_dl_runtime_resolve的地址保存在一个已知位置，而后者的则是用来将link_map和重定位项的下标放到栈上然后调用_dl_runtime_resolve，因为使用PLT0已经不可行。

		RELRO			
Signature	Implementation	N	P	H	F
write_memory $(destination, value)$	$*(destination) = value$	✓	✓	✓	✓
deref_write$(pointer, offset, value)$	$*(*(pointer)+offset) = value$			✓	✓
deref_save $(destination, pointer, offset)$	$*(destination) = *(*(pointer)+offset)$				✓
copy_to_stack $(offset, source)$	$*(stack\_pointer+offset) = *(source)$				✓

Table 2: 多种方法需要的Gadgets。”Signature”列代表gadget的名字和它接受的参数，”Implementation”代表gadget行为的类C伪代码。最后四列指示了某个gadget是否在第4节里对应的方法中需要。”N”表示没有RELRO，”P”表示部分RELRO，”H”表示部分RELRO且为使用huge pages的小型64位程序，”F”则表示完全RELRO。

对于感兴趣的读者，我们提供了Leakless在两组不同的缓解技术保护下进行利用的深度样例，放在Leakless代码库的文档中 [17].

==========

6 评估

我们使用4种方法对Leakless进行了评估。首先我们确定了我们的技术对于不同动态装载器实现的适用性。接着分析了多个流行的GNU/Linux以及BSD的发行版(Ubuntu, Debian, Fedora和FreeBSD)中的二进制文件，从而确定易受我们攻击影响的二进制文件所占的比率。然后我们将Leakless应用在对真实世界中Wireshark的一个有漏洞版本的利用中，以及一个针对Pidgin的更加复杂的攻击中。最后我们使用一个图灵完全的ROP编译器来实现Leakless的方法和两个以前使用的技术，并比较他们生成出的链的大小。

6.1 动态装载器

为了展示Leakless的普适性，尤其是针对不同的基于ELF的平台，我们调查了几种动态装载器的实现。特别地，我们发现GNU C标准库(也就是在GUN/Linux发行版中广泛使用的glibc)的动态装载部分，其他一些Linux实现例如dietlibc, uClibc和newlib(在嵌入式系统中广泛使用)，以及OpenBSD和NetBSD的实现都含有可以被Leakless利用的漏洞。另一种嵌入式库，musl，则不会受到我们方法的影响因为它不支持惰性装载。Bionic，Android中使用的C标准库，同样不可利用因为它只支持PIE的二进制文件。最有趣的例子，不同于所有我们分析的装载器，是FreeBSD的实现。实际上，它是唯一一个会对传进_dl_runtime_resolve的参数进行边界检查的。所有其他的装载器完全信任传入的参数。不仅如此，所有被分析的装载器都完全地信任控制结构，而Leakless会在多数攻击中破坏这个结构。

总结来说，在我们分析的装载器中，只有2个在设计上对Leakless是免疫的: musl, 它不支持惰性符号解析; 以及bionic, 它只支持PIE可执行文件。此外，因为FreeBSD的动态装载器会进行边界检查，4.2节中的技术已经不适用了。不过其他的技术还是可以起效。

6.2 操作系统调研

为了能明白Leakless对真实世界的操作系统的影响，我们对几个Linux和BSD发行版中默认安装的二进制程序进行了一次调研。特别地，我们检查了所有在/sbin,/bin,/usr/bin和/usr/bin目录下的程序，并将它们按照Leakless技术的适用性进行分类。我们考虑的发行版有Ubuntu 14.10, Debian Wheezy, Fedora 20, 和FreeBSD 10。我们同时使用了这些系统的x86和x86-64的版本。在Ubuntu和Debian上，我们另外安装了LAMP(Linux, Apache, MySQL, PHP)栈作为模拟一个典型服务器部署、配置的尝试。

我们将程序分为了以下5类：

[未保护] 此类包括没有启用RELRO或PIE的程序。对于这些程序，Leakless可以应用在4.1节中介绍的基础技术。

[部分RELRO] 开启了部分RELRO的程序，但是没有开启PIE。在这种情况下，Leakless可以应用4.2节中介绍的技术。

[部分RELRO(huge pages)] 开启了部分RELRO，且使用huge pages的小型程序，它们需要Leakless使用4.3节介绍的技术。

[完全RELRO] 开启了完全RELRO的程序，Leakless需要使用4.4节展示的技术。

[不受影响] 最后是使用PIE的程序，它们不受Leakless的影响(更多的讨论将在7.2中进行)

这项调研的结果在进行归一化后，如图5所示。我们能够确定，在Ubuntu中，84%的程序受到至少一项技术的影响，另有16%被PIE技术保护。在Debian中，Leakless可以应用在86%的程序中。Fedora则含有76%的易受影响程序。有趣的是，FreeBSD没有程序开启RELRO和PIE，所以100%的程序都会受到Leakless影响。

另外，我们还针对这些系统中的共享库进行了一次调研。我们发现，平均来说，只有11%的库启用了完全RELRO的保护。这对Leakless来说是个好消息: 对于一个给定的程序，能够找到加载的没有完全RELRO保护的库的概率极其地高，并且即使一个漏洞程序使用了RELRO，Leakless可以应用它的完全RELRO攻击来绕过。这样一来，RELRO作为一项缓解措施的作用基本已经微乎其微，除非它在系统中大范围使用。

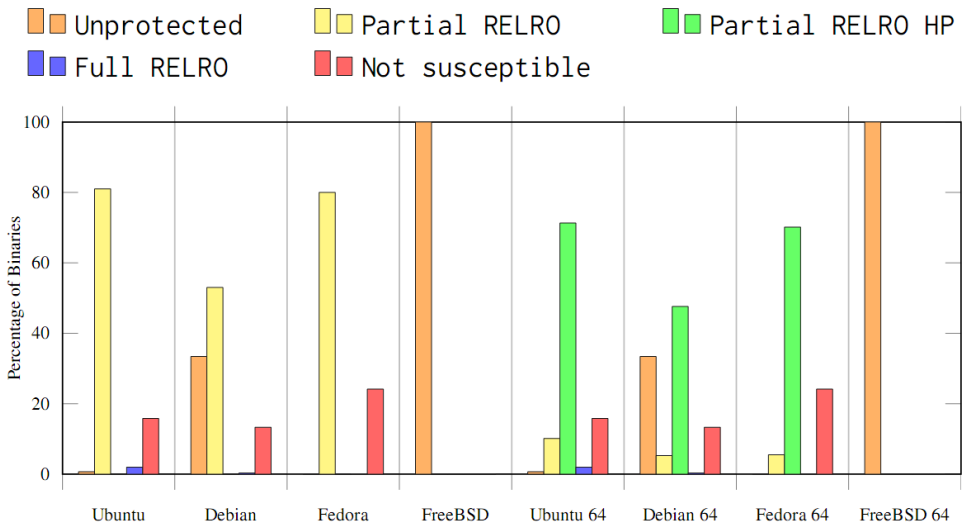

Figure 5: 目标发行版中默认安装程序的分类。被标识为未保护，部分RELRO，部分RELRO HP和完全RELRO分别需要应用4.1, 4.2, 4.3, 4.4节中的攻击技术。而对于不受影响的程序，Leakless的方法不适用。

6.3 案例研究: Wireshark

我们进行了一项将Leakless应用在一个不会与攻击者有直接联系的程序上的案例研究。换句话说，利用必须一击完成，并且没有地址布局信息以及库的内容信息。

我们选择了一个近期(2014年4月)的漏洞 [7]，一个在Wireshark 1.8.0到1.8.13以及1.10.0到1.10.6中MPEG协议解析器的栈缓冲区溢出漏洞。我们在用部分RELRO与完全RELRO编译的Wireshark 1.8.2上进行这项实验。二者都是在x86-64的Debian Wheezy系统上编译，使用GNU C语言库，并且没有其它类似PIE和stack canaries的防护。

我们使用Leakless的手动后端来确定需要的gadgets以构建4种必要的原始功能(在5.1节中描述): write_memory, deref_write, deref_save 和 copy_to_stack. 在Wireshark的例子中，找到能够满足这些原始功能的gadgets是非常容易的。

Leakless能够使用4.2节和4.4节的技术构建一个一击完成(one-shot)的利用。在两种情况下，它都能够使动态装载器调用glibc中的execve函数来启动一个我们选择的程序。

6.4 案例研究: Pidgin

我们同样将Leakless应用在Pidgin上来实行更加复杂的利用，这是一个流行的多协议即时消息客户端。而且，我们想在不进行任何异常系统调用(例如execve("/bin/sh")，这可能触发入侵检测系统报警)的情况下进行一次恶意操作。我们使用了x86架构下开启RELRO编译的Pidgin 2.10.7版本。

为了达到这个目的，我们进行了一个特殊设计的利用，使其假装自己进入应用逻辑里合法的函数中: 通过一个代理打开隧道连接。这种攻击的思想是一个即时通讯服务的提供者利用像CVE-2013-6487 [14] 这样的漏洞来获得代码执行，并且使用Pidgin的全局代理设置，将所有的即时通讯流量重定向到一个第三方的服务器上，从而造成聊天窃听。

一旦我们确定了必要的gadgets来使用Leakless对抗完全RELRO保护，调用libpurple.so(应用程序核心逻辑所在)中的函数就非常容易了，接着进行下面列表2中的C代码的等价操作

```
void *p, *a;
p = purple_proxy_get_setup(0);
purple_proxy_info_set_host(p, "legit.com");
purple_proxy_info_set_port(p, 8080);
purple_proxy_info_set_type(p, PURPLE_PROXY_HTTP);

a = purple_accounts_find("usr@xmpp", "prpl-xmpp");
purple_account_disconnect(a);
purple_account_connect(a);
```

Listing 2: Pidgin攻击

有趣的是，有些库提供的函数没有被导入到Pidgin程序中，这样在没有Leakless的情况下仅使用一个阶段的载荷完成攻击将非常具有挑战性。

6.5 ROP链的长度比较

为了证明Leakless方法的效率，我们将其与现有的允许攻击者调用任意库函数的两项技术进行了比较。第一项技术首先从.plt.got段向后扫描库，直到发现ELF的头部，随后向前扫描发现攻击者想要调用的函数的特征。这种方法是可行的，但并不十分可靠，因为一个库的不同版本(或实现)可能无法通过单一的特征就确定。第二项技术要可靠一些，因为它实现了一个完整的符号解析的过程，就像动态装载器一样。

我们实现了这两种方法，使用一种x86架构上基于Q [34]的图灵完全ROP编译器，称为ROPC [22]。我们在部分RELRO与完全RELRO两种情况下与Leakless的ROPC后端进行了比较。为了完整起见，我们同样包括了Leakless的手动后端，使用用户指定的gadgets.

事实上，ROP链长度是十分重要的，一个漏洞往往伴随着固定长度的载荷限制。为了测量Leakless的ROP链长度的影响，我们收集了Metasploit Framework [31] (一个可以自动进行多种软件已知漏洞利用的工具)中所有漏洞的载荷长度限制。我们发现1303个漏洞中的946个有最大载荷长度限制，它们的平均最大载荷长度为1332字节。为了展示自动生成复杂利用的可行性的增长，我们在图中增加了每种技术能够自动生成足够短的ROP链来利用的Metasploit漏洞占其总漏洞的百分比。

结果，在为ROPC的测试程序生成的ROP链长度方面和对Metasploit的漏洞利用的可行性方面，这些技术的表现如表3所示。Leakless全面超过了现有的技术，不仅是在进行初始调用的ROP链绝对长度上，在每次额外调用上的开销也更小。这使它在进行像6.4节中的复杂攻击中更加有效。

Technique	First call	Subsequent	Feasibility
ROPC - scan library	3468 bytes	+340 bytes	16.38%
ROPC - symbol resolution	7964 bytes	+580 bytes	8.67%
Leakless partial RELRO	648 bytes	+84 bytes	73.78%
Leakless full RELRO	2876 bytes	+84 bytes	17.44%
Leakless* partial RELRO	292 bytes	+48 bytes	95.24%
Leakless* full RELRO	448 bytes	+48 bytes	88.9%

Table 3: ROPC为6.5节中的每种技术生成的ROP链的长度，以及Leakless手动后端生成的结果(*)。第2列代表从攻击准备到第一次调用需要的字节数，第3列则表示后来的调用每次需要的字节数。最后，第4列显示了按照其第一次调用的大小生成的ROP链能够利用的Metasploit中的漏洞比率。

==========

7 讨论

在这节中，我们将讨论与Leakless有关的几个方面: 为什么它提供给攻击者的能力很有价值，在什么情况下它最适合，它的局限性在哪里，以及如何缓解其带来的影响。

7.1 Leakless的应用

Leakless作为漏洞利用开发者军械库中的一件强力工具，主要能够在三个领域帮助他们: 功能重用，一击利用，以及ROP链的缩短。

[一击利用] 所有利用都可以使用Leakless来简化，我们设计它的目标是使得那些需要信息泄漏漏洞的利用在无法进行信息泄漏时仍然可行。这类程序中有一大部分都是文件格式解析器。

解析文件格式的代码极其复杂，正因如此，当遇到不可信的输入时，就有内存破坏的倾向。有很多这样的例子: 过去10年解析图片的libpng库有27条CVE记录 [10], libtiff则有53条 [11]。复杂格式的解析器甚至更加糟糕: 过去5年多媒体库ffmpeg已经累计了170条CVE记录 [9].不仅仅局限于多媒体类的库。Wireshark，一个网络包分析器，有285条CVE记录，多数是因为网络协议分析插件的漏洞 [12].

这些库，以及和它们类似的其它库，往往是离线工作的。用户可以先下载一个媒体或者PCAP文件，然后使用这些库来解析。当漏洞触发的时候，攻击者无法与受害者建立一个直接连接来接受泄漏的信息或者发送额外的攻击载荷。不仅如此，大部分这种格式都是被动的，也就是说(不像PDF)，它们不含有攻击者模拟2个阶段利用需要的脚本。结果，即使这些库是有漏洞的，对它们的利用也是非常困难，不可靠，或者完全不可实行。通过避免信息泄漏的步骤，Leakless能够使得这些利用能更简单，可靠，可行。

[功能重用] Leakless让攻击者能够调用程序加载的库中的任意函数。事实上，漏洞程序不需要真正"导入"这些函数，它只需要对这些库进行链接即可(即可以调用库中的其它函数)。这就带来了一些便利。

首先，被大部分程序链接的C标准库，包含了几乎所有系统调用的封装函数(例如read()、execve()等等)。这就意味着Leakless能够进行任何系统调用，并且可以在没有系统调用gadget的情况下使用一条短小的ROP链完成。

不仅如此，就像6.4节中说的，Leakless能够轻易地重用程序逻辑中现有的功能。这很重要，原因有二。

第一个原因，通过将利用伪装成程序正常的行为能够帮助攻击者进行更加隐秘的攻击。当标准利用的套路被诸如seccomp [2], AppArmor [1] 或SELinux [25]这种保护机制封锁时，这就是决定性的。

第二个原因，根据攻击者的具体目标，重用函数功能往往比执行任意命令更好。除了在Pidgin的案例研究中的攻击之外，攻击者可以悄悄地在Firefox浏览器中启用不安全的密码套件或者SSL的不安全版本，仅仅需要一个SSL_CipherPrefSetDefault调用 [24].

[缩短ROP链] 就像6.5节说的，Leakless能够产生比现有技术更短的ROP链。实际上，在很多场合，Leakless通过产生小于1KB的ROP链就能完成任意函数的调用。在很多漏洞都有输入长度的限制的情况下，这是很重要的结果。例如，我们在Pidgin的案例研究中利用的漏洞允许的最大ROP链长度为1KB。普通的ROP编译技术无法自动生成这个漏洞的载荷，而Leakless却可以通过自动生成不超过长度限制的ROP链来调用任意函数。

7.2 限制

Leakless最大的限制就是在没有信息泄漏的情况下无法处理位置无关可执行文件(Position Independent Executables, PIEs)。这是使用ROP的技术的通病，因为ROP链中的gadgets的绝对地址必须确定。除此之外，在没有程序基地址的情况下，Leakless也无法定位它要破坏的动态装载器结构。

当遇到PIE的程序时，Leakless需要攻击者能够提供应用程序的基地址，这应该能够通过一个信息泄漏漏洞得到(或者使用BROP技术 [5])。这就打破了Leakless不需要信息泄漏就可以进行操作的能力。不过Leakless仍然是进行利用最方便的方式，因为不需要泄漏库的地址和内容。除此之外，依据具体情况，仅仅泄漏程序的地址或许比泄漏整个库的内容更加可行。与其它需要后者的技术不同，Leakless只需要前者。

实际当中，PIE因为关系到性能开销并不太常见。有测试结果显示PIE在x86处理器上的额外开销平均为10%，而在x86-64处理器上，得益于相对指令指针(instruction-pointer-relative)的寻址方式，平均有3.6%的额外开销 [28] 。

因为PIE相关的额外开销开销太大，大部分发行版只将那些"有风险"的程序上开启PIE。例如，资料显示，Ubuntu的官方支持程序包(即在主要仓库中的包)中只有27个开启了PIE，没有开启的则有超过27000个包 [40]。正如6.1节中说的，PIE可执行程序只占我们调研过的所有系统中很小一部分。

7.3 对抗措施

有多种能够对抗Leakless的方法，不过它们都有缺点。在本节中我们将会分析最适用的几种方法。

[位置无关可执行] 一个快速的对抗措施就是让所有系统中的可执行文件都是位置无关的。这虽然会封锁Leakless的自动操作(就像7.2节中讨论的那样)，但当存在任何信息泄漏时Leakless的技术还是可以应用的。为此，再加上PIE相关的性能开销，我们认为这节中讨论的其他对抗措施比它更为适用。

[禁用惰性装载] 当LD_BIND_NOW环境变量被设置时，动态装载器将会完全地禁用惰性装载。这就意味着，所有为了程序和其依赖库的导入将在程序开始时被解析。由此带来的副作用就是_dl_runtime_resolve函数的地址不会被装载到任何库的GOT中，Leakless就无法工作了。这就相当于在整个系统上应用了完全RELRO。所以，他也同样会带来不可忽略的性能上的额外开销。

[禁用DT_DEBUG] 最后，Leakless将会使用DT_DEBUG这个动态条目来绕过完全RELRO。而这个条目是调试器用来拦截与装载有关事件使用的。现在，这个域总是会被加载，为Leakless的完全RELRO绕过敞开大门。为了关闭这个坑，可以对动态装载器进行修改让它只在有调试器存在或有明确定义的环境变量存在时才去初始化这个值。

[对装载器控制结构更好的保护] Leakless非常依赖于“动态装载器的控制结构很好获得”这一事实，并且它们的位置也是众所周知的。与将它们装载到一个已知位置相比，把这些结构更好地保护或者隐藏在内存中会是个好主意。例如，在 [29] 中说的那样，对于这些结构以及任何会提供符号解析控制数据的段，应当在初始化后将它们标为只读。这样的改进能够彻底抹消Leakless破坏这些结构的能力，并能防止将控制流重定向到敏感函数的这类攻击。

此外，修改装载的过程让它使用一张表来记录link_map结构，并给_dl_runtime_resolve在表中赋予一个下标，而不是使用一个直接的指针，这样能够打破Leakless对完全RELRO的绕过。然而，这种改变将会破坏之前编译的任何二进制程序的兼容性。

[隔离动态装载器] 将动态装载器从目标程序的地址空间中隔离是一个有效的对抗措施。举例来说，诺基亚的塞班操作系统，其使用了微内核结构，动态装载器是在一个单独的进程中实现的，这个进程作为一个系统服务器(system server)与内核交互。这就保证了动态装载器的控制结构不会被程序破坏。所以，这实际上已经使得Leakless失效了。然而，这种对抗措施将会给程序的整体性能带来相当大的影响，因为进程间通信(Inter-Process Communication, IPC)也需要额外开销。

总体来说，这些缓解措施要么带来运行时的性能开销(PIE或装载器隔离)，要么带来装载时的性能开销(非惰性装载和系统范围的RELRO)，要么需要修改装载的过程(DT_DEBUG的禁用以及装载器控制结构隐藏).从长远来看，我们相信一个考虑安全的对动态装载器的重新设计将会让我们都受益匪浅。在短期内，也有几种针对Leakless可选的保护方案，不过都会有性能损失。

==========

8 结论

在这篇文章中，我们展示了Leakless这种新技术，能够使用动态装载器的功能来使得攻击者在利用中使用任意的、安全关键(security-critical)的库函数，而不需要知道这些函数在程序内存中的位置。而在以前，实现这种利用则需要先进行信息泄漏这一步骤。

因为Leakless使用的是ELF二进制格式规定的特性，所以攻击能够跨架构，跨操作系统，跨动态装载器来进行。此外，我们展示了我们的技术能够绕过像RELRO这种保护动态解析过程中重要控制结构的加固方案。最后，我们提出了几种对抗Leakless的措施，并讨论了它们的优缺点。

==========

参考文献

[1] AppArmor. http://wiki.apparmor.net/.

[2] A. Arcangeli. seccomp. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt.

[3] A. Baratloo, N. Singh, and T. K. Tsai. Transparent Run-Time Defense Against Stack-Smashing Attacks. In USENIX Annual Technical Conference, General Track, pages 251–262, 2000.

[4] M. Bauer. Paranoid penguin: an introduction to Novell AppArmor. Linux Journal, 2006(148):13, 2006.

[5] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazieres, and D. Boneh. Hacking blind. In Proceedings of the 35th IEEE Symposium on Security and Privacy, 2014.

[6] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In Security and Privacy (SP), 2012 IEEE Symposium on, pages

380–394. IEEE, 2012.

[7] Common Vulnerabilities and Exposures. CVE-2014-2299. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2299.

[8] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings, volume 2, pages 119–129. IEEE, 2000.

[9] CVEDetails.com. ffmpeg: CVE security vulnerabilities. http://www.cvedetails.com/product/6315/Ffmpeg-Ffmpeg.html.

[10] CVEDetails.com. Libpng: Security Vulnerabilities. http://www.cvedetails.com/vendor/7294/Libpng.html.

[11] CVEDetails.com. Libtiff: CVE security vulnerabilities. http://www.cvedetails.com/product/3881/Libtiff-Libtiff.html.

[12] CVEDetails.com. Wireshark: CVE security vulnerabilities. http://www.cvedetails.com/product/8292/Wireshark-Wireshark.html.

[13] CWE. CWE/SANS Top 25 Most Dangerous Software Errors. http://cwe.mitre.org/top25/.

[14] N. V. Database. NVD – Detail – CVE-2013-6487. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-6487.

[15] A. Di Federico, A. Cama, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Leakless source code repository. https://github.com/ucsb-seclab/leakless.

[16] S. Dudek. The Art Of ELF: Analysis and Exploitations. http://bit.ly/1a8MeEw.

[17] T. Dullien, T. Kornau, and R.-P. Weinmann. A Framework for Automated Architecture-Independent Gadget Search. In WOOT, 2010.

[18] M. Fox, J. Giordano, L. Stotler, and A. Thomas. Selinux and grsecurity: A case study comparing linux security kernel enhancements. 2009.

[19] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations. In USENIX Security, pages 49–64, 2013.

[20] C. Heitman and I. Arce. BARFgadgets. https://github.com/programa-stic/barf-project/tree/master/barf/tools/gadgets.

[21] inaz2. ROP Illmatic: Exploring Universal ROP on glibc x86-64. http://ja.avtokyo.org/avtokyo2014/speakers#inaz2.

[22] P. Kot. A Turing complete ROP compiler. https://github.com/pakt/ropc.

[23] P. Menage. Cgroups. Available on-line at: http://www.mjmwired.net/kernel/Documentation/cgroups.txt, 2008.

[24] Mozilla. SSL_CipherPrefSetDefault. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS/SSL_functions/sslfnc.html#_SSL_CipherPrefSetDefault_.

[25] National Security Agency. Security-Enhanced Linux. http://selinuxproject.org/.

[26] Nokia. Symbian OS Internals – The Loader. http://developer.nokia.com/community/wiki/Symbian_OS_Internals/10._The_Loader#The_loader_server.

[27] H. Orman. The Morris worm: a fifteen-year perspective. IEEE Security & Privacy, 1(5):35–43, 2003.

[28] M. Payer. Too much PIE is bad for performance. 2012. https://nebelwelt.net/publications/12TRpie/gccPIE-TR120614.pdf.

[29] M. Payer, T. Hartmann, and T. R. Gross. Safe Loading – A Foundation for Secure Execution of Untrusted Programs. In Proceedings of the 2012 IEEE Symposium on Security and Privacy, SP '12, pages 18–32, Washington, DC, USA, 2012. IEEE Computer Society.

[30] Phrack. Phrack – Volume 0xB, Issue 0x3a. http://phrack.org/issues/58/4.html.

[31] Rapid7, Inc. The Metasploit Framework. http://www.metasploit.com/.

[32] RedHat, Inc. Huge Pages and Transparent Huge Pages. https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Performance_Tuning_Guide/s-memory-transhuge.html.

[33] Santa Cruz Operation. System V Application Binary Interface, 2013. http://www.sco.com/developers/gabi/latest/contents.html.

[34] E. J. Schwartz, T. Avgerinos, and D. Brumley. Q: Exploit Hardening Made Easy. In USENIX Security Symposium, 2011.

[35] H. Shacham. 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, pages 552–561. ACM, 2007.

[36] R. Shapiro, S. Bratus, and S. W. Smith. "Weird Machines" in ELF: A Spotlight on the Underappreciated Metadata. In Proceedings of the 7th USENIX Conference on Offensive Technologies, WOOT'13, pages 11–11, Berkeley, CA, USA, 2013. USENIX Association.

[37] L. Szekeres, M. Payer, T. Wei, and D. Song. SoK: Eternal war in memory. In Security and Privacy (SP), 2013 IEEE Symposium on, pages 48–62. IEEE, 2013.

[38] The Avalanche Project. Avalange – a dynamic defect detection tool. https://code.google.com/p/avalanche/.

[39] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning. On the Expressiveness of Return-into-libc Attacks. In Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection, RAID'11, pages 121–141, Berlin, Heidelberg, 2011. Springer-Verlag.

[40] Ubuntu. Ubuntu Wiki – Security/Features. https://wiki.ubuntu.com/Security/Features#Built_as_PIE.

[41] R. N. Watson, J. Anderson, B. Laurie, and K. Kenn-away. Capsicum: Practical Capabilities for UNIX. In USENIX Security Symposium, pages 29–46, 2010.

==========

附录A 符号版本带来的挑战

在3.4节中我们介绍了符号版本的概念，在4.2节中我们提到了它的使用让Elf_Rel.r_info的含义有了新的限制。在本附录中我们会列出这几个限制，并解释Leakless如何自动地检验并满足它们。

A.1 符号版本的限制

当符号版本使用时，Elf_Rel.info域将同时作为动态符号表的下标以及符号版本表(.gnu.version段)的下标。符号版本表由Elf_Verneed值构成，一个0值或1值有着特殊的含义，它能够停止符号版本的过程，这对攻击者来说再好不过了。

为了理解这些限制到底是怎样的，我们将引入一些定义和命名规则。idx为Leakless计算出的Elf_Rel.r_info值所指示的下标，baseof(x)函数能够返回段x的基地址，sizeof(y)返回结构体y的大小，*是指针解引用操作(即取值)。我们定义了如下变量：

sym = baseof(.dynsym) + idx · sizeof(Elf Sym)

ver = baseof(.gnu.version) + idx · sizeof(Elf Verneed)

verdef = baseof(.gnu.version_r) + *(ver) · sizeof(Elf Verdef)

为了能够进行攻击，需要满足下面的条件:

1. sym指向攻击者控制的一块内存区域，并且
2. 下面的某一条能够满足:

(a) ver指向包含一个0或者一个1的内存区域，或

(b) ver指向攻击者控制的内存区域，那里会被写上0值，或

(c) verdef指向攻击者控制的内存区域，在那里将会放上一个正确构造的Elf_Verdef结构。

如果不能满足上述条件将会出现访问一段未被映射的内存区域或者符号解析过程失败的现象，并最终让程序终止。

Leakless可以在大多数情况下自动满足这些限制。一个典型的成功情形就是让idx值指向一个值为0的版本下标或者在.text段中(它大多跟在.gnu.version段后面)，并指向一个.data和.bss段中符号。一个值得注意的例外是x86-64架构下支持huge pages的ELF二进制文件 [32]。使用huge pages意味着内存页是以2MB的边界对齐的，所以包含只读段(特别是.gun.version和.text)的segment与可写的segment(含有.bss和.data)离得很远。这就让找到一个合适的idx值变得很难。

A.2 huge page的问题

huge page的效果如下面例子所示:

“`

$ readelf –wide -l elf-without-huge-pages

Program Headers :

Type VirtAddr MemSiz Flg Align

…

LOAD 0x00400000 0x006468 R E 0x1000

LOAD 0x00407480 0x0005d0 RW 0x1000

…

$ readelf –wide -l elf-with-huge-pages

Program Headers :

Type VirtAddr MemSiz Flg Align

…

LOAD 0x00400000 0x00610c R E 0x200000

LOAD 0x00606e10 0x0005d0 RW 0x200000

…

“`

在第一个例子里可写的segments与程序开头的距离是以KB计的，但如果用了huge pages距离就超过了2MB，这样就无法找到一个有效的idx值了。

对于这个问题，可以使用两种方法解决。

第一种选择是在只读的segment(一般是.text段)中为Elf_Verneed找到一个0值。令ro_start, ro_end, ro_size分别为只读segment的开始地址，结束地址以及大小，rw_start, rw_end, rw_size为可写segment的对应值。那么它们需要满足以下条件:

ro_start ≤ ver < ro_end

rw_start ≤ sym < rw_end

这里，最难满足的是.dynsym或.gnu.version位于ro_start的开头的情况。假设上述两条都能满足，我们就有:

idx · sizeof(Elf_Verneed) < ro end − ro start

idx · sizeof(Elf_Sym) ≥ rw start − ro start

也就是:

idx · sizeof(Elf_Verneed) < ro size

idx · sizeof (Elf_Sym) ≥ 2 MiB

我们知道Elf_Verneed和Elf_Sym在64位ELF中各占2个字节与24个字节，我们可以计算满足这个不等式组条件下ro_size的最小值。结果是170.7KB。如果.rodata段小于这个大小，那就必须使用其他方法。

第二种选择是将Elf_Verneed放置于可写的segment。在这种情况下，攻击的要求以下面的不等式组来定义:

rw_start ≤ ver < rw_end

rw_start ≤ sym < rw_end

如果我们同样来考虑最坏情况并作出之前那样的假设，我们得到:

idx · sizeof(Elf_Verneed) ≥ rw_start − ro_start

idx · sizeof(Elf_Sym) < rw_start − ro_start + rw_size

也就是:

idx · sizeof(Elf_Verneed) ≥ 2 MiB

idx · sizeof(Elf_Sym) < 2 MiB + rw_size

我们现在可以计算出能够使其成立的可写segment(rw_size)大小的下界: 22MB。然而，这个值没有道理地大，并让我们得出结论: 这种方法在使用huge pages的小型ELF文件中是不适用的