



单位代码 10006

学 号 10211044

分 类 号 TP312

北京航空航天大学

B E I H A N G U N I V E R S I T Y

毕业设计 (翻译)

计算机病毒的理论实验

学 院 名 称	<u>软件学院</u>
专 业 名 称	<u>软件工程专业</u>
学 生 姓 名	<u>康乔</u>
指 导 教 师	<u>原仓周</u>

2014 年 06 月

北京航空航天大学

本科生毕业设计（论文）任务书

I、毕业设计（论文）题目：

计算机病毒的理论实验

II、毕业设计（论文）使用的原始资料（数据）及设计技术要求：

III、毕业设计（论文）工作内容：

IV、主要参考资料：

____ 软件 ____ 学院 ____ 软件工程 ____ 专业类 ____ 102112 ____ 班

学生 ____ 康乔 ____

毕业设计(论文)时间： ____ 2013 ____ 年 ____ 11 ____ 月 ____ 15 ____ 日至 ____ 2014 ____ 年 ____ 06 ____ 月 ____ 10 ____ 日

答辩时间： ____ 2014 ____ 年 ____ 06 ____ 月 ____ 12 ____ 日

成 绩： ____

指导教师： ____

兼职教师或答疑教师（并指出所负责部分）：

____ 系（教研室）主任（签字）： ____

注：任务书应该附在已完成的毕业设计（论文）的首页。



本人声明

我声明，本论文及其研究工作是由本人在导师指导下独立完成的，在完成论文时所利用的一切资料均已在参考文献中列出。

作者： 康乔

签字：

时间： 2014 年 06 月



计算机病毒的理论实验

学 生： 康 乔

指导教师： 原仓周

摘 要

这篇文章介绍了“计算机病毒”并且检验了它们导致计算机系统的广泛破坏的潜力。文章展示了基本的理论结果、病毒防御在大型系统中的不可行性、防御方案以及一些实验结果。

关键词： 计算机病毒，系统完整性，数据完整性



Computer Virus Theory and Experiments

Author: Kang Qiao

Tutor: Yuan Cangzhou

Abstract

This paper introduces computer virus and examines their potential for causing widespread damage to computer system. Basic theoretical results are presented, and this infeasibility of viral defense in large classes of systems is shown. Defensive schemes are presented and several experiments are described.

Key words: Computer Virus, System Integrity, Data Integrity



目 录

1 介绍	1
2 代码插入与劫持技术分析	3
2.1 关于 ELF 格式的讨论	3
2.1.1 ELF 文件类型	3
2.1.2 链接视图与执行视图	3
2.1.3 文件空间和地址空间	6
2.2 ELF 插入技术分析	7
2.2.1 利用 <code>nop</code> 指令串插入代码	7
2.2.2 在代码段的最后分区插入代码	8
2.2.3 在代码段之前插入代码	10
2.3 二进制程序劫持技术分析	12
2.3.1 修改 ELF 文件入口点	12
2.3.2 在函数头增加跳转	15
2.3.3 通过篡改动态链接数据结构	16
3 VxWorks 代码插入方案分析与设计	18
3.1 VxWorks 镜像文件格式	18
3.2 VxWorks 运行时内存布局	19
3.3 VxWorks 代码插入方案设计	20
3.3.1 利用 <code>nop</code> 指令串插入	20
3.3.2 通过扩展 <code>.bss</code> 节进行插入	21
4 代码劫持方案的分析与设计	23
4.1 VxWorks 系统镜像种类	23
4.2 VxWorks 系统启动调用流程	24
4.3 VxWorks 下代码劫持方案设计	25
4.3.1 直接跳转及其可行性证明	25
4.3.2 间接跳转及其可行性证明	27



4.3.3 两种方案的对比和分析	28
5 实例: mini-notify——代码插入与函数劫持在 VxWorks 中的实现	30
5.1 背景	30
5.1.1 Linux 下的文件监控系统 inotify 介绍	30
5.1.2 VxWorks 下的 dosFs 文件系统	30
5.2 mini-notify 的目标	31
5.3 环境搭建与配置	32
5.3.1 Linux 主机	32
5.3.2 Windows 主机与 VxWorks 虚拟机	32
5.4 mini-notify 方案设计	33
5.4.1 劫持位置选择	33
5.4.2 函数劫持方案	34
5.4.3 代码插入方案	34
5.5 mini-notify 的实现	35
5.6 测试与评价	35
结论	37
参考文献	38
附录 A ELF 注入工具源代码	39
附录 B 实现文件监控功能的源代码	42



1 介绍

这篇文章解释了一个主要的计算机安全性问题，即病毒。病毒的有趣之处在于它能依附于其他的程序并使它们也成为病毒。考虑到现代计算机系统的广泛共享，携带着木马程序^[2]的病毒的威胁越来越值得注意。尽管在阻止信息非法传播^[2]的执行政策上有相当多的工作已经完成，并且许多系统已经采用它们来避免这种类型的攻击^{[2][3]}，但是在如何保持信息进入一个领域而不导致破坏方面的工作却很少^[2]。在计算机系统中有许多不同类型的信息路径，其中一些是合法授权的，另一些可能是隐秘的^[2]，这常常被用户所忽略。在这篇文章中我们将忽略隐秘性的信息路径。

一般设施提供正确的保护方案^[2]，但是它们依赖于只对正在进行的攻击能有效防御的安全性政策。就算是一些相当简单的防护系统也不能被证明是“安全的”^[2]。防止拒绝服务需要停止程序的检测^[2]。给系统中的信息流进行精确标记的问题已被证明是一个 NP-完全问题。对于在用户之间的不可信信息的传播，进行某种方式的防卫，这已经被测试过了^[2]，但是一般依赖能力来提高程序正确性，这也是一个众所周知的 NP 完全问题。

施乐公司的蠕虫程序已经说明，该程序可以通过网络繁殖，并且可以偶然地造成设备无法服务。在后来的变种中，一个“核心战争”的游戏中，能够让两个程序互相争斗。一些针对这一主题的，被匿名作者发布的其他的变种，是基于在程序之间进行的夜间游戏的上下文中的。术语病毒也被用于结合 APL 的作者的地方一般开始时调用每个函数反过来调用一个预处理器，以增加默认 APL 翻译。

一个广泛传播的安全问题的潜在威胁被分析了^[2]，对于政府、金融、商业和学术设施的危害是严重的。另外，这些机构倾向于使用特别的保护机制来应对这些特有的威胁，而不是研究全面而彻底的理论上的保护机制^[2]。当前的军队保护系统，很大程度上依赖隔离机制^[2]；然而，新的系统开始使用多层的用法^[2]。没有一个发布出来的被建议使用的系统可以定义或者部署一种方法，用来阻止一种病毒。

这篇文章中我们提出了一个防止计算机病毒的新问题。首先我们检测了病毒的感染特性，并且展示了共享信息的传递闭包可以被感染。当和木马一起使用的时候，可以导致设备广泛停止服务以及数据的非授权的操作。一些对计算机病毒的实验结果表明，病毒对于普通的和高安全性的操作系统都是一个强大的威胁。传播的途径、信息流的传递以及信息解释的一般性是防止计算机病毒的关键特性，这些特性将在下面一一解释。分析表明，有潜力阻止病毒攻击的系统具有有限的传递性和有限的共享性，或者



完全不能共享也不具有一般的信息解释（图灵能力）。只有前一种情形对于现代社会具有研究的意义。一般来说，使用先验和运行时分析对病毒的防护的研究是不可判定的，并且没有防护，治愈是很难或者无法实现的。

我们检查了一些被提出来的对策，这些对策都针对特定的情景，来对病毒的特质进行就事论事的分析。限制传播系统被认为是值得期待的，但是精确的部署是很棘手的，并且模糊的政策一般来说会导致有用的系统随着时间越来越少。系统范围内的病毒抗体的使用也被测试了，但普遍来说，还是要依赖针对特定棘手问题的解决方案。

综上可知对计算机病毒是一个非常重要的研究领域，因为它和其他领域的潜在性应用。现代系统对于病毒攻击提供很少或者没有提供防护，并且当代唯一有效的“安全”措施就是隔离了。



2 代码插入与劫持技术分析

2.1 关于 ELF 格式的讨论

对 ELF 格式文件进行二进制层面的修改，首先要对其文件格式有比较深入的了解。ELF 的文件格式在 [1] 中有着详尽的描述，在此不再对 ELF 文件格式的细节进行罗列，而是介绍一些关键的知识，并讨论这些知识与我们的插入工作有何关联，以及它们如何能决定插入的成败。

2.1.1 ELF 文件类型

ELF 文件主要有三种类型，分别是：

- 可重定位文件 (Relocatable File)
- 可执行文件 (Executable File)
- 共享目标文件 (Shared Object File)

可重定位文件就是常见的.o 文件，也叫目标文件。这类文件包含了代码和数据，但是其中的外部地址引用都未重定位，可以被与其他可重定位文件链接生成可执行文件或者共享目标文件。

可执行文件在 Linux 下往往没有扩展名，例如/bin/ls 就是典型的这一类文件。他们共同的特点是可以直接执行。但是 ELF 可执行文件在也分为两类：静态链接的和动态链接的。区别在于静态链接的文件所需要的库都已经被链接至可执行文件中。而动态链接的文件在运行时才与库链接。Gcc(GNU Compile Collection, GNU 编译器套装) 默认生成动态链接的可执行文件；通过增加 -static 选项，可以生成静态链接的可执行文件。

共享目标文件即 Linux 下的.so 文件，也常被叫做动态共享库。它们类似.o 文件，并没有被完全链接，但是区别在于，它们用于在运行时与动态链接的可执行文件进行链接。

我们的操作目标，即 VxWorks 系统镜像，可以归类为静态链接的可执行文件。它的特点是已经被完全链接和重定位。

2.1.2 链接视图与执行视图

每一种 ELF 文件都有着相似的基础结构。ELF 文件格式刚刚被生成的时候，是存储在磁盘中；然而它最终的用途是被加载和执行。这里有点类似从一个“程序”到“进



程”的转化。当 ELF 存在于磁盘中时，系统以某种视角来看待它；然而当它被加载到主存，系统就需要以另一种视角来看待它。这两种不同的视角，就是 ELF 文件的链接视图和执行视图，如图2.1。任何 ELF 文件都存在链接视图，而执行视图存在于共享目标文件和可执行文件中。

链接视图	执行视图
ELF 头部	ELF 头部
程序头部表（可选）	程序头部表
节区 1	段 1
...	
节区 n	段 2
...	
...	...
节区头部表	节区头部表（可选）

图 2.1 链接视图与执行视图

- 链接视图

在对 ELF 文件进行链接的过程中，链接器从链接视图来看待 ELF 文件，链接器认为 ELF 文件被划分为一个个节区（section），节区中包含代码或者数据。链接器通过合并这些段来形成新的 ELF 文件。

- 加载视图

当加载器对 ELF 可执行文件执行加载操作时，它眼中的 ELF 文件是被划分为了一个个段（segmentation）。在一个使用分页机制的系统中，每个段都是按照页大小（例如 4KB）进行对齐的。加载器的任务就是把每个段的页拷贝到主存的合适位置。（使用虚拟存储器的系统中，加载器不拷贝数据，只建立虚拟内存和磁盘文件的关联，在此可以简单认为加载器按照段结构进行了 ELF 文件到主存的拷贝）

ELF 文件中，存在以下的数据结构，来实现链接视图和执行视图。（在.o 文件中，不存在执行视图）

- 节区与节区头部表

图2.2展示了/bin/ls 程序的节区头部表的一部分。在链接视图中，节区用于划分文件中不同类型的数据。例如.text 节用于存放代码，.data 节用于存放数据等。链接两个目标文件的其中一步，就是把节区头表为是所有节区的一个索引，通过解析节区头表可以找到文件的所有节区。

- 段和程序头部表



```
Section Headers:
```

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.interp	PROGBITS	08048134	000134	000013	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	08048148	000148	000020	00	A	0	0	4
[3]	.note.gnu.build-id	NOTE	08048168	000168	000024	00	A	0	0	4
[4]	.hash	HASH	0804818c	00018c	000028	04	A	6	0	4
[5]	.gnu.hash	GNU_HASH	080481b4	0001b4	000020	04	A	6	0	4
[6]	.dynsym	DYNSYM	080481d4	0001d4	000050	10	A	7	1	4
[7]	.dynstr	STRTAB	08048224	000224	00004a	00	A	0	0	1
[8]	.gnu.version	VERSYM	0804826e	00026e	00000a	02	A	6	0	2
[9]	.gnu.version_r	VERNEED	08048278	000278	000020	00	A	7	1	4
[10]	.rel.dyn	REL	08048298	000298	000008	08	A	6	0	4
[11]	.rel.plt	REL	080482a0	0002a0	000018	08	A	6	13	4
[12]	.init	PROGBITS	080482b8	0002b8	000026	00	AX	0	0	4
[13]	.plt	PROGBITS	080482e0	0002e0	000040	04	AX	0	0	16
[14]	.text	PROGBITS	08048320	000320	000180	00	AX	0	0	16
[15]	.fini	PROGBITS	080484a0	0004a0	000017	00	AX	0	0	4
[16]	.rodata	PROGBITS	080484b8	0004b8	00000e	00	A	0	0	4
[17]	.eh_frame_hdr	PROGBITS	080484c8	0004c8	00001c	00	A	0	0	4
[18]	.eh_frame	PROGBITS	080484e4	0004e4	000060	00	A	0	0	4
[19]	.init_array	INIT_ARRAY	08049544	000544	000004	00	WA	0	0	4
[20]	.fini_array	FINI_ARRAY	08049548	000548	000004	00	WA	0	0	4
[21]	.jcr	PROGBITS	0804954c	00054c	000004	00	WA	0	0	4
[22]	.dynamic	DYNAMIC	08049550	000550	0000f0	08	WA	7	0	4
[23]	.got	PROGBITS	08049640	000640	000004	04	WA	0	0	4
[24]	.got.plt	PROGBITS	08049644	000644	000018	04	WA	0	0	4
[25]	.data	PROGBITS	0804965c	00065c	000008	00	WA	0	0	4
[26]	.bss	NOBITS	08049664	000664	000004	00	WA	0	0	4

图 2.2 节区头部表

图2.3是/bin/ls 的程序头部表。段是执行视图中的对 ELF 文件的划分，ELF 文件被加载和执行的就是一个一个段。一般来说，每个节区映射到唯一的一个段，而一个段包含不同的节区。段没有名字，但有类型和属性：类型一般指明该段是否需要加载至内存，例如需要加载器加载到主存中的段的类型为“LOAD”；而段的属性则表明了该段能否被读/写/执行。

```
Program Headers:
```

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00100	0x00100	R E	0x4
INTERP	0x000134	0x08048134	0x08048134	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x00544	0x00544	R E	0x1000
LOAD	0x000544	0x08049544	0x08049544	0x00120	0x00124	RW	0x1000
DYNAMIC	0x000550	0x08049550	0x08049550	0x000f0	0x000f0	RW	0x4
NOTE	0x000148	0x08048148	0x08048148	0x00044	0x00044	R	0x4
GNU_EH_FRAME	0x0004c8	0x080484c8	0x080484c8	0x0001c	0x0001c	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4

```
Section to Segment mapping:
Segment Sections...
```

00	
01	.interp
02	.interp .note.ABI-tag .note.gnu.build-id .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame_hdr .eh_frame
03	.init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04	.dynamic
05	.note.ABI-tag .note.gnu.build-id
06	.eh_frame_hdr
07	

图 2.3 程序头部表



在一个典型的 Linux 可执行文件中，第一个 LOAD 类型的段一般具有可读和可执行的属性，该段对应的节区为所有含有代码的节区，例如 .init 节、.text 节和 .fini 节。

与节区头部表类似，程序头部表是所有段的一个索引。程序头部表还指明了每个段所包含的节区。

2.1.3 文件空间和地址空间

上文介绍了 ELF 文件的两种不同的视图，理解两个视图的区别和联系能帮助我们找到可以插入代码的位置。然而，这还是不够的。我们插入的代码必须加载到一定的地址才能被运行。这就需要理解 ELF 文件与地址空间的关系。

在节区头部表和程序头部表中，不但指定了节区或者段在文件中的位置，还指定了节区或者段在内存中的加载地址。例如如图 2.2 所示，/bin/ls 程序包含的 .text 节在文件中的偏移是 0x3020，而加载地址为 0x8048320。

打个比方，如果在论文中新加入了一个章而没有更新目录，那么这一章很可能会被读者忽略。类似地，在 ELF 文件中插入代码，如果不更新相应的数据结构（节头表和程序头表），除了增大了文件的体积之外和导致文件不能运行之外，没有任何意义。

由于插入的代码会影响文件内容在文件中的偏移，因此相应节头表和段表的偏移字段必须被更新。如果这项工作没有出现错误，那么修改后的文件至少是可以正常地像原先一样执行的。然而，要让我们插入的代码也在加载时被映射到程序的地址空间中，那插入的部分必须拥有合法的载入地址。

这就带来了两个问题，一是哪些地址空间是空闲并可以使用的，二是如何让插入的代码刚好使用这部分地址空间。这两个问题将一直存在于后文对具体技术的讨论中。

第一个问题与操作系统和硬件平台有关，一个经典的方法是利用段间对齐产生的空间来插入代码。在一个运行在 x86 上的 Linux 系统来说，由于段的起始地址要按照 4KB 的页大小进行对齐，因此加载段之间将会有 0 ~ 4KB 的空余地址空间。插入的代码可以用于映射到这部分空间。

第二个问题包含很多细节性的修改，通过小心地把代码插入到一个段的最后，并修改段的大小，同时更新后面所有段和节的文件偏移，就可以让我们的代码刚好用上这一段地址空间。

如果解决了这两个问题，程序再次执行时，插入的代码就已经被加载到相应的地址空间了。剩下的问题则会根据需求变化。例如如何让插入的代码得到执行和在何时执行，以及如何在插入的代码中使用库函数等。这些问题将留在具体技术中讨论。



2.2 ELF 插入技术分析

2.2.1 利用 nop 指令串插入代码

首先介绍一种最简单的代码插入方式。一本优秀的反汇编教材 [2] 中讲述了利用 nop 串向 ELF 文件插入代码的方法。由于 gcc 编译器在生成 ELF 文件时, 会对函数的起始地址按照 0x10 进行对齐。多余的空间会使用 nop 指令 (操作码为 0x90) 进行填充。由于地址对齐而留下的 nop 指令串便可以供我们随意使用。

这些可用的 nop 指令串有这样两个条件: 在 nop 串第一个 nop 指令的前面必须是 ret 或者 jmp 指令, 而且最后一个 nop 串之后的地址能够被 10h 整除。代码 2.1 是 /bin/ls 文件反汇编之后的一部分代码, 代码中从地址 805ac43h 开始, 到 805ac4fh 结束的 nop 指令串满足上述的两个条件 (前面紧挨的指令为 jmp, nop 串结束后的地址为 805ac50h)。因此这些 nop 都可以被改写, 而不影响程序运行。

1	805ac3e:	5f	pop	edi
2	805ac3f:	5d	pop	ebp
3	805ac40:	c3	ret	
4	805ac41:	eb 0d	jmp	805ac50<__sprintf_chk@plt+0x10f90>
5	805ac43:	90	nop	
6	805ac44:	90	nop	
7	805ac45:	90	nop	
8	805ac46:	90	nop	
9	805ac47:	90	nop	
10	805ac48:	90	nop	
11	805ac49:	90	nop	
12	805ac4a:	90	nop	
13	805ac4b:	90	nop	
14	805ac4c:	90	nop	
15	805ac4d:	90	nop	
16	805ac4e:	90	nop	
17	805ac4f:	90	nop	
18	805ac50:	f3 c3	repz	ret

代码 2.1 /bin/ls 中存在的 nop 指令串

虽然这样的 nop 指令串在 /bin/ls 这样的文件中有很多, 但每一段都不是太长, 容不下很多代码。一个最简单的办法就是, 在每个 nop 串快要用完的时候, 使用 jmp 或其



他指令，跳转到下一个 `nop` 串的开始处。这样我们就可以利用几乎所有的 `nop` 指令串（除非有的实在太短，以至于写不下 `jmp` 指令）。文献 [2] 中给出了一个简单的手动修改的示例，劫持一个可执行文件，在运行它时先打印字符串 “hello”。示例的工作可以归结为以下几个步骤：

- （1）修改 ELF 文件入口点处的两条指令，替换为 `jmp` 指令，跳转地址为第一个 `nop` 指令串。
- （2）在 `nop` 串中写入使用系统调用打印 “hello\n” 的汇编代码。如果一个 `nop` 串不足，则跳转后继续使用下一个 `nop` 串。
- （3）打印代码的最后，加入 ELF 入口点被修改的两条指令。
- （4）使用 `jmp` 跳转回入口点处的第三条指令。

修改完成后，当我们再次在终端中执行该可执行文件时，终端中会首先打印出一行 “hello”，接着才是它原先的工作。

然而，如果要插入大量的代码，程序中仅有的一点 `nop` 指令串能够用吗？我们做一个简单的统计，来看看一个 ELF 可执行文件中，大概能留下多少可供使用的 `nop` 指令串。我们选取了一些典型的 ELF 文件，包括一个最简单的 C 语言 “hello,world!” 程序，若干/bin 目录下的实用程序和我们的目标 VxWorks 镜像。统计结果如表2.1所示。

表 2.1 典型 ELF 文件中的 `nop` 串占据的空间


ELF 文件	文件大小（字节）	<code>nop</code> 串大小（字节）
“hello,world!” 程序	7339	30
/bin/ls	108708	921
/bin/rm	303484	1653
/bin/bash	924892	13723
VxWorks 系统镜像	1337184	21114

由此可见，随着 ELF 文件体积的增大，可用的 `nop` 串空间也几乎在正比例扩张，二者的比例大概维持在 100:1 的数量级。除了在最简单的 “hello,world!” 程序中几乎什么都插不下之外，具有一些实用功能的应用程序都有着比较可观的空间。试想一下，假若/bin/bash 中 13KB 的 `nop` 串被全部塞满恶意代码，完全可以对系统产生严重的打击。

2.2.2 在代码段的最后分区插入代码

Silvio Cesare 由于在 1998 年提出的最原始的 UNIX 病毒模型^[3]，而曾被称为 “ELF 大师（ELF master）”。在这一病毒模型中，病毒在感染其他文件时，采取了一种不同

的方式。具体来说是在第一个可加载段（俗称的代码段）最后一节的最后插入代码。这一方案具有很大的通用性，因为不论 ELF 文件有多大，两个加载段之间都有大概 4KB 的地址空间供我们插入。即使总共才只有 7KB 的“hello,world!”程序也不例外，如图2.4所示。第一个 LOAD 段的结束地址为 0x80485b4(0x8048000+0x5b4), 而第二个 LOAD 段从 0x8049f08 才开始，两个段之间有多达 0x1954 的空闲地址空间。



Program Headers:							
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x08048034	0x08048034	0x00120	0x00120	R E	0x4
INTERP	0x000154	0x08048154	0x08048154	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x005b4	0x005b4	R E	0x1000
LOAD	0x000f08	0x08049f08	0x08049f08	0x00118	0x0011c	RW	0x1000
DYNAMIC	0x000f14	0x08049f14	0x08049f14	0x000e8	0x000e8	RW	0x4
NOTE	0x000168	0x08048168	0x08048168	0x00044	0x00044	R	0x4
GNU_EH_FRAME	0x0004d8	0x080484d8	0x080484d8	0x0002c	0x0002c	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x10
GNU_RELRO	0x000f08	0x08049f08	0x08049f08	0x000f8	0x000f8	R	0x1

图 2.4 hello 程序的程序头部表

之所以会出现这么大的空闲地址空间，来源于 Linux（包括一些其他的 UNIX）的特殊段加载机制。因为如果完全按照 0x1000 来对齐段的起始地址，将会对物理内存产生巨大的浪费。从图2.4可以看出，hello 程序的第一个段只有 0x5b4 字节大小，让它完全占有一个物理段，那就会产生 $0x1000 - 0x5b4 = 0xac4$ 字节的浪费，空间利用率只有 35.6%。

针对这种浪费，Linux 的做法是通过分配更多的虚拟内存来换取物理内存的节约。具体做法是把可执行文件从偏移 0 处开始，按照页大小进行划分，划分得到的每个页面作为一个物理页面加载。这样的问题在于有的物理页面会包含两个 ELF 段，这时只需要把这样的物理段映射两次到虚拟地址空间即可。如图2.5，假设一个 ELF 文件有三个段需要加载，在使用这种段合并机制的情况下，三个段之间会产生两段空余的地址空间。

显然，在物理地址空间中，页面利用率大幅提高，但在虚拟地址空间中留下了尺寸不小的碎片，图2.5的情况是碎片大概 1 页大小，但在有些 gcc 的版本中，这一大小甚至接近两个页。这一空间便可以被我们用于插入代码。要注意的是，虽然原理上，数据段最后的剩余空间同样可以利用，但这需要修改数据段的可写属性，不如插在代码段之后更为便捷。因此在 [3] 的算法中，代码的插入位置是在代码段之后。

下面简要描述了这一插入算法：

- 1、找到 ELF 中的第一个可加载段（代码段），将该段的 filez 字段和 memsz 字段增

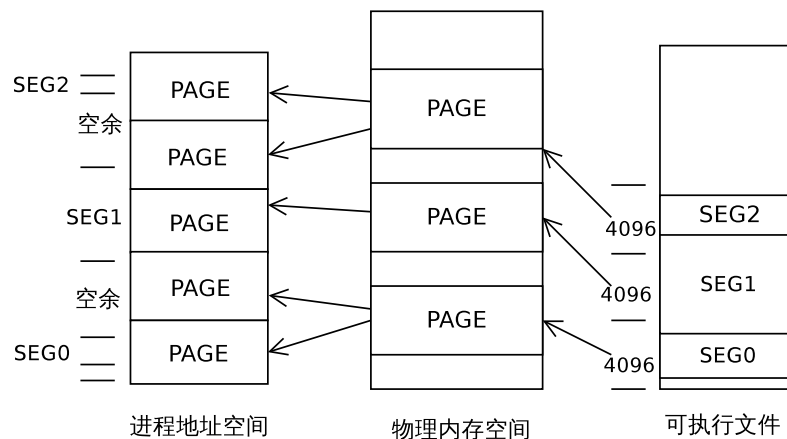


图 2.5 Linux 加载机制中的段合并

加要插入代码的大小。

- 2、对于所有在代码段之后的段，将其 offset 增加一页（4KB）的大小。
- 3、对于代码段的最后一个节区，将其 len 字段增加插入代码长度大小。
- 4、对于在代码段之后的所有节区，将其 offset 字段增加一页（4KB）的大小。
- 5、将插入代码的长度填充到 1 页（4KB），然后拷贝到预定的插入代码的位置。

这一算法提出之后，大量的针对 ELF 乃至 Linux 内核的研究都借鉴了其插入代码的思路，如 [4-8]。等

在这一算法的基础上，我实现和完善了这一方法的 python 版本并增加了一些自定义的功能。

下面借助一个例子来进一步说明其原理，假设我们有一个 ELF 可执行文件中的代码段最后插入了一小段代码。图2.6(a)和2.6(b)分别展示了在插入代码前后的程序头表的变化。我们可以很容易在这里发现一些上面的算法所描述的改变，例如第一个 LOAD 段的长度被增加了 0x26, 而从第二个 LOAD 段开始，所有的段的偏移量都增加了 0x1000，即 4K 等。类似地，节区头部表中的一些数据结构也被修改，在此就不再赘述了。

2.2.3 在代码段之前插入代码

虽然利用代码段最后可能空余的地址空间来插入代码，已经成为了一种比较成熟的手段，但这种方法的局限性仍然是不可忽视的。最重要的一点就是其插入的代码量存在限制。一般来说，如果插入代码量最好保证在 1KB 之内，一旦超过 2KB，几乎是不可能成功的。

然而，一个高效而便捷的工具 ELFsh^[9] 提供了一种新的插入思路，即把代码插入至

```
Program Headers:
Type      Offset      VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
PHDR      0x000034  0x08048034  0x08048034  0x00100 0x00100 R E  0x4
INTERP    0x000134  0x08048134  0x08048134  0x00013 0x00013 R   0x1
  [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD      0x000000  0x08048000  0x08048000  0x00544 0x00544 R E  0x1000
LOAD      0x000544  0x08049544  0x08049544  0x00120 0x00124 RW  0x1000
DYNAMIC   0x000550  0x08049550  0x08049550  0x000f0 0x000f0 RW  0x4
NOTE      0x000148  0x08048148  0x08048148  0x00044 0x00044 R   0x4
GNU_EH_FRAME 0x0004c8  0x080484c8  0x080484c8  0x0001c 0x0001c R   0x4
GNU_STACK 0x000000  0x00000000  0x00000000  0x00000 0x00000 RW  0x4
```

(a) 插入前

```
Program Headers:
Type      Offset      VirtAddr    PhysAddr    FileSiz MemSiz  Flg Align
PHDR      0x000034  0x08048034  0x08048034  0x00100 0x00100 R E  0x4
INTERP    0x000134  0x08048134  0x08048134  0x00013 0x00013 R   0x1
  [Requesting program interpreter: /lib/ld-linux.so.2]
LOAD      0x000000  0x08048000  0x08048000  0x0056a 0x0056a R E  0x1000
LOAD      0x001544  0x08049544  0x08049544  0x00120 0x00124 RW  0x1000
DYNAMIC   0x001550  0x08049550  0x08049550  0x000f0 0x000f0 RW  0x4
NOTE      0x001148  0x08048148  0x08048148  0x00044 0x00044 R   0x4
GNU_EH_FRAME 0x0014c8  0x080484c8  0x080484c8  0x0001c 0x0001c R   0x4
GNU_STACK 0x001000  0x00000000  0x00000000  0x00000 0x00000 RW  0x4
```

(b) 插入后

图 2.6 可执行文件插入代码前后的程序头表对比

代码段之前。我们知道，对于一个运行在 IA32 上的 Linux 系统来说，一个进程的地址空间中，代码段总是开始于 0x8048000，然而从 0 到 0x8047fff 这段空间一般不被使用。该工具的思路就是把需要执行的代码插入至这一段未被使用的地址空间中。

ELFsh 工具功能强大，它以一个.o 文件和一个可执行文件作为输入。不但能够讲.o 中的可执行代码插入至可执行文件中，还能插入.o 文件携带的数据节、bss 节和符号表。并能够对插入代码的所有重定位入口进行重定位。

下面简要说明其插入流程：

- 1、将.o 文件的.bss 节插入合并到可执行文件的.bss 节中。
- 2、遍历.o 文件的所有节区，判断是否具有以下条件
 - 存在标志位（说明该节区需要被加载到进程空间中）
 - 在文件中占据空间（排除.bss 节区）
 - 节区类型为值为 SHT_PROGBITS，说明该节区为程序节或者代码节。

对于同时满足这些条件的节区，再判断节区是否具有可写属性，对于具有可写属性的节区（数据节），插入到.bss 节之后；其他节区插入到第一个节区之前。

- 3、把.o 文件的符号表合并至可执行文件中。

- 4、对于.o 文件中的所有节区，如果已经被插入到了可执行文件中，并且需要重定



位，则对其进行重定位。

下面演示 ELFsh 工具的效果。我们首先编写一个简单的程序，源代码见2.2，然后把它编译为 myprintf.o 即可。我们将其注入到 ls 程序中。

```
1 #include<stdio.h>
2 void myprintf(){
3     printf("hi, you're hijacked!");
4 }
```

代码 2.2 myprintf.c 源代码

图2.7和2.8反映了 ls 程序被插入前后的节头表（只展示了前 20 个）和程序头表的变化。由图可以发现，在 ELFsh 向可执行文件注入了 9 个节区，并都位于可执行文件的第一个节区，即.interp 节之前。（此外还进行了符号表和.bss 节的合并，这里看不出来）。程序头表方面，可以发现段的数目并没有变化，但是第一个段的起始地址由 0x8048000 提前到了 0x803c000，这也说明注入的代码使用了 0x8048000 之前的地址空间。此外，位于后面的段在文件中的偏移也相应地增加了 0xc000，说明我们在.interp 段之前总共插入了 0xc000 字节的代码，即 12 个页，试想一下，如果使用 nop 插入或者插入到段间的空闲地址，这几乎是不可能完成的任务。

2.3 二进制程序劫持技术分析

2.3.1 修改 ELF 文件入口点

ELF 文件头中包含一些有关 ELF 文件的重要数据结构，一个典型的 ELF 文件头如图2.9所示。文件头对于 ELF 文件的链接视图和执行视图来说同样重要。除了标志该文件是 ELF 文件及其类型之外，头部最重要的作用在于标明了以下三点：

- 1 程序入口点位置
- 2 节区头部表位置和节区的数量
- 3 程序头部表位置和段的数量

图2.9中的“Entry Point address”即为该 ELF 文件的入口点地址。在 Linux 中，当 ELF 文件被加载完成之后，CPU 首先执行该地址处的第一条指令。因此我们只需要修改这一地址，就可以让系统首先跳转到我们的插入代码。

执行这一修改的步骤大致如下：

- 1、修改目标文件的 Entry Point address 字段为插入代码第一条指令的地址。
- 2、插入代码执行一定的操作。

Section Headers:										
[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.interp	PROGBITS	08048154	000154	000013	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	08048168	000168	000020	00	A	0	0	4
[3]	.note.gnu.build-id	NOTE	08048188	000188	000024	00	A	0	0	4
[4]	.gnu.hash	GNU_HASH	080481ac	0001ac	000020	04	A	5	0	4
[5]	.dynsym	DYNSYM	080481cc	0001cc	000050	10	A	6	1	4
[6]	.dynstr	STRTAB	0804821c	00021c	00004a	00	A	0	0	1
[7]	.gnu.version	VERSYM	08048266	000266	00000a	02	A	5	0	2
[8]	.gnu.version_r	VERNEED	08048270	000270	000020	00	A	6	1	4
[9]	.rel.dyn	REL	08048290	000290	000008	08	A	5	0	4
[10]	.rel.plt	REL	08048298	000298	000018	08	A	5	12	4
[11]	.init	PROGBITS	080482b0	0002b0	000023	00	AX	0	0	4
[12]	.plt	PROGBITS	080482e0	0002e0	000040	04	AX	0	0	16
[13]	.text	PROGBITS	08048320	000320	000192	00	AX	0	0	16
[14]	.fini	PROGBITS	080484b4	0004b4	000014	00	AX	0	0	4
[15]	.rodata	PROGBITS	080484c8	0004c8	00000e	00	A	0	0	4
[16]	.eh_frame_hdr	PROGBITS	080484d8	0004d8	00002c	00	A	0	0	4
[17]	.eh_frame	PROGBITS	08048504	000504	0000b0	00	A	0	0	4
[18]	.init_array	INIT_ARRAY	08049f08	000f08	000004	00	WA	0	0	4
[19]	.fini_array	FINI_ARRAY	08049f0c	000f0c	000004	00	WA	0	0	4
[20]	.jcr	PROGBITS	08049f10	000f10	000004	00	WA	0	0	4

(a) 插入前节头表

Program Headers:								
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align	
PHDR	0x000034	0x08048034	0x08048034	0x00120	0x00120	R E	0x4	
INTERP	0x000154	0x08048154	0x08048154	0x00013	0x00013	R	0x1	
[Requesting program interpreter: /lib/ld-linux.so.2]								
LOAD	0x000000	0x08048000	0x08048000	0x005b4	0x005b4	R E	0x1000	
LOAD	0x000f08	0x08049f08	0x08049f08	0x00118	0x0011c	RW	0x1000	
DYNAMIC	0x000f14	0x08049f14	0x08049f14	0x000e8	0x000e8	RW	0x4	
NOTE	0x000168	0x08048168	0x08048168	0x00044	0x00044	R	0x4	
GNU_EH_FRAME	0x0004d8	0x080484d8	0x080484d8	0x0002c	0x0002c	R	0x4	
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x10	
GNU_RELRO	0x000f08	0x08049f08	0x08049f08	0x000f8	0x000f8	R	0x1	

(b) 插入前程序头表

图 2.7 插入前节头表与程序头表

3、插入代码使用跳转指令，跳回原入口点的地址。

这一方法简单有效，文献 [3] 的第一代 Unix 病毒就使用了这种办法来获取控制权。图2.10展示了插入的代码如何通过修改程序入口点对程序原有的正常执行流进行篡改。在程序未被修改前，ELF 头的入口点指向代码段的起始位置，如图2.10(a)。通过修改 ELF 头的入口点，使它指向我们插入代码的位置；在插入代码的最后，跳转回原入口点的位置，以保证程序的正常执行，如图2.10(b)。

借助2.2.3中的例子，我们演示一下使用修改入口点进行劫持的效果。在完成插入后，我们只需要手动修改入口点地址，指向 myprintf 函数即可。然后手动修改该函数的最后几条指令，跳回原入口点即可。（回想一下，函数结尾往往有 nop 指令串，因此这很容易实现）。

最后的达到的效果如图2.11所示。



Section Headers:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.elfsh.hooks	PROGBITS	0803c154	000154	004fc7	00	AX	0	0	0
[2]	myprintf.o.eh_fra	PROGBITS	0804111b	00511b	001027	00	A	0	0	0
[3]	myprintf.o.rodata	PROGBITS	08042142	006142	000ffd	00	A	0	0	0
[4]	myprintf.o.text	PROGBITS	0804313f	00713f	000f94	00	AX	0	0	0
[5]	.elfsh.extplt	PROGBITS	080440d3	0080d3	000010	00	AX	0	0	0
[6]	.elfsh.altp1t	PROGBITS	08045113	009113	001008	00	AX	0	0	0
[7]	myprintf.o.eh_fra	PROGBITS	0804611b	00a11b	001027	00	A	0	0	0
[8]	myprintf.o.rodata	PROGBITS	08047142	00b142	000ffe	00	A	0	0	0
[9]	myprintf.o.text	PROGBITS	08048140	00c140	000014	00	AX	0	0	0
[10]	.interp	PROGBITS	08048154	00c154	000013	00	A	0	0	1
[11]	.note.ABI-tag	NOTE	08048168	00c168	000020	00	A	0	0	4
[12]	.note.gnu.build-i	NOTE	08048188	00c188	000024	00	A	0	0	4
[13]	.gnu.hash	GNU_HASH	080481ac	00c1ac	000020	04	A	14	0	4
[14]	.dynsym	DYNSYM	080481cc	00c1cc	000050	10	A	15	1	4
[15]	.dynstr	STRTAB	0804821c	00c21c	00004a	00	A	0	0	1
[16]	.gnu.version	VERSYM	08048266	00c266	00000a	02	A	14	0	2
[17]	.gnu.version_r	VERNEED	08048270	00c270	000020	00	A	15	1	4
[18]	.rel.dyn	REL	08048290	00c290	000008	08	A	14	0	4
[19]	.rel.plt	REL	08048298	00c298	000018	08	A	14	12	4
[20]	.init	PROGBITS	080482b0	00c2b0	000023	00	AX	0	0	4

(a) 插入后节头表

Program Headers:

Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
PHDR	0x000034	0x0803c034	0x0803c034	0x00120	0x00120	R E	0x4
INTERP	0x00c154	0x08048154	0x08048154	0x00013	0x00013	R	0x1
[Requesting program interpreter: /lib/ld-linux.so.2]							
LOAD	0x000000	0x0803c000	0x0803c000	0x0c5b4	0x0c5b4	R E	0x1000
LOAD	0x00cf08	0x08049f08	0x08049f08	0x0217c	0x0217c	RW	0x1000
DYNAMIC	0x00cf14	0x08049f14	0x08049f14	0x000e8	0x000e8	RW	0x4
NOTE	0x00c168	0x08048168	0x08048168	0x00044	0x00044	R	0x4
GNU_EH_FRAME	0x00c4d8	0x080484d8	0x080484d8	0x0002c	0x0002c	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x10
GNU_RELRO	0x00cf08	0x08049f08	0x08049f08	0x000f8	0x000f8	R	0x1

(b) 插入后程序头表

图 2.8 插入后程序头表和节头表

ELF Header:

Magic:	7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Class:	ELF32
Data:	2's complement, little endian
Version:	1 (current)
OS/ABI:	UNIX - System V
ABI Version:	0
Type:	EXEC (Executable file)
Machine:	Intel 80386
Version:	0x1
Entry point address:	0x804c070
Start of program headers:	52 (bytes into file)
Start of section headers:	107588 (bytes into file)
Flags:	0x0
Size of this header:	52 (bytes)
Size of program headers:	32 (bytes)
Number of program headers:	9
Size of section headers:	40 (bytes)
Number of section headers:	28
Section header string table index:	27

图 2.9 ELF 文件头部

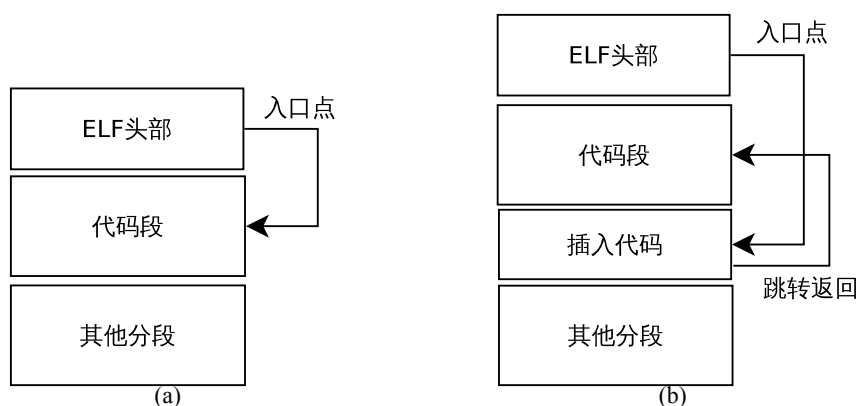


图 2.10 可执行文件代码插入前后执行流程对比

```
kq@kq:~$ ./modified_ls -l
hi, you're hijacked!
total 212948
drwxr-xr-x  4 kq  kq      4096  4月 23 13:09 buaathesis
drwxr-xr-x  3 kq  kq      4096  4月 24 13:31 codezero
drwxr-xr-x  2 kq  kq      4096  4月 24 10:01 Desktop
drwxr-xr-x  3 kq  kq      4096  5月 14 15:11 Documents
drwxr-xr-x  5 kq  kq      4096  5月 14 14:04 Downloads
drwxr-xr-x 28 kq  kq      4096  5月 14 22:59 eresl
```

图 2.11 修改后的 ls 程序运行结果

2.3.2 在函数头增加跳转

在 Linux 下的 ELF 文件中，我们可以通过修改 ELF 头部的入口点地址来实现劫持。然而使用这种方法，我们插入的代码将只会在程序开始时被运行一次。

我们可能不希望插入代码在程序执行的开始就运行，而是希望在某个函数被调用的时候，由它来截获这一函数。这时我们可以通过修改需要劫持的函数的开头几条指令来进行。

下面的例子中，我们假设要劫持某个函数，每当这个函数被调用时，都会首先输出一行“HELLO!”

一般来说，一个函数体开头的指令如代码2.3所示。

```
1 push ebp
2 mov ebp,esp
3 sub esp,0x??
```

代码 2.3 函数体开头的指令

我们把代码2.3替换为代码2.4:



```
1 push code_address      ;压入插入代码的起始地址
2 ret                    ;跳转到 code_address
```

代码 2.4 函数体开头的替换指令

我们在适合的位置插入代码2.5，

```
1 push ebp
2 mov ebp,esp
3 sub esp,0x??          ;补上被劫持函数的开头被修改的三条指令
4 push 0x000a214f
5 push 0x4c4c4548        ;压入字符串“HELLO!\n”的 ASCII 码
6 push esp               ;压入字符串的起始地址。
7 mov eax,0x????????    ;把 printf 函数的地址放入 eax 中
8 call eax               ;调用 printf 函数
9 add esp,0x8            ;恢复堆栈
10 push 0x????????      ;
11 ret                   ;返回被劫持函数第四条指令处。
```

代码 2.5 用于劫持某个函数的插入代码

这里的重点不在于我们插入的技术有多么先进，而是插入的代码提供了一种通用的劫持任意函数的方法。即无论我们在何种位置插入代码来劫持任意一个函数，总是可以通过以下几个步骤来劫持一个函数的入口：

1、修改函数的前三条指令（一般为 6 个字节）为 push 和 ret（刚好也是 6 个字节），类似代码2.4，压栈的内容为插入代码的地址，即 nop_address。

2、插入代码的前三条指令为 1 中被修改的三条指令。

3、在插入代码中进行操作。

4、恢复寄存器和堆栈（如果被修改过的话）。

5、使用 push+ret 组合返回被劫持函数原来的第四条指令处，即被劫持函数地址+6。

2.3.3 通过篡改动态链接数据结构

ELF 中文件关于动态链接的细节，应参见 [1]，然而，本条中讲述的劫持方法，不但简单明了，而且像教科书一般阐明了 ELF 中对动态库函数的引用机制。

动态链接的 ELF 可执行文件中，所有对于外部库函数的调用，都是通过一个 got 数据结构和一个段 plt 代码来实现的。



为了实现对外部函数引用的劫持工作，需要我们插入两段代码：A 段在函数入口点处执行，作用是帮助我们修改 `got`，定位到插入库函数的位置。B 段则是我们插入的库函数。这两段代码的插入可以用上一节中描述的三种方法的任意一中来进行。

两段代码的算法简要描述如下：

A:

- 1、使用系统调用，为代码段增加可写属性。
- 2、保存 `got` 中对应条目的地址，记为 `original_got`。
- 3、将 `got` 中条目的地址修改为插入的库函数的地址。

B:

- 1、执行目的操作，例如打印一行信息。
- 2、恢复 `got` 中旧的条目的地址。
- 3、恢复堆栈，调用原库函数。
- 4、保存 `got` 中的地址，覆盖 `original_got`。
- 5、将 `got` 中的地址替换为插入的库函数的地址。

在 B 代码的第 4 步要重新保存一遍 `original_got`，因为动态链接程序大都是用“延迟绑定”机制，在第一次调用库函数时，会经过 `plt` 进行一次间接跳转，并会调用动态链接器进行重定位，从而修改 `got` 中保存的地址。



3 VxWorks 代码插入方案分析与设计

VxWorks 操作系统是美国风河公司旗下的一款嵌入式实时操作系统。该系统由于其高实时性、高可靠性，在嵌入式实时操作系统领域占据一席之地，特别是应用于通信、军事、航空航天等高精尖领域中。

VxWorks 5.5 是 VxWorks 系统中一个较为经典的版本。本课题的所有研究都围绕 VxWorks 5.5 来展开，后文如果没有特别说明，所有的 VxWorks 都是指 5.5 版本。

VxWorks 作为嵌入式操作系统，与 PC 系统有很大的不同之处。体现在系统的加载启动流程，与硬件的交互等多个方面。虽然 VxWorks 也使用了 ELF 文件格式作为它的可执行文件格式，并且也是用 gcc 编译器。但是正如 2.1.3 中所述，代码插入决不只是针对 ELF 文件格式和内容的修改，还与系统的内存空间等有着密切的联系。

本章在前文列举的所有针对 ELF 代码插入技术的基础上，加上针对 VxWorks 若干特性的分析，考察这些代码插入技术在 VxWorks 中应用的可能性，并选择合适的代码插入方案。

3.1 VxWorks 镜像文件格式

VxWorks 的系统镜像文件本身就是一个 ELF 静态链接可执行文件，因此我们在前文中对 ELF 格式等方面的讨论在这里同样适用。为了寻找能够插入代码的位置，我们需要考察系统镜像文件的节头表和程序头表。图 3.1 展示了映像文件的节头表和程序头表，以及他们的映射关系。

该 ELF 文件的布局不同于一般的 Linux 下的 ELF 文件。Linux 下的 ELF 文件一般有两个加载段，一个只包含代码，拥有可读和可执行的权限；一个只包含数据，拥有可读和可写的权限。

而 VxWorks 系统镜像文件的布局要更加简单，只有一个可加载段，对应三个节，分别是 text 节，data 节和 bss 节。一般来说，ELF 文件中的代码和数据由于分别具有不同的属性（例如代码节不可写，数据节不可执行），这两种节会被映射到不同的段来映射，从而映射到不同的页面，并在页表中赋予他们不同的属性。然而 VxWorks 的做法比较特殊，代码和数据节映射到一个段，这也就意味着该段必须同时具有可读可写和可执行三类属性。

然而，由于 VxWorks 系统镜像文件只有一个段，也就导致我们无法利用段与段之间由于地址对齐而产生的地址空间。因此，我们在寻找空闲地址空间的过程中，可以

```
Section Headers:
[Nr] Name                Type          Addr      Off      Size    ES Flg Lk Inf Al
[ 0]                     NULL          00000000 000000 000000 00  0  0  0  0
[ 1] .text                PROGBITS     00308000 000060 0c3250 00 WAX 0  0 32
[ 2] .data                PROGBITS     003cb260 0c32c0 014780 00  WA 0  0 32
[ 3] .bss                 NOBITS       003df9e0 0d7a40 009880 00  WA 0  0 16
[ 4] .debug_aranges       PROGBITS     00000000 0d7a40 000060 00  0  0  0  1
[ 5] .debug_pubnames      PROGBITS     00000000 0d7aa0 001378 00  0  0  0  1
[ 6] .debug_info          PROGBITS     00000000 0d8e18 03a2f4 00  0  0  0  1
[ 7] .debug_abbrev         PROGBITS     00000000 11310c 000ee7 00  0  0  0  1
[ 8] .debug_line           PROGBITS     00000000 113ff3 00bca9 00  0  0  0  1
[ 9] .shstrtab             STRTAB       00000000 11fc9c 000071 00  0  0  0  1
[10] .symtab               SYMTAB       00000000 11fef0 017600 10 11 3028 4
[11] .strtab              STRTAB       00000000 1374f0 00f230 00  0  0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
Type      Offset    VirtAddr  PhysAddr  FileSiz MemSiz  Flg Align
LOAD      0x000060 0x00308000 0x00308000 0xd79e0 0xe1260 RWE 0x20

Section to Segment mapping:
Segment Sections...
00      .text .data .bss
```

图 3.1 VxWorks 系统映像文件的节头表和程序头表

利用两类办法:

- 1、利用.text 节之前或者.bss 节之后的地址空间
- 2、利用该加载段中的 nop 指令串

3.2 VxWorks 运行时内存布局

回顾在2.1.3中提到的关于 ELF 文件插入的两个问题: 文件空间与地址空间。要寻找合适的空闲的地址空间, 我们需要熟悉 VxWorks 运行中的详细内存布局。

图3.2是在 x86 硬件平台下的 VxWorsk 内存布局的一个简略版本。

我们的目标是寻找空闲的地址空间, 使得代码能够映射到这一块地址并被执行。然而, 从图中不难看出: 在 VxWorks 系统镜像的低地址和高地址两个方向, 均有明确的定义和用途。其中, RAM_LOW_ADRS 之前的低端内存区, 被用作初始化堆栈。系统的初始化流程借助这一区域实现函数调用。而 FREE_RAM_ADRS 之后的地址, 用于 WDB 内存池。这两段地址皆有明确的用途。这也就意味着, 我们在上一节中提到的第一类方法, 即利用.text 节之前或者.bss 节之后的地址空间进行插入的办法可能无法使用。然而利用 nop 串进行插入仍然是可行的办法之一。

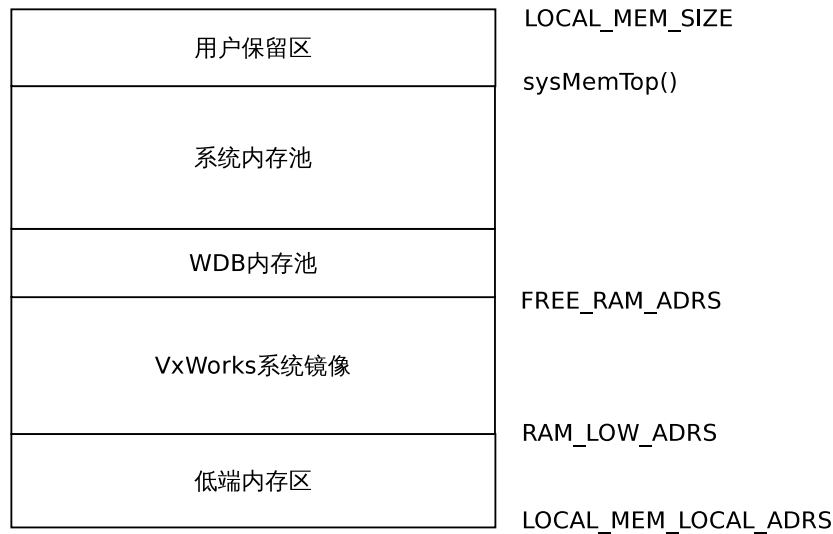


图 3.2 x86 下的 VxWorks 系统内存布局

3.3 VxWorks 代码插入方案设计

3.3.1 利用 nop 指令串插入

在2.2.1中我们曾指出，由于体积较大，VxWorks 系统镜像中存在较多的 nop 指令串可以用于插入代码。

然而手动进行插入少量代码方式虽然简单，但若对较大的文件注入大量的代码，手工操作的工作量会很大。文献 [10] 提供的 infelf 工具，可以用于在 Linux 下的 ELF 文件中自动寻找所有的 nop 指令串。我们在其基础上，完善了一个代码插入工具。该工具可以自动寻找 VxWorks 系统镜像中所有可以用于插入的 nop 指令串。该工具还可以以一个包含汇编代码的文本文件为输入，自动将代码汇编为二进制机器代码后，插入到 nop 指令串中。

该工具的完整代码参见附录A，工具的输入为 VxWorks 镜像文件和待插入的汇编代码文件。工具的主要算法如下：

- 1、反汇编 VxWorks 文件，找到所有的 ret 指令。
- 2、对于所有的 ret 指令之后的第一条指令，如果为 nop，则标记为 nop-start。
- 3、对于所有的 nop-start 之后的指令，第一条地址为 16 的倍数的指令，将其标记为 nop-end。
- 4、将输入的汇编代码文件汇编为二进制代码，从第一个 nop-start 处开始插入。
- 5、每插入一条指令，检测到 nop-end 的距离，如果小于 jmp 的长度或者下一条待插入指令的长度，则将上一条指令修改为 jmp，目的地址为下一个 nop-start。直到二进

制代码插入完成。

3.3.2 通过扩展.bss 节进行插入

前文中已经指出，.bss 节之后的地址用于 WDB 内存池，因此无法用于插入二进制代码。然而，我们可以通过修改 BSP 的办法，重新生成 bootloader。修改 bootloader 的目的是让它在加载镜像文件的过程中，为我们预留.bss 之后的一段地址。即把 WDB 内存池的起始地址略微后移，从而留出空闲的地址空间用于插入。

BSP 是 Board Support Package（板级支持包）的缩写，是 Wind River 公司推出的针对不同的硬件平台的一个程序。该程序针对不同的硬件进行抽象，为上层的 VxWorks 操作系统提供一致的接口，从而屏蔽了底层硬件的诸多细节，例如内存布局等等，在一定程度上方便了 VxWorks 驱动程序和应用程序的开发。BSP 和 VxWorks 的关系如图3.3所示。

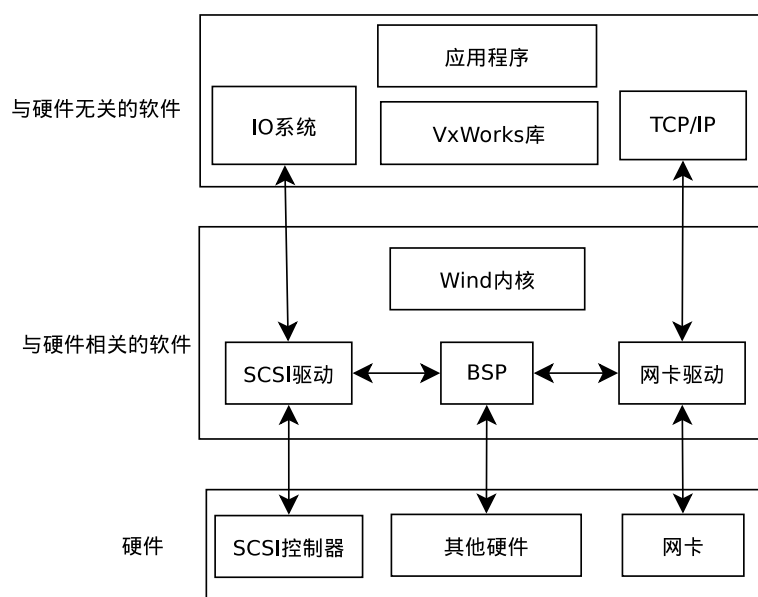


图 3.3 BSP 与 VxWorks 操作系统的关系

可见，BSP 直接或者间接地与硬件进行交互，并为上层的软件提供接口。BSP 对硬件的一个比较重要的抽象就是内存布局，3.2中的图3.2所描述的 VxWorks 内存布局，包括各个重要地址的位置，其实都是在 BSP 文件中定义的。

为了达到我们目标，即使得代码可以插入在.bss 节之后并不影响系统的正常运行。我们只需要修改 BSP 中的宏 FREE_RAM_ADDRESS 即可。例如，我们将该字段增大 0x1000。修改后，VxWorks 的内存布局将会如图3.4所示。

这样，我们就有了一定的内存空间用于插入代码。这时我们就可以使用附录A的插



图 3.4 修改 BSP 后的 VxWorks 系统内存布局

入工具对 VxWorks 镜像进行代码插入了。

这里的插入算法与5.6中有些许不同，具体的插入算法如下所示。

- 1、找到第一个（也是唯一一个）加载段的位置，将该段的 filez 字段和 memsz 字段增加要插入代码的大小。
- 2、将.bss 节的 len 字段增加插入代码长度大小。
- 3、对于在代码段之后的所有节区，将其 offset 字段增加一页（4KB）的大小。
- 4、将插入代码的长度填充到 1 页（4KB），然后插入到原.bss 节最后的位置。

4 代码劫持方案的分析与设计

上一章描述了针对 VxWorks 镜像的两种代码插入方式，本章将描述如何在代码插入的基础上，对目的函数进行劫持。从而当该函数被调用时，我们插入的代码将得到执行。插入代码执行完毕后，返回被劫持的函数，而不会影响整个系统的正常运行。

劫持的思路有两种。一种是劫持系统入口点，如图4.1所示。另一种是劫持某一函数，如图4.2所示。

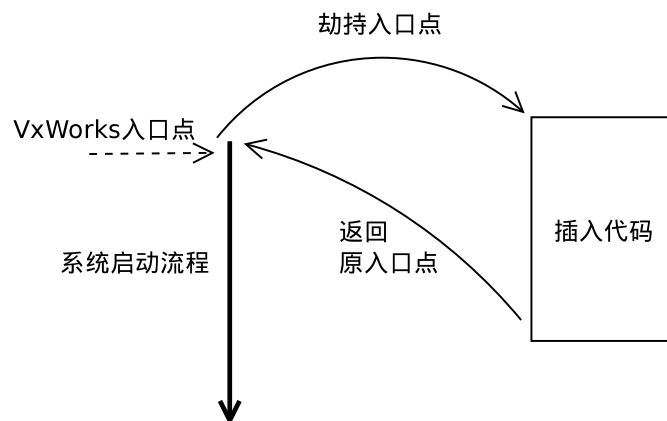


图 4.1 VxWorks 入口点劫持思路

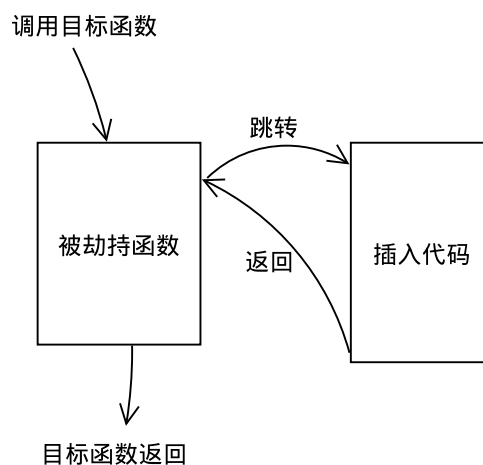


图 4.2 VxWorks 函数劫持思路

4.1 VxWorks 系统镜像种类

VxWorks 的系统映像分为三种类型，每一种映像类型都对应不同的加载方式。

- 可加载的映像类型

可引导型映像需要一段引导代码才能装载到 RAM 中，然后才能开始执行。

- 基于 ROM 的映像类型，

这类镜像首先把自己从 ROM 或者 Flash 中装载到 RAM 中，然后启动运行。

- ROM 驻留的映像类型

这类镜像与基于 ROM 的映像类型很相似，但是它在拷贝自身的时候只拷贝数据段，代码段仍然驻留在 ROM 中。适用于 RAM 较小的一些嵌入式系统，但运行速度会比较慢。

我们执行插入的对象是可加载的映像类型。这类映像本身其实是一个 ELF 文件，而不是平坦的二进制文件。其加载过程由一个加载程序（通常称为 **bootloader**）来完成。

4.2 VxWorks 系统启动调用流程

虽然我们的插入工作仅针对 ELF 文件来进行，并不修改 **bootloader** 的工作方式。但了解 **bootloader** 程序的工作方式是有必要的。就像在 Linux 下想要劫持 ELF 文件调用的共享库，需要熟悉加载器和动态链接器如何工作一样。

如图4.3所示，VxWorks 可加载型镜像的启动主要有三个步骤（不考虑 **bootloader** 被压缩的情况）：

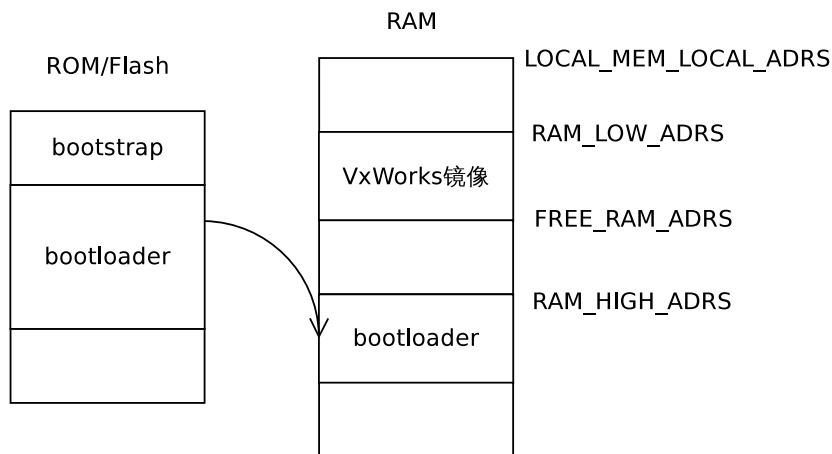


图 4.3 可加载型 VxWorks 镜像的启动

1、系统上电后，执行 **bootstrap** 的代码。该程序进行一些基本的硬件初始化，并负责将引导程序 **bootloader** 加载到 RAM 中的 **RAM_HIGH_ADRS** 处。

2、系统跳转到 RAM 中，执行 **bootloader** 的代码，该程序进行一些初始化工作之后，通过网络或其他途径下载可加载型 VxWorks 镜像，并加载到 RAM 中的 **RAM_LOW_ADRS** 处。

3、镜像加载完毕后，跳转到 RAM 中 VxWorks 的第一条指令处，开始进行内核初始化。完成后启动应用程序。

系统初始化开始后，会调用一系列的函数用于初始化。如图4.4所示。

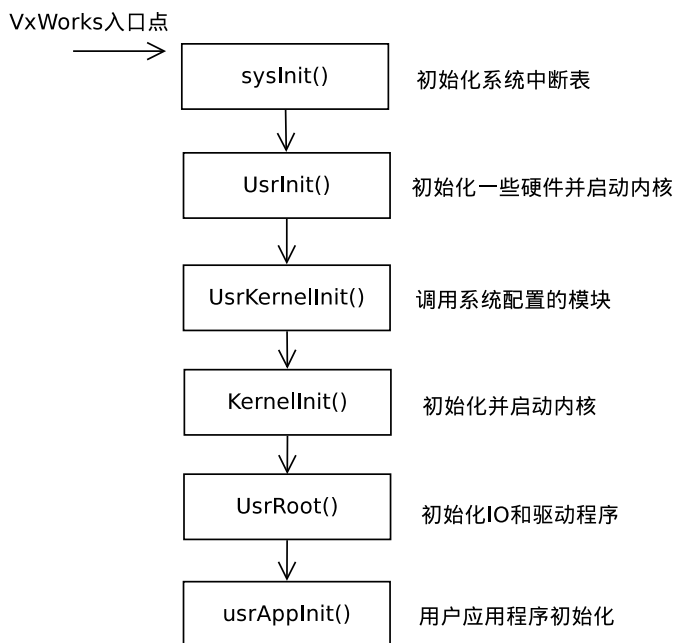


图 4.4 VxWorks 初始化函数调用流程

由图可知，从 VxWorks 入口点开始，进行了一些硬件方面的初始化工作。如果像在 Linux 下那样直接劫持入口点的话，我们的代码可能不能正常运行，例如不能正常使用堆栈。因此，函数的劫持应该有以下两个原则：

- 1、如果想要在系统启动后自动执行插入代码，应劫持 `usrAppInit` 函数
- 2、如果希望在某函数被调用执行被插入代码，应该在该函数头部插入跳转。

4.3 VxWorks 下代码劫持方案设计

经过前文论述，已经证明，在 VxWorks 下进行代码劫持其实只有一种情况，即劫持某一个函数的入口。本节在介绍的技术的基础上，提出两种在 VxWorks 下可行的函数劫持方案，分别是直接跳转和间接跳转。

4.3.1 直接跳转及其可行性证明

直接跳转类似2.3.2中介绍的跳转方案。我们发现，一个由 gcc 编译产生的 ELF 文件中，每个函数开头的几条指令和结尾的指令都是相似的。一般函数体都形如代码4.1所示。



```
1 push ebp
2 mov ebp, esp
3 sub esp, ??h      ;函数体开头的三条指令，保存 ebp，构建栈帧，分配空间。
4
5 ;函数体中间部分。
6
7 mov esp,ebp
8 pop ebp
9 ret               ;回收栈帧，取出 ebp，返回上一级函数。
```

代码 4.1 一般的 x86 函数体

函数体都分为三个部分。对于几乎每个函数来说，它们的开头的三条指令和结尾的三条指令（有时是两条，为 `leave` 和 `ret` 指令）几乎都是一样的。这无形中为劫持函数提供了一个方便之处。

我们使用一个例子来说明这一劫持方案。假设现有两个函数分别为 A 和 B，A 为被劫持的函数，B 为用于劫持 A 的插入函数。在劫持之前，他们的函数体如图4.5所示。

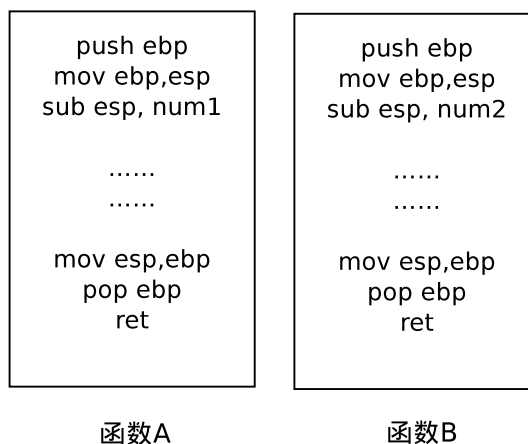


图 4.5 修改前的函数 A 和函数 B

利用直接跳转法进行修改，修改后的两个函数结果如图4.6所示。

劫持算法如下：

1、修改 A 函数的前三条指令（6 字节）为 `push+ret` 指令（6 字节），目的是跳转到 B 函数的第一条指令。

2、在 B 函数的 `ret` 指令前，添加两条指令：

第一条为 A 函数原来的第三条指令；

第二条为 `push` 指令，`push` 内容为 A 函数的第四条指令的地址。

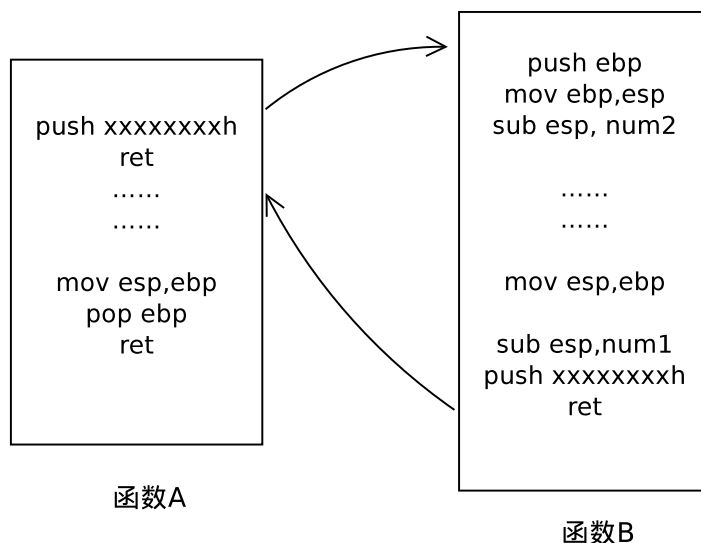


图 4.6 修改后的函数 A 和函数 B

下面从指令执行流和堆栈、寄存器恢复的角度，简要证明这种劫持方式的正确性：

1、控制流进入 A 函数时，还没有使用任何寄存器。因此在 B 中可以使用任意的调用者保存寄存器（eax 等）。而在 B 中编译生成的代码会保存被调用者保存寄存器（ebx 等）。故寄存器不会因为函数劫持而被污染。

2、B 中的倒数第四条指令，即 mov 指令，恢复了在 B 中发生变化的 esp 寄存器；而 B 中的开头两条指令和在倒数第三条添加的 sub 指令，刚好恢复了 A 中删掉的三条指令。故 A 的堆栈没有因为函数劫持而被污染。

3、A 中因为 push+ret 指令跳转到了 B 中，但最后又在 B 的最后跳回了 A 中。因此指令执行流最后也被复原。

综上所述，从函数 A 的角度看，堆栈、寄存器和指令执行流都没有因为函数的劫持而发生变化。因此 B 的代码相当于透明地插入到了 A 的开头部分。

4.3.2 间接跳转及其可行性证明

上一条中使用的劫持方法，由被劫持函数 A，直接跳转到劫持函数 B 中。为此付出的代价是仅仅是需要对 B 的结尾几条指令进行修改，以恢复 A 的栈帧。

然而这种方法仍然存在几种不足之处：

- 1、插入的位置仅限于函数体的开头。
- 2、需要小心的修改劫持函数的指令。如果需要插入大量函数，工作量较大。

本条提出一种新的劫持思路，即在使用一个中间层的思路，我们称这个中间层为 proxy。这需要我们插入两部分代码，一部分为 proxy，一部分为劫持函数。

proxy 劫持的原理非常简单。函数劫持无非就是要在不污染原函数堆栈和寄存器的情况下，改变指令的执行流。为了能够更从容地保存和恢复寄存器和堆栈，我们把这部分工作从劫持函数中分离出来，移到 proxy 中进行。

像4.3.1中的例子一样，同样有两个函数分别为 A 和 B，B 的任务是要劫持函数 A。利用 proxy 进行劫持的原理如图4.7所示。

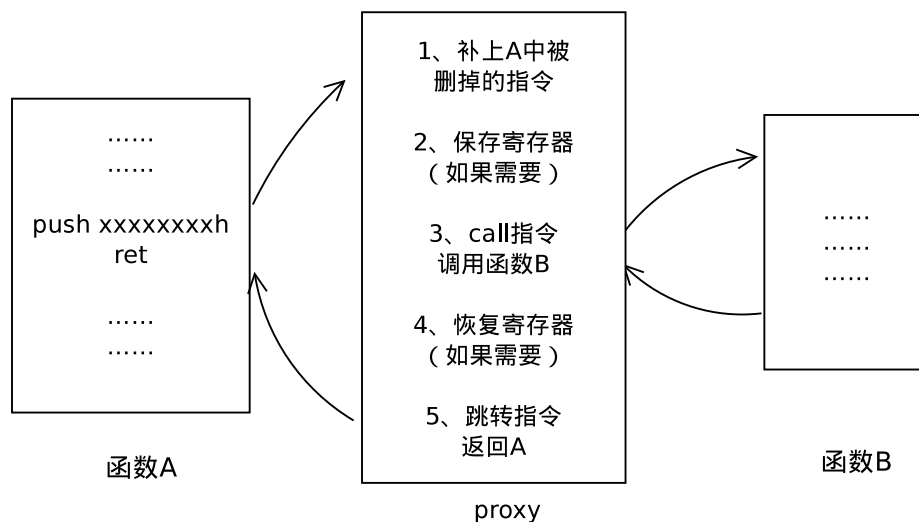


图 4.7 使用 proxy 劫持函数原理

在这一方法中，我们可以把 A 中的任意位置的进行劫持，只需要修改相应位置为 push+ret 指令即可（如果删掉的指令超过 6 字节，可用 nop 指令补齐）。

proxy 要进行一系列的任务。其中的核心任务是使用 call 指令调用函数 B。值得注意的是，由于对 A 的劫持可能来自任何一个地方，当劫持位置之前，有调用者保存寄存器被使用的话，B 函数有可能会污染 A 中的调用者保存寄存器，例如 eax。因此在 proxy 中需要按照需要进行寄存器的保存和恢复。

使用这种方法，我们很自然地拥有了比较多的回旋的余地。不必在狭小的 B 中进行寄存器保存的和恢复等工作。在实际工作中，我们在编写 B 这样的劫持函数的时候，就完全无需考虑任何有关劫持和插入的问题，而是自由地使用高级语言编写，然后像编译正常的程序一样编译即可。

4.3.3 两种方案的对比和分析

前文描述了两在 VxWorks 下进行函数劫持的思路。进行简单的对比和分析后，我们对两种方案的优劣总结如下：

1、直接跳转方案：



优点：插入代码量小。

缺点，逻辑比较复杂，劫持位置只能在函数体开头，对劫持函数的修改复杂。

2、间接跳转方案：

优点，逻辑清晰，劫持位置不受限制，`proxy` 通用性强；

缺点，插入代码量较大。

在接下来的实际的实现工作中，我们皆使用间接跳转方案。

5 实例：mini-notify——代码插入与函数劫持在 VxWorks 中的实现

5.1 背景

5.1.1 Linux 下的文件监控系统 inotify 介绍

Linux 2.6 内核中引入的 inotify，是一种监控文件系统变化的机制。它通过提供一定的接口，向用户空间报告底层文件系统发生的变化。用户空间通过系统调用和文件 IO 操作，来获取监视目标的变化情况。

5.1.2 VxWorks 下的 dosFs 文件系统

VxWorks 文件系统 dosFs 是 MS-DOS 兼容的文件系统，可基于块对物理介质进行操作。它提供极大的灵活性以满足实时应用的各种要求。在 VxWorks 操作系统中，文件系统的位置位于 IO 系统和驱动程序之间。它们之间的层次结构如图5.1所示。

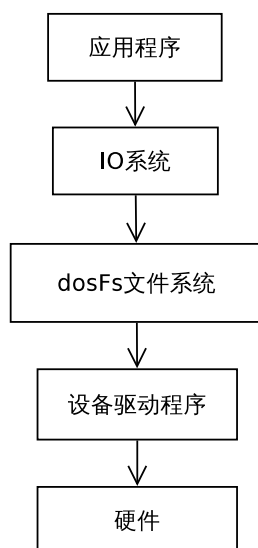


图 5.1 VxWorks 中的 IO 系统图示

具体到某个文件操作，例如 open 操作。应用程序可以调用库函数或者直接使用 IO 函数的时候，他们都会调用 dosFs 中的 dosFsOpen 函数，进而再调用驱动程序对应的函数。最后达到访问硬件的目的。整个流程如图5.2

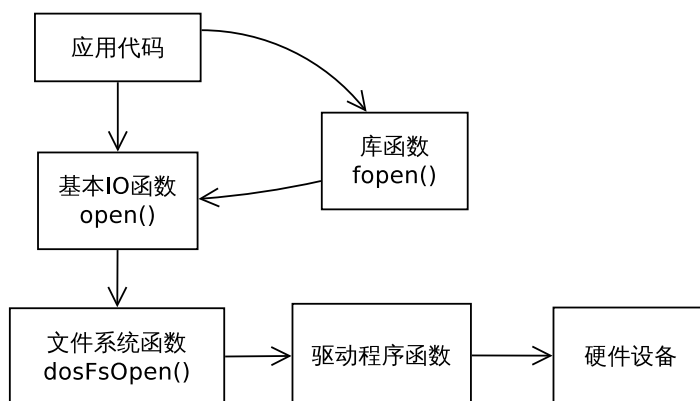


图 5.2 打开一个文件的函数调用流程

5.2 mini-notify 的目标

mini-notify 的目的是监控 VxWorks 下 dosFs 文件系统的变化，捕获所有针对该文件系统下的任何操作，例如对文件的读和写。并获取该操作的有关信息，并通过一定手段向“外界”传达该信息。

mini-notify 的原理如图5.3所示

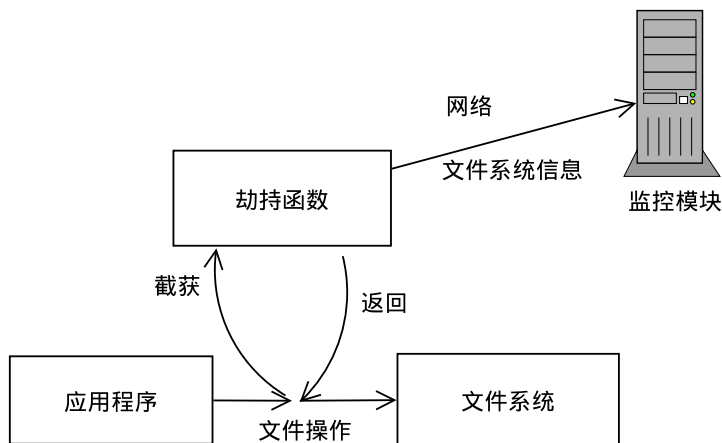


图 5.3 mini-notify 运行原理

其中，mini-notify 需要截获的文件系统操作包括：

- open
- close
- read
- write
- create
- delete

mini-notify 需要截获并向外界监控系统报告的信息有：



- 文件操作的名称（例如：read）
- 文件操作的具体路径（例如：/DOS/test）
- 系统的当前时间

5.3 环境搭建与配置

为了把精力集中于二进制层面的插入和劫持工作，而不是目标版的连接与调试，我们使用 VMware 虚拟机来运行一个 VxWorks 操作系统。这意味着我们只需要一台 x86 的 PC 就可以完成所有工作。然而实际上我使用了两个平台：

- 一台 x86 的 Linux 机器用于修改 VxWorks 系统映像。
- 一台 x86 的 Windows 机器用于运行 Tornado 和 VMware。

5.3.1 Linux 主机

Linux 主机的环境主要为我的修改工作提供便利的工具，特别是一些 GNU 实用工具，例如 readelf 和 objdump。

安装的 Linux 发行版为 Ubuntu 13.10 的 32 位版本，Linux 内核版本为 3.11。

5.3.2 Windows 主机与 VxWorks 虚拟机

Windows 主机的主要任务包含以下几点：

- 运行 Tornado 2.2，生成引导程序和 VxWorks 系统镜像。
- 运行 FTP Server，供目标机下载镜像。
- 运行 VMware 9，运行引导程序和 VxWorks 系统镜像。
- 运行 Target Server，用于与目标机通信。

图5.4说明了在 Windows 主机下各个软件虚拟机运行情况。

其中，在 VMware 中下载 VxWorks 镜像，需要首先载入 bootloader，实现的方式就不再赘述了。在此我们可以简单地认为 Tornado 生成了 VxWorks 镜像，而在虚拟机中即可通过 FTP 按照一定的路径找到该镜像并下载。因此，我们只需要修改该路径下的镜像文件，并替换掉原来的文件，当虚拟机重启后，就会下载我们修改过的镜像文件。

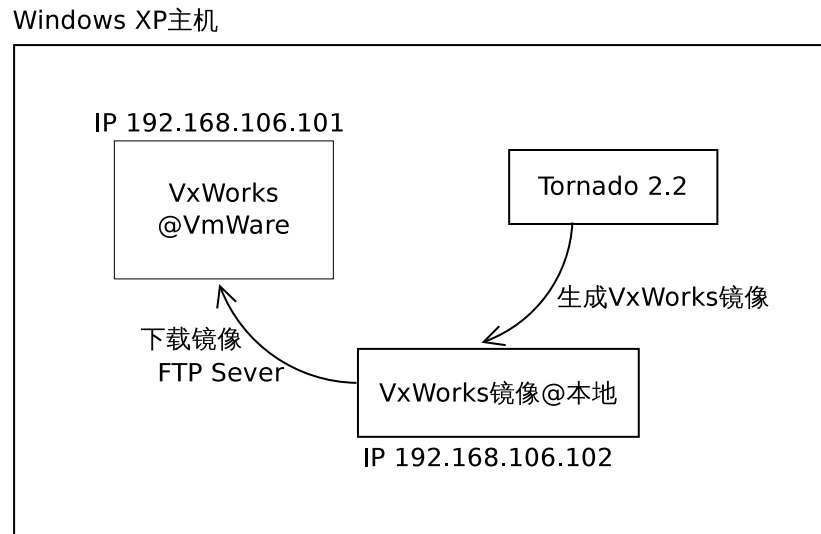


图 5.4 Windows 主机下的软件和虚拟机

5.4 mini-notify 方案设计

5.4.1 劫持位置选择

我们以劫持打开文件操作为例，阐述进行文件操作劫持的原理。首先我们要考察劫持的位置，即在图5.2所示的哪一层的相应函数中插入跳转，执行我们的插入代码。我们选择对文件系统函数进行劫持，即 `dosFsOpen()`。因为无论上层代码使用何种方式来进行文件操作，即无论使用哪一个库，都会最终调用到 `dosFsOpen()`，`dosFsOpen()` 函数的原型如代码5.1所示。

```
1 LOCAL DOS_FILE_DESC_ID dosFsOpen
2 (
3     DOS_VOLUME_DESC_ID pVolDesc, /* pointer to volume descriptor */
4     char *      pPath, /* dosFs full path/filename */
5     int      flags, /* file open flags */
6     int      mode /* file open permissions (mode) */
7 );
```

代码 5.1 `dosFsOpen()` 函数原型

可见，`dosFsOpen` 函数的传入参数包含了文件路径、文件打开的权限等信息。这也就意味着我们可以利用插入代码获取这些信息。而如果在更低层的驱动层来进行劫持，文件路径等信息就消失了，劫持工作很大程度上失去了意义。

5.4.2 函数劫持方案

为了不影响 `dosFsOpen` 的正常运行。插入代码需要保证不污染原先的堆栈和寄存器。我们继续使用中间层 `proxy` 作为劫持的中间代码。控制流首先从 `dosFsOpen` 转移到 `proxy`。用于劫持的功能的代码则被分离出来，单独写为一个函数，叫做 `hookFsOpen`。图5.5展示了劫持 `dosFsOpen` 函数的跳转过程。

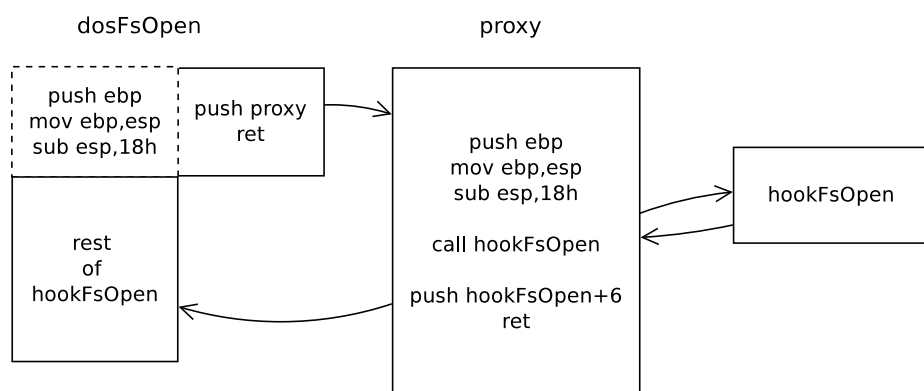


图 5.5 劫持 `dosFsOpen` 的函数跳转流程

我们将 `dosFsOpen` 函数开头的三条指令（位于虚线框中）替换为右边方框的指令。并在 `proxy` 开头补充这三条指令。这样一来，`proxy` 就很自然地拓展了 `dosFsOpen` 函数的空间，可以在其中写入任何代码，只需要在最后返回即可。我们在 `proxy` 中使用 `call` 指令调用 `hookFsOpen` 函数，一方面避免了在 `proxy` 中进行堆栈和寄存器的恢复；另一方面，我们 `hook` 函数不必真的像“插入”的函数那样小心翼翼，担心污染了寄存器和堆栈。总之，我们可以像编写正常的 C 语言函数一样，来编写 `hookFsOpen()`。

5.4.3 代码插入方案

在插入方案上，我们选择更简单的利用 `nop` 指令串的方案。在选择了插入方案和函数劫持方案的基础上，我们来考察如何能够更加方便地插入 `proxy` 和劫持函数。

附录A中针对 `VxWorks` 的代码插入工具，可以以汇编代码作为输入。我们可以将 `proxy` 和劫持函数写在一起，例如对于 `dosFsOpen` 函数来说，对应的插入代码如代码5.2所示。

```
1 global __start
2 section .text
3 __start:                ;proxy 部分
4     push ebp
5     mov ebp,esp
```



```
6      sub esp, 0x18          ; 补上 dosFsOpen 开头被替换的指令
7      call hookFsOpen        ; 调用 hookFsOpen, 可通过相对寻址定位
8      push dosFsOpen+6
9      ret                    ; 返回至 dosFsOpen 第四条指令处
10
11 hookFsOpen:
12      .....
13      .....                ; hookFsOpen 函数体
```

代码 5.2 为插入而准备的汇编代码

这一份汇编代码中, 同时包含了 proxy 和 hookFsOpen 函数。因此对于每一个被劫持的函数, 只需要使用一份汇编代码, 进行一次插入即可。此外, hook 函数体使用了汇编语言中的 label, proxy 对它的调用可以通过 label 来进行间接寻址, 从而为我们节省了手动重定位的工作。

5.5 mini-notify 的实现

完整的 mini-notify 包含针对 6 项基本文件操作 (create, delete, open, close, read 和 write) 的 6 个劫持函数。所有的劫持函数源代码见附录B。

劫持函数每次被调用, 主要进行以下两个方面的工作:

- 1、获取和解析 dosFs 函数的参数;
- 2、使用 ftp 向目标机器发送相应的信息, 包含操作名称, 文件路径 (pPath 参数) 和系统时间。

5.6 测试与评价

本节, 我们使用插入了 mini-notify 的系统镜像, 运行一个简单的测试用例, 测试用例代码见代码5.3。为了方便地获得 mini-notify 收集的信息, 我们将通过 ftp 传送改为直接在屏幕上打印。

```
1 void operate_file(){
2     int fd;
3     int nWritten,nRead;
4     char read_buf[20];
5     char *write_buf = "Hello dosFs!!!";
6
7     memset(read_buf,0,sizeof(read_buf));
```



```
8      fd = open("/DOSB/test1", O_CREAT | O_WRONLY, 0644);
9
10     nWritten = write(fd, write_buf, 15);
11     close(fd);
12     fd = open("/DOSB/test1", O_CREAT | O_RDONLY, 0644);
13     nRead = read(fd, read_buf, 20);
14
15     printf("fd:%d,write:%d,read:%d,contains:%s\n",fd, nWritten,nRead,read_buf);
16 }
```

代码 5.3 operate_file() 函数源代码

运行这一测试用例的结果如图5.6所示。

```
-> oprerate_file
Open    /DOSB/test1    FRI MAY 23 18:37:27 2014

Write   /DOSB/test1    FRI MAY 23 18:37:27 2014

Close   /DOSB/test1    FRI MAY 23 18:37:27 2014

Open    /DOSB/test1    FRI MAY 23 18:37:27 2014

Read    /DOSB/test1    FRI MAY 23 18:37:27 2014

Close   /DOSB/test1    FRI MAY 23 18:37:27 2014

fd:4,write:15,read:15,contains:Hello dosFs!!!
value = 0 = 0x0
-> _
```

图 5.6 测试程序运行结果

从该函数的输出中可以看出，测试用例中每次调用文件操纵函数，屏幕中都会打印一行输出，输出的信息包含三个部分，分别是操作的内容，目标文件路径和系统时间。

该测试用例证明 mini-notify 的基本功能运作正常。然而作为一个文件监控系统而言，mini-notify 仍有很多不足之处，下面列举出几条：

- 1、无法为应用程序提供接口，来由应用何时获取何种信息。而是强制地捕获文件系统信息。
- 2、获取的信息量有限，无法例如无法获知文件读写的内容等。
- 3、对于段时间内频繁的文件读写操作，mini-notify 只会简单地报告信息，而不会根据时间进行合并和整理。



结论

- 取得的成果

经过分析和实验证明,通过在 `nop` 指令串中写入代码,可以将代码插入到 VxWorks 二进制镜像文件中。劫持方法上,使用间接跳转法进行劫持,即利用一段插入代码 `proxy` 作为中间代码。可以简化劫持流程;另一方面,由于无需修改插入函数,也很大程度上方便了插入函数的编写。

利用选定的插入和劫持方案,在 VxWorks 中实现了一个类似 Linux 下 `inotify` 的文件监控机制——`mini-notify`。该机制能够捕获系统中所有的文件操作,并输出操作的文件路径等信息。从而证明了插入与劫持技术在 VxWorks 下的可行性。

- 尚存的不足

相对于在 Unix/Linux 下丰富的 ELF 插入技术,本文经分析认为只有利用 `nop` 串插入在 VxWorks 中是可行的。没有找到其他合适的代码插入方式。

VxWorks 系统已经实现了动态插入一个模块的功能,类似于 Linux 下的动态链接程序,本文缺少对这一方面的分析和研究。



参考文献

- [1] Committe T. Tool Interface Standard(TIS) Executable and Linking Formate(ELF) Specification[M]. 1.2.[S.l.]: [s.n.] , 1995.
- [2] Kaspersky K. 黑客反汇编揭秘 [M]. 第二版.[S.l.]: 电子工业出版社, 2010: 216–234.
- [3] Cesare S. Unix ELF parasites and virus[M].[S.l.]: [s.n.] , 1998. <http://vxheavens.com/lib/vsc01.html>.
- [4] Simple ELF Parasitic UNIX Virus[M].[S.l.]: [s.n.] . <http://academicunderground.org/virus/pnlVirus4.html>.
- [5] Dirk Gerrits R. G., Kooijmans P. An ELF virus prototype[A].[S.l.]: 2WC06 Hackers Hut 2005/2006, 2007.
- [6] grugq. Subversive Dynamic Linking to Libraries[M].[S.l.]: [s.n.] .
- [7] mayhem. The Cerberus ELF Interface[J]. phrack, 2003.
- [8] Cesare S. Shared Library Call Redirection via ELF PLT Infection[J]. phrack, 2000.
- [9] team E. .[S.l.]: [s.n.] . <http://www.eresi-project.org/wiki/TheELFsh>.
- [10] Z0mbie. Injected Evil (executable files infection)[M].[S.l.]: [s.n.] , 2004. <http://vxheavens.com/lib/vzo08.html>.
- [11] Ang Cui M. C., Stolfo S. J. When Firmware Modifications Attack: A Case Study of Embedded Exploitation[R].[S.l.]: Columbia University, 2012.
- [12] O'Neill R. Modern Day ELF Runtime infection via GOT poisoning[M].[S.l.]: [s.n.] , 2009. <http://vxheavens.com/lib/vrn00.html>.
- [13] Zaddach J. Implementation and Implications of a Stealth Hard-Drive Backdoor[J]. AC-SAC '13, 2013.
- [14] Chen K. Reversing and exploiting an Apple firmware update[J].
- [15] Inc. W. R. <http://www.windriver.com/products/vxworks.html>. Wind River VxWorks.



附录 A ELF 注入工具源代码

1、elfpatch.py 源代码

```
1 import struct
2 import os
3 import sys
4 import shutil
5 from elf_helper import *
6 from elfbin import elfbin
7
8 class elfpatch():
9     def __init__(self, FILE, OUTPUT, SHELLCODE, OBJECT_FILE):
10         self.FILE = FILE
11         self.OUTPUT = OUTPUT
12         self.bin_file = open(self.FILE, "r+b")
13         self.SHELLCODE = SHELLCODE
14         self.OBJECT_FILE = OBJECT_FILE
15         self.elfbin = elfbin(self.FILE)
16
17     def patch_elf(self):
18         shutil.copy2(self.FILE, self.OUTPUT)
19         print "[*] Patching Binary"
20         self.bin_file = open(self.OUTPUT, "r+b")
21
22         if not self.OBJECT_FILE:
23             shellcode = to_binary_code(self.SHELLCODE)
24         else:
25             shellcode = to_binary_code(get_text_section(self.OBJECT_FILE))
26         newBuffer = len(shellcode)
27         self.bin_file.seek(24, 0)
28
29         sh_addr = 0x0
30         offsetHold = 0x0
31         sizeOfSegment = 0x0
```



```
32     shellcode_vaddr = 0x0
33     headerTracker = 0x0
34     PAGE_SIZE = 4096
35     for header, values in self.elfbin.prog_hdr.iteritems():
36         if values['p_type'] == 0x1:
37             print "[+] Found text segment"
38             shellcode_vaddr = values['p_vaddr'] + values['p_filesz']
39             oldentry = self.elfbin.e_entry
40             headerTracker = header
41             newOffset = values['p_offset'] + values['p_filesz']
42             break
43     shellcode_vaddr = 0x00308000 + 0xce186
44     newOffset = 0x000060 + 0xce186
45     self.bin_file.seek(0)
46     file_1st_part = self.bin_file.read(newOffset)
47     newSectionOffset = self.bin_file.tell()
48     file_2nd_part = self.bin_file.read()
49
50     self.bin_file.close()
51     self.bin_file = open(self.OUTPUT, "w+b")
52     self.bin_file.write(file_1st_part)
53     self.bin_file.write(shellcode)
54     self.bin_file.write("\x00" * (PAGE_SIZE - len(shellcode)))
55     self.bin_file.write(file_2nd_part)
56     if self.elfbin.EI_CLASS == 0x01:
57         self.bin_file.seek(24, 0)
58         self.bin_file.seek(8, 1)
59         self.bin_file.write(struct.pack(self.elfbin.endian + "I", self.elfbin.
60             e_shoff + PAGE_SIZE))
61         self.bin_file.seek(self.elfbin.e_shoff + PAGE_SIZE, 0)
62         for i in range(self.elfbin.e_shnum):
63             if self.elfbin.sec_hdr[i]['sh_offset'] >= newOffset:
64                 self.bin_file.seek(16, 1)
65                 self.bin_file.write(struct.pack(self.elfbin.endian + "I", self
66                     .elfbin.sec_hdr[i]['sh_offset'] + PAGE_SIZE))
67                 self.bin_file.seek(20, 1)
```




```
66         elif self.elfbin.sec_hdr[i]['sh_size'] + self.elfbin.sec_hdr[i]['sh_addr'] == shellcode_vaddr:
67             self.bin_file.seek(20, 1)
68             self.bin_file.write(struct.pack(self.elfbin.endian + "I", self
        .elfbin.sec_hdr[i]['sh_size'] + newBuffer))
69             self.bin_file.seek(16, 1)
70         else:
71             self.bin_file.seek(40,1)
72         after_textSegment = False
73         self.bin_file.seek(self.elfbin.e_phoff,0)
74         for i in range(self.elfbin.e_phnum):
75             if i == headerTracker:
76                 after_textSegment = True
77                 self.bin_file.seek(16, 1)
78                 self.bin_file.write(struct.pack(self.elfbin.endian + "I", self
        .elfbin.prog_hdr[i]['p_filesz'] + newBuffer))
79                 self.bin_file.write(struct.pack(self.elfbin.endian + "I", self
        .elfbin.prog_hdr[i]['p_memsz'] + newBuffer))
80                 self.bin_file.seek(8, 1)
81             elif after_textSegment is True:
82                 self.bin_file.seek(4, 1)
83                 self.bin_file.write(struct.pack(self.elfbin.endian + "I", self
        .elfbin.prog_hdr[i]['p_offset'] + PAGE_SIZE))
84                 self.bin_file.seek(24, 1)
85             else:
86                 self.bin_file.seek(32,1)
87         self.bin_file.close()
88         print "[!] Patching Complete"
89         return (oldentry,shellcode_vaddr)
```



附录 B 实现文件监控功能的源代码

```
1 #include "stdio.h"
2 #include "hookFs.h"
3 #include "ioLib.h"
4 #include "private/dosFsLibP.h"
5 #include "vxWorks.h"
6 #include "Assert.h"
7 #include "hostLib.h"
8 #include "netDrv.h"
9
10 UINT32 read_ebp()
11 {
12     UINT32 __ebp;
13     __asm__ volatile("movl %%ebp, %0" : "=r"(__ebp));
14     return __ebp;
15 }
16
17 time_t biostime()
18 {
19     struct tm ahora;
20     unsigned char cHour, cMin, cSec;
21     unsigned char cDay, cMonth, cYear;
22
23     sysOutByte(0x70, 0x00 /*second*/);
24     cSec = sysInByte(0x71);
25     ahora.tm_sec = (cSec & 0x0F) + 10 * ((cSec & 0xF0) >> 4);
26
27     sysOutByte(0x70, 0x02 /*minut*/);
28     cMin = sysInByte(0x71);
29     ahora.tm_min = (cMin & 0x0F) + 10 * ((cMin & 0xF0) >> 4);
30
31     sysOutByte(0x70, 0x04 /*hour*/);
32     cHour = sysInByte(0x71);
33     ahora.tm_hour = (cHour & 0x0F) + 10 * ((cHour & 0xF0) >> 4);
```



```
34
35 sysOutByte(0x70,0x07/*day*/);
36 cDay = sysInByte(0x71);
37 ahora.tm_mday = (cDay&0x0F) + 10*((cDay&0xF0)>>4);
38
39 sysOutByte(0x70,0x08/*month*/);
40 cMonth = sysInByte(0x71);
41 ahora.tm_mon = (cMonth&0x0F) + 10*((cMonth&0xF0)>>4) - 1;
42
43 sysOutByte(0x70,0x09/*year*/);
44 cYear = sysInByte(0x71);
45 ahora.tm_year = 100 + (cYear&0x0F) + 10*((cYear&0xF0)>>4);
46
47 return mktime(&ahora);
48 }
49
50 void get_time(char datetime[])
51 {
52     int res;
53     struct timespec ts;
54     struct tm daytime;
55     time_t stime;
56
57     ts.tv_sec = biostime();
58     ts.tv_nsec = 0;
59     res = clock_settime(CLOCK_REALTIME, &ts);
60
61     stime = time(NULL);
62
63     daytime = *localtime(&stime);
64     strcpy(datetime, asctime(&daytime));
65 }
66
67 #define FUNC_HOOK_FS(name) \
68 void hookFs##name() \
69 {
```



```
70  UINT32 *_ebp; \
71  UINT32 arg; \
72  /*UINT32 arg0, arg1, arg2, arg3, arg4; */\
73  char nameBuf[MAX_PNAME_LEN]; \
74  PathNode *pn; \
75  DOS_FILE_DESC_ID pfd; \
76  if (!pHead) \
77  { \
78      printf("NOTIFY NOT INIT.\n"); \
79      return; \
80  } \
81  pn = pHead->next; \
82  if (!pn) \
83      return; \
84  memset(nameBuf, 0, sizeof(nameBuf)); \
85  __ebp = (UINT32 *) read__ebp(); \
86  __ebp = __ebp[0]; \
87  /* arg0 = __ebp[2];\
88  arg1 = __ebp[3];\
89  arg2 = __ebp[4];\
90  arg3 = __ebp[5];\
91  arg4 = __ebp[6];*/\
92  if ( !strcmp(#name, "Open") || !strcmp(#name, "Create") ) { \
93      arg = __ebp[4]; \
94      strcpy(nameBuf, arg); \
95  } \
96  else if (!strcmp(#name, "Delete")) \
97  { \
98      __ebp = (UINT32 *) read__ebp(); \
99      arg = __ebp[2]; \
100      strcpy(nameBuf, arg); \
101  } \
102  else if ( !strcmp(#name, "Close") ) \
103  { \
104      arg = __ebp[2]; \
105      pfd = (DOS_FILE_DESC_ID)arg; \
```



```
106     dosChkBuildPath(pfd); \
107     strcpy(nameBuf, pfd->pVolDesc->pChkDesc->chkPath); \
108 } \
109 else { \
110     /* Get the pFd */ \
111     __ebp = (UINT32 *) read__ebp(); \
112     /*arg0 = __ebp[2];\
113     arg1 = __ebp[3];\
114     arg2 = __ebp[4];\
115     arg3 = __ebp[5];\
116     arg4 = __ebp[6];*/\
117     arg = __ebp[2]; \
118     pfd = (DOS_FILE_DESC_ID)arg; \
119     dosChkBuildPath(pfd); \
120     strcpy(nameBuf, pfd->pVolDesc->pChkDesc->chkPath); \
121 } \
122 while(pn) \
123 { \
124     if (!strcmp(nameBuf, pn->pName) || !pathcmp(pn->pName, nameBuf)) \
125     { \
126         char datetime[64]; \
127         char result [MAX_PNAME_LEN + 80]; \
128         memset(datetime, 0, sizeof(datetime)); \
129         get__time(datetime); \
130         memset(result, 0, sizeof(result)); \
131         strcpy(result, #name); \
132         strcat(result, "\\t"); \
133         strcat(result, nameBuf); \
134         strcat(result, "\\t"); \
135         strcat(result, datetime); \
136         strcat(result, "\\r\\n\\0"); \
137         printf("%s\\n", result); \
138         break; \
139     } \
140     pn = pn->next; \
141 }
```



```
142 }  
143  
144 FUNC_HOOK_FS(Open);  
145 FUNC_HOOK_FS(Create);  
146 FUNC_HOOK_FS(Read);  
147 FUNC_HOOK_FS(Write);  
148 FUNC_HOOK_FS(Close);  
149 FUNC_HOOK_FS>Delete);
```