

Linux 网络入侵检测系统

刘文涛 编著

電子工業出版社

Publishing House of Electronics Industry

北京 · BEIJING

内 容 简 介

本书在介绍入侵检测系统的基本概念和原理的基础上,通过在 Linux 下设计一个典型的基于网络的入侵检测系统来更深入地探讨入侵检测技术。本书的一大特色是原理概念的讲述和系统的设计相辅相成,紧密联系。典型系统采用模块化设计思想,分别是网络数据包捕获模块、网络协议分析模块、存储模块、规则解析模块、入侵检测模块、响应模块和界面管理模块七个模块。另外,本书还深入讨论了网络数据包捕获技术、协议分析技术、入侵检测技术、入侵事件描述语言的建立、存储技术、多线程技术、界面设计技术等。

本书适合于计算机专业的本科生和研究生阅读,也可供从事计算机工程与应用的科技工作者或网络安全爱好者参考。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Linux 网络入侵检测系统/刘文涛编著. —北京:电子工业出版社,2004.10
ISBN 7-121-00477-1

.L... . 刘... .Linux 操作系统—安全技术 .TP316.89

中国版本图书馆 CIP 数据核字(2004)第 108223 号

责任编辑:竺南直

印 刷:

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

经 销:各地新华书店

开 本:787×1092 1/16 印张:18 字数:458 千字

印 次:2004 年 10 月第 1 次印刷

印 数: 册 定价: 元

略

凡购买电子工业出版社的图书,如有缺损问题,请向购买书店调换。若书店售缺,请与本社发行部联系。联系电话:(010) 68279077。质量投诉请发邮件至 zlts@phei.com.cn,盗版侵权举报请发邮件至 dbqq@phei.com.cn。

3000 4000 5000 6000 25.00 26.00 27.00 28.00 29.00

前 言

随着网络技术的飞速发展，网络安全问题也越来越突出。顺应这一趋势，涌现出了很多的网络安全技术，如网络防火墙、病毒检测、密码技术、身份认证等。但是，即使如此，还是有很多的服务器在不能够及时检测和预防的情况下被攻击，导致了巨大的经济损失。这种被动的防御系统显露出技术上很多的不足，于是，有人提出了主动的网络安全防御体系，这其中的代表者就是网络入侵检测系统。

网络入侵检测系统的根本功能就是实时地检测并分析网络的行为，根据分析的结果做出相应的响应。这样就可以及时地检测出网络的非法攻击，提早做出判断，减少损失。网络入侵检测系统的出发点在于其主动性，它不是被动的防御。主动性是入侵检测系统的一大特色，也是其发展迅速并被人们重视的一个重要原因。

本书介绍了网络入侵检测系统的基本概念和原理，并设计了一个简单又典型的网络入侵检测系统，通过设计与实现本系统可以使读者更好地理解网络入侵检测的相关概念。读者还可以在理解此系统的基础上设计功能更强大的网络入侵检测系统。系统的设计和实现与入侵检测系统相关概念的讲述是紧密联系、相辅相成的。作者在讲述一个概念的同时，将其与设计联系起来，这样有助于更加深入地理解概念的本质。本书中使用的是最新的基于协议分析的网络入侵检测技术，协议分析技术的实现是本书的一大重点。这个典型的入侵检测系统采用模块化设计，包括七个模块，分别为网络数据包捕获模块、网络协议分析模块、存储模块、规则解析模块、入侵事件检测模块、响应模块和界面管理模块。每个模块的设计与实现将分别放在不同的章节里进行详细的讨论。作者将此系统作为一个开发项目，其链接地址为 <http://tcpids.cosoft.org.cn>。

本书的主要特点

(1) 注重基础，重点突出。本书只有一个中心点，就是网络入侵检测系统，本书围绕这个中心展开论述，以便读者对网络入侵检测系统有一个最清楚的认识。包括入侵检测系统的各个方面，如其发展的原因、定义、入侵原理、入侵检测系统类型、研究发展方向、体系结构等。而本书的重点，就是理解一个完整的入侵检测系统的运行机理，包括它的各个部分，这是通过一个非常典型的系统来进行阐述的。

(2) 通过实例进行讲解。本书实现了一个完整的网络入侵检测系统，从最基本的部分开始设计，而这些组成部分几乎是每个大型网络入侵检测系统所具有的。作者深深感觉到，如果想很好地理解网络入侵检测系统的内部运行机理，通过编写实例来理解，是再好不过的，但是这个实例又不能太复杂，它实现了系统的最核心的功能。例如，学习操作系统，如果有一个小的完整的操作系统来进行学习，而且又可以进行裁减，这样学习起来就会理解得更深刻。本书设计的一个入侵检测系统就是为教学和研究所用，它实现了最核心的功能。读者可以根据这个小系统来进行更深入的开发和研究，特别是对于搞入侵检测系统方面开发的人员，有很好的借鉴作用。读者都有这样的体会，讲解一个问题（例如算法），不管你怎样描述，还不如给你一个 DEMO 让你来看，一下子就明白了，并且可以在这个 DEMO 的基础上设计自

己的系统。

(3) 程序讲解透彻。通过程序可以很好地描述一个算法和思想,因为程序是无国界的。本书给出了一些最关键的实现程序,并且对程序进行了详细的注解,读者不用担心看不懂程序。通过读本书的程序,可以更好地理解其实现的过程。本书的程序都是模块化的,结构层次都非常清楚。

本书的主要内容

首先我们介绍网络安全问题,讨论了传统的网络安全技术,提出了一种网络安全模型 PPDR。然后介绍了入侵检测系统的产生和定义。分析了两种类型的入侵检测系统:主机入侵检测系统和网络入侵检测系统。然后对入侵检测技术进行介绍,包括异常检测技术和误用检测技术。同时介绍入侵检测系统的标准化过程。

接下来介绍 Linux 下基于网络的入侵检测系统的设计原理和整体框架,分别介绍了各个模块的功能,讨论了数据源问题。

在网络数据包捕获模块中,本书详细讨论了数据包捕获机制。在此模块中也详细介绍了编写网络安全程序的函数库 LIBPCAP。LIBPCAP 是一个非常优秀的网络数据包捕获的封装函数库。

在协议分析模块中,本书对 TCP/IP 协议族进行了详细的分析,主要包括以太网协议,ARP/RARP,IP,UDP,TCP,ICMP,DNS,HTTP,DHCP。还对 IPX 协议也进行了分析。对 TCP 协议连接过程进行了详细的分析和实现。

在存储模块中,本书用 MySQL 数据库来存储网络信息,详细介绍了 MySQL 的使用方法,怎样用 PHPMYADMIN 来管理 MySQL 数据库,以及如何实现数据库的链接。介绍了怎样把网络信息存储到数据库中,还实现了对数据库的事后分析过程,实现了一些流量统计功能,着重分析了 HTTP 协议的相关内容,如 HTML 网页内容回放功能的实现。

在规则解析模块中,我们设计了一个入侵事件描述语言,入侵事件描述语言是 IDS 中一个比较重要的组成部分,在这个模块中详细介绍了入侵事件描述语言的设计和实现。对规则的解析过程进行了实现。

在入侵事件检测模块中,对基于协议的入侵检测方法进行了详细的介绍,实现了入侵规则匹配功能。还使用了非规则匹配的方式来检测入侵行为,如网络扫描行为的检测。在这个模块中,我们实现了入侵事件的检测功能。

在响应模块中,本书设计了几种响应方式,包括声音警报、灯光闪烁和被动记录等,还讨论了入侵响应原理、响应类型和响应方法。入侵响应是入侵检测系统中一个必要的组成部分。

在界面管理模块中,本书使用 GTK+ 技术来实现界面,介绍了 GTK+ 的相关内容,实现了界面管理模块的设计。在此模块中,还讨论了多线程技术,特别是对界面设计中的多线程技术进行了深入研究,例如界面的动态显示就使用了多线程技术。

由于本人知识有限,书中不免有错误之处,希望广大读者不吝赐教。

在此特别要感谢我的家人和朋友,他们的支持才使我有无穷的创作动力。

刘文涛
于武汉
2004 年 10 月

目 录

第 1 章	网络安全问题及其对策	(1)
1.1	网络安全问题	(1)
1.2	网络安全目标	(1)
1.3	网络面临的主要威胁	(2)
1.4	传统网络安全技术	(4)
1.5	网络安全模型——PPDR	(5)
第 2 章	入侵检测系统概述	(7)
2.1	入侵检测的产生及其定义	(7)
2.2	入侵检测系统的分类	(8)
2.3	入侵检测系统的标准化	(9)
2.3.1	入侵检测工作组 IDWG	(10)
2.3.2	公共入侵检测框架 CIDF	(10)
2.4	主要入侵检测系统介绍	(12)
第 3 章	入侵检测原理	(14)
3.1	入侵检测模型	(14)
3.1.1	IDES 模型	(14)
3.1.2	CIDF 模型	(15)
3.2	入侵检测技术	(15)
3.2.1	异常检测	(15)
3.2.2	误用检测	(16)
3.3	入侵检测的发展方向	(18)
第 4 章	Linux 网络入侵检测系统设计	(20)
4.1	系统设计原理	(20)
4.2	主要功能要求	(20)
4.3	检测器位置	(21)
4.4	数据源	(22)
4.5	系统总体结构	(24)
4.6	小结	(27)
第 5 章	网络数据包捕获模块设计与实现	(28)
5.1	Linux 内核中 TCP/IP 协议栈分析	(28)
5.2	BPF 机制	(30)
5.2.1	几种分组捕获机制介绍	(30)
5.2.2	BPF 过滤机制	(31)
5.3	使用 libpcap 函数库	(32)

5.3.1	主要函数介绍	(33)
5.3.2	编写步骤	(35)
5.3.3	bpf 过滤规则	(37)
5.4	实现数据包捕获模块	(41)
第 6 章	网络协议分析模块设计与实现	(46)
6.1	TCP/IP 协议分析基础	(46)
6.1.1	概述	(46)
6.1.2	IP 协议	(48)
6.1.3	TCP 协议	(49)
6.1.4	UDP 协议	(51)
6.1.5	ICMP 协议	(51)
6.2	协议分析模块的实现过程	(52)
6.2.1	协议分析过程	(52)
6.2.2	以太网协议分析	(63)
6.2.3	ARP 协议分析和 RARP 协议分析	(67)
6.2.4	IP 协议分析	(71)
6.2.5	TCP 协议分析	(79)
6.2.6	UDP 协议分析	(86)
6.2.7	ICMP 协议分析	(90)
6.3	其他协议的分析	(95)
6.3.1	DNS 协议	(95)
6.3.2	DHCP 协议	(103)
6.3.3	IPX/SPX 协议	(114)
6.4	使用 Libnids 库	(115)
6.4.1	Libnids 库简介	(115)
6.4.2	分析 TCP 连接过程	(121)
6.4.3	分析 HTTP 协议	(131)
第 7 章	存储模块设计与实现	(140)
7.1	设计原理	(140)
7.2	MySQL 数据库	(141)
7.2.1	安装 MySQL 数据库	(142)
7.2.2	基本操作	(142)
7.2.3	基本函数	(143)
7.3	存储模块实现	(147)
7.3.1	使用 PHPMYAdmin 管理数据库	(147)
7.3.2	设计数据库	(152)
7.3.3	实现数据库连接	(157)
7.4	数据库分析	(161)
7.4.1	分析 IP 数据包的分布状态	(162)

7.4.2	分析总体协议的分布状态	(170)
7.4.3	HTTP 流量分析	(177)
第 8 章	规则解析模块设计与实现	(181)
8.1	建立入侵事件描述语言	(181)
8.2	特征的选择	(182)
8.3	规则格式	(184)
8.4	规则选项	(187)
8.4.1	IP 协议变量	(187)
8.4.2	TCP 协议变量	(188)
8.4.3	UDP 协议变量	(189)
8.4.4	ICMP 协议变量	(190)
8.4.5	响应方式	(190)
8.5	规则解析模块实现	(191)
8.6	小结	(193)
第 9 章	入侵事件检测模块设计与实现	(194)
9.1	入侵检测方法	(194)
9.1.1	模式匹配方法的不足	(194)
9.1.2	使用协议分析方法	(196)
9.1.3	协议分析技术的优点	(197)
9.2	入侵事件检测模块实现	(198)
9.2.1	获取协议信息	(200)
9.2.2	规则匹配	(206)
9.2.3	检测扫描行为	(214)
9.3	小结	(218)
第 10 章	入侵响应模块设计与实现	(219)
10.1	响应的类型	(219)
10.2	入侵响应模块实现	(220)
10.2.1	采用声音警报的方式来响应	(220)
10.2.2	采用灯光闪烁的方式来发警报	(221)
10.2.3	使用日志来记录	(223)
第 11 章	界面模块设计与实现	(225)
11.1	GTK 概述	(225)
11.2	GTK 控件	(226)
11.3	使用 GTK	(229)
11.4	多线程技术	(231)
11.4.1	创建线程	(232)
11.4.2	结束线程	(233)
11.4.3	线程同步	(234)
11.4.4	GTKV+多线程	(235)

11.5 实现本系统界面模块	(237)
11.5.1 本系统界面分布情况	(237)
11.5.2 界面模块实现	(238)
11.6 小结	(272)
参考文献及进一步的读物	(273)

第 1 章 网络安全问题及其对策

1.1 网络安全问题

网络的飞速发展是有目共睹的，但是在这个飞速发展的过程中，也出现了种种不安全的因素。网络是一个双刃剑。网络给人类带来了无限好处，但同时也给人类带来了无限困扰。从各大媒体上大家都可以看到很多关于网络遭遇黑客攻击的报道，甚至把黑客的故事都搬上了银屏，这给现实生活中的人们产生了巨大的冲击。现在网络专家们对目前存在的安全隐患深感担忧。据报道，美国国防部已经花费了数十亿美元用以整治网络安全，但是网络安全问题还是不断涌现。现在全世界的公司都面临着网络被攻击的危险。在 2003 年，全球 13 台最重要的服务器中的 9 台都遭到了黑客袭击。

网络安全问题已经不再是一个新鲜的课题了，也已不再是一个高深的课题了，以前它基本上是跟网络人士打交道，但现在每一个人都可能与它打交道，只要他的电脑连在网络上。特别是随着网络应用范围的不断扩大，例如，政治、军事、文化、经济、教育、卫生、科技、公共服务等等都在普及网络，他们的网络安全问题也越来越突出，特别是在关键应用系统，如金融、电信、民航、电力等系统中。可以预见，随着网络的超常规的发展，网络安全问题会越来越严重，会越来越被人重视，当然网络安全技术也会越来越成熟。在这个过程中，必定会发生网络安全问题，这是不可避免的，但在遭遇网络安全问题的时候，不能够逃避，我们要力求找出应对网络安全问题的策略。

怎样解决网络安全问题，很多人在研究，也在探索。这其中出现了不少可喜的成果，但也有很多不足。随着网络的普及，网络攻击人员的技术也在不断提高，怎样应对这个现象，是一切志在解决网络安全问题的人士必须思考的一个问题。

本书讨论的是网络安全技术中的一个方面，即入侵检测系统。此技术的飞速发展，已经成为现在网络安全技术中的一个热门。本书只是起抛砖引玉的作用，希望能为网络安全的发展作出一点贡献。

1.2 网络安全目标

通俗地说，网络信息安全与保密主要是指保护网络信息系统，使其没有危险、不受威胁、不出事故。从技术角度来说，网络信息安全与保密的目标主要表现在系统的保密性、完整性、真实性、可靠性、可用性、不可抵赖性等方面。

1. 可靠性

可靠性是网络信息系统能够在规定条件下和规定时间内完成规定功能的特性。可靠性是系统安全的最基本要求之一，是所有网络信息系统的建设和运行目标。

2. 可用性

可用性是网络信息可被授权实体访问并按需求使用的特性。即网络信息服务在需要时，允许授权用户或实体使用的特性，或者是网络部分受损或需要降级使用时，仍能为授权用户提供有效服务的特性。可用性是网络信息系统面向用户的安全性能。

3. 保密性

保密性是指防止信息泄露给非授权个人或实体，信息只为授权用户使用的特性。保密性是在可靠性和可用性基础之上，保障网络信息安全的重要手段。

4. 完整性

完整性是网络信息未经授权不能进行改变的特性。即网络信息在存储或传输过程中保持不被偶然或蓄意地删除、修改、伪造、乱序、重放、插入等破坏和丢失的特性。完整性是一种面向信息的安全性，它要求保持信息的原样，即信息的正确生成、正确存储和正确传输。

5. 不可抵赖性

不可抵赖性也称做不可否认性，在网络信息系统的信息交互过程中，确信参与者的真实同一性。即所有参与者都不可能否认或抵赖曾经完成的操作和承诺。利用信息源证据可以防止发信方否认已发送信息，利用递交接收证据可以防止收信方事后否认已经接收的信息。

6. 可控性

可控性是对网络信息的传播及内容具有控制能力的特性。

概括地说，网络信息安全与保密的核心是通过计算机、网络、密码技术和安全技术，保护在公用网络信息系统中传输、交换和存储的消息的保密性、完整性、真实性、可靠性、可用性、不可抵赖性等。

1.3 网络面临的主要威胁

日益严重的网络信息安全问题，不仅使上网企业、机构以及用户蒙受巨大的经济损失，而且使国家的安全与主权面临严重威胁。要避免网络信息安全问题，首先必须搞清楚触发这一问题原因。总结起来，主要有以下几个方面原因。

1. 黑客的攻击

黑客对于大家来说，不再是一个高深莫测的人物，黑客技术逐渐被越来越多的人掌握和发展，目前，世界上有 20 多万个黑客网站，这些站点都介绍一些攻击方法和攻击软件的使用以及系统的一些漏洞，因而系统、站点遭受攻击的可能性就变大了，尤其是现在还缺乏针对网络犯罪很有成效的反击和跟踪手段，使得黑客攻击的隐蔽性好，破坏力大，是网络安全的主要威胁。

黑客的攻击方法有很多种，例如口令攻击，一种方法是通过网络监听非法得到用户口令，这类方法有一定的局限性，但危害性极大，监听者往往能够获得其所在网段的所有用户账号和口令，对局域网安全威胁巨大；二是在知道用户的账号后利用一些专门软件强行破解用户口令，这种方法不受网段限制，但黑客要有足够的耐心和时间；三是在获得一个服务器上的

用户口令文件后，用暴力破解程序破解用户口令，此方法在所有方法中危害最大；另外一个攻击方法就是放置特洛伊木马程序。特洛伊木马程序可以直接侵入用户的电脑并进行破坏，它常被伪装成工具程序或者游戏等诱使用户打开带有特洛伊木马程序的邮件附件或从网上直接下载，一旦用户打开了这些邮件的附件或者执行了这些程序之后，它们就会像古特洛伊人在敌人城外留下的藏满士兵的木马一样留在自己的电脑中，并在自己的计算机系统中隐藏一个可以在系统启动时悄悄执行的程序。当被植入特洛伊木马程序的机器连上网络时，木马程序就会把机器的信息发给控制端程序，这样控制端程序就会控制中木马的机器了。网络监听又是一个常用的攻击方法。再就是寻找系统的漏洞，其中某些是操作系统或应用软件本身具有的，如 Sendmail 漏洞，这些漏洞在补丁未被开发出来之前一般很难防御黑客的破坏。黑客的攻击方法多种多样，而且所用技术在不断创新。

2. 管理的疏忽

网络系统的管理是企业、机构及用户免受攻击的重要措施。事实上，很多企业、机构及用户的网络或系统都疏于这方面的管理。据 IT 界企业团体 ITAA 的调查显示，美国 90% 的 IT 企业对黑客攻击准备不足。目前，美国 75% ~ 85% 的网站都抓不住黑客的攻击，约有 75% 的企业网上信息失窃。

3. 网络的缺陷

Internet 的共享性和开放性使网上信息安全存在先天不足，因为其赖以生存的 TCP/IP 协议族缺乏相应的安全机制，而且因特网最初的设计考虑是该网络不会因局部故障而影响信息的传输，基本没有考虑安全问题，因此它在安全可靠、服务质量和带宽等方面存在着不适应性。

4. 软件的漏洞

随着软件系统规模的不断增大，系统中的安全漏洞或“后门”也不可避免地存在，比如我们常用的操作系统，无论是 Windows 还是 UNIX 几乎都存在或多或少的安全漏洞，众多的各类服务器、浏览器，一些桌面软件，等等，都被发现过存在安全隐患。可以说任何一个软件都可能会因为程序员的一个疏忽，设计中的一个缺陷等原因而存在漏洞，这也是网络安全的主要威胁之一。

由于漏洞的存在，黑客可以根据漏洞来进行攻击，病毒也可以根据漏洞来进行攻击。现在的病毒已经变得非常智能化，简直是无孔不入。这些病毒再加上黑客的技术就可能产生非常严重的后果。由于病毒传播的速度非常快，如果黑客把入侵代码加入病毒或者做成病毒，这样黑客的攻击范围就会变得非常大。病毒特征和黑客技术相结合的攻击方法会越来越普遍。现在的软件越来越复杂，漏洞就会越来越多，特别是黑客技术在不断提高，他们发现软件漏洞并利用漏洞来进行攻击的时间也会越来越短。

5. 网络内部攻击

网络内部用户的误操作、资源滥用和恶意行为。再完善的防火墙也无法抵御来自网络内部的攻击，也无法对网络内部的滥用做出反应。

1.4 传统网络安全技术

传统网络安全技术主要使用以下几种安全机制。

1. 加密机制

加密是一种最基本的安全机制，它能防止信息被非法读取。加密是一种在网络环境中对抗被动攻击的行之有效的安全机制。数据加密是保护数据的最基本的方法。但是，这种方法只能防止第三者获取真实数据，仅解决了安全问题的一个方面。而且，加密机制并不是牢不可破的。

2. 数据签名机制

签名与加密很相似，一般是签名者利用秘密密钥（私钥）对需签名的数据进行加密，验证利用签名者的公开密钥（公钥）对签名数据做解密运算。如果能保证一个签名者的签名只能惟一地从他自己产生，那么当收发双方发生争议的时候，仲裁机构就能够根据消息上的数字签名来裁定这条消息是否是由发送方发出的。

3. 访问控制机制

访问控制机制是按照事先确定的规则决定主体对客体的访问是否合法。当一个主体试图非法使用一个未经授权使用的客体（资源）时，访问控制功能将拒绝这一企图，并可附带报告这一事件给审计跟踪系统，审计跟踪系统产生一个报警或形成部分追踪审计。

4. 数据完整性机制

数据完整性机制可以发现网络上传输的数据已经被非法修改，从而使用户不会被非法数据欺骗。

5. 认证机制

认证是以交换信息的方式来确认实体身份的机制，是进行存取控制所必不可少的条件，因为不知道用户是谁，就无法判断其存取是否合法。

6. 系统脆弱性检测

系统中脆弱性的存在是系统受到各种安全威胁的根源。外部黑客的攻击主要利用了系统提供的网络服务中的脆弱性；内部人员作案则利用了系统内部服务及其配置上的脆弱性；而拒绝服务攻击主要利用资源分配上的脆弱性，长期占用有限资源不释放，使其他用户得不到应有的服务，或者是利用服务处理中的弱点，使该服务崩溃。

7. 防火墙

防火墙系统软件实现将定义好安全策略转换成具体的安全控制操作，它使得内部网络与因特网之间或者与其他外部网络互相隔离，限制网络互访。按照一定的安全策略规则对其检查网络包或服务请求，来决定网络之间的通信是否被允许，其中被保护的内部网络称为内部网络或私有网络，而与内部网络或私有网络相连的网络称为外部网络或公有网络。防火墙能有效地控制内部网络与外部网络之间的访问及数据传送，从而实现保护内部网络的信息不受外部

非授权用户的访问或者过滤信息的目的。防火墙的实现从层次上大概可以分两种：报文过滤和应用层网关。

1.5 网络安全模型——PPDR

PPDR (Policy Protection Detection Response) 的基本思想是：以安全策略为核心，通过一致性检查、流量统计、异常分析、模式匹配以及基于应用、目标、主机、网络的入侵检查等方法进行安全漏洞检测。检测使系统从静态防护转化为动态防护，为系统快速响应提供了依据。当发现系统有异常时，根据系统安全策略快速作出反应，从而达到保护系统安全的目的。PPDR 可适应网络安全模型如图 1-1 所示。

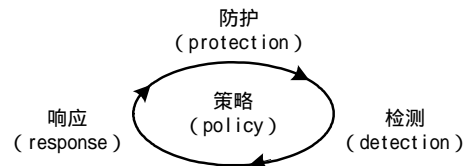


图 1-1 PPDR 网络安全模型

PPDR 模型由 4 个主要部分组成：安全策略 (Policy)、防护 (Protection)、检测 (Detection) 和响应 (Response)。PPDR 模型是在整体的安全策略的控制和指导下，在综合运用防护工具（如防火墙、操作系统身份认证、加密等手段）的同时，利用检测工具（如漏洞评估、入侵检测等系统）了解和评估系统的安全状态，通过适当的响应将系统调整到一个比较安全的状态。防护、检测和响应组成了一个完整的、动态的安全循环。

1. 策略

策略是这个模型的核心，在具体的实施过程中，策略意味着网络安全要达到的目标，它决定了各种措施的强度。因此追求安全是要付出代价的，一般会牺牲用户使用的舒适度，还有整个网络系统的运行性能，因此策略的制定要按照需要进行。

2. 防护

一般而言，防护是安全的第一步，它的基础是检测与响应的结果。具体包括：

安全规章的制定：在安全策略的基础上制定安全细则。

系统的安全配置：针对现有网络环境的系统配置，安装各种必要的补丁软件，并对系统进行仔细的配置，以达到安全策略规定的安全级别。

安全措施采用：安装防火墙软件和设备，VPN 软件和设备等。

3. 检测

采取了各种安全防护措施并不意味着网络系统的安全性就得到了安全的保障，网络的情况是动态变化的，而各种软件系统的漏洞层出不穷，都需要采取有效的手段对网络的运行进行监视。

防护相对于攻击者来说总是滞后的，一种漏洞的发现或者攻击手段的发明与相应的防护手段的采用之间，总会有一个时间差，检测就是弥补这个时间差的必要手段。

检测的作用包括：

- 异常监视：发现系统的异常情况如重要文件的修改，不正常的登录等；
- 模式发现：对已知的攻击模式进行发现。

4. 响应

在发现了攻击企图或者攻击之后，需要整个系统及时地作出反应，这包括：

报告：就是作出入侵事件响应的报告，通知安全管理员发生了入侵行为，可以利用各种各样的报告方式，如电子邮件、警报甚至手机短信，等等。

记录：当有入侵行为发生时，就把所有的系统活动都给记录下来，这样事后就可以进行进一步的分析，以找出系统的弱点和漏洞所在。

反应：反应可以是主动的，例如可以切断当前连接，也可以通过告诉防火墙更改控制策略，等等。其目的就是把损失控制到最低。

恢复：如果入侵行为成功，必然对系统造成一定的破坏，恢复就是修复系统，使系统能够正常运转。

第 2 章 入侵检测系统概述

2.1 入侵检测的产生及其定义

1. 入侵检测产生原因

传统的信息安全技术都集中在系统自身的加固和防护上。比如，采用 B 级操作系统和数据库，在网络出口配置防火墙，在信息传输和存储中采用加密技术，使用集中的身份认证产品等。

然而，单纯的防护技术有如下几方面的问题。

(1) 单纯的防护技术容易导致系统的盲目建设，这种盲目包括两方面：一方面是不了解安全威胁的严峻和当前的安全现状；另一方面是安全投入过大而又没有真正抓住安全的关键环节，导致不必要的浪费。

(2) 防火墙策略对于防范黑客有其明显的局限性。诸如：

防火墙难于防内。防火墙的安全控制只能作用于外对内或内对外，即：对外可屏蔽内部网的拓扑结构，封锁外部网上的用户连接内部网上的重要站点或某些端口，对内可屏蔽外部危险站点，但它很难解决内部网控制内部人员的安全问题。即防外不防内。而据权威部门统计结果表明，网络上的安全攻击事件有 70% 以上来自内部攻击。

防火墙难于管理和配置，易造成安全漏洞。防火墙的管理及配置相当复杂，要想成功地维护防火墙，要求防火墙管理员对网络安全攻击的手段及其与系统配置的关系有相当深刻的了解。防火墙的安全策略无法进行集中管理。一般来说，由多个系统（路由器、过滤器、代理服务器、网关、堡垒主机）组成的防火墙，管理上有所疏忽是在所难免的。根据美国财经杂志统计资料表明，30% 的入侵发生在有防火墙的情况下。

防火墙的安全控制主要是基于 IP 地址的，难于为用户在防火墙内外提供一致的安全策略。许多防火墙对用户的安全控制主要是基于用户所用机器的 IP 地址而不是用户身份，这样就很难为同一用户在防火墙内外提供一致的安全控制策略，限制了企业网的物理范围。

防火墙只实现了粗粒度的访问控制，且不能与企业内部使用的其他安全机制（如访问控制）集成使用，这样，企业就必须为内部的身份验证和访问控制管理维护单独的数据库。

(3) 保证信息系统安全的经典手段是“存取控制”或“访问控制”，这种手段在经典的以及现代的安全理论中都是实行系统安全策略的最重要的手段。但迄今为止，软件工程技术还不可能百分之百地保证任何一个系统（尤其是底层系统）中不存在安全漏洞。

而且，无论在理论上还是在实践中，试图彻底填补一个系统的安全漏洞都是不可能的，也还没有一种切实可行的办法解决合法用户在通过“身份鉴别”或“身份认证”后滥用特权的问题。

针对日益严重的网络安全问题和越来越突出的安全需求，“可适应网络安全模型”和“动态安全模型”应运而生。而入侵检测系统在动态安全模型中占有重要的地位。

入侵检测作为一种积极主动的安全防护技术，提供了对内部攻击、外部攻击和误操作的实时保护，在网络系统受到危害之前拦截和响应入侵。从网络安全立体纵深、多层次防御的角度出发，入侵检测理应受到人们的高度重视，这从国外入侵检测产品市场的蓬勃发展就可以看出。在国内，随着上网的关键部门、关键业务越来越多，迫切需要具有自主知识产权的入侵检测产品。但现状是入侵检测还不够成熟，处于发展阶段，或者是防火墙中集成较为初级的入侵检测模块，所以对于入侵检测系统的研究是很重要的。

2. 入侵检测的定义

入侵检测是指在特定的网络环境中发现和识别未经授权的或恶意的攻击和入侵，并对此作出反应的过程。而入侵检测系统 IDS 是一套运用入侵检测技术对计算机或网络资源进行实时检测的系统工具。IDS 一方面检测未经授权的对象对系统的入侵，另一方面还监视授权对象对系统资源的非法操作。

3. 入侵检测系统的作用

入侵检测系统的作用有以下几种：

- 监视、分析用户和系统的运行状况，查找非法用户和合法用户的越权操作；
- 检测系统配置的正确性和安全漏洞，并提示管理员修补漏洞；
- 对用户非正常活动的统计分析，发现攻击行为的规律；
- 检查系统程序和数据的一致性和正确性；
- 能够实时地对检测到的攻击行为进行响应；
- 对操作系统的审计跟踪管理，并识别用户违反安全策略的行为。

2.2 入侵检测系统的分类

入侵检测系统根据检测的对象可分为基于主机的入侵检测系统 HIDS (Host Intrusion Detection System) 和基于网络的入侵检测系统 NIDS (Network Intrusion Detection System)。

1. 主机入侵检测系统 HIDS

基于主机的入侵检测是根据主机系统的系统日志和审计记录来进行检测分析，通常在要受保护的主机上有专门的检测代理，通过对系统日志和审计记录不间断的监视和分析来发现攻击。

它的主要目的是在事件发生后提供足够的分析来阻止进一步的攻击。

其优点有：

能够监视特定的系统行为，基于主机的 IDS 能够监视所有的用户登录和退出甚至用户所做的所有操作，审计系统在日志里记录的策略改变，监视关键系统文件和可执行文件的改变等。

HIDS 能够确定攻击是否成功，由于使用含有已发生事件信息，它们可以比 NIDS 更加准确地判断攻击是否成功。

有些攻击在网络的数据流中很难发现，或者根本没有通过网络在本地进行。这时 NIDS 将无能为力，只能借助于 HIDS。

其缺点有：

HIDS 安装在我们需要保护的设备上，这会降低应用系统的效率。它依赖于服务器固有的日志与监视能力，如果服务器没有配置日志功能，则必须重新配置，这将会给运行中的业务系统带来不可预见的性能影响。

全面布署 HIDS 代价较大。

HIDS 除了监测自身的主机以外，根本不监测网络上的情况。

2. 网络入侵检测系统 NIDS

基于网络的入侵检测系统是使用原始网络数据包作为数据源。

其优点有：

有较低的成本。

能够检测到 HIDS 无法检测的入侵，例如 NIDS 能够检查数据包的头部而发现非法的攻击，NIDS 能够检测那些来自网络的攻击，它能够检测到超过授权的非法访问。

入侵对象不容易销毁证据，被截取的数据不仅包括入侵的方法，还包括可以定位入侵对象的信息。

能够做到实时检测和响应，一旦发现入侵行为就立即中止攻击。

NIDS 不依赖于被保护主机的操作系统。

其弱点有：

只检查它直接连接网段的通信，不能检测在不同网段的网络包。在使用交换以太网的环境中就会出现监测范围的局限，而安装多台网络入侵检测系统的传感器会使部署整个系统的成本大大增加。

网络入侵检测系统为了性能目标通常采用特征检测的方法，它可以检测出普通的一些攻击，而很难实现一些复杂的需要大量计算与分析时间的攻击检测。

网络入侵检测系统可能会将大量的数据传回分析系统中。在一些系统中监听特定的数据包会产生大量的分析数据流量。一些系统在实现时采用一定方法来减少回传的数据量，对入侵判断的决策由传感器实现，而中央控制台成为状态显示与通信中心，不再作为入侵行为分析器。这样的系统中的传感器协同工作能力较弱。

网络入侵检测系统处理加密的会话过程较困难，目前通过加密通道的攻击尚不多，但随着 IPv6 的普及，这个问题会越来越突出。

基于网络的入侵检测系统与基于主机的入侵检测系统都有各自的优点，应该互相取长补短，所以 IDS 必须把主机和网络两个部分紧密融合在一起，这样才能更好地进行检测。

2.3 入侵检测系统的标准化

为了提高 IDS 产品、组件及与其他安全产品之间的互操作性，美国国防高级研究计划署 (DARPA) 和互联网工程任务组 (IETF) 的入侵检测工作组 IDWG (Intrusion Detection Working Group) 发起制订了一系列建议草案，从体系结构、API、通信机制、语言格式等方面规范 IDS 的标准。

DARPA 提出的建议是公共入侵检测框架 CIDF (Common Intrusion Detection Framework)，

最早由加州大学戴维斯分校安全实验室主持起草工作。1999 年 6 月, IDWG 就入侵检测也出台了一系列草案。

2.3.1 入侵检测工作组 IDWG

入侵检测工作组 IDWG 的任务是: 定义数据格式和交换规程, 用于入侵检测与响应系统之间或与需要交互的管理系统之间的信息共享。IDWG 提出的建议草案包括三部分内容: 入侵检测消息交换格式 IDMEF (Intrusion Detection Message Exchange Format)、入侵检测交换协议 IDXP (Intrusion Detection Exchange Protocol) 以及隧道轮廓 (Tunnel Profile)。

(1) IDMEF

IDMEF 入侵检测消息交换格式描述了表示入侵检测系统输出信息的数据模型, 并解释了使用此模型的基本原理。该数据模型用可扩展标记语言 XML (Extensible Markup Language) 来实现, 并设计了一个 XML 文档类型定义。自动入侵检测系统可以使用 IDMEF 提供的标准数据格式对可疑事件发出警报, 提高商业、开放资源和研究系统之间的互操作性。IDMEF 主要适用于入侵检测分析器和接收警报的管理器之间的数据信道。

(2) IDXP

IDXP 入侵检测交换协议是一个用于入侵检测实体之间交换数据的应用层协议, 能够实现 IDMEF 消息、非结构文本和二进制数据之间的交换, 并提供面向连接协议之上的双方认证、完整性和保密性等安全特征。IDXP 是 BEEP 的一部分, 后者是一个用于面向连接的异步交互通用应用协议, IDXP 的许多特色功能 (如认证、保密性等) 都是由 BEEP 框架提供的。

2.3.2 公共入侵检测框架 CIDF

CIDF 所做的工作主要包括四部分: IDS 的体系结构、通信机制、描述语言和应用编程接口 API。

1. CIDF 的体系结构

CIDF 在 IDES 和 NIDES 的基础上提出了一个通用模型, 将入侵检测系统分为四个基本组件: 事件产生器、事件分析器、响应单元和事件数据库。结构如图 2-1 所示。

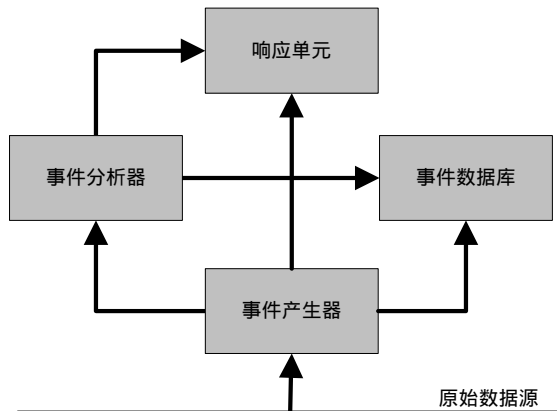


图 2-1 入侵检测系统 CIDF 模型

CIDF 将入侵检测系统需要分析的数据统称为事件 (event), 它可以是基于网络的入侵检测系统中网络中的数据, 也可以是从系统日志或其他途径得到的信息。

这四个组件只是逻辑实体, 例如, 一个组件可能是某台计算机上的一个进程或线程, 也可能是多个计算机上的多个进程, 它们以统一入侵检测对象 GIDO (generalized intrusion detection objects) 格式进行数据交换。GIDO 是对事件进行编码的标准通用格式。此格式是由 CIDF 描述语言 CISL 定义的。GIDO 数据流可以是发生在系统中的审计事件, 也可以是对审计事件的分析结果。

(1) 事件产生器 (Event generators)

事件产生器的任务是从入侵检测系统之外的计算环境中收集事件, 并将这些事件转换成 CIDF 的 GIDO 格式传送给其他组件。例如, 事件产生器可以是读取 C2 级审计踪迹并将其转换为 GIDO 格式的过滤器, 也可以是被动地监视网络并根据网络数据流产生事件的另一种过滤器, 还可以是 SQL 数据库中产生描述事务的事件的应用代码。

(2) 事件分析器 (Event analyzers)

事件分析器分析从其他组件收到的 GIDO, 并将产生的新 GIDO 再传送给其他组件。分析器可以是一个轮廓描述工具, 统计性地检查现在的事件是否可能与以前某个事件来自同一个时间序列; 也可以是一个特征检测工具, 用于在一个事件序列中检查是否有已知的滥用攻击特征; 此外, 事件分析器还可以是一个相关器, 观察事件之间的关系, 将有联系的事件放到一起, 以利于以后的进一步分析。

(3) 事件数据库 (Event databases)

用来存储 GIDO, 以备系统需要的时候使用。

(4) 响应单元 (Response units)

响应单元处理收到的 GIDO, 并据此采取相应的措施, 如杀死相关进程、将连接复位、修改文件权限等。

由于 CIDF 有一个标准格式 GIDO, 所以这些组件也适用于其他环境, 只需要将典型的环境特征转换成 GIDO 格式, 这样就提高了组件之间的消息共享和互通。

2. CIDF 的通信机制

为了保证各个组件之间安全高效地通信, CIDF 将通信机制构造成一个三层模型: GIDO 层、消息层和协商传输层。

要实现有目的的通信, 各组件就必须能正确理解相互之间传递的各种数据的语义, GIDO 层的任务就是提高组件之间的互操作性, 所以 GIDO 就如何表示各种各样的事件做了详细的定义。

消息层确保被加密认证消息在防火墙或 NAT 等设备之间传输过程中的可靠性。消息层只负责将数据从发送方传递到接收方, 而不携带任何有语义的信息; 同样, GIDO 层也只考虑所传递信息的语义, 而不关心这些消息怎样被传递。

单一的传输协议无法满足 CIDF 各种各样的应用需求, 只有当两个特定的组件对信道使用达成一致认识时, 才能进行通信。协商传输层规定 GIDO 在各个组件之间的传输机制。

3. CIDF 语言

CIDF 的总体目标是实现软件的复用和入侵检测与响应 IDR 组件之间的互操作性。首先,

IDR 组件基础结构必须是安全、健壮、可伸缩的，CIDF 的工作重点是定义了一种应用层的语言公共入侵规范语言 Cisl (Common Intrusion Specification Language)，用来描述 IDR 组件之间传送的信息，以及制定一套对这些信息进行编码的协议。Cisl 可以表示 CIDF 中的各种信息，如原始事件信息、分析结果、响应提示等。

Cisl 使用了一种被称为 S 表达式的通用语言构建方法，S 表达式可以对标记和数据进行简单的递归编组，即对标记加上数据，然后封装在括号内完成编组，这跟 LISP 有些类似。S 表达式的最开头是语义标识符（简称为 SID），用于显示编组列表的语义。例如下面的 S 表达式：

```
(HostName 'first.example.com')
```

该编组列表的 SID 是 HostName，它说明后面的字符串“first.example.com”将被解释为一个主机的名字。

有时候，只有使用很复杂的 S 表达式才能描述出某些事件的详细情况，这就需要使用大量的 SID。SID 在 Cisl 中起着非常重要的作用，用来表示时间、定位、动作、角色、属性等，只有使用大量的 SID，才能构造出合适的句子。Cisl 使用范例对各种事件和分析结果进行编码，把编码的句子进行适当的封装，就得到了 GIDO。

GIDO 的构建与编码是 Cisl 的重点。

4. CIDF 的 API 接口

CIDF 的 API 负责 GIDO 的编码、解码和传递，它提供的调用功能使得程序员可以在不了解编码和传递过程具体细节的情况下，以一种很简单的方式构建和传递 GIDO。

GIDO 的生成分为两个步骤：第一，构造表示 GIDO 的树型结构；第二，将此结构编成字节码。

CIDF 的 API 为实现者和应用开发者都提供了很多的方便，它分为两类：GIDO 编码/解码 API 和消息层 API。

2.4 主要入侵检测系统介绍

入侵检测系统的研究已经全面展开，世界各地都在投入资金进行研究和设计，本节就一些主要的入侵检测系统进行一般的介绍，有些系统是研究项目，而有些系统是商业产品。更详细的了解可以参考相关文档和网站。

1. Snort

Snort 是开放源码网络入侵检测系统 (The Open Source Network Intrusion Detection System) 的缩写。它能够在 IP 网络上执行实时的网络流量分析和数据包记录，它可以执行协议分析、内容检查和匹配，能够用来检测不同的攻击和探测，例如缓冲区溢出，秘密端口扫描，CGI 攻击，SMB 探测，操作系统识别尝试，等等。

Snort 使用灵活的规则语言来描述流量，来判断它是否应该收集或者通过，它有一个优秀的检测引擎，此检测引擎是利用模块插入框架。Snort 可以提供实时的预警能力，它的预警方式很多，例如有系统记录，UNIX 套接字，或者 WinPopup 消息方式。

Snort 有三个主要的用途，它能被用做一个轻量级的网络数据包分析器，就像 tcpdump 一样。也可以用作数据包记录器，这对于网络流量分析是很有帮助的，最后就是它可以作为

一个功能强大的入侵检测系统。

Snort 可以运行在具有 libpcap 库的所有平台上。例如 Linux, OpenBSD, FreeBSD, NetBSD, Solaris, SunOS, HP-UX, AIX, IRIX, Tru64, MacOS X Server, Win32 (Win9x/NT/2000)。

关于 Snort 的详细情况可以参考网站 <http://www.snort.org/>。

2 . LIDS

LIDS 是 Linux Intrusion Detection System 的缩写,即 Linux 入侵检测系统。它是一个内核模块和管理工具,它通过实现命令访问控制 (Mandatory Access Control , MAC) 来提高 Linux 内核的安全性。当它产生作用的时候,文件的访问,所有系统网络管理操作,任何性能使用,原始设备,内存和 I/O 访问都有可能被限制,即使对于超级用户。你可以通过 LIDS 来定义哪些程序可以访问哪些文件。

LIDS 扩大了系统的控制整个系统的绑定能力,增加了一些网络和文件系统安全特性,提高了内核的安全性。你可以通过网络在线调整安全特性,保护敏感的进程,通过网络接收安全警报来实现。LIDS 在 GPL 下发行。

关于 LIDS 的更多信息参考网站 <http://www.lids.org>。

3 . EMERALD

EMERALD 是 Event Monitoring Enabling Responses to Anomalous Live Disturbances 的缩写,它是针对于 Solaris 系统的,它是一个功能非常强大的商业 BSM 分析工具,可以检测用户行为和入侵行为。它具有以下一些特征,如可升级的网络监督,非常多的事件分析功能,轻量级的分析式检测器,基础架构和插件模块相结合,非常容易制定针对新的目的的用户策略。

相关信息参考 <http://www.sdl.sri.com/projects/emerald/>。

4 . AAFID

AAFID (Autonomous Agents For Intrusion Detection) 自治代理入侵检测系统,此系统由普渡大学开发。该系统是一种采用树型分层构造的代理群体,底部的是监视器代理,提供控制、管理以及分析功能,在树叶部分的代理专门用来收集信息。处在中间层的代理被称为收发器,这些收发器一方面实现对底层代理的控制,一方面可以起到信息的预处理过程,把精练的信息反馈给上层的监视器。这种结构采用了本地代理处理本地事件,中央代理负责整体分析的模式。AAFID 系统包含四个部件:监视器(Monitor)、收发器(Transceiver)、代理(Agent)和过滤器(Filter)。这些部件被称为 AAFID 的实体(Entity)。

相关信息参考 <http://www.cerias.purdue.edu/about/history/coast/projects/aafid.php>。

5 . GrIDS

GrIDS (Graph-based Intrusion Detection System) 基于图形的入侵检测系统。

GrIDS 是设计成为在网络系统上检测大规模自动攻击的一个系统。它使用的机制是建立活动图来近似地描述大规模分布式行为的因果结构。活动图的结点相当于系统中的主机,图的边相当于这些主机间的网络活动。用图来描述被监控的网络中的活动。它与入侵模式是不同的,即使他们警告的产生方式很相似。

相关信息参考 <http://seclab.cs.ucdavis.edu/projects/arpa/grids/welcome.html>。

第 3 章 入侵检测原理

3.1 入侵检测模型

3.1.1 IDES 模型

最早的入侵检测模型由 Dorothy Denning 在 1986 年提出。这个模型与具体系统和具体输入无关，对此后的大部分实用系统都很有借鉴价值。图 3-1 表示了这个通用模型的体系结构。

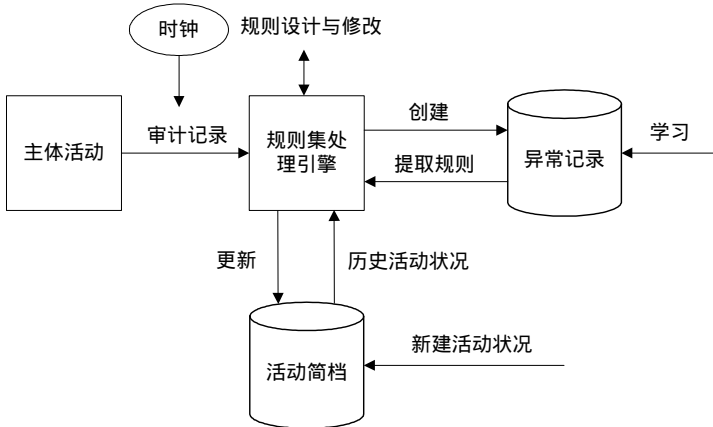


图 3-1 Denning 的入侵检测模型

入侵检测专家系统 IDES（Intrusion Detection Expert System）与它的后继版本 NIDES（Next-Generation Intrusion Detection Expert System）均完全基于 Denning 的模型。该模型的最大缺点是它没有包含已知系统漏洞或攻击方法的知识，而这些知识在许多情况下是非常有用的信息。

该模型由以下 6 个主要部分构成。

主体（Subjects）：启动在目标系统上活动的实体，如用户。

对象（Objects）：系统资源，如文件、设备、命令等。

审计记录（Audit records）：由<Subject, Action, Object, Exception-Condition, Resource-Usage, Time-Stamp>构成的六元组，活动（Action）是主体对目标的操作，对操作系统而言，这些操作包括读、写、登录、退出等；异常条件（Exception-Condition）是指系统对主体的该活动的异常报告，如违反系统读写权限；资源使用状况（Resource-Usage）是系统的资源消耗情况，如 CPU、内存使用率等；时标（Time-Stamp）是活动发生时间。

活动简档（Activity Profile）：用以保存主体正常活动的有关信息，具体实现依赖于检测方法，在统计方法中从事件数量、频度、资源消耗等方面度量，可以使用方差、马尔可夫模型等方法实现。

异常记录 (AnomalyRecord)：由<Event, Time-stamp, Profile>组成。用以表示异常事件的发生情况。

活动规则：规则集是检查入侵是否发生的处理引擎，结合活动简档用专家系统或统计方法等分析接收到的审计记录，调整内部规则或统计信息，在判断有入侵发生时采取相应的措施。

3.1.2 CIDE 模型

公共入侵检测框架 CIDE (Common Intrusion Detection Framework)是为了解决不同入侵检测系统的互操作性和共存问题而提出的入侵检测的框架。因为目前大部分的入侵检测系统都是独立研究与开发的，不同系统之间缺乏互操作性和互用性。一个入侵检测系统的模块无法与另一个入侵检测系统的模块进行数据共享，在同一台主机上两个不同的入侵检测系统无法共存，为了验证或改进某个部分的功能就必须重新构建整个入侵检测系统，而无法重用现有的系统和构件。

CIDE 阐述了一个入侵检测系统的通用模型。它的具体介绍见 2.3.2 节。

3.2 入侵检测技术

分析系统可以采用两种类型的检测技术：异常检测 (Anomaly Detection) 和误用检测 (Misuse Detection)。

3.2.1 异常检测

异常检测也被称为基于行为的检测，基于行为的检测指根据使用者的行为或资源使用状况来判断是否入侵。基于行为的检测与系统相对无关，通用性较强。它甚至有可能检测出以前未出现过的攻击方法，不像基于知识的检测那样受已知脆弱性的限制。但因为不可能对整个系统内的所有用户行为进行全面的描述，况且每个用户的行为是经常改变的，所以它的主要缺陷在于误检率很高。尤其在用户数目众多，或工作目的经常改变的环境中。其次由于统计简表要不断更新，入侵者如果知道某系统在检测器的监视之下，他们能慢慢地训练检测系统，以至于最初认为是异常的行为，经一段时间训练后也认为是正常的了。异常检测的模型如图 3-2 所示。

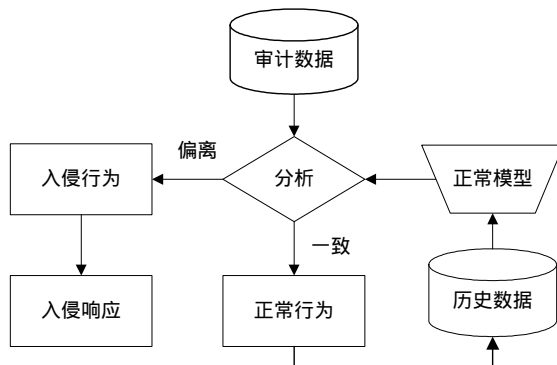


图 3-2 异常检测模型

异常检测方法主要有以下两种。

(1) 统计分析

概率统计方法是基于行为的入侵检测中应用最早也是最多的一种方法。首先，检测器根据用户对象的动作为每个用户都建立一个用户特征表，通过比较当前特征与已存储定型的以前特征，从而判断是否是异常行为。用户特征表需要根据审计记录情况不断地加以更新。用于描述特征的变量类型有：

- 操作密度：度量操作执行的速率，常用于检测通过长时间平均觉察不到的异常行为；
- 审计记录分布：度量在最新纪录中所有操作类型的分布；
- 范畴尺度：度量在一定动作范畴内特定操作的分布情况；
- 数值尺度：度量那些产生数值结果的操作，如 CPU 使用量，I/O 使用量。

这种方法的优越性在于能应用成熟的概率统计理论。但也有一些不足之处，如统计检测对事件发生的次序不敏感，也就是说，完全依靠统计理论可能漏检那些利用彼此关联事件的入侵行为。其次，定义是否入侵的判断阈值也比较困难。阈值太低则漏检率提高，阈值太高则误检率提高。

(2) 神经网络

神经网络的引入对入侵检测系统的研究开辟了新的途径，由于它有很多优点，如自适应性，自学习的能力，因此，在基于神经网络的入侵检测系统中，只要提供系统的审计数据，它就可以通过自学习从中提取正常的用户或系统活动的特征模式，而不必对大量的数据进行存取，精简了系统的设计。基于神经网络的检测方法具有普遍性，可以对多个用户采用相同的检测措施。神经网络适用于不精确模型，统计方法主要依赖用户行为的主观设计，所以此时描述的精确度很重要，不然会引起大量的误报。入侵检测系统可以利用神经网络的分类和识别能力，适用于用户行为的动态变化特征。但基于神经网络的入侵检测系统计算量大，将影响其实时性，可以采用和其他的技术相结合，来构造入侵检测系统。

3.2.2 误用检测

误用检测模型如图 3-3 所示。

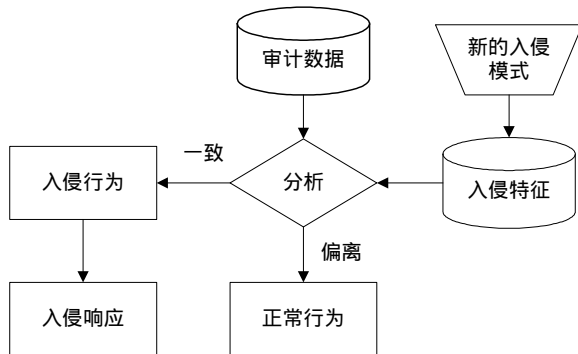


图 3-3 误用检测模型

误用检测也被称为基于知识的检测，它指运用已知攻击方法，根据已定义好的入侵模式，通过判断这些入侵模式是否出现来检测。因为很大一部分的入侵是利用了系统的脆弱性，通

通过分析入侵过程的特征、条件、排列以及事件间关系能具体描述入侵行为的迹象。这种方法由于依据具体特征库进行判断,所以检测准确度很高,并且因为检测结果有明确的参照,也为系统管理员采取相应措施提供了方便。主要缺陷在于与具体系统依赖性太强,不但系统移植性不好,维护工作量大,而且将具体入侵手段抽象成知识也很困难。并且检测范围受已知知识的局限,尤其是难以检测出内部人员的入侵行为,如合法用户的泄露,因为这些入侵行为并没有利用系统脆弱性。

误用检测方法有以下几种。

(1) 模式匹配

模式匹配就是将收集到的信息与已知的网络入侵和系统误用模式数据库进行比较,从而发现违背安全策略的行为。该过程可以很简单,如通过字符串匹配发现一个简单的条目或指令,也可以很复杂,如利用形式化的数学表达式来表示安全状态的变化。模式匹配方法的一大优点是只需收集与入侵相关的数据集合,可以显著减少系统负担,检测的准确率和效率比较高。

(2) 专家系统

专家系统是基于知识的检测中运用最多的一种方法。将有关入侵的知识转化成 if-then 结构的规则,即将构成入侵所要求的条件转化为 if 部分,将发现入侵后采取的相应措施转化成 then 部分。当其中某个或某部分条件满足时,系统就判断为入侵行为发生。其中的 if-then 结构构成了描述具体攻击的规则库,状态行为及其语义环境可根据审计事件得到,推理机根据规则和行为完成判断工作。在具体实现中,专家系统主要面临以下问题:

- 全面性问题,即难以科学地从各种入侵手段中抽象出全面的规则化知识;
- 效率问题,即所需处理的数据量过大,而且在大型系统上,如何获得实时连续的审计数据也是个问题。

(3) 模型推理

模型推理是指结合攻击脚本推理出入侵行为是否出现。其中有关攻击者行为的知识被描述为:攻击者目的,攻击者达到此目的的可能行为步骤,以及对系统的特殊使用等。根据这些知识建立攻击脚本库,每一脚本都由一系列攻击行为组成。检测时先将这些攻击脚本的子集看作系统正面临的攻击。然后通过一个称为预测器的程序模块根据当前行为模式,产生下一个需要验证的攻击脚本子集,并将它传给决策器。决策器收到信息后,根据这些假设的攻击行为在审计记录中的可能出现方式,将它们翻译成与特定系统匹配的审计记录格式。然后在审计记录中寻找相应信息来确认或否认这些攻击。初始攻击脚本子集的假设应满足:易于在审计记录中识别,并且出现频率很高。随着一些脚本被确认的次数增多,另一些脚本被确认的次数减少,攻击脚本不断地得到更新。

(4) 统计检测分析

在统计模型中使用的检测技术一般是异常检测,在统计模型中常用的检测值有下列几种,审计事件的数量,间隔时间,资源耗费情况,等等。现在出现的主要有以下5种统计模型。

操作模型,这种模型是假设异常情况可根据检测结果与一些固定指标相比较来取得,固定指标可以根据经验值或一段时间内的统计平均得到,例如,在短时间内的多次失败的登录很有可能是口令尝试攻击。

方差模型,此模型是根据计算参数的方差,此参数就是上面的检测值,然后设定其

置信区间，当检测值超过置信区间的范围时就表示可能有异常发生。

多元模型，是操作模型的扩展，根据在同一时间内分析多个参数实现检测。

马尔柯夫过程模型，此模型是针对每种类型的事件定义为系统状态，用状态转移矩阵来表示状态的变化，当一个事件发生的时候，或者状态矩阵转移的概率较小则表示可能有异常发生。

时间序列分析，将事件计数与资源耗用根据时间排成序列，如果一个新事件在该时间发生的概率较低，则该事件可能是入侵。

综上所述，统计检测分析方法可以记忆用户的使用习惯，从而具有较高的检测率和可用性，但是它的记忆能力也给入侵者以机会，通过逐步训练使入侵事件符合正常操作的统计规律，从而透过入侵检测系统。

(5) 状态转换分析

状态转换分析就是将状态转换图应用于入侵行为的分析。状态转换法将入侵过程看做一个行为序列，这个行为序列导致系统从初始状态转入被入侵状态。分析时首先针对每一种入侵方法确定系统的初始状态和被入侵状态，以及导致状态转换的转换条件，即导致系统进入被入侵状态必须执行的操作（特征事件）。然后用状态转换图来表示每一个状态和特征事件，这些事件被集成于模型中，所以检测时不需要一个个地查找审计记录。但是，状态转换是针对事件序列分析，所以不善于分析过分复杂的事件，而且不能检测与系统状态无关的入侵。

3.3 入侵检测的发展方向

入侵检测系统的研究还处于一个不完善的阶段，还有很多问题需要解决。但入侵检测系统的发展很快，已经出现了各种各样的新技术，在此就一些问题和发展方向进行阐述。

1. 宽带高速网络的实时入侵检测技术

大量高速网络技术如 ATM、千兆位以太网、G 比特光纤网等在近年内不断出现，在此背景下各种宽带接入手段层出不穷，其中很多已经得到了广泛的应用。如何实现高速网络下的实时入侵检测成为一个现实的问题。

这需要考虑两个方面的问题。首先，入侵检测系统的软件结构和算法需要重新设计，以适应高速网络的环境，重点是提高运行速度和效率。开发设计相应的专业硬件结构，加上配合设计的专业软件是解决这方面的一个途径。

另一个问题是，随着高速网络技术的不断进步和成熟，新的高速网络协议的设计也成为未来的一个发展趋势，如对 TCP/IP 协议的重新设计等，所以，现有的入侵检测系统如何适应和利用未来新的网络协议结构是一个全新的问题。

2. 大规模分布式入侵检测

传统的 IDS 局限于单一的主机或网络架构，对异构系统及大规模的网络检测不是很好，不同的 IDS 系统之间及与其他网络安全系统不能协同工作。为解决这一问题，需要发展分布式入侵检测技术。

传统的集中式模型具有几个明显的缺陷。首先，面对在大规模、异质网络基础上发起的复杂攻击行为，中央控制台的业务负荷将会达到不可承受的地步。以至于无法具有足够能力

来处理来自四面八方的消息事件。其次，由于网络传输的延时问题，到达中央控制台的数据包中的事件消息只是反映了它刚被生成时的环境状态。不能反映随着时间的推移可能已经改变的当前状态。异质网络环境所带来的平台差异性也将给集式模型带来困难。因为每一种攻击行为在不同的平台操作环境中都出现不同类型的模式特征，而且已知的攻击方法数目非常多，这样，在集中式模式的系统中，想要进行较完全的攻击模式的匹配就已经非常困难，更何况要面对不断出现新的攻击手段所带来的更多新的攻击模式的匹配问题。

3. 其他方面

(1) 应用层入侵检测的研究。许多入侵的语义只有在应用层才能理解，而目前的 IDS 仅能检测如 Web 之类的通用协议，而不能处理数据库系统等其他的应用系统。

(2) 智能入侵检测技术的研究。入侵检测方法越来越多样化与综合化，尽管已经有智能体、神经网络等在入侵检测领域应用研究，但智能入侵检测系统研究工作处于尝试性阶段，还不能形成真正实用的产品。

(3) 入侵检测的评测方法的研究。用户需对众多的 IDS 系统进行评价，评价指标包括 IDS 检测范围、系统资源占用和 IDS 系统自身的可靠性。从而设计通用的入侵检测测试与评估方法与平台，实现对多种 IDS 系统的检测已成为当前 IDS 的另一重要研究与发展领域。

(4) 与其他网络安全技术相结合的研究。如与防火墙，病毒检测等网络安全技术相结合，充分发挥各自的长处，协同配合，共同提供一个完整的以防火墙为核心的网络安全体系。

(5) 自身安全性的研究。IDS 本身的健壮性是 IDS 系统好坏的重要指标。IDS 的健壮性要求系统本身在各种网络环境下都能正常工作，并且系统的各个模块之间的通信能够不被破坏。

第 4 章 Linux 网络入侵检测系统设计

本章开始讨论 Linux 下基于网络环境的入侵检测系统的设计与实现。讨论了该系统模型的体系结构，并对各个模块进行了介绍，包括网络数据包捕获模块、网络协议分析模块、存储模块、规则解析模块、入侵事件检测模块、响应模块和界面管理模块。

4.1 系统设计原理

随着计算机网络技术的发展，单独地依靠主机审计信息进行入侵检测难以适应网络安全需求。人们提出了基于网络入侵检测系统的体系结构，这种检测系统根据网络流量，网络数据包和协议来分析来检测入侵，其基本原理如图 4-1 所示。

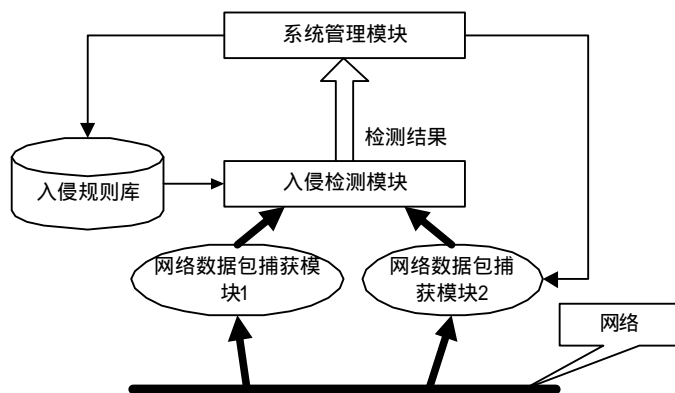


图 4-1 基于网络的入侵检测系统模型

图中网络数据包捕获模块的功能是按一定的规则从网络上获取与安全事件相关的数据包，然后传递给入侵分析引擎模块进行安全分析判断。入侵检测模块将根据网络数据包捕获模块上接收到的包并结合网络入侵规则库进行分析，把分析的结果传递给系统管理模块。管理模块的主要功能是管理其他模块的配置工作，将分析引擎器的结果以有效的方式通知网络管理员与维护入侵标签等。

4.2 主要功能要求

一般而言，一个网络入侵检测系统要满足以下几个主要功能要求。

实时性要求：尽量缩短发生攻击行为和被检测出来之间的时间，及时作出反应。

可扩展性要求：由于攻击方式的多样化，所以可以建立一套能够描述入侵特征的入侵事件描述语言。用此语言可以动态建立入侵规则库。这样，当有新的入侵事件时，就可以使用入侵描述语言动态加载入侵特征到规则库中。

适应性要求：由于网络环境的复杂性，所以要求入侵检测系统也必须能够适应不同的环境。

安全性与可用性要求：入侵检测系统自身的安全性也很重要，不要成为被容易攻击的目标。

有效性要求：系统在一定的规定之内能够满足性能的要求。

4.3 检测器位置

一个比较典型的 IDS 配置示意图如图 4-2 所示。

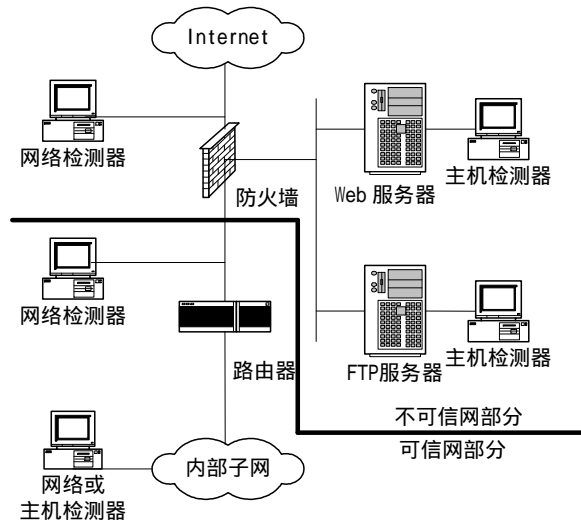


图 4-2 IDS 配置示意图

对于主机型 IDS，其数据采集部分当然位于其所监测的主机上。

基于网络的入侵检测系统需要有检测器才能工作。如果检测器放的位置不正确，入侵检测系统也无法工作在最佳状态。检测器可以有以下几种位置。

(1) 放在防火墙之外

这种安排使检测器可以看见所有来自 Internet 的攻击，然而如果攻击类型是 TCP 攻击，而防火墙或过滤路由器能封锁这种攻击，那么入侵检测系统可能就检测不到这种攻击的发生。因为许多攻击类型只能通过检测是否与字符串特征一致才能被发现，而字符串的传送只有在 TCP 3 次握手才能进行。其优点就是可以看到自己的站点和防火墙暴露在多少种攻击之下。

(2) 放在防火墙之内

如果攻击者能够发现检测器，就可能会对检测器进行攻击，从而减少攻击者的行动被审计的机会，防火墙内的系统会比外面的系统脆弱性少一些，如果检测器在防火墙内就会少一些干扰，从而有可能减少误报警。如果本应该被防火墙封锁的攻击渗透进来，检测器在防火墙内检测到，就能发现防火墙的设置失误。其优点就是设置良好的防火墙能够阻止大部分的“幼稚脚本”的攻击，使检测器不用将大部分的注意力分散在这类攻击上。

(3) 防火墙内外都有检测器

其优点有：

你无须猜测是否有攻击渗透过防火墙；

你可以检测来自内部和外部的攻击。

还有一个问题就是检测器与谁连接。对于交换式以太网交换机，问题则会变得复杂。由于交换机不采用共享媒质的办法，传统的采用一个 sniffer 来监听整个子网的办法不再可行。可解决的办法有：

交换机的核心芯片上一般有一个用于调试的端口（span port），任何其他端口的进出信息都可从此得到。如果交换机厂商把此端口开放出来，用户可将 IDS 系统接到此端口上。优点就是无需改变 IDS 体系结构。缺点是采用此端口会降低交换机性能。

把入侵检测系统放在交换机内部或防火墙内部等数据流的关键入口、出口。其好处是可得到几乎所有关键数据。缺点就是必须与其他厂商紧密合作，且会降低网络性能。

采用分接头（Tap），将其接在所有要监测的线路上。网络分接头（Network tap）被用于建立进行网络监测的永久接入端口。一个分接头，可以安装在任意两台网络设备（如交换机、路由器和防火墙）之间。它可以作为用于采集联机数据的任何监测设备的接入端口，包括入侵检测、协议分析、拒绝服务和远程监测工具等等。由于分接头不对网络传输流产生影响，因此被称为被动设备。如果一个分接头发生故障，传输流将继续传送，网络不会受到影响。在使用光纤分接头的情况下，关键内部部件如光纤分离器，不需要电源，因此不易受到断电的影响。采用分接头的网络结构如图 4-3 所示。优点就是在不降低网络性能的前提下收集了所需的信息。缺点是必须购买额外的设备（Tap）。

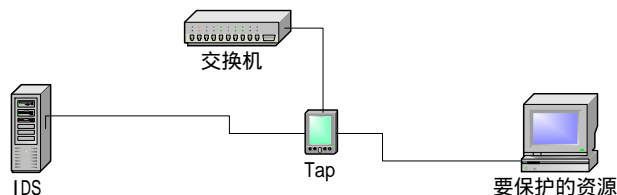


图 4-3 分接头部署

4.4 数据源

前面讲过，现在基本上有两种入侵检测系统类型，一种是基于主机的入侵检测系统（HIDS），另一种是基于网络的入侵检测系统（NIDS），这两种入侵检测系统的数据源总体上是不同的。前者主要针对主机方面的信息来处理的，如系统日志、特殊应用程序日志、特定的进程以及系统调用等等信息。所有数据来源主要是依托于主机，主机环境不同，则其系统的设计也相应改变，其具有很多个体特性。这样针对不同的主机类型，就需要不同的入侵检测模块。例如，如果主机提供 Web 服务，就需要有专门针对此服务的入侵检测模块，如果另外一主机提供了数据库服务，就需要有专门针对数据库服务的入侵检测模块。但基于网络的入侵检测系统是针对于网络而言的，其数据源就是一切网络数据包。

网络数据是目前商业入侵检测系统最为通用的信息来源。基本原理是：当网络数据流在网段中传播时，采用特殊的数据提取技术，收集网络中传输的数据，作为入侵检测系统的数

据来源。许多著名的入侵检测系统，如 RealSecure, NFR, NetRanger, Snort 等都是采用或者部分采用了网络数据作为入侵分析的数据来源。网络数据来源之所以能得到如此广泛的应用，正是由于它本身所具有的一些优势：

通过网络监听的方式获取信息，由于监视器所做的工作仅仅是从网络中读取传输的数据包，因此对受保护系统的性能影响很小或几乎没有，并且无须改动原先的系统和网络结构，只需在网络中添加一个网络监视器就可以了。

网络监视器对网络中用户是透明的，降低了监视器本身遭受入侵者攻击的可能性。

网络监听相对于基于主机的 IDS 更容易检测到某些网络协议的攻击方法。典型的是通过向目标主机发送畸形的和大量的网络包从而造成 DoS 攻击的方法。

网络监视器可以针对一个网络段的数据进行入侵分析，与受保护主机的操作系统无关。与之相比，基于主机的 IDS 必须首先保证操作系统的正常工作，并且需要针对不同的操作系统开发不同的版本。

在计算机网络系统中，局域网普遍采用的是基于广播机制的 IEEE802.3 协议，即以太网（Ethernet）协议。该协议保证传输的数据包能被同一冲突域内的所有主机接收，基于网络的入侵检测正是利用了以太网的这一特性。

以太网卡通常有正常模式（normal mode）和混杂模式（promiscuous mode）这两种工作模式。网卡的工作原理如图 4-4 所示。

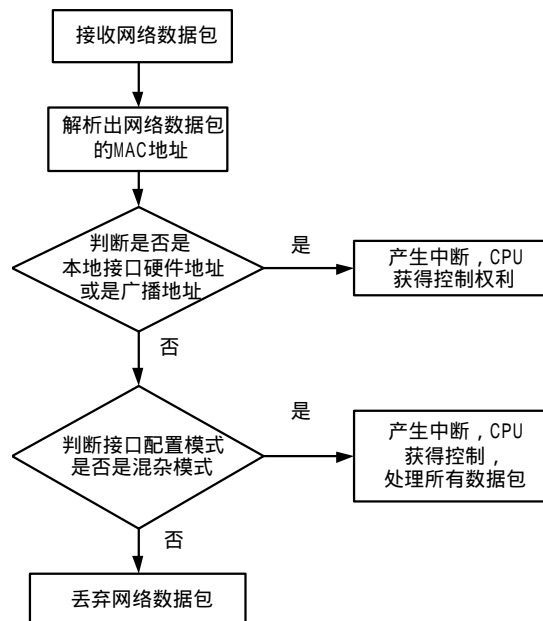


图 4-4 网卡工作原理

在正常模式下，网卡每接收到一个到达的数据包，就会检查该数据包的目的地址，如果是本机地址和广播地址，则将接收数据包放入缓冲区，其他目的地址的数据包则直接丢掉。因此，正常模式下主机仅处理以本机为目的的数据包。网卡的混杂模式则不同，在此种模式下，网卡可以接收本网段内传输的所有数据包，无论这些数据包的目的地址是否是本机。基于网络的入侵检测系统必须利用网卡的混杂模式，获得经过本网段的所有数据信息，从而实

现获取数据的功能。图 4-5 是混杂模式的示意图。

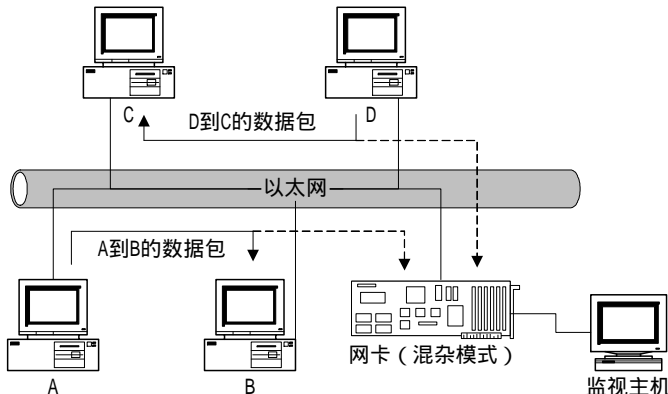


图 4-5 网卡混杂模式

网卡的混杂模式用于在基于广播机制的局域网获取网络数据非常有效，但对于现代的基于交换的网络环境就会出现问题。交换机制使得网络数据包不能在整个局域网内广播，而只能在预先设定的虚网（VLAN）内广播，这就使得网络监听器只能获取本虚网内传输的数据，无法保障整个局域网的安全监视。这是基于网络的入侵检测系统所普遍面临的难题。目前，一些研究团体提出了若干解决方法，采用某些特殊手段对网络监视机制进行改造，以获取我们所感兴趣的数据，具体方法有使用分接头（tap）和利用交换机的监视口（span port）。详细介绍见 4.3 节。

4.5 系统总体结构

系统体系结构如图 4-6 所示。它从逻辑上分为数据采集、数据分析和结果显示三部分，符合 CIDF 的规范。

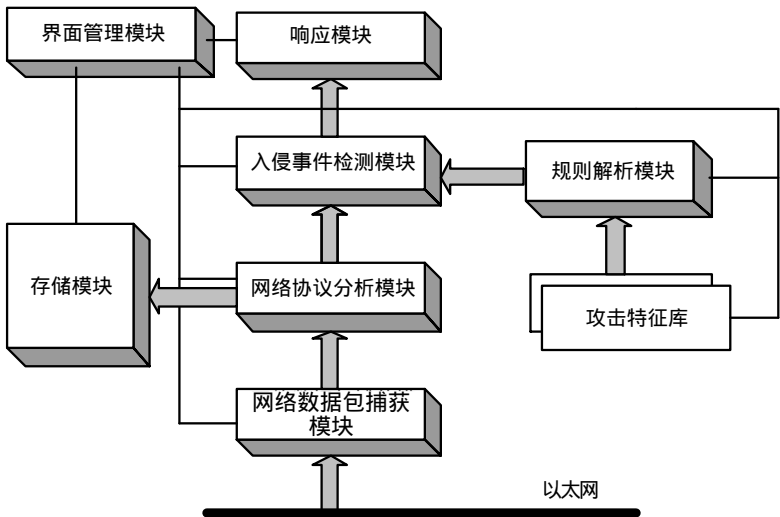


图 4-6 系统体系结构框图

此系统由 7 个模块组成，下面分别对它们进行介绍。

1. 网络数据包捕获模块

此模块是本系统最开始的部分，因为网络入侵检测系统的操作对象是网络数据包，那么首先就必需把所有的网络数据包捕获下来。所以此模块的主要功能就是从以太网中捕获数据包。怎样捕获数据包，不同的操作系统和不同的环境都有不同的实现方式，在 Linux 操作系统中，我们可以使用系统的底层调用来实现，也可以使用相应的高层调用来实现。由于此模块的性能要求很高，因为网络中的数据包很多，如果捕获不及时，就会有漏包的情况出现，这是我们要竭力避免的。怎样提高数据包的捕获性能呢？我们使用一个在 Linux 下非常流行的捕获机制，即 BPF 机制。此数据包捕获机制的性能非常优越，我们就不需要再用底层的调用来编写代码，BPF 封装了底层的调用，并且作了优化处理。在后面章节有详细介绍。

2. 网络协议分析模块

此模块是本系统中一个非常重要的部分，只有在完全对捕获到的数据包进行详细的分析之后，才能在此基础上进一步地分析其是否有入侵行为发生。而此模块的设计功能是否齐全，是否优良就直接影响到入侵检测的性能了。所以此模块是入侵检测系统的基础。

在网络协议分析模块中，必须对捕获到的数据包进行详细的分析，这里就有很多问题出现。例如，由于网络协议非常多，所以就必须分析很多的协议。在本系统中，我们分析了网络中大部分的协议，包括：

- 以太网协议；
- ARP/RARP (Address Resolution Protocol)；
- IP (Internet Protocol)；
- ICMP (Internet Control Message Protocol)；
- TCP (Transmission Control Protocol)；
- UDP (User Datagram Protocol)；
- HTTP (Hyper Text Transfer Protocol)；
- DNS (Domain Name System)；
- IPX/SPX (Internetwork Packet eXchange) / (Sequenced Pakcet eXchange)；
- DHCP (Dynamic Host Configuration Protocol)。

网络协议分析模块对捕获到的数据包进行协议分析，检测出每个数据包的类型和特征。这是此系统的核心部分。

3. 存储模块

此模块主要是存储网络信息。由于网络数据包很多，而且是稍纵即逝的，所以必须及时地把它们存储起来。存储起来有很多用处，最大的好处可以供事后分析，在此基础上可以分析出网络的流量情况，例如分析 IP 协议的分布情况，分析某个 IP 地址的活动情况，等等。

在此模块中，我们使用了 MySQL 数据库。MySQL 数据库是 Linux 下一个非常优秀的数据库系统。其详细介绍可参考相关章节。在这个模块中主要考虑的是存储哪些信息以及存储的手段。

4. 响应模块

响应是入侵检测系统中一个不可分割的部分，因为当系统检测到入侵的时候，只有通过响应来处理相关的事情。响应有主动响应和被动响应之分，在本系统中，采用被动响应方式，通过不同的响应技术来实现响应模块。此响应是实时的。

5. 入侵事件检测模块

此模块是入侵检测系统的主要模块。入侵检测功能就是由此模块实现的，主要的思路就是根据入侵规则库进行协议分析，看是否有入侵行为发生。在这里就可以转化为规则库的匹配问题了。所以如定义好了入侵规则库，并且协议分析完成了，那么现在就是对这两部分进行匹配了。如果协议分析的结果跟入侵规则库匹配成功，就说明有入侵行为发生。入侵规则库的建立是必须完成的。它是一个入侵检测系统的知识库，它的丰富与否，就决定了入侵检测系统的性能。只有入侵规则库越丰富，那么系统检测到的入侵行为就会越多。

另外此模块除了使用入侵规则库来检测入侵之外，还使用了一些异常检测功能。如对扫描入侵行为的检测就使用了异常检测技术。

6. 规则解析模块

规则解析模块在第 8 章详细介绍。此章中详细介绍了入侵规则库的建立。

入侵规则库中有很多种入侵事件，怎样描述入侵事件呢？就需要一个入侵事件描述语言。入侵事件描述语言是入侵检测系统重要的一部分，很多大型的入侵检测系统都有自己的入侵事件描述语言。在本系统中，我们设计了相对简单的入侵事件描述语言，用此语言描述的入侵种类有一定的限制，但基本上可以描述一般的入侵行为，如常见的 UNICODE 攻击等等。读者通过此入侵事件描述语言的学习，就可以了解大型的入侵检测系统的入侵事件描述语言的设计思路了，因为其功能都是大同小异的。

此模块的功能就是把定义好的入侵规则库从文件中读取出来，然后进行解析，读入内存。也就是读入相应的变量之中。怎样解析入侵规则库？它跟入侵规则的入侵事件描述语言密切相关。在第 8 章中都有详细的介绍。

7. 界面管理模块

界面是为了控制操作的方便而设计的。在本系统中使用了 GTK+ 技术来设计界面，GTK+ 是 Linux 系统中一个应用非常广泛的界面开发技术。

界面管理模块的实现中有一个问题要提到的就是动态数据的显示问题。因为我们捕获的数据包是实时的，分析出来的网络数据信息也要实时地显示出来。但使用 GTK+ 时，如果不使用多线程技术，就不可能动态地显示数据信息，因为这时只能当捕获到足够的数据后才能显示出来，在这个过程中，GTK+ 界面是僵死不动的，不能够进行其他的任何操作。所以为了消除这个缺点，在界面的设计中，我们使用了多线程技术，这样用一个线程来捕获数据包，用一个线程来分析数据包，再用一个线程来显示分析后的数据包信息。不仅在界面设计的时候用到多线程，在本系统中到处都用到了多线程技术，例如，刚才讲的，数据包捕获和数据包分析就分别是一个线程，另外入侵检测的功能的实现也是多线程的。多线程技术的使用大大增加了系统的性能。

4.6 小结

由于本系统只是作为研究和教学所用，所以很多技术细节没有考虑，如远程控制问题，数据的安全通信问题和内容显示的美观问题，等等。再就是本系统不是分布式的，如果要扩展成分布式的，还要做很多工作。但本系统实现了几乎所有大型的入侵检测系统的一些基本功能和模块，也是最基础和核心的。所以说“麻雀虽小，五脏俱全”，一般的入侵检测系统大部分也是由这几个模块构成的。

在这个系统中主要对 TCP/IP 协议分析进行了详细探讨，因为 TCP/IP 协议族是互联网的核心协议。对整个协议分析过程进行了详细研究，包括网络数据包捕获机制和协议解析过程。主要对 IP、ARP、TCP、UDP、ICMP 等协议进行了分析。

下面各个章节分别对每个模块的设计和实现进行详细的介绍。

第 5 章 网络数据包捕获模块设计与实现

数据包捕获流程如图 5-1 所示。

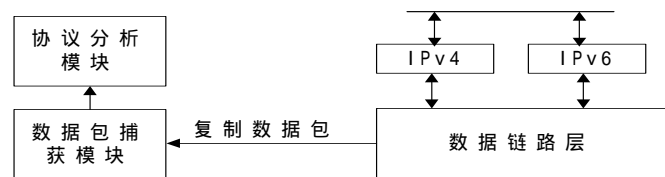


图 5-1 数据包捕获示意图

网络数据包捕获模块将网络接口设置为混杂模式，将网络上传输的数据包截取下来，供协议解析模块使用。

由于效率的需要，有时要根据设置过滤网络上的一些数据包，如特定 IP、特定 MAC 地址、特定协议的数据包。网络监听模块的过滤功能的效率是该网络监听的关键，因为对于网络上的每一数据包都会使用该模块过滤，判断是否符合过滤条件。低效率的过滤程序会导致数据包丢失、分析部分来不及处理等。

为提高效率，数据包过滤应该在系统内核里来实现。本系统采用了专门为数据监听应用程序设计的开发包 Libpcap 来实现该功能，开发包中内置了内核层实现的 BPF 过滤机制和许多接口函数，它们不但能够提高监听部分的效率，也降低了开发的难度，并增强了系统的移植性。

5.1 Linux 内核中 TCP/IP 协议栈分析

Linux 内核中 TCP/IP 协议栈的层次如图 5-2 所示。

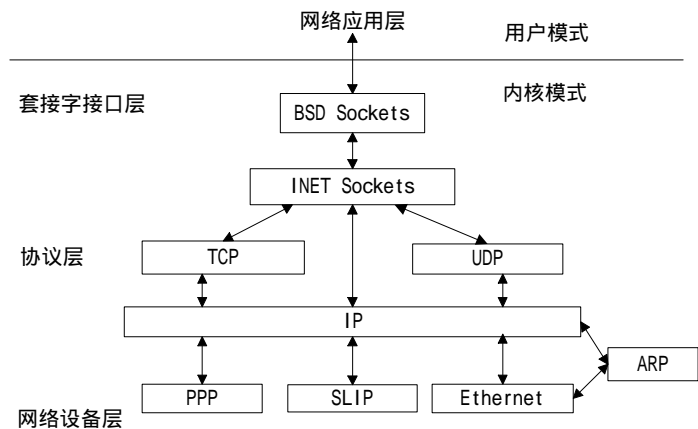


图 5-2 Linux 下 TCP/IP 协议层

1. 网络设备层

在 IP 层之下是 Linux 的网络设备层,其中包括以太网设备或 PPP 设备等。和 Linux 系统中的其他设备不同,网络设备并不总代表实际的物理设备,例如,回环设备就是一个纯软件设备。ARP 协议提供地址解析功能,因此它处于 IP 层和网络设备层之间。

2. 协议层

Linux 用一系列相互连接层的软件实现 Internet 协议族。BSD 套接字 (BSD sockets) 由专门处理 BSD sockets 通用套接字管理软件处理。它由 INET sockets 层来支持,这一层为基于 IP 的协议 TCP 和 UDP 管理传输端点。用户数据报协议 UDP 是一个无连接协议,而传输控制协议 TCP 是一个可靠的端对端协议。TCP 包则被 TCP 连接两端编号以保证传输的数据被正确接收。IP 层包含了实现 Internet 协议的代码。这些代码给要传输的数据加上 IP 头,并知道如何把传入的 IP 包送给 TCP 或 UDP。

3. 套接字接口层

BSD Sockets 是一个通用的接口,它不仅支持各种网络工作形式,而且还是一个交互式通信机制。一个套接字描述一个通信连接的一端,两个通信程序中各自有一个套接字来描述它们自己那一端。Linux 支持多种类型的套接字。这是因为每一类型的套接字有它自己的通信寻址方法。Linux 支持下列套接字地址族或域:

- UNIX : UNIX 域套接字;
- INET : Internet 地址族,支持通过 TCP/IP 协议的通信;
- AX25 : Amateur radio X25 ;
- IPX : Novell IPX ;
- APPLETALK : Appletalk DDP ;
- X25 : X25。

有一些套接字类型支持面向连接的服务类型。但并非所有的地址族能支持所有的服务类型。Linux BSD 套接字支持下列套接字类型。

(1) Stream

这些套接字提供可靠的双工顺序数据流,能保证传送过程中数据不丢失,不被弄混和复制。Internet 地址中的 TCP 协议支持流套接字。

(2) Datagram

这些套接字提供双工数据传送,但与流套接字不同,这里不保证信息的到达。即使它们到达了,也不能保其到达的顺序,甚至不能保证被复制和弄混。这类套接字由 Internet 地址族中的 UDP 协议支持。

(3) Raw

允许直接处理下层协议。例如,有可能打开一个 raw 套接字到以太网设备,看 raw IP 数据传输。

(4) Reliable Delivered Messages

与 Datagram 很相似,但它能保证数据的到达。

(5) Sequenced Packets

与 Stream 很相似，但它的数据包大小是固定的。

(6) Packet

这不是一个标准的 BSD 套接字类型，而是一个 Linux 特定的扩展，它允许在设备级上直接处理包。

INET socket 层支持包括 TCP/IP 协议在内的 internet 地址族。如前所述，这些协议是分层的，一个协议使用另一个协议的服务。Linux 的 TCP/IP 代码和数据结构反映了这一分层模型。它与 BSD socket 层的接口要通过一系列 Internet 地址族 socket 操作，这一操作是在网络初始化时就已经注册到 BSD socket 层。

5.2 BPF 机制

5.2.1 几种分组捕获机制介绍

操作系统所提供的分组捕获机制主要有以下三种。

(1) 数据链路提供者接口 DLPI (Data Link Provider Interface)

数据链路提供者接口 DLPI 定义了数据链路层向网络层提供的服务，是数据链路服务 DLS (Data Link Service) 的提供者 and 使用者间的一种标准接口，在实现上基于 UNIX 的流 STREAMS 机制。DLS 使用者既可以是用户的应用程序，也可以是访问数据链路服务的高层协议如 TCP/IP 和 IPX 等。IEEE802 标准数据链路层划分为两个子层：逻辑链路控制 LLC 子层和介质访问控制 MAC 子层，DLPI 正是 LLC 子层服务的一种基于流的实现。DLPI 实现了 IEEE802.2 标准定义的 LLC 子层类型 1、类型 2 和类型 3 操作，即支持有连接、无连接和有确认无连接 3 种通信服务。使用其他高层协议相比，用 DLPI 开发网络通信程序，具有协议复杂度底、使用简便、传输速度快、实时性较强等诸多优点。

(2) Linux 下的 SOCK_PACKET 类型套接口

Linux 中套接字类型由很多种，详细介绍见 5.1 节。SOCK_PACKET 类型的 socket 可以接收网络上所有数据包。捕获所有网络数据包的过程如下。

建立套接字

```
if_fd = socket(AF_INET, SOCK_PACKET, htons(ETH_P_ALL))
```

其中 AF_INET 表示了 Internet 地址族，SOCK_PACKET 表示了 socket 类型，htons(ETH_P_ALL)是建立 socket 时使用的协议参数，表示该 socket 将接收所有类型的以太网报文，包含 IP 和 ARP。

设置混杂模式

```
struct ifconf ifr;
```

```
ioctl(if_fd, SIOCGIFFLAGS, &ifr)
```

若建立 socket 成功，则对其标志进行操作，首先将标志取出。

```
ifr.ifr_flags |= (IFF_PROMISC|IFF_UP|IFF_RUNNING)
```

设置成 IFF_PROMISC|IFF_UP|IFF_RUNNING，即启动该接口，并将其设成混杂模式。

```
ioctl(if_fd, SIOCSIFFLAGS, &ifr);
```

再将标志写回接口，这样该 socket 以混杂方式运行，可听到连接在该网络接口上的以太网上的所有报文。

(3) 伯克利数据包过滤器 BPF (Berkeley Packet Filter)

基于 BSD 的系统使用 BPF，基于 SVR4 的系统一般使用 DLPI。从文献上看 BPF 比 DLPI 性能好很多，而 SOCK_PACKET 更弱。SCO OpenServer 虽然本身没有内核过滤模块 BPF，但有作为可压入内核的 STREAMS 模块 BPF，采用与伯克利 BPF 相一致的概念。但在 ioctl 操作上，SCO 的 BPF 并不完全提供伯克利 BPF 的所有功能。

图 5-3 是 BPF 的模型及其接口。BPF 有两个主要部件：Network tap 和 Packet filter。Network tap 从网络设备驱动程序中搜集数据拷贝并转发到监听程序。过滤器决定是否接受该数据包和复制数据包的哪些部分。BPF 在内核设置了过滤器，预先可对数据包进行过滤，并且只将用户需要的数据提交给用户进程。每个 BPF 都有一个 buffer，如果过滤器判断接收某个包，BPF 就将它复制到相应的 buffer 中暂存起来，等收集到足够的数据后再一起提交给用户进程，提高了效率。它的缓存机制使用循环双缓存，若其中一缓存满时则将两缓存调换。这种缓存机制对提高效率有很大作用。

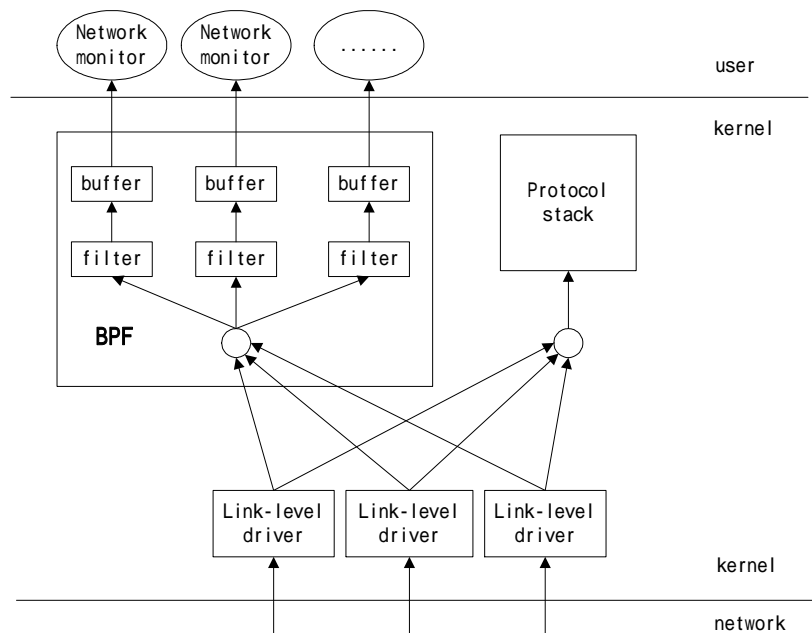


图 5-3 BPF 框架

5.2.2 BPF 过滤机制

BPF 改进了过滤方式。目前基本的过滤规则描述方式有两种：一种是布尔表达树，另一种是 BPF 使用的可控制流图 CFG (Control Flow Graph)。

布尔表达树方式如图 5-4 所示。在树型模型中，每一节点代表一个布尔关系如 AND 和 OR，每个叶子代表一个谓词断语，如 type=IP，边表示布尔操作和操作数的关系。

可控制流图 CFG 方式如图 5-5 所示。在 CFG 模型中，每个节点代表一个谓词断语，而

边代表控制转换。如果谓词断语为 true 则转向右边,反之转向左边。每个 CFG 图有 2 个终结节点,表示返回 true 或者 false。

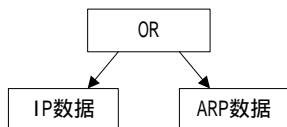


图 5-4 布尔表达树方式

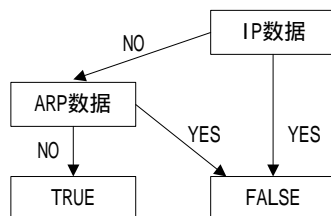


图 5-5 可控制流图 CFG 方式

以上两个模型的计算能力大致相同,用一个模型可以表示的函数用另一个模型也同样可以表示,但是在实现的时候却大不相同,树型模型的操作往往基于数堆栈操作,而 CFG 模型可以基于寄存器操作。树型模型基于操作数堆栈运算,指令将常数或包数据压入堆栈,在堆栈顶层执行布尔运算或位运算,通过连续运算,在堆栈清空时返回结果。堆栈需要内存模拟,其中的 pop 和 push 操作需要指针的加减运算将数据从内存读取和写入内存。另外树型结构可能进行重复操作,例如某个值可能在多个叶子中分别从内存读取并计算,因此这种方式效率较低。BPF 采用 CFG 过滤模型而非过滤树模型。这是因为过滤树模型在解释包时可能进行重复计算,而 CFG 模型解释包时数据包的解释状态和路径是可记忆的,不需要重复计算。CFG 过滤模型比树型过滤模型要快许多倍。

BPF 过滤器的过滤功能是通过虚拟机 (pseudo machine) 执行过滤程序来实现的,过滤机主要由累加器、索引寄存器、数据存储器和隐含的程序计数器几部分组成。过滤程序 (filter program) 实际上是一组过滤规则。过滤规则由用户定义,以决定是否接受数据包和需要接收多少数据。每一条规则执行一组操作,具体操作可以分为指令装载、指令存储、执行算术指令、执行跳转指令、执行返回指令等几个类别。

过滤过程是:当一个数据包到达网络接口时,链路层驱动程序将其提交到系统协议栈。如果 BPF 正在接口监听,驱动程序将首先调用 BPF。BPF 将数据包发送给过滤器 filter,过滤器对数据包过滤,并将数据提交给过滤器关联的上层应用程序。然后链路层驱动程序重新取得控制权,将数据包提交给上层的系统协议栈处理。

网络数据捕获模块将获取所有的网络流量,包括所有协议端口所有子网主机的所有交互数据,但在实际应用中,其中存在若干用户不需要关心的数据,或者称为垃圾数据,垃圾数据在所有流量中占有极大比重,严重影响了系统工作效率的提高,因此高效率的数据包过滤机制是数据包监听的重要组成部分,它使得用户可以指定特定的子网主机,以及特定的协议端口如 HTTP、FTP 等进行过滤,只将用户关心的敏感数据向上层提交,从而提高系统工作效率。而 BPF 的过滤机制很好地完成了此项任务。

5.3 使用 libpcap 函数库

Libpcap (Packet Capture Library) 是一个与实现无关的访问操作系统所提供的分组捕获机制的分组捕获函数库,用于访问数据链路层。其捕获机制就是 BPF 机制。Libpcap 是由 Berkeley 大学 Lawrence Berkeley National Laboratory 研究院的 Van Jacobson、Craig Leres 和

Steven McCanne 编写的。

这个库为不同的平台提供了一致的编程接口，在安装了 libpcap 的平台上，以 libpcap 为接口写的程序、应用，能够自由地跨平台使用。图 5-6 是一个标准的基于 libpcap 库的应用程序流程。

5.3.1 主要函数介绍

下面介绍 libpcap 的一些重要的函数。

`pcap_t *pcap_open_live(char *device, int snaplen, int promisc, int to_ms, char *ebuf)`

获得用于捕获网络数据包的数据包捕获描述字。device 参数为指定打开的网络设备名。snaplen 参数定义捕获数据的最大字节数。promisc 指定是否将网络接口置于混杂模式。to_ms 参数指定超时时间（毫秒）。ebuf 参数则仅在 pcap_open_live() 函数出错返回 NULL 时用于传递错误消息。

`pcap_t *pcap_open_offline(char *fname, char *ebuf)`

打开以前保存捕获数据包的文件，用于读取。fname 参数指定打开的文件名。该文件中的数据格式与 tcpdump(Linux 一个有名的数据包捕获程序)兼容。“-”为标准输入。ebuf 参数则仅在 pcap_open_offline() 函数出错返回 NULL 时用于传递错误消息。

`pcap_dumper_t *pcap_dump_open(pcap_t *p, char *fname)`

打开用于保存捕获数据包的文件，用于写入。fname 参数为“-”时表示标准输出。出错时返回 NULL。p 参数为调用 pcap_open_offline() 或 pcap_open_live() 函数后返回的 pcap 结构指针。fname 参数指定打开的文件名。如果返回 NULL，则可调用 pcap_geterr() 函数获取错误消息。

`char *pcap_lookupdev(char *errbuf)`

用于返回可被 pcap_open_live() 或 pcap_lookupnet() 函数调用的网络设备名指针。如果函数出错，则返回 NULL，同时 errbuf 中存放相关的错误消息。

`int pcap_lookupnet(char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char *errbuf)`

获得指定网络设备的网络号和掩码。netp 参数和 maskp 参数都是 bpf_u_int32 指针。如果函数出错，则返回 -1，同时 errbuf 中存放相关的错误消息。

`int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user)`

捕获并处理数据包。cnt 参数指定函数返回前所处理数据包的最大值。cnt=-1 表示在一个缓冲区中处理所有的数据包。cnt=0 表示处理所有数据包，直到产生以下错误之一：读取到 EOF；超时读取。callback 参数指定一个带有三个参数的回调函数，这三个参数为：一个从 pcap_dispatch() 函数传递过来的 u_char 指针，一个 pcap_pkthdr 结构的指针，和一个数据包大小的 u_char 指针。如果成功则返回读取到的字节数。读取到 EOF 时则返回零值。出错时则返回 -1，此时可调用 pcap_perror() 或 pcap_geterr() 函数获取错误消息。

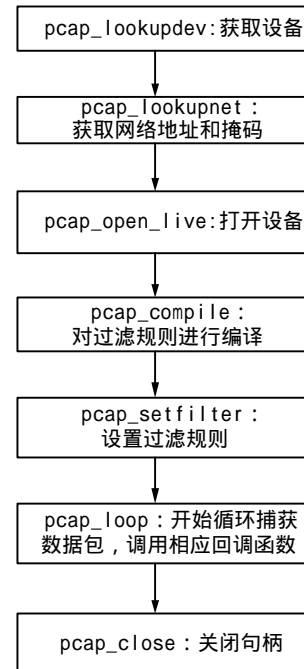


图 5-6 基于 libpcap 库的应用程序流程

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)
```

功能基本与 pcap_dispatch() 函数相同, 只不过此函数在 cnt 个数据包被处理或出现错误时才返回, 但读取超时不会返回。而如果为 pcap_open_live() 函数指定了一个非零值的超时设置, 然后调用 pcap_dispatch() 函数, 则当超时发生时 pcap_dispatch() 函数会返回。cnt 参数为负值时 pcap_loop() 函数将始终循环运行, 除非出现错误。

```
void pcap_dump(u_char *user, struct pcap_pkthdr *h, u_char *sp)
```

向调用 pcap_dump_open() 函数打开的文件输出一个数据包。该函数可作为 pcap_dispatch() 函数的回调函数。

```
int pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask)
```

将 str 参数指定的字符串编译到过滤程序中。fp 是一个 bpf_program 结构的指针, 在 pcap_compile() 函数中被赋值。optimize 参数控制结果代码的优化。netmask 参数指定本地网络的网掩码。

```
int pcap_setfilter(pcap_t *p, struct bpf_program *fp)
```

指定一个过滤程序。fp 参数是 bpf_program 结构指针, 通常取自 pcap_compile() 函数调用。出错时返回 -1; 成功时返回 0。

```
u_char *pcap_next(pcap_t *p, struct pcap_pkthdr *h)
```

返回指向下一个数据包的 u_char 指针。

```
int pcap_datalink(pcap_t *p)
```

返回数据链路层类型, 例如 DLT_EN10MB。

```
int pcap_snapshot(pcap_t *p)
```

返回 pcap_open_live 被调用后的 snapshot 参数值。

```
int pcap_is_swapped(pcap_t *p)
```

返回当前系统主机字节与被打开文件的字节顺序是否不同。

```
int pcap_major_version(pcap_t *p)
```

返回写入被打开文件所使用的 pcap 函数的主版本号。

```
int pcap_minor_version(pcap_t *p)
```

返回写入被打开文件所使用的 pcap 函数的辅版本号。

```
int pcap_stats(pcap_t *p, struct pcap_stat *ps)
```

向 pcap_stat 结构赋值。成功时返回 0。这些数值包括了从开始捕获数据以来至今共捕获到的数据包统计。如果出错或不支持数据包统计, 则返回 -1, 且可调用 pcap_perror() 或 pcap_geterr() 函数来获取错误消息。

```
FILE *pcap_file(pcap_t *p)
```

返回被打开文件的文件名。

```
int pcap_fileno(pcap_t *p)
```

返回被打开文件的文件描述符。

```
void pcap_perror(pcap_t *p, char *prefix)
```

在标准输出设备上显示最后一个 pcap 库错误消息。以 prefix 参数指定的字符串为消息头。

```
char *pcap_geterr(pcap_t *p)
```

返回最后一个 pcap 库错误消息。

```
char *pcap_strerror(int error)
```

如果 `strerror()` 函数不可用，则可调用 `pcap_strerror` 函数替代。

```
void pcap_close(pcap_t *p)
```

关闭 `p` 参数相应的文件，并释放资源。

```
void pcap_dump_close(pcap_dumper_t *p)
```

关闭相应的被打开文件。

5.3.2 编写步骤

使用 `libpcap` 库文件来编写网络数据包捕获程序有一定的规律性，如图 5-6 所示。一般由以下五个部分组成。

设置一些变量和句柄，以便后面的程序所用，大概有以下一些变量。

```
pcap_t *handle
```

会话句柄，这个句柄很重要，其结构体如下：

```
typedef struct pcap pcap_t;
struct pcap {
    int fd;
    int snapshot;
    int linktype;
    int tzoff;      /* timezone offset */
    int offset;     /* offset for proper alignment */
    struct pcap_sf sf;
    struct pcap_md md;
    /*
     * Read buffer.
     */
    int bufsize;
    u_char *buffer;
    u_char *bp;
    int cc;
    /*
     * Place holder for pcap_next().
     */
    u_char *pkt;
    /*
     * Placeholder for filter code if bpf not in kernel.
     */
    struct bpf_program fcode;
    char errbuf[PCAP_ERRBUF_SIZE];
}
```

```
char *dev
```

要监听的设备，如 eth0 等等。

```
char errbuf[PCAP_ERRBUF_SIZE]
```

保存出错信息。

```
struct bpf_program filter
```

存放被编译后的过滤规则。bpf_program 结构体类型描述如下：

```
struct bpf_program {
    u_int bf_len;
    struct bpf_insn *bf_insns;
}
```

```
char filter_app[] = "port 23"
```

过滤表达式，就是 bpf 过滤规则，后面有详细介绍。

```
bpf_u_int32 mask
```

本地网络掩码。bpf_u_int32 类型描述如下：

```
typedef u_int bpf_u_int32
```

```
bpf_u_int32 net
```

存放本地 IP 地址。

```
struct pcap_pkthdr header
```

由 pcap 提供的头。pcap_pkthdr 结构体类型描述如下：

```
struct pcap_pkthdr {
    struct timeval ts; /* 时间标志 */
    bpf_u_int32 caplen; /* 捕获包部分长度 */
    bpf_u_int32 len; /* 当前数据包的长度 */
}
```

```
const u_char *packet
```

用此变量来存放实际捕获到的数据包，后面就可以对此字符串进行分析了。

```
int number
```

设置捕获数据包的个数。就是说要捕获多少个网络数据包。

定位网络设备，以便进行数据包的捕获。

```
dev = pcap_lookupdev(errbuf);
```

用此函数定义设备，把设备赋给 dev 变量，就是前面定义的网络设备变量。如果出错，就把出错信息存放到 errbuf 中，然后可以查看出错的原因。

```
pcap_lookupnet(dev, &net, &mask, errbuf)
```

接着调用此函数，获得指定网络设备的网络号和掩码，同理，如果出错，出错信息就存放在 errbuf 中，以便查找。

```
handle = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf)
```

最后调用此函数以混杂模式打开会话，这个函数就对网络设备进行了参数配置，并打开。具体介绍可以参考上面的对函数的介绍部分。

对过滤规则进行设置。

```
pcap_compile(handle, &filter, filter_app, 0, net)
```

将 filter_app 指定的字符串编译到过滤程序中，以便进行数据包的过滤捕获。此时 filter 参数就是上面介绍的以 bpf_program 类型定义的变量。

```
pcap_setfilter(handle, &filter)
```

此函数就是使刚才定义的过滤规则生效。

捕获数据包。有几种类型的捕获机制，下面分别介绍。

```
packet = pcap_next(handle, &header)
```

此函数是每次只是捕获一个数据包，捕获的数据包就存放给 packet，这样就可以对 packet 表示的数据包进行分析。

```
pcap_loop(handle, number, my_callback_name, packet)
```

此函数很重要，参数 number 就是捕获数据包的个数。此函数注册了自己的回调函数。回调函数的名字即使 my_callback_name，然后就可以在整个回调函数中对数据包进行解析。但要注意此回调函数的类型。其定义如下：

```
void my_callback_name (u_char *, const struct pcap_pkthdr *, const u_char *)
```

my_callback_name 必须这样定义，才能够带入 pcap_loop 函数之中，然后就可以在此函数中来编写代码解析网络数据包。

关闭会话句柄。

```
pcap_close(handle)
```

用此函数来关闭会话，参数 handle 是由 pcap_open_live 函数返回的。

5.3.3 bpf 过滤规则

设置过滤规则就是让网络设备只是捕获我们感兴趣的网络数据包，如果没有设置过滤规则，即上面的 filter_app 是空字符串，那么网络设备就捕获所有类型的数据包，否则只是捕获过滤规则设置的数据包，此时过滤规则的逻辑值为真。此过滤规则是通用的，由著名的网络程序 tcpdump 推出，其他很多的网络程序都是基于此规则进行设计的。此过滤规则的内部解析机制上面介绍过，下面我们参考 tcpdump 的过滤规则形式着重介绍一下过滤规则的定义形式。

过滤规则由一个或多个原语组成。原语通常由一个标识（id，名称或数字）和标识前面的一个或多个限定词组成。

过滤规则有三种类型的限定词，分别为 type、dir 和 proto。

(1) type

类型限定词，说明标识的类型，即名字的类型或数字的类型。大概有 host、net 和 port 三种类型。例如 'host foo', 'net 128.3', 'port 20'。如果不指定类型修饰字，就使用默认的 host。

(2) dir

方向限定词，说明相对于标识的传输方向，即数据是传入还是传出标识。可以使用的方向有 src, dst, src or dst 和 src and dst。例如，'src foo', 'dst net 128.3', 'src or dst port ftp-data'。如果不指定方向限定词，就使用默认的 src or dst。对于 'null' 链路层（就是说像 slip 之类的点到点协议），用 inbound 和 outbound 限定词指定所需的传输方向。

(3) proto

协议限定词。说明过滤规则指定的协议。可以使用的协议有 ether, fddi, ip, arp, rarp, decnet, lat, sca, moprc, mopdl, tcp 和 udp。例如, 'ether src foo', 'arp net 128.3', 'tcp port 21'。如果不指定协议限定词, 就使用所有符合类型的协议。例如, 'src foo' 指 (ip 或 arp 或 rarp) src foo', 'net bar' 指 '(ip 或 arp 或 rarp) net bar', 'port 53' 指 (tcp 或 udp) port 53'。

更复杂的过滤器表达式可以通过 and, or 和 not 连接原语来组建。例如, 'host foo and not port ftp and not port ftp-data'。为了简便, 可以忽略相同的限定词。例如, 'tcp dst port ftp or ftp-data or domain' 实际上等价于 'tcp dst port ftp or tcp dst port ftp-data or tcp dst port domain'。

下面介绍过滤规则的原语, 允许的原语有以下几种。

dst host host1

如果报文中 IP 的目的地址域是 host1, 则逻辑为真。host1 既可以是地址, 也可以是主机名。此过滤规则就是捕获所有目的地址是 host1 的网络数据包。

src host host1

如果报文中 IP 的源地址域是 host1, 则逻辑为真。即捕获所有源地址是 host1 的所有网络数据包。

host host1

如果报文中 IP 的源地址或者目的地址是 host1, 则逻辑为真。即捕获所有源地址或者目的地址是 host1 的网络数据包。

上面所有的 host 表达式都可以加上 ip, arp 或 rarp 关键字做前缀, 就像:

ip host host1

它等价于:

ether proto \ip and host host1

如果 host1 是拥有多个 IP 地址的主机名, 它的每个地址都会被查验。

ether dst ehost

如果报文的以太网目的地址是 ehost, 则逻辑为真。即捕获所有以太网目的地址是 ehost 的网络数据包。其中 ehost 既可以是名字, 如 eth0 (/etc/ethers 里有), 也可以是数字。

ether src ehost

如果报文的以太网源地址是 ehost, 则逻辑为真。即捕获所有以太网源地址是 ehost 的网络数据包。

ether host ehost

如果报文的以太源地址或以太网目的地址是 ehost, 则逻辑为真。即捕获所有以太网源地址或者目的地址是 ehost 的网络数据包。

gateway host

如果报文把 host 当做网关, 则逻辑为真。也就是说, 报文的以太网源或目的地址是 host, 但是 IP 的源目地址都不是 host。此 host 必须是个主机名, 而且必须存在于 /etc/hosts 和 /etc/ethers 中。一个等价的表达式是:

ether host ehost and not host host1

对于 host1 和 ehost, 它既可以是名字, 也可以是数字。

dst net net1

如果报文的 IP 目的地址属于网络号 net1，则逻辑为真。net1 既可以是名字（存在 /etc/networks 中），也可以是网络号。即捕获所有目的地址属于网络 net1 的网络数据包。

src net net1

如果报文的 IP 源地址属于网络号 net1，则逻辑为真。即捕获所有源地址属于网络 net1 的网络数据包。

net net1

如果报文的 IP 源地址或目的地址属于网络号 net1，则过滤规则逻辑为真。即捕获所有源地址或者目的地址属于网络 net1 的网络数据包。

net net1 mask mask1

如果 IP 地址匹配指定网络掩码 mask1 的 net1，则逻辑为真。本原语可以用 src 或 dst 修饰。

net net1/len

如果 IP 地址匹配指定网络掩码的 net1，则逻辑为真，掩码的有效位宽为 len。本原语可以用 src 或 dst 修饰。

dst port port1

如果报文是 ip/tcp 或 ip/udp，并且目的端口是 port1，则逻辑为真。port1 是一个数字，也可以是/etc/services 中定义过的名字。如果使用名字，则检查端口号和协议。如果使用数字，或者有二义的名字，则只检查端口号。例如，dst port 513 将显示 tcp/login 的数据和 udp/who 的数据，而 port domain 将显示 tcp/domain 和 udp/domain 的数据。

src port port1

如果报文的源端口号是 port1，则逻辑为真。即捕获所有源端口号是 port1 的网络数据包。

port port

如果报文的源端口或目的端口是 port，则逻辑为真，上述的任意一个端口表达式都可以用关键字 tcp 或 udp 做前缀，就如：

tcp src port port1

它只匹配源端口是 port1 的 TCP 报文。即捕获所有源端口号是 port1 的协议是 tcp 的网络数据包。

less length

如果报文的长度小于等于 length，则逻辑为真。即只捕获所有长度小于 length 的网络数据包。它等同于：

len <= length.

greater length

如果报文的长度大于等于 length，则逻辑为真。即只捕获所有长度大于 length 的网络数据包。它等同于：

len >= length.

ip proto protocol

如果报文是 IP 数据报，其内容的协议类型是 protocol，则逻辑为真。即只捕获网络协议类型是 protocol 的 ip 数据包。protocol 可以是数字，也可以是下列名称中的一个：icmp, igrp, udp, nd 或 tcp。注意这些标识符 tcp, udp 和 icmp 也同样是关键字，所以必须用反斜杠 (\) 转

义，在 C-shell 中应该是\\。

ether broadcast

如果报文是以太网广播报文，则逻辑为真。即捕获以太网广播数据包。关键字 ether 是可选的。

ip broadcast

如果报文是 IP 广播报文，则逻辑为真。检查全 0 和全 1 的广播约定，并且检查本地的子网掩码。

ether multicast

如果报文是以太多目传送报文 (multicast)，则过滤规则逻辑为真。关键字 ether 是可选的。这实际上是 'ether[0] & 1 != 0' 的简写。

ip multicast

如果报文是 IP 多目传送报文，则过滤规则逻辑为真。即捕获 ip 多目传送报文。

ether proto protocol

如果报文协议属于以太类型的 protocol，则逻辑为真。Protocol 可以是数字，也可以是名字，如 ip, arp 或 rarp。注意这些标识符也是关键字，所以必须用反斜杠(\)转义。如果是 FDDI (例如，'fddi protocol arp')，协议标识来自 802.2 逻辑链路控制 (LLC) 报头，它通常位于 FDDI 报头的顶层。当根据协议标识过滤报文时，那么就假设所有的 FDDI 报文含有 LLC 报头，而且 LLC 报头用的是 SNAP 格式。

decnet src host

如果 DECNET 的源地址是 host，则逻辑为真，该主机地址的形式可能是 "10.123"，或者是 DECNET 主机名。只有配置成运行 DECNET 的 Ultrix 系统支持 DECNET 主机名。

decnet dst host

如果 DECNET 的目的地址是 host，则逻辑为真。

decnet host host

如果 DECNET 的源地址或目的地址是 host，则逻辑为真。

ip, arp, rarp, decnet

是 ether proto p 的简写形式，其中 p 为上述协议的一种。

tcp, udp, icmp

是 ip proto p 的简写形式，其中 p 为上述协议的一种。

expr relop expr

如果这个关系成立，则逻辑为真，其中 relop 是 >, <, >=, <=, =, != 之一，expr 是数学表达式，由常整数 (标准 C 语法形式)，普通的二进制数运算符 [+ , - , * , / , & , |]，一个长度运算符，和指定的报文数据访问算符组成。要访问报文内的数据，使用下面的语法：

proto [expr : size]

proto 是 ether, fddi, ip, arp, rarp, tcp, udp, or icmp 之一，同时也指出了下标操作的协议层。expr 给出字节单位的偏移量，该偏移量相对于指定的协议层。Size 是可选项，指出感兴趣的字节数；它可以是 1, 2, 4，默认为 1 字节。由关键字 len 给出的长度运算符指明报文的长度。

例如，'ether[0] & 1 != 0' 捕捉所有的多目传送报文。表达式 'ip[0] & 0xf != 5' 捕捉所有带可选域的 IP 报文。表达式 'ip[6:2] & 0x1fff = 0' 只捕捉未分片和片偏移为 0 的数据报。这种

检查隐含在 tcp 和 udp 下标操作中。例如, tcp[0] 一定是 TCP 报头的第一个字节, 而不是其中某个 IP 片的第一个字节。

原语可以用下述方法结合使用:

圆括弧括起来的原语和操作符。取反操作 ('!' or 'not'), 联结操作 ('&&' or 'and') 或操作 ('||' or 'or')。取反操作有最高优先级。或操作和联结操作有相同的优先级, 运算时从左到右结合。注意联结操作需要显式的 and 算符, 而不是并列放置。

如果给出标识符, 但没给关键字, 那么暗指最近使用的关键字。例如, not host vs and ace 作为 not host vs and host ace 的简写形式, 不应该和 not (host vs or ace)混淆。表达式参数可以作为单个参数传给捕获程序, 也可以作为复合参数, 后者更方便一些。

下面使用 tcpdump 程序来对过滤规则进行举例介绍。

显示所有进出 sundown 的报文:

```
tcpdump host sundown
```

显示 helios 和主机 hot, ace 之间的报文传送:

```
tcpdump host helios and \( hot or ace \)
```

显示 ace 和除了 helios 以外的所有主机的 IP 报文:

```
tcpdump ip host ace and not helios
```

显示本地的主机和 Berkeley 的主机之间的网络数据:

```
tcpdump net ucb-ether
```

显示所有通过网关 snup 的 ftp 报文(注意这个表达式被单引号括起, 防止 shell 解释圆括弧):

```
tcpdump 'gateway snup and (port ftp or ftp-data)'
```

显示既不是来自本地主机, 也不是传往本地主机的网络数据(如果你把网关通往某个其他网络, 这个做法将不会把数据发往你的本地网络):

```
tcpdump ip and not net localnet
```

显示每个 TCP 会话的起始和结束报文(SYN 和 FIN 报文) , 而且会话方中有一个远程主机:

```
tcpdump 'tcp[13] & 3 != 0 and not src and dst net localnet'
```

显示经过网关 snup 中大于 576 字节的 IP 数据报:

```
tcpdump 'gateway snup and ip[2:2] > 576'
```

显示 IP 广播或多目传送的数据报, 这些报文不是通过以太网的广播或多目传送形式传送的:

```
tcpdump 'ether[0] & 1 = 0 and ip[16] >= 224'
```

显示所有不是回响请求/应答的 ICMP 报文:

```
tcpdump 'icmp[0] != 8 and icmp[0] != 0'
```

5.4 实现数据包捕获模块

本系统是有界面的, 请参考第 11 章。当系统运行后, 此时系统还是没有实际工作的。当在单击 “ Start ” 按钮的时候, 就调用其回调函数。此回调函数就是本系统的入口。此函数就开始调用响应的一序列函数来实现系统的相关功能。图 5-7 是函数相互调用关系。

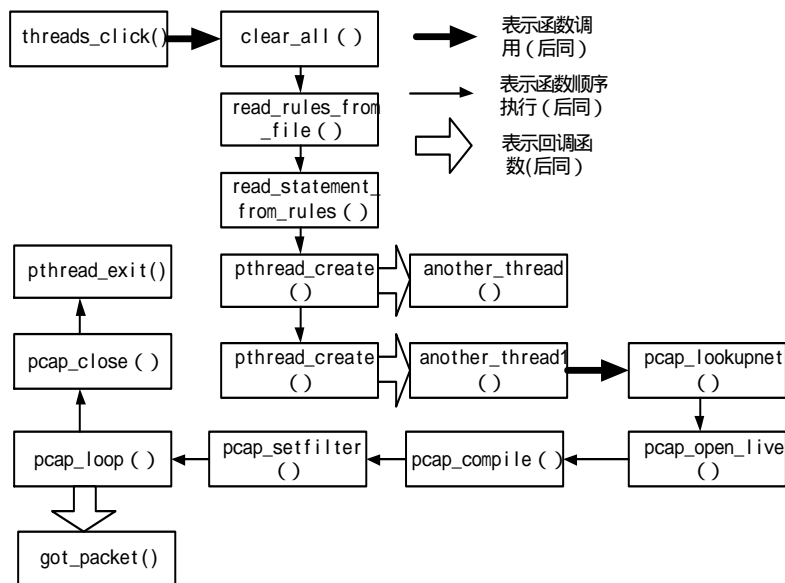


图 5-7 捕获模块中函数相互调用关系

下面是系统入口函数的介绍：

```

void
threads_click (GtkWidget * widget, gpointer data)
{
    //按钮 “ start ” 的回调函数
    pthread_t another, another1;
    //线程句柄，请参考 11.4 章节，多线程技术
    if (sniffer_active == 1) //判断是否运行，如果单击 “ stop ” 按钮就停止
        return;
    threadstop = 0;
    count = 1;
    clear_all (NULL, NULL);
    //清除所有界面的内容
    read_rules_from_file ("rules");
    //从规则库中读取规则
    read_statement_from_rules ();
    //规则解析
    pthread_create (&another, NULL, another_thread, NULL);
    //创建一个线程，其回调函数为 another_thread ( )
    pthread_create (&another1, NULL, another_thread1, NULL);
    //创建另外一个线程，其回调函数为 another_thread1 ( )
}
  
```

在本系统中我们使用了 libpcap 库，其核心代码如下：

```

void *another_thread1(void *args)
//此函数在 threads_click()函数中被设置为一个线程的回调函数
{
    char *dev;
    //网络设备
    char errbuf[PCAP_ERRBUF_SIZE];
    //存储出错信息
    pcap_t *descr;
    //捕获句柄
    struct bpf_program fp;
    // 存储编译过的过滤规则
    bpf_u_int32 maskp;
    // 子网掩码
    bpf_u_int32 netp;
    // ip 地址
    char filter_app[1024] = "";
    // 规律规则，初始值为空
    char string[1024];
    struct hook_and_sinkerhs;
    //定义参考后面
    get_tcp_flags |= GET_TCPD_COUNT_LINKSIZE;
    hs.hook = my_hook;
    //对 hs 进行初始化，设置回调函数，参考后面
    hs.proc_flags = get_tcp_flags ;
    what_to_show |= PP_SHOW_IPHEADER|
                    PP_SHOW_BASICINFO | PP_SHOW_LINKLAYER
                    |PP_SHOW_PACKETCONTENT|PP_SHOW_TCPHEADER
                    |PP_SHOW_UDPHEADER|PP_SHOW_ICMPHEADER;

    sniffer_active=1;
    dev=(char *)malloc(sizeof(char)*1024);
    strcpy(dev,device_total);
    //设置网络设备，由 device_total 定义
    strcpy(filter_app,filter_total);
    //设置过滤规则，由 filter_total 定义
    pcap_lookupnet(dev, &netp, &maskp, errbuf);
    strcpy(buffer,"");
    sprintf(string,"Device: [%s]\n", dev);
    strcpy(buffer,string);
    sprintf(string,"Num of packets: [%d]\n", sniffer_number);
    strcat(buffer,string);
    sprintf(string,"Filter app: [%s]\n", filter_app);
    strcat(buffer,string);

```

```
insert_text1("green");
descr = pcap_open_live(dev, BUFSIZ, 1, 0, errbuf);
//打开网络设备
if (descr == NULL)
{
    printf("pcap_open_live(): %s\n", errbuf);
    exit(1);
}
if (pcap_compile(descr, &fp, filter_app, 0, netp) == -1)
// 应用过滤规则
{
    printf("pcap_compile borqed\n");
    exit(1);
}
if (pcap_setfilter(descr, &fp) == -1)
{
    printf("pcap_setfilter said 'eat shit'\n");
    exit(1);
}
hs.linktype = pcap_datalink(descr);
dumper_filename=pcap_dump_open(descr,savefile_string);
if(dumper_filename==NULL)
printf("dumper_filename erro\n");
pcap_loop(descr, sniffer_number, got_packet, (u_char *)&hs);
//定义回调函数 got_packet，我们在此函数中对捕获到的网络数据包进行分
//析，在此基础上进行入侵检测分析，参考第 6.2 节。
pcap_close(descr);
//关闭会话
pcap_dump_close(dumper_filename);
//关闭捕获文件
sprintf(string,"Done sniffing\n");
strcpy(buffer,string);
insert_text1("purple");
pthread_exit(0);
//退出线程
}
```

一种捕获网络数据包的运行界面如图 5-8 所示。

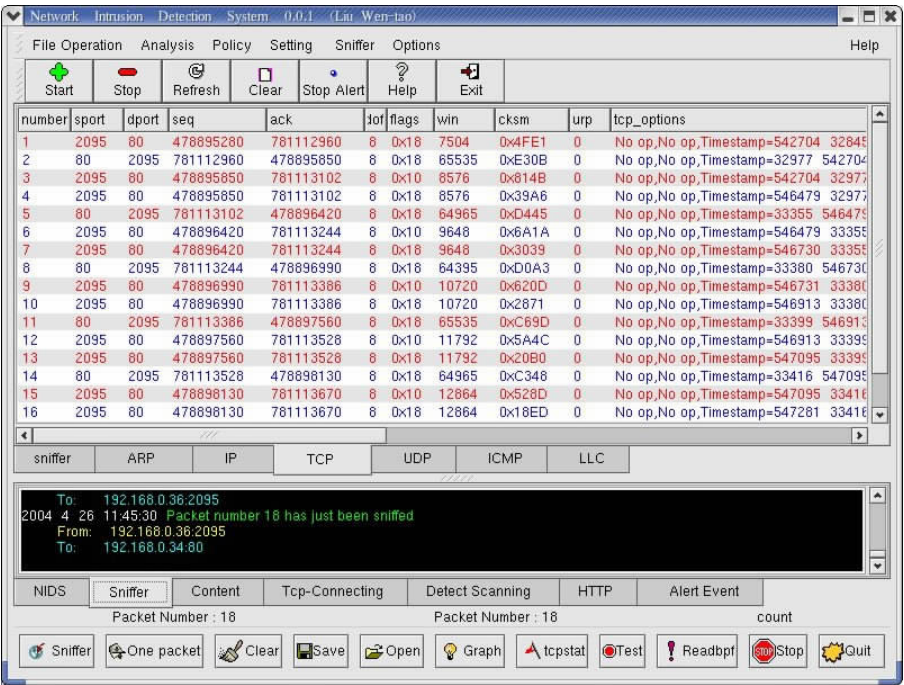


图 5-8 数据包捕获界面

第 6 章 网络协议分析模块设计与实现

网络协议分析模块是整个系统最重要的一部分，而只有很好地对网络协议进行了详细正确的分析，才能够检测出入侵行为。一个优良的入侵检测系统，它的协议分析模块肯定也是优良的。从几个大型的入侵检测系统可以看出，如 Snort 入侵检测系统，还有 Symantec ManHunt、Dragon 等商业入侵检测系统。

在本章，主要详细介绍了怎样设计和实现以太网协议，ARP/RARP，IP，TCP，UDP，ICMP，DHCP，IPX，HTTP 等协议的分析。并仔细实现了对 TCP 协议连接过程的分析，可以分析所有基于 TCP 协议的应用层协议的连接过程，在此基础上我们对 FTP 协议的连接过程进行了举例分析。

6.1 TCP/IP 协议分析基础

6.1.1 概述

1. TCP/IP 模型

TCP/IP 协议并不完全符合 OSI (Open Systems Interconnection) 的 7 层参考模型。传统的开放系统互连参考模型，是一种通信协议的 7 层抽象的参考模型，其中每一层执行某一特定任务。该模型的目的是使各种硬件在相同的层次上相互通信。这 7 层是：物理层 (Physical Layer)、数据链路层 (Data Link Layer)、网络层 (Network Layer)、传输层 (Transport Layer)、会话层 (Session Layer)、表示层 (Presentation Layer) 和应用层 (Application Layer)。而 TCP/IP 通信协议采用了 4 层的层次结构，每一层都呼叫它的下一层所提供的网络来完成自己的需求。这 4 层分别为：

应用层。应用程序间沟通的层，如简单电子邮件传输 (SMTP)、文件传输协议 (FTP)、网络远程访问协议 (Telnet) 等。

传输层。在此层中，它提供了节点间的数据传送服务，如传输控制协议 TCP、用户数据报协议 UDP 等。它的主要功能是格式化传输数据并为数据的传输提供一种机制。为了解决不同应用程序的识别问题，传输层在每个数据报中增加识别信源和信宿应用程序的信息。

网络层。是基于 TCP/IP 协议组的任意网络的核心。负责提供基本的数据封包传送功能，让每一块数据包都能够到达目的主机（但不检查是否被正确接收），如网际协议 (IP)。

链路层。又称网络接口层，对实际的网络媒体的管理，定义如何使用实际网络（如 Ethernet、Serial Line 等）来传送数据。它负责接收网络层数据报并通过传输线发送，或者从传输线上接收数据帧，从中取出数据报，交给网络层。

2. TCP/IP 分层

如上所说 TCP/IP 协议模型由 4 层组成，它是由很多协议共同组成的。图 6-1 是 TCP/IP 协议族分层结构示意图。

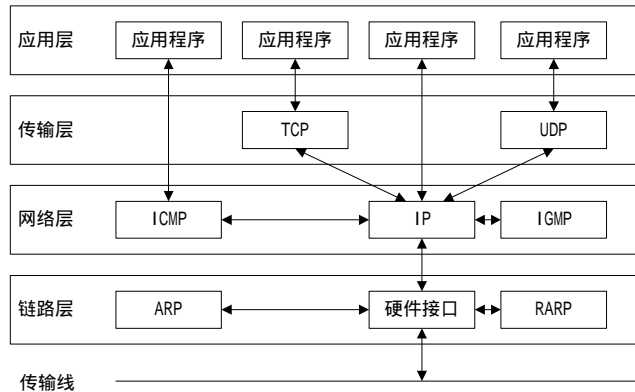


图 6-1 TCP/IP 协议族分层结构示意图

3. 数据包封装与分解

当需要发送数据时候，就要进行封装的过程，如图 6-2 所示。封装的过程就是把用户数据用协议来进行封装，首先由应用层协议进行封装，如 HTTP 协议。而 HTTP 协议是基于 TCP 协议的。它就被 TCP 协议进行封装，http 包作为 TCP 数据段的数据部分。而 TCP 协议是基于 IP 协议的，所以 TCP 段就作为 IP 协议的数据部分，加上 IP 协议头，就构成了 IP 数据报，而 IP 数据报是基于以太网的（假设是以太网环境），所以这个时候就被封装成了以太网帧，这个时候就可以发送数据了。通过物理介质进行传送。

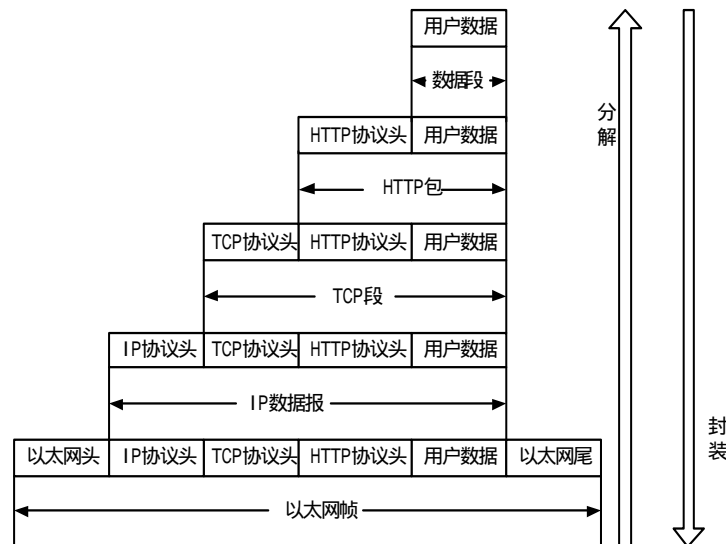


图 6-2 数据包封装与分解

当接收网络数据时候，就要进行数据包分解的过程，如图 6-2 所示。分解的过程与封装的过程恰恰相反，这个时候就需要从一个以太网帧中读出用户数据，就需要一层一层地进行分解，首先是去掉以太网头和以太网尾，在把剩下的部分传递给 IP 层软件进行分解，去掉 IP 头，然后把剩下的传递给传输层，例如 TCP 协议，此时就去掉 TCP 头，剩下应用层协议部

分数据包了，例如 HTTP 协议，此时 HTTP 协议软件模块就会进一步地分解，把用户数据给分解出来，例如是 HTML 代码。这样应用软件就可以操作用户数据了，如用浏览器来浏览 HTML 页面。

6.1.2 IP 协议

网际协议 IP (Internet Protocol) 是 TCP/IP 的核心，也是网络层中最重要的协议。网际协议 IP 是一种不可靠，无连接的协议。它允许两个主机系统在没有任何预先调用设置的情况下交换信息。因为 IP 是无连接的，所以数据报有可能在两个末端用户站点之间丢失。网际协议 IP 对它的末端用户把其底层子网隐藏起来。从这个角度上来看，它给末端用户建立了一个虚拟网络，允许不同类型的网络和 IP 网关相连。其结果使得 IP 的安装非常简单，并且由于 IP 的无连接设计，它的扩展性也非常好。但是，因为 IP 是一个不可靠的数据类型协议，所以它没有可靠性机制。它不给底层子网提供容错恢复功能，也没有流量控制。用户数据可能会丢失，甚至不按顺序到达。

IP 数据报的格式如图 6-3 所示。

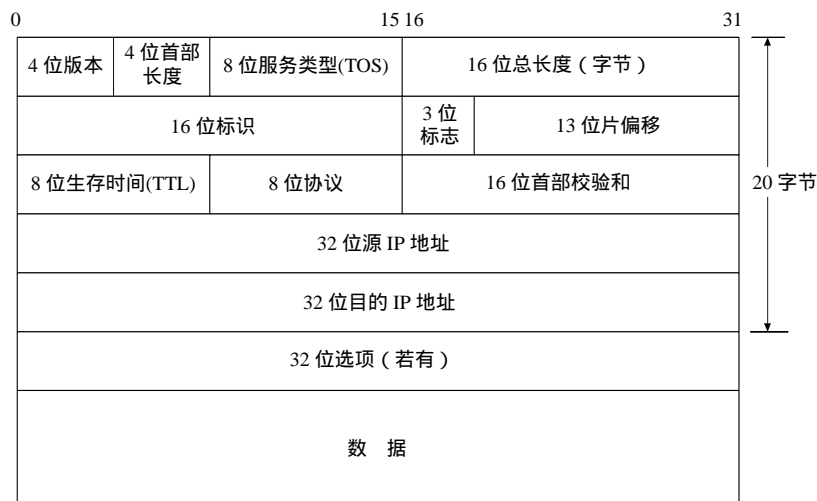


图 6-3 IP 数据报格式

每个字段的意思解释如下。

版本：4 位，表明包头的版本，当前是 4。

首部长度：4 位，以 32 位为单位标记数据包头长度。

服务类型：8 位，标明服务质量的参数，此服务类型用于在特定网络指示所需要的服务。

总长度：16 位，总长度指的是数据报的长度，由字节计，包括数据和报头。允许数据报的大小为 64K 字节。这么大的数据报对大多数主机和网络来说是不适用的。但是，所有主机必须能够接收大于 576 字节的数据报，无论它们是一起来，还是分段来。如果知道对方主机能够接收大于 576 字节的数据报，最好在发送时不要发送小于 576 字节的数据报。选择 576 是因为 $576=512(\text{数据})+64(\text{报头})$ 。报头最长不超过 60 字节，通常为 20 字节。

标识：16 位，标记是发送用于帮助重组分段的包的。

标志：3 位，0 位保留，必须为 0。1 位 (DF)，若为 0 表示可分段，若为 1 表示不可分段。2 位 (MF)，若为 0 表示最后一段，若为 1 表示还有多段。

片偏移：13 位，此域指示这个段在应该在数据报中什么位置，它以 64 位为单位计算，首段的偏移为零。

生存时间：8 位，此域说明数据报在互联网系统生存的最大时间。如果此域的值为零，抛弃此数据报。在处理报头的同时也处理此域。时间以秒计，但每个处理单元都至少会对 TTL 减一，即使时间小于一秒。

协议：8 位，此域指示用于数据报数据部分的下一层协议。

首部校验和：16 位，校验码只在头部，因此头域会在处理时改变，因此头会经常改变。这种校验方法比较容易计算，实验证明它也是适用的，但它可能在未来被 CRC 校验过程取代。

源 IP 地址：32 位，表示发送端的 IP 地址。

目的 IP 地址：32 位，表示接收端的 IP 地址。

选项：长度不定，在数据报中可以有选项也可以没有，但 IP 模块中必须有处理选项的功能。有些情况下，安全选项是必需的。它的长度不定，可以没有也可以是多个。选项有两种格式。一种是单独一个选项类型字节。第二种是一个选项类型字节，一个选项长度字节和实际选项数据字节。选项长度是选项类型，长度本身和数据的长度。选项类型可被看做有 3 个域，分别为复制标记 (1 位)、选项类 (2 位) 和选项号 (5 位)。

6.1.3 TCP 协议

传输控制协议 TCP (Transmission Control Protocol) 是一种可靠的面向连接的传送服务。它在传送数据时是分段进行的，主机交换数据必须建立一个会话。它用比特流通信，即数据被作为无结构的字节流。

通过每个 TCP 传输的字段指定顺序号，以获得可靠性。如果一个分段被分解成几个小段，接收主机知道是否所有小段都已收到。通过发送应答，用以确认别的主机收到了数据。对于发送的每一个小段，接收主机必须在一个指定的时间返回一个确认。如果发送者未收到确认，数据会被重新发送；如果收到的数据包损坏，接收主机舍弃它，因为确认未被发送，发送者会重新发送分段。

TCP 首部的数据格式如图 6-4 所示。

各字段含义说明如下。

源端口和目的端口：都是 16 位，每个 TCP 段都包括源端和目的端的端口号，用于寻找发送端和接收端的应用进程。这两个值加上 IP 首部的源端 IP 地址和目的端 IP 地址惟一确定一个 TCP 连接。

序列号：32 位。当 SYN 出现，序列码实际上是初始序列码 (ISN)，而第一个数据字节是 ISN+1。序号用来标识从 TCP 发送端向接收端发送的数据字节流，它表示在这个报文段中的第一个数据字节。如果将字节流看作在两个应用程序间的单向流动，则 TCP 用序号对每个字节进行计数。

确认号：32 位，如果设置了 ACK 控制位，这个值表示一个准备接收的包的序列码。当建立一个新连接时，SYN 标志变 1。序号字段包含由这个主机选择的该连接的初始序号 ISN，该主机要发送数据的第一个字节的序号为这个 ISN 加 1，因为 SYN 标志使用了一个序号。

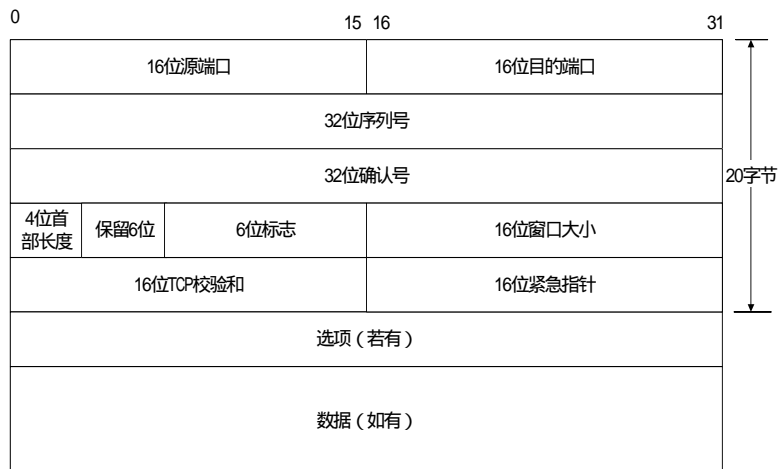


图 6-4 TCP 首部的数据格式

既然每个被传输的字节都被计数，确认序号包含发送确认的一端所期望收到的下一个序号。因此，确认序号应当是上次已成功收到数据字节序号加 1。只有 ACK 标志为 1 时确认序号字段才有效。

发送 ACK 无须任何代价，因为 32 位的确认序号字段和 ACK 标志一样，总是 TCP 首部的一部分。因此一旦一个连接建立起来，这个字段总是被设置，ACK 标志也总是被设置为 1。

TCP 为应用层提供全双工的服务。因此，连接的每一端必须保持每个方向上的传输数据序号。

TCP 可以表述为一个没有选择确认或否认的滑动窗口协议。因此 TCP 首部中的确认序号表示发送方已成功收到字节，但还不包含确认序号所指的字节。当前还无法对数据流中选定的部分进行确认。

首部长度：4 位。首部长度需要设置，因为任选字段的长度是可变的。TCP 首部最多 60 个字节。

标志位：有 6 个，介绍如下：

- URG：紧急指针有效；
- ACK：确认序号有效；
- PSH：接收方应尽快将这个报文段交给应用层；
- RST：重建连接；
- SYN：同步序号用来发起一个连接；
- FIN：发送端完成发送任务。

窗口大小：16 位。TCP 的流量控制由连接的每一端通过声明的窗口大小来提供。窗口大小为字节数，起始于确认序号字段指明的值，这个值是接收端期望接收的字节数。窗口大小是一个 16 位的字段，因而窗口大小最大为 65535 字节。

校验和：16 位。检验和覆盖整个 TCP 报文段：TCP 首部和 TCP 数据。这是一个强制性的字段，一定是由发送端计算和存储，并由接收端进行验证。TCP 校验和的计算和 UDP 首部校验和的计算一样，也使用伪首部。

紧急指针：16 位。紧急指针是一个正的偏移量，与序号字段中的值相加表示紧急数据最

后一个字节的序号。TCP 的紧急方式是发送端向另一端发送紧急数据的一种方式。

6.1.4 UDP 协议

用户数据报协议 UDP(User Datagram Protocol)提供了无连接的数据报服务。它适用于无须应答并且通常一次只传送少量数据的应用软件。

UDP 首部的数据格式如图 6-5 所示。

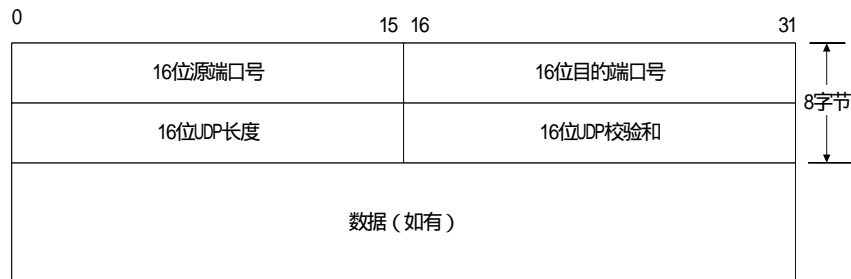


图 6-5 UDP 首部的数据格式

各个字段描述如下。

端口号表示发送和接收的进程。

UDP 长度字段指的是 UDP 首部和 UDP 数据的字节长度，该字段的最小值是 8，表示没有数据。该长度不包括 IP 首部长度。

UDP 校验和包括 UDP 首部和 UDP 数据的校验和。由于 UDP 数据报的长度可以为奇数字节数，因此计算时在最后增加填充字节 0，这只是为了校验和的计算。

6.1.5 ICMP 协议

Internet 控制消息协议 ICMP 是用于报告错误并代表 IP 对消息进行控制的。ICMP 报文在 IP 数据报内部被传输。ICMP 报文格式如图 6-6 所示，所有报文的前 4 个字节都是一样的，但是剩下的其他字节则互不相同。类型字段可以有 15 个不同的值，以描述特定类型的 ICMP 报文。某些 ICMP 报文还使用代码字段的值来进一步描述不同的条件。校验和字段覆盖整个 ICMP 报文。

ICMP 报文的格式如图 6-6 所示。

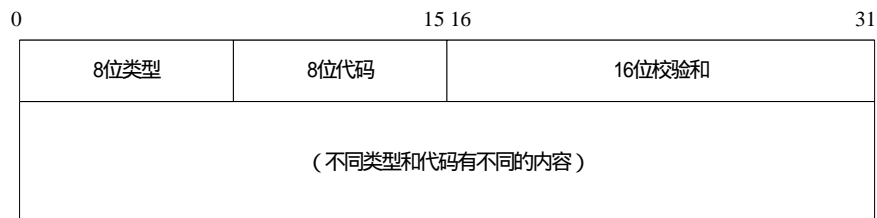


图 6-6 ICMP 报文格式

类型字段可以有 15 个不同的值 (0、3~5、8~18)。某些报文还使用代码字段来进一步描述不同的条件。校验和字段覆盖整个 ICMP 报文，与 IP 首部校验和算法是一样的。

6.2 协议分析模块的实现过程

6.2.1 协议分析过程

图 6-7 是整个协议分析的基本过程。

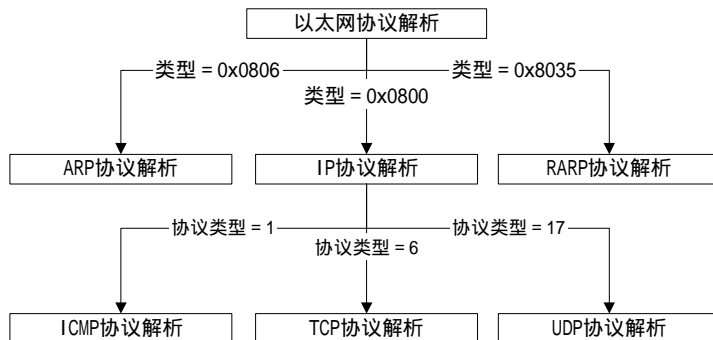


图 6-7 协议分析过程的基本流程

协议分析的主要内容就是把各个协议的内容分析出来。下面以 TCP 协议分析为例子进行介绍。

以字节为单位，不考虑选项的问题。假设我们捕获的以太网数据帧的起始地址为 X ，那么 IP 数据包的起始地址为 $X+14$ ，其中 14 为以太网首部长度。而 TCP 的数据包的起始地址为 $X+14+20$ ，其中 20 为 IP 首部长度。那么 TCP 的负载就是 $X+14+20+20$ ，其中后一个 20 是 TCP 首部长度。这样就能依次读出以太网帧、IP、TCP 的首部内容和 TCP 的负载内容。

第 5 章在设计网络数据包捕获的时候我们使用了下面的一个函数（参考第 5.4 节）：

```
pcap_loop(descr, sniffer_number, got_packet, (u_char *)&hs);
```

在此函数中 `got_packet` 是回调函数。此回调函数就是我们分析协议的位置，在此函数中对协议进行非常详细的分析。此函数实现如下：

```
//----- callback function -----//
void got_packet(u_char *args, const struct pcap_pkthdr *header, const u_char *packet)
{
    //为数据包的特性定义相应的指针变量
    const struct sniff_ethernet *ethernet;
    // 以太网头
    const struct sniff_ip *ip;
    // IP 头 const struct sniff_tcp *tcp;
    // TCP 头
    // 得到以太网头的大小
    int size_ethernet = sizeof(struct sniff_ethernet);
    int size_ip = sizeof(struct sniff_ip);
    //得到 IP 头的大小
    int size_tcp = sizeof(struct sniff_tcp);
```

```
// 得到 TCP 头的大小
char string[1024];
char timestr[1024];
char number[1024];
char destip[1024];
char ether_type_string[1024];
GtkWidget *list;
//界面操作，可以参考后面
GtkWidget *label;
int ether_type;
packetnumber=count;
sprintf(packet_number_string,"%d",packetnumber);
//clear all header_string_object
clear_all_variable();
//清零
analysis_ethernet(args,header,packet);
//分析以太网协议。也是一个回调函数，跟 got_packet 函数类型一样
//another callback is called
if(show_packet_content==1)
proc_pcap(args,header,packet);
//显示网络数据包的内容，也是一个回调函数。参考后面
//another callback is called
process_pcap(args,header,packet);
//协议分析，也是一个回调函数，跟 got_packet 的类型也是一样的。参考后面
if(savefile_yesno==1) //判断是否存储文件
pcap_dump((void *)dumper_filename,header,packet);
//把捕获到的数据包存放到文件中，目的是供事后分析
ethernet = (struct sniff_ethernet*)(packet);
ip = (struct sniff_ip*)(packet + size_ethernet);
tcp = (struct sniff_tcp*)(packet + size_ethernet + size_ip);
ether_type=ntohs(ethernet->ether_type);
switch (ntohs(ethernet->ether_type)) {
case ETHERTYPE_IP:
//以太网类型字段是 IPv4 协议
    strcpy(ether_type_string,"IP");
    break;
#ifdef INET6
case ETHERTYPE_IPV6:
//以太网类型字段是 IPv6 协议
    strcpy(ether_type_string,"IPv6");
    break;
#endif
}
```

```
case ETHERTYPE_ARP:
//以太网类型字段是 ARP 协议
    strcpy(ether_type_string,"ARP");
    gdk_threads_enter();
//多线程操作
    add_list_to_clist1();
//界面动态显示信息（参考界面模块设计）
    gdk_threads_leave();
    break;
case ETHERTYPE_REVARP:
//以太网类型字段是 REVARP 协议
    strcpy(ether_type_string,"REVARP");
    break;
case ETHERTYPE_IPX:
//以太网类型字段是 IPX 协议
    printf("ipx \n");
    strcpy(ether_type_string,"IPX");
    break;
case ETHERTYPE_AT:
//以太网类型字段是 AT 协议
    strcpy(ether_type_string,"AT");
    break;
case ETHERTYPE_AARP:
//以太网类型字段是 AARP 协议
    strcpy(ether_type_string,"AARP");
    break;
default:
    break;
}
//停止捕获
if(threadstop==1)
    pthread_exit(0);
//退出线程的回调函数
gdk_threads_enter();
//线程
inserttime();
//插入时间信息
gdk_threads_leave();
getcurrenttime(timestr);
//获取当前时间
strcpy(snifferpacket.time,timestr);
sprintf(string," Packet number %d has just been sniffed\n", count);
```

```
strcpy(buffer,string);
insert_text1("green");
sprintf(string,"\tFrom:    %s:%d\n", inet_ntoa(ip->ip_src), ntohs(tcp->th_sport));
if(strcmp(ether_type_string,"ARP")==0){
    sprintf(snifferpacket.source,"%s",arp_header_string_object.source_ip);
    sprintf(snifferpacket.sport,"%s","");
}
else{
    sprintf(snifferpacket.source,"%s",inet_ntoa(ip->ip_src));
    sprintf(snifferpacket.sport,"%d",ntohs(tcp->th_sport));
}
strcpy(buffer,string);
insert_text1("yellow");
sprintf(string,"\tTo:      %s:%d\n", inet_ntoa(ip->ip_dst), ntohs(tcp->th_dport));
//保存目的 IP 地址
sprintf(destip,"%s",inet_ntoa(ip->ip_dst));
if(strcmp(ether_type_string,"ARP")==0)
{
    sprintf(snifferpacket.destination,"%s",arp_header_string_object.destination_ip);
    sprintf(snifferpacket.dport,"%s","");
}
else
{
    sprintf(snifferpacket.destination,"%s",inet_ntoa(ip->ip_dst));
    sprintf(snifferpacket.dport,"%d",ntohs(tcp->th_dport));
}
strcpy(buffer,string);
insert_text1("cyan");
gdk_threads_enter();
count++;
//把数据存储到数据库种，参考后面章节
if(use_database_yesno==1)
//判断是否使用数据库
    insert_sniffer_into_database();
//把网络数据包存放到数据库中，参考后面章节
button_add_clicked();
label=lookup_widget(window,"label");
sprintf(number,"Packet Number : %d",packetnumber);
gtk_label_set_text(GTK_LABEL(label),number);
label=lookup_widget(window,"baojinlabel");
if(strcmp(destip,"192.168.0.1")==0)
{
```

```

        gtk_label_set_text(GTK_LABEL(label),"destip is 192.168.0.1");
    }
    gdk_threads_leave();
    get_ip_variable();
    //获取 IP 数据包信息
    get_tcp_variable();
    //获取 TCP 数据包信息
    get_udp_variable();
    //获取 UDP 数据包信息
    get_icmp_variable();
    //获取 ICMP 数据包信息
    whole_parse_rules();
    //入侵规则匹配，在此函数中进行入侵检测，参考后面章节
}

```

图 6-8 是函数关系图。

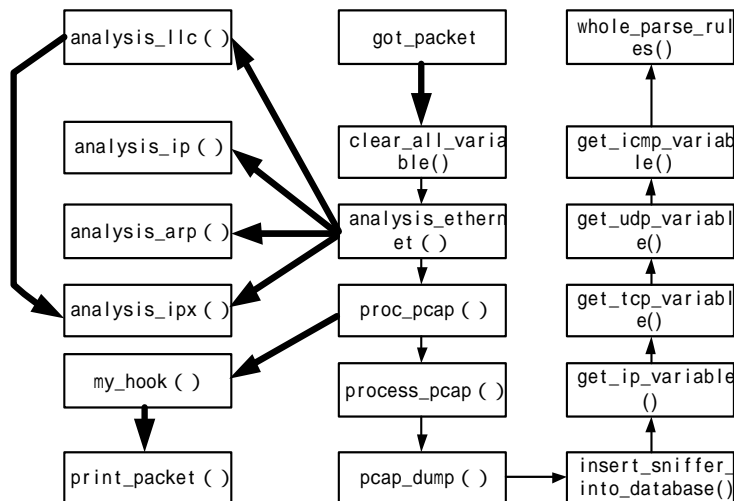


图 6-8 函数关系图

此函数是本系统的最核心的函数，在这个函数里面我们定义了一些核心变量，调用了一些核心的函数，这些函数实现了本系统的主要功能，可以说本系统的大体框架就由此函数决定的。在这里详细的介绍一下里面调用的一些函数，后面有的地方还要对这些函数作进一步的讨论实现。

```
analysis_ethernet(args,header,packet)
```

此函数是对网络数据包进行以太协议分析，参考 6.2.2 节。在 analysis_ethernet 函数中在进一步对其他协议进行分析，如图 6-8 所示，具体每个协议的分析函数的实现都在后面有介绍。

```
proc_pcap(args,header,packet)
```

此函数是对网络数据包进行内容显示。可以以十六进制数形式显示，也可以以十进制数

形式显示，这样就可以分析到具体每个字节的内容是什么。其具体实现请参考 6.2.3 节。

```
process_pcap(args,header,packet)
```

此函数也是分析网络数据包。下面重点介绍此函数的实现：

```
void
process_pcap (u_char * user, const struct pcap_pkthdr *h, const u_char * p)
{
    struct hook_and_sinker *hs;
    //hook_and_sinker 结构体类型（后面有介绍）指针变量
    struct ether_header *ep;
    //以太网协议结构体类型
    u_int length = h->caplen, x;
    u_char *packet;
    hs = (struct hook_and_sinker *) user;
    //把 user 参数强制转化为 hook_and_sinker 类型的，参考本函数的最后一条语句。
    packet = (u_char *) p;
    //存放网络数据包的内容
    ep = (struct ether_header *) p;
    //强制转换成以太网结构体类型
    pdata.link_type = 0;
    switch (hs->linktype)
    {
        case DLT_NULL:
            pdata.link_type |= LINK_NONE;
            switch (*(int *) packet)
            {
                case AF_INET:           //IPv4
                    pdata.link_type |= LINK_NONE_IP;
                    break;
                #ifdef INET6           //IPV6
                case AF_INET6:
                    pdata.link_type |= LINK_NONE_IP6;
                    break;
                #endif
                default:
                    break;
            }
            packet += sizeof (int);
            pdata.packet_len = h->len;
            break;
        case DLT_EN10MB:
            printf ("DLT_EN10MB\n");
```

```
packet += sizeof (struct ether_header);
length -= sizeof (struct ether_header);
bcopy (&(ep->ether_shost), &(pdata.ether.ether_shost),
sizeof (struct ether_addr));
bcopy (&(ep->ether_dhost), &(pdata.ether.ether_dhost),
sizeof (struct ether_addr));
pdata.link_type |= LINK_ETHERNET;
//链路层是以太网类型
switch (ntohs (ep->ether_type))
{
    case ETHERTYPE_IP:
        //网络层协议是 IPv4 类型
        pdata.link_type |= LINK_ETHERNET_IP;
        break;
#ifdef INET6
    case ETHERTYPE_IPV6:
        //网络层协议是 IPv6 类型
        pdata.link_type |= LINK_ETHERNET_IP6;
        break;
#endif
    case ETHERTYPE_ARP:
        //网络层协议是 ARP
        pdata.link_type |= LINK_ETHERNET_ARP;
        break;
    case ETHERTYPE_REVARP:
        //网络层协议是 RARP
        pdata.link_type |= LINK_ETHERNET_REVARP;
        break;
    case ETHERTYPE_IPX:
        //网络层协议是 IPX
        pdata.link_type |= LINK_ETHERNET_IPX;
        break;
    case ETHERTYPE_AT:
        //网络层协议是 AT
        pdata.link_type |= LINK_ETHERNET_AT;
        break;
    case ETHERTYPE_AARP:
        //网络层协议是 AARP
        pdata.link_type |= LINK_ETHERNET_AARP;
        break;
    default:
        break;
}
```

```
    }
    pdata.ether.ether_type = ntohs (ep->ether_type);
    pdata.packet_len = h->len;
    if (!(hs->proc_flags & GET_TCPD_COUNT_LINKSIZE))
        pdata.packet_len -= ETHER_HDR_LEN;
    break;
case DLT_PPP:
//链路层协议是 PPP
    pdata.link_type |= LINK_PPP;
    x = (packet[2] << 8) | packet[3];
    switch (x)
    {
        case 0x0021:
            //协议是 IP
            pdata.link_type |= LINK_PPP_IP;
            break;
        case 0x8021:
            //协议是 IPCP
            pdata.link_type |= LINK_PPP_IPCP;
            break;
#ifdef INET6
        case 0x0057:
            //协议是 IP6
            pdata.link_type |= LINK_PPP_IP6;
            break;
        case 0x8057:
            //协议是 IPCP6
            pdata.link_type |= LINK_PPP_IPCP6;
            break;
#endif
        case 0x80fd:
            //协议是 CCP
            pdata.link_type |= LINK_PPP_CCP;
            break;
        case 0xc021:
            //协议是 LCP
            pdata.link_type |= LINK_PPP_LCP;
            break;
        case 0xc023:
            //协议是 PAP
            pdata.link_type |= LINK_PPP_PAP;
            break;
```

```

        case 0xc223:
            //协议是 CHAP
            pdata.link_type |= LINK_PPP_CHAP;
            break;
        default:
            //协议是其他类型的
            pdata.link_type |= LINK_PPP_OTHER;
            break;
    }
    packet += PPP_HDRLEN;
    length -= PPP_HDRLEN;
    pdata.packet_len = h->len;
    if (!(hs->proc_flags & GET_TCPD_COUNT_LINKSIZE))
        pdata.packet_len -= PPP_HDRLEN;
    break;
default:
#ifdef DEBUG
    printf ("Unknown Link Type: %X\n", hs->linktype);
#endif
    pdata.packet_len = h->len;
    break;
}
length = (length < PAK_SIZ) ? length : PAK_SIZ;
bcopy ((void *) &(h->ts), &(pdata.timestamp), sizeof (struct timeval));
bcopy (packet, &(pdata.data.raw), length);
pdata.buffer_len = length;
hs->hook (&pdata, hs->args);
//调用 hook 指针指向的函数。
}

```

此函数中用到 struct hook_and_sinker 数据结构体类型，此类型定义如下：

```

struct hook_and_sinker
{
    void (*hook) (packet_data *, void **);    //指向回调函数
    void **args;                               //参数
    int proc_flags;                            //标志
    bpf_u_int32 linktype;                     //链路层类型
};

```

这个结构体里面，最重要的是第一个成员，是一个指向回调函数类型的指针变量。

此函数中最重要的一个语句就是最后一行语句：

```
hs->hook (&pdata, hs->args)
```

此行语句就是调用 hook 指针指向的函数，而在这里可以看出就是此时 hook 指向的函数

名字就是本函数的形式参数 user 里面所代表的函数，注意 user 是字符指针类型。再看一下，在函数 got_packet 里面调用本函数语句为：process_pcap(args, header, packet)；所以就是 args 代表的函数，而 args 又是 got_packet 的形式参数，接下来看一下，在函数 another_thread1 (参考第5章)里面调用 got_packet 函数语句为 pcap_loop(descr, sniffer_number, got_packet, (u_char *)&hs)；此函数的最后一个参数为(u_char *)&hs，此参数虽说是一个字符指针类型，但其存放的是一个 hook_and_sinker 结构体类型的内容。another_thread1 函数里面对变量 hs 进行了初始化，其语句为：hs.hook = my_hook；所以用到的回调函数应该是 my_hook 函数。此函数的定义实现如下：

```
void
my_hook (packet_data * pd, void **args)
{
    print_packet (pd, what_to_show);
}
```

此函数又调用了 print_packet 函数，此函数定义实现如下：

```
void
print_packet (packet_data * p, int what_to_print)
{
    struct ip *ip;           //IP 结构体指针变量
    struct tcphdr *tcp;      //TCP 结构体指针变量
    struct udphdr *udp;      //UDP 结构体指针变量
    struct icmp *icmp;       //ICMP 结构体指针变量
#ifdef INET6
    struct ip6_hdr *ip6;     //Ipv6 结构体指针变量
    struct icmp6_hdr *icmp6; //ICMP6 结构体指针变量
#endif
    if (what_to_print & PP_SHOW_BASICINFO)
    {
        printf ("PACKET SIZE: %d", p->packet_len);
        //数据包大小
        if (p->packet_len != p->buffer_len)
            printf (" (BUFFER SIZE: %d)", p->buffer_len);
        //缓冲区大小
        printf ("\n");
    }
    if (what_to_print & PP_SHOW_LINKLAYER)
    {
        if (p->ether.ether_type == ETHERTYPE_ARP)
            //网络层是 ARP 协议
            {
                print_arp_header (&(p->data.arp));
                //分析 ARP 协议，并增添附加功能
            }
    }
}
```

```

        printf ("-----\n");
        return;
    }
}
if ((p->link_type & GENERIC_LINK_OTHER) != GENERIC_LINK_IP
#ifdef INET6
    && (p->link_type & GENERIC_LINK_OTHER) != GENERIC_LINK_IP6
#endif
)
    return;
ip = &(p->data.ip.hdr);
if (is_ip_packet (p))
    tcp = &(p->data.ip.body.tcphdr);
    //定位 TCP 协议指针
if (is_ip_packet (p))
    //判断是否是 ip 包
    udp = &(p->data.ip.body.udphdr);
    //定位 UDP 协议指针
icmp = &(p->data.ip.body.icmp);
//定位 ICMP 协议指针
#ifdef INET6
ip6 = &(p->data.ip6.hdr);
if (is_ip6_packet (p))
{
    tcp = &(p->data.ip6.body.tcphdr);
    udp = &(p->data.ip6.body.udphdr);
    icmp6 = &(p->data.ip6.body.icmp6hdr);
}
#endif
if (what_to_print & PP_SHOW_IPHEADER)
{
    if (is_ip_packet (p))
        print_ip_header (ip);
        //分析 IP 协议，并增添附加功能
#ifdef INET6
        if (is_ip6_packet (p)) //分析 IPv6 协议
            print_ip6_header (ip6);
        #endif
    }
p->buffer_len -= sizeof (struct ip);
switch (get_ip_proto (p))
{

```

```
case IPPROTO_TCP:
    print_tcp_header(tcp, p->buffer_len, what_to_print);
    //分析 TCP 协议，并增添附加功能
    break;
case IPPROTO_UDP:
    print_udp_header(udp, p->buffer_len, what_to_print);
    //分析 UDP 协议，并增添附加功能
    break;
case IPPROTO_ICMP:
    print_icmp_header(icmp, p->buffer_len, what_to_print);
    //分析 ICMP 协议，并增加附加功能
    break;
#ifdef INET6
case IPPROTO_ICMPV6:
    print_icmp6_header(icmp6, p->buffer_len, what_to_print);
    //分析 ICMPv6 协议
    break;
#endif
default:
    printf("UNKNOWN IP PROTOCOL! (0x%.4X)\n", get_ip_proto(p));
    break;
}
}
```

在此 `print_packet` 函数里面，具体进行了协议分析，在这个分析过程中，更增加了一些操作功能，如把分析到的内容以不同的形式显示出来，可以在终端上显示，也可以在界面上动态地彩色地显示出来，还有就是把分析的结果用 MySQL 数据库存储起来，供事后分析。所以这里的分析添加了额外的处理。

下面对各个协议的分析实现过程分别进行详细介绍。

6.2.2 以太网协议分析

以太网采用 CSMA/CD (Carrier Sense Multiple Access with Collision Detection) 技术。IEEE802 委员会制定了一个标准集，包括针对整个 CSMA/CD 网络的 802.3、针对令牌总线网络的 802.4 和针对令牌环网络的 802.5 等。

以太网的封装格式如图 6-9 所示。

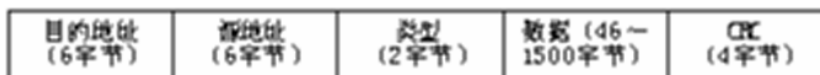


图 6-9 以太网数据帧格式

用来描述以太网数据帧格式的变量和数据类型如下：

```

//一个以太网 MAC 地址的字节数
#define   ETHER_ADDR_LEN      6
//类型字段的字节数
#define   ETHER_TYPE_LEN      2
//CRC 校验字段的字节数
#define   ETHER_CRC_LEN      4
//捆绑头的长度
#define   ETHER_HDR_LEN      (ETHER_ADDR_LEN*2+ETHER_TYPE_LEN)
//最小数据包长度
#define   ETHER_MIN_LEN      64
//最大数据包长度
#define   ETHER_MAX_LEN      1518
//定义一个验证长度的宏
#define   ETHER_IS_VALID_LEN(foo) \
    ((foo) >= ETHER_MIN_LEN && (foo) <= ETHER_MAX_LEN)
//以太网头的数据结构体
struct ether_header
{
    u_char ether_dhost[ETHER_ADDR_LEN]; //目的地址
    u_char ether_shost[ETHER_ADDR_LEN]; //源端地址
    u_short ether_type; //类型字段
};
//48 位以太网地址的数据结构体
struct ether_addr
{
    u_char octet[ETHER_ADDR_LEN];
};
#define   ETHERTYPE_PUP    0x0200           // PUP 协议
#define   ETHERTYPE_IP     0x0800           // IP 协议
#define   ETHERTYPE_ARP    0x0806           // ARP 协议
#define   ETHERTYPE_REVARP  0x8035          // RARP 协议
#define   ETHERTYPE_VLAN   0x8100          // IEEE 802.1Q
#define   ETHERTYPE_IPV6   0x86dd          // IPv6 协议
#define   ETHERTYPE_LOOPBACK 0x9000        // 回地网
#define   ETHERTYPE_IPX    0x8137          // IPX 协议族
//以太网头的数据结构体
typedef struct _EtherHdr
{
    unsigned char ether_dst[6]; //目的地址
    unsigned char ether_src[6]; //源地址
    unsigned short ether_type; //类型
} EtherHdr;

```


本系统是在以太网环境下进行设计的，首先要对以太网协议进行分析。其协议分析代码如下：

```
void analysis_ethernet(u_char *user, const struct pcap_pkthdr *h, u_char *p)
{
    int length;
    int caplen;
    int ether_type;
    EtherHdr *ep;
    //获得整个数据包长度和捕获部分长度
    length = h->len;
    caplen = h->caplen;
    printf("          Ethernet Header(%u.%06u)\n",
        (u_int32_t) h->ts.tv_sec, (u_int32_t) h->ts.tv_usec);
    //检测被删减的头部
    if (caplen < sizeof(EtherHdr))
    {
        printf("Ethernet header too short! (%d bytes)\n", length);
        return;
    }
    ep = (EtherHdr *) p;
    //以太网协议头数据结构
    ether_type = ntohs(ep->ether_type);
    printf("Hardware source: %x: %x: %x: %x: %x: %x \n", *(p+6), *(p+7), *(p+8), *(p+9), *(p+10),
        *(p+11));
    //输出以太网源硬件地址
    printf("Hardware destination: %x: %x: %x: %x: %x: %x \n", *(p), *(p+1), *(p+2), *(p+3),
        *(p+4), *(p+5));
    //输出以太网目的地址
    printf("Protocol type:          %xH \n", ether_type);
    //输出以太网协议类型
    printf("Length:          %d\n", length+4); // add FCS
    packet_ptr = p;
    packet_end = p + caplen;
    //核对 IEEE 802(LLC)封装，如果不是，确保是一个正确的以太网数据包
    p += sizeof(EtherHdr);
    if (ether_type <= ETHERMTU)
    {
        analysis_llc(p,length);
        //分析 LLC
    }
    else
```

```

    {
        switch (ether_type)
        {
            case ETHERTYPE_IP:
                printf("ip in analysis_ethernet \n");
                analysis_ip(p, length);
                //分析 IP 协议
                return;
            case ETHERTYPE_ARP:
            case ETHERTYPE_REVARP:
                analysis_arp(p, length, caplen);
                //分析 ARP 协议
                return;
            case ETHERTYPE_IPX:
                printf("ipx in analysis_ethernet \n");
                analysis_ipx(p, length-sizeof(EtherHdr));
                //分析 IPX 协议
                return;
            default:
                return;
        }
    } // else
}

```

我们来看一下分析 LLC 的过程。其数据结构如下所示：

```

typedef struct llc_header
{
    u_int8_t  dsap;
    u_int8_t  ssap;
} llc_header_t;

```

其分析的实现过程由以下函数来完成。

```

void analysis_llc(const u_char *bp, int length)
{
    llc_header_t  *llc;          //LLC 头
    u_int8_t      org_id[3];
    u_int16_t     ethertype;
    u_int8_t      control;
    llc=(llc_header_t *)bp; //LLC 协议头数据结构
    printf("LLC header \n");
    printf("DSAP : %d",llc->dsap);
    printf("SSAP : %d",llc->ssap);
    if (llc->dsap == 255 && llc->ssap == 255)

```

```

    {
        printf("analysis_ipx\n");
        return;
    }
    control=*(bp+2);
    printf("Control : %d",control);
    gdk_threads_enter();
    add_list_to_clist6();
    //动态显示数据
    gdk_threads_leave();
    if (llc->dsap == 240 && llc->ssap == 240)
    {
        return;
    }
    if (llc->dsap == 224 && llc->ssap == 224)
    {
        analysis_ipx(pkt);
        //分析 IPX 协议
        return;
    }
    if (llc->dsap == 170 && llc->ssap == 170)
    {
        org_id[0]=*(bp+3);
        org_id[1]=*(bp+4);
        org_id[2]=*(bp+5);
        ethertype=*(bp+6);
        ethertype = ntohs(ethertype);
        printf("SNAP header");
        printf("Organization ID   %d",org_id);
        printf("Protocol :   %d",ethertype);
    }
    return;
}

```

6.2.3 ARP 协议分析和 RARP 协议分析

ARP (Address Resolution Protocol) 地址分析协议，其作用是用来分析或映射已知的目的 IP 地址到物理地址 (MAC 子层地址)，以允许在以太网等多种访问介质上通信。网络间的主机相互通信，不仅需要知道彼此的 IP 地址，还必须知道彼此的物理地址，这样它们才能应用数据链路层协议在本地介质上传送数据。

ARP 数据格式如图 6-10 所示。



图 6-10 ARP 请求或应答格式

各字段描述如下。

硬件类型：表示硬件地址的类型，以太网为 1。

协议类型：表示要映射的协议地址类型，IP 地址对应的值为 0x0800。

硬件地址长度：以太网为 6。

协议地址长度：以太网为 4，即 IP 地址长度。

操作字段：ARP 请求=1，ARP 应答=2，RARP 请求=3，RARP 应答=4。

ARP 的工作过程包括：

ARP 请求：信源主机广播一个包含目的 IP 地址的 ARP 请求包。

ARP 响应：当系统收到一份目的端为本机的 ARP 请求报文时，它就把硬件地址填进去，然后用两个目的端地址分别替换两个发送端地址，并把操作字段置为 2，最后把它发送出去。信宿主机识别其 IP 地址，并回应一个包含其物理地址的 ARP 应答来响应信源主机的请求。ARP 请求必须作为广播发送，是以太网中常见的广播流量。ARP 消息不会离开逻辑网络，且从不需要路由。最后为了限制 ARP 广播，主机将保留 ARP 映射在自己的缓存中。

RARP 逆地址分析：为已知物理地址请求一个 IP 地址，如无盘工作站请求服务器提供其 IP 地址。

与 ARP/RARP 协议相关的变量和数据结构定义如下：

```
// ARP 协议操作代码
#define ARPOP_REQUEST      1          // ARP 请求
#define ARPOP_REPLY        2          // ARP 应答
#define ARPOP_RREQUEST     3          // RARP 请求
#define ARPOP_RREPLY       4          // RARP 应答

//ARP 协议头
typedef struct _ARPHdr
{
    unsigned short ar_hrd;           // 硬件地址类型
    unsigned short ar_pro;           // 协议地址类型
    unsigned char ar_hln;            // 硬件地址长度
    unsigned char ar_pln;            // 协议地址长度
    unsigned short ar_op;            // ARP 操作代码
} ARPHdr;

//以太网 ARP 格式
typedef struct _EtherARP
{
    ARPHdr ea_hdr;                  // ARP 头
    unsigned char arp_sha[6];        // 发送方硬件地址
    unsigned char arp_spa[4];        // 发送方协议地址
```

```

        unsigned char arp_tha[6];           // 目的硬件地址
        unsigned char arp_tpa[4];          // 目的协议地址
    } EtherARP;
//存储 ARP 协议信息的数据结构体
struct my_arp_header_string
{
    char hrd[1024];
    char pro[1024];
    char hln[1024];
    char plen[1024];
    char op[1024];
    char source_hardware[1024];
    char source_ip[1024];
    char destination_hardware[1024];
    char destination_ip[1024];
    char information[1024];
};
//定义一个全局变量，后面入侵分析模块也要用到
struct my_arp_header_string arp_header_string_object
ARP/RARP 协议解析的实现由下面的函数来实现：
void analysis_arp(u_char *bp, int length, int caplen)
{
    EtherARP *ap;
    u_short pro, hrd, op;
    struct in_addr spa, tpa;
    char *etheraddr_string(u_char *ep);
    printf("-----ARP Header-----\n");
    ap = (EtherARP *) bp;    //ARP 协议数据结构
    if (length < sizeof(EtherARP))
    {
        printf("Truncated packet\n");
        return;
    }
    hrd = ntohs(ap->ea_hdr.ar_hrd); //硬件类型
    pro = ntohs(ap->ea_hdr.ar_pro); //协议
    op = ntohs(ap->ea_hdr.ar_op);   //操作类型
    printf("Hardware type:          %d\n", hrd);
    printf("Protocol:              %d\n", pro);
    printf("Operation:              %d ", op);
    switch (op)    //判断 ARP 操作类型
    {
        case ARPOP_REQUEST:

```

```

        printf("(ARP request)\n");
        break;
    case ARPOP_REPLY:
        printf("(ARP reply)\n");
        break;
    case ARPOP_RREQUEST:
        printf("(RARP request)\n");
        break;
    case ARPOP_RREPLY:
        printf("(RARP reply)\n");
        break;
    default:
        printf("(unknown)\n");
        return;
}
memcpy((void *) &spa, (void *) &ap->arp_spa, sizeof (struct in_addr));
memcpy((void *) &tpa, (void *) &ap->arp_tpa, sizeof (struct in_addr));
printf("Sender hardware:      %s\n", etheraddr_string(ap->arp_sha));
// 源硬件地址
printf("Sender IP:           %s\n", inet_ntoa(spa));
//源 IP 地址
printf("Target hardware:     %s\n", etheraddr_string(ap->arp_tha));
//目的硬件地址
printf("Target IP:          %s\n", inet_ntoa(tpa));
//目的 IP 地址
}

```

除了本函数实现了分析 ARP 协议数据包以外，还有下面函数也实现了分析 ARP 协议的功能并增加了一些附加的功能。

```

void
print_arp_header (struct arphdr *arp)
{
    printf ("ARP PACKET:\n");
    printf ("\thrd=%d pro=%d hln=%d plen=%d op=%d",
        htons (arp->ar_hrd),
        htons (arp->ar_pro), arp->ar_hln, htons (arp->ar_pln, htons (arp->ar_op));
    sprintf (arp_header_string_object.hrd, "%d", htons (arp->ar_hrd));
    sprintf (arp_header_string_object.pro, "%d", htons (arp->ar_pro));
    sprintf (arp_header_string_object.hln, "%d", arp->ar_hln);
    sprintf (arp_header_string_object.plen, "%d", arp->ar_pln);
    sprintf (arp_header_string_object.op, "%d", htons (arp->ar_op));
    if (use_database_yesno == 1)

```

```

insert_arp_into_database ();
//把 ARP 协议内容存储到数据库中，参考后面章节
printf ("\n");
}

```

6.2.4 IP 协议分析

IP 协议是一个非常重要的协议，对其协议分析实现，也是本系统中最关键的部分。由于 IP 协议的内容较多，在这里我们分几个方面进行实现。图 6-11 是函数关系图。

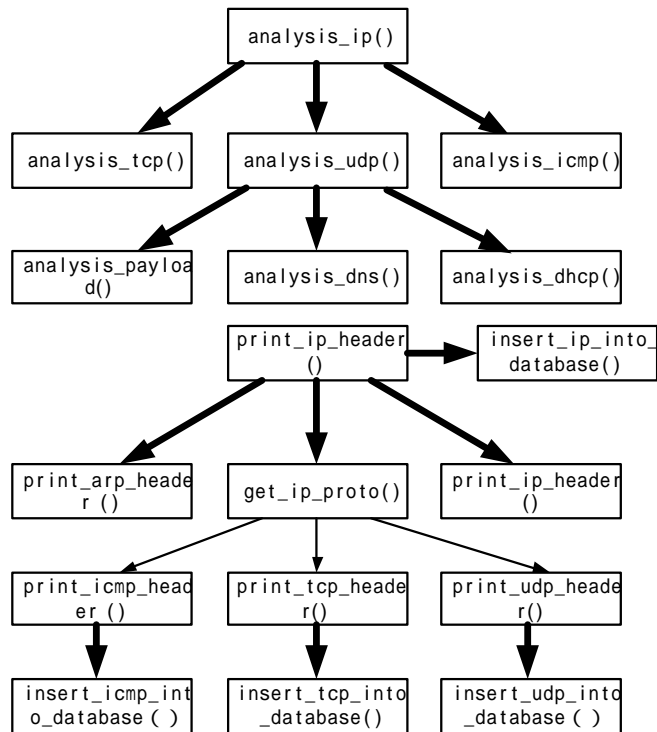


图 6-11 函数关系图

IP 协议的介绍和数据包格式参考上面介绍，此处是实现 IP 协议分析的部分，首先把用来描述 IP 数据包的变量和数据类型定义如下：

```

typedef struct _IPHdr
{
    #if defined(WORDS_BIGENDIAN)
    u_int8_t ip_v:4, ip_hl:4;           //版本和长度
    #else
    u_int8_t ip_hl:4, ip_v:4;
    #endif
    u_int8_t ip_tos;                    //服务类型
    u_int16_t ip_len;                   //总长度
    u_int16_t ip_id;                   //标识

```

```

        u_int16_t ip_off;           //标志和片偏移
        u_int8_t ip_ttl;           //TTL
        u_int8_t ip_p;             //协议类型
        u_int16_t ip_csum;         // 首部校验和
        struct in_addr ip_src;     //源 IP 地址
        struct in_addr ip_dst;     //目的 IP 地址
    } IPHdr;
//下面表示 IP 封装的下一个协议头类型
#define ICMP_NEXT_HEADER 1        //ICMP 协议
#define IP_NEXT_HEADER 4          //IP 协议
#define TCP_NEXT_HEADER 6         //TCP 协议
#define UDP_NEXT_HEADER 17        //UDP 协议
#define GRE_MEXT_HEADER 47        //GRE
#define ESP_NEXT_HEADER 50        //ESP
#define AH_NEXT_HEADER 51         //AH
//存储 IP 数据包信息的数据结构体
struct my_ip_header_string
{
    char version[1024];
    char header_length[1024];
    char tos[1024];
    char total_length[1024];
    char id[1024];
    char off[1024];
    char ttl[1024];
    char protocol[1024];
    char checksum[1024];
    char source_ip[1024];
    char destination_ip[1024];
};

```

//定义一个全局变量，用来存储 IP 信息，后面入侵检测模块也要用到

```
struct my_ip_header_string ip_header_string_object;
```

下面是分析 IP 数据包的核心代码，主要由下面的一个函数实现：

```

void analysis_ip(const u_char *bp, int length)
{
    IPHdr *ip, ip2;
    u_int hlen, len, off;
    u_char *cp = NULL;
    u_int frag_off;
    u_char tos;
    u_int16_t csum;

```



```
u_int16_t my_csum;
ip = (IPHdr *) bp;          //定位 IP 数据报，强制转换
len = ntohs(ip->ip_len);
csum = ntohs(ip->ip_csum);
hlen = ip->ip_hl*4;
printf("Version:           %d\n", ip->ip_v);
//读取版本
printf("Header length:      %d\n", hlen);
//读取首部长度
tos = ip->ip_tos;
printf("Type of service:    %d", tos);
//读取服务类型
printf("Total length:       %d\n", ntohs(ip->ip_len));
//读取总长度
printf("Identification #:   %d\n", ntohs(ip->ip_id));
//读取标识
frag_off = ntohs(ip->ip_off);
printf("Fragmentation offset: %d", (frag_off & 0x1fff) * 8);
//读取片偏移
frag_off &= 0xe000;
printf(" (U=%d, DF=%d, MF=%d)\n", (frag_off & 0x8000) >> 15,
      (frag_off & 0x4000) >> 14, (frag_off & 0x2000) >> 13);
//读取 3 位标志
printf("Time to live:      %d\n", ip->ip_ttl);
//读取生存时间
printf("Protocol:         %d\n", ip->ip_p);
//读取协议
printf("Header checksum:    %d ", csum);
//读取首部校验和
memcpy((void *) &ip2, (void *) ip, sizeof(IPHdr));
ip2.ip_csum = 0;
//校验和清零
my_csum = ntohs(in_cksum((u_int16_t *) &ip2, sizeof(IPHdr)))
//计算校验和
if (my_csum != csum)
//判断校验和
printf("(error: should be %d)", my_csum);
printf("Source address   %s\n", inet_ntoa(ip->ip_src));
//读取源 IP 地址
printf("Destination address %s\n", inet_ntoa(ip->ip_dst));
//读取目的 IP 地址
len -= hlen;
```

```

off = ntohs(ip->ip_off);
if ((off & 0x1fff) == 0)
{
    cp = (u_char *) ip + hlen;
    switch (ip->ip_p)
    {
        case TCP_NEXT_HEADER:
            analysis_tcp(cp, len);
            //分析 TCP 协议(见 6.2.5 节)
            break;
        case UDP_NEXT_HEADER:
            analysis_udp(cp, len);
            //分析 UDP 协议(见 6.2.6 节)
            break;
        case ICMP_NEXT_HEADER:
            analysis_icmp(cp);
            //分析 ICMP 协议(见 6.2.7 节)
            break;
        default:
            break;
    }
}
}
}

```

IP 首部校验和字段是根据 IP 首部计算的校验和码，它不对首部后面的数据进行计算。其算法为：首先，在计算前将校验和域的所有 16 位置为 0，然后 IP 包头从头开始每两个字节位单位相加，若有进位，则和加 1。如此重复直至所有包头中的信息都加完为止，将最后的和求反，即得 16 位的校验和。其中计算校验和的主要代码如下：

```

u_int16_t in_cksum(u_int16_t *addr, int len)
{
    int nleft = len;
    u_int16_t *w = addr;
    u_int32_t sum = 0;
    u_int16_t answer = 0;
    while (nleft > 1)
    {
        sum += *w++; //每 16 位相加
        nleft -= 2;
    }
    if (nleft == 1)
    {
        *(u_int8_t *)&answer = *(u_int8_t *)w;
        sum += answer;
    }
}

```

```

sum = (sum >> 16) + (sum & 0xffff);
// 把高 16 位加低 16 位
sum += (sum >> 16);
// 加上进位
answer = ~sum;
// 求反
return(answer);
}

```

分析 IP 协议的还有下面一个函数，此函数在 print_packet 函数里面被调用。

```

void
print_ip_header (struct ip *ip)
{
    char indnt[64] = " ";
    struct protoent *ip_prot;
    printf (" IP HEADER:\n");
    ip_prot = getprotobyname (ip->ip_p);
    if (ip_prot == NULL)
    {
        printf ("Couldn't get IP Protocol\n");
        return;
    }
    printf ("%sver=%d hlen=%d TOS=%d len=%d ID=0x%.2X", indnt,
#ifdef _IP_VHL
        ip->ip_vhl >> 4, (ip->ip_vhl & 0x0f) << 2,
#else
        ip->ip_v, ip->ip_hl << 2,
#endif
        ip->ip_tos, htons (ip->ip_len), htons (ip->ip_id));
    sprintf (ip_header_string_object.header_length, "%d",
#ifdef _IP_VHL
        ip->ip_vhl >> 4,
#else
        ip->ip_hl << 2
#endif
    );
    sprintf (ip_header_string_object.version, "%d",
#ifdef _IP_VHL
        (ip->ip_vhl & 0x0f) << 2,
#else
        ip->ip_v
#endif
    );
    //存储 IP 协议信息到 ip_header_string_object
}

```

```

sprintf (ip_header_string_object.tos, "%d", ip->ip_tos);
sprintf (ip_header_string_object.total_length, "%d", htons (ip->ip_len));
sprintf (ip_header_string_object.id, "%d", htons (ip->ip_id));
printf ("\n%sFRAG=0x%.2x TTL=%u Proto=%s cksum=0x%.2X\n", indnt,
        htons (ip->ip_off),
        ip->ip_ttl, ip_prot->p_name, htons (ip->ip_sum));
sprintf (ip_header_string_object.off, "%d", htons (ip->ip_off));
sprintf (ip_header_string_object.ttl, "%u", ip->ip_ttl);
sprintf (ip_header_string_object.protocol, "%s", ip_prot->p_name);
sprintf (ip_header_string_object.checksum, "0x%.2X", htons (ip->ip_sum));
printf ("%s%s", indnt, inet_ntoa (ip->ip_src));
sprintf (ip_header_string_object.source_ip, "%s", inet_ntoa (ip->ip_src));
printf (" -> %s\n", inet_ntoa (ip->ip_dst));
sprintf (ip_header_string_object.destination_ip, "%s",
        inet_ntoa (ip->ip_dst));
if (use_database_yn == 1)
    insert_ip_into_database ();
//把 IP 协议内容存放到数据库中，参考后面章节
gdk_threads_enter ();
add_list_to_clist2 ();
//把 IP 协议内容在界面上显示出来，界面参考后面章节
gdk_threads_leave ();
}

```

图 6-12 是分析整个数据包的运行界面，分别用十六进制和字符形式显示出来。

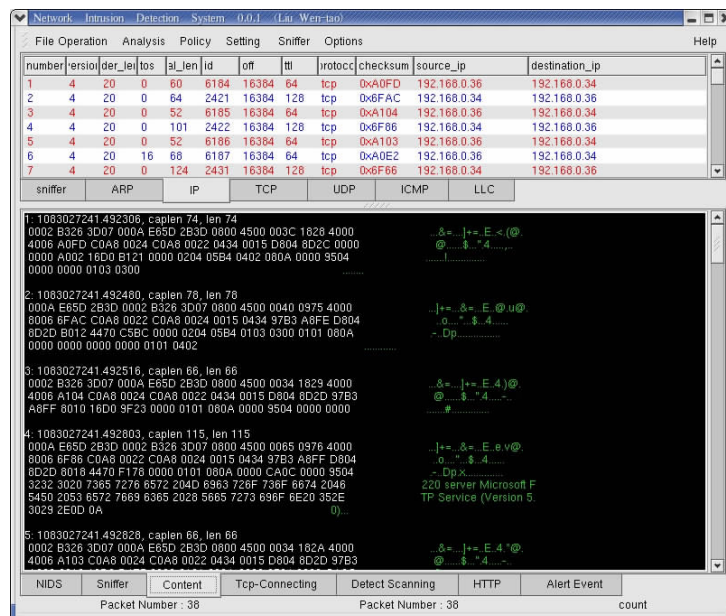


图 6-12 分析整个数据包内容

网络数据包的内容怎样用十六进制和字符形式显示出来，其核心代码如下：

```
void
proc_pcap (u_char * user, const struct pcap_pkthdr *h, const u_char * p)
{
    //此函数在 got_packet 函数被调用，参考前面的介绍。
    u_int length = h->caplen, i, j, k, step;
    u_char *r, *s;
    char c;
    char content[1024];
    char content_str[1024];
    char content_string[1024];
    r = (u_char *) p;
    s = (u_char *) p;
    step = 22;
    printf ("%u: %lu.%6lu, caplen %u, len %u\n",
        count,
        (long unsigned int) h->ts.tv_sec,
        (long unsigned int) h->ts.tv_usec, h->caplen, h->len);
    sprintf (content, "%u: %lu.%6lu, caplen %u, len %u\n",
        count,
        (long unsigned int) h->ts.tv_sec,
        (long unsigned int) h->ts.tv_usec, h->caplen, h->len);
    gdk_threads_enter ();    //多线程操作
    insert_text2 (content);    //动态显示信息
    gdk_threads_leave ();
    for (i = 0; i < length;)
    {
        sprintf (content, " ");
        gdk_threads_enter ();
        insert_text2 (content);
        gdk_threads_leave ();
        for (j = 0; j < step && (j + i) < length;)
        {
            sprintf (content, "%.2X", *r++);
            gdk_threads_enter ();
            insert_text2 (content);
            gdk_threads_leave ();
            j++;
            if ((j + i) == length)
            {
                sprintf (content, " ");
                gdk_threads_enter ();
```

```

        insert_text2 (content);
        gdk_threads_leave ();
        j++;
        break;
    }
    sprintf (content, "%.2X ", *r++);
    gdk_threads_enter ();
    insert_text2 (content); //动态显示数据
    gdk_threads_leave ();
    j++;
}
for (k = j; k < step; k++, k++)
{
    sprintf (content, "          ");
    gdk_threads_enter ();
    insert_text2 (content); //动态显示数据
    gdk_threads_leave ();
}
sprintf (content, "          ");
gdk_threads_enter ();
insert_text2 (content); //动态显示数据
gdk_threads_leave ();
for (j = 0; j < step && (j + i) < length; j++)
{
    c = *p++;
    sprintf (content_string, "%c", char_conv (c)); //返回字符
    gdk_threads_enter ();
    insert_text2_green (content_string); //动态显示数据
    gdk_threads_leave ();
}
sprintf (content, "\n");
gdk_threads_enter ();
insert_text2 (content); //动态显示数据
gdk_threads_leave ();
i += j;
}
sprintf (content, "\n");
gdk_threads_enter ();
insert_text2 (content); //动态显示数据
gdk_threads_leave ();
}

```

6.2.5 TCP 协议分析

此部分是实现 TCP 协议分析的功能。用来描述 TCP 数据包的变量和数据类型如下：

```
#define TH_FIN 0x01          //定义 FIN 标志
#define TH_SYN 0x02          //定义 SYN 标志
#define TH_RST 0x04          //定义 RST 标志
#define TH_PUSH 0x08         //定义 PUSH 标志
#define TH_ACK 0x10          //定义 ACK 标志
#define TH_URG 0x20          //定义 URG 标志
#define TCPOPT_EOL 0
#define TCPOPT_NOP 1
#define TCPOPT_MAXSEG 2
//定义 TCP 头的数据结构体
typedef struct _TCPHdr
{
    u_int16_t th_sport;        // 源端口
    u_int16_t th_dport;        // 目的端口
    u_int32_t th_seq;          // 序列号
    u_int32_t th_ack;          // 确认序号
#ifdef WORDS_BIGENDIAN
    u_int8_t th_off:4,         // 首部长度
    th_x2:4;                   // 保留位
#else
    u_int8_t th_x2:4,          // 保留位
    th_off:4;                   // 首部长度
#endif
    u_int8_t th_flags;
    u_int16_t th_win;           // 窗口大小
    u_int16_t th_sum;           // 检验和
    u_int16_t th_urp;           // 紧急指针
} TCPHdr;

#define EXTRACT_16BITS(p) ((u_int16_t) ntohs (*(u_int16_t *) (p)))
#define EXTRACT_32BITS(p) ((u_int32_t) ntohl (*(u_int32_t *) (p)))
#ifdef TCPOPT_WSCALE
#define TCPOPT_WSCALE 3 // 窗口参数
#endif
#ifdef TCPOPT_ECHO
#define TCPOPT_ECHO 6 // 回显
#endif
#ifdef TCPOPT_ECHOREPLY
#define TCPOPT_ECHOREPLY 7 // 回显
```

```

#endif

#ifndef TCPOPT_TIMESTAMP
#define TCPOPT_TIMESTAMP      8    // 事件戳
#endif

//用来存储 TCP 信息的数据结构体。
struct my_tcp_header_string
{
    char sport[1024];
    char dport[1024];
    char seq[1024];
    char ack[1024];
    char doff[1024];
    char flags[1024];
    char win[1024];
    char cksum[1024];
    char urp[1024];
    char options[1024];
    char information[1024];
};

//定义一个全局变量，用来存储 TCP 数据包信息，后面的入侵检测模块也要用到。
struct my_tcp_header_string tcp_header_string_object;

```

图 6-13 是分析 TCP 协议的运行界面。

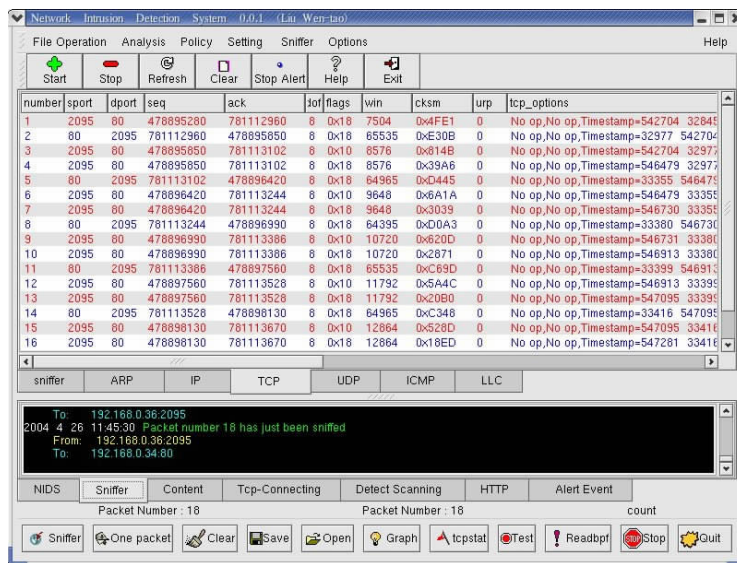


图 6-13 运行结果

TCP 协议分析主要由以下一个函数来完成，在上节中，我们看到当 `ip->ip_p` 是 `TCP_NEXT_HEADER` 的时候，就调用 `analysis_tcp` 函数，此函数就完成对 TCP 协议头的协议分析过程。


```

void analysis_tcp(u_char *bp, int length)
{
    TCPhdr *tp;
    u_char flags;
    int hlen, total_hlen;
    u_short sport, dport, win, urp;
    u_int seq, ack;
    tp = (TCPhdr *) bp;    //TCP 头数据结构
    printf("----- TCP Header-----\n");
    if (length < sizeof(TCPhdr))
    {
        printf("Truncated TCP header: %d bytes\n", length);
        return;
    }
    sport = ntohs(tp->th_sport);    //源端口
    dport = ntohs(tp->th_dport);    //目的端口
    hlen = tp->th_off * 4;    //首部长度
    seq = ntohl(tp->th_seq);    //32 位序列号
    ack = ntohl(tp->th_ack);    //32 位确认号
    win = ntohs(tp->th_win);    //16 位窗口大小
    urp = ntohs(tp->th_urp);    //紧急指针
    flags = tp->th_flags;    //标志位
    printf("Source port:          %d", sport);
    if (sport < 1024)
        printf(" (%s)\n", tcpport_string(sport));    //输出源端口
    else
        printf("\n");
    printf("Destination port:      %d", dport);    //输出目的端口
    if (dport < 1024)
        printf(" (%s)\n", tcpport_string(dport));
    else
        printf("\n");
    printf("Sequence number:        %u\n", seq);    //输出序列号
    printf("Acknowledgement number: %u\n", ack);    //输出确认号
    printf("Header length:          %d\n", hlen);    //输出首部长度
    printf("Reserved bits:          %d\n", tp->th_x2);    //输出保留字节
    printf("Flags:                  ");    //输出标志位
    if (flags & TH_SYN) printf("SYN ");    //标志位是否 TH_SYN
    if (flags & TH_FIN) printf("FIN ");    //标志位是否 TH_FIN
    if (flags & TH_RST) printf("RST ");    //标志位是否 TH_RST
    if (flags & TH_PUSH) printf("PSH ");    //标志位是否 TH_PUSH
    if (flags & TH_ACK) printf("ACK ");    //标志位是否 TH_ACK
}

```

```

if (flags & TH_URG) printf("URG ");      //标志位是否    TH_URG
printf("\n");
printf("Window size:          %d\n", win);    //输出窗口大小
printf("Checksum:            %d\n", ntohs(tp->th_sum));//输出校验位
printf("Urgent pointer:      %d\n", urp);//输出紧急指针
printf("Options:              ");
if (hlen > length)            //检查头长度
{
    printf("none\nBad header length\n");
    return;
}
length -= hlen;
total_hlen = hlen;
//处理各种选项
strcpy(tcp_header_string_object.options,"");
if ((hlen -= sizeof(TCPHdr)) > 0)
{
    u_char *cp;
    char string_local[1024];
    int i, opt, len, datalen;
    strcpy(string_local,"");
    cp = (u_char *) tp + sizeof(TCPHdr);
    while (hlen > 0)
    {
        opt = *cp++;
        if (opt == TCPOPT_EOL || opt == TCPOPT_NOP)
            len = 1;
        else
        {
            len = *cp++; // total including type, len
            if (len < 2 || len > hlen)
            {
                printf("Bad option\n");
                strcat(tcp_header_string_object.options,"Bad option");
                break;
            }
            hlen--;
            length--;
        } // else
        //计算类型字节
        hlen--;
        length--;
    }
}

```

```
//处理余下的选项
datalen = 0;
switch (opt)
{
    case TCPOPT_MAXSEG:
        datalen = 2;
        printf("Maximum segment size = ");
        printf("%u\n", EXTRACT_16BITS(cp));
        sprintf(string_local,"Max seg size = %u , ",
            EXTRACT_16BITS ( cp ) );
        //最大分片长度
        strcat(tcp_header_string_object.options, string_local);
        break;
    case TCPOPT_EOL:
        printf("End of options list\n");
        break;
    case TCPOPT_NOP:
        printf("No op\n");
        strcat(tcp_header_string_object.options,"No op,");
        break;
    case TCPOPT_WSCALE:
        datalen = 1;
        printf("Window scale = ");
        printf("%u\n", *cp);
        sprintf(string_local,"Window scale= %u,",*cp);
        strcat(tcp_header_string_object.options,string_local);
        break;
    case TCPOPT_ECHO: //回显
        datalen = 4;
        printf("Echo = ");
        printf("%u\n", EXTRACT_32BITS(cp));
        sprintf(string_local,"Echo=%u",EXTRACT_32BITS(cp));
        strcat(tcp_header_string_object.options,string_local);
        break;
    case TCPOPT_ECHOREPLY: //回显
        datalen = 4;
        printf("Echo reply = ");
        printf("%u\n", EXTRACT_32BITS(cp));
        sprintf(string_local,"Echo reply=%u",
            EXTRACT_32BITS(cp));
        strcat(tcp_header_string_object.options,string_local);
        break;
```

```

        case TCPOPT_TIMESTAMP: //事件戳
            datalen = 8;
            printf("Timestamp = ");
            printf("%u", EXTRACT_32BITS(cp));
            printf(" %u\n", EXTRACT_32BITS(cp + 4));
            sprintf(string_local, "Timestamp=%u %u ",
                EXTRACT_32BITS(cp), EXTRACT_32BITS(cp+4));
            strcat(tcp_header_string_object.options, string_local);
            break;
        default:
            datalen = len - 2;
            printf("Option (I don't know) %d:", opt);
            for (i = 0; i < datalen; ++i)
                printf("%02x\n", cp[i]);
            break;
    }
    cp += datalen;
    hlen -= datalen;
}
}
else
{
    printf("none\n");
    strcat(tcp_header_string_object.options, "none");
}
return;
}

```

分析 TCP 协议还有下面一个函数，此函数在 print_packet 函数里面被调用。

```

void
print_tcp_header (struct tcphdr *tcp, int len, int wtp)
{
    char indnt[64] = "    ";
    if (wtp & PP_SHOW_TCPHEADER)
    {
        printf ("%sTCP HEADER:\n", indnt);
        printf ("%ssport=%d dport=%d seq=0x%lX ack=0x%lX doff=0x%.2X\n",
            indnt,
            htons (tcp->th_sport),
            htons (tcp->th_dport),
            (unsigned long int) htonl (tcp->th_seq),
            (unsigned long int) htonl (tcp->th_ack), tcp->th_off);
    }
}

```

```
//把 TCP 信息存储到 tcp_header_string_object
sprintf (tcp_header_string_object.sport, "%d", htons (tcp->th_sport));
sprintf (tcp_header_string_object.dport, "%d", htons (tcp->th_dport));
sprintf (tcp_header_string_object.seq, "%u",
        (unsigned long int) htonl (tcp->th_seq));
sprintf (tcp_header_string_object.ack, "%u",
        (unsigned long int) htonl (tcp->th_ack));
sprintf (tcp_header_string_object.doff, "%d", tcp->th_off);
printf ("%sflags=(0x%X)", indnt, tcp->th_flags);
sprintf (tcp_header_string_object.flags, "0x%X", tcp->th_flags);
if (tcp->th_flags & TH_FIN)
{
    printf (",FIN");    //FIN 标志
}
if (tcp->th_flags & TH_SYN)
{
    printf (",SYN");    //SYN 标志
}
if (tcp->th_flags & TH_RST)
{
    printf (",RST");    //RST 标志
}
if (tcp->th_flags & TH_PUSH)
{
    printf (",PUSH");    //PUSH 标志
}
if (tcp->th_flags & TH_ACK)
{
    printf (",ACK");    //ACK 标志
}
if (tcp->th_flags & TH_URG)
{
    printf (",URG");    //URG 标志
}
printf (" win=%u cksm=0x%.4X urp=0x%.4X",
        htons (tcp->th_win), htons (tcp->th_sum), htons (tcp->th_urp));
sprintf (tcp_header_string_object.win, "%u", htons (tcp->th_win));
sprintf (tcp_header_string_object.cksm, "0x%.4X", htons (tcp->th_sum));
sprintf (tcp_header_string_object.urp, "%d", htons (tcp->th_urp));
gdk_threads_enter ();
add_list_to_clist3 (); //动态显示数据
gdk_threads_leave ();
```

```

        if (use_database_yesno == 1)
            insert_tcp_into_database ();
        //把 TCP 协议内容存放到数据库中
        printf ("\n");
    }
    if (1)
    {
        u_char *p;
        int i;
        strcpy (tcp_content_object, "");
        p = (char *) tcp;
        p += (tcp->th_off) * sizeof (int);
        len -= sizeof (struct tcphdr) + sizeof (int);
        // Options (sizeof(int)) is there for padding
        if (len > 0)
        {
            for (i = 0; i < len; i++)
            {
                printf ("%s", my_charconv (p[i]));
                strcat (tcp_content_object, my_charconv (p[i]));
            }
            printf ("\n");
        }
    }
}

```

6.2.6 UDP 协议分析

此部分实现 UDP 协议分析。用来描述 UDP 数据包的变量和数据类型如下：

```

#define L2TP_PORT 1701           //L2TP 协议端口
#define DHCP_CLIENT_PORT 68     //DHCP 客户端端口
#define DHCP_SERVER_PORT 67    //DHCP 服务器端口
#define SIP_PORT 5060           //SIP 协议端口
#define RIP_PORT 520            //RIP 协议端口
#define ISAKMP_PORT 500        //ISAKMP 端口
typedef struct _UDPHdr
{
    u_int16_t uh_src;           //源端口号
    u_int16_t uh_dst;           //目的端口号
    u_int16_t uh_len;           //UDP 长度
    u_int16_t uh_chk;           //UDP 检验和
} UDPHdr;

```

//用来存储 UDP 数据包信息的数据结构体。

```
struct my_udp_header_string
{
    char sport[1024];
    char dport[1024];
    char len[1024];
    char cksum[1024];
};
```

//定义一个全局变量，用来存储 UDP 数据包信息，后面的入侵检测模块也要用到。

```
struct my_udp_header_string udp_header_string_object;
```

图 6-14 是分析 UDP 协议的运行界面。

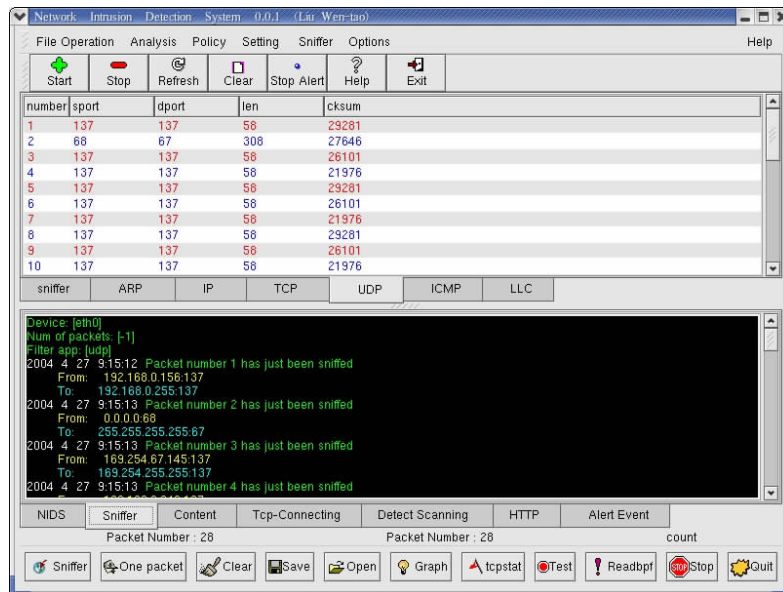


图 6-14 UDP 协议分析运行结果

在分析 IP 协议时，当 ip->ip_p 为 UDP_NEXT_HEADER 时，就分析 UDP 协议，调用 analysis_udp() 函数。其核心代码如下：

```
void analysis_udp(u_char *bp, int length)
{
    UDPhdr *up;
    u_char *ep = bp + length;
    u_short sport, dport, len;
    struct servent *se;
    char *udpport_string(u_short);
    if (ep > packet_end) ep = packet_end;
    if (length < sizeof(UDPhdr))
    {
        printf("Truncated header, length = %d bytes\n", length);
        return;
    }
```

```

    }
    up = (UDPHdr *) bp; //定位 UDP 协议头，强制转换为 UDP 数据结构
    sport = ntohs(up->uh_src);
    dport = ntohs(up->uh_dst);
    len = ntohs(up->uh_len);
    printf("----- UDP Header-----\n");
    printf("Source port:          %d", sport); //源端口号
    if (sport < 1024)
        printf(" (%s)\n", udpport_string(sport));
    else
        printf("\n");
    printf("Destination port:      %d", dport); //目的端口号
    if (dport < 1024)
        printf(" (%s)\n", udpport_string(dport));
    else
        printf("\n");
    printf("Length:      %d\n", len); //长度
    printf("Checksum:    %d\n", ntohs(up->uh_chk)); //校验和
    se = getservbyname("domain", "udp");
    if (se == NULL)
        syserr("can't get services entries");
    if (sport == ntohs(se->s_port) || dport == ntohs(se->s_port))
        analysis_dns((u_char *) bp + sizeof(UDPHdr), len);
        //分析 DNS 数据包 (参考 6.3.1 节)
    if (sport == L2TP_PORT || dport == L2TP_PORT)
        analysis_l2tp((u_char *) bp + sizeof(UDPHdr), len);
        //分析 L2TP 数据包 (本书不作介绍)
    if (sport == DHCP_CLIENT_PORT || dport == DHCP_CLIENT_PORT)
        analysis_dhcp((u_char *) bp + sizeof(UDPHdr), len);
        //分析 DHCP 数据包 (参考 6.3.2 节)
    if (sport == RIP_PORT || dport == RIP_PORT)
        analysis_rip((u_char *) bp + sizeof(UDPHdr), len);
        //分析 RIP 数据包 (本书不作介绍)
    if (sport == SIP_PORT || dport == SIP_PORT)
        analysis_sip((u_char *) bp + sizeof(UDPHdr), len);
        //分析 SIP 数据包 (本书不作介绍)
    if (sport == ISAKMP_PORT || dport == ISAKMP_PORT)
        analysis_isakmp((u_char *) bp + sizeof(UDPHdr), len);
        //分析 IKE 数据包 (本书不作介绍)
    analysis_payload((u_char *) bp + sizeof(UDPHdr), len);
    //分析数据内容
    return;
}

```


分析 UDP 协议还有下面一个函数，此函数在 print_packet 函数里面被调用。

```
void
print_udp_header (struct udphdr *udp, int len, int wtp)
{
    if (wtp & PP_SHOW_UDPHEADER)
    {
        printf ("\tUDP HEADER:\n");
        printf ("\t\t sport=%d dport=%d len=%d cksum=0x%.2X\n",
            htons (udp->uh_sport),
            htons (udp->uh_dport),
            htons (udp->uh_ulen), htons (udp->uh_sum));
        //把 UDP 协议信息存储到 udp_header_string_object
        sprintf (udp_header_string_object.sport, "%d", htons (udp->uh_sport));
        sprintf (udp_header_string_object.dport, "%d", htons (udp->uh_dport));
        sprintf (udp_header_string_object.len, "%d", htons (udp->uh_ulen));
        sprintf (udp_header_string_object.cksum, "%d", htons (udp->uh_sum));
        gdk_threads_enter ();
        add_list_to_clist4 ();
        //把数据动态显示到 UDP 页面控件中
        gdk_threads_leave ();
        if (use_database_yesno == 1)
            insert_udp_into_database ();
        //把 UDP 数据包内容存放到数据库中
    }
    if (1)
    {
        u_char *p;
        int i;
        strcpy (udp_content_object, "");
        p = (char *) udp;
        p += sizeof (struct udphdr);
        len -= sizeof (struct udphdr);
        for (i = 0; i < len; i++)
        {
            printf ("%s", my_charconv (p[i]));
            strcat (udp_content_object, my_charconv (p[i]));
        }
        printf ("\n");
    }
}
```

6.2.7 ICMP 协议分析

此部分实现 ICMP 协议分析。用来描述 ICMP 数据包的变量和数据类型如下：

```
typedef struct _ICMPHdr
{
    u_int8_t type;           //类型
    u_int8_t code;           //代码
    u_int16_t csum;          //校验和
} ICMPHdr;
typedef struct _echoext      //回显格式
{
    u_int16_t id;
    u_int16_t seqno;
} echoext;
//用来存储 ICMP 信息的数据结构体。
struct my_icmp_header_string
{
    char type[1024];
    char code[1024];
    char cksum[1024];
    char information[1024];
    struct my_echo_string
    {
        char id[1024];
        char sequence[1024];
    } echo_string_object;
};
//定义一个全局变量，用来存储 ICMP 信息，后面的入侵检测模块也要用到。
struct my_icmp_header_string icmp_header_string_object;
```

从上面可以看出在分析 IP 协议时，当 ip->ip_p 为 ICMP_NEXT_HEADER 时，就分析 ICMP 协议，调用 analysis_icmp() 函数。其核心代码如下：

```
void analysis_icmp(u_char *bp)
{
    ICMPHdr *icmph;
    echoext *echo;
    icmph = (ICMPHdr *) bp;
    printf("Type:  %d ", icmph->type); //ICMP 协议的类型
    switch (icmph->type)
    {
        case 0:
            printf("(echo reply)\n"); //echo 应答
```

```
printf("Code          %d\n", icmp->code);
echo = (echoext *) (bp + sizeof(ICMPHdr));
printf("Identifier:   %d\n", echo->id);
printf("Sequence number : %d\n", echo->seqno);
break;
case 3:
printf("(dest unreachable)\n"); //目的不可达
printf("Code:          %d ", icmp->code);
switch (icmp->code) //判断代码类型
{
    case 0:
        printf("(network)\n"); //网络
        break;
    case 1:
        printf("(host)\n"); //主机
        break;
    case 2:
        printf("(protocol)\n"); //协议
        break;
    case 3:
        printf("(port)\n"); //端口
        break;
    case 4:
        printf("(fragment needed)\n"); //碎片
        break;
    case 5:
        printf("(source route failed)\n"); //源路由失败
        break;
    case 6:
        printf("(network unknown)\n"); //网络不可知
        break;
    case 7:
        printf("(host unknown)\n"); //主机不可知
        break;
    case 9:
        printf("(network prohibited)\n"); //网络禁止
        break;
    case 10:
        printf("(host prohibited)\n"); //主机禁止
        break;
    case 11:
        printf("(network unreachable for TOS)\n");
```

```
        //对 TOS 网络不可达
        break;
    case 12:
        printf("(host unreachable for TOS)\n");
        //对 TOS 主机不可达
        break;
    case 13:
        printf("(communication filtered)\n");    //通信过滤
        break;
    case 14:
        printf("(host precedence violation)\n");
        //违反主机优先
        break;
    case 15:
        printf("(precedence cutoff)\n");          //优先中止
        break;
    default:
        printf("(unknown code)\n");
        //代码不可知
        break;
} // code
break;
case 4:
    printf("(source quench)\n");
    printf("Code: %d ", icmph->code);
    break;
case 5:
    printf("(redirect)\n");          //重定向
    printf("Code: %d ", icmph->code);
    switch(icmph->code)
    {
        case 0:
            printf("(network)\n");          //网络
            break;
        case 1:
            printf("(host)\n");              //主机
            break;
        case 2:
            printf("(TOS and network)\n");    //TOS 和网络
            break;
        case 3:
            printf("(TOS and host)\n");        //TOS 和主机
```

```
        break;
    }
    break;
case 8:
    printf("(echo request)\n");           //回显查询
    printf("Code   %d\n", icmp->code);
    echo = (echoext *) (bp + sizeof(ICMPHdr));
    printf("Identifier          %d\n", echo->id);
    printf("Sequence number      %d\n", echo->seqno);
    break;
case 9:
    printf("(router advertisement)\n");    //路由通告
    printf("Code:                %d ", icmp->code);
    break;
case 10:
    printf("(router solicitation)\n");     //路由恳求
    printf("Code:                %d ", icmp->code);
    break;
case 11:
    printf("(time exceeded)\n");           //时间超出
    printf("Code:                %d ", icmp->code);
    break;
case 12:
    printf("(parameter error)\n");         //参数错误
    printf("Code:                %d ", icmp->code);
    break;
case 13:
    printf("(timestamp request)\n");       //时间戳查询
    printf("Code:                %d ", icmp->code);
    break;
case 14:
    printf("(timestamp reply)\n");         //时间戳应答
    printf("Code:                %d ", icmp->code);
    break;
case 17:
    printf("(address mask request)\n");     //地址掩码查询
    printf("Code:                %d ", icmp->code);
    break;
case 18:
    printf("(address mask reply)\n");       //地址掩码应答
    printf("Code:                %d ", icmp->code);
    break;
```

```

default:
    printf("(unknown type)\n");    //类型不可知
    printf("Code: %d ", icmph->code);
    break;
}
printf("Checksum: %d\n", ntohs(icmph->csum));
return;
}

```

图 6-15 是分析 ICMP 协议的运行界面。

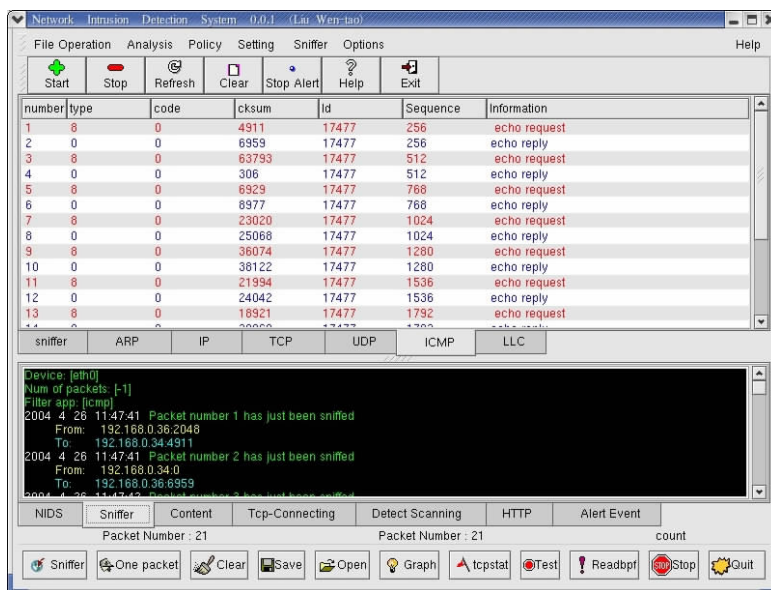


图 6-15 分析 ICMP 协议的运行结果

分析 ICMP 协议还有下面一个函数，此函数在 print_packet 函数里面被调用。

```

void
print_icmp_header (struct icmp *icmp, int len, int wtp)
{
    if (wtp & PP_SHOW_ICMPHEADER)
    {
        printf ("tICMP HEADER:\n");
        printf ("t\ttype=0x%X code=0x%X cksum=0x%.4X\n",
            icmp->icmp_type, icmp->icmp_code, htons (icmp->icmp_cksum));
        //把 ICMP 信息存储到 icmp_header_string_object
        sprintf (icmp_header_string_object.type, "%d", icmp->icmp_type);
        sprintf (icmp_header_string_object.code, "%d", icmp->icmp_code);
        sprintf (icmp_header_string_object.cksum, "%d",
            htons (icmp->icmp_cksum));
        gdk_threads_enter ();
    }
}

```

```

        add_list_to_clist5 ();
        //把 ICMP 信息动态显示到 ICMP 页面控件中
        gdk_threads_leave ();
        if (use_database_yesno == 1)
            insert_icmp_into_database ();
        //把 ICMP 协议内容存放到数据库中
    }
    if (1)
    {
        u_char *p;
        int i;
        strcpy (icmp_content_object, "");
        p = (char *) icmp;
        p += sizeof (struct icmp);
        len -= sizeof (struct icmp);
        for (i = 0; i < len; i++)
        {
            printf ("%s", my_charconv (p[i]));
            strcat (icmp_content_object, my_charconv (p[i]));
        }
        printf ("\n");
    }
}

```

6.3 其他协议的分析

6.3.1 DNS 协议

DNS (Domain Name System) 是寻找 Internet 域名并将它转化为 IP 地址的系统。域名是有意义的、容易记忆的 Internet 地址。因为支持一个集中式的域名列表是不现实的，域名和 IP 地址是分布式存放的。您的请求首先到达地理上比较近的主机，如果寻找不到此域名，主机将您的请求向远方的主机发送。如果找到相应的地址就返回给请求方；如果找不到则返回错误信息给请求方。

DNS 是一个分布式的数据库，它是为了定义 Internet 上的主机而提供的一个层次性的命名系统。利用 DNS 能进行域名的解析，域名解析过程包括：

DNS 客户向本地的 DNS 服务器发出查询请求。

如果该 DNS 本身具有客户想要查询的数据，则直接返回给客户，如果没有，则该服务器和其他命名服务器联系，从其他服务器上获取信息，然后返回给用户。

最不妙的情况是，本地的 DNS 服务器查询了所有其他的命名服务器，没有获得用户要查询的信息。

DNS 报文格式由 12 个字节首部和 4 个长度可变的字段组成，如图 6-16 所示。

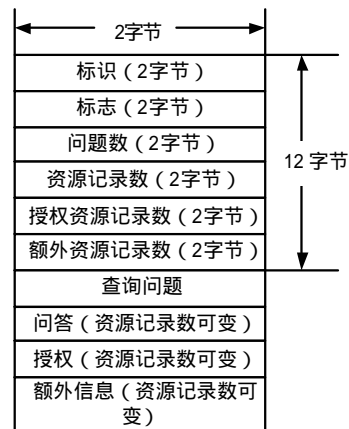


图 6-16 DNS 格式

标识字段由客户程序设置并由路由器返回结果。客户程序通过它来确定响应与查询是否匹配。

标志字段被划分为若干子字段，如图 6-17 所示。

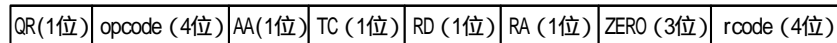


图 6-17 标志字段格式

标志字段格式说明如下。

QR：1 位，0 表示查询，1 表示响应报文。

Opcode：4 位，0 表示标准查询，1 表示反向查询，2 表示服务器状态请求。

AA：1 位，表示授权回答。

TC：1 位，表示可截断的，使用 UDP 时，它表示当的总长度超过 512 字节时，只返回前 512 个字节。

RD：1 位，表示期望递归。该位能在一个查询中设置，并在响应中返回。这个标志告诉名字服务器必须处理这个查询。如果该位为 0，且被请求的名字服务器没有一个授权回答，它就返回一个能解答该查询的其他名字服务器列表，这称为叠代查询。

RA：1 位，表示可用递归。如果名字服务器支持递归查询，则在响应中将该位置为 1。

ZERO：3 位，必须为 0。

Rcode：4 位返回码，0 表示没有差错，3 表示名字差错。名字差错只有从一个授权服务器上返回，它表示在查询中指定的域名不存在。

随后的 4 个 16 位的字段说明最后 4 个变长字段中包含的条目数。对于查询报文，问题数通常是 1，而其他 3 项均为 0。对于应答报文，问答数至少为 1，剩下的 2 项可以是 0 或非 0。

有关 DNS 协议更详细的介绍可参考其他资料。

实现 DNS 协议分析的相关变量和数据结构定义如下：

```
typedef struct _DNSHdr
{
    u_int16_t dns_id;
    #if defined (WORDS_BIGENDIAN)
```



```

u_int16_t dns_fl_qr:1,      //0 表示查询, 1 表示响应报文。
dns_fl_opcode:4,  //0 表示标准查询, 1 表示反向查询, 2 表示服务器状态请求。
dns_fl_aa:1,      //表示授权回
dns_fl_tc:1,      //表示可截断的
dns_fl_rd:1,      //表示期望递归
dns_fl_ra:1,      //表示可用递归
dns_fl_zero:3,     //全部为 0
dns_fl_rcode:4;    //返回码
#else
u_int16_t dns_fl_rcode:4,
dns_fl_zero:3,
dns_fl_ra:1,
dns_fl_rd:1,
dns_fl_tc:1,
dns_fl_aa:1,
dns_fl_opcode:4,
dns_fl_qr:1;
#endif
u_int16_t dns_num_q;
u_int16_t dns_num_ans;
u_int16_t dns_num_auth;
u_int16_t dns_num_add;
} DNSHdr;

```

实现 DNS 协议分析的函数实现如下：

```

void analysis_dns(u_char *bp, int length)
{
    u_char *ep = bp + length;
    u_char *p;
    DNSHdr dns_fixed;
    int i, t;
    u_int16_t qt, qc;
    char *dns_query_type(u_int16_t);
    u_char *analysis_rr(u_char *, u_char *, u_char *);
    u_char *parse_labels(u_char *, u_char *, u_char *);
#ifdef DEBUG
    void analysis_ascii(u_char *, u_char *);
#endif
    if (ep > packet_end)
        ep = packet_end;
    memcpy((void *) &dns_fixed, bp, sizeof(dns_fixed));
    printf(" -----DNS Packet-----\n");
}

```

```

printf("Identification:      %d\n", ntohs(dns_fixed.dns_id));
printf("Flags: query/response: %d ", dns_fixed.dns_fl_qr);
if (dns_fixed.dns_fl_qr == 0)
    printf("(query)\n");
else
    printf("(response)\n");
printf("      opcode          %d ", dns_fixed.dns_fl_opcode);
switch(dns_fixed.dns_fl_opcode)
{
    case 0: printf("(standard)\n");
            break;
    case 1: printf("(inverse)\n");
            break;
    case 2: printf("(server status)\n");
            break;
}
printf("auth answer      %d\n", dns_fixed.dns_fl_aa);
printf("truncated        %d\n", dns_fixed.dns_fl_tc);
printf("recursion req     %d\n", dns_fixed.dns_fl_rd);
printf("recursion avail   %d\n", dns_fixed.dns_fl_ra);
printf("zero              %d\n", dns_fixed.dns_fl_zero);
printf("return code       %d ", dns_fixed.dns_fl_rcode);
switch(dns_fixed.dns_fl_rcode)
{
    case 0: printf("(no error)\n");
            break;
    case 1: printf("(format error)\n");
            break;
    case 2: printf("(server error)\n");
            break;
    case 3: printf("(name error)\n");
            break;
    case 4: printf("(not implemented)\n");
            break;
    case 5: printf("(service refused)\n");
            break;
}
printf("# of questions      %d\n", ntohs(dns_fixed.dns_num_q));
printf("# of answer RRs       %d\n", ntohs(dns_fixed.dns_num_ans));
printf("# of authorization RRs %d\n", ntohs(dns_fixed.dns_num_auth));
printf("# of additional RRs    %d\n", ntohs(dns_fixed.dns_num_add));
p = bp + sizeof(DNSHdr);

```

```
i = ntohs(dns_fixed.dns_num_q);
while (i > 0)
{
    printf("Question: query name      ");
    p = parse_labels(p, bp, ep);
    memcpy((void *) &qt, p, sizeof(qt));
    p = p + sizeof(qt);
    memcpy((void *) &qc, p, sizeof(qc));
    p = p + sizeof(qc);
    printf("query type      %d %s\n", ntohs(qt),
dns_query_type(ntohs(qt)));
    printf("query class    %d", ntohs(qc));
    if (ntohs(qc) == 1)
        printf(" (Internet)");
    printf("\n");
    i--;
}
//分析应答的资源记录
i = ntohs(dns_fixed.dns_num_ans);
t = i;
while (i > 0)
{
    printf("Answer %d: ", t-i+1);
    p = analysis_rr(p, bp, ep);
    i--;
}
//分析命令应答的资源记录
i = ntohs(dns_fixed.dns_num_auth);
t = i;
while (i > 0)
{
    printf("Auth %d:      ", t-i+1);
    p = analysis_rr(p, bp, ep);
    i--;
}
//分析附加信息的资源记录
i = ntohs(dns_fixed.dns_num_add);
t = i;
while (i > 0)
{
    printf("Adtnl %d:   ", t-i+1);
    #ifdef DEBUG
```

```

        analysis_ascii(p, ep);
    #endif
    p = analysis_rr(p, bp, ep);
    i--;
}
}

```

此函数中用到下面三个函数。

```

char *dns_query_type(u_int16_t t)
//此函数是返回一个 DNS 查询类型值的文本描述。
{
    static char answer[32];
    switch(t)
    {
        case 1: strncpy(answer, "(IP address)", 32);
                break;
        case 2: strncpy(answer, "(name server)", 32);
                break;
        case 5: strncpy(answer, "(canonical name)", 32);
                break;
        case 12: strncpy(answer, "(pointer record)", 32);
                break;
        case 13: strncpy(answer, "(host info)", 32);
                break;
        case 15: strncpy(answer, "(MX record)", 32);
                break;
        case 252: strncpy(answer, "(request zone transfer)", 32);
                break;
        case 255: strncpy(answer, "(request all records)", 32);
                break;
        default: answer[0] = '\0';
                break;
    }
    return answer;
}

u_char *analysis_rr(u_char *p, u_char *bp, u_char *ep)
{
    //此函数是输出资源记录的内容
    int i;
    u_int16_t qt, qc;
    u_int32_t ttl;
    u_int16_t reslen;
    struct in_addr ipaddr;

```

```

u_char *parse_labels(u_char *, u_char *, u_char *);
printf("server name ");
p = parse_labels(p, bp, ep);
memcpy((void *) &qt, p, sizeof(qt));
p = p + sizeof(qt);
memcpy((void *) &qc, p, sizeof(qc));
p = p + sizeof(qc);
printf("          type          %d %s\n", ntohs(qt),
dns_query_type(ntohs(qt)));
printf("          class          %d", ntohs(qc));
if (ntohs(qc) == 1)
    printf(" (Internet)");
printf("\n");
//TTL
memcpy((void *) &ttd, p, sizeof(ttd));
p = p + sizeof(ttd);
printf("          ttl          %d seconds\n", ntohs(ttd));
//资源长度
memcpy((void *) &reslen, p, sizeof(reslen));
p = p + sizeof(reslen);
printf("          length          %d\n", ntohs(reslen));
//资源数据
switch(ntohs(qt))
{
    case 1:
        for (i=1; i<= ntohs(reslen); i += 4)
        {
            memcpy((void *) &ipaddr, p, sizeof(ipaddr));
            p = p + sizeof(ipaddr);
            printf("          IP address      %s\n", inet_ntoa(ipaddr));
        }
        break;
    case 2:
        printf("          auth host      ");
        p = parse_labels(p, bp, ep);
        break;
    case 5:
        printf("          canon host      ");
        p = parse_labels(p, bp, ep);
        break;
    default:
        p = p + ntohs(reslen);
}

```

```
        return p;
    }
    u_char *parse_labels(u_char *p, u_char *bp, u_char *ep)
    {
        //分析在一个 DNS 数据包中的标记
        u_int8_t count;
        u_int16_t offset;
        while(1)
        {
            count = (u_int8_t) *p;
            if (count >= 192)
            {
                p++;
                offset = count - 192;
                offset = offset << 8;
                offset = offset + *p;
                p++;
                parse_labels(bp+offset, bp, ep);
                return p;
            }
            else
                p++;
            if (count == 0)
                break;
            while (count > 0)
            {
                if (p <= ep)
                    printf("%c", *p);
                else
                {
                    printf("\nPacket length exceeded\n");
                    return p;
                }
                p++;
                count--;
            }
            printf(".");
        }
        printf("\n");
        return p;
    }
}
```

6.3.2 DHCP 协议

DHCP (Dynamic Host Configuration Protocol) 是动态主机配置协议,为因特网主机提供配置参数,它包括两个部分,一个从 DHCP 服务器向主机传输主机配置参数的协议和一个向主机分配网络地址的机制。其设计的目的是为了减轻 TCP/IP 网络的规划、管理和维护的负担,解决 IP 地址空间缺乏问题。配置 DHCP 协议的服务器把 TCP/IP 网络设置集中起来,动态处理工作站 IP 地址的配置,用 DHCP 租约和预置的 IP 地址相联系,DHCP 租约提供了自动在 TCP/IP 网络上安全地分配和租用 IP 地址的机制,实现 IP 地址的集中式管理,基本上不需要网络管理人员的人为干预。而且,DHCP 本身被设计成 BOOTP (自举协议)的扩展,支持需要网络配置信息的无盘工作站,对需要固定 IP 的系统也提供了相应支持。这个系统中,DHCP 客户是一通过 DHCP 来获得网络配置参数的 Internet 主机,通常就是普通用户的工作站。DHCP 服务器是提供网络设置参数给 DHCP 客户的 Internet 主机。DHCP/BOOTP 中继代理是在 DHCP 客户和服务器之间转发 DHCP 消息的主机或路由器。DHCP 是基于客户机/服务器模型设计的,DHCP 客户和 DHCP 服务器之间通过收发 DHCP 消息进行通信。DHCP 消息的格式与 BOOTP 消息大部分相同,这样设计可以增强 BOOTP 服务器工具,同时为 BOOTP 和 DHCP 两种客户服务。另外,BOOTP 的中继代理可用来转发跨子网的 DHCP 请求。



图 6-18 DHCP 协议格式

DHCP 协议格式如图 6-18 所示。各字段描述如下。

op: 消息操作代码, 值为 1 代表自举请求 (BOOTREQUEST), 值为 2 代表自举响应 (BOOTREPLY)。在 DHCP 客户和 DHCP 服务器对话期间, op 段被 DHCP 客户设置为 1, 被 DHCP 服务器设置为 2。

htype: 硬件地址类型。

hlen: 硬件地址长度。

hops: DHCP 客户置为零。

xid: DHCP 客户在寻求时产生的一个随机数, 它提供了对所有后续的 DHCP 消息中的客户请求和服务器响应的一种联合。

ciaddr: 客户机用来请求一个特定的 IP 地址, 这个地址以前曾经分配给该客户机, 希望保留。

yiaddr: 由 DHCP 服务器填写, 包含它提供给某一 DHCP 客户的 IP 地址。

siaddr: 服务器的主机地址。

giaddr: 中继代理的 IP 地址。

chaddr: DHCP 客户硬件地址。

sname: 服务器主机名。

file：启动文件名。

options：选项。

在获得 IP 地址前，DHCP 客户用 htype, hlen 和 chaddr 段表明它的硬件地址，这个值由向客户硬件地址作出响应的服务器和中继代理利用。以前 BOOTP 协议中的两个没有用到的 8 位组的 flags 段在 DHCP 消息里有了定义。这个段的高位比特用于表明客户机能不能在 IP 地址没有被配置前接收 Unicast 回应，剩下的低位比特保留且必须置为零。hops 和 secs 段在初始化过程中被中继代理有选择地利用。sname 和 file 域可以被 BOOTP 或无盘站利用。选项附加在 DHCP 消息的固定长度段之后，为了与 BOOTP 工具兼容，选项段的前四个 8 位组包含了 RFC1497 中定义的 magic cookies，余下的段就都是 DHCP 选项。在 RFC1533 里定义了 DHCP 的所有选项的格式。大多数选项用于标志网络传输设置值，例如子网掩码（mask）、DNS 服务器地址等其他选项被 DHCP 协议利用，且在大多数消息中是必需的。选项可以固定长度或可变长度，所有的选项都以一个 8 位组标识码开始，这个标识码用来标识选项。不带数据的固定长度选项就只由一个标识码构成。而且只有选项 0 和 255 是固定长度，其他的选项都可变长度的，为了标明选项数据的长度，在标识码后面是一个长度 8 位组，这个长度 8 位组的值不包含标识码和长度码本身。

DHCP 消息类型选项的标识码是 53，长度是 1 个 8 位组，值是从 1 到 7，分别代表不同的 DHCP 消息类型。从大到小，依次表示类型为 DHCPDISCOVER、DHCP OFFER、DHCPREQUEST、DHCPDECLINE、DHCPACK、DHCPNAK 和 DHCPRELEASE。

最后一项选项是零长度的 End，选项为 255，表明这是选项的结束以便 DHCP 客户处理。采用选项编码的好处是不论选项有多长，DHCP 客户都可以正确接收，即使是它不认识的选项。不论是 DHCP 客户还是 DHCP 服务器，都是通过按 DHCP 消息格式要求来填写各个段形成具体的 DHCP 消息，DHCP 用的传输协议的非面向连接的 UDP（用户数据报协议），从 DHCP 客户发出的 DHCP 消息送往 DHCP 服务器的端口 67，DHCP 服务器发给客户的 DHCP 消息送往 DHCP 客户的端口 68，由于在取得服务器赋予的 IP 之前，DHCP 客户并没有自己的 IP，所以包含 DHCP 消息的 UDP 数据报的 IP 头的源地址段是 0.0.0.0，目的地址则是 255.255.255.255。

分析 DHCP 用到的辅助变量和数据结构定义如下：

```
#define DHCP_BOOTREQUEST 1
#define DHCP_BOOTREPLY 2
#define BOOTP_COOKIE      0x63825363
#define DHCP_OPT_PAD      0
#define DHCP_OPT_NETMASK  1
#define DHCP_OPT_TIMEOFFSET 2
#define DHCP_OPT_ROUTER   3
#define DHCP_OPT_TIMESERVER 4
#define DHCP_OPT_NAMESERVER 5
#define DHCP_OPT_DNS      6
#define DHCP_OPT_LOGSERVER 7
#define DHCP_OPT_COOKIESESERVER 8
```

#define DHCP_OPT_LPRSERVER	9
#define DHCP_OPT_IMPRESSSERVER	10
#define DHCP_OPT_RESLOCSERVER	11
#define DHCP_OPT_HOSTNAME	12
#define DHCP_OPT_BOOTFILESIZE	13
#define DHCP_OPT_MERITDUMP	14
#define DHCP_OPT_DOMAINNAME	15
#define DHCP_OPT_SWAPSERVER	16
#define DHCP_OPT_ROOTPATH	17
#define DHCP_OPT_EXTSPATH	18
#define DHCP_OPT_IPFORWARD	19
#define DHCP_OPT_NONLOCALSR	20
#define DHCP_OPT_POLICYFILTER	21
#define DHCP_OPT_MAXREASSEMBLE	22
#define DHCP_OPT_IPTTL	23
#define DHCP_OPT_PATHMTUAGING	24
#define DHCP_OPT_PATHMTUPLATEAU	25
#define DHCP_OPT_INTERFACEMTU	26
#define DHCP_OPT_SUBNETSLOCAL	27
#define DHCP_OPT_BCASTADDRESS	28
#define DHCP_OPT_MASKDISCOVERY	29
#define DHCP_OPT_MASKSUPPLIER	30
#define DHCP_OPT_ROUTERDISCOVERY	31
#define DHCP_OPT_ROUTERSOLIC	32
#define DHCP_OPT_STATICROUTE	33
#define DHCP_OPT_TRAILERENCAPS	34
#define DHCP_OPT_ARPTIMEOUT	35
#define DHCP_OPT_ETHERNETENCAPS	36
#define DHCP_OPT_TCPCTL	37
#define DHCP_OPT_TCPKEEPALIVEINT	38
#define DHCP_OPT_TCPKEEPALIVEGRBG	39
#define DHCP_OPT_NISDOMAIN	40
#define DHCP_OPT_NISSERVERS	41
#define DHCP_OPT_NTPSERVERS	42
#define DHCP_OPT_VENDORSPECIFIC	43
#define DHCP_OPT_NETBIOSNAMESERV	44
#define DHCP_OPT_NETBIOSDGDIST	45
#define DHCP_OPT_NETBIOSNODETYPE	46
#define DHCP_OPT_NETBIOSSCOPE	47
#define DHCP_OPT_X11FONTS	48
#define DHCP_OPT_X11DISPLAYMNGR	49

```

#define DHCP_OPT_REQUESTEDIPADDR    50
#define DHCP_OPT_IPADDRLEASE        51
#define DHCP_OPT_OVERLOAD            52
#define DHCP_OPT_MESSAGE_TYPE        53
#define DHCP_OPT_SERVERID            54
#define DHCP_OPT_PARAMREQLIST        55
#define DHCP_OPT_MESSAGE              56
#define DHCP_OPT_MAXDHCPMSGSIZE      57
#define DHCP_OPT_RENEWALTIME          58
#define DHCP_OPT_REBINDINGTIME        59
#define DHCP_OPT_VENDORCLASSID        60
#define DHCP_OPT_CLIENTID             61
#define DHCP_OPT_NISPLUSDOMAIN        64
#define DHCP_OPT_NISPLUSSERVERS      65
#define DHCP_OPT_TFTP_SERVER          66
#define DHCP_OPT_BOOTFILE             67
#define DHCP_OPT_MOBILEIPHOME        68
#define DHCP_OPT_SMTP_SERVER          69
#define DHCP_OPT_POP3_SERVER          70
#define DHCP_OPT_NNTP_SERVER          71
#define DHCP_OPT_WWW_SERVER           72
#define DHCP_OPT_FINGER_SERVER        73
#define DHCP_OPT_IRC_SERVER           74
#define DHCP_OPT_ST_SERVER            75
#define DHCP_OPT_STDASERVER           76
#define DHCP_OPT_END                  255
#define DHCP_MAX_HOSTNAME             64
//定义 DHCP 头的数据结构体
typedef struct _DHCPHdr
{
    #if defined(WORDS_BIGENDIAN)
        u_int8_t hops;    //DHCP 客户为 0
        u_int8_t hlen;     //硬件地址长度
        u_int8_t htype;    //硬件地址类型
        u_int8_t op;       //消息操作代码
    #else
        u_int8_t op;
        u_int8_t htype;
        u_int8_t hlen;
        u_int8_t hops;
    #endif

```

```

        u_int32_t xid;           //随机数
        u_int16_t secs;         //时间
        u_int16_t flags;        // 标志
        u_int32_t ciaddr;       //IP 地址
        u_int32_t yiaddr;       //IP 地址
        u_int32_t siaddr;       // 服务器的主机 IP 地址
        u_int32_t giaddr;       //中继代理的 IP 地址
        char chaddr [16];       //DHCP 客户硬件地址
        char sname [64];        //服务器主机名
        char file [128];        //启动文件名
    } DHCPHdr;
DHCP 协议的分析实现由下面一个函数来完成。
void analysis_dhcp(u_char *bp, int length)
{
    u_char *ep = bp + length;
    DHCPHdr *dhcp;
    int end_pad_cnt = 0;
    int i;
    int c;
    int j;
    int opt;
    int done = 0;
    u_int16_t s;
    char hostname [DHCP_MAX_HOSTNAME];
    struct in_addr in_holder;
    if (ep > packet_end)
        ep = packet_end;
    printf("-----DHCP Header-----\n");
    dhcp = (DHCPHdr *) bp;
    printf("OP                %d ", (int) dhcp->op);
    switch (dhcp->op)
    {
        case DHCP_BOOTREQUEST:    //启动请求
            printf("(boot request)\n");
            break;
        case DHCP_BOOTREPLY:      //启动应答
            printf("(boot reply)\n");
            break;
        default:
            printf("(unknown)\n");
    }
}

```

```

printf("Hardware addr type:  %d\n", (int) dhcp->htype);    //硬件地址类型
printf("Hardware addr length:  %d\n", (int) dhcp->hlen);    //硬件地址长度
printf("Hops:      %d\n", (int) dhcp->hops);                //跳
printf("Transaction ID: %x\n", ntohl(dhcp->xid));          //ID
printf("Seconds: %d\n", ntohs(dhcp->secs));                //时间
printf("Flags:    %x\n", ntohs(dhcp->flags));              //标志
memcpy((void *) &in_holder, (void *) &dhcp->ciaddr, sizeof(struct in_addr));
printf("Client addr:  %s\n", inet_ntoa(in_holder));        //客户地址
memcpy((void *) &in_holder, (void *) &dhcp->yiaddr, sizeof(struct in_addr));
printf("Your addr:   %s\n", inet_ntoa(in_holder));        //本身地址
memcpy((void *) &in_holder, (void *) &dhcp->siaddr, sizeof(struct in_addr));
printf("Next server addr: %s\n", inet_ntoa(in_holder));    //下一个服务器地址
memcpy((void *) &in_holder, (void *) &dhcp->giaddr, sizeof(struct in_addr));
printf("Relay agent addr: %s\n", inet_ntoa(in_holder));    //代理地址
printf("Client hardware addr:  \n");
printf("Server host name: %s\n", dhcp->sname);              //服务器主机名字
printf("Boot file name:  %s\n", dhcp->file);                //引导文件名字
bp = bp + sizeof(DHCPHdr);
i = htonl(BOOTP_COOKIE);
memcpy((void *) &c, bp, 4);
if (i == c)
{
    printf("BOOTP Cookie:  %x\n", i);
    bp = bp + sizeof(BOOTP_COOKIE);
}
while (bp < ep)                //分析选项
{
    int len=0;
    opt = (int) *bp;
    if (opt > 0 && opt < 255)
        len = *++bp;
    if (!done)
        printf("Option:      %d ", opt);
    bp++;
    switch(opt)
    {
        //判断选项类型
        case DHCP_OPT_PAD:
            if (!done)
                printf("(pad)\n");
            else

```

```
        end_pad_cnt++;
    break;
case DHCP_OPT_NETMASK:    //网络掩码
    printf("(netmask)\n");
    printf("  Length:      %d\n", len);
    memcpy((void *) &in_holder, bp, sizeof(struct in_addr));
    printf("  Mask:          %s\n", inet_ntoa(in_holder));
    bp = bp + sizeof(struct in_addr);
    break;
case DHCP_OPT_TIMEOFFSET: //偏移
    printf("(time offset)\n");
    printf("  Length:      %d\n", len);
    memcpy((void *) &i, bp, sizeof(int));
    printf("  Offset:    %d\n", ntohl(i));
    bp = bp + sizeof(u_int32_t);
    break;
case DHCP_OPT_ROUTER:     //路由地址
    printf("(routers)\n");
    printf("  Length:      %d\n", len);
    j = 0;
    while (j < len / 4)
    {
        memcpy((void *) &in_holder, bp, sizeof(struct in_addr));
        printf("  Address:    %s\n", inet_ntoa(in_holder));
        bp = bp + sizeof(struct in_addr);
        j++;
    }
    break;
case DHCP_OPT_DNS:        //DNS 地址
    printf("(DNS)\n");
    printf("  Length:      %d\n", len);
    j = 0;
    while (j < len / 4)
    {
        memcpy((void *) &in_holder, bp, sizeof(struct in_addr));
        printf("  Address: %s\n", inet_ntoa(in_holder));
        bp = bp + sizeof(struct in_addr);
        j++;
    }
    break;
case DHCP_OPT_HOSTNAME:   //主机名字
```

```

    printf("(host name)\n");
    printf("  Length:   %d\n", len);
    memcpy(hostname, bp, len);
    hostname[len] = '\0';
    printf("  Host name:   %s\n", hostname);
    bp = bp + len;
    break;
case DHCP_OPT_DOMAINNAME: //域名
    printf("(domain name)\n");
    printf("  Length:   %d\n", len);
    memcpy(hostname, bp, len);
    hostname[len] = '\0';
    printf("  Domain name: %s\n", hostname);
    bp = bp + len;
    break;
case DHCP_OPT_VENDORSPECIFIC: //参数
    printf("(vendor specific parameters)\n");
    printf("  Length:   %d\n", len);
    printf("  Parameters:           ");
    for (i=0; i<len; i++)
    {
        printf("%x ", *bp);
        bp++;
    }
    printf("\n");
    break;
case DHCP_OPT_NETBIOSNAMESERV:
    //netbios 名字服务器
    printf("(netbios name servers)\n");
    printf("  Length: %d\n", len);
    j = 0;
    while (j < len / 4)
    {
        memcpy((void *) &in_holder, bp, sizeof(struct in_addr));
        printf("  Address: %s\n", inet_ntoa(in_holder));
        bp = bp + sizeof(struct in_addr);
        j++;
    }
    break;
case DHCP_OPT_NETBIOSNODETYPE: //netbios 结点类型
    printf("(netbios node type)\n");

```

```
printf(" Length:  %d\n", len);
printf(" Node type:  ");
switch(*bp)
{
    case 0x1:
        printf("B\n");
        break;
    case 0x2:
        printf("P\n");
        break;
    case 0x4:
        printf("M\n");
        break;
    case 0x8:
        printf("H\n");
        break;
}
bp++;
break;
case DHCP_OPT_REQUESTEDIPADDR: //询问 IP 地址
    printf("(requested IP address)\n");
    printf(" Length:  %d\n", len);
    memcpy((void *) &in_holder, bp, sizeof(struct in_addr));
    printf(" Address:  %s\n", inet_ntoa(in_holder));
    bp = bp + sizeof(struct in_addr);
    break;
case DHCP_OPT_IPADDRLEASE: //IP 地址租用时间
    printf("(IP address lease time)\n");
    printf(" Length:  %d\n", len);
    memcpy((void *) &i, bp, sizeof(int));
    printf(" Lease time:  %d\n", ntohl(i));
    bp = bp + sizeof(u_int32_t);
    break;
case DHCP_OPT_MESSAGE_TYPE: //消息类型
    printf("(message type)\n");
    printf(" Length:  %d\n", len);
    printf(" Type: ");
    switch(*bp)
    {
        case 1:
            printf("DHCPDISCOVER\n");
```

```
        break;
    case 2:
        printf("DHCPOFFER\n");
        break;
    case 3:
        printf("DHCPREQUEST\n");
        break;
    case 4:
        printf("DHCPDECLINE\n");
        break;
    case 5:
        printf("DHCPACK\n");
        break;
    case 6:
        printf("DHCPNAK\n");
        break;
    case 7:
        printf("DHCPRELEASE\n");
        break;
    case 8:
        printf("DHCPINFORM\n");
        break;
    default: //未知类型
        printf("unknown\n");
        break;
}
bp++;
break;
case DHCP_OPT_SERVERID: //服务 ID
    printf("(server ID)\n");
    printf("  Length:  %d\n", len);
    memcpy((void *) &in_holder, bp, sizeof(struct in_addr));
    printf("  Address: %s\n", inet_ntoa(in_holder));
    bp = bp + sizeof(struct in_addr);
    break;
case DHCP_OPT_PARAMREQLIST: //需要 DHCP 选项
    printf("(DHCP options requested)\n");
    printf("  Length: %d\n", len);
    for (i=0; i < len; i++)
    {
        printf("  Option requested:      %d\n",
```



```

        (u_int32_t) *bp);
        bp++;
    }
    break;
case DHCP_OPT_MAXDHCPMSGSIZE: //最大消息大小
    printf("(max message size)\n");
    printf("  Length:  %d\n", len);
    memcpy((void *) &s, bp, sizeof(u_int16_t));
    printf("  Size: %d\n", ntohs(s));
    bp = bp + sizeof(u_int16_t);
    break;
case DHCP_OPT_RENEWALTIME: //更新时间
    printf("(renewal time)\n");
    printf("  Length:  %d\n", len);
    memcpy((void *) &i, bp, sizeof(u_int32_t));
    printf("  Renewal time:  %d\n", ntohl(i));
    bp = bp + sizeof(u_int32_t);
    break;
case DHCP_OPT_REBINDINGTIME: //重新绑定时间
    printf("(rebinding time)\n");
    printf("  Length:      %d\n", len);
    memcpy((void *) &i, bp, sizeof(u_int32_t));
    printf("  Rebinding time:  %d\n", ntohl(i));
    bp = bp + sizeof(u_int32_t);
    break;
case DHCP_OPT_VENDORCLASSID: //提供类型 ID
    printf("(vendor class ID)\n");
    printf("  Length:      %d\n", len);
    printf("  Parameters:  ");
    for (i=0; i<len; i++)
    {
        printf("%x ", *bp);
        bp++;
    }
    printf("\n");
    break;
case DHCP_OPT_CLIENTID: //客户 ID
    printf("(client ID)\n");
    printf("  Length:      %d\n", len);
    printf("  Parameters: ");
    for (i=0; i<len; i++)

```

```

    {
        printf("%x ", *bp);
        bp++;
    }
    printf("\n");
    break;
case DHCP_OPT_END: //选项末尾
    printf("(end of options)\n");
    done = 1;
    bp++;
    break;
default: //未知选项
    printf("(unknown)\n");
    printf("  Length:    %d\n", len);
    bp = bp + len;
} // switch
} // while
printf("%d padding bytes\n", end_pad_cnt);
}

```

6.3.3 IPX/SPX 协议

IPX (Internetwork Packet eXchange) 协议属于 NetWare 协议组中，NetWare 协议与 OSI 模型的对照如图 6-19 所示，可以看到 IPX 属于网络层数据报传输协议。

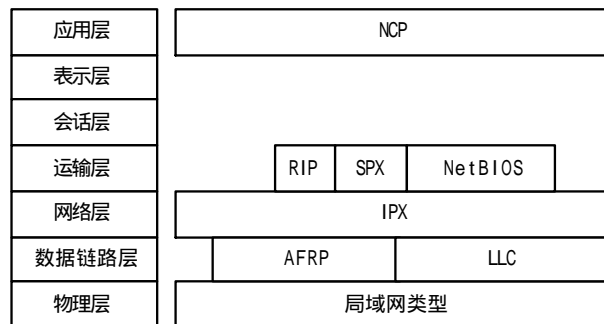


图 6-19 IPX/SPX 协议栈

IPX 协议与 IP 协议是两种不同的网络层协议。从图 6-19 可以看出 SPX(Sequenced Pakcet eXchange) 对应于传输层。IPX 负责从发者向接收者传送数据包，这些数据包也包括路由。SPX 通过对包传送的确认来监视包传送的过程，它负责提供差错控制能力，如果数据包的内容不可用，则需要重新发包，IPX/SPX 在 NetWare 的局域网上提供传输服务，同时也提供网络的控制信息和网络上可用服务的信息。

IPX/SPX 数据包的格式如图 6-20 所示。

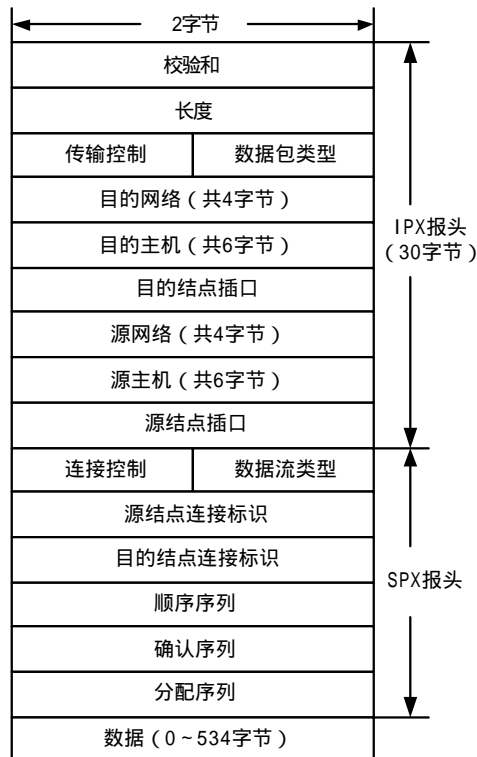


图 6-20 IPX/SPX 数据包格式

6.4 使用 Libnids 库

6.4.1 Libnids 库简介

Libnids (Network Intrusion Detect System library) 库提供的接口函数主要实现了开发网络入侵监测系统所使用的一些最基本的结构框架。它是在函数库 libnet 和 libpcap 的基础上开发的，它封装了开发 NIDS 所需的一些基本函数。Libnids 库提供的函数能够监视通过本地的所有网络通信，检查数据包等。这是因为它使用了 libpcap 的一些数据包捕获功能。刚才说过，libpcap 也使用了 libnet 库，libnet 库是一个编写网络安全程序的一个非常有用的接口库，libnet 的介绍参考后面章节。由于它使用了 libnet 库，所以具有了重构网络数据包的功能，具体的是它可以重新构造 TCP 数据段等。另外它还增加了一些自己的功能，它可以处理 IP 分片包，监测 TCP 端口扫描的功能。这些都是 libnids 的优点。Libnids 的实现模拟了 Linux kernel 2.0.x 的 IP 协议栈。

如果使用 libnids 接口函数库，那么程序开发者就不用再编写底层的网络处理代码，只需开发入侵检测的实质功能就可以了，由于 libnids 只是提供了一些框架，这个框架没有内容，那么就需要开发者来自己添加内容，可以在此基础上设计自己的入侵检测系统。本系统就用到了 libnids，我们扩展了它的功能，在它的基础上设计了具体入侵检测功能。只有加上了具体的入侵检测模块，才能够实现入侵检测。

下面具体介绍一下 libnids 的基本构架，介绍一下它实现的主要功能，更详细的内容可以参考其源代码，源代码是最好的文档。Libnids 主要实现了以下一些功能。

(1) 接受 IP 包或 IP 分片数据包

```
void ip_frag_func(struct ip * a_packet)
```

使用此函数就可以接收所有的 ip 网络数据包，此数据包包括分片包、畸形网络包，等等。参数 a_packet 就是用来存网络数据包。

```
nids_register_ip_frag(ip_frag_func);
```

使用此函数进行注册，这样就 libnids 可以进行捕获了。这个函数的参数就是我们刚才定义的函数 ip_frag_func，这样当有 ip 数据的时候就会调用 ip_frag_func 函数，此时用到的机制也是回调机制。在这之前必须使用 nids_init() 函数进行初始化，每个基于 libnids 的程序初始化后，都必须使用 nids_init() 函数，这样注册的函数才能够使用。

如果只是接收正确的 IP 数据包，即只接受不是碎片包，头部校验正确的数据包，等等，就使用如下的回调函数：

```
void ip_func(struct ip * a_packet)
```

再调用以下函数进行注册：

```
nids_register_ip(ip_func);
```

(2) TCP 数据流重组

```
void tcp_callback(struct tcp_stream * ns, void ** param)
```

使用此函数来接收 TCP 数据包，这是个回调函数。参数 ns 表示一个 TCP 连接，那么 TCP 连接的所有内容就都由此 ns 参数提供。它是 tcp_stream 结构类型的，下面分析此结构体：

```
struct tcp_stream
{
    struct tuple4 addr; //连接参数，包括 saddr, daddr, sport, dport
    char nids_state; //连接的逻辑状态
    struct half_stream client, server; //表示客户端和服务端端的机构体
    ... //其他是 libnids 使用的辅助字段
};
```

此结构体的第一个参数 addr 包括了 TCP 连接的一些参数，包括源地址、目的地址、源端口和目的端口。其结构体 struct tuple4 类型定义如下：

```
struct tuple4 // TCP connection parameters
{
    unsigned short source, dest; // 客户端和服务端端的端口号
    unsigned long saddr, daddr; // 客户端和服务端端的地址
};
```

结构体 tcp_stream 的第二个参数是 nids_state 表示 nids 的状态。此状态将决定 tcp_callback 的动作，此状态有以下几种类型：

当此状态是 NIDS_JUST_EST 时，ns 表示一个刚刚建立的连接。那么 tcp_callback 就可以根据这个决定是否对该连接的后续数据进行检查。如果需要检查，tcp_callback 回调函数将

通知 libnids 它希望接收哪些数据，例如流向客户端的数据、流向服务器端的数据、流向客户端的紧急数据或流向服务器端的紧急数据，等等，最后返回。

当此状态是 NIDS_DATA 时，则表示此连接接收到新的数据。

当是其他的状态时，例如 NIDS_CLOSE、NIDS_RESET、NIDS_TIMEOUT，则表示该连接已经关闭了。

结构体 tcp_stream 的第三个参数是 client，表示客户端的连接信息，第四个参数是 server，表示服务器端的 TCP 连接信息。这两个参数的类型是结构体 half_stream，下面分析此结构体。

```
struct half_stream    // 描述一端 TCP 连接的结构体
{
    char state;        // 套接字状态，如 TCP_ESTABLISHED
    char collect;      // 如果大于 0，则保存数据到 data 中，否则此数据流将被忽略掉
    char collect_urg;   // 决定是否捕获紧急数据
    char * data;        // 用来存储正常数据流
    unsigned char urgdata; // 用来存放紧急数据
    int count;          // 显示从连接开始已经存放到 data 中多少个字节
    int offset;         // data 中首字节数据偏移量
    int count_new;
    // 显示最近一次存放到 data 中的字节数，如果为 0，表示没有数据
    char count_new_urg; // 如果不是 0，则表示有新的紧急数据到达
    ...                // 其他的是 libnids 用到的辅助字段
};
```

libnids 允许 nids_register_tcp 和 nids_register_ip 两个函数可以被调用多次，这样就可以注册不同的回调函数，在每个回调函数中就可以实现不同的功能了。

libnids 库还定义了一个结构体 nids_prm，用此结构体声明一个全局变量 nids_params，介绍如下：

```
struct nids_prm
{
    int n_tcp_streams;
    // 表示用来存放 tcp_stream 结构的 hash 表的大小，默认为 1024。如果设为 0， //libnids
    // 将不能够重组 TCP 流。
    int n_hosts;
    // 表示存放 IP 分片信息的 hash 表大小，默认为 256。
    char * device;
    // 表示 libnids 所要监听的网络设备，默认为 NULL，这时候就由 pcap_lookupdev 函数；来
    // 确定，如果为 all，就表示 libnids 试着捕获所有网络设备上的数据包。
    int sk_buff_size;
    // 表示 sk_buff 结构大小，此结构体是由 linux 内核定义的，内核用它来对数据包进行排队，
    // 如果 sk_buff_size 的值跟 sizeof(struct sk_buff)的数值不一样，libnids 可能被旁路，所以如果你不知
    // 到大小，那么你就在你的主机上检查一下 sizeof(struct sk_buff)的大小，然后更正此参数值，默认
    // 为 168。
    int dev_addon;
```

```

//表示在结构体 sk_buff 中为网络接口上的信息保留的字节数, 如果其值为-1, 它将由
nids_nit()函数根据 libnids 将监听的网络接口类型来进行更正, 默认为-1。
void (*syslog)();
// 指向日志函数
int syslog_level;
// 如果 nids_params.syslog==nids_syslog, 则此字段将确定日志等级, 此日志登记是为系统后台程序 syslogd 来记录事件所使用的, 默认值为 LOG_ALERT。
int scan_num_hosts;
// 表示用来存放端口扫描信息的 hash 表的大小。如果值为 0, 则端口扫描监测功能将被关闭, 默认值为 256。
int scan_num_ports;
// 表示从同一个 ip 地址有多少个 tcp 端口号必须被扫描到, 默认值 10。
int scan_delay;
// 为了使 libnids 报告端口扫描攻击, 在这里定义一个在两次端口扫描中的间隔时间。默认值为 3000。
void (*no_mem)();
// 当 libnids 运行时内存不够时被调用, 此时应该终止当前进程。
int (*ip_filter)(struct ip*);
// 是指向一个函数, 此函数当接收到一个 IP 数据包时被调用。如果此函数返回一个非零值, 则处理该数据包, 否则就忽略该数据包。 这种方式就可以只检测某一个特定主机的数据包, 而不是整个子网。默认函数为 nids_ip_filter, 它永远返回值 1。
char *pcap_filter;
// 用来表是过滤规则字符串, 此过滤规则是由 pcap 使用的。默认为 NULL。注意这个规则是应用于网络链路层, 所以象 " tcp dst port 23 " 将不能捕获到碎片数据包。
int promisc;
//如果此参数不是 0 ,那么 libnids 读取数据包的网络设备将被设置为混杂模式, 默认值为 1。
int one_loop_less ;
//默认不可用。
} nids_params;

```

刚才提到过 nids_params 的 syslog 字段默认时是指向 nids_syslog 函数, 此函数声明如下:

```
void nids_syslog (int type, int errnum, struct ip *iph, void *data);
```

nids_params.syslog 函数用于记录异常情况, 如端口扫描企图, 无效 TCP 头标志等。该字段应指向自定义的日志处理函数。nids_syslog()仅作为一个例子。nids_syslog()函数向系统守护服务 syslogd 发送日志消息。

下面举一个简单的例子来分析一下怎样使用 libnids 来编写程序, 此例子的功能是分析出所有 tcp 连接的信息。

```

// -----start programe -----
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/in_sysm.h>

```

```

#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include "nids.h" //必须包含此头文件
#define int_ntoa(x)    inet_ntoa(((struct in_addr *)&x))
// 结构体 tuple4 包含 TCP 连接的地址和端口号
char *
adres (struct tuple4 addr)
// 此函数是把 IP 地址转化为字符串
{
    static char buf[256];
    strcpy (buf, int_ntoa (addr.saddr));
    sprintf (buf + strlen (buf), "%i,", addr.source);
    strcat (buf, int_ntoa (addr.daddr));
    sprintf (buf + strlen (buf), "%i", addr.dest);
    return buf;
}
void
tcp_callback (struct tcp_stream *a_tcp, void ** this_time_not_needed)
//分析 TCP 连接的回调函数
{
    char buf[1024];
    strcpy (buf, adres (a_tcp->addr));
    if (a_tcp->nids_state == NIDS_JUST_EST)
    {
        // 一个 TCP 连接已经建立，可以在此处加上代码进行其他分析。例如加上代码：if
        (a_tcp->addr.dest!=25) return;表示不处理目标端口号是 25 的网络数据包。在这个程序中我们处理
        所有的网络数据包。
        a_tcp->client.collect++;
        //处理客户端网络数据包
        a_tcp->server.collect++;
        // 处理服务器端网络数据包
        a_tcp->server.collect_urg++;
        // 处理服务器端的紧急网络数据包
#ifdef WE_WANT_URGENT_DATA_RECEIVED_BY_A_CLIENT
        a_tcp->client.collect_urg++;
        // 处理客户端的紧急网络数据包。
#endif
        fprintf (stderr, "%s established\n", buf);
        return;
    }
    if (a_tcp->nids_state == NIDS_CLOSE)

```

```
{
    // 表示 TCP 连接正常关闭
    fprintf(stderr, "%s closing\n", buf);
    return;
}
if (a_tcp->nids_state == NIDS_RESET)
{
    // TCP 连接因 RST 而关闭
    fprintf(stderr, "%s reset\n", buf);
    return;
}
if (a_tcp->nids_state == NIDS_DATA)
{
    //表示接收到新的网络数据包;
    struct half_stream *hlf;
    if (a_tcp->server.count_new_urg)
    {
        // 有新的紧急数据到达服务器端
        strcat(buf, "(urgent->");
        buf[strlen(buf)+1]=0;
        buf[strlen(buf)]=a_tcp->server.urgdata;
        write(1,buf,strlen(buf));
        return;
    }
    if (a_tcp->client.count_new)
    {
        // 有新的数据到达客户端
        hlf = &a_tcp->client;
        strcat (buf, "<-");
    }
    else
    {
        //有新的网络数据到达服务器端
        hlf = &a_tcp->server;
        strcat (buf, "->");
    }
    fprintf(stderr,"%s",buf);
    //显示 TCP 连接两端的参数，包括 IP 地址和端口号
    write(2,hlf->data,hlf->count_new);
    //显示新到的数据
}
return ;
```



```

    }
    int main () //主函数
    {
        //此处可以对 libnids 的全局变量进行赋值
        // 例如：nids_params.n_hosts=256;
        if (!nids_init ()) //初始化，必须的
        {
            fprintf(stderr,"%s\n",nids_errbuf);
            exit(1);
        }
        nids_register_tcp (tcp_callback); //注册回调函数
        nids_run (); //开始运行，libnids 的实际运行函数，必须的
        return 0;
    }
    // -----end programe -----

```

6.4.2 分析 TCP 连接过程

TCP 连接的状态图如图 6-21 所示。

(1) TCP 连接的建立

建立连接应用的是三消息握手。如果双方同时都发送 SYN 也没有关系，双方会发现这个 SYN 中没有确认，于是就知道了这种情况，通常来说，应该发送一个“reset”段来解决这种情况。三消息握手减少了连接失败的可能性。

请求端发送一个 SYN 段指明客户打算连接的服务器的端口，以及初始序号（ISN），这个 SYN 报文段为报文段 1。

服务器端发回包含服务器的初始序号的 SYN 报文段（报文段 2）作为应答。同时将确认序号设置为客户的 ISN 加 1 以对客户的 SYN 报文段进行确认。一个 SYN 将占用一个序号。

客户必须将确认序号设置为服务器的 ISN 加 1 以对服务器的 SYN 报文段进行确认（报文段 3）。

这 3 个报文段完成连接的建立，称为三次握手。发送第一个 SYN 的一端将执行主动打开，接收这个 SYN 并发回下一个 SYN 的另一端执行被动打开。

使用三消息握手的主要原因是为了防止使用过期的数据段。为了这个目的，必须引入新的控制消息，RESET。如果接收 TCP 处理非同步状态，在接收到 RESET 后返回到 LISTEN 状态。如果 TCP 处理下面几种状态 ESTABLISHED、FIN-WAIT-1、FIN-WAIT-2、CLOSE-WAIT、CLOSING、LAST-ACK、TIME-WAIT 时，放弃连接并通过用户。

(2) TCP 连接的终止

CLOSE 是一个操作，它的意思就是“本方已经有数据发送”。由于是全双工的，所以会造成一些麻烦，因为接收方对于处理接收方的连接有点麻烦。我们以一种简单的方式对待 CLOSE，发送 CLOSE 的一方在接收到对方的 CLOSED 之前，还要继续接收数据。因此程序

可以在一个 CLOSE 之后初始化几个 SEND , 然后开始 RECEIVE , 直到接收到对方的 CLOSED 而 RECEIVE 失败为止。我们假设 TCP 可以通知用户连接关闭, 即使仍在 RECEIVE 也可以, 这样用户就可以正常关闭了。这样, TCP 可以在连接关闭前可靠地发送数据。

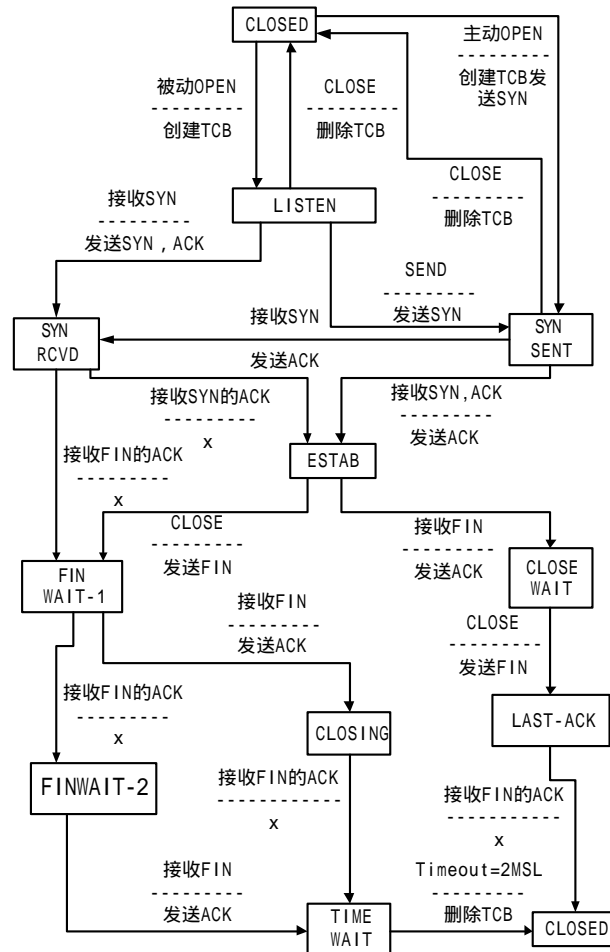


图 6-21 TCP 连接状态图

由于 TCP 连接是全双工的, 因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个 FIN 来终止这个方向的连接。收到一个 FIN 只意味着这一方向上没有数据流动, 一个 TCP 连接在收到一个 FIN 后仍能发送数据。首先进行关闭的一方将执行主动关闭, 而另一方执行被动关闭。

TCP 客户端发送一个 FIN, 用来关闭客户到服务器的数据传送 (报文段 4)。

服务器收到这个 FIN, 它发回一个 ACK, 确认序号为收到的序号加 1 (报文段 5)。和 SYN 一样, 一个 FIN 将占用一个序号。

服务器关闭客户端的连接, 发送一个 FIN 给客户端 (报文段 6)。

客户端发回确认, 并将确认序号设置为收到序号加 1 (报文段 7)。

图 6-22 是分析 TCP 连接过程的运行界面。在这个画面中, TCP 连接的是 HTTP 协议。我们的测试环境是在一个机器上运行 Web 服务器, 其 IP 地址为 192.168.0.34, 在另一个机器

上运行浏览器，其 IP 地址 192.168.0.36。此时展示的 TCP 连接过程是一个 HTTP 会话过程，首先建立连接，然后开始通信，最后切断连接。可以看到 TCP 连接的整个运行细节。

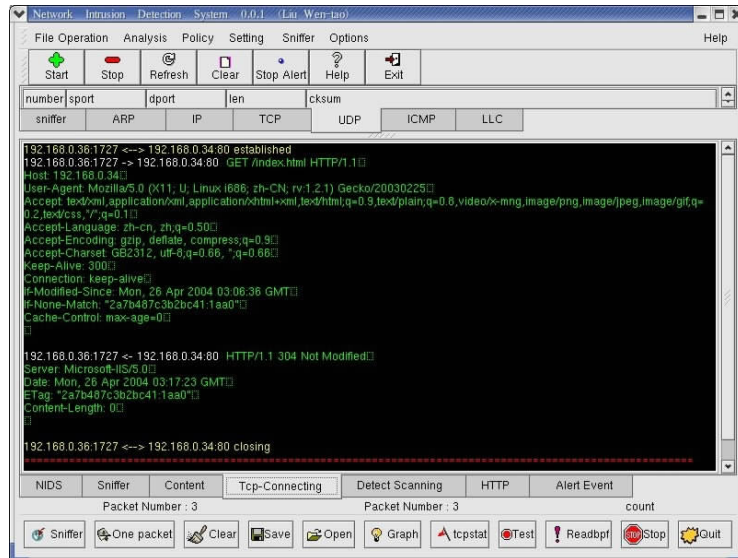


图 6-22 TCP 的连接过程分析

分析 TCP 连接过程的入口函数是 `show_tcp_connecting()`，其函数关系如图 6-23 所示。

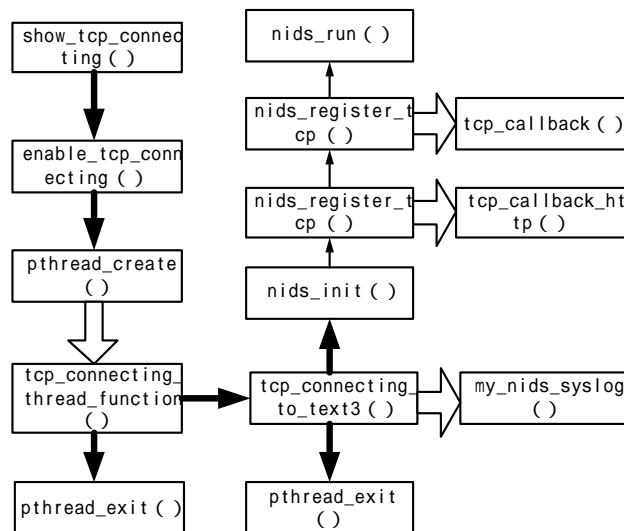


图 6-23 TCP 连接分析中函数调用关系

```

void
show_tcp_connecting(GtkWidget * widget, gpointer * data)
//此函数是菜单的回调函数，参考界面设计部分
{
    static int status = 1;    //菜单的状态，是否被选择
    if (status)               //菜单被选择

```

```

    {
        show_tcp_connecting_yesno = 1;    //TCP 连接标志为 1
        enable_tcp_connecting ();          //分析 TCP 连接过程，下面有实现
    }
    else
    {
        show_tcp_connecting_yesno = 0;    //否则 TCP 连接标志就为 0
    }
    status = !status;                      //取反，此菜单是选择菜单类型的
}

```

函数中调用了下面一个函数：

```

void
enable_tcp_connecting (void)
{
    pthread_t tcp_connecting_thread;
    //线程句柄
    pthread_create (&tcp_connecting_thread, NULL,
        tcp_connecting_thread_function, NULL);
    // 创建一个线程，其回调函数为 tcp_connecting_thread_function ( )
    // 下面有它的实现
}

void *
tcp_connecting_thread_function (void *args)
//此是线程的回调函数
{
    if (show_tcp_connecting_yesno == 1) //判断是否需要分析 TCP 连接过程
    {
        tcp_connecting_to_text3 ();    //分析 TCP 连接过程，其实现在后面介绍
    }
    if (show_tcp_connecting_yesno == 0) //不需要就退出线程
    {
        pthread_exit (0);              //退出线程
    }
}

void
tcp_connecting_to_text3 (void)
{
    if (show_tcp_connecting_yesno == 0)
        //设置的标志，如果为 1 就分析 TCP 连接过程
        //如果为 0，就退出此线程，在此系统中我们使用了多线程技术。请参考后面章节。
    {

```

```

        pthread_exit (0); //退出线程
    return;
}
nids_params.syslog = my_nids_syslog;
//设置日志函数，此函数在第 9 章有实现。
openlog ("nids", 0, LOG_LOCAL0);
nids_params.device = device_total;
//设置捕获设备
nids_params.pcap_filter = "";
//设置过滤规则
if (!nids_init ()) //NIDS 库函数初始化
{
    fprintf (stderr, "%s\n", nids_errbuf);
    exit (1);
}
nids_register_tcp (tcp_callback);
//注册函数 tcp_callback，后面有实现
nids_register_tcp (tcp_callback_http);
//注册函数 tcp_callback_http，后面有实现
nids_run ();
//运行 NIDS
}

```

下面是实现 TCP 连接过程分析的函数：

```

void
tcp_callback (struct tcp_stream *a_tcp, void ** this_time_not_needed)
{
    char buf[1024];
    char content[1024];
    if(show_tcp_connecting_yn==0)
    {
        pthread_exit(0);
        return ;
    }
    strcpy (buf, adres (a_tcp->addr)); //存储信息
    if (a_tcp->nids_state == NIDS_JUST_EST)
    {
        //表示建立 TCP 连接开始建立，如图 6-22 所示
        //显示 192.168.0.36<-->192.168.0.34 established
        a_tcp->client.collect++;
        a_tcp->server.collect++;
        a_tcp->server.collect_urg++;
    }
}

```

```

#ifdef WE_WANT_URGENT_DATA_RECEIVED_BY_A_CLIENT
a_tcp->client.collect_urg++;
#endif
sprintf (content, "%s established\n", buf);
gdk_threads_enter();
insert_tcp_connecting_to_text3_red(content); //动态显示 TCP 连接状态
gdk_threads_leave();
return;
}
if (a_tcp->nids_state == NIDS_CLOSE)    //关闭状态
{
    //TCP 连接关闭，如图 6-22 所示，192.168.0.36<-->192.168.0.34 closing
    sprintf (content, "%s closing\n", buf);
    gdk_threads_enter();
    insert_tcp_connecting_to_text3_red(content); //动态显示 TCP 连接状态
    gdk_threads_leave();
    sprintf(content, "===== \n");
    gdk_threads_enter();
    insert_tcp_connecting_to_text3_red1(content); //动态显示 TCP 连接状态
    gdk_threads_leave();
    return;
}
if (a_tcp->nids_state == NIDS_RESET)    //重置状态
{
    sprintf (content, "%s reset\n", buf);
    gdk_threads_enter();
    insert_tcp_connecting_to_text3_red(content); //动态显示 TCP 连接状态
    gdk_threads_leave();
    return;
}
if (a_tcp->nids_state == NIDS_DATA)    //有数据传输
{
    //TCP 连接过程中的数据传输状态，如图 6-22 所示
    struct half_stream *hlf;
    if (a_tcp->server.count_new_urg)
    {
        strcat(buf, "(urgent->)");
        buf[strlen(buf)+1]=0;
        buf[strlen(buf)]=a_tcp->server.urgdata;
        snprintf(content, strlen(buf), "%s", buf);
        gdk_threads_enter();
        insert_tcp_connecting_to_text3_red(content); //动态显示 TCP 连接状态
    }
}

```

```

        gdk_threads_leave();
        return;
    }
    if (a_tcp->client.count_new)
    {
        hlf = &a_tcp->client;
        strcpy (buf, adres_left (a_tcp->addr));
    }
    else
    {
        hlf = &a_tcp->server;
        strcpy (buf, adres_right (a_tcp->addr));
    }
    sprintf(content,"%s  ",buf);
    gdk_threads_enter();
    insert_tcp_connecting_to_text3(content);    //动态显示 TCP 连接状态
    gdk_threads_leave();
    snprintf(content,hlf->count_new+1,"%s",hlf->data);
    strcat(content,"\n");
    gdk_threads_enter();
    insert_tcp_connecting_to_text3_green(content);
    //动态显示 TCP 连接状态
    gdk_threads_leave();
}
return ;
}

```

里面使用了下面几个函数，实现如下：

```

char *
adres (struct tuple4 addr)
{
    //返回类似 192.168.0.36 <--> 192.168.0.34 的字符串，
    //表示两个地址之间有信息传输
    static char buf[256];
    strcpy (buf, int_ntoa (addr.saddr));
    sprintf (buf + strlen (buf), ":%i", addr.source);
    strcat(buf," <--> ");
    strcat (buf, int_ntoa (addr.daddr));
    sprintf (buf + strlen (buf), ":%i", addr.dest);
    return buf;
}
char *

```

```
adres_left (struct tuple4 addr)
{
    //返回类似 192.168.0.36 <-192.168.0.34 的字符串 ,
    //表示数据从地址 192.168.0.34 传送到 192.168.0.36
    static char buf[256];
    strcpy (buf, int_ntoa (addr.saddr));
    sprintf (buf + strlen (buf), ":%i", addr.source);
    strcat(buf, " <- ");
    strcat (buf, int_ntoa (addr.daddr));
    sprintf (buf + strlen (buf), ":%i", addr.dest);
    return buf;
}

char *
adres_right (struct tuple4 addr)
{
    //返回类似 192.168.0.36-> 192.168.0.34 的字符串 ,
    //表示数据从地址 192.168.0.36 传送到 192.168.0.34
    static char buf[256];
    strcpy (buf, int_ntoa (addr.saddr));
    sprintf (buf + strlen (buf), ":%i", addr.source);
    strcat(buf, " -> ");
    strcat (buf, int_ntoa (addr.daddr));
    sprintf (buf + strlen (buf), ":%i", addr.dest);
    return buf;
}
```

只要是基于 TCP 协议的应用层协议，都可以通过上面的实现来分析 TCP 连接过程。我们通过 FTP 协议来实际分析一下 TCP 协议的连接过程。我们的测试环境是在 IP 地址为 192.168.0.34 的主机上运行 FTP 服务器，FTP 服务器的名字和版本为 Microsoft FTP Service (Version 5.0)。在另一台 IP 地址为 192.168.0.36 的机器上执行 FTP 操作，如图 6-24 所示，其操作过程如下：

```
[root@peterwuhan root]#ftp 192.168.0.34
Connected to 192.168.0.34 (192.168.0.34).
220 server Microsoft FTP Service (Version 5.0).
Name (192.168.0.34:root):anonymous
331 Anonymous access allowed,send identity (e-mail name ) as pawword.
Password:
230-hello.
230-This is a FTP server.
230 Anonymous user logged in.
Remote system type is Windows_NT.
ftp>put test.c
```



```

local:test.c remote test.c
227 Entering Passive Mode (192,168,0,34,8,31).
125 Data connection already open:Transfer starting.
226 Transfer complete.
162 bytes sent in 0.0121 secs ( 13 Kbytes/sec)
ftp>quit
221 You are quit. Bye!

```

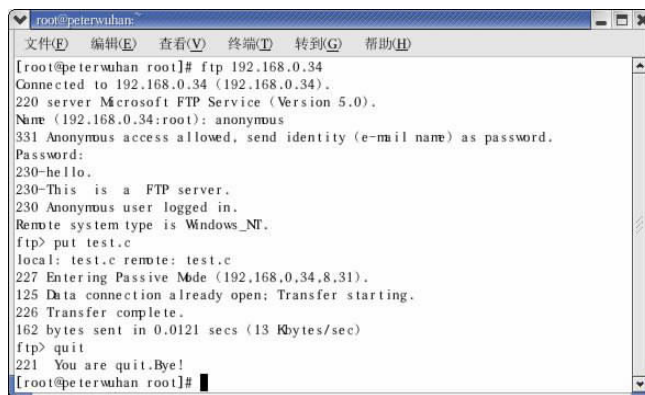


图 6-24 FTP 操作测试

在这里首先连接 FTP 服务器 192.168.0.34，输入用户名，匿名（anonymous），然后输入密码 12345，显示欢迎信息，然后用 put 命令向 FTP 服务器传送一个文件 test.c，然后执行退出命令，显示结束信息。

用本入侵检测系统分析此 TCP 连接过程的结果如图 6-25 所示，其分析过程显示如下：

```

192.168.0.36:1048 <--> 192.168.0.34:21 established
192.168.0.36:1048 <- 192.168.0.34:21  220 server Microsoft FTP Service (Version 5.0).
192.168.0.36:1048 -> 192.168.0.34:21  USER anonymous
192.168.0.36:1048 <- 192.168.0.34:21  331 Anonymous access allowed, send identity (e-mail name) as
password.
192.168.0.36:1048 -> 192.168.0.34:21  PASS 12345
192.168.0.36:1048 <- 192.168.0.34:21  230-hello.
192.168.0.36:1048 <- 192.168.0.34:21  230-This is a FTP server.
230 Anonymous user logged in.
192.168.0.36:1048 -> 192.168.0.34:21  SYST
192.168.0.36:1048 <- 192.168.0.34:21  215 Windows_NT version 5.0
192.168.0.36:1048 -> 192.168.0.34:21  PASV
192.168.0.36:1048 <- 192.168.0.34:21  227 Entering Passive Mode (192,168,0,34,8,31).
192.168.0.36:1056 <--> 192.168.0.34:2079 established
192.168.0.36:1048 -> 192.168.0.34:21  STOR test.c
192.168.0.36:1048 <- 192.168.0.34:21  125 Data connection already open; Transfer starting.
192.168.0.36:1056 -> 192.168.0.34:2079  #include <stdio.h>
int max(int x,int y)

```

```

{
return x+y;
}
int main ()
{
int a,b,c;
a=1;
b=2;
c=0;
c=max(a,b);
printf ("a+b=%d\n",c);
return 1;
}
192.168.0.36:1056 <--> 192.168.0.34:2079 closing
=====

192.168.0.36:1048 <- 192.168.0.34:21 226 Transfer complete.
192.168.0.36:1048 -> 192.168.0.34:21 QUIT
192.168.0.36:1048 <- 192.168.0.34:21 221 You are quit.Bye!
192.168.0.36:1048 <--> 192.168.0.34:21 closing
=====

```

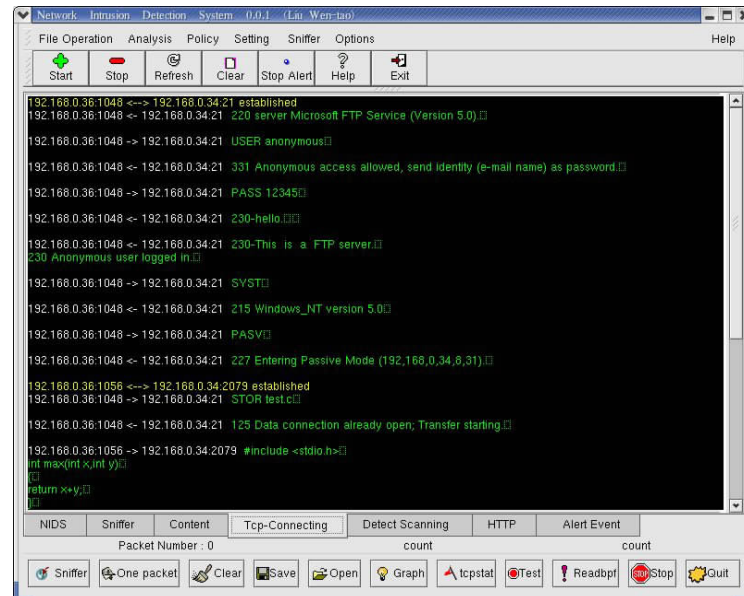


图 6-25 分析 FTP 协议的连接过程

我们对此分析结果解释一下，开始一行就是说明两个机器建立了 TCP 连接。IP 地址之间的连接是双箭头。然后 FTP 服务器 192.168.0.34 和客户端 192.168.0.36 之间进行控制信息传输。IP 地址之间的连接是单箭头，箭头指向谁就表示信息传送到那个机器，例如 192.168.0.36:1048 -> 192.168.0.34:21 就表示信息从机器 192.168.0.36 传送到 192.168.0.34，冒

号后面跟的是端口号，表示从机器 192.168.0.36 的端口 1048 传送信息到机器 192.168.0.34 的端口 21。21 就是 FTP 服务器的端口号。

建立连接之后，要求输入用户名，我们使用的是匿名 anonymous，然后要求输入密码。在控制台上输入密码是不显示的，但是在我们这个系统中，使用 TCP 连接分析就可以分析出用户的密码了，显示如下：

```
192.168.0.36:1048 -> 192.168.0.34:21 PASS 12345
```

可以看到密码就是 12345。这是因为其是明文传输的，如果是密文传输，就不可能显示出密码了。所以现在文件传输都是用加密形式传输的，可以使用 SSH 协议。如果用此系统来分析 SSH 协议，就会发现其显示的信息都是乱码，就是因为其经过加密，显示的是杂乱无章的内容。

再看其中一句如下：

```
192.168.0.36:1056 <--> 192.168.0.34:2079 established
```

这句表示又建立了一个 TCP 连接，端口分别是 1056 和 2079，这是由 FTP 协议决定的。因为 FTP 协议的控制信息和数据信息的传送是两个 TCP 连接过程。当建立了数据信息传输的 TCP 连接之后，就可以开始数据传输了。我们用 put 命令上传了一个名字为 test.c 的文件，其内容就是几行 C 语言程序。可以看出其数据传输的显示信息如下：

```
192.168.0.36:1056 -> 192.168.0.34:2079 #include <stdio.h>
int max(int x,int y)
{
    return x+y;
}
int main ()
{
    int a,b,c;
    a=1;
    b=2;
    c=0;
    c=max(a,b);
    printf ("a+b=%d\n",c);
    return 1;
}
```

这段显示结果，就是说明从客户端 192.168.0.36 的端口 1056 传送数据信息到服务器 192.168.0.34 的端口 2079。而此数据信息就是文件 test.c 的内容。

6.4.3 分析 HTTP 协议

HTTP (Hyper Text Transfer Protocol) 超文本传输协议是用来在 WWW 上交换文件（文本，图形，声音，动画等）的规则集。相对于 TCP/IP 协议族，HTTP 是应用协议。HTTP 协议采用了请求/响应模型，HTTP 是一个基于消息的协议，当客户机向服务器发送请求或服务器向客户机返回响应消息时，都要使用到 HTTP 协议。图 6-26 表示了一个典型的 HTTP 协议

的客户机与服务器之间的交互过程。

HTTP 消息格式如图 6-27 所示。

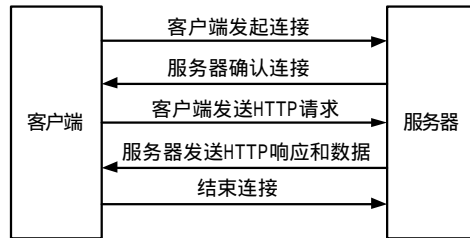


图 6-26 HTTP 客户端和服务会话

请求消息	响应消息
请求行	状态行
通用头	通用头
请求头	请求头
实体头	实体头
CR/LF	CR/LF
实体	实体

图 6-27 HTTP 消息格式

通常 HTTP 消息包括客户机向服务器的请求消息和服务器向客户机的响应消息。这两种类型的消息由一个起始行，一个或者多个头域，一个只是头域结束的空行和可选的消息体组成。HTTP 的头域包括通用头、请求头、响应头和实体头四个部分。每个头域由一个域名、冒号 (:) 和域值三部分组成。域名是大小写无关的，域值前可以添加任何数量的空格符，头域可以被扩展为多行，在每行开始处，使用至少一个空格或制表符。

客户端发往服务端的信息是请求消息。一个请求消息有以下几个部分组成：请求行 (REQUEST LINE)、一个或多个头 (HEAD)、一个隔离的 CR/LF 对和实体 (ENTITY BODY)。

服务器端发往客户端的消息是响应消息。一个响应消息有以下几个部分组成：状态行 (STATUS LINE)、一个或多个头 (HEADER)、一个隔离的 CR/LF (CARRIAGE RETURN/LINE FEED) 和实体 (ENTITY BODY)。

请求消息的请求行为下面的格式：

Method REQUEST-URI HTTP-Version CRLF

Method 表示对于 Request-URI 完成的方法，这个字段是大小写敏感的，包括 OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE。方法 GET 和 HEAD 应该被所有的通用 WEB 服务器支持，其他所有方法的实现是可选的。GET 方法取回由 Request-URI 标识的信息。HEAD 方法也是取回由 Request-URI 标识的信息，只是可以在响应时，不返回消息体。POST 方法可以请求服务器接收包含在请求中的实体信息，可以用于提交表单，向新闻组、BBS、邮件群组 and 数据库发送消息。

Request-URI 遵循 URI 格式，在此字段为星号 (*) 时，说明请求并不用于某个特定的资源地址，而是用于服务器本身。HTTP-Version 表示支持的 HTTP 版本，例如为 HTTP/1.1。CRLF 表示换行回车符。请求头域允许客户端向服务器传递关于请求或者关于客户机的附加信息。请求头域可能包含下列字段 Accept, Accept-Charset, Accept-Encoding, Accept-Language, Authorization, From, Host, If-Modified-Since, If-Match, If-None-Match, If-Range, If-Range, If-Unmodified-Since, Max-Forwards, Proxy-Authorization, Range, Referer, User-Agent。对请求头域的扩展要求通信双方都支持，如果存在不支持的请求头域，一般将会作为实体头域处理。

响应消息的状态行为下面的格式：

HTTP-Version Status-Code Reason-Phrase CRLF

HTTP-Version 表示支持的 HTTP 版本，例如为 HTTP/1.1。Status-Code 是一个三个数字

的结果代码。Reason-Phrase 给 Status-Code 提供一个简单的文本描述。Status-Code 主要用于机器自动识别，Reason-Phrase 主要用于帮助用户理解。

状态码 (Status-Code) 的第一个数字定义响应的类别，后两个数字没有分类的作用。第一个数字可能取 5 个不同的值：

- 1xx：信息，保留以便将来使用。
- 2xx：处理成功响应类，动作已经被成功接收、理解并接受了。
- 3xx：重新定向，还需要进一步的动作来完成请求。
- 4xx：客户端错误，请求语法错误或不能完成。
- 5xx：服务端错误，服务器不能完成一个明显合理的请求。

典型的响应消息包括以下几个部分。

通用头域 (GENERAL HEADERS)

通用头域包含请求和响应消息都支持的头域，通用头域包含 Cache-Control, Connection, Date, Pragma, Transfer-Encoding, Upgrade, Via。对通用头域的扩展要求通信双方都支持此扩展，如果存在不支持的通用头域，一般将会作为实体头域处理。其中 DATE 表示请求或响应的日期和时间。PRAGMA 表示允许客户发送关于自己或对服务器请求的其他信息的头。

请求头 (REQUEST HEADERS)

一个请求头允许客户向服务器发送附加消息，内容可以是关于请求或客户本身的。有以下合法的请求头。AUTHORIZATION：允许一个客户向服务器证实自己。FORM：允许客户向服务器发送一个用户的电子邮件地址。IF-MODIFIED-SINCE：允许客户有条件的根据最终修改时间重新得到请求信息。REFERER：允许客户向服务器通报请求 URL 资源文档。USER-AGENT：允许客户发送关于本身的信息，使得服务器可以根据请求客户应用的限制来修正响应。

响应头 (RESPONSE HEADERS)

相应头允许服务器传递关于响应的附加信息给客户。这些消息不可以放在状态行里。LOCATION：定义资源的确切位置。SERVER：包含关于服务器软件的信息。WWW-AUTHENTICATE：包含未授权响应的消息。

实体头 (ENTITY HEADERS)

请求消息和响应消息都可以包含实体信息，实体信息一般由实体头域和实体组成。实体头域包含关于实体的原信息，实体头包括 Allow, Content-Base, Content-Encoding, Content-Language, Content-Length, Content-Location, Content-MD5, Content-Range, Content-Type, Etag, Expires, Last-Modified, extension-header。extension-header 允许客户端定义新的实体头，但是这些域可能无法未接受方识别。实体可以是一个经过编码的字节流，它的编码方式由 Content-Encoding 或 Content-Type 定义，它的长度由 Content-Length 或 Content-Range 定义。Content-Type 用于向接收方指示实体的介质类型，指定 HEAD 方法送到接收方的实体介质类型，或 GET 方法发送的请求介质类型。Content-Range 用于指定整个实体中的一部分的插入位置，他也指示了整个实体的长度。在服务器向客户返回一个部分响应，它必须描述响应覆盖的范围和整个实体长度。Last-modified 实体头指定服务器上保存内容的最后修订时间。

图 6-28 是分析 HTTP 协议的运行界面。

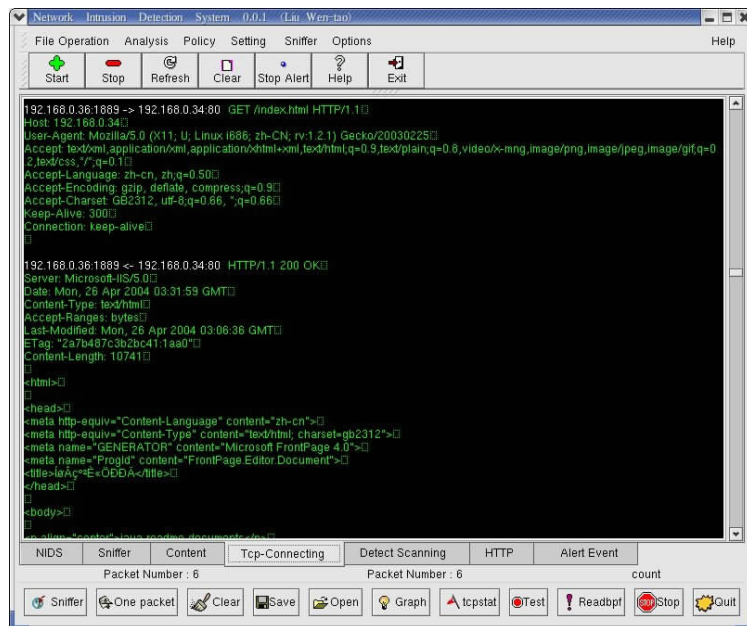


图 6-28 分析 HTTP 协议

察看 HTTP 通信的内容，查看分析后的网页内容，重建通过 HTTP 协议传输的文件。其中使用的结构体如下：

```

struct http_packet_string
{
    char time[1024];
    char source_ip[1024];
    char destination_ip[1024];
    char sport[1024];
    char dport[1024];
    char destination_hostname[1024];
    char url[1024];
    char data_type[1024];
    char data_length[1024];
};

```

其函数实现如下：

```

void
tcp_callback_http (struct tcp_stream *a_tcp, void ** this_time_not_needed)
{
    //此函数在函数 tcp_connecting_to_text3 (void)中被注册。
    char buf[1024];
    char content[1024];
    int source_port_local;
    if(show_tcp_connecting_yn==0)

```

```

{
    pthread_exit(0);
    return ;
}
strcpy (buf, adres (a_tcp->addr));
source_port_local=a_tcp->addr.source;
if (a_tcp->nids_state == NIDS_JUST_EST)
{
    if (a_tcp->addr.dest!=80) return;
    //只捕获 http 协议的数据包
    a_tcp->client.collect++;
    a_tcp->server.collect++;
    a_tcp->server.collect_urg++;
#ifdef WE_WANT_URGENT_DATA_RECEIVED_BY_A_CLIENT
    a_tcp->client.collect_urg++;
#endif
    sprintf (content, "%s established\n", buf);
    number_connecting++;
    source_port[number_connecting]=source_port_local;
    sprintf(http_content_string[number_connecting], "%s", "");
    gdk_threads_enter();
    insert_text_to_text5_red(content);
    gdk_threads_leave();
    return;
}
if (a_tcp->nids_state == NIDS_CLOSE)
{
    sprintf (content, "%s closing\n", buf);
    gdk_threads_enter();
    insert_text_to_text5_red(content);
    gdk_threads_leave();
    sprintf(content, "=====\n");
    gdk_threads_enter();
    insert_text_to_text5_red1(content);
    gdk_threads_leave();
    fprintf(stderr, "the 2");
    http_content[number_connecting]=http_content_string[number_connecting];
    fprintf(stderr, "the 3");
    fprintf(stderr, "the http_content[%d] is:", number_connecting);
    fprintf(stderr, "the 4");
    fprintf(stderr, "%s\n", http_content[number_connecting]);
    fprintf(stderr, "the 5");
    html=get_html(http_content[number_connecting]);

```

```

//获取 HTML 内容，由函数 http_content 来实现
fprintf(stderr,"the 6");
fprintf(stderr,"the later:%s\n",html);
{
//这断内容是创建一个文件用来存储 HTML 信息，这样这个文件就是后缀为.html 格式
//的文件，可以用浏览器来显示出来。请参考存储模块实现章节。
    char filename[1024];
    //文件名字
    int logfd;
    //文件句柄
    guint length;
    char dirname1[1024];
    //目录
    char dirname2[1024];
    //目录
    sprintf(dirname1,"./%s",int_ntoa(a_tcp->addr.saddr));
    //把当前目录加上以源 ip 地址为名字的目录赋值给 dirname1
    mkdir(dirname1,O_RDWR|O_CREAT|O_TRUNC);
    //以 dirname1 为名字来创建目录
    sprintf(dirname2,"%s/%s",dirname1,int_ntoa(a_tcp->addr.daddr));
    //把当前目录加上 dirname1 再加上目的 ip 地址为名字的目录赋值给 dirname2
    mkdir(dirname2,O_RDWR|O_CREAT|O_TRUNC);
    //创建以 dirname2 为名字的目录
    sprintf(filename,"%s/%i.html",dirname2,a_tcp->addr.source);
    //把 dirname2 加上源端口为名字再加上.html 的字符串赋值给 filename
    logfd=open(filename,O_RDWR|O_CREAT|O_TRUNC,0600);
    //打开以 filename 的文件，如果不存在，就创建
    write(logfd,html,strlen(html));
    //向以 logfd 表示的文件中写内容，内容就是 html 代码
    close(logfd);
    //关闭文件
}
{
    char timestr[1024];
    getcurrenttime(timestr);
    //获取当前时间
    strcpy(http_packet.time,timestr);
    //存储当前时间到 HTTP 信息中
    sprintf(http_packet.source_ip,"%s",int_ntoa(a_tcp->addr.saddr));
    //存储源 ip 地址
    sprintf(http_packet.destination_ip,"%s",int_ntoa(a_tcp->addr.daddr));
    //存储目的 ip 地址
    sprintf(http_packet.sport,"%i",a_tcp->addr.source);

```



```

        //存储源端口
        sprintf(http_packet.dport,"%i",a_tcp->addr.dest);
        //存储目的端口
        sprintf(http_packet.destination_hostname,"%s","");
        //存储目的主机名字
        sprintf(http_packet.url,"%s","");
        //存储目的 rul
        sprintf(http_packet.data_type,"%s","");
        //存储数据类型
        sprintf(http_packet.data_length,"%d",strlen(html));
        //存储 html 内容
    }
    //显示信息
    g_print("%s",http_packet.time);
    g_print("%s",http_packet.source_ip);
    g_print("%s",http_packet.destination_ip);
    g_print("%s",http_packet.sport);
    g_print("%s",http_packet.dport);
    g_print("%s",http_packet.destination_hostname);
    g_print("%s",http_packet.url);
    g_print("%s",http_packet.data_type);
    insert_http_content_into_database();
    //把分析后的 http 信息存储到数据库中，请参考后面章节
    return;
}
if (a_tcp->nids_state == NIDS_RESET)
{
    sprintf (content, "%s reset\n", buf);
    gdk_threads_enter();
    insert_text_to_text5_red(content);
    //把信息动态显示到界面(显示 http 信息的页面，请参考界面设计模块)中
    gdk_threads_leave();
    return;
}
if (a_tcp->nids_state == NIDS_DATA)
//数据传输状态
{
    struct half_stream *hlf;
    if (a_tcp->server.count_new_urg)
    {
        strcat(buf,"(urgent->)");
        buf[strlen(buf)+1]=0;
        buf[strlen(buf)]=a_tcp->server.urgdata;
    }
}

```

```

        snprintf(content, strlen(buf), "%s", buf);
        gdk_threads_enter();
        insert_text_to_text5_red(content);
        //把信息动态显示到界面(显示 http 信息的页面，请参考界面设计模块)中
        gdk_threads_leave();
        return;
    }
    if (a_tcp->client.count_new)
    {
        hlf = &a_tcp->client;
        strcpy (buf, adres_left (a_tcp->addr));
        sprintf(content, "%s ", buf);
        gdk_threads_enter();
        insert_text_to_text5(content);
        gdk_threads_leave();
        snprintf(content, hlf->count_new+1, "%s", hlf->data);
        strcat(content, "\n");
        fprintf(stderr, "the 7");
        strcat(http_content_string[number_connecting], content);
        gdk_threads_enter();
        insert_text_to_text5_red(content);
        //把信息动态显示到界面(显示 http 信息的页面，请参考界面设计模块)中
        gdk_threads_leave();
    }
    else
    {
        hlf = &a_tcp->server;
        strcpy (buf, adres_right (a_tcp->addr));
        sprintf(content, "%s ", buf);
        gdk_threads_enter();
        insert_text_to_text5(content);
        gdk_threads_leave();
        snprintf(content, hlf->count_new+1, "%s", hlf->data);
        strcat(content, "\n");
        gdk_threads_enter();
        insert_text_to_text5_green(content);
        gdk_threads_leave();
    }
}
return ;
}

```

在这个函数中，有分析出 HTML 信息的函数。其实现如下：

```
char *
get_html(char *line)
{
    int i = 0;
    int j;
    char *string;
    while (1)
    {
        if ((line[i] == '<') && ((line[i + 1] == 'h') || (line[i + 1] == 'H'))
            && ((line[i + 2] == 't') || (line[i + 2] == 'T'))
            && ((line[i + 3] == 'm') || (line[i + 3] == 'M'))
            && ((line[i + 4] == 'l') || (line[i + 4] == 'L'))))
            break;
        if (i == strlen(line))
            break;
        i++;
    }
    if (i == strlen(line))
    {
        return "";
    }
    string = line + i;
    return string;
}
```

在这里只是简单的读取其内容的开始处，然后就可以把从此开始的所有内容都读出来，然后存放到文件中。此时文件中就是 html 代码了。如果用浏览器来读取代码就可以显示网页内容了。请参考 7.4.3 节。

根据同样的道理，如果想分析 FTP, TELET, POP3 等协议，然后把他们的内容也读出来，都可以根据此方法来进行分析。限于篇幅，在这里就不再详细分析和实现。

第7章 存储模块设计与实现

7.1 设计原理

存储模块把入侵检测系统检测到的数据信息记录下来，以便于以后的分析。前面设计的网络协议分析模块，它产生了很多有用的原始信息，需要及时把它们存储下来，不然它们就会丢失。而这个功能就是由存储模块来完成的。我们在设计存储模块的时候，考虑到模块化设计的思想，把存储模块作为一个单独的模块来进行设计。在入侵检测系统需要把分析后的数据信息存储到数据库的时候，才挂用此存储模块，所以此存储模块可以动态的挂载和卸载。

我们可以创建一个专用的存储系统来存储网络信息，但是这样会使系统变得很复杂，也没有那个必要。所以决定选择现有的数据库系统来实现存储模块。在本系统中我们使用MySQL来建立存储系统。MySQL数据库是Linux下一个非常流行的数据库。

在设计存储模块的时候，通常要考虑以下几个问题。

1. 数据存储

使用数据库系统，数据信息进入数据库的时候可以有如下两种选择：

把原始数据直接写入数据库系统中进行存储，并在数据库完成过滤分析。

在数据库外进行过滤分析，在数据库中只存放分析结果。

第一种选择的优点在于，一旦制定了载入原始数据的方法并对它们进行处理，就解决了数据载入的所有问题。第二种方法的优点就是能够得到更好的性能和灵活性，但只要任何事情发生改变就必须重新处理数据载入的问题。

数据是实时提交还是分批提交，各有利弊。主要原因是在数据载入数据库后，要把该数据写到数据库的物理存储中，就必须调用提交命令。如果得到每个数据就提交一次，这样对信息处理会提供方便，但影响系统的运行性能。如果分批提交，你就可以获得更高的性能，但这样延长了系统的等待时间。分批提交降低了系统实时性的要求。可以通过调节每次提交的数量来控制实时性的要求，如果系统硬件条件允许，如有比较高的处理器速度、比较大的内存、高速的外部存储设备等，就可以将批量提交的数量减小，甚至可以是减小到每个数据就提交一次，以得到更好的实时性。为了提高检测粒度，就可以将提交的数量适量增大。

在本系统中，我们使用的是实时提交的方法。

2. 数据处理

由于某种原因，例如误报警或者漏报警，这时候安全管理员就需要入侵检测系统提供所有的可用数据信息，包括数据包头和内容，以便于进行手工分析。但是我们无法永久保存这样的高保真数据，因为这些数据包是非常庞大的，这时就需要对数据库进行一次或多次处理。

对数据信息进行处理的一种有效的办法是把数据库分为两个主要的存储方式：原始数据数据库和长期记录数据库。

原始数据数据库：原始数据数据库中存储了近一段时间的高保真数据，这些数据包括源 IP 地址、目标 IP 地址、原始数据、欺骗包的特征、新的攻击特征、最后一次见到的时间等。这个数据库应该建立多重索引和优化，以便能够进行最有效的搜索。原始数据数据库中存储很短一段时间内的数据包。

原始数据数据库中还有一个重要的问题是存储多少数据、原始数据要保存多长时间。一般认为原始数据应该保存 3 天到一周，总是尽可能长地保存原始数据，因为数据处理后进行手工分析就会受到限制。但原始数据要占用大量的存储器，同时原始数据的增加会延长数据库检索的时间，降低数据分析速度。

可以让系统提供两种保存原始数据的设置方式：按时间和按存储容量。按时间设置以分析为依据设置原始数据应该保存的时间，而按存储容量是以保证检索性能为前提来配置系统。

长期记录数据库：长期记录数据库以缩减数据的格式记录了很长一段时间的检测情况。长期记录数据库主要支持产生报告而不是交互式的查询，还能帮助可能漏掉的事件进行检测。缩减数据格式一般只包括时间、源 IP 地址、目标 IP 地址、源端口、目标端口、协议标志等。

在本系统中两种数据库类型都具有。

3. 存储方法

存储模块从网络协议分析模块和其他模块（如入侵检测模块）中获取数据信息有两种方法：推方法（PUSH）和拉方法（PULL）。推方法是在协议分析模块分析一个数据包的时候就把数据信息推给存储模块。拉方法是存储模块在需要数据时查询协议分析模块，一起获得最新的数据信息。推方法对于入侵检测系统来说是一种比较好的结构。

在本系统中使用的就是推方法，当协议分析模块在分析一个网络数据包的时候就把分析后的数据信息马上存储到数据库中。

7.2 MySQL 数据库

MySQL 是一个快速的客户-服务器结构的 SQL 数据库管理系统，由一个服务器守护程序 mysqld 及很多不同的客户程序和库组成。虽然它不是开放的源代码产品，但可以自由使用。其功能强大、灵活性好、应用编程接口丰富并且系统结构精巧。

MySQL 具有下述一些重要特征：

- 使用核心线程的完全多线程，这样它就能很容易地利用多 CPU；
- 具有 C, C++, Eiffel, Java, Perl, PHP, Python 和 TCL 等多种语言的 API；
- 可运行在多种不同的平台上，包括 Windows；
- 利用一个优化的一遍扫描多重联结能够快速地进行联结；
- 在查询的 SELECT 和 WHERE 部分，支持全部运算符和函数；
- 全面支持 SQL 的 GROUPBY 和 ORDERBY 子句，支持聚合函数 COUNT(), COUNT(DISTINCT), AVG(), STD(), SUM(), MAX()和 MIN()；
- 有一个非常灵活且安全的权限和口令系统，允许基于主机的认证，而且口令是安全的，因为当与一个服务器连接时，所有口令传送被加密。

7.2.1 安装 MySql 数据库

安装 MySql 比较简单，其步骤如下：

```
rpm -ivh MySQL-server-4.0.18-0.i386.rpm
rpm -ivh MySQL-client-4.0.18-0.i386.rpm
rpm -ivh MySQL-devel-4.0.18-0.i386.rpm
rpm -ivh MySQL-shared-4.0.18-0.i386.rpm
rpm -ivh MySQL-shared-compat-4.0.18-0.i386.rpm
```

如果不出什么差错，这样就安装完成了。

7.2.2 基本操作

在这里不打算详细介绍 MySql 的所有命令，只是介绍常用的和与本系统有关的命令。

(1) 显示数据库信息

使用 mysqlshow 命令：

```
[root@peterwuhan mysql]# mysqlshow //显示所有数据库信息
+-----+
| Databases |
+-----+
| mysql     |
| test      |
+-----+

[root@peterwuhan mysql]# mysqlshow mysql //只显示 mysql 数据库信息
Database: mysql
+-----+
| Tables |
+-----+
| columns_priv |
| db           |
| func         |
| host         |
| tables_priv  |
| user        |
+-----+
```

(2) 启动和关闭 MySql 服务器

使用 mysql.server 脚本就可以，mysql.server 脚本可以被用来启动或停止服务器，通过用 start 或 stop 参数调用它：

```
mysql.server start //启动 MySql 服务器
mysql.server stop  //关闭 MySql 服务器
```

(3) 改变口令

可以有很多种方法，第一种使用 set 命令：

```
mysql> set password for root@"localhost"=password('zzz');
Query OK, 0 rows affected (0.00 sec)

mysql> flush privileges;
Query OK, 0 rows affected (0.00 sec)
```

第二种使用 update 命令：

```
mysql> update user set password=password('zz') where user='root';
Query OK, 2 rows affected (0.00 sec)

Rows matched: 2   Changed: 2   Warnings: 0

mysql> flush privileges;
Query OK, 0 rows affected (0.08 sec)
```

这两种方法在 user 表里直接更新口令，你必须告诉服务器再次读入授权表（用 FLUSH PRIVILEGES），否则改变将不被注意到。

第三种方式是使用 mysqladmin 命令：

```
[root@peterwuhan linux_ids]# mysqladmin -u root -p password zzz
Enter password:
[root@peterwuhan linux_ids]#
```

此时不需要用 password 函数。因为使用 mysqladmin password 命令设置口令的时候，password()函数是不必要的。他们都自动加密你设置的口令，上面的一个例子设置了一个口令 ' zzz '：

7.2.3 基本函数

MySQL 支持很多语言，在本系统中使用的是 C 语言，所以在这里着重介绍一下 MySQL 数据库中与 C 语言相关的 API 函数。

```
MYSQL *mysql_connect(MYSQL *mysql, const char *host, const char *user, const char *passwd)
```

此函数是与一个 MySQL 服务器进行连接。也可以使用 mysql_real_connect()函数来进行连接数据库服务器。

```
void mysql_close(MYSQL *mysql)
```

此函数是关闭一个正在运行的服务器连接。此函数也释放由 mysql_init() 和 mysql_connect()两个函数分配的句柄。

```
my_bool mysql_change_user(MYSQL *mysql, const char *user, const char *password, const char *db)
```

此函数改变正在连接的数据库上的一个用户。

```
int mysql_create_db(MYSQL *mysql, const char *db)
```

此函数是创建一个数据库，数据库的名字是 db。也可以使用 SQL 命令 CREATE DATABASE 来创建数据库。

```
void mysql_data_seek(MYSQL_RES *result, unsigned long long offset)
```

此函数是在一个数据库查询结果集合中定位任意一行。

```
void mysql_debug(char *debug)
```

此函数用于调试，即使用给定字符串来做一个 DBUG_PUSH。

```
int mysql_drop_db(MYSQL *mysql, const char *db)
```

此函数释放一个数据库名字是 db 的数据库。也可以使用 SQL 命令 DROP DATABASE 来释放一个数据库。

```
int mysql_dump_debug_info(MYSQL *mysql)
```

此函数是把调试信息记录下来。

```
my_bool mysql_eof(MYSQL_RES *result)
```

此函数判断是不是已经读到一个结果集的最后一行。可以使用 mysql_errno() 和 mysql_error() 来完成相似的功能。

```
unsigned int mysql_errno(MYSQL *mysql)
```

此函数是得到最近被调用的 MySQL 函数的出错的错误代码。

```
char *mysql_error(MYSQL *mysql)
```

此函数是返回最近被调用的 MySQL 调用中的出错消息。

```
unsigned int mysql_escape_string(char *to, const char *from, unsigned int length)
```

此函数是用在 SQL 语句中的字符串的转义特殊字符。把在 from 中的字符串编码为在一条 SQL 语句中可以发给服务器的转义的 SQL 字符串，将结果放在 to 中，并且加上一个终止的空字节。编码的字符是 NUL (ASCII 0) ' ' '\n ' '\r' '\ ' '\ ' '\ ' '\ ' 和 Control-Z。由 from 指向的字符串必须是 length 个字节长。你必须分配 to 的缓冲区至少 length*2+1 个字节长。当 mysql_escape_string() 返回时，to 的内容将是空字符终止的字符串。返回值是编码后的字符串的长度，不包括终止空字符。

```
MYSQL_FIELD *mysql_fetch_field(MYSQL_RES *result)
```

返回下一个表字段的类型。返回作为一个 MYSQL_FIELD 结构的一个结果集的一个列的定义。重复调用这个函数在结果集中检索所有关于列的信息。当没有剩下更多的字段时，mysql_fetch_field() 返回 NULL。

```
MYSQL_FIELD *mysql_fetch_fields(MYSQL_RES *result)
```

返回一个表字段的类型，给出一个字段编号。返回一个结果集的所有 MYSQL_FIELD 结构的数组。每个结构提供结果集中一列的字段定义。

```
MYSQL_FIELD *mysql_fetch_fields(MYSQL_RES *result)
```

返回一个所有字段结构的数组。返回一个结果集的所有 MYSQL_FIELD 结构的数组。每个结构提供结果集中一列的字段定义。

```
MYSQL_FIELD *mysql_fetch_field_direct(MYSQL_RES *result, unsigned int fieldnr)
```

给定在一个结果集中的一个列的字段编号 fieldnr，返回作为 MYSQL_FIELD 结构的列的字段定义。你可以使用这个函数检索任意列的定义。fieldnr 的值应该在从 0 到 mysql_num_fields(result)-1 范围内。

```
unsigned long *mysql_fetch_lengths(MYSQL_RES *result)
```

返回当前行中所有列的长度。返回在结果集合内的当前行的列长度。

```
MYSQL_ROW mysql_fetch_row(MYSQL_RES *result)
```

从结果集中取得下一行。检索一个结果集的下一行。

MYSQL_FIELD_OFFSET mysql_field_seek(MYSQL_RES *result, MYSQL_FIELD_OFFSET offset)

把列光标放在一个指定的列上。将字段光标设置到给定的偏移量。下一次调用 mysql_fetch_field() 将检索与该偏移量关联的列的字段定义。

unsigned int mysql_field_count(MYSQL *mysql)

返回最近查询的结果列的数量。返回在连接上的最近查询的列的数量。

MYSQL_FIELD_OFFSET mysql_field_tell(MYSQL_RES *result)

返回用于最后一个 mysql_fetch_field() 的字段光标的位置。返回用于最后一个 mysql_fetch_field() 的字段光标的位置。

void mysql_free_result(MYSQL_RES *result)

释放一个结果集合使用的内存。

char *mysql_get_client_info(void)

返回客户版本信息。返回代表客户库的版本的字符串。

char *mysql_get_host_info(MYSQL *mysql)

返回一个描述连接的字符串。返回描述正在使用的连接类型的字符串，包括服务器主机名。

char *mysql_get_server_info(MYSQL *mysql)

返回连接使用的协议版本。

char *mysql_get_server_info(MYSQL *mysql)

返回服务器版本号。返回表示服务器版本号的字符串。

char *mysql_info(MYSQL *mysql)

返回关于最近执行的查询信息。

MYSQL *mysql_init(MYSQL *mysql)

获得或初始化一个 MYSQL 结构。

my_ulonglong mysql_insert_id(MYSQL *mysql)

返回有前一个查询为一个 AUTO_INCREMENT 列生成的 ID。

int mysql_kill(MYSQL *mysql, unsigned long pid)

杀死一个给定的线程。

MYSQL_RES *mysql_list_dbs(MYSQL *mysql, const char *wild)

返回匹配一个简单的正则表达式的数据库名。

MYSQL_RES *mysql_list_fields(MYSQL *mysql, const char *table, const char *wild)

返回匹配一个简单的正则表达式的列名。

MYSQL_RES *mysql_list_processes(MYSQL *mysql)

返回当前服务器线程的一张表。返回一个描述当前服务器线程的结果集合。

MYSQL_RES *mysql_list_tables(MYSQL *mysql, const char *wild)

返回匹配一个简单的正则表达式的表名。

unsigned int mysql_num_fields(MYSQL_RES *result)

返回一个结果集中的列的数量。在结果集中返回列的数量。

```
my_ulonglong mysql_num_rows(MYSQL_RES *result)
```

返回一个结果集中的行的数量。在结果集中返回行的数量。

```
int mysql_options(MYSQL *mysql, enum mysql_option option, const char *arg)
```

设置对 mysql_connect() 的连接选项。能用于设置额外连接选项并且影响一个连接的行为。

```
int mysql_ping(MYSQL *mysql)
```

检查对服务器的连接是否正在工作。

```
int mysql_query(MYSQL *mysql, const char *query)
```

执行指定为一个空结尾的字符串的 SQL 查询。

```
MYSQL *mysql_real_connect(MYSQL *mysql, const char *host, const char *user, const char *passwd,
const char *db, unsigned int port, const char *unix_socket, unsigned int client_flag)
```

此函数是一个非常重要的函数，其功能是连接一个 MySQL 数据库服务器。mysql_real_connect() 试图建立到运行 host 的一个 MySQL 数据库引擎的一个连接。第一个参数应该是一个现存 MYSQL 结构的地址。在调用 mysql_real_connect() 之前，你必须调用 mysql_init() 初始化 MYSQL 结构。参数 host 的值可以是一个主机名或一个 IP 地址。如果 host 是 NULL 或字符串 "localhost"，假定是到本地主机的一个连接。如果 OS 支持套接字 (Unix) 或命名管道 (Win32)，则使用它们而不是 TCP/IP 与服务器连接。参数 user 包含用户的 MySQL 登录 ID。如果 user 是 NULL，假定是当前用户。参数 passwd 为 user 包含口令。如果 passwd 是 NULL，只有在 user 表中对于有一个空白口令字段的用户的条目将被检查一个匹配。这允许数据库主管设置 MySQL 权限，使用户获得不同的口令，取决于他们是否已经指定一个口令。参数 db 是数据库名。如果 db 不是 NULL，则连接将默认数据库设置为这个值。参数 port 如果不是 0，值对于 TCP/IP 连接将用作端口号。参数 unix_socket 如果不是 NULL，字符串指定套接字或应该被使用的命名管道。参数 client_flag 的值通常是 0。

```
int mysql_real_query(MYSQL *mysql, const char *query, unsigned int length)
```

执行指定为带计数的字符串的 SQL 查询。

```
int mysql_reload(MYSQL *mysql)
```

告诉服务器重装授权表。

```
MYSQL_ROW_OFFSET mysql_row_seek(MYSQL_RES *result, MYSQL_ROW_OFFSET offset)
```

搜索在结果集中的行，使用从 mysql_row_tell() 返回的值。

```
MYSQL_ROW_OFFSET mysql_row_tell(MYSQL_RES *result)
```

返回行光标位置。

```
int mysql_select_db(MYSQL *mysql, const char *db)
```

连接一个数据库。

```
int mysql_shutdown(MYSQL *mysql)
```

关掉数据库服务器。

```
char *mysql_stat(MYSQL *mysql)
```

返回作为字符串的服务器状态。

```
MYSQL_RES *mysql_store_result(MYSQL *mysql)
```

检索一个完整的结果集合给客户。mysql_store_result()读取一个到客户的查询的全部结果,分配一个MYSQL_RES结构,并且把结果放进这个结构中。当调用了mysql_store_result(),就可以调用mysql_num_rows()找出结果集中有多少行。

```
unsigned long mysql_thread_id(MYSQL *mysql)
```

返回当前线程的ID。

```
MYSQL_RES *mysql_use_result(MYSQL *mysql)
```

初始化一个一行一行地结果集合的检索。mysql_use_result()初始化一个结果集合的检索,但不真正将结果集合读入客户,就像mysql_store_result()那样。相反,必须通过调用mysql_fetch_row()单独检索出每一行,这直接从服务器读出结果而不在一个临时表或本地缓冲区中存储它,它比mysql_store_result()更快一点并且使用较少的内存。客户将只为当前行和一个可能最大max_allowed_packet字节的通信缓冲区分配内存。

更详细的函数介绍,请参考MySQL相关文档。

7.3 存储模块实现

在本系统中,存储模块是用MySQL数据库来实现数据库存储的。为了方便管理,我们使用了一个非常优秀的MySQL管理系统PHPMyAdmin。PHPMyAdmin是用来远程连接MySQL数据库的客户端软件。使用此系统就要用到PHP和Apache。下面详细介绍一下这些内容。

7.3.1 使用PHPMyAdmin管理数据库

PHPMyAdmin可以管理整个MySQL服务器,也可以管理单个数据库。为了实现后一种,你将需要合理设置MySQL用户,他只能对允许的数据库进行读/写。在Linux下使用PHPMyAdmin之前要安装Web服务器,因为PHPMyAdmin是基于Web模式的。所以必须安装Web服务器,在本系统中,我们使用Apache服务器。下面简单介绍一下Apache服务器和PHPMyAdmin的安装和配置。

(1) 安装Apache服务器

解开安装包,使用命令tar -zxvf apache_1.3.24.tar.gz,其执行过程如下:

```
[root@peterwuhan apache]# ls
```

```
apache_1.3.24.tar.gz
```

```
[root@peterwuhan apache]# tar -zxvf apache_1.3.24.tar.gz
```

经过上面解包后,出现目录apache_1.3.24,进入此目录,使用命令如下:

```
[root@peterwuhan apache]# ls
```

```
apache_1.3.24  apache_1.3.24.tar.gz
```

```
[root@peterwuhan apache]# cd apache_1.3.24
```

```
[root@peterwuhan apache_1.3.24]# ls
```

```
ABOUT_APACHE  config.layout  INSTALL  README  WARNING-WIN.TXT
```

```
Announcement  configure  LICENSE  README.configure
```

```
cgi-bin      htdocs      logs          README-WIN.TXT
conf         icons        Makefile.tmpl src
```

对安装进行配置，使用 `configure` 命令，执行过程如下：

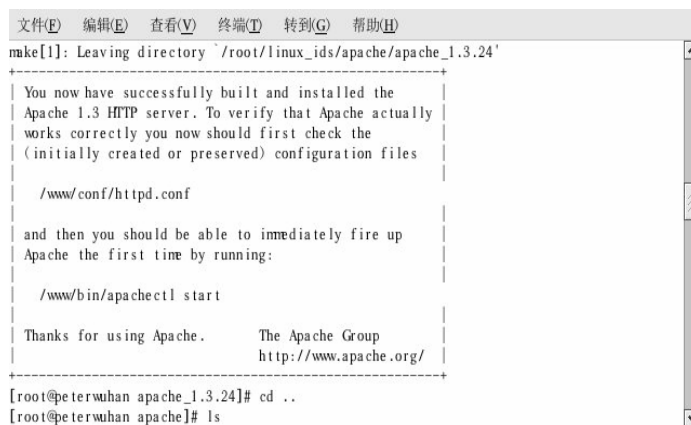
```
[root@peterwuhan apache_1.3.24]# ./configure --prefix=/www --enable-module=so
```

使用 `make` 命令，执行过程如下：

```
[root@peterwuhan apache_1.3.24]# make
```

使用 `make install` 命令进行安装，安装成功后如图 7-1 所示，其执行过程如下：

```
[root@peterwuhan apache_1.3.24]# make install
```



```
文件(E)  编辑(E)  查看(V)  终端(T)  转到(G)  帮助(H)
make[1]: Leaving directory `/root/linux_ids/apache/apache_1.3.24'
+-----+
| You now have successfully built and installed the |
| Apache 1.3 HTTP server. To verify that Apache actually |
| works correctly you now should first check the |
| (initially created or preserved) configuration files |
|-----+
| /www/conf/httpd.conf |
|-----+
| and then you should be able to immediately fire up |
| Apache the first time by running: |
|-----+
| /www/bin/apachectl start |
|-----+
| Thanks for using Apache.      The Apache Group |
|                               http://www.apache.org/ |
+-----+
[root@peterwuhan apache_1.3.24]# cd ..
[root@peterwuhan apache]# ls
```

图 7-1 apache 安装

至此，apache 服务器安装成功。接下来就是安装 PHP，因为 PHPMyAdmin 要使用到 PHP。

(2) 安装 PHP

第一步就是解压缩，使用 `tar` 命令，其执行过程如下：

```
[root@peterwuhan php]# tar -zxvf php4-latest.tar.gz
```

经过第一步后，产生一个目录 `php4-200208281800`，进入此目录：

```
[root@peterwuhan php]# ls
```

```
php4-200208281800  php4-latest.tar.gz
```

```
[root@peterwuhan php]# cd php4-200208281800
```

再使用 `configure` 命令来配置安装过程，其执行过程如下：

```
[root@peterwuhan php4-200208281800]# ./configure --with-mysql=/usr --with-apxs=/www/bin/
```

使用 `make` 命令，其执行过程如下：

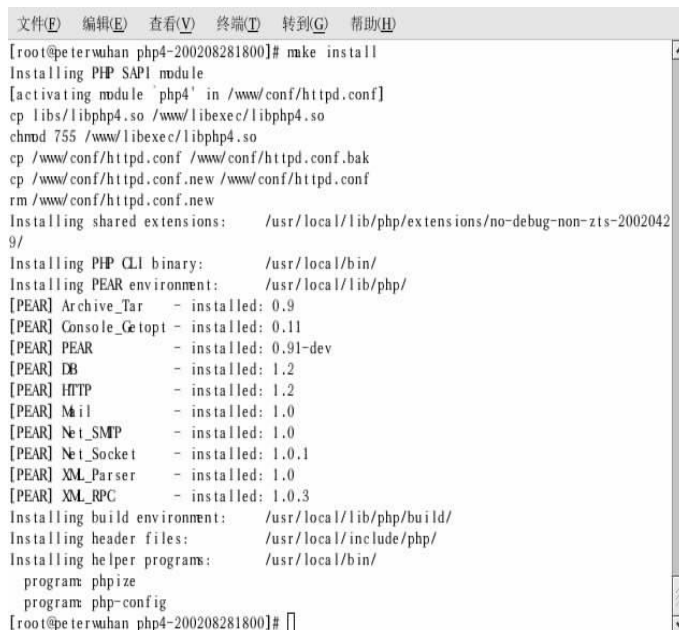
```
[root@peterwuhan php4-200208281800]# make
```

使用 `make install` 命令进行安装，安装成功后如图 7-2 所示，其执行过程如下：

```
[root@peterwuhan php4-200208281800]# make install
```

配置 `php.ini` 文件，使用命令：

```
[root@peterwuhan php4-200208281800]# cp php.ini-dist /usr/local/lib/php.ini
```



```

文件(F) 编辑(E) 查看(V) 终端(T) 转到(G) 帮助(H)
[root@peterwuhan php4-200208281800]# make install
Installing PHP API module
[activating module 'php4' in /www/conf/httpd.conf]
cp libs/libphp4.so /www/libexec/libphp4.so
chmod 755 /www/libexec/libphp4.so
cp /www/conf/httpd.conf /www/conf/httpd.conf.bak
cp /www/conf/httpd.conf.new /www/conf/httpd.conf
rm /www/conf/httpd.conf.new
Installing shared extensions:    /usr/local/lib/php/extensions/no-debug-non-zts-20020429/
Installing PHP CLI binary:      /usr/local/bin/
Installing PEAR environment:    /usr/local/lib/php/
[PEAR] Archive_Tar      - installed: 0.9
[PEAR] Console_Getopt  - installed: 0.11
[PEAR] PEAR             - installed: 0.91-dev
[PEAR] DB               - installed: 1.2
[PEAR] HTTP            - installed: 1.2
[PEAR] Mail            - installed: 1.0
[PEAR] Net_SMP         - installed: 1.0
[PEAR] Net_Socket      - installed: 1.0.1
[PEAR] XML_Parser      - installed: 1.0
[PEAR] XML_RPC         - installed: 1.0.3
Installing build environment:   /usr/local/lib/php/build/
Installing header files:       /usr/local/include/php/
Installing helper programs:    /usr/local/bin/
    program: phpbize
    program: php-config
[root@peterwuhan php4-200208281800]#

```

图 7-2 安装 php

(3) 配置 Apache

要使刚才的 PHP 生效，就必须对 apache 进行配置。对 apache 服务器进行配置，即修改 httpd.conf 文件的内容。读者可以自己查看此文件内容。

此文件中的选项很多，在这里不对他们进行详细的介绍，感兴趣的读者可以查阅相关文档。现在要使 PHP 生效，就要在此文件中加上下面两行语句：

```
AddType application/x-httpd-php .php
LoadModule php4_module      libexec/libphp4.so
```

第一句表示使 apache 服务器识别以 .php 为后缀的文件类型。第二句是加载模块的功能，在这里加入的是 php4_module 模块。

至此，我们对 apache 和 php 都安装和配置完了。接下来就是对 PHPMyAdmin 进行安装和配置。

(4) 安装和配置 PHPMyAdmin

安装 PHPMyAdmin 比较容易，首先把安装包拷贝到目录 /www/htdocs 中，此目录是在安装 apache 时生成的。然后进入 /www/htdocs 目录，其执行过程如下：

```

[root@peterwuhan phpMyAdmin]# ls
phpMyAdmin-2.2.0-php.tar.gz
[root@peterwuhan phpMyAdmin]# cp phpMyAdmin-2.2.0-php.tar.gz /www/htdocs
[root@peterwuhan phpMyAdmin]# cd /www/htdocs

```

然后，在此目录中解压缩 PHPMyAdmin 文件包，其执行过程如下：

```
[root@peterwuhan htdocs]# tar -zxvf phpMyAdmin-2.2.0-php.tar.gz
```

这样就会在当前目录生成一个 PHPMyAdmin 目录。这个时候 PHPMyAdmin 就安装成功了。现在来测试一下，首先运行 apache 服务器。然后打开浏览器（例如 mozilla），在地址栏

中输入下面一行：

`http://127.0.0.1/phpMyAdmin/index.php`

就会出现如图 7-3 所示的画面。



图 7-3 phpMyAdmin 运行结果

可以看到出现错误的信息。提示说没有密码，怎样处理这个问题呢？就需要对 PHPMyAdmin 进行配置。配置 PHPMyAdmin 就是要修改文件 `config.inc.php`。此文件就在 PHPMyAdmin 目录下面，读者可以查看此文件的内容。

下面简要介绍一下此文件中各字段代表什么意思。

`$cfgServers` 数组

从 1.4.2 版本开始，phpMyAdmin 支持对多个 MySQL-server 的管理。所以，增加了 `$cfgServers` 数组来存放不同服务器的登录信息。`$cfgServers[1]["host"]` 包含了第一个服务器的主机名，`$cfgServers[2]["host"]` 为第二个服务器的主机等，等等。如果你只有一个服务器要管理，可以简单地不去理会其他 `$cfgServers` 入口的主机名。

`$cfgServers[n]["port"]` 字符串

第 n 个 MySQL 服务器的端口号。默认值为 3300。

`$cfgServers[n]["host"]` 字符串

第 n 个 MySQL 服务器的主机名。例如，localhost。

`$cfgServers[n]["adv_auth"]` 布尔值

对这个服务器应该使用基本或是高级认证方式。基本认证方式(`$adv_auth = false`)是普通的老的作法：用户名和口令被存储在 `config.inc.php3` 中。高级认证方式(`$adv_auth = true`)从 1.3.0 版开始引入，允许你通过 HTTP-Auth 来作为合法的 MySQL 的用户进行登录。在 `config.inc` 中你只需要提供一个标准用户，他能够连接到 MySQL 上并且可以读出 mysql 库的 user/db 表。

`$cfgServers[n]["user"]` 字符串

`$cfgServers[n]["password"]` 字符串

当使用基本认证方式时，PHPMyAdmin 将使用用户名/口令与这个 MySQL 服务器连接。当使用高级认证方式时则不需要。

`$cfgServers[n]["stduser"]` 字符串

`$cfgServers[n]["stdpass"]` 字符串

当使用高级认证方式时，用户名/口令对被用于校验真正的用户名/口令对。这个用户必须能够连接 MySQL，而且可以读取 mysql 库的 user 表。当使用基本认证方式时则不需要。

\$cfgServers[n]["only_db"] 字符串

如果设置了一个数据库名，只有这个数据库将显示给用户。

\$cfgServers[n]["verbose"] 字符串

只有在多服务器入口时使用 PHPMyAdmin 才有用。如果设置了，这个字符串将被显示出来，用来代替在主页面中的下接菜单中的主机名。

\$cfgManualBase 字符串

如果设为一个 URL，就会创建相应的帮助链接。

\$cfgPersistentConnections 布尔值

是否使用持续链接。

\$cfgConfirm 布尔值

当你将要丢失数据时是否应该显示一个警告信息。

\$cfgMaxRows 整数

当浏览一个结果集时显示的记录数。如果结果集包含了更多的数据，将显示前页/后页的链接。

\$cfgMaxInputsize 整数

当向一个表增加一条新的记录时，编辑字段的大小。

\$cfgBorder 整数

表格边界的大小。

\$cfgThBgcolor 字符串 [HTML 颜色]

用在表头的颜色。

\$cfgBgcolorOne 字符串 [HTML 颜色]

表格行第一行的颜色。

\$cfgBgcolorTwo 字符串 [HTML 颜色]

表格行第二行的颜色。

\$cfgOrder 字符串 ["DESC"] ["ASC"]

定义了当你点击字段名时，字段是以升序("ASC")显示还是以降序("DESC")显示。

\$cfgShowBlob 布尔值

定义了当浏览一个表的内容时，是否显示 BLOB 字段。

\$cfgShowSQL 布尔值

定义了是否显示 phpMyAdmin 所生成的 sql 查询语句。

\$cfgColumnTypes 数组

MySQL 列的所有可能的类型。大多数情况下你不需要编辑它。

\$cfgFunctions 数组

MySQL 支持函数的列表。大多数情况下你不需要编辑它。

\$cfgAttributeTypes 数组

字段可能的属性。大多数情况下你不需要编辑它。

那么怎样修改这些属性来消除错误信息呢？在这个文件中有两个字段比较重要，一个是

\$cfgServers[1]['user'] = 'root';

这个是 MySQL 的用户名，改写成可用的用户名，在这里我们使用默认的用户名 root 就

可以了。

另外一个就是下面的一个：

```
$cfgServers[1]['password'] = '';
```

这个是对应 MySQL 的用户名的密码，必须修改为你可用的密码，不然就不能访问，就会出现图 7-3 所示的情况。

我们设置的密码是 zz，修改语句如下：

```
$cfgServers[1]['password'] = 'zz';
```

这样再在地址栏上输入

http://127.0.0.1/phpMyAdmin/index.php

就会出现如图 7-4 所示的结果。

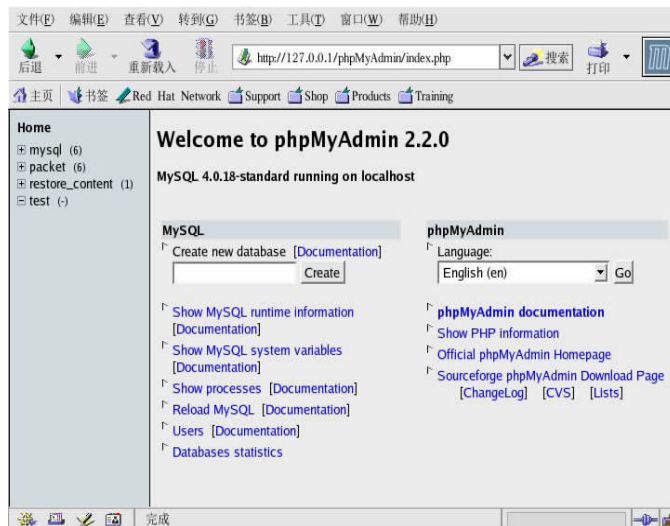


图 7-4 PhpMyAdmin 配置

现在就可以使用 PhpMyAdmin 来管理 MySql 数据库了。具体管理 MySql 数据库的操作都很简单，读者可以参考相关资料。

7.3.2 设计数据库

本系统采用 MySQL 作为存储数据库。在捕获到数据包的时候，就根据需要调用存储模块来存储网络数据包信息，这些信息是经过网络协议分析模块分析得到的，不是原始的字节流信息。这里使用的是推方法。在把数据包信息存储到数据库后，可以进行事后分析过程，对存储在数据库中的网络数据包进行分析，查找可疑事件或者进行网络流量分析等等。如分析 IP 流量，HTTP 协议内容，显示 HTML 文件等。这样可以提供更详细的网络性能描述，找出一些异常情况，以便及时查找原因所在。

在本系统中使用了两个数据库，一个是名字为 packet 的数据库，其功能是存放已经分析过的网络数据包信息。另一个是名字为 restore_content 的数据库，其功能是存放 HTTP 协议分析后的信息，分析后他可以把 HTTP 协议访问的页面给显示出来。下面详细介绍怎样创建本系统的 packet 数据库。

使用 create database 命令来创建数据库。

以下是在 pakcet 数据库中创建数据表的 MySQL 语句。

(1) 创建 arp 表

此表的内容是存放分析 ARP 协议后的数据包信息。

```
CREATE TABLE arp (
  packetnumber char(15) NOT NULL default "",
  hrd char(10) NOT NULL default "",
  pro char(10) NOT NULL default "",
  hln char(10) NOT NULL default "",
  plen char(10) NOT NULL default "",
  op char(10) NOT NULL default "",
  PRIMARY KEY (packetnumber),
  UNIQUE KEY packetnumber (packetnumber)
) TYPE=MyISAM;
```

创建后的结果如图 7-5 所示。

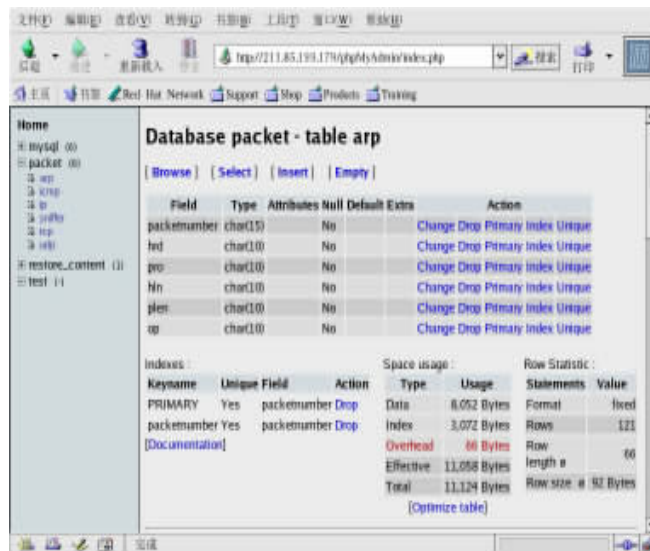


图 7-5 创建 arp 表

(2) 创建 icmp 表

此表用来存储分析 ICMP 协议后的信息。

```
CREATE TABLE icmp (
  packetnumber char(15) NOT NULL default "",
  type char(10) NOT NULL default "",
  code char(10) NOT NULL default "",
  cksum char(10) NOT NULL default "",
  PRIMARY KEY (packetnumber),
  UNIQUE KEY packetnumber (packetnumber)
) TYPE=MyISAM;
```

创建后的画面如图 7-6 所示。

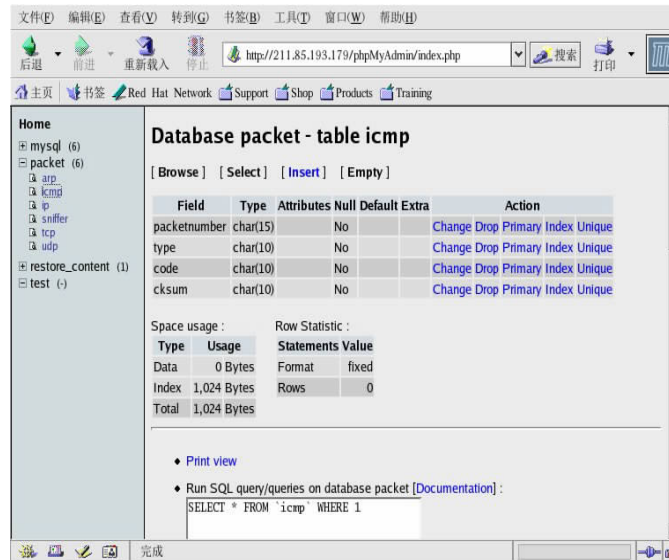


图 7-6 创建 icmp 表

(3) 创建 ip 表

此表是用来存储分析 IP 协议后的信息。

```
CREATE TABLE ip (
    packetnumber char(15) NOT NULL default "",
    version char(10) NOT NULL default "",
    headerlength char(10) NOT NULL default "",
    tos char(10) NOT NULL default "",
    totallength char(10) NOT NULL default "",
    id char(10) NOT NULL default "",
    off char(10) NOT NULL default "",
    ttl char(10) NOT NULL default "",
    protocol char(10) NOT NULL default "",
    checksum char(10) NOT NULL default "",
    sourceip char(15) NOT NULL default "",
    destinationip char(15) NOT NULL default "",
    PRIMARY KEY (packetnumber),
    UNIQUE KEY packetnumber (packetnumber)
) TYPE=MyISAM;
```

创建后的画面如图 7-7 所示。

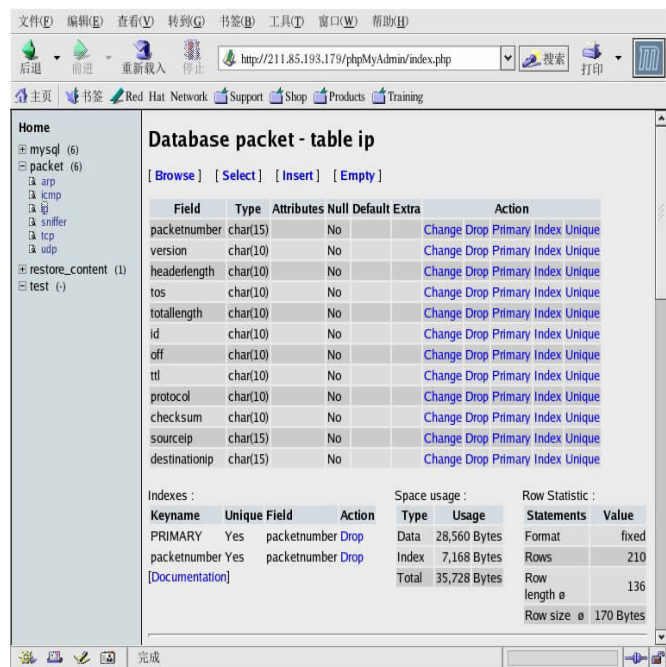


图 7-7 创建 ip 表

(4) 创建 tcp 表

此表用来存放分析 TCP 协议后的信息。

```
CREATE TABLE tcp (
  packetnumber char(15) NOT NULL default "",
  sport char(10) NOT NULL default "",
  dport char(10) NOT NULL default "",
  seq char(15) NOT NULL default "",
  ack char(15) NOT NULL default "",
  doff char(10) NOT NULL default "",
  flags char(15) NOT NULL default "",
  win char(15) NOT NULL default "",
  cksum char(10) NOT NULL default "",
  urp char(10) NOT NULL default "",
  PRIMARY KEY (packetnumber),
  UNIQUE KEY packetnumber (packetnumber)
) TYPE=MyISAM;
```

创建后的画面如图 7-8 所示。

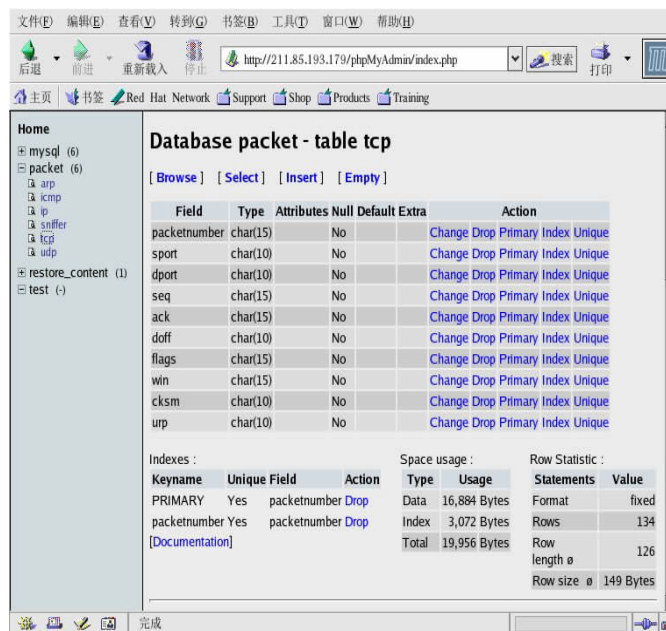


图 7-8 创建 tcp 表

(5) udp 表

此表用来存储分析 UDP 协议后的信息。

```
CREATE TABLE udp (
  packetnumber char(15) NOT NULL default "",
  sport char(10) NOT NULL default "",
  dport char(10) NOT NULL default "",
  len char(10) NOT NULL default "",
  cksum char(10) NOT NULL default "",
  PRIMARY KEY (packetnumber),
  UNIQUE KEY packetnumber (packetnumber)
) TYPE=MyISAM;
```

创建后的画面如图 7-9 所示。

(6) 创建 sniffer 表，此表是存放所有数据包的大体信息的。其创建语句如下：

```
CREATE TABLE sniffer (
  packetnumber char(50) default NULL,
  time char(20) default NULL,
  sourceip char(15) default NULL,
  sourceport char(6) default NULL,
  destinationip char(15) default NULL,
  destinationport char(6) default NULL,
  ether_type char(10) default NULL
) TYPE=MyISAM
```

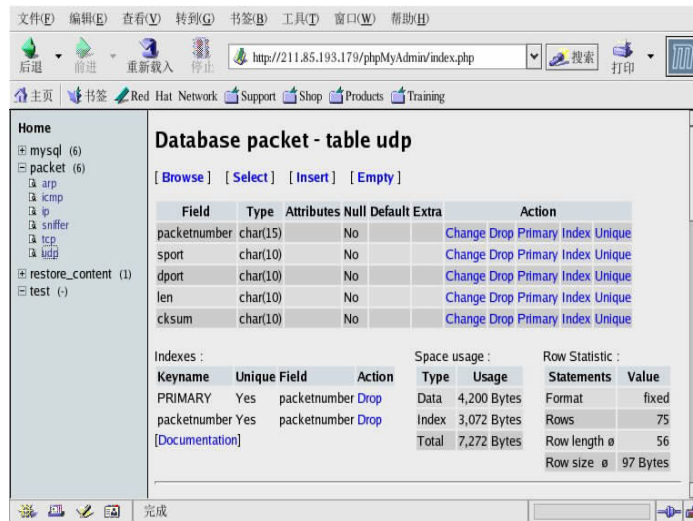


图 7-9 创建 UDP 表

7.3.3 实现数据库连接

实现数据库连接，我们使用的是 C 语言，所以就要用到 MySQL 数据库的 C 语言的 API 函数。实现数据库的连接，其基本思想是这样的，当有数据到达时，如果存储功能是要求的（这可以用一个开关语句来实现），那么就及时地把分析后的网络信息存放到数据库中。

此时要考虑存储信息到数据库的语句应放在哪里呢？当然是放在协议分析函数里面，当协议分析函数分析出协议信息之后，就马上调用存储信息的函数来把分析后的网络信息存储到数据库中。

以下是实现存储网络信息的相关函数：

```
insert_arp_into_database()
```

把 ARP 协议信息载入数据库。

```
insert_ip_into_database()
```

把 IP 协议信息载入数据库。

```
insert_tcp_into_database()
```

把 TCP 协议信息载入数据库。

```
insert_udp_into_database()
```

把 UDP 协议信息载入数据库。

```
insert_icmp_into_database()
```

把 ICMP 协议信息载入数据库。

下面来详细地介绍这些函数的实现。

(1) insert_arp_into_database()

此函数就是把捕获到的 ARP 数据包信息存入到数据库中。在函数 print_arp_header (struct arphdr *arp)中被调用。请参考 6.2.3 节。

```

void insert_arp_into_database(void)
{
    char *query_str; char str[1024];
    unsigned long packet_number_local;
    char number_string_local[1024];
    mysql_init(&mysql);
    //初始化数据库。
    if(!(mysql_real_connect(&mysql,"localhost","root","zz",NULL,3306,"/var/lib/mysql/mysql.sock",
    0)))
    //连接数据库服务器。
        exiterr(1);
    if (mysql_select_db(&mysql,"packet")) //连接数据库 packet。
        exiterr(2);
    packet_number_local=sniffer_rows_number+packetnumber;
    sprintf(number_string_local,"%lu",packet_number_local);
    sprintf(str,"insert into arp (packetnumber,hrd,pro,hln,plen,op) values
    (%s',%s',%s',%s',%s',%s')",number_string_local,
    arp_header_string_object.hrd,
    arp_header_string_object.pro,arp_header_string_object.hln,
    arp_header_string_object.plen,arp_header_string_object.op);
    if(mysql_real_query(&mysql,str,strlen(str))) exiterr(3);
    //数据库执行由 str 指定的 SQL 语句。
    mysql_close(&mysql);
    //关闭数据库连接。
}

```

(2) insert_ip_into_database()

此函数就是把捕获到的 IP 网络数据包存放到数据库中。此函数在函数 print_ip_header (struct ip *ip)中被调用。请参考 6.2.4 节。

```

void insert_ip_into_database(void)
{
    char *query_str;
    char str[1024];
    unsigned long packet_number_local;
    char number_string_local[1024];
    mysql_init(&mysql);//初始化数据库
    if(!(mysql_real_connect(&mysql,"localhost","root","zz",NULL,3306,"/var/lib/mysql/
    mysql.sock",0)))
    //连接数据库
        exiterr(1);
    if (mysql_select_db(&mysql,"packet"))
    //连接 packet 数据库

```

```

        exiterr(2);
    printf("insert a ip data \n");
    packet_number_local=sniffer_rows_number+packetnumber;
    sprintf(number_string_local,"%lu",packet_number_local);
    sprintf(str,"insert into ip (packetnumber, version, headerlength, tos,totallength, id, off, ttl, protocol,
checksum, sourceip, destinationip) values ('%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s', '%s',
'%s', '%s')", number_string_local, ip_header_string_object.version,
ip_header_string_object.header_length,ip_header_string_object.tos,
ip_header_string_object.total_length,ip_header_string_object.id,
ip_header_string_object.off,ip_header_string_object.ttl,
ip_header_string_object.protocol,ip_header_string_object.checksum,
ip_header_string_object.source_ip,ip_header_string_object.destination_ip);
    //构造 SQL 语句
    if(mysql_real_query(&mysql,str,strlen(str)))
    //执行 SQL 语句
        exiterr(3);
    mysql_close(&mysql);
    //关闭数据库连接
}

```

(3) insert_tcp_into_database()

此函数就是把捕获到的 TCP 网络数据包存放到数据库中。此函数在函数 print_tcp_header (struct tcphdr *tcp, int len, int wtp)中被调用。请参考 6.2.5 节。

```

void insert_tcp_into_database(void)
{
    char *query_str;
    char str[1024];
    unsigned long packet_number_local;
    char number_string_local[1024];
    mysql_init(&mysql);
    //初始化数据库
    if(!(mysql_real_connect(&mysql,"localhost","root","zz",NULL,3306,"/var/lib/
mysql/mysql.sock",0)))
    //连接数据库服务器
        exiterr(1);
    if (mysql_select_db(&mysql,"packet"))
    //打开 packet 数据库
        exiterr(2);
    printf("insert a arp data \n");
    packet_number_local=sniffer_rows_number+packetnumber;
    sprintf(number_string_local,"%lu",packet_number_local);
    sprintf(str,"insert into tcp (packetnumber,sport,dport,seq,ack,doff,flags,win,cksm,urp) values

```

```

        ("%s','%s','%s','%s','%s','%s','%s','%s','%s','%s')",number_string_local,
tcp_header_string_object.sport,tcp_header_string_object.dport,
tcp_header_string_object.seq,tcp_header_string_object.ack,
tcp_header_string_object.doff,tcp_header_string_object.flags,
tcp_header_string_object.win,
tcp_header_string_object.cksm,tcp_header_string_object.urp);
//填充 SQL 语句
if(mysql_real_query(&mysql,str,strlen(str)))
//执行 SQL 语句
        exiterr(3);
mysql_close(&mysql);
//关闭数据库连接
}

```

(4) insert_udp_into_database()

此函数是把捕获到的 UDP 网络数据包存放到数据库中。此函数在函数 print_udp_header (struct udphdr *udp, int len, int wtp)中被调用。请参考 6.2.6 节。

```

void insert_udp_into_database(void)
{
    char *query_str;
    char str[1024];
    unsigned long packet_number_local;
    char number_string_local[1024];
    mysql_init(&mysql);
    //初始化连接
    if(!(mysql_real_connect(&mysql,"localhost","root","zz",NULL,3306,"/var/lib/mysql/
mysql.sock",0)))
    //连接数据库连接
    exiterr(1);
    if (mysql_select_db(&mysql,"packet"))
    //选择 packet 数据库
        exiterr(2);
    printf("insert a udp data \n");
    packet_number_local=sniffer_rows_number+packetnumber;
    sprintf(number_string_local,"%lu",packet_number_local);
    sprintf(str,"insert into udp (packetnumber,sport,dport,len,cksum) values
        ("%s','%s','%s','%s','%s')",number_string_local,udp_header_string_object.sport,
udp_header_string_object.dport,udp_header_string_object.len,
udp_header_string_object.cksum);
    //填充 SQL 语句
    if(mysql_real_query(&mysql,str,strlen(str)))
    //执行 SQL 语句

```



```

        exiterr(3);
        mysql_close(&mysql);
        //关闭数据库连接
    }
}

```

(5) insert_icmp_into_database()

此函数把 ICMP 捕获到的网络数据包存放到数据库中。此函数在函数 print_icmp_header (struct icmp *icmp, int len, int wtp)中被调用。请参考 6.2.7 节。

```

void insert_icmp_into_database(void)
{
    char *query_str;
    char str[1024];
    unsigned long packet_number_local;
    char number_string_local[1024];
    mysql_init(&mysql);
    //初始化
    if(!(mysql_real_connect(&mysql,"localhost","root","zz",NULL,3306,"/var/lib/mysql/
mysql.sock",0)))
        //连接数据库
        exiterr(1);
    if (mysql_select_db(&mysql,"packet"))
        //选择 packet 数据库
        exiterr(2);
    printf("insert a icmp data \n");
    packet_number_local=sniffer_rows_number+packetnumber;
    sprintf(number_string_local,"%lu",packet_number_local);
    sprintf(str,"insert into icmp (packetnumber,type,code,cksum) values \
        (%s',%s',%s',%s')",number_string_local,icmp_header_string_object.type,
        icmp_header_string_object.code,icmp_header_string_object.cksum);
    //使用 icmp 信息来填充 SQL 语句
    if(mysql_real_query(&mysql,str,strlen(str)))
        //执行 SQL 语句
        exiterr(3);
    mysql_close(&mysql);
    //关闭数据库连接
}

```

7.4 数据库分析

建立存储系统的目的就是为存储信息以便以后进行进一步的分析。可以对数据库进行各种各样的分析操作，通过数据库分析可以得到很多有价值的信息，可以实现很多功能。例如，可以进行流量分析，数据包分布情况分析，协议大体分布情况分析，特定协议的内容分

析等等。下面我们只是实现一部分分析过程，读者还可以参考这些分析提出更多的分析过程，以实现更多的功能。

7.4.1 分析 IP 数据包的分布状态

通过对数据库的分析可以得到网络中 IP 网络数据包的分布情况。实际上也就是调用 SQL 语句来对数据库进行查询。然后就可以得到很多的分析结果，如可以分析 IP 源地址是某一个固定 IP 地址的数目是什么，它访问了哪些 IP 主机。它的访问时间是什么，访问的服务是什么，可以从端口进行查询。例如，如果端口是 80 就说明访问的是 WEB 服务器。这样就可以进行很多类型的分析。

在这里我们通过界面显示的手段来分析。

1. 获取源 IP 的分布情况

```
void
get_source_ip_from_database (char *result[], uint * number_p)
{
    MYSQL mysql_source_ip;
    MYSQL_FIELD *field_source_ip;
    MYSQL_ROW row_source_ip;
    MYSQL_RES *res_source_ip;
    uint i = 0;
    uint j = 0;
    char query_str[1024];
    char str_source_ip[1024];
    mysql_init (&mysql_source_ip);
    //初始化
    if (!(mysql_real_connect(&mysql_source_ip, "localhost", "root", "zz", NULL, 3306,
    "/var/lib/mysql/mysql.sock", 0))) //连接数据库服务器
    {
        exiterr (1);
        return;
    }
    if (mysql_select_db (&mysql_source_ip, "packet")) //打开 packet 数据库
        exiterr (2);
    strcpy (query_str, "SELECT sourceip FROM ip"); //生成 SQL 语句
    if (mysql_real_query (&mysql_source_ip, query_str, strlen (query_str)))
        //执行 SQL 语句
        exiterr (3);
    if (!(res_source_ip = mysql_store_result (&mysql_source_ip)))
        exiterr (4);
    *number_p = mysql_num_rows (res_source_ip);
    printf ("number_p is %d\n", *number_p);
```

```
while ((row_source_ip = mysql_fetch_row (res_source_ip)))
{
    //获取 ip 数据表中的记录
    for (i = 0; i < mysql_num_fields (res_source_ip); i++)
    {
        printf ("%s      ", row_source_ip[i]);
        printf ("haha\n ");
        result[j] = row_source_ip[i];
        printf ("j is %d\n", j);
        printf ("%s", result[j]);
        printf ("2 \n");
        printf ("\n");
    }
    j++;
}
mysql_free_result (res_source_ip);
//释放
mysql_close (&mysql_source_ip);
//关闭数据库连接
}

void
source_ip_item_expand (GtkWidget * item, gchar * signame)
{
    //此函数时当选项展开时显示源 IP 地址信息
    printf ("haha gchar0 \n");
    uint i = 0;
    uint number_packet = 0;
    char *str[10000];
    int k = 1;
    int j;
    int same = 1;
    printf ("haha gchar1 \n");
    gchar string_local_single[10000][20];
    gchar string_local[10000][20];
    GtkWidget *box_xpm;
    GtkWidget *dst_ip_subitem;
    GtkWidget *dst_ip_subtree;
    get_source_ip_from_database (str, &number_packet);
    //从数据库中获取 ip 协议信息
    uint my_number_haha = number_packet;
    for (i = 0; i < my_number_haha; i++)
    {
```

```
        sprintf (string_local[i], "%s", str[i]);
        printf ("%d\n", i);
    }
    sprintf (string_local_single[0], "%s", string_local[0]);
    k = 1;
    for (i = 0; i < my_number_haha; i++)
    {
        printf ("i : %d \n", i);
        for (j = 0; j < k; j++)
        {
            if (strcmp (string_local_single[j], string_local[i]) == 0)
            {
                printf ("==\n");
                same = 1;
                break;
            }
            else
                same = 0;
        }
        if (same == 0)
        {
            sprintf (string_local_single[k], "%s", string_local[i]);
            k++;
        }
    }
    dst_ip_subtree =
    gtk_object_get_data (GTK_OBJECT (analysize_window), "source_ip_subtree");
    for (i = 0; i < k; i++)
    {
        dst_ip_subitem = gtk_tree_item_new ();
        box_xpm =
        xpm_label_box (analysize_window, "/xpm/scanning.xpm",
            string_local_single[i]);
        gtk_widget_show (box_xpm);
        gtk_container_add (GTK_CONTAINER (dst_ip_subitem), box_xpm);
        gtk_signal_connect (GTK_OBJECT (dst_ip_subitem), "select",
            GTK_SIGNAL_FUNC (source_ip_selected),
            string_local_single[i]);
        gtk_tree_append (GTK_TREE (dst_ip_subtree), dst_ip_subitem);
        gtk_widget_show (dst_ip_subitem);
    }
}
```

```
void
source_ip_item_collapse (GtkWidget * item, gchar * signame)
{
    //此函数时当选项折叠时被调用。
    uint i;
    uint number_packet = 0;
    uint my_number = 0;
    char *str[10000];
    gchar string_local[10000][20];
    GtkWidget *tree;
    GtkWidget *box_xpm;
    GtkWidget *dst_ip_subitem;
    GtkWidget *dst_ip_subtree;
    GtkWidget *dst_ip_item;
    get_source_ip_from_database (str, &number_packet);
    //从数据库中获取 ip 协议信息
    my_number = number_packet;
    for (i = 0; i < my_number; i++)
    {
        sprintf (string_local[i], "%s", str[i]);
        printf ("before4 %s \n", str[i]);
    }
    tree = gtk_object_get_data (GTK_OBJECT (analysize_window), "tree");
    gtk_tree_clear_items (GTK_TREE (tree), 0, my_number);
    dst_ip_item = gtk_tree_item_new ();
    gtk_object_set_data (GTK_OBJECT (analysize_window), "source_ip_item",
        dst_ip_item);
    gtk_signal_connect (GTK_OBJECT (dst_ip_item), "expand",
        GTK_SIGNAL_FUNC (source_ip_item_expand), NULL);
    gtk_signal_connect (GTK_OBJECT (dst_ip_item), "collapse",
        GTK_SIGNAL_FUNC (source_ip_item_collapse), NULL);
    box_xpm =
    xpm_label_box (analysize_window, "./xpm/database.xpm", "Destination IP");
    gtk_widget_show (box_xpm);
    gtk_container_add (GTK_CONTAINER (dst_ip_item), box_xpm);
    gtk_tree_append (GTK_TREE (tree), dst_ip_item);
    gtk_widget_show (dst_ip_item);
    dst_ip_subtree = gtk_tree_new ();
    gtk_object_set_data (GTK_OBJECT (analysize_window), "source_ip_subtree",
        dst_ip_subtree);
    gtk_tree_set_selection_mode (GTK_TREE (dst_ip_subtree),
        GTK_SELECTION_SINGLE);
}
```

```

gtk_tree_set_view_mode(GTK_TREE(dst_ip_subtree),
GTK_TREE_VIEW_ITEM);
gtk_tree_item_set_subtree (GTK_TREE_ITEM (dst_ip_item), dst_ip_subtree);
}

```

图 7-10 是显示源 IP 的分析情况。

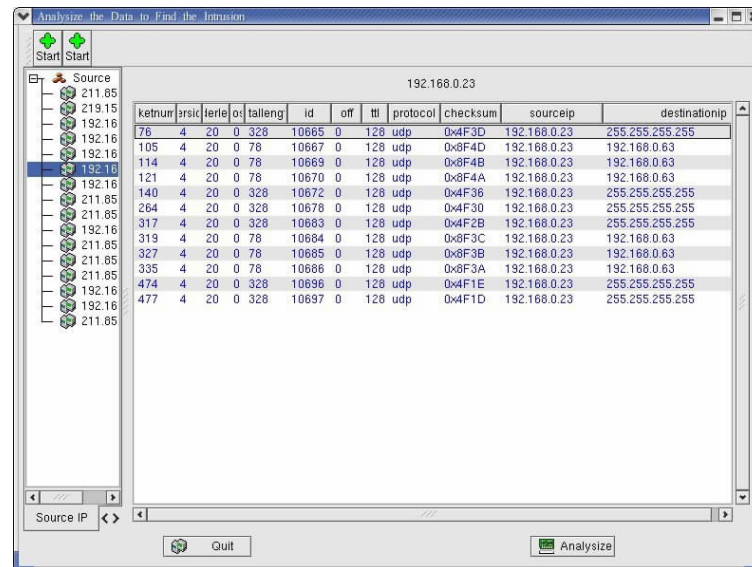


图 7-10 显示源 IP 的分布情况

2. 获取目的 IP 的分布情况

```

void
get_destination_ip_from_database (char *result[], uint * number_p)
{
    MYSQL mysql_dst_ip;
    MYSQL_FIELD *field_dst_ip;
    MYSQL_ROW row_dst_ip;
    MYSQL_RES *res_dst_ip;
    uint i = 0;
    uint j = 0;
    char query_str[1024];
    char str_dst_ip[1024];
    mysql_init (&mysql_dst_ip);
    //初始化
    if (!(mysql_real_connect
        (&mysql_dst_ip, "localhost", "root", "zz", NULL, 3306,
         "/var/lib/mysql/mysql.sock", 0))) //连接数据库服务器
    {
        exiterr (1);
    }
}

```

```

        return;
    }
    if (mysql_select_db (&mysql_dst_ip, "packet")) //打开 packet 数据库
        exiterr (2);
    strcpy (query_str, "SELECT destinationip FROM ip"); //生成 SQL
    if (mysql_real_query (&mysql_dst_ip, query_str, strlen (query_str)))
        exiterr (3); //执行 SQL
    if (!(res_dst_ip = mysql_store_result (&mysql_dst_ip)))
        exiterr (4);
    *number_p = mysql_num_rows (res_dst_ip);
    while ((row_dst_ip = mysql_fetch_row (res_dst_ip)))
    {
        //读取 ip 表中的记录
        for (i = 0; i < mysql_num_fields (res_dst_ip); i++)
        {
            printf ("%s\t", row_dst_ip[i]);
            printf ("1\n ");
            result[j] = row_dst_ip[i];
            printf ("%s", result[j]);
            printf ("2\n");
            printf ("\n");
        }
        j++;
    }
    mysql_free_result (res_dst_ip);
    mysql_close (&mysql_dst_ip);
}

void
dst_ip_item_expand (GtkWidget * item, gchar * signame)
{
    //此函数是当选项折叠时被调用。
    uint i;
    uint number_packet = 0;
    uint my_number = 0;
    char *str[10000];
    int k = 1;
    int j;
    int same = 1;
    gchar string_local_single[10000][20];
    gchar string_local[10000][20];
    GtkWidget *box_xpm;
    GtkWidget *dst_ip_subitem;

```

```
GtkWidget *dst_ip_subtree;
get_destination_ip_from_database (str, &number_packet);
// 从数据库中获取目的 IP 信息。
my_number = number_packet;
for (i = 0; i < my_number; i++)
{
    sprintf (string_local[i], "%s", str[i]);
    printf ("before1 %s \n", str[i]);
}
sprintf (string_local_single[0], "%s", string_local[0]);
k = 1;
for (i = 0; i < my_number; i++)
{
    printf ("i : %d \n", i);
    for (j = 0; j < k; j++)
    {
        if (strcmp (string_local_single[j], string_local[i]) == 0)
        {
            printf ("==\n");
            same = 1;
            break;
        }
        else
            same = 0;
    }
    if (same == 0)
    {
        sprintf (string_local_single[k], "%s", string_local[i]);
        k++;
        printf ("k is :%d\n", k);
    }
}
printf ("the k %d \n", k);
dst_ip_subtree =
gtk_object_get_data (GTK_OBJECT (analysize_window), "dst_ip_subtree");
for (i = 0; i < k; i++)
{
    dst_ip_subitem = gtk_tree_item_new ();
    box_xpm =
xpm_label_box (analysize_window, "/xpm/scanning.xpm",
                string_local_single[i]);
    gtk_widget_show (box_xpm);
```



```

        gtk_container_add (GTK_CONTAINER (dst_ip_subitem), box_xpm);
        gtk_signal_connect (GTK_OBJECT (dst_ip_subitem), "select",
            GTK_SIGNAL_FUNC (dst_ip_selected),
            string_local_single[i]);
        gtk_tree_append (GTK_TREE (dst_ip_subtree), dst_ip_subitem);
        gtk_widget_show (dst_ip_subitem);
    }
}

void
dst_ip_item_collapse (GtkWidget * item, gchar * signame)
{
    //此函数是当选项展开时显示的信息。
    uint i;
    uint number_packet = 0;
    uint my_number = 0;
    char *str[10000];
    gchar string_local[10000][20];
    GtkWidget *tree;
    GtkWidget *box_xpm;
    GtkWidget *dst_ip_subitem;
    GtkWidget *dst_ip_subtree;
    GtkWidget *dst_ip_item;
    get_destination_ip_from_database (str, &number_packet);
    //从数据库中获取目的 IP 信息
    my_number = number_packet;
    for (i = 0; i < my_number; i++)
    {
        sprintf (string_local[i], "%s", str[i]);
        printf ("before6 %s \n", str[i]);
    }
    tree = gtk_object_get_data (GTK_OBJECT (analyze_window), "tree");
    gtk_tree_clear_items (GTK_TREE (tree), 0, my_number);
    dst_ip_item = gtk_tree_item_new ();
    gtk_object_set_data (GTK_OBJECT (analyze_window), "dst_ip_item",
        dst_ip_item);
    gtk_signal_connect (GTK_OBJECT (dst_ip_item), "expand",
        GTK_SIGNAL_FUNC (dst_ip_item_expand), NULL);
    //回调函数 dst_ip_item_expand ( )
    gtk_signal_connect (GTK_OBJECT (dst_ip_item), "collapse",
        GTK_SIGNAL_FUNC (dst_ip_item_collapse), NULL);
    //回调函数 dst_ip_item_collapse ( )
    box_xpm =

```

```

xpm_label_box (analysize_window, "/xpm/database.xpm", "Destination IP");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (dst_ip_item), box_xpm);
gtk_tree_append (GTK_TREE (tree), dst_ip_item);
gtk_widget_show (dst_ip_item);
dst_ip_subtree = gtk_tree_new ();
gtk_object_set_data (GTK_OBJECT (analysize_window), "dst_ip_subtree",
    dst_ip_subtree);
gtk_tree_set_selection_mode (GTK_TREE (dst_ip_subtree),
    GTK_SELECTION_SINGLE);
gtk_tree_set_view_mode (GTK_TREE (dst_ip_subtree),
    GTK_TREE_VIEW_ITEM);
gtk_tree_item_set_subtree (GTK_TREE_ITEM (dst_ip_item), dst_ip_subtree);
}

```

图 7-11 是显示目的 IP 分布情况的运行界面。

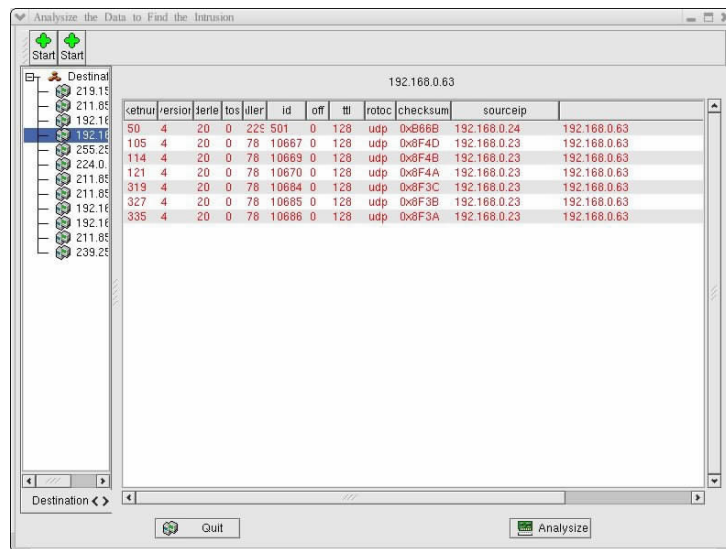


图 7-11 分析目的 IP 的分布情况

7.4.2 分析总体协议的分布状态

由于网络中的协议类型是很多的，如 ARP，IP，TCP，UDP 等。所以我们想知道数据包是基于什么协议的，它们的分布情况如何。例如，我们想知道一段时间中，捕获到的网络数据包中 IP 协议占多少，并且其分布情况如何，有哪些数据包是基于 IP 协议的，并且这些数据包是怎样进行传送的，它们的源 IP 地址，目的 IP 地址是什么，上一层协议是什么等等。这些分析都可以实现的。

图 7-12 是显示 IP 协议分布情况的运行界面，也可以分析 ARP，TCP，UDP 等协议的分布情况。下面我们来具体实现这个操作。此功能由函数 create_tabel_clist () 来完成，它在一

个回调函数中被调用。

ketrnum	version	saderleng	tos	totallength	id	off	ttl	protocol
33	4	20	0	1500	46656	16384	108	tcp
34	4	20	0	52	60152	16384	64	tcp
35	4	20	0	1500	46657	16384	108	tcp
36	4	20	0	52	60153	16384	64	tcp
37	4	20	0	1500	46658	16384	108	tcp
38	4	20	0	52	60154	16384	64	tcp
39	4	20	0	1500	46659	16384	108	tcp
40	4	20	0	52	60155	16384	64	tcp
41	4	20	0	1500	46670	16384	108	tcp
42	4	20	0	52	60156	16384	64	tcp
43	4	20	0	1500	46671	16384	108	tcp
50	4	20	0	229	501	0	128	udp
75	4	20	0	276	959	0	128	udp
76	4	20	0	328	10665	0	128	udp
98	4	20	0	47	115	0	2	udp
105	4	20	0	78	10667	0	128	udp
114	4	20	0	78	10669	0	128	udp
121	4	20	0	78	10670	0	128	udp
123	4	20	0	78	45104	0	128	udp
131	4	20	0	78	45105	0	128	udp
138	4	20	0	78	45106	0	128	udp
139	4	20	0	276	959	0	128	udp
140	4	20	0	328	10672	0	128	udp
144	4	20	0	647	23619	16384	64	tcp
151	4	20	0	1500	6474	16384	126	tcp

图 7-12 显示协议分布情况

GtkWidget *

create_tabel_clist (gchar * table_name)

```
{
    GtkWidget *vbox;
    GtkWidget *label;
    GtkWidget *scrolled_window;
    GtkWidget *clist;
    GtkWidget *label_column;
    MYSQL con;
    MYSQL_FIELD *field;
    MYSQL_ROW db_row;
    MYSQL_RES *result_set;
    guint counter;
    guint cols;
    char query_str[1024];
    gchar *row[20] = { "", "", "", "", "", "", "", "", "", "",
        "", "", "", "", "", "", "", "", "", "" };
    int j = 0;
    GdkColor color;
    GdkColor color_1;
    gdk_color_parse ("grey90", &color);
    gdk_colormap_alloc_color (gdk_colormap_get_system (), &color, FALSE, TRUE);
    gdk_color_parse ("blue1", &color_1);
    gdk_colormap_alloc_color (gdk_colormap_get_system (), &color_1, FALSE,
```

```
        TRUE);
vbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
label = gtk_label_new (table_name);
gtk_box_pack_start (GTK_BOX (vbox), label, FALSE, FALSE, 5);
gtk_widget_show (label);
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW
(scrolled_window),GTK_POLICY_AUTOMATIC,  GTK_POLICY_ALWAYS);
gtk_box_pack_start (GTK_BOX (vbox), scrolled_window, TRUE, TRUE, 5);
gtk_widget_show (scrolled_window);
clist = gtk_clist_new (20);
//建立列表控件
gtk_clist_set_shadow_type (GTK_CLIST (clist), GTK_SHADOW_IN);
gtk_container_add (GTK_CONTAINER (scrolled_window), clist);
gtk_widget_show (clist);
mysql_init (&con); //初始化
if (!
    (mysql_real_connect
    (&con, "localhost", "root", "zz", NULL, 3306,
    "/var/lib/mysql/mysql.sock", 0)))
//建立数据库服务器连接
{
    exiterr (1);
    return;
}
if (mysql_select_db (&con, "packet"))
//选择数据库 packet
    exiterr (2);
sprintf (query_str, "select * from %s", table_name);
//建立 SQL 语句
if (mysql_real_query (&con, query_str, strlen (query_str)))
//执行 SQL 语句
    exiterr (3);
if (!(result_set = mysql_store_result (&con)))
//保存结果
    exiterr (4);
cols = mysql_num_fields (result_set);
//得到结果的数量，即行的数量
for (counter = 0; counter < cols; counter++)
{
    //把信息添加到列表中
    mysql_field_seek (result_set, counter);
```

```

        field = mysql_fetch_field (result_set);
        label_column = gtk_label_new (field->name);
        gtk_widget_show (label_column);
        gtk_clist_set_column_widget (GTK_CLIST (clist), counter, label_column);
        gtk_clist_set_column_width (GTK_CLIST (clist), counter, 80);
    }
    gtk_clist_column_titles_show (GTK_CLIST (clist));
    while ((db_row = mysql_fetch_row (result_set)) != 0L)
    {
        for (counter = 0; counter < cols; counter++)
        {
            row[counter] = db_row[counter];
        }
        gtk_clist_append (GTK_CLIST (clist), row);
        if (j % 2 == 0)
        {
            gtk_clist_set_background ((GtkCLIST *) clist, j, &color);
            gtk_clist_set_foreground ((GtkCLIST *) clist, j, &color_1);
            j++;
        }
    }
    return vbox;
}

```

另外，我们可以分析网络数据包中的协议相对分布情况，例如，在网络数据包中 IP 协议和 ARP 协议的比例是多少，可以使用饼状图来显示其比例，如图 7-13 所示。

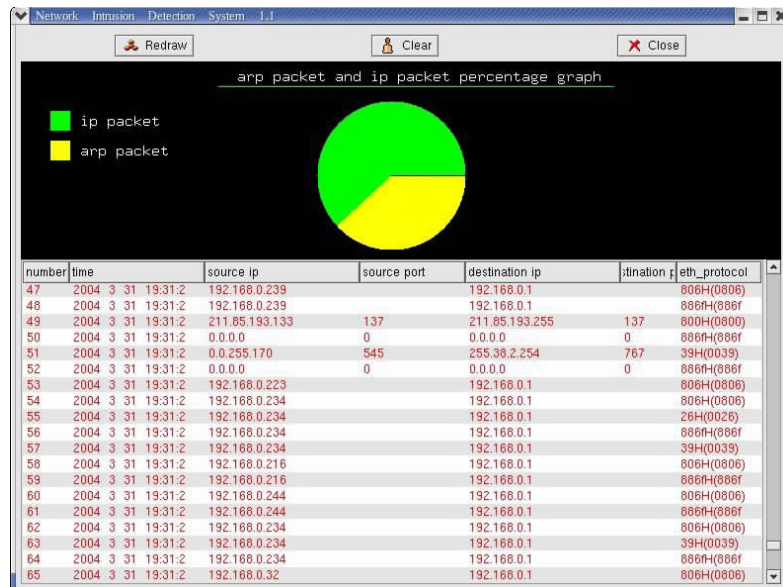


图 7-13 流量分析结果

其实现主要由函数 show_table_sniffer () 和 draw_test () 来完成。

函数 show_table_sniffer () 是实现数据库连接的。其实现如下：

```
void
show_table_sniffer (void)
{
    uint i = 0;
    uint flags = 0;
    static gint j = 0;
    char query_str[1024];
    char str[1024];
    GtkWidget *list;
    GdkColor color;
    GdkColor color_1;
    gchar *drink[6];
    gdk_color_parse ("grey91", &color);
    gdk_colormap_alloc_color (gdk_colormap_get_system (), &color, FALSE, TRUE);
    gdk_color_parse ("blue violet", &color_1);
    gdk_colormap_alloc_color (gdk_colormap_get_system (), &color_1, FALSE,
        TRUE);
    list = gtk_object_get_data (GTK_OBJECT (window_database), "clist");
    mysql_init (&mysql);
    //初始化
    if (! (mysql_real_connect (&mysql, "localhost", "root", "zz", NULL, 3306,
        "/var/lib/mysql/mysql.sock", 0)))
    //连接数据库服务器
    {
        exiterr (1);
        return;
    }
    if (mysql_select_db (&mysql, "packet"))
    //选择数据库
        exiterr (2);
    strcpy (query_str, "SELECT * FROM sniffer");
    //建立 SQL 语句，语句功能是从 sniffer 表中查询所有信息
    if (mysql_real_query (&mysql, query_str, strlen (query_str)))
    //执行 SQL 语句
        exiterr (3);
    if (!(res = mysql_store_result (&mysql)))
    //存储执行结果
        exiterr (4);
    sniffer_rows_number = mysql_num_rows (res);
    //共有多少数据包
    printf ("Number of rows: %lu\n", (unsigned long) sniffer_rows_number);
    while ((row = mysql_fetch_row (res)))
```

```

{
    for (i = 0; i < mysql_num_fields (res); i++)
    {
        drink[i] = row[i];
    }
    gtk_clist_append ((GtkCList *) list, drink);
    //添加数据信息到列表中
    if (j % 2 == 0)gtk_clist_set_background ((GtkCList *) list, j, &color);
    gtk_clist_set_foreground ((GtkCList *) list, j, &color_1);
    j++;
}
mysql_free_result (res);
//释放结果
strcpy (query_str, "SELECT * FROM sniffer where ether_type='806H(0806)'");
//建立 SQL 语句，语句功能是从 sniffer 表中查询所有 ARP 协议信息
if (mysql_real_query (&mysql, query_str, strlen (query_str)))
//执行 SQL 语句
    exiterr (3);
if (!(res = mysql_store_result (&mysql)))
//存储结果
    exiterr (4);
arp_number = 0;
while ((row = mysql_fetch_row (res)))
{
    arp_number++;
    //得到 ARP 数据包的数量
}
mysql_free_result (res);
//释放结果
strcpy (query_str, "SELECT * FROM sniffer where ether_type='800H(0800)'");
//建立 SQL 语句，语句功能是从 sniffer 表中查询所有 IP 协议信息
if (mysql_real_query (&mysql, query_str, strlen (query_str)))
//执行 SQL 语句
    exiterr (3);
if (!(res = mysql_store_result (&mysql)))
//存储结果
    exiterr (4);
ip_number = 0;
while ((row = mysql_fetch_row (res)))
{
    ip_number++;
    //获得 IP 数据包的数量
}

```

```

mysql_free_result(res);
//释放结果
printf("ip is :%d\n", ip_number);
printf("arp is :%d\n", arp_number);
mysql_close(&mysql);
//关闭数据库连接
}

```

函数 draw_test () 是实现化饼状图形的。其实现如下：

```

void
draw_test(GtkWidget *item, gchar *signame)
{
    GtkWidget *draw;
    GdkFont *fixed_font;
    GdkRectangle draw_area_total;
    GtkWidget *list;
    int j = 0;
    GdkColor color;
    GdkColor color_1;
    gdk_color_parse("grey90", &color);
    gdk_colormap_alloc_color(gdk_colormap_get_system(), &color, FALSE, TRUE);
    gdk_color_parse("red", &color_1);
    gdk_colormap_alloc_color(gdk_colormap_get_system(), &color_1, FALSE,
        TRUE);
    list = gtk_object_get_data(GTK_OBJECT(window_database), "clist");
    for(j = 0; j < sniffer_rows_number; j++)
    {
        if(j % 2 == 0)
            gtk_clist_set_background((GtkCList *) list, j, &color);
            gtk_clist_set_foreground((GtkCList *) list, j, &color_1);
    }
    draw = gtk_object_get_data(GTK_OBJECT(window_database), "drawing_area");
    draw_area_total.x = 0;
    draw_area_total.y = 0;
    draw_area_total.width = draw->allocation.width;
    draw_area_total.height = draw->allocation.height;
    fixed_font = gdk_font_load("-misc-fixed-medium-r-*-*140-*-*-*-*");
    //设置字体
    gdk_draw_rectangle(pixmap, get_gc("green"), TRUE, 30, 50, 20, 20);
    //画矩形
    gdk_draw_string(pixmap, fixed_font, get_gc("white"), 60, 65, "ip packet");
    //画字符串
    gdk_draw_rectangle(pixmap, get_gc("yellow"), TRUE, 30, 80, 20, 20);
    //画矩形
}

```



```

gdk_draw_string (pixmap, fixed_font, get_gc ("white"), 60, 95, "arp packet");
//画字符串
gdk_draw_string (pixmap, fixed_font, get_gc ("white"),
    210, 20, " arp packet and ip packet percentage graph ");
gdk_draw_line (pixmap, get_gc ("green"), 200, 25, 600, 25);
//画直线
printf (" draw test ip is :%d\n", ip_number);
printf ("draw test arp is :%d\n", arp_number);
gdk_draw_arc (pixmap, get_gc ("green"), TRUE, 300, 40, 150, 150, 0,
    ip_number / (ip_number + arp_number) * 360 * 64);
//画表示 IP 信息的圆饼
gdk_draw_arc (pixmap, get_gc ("yellow"), TRUE, 300, 40, 150, 150,
    ip_number / (ip_number + arp_number) * 360 * 64,
    (360 - ip_number / (ip_number + arp_number) * 360) * 64);
//画表示 ARP 信息的圆饼
gtk_widget_draw (draw, &draw_area_total);
}

```

7.4.3 HTTP 流量分析

在此系统中，我们着重分析了 HTTP 协议。通过对 HTTP 协议的分析可以查看访问网页的过程。通过对 HTTP 的分析，把里面的协议内容存储到数据库中，然后再还原成 HTML 网页，就可以查看网络中谁访问了哪些网页。

图 7-14 是显示 HTTP 协议信息的运行结果。

time	source_ip	destination_ip	sport	dport	tination_hostn	url
2004 4 26 1	192.168.0.36	192.168.0.34	1668	80		
2004 3 31 1	211.85.193.17	202.205.11.70	2554	80		
2004 3 31 1	211.85.193.17	202.205.11.70	2555	80		
2004 4 26 1	192.168.0.36	192.168.0.34	1701	80		
2004 4 26 1	192.168.0.36	192.168.0.34	1727	80		
2004 4 26 1	192.168.0.36	192.168.0.34	1665	80		

图 7-14 HTTP 协议流量分析

首先建立 restore_content 数据库，然后建立数据表如下：

```
CREATE TABLE http (
    time char(20) NOT NULL default "",
    source_ip char(15) NOT NULL default "",
    destination_ip char(15) NOT NULL default "",
    sport char(10) NOT NULL default "",
    dport char(10) NOT NULL default "",
    destination_hostname char(50) NOT NULL default "",
    url char(100) NOT NULL default "",
    data_type char(50) NOT NULL default "",
    data_length char(20) NOT NULL default ""
) TYPE=MyISAM
```

存储 HTTP 协议信息的功能由下面这个函数来实现：

```
void
insert_http_content_into_database (void)
{
    //此函数在函数 tcp_callback_http 中被调用，请参考 6.3.3 节。
    MYSQL mysql_content;
    MYSQL_FIELD *field_content;
    MYSQL_ROW row_content;
    MYSQL_RES *result_set_content;
    uint i = 0;
    char *query_str;
    char str[1024];
    unsigned long packet_number_local;
    char number_string_local[1024];
    mysql_init (&mysql_content);
    //初始化
    if (!
        (mysql_real_connect
         (&mysql_content, "localhost", "root", "zz", NULL, 3306,
          "/var/lib/mysql/mysql.sock", 0)))
        //连接数据库服务器
        exiterr (1, mysql_content);
    if (mysql_select_db (&mysql_content, "restore_content"))
        //连接 restore_content 数据库
        exiterr (2, mysql_content);
    printf ("insert a http_content data \n");
    sprintf(str,"insert into http( time, source_ip, destination_ip, sport, dport, destination_hostname, url,
    data_type, data_length) values \
    ('%s','%s','%s','%s','%s','%s','%s','%s','%s')",
    http_packet.time, http_packet.source_ip, http_packet.destination_ip, http_packet.sport,
```

```

    http_packet.dport, http_packet.destination_hostname, http_packet.url, http_packet.data_type,
    http_packet.data_length);
//用 http 信息填充 SQL 语句
if (mysql_real_query (&mysql_content, str, strlen (str)))
//执行 SQL 语句
    exiterr (3);
mysql_close (&mysql_content);
//关闭数据库连接
}

```

当要显示网页的内容时，就调用如下函数来实现：

```

void
restore_content_button_click (GtkWidget * widget, gchar * signame)
{
    // “ restore content ” 按钮的回调函数，如图 7-14
    char *tmpbuf;
    GtkWidget *clist;
    gchar *column_text;
    gchar *column_text_destip;
    tmpbuf = malloc (1024);
    clist = gtk_object_get_data (GTK_OBJECT (restore_content_window), "clist");
    gtk_clist_get_text (GTK_CLIST (clist), content_clist_selected_row, 3,
        &column_text);
    gtk_clist_get_text (GTK_CLIST (clist), content_clist_selected_row, 2,
        &column_text_destip);
    sprintf (tmpbuf,
        "mozilla file:/root/linux_ids/ids_text1/192.168.0.133/%s/%s.html",
        column_text_destip, column_text);
    system (tmpbuf);
    //运行浏览器程序
    free (tmpbuf);
}

```

可以看到本函数就是用 mozilla 浏览器来打开一个文件，此文件是在程序中动态生成的，是在分析 HTTP 协议的基础上生成的。请参考 6.4.3 节。

在这里我们来做个实验，在 IP 地址为 192.168.0.34 的 PC 上开启 Web 服务器，运行一个测试页面，在另外一个 IP 地址为 192.168.0.36 的 PC 上访问此 Web 服务器，如图 7-15 所示。注意此时浏览器的地址栏里的内容如下：

http://192.168.0.34/index..html

在此窗口中，我们可以看到所有 HTTP 协议的连接状态，选择其中 IP 地址分别是 192.168.0.34 和 192.168.0.36 的一个 HTTP 连接，然后单击“ restore content ”按钮，就会出现如图 7-16 所示的运行结果。

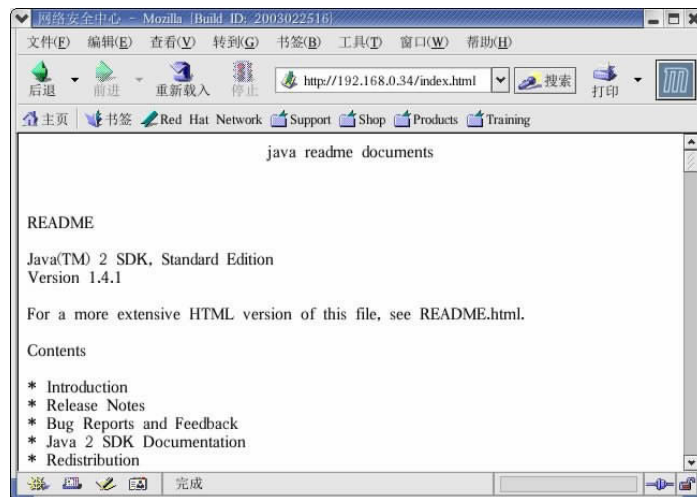


图 7-15 访问 Web 服务器

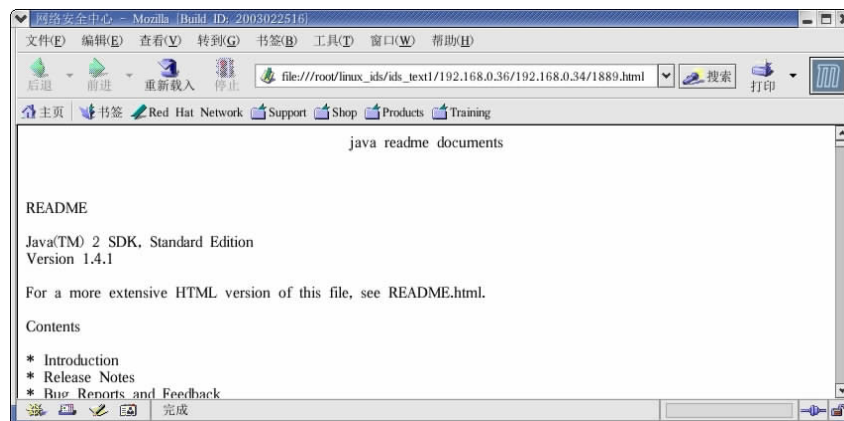


图 7-16 HTTP 内容回放

注意，在此画面中，浏览器的地址栏里的内容如下：

file:///root/linux_ids/ids_text1/192.168.0.36/192.168.0.34/1889.html

可以看到，此时浏览器打开的是本地 HTML 文件，而不是远程 Web 服务器上的文件。

这样，通过分析 HTTP 协议，提取 HTTP 协议内容，重现 HTML 文件，就可以查看到网络上所有主机浏览网站的活动了。同样的道理，可以重现 FTP, POP3, TELNET 等协议的内容。限于篇幅，就不在此一一介绍，读者可以自己来实现。

第 8 章 规则解析模块设计与实现

本章实现了一个入侵事件描述语言，并讨论了怎样用此入侵事件描述语言来建立规则，用规则来描述一个入侵事件。既然规则是用入侵描述语言建立的，那么对规则的解析就相当于对入侵事件描述语言的解析了。其实可以把入侵描述语言理解为 C 语言，而把规则理解为用 C 语言写的源代码，对规则的解析就相当于对 C 语言源代码进行编译。所以规则解析过程的实现是非常重要的。

入侵描述语言的建立是一个重要的工程。因为它的产生有很多好处。

第一，它可以动态建立规则，当有新的入侵攻击出现的时候，就可以使用入侵事件描述语言来建立新的规则，然后把它添加到旧的规则库中。这样规则库就可以动态更新。

第二，它可以使整个系统变得短小精悍。因为，如果不使用入侵描述语言，那么就需要用代码来实现各种不同的入侵攻击事件。入侵攻击方式是如此之多，而且还会不断出现新的攻击方式，如果都用代码来实现，势必会使整个系统变得非常庞大。

第三，不需要更改程序。当有新的攻击方式出现的时候（这是非常普遍的，因为现在黑客技术在不断提高），我们不需要编写新的代码来检测这种新的攻击方式，只需要用入侵描述语言来描述这个入侵方式就可以了，然后把定义好的规则添加到规则库中。这样，整个系统都不需要更改，只是更新规则库而已。

既然可以用入侵描述语言来建立新的规则，用新的规则来描述新的攻击方式，那么入侵描述语言就必须非常的优秀才行，不然的话，它不可能描述所有的攻击方式。因为攻击方式是千变万化的，并且新的攻击方式会变得越来越复杂，入侵描述语言能否描述所有的攻击方式，这是衡量一个入侵描述语言性能是否优良的重要指标。另外，上面也提到过，就是对入侵规则的解析也很重要。因为如果解析的过程非常的慢，就会影响整个系统的运行效率。

本章就是讨论怎样实现入侵描述语言，怎样用入侵描述语言来建立规则，怎样实现规则的解析过程。

8.1 建立入侵事件描述语言

如何表示入侵行为，这就是入侵描述语言的责任。IDS 事件描述语言就是建立一种 IDS 的事件描述方式，使 IDS 事件分析人员不必修改程序，在现场通过一定的设置工作，就可以定义任何新出现的 IDS 事件，并把事件库发送出去，使整个 IDS 系统在线升级，从而达到快速反应的效果。

每一个基于模式匹配的入侵检测方法都需要一个已定的入侵模式。这就需要一种对入侵行为的描述方法。现在的各种入侵检测系统中的描述方法各有不同，每个厂商定义自己的描述方法，每种方法各有长短。

目前，绝大多数的 IDS 系统中，都是采用程序判断的方式，就是每一个 IDS 事件，必须需要一个特定的子程序进行判断。其优点就是设计开发方便，每一个事件可以充分利用系统

的所有资源，事件之间没有关联，每一个事件可以独立开发，充分发挥每个事件开发人员的特长。其缺点也很多，就是每一个事件都对应一个子程序，开发量大，维护量大，升级困难，系统可靠性、事件的准确性都由一个个事件子程序决定，因此作为一个需要长时间维护补充数据的系统，就不合适。

要建立 IDS 事件描述语言是一项很困难和复杂的工作。要采用 IDS 事件描述方式，首先必须建立一个有效、可扩展的事件描述方法，能够描述所有 IDS 事件，这必须建立在对所有网络协议、IDS 事件彻底精通的基础上。

这些描述方法中最引人注目的是 NFR 使用的描述方法，它把每一种入侵行为描述为一种按特定语法书写的函数，在该函数进入系统时被编译成为二进制代码，计算机能够直接执行。

描述方法的不一致使得用户只能依赖于开发厂商来升级入侵检测模式库。如果有一种新的入侵方法出现，各个厂商对这种入侵方法的反应时间是不相同的，他们发布自己的描述方法总是有先有后的，因此总是存在一个可以被黑客利用的时间差。

在本系统中，设计了基于协议分析的入侵事件描述语言，用此语言来定义入侵规则。这样很好地解决了上述问题，能够动态定义入侵规则，当有新的入侵活动出现时，就能够利用入侵事件描述语言来写入侵规则，并把这个规则加入规则库中。这样就不用改变整个程序的内容，不需重新编译程序，只要更新系统中的规则库就可以了。

下面对建立入侵事件描述语言的过程进行详细介绍。

8.2 特征的选择

IDS 要有效地捕捉入侵行为，必须拥有一个强大的入侵特征数据库。而特征（signature）的选择至关重要。IDS 中的特征就是指用于判别通信信息种类的样板数据，通常有多种。例如：

- 来自保留 IP 地址的连接企图：可通过检查 IP 报头的来源地址轻易地识别。
- 带有非法 TCP 标志联合物的数据包：可通过对比 TCP 报头中的标志集与已知正确和错误标志联合物的不同点来识别。
- 含有特殊病毒信息的 E-mail：可通过对比每封 E-mail 的主题信息和病态 E-mail 的主题信息来识别。
- 查询负载中的 DNS 缓冲区溢出企图：可通过解析 DNS 域及检查每个域的长度来识别利用 DNS 域的缓冲区溢出企图。
- 通过提交上千次相同命令来实施对 POP3 服务器的拒绝服务攻击：可以设定命令提交的次数，一旦超过设定的次数系统将会发生报警。

可以看出，特征的选择范围非常广泛，有简单的报头域数值、有高度复杂的连接状态跟踪、有扩展的协议分析。这里是根据上面协议分析得到的一些协议量来定义特征的。因为这些特征值是根据 RFC 而来的，是有据可依的。发现任何违背 RFC 规定的异常特征，就可以断定有入侵企图。当然还要根据实际情况而定，因为并不是所有的操作系统和应用程序都是完全继承 RFC 的。再就是报头值的结构比较简单，而且可以很清楚地识别出异常报头信息。

下面举一个实例，看是怎样把协议量作为特征的。

Synscan 是一个流行的用于扫描和探测系统的工具，由于它的代码可以被用于创建蠕虫

Ramen 的开始片断。Synscan 的执行行为具有典型性，它发出的信息包具有多种可分辨的特性，包括：

- 不同的来源 IP 地址信息；
- TCP 的源端口 21，目标端口 21；
- 服务类型 0；
- IP 标识为 39426 (IP identification number)；
- 设置了 SYN 和 FIN 标志位；
- 不同的序列号集合 (sequence numbers set)；
- 不同的确认号码集合 (acknowledgment numbers set)；
- TCP 窗口尺寸为 1028。

对以上数据进行筛选，比较哪个更合适做特征数据。一般要寻找的是非法、异常或可疑数据，这都反映出攻击者利用的漏洞或者他们使用的特殊技术。

只具有 SYN 和 FIN 标志集的数据包，这是公认的恶意行为迹象。

没有设置 ACK 标志，但却具有不同确认号码数值的数据包，而正常情况应该是 0。

来源端口和目标端口都被设置为 21 的数据包，经常与 FTP 服务器关联。这种端口相同的情况一般被称为“反身”(reflexive)，除了个别时候如进行一些特别 NetBIOS 通信外，正常情况下不应该出现这种现象。“反身”端口本身并不违反 TCP 标准，但大多数情况下它们并非是预期数值。例如，在一个正常的 FTP 对话中，目标端口一般是 21，而来源端口通常都高于 1023。

TCP 窗口尺寸为 1028，IP 标识在所有数据包中为 39426。根据 IP RFC 的定义，这两类数值应在数据包间有所不同。因此，如果持续不变，就表明可疑。

从以上 4 个候选对象中，我们可以单独选出一项作为基于报头的特征数据，也可以选出多项组合作为特征数据。

选择一项数据作为特征有很大的局限性。选择以上 4 项数据联合作为特征也不现实，因为这显得有些太特殊了，而且也缺乏效率。特征定义要在效率和精确度之间取得折中。简单特征比复杂特征更倾向于误报 (false positives)，复杂特征比简单特征更倾向于漏报 (false negatives)。特征的选择应根据实际情况而定。还是以这个例子来看，要想判断攻击可能采用的工具是什么，那么除了 SYN 和 FIN 标志以外，还需要其他一些属性。“反身”端口虽然可疑，但是许多工具都使用到它，而且一些正常通信也有此现象，因此不适宜选为特征。TCP 窗口尺寸为 1028 和 IP 标识为 39426 都很正常。没有 ACK 标志的 ACK 数值很明显是非法的，因此非常适于选为特征数据。这样基于以上个很多因素的考虑可以创建一个特征，用于寻找并确定 synscan 发出的每个 TCP 信息包中的以下属性：

- 只设置了 SYN 和 FIN 标志；
- IP 标识为 39426；
- TCP 窗口尺寸为 1028。

从上面的例子中，可以看出，用于创建 IDS 特征报头值信息有很多种。通常，最有可能用于生成报头相关特征的元素为以下几种：

- IP 地址，特别保留地址、非路由地址、广播地址；
- 不应被使用的端口号，特别是众所周知的协议端口号和木马端口号；

- 异常信息包片段；
- 特殊 TCP 标志组合值；
- 不应该经常出现的 ICMP 字节或代码。

8.3 规则格式

基于协议分析的入侵事件语言，是以网络协议变量为基础，把网络协议中的各个数据域分解为一个个标准变量，即网络变量，并赋予所能够进行的各种运算操作。所谓数据模式，就是指各种变量的数据匹配形式。

当把网络协议中的所有数据域都定义为协议变量后，就可以根据 IDS 事件的产生方式，用一个运算公式定义一个 IDS 事件了，示例如下：

ip_ttl=1：这里定义了一个 IP 路由跟踪事件，即当 IP 协议中的最大跳转值等于 1 时，产生该事件，此事件往往被黑客用于确定一个 IP 地址的路由信息。

入侵事件描述语言就是用来描述规则的。其语法可以用文本格式表示如下：

事件定义 ::= 事件定义运算式 [关系符号 事件定义]

关系符号 ::= = & 或 |

事件定义运算式 ::= 协议变量 协议运算符 事件数据

协议变量 ::= 所有的协议变量

所有的协议变量 ::= ip_ttl 或者 ip_flags 或.....

协议运算符 ::= = = 或 ~ 或 > 或 < 或 ^

事件数据 ::= 十进制数字 或 字符串

其中，[] 中的内容可以有，也可以无。

事件定义：由事件定义运算式组成，或者由事件运算是通过关系符号组合在一起的组合事件运算式。

关系符号：只有两种关系符号，一个是 &，表示必须同时满足两个事件定义运算式。另一个是 |，表示只要满足一种事件定义运算式就可以。

事件定义运算式：由协议变量、协议运算符和事件数据三个部分组成。

协议变量：表示每个协议特征。有很多个协议变量，详细介绍在后面。

协议运算符：有五种协议运算符，分别如下：

= 表示等于；

~ 表示不等于；

> 表示大于；

< 表示小于；

^ 表示包含。

事件数据：表示协议特征的值，一般为 IDS 事件的特征数。可以是十进制形式或者字符串形式。

上面只是事件定义的语法规则，下面再来看一个完整的入侵事件是怎样定义的。

一个完整的事件描述格式包括事件名称、事件协议、事件代码、事件定义、事件说明和

响应方式，如下所示：

事件名称	事件协议	事件代码	事件定义	事件说明	响应方式
------	------	------	------	------	------

其中各个部分描述如下。

事件名称：是指事件的简写名字。可以用一个比较有代表性的名字来说明入侵的类型等。如 DOS，表示拒绝服务攻击，UNICODE，表示 UNICODE 漏洞攻击事件。

事件协议：是指该事件所使用的协议。如 TCP, UDP, ICMP, IP 等。

事件代码：是指该事件指定的代码，每个事件的代码不同。这样就可以区分各个不同的事件了。

事件定义：就是上面定义的事件定义，这是规则格式中最重要的一部分。

事件说明：是对此事件的详细说明。

响应方式：是指对此事件所采用的何种响应方式。

在各个部分中间以 TAB 键隔开，一行代表一个事件，存放在一个文件中。程序在运行时要读取这个文件中的内容，进行解析，供入侵检测系统所用。

可以举一个例子，下面就是一个事件：

```
ICMP_PING Event
ICMP
100000
icmp_type=8
This is a ping event.
sound
```

此规则可以描述一个 ICMP PING 事件，特征是 ICMP 协议并且其类型为 8。事件名称为“ICMP_PING Event”，事件协议是“ICMP”，事件代码是“100000”，事件定义是“icmp_type=8”，事件说明是“This is a ping event.”，事件响应方式是“sound”。

又如下面一个事件：

```
UNICODE Event
TCP
100001
tcp_dport = 80&tcp_content^%c0%af
This is unicode event.
sound
```

此规则可以描述一个 unicode 事件，特征是 TCP 协议，目的端口号为 80，并且其负载中包含%c0%af 字符串。事件名称为“UNICODE Event”，事件协议是“TCP”，事件代码是“100001”，事件定义是“tcp_dport = 80&tcp_content^%c0%af”，此时的事件定义就有两个部分，分别是“tcp_dport = 80”和“tcp_content^%c0%af”，它们的关系是“&”，说明只有这两个事件定义都是真的时候整个事件的定义才是真。事件说明是“This is unicode event.”，事件响应方式是“sound”。

图 8-1 是规则操作的运行界面。可以任意增加或者删除一个或多个规则，在这个过程中，不需要终止整个系统的运行状态。这极大地增加了系统的灵活性和扩展性。

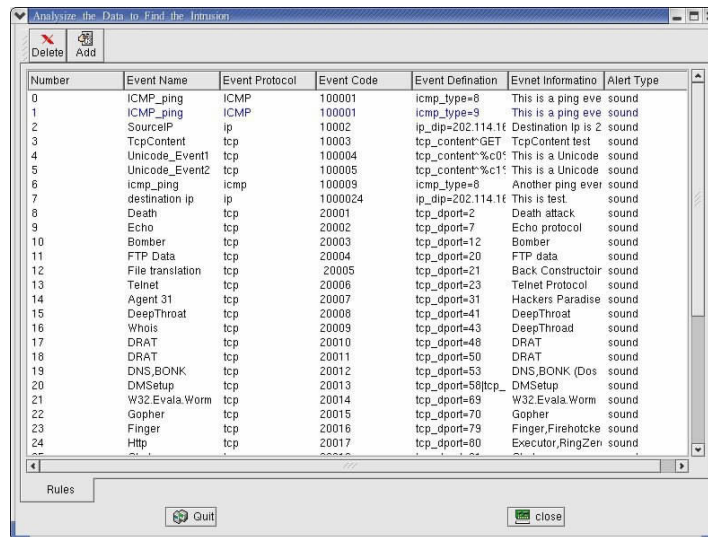


图 8-1 规则操作界面

我们可以在这个界面对规则进行各种操作，如添加、删除、修改、保存等。这些都比较容易实现。需要注意的是，这些规则实际存储在一个文件之中，称之为 rules。在这个文件中存放了很多的规则，每一项规则就表示了一种入侵行为，就如上面描述的两个例子一样，例如，UNICODE 漏洞攻击，就可以用上述规则进行定义，然后把它添加到规则库中，当然可以手工修改 rules 文件，把规则添加到规则库之中。但是，这样非常容易出错，而且出错之后就不容易查找。所以最好在图 8-1 所示的界面中进行添加，当添加之后，规则就会按规定的格式存储到文件 rules 之中了。所以这样既安全又迅速。

也可以在规则中添加一些测试规则，用于分析我们感兴趣的网络数据。例如，如果想分析所有的 HTTP 数据包，就可以定义如下规则。

```
HTTP Event
HTTP
20017
tcp_dport = 80
This is HTTP event.
sound
```

这样就可以检测到所有的 HTTP 信息了。这对于分析特定的数据包是非常有好处的。可以检测到网络中有什么信息在流动。只要对什么感兴趣，都可以定义相应的规则来进行检测。可以看出这个规则的定义是非常灵活，非常方便的。而且规则之间互不干扰，可以相互独立。如果你想对几种类型的数据感兴趣，你可以添加多个规则，一个规则就对应一种类型的数据，当有此数据到达时，就响应，响应的方式由“响应方式”字段进行定义，例如 sound，就表示声音报警。还有很多其他响应方式，请参考第 10 章响应模块。

入侵事件描述语言的建立对于入侵规则来说是一个非常重要的部分，通过使用入侵描述语言，就可以方便地定义入侵攻击特征，添加规则到规则库中。如果出现了新的黑客攻击方式，就可以用此入侵事件描述语言来建立一个新的规则到规则库中，这样就可以更新规则库了。很多商业的入侵检测系统，都可以通过因特网来动态更新规则库，其原理就是如此。

增加一个规则的窗口如图 8-2 所示。包括事件名称、事件协议、事件代码、事件定义、事件信息、响应类型。可以动态添加任意的事件规则到规则库中。

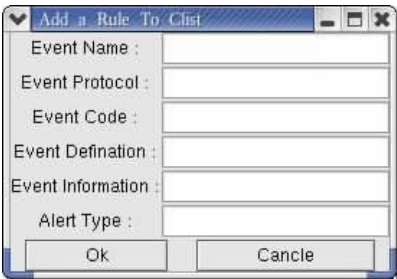


图 8-2 添加规则

8.4 规则选项

如上所说，基于协议分析的描述语言，是以网络协议变量为基础的，下面详细讨论网络协议变量的指定以及相关规则选项的意义。

8.4.1 IP 协议变量

IP 协议变量的数据结构如下所示：

```
struct ip_variable
{
    int      ip_hlength;
    int      ip_length;
    char     ip_sip[1024];
    char     ip_dip[1024];
    char     ip_type[1024];
    int      ip_version;
    int      ip_service;
    int      ip_ident;
    int      ip_flags;
    int      ip_distance;
    int      ip_ttl;
}
```

其中各个变量的定义和相应的运算操作种类如表 8-1 所示。

表 8-1 IP 协议变量定义规则

变量名称	变量含义	运算操作	操作含义
ip_hlength	IP 报文头长度	=	等于运算
		~	不等于运算
		>	大于运算
		<	小于运算

续表

变量名称	变量含义	运算操作	操作含义
ip_length	IP 报文长度	=	等于运算
		~	不等于运算
		>	大于运算
		<	小于运算
ip_sip	源 IP 地址	=	等于运算
		~	不等于运算
ip_dip	目的 IP 地址	=	等于运算
		~	不等于运算
ip_type	IP 协议类型	=	等于运算
		~	不等于运算
ip_version	IP 报文版本	=	等于运算
		~	不等于运算
		>	大于运算
		<	小于运算
ip_service	IP 报文服务类型	=	等于运算
		~	不等于运算
ip_ident	IP 报文标志	=	等于运算
		~	不等于运算
ip_flags	IP 报文偏移标志	=	等于运算
		~	不等于运算
ip_distance	IP 报文碎片偏移量	=	等于运算
		>	大于运算
		<	小于运算
ip_ttl	IP 报文最大跳转值	=	等于运算
		>	大于运算
		<	小于运算

8.4.2 TCP 协议变量

TCP 协议变量的数据结构如下所示：

```

struct tcp_variable
{
    int      tcp_sport;
    int      tcp_dport;
    unsigned long int    tcp_sequence;
    unsigned long int    tcp_ack;
    int      tcp_hlength;
    char      tcp_flags[1024];
    int      tcp_window;

```

```
int      tcp_checksum;
int      tcp_urge;
char     tcp_content[65535];
}
```

其中各个变量的定义和相应的运算操作种类如表 8-2 所示。

表 8-2 TCP 协议变量定义规则

变量名称	变量含义	运算操作	操作含义
tcp_sport	源端口	=	等于运算
		~	不等于运算
tcp_dport	目的端口	=	等于运算
		~	不等于运算
tcp_sequence	序号	=	等于运算
		~	不等于运算
tcp_ack	确认序号	=	等于运算
		~	不等于运算
tcp_hlength	首部长度	=	等于运算
		~	不等于运算
		>	大于运算
		<	小于运算
tcp_flags	标志	=	等于运算
		~	不等于运算
tcp_window	窗口	=	等于运算
		~	不等于运算
		>	大于运算
		<	小于运算
tcp_checksum	校验和	=	等于运算
		~	不等于运算
tcp_urge	紧急指针	=	等于运算
		~	不等于运算
tcp_content	数据内容	^	包含

8.4.3 UDP 协议变量

UDP 协议变量的数据结构如下所示：

```
struct udp_variable
{
int      udp_sport;
int      udp_dport;
int      udp_length;
int      udp_checksum;
char     udp_content[65535];
}
```

其中各个变量的定义和相应的运算操作种类如表 8-3 所示。

表 8-3 UDP 协议变量定义规则

变量名称	变量含义	运算操作	操作含义
udp_sport	源端口	=	等于运算
		~	不等于运算
udp_dport	目的端口	=	等于运算
		~	不等于运算
udp_length	长度	=	等于运算
		~	不等于运算
		>	大于运算
		<	小于运算
udp_checksum	校验和	=	等于运算
		~	不等于运算
udp_content	数据内容	^	包含

8.4.4 ICMP 协议变量

ICMP 协议变量的数据结构如下所示：

```
struct icmp_variable
{
    int      icmp_type;
    int      icmp_code;
    int      icmp_checksum;
    char     icmp_content[65535];
}
```

其中各个变量的定义和相应的运算操作种类如表 8-4 所示。

表 8-4 ICMP 协议变量定义规则

变量名称	变量含义	运算操作	操作含义
icmp_type	类型	=	等于运算
		~	不等于运算
icmp_code	代码	=	等于运算
		~	不等于运算
icmp_checksum	校验和	=	等于运算
		~	不等于运算
icmp_content	数据内容	^	包含

8.4.5 响应方式

alert_type

此是代表报警方式的变量，共有如下几种：

sound : 采用声音报警。
 flash : 采用闪烁的画面报警。
 email : 采用发送 E-mail 的方式报警。
 log : 只是记录下来。

8.5 规则解析模块实现

规则解析模块的主要功能就是对规则进行解析。规则解析就是把规则库中的规则解析成入侵检测模块能够识别的形式。由于规则库中存放很多条规则,这些规则是用事件描述语言写的,那么现在的任务就是把每条规则都分解出来,存入到各个变量中,以便进行入侵检测分析。

规则解析中用来表示事件描述格式中各段的数据结构如下:

```
char event_name[1024][1024];
```

表示事件名称,例如“UNICODE Event”。

```
char event_protocol[1024][1024];
```

表示事件协议,例如“TCP”。

```
char event_code[1024][1024];
```

表示事件代码,例如“100001”。

```
char event_defination[1024][1024];
```

表示事件定义,例如“tcp_dport=80&tcp_content^%c0%af”。

```
char event_information[1024][1024];
```

表示事件说明,例如“This is a unicode event.”。

```
char alert_type[1024][1024];
```

表示响应方式,例如“sound”。

首先在解析规则之前必须把上面所有清零,其函数实现如下:

```
void
clear_all_rules(void)
{
    //清除所有规则变量的内容
    int i;
    for (i = 0; i < 1024; i++)
    {
        strcpy (total_rules[i], "");
        strcpy (event_name[i], "");
        strcpy (event_protocol[i], "");
        strcpy (event_code[i], "");
        strcpy (event_defination[i], "");
        strcpy (event_information[i], "");
        strcpy (alert_type[i], "");
    }
}
```

下面是把事件定义解析的核心代码，主要是把事件定义中的事件定义运算式和关系符号都解析出来。

```
void read_event_definition(char *event)
/* 解析事件定义，如把“icmp_type=8&ip_sip=202.114.165.24”解析成：事件定义运算式
“icmp_type=8”和“ip_sip=202.114.165.24”，分别存储到 event_statement[]中。把&符号存储到
relation 中*/
{
    int total_length;
    int length;
    int i;
    int j;
    int relation_number=0;
    total_length=strlen(event);
    for(i=0;i<total_length;i++) /*把符号先解析出来*/
    {
        if(*(event+i)=='&') /*判断关系符号是否是&*/
        {
            relation[relation_number]='&';
            /*把关系符号&存入 relation 中*/
            relation_number++;
        }
        if(*(event+i)=='|') /*判断关系符号是否是|*/
        {
            relation[relation_number]='|';
            /*把关系符号&存入 relation 中*/
            relation_number++;
        }
    }
    state_number=0;
    state_number=relation_number+1;
    for(i=0;i<state_number;i++) /*先清空*/
    {
        strcpy(event_statement[i],"");
    }
    for(j=0;j<state_number;j++) /*解析出事件定义运算式*/
    {
        length=strlen(event);
        for(i=0;i<length;i++)
        {
            if((*(event+i)=='&')||(*(event+i)=='|'))
            /*如果是&或|就跳过读取下一字节*/
```



```
        {  
            event=event+i+1;  
            break;  
        }  
        event_statement[j][i]=*(event+i);  
    }  
    event_statement[j][i]='\0';  
}
```

规则解析的过程和规则匹配过程是一起执行的，它们之间的联系非常密切。在这里只是简单介绍了其实现机理，更详细的内容可以参考第 9 章的入侵检测模块设计，在那里更详细地实现了规则解析和规则匹配的全过程。

8.6 小结

本章我们设计了一个入侵事件描述语言，然后用此语言来定义入侵规则。事件描述语言是入侵检测系统中一个非常重要的内容，在一般的入侵检测系统中都有自己相应的事件描述语言，如 Snort 系统。在这里我们只是为了理解怎样建立入侵事件描述语言，事件描述语言的使用方法，怎样用事件描述语言定义规则，怎样解析规则。所以此描述语言是一个比较简单的实现，不是很复杂，有利于读者更好地掌握入侵事件描述语言技术。

使用入侵事件描述语言的好处是显而易见的。由于黑客技术在不断提高，他们攻击的方式也在不断变化，当出现新的攻击方式的时候，就可以通过入侵事件描述语言来定义新的规则，把此规则添加到现有的规则库中，以更新规则库。这样，新的攻击方式也可以检测出来了。这样系统不需要重新更改，就可以检测到新的攻击方式了。可以看到，这是非常方便和灵活的。

对入侵描述语言的解析，就是编译入侵描述语言定义的规则。就像编译 C 语言程序一样。在这里入侵描述语言就相当于 C 语言，而用入侵描述语言定义的规则就相当于用 C 语言写的一段源程序，而规则的解析就相当于编译 C 语言源程序了。大家都知道，C 语言编译器的好坏就决定了编译后 C 语言程序的性能。一样的道理，规则解析的好坏就决定了入侵检测模块的运行性能了。

规则解析模块与入侵事件检测模块联系得非常紧密，在这里只是简单介绍了规则解析的实现过程，在入侵事件检测模块中还要结合规则匹配进行详细的讨论规则解析的实现过程。

第9章 入侵事件检测模块设计与实现

入侵事件检测模块是入侵检测系统的核心模块，其主要功能就是检测是否有入侵行为发生。怎样检测有入侵行为发生，这就是入侵检测方法。现在出现了很多种入侵检测方法，其中用得最多的就是模式匹配方法，这是传统的入侵检测方法。模式匹配方法有很多缺点，为此我们提出了基于协议分析的入侵检测方法，它是一个新的入侵检测方法，其性能要比传统的模式匹配方法优良。

基于协议分析的入侵检测方法，其原理很简单，就是在协议分析的基础上进行入侵事件检测。在前面几章中，我们实现了网络协议分析模块，而此模块的作用就是对网络数据包进行详细的协议分析。其分析的结果，即协议信息，是入侵检测模块的分析对象。所以现在就能够轻松进行入侵检测的实现了。另外，我们还实现了规则解析模块，在此模块中，我们实现了一个入侵事件描述语言，而此描述语言也是基于协议而建立起来的。用此入侵事件描述语言建立的规则，也是关于协议的规则。那么在入侵事件检测模块中，我们就可以使用规则来匹配协议信息的方式来进行入侵检测。这就是入侵事件检测模块的实现机理。

9.1 入侵检测方法

9.1.1 模式匹配方法的不足

模式匹配就是将收集到的信息与已知的网络入侵和系统误用模式数据库进行比较，从而发现违背安全策略的行为。模式匹配是第一代和第二代入侵检测系统所使用的网络数据包分析技术。他的优点就是分析速度快、误报率小。但是单纯使用模式匹配的方法有很多的不足之处。单纯的模式匹配方法就是判断分析网络上的每一个数据包是否具有某种攻击特征，即字符串，其工作步骤如下：

- 从网络数据包的包头开始与攻击特征字符串比较；
- 若比较结果相同，那么就检测到一个可能的攻击；
- 若比较结果不同，则下移一个字节再进行比较；
- 直到把攻击特征中所有字节都匹配完；
- 对于每一个攻击特征，重复第二个步骤；
- 直到每一个攻击特征都匹配完。

下面给出一个例子说明其工作原理。

```
0020 DAD3 C580 5254 AB27 C004 0800 4500 0034 07BA 4000
4006 A0F6 CA72 A518 CA72 5816 040B 0050 23E5 241E 50BB
07B0 8010 9310 BD0A 0000 0101 080A 0002 259E 01A3 C881
```

以上为捕获到的数据包，对于攻击模式“GET /cgi-bin/.phf”，根据以上次序，首先从数据包头部开始比较：

```
GET /cgi-bin/./phf--
0020 DAD3 C580 5254 AB27 C004 0800 4500 0034 07BA 4000
4006 A0F6 CA72 A518 CA72 5816 040B 0050 23E5 241E 50BB
07B0 8010 9310 BD0A 0000 0101 080A 0002 259E 01A3 C881
```

比较不成功，移动一个字节重新比较。

```
- GET /cgi-bin/./phf-
0020 DAD3 C580 5254 AB27 C004 0800 4500 0034 07BA 4000
4006 A0F6 CA72 A518 CA72 5816 040B 0050 23E5 241E 50BB
07B0 8010 9310 BD0A 0000 0101 080A 0002 259E 01A3 C881
```

还是不成功，再移动一次。

```
--GET /cgi-bin/./phf-
0020 DAD3 C580 5254 AB27 C004 0800 4500 0034 07BA 4000
4006 A0F6 CA72 A518 CA72 5816 040B 0050 23E5 241E 50BB
07B0 8010 9310 BD0A 0000 0101 080A 0002 259E 01A3 C881
```

重复比较，没有一次匹配成功。

这种单纯的模式匹配方法存在以下几个问题。

(1) 计算量大

对于一个特定网络的每秒需要比较的最大次数估算为：

攻击特征字节数 × 网络数据包字节数 × 每秒数据包数量 × 攻击特征数量

如果攻击特征平均长度为 10 字节，网络数据包平均长度为 30 字节，每秒 30 000 数据包，特征库中有 2000 条特征，那么，每秒比较的次数大约为：

$$10 \times 30 \times 30,000 \times 2,000 = 18,000,000,000$$

(2) 检测准确性低

简单的模式匹配只能检测特定类型的攻击。对攻击特征微小的变形都将使得检测失败。

例如，对于 Web 服务器

```
GET /cgi-bin/phf
HEAD /cgi-bin/phf
GET //cgi-bin/phf
GET /cgi-bin/foobar/./phf
GET %00/cgi-bin/phf
GET /%63%67%69%2d%62%69%6e/phf
```

这些攻击特征用以上的匹配方法都不能检测出来。

传统的模式匹配的问题根本是它把网络数据包看做是无序的随意的字节流。它对该网络数据包的内部结构完全不了解。它对于网络中传输的图像或音频流同样进行匹配。可是网络通信协议是一个高度格式化的、具有明确含义和取值的数据流，如果将协议分析和模式匹配方法结合起来，可以获得更高效率、更精确的结果。

而协议分析技术则不同于简单的模式匹配技术。协议分析有效利用了网络协议的层次性和相关协议的知识快速地判断攻击特征是否存在。它的高效使得匹配的计算量大幅度减小。即使在 100Mb/s 的网络中，也可以充分地检测每一个数据包。

下面还是以上述数据包为例，讨论一下基于协议分析的入侵检测系统是如何处理的：

```
0020 DAD3 C580 5254 AB27 C004 0800 4500 0034 07BA 4000
4006 A0F6 CA72 A518 CA72 5816 040B 0050 23E5 241E 50BB
07B0 8010 9310 BD0A 0000 0101 080A 0002 259E 01A3 C881
```

协议规范指出以太网数据包中第 13 字节处包含了两个字节的网络层协议标识，如图 9-1 所示。基于协议分析的入侵检测系统利用这个知识开始第一步检测：跳过数据包前面 12 个字节，读取第 13 字节处的 2 字节协议标识为 0x0800。根据协议规范可以判断这个网络数据包是 IP 包。

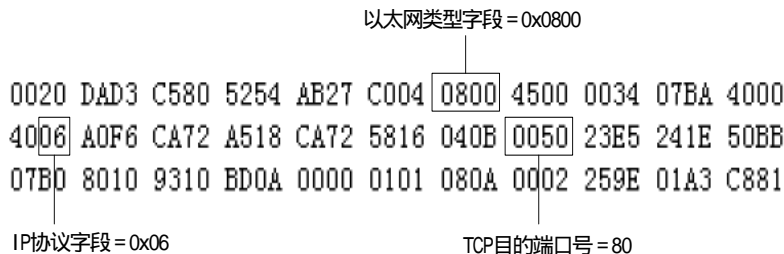


图 9-1 数据包分析

根据 IP 协议规定，可以知道在数据包第 24 字节处有一个 1 字节的传输层协议标识。因此系统跳过数据包第 15 到第 24 字节直接读取传输层协议标识为 0x06，这个数据包是 TCP 协议。

再根据 TCP 协议规定，可以知道在数据包第 37 字节处有一个 2 字节的应用层协议标识，即端口号。于是系统跳过第 25 到 36 字节直接读取第 37 字节的端口号为 0x0050，即 80。该数据包是一个 HTTP 协议的数据包。

再根据 HTTP 协议规定，可以知道在数据包第 55 字节是 URL 开始处，我们要检测入侵特征“GET /cgi-bin/./phf”，只需要仔细检测这个 URL 就可以了。

可以看出，利用协议分析可以大大减小模式匹配的计算量，提高匹配的精确度，减少误报率。下面详细介绍协议分析技术。

9.1.2 使用协议分析方法

基于协议分析的规则与其他规则相比，提供了一种高级的网络入侵解决方案，因为它们可以检测更广泛的攻击，包括已知和未知的。协议分析可以有效地针对欺骗性攻击。

使用基于协议分析方法制订的网络入侵检测规则对于与类似 DNS，HTTP，FTP，SMTP 等协议相关的已知和未知的攻击非常有效。协议分析的过程意味着 IDS 传感器知道各种不同协议的工作方法，近似地分析这些协议的通信来得出是否有可疑或不正常的行为存在。对于每个协议来说，分析不仅要依赖协议标准如 RFCs，还要依赖于实际中真正发生的事件。很多行为是违背标准的，所以规则反应出事实的本来面目非常重要，而不是理想状态下它们做了什么或者有多少积极和被动的错误会发生。协议分析技术是观察在特定协议中的通信，然后再进行验证，如果通信状态不如预期则发出警报。

单纯进行数据包查找的规则，它们必须是针对某些特定的已知攻击。并且由于有很多的

攻击程序和脚本，也有很多这些程序的变体，那么就无法制订一个规则来囊括所有的攻击程序。还有就是时间问题，因为规则是在知道攻击发生后才制订出来的，存在滞后性。在攻击公布前，规则是无法识别它的。在很多事件中，攻击最初的实施比 IDS 能够识别它的行为要早很多。而协议分析可以查找任何违背标准或预期行为的行动，它增强了网络 IDS 传感器侦测已知和未知攻击方法的能力。

制订一种基于已知和潜在脆弱性的规则，替代着重于匹配攻击程序的规则。基于协议分析所设置的规则非常了解协议的预期行为。基于协议分析制订的规则还可以附加确认信息和各种命令，来寻找任何不正常或可疑行为的迹象。这些确认信息应该符合已知和未知的脆弱性，如缓冲区溢出和非法属性。虽然无法预知未知的脆弱性是否存在，但可以通过检查各种不正常的领域来预期。

对于各种不同分类的攻击都可以使用协议分析方法识别，可以通过确认协议数据来捕获攻击。在最初的分类中，可以制订一种可以识别可能的攻击程序的规则，然后，进行进一步分类，不需要决定攻击的目的，而只需要确定是否有不正常的行为发生以及进行更进一步的研究。只要着重于对已知协议进行仔细分析，而不需要研究攻击程序的特征。

攻击者可以迷惑使用包查找功能建立的 IDS，也就是上面的模式匹配方法，而对于协议分析方法却无法欺骗。例如，假设需要对含有 scripts/iisadmin 的 URL 发出警报。攻击者可以使用反斜杠代替斜杠。大多数 Web 服务器对于使用反斜杠还是斜杠来划分目录不敏感，所以 WEB 服务器认为含有 scripts/iisadmin 和含有 scripts\iisadmin 的 URL 是一回事。不幸的是，IDS 规则只是进行简单的字符匹配工作，所以对出现 scripts\iisadmin 不加理睬。但协议分析方法就可以识别这种变体入侵。当一个 HTTP 通信开始被监控时，IDS 传感器从 URL 中读出路径，然后对其进行分析。它适当地分析反斜杠，使 URL 标准化，然后找出可疑的目录内容，如 scripts/iisadmin。例如在 unicode 攻击中，与 ASCII 字符相关的 HEX 编码一直到 %7f，Unicode 编码的值要高于它。以包含 scripts/..%c0%af../winnt 的 URL 路径为例。%c0%af 在 Unicode 中相当于斜杠/，这样就是 scripts/../../winnt。这也是攻击特征。使用包含 Unicode 编码解析的协议分析，就可以识别这种欺骗。

举例说明协议分析如何识别包含 /hidden/admin/ 的 URL，从而识别欺骗的，其流程如图 9-2 所示。

对 IP 包头进行解析，确定 IP 有效荷载所包含的协议。例如 IP 协议号为 6 的 TCP 协议。

对 TCP 头进行解析，确定 TCP 目的端口。例如 Web 服务器端口 80。

对 HTTP 协议进行解析，识别 URL 路径。

对 URL 进行解析，识别路径欺骗，HEX 编码，二次 HEX 编码和 Unicode 编码。

最后判断路径中是否包含 “/hidden/admin/”，若有则报警。

9.1.3 协议分析技术的优点

传统的特征模式匹配技术的特点就是检测慢、不准确、消耗系统资源，它的最大缺陷及时计算负载大和准确性低，而协议分析技术的特点就是检测快、准确，极少的资源消耗。

协议分析是新一代 IDS 系统探测攻击手法的主要技术，它利用网络协议的高度规则性快速探测攻击的存在。协议分析技术的优势在于：

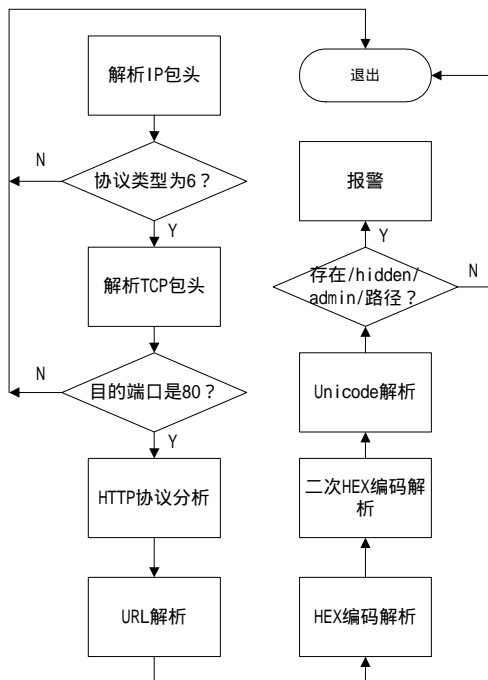


图 9-2 识别 URL 的流程

解析命令字符串。URL 第一个字节的位置给予解析器。解析器是一个命令解析程序，最新一代 IDS 网络入侵检测引擎包含超过 70 个不同的命令解析器，可以对在不同的上层应用协议，对每一个用户命令作出详细分析。

探测碎片攻击和协议确认。在基于协议分析的 IDS 中，各种协议都被解析，如果出现 IP 碎片设置，则数据包将首先被重装，然后详细分析来了解潜在的攻击行为。由于协议被完整解析，这还可以用来确认协议的完整性。

降低误报率。协议解析也大大降低了模式匹配 IDS 系统中常见的误报现象。使用命令解析器可以确保一个特征串的实际意义被真正理解，辨认出串是不是攻击或可疑的。

真正高性能。新一代基于协议分析的 IDS 系统网络传感器采用新设计的高性能数据包驱动器，使其不仅支持线速百兆流量检测，而且千兆网络传感器具有高达 900Mbit/s 网络流量的 100% 检测能力，不会忽略任何一个数据包。

新一代基于协议分析的入侵检测系统解决了 IDS 领域长期以来的应用瓶颈问题：检测准确性以及大流量应用网络环境下的系统性能。新一代 IDS 提供商将凭借此项最新技术，融合传统特征模式匹配技术的优点，为用户提供更加完善、优秀的入侵检测与防护系统。

9.2 入侵事件检测模块实现

入侵行为的检测技术是入侵检测系统的核心。根据入侵特征对数据包进行匹配，检测是否有入侵行为发生，这就是入侵事件检测模块要实现的功能。

本系统采用在以上协议分析的基础上，运用入侵事件描述语言，对规则库进行匹配来检测是否有入侵行为的存在。所有的攻击方法被表示为入侵规则存放在入侵特征数据库中，当

前的数据如果和规则库中某种特征匹配，就指出是某种入侵行为发生。

其流程如图 9-3 所示。

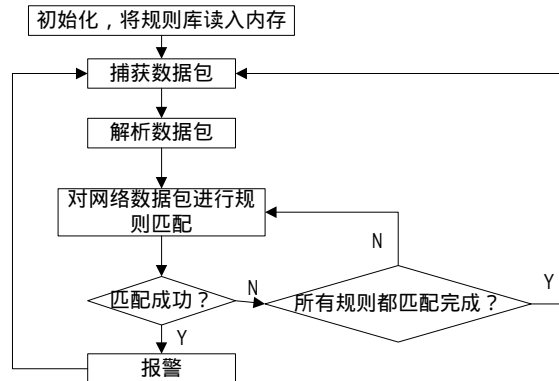


图 9-3 规则库的匹配流程

系统初始化时，将制定规则文件中的规则读入内存中。

捕获数据包。

解析数据包，把每个数据包的协议变量全部解析出来。

对内存中的一条规则进行匹配。

若规则匹配成功，则调用相应的响应程序。接着再运行步骤 。不成功则继续下一步骤。

匹配下一条规则，直到所有的规则都匹配完成为止。

图 9-4 是入侵事件检测模块检测到入侵行为的一个运行界面。

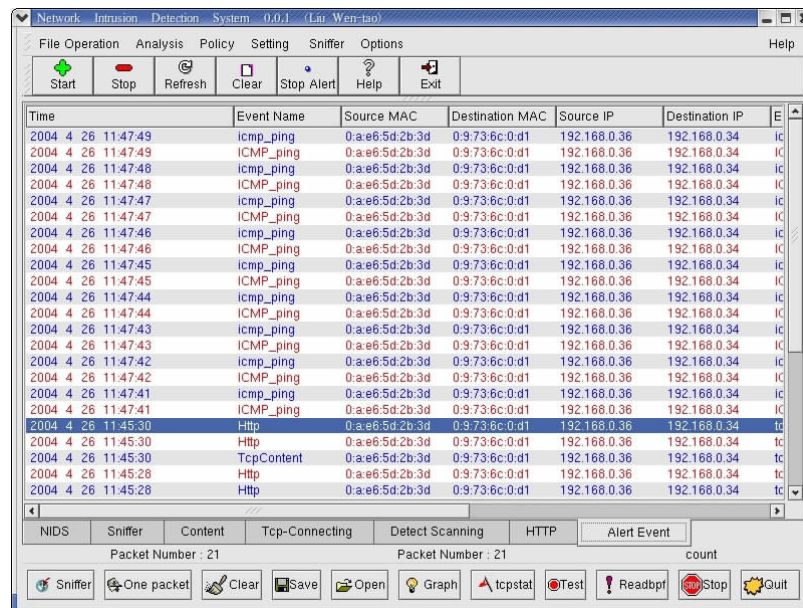


图 9-4 规则库操作界面

9.2.1 获取协议信息

在前面的网络协议分析模块中，我们分析了很多协议，把它们的信息存储起来，在这里我们要获取其信息，然后与规则进行匹配。这些信息都是存储在一些全局变量中。在前面的协议分析模块中定义了下面几个全局变量。

```
struct my_ip_header_string      ip_header_string_object;
struct      my_tcp_header_string      tcp_header_string_object;
struct      my_udp_header_string      udp_header_string_object;
struct my_icmp_header_string      icmp_header_string_object;
```

这些变量里面存放了分析协议之后的所有信息，现在就要从这些变量中读取信息，然后赋值给入侵事件检测模块中的相关变量，并在入侵事件检测模块中再调用相关变量。这样协议分析模块和入侵事件检测模块之间就不会有太多的牵连。这是模块化设计的具体体现。这样的思想在整个系统中都有体现。

在入侵事件检测模块中使用了下面一些变量。

```
struct      ip_variable      nids_ip_variable;
struct tcp_variable      nids_tcp_variable;
struct      udp_variable      nids_udp_variable;
struct      icmp_variable      nids_icmp_variable;
```

这些变量的结构体描述请参考规则匹配模块部分。它们就是存放了入侵事件检测模块所要使用的信息。现在的任务就是把在协议分析模块中使用到的变量中的内容读入入侵事件检测模块中所使用到的变量，而且在这个过程中要进行一定的类型转换。例如把字符串类型的值转化为数值型的。

读取网络信息的功能由下面几个函数来实现：

```
ip_header_string_object ( );
tcp_header_string_object.port ( );
udp_header_string_object ( );
icmp_header_string_object ( );
```

在这个系统中，只考虑了四种协议类型的数据，分别是 IP, TCP, UDP, ICMP。读者可以在此基础上加上其他协议类型，这样系统就会更强大了。首先对这些值清零。函数实现如下：

```
void
clear_all_variable (void) //对全局变量清零
{
    strcpy (ip_header_string_object.header_length, "");
    strcpy (ip_header_string_object.off, "");
    strcpy (ip_header_string_object.checksum, "");
    strcpy (ip_header_string_object.id, "");
    strcpy (ip_header_string_object.version, "");
    strcpy (ip_header_string_object.protocol, "");
    strcpy (ip_header_string_object.destination_ip, "");
```



```

strcpy (ip_header_string_object.source_ip, "");
strcpy (ip_header_string_object.total_length, "");
strcpy (ip_header_string_object.tos, "");
strcpy (ip_header_string_object.ttl, "");
strcpy (tcp_header_string_object.sport, "");
strcpy (tcp_header_string_object.dport, "");
strcpy (tcp_header_string_object.seq, "");
strcpy (tcp_header_string_object.ack, "");
strcpy (tcp_header_string_object.doff, "");
strcpy (tcp_header_string_object.flags, "");
strcpy (tcp_header_string_object.win, "");
strcpy (tcp_header_string_object.cksm, "");
strcpy (tcp_header_string_object.urp, "");
strcpy (tcp_header_string_object.options, "");
strcpy (tcp_header_string_object.information, "");
strcpy (tcp_content_object, "");
strcpy (udp_header_string_object.sport, "");
strcpy (udp_header_string_object.dport, "");
strcpy (udp_header_string_object.len, "");
strcpy (udp_header_string_object.cksum, "");
strcpy (udp_content_object, "");
strcpy (icmp_header_string_object.type, "");
strcpy (icmp_header_string_object.code, "");
strcpy (icmp_header_string_object.cksum, "");
strcpy (icmp_header_string_object.information, "");
strcpy (icmp_content_object, "");
}

```

怎样从这些变量中获取信息，是由下面这些函数完成的。

(1) 获取 IP 信息

```

void
get_ip_variable (void)
{
    int a_number;
    a_number = atoi (ip_header_string_object.header_length);
    nids_ip_variable.ip_hlength = a_number;
    printf ("nids_ip_variable.ip_hlength=%d\n", nids_ip_variable.ip_hlength);
    a_number = atoi (ip_header_string_object.tos);
    nids_ip_variable.ip_service = a_number;
    printf ("nids_ip_variable.ip_service=%d\n", nids_ip_variable.ip_service);
    a_number = atoi (ip_header_string_object.total_length);
    nids_ip_variable.ip_length = a_number;
}

```

```

printf ("nids_ip_variable.ip_length=%d\n", nids_ip_variable.ip_length);
sprintf (nids_ip_variable.ip_sip, "%s", ip_header_string_object.source_ip);
printf ("nids_ip_variable.ip_sip=%s\n", nids_ip_variable.ip_sip);
sprintf (nids_ip_variable.ip_dip, "%s",
        ip_header_string_object.destination_ip);
printf ("nids_ip_variable.ip_dip=%s\n", nids_ip_variable.ip_dip);
sprintf (nids_ip_variable.ip_type, "%s", ip_header_string_object.protocol);
printf ("nids_ip_variable.ip_type=%s\n", nids_ip_variable.ip_type);
a_number = atoi (ip_header_string_object.version);
nids_ip_variable.ip_version = a_number;
printf ("nids_ip_variable.ip_version=%d\n", nids_ip_variable.ip_version);
a_number = atoi (ip_header_string_object.id);
nids_ip_variable.ip_ident = a_number;
printf ("nids_ip_variable.ip_ident=%d\n", nids_ip_variable.ip_ident);
a_number = atoi (ip_header_string_object.off);
nids_ip_variable.ip_distance = a_number;
printf ("nids_ip_variable.ip_distance=%d\n", nids_ip_variable.ip_distance);
a_number = atoi (ip_header_string_object.ttl);
nids_ip_variable.ip_ttl = a_number;
printf ("nids_ip_variable.ip_ttl=%d\n", nids_ip_variable.ip_ttl);
}

```

(2) 获取 TCP 信息

```

void
get_tcp_variable (void)
{
    int a_number;
    long int a_long_number;
    a_number = atoi (tcp_header_string_object.sport);
    nids_tcp_variable.tcp_sport = a_number;
    printf ("nids_tcp_variable.tcp_sport=%d\n", nids_tcp_variable.tcp_sport);
    a_number = atoi (tcp_header_string_object.dport);
    nids_tcp_variable.tcp_dport = a_number;
    printf ("nids_tcp_variable.tcp_dport=%d\n", nids_tcp_variable.tcp_dport);
    a_long_number = atoll (tcp_header_string_object.seq);
    nids_tcp_variable.tcp_sequence = a_long_number;
    printf ("nids_tcp_variable.tcp_sequence=%u\n",
            nids_tcp_variable.tcp_sequence);
    a_long_number = atoll (tcp_header_string_object.ack);
    nids_tcp_variable.tcp_ack = a_long_number;
    printf ("nids_tcp_variable.tcp_ack=%u\n", nids_tcp_variable.tcp_ack);
    a_number = atoi (tcp_header_string_object.doff);
}

```

```

nids_tcp_variable.tcp_hlength = a_number;
printf ("nids_tcp_variable.tcp_hlength=%d\n",
        nids_tcp_variable.tcp_hlength);
sprintf (nids_tcp_variable.tcp_flags, "%s", tcp_header_string_object.flags);
printf ("nids_tcp_variable.tcp_flags=%s\n", nids_tcp_variable.tcp_flags);
a_number = atoi (tcp_header_string_object.win);
nids_tcp_variable.tcp_window = a_number;
printf ("nids_tcp_variable.tcp_window=%d\n", nids_tcp_variable.tcp_window);
a_number = atoi (tcp_header_string_object.urp);
nids_tcp_variable.tcp_urge = a_number;
printf ("nids_tcp_variable.tcp_urge=%d\n", nids_tcp_variable.tcp_urge);
sprintf (nids_tcp_variable.tcp_content, "%s", tcp_content_object);
printf ("nids_tcp_variable.tcp_content=%s\n",
        nids_tcp_variable.tcp_content);
}

```

(3) 获取 UDP 信息

```

void
get_udp_variable (void)
{
    int a_number;
    a_number = atoi (udp_header_string_object.sport);
    nids_udp_variable.udp_sport = a_number;
    printf ("nids_udp_variable.udp_sport=%d\n", nids_udp_variable.udp_sport);
    a_number = atoi (udp_header_string_object.dport);
    nids_udp_variable.udp_dport = a_number;
    printf ("nids_udp_variable.udp_dport=%d\n", nids_udp_variable.udp_dport);
    a_number = atoi (udp_header_string_object.len);
    nids_udp_variable.udp_length = a_number;
    printf ("nids_udp_variable.udp_length=%d\n", nids_udp_variable.udp_length);
    a_number = atoi (udp_header_string_object.cksum);
    nids_udp_variable.udp_checksum = a_number;
    printf ("nids_udp_variable.udp_checksum=%d\n",
            nids_udp_variable.udp_checksum);
    sprintf (nids_udp_variable.udp_content, "%s", udp_content_object);
    printf ("nids_udp_variable.udp_content=%s\n",
            nids_udp_variable.udp_content);
}

```

(4) 获取 ICMP 信息

```

void
get_icmp_variable (void)
{

```

```

int a_number;
a_number = atoi (icmp_header_string_object.type);
nids_icmp_variable.icmp_type = a_number;
printf ("nids_icmp_variable.icmp_type=%d\n", nids_icmp_variable.icmp_type);
a_number = atoi (icmp_header_string_object.code);
nids_icmp_variable.icmp_code = a_number;
printf ("nids_icmp_variable.icmp_code=%d\n", nids_icmp_variable.icmp_code);
a_number = atoi (icmp_header_string_object.cksum);
nids_icmp_variable.icmp_checksum = a_number;
printf ("nids_icmp_variable.icmp_checksum=%d\n",
        nids_icmp_variable.icmp_checksum);
sprintf (nids_icmp_variable.icmp_content, "%s", icmp_content_object);
printf ("nids_tcp_variable.tcp_content=%s\n",
        nids_icmp_variable.icmp_content);
}

```

(5) 获取协议变量对应类型的数值，即从规则中读出表达式的变量的存储的值是什么，例如，“tcp_dport=80”中的变量就是tcp_dport，而值就是80。

```

int
get_protocol_variable (char variable[1024])
{
if (strcmp (variable, "ip_hlength") == 0)
    //协议变量是 ip_hlength
return nids_ip_variable.ip_hlength;
if (strcmp (variable, "ip_length") == 0)
    //协议变量是 ip_length
return nids_ip_variable.ip_length;
if (strcmp (variable, "ip_version") == 0)
    //协议变量是 ip_version
return nids_ip_variable.ip_version;
if (strcmp (variable, "ip_service") == 0)
    //协议变量是 ip_service
return nids_ip_variable.ip_service;
if (strcmp (variable, "ip_ident") == 0)
    //协议变量是 ip_ident
return nids_ip_variable.ip_ident;
if (strcmp (variable, "ip_flags") == 0)
    //协议变量是 ip_flags
return nids_ip_variable.ip_flags;
if (strcmp (variable, "ip_distance") == 0)
    //协议变量是 ip_distance
return nids_ip_variable.ip_distance;
if (strcmp (variable, "ip_ttl") == 0)

```

```
//协议变量是 ip_ttl
return nids_ip_variable.ip_ttl;
if (strcmp (variable, "tcp_sport") == 0)
//协议变量是 tcp_sport
return nids_tcp_variable.tcp_sport;
if (strcmp (variable, "tcp_dport") == 0)
//协议变量是 tcp_dport
return nids_tcp_variable.tcp_dport;
if (strcmp (variable, "tcp_hlength") == 0)
//协议变量是 tcp_hlength
return nids_tcp_variable.tcp_hlength;
if (strcmp (variable, "tcp_window") == 0)
//协议变量是 tcp_window
return nids_tcp_variable.tcp_window;
if (strcmp (variable, "tcp_checksum") == 0)
//协议变量是 tcp_checksum
return nids_tcp_variable.tcp_checksum;
if (strcmp (variable, "tcp_urge") == 0)
//协议变量是 tcp_urge
return nids_tcp_variable.tcp_urge;
if (strcmp (variable, "udp_sport") == 0)
//协议变量是 udp_sport
return nids_udp_variable.udp_sport;
if (strcmp (variable, "udp_dport") == 0)
//协议变量是 udp_dport
return nids_udp_variable.udp_dport;
if (strcmp (variable, "dup_length") == 0)
//协议变量是 dup_length
return nids_udp_variable.udp_length;
if (strcmp (variable, "udp_checksum") == 0)
//协议变量是 udp_checksum
return nids_udp_variable.udp_checksum;
if (strcmp (variable, "icmp_type") == 0)
//协议变量是 icmp_type
return nids_icmp_variable.icmp_type;
if (strcmp (variable, "icmp_code") == 0)
//协议变量是 icmp_code
return nids_icmp_variable.icmp_code;
if (strcmp (variable, "icmp_checksum") == 0)
//协议变量是 icmp_checksum
return nids_icmp_variable.icmp_checksum;
return -1;
}
```

9.2.2 规则匹配

在判断规则匹配时,最主要的是判断“事件定义”的真假,事件定义存储在 event_definition 中。还是以 UNICODE Event 事件为例,即判断事件定义“tcp_dport=80&tcp_content^%c0%af”的真假。其步骤就是先判断事件定义运算式“tcp_dport=80”的真假,再判断事件定义运算式“tcp_content^%c0%af”的真假,再求两个真假值的“与”。若为真,则匹配成功。

判断事件定义运算式的真假,例如“tcp_dport=80”。

在实现规则匹配开始,调用了下面两个函数。

```
read_rules_from_file("rules");
read_statement_from_rules();
```

这两个函数分别在函数 threads_click (GtkWidget * widget, gpointer data)中被调用,请参考前面的章节。图 9-5 是入侵检测模块中使用到的相关函数的调用关系。第一个函数的作用是从文件中读取入侵规则,因为我们定义入侵规则都放在文件中,这个文件的名字是 rules。所以必须开始把规则读入内存中。此函数就实现了这个功能。

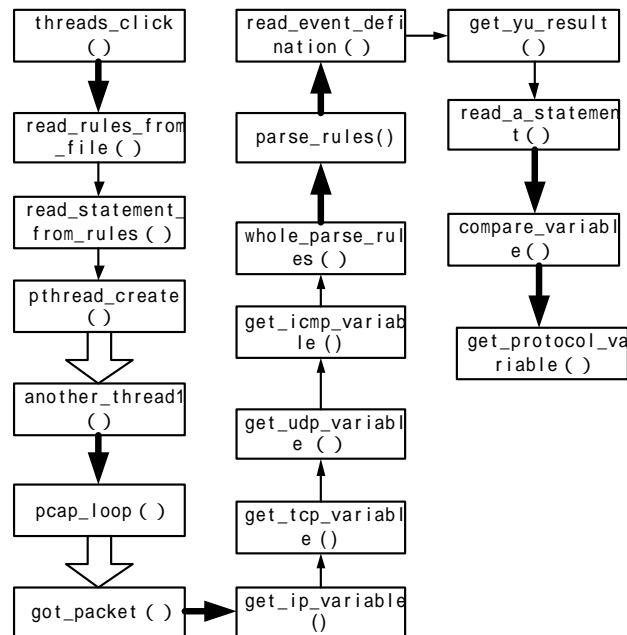


图 9-5 函数关系

```
void
read_rules_from_file(char *filename)
{
    FILE *fp;
    fp = fopen(filename, "r"); //打开文件
    total_rules_number = 0;
    clear_all_rules();
    //清除所有规则变量信息
}
```

```

        while (!feof (fp))
        {
            sprintf (total_rules[total_rules_number], "%s", ReadFile (fp, '\n'));
            //从规则库中读出规则，由 ReadFile()函数完成
            total_rules_number++;
        }
        total_rules_number--;
    }
}

```

此函数中调用了函数 `clear_all_rules ()`;其作用是清零。在调用这个函数后，入侵规则就存放在变量 `total_rules` 中。变量 `total_rules_number` 是入侵规则的个数。其中从规则库中读入规则的函数 `ReadFile ()` 实现如下：

```

char *
ReadFile (FILE * fp, char flag)
{
    char word[10000];
    char *string;
    int x;
    x = 0;
    while (1)
    {
        word[x] = (char) fgetc (fp);
        if ((word[x] == flag) || (feof (fp)))
        {
            if (word[x] != flag)
                x++;
            word[x] = '\0';
            string = word;
            return string;
        }
        ++x;
    }
}

```

函数 `read_statement_from_rules ()`是从一个规则中读出事件的各个部分，分别为事件名称、事件协议、事件代码、事件定义、事件说明和响应方式。其实现如下：

```

void
read_statement_from_rules (void)
{
    int i;
    char *rules;
    for (i = 0; i < total_rules_number; i++)
    {

```

```

        rules = total_rules[i];
        printf ("total_rules[%d] is %s\n", i, rules);
        sprintf (event_name[i], "%s", ReadData (rules, '\t'));
        printf ("event_name[%d] is %s\n", i, event_name[i]);
        sprintf (event_protocol[i], "%s", ReadData (rules, '\t'));
        printf ("event_protocol[%d] is %s\n", i, event_protocol[i]);
        sprintf (event_code[i], "%s", ReadData (rules, '\t'));
        printf ("event_code[%d] is %s\n", i, event_code[i]);
        sprintf (event_defination[i], "%s", ReadData (rules, '\t'));
        printf ("event_defination[%d] is %s\n", i, event_defination[i]);
        sprintf (event_information[i], "%s", ReadData (rules, '\t'));
        printf ("event_information[%d] is %s\n", i, event_information[i]);
        sprintf (alert_type[i], "%s", rules);
        printf ("alert_type[%d] is %s\n", i, alert_type[i]);
    }
    printf (" test in read_statement_from_rules\n");
}

```

其中有个函数 ReadData (), 其作用是一次读出一个规则的各个部分并返回其值。其函数实现如下 :

```

char *
ReadData (char *line, char flag)
{
    //参数 line 是一个规则 , 也就是一行 , flag 是规则中每个部分相隔的分隔符
    int i = 0;
    int j;
    char word[1024];
    char *string;
    for (i = 0; ((line[i]) && (line[i] != flag)); i++)
        word[i] = line[i];
    word[i] = '\0';
    while (line[i] == flag)
        ++i;
    j = 0;
    while (line[j++] = line[i++]);
    string = word;
    return string;
}

```

然后在函数 got_packet(此函数的实现 , 请参考前面章节) 中调用了函数 whole_parse_rules , 此函数完成了入侵规则匹配的功能。其中相关的函数调用关系如图 9-5 所示。

whole_parse_rules () 函数实现如下 :


```

void
whole_parse_rules (void)
{
    char *rules;
    int i;
    printf ("whole_parse_rules\n");
    for (i = 0; i < total_rules_number; i++)
    {
        printf ("total_rules_number is %d\n", total_rules_number);
        rules = event_defination[i];
        if (parse_rules (rules) == 1)
        {
            gdk_threads_enter ();
            add_event_to_clist (i); //把入侵事件填入到界面
            gdk_threads_leave ();
            make_log (i);          //把入侵事件记录下来，供事后分析
            if (strcmp (alert_type[i], "sound") == 0) //响应方式
                make_sound ();    //声音警报，请参考响应模块实现
            if (strcmp (alert_type[i], "flash") == 0)
                make_flash ();    //灯光闪烁警报，请参考响应模块实现
            if (strcmp (alert_type[i], "email") == 0)
                make_email ();    //通过发 e-mail 通知
        }
    }
}

```

前面讲过要判断定义运算式的真假，如定义运算式“tcp_dport=80”的值是否是“真”还是“假”。此功能由如下函数实现。

```
int compare_variable(char variable[1024],char fuhao,char *result)
```

/*求出事件定义运算式的真假，这里是判断事件数据是十进制数字的情况。参数 variable 表示协议变量，如“tcp_dport”，fuhao 表示协议运算符，如“=”，result 表示事件数据，如“80”。*/

```

{
    int number_result=0; int variable_value;
    number_result=atoi(result);
    variable_value=get_protocol_variable(variable);
    /*取得协议变量的值*/
    if(variable_value==--1)
        return 0;
    if(fuhao=='=') /*判断协议运算符是否为'='*/
    {if(variable_value==number_result)
        /*判断规则中的协议值与真实的协议变量值是否相等*/
    }
}

```

```

return 1;}
if(fuhao=='~') /*判断协议运算符是否为'~'*/
{if(variable_value!=number_result)
/*判断规则中的协议值与真实的协议变量值是否不相等*/
return 1;}
if(fuhao=='>') /*判断协议运算符是否为'>'*/
{if(variable_value>number_result)
/*判断规则中的协议值与真实的协议变量值是否大于*/
return 1;}
if(fuhao=='<') /*判断协议运算符是否为'<'*/
{if(variable_value<number_result)
/*判断规则中的协议值与真实的协议变量值是否大于*/
return 1;}
return 0; /*默认返回 0*/
}

```

判断一个事件定义的真假，例如“tcp_dport=80&tcp_content^%c0%af”。其核心代码如下：

```

int parse_rules(char *rules)
/*求出事件定义的真假，参数 rules 表示事件定义*/
{
    int i=0;
    int j=0;
    int result;
    int yu[1024];
    int yu_number=0;
    read_event_definition(rules); /*从规则中读取事件定义，请参考第 8 章*/
    for(j=0;j<state_number-1;j++)
    {
        if(relation[j]=='&') /*判断关系符号是否为'&'*/
        {yu[yu_number]=get_yu_result(event_statement[j],
event_statement[j+1]);
/*求出两个事件定义运算式相与后的真假*/
        yu_number++;
        }
    }
    if(yu_number==0)
/*判断事件定义是否就一个事件定义运算式*/
    return read_a_statement(rules); /*返回事件定义运算式的真假*/
    for(i=0;i<yu_number;i++)
    {if(yu[yu_number]==1) /*判断事件定义的真假*/
        {
            return 1;
        }
    }
    return 0; /*默认返回 0 值*/
}

```

在此函数中 `get_yu_result ()` 被调用，其实现如下：

```
int
get_yu_result (char *str1, char *str2)
{
    if (read_a_statement (str1) && (read_a_statement (str2)))
        return 1;
    else
        return 0;
}
```

在此函数中还有一个函数 `read_a_statement ()` 被调用，此函数的实现如下：

```
int
read_a_statement (char *statement)
{
    int i = 0;
    int length = strlen (statement);
    char fuhao;
    char *result;
    int number_result = 0;
    char variable[1024];
    for (i = 0; i < length; i++)
    {
        if (*(statement + i) == '=') //协议运算符是 =
        {
            fuhao = '=';
            break;
        }
        if (*(statement + i) == '~') //协议运算符是 ~
        {
            fuhao = '~';
            break;
        }
        if (*(statement + i) == '>') //协议运算符是 >
        {
            fuhao = '>';
            break;
        }
        if (*(statement + i) == '<') //协议运算符是 <
        {
            fuhao = '<';
            break;
        }
    }
}
```

```
        if (*(statement + i) == '^')           //协议运算符是 ^
        {
            fuhao = '^';
            break;
        }
        variable[i] = *(statement + i);
    }
    printf ("i= %d\n", i);
    variable[i] = '\0';
    result = statement + i + 1;
    if (strcmp (variable, "ip_sip") == 0) //协议变量是 ip_sip
    {
        if (fuhao == '=')
        {
            if (strcmp (nids_ip_variable.ip_sip, result) == 0)
                return 1;
        }
        if (fuhao == '~')
        {
            if (strcmp (nids_ip_variable.ip_sip, result) != 0)
                return 1;
        }
    }
    if (strcmp (variable, "ip_dip") == 0) //协议变量是 ip_dip
    {
        if (fuhao == '=')
        {
            if (strcmp (nids_ip_variable.ip_dip, result) == 0)
                return 1;
        }
        if (fuhao == '~')
        {
            if (strcmp (nids_ip_variable.ip_dip, result) != 0)
                return 1;
        }
    }
    if (strcmp (variable, "ip_type") == 0) //协议变量是 ip_type
    {
        if (fuhao == '=')
        {
            if (strcmp (nids_ip_variable.ip_type, result) == 0)
                return 1;
```

```
    }
    if (fuhao == '~')
    {
        if (strcmp (nids_ip_variable.ip_type, result) != 0)
            return 1;
    }
}

if (strcmp (variable, "tcp_flags") == 0)    //协议变量是 tcp_flags
{
    if (fuhao == '=')
    {
        if (strcmp (nids_tcp_variable.tcp_flags, result) == 0)
            return 1;
    }
    if (fuhao == '~')
    {
        if (strcmp (nids_tcp_variable.tcp_flags, result) != 0)
            return 1;
    }
}

if (strcmp (variable, "tcp_content") == 0)    //协议变量是 tcp_content
{
    if (fuhao == '^')
    {
        if (strstr (nids_tcp_variable.tcp_content, result) != NULL)
            return 1;
    }
}

if (strcmp (variable, "udp_content") == 0)    //协议变量是 udp_content
{
    if (fuhao == '^')
    {
        if (strstr (nids_udp_variable.udp_content, result) != NULL)
            return 1;
    }
}

if (strcmp (variable, "icmp_content") == 0)    //协议变量是 icmp_content
{
    if (fuhao == '^')
    {
        if (strstr (nids_icmp_variable.icmp_content, result) != NULL)
            return 1;
    }
}
```

```

    }
}
if (compare_variable (variable, fuhao, result) == 1) //判断定义运算式的真假
    return 1;
return 0;
}

```

图 9-6 是实时检测到入侵后的运行界面。当检测到入侵行为时就把入侵事件写入入侵事件窗口，同时也把这些入侵事件写入日志。

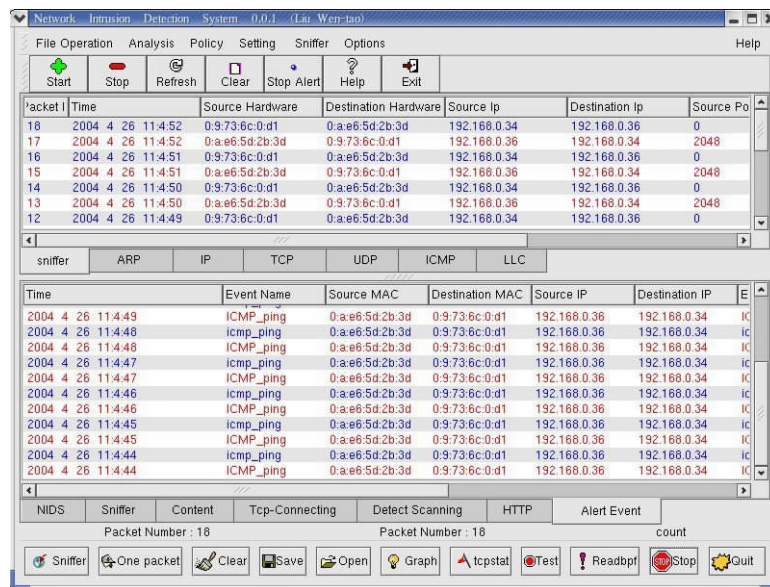


图 9-6 入侵事件显示窗口

9.2.3 检测扫描行为

为了检测到网络入侵行为，除了使用上面讲的基于协议分析的规则模式匹配算法之外，还可以不用规则来描述入侵行为，这个时候就可以单个地对入侵行为编写一个检测函数来检测。例如，对网络扫描行为的判断。我们可以实现一个函数来检测入侵行为。

检测扫描行为的函数实现如下，此函数在函数 `tcp_connecting_to_text3()` 中被设置，请参考 6.3.2 节。

```

static void
my_nids_syslog (int type, int errnum, struct ip *iph, void *data)
{
    static int scan_number = 0;
    char saddr[20], daddr[20];
    char buf[1024];
    struct host *this_host;
    unsigned char flagsand = 255, flagsor = 0;
    int i;

```

```
char content[1024];
if (detect_scanning_yesno == 0) //判断是否检测
{
    return;
}
switch (type)
{
case NIDS_WARN_IP:
    if (errnum != NIDS_WARN_IP_HDR)
    {
        strcpy (saddr, int_ntoa (iph->ip_src.s_addr));
        strcpy (daddr, int_ntoa (iph->ip_dst.s_addr));
        syslog (nids_params.syslog_level,
            "%s, packet (apparently) from %s to %s\n",
            nids_warnings[errnum], saddr, daddr);
    }
    else
        syslog (nids_params.syslog_level, "%s\n", nids_warnings[errnum]);
    break;
case NIDS_WARN_TCP:
    strcpy (saddr, int_ntoa (iph->ip_src.s_addr));
    strcpy (daddr, int_ntoa (iph->ip_dst.s_addr));
    if (errnum != NIDS_WARN_TCP_HDR)
        syslog (nids_params.syslog_level,
            "%s,from %s:%hi to %s:%hi\n", nids_warnings[errnum], saddr,
            ntohs (((struct tcphdr *) data)->th_sport), daddr,
            ntohs (((struct tcphdr *) data)->th_dport));
    else
        syslog (nids_params.syslog_level, "%s,from %s to %s\n",
            nids_warnings[errnum], saddr, daddr);
    break;
case NIDS_WARN_SCAN:
    scan_number++;
    this_host = (struct host *) data;
    sprintf (buf, "Scan from :\n");
    gdk_threads_enter ();
    insert_text_to_text4 (buf);
    gdk_threads_leave ();
    sprintf (buf, "%s\n", int_ntoa (this_host->addr));
    gdk_threads_enter ();
    insert_text_to_text4_green (buf);
    gdk_threads_leave ();
```

```
    sprintf (buf, "Scanned Ports :\n");
    gdk_threads_enter ();
    insert_text_to_text4 (buf);
    gdk_threads_leave ();
    sprintf (buf, "");
    for (i = 0; i < this_host->n_packets; i++)
    {
        strcat (buf, int_ntoa (this_host->packets[i].addr));
        sprintf (buf + strlen (buf), ":%hi\n", this_host->packets[i].port);
        flagsand &= this_host->packets[i].flags;
        flagsor |= this_host->packets[i].flags;
    }
    gdk_threads_enter ();
    insert_text_to_text4_green (buf);
    gdk_threads_leave ();
    sprintf (buf, "");
    if (flagsand == flagsor)
    {
        i = flagsand;
        switch (flagsand)
        {
            case 2:
                strcat (buf, "scan type: SYN\n");
                break;
            case 0:
                strcat (buf, "scan type: NULL\n");
                break;
            case 1:
                strcat (buf, "scan type: FIN\n");
                break;
            default:
                sprintf (buf + strlen (buf), "flags=0x%x\n", i);
        }
    }
    else
    strcat (buf, "various flags\n");
    syslog (nids_params.syslog_level, "%s", buf);
    gdk_threads_enter ();
    insert_text_to_text4 (buf);
    gdk_threads_leave ();
    sprintf (buf,
        "----- %d -----\n",
```



```

        scan_number);
gdk_threads_enter ();
insert_text_to_text4_red (buf);
gdk_threads_leave ();
break;
default:
syslog (nids_params.syslog_level, "Unknown warning number ?\n");
sprintf (content, "default");
gdk_threads_enter ();
insert_text_to_text4 (content);
gdk_threads_leave ();
break;
}
}

```

其中一个结构体定义如下：

```

struct host
{
    struct host *next;
    struct host *prev;
    u_int addr;
    int modtime;
    int n_packets;
    struct scan *packets;
};

```

为了检测到扫描行为，我们使用 nmap 软件来对另外一台机器进行扫描，如图 9-7 所示。

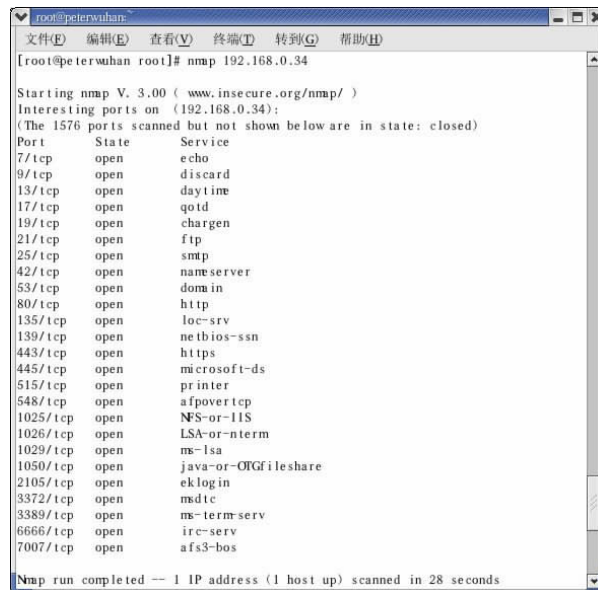


图 9-7 扫描主机

图 9-8 是检测到扫描行为的运行界面。可以看到不同的扫描行为，例如，SYN 扫描或者 FIN 扫描，都可以检测出来。

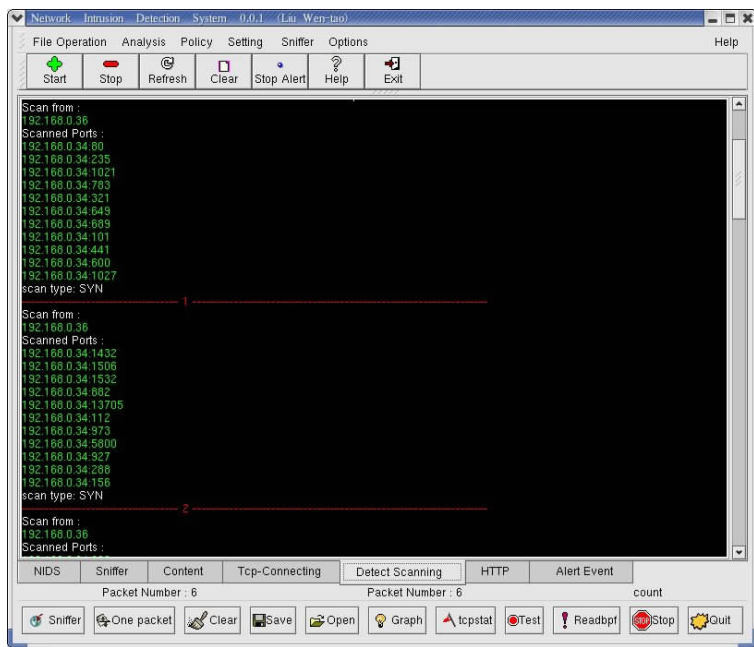


图 9-8 检测到的扫描行为

9.3 小结

入侵事件检测模块是入侵检测系统的核心模块，它实现了入侵检测的功能，所以它是入侵检测系统中最有价值的一部分。前面的几个模块，如数据包捕获模块、网络协议分析模块、规则解析模块等都是此模块的基础。此模块实现了系统的主要功能。因为入侵检测模块的实现是在前面几个模块的基础上进行，所以其实现不是很复杂。因为前面几个模块都实现了大量的功能，例如，网络协议分析模块把捕获到的数据包进行了协议分析，因而其分析的结果就是为入侵事件检测模块所要用的，也就是入侵事件检测模块的分析对象。另外在规则解析模块中，对规则的解析也是入侵事件检测模块所需要的，因为入侵事件检测模块使用的规则匹配功能就必须先要把规则给解析出来才能实现。

第 10 章 入侵响应模块设计与实现

响应就是当入侵检测系统检测到入侵行为时所作出的反应动作。响应是一个入侵检测系统必不可少的一部分，没有它入侵检测就失去了存在的价值。响应是网络安全模型 PPDR 中的 R，它在整个动态安全模型中扮演一个重要的角色。

IDS 作为一种积极主动的安全防护技术，不仅能够检测来自网络外部的入侵，同时还能够监督网络内部用户的活动。

在 IDS 系统的所有功能中，响应入侵是最要的部分。

10.1 响应的类型

入侵检测系统的响应可以分为主动响应(active responses)和被动响应(passive response)两种类型。

1. 主动响应

在主动响应中，系统自动地或以用户设置的方式来阻断攻击过程或以其他方式影响攻击过程。可以选择的措施有几种。第一种就是针对入侵者采取的措施。这是最具侵略性的形式，即追踪入侵者实施攻击的发起地，并采取措施以禁用入侵者的机器或网络连接。但这种措施会带来负面影响，这种网络反击带来的危险包括：

- 根据网络黑客的习惯，系统鉴别出的攻击发起地很可能只是该黑客的另一受害人；
- 就算攻击者来自他自己的系统，但攻击方用的也许是一个假 IP 地址；
- 反击的结果可能挑起更猛烈的攻击；
- 在很多情况下，反击会冒很大的风险，可能成为刑事起诉或民事诉讼的对象。

第二种措施就是修正系统。修正系统以弥补引起攻击的缺陷，这种保护自身安全而装备的“自疗”系统类似于生物体的免疫系统，可以辨认出问题所在并将引起问题的部分隔离起来。产生合适的响应对于鉴定问题所在也是很有帮助的。在一些入侵检测系统中，这类响应可以改变分析引擎的一些参数设置和操作方法，例如提高敏感级别。也可以通过添加规则，如提高某类攻击的可疑级别和扩大监控范围，来改变专家系统，从而达到在更好的粒度层次上收集信息的目的。这种策略类似于在实时过程控制系统中，利用当前系统进程的结果来调整和优化以后的进程。

第三种措施就是收集更详细的信息。当被保护的系统关系重大而且当事人要求法律赔偿时，这种措施就很有价值。有时，这种记录日志的响应与专用的服务器结合起来使用，作为可以将入侵者转移的环境来安置。专用服务器在这里是指诱骗系统，它有很多名称，最常见的是蜜罐(honey pots)，鸟饵(decoys)或玻璃鱼缸(fishbowls)。这些服务器可以模拟关键系统的文件系统和其他系统特征，引诱攻击者进入，记录下攻击者的行为，从而获得关于攻击者的详细信息，作为进一步采取法律措施的证据。用这种方法收集的信息对网络安全威胁

趋势分析也是很有价值的，而对于必须在敌对威胁环境中运行的系统或面临大量攻击的系统尤其有用。如政府的 Web 服务器或高收入的电子商务站点。

主动响应能够阻止正在进行的攻击，使得攻击者不能够继续访问。更为主动的响应则是 IDS 系统在检测到攻击时会对攻击者进行反击，这种响应存在一定风险，有可能影响网络上的无辜用户，应该谨慎采用。

2. 被动响应

被动响应是指为用户提供信息，由用户决定接下来应该采取什么措施。被动响应根据危险程度高低的次序提交给用户，这里的危险程度是警报机制与问题汇报最主要的区别。

大多数入侵检测系统提供以多种形式生成报警的响应方式。这种机动性使用户可以定制适合自己的系统运行过程的报警。

入侵检测系统可以被设计成与网络管理工具一起协同运作。这些系统可以更好地利用网络管理的基础设备，在网络管理控制台上发送和显示警报警告。一些产品利用简单网络管理协议(SNMP)的消息或陷阱作为一种报警方式。这种结合可以带来几个好处，如可以利用普通通信信道的功能，提供适应当时网络环境的主动响应。此外，SNMP 陷阱让用户可以将响应检测到的问题的处理工作转移到接受陷阱的系统上。

10.2 入侵响应模块实现

通过警告的方式来说明有入侵行为发生。警告的种类有很多种，在此系统中用到了三种方式，一种是声音警告，一种是灯光闪烁警告，另一种是发送 E-mail。最后把入侵行为记录下来供事后分析。当然还可以使用其他形式的警告方式，其宗旨就是尽快把入侵行为报告给相关管理人员，以便管理人员来及时的处理。

在此系统中采用被动响应的方式。对检测到的入侵行为，进行实时的响应。

10.2.1 采用声音警报的方式来响应

当入侵规则中的响应方式 alert type 为 sound 时，表示用声音发警报。

```
void
make_sound (void)
{
    //此函数在函数 whole_parse_rules (void)中被调用。
    pthread_t another;
    sound_thread_stop = 0;
    pthread_create (&another, NULL, sound_thread, NULL);
    printf ("gdk_beep()\n");
}
```

在这里调用了创建线程的函数来创建一个线程，此线程回调函数为 sound_thread，其实现如下：

```
void *
sound_thread (void *args)
```

```

{
    if (sound_thread_stop == 1)
        pthread_exit (0);
    sound_timer = gtk_timeout_add (150, show_sound, NULL);
}

```

在此函数里面定义了一个定时器，定时器的回调函数是 showa_sound ()。此函数实现如下：

```

gint
show_sound (gpointer data)
{
    if (sound_thread_stop == 1)
    {
        return FALSE;
    }
    gdk_beep (); //此是发声函数
    return TRUE;
}

```

10.2.2 采用灯光闪烁的方式来发警报

当入侵规则中的响应方式 alert type 为 flash 时，表示灯光闪烁来发警报。

```

void
make_flash (void)
{ //此函数在函数 whole_parse_rules (void)中被调用。
    create_flash ();
}

```

在此函数中调用了 create_flash()函数，其实现如下：

```

void
create_flash (void)
{
    //first remove the timer,so can call again
    gtk_timeout_remove (ptimer_flash);
    ptimer_flash = gtk_timeout_add (200, flash_timeout, NULL);
}

```

在此函数中用定时器，产生动画效果。其回调函数为 flash_timeout ()，其实现如下：

```

gint
flash_timeout (gpointer data)
{
    if (bian_flash == 1) //显示一种效果
    {
        show_window1 (); //显示图像
        if (window_flash2 != NULL)

```

```

gtk_widget_destroy (window_flash2);
    bian_flash = 2;
}
else if (bian_flash == 2) //显示另一种效果，产生动画
{
    show_window2 ();    //显示图像
    if (window_flash1 != NULL)
gtk_widget_destroy (window_flash1);
    bian_flash = 1;
}
return TRUE;
}

```

在此函数中，有两种效果交替显示，这样就产生了动画的效果。两个显示图像的函数实现如下：

```

void
show_window1 ()
{
    //使用 GTK + 显示第一种效果图像
    GtkWidget *pixmap, *fixed;
    GdkPixmap *gdk_pixmap;
    GdkBitmap *mask;
    GtkStyle *style;
    GdkGC *gc;
    window_flash1 = gtk_window_new (GTK_WINDOW_POPUP);
    gtk_widget_set_uposition (window_flash1, 20, 400);
    gtk_widget_set_usize (GTK_WIDGET (window_flash1), 60, 60);
    /* Now for the pixmap and the pixmap widget */
    style = gtk_widget_get_default_style ();
    gc = style->black_gc;
    gdk_pixmap = gdk_pixmap_create_from_xpm_d (window_flash1->window, &mask,
                                                &style->bg[GTK_STATE_NORMAL],
                                                magick);
    pixmap = gtk_pixmap_new (gdk_pixmap, mask);
    gtk_widget_show (pixmap);
    /* To display the pixmap, we use a fixed widget to place the pixmap */
    fixed = gtk_fixed_new ();
    gtk_widget_set_usize (fixed, 200, 200);
    gtk_fixed_put (GTK_FIXED (fixed), pixmap, 0, 0);
    gtk_container_add (GTK_CONTAINER (window_flash1), fixed);
    gtk_widget_show (fixed);
    /* This masks out everything except for the image itself */
    gtk_widget_shape_combine_mask (window_flash1, mask, 0, 0);
    /* show the window */
}

```

```
    gtk_widget_show (window_flash1);
}
void
show_window2 ()
{
    //显示第二种效果
    GtkWidget *pixmap, *fixed;
    GdkPixmap *gdk_pixmap;
    GdkBitmap *mask;
    GtkStyle *style;
    GdkGC *gc;
    window_flash2 = gtk_window_new (GTK_WINDOW_POPUP);
    gtk_widget_set_uposition (window_flash2, 20, 400);
    gtk_widget_set_usize (GTK_WIDGET (window_flash2), 60, 60);
    /* Now for the pixmap and the pixmap widget */
    style = gtk_widget_get_default_style ();
    gc = style->black_gc;
    gdk_pixmap = gdk_pixmap_create_from_xpm_d (window_flash2->window, &mask,
                                                &style->bg[GTK_STATE_NORMAL],
                                                magick1);
    pixmap = gtk_pixmap_new (gdk_pixmap, mask);
    gtk_widget_show (pixmap);
    fixed = gtk_fixed_new ();
    gtk_widget_set_usize (fixed, 200, 200);
    gtk_fixed_put (GTK_FIXED (fixed), pixmap, 0, 0);
    gtk_container_add (GTK_CONTAINER (window_flash2), fixed);
    gtk_widget_show (fixed);
    gtk_widget_shape_combine_mask (window_flash2, mask, 0, 0);
    gtk_widget_show (window_flash2);
}
```

10.2.3 使用日志来记录

```
void
make_log (int number)
{
    int log;
    char filename[1024];
    char today[1024];
    char timestr[1024];
    char log_string[1024];
    strcpy (log_string, "");
    getcurrenttime (timestr);
```

```

    strcat (log_string, timestr);
    strcat (log_string, " event_name:");
    strcat (log_string, event_name[number]);
    strcat (log_string, " source_hardware:");
    strcat (log_string, snifferpacket.source_hardware);
    strcat (log_string, " destination_hardware:");
    strcat (log_string, snifferpacket.destination_hardware);
    strcat (log_string, " source_ip:");
    strcat (log_string, ip_header_string_object.source_ip);
    strcat (log_string, " destination_ip:");
    strcat (log_string, ip_header_string_object.destination_ip);
    strcat (log_string, " event_protocol:");
    strcat (log_string, event_protocol[number]);
    strcat (log_string, " event_code:");
    strcat (log_string, event_code[number]);
    strcat (log_string, " event_defination:");
    strcat (log_string, event_defination[number]);
    strcat (log_string, " event_information:");
    strcat (log_string, event_information[number]);
    strcat (log_string, " alert_type:");
    strcat (log_string, alert_type[number]);
    strcat (log_string, "\n*****\n");
    get_current_day (today);
    sprintf (filename, "./event_log/%s.log", today);
    log = open (filename, O_WRONLY | O_CREAT | O_APPEND, 0666);
    write (log, log_string, strlen (log_string));
    close (log);
}

```

当检测到入侵行为时就把入侵事件写入入侵事件窗口。同时也把这些入侵事件写入日志，如下所示：

2002 10 22 16:54:48	事件发生的时间
event_name:ICMP_ping	事件名称
source_hardware:52:54:ab:27:c0:4	源硬件地址
destination_hardware:0:20:da:d3:c5:80	目的硬件地址
source_ip:192.168.1.13	源 IP 地址
destination_ip:192.168.1.4	目的 IP 地址
event_protocol:ICMP	事件协议
event_code:100001	事件代码
event_defination:icmp_type=8	事件定义
event_information:This is a ping event.	事件说明
alert_type:sound	事件响应方式

第 11 章 界面模块设计与实现

界面模块是用来管理其他模块的，它的实现是为了更好地管理其他模块，使系统更容易表示和使用。通过界面可以操作任何其他的模块，例如，可以通过界面操作动态挂载和卸载存储模块，这样，当需要存储信息的时候就启动存储模块，当不需要的时候，就可以卸载存储模块，操作非常方便。界面模块的另外一个主要功能就是显示信息，有很多信息要显示，例如网络协议信息，在协议分析之后，把分析的结果显示出来。协议分析结果显示方式各种各样，例如用十六进制或者字符形式显示协议内容。还有网络数据库信息，把数据库信息显示出来，当然可以显示各种数据库信息，根据需要进行。还有就是入侵事件的显示，当有入侵发生的时候，就显示到事件界面中。还有规则库操作界面，可以用来显示规则库。

Linux 下设计界面的技术很多，在本系统中使用 GTK 来设计界面。

11.1 GTK 概述

在 Linux 下用 GTK+(GIMP Toolkit) 来编制窗口程序，目的主要是给用户一个更方便友好的界面来控制其他模块和浏览结果。GTK+是一套用于创建图形用户界面的工具包。GIMP (GNU Image Manipulation Program) 是 GNU 图像处理程序。GTK+最早用于 GIMP 的开发，现在已广泛用于开发 Linux 的图形用户界面。它是开放源代码软件。

图 11-1 表示 GTK+的应用程序层次结构。从图中可以看到 GTK+在开发系统中所处的位置，GTK+是建立在 GDK (GIMP Drawing Kit，简称 GDK) 上的构件库。GDK 是依赖于平台的应用软件开发工具集，基本上是将 XLIB 的绘图功能包装起来。XLIB 是 X 窗口系统的底层函数库。因为 GTK+频繁用到 GLIB 库，所以 GTK+的可移植性和功能依赖于 GLIB 库。GLIB 库定义了许多有用的数据结构和函数，如连接表操作、树、内存管理等。

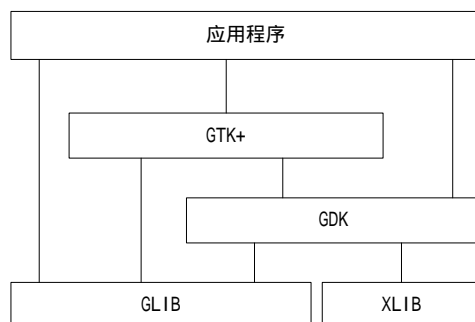


图 11-1 GTK+的应用程序层次

GTK+库是用 C 语言编写的、面向对象的、由事件驱动的工具库，它提供了多种语言使用 GTK+的接口，类似大多数现代的图形用户界面工具集，屏幕用控件建立，如窗口、组合框、正文框和按钮等等。在这些控件中设置回调函数来完成对信号的处理，信号通常是鼠标

或键盘产生的事件引起的。当回调函数收到信号时，应用程序响应信号，并完成相应处理。GTK 是一个事件驱动的工具集，一个 GTK 实例一般是在 `gtk_main` 上等待直到一个事件发生，这时控制权就被传递给响应的函数。注意一点，GTK 虽然是用 C 语言来写的，即它只是提供了 C 语言的函数调用接口，但是你可以使用很多语言来使用 GTK，这是因为 GTK+ 已经被绑定到几乎所有流行的语言上了，如：C++，Guile，Perl，Python，TOM，Ada95，Objective C，Free Pascal，and Eiffel。所以你可以使用你熟悉的语言来编写 GTK 程序。

11.2 GTK 控件

GTK 里面包括很多控件，下面分别介绍最常用的一些控件，并介绍最常用的一些函数。读者可以参考其他更详细的资料来了解相关内容。GTK 很好使用，它的函数很多，很丰富，在使用了一段时间之后，就能很好的把握用 GTK 来设计 LINUX 界面的技巧。它没有很深的理论，也不像其他高级语言来设计界面的复杂消息响应过程。很多知识都可以举一反三，掌握 GTK 不需要花很长的时间。最关键的就是函数的调用，读者如果把各个控件的相关函数弄懂之后，那么用 GTK 来编写界面就非常容易。

1. GtkWidget

它是一般控件的数据结构，它相关的操作函数如下。

```
GtkWidget* gtk_widget_new (GtkType type,const gchar *first_property_name,...);
```

创建一个新的控件。其属性由所带的参数控制。

```
GtkWidget* gtk_widget_ref (GtkWidget *widget);
```

设置一个控件的引用。

```
void gtk_widget_unref (GtkWidget *widget);
```

取消一个控件的引用。

```
void gtk_widget_destroy (GtkWidget *widget);
```

撤销一个控件。

```
void gtk_widget_show (GtkWidget *widget);
```

显示一个控件。

```
void gtk_widget_hide (GtkWidget *widget);
```

隐藏一个控件。

```
void gtk_widget_show_all (GtkWidget *widget);
```

显示一个控件，并显示其所有子控件。

```
void gtk_widget_hide_all (GtkWidget *widget);
```

隐藏一个控件，并隐藏其所有子控件。

```
void gtk_widget_set_name (GtkWidget *widget,const gchar *name);
```

设置控件的名字。

```
G_CONST_RETURN gchar* gtk_widget_get_name (GtkWidget *widget);
```

获取控件的名字。

2. 按钮控件

GtkButton 就是按钮控件的数据结构。它相关的操作函数如下：

```
GtkWidget* gtk_button_new      (void);
```

创建一个新的按钮控件。

```
GtkWidget* gtk_button_new_with_label (const gchar *label);
```

创建一个包含一个标签控件的按钮控件，标签的名字由 label 给定。

```
void        gtk_button_clicked      (GtkButton *button);
```

发送一个 GtkButton::clicked 信号给一个相应的按钮。

```
void        gtk_button_enter        (GtkButton *button);
```

发送一个 GtkButton::enter 信号给按钮 button。

```
void        gtk_button_leave        (GtkButton *button);
```

发送一个 GtkButton::leave 信号给按钮 button。

```
G_CONST_RETURN gchar* gtk_button_get_label (GtkButton *button);
```

获取按钮的标签内容。

```
void        gtk_button_set_label      (GtkButton *button,  
                                       const gchar *label);
```

设置按钮的标签内容。

3. 文本输入控件

GtkEntry 就是单行文本输入控件的数据结构。它相关的操作函数如下：

```
GtkWidget* gtk_entry_new      (void);
```

创建一个新的文本输入控件。

```
GtkWidget* gtk_entry_new_with_max_length (gint max);
```

创建一个新的文本输入控件，其长度最大值是 max。

```
void        gtk_entry_set_text (GtkEntry *entry, const gchar *text);
```

用给定的值来设置控件中的文本来代替当前的内容。

```
void        gtk_entry_append_text (GtkEntry *entry, const gchar *text);
```

用给定的文本来增加控件中的内容，放在后面。

```
void        gtk_entry_prepend_text (GtkEntry *entry, const gchar *text);
```

用给定的文本来增加控件中的内容，放在前面。

```
G_CONST_RETURN gchar* gtk_entry_get_text (GtkEntry *entry);
```

获取控件中的文本内容。

```
void        gtk_entry_set_visibility (GtkEntry *entry, gboolean visible);
```

根据 visible 设置控件中的内容是否可见。

```
void        gtk_entry_set_editable (GtkEntry *entry, gboolean editable);
```

根据 `editable` 来设置控件是否可以编辑。

```
void gtk_entry_set_max_length (GtkEntry *entry, gint max);
```

用 `max` 来设置控件内容的最大长度。

```
gint gtk_entry_get_max_length (GtkEntry *entry);
```

获取控件中内容允许的最大长度。

4. 弹出窗口

`GtkDialog` 就是弹出窗口的数据结构。它相关的操作函数如下：

```
GtkWidget* gtk_dialog_new (void);
```

创建一个新的对话弹出窗口。

```
GtkWidget* gtk_dialog_new_with_buttons (const gchar *title, GtkWidget *parent, GtkDialogFlags flags, const gchar *first_button_text,...);
```

创建一个含有按钮的弹出窗口。

```
gint gtk_dialog_run (GtkDialog *dialog);
```

运行弹出窗口直到被撤销。

5. GtkWidget

这是窗口控件的数据结构。它相关的操作函数如下：

```
GtkWidget* gtk_window_new (GtkWindowType type);
```

根据 `type` 来创建一个新的窗口。

```
void gtk_window_set_title (GtkWidget *window, const gchar *title);
```

用 `title` 来设置窗口的标题。

```
void gtk_window_set_default_size (GtkWidget *window, gint width, gint height);
```

用 `width` 和 `height` 来设置窗口的大小。

6. GtkMenu, GtkMenuBar, GtkMenuItem

分别是下拉菜单控件、菜单条和菜单项的数据结构。它们的相关函数如下：

```
GtkWidget* gtk_menu_new (void);
```

创建一个新的菜单。

```
#define gtk_menu_append(menu,child) gtk_menu_shell_append ((GtkMenuShell *) (menu), (child))
```

增加一个菜单项到一个菜单的后面。

```
#define gtk_menu_prepend(menu,child) gtk_menu_shell_prepend ((GtkMenuShell *) (menu), (child))
```

增加一个菜单项到一个菜单的前面。

```
#define gtk_menu_insert(menu,child,pos) gtk_menu_shell_insert ((GtkMenuShell *) (menu), (child), (pos))
```

插入一个菜单项到一个菜单中。

```
void gtk_menu_popup (GtkMenu *menu, GtkWidget *parent_menu_shell, GtkWidget *parent_menu_item, GtkMenuPositionFunc func, gpointer data, guint button, guint32 activate_time);
```

显示一个菜单。

```
void gtk_menu_set_title (GtkMenu *menu,const gchar *title);
```

用 title 来设置菜单内容。

```
G_CONST_RETURN gchar* gtk_menu_get_title (GtkMenu *menu);
```

获取菜单内容。

```
void gtk_menu_popdown (GtkMenu *menu);
```

从屏幕上移出菜单。

```
GtkWidget* gtk_menu_bar_new (void);
```

创建一个新的菜单条。

```
#define gtk_menu_bar_append(menu,child) gtk_menu_shell_append ((GtkMenuShell *) (menu),(child))
```

增加一个新的菜单项到菜单条的末尾。

```
#define gtk_menu_bar_prepend(menu,child) gtk_menu_shell_prepend ((GtkMenuShell *) (menu),(child))
```

增加一个新的菜单项到菜单条的开始。

```
#define gtk_menu_bar_insert(menu,child,pos) gtk_menu_shell_insert ((GtkMenuShell *) (menu),(child),(pos))
```

插入一个新的菜单项到菜单的中间位置。

```
GtkWidget* gtk_menu_item_new (void);
```

创建一个新的菜单项。

```
GtkWidget* gtk_menu_item_new_with_label (const gchar *label);
```

创建一个带有标签的新的菜单项。

```
void gtk_menu_item_set_submenu (GtkMenuItem *menu_item,GtkWidget *submenu);
```

设置一个新的子菜单项。

```
void gtk_menu_item_remove_submenu (GtkMenuItem *menu_item);
```

删除一个子菜单项目。

11.3 使用 GTK

1. 信号驱动

前面说过，GTK 是使用信号驱动来运行的，它创建的控件只有消息产生的时候才调用相应的响应函数。怎样把消息和一个控件相连接，这就是信号绑定的问题。它由下面几个函数来完成。

```
gint gtk_signal_connect( GtkWidget *object,gchar *name,GtkSignalFunc func,gpointer func_data );
```

该函数把一个控件和一个响应函数进行绑定。

```
void gtk_signal_disconnect( GtkWidget *object,gint id );
```

该函数是把一个控件的信号给撤销掉。

那么一个信号是怎样处理的呢？下面是处理一个信号的流程。

首先是信号的发出，信号发出是说 GTK 为一个特定的对象和一个特定的信号执行所有的响应函数的过程，绑定在一个信号上的所有响应函数的执行顺序与它们被设置的顺序是一致

的。再就是信号的传播，从事件发生的那个控件开始，发出一个普通的事件。如果它的信号处理程序返回真，则返回，否则，发出一个特定的“button_press_event”信号。如果返回真，则返回，否则，将控制转移到这个控件的父控件，继续上面的操作，直到某个响应函数返回真或者已经上到最高的控件。

2. 创建菜单过程

创建一个菜单的过程如下：

首先使用函数 `gtk_menu_new()` 来产生一各新的菜单。
再使用函数 `gtk_menu_item_new()` 来产生一个新的菜单项目。
再用函数 `gtk_menu_item_append()` 把所产生的菜单放在一起构成一个菜单列。
再使用函数 `gtk_menu_item_set_submenu()` 把产生的菜单列连接到主菜单上去。
再使用函数 `gtkmenu_bar_new()` 来产生一个菜单条。
最后使用函数 `gtk_menu_bar_append()` 来将主菜单放到菜单条上。

3. 一个使用 GTK+ 的例子

这是一个最简单的例子。创建一个窗口，窗口中有一个按钮，当用鼠标点击按钮时，弹出另外一个窗口。

```
#include <gtk/gtk.h>
//自己的回调函数
void my_button_clicked(GtkWidget* button, gpointer userdata)
{
    GtkWidget *dialog; //对话框指针变量
    dialog = gtk_message_dialog_new(NULL,
                                    GTK_DIALOG_MODAL | GTK_DIALOG_DESTROY_WITH_PARENT,
                                    GTK_MESSAGE_INFO,
                                    GTK_BUTTONS_OK,
                                    (gchar*)userdata);
    //创建一个对话框
    gtk_dialog_run(GTK_DIALOG(dialog));
    //运行对话框
    gtk_widget_destroy(dialog);
    //销毁对话框
}
//主函数
int main(int argc, char* argv[])
{
    GtkWidget *window, *button; //定义两个控件变量指针
    gtk_init(&argc, &argv); //初始化
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    //创建窗口
    g_signal_connect(G_OBJECT(window), "delete_event",
```

```

        G_CALLBACK(gtk_main_quit),NULL);
//捆绑回调函数。
    gtk_window_set_title(GTK_WINDOW(window),"Hello World!");
//为窗口设置标题
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
//设置窗口的宽度
    button=gtk_button_new_with_label("OK");
//创建按钮控件，
    g_signal_connect(G_OBJECT(button),"clicked",
        G_CALLBACK(my_button_clicked),(gpointer)"information");
//为按钮捆绑回调函数 my_button_clicked(),此函数在上面有实现。
    gtk_container_add(GTK_CONTAINER(window),button);
//把按钮控件添加到 window 中。
    gtk_widget_show_all(window);
//显示窗口
gtk_main();
//GTK 主函数。
return FALSE;
}

```

11.4 多线程技术

在本系统中用到了多线程技术，使用多线程可以有很多好处，特别在编写网络程序的时候，为了提高程序的性能，使用多线程是一个非常好的选择。在本系统中有好几个线程，在使用线程的时候，借用了 GTK 中的线程函数，因为其封装了底层的线程操作。

相对进程而言，线程是一个更加接近于执行体的概念，它可以与同进程中的其他线程共享数据，但拥有自己的栈空间，拥有独立的执行序列。在串行程序基础上引入线程和进程是为了提高程序的并发度，从而提高程序运行效率和响应时间。

线程和进程在使用上各有优缺点：线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在 SMP 机器上运行，而进程则可以跨机器迁移。

另外使用多线程还有一个好处。在显示数据信息的时候，我们在捕获网络数据包的时候，只要捕获了一个网络数据包就显示出来，这样就出现了动态显示的效果。可以非常实时的显示。而不是在抓到了很多网络数据包之后再一起全部显示出来，如果这样的话，在捕获数据包的过程中，程序就会死在那里，一动不动，也不能执行其他的操作。

本系统的入口就使用线程操作。具体操作函数如下：

```

void
threads_click (GtkWidget * widget, gpointer data)
{
    //当按“start”按钮时，系统就开始运行，就是调用本函数，所以此函数是系统
    //的入口函数。详细解释请参考 5.4 节
    pthread_t another, another1;

```

```

//线程句柄，多线程技术
if (sniffer_active == 1)
return;
threadstop = 0;
count = 1;
clear_all (NULL, NULL);
read_rules_from_file ("rules");
read_statement_from_rules ();
pthread_create (&another, NULL, another_thread, NULL);
//创建另外一个线程，其回调函数为 another_thread ( )，
//这里就是多线程技术
pthread_create (&another1, NULL, another_thread1, NULL);
//创建另外一个线程，其回调函数为 another_thread1 ( )，
//这里就是多线程技术
}

```

11.4.1 创建线程

从 `threads_click()` 函数中可以看出，创建线程是通过 `pthread_create()` 函数来完成的，其函数原型定义如下：

```

int pthread_create(pthread_t * thread, pthread_attr_t * attr,
void * (*start_routine)(void *), void * arg)

```

参数 `thread` 返回创建的线程 ID，而 `attr` 是创建线程时设置的线程属性。`pthread_create()` 的返回值表示线程创建是否成功。尽管 `arg` 是 `void *` 类型的变量，但它同样可以作为任意类型的参数传给 `start_routine()` 函数；同时，`start_routine()` 可以返回一个 `void *` 类型的返回值，而这个返回值也可以是其他类型，并由 `pthread_join()` 获取。

`pthread_create()` 中的 `attr` 参数是一个结构指针，结构中的元素分别对应着新线程的运行属性，主要包括以下几项：

`__detachstate`，表示新线程是否与进程中其他线程脱离同步，如果置位则新线程不能用 `pthread_join()` 来同步，且在退出时自行释放所占用的资源。默认为 `PTHREAD_CREATE_JOINABLE` 状态。这个属性也可以在线程创建并运行以后用 `pthread_detach()` 来设置，而一旦设置为 `PTHREAD_CREATE_DETACH` 状态（不论是创建时设置还是运行时设置）则不能再恢复到 `PTHREAD_CREATE_JOINABLE` 状态。

`__schedpolicy`，表示新线程的调度策略，主要包括 `SCHED_OTHER`（正常、非实时）、`SCHED_RR`（实时、轮转法）和 `SCHED_FIFO`（实时、先入先出）三种，默认为 `SCHED_OTHER`，后两种调度策略仅对超级用户有效。运行时可以用 `pthread_setschedparam()` 来改变。

`__schedparam`，一个 `struct sched_param` 结构，目前仅有一个 `sched_priority` 整型变量表示线程的运行优先级。这个参数仅当调度策略为实时（即 `SCHED_RR` 或 `SCHED_FIFO`）时才有效，并可以在运行时通过 `pthread_setschedparam()` 函数来改变，默认为 0。

`__inheritsched`，有两种值可供选择：`PTHREAD_EXPLICIT_SCHED` 和 `PTHREAD_INHERIT_SCHED`，前者表示新线程使用显式指定调度策略和调度参数（即 `attr` 中的值），

而后者表示继承调用者线程的值。默认为 PTHREAD_EXPLICIT_SCHED。

__scope, 表示线程间竞争 CPU 的范围, 也就是说线程优先级的有效范围。POSIX 的标准中定义了两个值: PTHREAD_SCOPE_SYSTEM 和 PTHREAD_SCOPE_PROCESS, 前者表示与系统中所有线程一起竞争 CPU 时间, 后者表示仅与同进程中的线程竞争 CPU。目前 LinuxThreads 仅实现了 PTHREAD_SCOPE_SYSTEM 一值。

pthread_attr_t 结构中还有一些值, 但不使用 pthread_create() 来设置。

11.4.2 结束线程

线程终止有两种情况: 正常终止和非正常终止。线程主动调用 pthread_exit() 或者从线程函数中 return 都将使线程正常退出, 这是可预见的退出方式; 非正常终止是线程在其他线程的干预下, 或者由于自身运行出错 (比如访问非法地址) 而退出, 这种退出方式是不可预见的。

线程终止函数原型如下:

```
void pthread_exit(void *retval)
```

参数 retval 是 pthread_exit() 调用者线程 (线程 ID 为 th) 的返回值。

一般情况下, 线程在其主体函数退出的时候会自动终止, 但同时也可以因为接收到另一个线程发来的终止 (取消) 请求而强制终止。

线程取消的方法是向目标线程发 Cancel 信号, 但如何处理 Cancel 信号则由目标线程自己决定, 或者忽略、或者立即终止、或者继续运行至 Cancellation-point (取消点), 由不同的 Cancellation 状态决定。

线程接收到 CANCEL 信号的默认处理 (即 pthread_create() 创建线程的默认状态) 是继续运行至取消点, 也就是说设置一个 CANCELED 状态, 线程继续运行, 只有运行至 Cancellation-point 的时候才会退出。

pthread_join()、pthread_testcancel()、pthread_cond_wait()、pthread_cond_timedwait()、sem_wait()、sigwait() 等函数以及 read()、write() 等会引起阻塞的系统调用都是 Cancellation-point, 而其他 pthread 函数都不会引起 Cancellation 动作。如果线程处于无限循环中, 且循环体内没有执行至取消点的必然路径, 则线程无法由外部其他线程的取消请求而终止。因此在这样的循环体的必经路径上应该加入 pthread_testcancel() 调用。

与结束线程相关的 pthread 函数还有如下:

```
int pthread_cancel(pthread_t thread)
```

发送终止信号给 thread 线程, 如果成功则返回 0, 否则为非 0 值。发送成功并不意味着 thread 会终止。

```
int pthread_setcancelstate(int state, int *oldstate)
```

设置本线程对 Cancel 信号的反应, state 有两种值: PTHREAD_CANCEL_ENABLE (默认) 和 PTHREAD_CANCEL_DISABLE, 分别表示收到信号后设为 CANCELED 状态和忽略 CANCEL 信号继续运行; old_state 如果不为 NULL 则存入原来的 Cancel 状态以便恢复。

```
int pthread_setcanceltype(int type, int *oldtype)
```

设置本线程取消动作的执行时机, type 有两种取值: PTHREAD_CANCEL_DEFERRED

和 PTHREAD_CANCEL_ASYNCHRONOUS，仅当 Cancel 状态为 Enable 时有效，分别表示收到信号后继续运行至下一个取消点再退出和立即执行取消动作（退出）；oldtype 如果不为 NULL 则存入运来的取消动作类型值。

```
void pthread_testcancel(void)
```

检查本线程是否处于 Canceld 状态，如果是，则进行取消动作，否则直接返回。

11.4.3 线程同步

在使用线程的时候就要考虑线程的同步问题，可以使用很多方法来保持线程的同步。一般使用互斥锁、条件变量和信号灯技术。

（1）互斥锁

有两种方式创建互斥锁：静态方式和动态方式。静态方式就是使用 pthread_mutex_t 结构体，其中定义了一个常量 PTHREAD_MUTEX_INITIALIZER；动态方式是使用 pthread_mutex_init() 函数来初始化互斥锁。其函数原型如下：

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)
```

参数 mutexattr 用于指定互斥锁属性，如果为 NULL 就使用默认属性。

撤销一个互斥锁使用函数 pthread_mutex_destroy() 函数。其函数原型如下：

```
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```

对互斥锁的操作有下面一些相关函数：

```
int pthread_mutex_lock(pthread_mutex_t *mutex) //加锁
```

```
int pthread_mutex_unlock(pthread_mutex_t *mutex) //解锁
```

```
int pthread_mutex_trylock(pthread_mutex_t *mutex) //测试加锁
```

（2）条件变量

条件变量是利用线程间共享的全局变量进行同步的一种机制，主要包括两个动作：一个线程等待“条件变量的条件成立”而挂起；另一个线程使“条件成立”。为了防止竞争，条件变量的使用总是和一个互斥锁结合在一起。也有两种方式，一种使用静态方式，使用 PTHREAD_COND_INITIALIZER 常量。另一种使用动态方式，使用函数 pthread_cond_init()，其函数原型如下：

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr)
```

撤销一个条件变量使用函数 pthread_cond_destroy()。其原型定义如下：

```
int pthread_cond_destroy(pthread_cond_t *cond)
```

相关的操作函数定义如下：

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) //无条件等待
```

```
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime) //计时等待
```

（3）信号灯

信号灯有两种状态，一个是灯灭，一个是灯亮，每个状态就表示了系统资源的共享与否。也可以使用状态大于 2 的信号灯，此时表示资源数大于 1。创建一个信号灯使用函数 sem_init()，其函数原型如下：

```
int sem_init(sem_t *sem, int pshared, unsigned int value)
```

其中，参数 value 是信号灯的初值，pshared 表示是否为多进程共享。撤销信号灯使用函数 sem_destroy ()，其函数原型如下：

```
int sem_destroy(sem_t * sem)
```

与信号灯相关的操作函数说明如下：

```
int sem_post(sem_t * sem) //点灯
```

```
int sem_wait(sem_t * sem) //阻塞等灯亮
```

```
int sem_trywait(sem_t * sem) //非阻塞等灯亮
```

```
int sem_getvalue(sem_t * sem, int * sval) //获取灯值
```

关于多线程技术的介绍读者可以进一步参考其他相关资料。

11.4.4 GTKV+多线程

本系统中大量使用了多线程技术。在界面设计中，也使用到了线程技术，以达到动态的显示数据的效果，下面介绍一下它的实现过程。

本系统界面是使用 GTK+库的。在 GTK+中也可以使用线程技术，GTK+封装了一些线程操作函数。由于 GTK+是基于 GLIB 的。在 GLIB 中就使用了线程技术。它里面有很多的线程操作函数。下面对其中比较常见函数进行了一些介绍。在 GTK 中有两个重要的线程操作函数，分别是 gdk_thread_enter()和 gdk_thread_leave()，这两个函数是成对出现的，当要对代码进行线程操作时，就可以在这两个函数之间添加代码了。第一个函数是进入线程操作，第二个函数是离开线程操作。封装成这两个函数的目的是增强线程的安全性。

首先要调用初始化线程函数，其原型定义如下：

```
void g_thread_init (GThreadFunctions *vtable);
```

在调用其他线程函数的时候，就必须使用此函数进行初始化。一般情况下参数 vtable 可以赋值为 NULL。注意此函数不能直接或间接的作为回调函数来调用。而且此函数只可能被调用一次。如果第二次调用就会错误终止。如果不能确定线程是否已经初始化，可以使用如下语句：

```
if (!g_thread_supported ()) g_thread_init (NULL);
```

在使用线程的时候 程序编译的时候需要使用参数 pkg-config --libs gthread-2.0 进行编译。

在 GTK 中创建一个线程使用函数 g_thread_create ()，其原型如下：

```
GThread* g_thread_create (GThreadFunc func,gpointer data, gboolean joinable, GError **error);
```

参数 joinable 的值为 TRUE 时，可以使用函数 g_thread_join()来等待线程结束。参数 func 就是线程的回调函数。此回调函数的参数就是参数 data。如果忽略错误，参数 error 就可以赋值为 NULL。非 NULL 就可以返回出错原因。此函数的返回值是一个 Gthread 结构体，它表示当前运行的进程。

下面看线程回调函数的定义形式：

```
gpointer (*GthreadFunc) (gpointer data)
```

其中 data 就是 g_thread_create()中传递的参数。

线程的优先级由枚举类型 GthreadPriority 决定，其定义如下：

```
typedef enum
{
    G_THREAD_PRIORITY_LOW,    //优先级低于正常值
    G_THREAD_PRIORITY_NORMAL, //默认的优先级
    G_THREAD_PRIORITY_HIGH,   //优先级高于正常值
    G_THREAD_PRIORITY_URGENT  //最高优先级
}GThreadPriority;
```

设置优先级的函数是 `g_thread_set_priority ()`，其原型如下：

```
void g_thread_set_priority (Gthread *thread , GthreadPriority priority)
```

终止线程使用函数 `g_thread_exit ()`，其原型如下：

```
void g_thread_exit (gpointer retval);
```

此函数的功能就是退出当前线程。

在 GTK 中线程的同步也可以用互斥锁和条件值。在 GTK 中与互斥锁相关的函数介绍如下：

```
Gmutex * g_mutex_new();
```

创建一个新的互斥体 `GMutex`，`GMutex` 结构体是一个不透明的结构体用来表示一个互斥体。可以用来保护私有数据不被共享访问。此函数当没有调用 `g_thread_init()`时将终止。

```
void g_mutex_lock (GMutex *mutex);
```

锁定互斥体。如果互斥体已经被另外一个线程锁定，那么当前线程将被阻塞直到此互斥体被其他线程解锁。

```
Gboolean g_mutex_trylock (GMutex *mutex);
```

非阻塞形式的锁定互斥体。如果互斥体被另外一个线程锁定，那么它直接返回一个 `FALSE`。否则就锁定一个互斥体，并且返回 `TRUE`。

```
Void g_mutex_unlock (GMutex *mutex);
```

解锁互斥体。如果另外一个线程在调用函数 `g_mutex_lock()`时被阻塞了，那么这个时候线程将被唤醒，并且能够锁定此互斥体。

```
Void g_mutex_free (GMutex *mutex);
```

撤销一个互斥体。

在 GTK 中使用的第二种同步技术就是使用条件值。其数据结构体如下：

```
struct Gcond ;
```

`Gcode` 结构体是一个不透明的结构体用来描述一个条件值。一个 `Gcode` 是一个对象，当线程发现一个条件值为假的时候可以在此条件上阻塞。如果另外的线程改变此条件的状态，他们可以发送信号给 `Gcnode`，例如等待线程被唤醒事件。与条件值相关的函数介绍如下：

```
Gcond * g_cond_new ();
```

创建一个新的 `Gcond`，此函数如果在函数 `g_thread_init()`之前调用会被终止。

```
void g_cond_signal( Gcond *cond);
```

如果线程正在等待条件值，其中一个被唤醒。

```
Void g_cond_wait(Gcond *cond, GMutex *mutex);
```

等待一直到线程在一个条件值被唤醒。

```
Gboolean g_cond_timed_wait (Gcond *cond , GMutex *mutex , GtimeVal *abs_time);
```

计时等待,等待线程唤醒,但当到达一定时间的时候就不再等待。时间值由参数 abs_time 来决定。

```
Void g_cond_free (Gcond *cond);
```

撤销一个条件值。

11.5 实现本系统界面模块

11.5.1 本系统界面分布情况

本系统的界面用 GTK+来设计,基本上都具有一般窗口的大部分内容。它有菜单、工具条、主窗口,等等,如图 11-2 所示。

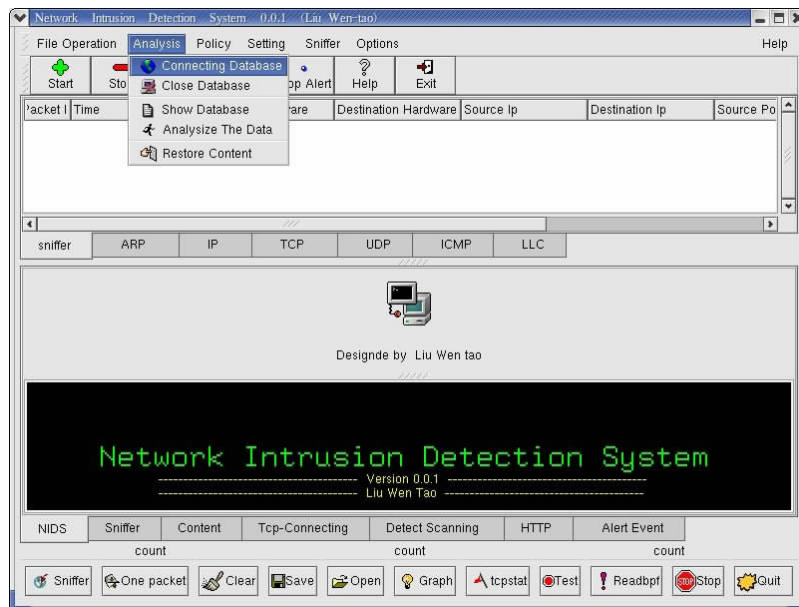


图 11-2 界面模块

其中主窗口包括两部分,上部分是显示协议分析内容的。其中 sniffer 页面显示所有捕获数据包的基本信息,包括数据包个数、时间源硬件地址、目的硬件地址、源 IP 地址、目的 IP 地址、源端口和目的端口。ARP 页面显示分析 ARP 协议的结果,其中包括所有的 ARP 字段信息。IP 页面显示分析 IP 协议的结果,其中包括所有的 IP 字段信息。TCP 页面显示分析 TCP 协议的结果,其中包括所有的 TCP 字段信息。UDP 页面显示分析 UDP 协议的结果,其中包括所有的 UDP 字段信息。ICMP 页面显示分析 ICMP 协议的结果,其中包括所有的 ICMP 字段信息。LLC 页面显示分析 LLC 协议的结果。当有相应协议的数据包捕获的时候,就把相应的协议显示到相应的页面之中。

下部分也是显示信息的，其中 sniffer 页面也是显示捕获数据包的基本信息。Content 页面显示数据包内容的，以十六进制和字符形式。Tcp-Connecting 页面显示 TCP 连接过程。Detect Scanning 是显示捕获扫描行为的信息。HTTP 页面是显示分析 HTTP 协议的内容。Alert Event 是显示发现入侵事件的信息。

各个页面的显示结果可以参考相关章节。

11.5.2 界面模块实现

本系统使用 GTK 来实现界面，其代码很多，在这里仅举几个实现过程。

1. 添加规则的窗口

这个窗口最典型也比较容易理解，因为它规模比较小，所以先介绍它的实现过程。此窗口的功能就是添加入侵规则事件，一次添加一个事件。此事件的定义如前所述，有几个部分，分别是事件名字（Event Name）、事件协议（Event Protocol）、事件代码（Event Code）、事件定义（Event Definition）、事件信息（Event Information）和响应类型（Alert Type）。

```
void
show_add_a_rule_window(void)
{
    GtkWidget *hbox;           //水平 box 控件
    GtkWidget *entry1;         //编辑控件 1，表示 Event Name：
    GtkWidget *entry2;         //编辑控件 2，表示 Event Protocol
    GtkWidget *entry3;         //编辑控件 3，表示 Event Code
    GtkWidget *entry4;         //编辑控件 4，表示 Event Definition
    GtkWidget *entry5;         //编辑控件 5，表示 Event Information
    GtkWidget *entry6;         //编辑控件 6，表示 Alert Type
    GtkWidget *button;         //按钮控件
    GtkWidget *label;          //标签控件
    GtkWidget *table;          //表格控件
    add_a_rule_window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    //创建新的窗口
    gtk_window_set_title(GTK_WINDOW(add_a_rule_window),
        "Add a Rule To Clist");
    //设置窗口的标题
    table = gtk_table_new(7, 2, FALSE);
    //创建一个表格控件
    gtk_table_set_row_spacings(GTK_TABLE(table), 2);
    //设置此表格有 2 行
    gtk_table_set_col_spacings(GTK_TABLE(table), 2);
    //设置此表格有 2 列
    gtk_container_add(GTK_CONTAINER(add_a_rule_window), table);
    //把此表格控件添加到窗口控件
    gtk_widget_show(table);
}
```

```
//显示表格控件
label = gtk_label_new ("Event Name :");
//创建一个标签控件
gtk_table_attach (GTK_TABLE (table), label, 0, 1, 0, 1, GTK_FILL, GTK_FILL,
0, 0);
//把标签控件添加到表格中相应位置，由参数决定
gtk_widget_show (label);
//显示标签
entry1 = gtk_entry_new_with_max_length (50);
//创建一个编辑控件
gtk_object_set_data (GTK_OBJECT (add_a_rule_window), "entry1", entry1);
//设置一个数据
gtk_entry_set_text (GTK_ENTRY (entry1), "");
//设置空字符串
gtk_entry_select_region (GTK_ENTRY (entry1), 0,
GTK_ENTRY (entry1)->text_length);
//设置编辑控件属性
gtk_table_attach (GTK_TABLE (table), entry1, 1, 2, 0, 1, GTK_FILL, GTK_FILL,
0, 0);
//把此控件添加到表格中
gtk_widget_show (entry1);
//显示编辑控件
label = gtk_label_new ("Event Protocol :");
//创建标签
gtk_table_attach (GTK_TABLE (table), label, 0, 1, 1, 2, GTK_FILL, GTK_FILL,
0, 0);
//把此标签添加到表格中
gtk_widget_show (label);
//显示此标签
entry2 = gtk_entry_new_with_max_length (50);
//创建编辑控件
gtk_object_set_data (GTK_OBJECT (add_a_rule_window), "entry2", entry2);
//设置一个数据
gtk_entry_set_text (GTK_ENTRY (entry2), "");
//设置空字符串
gtk_entry_select_region (GTK_ENTRY (entry2), 0,
GTK_ENTRY (entry2)->text_length);
//设置编辑控件 2 属性
gtk_table_attach (GTK_TABLE (table), entry2, 1, 2, 1, 2, GTK_FILL, GTK_FILL,
0, 0);
//将编辑控件 2 添加到表格中
gtk_widget_show (entry2);
```

```
//显示编辑控件
label = gtk_label_new ("Event Code :");
//创建标签
gtk_table_attach (GTK_TABLE (table), label, 0, 1, 2, 3, GTK_FILL, GTK_FILL,
0, 0);
//把标签控件添加到表格中相应位置
gtk_widget_show (label);
entry3 = gtk_entry_new_with_max_length (50);
//创建一个编辑控件 3
gtk_object_set_data (GTK_OBJECT (add_a_rule_window), "entry3", entry3);
//设置一个数据
gtk_entry_set_text (GTK_ENTRY (entry3), "");
//设置空字符串
gtk_entry_select_region (GTK_ENTRY (entry3), 0,
GTK_ENTRY (entry3)->text_length);
//设置编辑控件 3 属性
gtk_table_attach (GTK_TABLE (table), entry3, 1, 2, 2, 3, GTK_FILL, GTK_FILL,
0, 0);
//把此控件添加到表格中
gtk_widget_show (entry3);
//显示编辑控件 3
label = gtk_label_new ("Event Defination :");
//创建标签控件
gtk_table_attach (GTK_TABLE (table), label, 0, 1, 3, 4, GTK_FILL, GTK_FILL,
0, 0);
//把标签控件添加到表格中相应位置
gtk_widget_show (label);
//显示标签控件
entry4 = gtk_entry_new_with_max_length (50);
//创建一个编辑控件
gtk_object_set_data (GTK_OBJECT (add_a_rule_window), "entry4", entry4);
//设置一个数据
gtk_entry_set_text (GTK_ENTRY (entry4), "");
//设置空字符串
gtk_entry_select_region (GTK_ENTRY (entry4), 0,
GTK_ENTRY (entry4)->text_length);
//设置编辑控件属性
gtk_table_attach (GTK_TABLE (table), entry4, 1, 2, 3, 4, GTK_FILL, GTK_FILL,
0, 0);
//把此控件添加到表格中
gtk_widget_show (entry4);
//显示编辑控件
```



```
label = gtk_label_new ("Event Information :");
//创建标签
gtk_table_attach (GTK_TABLE (table), label, 0, 1, 4, 5, GTK_FILL, GTK_FILL,
0, 0);
//把标签控件添加到表格中相应位置
gtk_widget_show (label);
//显示标签控件
entry5 = gtk_entry_new_with_max_length (50);
//创建一个编辑控件
gtk_object_set_data (GTK_OBJECT (add_a_rule_window), "entry5", entry5);
//设置一个数据
gtk_entry_set_text (GTK_ENTRY (entry5), "");
//设置空字符串
gtk_entry_select_region (GTK_ENTRY (entry5), 0,
GTK_ENTRY (entry5)->text_length);
//设置编辑控件属性
gtk_table_attach (GTK_TABLE (table), entry5, 1, 2, 4, 5, GTK_FILL, GTK_FILL,
0, 0);
//把此控件添加到表格中
gtk_widget_show (entry5);
//显示编辑控件
label = gtk_label_new ("Alert Type :");
//创建标签
gtk_table_attach (GTK_TABLE (table), label, 0, 1, 5, 6, GTK_FILL, GTK_FILL,
0, 0);
//把标签控件添加到表格中相应位置
gtk_widget_show (label);
//显示标签控件
entry6 = gtk_entry_new_with_max_length (50);
//创建一个编辑控件
gtk_object_set_data (GTK_OBJECT (add_a_rule_window), "entry6", entry6);
//设置一个数据
gtk_entry_set_text (GTK_ENTRY (entry6), "");
//设置空字符串
gtk_entry_select_region (GTK_ENTRY (entry6), 0,
GTK_ENTRY (entry6)->text_length);
//设置编辑控件属性
gtk_table_attach (GTK_TABLE (table), entry6, 1, 2, 5, 6, GTK_FILL, GTK_FILL,
0, 0);
//把此控件添加到表格中
gtk_widget_show (entry6);
//显示编辑控件
```

```

hbox = gtk_hbox_new (FALSE, 0);
//创建一个水平 box
gtk_table_attach (GTK_TABLE (table), hbox, 0, 2, 6, 7,
                  GTK_FILL | GTK_EXPAND, GTK_FILL | GTK_EXPAND, 0, 0);
//把 box 控件添加到表格控件中
gtk_widget_show (hbox);
//显示此水平 box
button = gtk_button_new_with_label ("Ok");
//创建一个 ( ok ) 按钮控件
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (add_a_rule_window_ok_clicked), NULL);
//为此按钮控件捆绑一个回调函数 add_a_rule_window_ok_clicked ( ), 当有"clicked"
//事件发生时, 调用此函数
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (gtk_widget_destroy),
                           GTK_OBJECT (add_a_rule_window));
//为 ok 按钮捆绑另外一个回到函数 gtk_widget_destroy ( )
//此函数的功能时销毁此窗口
gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 10);
//把此按钮添加入水平 box 控件中
gtk_widget_show (button);
//显示此按钮控件
button = gtk_button_new_with_label ("Cancle");
//创建一个"Cancle"按钮控件
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
                           GTK_SIGNAL_FUNC (gtk_widget_destroy),
                           GTK_OBJECT (add_a_rule_window));
//为 Cancle 按钮控件捆绑一个回调函数 gtk_widget_destroy()
gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 10);
//把此按钮添加入水平 box 控件中
gtk_widget_show (button);
//显示此按钮控件
gtk_widget_show (add_a_rule_window);
}

```

2. 主界面

下面的函数是整个系统的主函数。

```

gint
main (gint argc, gchar * argv[])
{
    //定义一些控件变量。
    GtkWidget *mnu_1;

```

```
GtkWidget *mnu_2;
GtkWidget *mnu_3;
GtkWidget *text;
GtkWidget *vbox_text;
GtkWidget *text3;
GtkWidget *text4;
GtkWidget *text5;
GtkWidget *vpaned;
GtkWidget *vpaned1;
GtkWidget *menu;
GtkWidget *menu_bar;
GtkWidget *root_menu;
GtkWidget *separator1;
GtkWidget *menu_items;
GtkWidget *handlebox1;
GtkWidget *handlebox2;
GtkWidget *toolbar2;
GtkWidget *tmp_toolbar_icon;
GtkAccelGroup *accel_group;
GtkWidget *pixmapwid;
GdkPixmap *pixmap;
GdkBitmap *mask;
GtkStyle *style;
GtkWidget *button;
GtkWidget *box_xpm;
GtkWidget *vbox, *listvbox;
GtkWidget *hbox;
GtkWidget *buttons_hbox;
GtkWidget *vscrollbar;
GtkTooltips *tooltips;
GtkWidget *showcount_label;
GtkWidget *label;
GtkWidget *baojinlabel;
GtkWidget *separator;
GdkPixmap *icon_pixmap;
GdkBitmap *icon_mask;
GtkWidget *notebook;
GtkWidget *text_notebook;
GtkWidget *text_scrolledwindow;
GtkWidget *frame;
GtkWidget *scrolled_window;
GtkWidget *clist;
```

```

GtkWidget *clist1;
GtkWidget *clist2;
GtkWidget *clist3;
GtkWidget *clist4;
GtkWidget *clist5;
GtkWidget *clist6;
GtkWidget *clist_event;
GtkWidget *button_add, *button_clear, *button_hide_show;
gchar *titles_arp[11] =
{ "number", "Hardware type", "Protocol", "hln", "plen", "Operation",
  "Source Hardware", "Source Ip", "Destination Hardware", "Destination Ip", "Information" };
gchar *titles_ip[12] =
{ "number", "version", "header_length", "tos", "total_length", "id",
  "off", "ttl", "protocol", "checksum", "source_ip", "destination_ip" };
gchar *titles_tcp[12] =
{ "number", "sport", "dport", "seq", "ack", "doff", "flags", "win",
  "cksm", "urp", "tcp_options", "information" };
gchar *titles_udp[5] = { "number", "sport", "dport", "len", "cksum" };
g_thread_init (NULL);
gtk_init (&argc, &argv);
//gtk 初始化，此函数是必需的。
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_set_usize (GTK_WIDGET (window), 790, 580);
gtk_window_set_title (GTK_WINDOW (window),
    "Network Intrusion Detection System 0.0.1 (Liu Wen-tao)");
gtk_widget_realize (window);
icon_pixmap = gdk_pixmap_create_from_xpm_d (window->window, &icon_mask,
    &window->style->
    bg[GTK_STATE_NORMAL],
    large_kvt_xpm);
gdk_window_set_icon (window->window, NULL, icon_pixmap, icon_mask);
gdk_window_set_icon_name (window->window, "NIDS");
vbox = gtk_vbox_new (FALSE, 0);
//===== 创建菜单条控件 =====//
menu_bar = gtk_menu_bar_new ();
//=====创建 File Operation 菜单 =====//
menu = gtk_menu_new ();
root_menu = gtk_menu_item_new_with_label ("File Operation");
gtk_widget_show (root_menu);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "./xpm/open.xpm", "Open (.bpf)");
gtk_widget_show (box_xpm);

```

```
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
    GTK_SIGNAL_FUNC (set_readfile_callback), NULL);
//设置回调函数 set_readfile_callback ( ), 其功能是设置读取的默认文件
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/filesave.xpm", "Save(.bpf)");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
    GTK_SIGNAL_FUNC (set_savefile_callback), NULL);
//设置回调函数 set_savefile_callback ( ), 其功能是设置打开的默认文件
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
separator1 = gtk_menu_item_new ();
gtk_widget_show (separator1);
gtk_menu_append (GTK_MENU (menu), separator1);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/open.xpm", "Open (.log)");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
    GTK_SIGNAL_FUNC (openfile_test), NULL);
//设置回调函数 openfile_test ( ), 其功能是打开文件
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/filesave.xpm", "Save (.log)");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
    GTK_SIGNAL_FUNC (savetofile), NULL);
//设置回调函数 savetofile ( ), 其功能是把信息存储到文件
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
separator1 = gtk_menu_item_new ();
gtk_widget_show (separator1);
gtk_menu_append (GTK_MENU (menu), separator1);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/open.xpm", "Open Event Log");
gtk_widget_show (box_xpm);
```

```
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (open_log_file), NULL);
//设置回调函数 open_log_file ( ), 其功能是打开日志文件
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
separator1 = gtk_menu_item_new ();
gtk_widget_show (separator1);
gtk_menu_append (GTK_MENU (menu), separator1);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/quits.xpm", "Quit");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (destroy), NULL);
//设置回调函数 destroy ( ), 其功能是销毁窗口, 退出程序
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
gtk_menu_item_set_submenu (GTK_MENU_ITEM (root_menu), menu);
gtk_menu_bar_append (GTK_MENU_BAR (menu_bar), root_menu);
//=====创建 analysis 菜单=====//
menu = gtk_menu_new ();
root_menu = gtk_menu_item_new_with_label ("Analysis");
gtk_widget_show (root_menu);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/url.xpm", "Connecting Database");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_object_set_data (GTK_OBJECT (window), "menu_connect_database",
                    menu_items);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (connect_database), NULL);
//设置回调函数 connect_database ( ), 其功能是打开数据库连接
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/cross.xpm", "Close Database");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_object_set_data (GTK_OBJECT (window), "menu_close_database",
                    menu_items);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
```

```
GTK_SIGNAL_FUNC (close_database), NULL);
//设置回调函数 close_database ( ), 其功能是关闭数据库连接
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
separator1 = gtk_menu_item_new ();
gtk_widget_show (separator1);
gtk_menu_append (GTK_MENU (menu), separator1);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "./xpm/file.xpm", "Show Database");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_object_set_data (GTK_OBJECT (window), "menu_show_database", menu_items);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
GTK_SIGNAL_FUNC (show_database), NULL);
//设置回调函数 show_database ( ), 其功能是显示数据库信息窗口
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_menu_item_new ();
box_xpm =
xpm_label_box (window, "./xpm/mini-run.xpm", "Analyze The Data");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_object_set_data (GTK_OBJECT (window), "menu_show_database", menu_items);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
GTK_SIGNAL_FUNC (show_analyze_window), NULL);
//设置回调函数 show_analyze_window ( ), 其功能是显示数据库分析窗口
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
separator1 = gtk_menu_item_new ();
gtk_widget_show (separator1);
gtk_menu_append (GTK_MENU (menu), separator1);
menu_items = gtk_menu_item_new ();
box_xpm =
xpm_label_box (window, "./xpm/restore_content.xpm", "Restore Content");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_object_set_data (GTK_OBJECT (window), "menu_show_database", menu_items);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
GTK_SIGNAL_FUNC (show_restore_content_window), NULL);
//设置回调函数 show_restore_content_window ( ),
//功能是显示 HTTP 数据库信息
gtk_menu_append (GTK_MENU (menu), menu_items);
```

```

gtk_widget_show (menu_items);
gtk_menu_item_set_submenu (GTK_MENU_ITEM (root_menu), menu);
gtk_menu_bar_append (GTK_MENU_BAR (menu_bar), root_menu);
//===== 创建 policy 菜单 =====/
menu = gtk_menu_new ();
root_menu = gtk_menu_item_new_with_label ("Policy");
gtk_widget_show (root_menu);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "./xpm/mini-zoom.xpm", "Rules");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (show_rules_window), NULL);
//设置回调函数 show_rules_window ( ), 功能是显示规则显示窗口
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "./xpm/mini-colors.xpm", "Defile Rules");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (show_define_rules_window), NULL);
//设置回调函数 show_define_rules_window ( ), 其功能是显示定义规则窗口 ,
//在这个窗口中就可以动态的定义规则了 ,
//然后把规则加入到规则库中。
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
gtk_menu_item_set_submenu (GTK_MENU_ITEM (root_menu), menu);
gtk_menu_bar_append (GTK_MENU_BAR (menu_bar), root_menu);
//===== 创建 setting 菜单 =====/
menu = gtk_menu_new ();
root_menu = gtk_menu_item_new_with_label ("Setting ");
gtk_widget_show (root_menu);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "./xpm/bpf.xpm", "Setting Sniffer ");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (setting_sniffer_clicked), NULL);
//设置回调函数 setting_sniffer_clicked ( ),
//其功能是设置捕获参数, 如捕获个数等
gtk_menu_append (GTK_MENU (menu), menu_items);

```



```
gtk_widget_show (menu_items);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/mini-display.xpm", "Setting Bpf");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (setting_bpf), NULL);
//设置回调函数 setting_bpf ( ), 其功能是设置 pbk 文件
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/delete.xpm", "Clear All");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (clear_all), NULL);
//设置回调函数 clear_all ( ), 其功能是清除所有页面信息
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
gtk_menu_item_set_submenu (GTK_MENU_ITEM (root_menu), menu);
gtk_menu_bar_append (GTK_MENU_BAR (menu_bar), root_menu);
//===== 创建 sniffer 菜单 =====/
menu = gtk_menu_new ();
root_menu = gtk_menu_item_new_with_label ("Sniffer");
gtk_widget_show (root_menu);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/getdevice.xpm", "Get device");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (get_device_callback), NULL);
//设置回调函数 get_device_callback ( ), 其功能是获取可用的网络设备
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/setfilter.xpm", "Set filter");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (set_filter_callback), NULL);
//设置回调函数 set_filter_callback ( ), 其功能是设置过滤规则
gtk_menu_append (GTK_MENU (menu), menu_items);
```

```
gtk_widget_show (menu_items);
separator1 = gtk_menu_item_new ();
gtk_widget_show (separator1);
gtk_menu_append (GTK_MENU (menu), separator1);
menu_items = gtk_check_menu_item_new ();
box_xpm = xpm_label_box (window, "./xpm/setsavefile.xpm", "Set Savefile");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (set_savefile_callback), NULL);
//设置回调函数 set_savefile_callback ( ), 其功能是设置保存的默认文件
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "./xpm/setreadfile.xpm", "Set Readfile");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (set_readfile_callback), NULL);
//设置回调函数 set_readfile_callback ( ), 其功能是设置读取的默认文件
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
separator1 = gtk_menu_item_new ();
gtk_widget_show (separator1);
gtk_menu_append (GTK_MENU (menu), separator1);
menu_items = gtk_check_menu_item_new ();
box_xpm =
xpm_label_box (window, "./xpm/showtotal.xpm",
                "Show total packet content");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (show_total_packet_content), NULL);
//设置回调函数 show_total_packet_content ( ),
//其功能是显示数据包内容 ( 十六进制和字符形式 )
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_check_menu_item_new ();
box_xpm =
xpm_label_box (window, "./xpm/showtcp.xpm", "Show Tcp Connecting");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
```

```
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (show_tcp_connecting), NULL);
//设置回调函数 show_tcp_connecting ( ),
//其功能是分析 TCP 协议连接过程, 请参考 6.3.2 节
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_check_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/scanning.xpm", "Detect Scanning");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (detect_scanning), NULL);
//设置回调函数 detect_scanning ( ), 其功能是检测扫描行为, 请参考 9.6.4 节
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
gtk_menu_item_set_submenu (GTK_MENU_ITEM (root_menu), menu);
gtk_menu_bar_append (GTK_MENU_BAR (menu_bar), root_menu);
menu = gtk_menu_new ();
root_menu = gtk_menu_item_new_with_label ("Options");
gtk_widget_show (root_menu);
menu_items = gtk_check_menu_item_new ();
box_xpm =
xpm_label_box (window, "/xpm/hidetoolbar.xpm", "Hide the toolbar");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (hidetoolbar_callback), NULL);
//设置回调函数 hidetoolbar_callback(), 其功能是隐藏工具栏
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_check_menu_item_new ();
box_xpm =
xpm_label_box (window, "/xpm/hidebutton.xpm", "Hide the Buttons");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (hidebuttons_callback), NULL);
//设置回调函数 hidebuttons_callback ( ), 其功能是隐藏按钮行
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
separator1 = gtk_menu_item_new ();
gtk_widget_show (separator1);
```

```

gtk_menu_append (GTK_MENU (menu), separator1);
gtk_menu_item_set_submenu (GTK_MENU_ITEM (root_menu), menu);
gtk_menu_bar_append (GTK_MENU_BAR (menu_bar), root_menu);
//===== 创建 help 菜单 =====/
menu = gtk_menu_new ();
root_menu = gtk_menu_item_new_with_label ("Help");
gtk_widget_show (root_menu);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "./xpm/help.xpm", "Help");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (on_help_about_menubar_activate), NULL);
//设置回调函数 on_help_about_menubar_activate ( ), 显示帮助对话框
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
menu_items = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "./xpm/about.xpm", "About....");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (menu_items), box_xpm);
gtk_signal_connect (GTK_OBJECT (menu_items), "activate",
                    GTK_SIGNAL_FUNC (about), NULL);
//设置回调函数 about ( ), 显示关于对话框
gtk_menu_append (GTK_MENU (menu), menu_items);
gtk_widget_show (menu_items);
gtk_menu_item_set_submenu (GTK_MENU_ITEM (root_menu), menu);
gtk_menu_bar_append (GTK_MENU_BAR (menu_bar), root_menu);
gtk_menu_item_right_justify (GTK_MENU_ITEM (root_menu));
handlebox1 = gtk_handle_box_new ();
gtk_object_set_data (GTK_OBJECT (window), "handlebox1", handlebox1);
gtk_widget_show (handlebox1);
gtk_box_pack_start (GTK_BOX (vbox), handlebox1, FALSE, TRUE, 0);
gtk_container_add (GTK_CONTAINER (handlebox1), menu_bar);
gtk_widget_show (menu_bar);
//===== 创建工具栏控件 toolbar =====//
accel_group = gtk_accel_group_new ();
gtk_window_add_accel_group (GTK_WINDOW (window), accel_group);
handlebox2 = gtk_handle_box_new ();
gtk_object_set_data (GTK_OBJECT (window), "handlebox2", handlebox2);
gtk_widget_show (handlebox2);
gtk_box_pack_start (GTK_BOX (vbox), handlebox2, FALSE, TRUE, 0);
toolbar2 = gtk_toolbar_new (GTK_ORIENTATION_HORIZONTAL, GTK_TOOLBAR_BOTH);

```

```

gtk_object_set_data (GTK_OBJECT (window), "toolbar2", toolbar2);
gtk_widget_show (toolbar2);
gtk_container_add (GTK_CONTAINER (handlebox2), toolbar2);
tmp_toolbar_icon = create_pixmap (window, "allow.xpm");
if (tmp_toolbar_icon == NULL)
g_error ("Couldn't create pixmap");
button = gtk_toolbar_append_element (GTK_TOOLBAR (toolbar2),
GTK_TOOLBAR_CHILD_BUTTON,
NULL,
"Start",
"Allow any host", NULL,
tmp_toolbar_icon, NULL, NULL);
gtk_object_set_data (GTK_OBJECT (window), "btn_allowall", button);
gtk_widget_show (button);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
GTK_SIGNAL_FUNC (threads_click), NULL);
//设置回调函数 threads_click ( ), 其功能是开始系统运行, 系统入口函数
gtk_widget_add_accelerator (button, "clicked", accel_group,
GDK_A, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
tmp_toolbar_icon = create_pixmap (window, "deny.xpm");
if (tmp_toolbar_icon == NULL)
g_error ("Couldn't create pixmap");
button = gtk_toolbar_append_element (GTK_TOOLBAR (toolbar2),
GTK_TOOLBAR_CHILD_BUTTON,
NULL,
"Stop",
"Deny all hosts", NULL,
tmp_toolbar_icon, NULL, NULL);
gtk_widget_show (button);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
GTK_SIGNAL_FUNC (stop1), NULL);
//设置回调函数 stop1 ( ), 其功能是停止系统运行
gtk_widget_add_accelerator (button, "clicked", accel_group,
GDK_D, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
tmp_toolbar_icon = create_pixmap (window, "refresh.xpm");
if (tmp_toolbar_icon == NULL)
g_error ("Couldn't create pixmap");
button = gtk_toolbar_append_element (GTK_TOOLBAR (toolbar2),
GTK_TOOLBAR_CHILD_BUTTON,
NULL,
"Refresh",
"Refresh allowed hosts", NULL,

```

```
        tmp_toolbar_icon, NULL, NULL);

gtk_widget_show (button);
gtk_widget_add_accelerator (button, "clicked", accel_group,
                           GDK_R, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
tmp_toolbar_icon = create_pixmap (window, "clear.xpm");
if (tmp_toolbar_icon == NULL)
    g_error ("Couldn't create pixmap");
button = gtk_toolbar_append_element (GTK_TOOLBAR (toolbar2),
                                    GTK_TOOLBAR_CHILD_BUTTON,
                                    NULL,
                                    "Clear",
                                    "Help", NULL,
                                    tmp_toolbar_icon, NULL, NULL);

gtk_widget_show (button);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   GTK_SIGNAL_FUNC (button_clear_clicked), NULL);
//设置回调函数 button_clear_clicked ( ), 其功能是清除页面内容
gtk_widget_add_accelerator (button, "clicked", accel_group,
                           GDK_H, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
tmp_toolbar_icon = create_pixmap (window, "stop_alert.xpm");
if (tmp_toolbar_icon == NULL)
    g_error ("Couldn't create pixmap");
button = gtk_toolbar_append_element (GTK_TOOLBAR (toolbar2),
                                    GTK_TOOLBAR_CHILD_BUTTON,
                                    NULL,
                                    "Stop Alert",
                                    "Stop the Alert", NULL,
                                    tmp_toolbar_icon, NULL, NULL);

gtk_widget_show (button);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   GTK_SIGNAL_FUNC (stop_alert), NULL);
//设置回调函数 stop_alert ( ), 其功能是停止事件响应方式
gtk_widget_add_accelerator (button, "clicked", accel_group,
                           GDK_S, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
tmp_toolbar_icon = create_pixmap (window, "help.xpm");
if (tmp_toolbar_icon == NULL)
    g_error ("Couldn't create pixmap");
button = gtk_toolbar_append_element (GTK_TOOLBAR (toolbar2),
                                    GTK_TOOLBAR_CHILD_BUTTON,
                                    NULL,
                                    "Help",
                                    "Help", NULL,
```

```

        tmp_toolbar_icon, NULL, NULL);
gtk_widget_show (button);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (on_help_about_menubar_activate), NULL);
//设置回调函数 on_help_about_menubar_activate ( ),
//其功能是显示帮助对话框
gtk_widget_add_accelerator (button, "clicked", accel_group,
GDK_H, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
tmp_toolbar_icon = create_pixmap (window, "exit.xpm");
if (tmp_toolbar_icon == NULL)
g_error ("Couldn't create pixmap");
button = gtk_toolbar_append_element (GTK_TOOLBAR (toolbar2),
                                    GTK_TOOLBAR_CHILD_BUTTON,
                                    NULL,
                                    "Exit",
                                    "Quit this program", NULL,
                                    tmp_toolbar_icon, NULL, NULL);

gtk_widget_show (button);
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                    GTK_SIGNAL_FUNC (destroy), NULL);
//设置回调函数 destroy ( ), 其功能是销毁窗口, 退出程序
gtk_widget_add_accelerator (button, "clicked", accel_group,
GDK_Q, GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
gtk_widget_add_accelerator (button, "clicked", accel_group,
                            GDK_X, GDK_MOD1_MASK, GTK_ACCEL_VISIBLE);
// ===== 创建面板控件 =====//
vpaned = gtk_vpaned_new ();
gtk_box_pack_start (GTK_BOX (vbox), vpaned, TRUE, TRUE, 0);
gtk_paned_set_handle_size (GTK_PANED (vpaned), 10);
gtk_paned_set_gutter_size (GTK_PANED (vpaned), 15);
gtk_widget_show (vpaned);
// =====-创建记事本控件 notebook=====//
notebook = gtk_notebook_new ();
gtk_object_set_data (GTK_OBJECT (window), "notebook", notebook);
gtk_notebook_set_tab_pos (GTK_NOTEBOOK (notebook), GTK_POS_BOTTOM);
gtk_paned_add1 (GTK_PANED (vpaned), notebook);
gtk_widget_show (notebook);
//=====创建显示捕获信息的页面控件 =====//
listvbox = gtk_vbox_new (FALSE, 0);
label = gtk_label_new ("  sniffer  ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (notebook), listvbox, label);

```

```

gtk_widget_show (listvbox);
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
gtk_box_pack_start (GTK_BOX (listvbox), scrolled_window, TRUE, TRUE, 0);
gtk_widget_show (scrolled_window);
clist = gtk_clist_new (9);
gtk_object_set_data (GTK_OBJECT (window), "clist", clist);
gtk_clist_set_column_title (GTK_CLIST (clist), 0, "Packet Id");
gtk_clist_set_column_title (GTK_CLIST (clist), 1, "Time");
gtk_clist_set_column_title (GTK_CLIST (clist), 2, "Source Hardware");
gtk_clist_set_column_title (GTK_CLIST (clist), 3, "Destination Hardware");
gtk_clist_set_column_title (GTK_CLIST (clist), 4, "Source Ip");
gtk_clist_set_column_title (GTK_CLIST (clist), 5, "Destination Ip");
gtk_clist_set_column_title (GTK_CLIST (clist), 6, "Source Port");
gtk_clist_set_column_title (GTK_CLIST (clist), 7, "Destination Port");
gtk_clist_set_column_title (GTK_CLIST (clist), 8, "Ethernet Type");
gtk_clist_column_titles_show (GTK_CLIST (clist));
gtk_widget_ref (clist);
gtk_object_set_data_full (GTK_OBJECT (window), "clist", clist,
                          (GtkDestroyNotify) gtk_widget_unref);
gtk_signal_connect (GTK_OBJECT (clist), "select_row",
                   GTK_SIGNAL_FUNC (selection_made), NULL);
//设置回调函数 selection_made ( )
gtk_clist_set_shadow_type (GTK_CLIST (clist), GTK_SHADOW_IN);
gtk_clist_set_column_width (GTK_CLIST (clist), 0, 40);
gtk_clist_set_column_width (GTK_CLIST (clist), 1, 130);
gtk_clist_set_column_width (GTK_CLIST (clist), 2, 120);
gtk_clist_set_column_width (GTK_CLIST (clist), 3, 120);
gtk_clist_set_column_width (GTK_CLIST (clist), 4, 120);
gtk_clist_set_column_width (GTK_CLIST (clist), 5, 120);
gtk_clist_set_column_width (GTK_CLIST (clist), 6, 120);
gtk_clist_set_column_width (GTK_CLIST (clist), 7, 120);
gtk_clist_set_column_width (GTK_CLIST (clist), 8, 120);
gtk_container_add (GTK_CONTAINER (scrolled_window), clist);
gtk_widget_show (clist);
//=====创建显示 ARP 信息的页面控件=====//
listvbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
label = gtk_label_new ("      ARP      ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (notebook), listvbox, label);

```


[illegible]

```

gtk_clist_set_shadow_type (GTK_CLIST (clist2), GTK_SHADOW_IN);
gtk_clist_set_column_width (GTK_CLIST (clist2), 0, 40);
gtk_clist_set_column_width (GTK_CLIST (clist2), 1, 80);
gtk_clist_set_column_width (GTK_CLIST (clist2), 2, 80);
gtk_clist_set_column_width (GTK_CLIST (clist2), 3, 80);
gtk_clist_set_column_width (GTK_CLIST (clist2), 4, 80);
gtk_clist_set_column_width (GTK_CLIST (clist2), 5, 80);
gtk_clist_set_column_width (GTK_CLIST (clist2), 6, 40);
gtk_clist_set_column_width (GTK_CLIST (clist2), 7, 80);
gtk_clist_set_column_width (GTK_CLIST (clist2), 8, 80);
gtk_clist_set_column_width (GTK_CLIST (clist2), 10, 120);
gtk_clist_set_column_width (GTK_CLIST (clist2), 11, 120);
gtk_container_add (GTK_CONTAINER (scrolled_window), clist2);
gtk_widget_show (clist2);
//=====创建显示 TCP 信息的页面控件 tcp page=====//
listvbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
label = gtk_label_new ("      TCP      ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (notebook), listvbox, label);
gtk_widget_show (listvbox);
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
gtk_box_pack_start (GTK_BOX (listvbox), scrolled_window, TRUE, TRUE, 0);
gtk_widget_show (scrolled_window);
clist3 = gtk_clist_new_with_titles (12, titles_tcp);
gtk_object_set_data (GTK_OBJECT (window), "clist_3", clist3);
gtk_widget_ref (clist3);
gtk_object_set_data_full (GTK_OBJECT (window), "clist3", clist3,
                          (GtkDestroyNotify) gtk_widget_unref);
gtk_clist_set_shadow_type (GTK_CLIST (clist3), GTK_SHADOW_IN);
gtk_clist_set_column_width (GTK_CLIST (clist3), 0, 40);
gtk_clist_set_column_width (GTK_CLIST (clist3), 1, 40);
gtk_clist_set_column_width (GTK_CLIST (clist3), 2, 40);
gtk_clist_set_column_width (GTK_CLIST (clist3), 3, 80);
gtk_clist_set_column_width (GTK_CLIST (clist3), 4, 80);
gtk_clist_set_column_width (GTK_CLIST (clist3), 5, 80);
gtk_clist_set_column_width (GTK_CLIST (clist3), 6, 80);
gtk_clist_set_column_width (GTK_CLIST (clist3), 7, 80);
gtk_clist_set_column_width (GTK_CLIST (clist3), 8, 80);
gtk_clist_set_column_width (GTK_CLIST (clist3), 9, 80);

```

```

gtk_clist_set_column_width (GTK_CLIST (clist3), 10, 600);
gtk_clist_set_column_width (GTK_CLIST (clist3), 11, 600);
gtk_container_add (GTK_CONTAINER (scrolled_window), clist3);
gtk_widget_show (clist3);
//=====创建显示 UDP 信息的页面控件 =====//
listvbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
label = gtk_label_new ("      UDP      ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (notebook), listvbox, label);
gtk_widget_show (listvbox);
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
gtk_box_pack_start (GTK_BOX (listvbox), scrolled_window, TRUE, TRUE, 0);
gtk_widget_show (scrolled_window);
clist4 = gtk_clist_new_with_titles (5, titles_udp);
gtk_object_set_data (GTK_OBJECT (window), "clist_4", clist4);
gtk_widget_ref (clist4);
gtk_object_set_data_full (GTK_OBJECT (window), "clist4", clist4,
                          (GtkDestroyNotify) gtk_widget_unref);
gtk_clist_set_shadow_type (GTK_CLIST (clist4), GTK_SHADOW_IN);
gtk_clist_set_column_width (GTK_CLIST (clist4), 0, 40);
gtk_clist_set_column_width (GTK_CLIST (clist4), 1, 80);
gtk_clist_set_column_width (GTK_CLIST (clist4), 2, 80);
gtk_clist_set_column_width (GTK_CLIST (clist4), 3, 80);
gtk_clist_set_column_width (GTK_CLIST (clist4), 4, 80);
gtk_container_add (GTK_CONTAINER (scrolled_window), clist4);
gtk_widget_show (clist4);
//=====创建显示 ICMP 信息的页面=====//
listvbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
label = gtk_label_new ("      ICMP      ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (notebook), listvbox, label);
gtk_widget_show (listvbox);
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
gtk_box_pack_start (GTK_BOX (listvbox), scrolled_window, TRUE, TRUE, 0);
gtk_widget_show (scrolled_window);
clist5 = gtk_clist_new (7);

```

```

gtk_object_set_data (GTK_OBJECT (window), "clist_5", clist5);
gtk_widget_ref (clist5);
gtk_object_set_data_full (GTK_OBJECT (window), "clist5", clist5,
                          (GtkDestroyNotify) gtk_widget_unref);
gtk_clist_set_shadow_type (GTK_CLIST (clist5), GTK_SHADOW_IN);
gtk_clist_set_column_title (GTK_CLIST (clist5), 0, "number");
gtk_clist_set_column_title (GTK_CLIST (clist5), 1, "type");
gtk_clist_set_column_title (GTK_CLIST (clist5), 2, "code");
gtk_clist_set_column_title (GTK_CLIST (clist5), 3, "cksum");
gtk_clist_set_column_title (GTK_CLIST (clist5), 4, "Id");
gtk_clist_set_column_title (GTK_CLIST (clist5), 5, "Sequence");
gtk_clist_set_column_title (GTK_CLIST (clist5), 6, "Information");
gtk_clist_column_titles_show (GTK_CLIST (clist5));
gtk_clist_set_column_width (GTK_CLIST (clist5), 0, 40);
gtk_clist_set_column_width (GTK_CLIST (clist5), 1, 80);
gtk_clist_set_column_width (GTK_CLIST (clist5), 2, 80);
gtk_clist_set_column_width (GTK_CLIST (clist5), 3, 80);
gtk_clist_set_column_width (GTK_CLIST (clist5), 4, 80);
gtk_clist_set_column_width (GTK_CLIST (clist5), 5, 80);
gtk_clist_set_column_width (GTK_CLIST (clist5), 6, 80);
gtk_container_add (GTK_CONTAINER (scrolled_window), clist5);
gtk_widget_show (clist5);
//===== 创建显示 LLC 信息的页面 llc page=====//
listvbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
label = gtk_label_new ("      LLC      ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (notebook), listvbox, label);
gtk_widget_show (listvbox);
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
gtk_box_pack_start (GTK_BOX (listvbox), scrolled_window, TRUE, TRUE, 0);
gtk_widget_show (scrolled_window);
clist6 = gtk_clist_new (4);
gtk_object_set_data (GTK_OBJECT (window), "clist_6", clist6);
gtk_widget_ref (clist6);
gtk_object_set_data_full (GTK_OBJECT (window), "clist6", clist6,
                          (GtkDestroyNotify) gtk_widget_unref);
gtk_clist_set_shadow_type (GTK_CLIST (clist5), GTK_SHADOW_IN);
gtk_clist_set_column_title (GTK_CLIST (clist6), 0, "Packet ID");
gtk_clist_set_column_title (GTK_CLIST (clist6), 1, "DSAP");

```

```

gtk_clist_set_column_title (GTK_CLIST (clist6), 2, "SSAP");
gtk_clist_set_column_title (GTK_CLIST (clist6), 3, "Control");
gtk_clist_column_titles_show (GTK_CLIST (clist6));
gtk_clist_set_column_width (GTK_CLIST (clist6), 0, 100);
gtk_clist_set_column_width (GTK_CLIST (clist6), 1, 100);
gtk_clist_set_column_width (GTK_CLIST (clist6), 2, 100);
gtk_clist_set_column_width (GTK_CLIST (clist6), 3, 100);
gtk_container_add (GTK_CONTAINER (scrolled_window), clist6);
gtk_widget_show (clist6);
//===== 创建弹出菜单 create popup menu=====//
popup_menu = gtk_menu_new ();
mnu_1 = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/delete.xpm", "      Clear      ");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (mnu_1), box_xpm);
mnu_2 = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/filesave.xpm", "      Save      ");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (mnu_2), box_xpm);
mnu_3 = gtk_menu_item_new ();
box_xpm = xpm_label_box (window, "/xpm/open.xpm", "      Open      ");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (mnu_3), box_xpm);
gtk_signal_connect (GTK_OBJECT (mnu_1), "activate", GTK_SIGNAL_FUNC (clear),
                    NULL);
//设置回调函数 clear ( ), 其功能是清除页面内容
gtk_signal_connect (GTK_OBJECT (mnu_2), "activate",
                    GTK_SIGNAL_FUNC (savetofile), NULL);
//设置回调函数 savetofile ( ), 其功能是保存信息到文件中
gtk_signal_connect (GTK_OBJECT (mnu_3), "activate",
                    GTK_SIGNAL_FUNC (openfile_test), NULL);
//设置回调函数 openfile_test ( ), 其功能代开文件读信息
gtk_menu_append (GTK_MENU (popup_menu), mnu_1);
gtk_menu_append (GTK_MENU (popup_menu), mnu_2);
gtk_menu_append (GTK_MENU (popup_menu), mnu_3);
gtk_widget_show (mnu_1);
gtk_widget_show (mnu_2);
gtk_widget_show (mnu_3);
//=====显示第二个记事本控件 ( second notebook )=====//
text_notebook = gtk_notebook_new ();
gtk_object_set_data (GTK_OBJECT (window), "text_notebook", text_notebook);
gtk_notebook_set_tab_pos (GTK_NOTEBOOK (text_notebook), GTK_POS_BOTTOM);

```

```

gtk_paned_add2 (GTK_PANED (vpaned), text_notebook);
gtk_widget_show (text_notebook);
//=====显示标志的页面=====//
vbox_text = gtk_vbox_new (FALSE, 0);
gtk_widget_show (vbox_text);
gtk_widget_show (window);
vpaned1 = gtk_vpaned_new ();
gtk_box_pack_start (GTK_BOX (vbox_text), vpaned1, TRUE, TRUE, 0);
gtk_paned_set_handle_size (GTK_PANED (vpaned1), 10);
gtk_paned_set_gutter_size (GTK_PANED (vpaned1), 15);
gtk_widget_show (vpaned1);
frame = gtk_frame_new (NULL);
gtk_container_add (GTK_CONTAINER (frame), vbox_text);
gtk_frame_set_label_align (GTK_FRAME (frame), 1.0, 0.0);
gtk_frame_set_shadow_type (GTK_FRAME (frame), GTK_SHADOW_IN);
gtk_widget_show (frame);
hbox = gtk_vbox_new (FALSE, 0);
gtk_widget_show (hbox);
gtk_widget_show (window);
style = gtk_widget_get_style (window);
pixmap =
    gdk_pixmap_create_from_xpm_d (window->window, &mask,
                                   &style->bg[GTK_STATE_NORMAL],
                                   large_kvt_xpm);
pixmapwid = gtk_pixmap_new (pixmap, mask);
gtk_box_pack_start (GTK_BOX (hbox), pixmapwid, TRUE, TRUE, 10);
label = gtk_label_new ("    Designed    by    Liu Wen tao    ");
gtk_box_pack_start (GTK_BOX (hbox), label, TRUE, TRUE, 10);
gtk_widget_show (label);
gtk_widget_show (pixmapwid);
gtk_paned_add1 (GTK_PANED (vpaned1), hbox);
hbox = gtk_hbox_new (FALSE, 0);
gtk_widget_show (hbox);
text_scrolledwindow = gtk_scrolled_window_new (NULL, NULL);
gtk_widget_ref (text_scrolledwindow);
gtk_object_set_data_full (GTK_OBJECT (window), "text_scrolledwindow",
                           text_scrolledwindow,
                           (GtkDestroyNotify) gtk_widget_unref);
gtk_widget_show (text_scrolledwindow);
gtk_box_pack_start (GTK_BOX (hbox), text_scrolledwindow,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL), 0);

```

```

gtk_container_set_border_width (GTK_CONTAINER (text_scrolledwindow), 0);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (text_scrolledwindow),
GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
text = gtk_text_new (NULL, NULL);
gtk_object_set_data (GTK_OBJECT (window), "text", text);
gtk_text_set_editable (GTK_TEXT (text), FALSE);
gtk_widget_show (text);
gtk_container_add (GTK_CONTAINER (text_scrolledwindow), text);
gtk_paned_add2 (GTK_PANED (vpaned1), hbox);
label = gtk_label_new ("  NIDS  ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (text_notebook), frame, label);
//=====显示捕获到的大体信息的页面=====//
hbox = gtk_hbox_new (FALSE, 0);
gtk_widget_show (hbox);
frame = gtk_frame_new (NULL);
gtk_container_add (GTK_CONTAINER (frame), hbox);
gtk_frame_set_label_align (GTK_FRAME (frame), 1.0, 0.0);
gtk_frame_set_shadow_type (GTK_FRAME (frame), GTK_SHADOW_IN);
gtk_widget_show (frame);
text_scrolledwindow = gtk_scrolled_window_new (NULL, NULL);
gtk_widget_ref (text_scrolledwindow);
gtk_object_set_data_full (GTK_OBJECT (window), "text_scrolledwindow",
text_scrolledwindow,
(GtkDestroyNotify) gtk_widget_unref);
gtk_widget_show (text_scrolledwindow);
gtk_box_pack_start (GTK_BOX (hbox), text_scrolledwindow,
(GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
(GtkAttachOptions) (GTK_EXPAND | GTK_FILL), 0);
gtk_container_set_border_width (GTK_CONTAINER (text_scrolledwindow), 0);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (text_scrolledwindow),
GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
text1 = gtk_text_new (NULL, NULL);
gtk_signal_connect (GTK_OBJECT (text1), "event",
GTK_SIGNAL_FUNC (text_click_event), popup_menu);
//设置回调函数 text_click_event ( ), 其功能是弹出菜单
gtk_object_set_data (GTK_OBJECT (window), "text1", text1);
gtk_text_set_editable (GTK_TEXT (text1), FALSE);
gtk_widget_show (text1);
gtk_container_add (GTK_CONTAINER (text_scrolledwindow), text1);
label = gtk_label_new ("  Sniffer  ");
gtk_widget_show (label);

```

```

gtk_notebook_append_page (GTK_NOTEBOOK (text_notebook), frame, label);
//=====显示协议内容（ 分别用 16 进制和字符形式 ）的页面 =====//
hbox = gtk_hbox_new (FALSE, 0);
gtk_widget_show (hbox);
text_scrolledwindow = gtk_scrolled_window_new (NULL, NULL);
gtk_widget_ref (text_scrolledwindow);
gtk_object_set_data_full (GTK_OBJECT (window), "text_scrolledwindow",
                           text_scrolledwindow,
                           (GtkDestroyNotify) gtk_widget_unref);
gtk_widget_show (text_scrolledwindow);
gtk_box_pack_start (GTK_BOX (hbox), text_scrolledwindow,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL), 0);
gtk_container_set_border_width (GTK_CONTAINER (text_scrolledwindow), 0);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (text_scrolledwindow),
                                GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
text2 = gtk_text_new (NULL, NULL);
gtk_signal_connect (GTK_OBJECT (text2), "event",
                    GTK_SIGNAL_FUNC (text_click_event), popup_menu);
//设置回调函数 text_click_event ( ) , 其功能是弹出菜单
gtk_text_set_editable (GTK_TEXT (text2), FALSE);
gtk_object_set_data (GTK_OBJECT (window), "text2", text2);
gtk_widget_ref (text2);
gtk_object_set_data_full (GTK_OBJECT (window), "text2", text2,
                           (GtkDestroyNotify) gtk_widget_unref);
gtk_widget_show (text2);
gtk_container_add (GTK_CONTAINER (text_scrolledwindow), text2);
label = gtk_label_new ("    Content    ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (text_notebook), hbox, label);
//===== 显示 TCP 连接过程 =====//
hbox = gtk_hbox_new (FALSE, 0);
gtk_widget_show (hbox);
text_scrolledwindow = gtk_scrolled_window_new (NULL, NULL);
gtk_widget_ref (text_scrolledwindow);
gtk_object_set_data_full (GTK_OBJECT (window), "text_scrolledwindow",
                           text_scrolledwindow,
                           (GtkDestroyNotify) gtk_widget_unref);
gtk_widget_show (text_scrolledwindow);
gtk_box_pack_start (GTK_BOX (hbox), text_scrolledwindow,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL), 0);

```



```

gtk_container_set_border_width (GTK_CONTAINER (text_scrolledwindow), 0);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (text_scrolledwindow),
GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
text3 = gtk_text_new (NULL, NULL);
gtk_signal_connect (GTK_OBJECT (text3), "event",
                    GTK_SIGNAL_FUNC (text_click_event), popup_menu);
//设置回调函数 text_click_event ( ) , 其功能是弹出菜单
gtk_text_set_editable (GTK_TEXT (text3), FALSE);
gtk_object_set_data (GTK_OBJECT (window), "text3", text3);
gtk_widget_show (text3);
gtk_container_add (GTK_CONTAINER (text_scrolledwindow), text3);
label = gtk_label_new ("    Tcp-Connecting    ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (text_notebook), hbox, label);
//===== 创建检测扫描信息的页面控件 =====//
hbox = gtk_hbox_new (FALSE, 0);
gtk_widget_show (hbox);
text_scrolledwindow = gtk_scrolled_window_new (NULL, NULL);
gtk_widget_ref (text_scrolledwindow);
gtk_object_set_data_full (GTK_OBJECT (window), "text_scrolledwindow",
                          text_scrolledwindow,
                          (GtkDestroyNotify) gtk_widget_unref);
gtk_widget_show (text_scrolledwindow);
gtk_box_pack_start (GTK_BOX (hbox), text_scrolledwindow,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL), 0);
gtk_container_set_border_width (GTK_CONTAINER (text_scrolledwindow), 0);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (text_scrolledwindow),
GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
text4 = gtk_text_new (NULL, NULL);
gtk_signal_connect (GTK_OBJECT (text4), "event",
                    GTK_SIGNAL_FUNC (text_click_event), popup_menu);
//设置回调函数 text_click_event ( )
gtk_object_set_data (GTK_OBJECT (window), "text4", text4);
gtk_text_set_editable (GTK_TEXT (text4), FALSE);
gtk_widget_show (text4);
gtk_container_add (GTK_CONTAINER (text_scrolledwindow), text4);
label = gtk_label_new ("    Detect Scanning    ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (text_notebook), hbox, label);
//===== 显示 http 连接过程页面 =====//
hbox = gtk_hbox_new (FALSE, 0);

```

```

gtk_widget_show (hbox);
text_scrolledwindow = gtk_scrolled_window_new (NULL, NULL);
gtk_widget_ref (text_scrolledwindow);
gtk_object_set_data_full (GTK_OBJECT (window), "text_scrolledwindow",
                          text_scrolledwindow,
                          (GtkDestroyNotify) gtk_widget_unref);
gtk_widget_show (text_scrolledwindow);
gtk_box_pack_start (GTK_BOX (hbox), text_scrolledwindow,
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL),
                    (GtkAttachOptions) (GTK_EXPAND | GTK_FILL), 0);
gtk_container_set_border_width (GTK_CONTAINER (text_scrolledwindow), 0);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (text_scrolledwindow),
                                GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
text5 = gtk_text_new (NULL, NULL);
gtk_object_set_data (GTK_OBJECT (window), "text5", text5);
gtk_text_set_editable (GTK_TEXT (text5), FALSE);
gtk_widget_show (text5);
gtk_container_add (GTK_CONTAINER (text_scrolledwindow), text5);
label = gtk_label_new ("    HTTP    ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (text_notebook), hbox, label);
//===== 设置事件显示页面 =====//
listvbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
label = gtk_label_new ("        Alert Event        ");
gtk_widget_show (label);
gtk_notebook_append_page (GTK_NOTEBOOK (text_notebook), listvbox, label);
gtk_widget_show (listvbox);
scrolled_window = gtk_scrolled_window_new (NULL, NULL);
gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
gtk_box_pack_start (GTK_BOX (listvbox), scrolled_window, TRUE, TRUE, 0);
gtk_widget_show (scrolled_window);
clist_event = gtk_clist_new (11);
gtk_object_set_data (GTK_OBJECT (window), "clist_event", clist_event);
gtk_widget_ref (clist_event);
gtk_object_set_data_full (GTK_OBJECT (window), "clist_event", clist_event,
                          (GtkDestroyNotify) gtk_widget_unref);
gtk_clist_set_shadow_type (GTK_CLIST (clist_event), GTK_SHADOW_IN);
gtk_clist_set_column_title (GTK_CLIST (clist_event), 0, "Time");
//显示时间
gtk_clist_set_column_title (GTK_CLIST (clist_event), 1, "Event Name");

```

```
//显示事件名字
gtk_clist_set_column_title (GTK_CLIST (clist_event), 2, "Source MAC");
//显示源 MAC 地址
gtk_clist_set_column_title (GTK_CLIST (clist_event), 3, "Destination MAC");
//显示目的 MAC 地址
gtk_clist_set_column_title (GTK_CLIST (clist_event), 4, "Source IP");
//显示源 IP 地址
gtk_clist_set_column_title (GTK_CLIST (clist_event), 5, "Destination IP");
//显示目的 IP 地址
gtk_clist_set_column_title (GTK_CLIST (clist_event), 6, "Event Protocol");
//显示事件协议
gtk_clist_set_column_title (GTK_CLIST (clist_event), 7, "Event Code");
//显示事件代码
gtk_clist_set_column_title (GTK_CLIST (clist_event), 8, "Event Defination");
//显示事件定义
gtk_clist_set_column_title (GTK_CLIST (clist_event), 9,
                           "Evnet Informatino");
//显示事件描述
gtk_clist_set_column_title (GTK_CLIST (clist_event), 10, "Alert Type");
//显示事件响应方式
gtk_clist_column_titles_show (GTK_CLIST (clist_event));
gtk_clist_set_column_width (GTK_CLIST (clist_event), 0, 200);
gtk_clist_set_column_width (GTK_CLIST (clist_event), 1, 100);
gtk_clist_set_column_width (GTK_CLIST (clist_event), 2, 100);
gtk_clist_set_column_width (GTK_CLIST (clist_event), 3, 100);
gtk_clist_set_column_width (GTK_CLIST (clist_event), 4, 100);
gtk_clist_set_column_width (GTK_CLIST (clist_event), 5, 100);
gtk_clist_set_column_width (GTK_CLIST (clist_event), 6, 100);
gtk_clist_set_column_width (GTK_CLIST (clist_event), 7, 100);
gtk_clist_set_column_width (GTK_CLIST (clist_event), 8, 100);
gtk_clist_set_column_width (GTK_CLIST (clist_event), 9, 100);
gtk_clist_set_column_width (GTK_CLIST (clist_event), 10, 100);
gtk_container_add (GTK_CONTAINER (scrolled_window), clist_event);
gtk_widget_show (clist_event);
hbox = gtk_hbox_new (FALSE, 0);
showcount_label = gtk_label_new ("count");
gtk_widget_ref (showcount_label);
gtk_object_set_data_full (GTK_OBJECT (window), "showcount_label",
                           showcount_label,
                           (GtkDestroyNotify) gtk_widget_unref);
gtk_box_pack_start (GTK_BOX (hbox), showcount_label, TRUE, TRUE, 0);
gtk_widget_show (showcount_label);
```

```

label = gtk_label_new ("count");
gtk_widget_ref (label);
gtk_object_set_data_full (GTK_OBJECT (window), "label", label,
                          (GtkDestroyNotify) gtk_widget_unref);
gtk_box_pack_start (GTK_BOX (hbox), label, TRUE, TRUE, 0);
gtk_widget_show (label);
baojinlabel = gtk_label_new ("count");
gtk_widget_ref (baojinlabel);
gtk_object_set_data_full (GTK_OBJECT (window), "baojinlabel", baojinlabel,
                          (GtkDestroyNotify) gtk_widget_unref);
gtk_box_pack_start (GTK_BOX (hbox), baojinlabel, TRUE, TRUE, 0);
gtk_widget_show (baojinlabel);
gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, FALSE, 0);
gtk_widget_show (hbox);
separator = gtk_hseparator_new ();
gtk_box_pack_start (GTK_BOX (vbox), separator, FALSE, TRUE, 0);
gtk_widget_show (separator);
//===== 创建按钮行 =====//
buttons_hbox = gtk_hbox_new (FALSE, 0);
gtk_object_set_data (GTK_OBJECT (window), "buttons_hbox", buttons_hbox);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (button), "clicked",
                   GTK_SIGNAL_FUNC (threads_click), NULL);
//设置回调函数 threads_click ( ), 其功能是开始系统运行
gtk_box_pack_start (GTK_BOX (buttons_hbox), button, TRUE, TRUE, 5);
gtk_tooltips_set_tip (tooltips, button, "Show the data in graph forms.",
                      NULL);
box_xpm = xpm_label_box (window, "/xpm/sniffer.xpm", "Sniffer");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button), box_xpm);
gtk_widget_show (button);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button1 = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (button1), "clicked",
                   GTK_SIGNAL_FUNC (ok_clicked), NULL);
//设置回调函数 ok_clicked ( )
gtk_signal_connect_object (GTK_OBJECT (button1), "clicked",
                          GTK_SIGNAL_FUNC (ok_clicked_1),
                          GTK_OBJECT (window));

```

```
//设置回调函数 ok_clicked_1 ( )
gtk_box_pack_start (GTK_BOX (buttons_hbox), button1, TRUE, TRUE, 5);
gtk_tooltips_set_tip (tooltips, button1, "Sniffer one packet a time.",
    NULL);

box_xpm =
xpm_label_box_button (window, "/xpm/onepacket.xpm", "One packet");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button1), box_xpm);
gtk_widget_show (button1);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (button), "clicked", GTK_SIGNAL_FUNC (clear),
    NULL);
//设置回调函数 clear ( )
gtk_signal_connect (GTK_OBJECT (button), "clicked",
    GTK_SIGNAL_FUNC (clearfile), NULL);
//设置回调函数
gtk_box_pack_start (GTK_BOX (buttons_hbox), button, TRUE, TRUE, 5);
gtk_tooltips_set_tip (tooltips, button, "Clear all content in the text.",
    NULL);

box_xpm = xpm_label_box_button (window, "/xpm/delete.xpm", "Clear");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button), box_xpm);
gtk_widget_show (button);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (button), "clicked",
    GTK_SIGNAL_FUNC (savetofile), NULL);
//设置回调函数 savetofile ( ), 功能是保存当前信息
gtk_box_pack_start (GTK_BOX (buttons_hbox), button, TRUE, TRUE, 5);
gtk_tooltips_set_tip (tooltips, button, "Save the content to the file.",
    NULL);

box_xpm = xpm_label_box_button (window, "/xpm/filesave.xpm", "Save");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button), box_xpm);
gtk_widget_show (button);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (button), "clicked",
```

```
GTK_SIGNAL_FUNC (openfile_test), NULL);
//设置回调函数 openfile_test ( ), 其功能是打开文件
gtk_box_pack_start (GTK_BOX (buttons_hbox), button, TRUE, TRUE, 5);
gtk_tooltips_set_tip (tooltips, button, "Show the data in text forms.",
    NULL);
box_xpm = xpm_label_box_button (window, "/xpm/open.xpm", "Open");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button), box_xpm);
gtk_widget_show (button);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (button), "clicked", GTK_SIGNAL_FUNC (plot),
    NULL);
//设置回调函数 plot ( ), 其功能是显示图形
gtk_box_pack_start (GTK_BOX (buttons_hbox), button, TRUE, TRUE, 5);
gtk_tooltips_set_tip (tooltips, button, "Show the data in graph forms.",
    NULL);
box_xpm = xpm_label_box_button (window, "/xpm/idea.xpm", "Graph");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button), box_xpm);
gtk_widget_show (button);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (button), "clicked",
    GTK_SIGNAL_FUNC (tcpstat), NULL);
//设置回调函数 tcpstat ( ), 其功能是显示画面
gtk_box_pack_start (GTK_BOX (buttons_hbox), button, TRUE, TRUE, 5);
gtk_tooltips_set_tip (tooltips, button, "Show the data in graph forms.",
    NULL);
box_xpm = xpm_label_box_button (window, "/xpm/flag.xpm", "tcpstat");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button), box_xpm);
gtk_widget_show (button);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (button), "clicked", GTK_SIGNAL_FUNC (test),
    NULL);
//设置回调函数 test ( ), 其功能是测试响应方式
gtk_box_pack_start (GTK_BOX (buttons_hbox), button, TRUE, TRUE, 5);
```

```
gtk_tooltips_set_tip (tooltips, button, "Show the data in graph forms.",
    NULL);
box_xpm = xpm_label_box_button (window, "/xpm/minibreak.xpm", "Test");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button), box_xpm);
gtk_widget_show (button);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (button), "clicked",
    GTK_SIGNAL_FUNC (read_bpf), NULL);
//设置回调函数 read_bpf ( ), 其功能是读取 pbf 文件
gtk_box_pack_start (GTK_BOX (buttons_hbox), button, TRUE, TRUE, 5);
gtk_tooltips_set_tip (tooltips, button, "Show the data in graph forms.",
    NULL);
box_xpm = xpm_label_box_button (window, "/xpm/run.xpm", "Readbpf");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button), box_xpm);
gtk_widget_show (button);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (button), "clicked", GTK_SIGNAL_FUNC (stop1),
    NULL);
//设置回调函数 stop1 ( ), 其功能是停止系统运行
gtk_box_pack_start (GTK_BOX (buttons_hbox), button, TRUE, TRUE, 5);
gtk_tooltips_set_tip (tooltips, button, "Stop sniffing.....", NULL);
box_xpm = xpm_label_box_button (window, "/xpm/break.xpm", "Stop");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button), box_xpm);
gtk_widget_show (button);
tooltips = gtk_tooltips_new ();
gtk_tooltips_set_delay (tooltips, 0);
button = gtk_button_new ();
gtk_signal_connect (GTK_OBJECT (window), "destroy",
    GTK_SIGNAL_FUNC (gtk_exit), NULL);
//设置回调函数 gtk_exit ( ), 其功能是退出 GTK
gtk_signal_connect (GTK_OBJECT (window), "delete_event",
    GTK_SIGNAL_FUNC (gtk_exit), NULL);
//设置回调函数 gtk_exit ( ), 其功能是退出 GTK
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
    GTK_SIGNAL_FUNC (gtk_widget_destroy),
```

```
        GTK_OBJECT (window));
//设置回调函数 gtk_widget_destroy ( ), 其功能是销毁窗口
gtk_signal_connect (GTK_OBJECT (button), "clicked",
        GTK_SIGNAL_FUNC (destroy), NULL);
//设置回调函数 destroy ( ), 其功能是销毁窗口
gtk_box_pack_start (GTK_BOX (buttons_hbox), button, TRUE, TRUE, 5);
gtk_tooltips_set_tip (tooltips, button, "Quit the programe.", NULL);
box_xpm = xpm_label_box_button (window, "/xpm/quit.xpm", "Quit");
gtk_widget_show (box_xpm);
gtk_container_add (GTK_CONTAINER (button), box_xpm);
gtk_widget_show (button);
gtk_box_pack_start (GTK_BOX (vbox), buttons_hbox, FALSE, FALSE, 5);
gtk_widget_show (buttons_hbox);
gtk_container_add (GTK_CONTAINER (window), vbox);
gtk_widget_show (vbox);
gtk_widget_show (window);
mynids_init ();
srand ((unsigned int) time (NULL));
gdk_threads_enter ();
gtk_main ();
gdk_threads_leave ();
return 0;
}
```

11.6 小结

本系统的界面模块设计没有很复杂的理论，使用 GTK 来设计界面是很容易的。从上述所述，读者也可以看出其大部分功能是重复的操作，不同的只是其回调函数的设计。例如，不同的菜单，其界面设计都是一样的，只是其回调函数不同而已，那么只要实现了其回调函数，界面就具有了实际的功能，回调函数才真正实现了功能。这样界面的设计和实际功能的设计就可以分开进行，然后把他们联系在一起，最后就是一个完整的系统了。

参考文献及进一步的读物

- 1 Denning, D. An Intrusion Detection Model. IEEE Transactions on Software Engineering. 1987,13(2):222-232
- 2 State Transition Analysis: A Rule-Based Intrusion Detection Approach. K. Ilgun, R. A. Kemmerer, and P. A. Porras. IEEE Transactions on Software Engineering, 21(3). March, 1995
- 3 A Sense of Self for Unix Processes. S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. In Proceedings of the 1996 IEEE Symposium on Security and Privacy. 1996
- 4 Intrusion Detection via Static Analysis. D. Wagner and D. Dean. In Proceedings of the 2001 IEEE Symposium on Security and Privacy. 2001
- 5 A Framework for Constructing Features and Models for Intrusion Detection Systems. W. Lee and S. J. Stolfo. ACM Transactions on Information and System Security, 3(4). 2000
- 6 Detecting Intrusion Using System Calls: Alternative Data Models. C. Warrender, S. Forrest, and B. Perlmutter. In Proceedings of the 1999 IEEE Symposium on Security and Privacy. 1999
- 7 Logic Induction of Valid Behavior Specifications for Intrusion Detection. C. Ko. In Proceedings of the 2000 IEEE Symposium on Security and Privacy. 2000
- 8 Temporal Sequence Learning and Data Reduction for Anomaly Detection. T. Lane and C. E. Brodley. ACM Transactions on Information and System Security, 2(3). August, 1999
- 9 Network Intrusion Detection. B. Mukherjee, L. T. Heberlein, and K. N. Levitt. IEEE Network, May/June, 1994
- 10 Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. T. H. Ptacek and T. N. Newsham. Technical Report. 1998
- 11 <http://www.snort.org>
- 12 LIDS,<http://www.lids.org>
- 13 EMERALD,<http://www.sdl.sri.com/projects/emerald/>
- 14 AAFID,<http://www.cerias.purdue.edu/about/history/coast/projects/aafid.php>
- 15 GrIDS,<http://seclab.cs.ucdavis.edu/projects/arpa/grids/welcome.html>
- 16 RFC791, INTERNET PROTOCOL. 1981
- 17 RFC786, User Datagram Protocol. 1980
- 18 RFC826, An Ethernet Address Resolution Protocol. 1982
- 19 RFC792, INTERNET CONTROL MESSAGE PROTOCOL. 1981
- 20 RFC793, TRANSMISSION CONTROL PROTOCOL
- 21 Tim Carstens. Programming with pcap. <http://www.tcpdump.org/pcap.htm>
- 22 <http://www.isi.edu/gost/cidf/>
- 23 <http://www.ietf.org>
- 24 The TUNNEL Profile. <http://www.ietf.org/internet-drafts/draft-ietf-idwg-beep-tunnel-05.txt>. 2003
- 25 The Intrusion Detection Exchange Protocol. <http://www.ietf.org/internet-drafts/draft-ietf-idwg-beep-idxp-07.txt>. 2003
- 26 Intrusion Detection Message Exchange Format. [http://www.ietf.org/internet-drafts/](http://www.ietf.org/internet-drafts/draft-ietf-idwg-) draft-ietf-idwg-

- idmef-xml-10. txt. 2003
- 27 A Common Intrusion Specification Language. <http://www.isi.edu/gost/cidf/drafts/language.txt>. 1999
- 28 Communication in the Common Intrusion Detection Framework. <http://www.isi.edu/gost/cidf/drafts/communication.txt>. 1998
- 29 CIDF APIs. <http://www.isi.edu/gost/cidf/drafts/api.txt>
- 30 The Common Intrusion Detection Framework Architecture.
<http://www.isi.edu/gost/cidf/drafts/architecture.txt>
- 31 Network Programming. <http://www.ecst.csuchico.edu/~beej/guide/net/html/>
- 32 Graph-based Intrusion Detection System. <http://seclab.cs.ucdavis.edu/arpa/grids/welcome.html>
- 33 Next-Generation Intrusion Detection Expert System (NIDES). <http://www.sdl.sri.Com/projects/nides/index.html>
- 34 <http://www.mysql.com/>
- 35 <http://www.gtk.org/>
- 36 <http://www.linuxforum.net/>
- 37 GTK+ Reference Manual , <http://developer.gnome.org/doc/API/gtk/>
- 38 <http://www.snort.org/>
- 39 <http://www.lids.org/>
- 40 FAQ: Network Intrusion Detection Systems, <http://www.ticm.com/kb/faq/idsfaq.html>
- 41 Performance Adaptation in Real-Time Intrusion Detection Systems. Wenke Lee, Joao B. D. Cabrera, Ashley Thomas, Niranjana Balwalli, Sunmeet Saluja, and Yi Zhang. In Proceedings of The 5th International Symposium on Recent Advances in Intrusion Detection (RAID 2002), Zurich, Switzerland, October 2002
- 42 Using Embedded Sensors for Detecting Network Attacks. F. Kerschbaum, E. H. Spafford, and D. Zamboni. Purdue University Technical Report. 2000
- 43 EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. P. A. Porras and Peter G. Neumann. In Proceedings of the National Information Systems Security Conference. 1997
- 44 The Design of GrIDS: A Graph-Based Intrusion Detection System. S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, J. Rowe, S. Staniford, R. Yip, D. Zerkle. UC Davis Technical Report CSE-99-2. 1999
- 45 Experiences with Tripwire: Using Integrity Checkers for Intrusion Detection. G. H. Kim and E. H. Spafford. In USENIX Systems Administration, Networking and Security Conference III. 1994
- 46 Information Modeling for Intrusion Report Aggregation. R. P. Goldman, W. Heimerdinger, S. A. Harp, C. W. Geib, V. Thomas, and R. L. Carter. In Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX II). 2001
- 47 Probabilistic Alert Correlation. A. Valdes and K. Skinner. In Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID) 2001
- 48 Aggregation and Correlation of Intrusion-Detection Alerts. H. Debar and A. Wespi. In Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID). 2001
- 49 A Methodology for Testing Intrusion Detection Systems. N. J. Puketza, K. Zhang, M. Chung, B. Mukherjee, and R. A. Olsson. IEEE Transactions on Software Engineering, 22(10). October, 1996
- 50 Toward Cost-Sensitive Modeling for Intrusion Detection and Response. W. Lee, W. Fan, M. Miller, S. Stolfo, and E. Zadok. Journal of Computer Security 10(1,2), 2002

-
- 51 Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST) . U. Lindqvist and P. A. Porras. In Proceedings of the 1999 IEEE Symposium on Research in Security and Privacy. 1999
 - 52 The 1999 DARPA Off-line Intrusion Detection Evaluation. R. P. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. MIT Lincoln Lab Technical Report. 2000
 - 53 A Mission-Impact-Based Approach to INFOSEC Alarm Correlation. P. A. Porras, M. W. Fong, A. Valdes. In Proceedings of the 5th International Symposium on Recent Advances in Intrusion Detection (RAID). 2002
 - 54 A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems. K. Kendall. Master Thesis. MIT. 1999
 - 55 Attack Development for Intrusion Detection Evaluation. . K. Das. B. S. Thesis. MIT. 2000
 - 56 Statistical Foundations of Audit Trail Analysis for the Detection of Computer Misuse. P. Helman and G. Liepins. IEEE Transactions on Software Engineering, 19(9), September, 1993
 - 57 The Base-Rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. S. Axelsson. In Proceedings of the ACM Conference on Computer and Communication Security. November, 1999
 - 58 Information-Theoretic Measures for Anomaly Detection. W. Lee and D. Xiang. In Proceedings of the 2001 IEEE Symposium on Security and Privacy. May, 2001
 - 59 Benchmarking Anomaly-Based Detection Systems. R. Maxion and K. M. C Tan. In Proceedings of the 1st International Conference on Dependable Systems & Networks. 2000
 - 60 A Revised Taxonomy for Intrusion-Detection Systems. H. Debar, M. Dacier, and A. Wepsi. IBM Research Report. 1999
 - 61 Artificial Intelligence and Intrusion Detection: Current and Future Directions. J. Frank. In Proceedings of the 17th National Computer Security Conference. 1994
 - 62 Research in Intrusion Detection Systems: A Survey. S. Axelsson. Technical Report. 1999
 - 63 State of the Practice of Intrusion Detection Technologies. J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner. CMU/SEI Technical Report (CMU/SEI-99-TR-028. 1999
 - 64 An Application of Pattern Matching in Intrusion Detection. S. Kumar and E. H. Spafford. Purdue University Technical Report CSD-TR-94-013. 1994
 - 65 Lee W, Stolfo S J, Mok K W. A data mining framework for building intrusion detection models . Proceedings of the 1999 IEEE Symposium on Security and Privacy, Oakland, California . 1999: 120-132
 - 66 Autonomous Agents for Intrusion Detection , <http://www.cerias.purdue.edu/about/projects/aafid/>
 - 67 Douglas E. Comer, David L. Stevens. Internetworking with TCP/IP Volume II: Design, Implementation, and Internals, Third Edition, Vol. 2 . Prentice Hall PTR . 1998. 20 ~ 76
 - 68 W. Richard Stevens . The TCP/IP Illustrated, Volume 1: The Protocols, Vol. 1 Addison Wesley Longman, Inc. 1993. 14 ~ 50
 - 69 Douglas E. Comer. Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architectures, Fourth Edition, Vol. 1 . Prentice Hall PTR . 2000. 31 ~ 98
 - 70 Sean Walton . Linux Socket Programming . Sams. 2001. 2 ~ 67
 - 71 Bro: A System for Detecting Network Intruders in Real-Time. V. Paxson. Computer Networks, 31(23-24). December, 1999
 - 72 Marcus Ranum, "Intrusion Detection System: Expectation, Ideals and Realities. ", Computer Security Journal, Volume XIV, Number 4, 1999. 12 ~ 54

-
- 73 www.cert.org
- 74 M. F. Arnet, M. Coulombe. Inside TCP/IP, Second Edition. New Rider Publishing. 1997
- 75 The BSD Packet Filter: A New Architecture for User-level Packet Capture, Steven McCanne and Van Jacobson, Lawrence Berkeley Laboratory One Cyclotron Road Berkeley, 1992. 1 ~ 10
- 76 Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Off-line Intrusion Detection System Evaluation as Performed by Lincoln Laboratory. John McHugh. ACM Transactions on Information and System Security, 3(4). November, 2000
- 77 Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. R. P. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. P. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogrod, R. K. Cunningham, and M. A. Zissman. In Proceedings of the 2000 DARPA Information Survivability Conference and Exposition (DISCEX). 2000
- 78 Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. M. Handley and V. Paxson. In Proceedings of the 10th USENIX Security Symposium. August, 2001
- 79 Detecting Intruders in Computer Systems. T. Lunt. In Proceedings of the 1993 Conference on Auditing and Computer Technology. 1993
- 80 An Architecture for Intrusion Detection Using Autonomous Agents. J. S. Balasubramaniyan, J. O. Garcia-Fernandez, D. Isacoff, E. H. Spafford, and D. Zamboni. Purdue University Technical Report. 1998
- 81 Live Traffic Analysis of TCP/IP Gateways. P. A. Porras and A. Valdes. In Proceedings of the Internet Society Symposium on Network and Distributed System Security (NDSS). 1998
- 82 W. Ricjard Stevens. UNIX Network Programming Volume 1 Networking APIs: Sockets and XTI(second edition). Prentice Hall PTR, 1998. 4 ~ 34
- 83 The SRI IDES Statistical Anomaly Detector. H. S. Javitz and A. Valdes. In Proceedings of the IEEE Symposium on Research in Security and Privacy. 1991
- 84 Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. C. Ko, M. Ruschitzka, and K. Levitt. In Proceedings of the 1997 IEEE Symposium on Security and Privacy. 1997
- 85 A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. In Proceedings of the 2001 IEEE Symposium on Security and Privacy. 2001
- 86 Probabilistic Alert Correlation. A. Valdes and K. Skinner. In Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID 2001). 2001
- 87 蒋建春, 冯登国. 网络入侵检测原理与技术. 北京: 国防工业出版社, 2001. 42 ~ 63
- 88 戴英侠, 连一峰, 王航. 系统安全与入侵检测. 北京: 清华大学出版社, 2002. 33 ~ 58
- 89 蒋建春, 马恒太, 任党思等. 网络安全入侵检测: 研究综述. 软件学报, 2000,11(11): 1460 ~ 1466
- 90 (美)Stephen T. Satchell, H. B. J. Clifford. Linux IP 协议栈源代码分析. 机械工业出版社, 2000. 2 ~ 23
- 91 (美)Gary R. Wright, W. Richard Stevens, 范建华等(译). TCP/IP 详解 卷 1: 协议. 机械工业出版社, 2000. 3 ~ 37
- 92 (美)Gary R. Wright, W. Richard Stevens, 胡谷雨等(译). TCP/IP 详解 卷 3: TCP 事务协议, HTTP, NNTP 和 UNIX 域协议. 机械工业出版社, 2000. 15 ~ 89
- 93 韩东海, 王超, 李群. 入侵检测系统实例剖析. 北京: 清华大学出版社, 2002. 3 ~ 26
- 94 鲁士文. 计算机网络协议和实现技术. 清华大学出版社, 2000. 3 ~ 35

-
- 95 (美)Douglas E. Comer 著. 林瑶等译. 用 TCP/IP 进行网际互联. 第一卷: 原理、协议与结构. 电子工业出版社, 2001. 3 ~ 65
 - 96 林生著. 计算机通信与网络教程. 清华大学出版社, 1999. 13 ~ 97
 - 97 (美)Douglas E. Comer, David L. Stevens 著, 张娟等译. 用 TCP/IP 进行网际互联. 第二卷: 设计、实现与内核. 电子工业出版社, 2001. 20 ~ 75
 - 98 (美)Douglas E. Comer, David L. Stevens 著, 赵刚等译. 用 TCP/IP 进行网际互联. 第三卷: 客户——服务器设计. 电子工业出版社, 2001. 12 ~ 60
 - 99 雷澎等著. Linux 的内核与编程. 机械工业出版社, 2000. 15 ~ 80