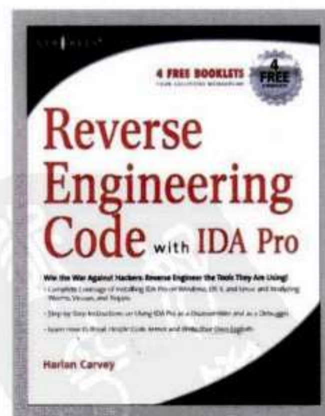
TURING 图灵程序设计丛书 网络安全系列

Reverse Engineering Code with IDA Pro

IDA Pro 代码破解揭秘

[美] Dan Kaminsky
Justin Ferguson
Jason Larsen
Luis Miras
Walter Pearce 著

看雪论坛翻译小组 译



- 安全编程修炼之道！
- 看雪学院等著名安全论坛强烈推荐
- 安全专家兼IOActive公司渗透测试总监 Dan Kaminsky经典力作

更多资源请点击访问稀.酷客(www.ckook.com)

“多年来我曾遇到许多高级的逆向工程问题，而它已在我的参考资料中占有一席之地。其中有些从前我没有意识到的有趣话题，它们使我获益匪浅。”

——亚马逊读者评论

Reverse Engineering Code with IDA Pro

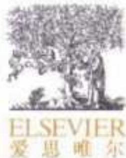
IDA Pro代码破解揭秘

如果你想掌握IDA Pro，如果你想掌握逆向工程编码的科学和艺术，如果你想进行更高效的安全研发和软件调试，本书正适合你！

本书是安全领域内的权威著作，也是少有的一本面向逆向工程编码的书籍！

书中阐述了IDA Pro逆向工程代码破解的精髓，细致而全面地讲述了利用IDA Pro挖掘并分析软件中的漏洞、逆向工程恶意代码、使用IDC脚本语言自动执行各项任务，指导读者在理解PE文件和ELF文件的基础上分析逆向工程的基本组件，使用IDA Pro调试软件和修改堆和栈的数据，利用反逆向功能终止他人对应用的逆向，还介绍了如何跟踪执行流、确定协议结构、分析协议中是否仍有未文档化的消息，以及如何编写IDC脚本和插件来自动执行复杂任务等内容。本书注重实践，提供了大量图示和示例代码供大家参考使用，可读性和可操作性极强。

本书译自原版*Reverse Engineering Code with IDA Pro*，并由Elsevier授权出版。



图灵网站：www.turingbook.com 热线：(010)51095186
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

分类建议 计算机/网络技术/网络安全

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-23416-2

9 787115 234162 >

ISBN 978-7-115-23416-2

定价：49.00元

更多资源请点击访问稀.酷客(www.ckook.com)

版 权 声 明

Reverse Engineering Code with IDA Pro by Dan Kaminsky, Justin Ferguson, Jason Larsen, Luis Miras, Walter Pearce, ISBN: 978-1-59749-237-9.

Copyright © 2008 by Elsevier. All rights reserved.

Authorized Simplified Chinese translation edition published by the Proprietor.

ISBN: 978-981-272-219-5.

Copyright© 2010 by Elsevier (Singapore) Pte Ltd. All rights reserved.

Elsevier (Singapore) Pte Ltd.
3 Killiney Road
#08-01 Winsland House I
Singapore 239519
Tel：(65)6349-0200
Fax：(65)6733-1817

First Published 2010
2010年初版

Printed in China by POSTS & TELECOM PRESS under special arrangement with Elsevier (Singapore) Pte Ltd. This edition is authorized for sale in China only, excluding Hong Kong SAR and Taiwan. Unauthorized export of this edition is a violation of the Copyright Act. Violation of this Law is subject to Civil and Criminal Penalties.

本书简体中文版由Elsevier (Singapore) Pte Ltd. 授权人民邮电出版社在中华人民共和国境内（不包括香港特别行政区和台湾地区）出版与销售。未经许可之出口，视为违反著作权法，将受法律之制裁。

更多资源请点击访问稀.酷客(www.ckook.com)

看雪致序

IDA Pro是一款非常优秀的反汇编工具，功能强大，对于有经验的程序员和软件安全相关技术人员来说是必用的工具。也许因为它过于强大和专业了，虽然很多人拥有它，但却难以熟练使用它，以至于很难发挥它强大的功能。同时很遗憾的是，IDA Pro附带提供的用户手册内容比较简洁，这同样增加了使用者了解和自如应用这款工具解决问题并将其变为利器的难度。

幸运的是，Dan Kaminsky（知名的安全专家，或者说世界著名的黑客，你可以在网上查到他的个人信息）和他的朋友不辞辛劳，为我们撰写了一部针对 IDA Pro 的应用指导性书籍 *Reverse Engineering Code with IDA Pro*。也许你已经注意到，国内外的读者和专业人士对本书的评价有很大争议。不可否认的是，作为 IDA Pro 的技术参考书，它的价值还是被大多数读者认可的，至少我个人是这么认为的。当然，其原著是英文版，对于国内的读者来说，这同样是个问题。

我很高兴看到 *Reverse Engineering Code with IDA Pro* 的中文版能够出版，这是看雪论坛翻译小组的兄弟们经过不懈努力带给我们大家的一份厚礼。更为难能可贵的是，他们都是利用业余时间凭借坚韧的毅力来完成这项工作的，这种吃苦和执着的精神正是我们这些程序员最为可贵的品质，我对他们的努力和成果表示由衷的祝贺和欣喜。

对于一部翻译作品，特别是技术书籍的翻译作品，在技术术语和基本含义正确的情况下，我们也许不该过多苛求信、达、雅的意境。同时我也注意到，本书的译者在一些非技术性描述的语句和章节中融入了意译，我认为这是可以接受和理解的。我认为在翻译技术书籍时，也能够并且应该可以表达译者的情感和思想，而非简单的直译。如果一味直译，读者可能会感到乏味和枯燥；在译作中融入译者的思想，就体现了译者的个性和精神。

看雪软件安全论坛已经走过十年，出版了不少广受读者欢迎的专业书籍，我希望本书能够继往开来，成为我们辉煌历程的新里程碑。再次感谢本书的全体译者，我坚信你们的努力能够被读者认可。

为了看雪下一个更加精彩的十年，让我们一起携手共进。

段钢

于北京

更多资源请点击访问稀.酷客(www.ckook.com)

序

翻译很苦，作为翻译了三本书的人，我深有感触；但组织者更难，我是第一次体会。其中的酸甜苦辣不说也罢。这本书能和大家见面，我作为组织者首先要感谢全心参与的四位译者，他们是余洋、任晓枫、崔孝晨和李军。

四位译者虽然专业不同，翻译技能的熟练度不一样，但都一样认真，一样按要求细心修订。我想，所有的荣耀归于他们；如果有责难，就冲我来吧，我作为组织者，责无旁贷。在四位译者中，我最欣赏的是第 7 章和第 8 章的译者崔孝晨。他的行文很流畅，翻译也很到位，最后才了解他原来翻译过《Windows 取证分析》。当然，最令我钦佩的是他的一颗爱心，他主动提出要把稿费捐给地震灾区。

其次，要感谢看雪为这些志同道合的朋友提供了一个自由交流的空间；接下来，要感谢北京图灵文化发展有限公司的各位编辑，在与他们打交道的过程中，虽然一些细节有待商榷，但他们认真负责的态度深深感染了我，我想凭此一条，其他的就已不重要了；我还要借此机会向我的妻子表示感谢，虽然此次组织活动纯属业余活动，但她仍给予我一如既往的支持，理解万岁。

最后，请允许我向论坛上各位等待此书的朋友表示歉意，你们的支持是动力，也是鞭策。thinkSJ、colboy、Anplando、bwin、fool、shellwolf、yingyue、terren、batter、elance、brodbus、combojiang、lixianhuei、wzmooo、鹩蛋壳、太难了、zapline、jordanpz、Second、icetowater、magicfx、iawen、wtxpwh、KENW、安摧、不尽湘江、seagull、gooogleman、克里克里、duanzhu、chinazgj、xuheping、charme、jwtck、lovesuae、hcaihao、riusksk、adomore，感谢你们。

罗爱国

更多资源请点击访问稀.酷客(www.ckook.com)

关于 IOActive

IOActive 成立于 1998 年，现已成功跻身于西北部计算机安全界的领导者行列（西北部计算机安全界致力于基础架构评估服务、应用程序安全服务、可管理服务、事件响应服务及教育服务）。公司为财富 500 强中的多数公司提供从企业风险管理到中立的安全硬件及众多应用程序验证服务。此外，公司还为大多数保险公司、州政府及能源公司提供 IT 灾难恢复及业务持续性计划服务。IOActive 的顾问都是公认的本地区或全国性计算机安全组织的成员和积极的撰稿人，这些安全组织包括 SANS、Agora、CRIME、ISSA、CTIN、WSA、HoneyNet 研究联盟、OWASP、华盛顿州立大学信息保险学院等。

更多资源请点击访问稀.酷客(www.ckook.com)

技术编辑及著作者

Dan Kaminsky IOActive 公司的渗透测试主管。Dan 自 1999 年起（在去 Cisco 及 Avaya 上班前）在安全圈内就非常活跃。使他广为人知的是他在黑帽子大会上一系列的“Black Ops”演讲，此外，他还是唯一一位出席并在每届微软内部训练活动“Blue Hat”上发言的人。Dan 致力于设计层面的故障分析，特别针对大规模的网络应用程序。Dan 经常收集世界各地互联网的详细健康数据，最近用这些数据检测大部分 rootkit 在世界各地的繁殖情况。Dan 是这个世界上少数几个同时拥有技术专长及执行层咨询技巧和能力的人。

更多资源请点击访问稀.酷客(www.ckook.com)

特约作者

Justin Ferguson IOActive公司的安全顾问及研究员。他通过IOActive的应用安全实践，帮助财富500强公司理解并减轻复杂软件计算环境里的风险。Justin涉足从金融行业到联邦政府的各行业，在逆向工程、源代码审计、恶意程序分析、企业安全分析等方面有超过6年的经验。

> 我非常感激我的父亲Bruce Dennis Ferguson，他是我心目中的伟人；我从来没有后悔，也不会为成为现在这样的人而感到歉意。我感谢所有来自波士顿的蓝领阶层，他们为了使儿女过上好的生活日夜操劳。当然，还有他们的另一半，每天在旁边默默支持他们的女士，你们在我心目中是最美的。我想借此机会问候来自南端和布罗克顿/南岸，仍在苦苦挣扎的每一个人，他们相信忍受不应得的苦难是一种赎罪。圣徒犹大，为我们祈祷吧。

Jason Larsen 渗透并拥有这个星球上最严密的一些系统。他的职业生涯开始于他在爱达荷州立大学发现互联网范围的秘密扫描时。为了支持他深入研究并开发检测系统，以及奖励他作为真能屏蔽渗透的第一代入侵保护系统的著作者之一，他被授予二次奖学金。出于国家安全的考量，Larsen先生不能公开所做的大部分工作。他通过爱达荷国家实验室，为能源部的大多数SCADA和PCS系统的安全问题开发更简练的解决方案。几个行业中的数百家客户，包括美国和国外的都从他的安全工作中获益。

> 我想把本书献给有着无限忍耐力并理解我的女友。感谢你听到最近的问题时微微点头示意，感谢你偶尔推开房门放进一缕缕阳光。每个电脑迷都应该拥有一位纹身的伴侣。

Luis Miras 一位独立安全研究员。他曾为一些安全产品厂商及主流咨询公司工作。他的兴趣包括漏洞研究、二进制分析和硬件/软件逆向。在过去，他曾涉足数字化设计和嵌入式编程。他曾出席CanSecWest、Black Hat、CCC Congress、XCon、Recon、DefCon及其他世界级的会议。他若偶尔放下IDA或集成电路工作，你便很可能发现他在制做一些甜粉。

> 谨以此书献给我的父母与兄弟。我要感谢Don Omar、Nancy和Nas，他们为我提供了编码配乐。我还想问候我所有的朋友并告诉他们我还活着，而且不会再消失了。

更多资源请点击访问稀.酷客(www.ckook.com)

感谢 Sebastian “topo” Muniz 与我讨论 IDA 及他们那些跳跃的思维。

Walter Pearce 为 IOActive 提供应用安全和渗透测试服务，研究和开发自动化 IT 安全测试和保护功能的高级工具。他的职业生涯从 12 岁开始，他的第一个职业角色是在线零售商数据中心群集的操作员，而这使他最终成为金融服务公司和行业里的资深编程工程师。他在金融行业工作期间，专攻内部威胁的概念，并从事设计工作以降低此类事件发生的概率。客户经常要求 Pearce 先生提供专家级的应用安全服务，这涉及多种平台和（编程）语言。

致 Becca、妈妈和 David。我爱你们

更多资源请点击访问稀.酷客(www.ckook.com)

溜客安全网
WWW.176KU.COM
全力打造最优秀中国黑客技术资源共享平台
溜客精神：
技術共享，資源共享，資料共享
不求最好，只求較好
做中國較好的網絡安全資料站
300G成套精品教程免费下载
每月网络期刊，黑客期刊发布
请将本站推荐给更多的好友
让大家都成为溜客一员
溜客資料共享群：
访问溜客安全网最下方
查看本站最新共享QQ群
溜客网络安全技术人才培训进行中
做一个通过正道可以养活自己的黑客
从我做起，不做伪黑客
WWW.176KU.COM/VIP.HTML

目　　录

第 1 章　导言 …… 1
1.1　代码调试器概述 …… 2
1.2　小结 …… 3
第 2 章　汇编及逆向工程基础 …… 5
2.1　导言 …… 6
2.2　汇编语言及 IA-32 处理器 …… 6
2.3　栈、堆及二进制可执行文件中的其他区段 …… 14
2.4　最新的 IA-32 指令集及参考资料 …… 19
2.5　小结 …… 25
第 3 章　可移植可执行文件格式和可执行链接格式 …… 27
3.1　导言 …… 28
3.2　可移植可执行文件格式 …… 28
3.3　可执行链接格式 …… 35
3.4　小结 …… 47
第 4 章　实战 1 …… 49
4.1　导言 …… 50
4.2　跟踪执行流 …… 50
4.3　快速跟踪并找出解决方案 …… 63
4.4　常见问题 …… 64
第 5 章　调试 …… 65
5.1　导言 …… 66
5.2　调试的基础知识 …… 66
5.2.1　断点 …… 67
5.2.2　单步 …… 68
5.2.3　监视 …… 68
5.2.4　异常 …… 68
5.2.5　跟踪 …… 69
5.3　使用 IDA Pro 进行调试 …… 69
5.4　调试技术在逆向工程中的应用 …… 71
5.5　堆和栈的访问和修改 …… 78
5.6　其他调试器 …… 80
5.6.1　Windbg …… 80
5.6.2　Ollydbg …… 80
5.6.3　immdbg …… 81
5.6.4　PaiMei/PyDbg …… 81
5.6.5　GDB …… 81
5.7　小结 …… 82
第 6 章　反逆向技术 …… 83
6.1　导言 …… 84
6.2　调试 …… 84
6.3　举例阐述 …… 87
6.4　混淆技术 …… 87
6.5　小结 …… 104
第 7 章　实战 2 …… 105
7.1　协议问题 …… 106
7.2　协议结构 …… 106
7.2.1　分帧与重组 …… 106
7.2.2　自相似性 …… 108
7.2.3　Hit Marking …… 120

更多资源请点击访问稀.酷客(www.ckook.com)

7.2.4 Hitlist 示例 …… 124

第 8 章 高级攻略 …… 129

8.1 导言 …… 130

8.2 逆向分析恶意软件 …… 131

第 9 章 IDA 脚本编写和插件 …… 161

9.1 导言 …… 162

9.2 IDA 脚本编写基础 …… 162

9.3 IDC 语法 …… 162

9.3.1 输出 …… 163

9.3.2 变量 …… 164

9.3.3 条件 …… 165

9.3.4 循环 …… 165

9.3.5 函数 …… 166

9.3.6 全局变量 …… 168

9.4 简单脚本示例 …… 170

9.5 编写 IDC 脚本 …… 173

9.5.1 用 IDC 解决问题 …… 173

9.5.2 新的 IDC 调试器功能 …… 180

9.5.3 有用的 IDC 函数 …… 181

9.6 IDA 插件基础 …… 185

9.6.1 模块/插件资源 …… 186

9.6.2 IDA Pro SDK 介绍 …… 187

9.7 插件语法 …… 188

9.8 设置开发环境 …… 189

9.9 简单插件示例 …… 191

9.9.1 Hello World 插件 …… 191

9.9.2 find memcpy 插件 …… 194

9.10 间接调用插件 …… 209

9.10.1 收集数据 …… 210

9.10.2 用户接口 …… 211

9.10.3 实现回调 …… 213

9.10.4 显示结果 …… 215

9.11 插件开发和调试策略 …… 250

9.11.1 创建一个新的 IDA 开发目录 …… 250

9.11.2 编辑配置文件 …… 250

9.12 加载器 …… 255

9.13 处理器模块 …… 256

9.14 第三方脚本插件 …… 256

9.14.1 IDAPython …… 256

9.14.2 IDARub …… 257

9.15 常见问题 …… 257

更多资源请点击访问稀.酷客(www.ckook.com)

你
想
換
嗎
？

www.17huan.com

更多资源请点击访问稀.酷客(www.ckook.com)

第 1 章

导　言

曾几何时，信息安全专家驰骋的疆场已经发生了翻天覆地的变化。我们的任务不再是防范那些窥视我们重要资产的好奇的年轻人，而是变成了防御那些在金融或地缘政治利益驱使下的有组织犯罪，这样的攻击尤其残酷无情又手段高超。

应用程序和通信协议中的安全漏洞层出不穷，因特网也变得日益庞大而复杂，敏感信息随处可见，所有这些都为我们的对手创造了“多目标作战环境”。这些“对手”使用不同的高级软件规避 IDS、IPS 和 AV 的检测引擎，并在“肉鸡”上实现完整的远程控制和窃取数据的功能。为了了解和预测这些恶意软件会产生怎样的影响，我们免不了会利用高级逆向工程技术，使用 Data Rescue 和 Zynamics 等公司开发的行业标准工具。

本书作者为业内资深人士。本书为我们带来了逆向工程领域的前沿思想，我相信大家一定可以从中找到必要的信息，从而走向计算机安全的最前沿。

非常感谢 Lauren Vogt、Ted Ipsen、Dan Kaminsky 、Jason Larsen、Walter Pearce、Justin Ferguson、Luis Miras，以及使这本书顺利出版的 Syngress 的热心人。

Joshua J. Pennell

IOActive 公司创始人与 CEO

更多资源请点击访问稀.酷客(www.ckook.com)

1.1 代码调试器概述

对我们来说，迟早都需要了解可执行文件的一切底细。比如说，你可能想知道：

- 它调用的精确内存地址；
- 它正写入数据的准确内存区域；
- 它正在从哪里读取数据；
- 它正在使用哪个寄存器。

当你没有源代码而又想分析可执行文件时，调试器可以通过对文件进行反汇编帮助你。这一般发生在分析恶意软件的时候，在这种情形下，你还指望找到它的源代码？本部分的内容不是教你熟练使用这些调试器，而是介绍一些可能会用到的调试器。调试器非常强大，要想熟练掌握，需要长时间的练习才行。

调试器中的“精英”也即本书介绍的重点是 Data Rescuse 公司的 IDA Pro（Interactive Disassembler Pro）。如果你打算在企业级环境下使用调试器，它将是你的首选。总的来说，它的价格还不算太贵，可以说是物有所值。

提示

Data Rescuse 在其网站 www.datarescue.com/idabase/index.htm 提供了 IDA Pro 的试用版。当然，试用版会有一些限制。比如说，只能处理少数几种格式的文件和适用于少数处理器类型，有使用时间的限制，只能在 Windows 平台上使用等。

IDA Pro 不仅是简单的调试器，而且是一个可编程的交互式反汇编器和调试器。有了 IDA Pro，你所能见到的可执行文件格式或应用程序应该都不在话下。IDA Pro 可以处理各种平台上的文件，从 Xbox、Playstation、Nintendo 到 Macintosh，从 PDA 到 Windows、UNIX 等，几乎包含所有类型的平台。图 1-1 显示了 IDA Pro 启动时出现的界面。注意，界面上显示了各种文件类型及选项卡，帮助你为反汇编的文件选择合适的分析引擎。

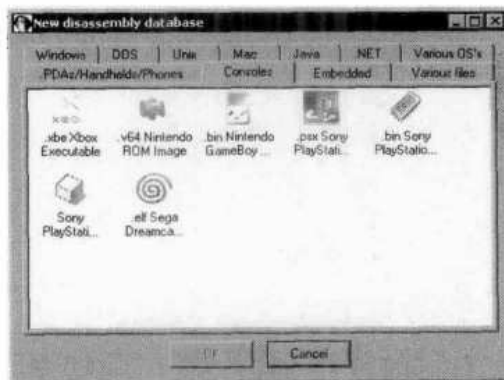


图 1-1 IDA Pro 反汇编数据库选择器加载启动界面

更多资源请点击访问稀.酷客(www.ckook.com)

在图 1-2 中，IDA Pro 加载了名为 instantmsgre.exe 的 WootBot 变种。从图 1-2 中我们看到，它被打包软件 Molebox 处理过。此外，你还可以看到它正在生成的内存调用，以及调用了哪些 Windows DLL。当你需要击退病毒或恶意软件，特别是为了修复系统而编写专杀工具时，这些信息可都是无价之宝。

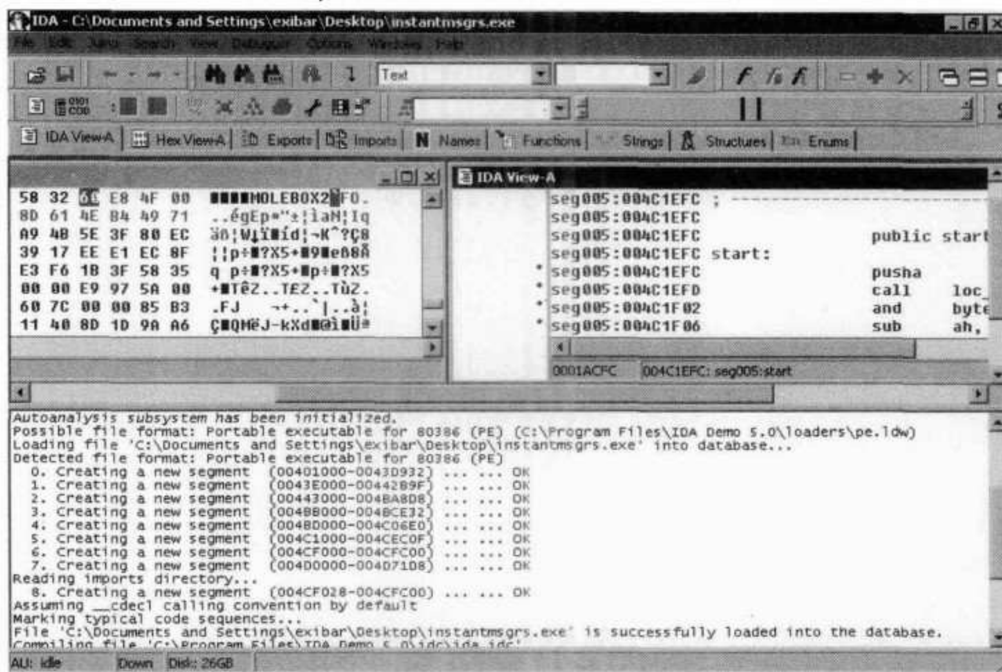


图 1-2 IDA Pro 反汇编 WootBot 的变种——instantmsgre.exe

1.2 小结

IDA 是 Windows 下众多调试工具中颇受大家欢迎的一种。IDA Pro 首先是一个反汇编器，可以显示二进制汇编码（可执行文件或 DLL（Dynamic Link Library，动态链接库）），它提供的某些高级功能使我们更容易理解汇编代码。其次，它又是一个调试器，用户可以逐条调试二进制文件中的指令，从而确定当前正在执行哪条指令，以及执行的顺序等。在本书中，我们会一一介绍这些内容。IDA Pro 广泛应用于恶意软件分析、软件漏洞研究等目的。你可以直接在 www.datarescue.com 上订购 IDA Pro。

更多资源请点击访问稀.酷客(www.ckook.com)

第 2 章

汇编及逆向工程基础

本章内容：

- 汇编语言及 IA-32 处理器
- 堆、栈及二进制可执行文件中的其他区段
- 最新的 IA-32 指令集及参考资料

☑ 小结

更多资源请点击访问稀.酷客(www.ckook.com)

2.1 导言

本章，我们将介绍一些基础性的内容，包括汇编语言、Intel 架构的处理器，以及进一步学习时需要掌握的概念。本书着眼于 32 位 Intel 架构（IA-32）的汇编语言，涉及 Windows 及 Linux 两种操作系统。我们假设读者至少了解一些 IA-32 汇编语言（尽管本章在一定程度上覆盖了架构及指令集）的知识，并能熟练使用 C/C++。本章为入门者提供了开展工作所需要掌握的基础知识，以及作者认为有必要提及的参考资料。

2.2 汇编语言及 IA-32 处理器

汇编是与计算机进行交互的一种有趣的方法。Donald Knuth 曾说过：“科学就是我们已经充分理解并可以向计算机解释的，除此之外的就是艺术。”对我而言，这也普遍存在于汇编语言程序设计中，因为在编写汇编程序时，通常会滥用指令，而不是让它做“本职工作”（例如，用 LEA（Load Effective Address，取有效地址指令）做与指针算法无关的事情）。但到底什么是汇编呢？汇编是指一些与处理器指令集一一对应的指令助记符的使用。也就是说，代码与处理器之间没有抽象层：所写即所得（尽管在有些平台上有例外：汇编器先输出伪指令，然后再把它们转换成多条处理器指令——但前面所说的依然正确）。

以一条 C 语句为例：

```
return 0;
```

我们最终会得到如下的汇编指令：

```
leave
xor    eax,    eax
ret
```

注意

各种硬件平台的汇编语言语法不尽相同。上面的汇编代码段及本书中使用的大多数汇编代码段采用的是 Intel 语法；另外比较流行的、在 Unix 里大量使用的是 AT&T 语法，它和 Intel 语法有一些区别。我们在这里之所以选用 Intel 语法，主要是因为它被用于 IDA 的反汇编，而且它要比 AT&T 语法更普及些。这也意味着使用 Intel 语法，可以找到更多的参考书、官方文档及可以帮助解答问题的人。

本书并不准备介绍 AT&T 语法与 Intel 语法之间的异同，但通过下面的例子你可以了解个大概：

Intel:

```
  leave
xor    eax, eax
ret
```

更多资源请点击访问稀.酷客(www.ckook.com)

AT&T:

```
    leave
  xorl    %eax, %eax
  ret
```

值得注意的是，很多地方仍在使用 AT&T 语法，特别是 Unix 中它通常被用作标准语法(但随着IA-32计算机上的Unix及类Unix操作系统的兴起，这种情形正在慢慢改变)。因此，花些时间学习它肯定不会错，特别是当你对 Unix 或其他硬件平台感兴趣时。

即使你对我们前面所列的汇编代码一无所知，也不用太担心。我只是想让你和汇编代码混个脸熟，知道它们就是所谓的 IA-32 汇编，并且知道它等同于 C 语言里的“return 0”语句就可以了。虽然汇编属于低级语言，但处理器本身并不理解汇编代码，汇编代码只是为了便于人们阅读而采用的等同于机器码的助记符。汇编代码经编译器处理后输出为操作码（opcodes），操作码是指令或执行单条指令所必需的开关序列的二进制表示形式。为了符合人们的胃口，操作码一般用十六进制表示，因为这种表示形式比二进制更容易读一些。例如，前面的汇编指令对应的操作码如下所示：

```
0xC9            (leave)
0x31, 0xc0      (xor eax,eax)
0xC9            (ret)
```

正如我们所看到的，这里是三个基本的抽象层，但随着计算技术的发展，我们增加了越来越多的抽象层，比如说 Java 及.NET 应用程序使用的虚拟机概念。不过，它们最终还是要还原成汇编代码，并最终以0、1序列的形式进入处理器。关于操作码的更完整说明请参阅 Intel® *64 and IA-32 Architectures Software Developer's Manual Volume 2A, section 2.1.2*。

至此，我们已经对汇编指令有了一些初步了解，但怎样使用它们呢？汇编指令的指令后面通常会跟一个参数（也称为操作数），视具体的指令而定，操作数可能是常量、内存变量或寄存器。常量是最简单的形式，一般是在源代码中定义的。例如，如果一段代码中用到了如下指令：

```
mov eax, 0x1234
```

那么十六进制数 0x1234 就是一个常量。常量简单明了，一般直接编码到指令中，除此以外几乎没有什么要额外说明的了。但有一点比较有意思，如果思考过我们在前面提到的返回零的 C 语句，聪明的读者可能已经注意到了，编译器生成的汇编语句并没有包含常量——即使源代码中有。这是编译器优化导致的结果，它认为复制零的操作比执行异或运算更消耗资源（两者的执行结果是一样的）。

我们接下来要介绍的是寄存器。寄存器与 C/C++中的变量类似。通用寄存器可以保存整数、偏移量、立即数、指针，或任何能用 32 位二进制表示的东西。它本质上是预分配的、物理上存在于处理器中的变量，总是位于一定范围内。它们的用法与变量还是有些差异；对寄存器而言，它们可以被反复使用，但在 C 或 C++程序里，定义的变量通常只用于一个目的，且不再把它用在

更多资源请点击访问稀.酷客(www.ckook.com)

其他地方。

> **工具和陷阱**
>
> **操作码和Shellcode**
>
> 当我们刚进入数字世界，初次面对 shellcode 时几乎都会发懵。这些神秘的十六进制数字组成的数组在我们面前跳来跳去，而我们却不了解它到底要做什么。研究过破解程序（exploit）的人应该都见过它，但我们不建议读者过早地接触它们，因为欲速则不达。
>
> 不过可以肯定的是，一些关于它的说法被神秘化复杂化了。Shellcode 其实只是一系列的操作码，一般保存在 C 字符串数组里。称它为 shellcode 主要是因为这一系列的操作码是执行 shell（例如/bin/sh 或 cmd.exe）所必需的指令。在这里，如果我们以下面的 C 语句为例生成 shellcode:
>
> ```
> return 0;
> ```
>
> 则使用的是这条指令所对应的操作码，即 0xC9,0x31,0xC0,0xC9。如果把它放到 C 程序里，看起来应该和下面差不多：
>
> ```
> unsigned char shellcode[]="\xc9\x31\xc0\xc9";
> ```
>
> 现在知道这是怎么一回事了吧？你现在可能会因为以前把它们想得太复杂而感到自己有点愚蠢，不必这样！我认为，这是每个人认知过程中必定会经历的阶段——至少我是这样的。

IA-32 中有 8 个 32 位通用寄存器，6 个 16 位段寄存器，1 个 32 位指令指针寄存器，1 个 32 位状态寄存器，5 个控制寄存器，3 个内存管理寄存器，8 个调试寄存器，等等。在大多数情况下，我们只用到通用寄存器、指令指针寄存器、段寄存器及状态寄存器。如果处理 OS 的驱动程序之类的，则更有可能会碰到其他的寄存器。在这里，我们只准备介绍通用寄存器、指令指针寄存器、状态寄存器及段寄存器。对于其他的寄存器，只要知道它们存在就可以了。当然，如果你有兴趣进一步学习，则可以阅读 Intel 文档。

8 个 32 位通用寄存器分别是：EAX、EBX、ECX、EDX、ESI、EDI、EBP 和 ESP。这些寄存器中除了几个有专门的用处外，其他的可以随便用。例如，不少指令会默认把某些寄存器作为参数（操作数）。比如，多数字符串指令通常把 ECX 用作计数器，把 ESI 作为源指针，把 EDI 作为目的指针。此外，在某些内存模型中，有些指令默认把某些寄存器作为段的基址（这些下面很快会讲到）。最后，有些操作会涉及一些寄存器，虽然它们并没有在指令中体现出来。例如，栈操作通常会使用 EBP 和 ESP 寄存器，如果它们包含的值没有映射到当前进程的地址空间里，通常会导致应用程序崩溃。IA-32 架构几乎完全向后兼容 8086 处理器，寄存器的使用上也反映出

更多资源请点击访问稀.酷客(www.ckook.com)

这一点。我们可以访问所有通用寄存器的 32 位内容，也可以访问它的低 16 位，更甚之，像 EAX，EBX，ECX 和 EDX 寄存器的低 16 位又可以分为高 8 位及低 8 位而分别访问之。图 2-1 中使用的名字反映了这种情况。例如，为了访问 EAX 寄存器的低 8 位，需要用 AL 替换指令中的 EAX；为了访问 EBP 寄存器的低 16 位，需要用 BP 替换 EBP；为了访问 EDX 寄存器低 16 位中的高 8 位，需要用 DH 替换 EDX。除了通用寄存器外，不要忘了还有指令指针——EIP。EIP 寄存器指向处理器将要执行的下一条指令，在几乎所有的基于应用程序的攻击中，目标都是控制这个寄存器。然而，它和通用寄存器不一样，我们并不能直接修改它。也就是说，你并不能通过执行一条指令修改其中的值，但是你可以通过执行一组操作来间接地修改它的值。例如，在压入栈段后紧接着执行 `ret` 指令。如果你不理解这些叙述，不要担心，我们稍后就会介绍刚才提到的指令以及栈段的概念。现在，你只需知道不能直接修改指令指针的值就可以了。

32-bit	16-bit	31–16	15–8	7–0
EAX	AX		AH	AL
EBX	BX		BH	BL
ECX	CX		CH	CL
EDX	DX		DH	DL
ESI			SI	
EBP			BP	
ESP			SP	
EDI			DI	

图 2-1 通用寄存器

除了 EIP 寄存器和通用寄存器，还有 6 个 16 位的段寄存器：代码段（CS）、数据段（DS）、栈段（SS）、额外段（ES）、FS 及 GS。后 3 个是额外的通用目的段。段寄存器包含我们称之为段选择子的指针，通常以偏移量的基址形式出现。例如，看下面的指令：

```
mov DS:[eax], ebx
```

在这条指令中，EBX 寄存器中的内容被复制到 EAX 指定的数据段里的一个偏移地址。这个地址也可以解释成“DS 的地址加上 EAX 的值”。段选择子是 16 位的段标识符，也就是说段选择子不直接指向段，而是指向定义段的段描述符。因此，段寄存器指向段选择子，用于确定 8 192 个可能的标识段的段描述符中的一个。没把你绕糊涂吧？

段选择子的结构相对比较简单。3 至 15 位用作描述符表的索引（三个内存管理寄存器之一），第 2 位指定正确的描述符表，低 2 位指定请求的特权级（从 0 到 3——本章后面将讨论特权级）。段描述符非常有趣，对 OS 设计而言也非常重要，但为了确保本章中相关内容的唯一性，我们就不过多介绍了。当然，我们鼓励有兴趣的读者把 Intel 开发者手册找来读一读。此外，EFLAGS 寄存器也很重要，它包含了各种标志，指示前一条指令执行后的状态、情形、当前的特权级，以及是否允许中断等内容。在上下文里，如果我们没有领会使用它的那些指令，EFLAGS 寄存器对我们而言就没什么意义。我们稍后会完整介绍它。

到现在为止，我们介绍了常量及寄存器操作数，接下来讨论内存操作数。尽管我们对内存操

更多资源请点击访问稀.酷客(www.ckook.com)

作数的描述比较有限，但相比较而言它要复杂一些。内存操作数基本上相当于高级语言程序员眼中的变量。也就是说，当你在像C或C++这样的语言里声明一个变量，它基本上就会以内存操作数的形式出现在内存中。在程序中通常以指针的形式访问它们，如果取消了指向它们的指针，则需要通过寄存器或直接从内存访问。这个概念本身很简单，但要真正理解它还需要有扎实基本功，比如说了解内存是怎样寻址的，而这又依赖于内存模型、操作模式以及使用的特权级。这为接下来要介绍的操作模式起了个好头。

在IA-32中，有3种基本的操作模式以及1种伪操作模式。分别是保护模式、实地址模式、系统管理模式，以及是保护模式子集的伪模式（称为虚拟8086模式）。本着从简的原则，我们将着重讨论保护模式。各种操作模式之间最大的不同是它们修改指令或体系结构特征的表示。例如，RM（Real Mode，实模式）意味着向后兼容，在RM里，只支持实地址模式内存模型。这里要着重注意的是（除非你打算逆向分析古老的DOS应用程序或类似的东西），重启或冷启动IA-32机器时，它肯定处于实模式。SMM（System Management Mode，系统管理模式），自80386以来就出现在Intel架构里了，常用于电源管理、系统硬件控制等地方。它基本上会阻塞所有其他的操作，并切换到新的地址空间。不过一般而言，你碰到和使用的大都是保护模式。

Intel在80286处理器中引入了保护模式，并在80386中进一步完善，体现了Intel架构质的飞跃。在此之前的关键问题是，这些老的处理器只支持一种操作模式，没有在硬件上强制保护指令和内存。这不仅给恶意操作者肆意妄为的机会，也加大了因缺陷的程序出错而导致整个系统崩溃的机率；因此，它是可靠性与安全性的关键所在。这些老处理器另外的问题是640KB限制；不过，这些在PM中早已不存在了。此外，它还有另外一些好处，例如，从硬件上支持多任务，修改了中断的处理方式等。286和386代表了个人计算机的巨大成就。

无论是在较早的8086/80186处理器时代，还是现代处理器位于实地址模式的今天，段寄存器都是表示线性地址的高16位，而在保护模式里，选择子是描述表里的索引。此外，就像前面提过的，以前的CPU没有内存保护或指令限制；但在保护模式里，有四个称为环（ring）的特权级。一般用0至3表示它们，数越小表示特权级越高。环0一般是操作系统使用，而应用程序一般运行在环3，这将通过中止恶意程序运行来防止它们修改操作系统的数据结构和对象，也限制应用程序可以运行的指令（如果环3应用程序可以切换其特权级，有什么好处呢？）。在IA-32里，在三个地方可以找到特权级：CS寄存器的低2位——CPL（Current Privilege Level，当前特权级），段描述符的低2位——DPL（Descriptor Privilege Level，描述符特权级），段选择子的低2位——RPL（Requestor's Privilege Level，请求特权级）。CPL是当前正在执行代码的特权级，DPL是给定描述符的特权级，而RPL则显然是创建段的代码的特权级。

特权级限制了对系统数据的可信组件的访问，例如，环3应用程序就不能访问环2组件的数据，不过，环2组件可以访问环3组件的数据。这就是你为什么不能从Windows或Linux内核里任意读取数据的原因了，但它们可以读你的数据。特权级分离的另一种作用是它对执行转移控制进行检查。程序请求从当前段改变到其他段时，将引发系统对其进行检查，确保CPL和段的DPL是一样的。间接执行转移通过诸如调用门（call gates，后面会作简短的介绍）之类的机制进行。

更多资源请点击访问稀.酷客(www.ckook.com)

最后，特权级会限制程序访问某些指令，比如说那些从根本上改变操作系统环境或为操作系统保留的操作（诸如读写串口）。

保护模式下有三个不同的内存模型：平坦、分段及实地址。实地址模式常用于启动时且保持向后兼容，不过，你很有可能从来不会（或很少）碰到它，因此，本章不准备讨论它。平坦内存模型和它名字表述含义差不多：它是平坦的（参见图 2-2）！这意味着系统内存看起来像一片连续的地址空间，通常是地址 0~4294967296。这里的地址空间被称为线性地址空间，任何单独的地

更多资源请点击访问稀.酷客(www.ckook.com)

或应用程序中的表现形式。在这两种情形下，数据仍是以线性的方式保存，但数据的视图改变了。对分段内存来说，不是以线性地址访问内存，而是使用了逻辑地址（也称为 far 指针）。逻辑地址是基址——保存在段选择子里——加上偏移量的结合。两者（基址及偏移量）相加后对应段里的一个地址，然后映射到线性地址空间里。这样做可以保证高度分离——由处理器强制执行，确保一个进程不会跑到另一个进程的地址空间里（不过，在平坦内存模型里，由操作系统做同样的事情）。因此，基址加上偏移量等于处理器地址空间里的线性地址。此外，除了每个应用程序被给定它自己的段集及段描述符外，多段模型也可以用于使用单一分段模型的地方。不过目前我想不起来还有哪个操作系统在使用它，因此，进一步讨论它怎样工作也没什么意义。

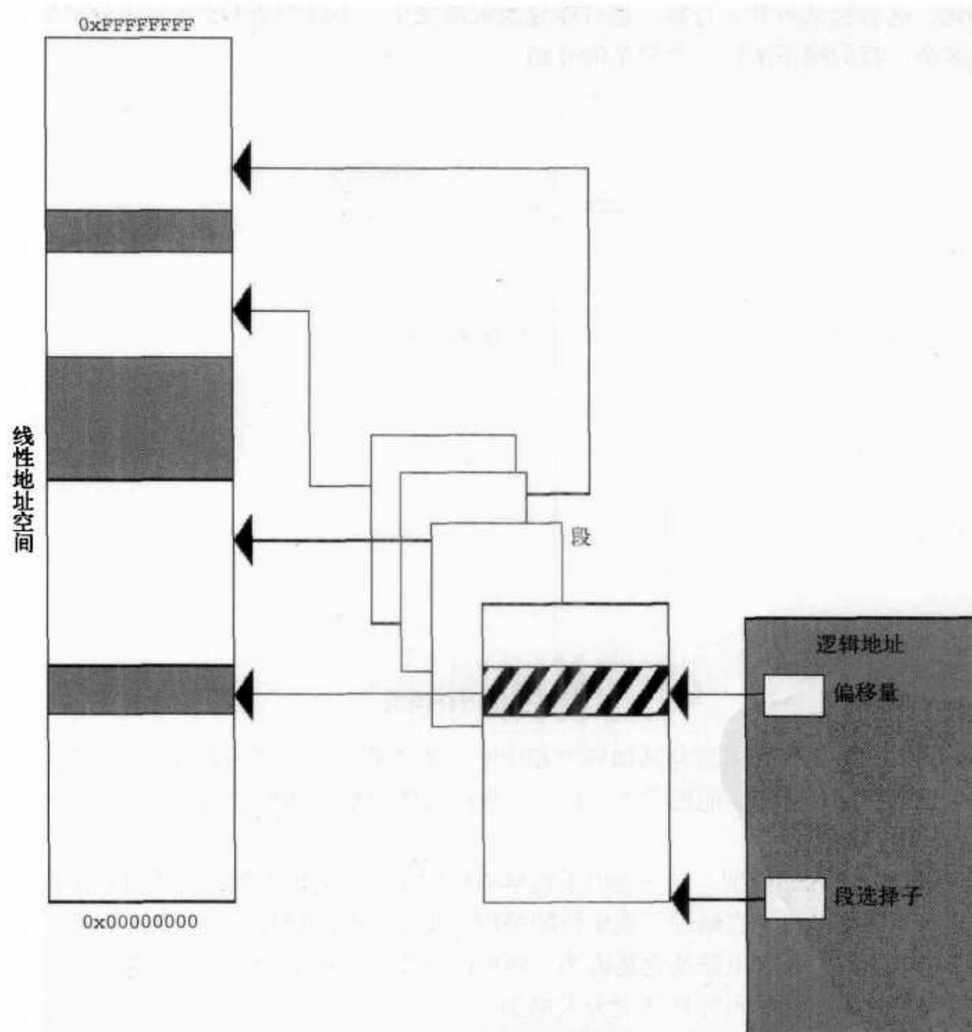


图 2-3　分段内存模型

更多资源请点击访问稀.酷客(www.ckook.com)

破坏与防护

分段的安全性

我们前面提到过，分段内存模型最近在一些团体中的吸引力有所回升，特别是grsecurity(http:// grsecurity.com/)和 PaX(http://pax.grsecurity.net/)——第三方 Linux 内核补丁，为 Vanilla 内核提供卓越的安全性。grsecurity 的首席开发者 Brad Spengler 演示了平坦内存模型带来的不安全性——被认为是第一个可利用的 Linux 内核 NULL 指针解除索引漏洞，在下面的 URL 里可以了解详情：http://marc.info/_?l=dailydave&m=117294179528847&w=2。此外，PaX 的匿名作者已经实现了称为 UDEREF 的特性，此特性企图阻止意外解除内核里由用户空间指针提供的指针索引（从而导致潜在的可利用条件）。已有人撰文介绍它，建议有兴趣的读者找出来读一读，进一步理解 UDEREF 对平坦内存模型所做的安全改进。在此刻写作之时，在http://grsecurity. net/~spender/uderef.txt 可以找到这篇文章。

现代操作系统能使用的地址空间往往比物理内存更大一些，而寻址的数据并不一定保存在物理内存里。这是如何做到的呢？答案是分页和虚拟内存—— 一个现代计算的基础术语，经常会被一些有很多操作经验的人们所误解。不难碰到能理解给定的应用程序可以访问 4GB 内存的人，但他们并不理解仅仅给他们 1GB 或 2GB 物理内存时，程序怎么工作。

简而言之，分页利用了这样的事实：只有当前必需的数据才需要随时保存在物理内存里。它把需要的数据保存在物理内存里，把暂时用不上的数据保存到硬盘上。把数据从磁盘读入内存的过程，或向磁盘写数据，被称为交换数据，这就不难理解 Windows 为什么会有交换文件，而 Linux 有交换分区。当不启用分页时，线性地址（不管它是否由 far 指针组成）与物理地址一一对应，它和使用的地址空间之间没有转换。然而，当启用分页时，应用程序使用的所有指针都是虚拟地址。（这就是为什么在保护模型里平坦内存模型中的两个应用程序使用分页访问完全一样的地址却不会彼此破坏。）这些虚拟地址与物理内存地址间不存在一一对应的关系。

使用分页技术时，处理器把物理内存分成 4KB、2MB 或 4MB 大小的页。地址转换成线性地址时，将通过分页机制查找它，如果这个地址不在当前的物理内存里，将抛出 page-fault（页面故障）异常，操作系统收到此异常后，将把指定的页面加载到内存里，然后接着执行刚才导致此异常的指令。

把虚拟地址转换成物理地址的过程依赖使用的页面大小，但基本概念是一样的。线性地址分成 2 个或 3 个部分。首先用 PDBR（Page Directory Base Register，页面目录基址寄存器）或 CR3（Control Register 3，控制寄存器 3）查找 Page Directory（页面目录）。在这之后，线性地址里的第 22~31 位将作为 Page Directory 里的偏移量，标识使用的 Page Table（页面表）。（参见图 2-4）一旦找到 Page Table，将用第 12~21 位查找 PTE（Page Table Entry，页面表项）（标识使用的内存

更多资源请点击访问稀.酷客(www.ckook.com)

页)。最后，线性地址的第 0~11 位用作在页面里定位请求的数据的偏移量。当使用其他页面时，除了省略一个间接层外，过程几乎是一样的；在这种过程中，目录项直接指向页面、页面表且 PTE 被完全省略了。PDE（Page Directory Entries，页面目录项）和 PTE 的内容对我们来说不是很重要。如果你在工作时碰到它很重要的情况，或者你很好奇它的原理，请参考处理器的文档。

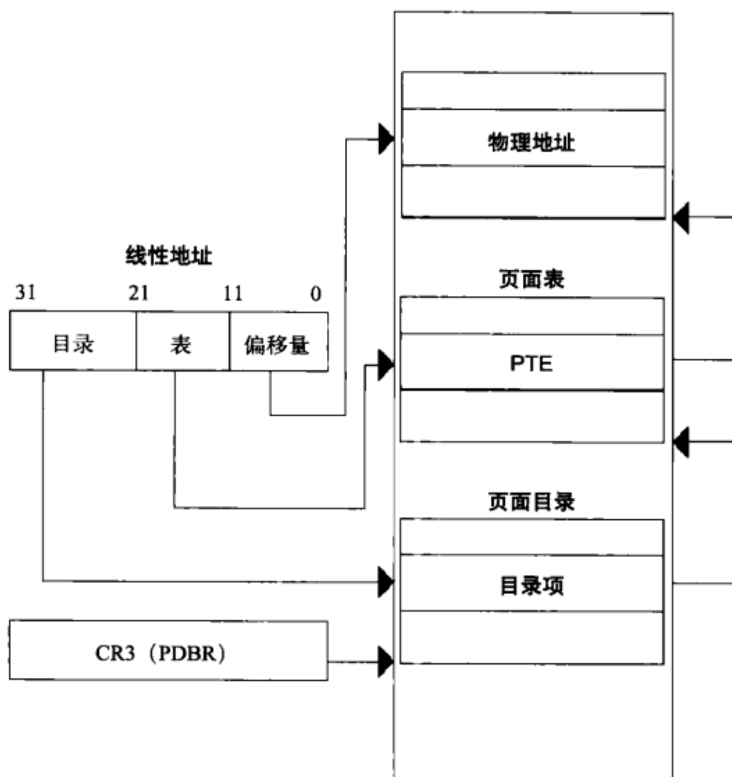


图 2-4　4KB 地址转换

现在，我们对指令、操作数、内存模型、操作模式等概念有了一些理解，可以继续学习其他内容了。全书及本章稍后将使用的大多数术语都已定义了，因此，当你阅读全书时，如果感到有些内容还不理解，可以再回到这里看看。

2.3　栈、堆及二进制可执行文件中的其他区段

前一节介绍了段、段寄存器、段描述符及段选择子，但我们并没有真正深入研究它们包含的数据。而理解这些部分对理解二进制可执行文件的布局相当重要。在本节，我们将尽量深入地讨论这些概念。当然，因为篇幅的关系，有些内容——例如堆，我们并没有对实现细节进行深入分析。碰到这种情形，我们通常会提供一般性的介绍，而对细节的理解就由读者自己练习了。

更多资源请点击访问稀.酷客(www.ckook.com)

警告 读者应该理解，这里所定义的区和段与分段内存模型并没有什么关系。在内存模型里，应用程序和操作系统所享受的待遇是不一样的，应该说，应用程序不知道实现细节是幸运的。此外，在本章，术语段和区是可以互换使用的。

我们在前面曾讨论过段寄存器，特别是 CS、DS 及 SS 段寄存器，但我们并没有介绍代码段、数据段、栈段是些什么。在传统的设计里，应用程序有一些基本的组成部分（和一些与具体实践有关的部分），比如说代码段（或 text 段，或简单地记成 .text）、数据段（通常简单地记成 .data），以符号开始的块（BSS/.bss）段、栈段及堆段等。我们来看一个例子，下面的 C 代码将演示各种段之间的不同点：

```
unsigned int variable_zero;
unsigned int variable_one = 0x1234;
int
main(void)
{
        void*variable_two = malloc(0x1234);
        [...]
}
```

在这个例子里，我们定义了 3 个变量和 1 个函数。第一个变量被命名为 variable_zero，是一个全局变量，未被初始化。在这个例子里，C 编译器将为其在二进制文件里分配空间并填充零，它将位于二进制区段里的 BSS 区段。名为 variable_one 的变量是另一个全局变量。不过，在这个例子里它被初始化为 0x1234。在这个例子里，编译器将在二进制文件里为其预分配空间，把数值保存在数据段里。之后是函数 main。main 明显是一个函数，因此它位于代码段里。在 main 之后我们发现了称为 variable_two 的变量，它给我们出了个难题：我们有一个指针（它的作用域是 main 之内和它指向的内存地址）。这个指针本身是函数的一部分，是动态分配的，存在于栈段之中，与函数的生命周期相同；malloc()返回的指针存在于堆上，是动态分配的，具有全局作用域及“use-until-free()”生命周期。此外，也常见一些其他的区段，例如，GCC（GNU Compiler Collection，GNU 编译器套装）编译生成的程序里，在源文件里声明的常量字符串通常保存在名为.rodata 的区段里，而只读数据可能会保存在代码段里。

栈是一种更重要的区段，在应用程序的例行操作中扮演着非常重要的角色。计算机科班出身的人肯定都知道栈是什么以及它是怎样工作的。但出于完整性的考虑，我们在此还是要大致介绍一下。栈是一种简单的数据结构，栈数据彼此之间紧挨着，元素以 LIFO（Last-In First-Out，后进先出）的方式增加和移除。当你向栈上增加数据时，是把它压入栈；当你从栈上移除数据时，则是从栈上弹出它。（参见图 2-5）在大多数计算平台上，栈之所以如此重要有两个原因。第一个原因是所有的自动或局部变量都保存在它上面，也就是说，当一个函数被调用时，任何它声明的不是静态或类似的局部变量都会在栈上分配相应的空间。这通常通过把当前栈指针加或减一个数

更多资源请点击访问稀.酷客(www.ckook.com)

值来实现。栈指针是 ESP 寄存器，通常指向栈段的顶端，或者更严格地说 SS:ESP 指向栈段的顶端。栈的顶端是栈使用地址中的最小值——之所以是最小值，是因为栈在 IA-32 上是向下增长的。栈的底部通常也是有限制的，而不是我们想象中的绝对的底，它一般是当前栈帧的底，由 EBP 寄存器指向。

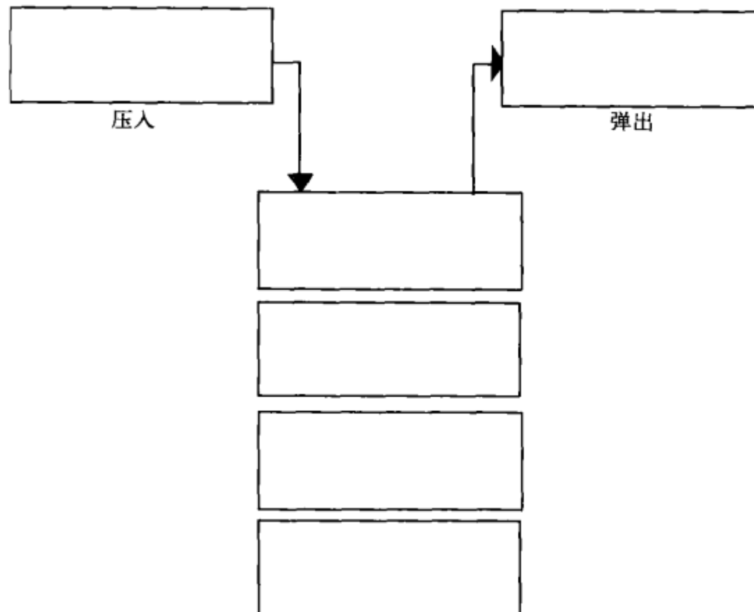


图 2-5　栈

栈帧指的是与当前正在执行的函数有关的当前视图；当处理器进入新过程时，会执行我们称之为预处理例程（procedure prologue）的步骤（参见图 2-6）。预处理执行如下操作：首先把调用帧里下一条指令的地址压入栈，接下来把当前栈的基址（EBP）保存到栈上，然后把 ESP 寄存器复制到 EBP 寄存器里，最后，把 ESP 寄存器里的值减去一部分，从而为函数里的变量分配空间。当函数被调用时，函数的参数以逆序压入栈（或者说最后的先压入）。预处理对应汇编指令如下：

```
push ebp
mov ebp, esp
sub esp, 0x1234
```

看到这里，我们肯定会奇怪这里为什么没有保存返回地址，或我们在调用例程里继续执行的地址。例如，假如有下面这样的 C 代码：

```
A();
B();
```

更多资源请点击访问稀.酷客(www.ckook.com)

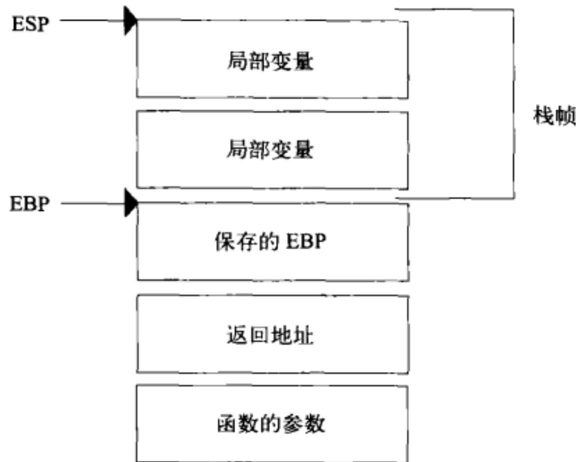


图 2-6　栈帧

当进入函数 A()时返回地址将是指令 B()的地址。这么说理解起来可能有些困难，特别是我们并没有看到把地址保存到栈上的指令，但其实调用指令背地里帮我们做了这件事，稍后将会讨论。函数除了预处理例程外，还有对应的扫尾例程（procedure epilogue）。扫尾例程所做的基本上是恢复预处理所做的改变，包括调大栈指针的值（作用是取消分配的局部变量），调用 leave 和 ret 指令（作用是移去保存的帧指针和返回地址，并把执行流返回给调用函数）。因此，栈的要点可以总结如下：

- 在 IA-32 平台上，栈向着较小的地址增长；
- 栈以 LIFO 的顺序移去或增加数据；
- 局部变量及只存在于函数生命周期的变量随着栈的取消而结束；
- 每个函数都会有包含局部变量的栈帧（除非是编译器故意忽略了）；
- 在每个函数的栈帧之前有保存的帧指针、返回地址和例程的参数；
- 栈帧在预处理期间构造，在扫尾例程执行期间取消。

堆是另一种重要的数据结构，不是说处理器对它有多依赖，而是因为它被使用的频率很高。堆只是内存中一个区段（主要供那些动态分配的变量使用，这些变量需存在于当前栈帧之外）。由此，大多数对象和应用程序使用的大量数据都保存在堆上。堆通常是随机映射的，或（在更经典的例子里）是数据段的动态扩展（尽管 DS 几乎不会指向堆）。从此种意义上说，处理器是不知道这些细节的。此外，操作系统也不太了解用户区的堆，它只在需要时尽可能地分给应用程序更多的内存，否则操作失败。它是典型的提供堆操作并定义相应语义的 libc 或类似的东西。

特别是在初始化时，堆将向操作系统请求尽可能大的内存区段，然后按应用程序的需求分给小内存块。这些块包含内联的元数据，指示块大小及其他元素，例如前一块内存的大小。

通过指向给定块的指针，再加上它（给定块）的大小就可以找到下一个块，或者通过把块开头地址减去前一个块的大小来发现前一个块。例如，在图 2-7 里你会看到 Glibc 分配的块。在这

更多资源请点击访问稀.酷客(www.ckook.com)

个例子里，标记为 mem 的指针指示数据的开始（由 malloc()或类似方式返回给 API 用户），而标记为 chunk 的指针则是实际块的开始。在这里，我们发现了包含前一个块的大小、当前块的大小，以及指示各种状态条件的元数据。这个块（通常是 Linux 使用）与大多数操作系统及动态内存分配实现（当然，还是有一些区别）所用的块是相似的。因为从操作系统获取大内存块或扩展数据段的大小需要花费相当多的操作，因此，通常会维护分类的 cache（高速缓冲存储器）。这个 cache 通常以指向前面 free()的内存块的指针链表形式出现。这个链表一般会很复杂，相邻的空闲块会合并在一起，从而减少碎片。通过大小或其他特性对链表进行排序，从而在请求发生时以最有效的手段尽可能定位候选的内存块。

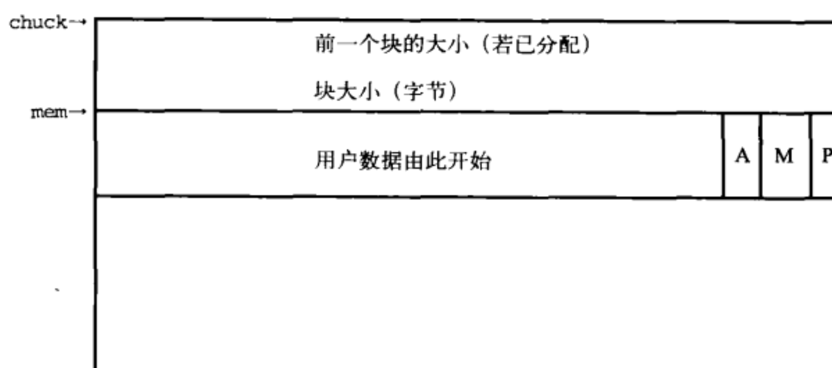


图 2-7　Glibc 分配 chunk

为了和前一个例子保持一致，我们在图 2-8 里提供了用 Glibc 表示的空闲内存块。在这个例子里，标为 mem 的指针指示返回给 API 用户的指针通常所处的位置，标为 chunk 的指针指向使用的物理数据结构的开头。两者最大的不同之处是使用过的用户数据中保存着两个指针，一个指向链表里下一个空闲的内存块，另一个指向链表里前一个内存块。这暗示，不像已分配的块是通过大小来遍历的，空闲的内存块直接通过链表遍历。虽然结构列表的细节是关于 Glibc 的，但概念本身在大多数实现中是雷同的。因此，在普通应用程序生命周期里频繁发生分配和释放操作时，把原来从操作系统里获取的内存块取出来，返回给空闲列表，如果可能的话，当再有分配请求时，将会利用空闲列表里的内存块，如此这般，直到所有的内存都被使用或应用程序结束。

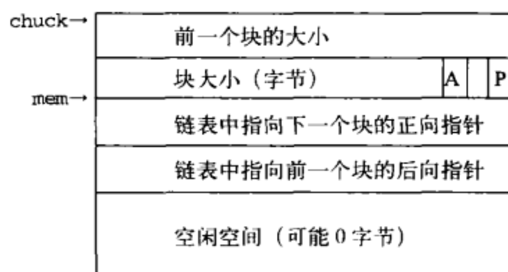


图 2-8　Glibc 空闲块

更多资源请点击访问稀.酷客(www.ckook.com)

本节，我们讨论了二进制可执行文件中常见的区段，以及它们的作用，深入探讨了栈段与堆段，并在一定程度上讨论了它们的工作原理。这对于本书的学习已经够用了，但我们仍建议感兴趣的读者仔细阅读相关的参考文档。

2.4 最新的IA-32指令集及参考资料

我们在前面的几节中简单地讨论了指令及操作数，着重讨论了IA-32的架构设计，之后又深入探讨了二进制可执行文件内存的常见布局及它们的用途与使用方式。在这一节，我们将介绍一些经常使用的指令，及其使用方式和操作数。如果你已经读过Intel开发者手册，可以直接跳到下一章。在本节，我们将使用如表2-1所列的术语。

表2-1 所用术语

术　语	意　义
Reg32	32位寄存器
Reg16	16位寄存器
Reg8	8位寄存器
Mem32	32位内存操作数
Mem16	16位内存操作数
Mem8	8位内存操作数
Sreg	段寄存器
Memoffs8	8位内存偏移量
Memoffs16	16位内存偏移量
Memoffs32	32位内存偏移量
Imm8	8位立即数(常量)
Imm16	16位立即数
Imm32	32位立即数
ptr16:16	操作数给出的绝对地址
ptr16:32	操作数给出的绝对地址
mem16:16	以mem16:16形式给出的绝对间接地址
mem16:32	以mem16:32形式给出的绝对间接地址
rel8	8位相对寻址
rel16	16位相对寻址
rel32	32位相对寻址
寄存器名	引入的寄存器名

我们准备介绍的第一条指令是mov，它非常常见，其作用是把一个操作数复制到另一个操作数中。它可以使用的格式及允许的操作数如表2-2所列。

更多资源请点击访问稀.酷客(www.ckook.com)

表2-2 mov指令

目的操作数	源操作数	目的操作数	源操作数
reg8/mem8	reg8	EAX	memoffs32
reg16/mem16	reg16	memoffs8	AL
reg32/mem32	reg32	memoffs16	AX
reg8	reg8/mem8	memoffs32	EAX
reg16	reg16/mem16	reg8	imm8
reg32	reg32/mem32	reg16	imm16
reg16/mem16	Sreg	reg32	imm32
Sreg	reg16/mem16	reg8/mem8	imm8
AL	memoffs8	reg16/mem16	imm16
AX	memoffs16	reg32/mem32	imm32

mov 把源操作数复制到目的操作数，当然，它只能复制特定类型的操作数。例如，不能用它设置代码段，不能用它修改 EIP 寄存器。如果目的操作数是 Sreg 类型，它还必须指向一个有效的段选择子。接下来介绍几个位操作指令，例如 and 和 xor。

and 指令也很简单。它把目的操作数与源操作数按位做 AND 操作，并把结果保存在目的操作数里。它支持的操作数如表 2-3 所示。按位 AND 操作比较两个操作数的二进制表示，如果两个位都是 1 则输出 1，否则输出 0。

表2-3 and指令

目的操作数	源操作数	目的操作数	源操作数
reg8/mem8	reg8	EAX	imm32
reg16/mem16	reg16	reg8	imm8
reg32/mem32	reg32	reg16	imm16
reg8	reg8/mem8	reg32	imm32
reg16	reg16/mem16	reg8/mem8	imm8
reg32	reg32/mem32	reg16/mem16	imm16
AL	imm8	reg32/mem32	imm32
AX	imm16		

接下来介绍 not 指令——另一个很简单但也经常使用的指令，它逐位对单一操作数做 NOT 操作，它允许的操作数如表 2-4 所示。它简单地按位把 1 设成 0，反之亦然。

表2-4 not指令

目的操作数	源操作数
reg8/mem8	N/A
reg16/mem16	N/A
reg32/mem32	N/A

更多资源请点击访问稀.酷客(www.ckook.com)

再来看 or 指令，它把操作数逐位做 OR 操作，它能接受的操作数如表 2-5 所示。位操作 OR 很简单且经常用到。它按位比较两个操作数，当被比较的两个位都为 0 时它才输出 0，否则输出 1。

表2-5 or指令

目的操作数	源操作数	目的操作数	源操作数
reg8/mem8	reg8	EAX	imm32
reg16/mem16	reg16	reg8	imm8
reg32/mem32	reg32	reg16	imm16
reg8	reg8/mem8	reg32	imm32
reg16	reg16/mem16	reg8/mem8	imm8
reg32	reg32/mem32	reg16/mem16	imm16
AL	imm8	reg32/mem32	imm32
AX	imm16		

我们再来看看 xor 指令。它逐位对操作数做 XOR 操作，它接受的操作数如表 2-6 所示。xor 指令逐位比较源操作数与目的操作数，并把结果保存在目的操作数里。如果被比较的两个位不一样才输出 1，否则输出 0。

表2-6 xor指令

目的操作数	源操作数	目的操作数	源操作数
reg8/mem8	reg8	EAX	imm32
reg16/mem16	reg16	reg8	imm8
reg32/mem32	reg32	reg16	imm16
reg8	reg8/mem8	reg32	imm32
reg16	reg16/mem16	reg8/mem8	imm8
reg32	reg32/mem32	reg16/mem16	imm16
AL	imm8	reg32/mem32	imm32
AX	imm16		

test 指令常用于确定特殊的条件，并根据结果改变控制流（参见表 2-7）。test 指令逐位对第一个与第二个操作数做 AND 操作，根据操作结果设置 EFLAGS 寄存器中的标志。注意，此指令并不保存操作结果。

表2-7 test指令

目的操作数	源操作数	目的操作数	源操作数
reg8/mem8	reg8	reg8	imm8
reg16/mem16	reg16	reg16	imm16
reg32/mem32	reg32	reg32	imm32
AL	imm8	reg8/mem8	imm8
AX	imm16	reg16/mem16	imm16
EAX	imm32	reg32/mem32	imm32

更多资源请点击访问稀.酷客(www.ckook.com)

cmp 指令用于比较两个操作数的大小，它用目的操作数减去源操作数，并根据结果设置 EFLAGS 寄存器中的标志（参见表 2-8）。它的使用方式与 test 类似，主要用于比较像用户输入、例程中的返回值等数值。如果一个操作数是立即数，将把它做符号扩展，从而使之匹配另一个操作数的大小。

表2-8　cmp指令

目的操作数	源操作数	目的操作数	源操作数
reg8/mem8	reg8	EAX	imm32
reg16/mem16	reg16	reg8	imm8
reg32/mem32	reg32	reg16	imm16
reg8	reg8/mem8	reg32	imm32
reg16	reg16/mem16	reg8/mem8	imm8
reg32	reg32/mem32	reg16/mem16	imm16
AL	imm8	reg32/mem32	imm32
AX	imm16		

lea（Load Effective Address，取有效地址指令）指令计算源操作数指定的地址，并把结果保存在目的操作数中（参见表 2-9）。它也用于在多个寄存器间做算术运算，且不修改源操作数。

表2-9　lea指令

目的操作数	源操作数
reg8	mem8
reg16	mem16
reg32	mem32

jmp 指令把执行控制权交给它的操作数。它可以执行 4 种不同类型的跳转：近跳转，短跳转，远跳转及任务切换（参见表 2-10）。近跳转的目的地址只能在当前代码段里；短跳转可以跳到与当前地址相距-128 到 127 的地址内；远跳转可以跳到地址空间里任何一个段里，前提是它们有与当前代码段同样的特权级；任务切换跳转则可以跳到不同的任务里。

表2-10　jmp指令

目的操作数	源操作数	目的操作数	源操作数
rel8	N/A	ptr16:16	N/A
rel16	N/A	ptr16:32	N/A
rel32	N/A	mem16:16	N/A
reg16/mem16	N/A	mem16:32	N/A
reg32/mem32	N/A		

jcc 不是指一条指令，而是一系列条件跳转指令的统称。具体的条件随着使用的指令会有所变化，但基本上都是配合 test 与 cmp 指令使用的。表 2-11 显示了 jcc 指令使用的目的操作数。在表 2-12 里你还会发现条件跳转的列表和标志（用于确定条件是否为真）。不了解标志的意思？不必担心，我们稍后就会介绍 EFLAGS。

更多资源请点击访问稀.酷客(www.ckook.com)

表2-11 jcc指令

目的操作数	源操作数
rel8	N/A
rel16	N/A
rel32	N/A

表2-12 条件跳转寄存器

指令	标志条件	描述
ja	CF = 0 && ZF = 0	大于则跳转
jae	CF = 0	大于或等于时跳转
jb	CF = 1	小于则跳转
jbe	CF = 1 || ZF = 1	小于或等于时跳转
jc	CF = 1	有进位时跳转
jcxz	CX = 0	CX 为0时跳转
jecxz	ECX = 0	ECX为0时跳转
je	ZF = 1	相等则跳转
jg	ZF = 0 && SF = OF	大于则跳转
jge	SF = OF	大于等于时跳转
jl	SF != OF	小于时跳转
jle	ZF = 1 || SF != OF	小于等于时跳转
jna	CF = 1 || ZF = 1	不大于则跳转
jnae	CF = 1	不大于或等于时跳转
jnb	CF = 0	不小于时跳转
jnbe	CF = 0 && ZF = 0	不小于或等于时跳转
jnc	CF = 0	无进位跳转
jne	ZF = 0	不相等跳转
jng	ZF = 1 || SF != OF	不大于则跳转
jnge	SF != OF	不大于等于时跳转
jnl	SF = OF	不小于时跳转
jnle	ZF = 0 && SF = OF	大于则跳转
jno	OF = 0	不溢出跳转
jnp	PF = 0	非奇偶跳转
jns	SF = 0	无符号跳转
jnz	ZF = 0	不为0跳转
jo	OF = 1	溢出跳转
p	PF = 1	奇偶跳转
jpe	PF = 1	奇偶性为偶数时跳转
jpo	PF = 0	奇偶性为奇数时跳转
js	SF = 1	有符号跳转
jz	ZF = 1	为0跳转

更多资源请点击访问稀.酷客(www.ckook.com)

像你在表 2-12 里所看到那样，大多数条件跳转指令都是根据 EFLAGS 寄存器的状态决定后续操作的，因此，有必要介绍一下 EFLAGS。EFLAGS 寄存器是 32 位寄存器，包含一组状态、系统标志及控制标志。每个标志由寄存器里一位代表，从 0 位到 31 位我们有下面这些标志。

CF：进位标志，指示在算术运算中是否有进位或借位。用于无符号算数运算。

PF：奇偶标志，为机器中传送信息时可能出错提供校验。当目的操作数中 1 的个数为偶数时置 1（PE），否则置 0（PO）。

AF：辅助进位标志，记录运算时低 4 位（半个字节）产生的进位值。有进位时置 1（AC），否则置 0（NA）。

ZF：零标志。运算结果为 0 时置 1（ZR），否则置 0（NZ）。

SF：符号标志，记录运算结果的符号。结果为负时置 1（NG），否则置 0（PL）。

TF：陷阱标志，用于单步方式操作。当 TF 为 1 时，每条指令执行完后产生陷阱，由系统控制计算机；当 IF 为 0 时，CPU 正常工作，不产生陷阱。

IF：允许中断标志。当 IF 为 1（EI）时，允许中断；IF 为 0（DI）时关闭中断。

DF：方向标志，在串处理指令中控制处理信息的方向。当 DF 置 1（DN）时每次操作后，变址寄存器 SI 和 DI 减量，这样就使串处理从高地址向低地址方向处理；当 DF 置 0（UP）时，则反之。

OF：溢出标志。在运算过程中，若操作数超出了机器能表示的范围则称为溢出，此时 OF 标志位为 1(OV)；否则置 0(NV)。

IOPL（12 位到 13 位）：I/O 特权级标志。指出当前运行任务的 I/O 端口的特权级。

NF：嵌套任务标志。只在当前任务是前一任务的子任务时设置。

RF：恢复标志。控制处理器对调试异常的响应。

VM：虚拟 8086 标志。控制是否启用虚拟 8086 模式。

AC：对齐检查标志。设置为启用存储器的对齐检查的参考。

VIF：虚拟中断标志。IF 的虚拟映像，与 VIP 标志联合使用。

VIP：虚拟中断标志。确定是否有中断被挂起。

ID：标识标志，确定 CUP 是否支持 CUPID 指令。

第 22 到 31 位当前被保留。

call 指令有点像更正式一些的跳转指令；它像前面描述的那样设置栈，使处理器在被调用的函数执行完成后可以恢复执行。（参见表 2-13）

表2-13 call 指令

目的操作数	源操作数	目的操作数	源操作数
rel16	N/A	ptr16:16	N/A
rel32	N/A	ptr16:32	N/A
reg16/mem16	N/A	mem16:16	N/A
reg32/mem32	N/A	mem16:32	N/A

更多资源请点击访问稀.酷客(www.ckook.com)

ret 指令是 call 指令的搭档。它从栈里获取保存的元数据，弹出元数据并返回那个地址（参见 2-14）。它后面带的立即操作数指定了在执行返回后从栈上弹出多少个字节。

表2-14 ret指令

目的操作数	源操作数
N/A	N/A
imm16	N/A

2

像你看到的那样，我们已经介绍了许多指令，而且它们中的大多数还接受各种形式的操作数，从而导致即使是同一条指令也会有不同的形式（对应不同的操作码）。我们希望论述一些常用指令的基础知识，至少让大家熟悉它们。除了这些指令之外，还有很多指令，如果还不熟悉它们的话，建议你阅读 Intel 开发者手册，特别是卷 3A 及 3B。

2.5 小结

现在，你至少对汇编及 Intel 架构有所了解了，并在一定程度上了解了内存模型及操作模式的工作方式。然而，你可能会想："是的，我现在有些了解汇编了，但什么是逆向工程呢？"嗯，逆向工程是个广义的术语，对不同的人有不同的意思。就一般意义而言，它是指把一个不可读、不可分析的应用程序通过一系列的处理，使它可读、可分析的过程。逆向工程的目的有很多种，一些人是为了找回丢失的源代码，一些人通过它复制具有所有权的产品，还有一些人通过逆向分析恶意程序来了解它们做些什么，当然，更多的是通过逆向工程找出软件中存在的 bug 和不安全因素。

简而言之，对本书来讲，逆向工程就是把计算机可读的二进制文件（利用操作码）转换成汇编语言，然后通过阅读汇编指令来达成目标。从这种意义上，可以说对逆向工程师而言没有闭源软件——因为处理器可以看见每一条指令，逆向工程师也可以。

更多资源请点击访问稀.酷客(www.ckook.com)

第 3 章

可移植可执行文件格式和可执行链接格式

本章内容：

- 可移植可执行文件格式
- 可执行链接格式

☑ 小结

更多资源请点击访问稀.酷客(www.ckook.com)

3.1　导言

在本章，我们将介绍两种常见的二进制可执行文件格式：PE（Portable Executable，可移植可执行文件格式）和ELF(Executable and Linkable Format，可执行链接格式)。PE主要用于Windows环境，而ELF主要在Unix中使用，ELF替代了之前的不支持共享库标准的a.out格式。值得一提的是，PE是COFF格式（用在较早的Unix中）的改进版。作者个人认为，这跟微软雇用了许多以前为Unix做开发的程序员不无关系。

本章将介绍文件的实体结构，以及逆向工程师感兴趣的和重要的一些细节。这两种文件格式都有开放的文档，如果有些内容本章未曾涉及，建议有兴趣的读者把它们的规范找来读一读。

3.2　可移植可执行文件格式

PE格式，更恰当的术语应该是PE-COFF(Portable Executable and Common Object File Format，可移植可执行和通用对象文件格式)，很简单也很好理解。我们不准备在前面章节里已经涉及的主题上花很多时间（例如，会一笔带过.text或代码区段，或.data区段），而把精力放在最基本的概念上，使读者迅速掌握用十六进制编辑器打开文件的方法并熟悉文件头的结构。极少数没有介绍的区段及内部格式当练习留给读者。

图3-1显示了典型的PE文件布局。在开头，你将发现一个带头部的DOS stub程序，它非常

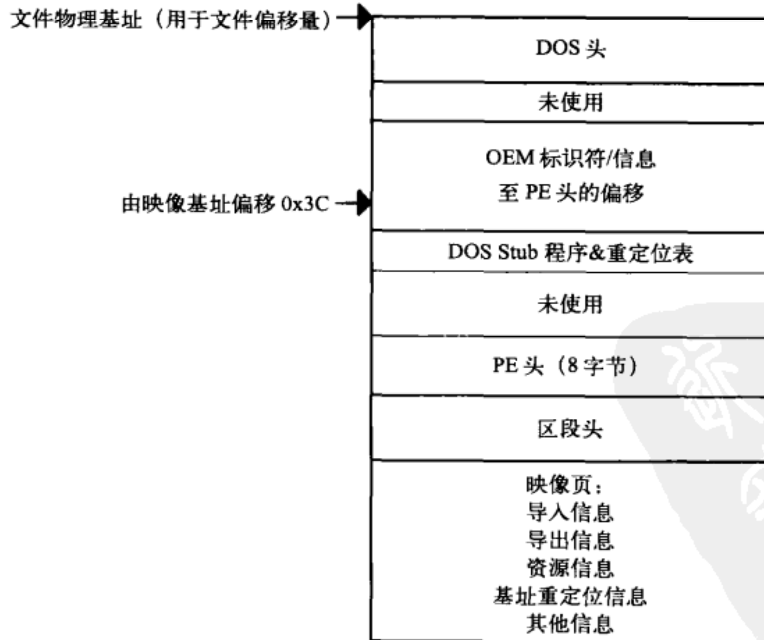


图 3-1　PE 文件布局

更多资源请点击访问稀.酷客(www.ckook.com)

简单，用于在 DOS 里运行应用程序时使用，其主要目的是保持向后兼容。对逆向工程师而言，它最重要的部分是一个指向 PE 头部的偏移量。从文件头开始向后偏移 0x3C 处就是这个偏移量，它指示从文件开头偏移多少是 PE 头部。

在这里需要注意，DOS 头部本身只存在于映像文件里，不会出现在其他文件里，如目标文件等。DOS 文件头的一个特性是其开始的几个字节是“魔术”字节，一般是字符 M 和 Z，表明它是 DOS 映像。受此及其他因素启发，我们可以识别一个文件是否是 PE。跳过 DOS 头（实际上它对我们没有什么用处），就是 PE 头。PE 头以另外的字段标识 PE 文件的开始——通常是 PE\0\0（注意，这里的\0 是二进制的 0），在此之后是 COFF 文件头（参见图 3-2）。该格式第一个字段是机器类型，它有 2 字节长，几乎总是 0x8664（AMD64）或 0x14c（IA32），很少是 0x200（IA64，安腾处理器）。之后是又一个 2 字节的字段，指示这个文件有多少个区段。按微软的说法，它的最大值为 96。下一个字段比较有趣，是一个指向 COFF 符号表的文件偏移量。作为一名逆向工程师，几乎可以肯定碰到的文件并没有符号表，因此，这个字段为 0，表示没有符号表。然而，如果有符号表的话，它的偏移量将在此指定。在这个字段之后是一个 4 字节的字段，指示有多少符号。最后是两个标准的 COFF 字段，都是 2 个字节，一个指示可选头部的大小（只在映像中存在），另一个字段称为特性（characteristics），定义文件中特殊的属性。

偏移量	大小	字段名
0	2	**Machine**
2	2	**Number Of Sections**
4	4	Time Date Stamp
8	4	**Pointer To Symbol Table**
12	4	**Number Of Symbols**
16	2	**Size Of Optional Header**
18	4	**Characteristics**

黑体表示已讨论过

图 3-2 COFF 文件头

特性字段指定文件的各种属性，例如，文件是否是 DLL（Dynamically Loaded Library，动态加载库），是否是系统文件的一分子，文件中的重定位信息是否被剥离，文件是否使用了 32 位字，等等。微软的正式文档对此有详细说明。

更多资源请点击访问稀.酷客(www.ckook.com)

如果有可选头部的话（目标文件里没有它），它将由三个主要的区段组成，第一个是8个字段，一般是COFF格式；接下来是21个Windows特定字段；最后是数据目录。（参见图3-3）在COFF区段里，比较有意思的字段是magic、代码大小、初始化数据的大小、未初始化数据的大小、进入点和代码基址、数据基址。magic是另一个可用于标识PE版本的标志，对于PE32或PE32+来说，其对应的有效字段是0x10B和0x20B。本章只绍介PE32格式（因为它更常见一些），而PE32+格式就当练习留给读者了。size字段比较好理解，它指示文件中.text/code、.data和.bss区段的大小。最后是进入点的RVA（Relative Virtual Address，相对虚拟地址）或是应用程序应该从哪里开始运行的地址。RVA是一个比较复杂的机制，一般用它表示到文件载入的VA（Virtual Address，虚拟地址）之间的偏移量。除了上面介绍的，接下来是code和data区段的虚拟基址。接下来，在可选头部的Windows特定字段里，我们发现有下列字段：映像基址，映像大小，头部的大小，DLL特性，最后是数据目录条目的数量及其大小。

偏移量	大小	可选头部区段
0	2	标准COFF字段
28	68	Windows特定字段
96	变长	数据目录

黑体表示已讨论过

图3-3　可选头部布局

映像基址是指加载时映像第一个字节的首地址。按照微软的说法，它必须是64K的倍数。其默认值视操作系统而定，并不是很重要；它仅是优先选择的对象，而加载器有权视具体情况选择其他值。接下来是映像大小及头部大小；映像大小是指文件的总体大小，包括了所有载入内存的头部的大小，以及指定头部大小的头部的大小。DLL特性字段明显只适用于DLL，指定与DLL有关的特性。对于逆向工程师，有意义的字段是0x0040，它表明基地可以动态分配并允许ASLR（Address Space Layout Randomization，地址空间布局随机化），0x0080表明已经做过代码完整性检查，0x0100表明这个映像兼容NX（no-execute），最后是0x0400，表明这个文件不使用SEH（Structured Exception Handling，结构化异常处理），从而预防SEH处理程序指向这个DLL。最后一个元素指定了可选头部中下一个子区段包含的元素数量（参见图3-4）。

数据目录有点像一个与众不同的怪物，指定映像中其他一些数据类型。例如，输入表数据目录指定应用程序将输入哪些库和函数。可选头部的数据目录区段是由16个结构（包含双字）组成的数组，指定给定数据目录的VA及大小。数据目录的详细说明见表3-1。

更多资源请点击访问稀.酷客(www.ckook.com)

偏移量	大小	字段名
0	2	**Magic**
2	1	MajorLinkerVersion
3	1	MinorLinkerVersion
4	4	**SizeOfCode**
8	4	**SizeOfInitializedDat**
12	4	**SizeOfUninitializedDate**
16	4	**EntryPoint**
20	4	**BaseOfCode**
24	4	**BaseOfData**
28	4	**ImageBase**
32	4	SectionAlignment
36	4	FileAlignment
40	2	MajorOSVersion
42	2	MinorOSVersion
44	2	MajorImageVersion
46	2	MinorImageVersion
48	2	MajorSubsystemVersion
50	2	MinorSubsystemVersion
52	4	Win32VersionValue
56	4	**SizeOfImage**
60	4	**SizeOfHeaders**
64	4	Checksum
68	2	Subsystem
70	2	**DLLCharacteristics**
72	4	SizeOfStackReserve
76	4	SizeOfStackCommt
80	4	SizeOfHeapReserve
84	4	SizeOfHeapCommt
88	4	LoaderFlags
92	4	**NumberOfRVAsAndSizes**

COFF 标准字段（偏移量 0～24）

Windows 特定字段（偏移量 28～92）

黑体表示已讨论过

图 3-4 可选头部

更多资源请点击访问稀.酷客(www.ckook.com)

表3-1　数据目录

名　字	描　述
输出表	文件输出函数的表
输入表	文件输入函数的表
资源表	文件使用的各种资源，如icon
异常表	文件使用的已注册异常处理程序
证书表	属性证书表
基址重定位表	文件中所有的基址重定位
调试	存储编译产生的调试信息
架构	保留，必须为0
全局指针	存储在全局指针寄存器中的数值的RVA
线程本地存储(TLS)表	线程数据存储中用到的信息
加载配置表	不同Windows版本中有不同的应用，从XP开始使用 SafeSEH 寄存器功能
绑定输入	绑定输入表
输入地址表(IAT)	运行前是同一输入查询表、运行时其中是已确定的符号地址
延迟输入描述符	与输入表相似，但延迟输入
CLR运行时头部	CLR运行时头部
保留	保留，必须为0

这里只详细描述了输出表、输入表及加载配置表，其他的表当练习留给读者。输出表指定了文件输出的函数。输出表（也称为.edata 区段）的格式如表 3-2 所示。

表3-2　输出表的格式

名　字	描　述
输出目录表	描述了全部的输出信息。包含了在映像中用于决定输入到输出函数的地址信息
输出地址表	包括了输出入口点的地址、相关数据和绝对地址
名字指针表	输出名表中对应的RVA组成的数组
序数表	输出地址表的16位序数数组
输出名表	已输出函数/数据等的以NULL结束的变长字符串名

需要注意的是，这些表并不需要都存在；如果输出只通过序数完成，那么只需 EDT（Export Directory Table，输出目录表）及 EAT（Export Address Table，输出地址表）。EDT 中有意思的字段是：name RVA、ordinal base、address table entries、number of name pointers、export address table (EAT) R V A、name pointer RVA 和 ordinal table RVA。参见图 3-5。

name RVA 是指向讨论中的 DLL 名字的 RVA。ordinal base 只是顺序索引起始的基址索引；一般设为 1。address table entries 和 number of name pointers 字段指出地址表和名字表里分别有多少条目，EAT RVA 、name pointer RVA 和 ordinal table RVA 就像它们的字面意思一样；它们指出 .edata 区段中剩下的表的 RVA。export address table 是非常简单的结构——只有一个元素，有两种表示方法，且通常联合使用两种方法。如果这个地址不在输出区段里（它由这个地址与可选头部里提供的长度相加所确定），这个字段是代码或数据真正的地址。

更多资源请点击访问稀.酷客(www.ckook.com)

否则这个字段是转发 RVA（forwarder RVA），指出在其他 DLL 里的符号。export name pointer table 是另一个比较简单的结构，如果已定义，那么对于每个输出项它简单地包含一个指向输出名字表的 RVA。输出名字只在这个表里包含了指针时有意义。ordinal table 是一个序数基址偏离 EAT 的 16 位索引的数组。序数表与输出名字表本质上互为镜像，也就是说其中一个表里面的索引提供了指向另一个表的索引，假如存在输出的名字。见图 3-6。

偏移量	大小	字段名
0	4	Export Flags
4	4	Time/Date Stamp
8	2	Major Version
10	2	Minor Version
12	4	**Name RVA**
16	4	**Ordinal Base**
20	4	**Address Table Entries**
24	4	**Number Of Name Pointers**
28	4	**Export Address Table RVA**
32	4	**Name Pointer RVA**
36	4	**Ordinal Table RVA**

黑体表示已讨论过

图 3-5 EAT

偏移量	大小	字段名
0	4	Export RVA
0	4	Forwarder RVA

黑体表示已讨论过

图 3-6 EAT

更多资源请点击访问稀.酷客(www.ckook.com)

最后，export name table 包含真正的为输出符号拼凑公共名（public name）的字符串输出；公共意味着如果它存在，应用程序可以按名字输入函数/数据。因此，综合来说，可以用下面的步骤把名字解析成符号：

(1) 获取 VA 或可选头部里的输出目录表。

(2) 用这个 VA 定位序数基址、输出目录表和序数表 RVA。

(3) 检索名字指针 RVA 的 RVA。

(4) 搜索输出名字指针表来确定函数是否是按名字输出的。

(5) 把指向名字指针表的索引作为指向序数表的索引来检索序数。

(6) 找到序数后用它减去序数基址，把结果作为指向 EAT 的索引。

(7) 这个索引中的数据就是输出函数的 RVA。

通过序数获取符号的过程和上面完全一样。只不过按名字寻找输出的步骤被删除了，从名字指针索引到序数索引的转换也被删除了。

输入表（也称为.idata 区段）使用的方法与输出表类似，相比之下还要简单一些。输入符号时会用到三个主要结构：IDT（Import Directory Table，输入目录表，如图 3-7 所示），ILT（Import Lookup Table，输入查找表，如图 3-8 所示）和提示/名称表。IDT 包含一些字段，我们在这里只讨论其中的 ILT RVA、名称 RVA 和 IAT RVA。除了名称 RVA 外，其他几个都可按字面理解，其实名称 RVA 也很简单，它是指向输入的以 NULL 结尾的 DLL ASCII 文件名的 RVA。ILT 和 IAT 是 32 位整数数组（在 PE32 上），其每个条目是位字段。条目的高位指出是按名字还是按序数输入的，如果它被置位那么表示是按序数输入的。如果是按序数输入，那么用 0~15 位表示输入的序数。如果是按名字输入的，则用 0~30 位表示输入的名字到提示/名称表的 RVA（共 31 位）。提示/名称 表也很简单，其每个条目的前两个字节充当到加载器的提示。

偏移量	大小	字段名
0	4	**Import Lookup Table RVA**
4	4	Time/Date Stamp
8	4	Forwarder Chain
12	4	**名称 RVA**
16	4	**Import Address Table RVA**

黑体表示已讨论过

图 3-7　IDT

更多资源请点击访问稀.酷客(www.ckook.com)

位	大小	字段名
31	1	**Ordinal/Name Flag**
15-0	16	**Ordinal Number**
30-0	31	**Hint/Name Table RVA**

黑体表示已讨论过

图 3-8 ILT

该提示就成了指向目标 DLL 中的输出名字指针表的索引。如果条目匹配，则使用它，否则就会搜索名字。接下来的元素是可变长度的、以 NULL 终止的 ASCII 字符串，也就是输入的函数名，为了与下一个条目对齐，后面可能还会填充 NULL 字节。

注意

虽然我们假定符号在真正捆绑前 IAT 与 ILT 包含相同的数据，但我发现事实并不总是这样。在很久以前，当我编写分析 PE 格式的工具时，就发现有些编译器会把 ILT 移到.txet 区段里，它的内容与预期的会完全不一样！我还发现有些编译器根本就没有使用 ILT，而只用了 IAT。在手动分析文件格式时，要注意这样的细微差别。微软和 Borland 都会尽可能走捷径。

最后我们再来谈谈加载配置结构，如图 3-9 所示。它在 Windows NT 中使用很少，而且其使用方式和 Windows 2000 后期系统中的使用方式也大不一样。在 Windows XP 及更新系统中，SafeSEH 用这个区段注册有效的系统异常处理程序，从而避免攻击者覆盖 SEH 条目，引发系统异常并执行代码的问题。如果可选头部中的 DLL 特性字段里的 IMAGE_DLLCHARACTERISTICS_NO_SHE 字段没有置位，且异常处理程序在系统试图调用它时不在这个列表里，将停止处理。在加载配置结构里，我们只对三个字段感兴趣：Security cookie、structured exception (SE) handler table、及 structured exception handler count。security cookie 并不真的是 cookie，而是一个指向它的指针。这个 cookie 有多种使用方式，最知名的就是在微软编译器中指定/GS 标志时，它实现栈 cookie 阻止基于栈的缓冲区溢出。SE handler table 是排过序的 RVA 表，它对应特定映像的有效 SEH 处理程序。SE handler count 是处理程序总数。

3.3 可执行链接格式

ELF (Executable and Linking Format，可执行链接格式) 诞生于 Unix System LaboratoriesUNIX 系统实验室，并最终作为 System V Application Binary Interface (ABI)的一部分面世，后来被 Tool Interface Standard (工具接口标准) 采用。有趣的是，此格式最初的名字是 Extensible Linking Format (可扩展链接格式)，很可能是因为以前的文件格式不支持扩展库动态链接的缘故。自从它作为可

更多资源请点击访问稀酷客(www.ckook.com)

选的文件格式被 Unix 和类 Unix 操作系统正式采用，它已成为 Unix 世界中的事实标准，几乎每个厂商都会把它作为原生格式，或者通过薄抽象层支持它。从 Linux、Solaris、IRIX、BSD 到 Playstation 几乎所有系统都在使用它。因此，除非你生活在纯 Windows 世界中，否则肯定会碰到 ELF 格式的文件。

偏移量	大小	字段名
0	4	Characteristics
4	4	TimeDateStamp
8	2	MajorVersion
10	2	MinorVersion
12	4	GlobalFlagsClear
16	4	GlobelFlagsSet
20	4	CriticalSectionDefaultTimeout
24	8	DeCommitFreeBlockThreshold
32	8	DeCommitTotalFreeThreshold
40	8	LockPrefixTable
48	8	MaximumAllocationSize
56	8	VirtualMemoryThreshold
64	8	ProcessAffinityMask
72	4	ProcessHeapFlags
76	2	CSDVersion
78	2	Reserved
80	8	EditList
88	4	**SecurityCookie**
96	4	**SEHandlerTable**
104	4	**SEHandlerCount**

黑体表示已讨论过

图 3-9　加载配置结构

更多资源请点击访问稀.酷客(www.ckook.com)

在 ELF 头部（如图 3-10 所示），没有严格规定好的大小；每部分的大小都是根据处理器的原生大小而定的，多种处理器都这样用。出于这个原因，本章只介绍 IA32。对我们这些从事逆向分析的人来说，ELF 头部和格式很直观也很有意思，希望你能弄明白本章谈及的头部中的各元素。e_ident 字段用于确认 ELF 文件：它确认此文件是 ELF 文件，确定处理器原生字的大小，指定字节顺序，及 ELF 头部的版本。这个字段是一个无符号字符的数组，共 16 个元素。开始的四个字节是"magic"字段，其值是 0x7F、E、L 和 F。接下来的一个字节指定字大小或类，其值为 0 时表示它是一个无效的类，其值为 1 表示它是 32 位，其值为 2 表示它是 64 位的。接下来的字节指定文件内数据的编码，其值可能为 0、1 或 2，0 表示无效，1 表示数据是小端字节序的二进制

偏移量	大小	字段名
0	16	e_ident
16	2	e_type
18	4	e_machine
22	4	e_version
26	4	e_entry
30	4	e_phoff
34	4	e_shoff
38	4	e_flags
42	2	e_ehsize
44	2	e_phentsize
46	2	e_phnum
48	2	e_shentsize
50	2	e_shnum
52	2	e_shstrndx

黑体表示已讨论过

图 3-10 ELF 头部

PDG

更多资源请点击访问稀.酷客(www.ckook.com)

补码，2 表示数据是大端字节序的二进制补码。接下来的字段指定 ELF 头部的版本，我们稍后就会看到此举有些多余。它一般应当设为 1，表示它正在使用 ELF 规范的当前版本。e_ident 字段里剩下的字节都是未使用的、保留的，规范建议程序在分析头部时忽略这些字节（它们通常以 0 填充）。e_ident 字段之后是 e_type 字段，它确定此文件是何种类型的可执行文件，可能的值有：0、1、2 和 3。0 表示它不是文件类型，1 表示这个文件是可重定位的文件，2 表示它是一个可执行文件，3 表示它是一个共享目标文件，0xFF00 和 0xFFFF 由处理器的规范确定。

e_type 字段之后是 e_machine 字段，它确定此文件在何种处理器上运行。对我们而言唯一有意义的值是 3，它表示是在 IA32 机器上运行的；其他的值对应一些不太流行的硬件平台，如 SPARC、Motorola、MIPS 及 IA8086。e_version 和 e_ident 头部里的版本值是一样的，0 表示这个版本是无效的，1 表示这个版本是当前版本。e_entry 字段完全可以从字面理解，如果适用的话，它保存应用程序进入点的 VA，否则被设为 0。e_phoff 字段保存的是此文件程序头部表（program head table）的用字节表示的偏移量。e_shoff 字段保存的是此文件区段头部表（section head table）的用字节表示的偏移量。只有当文件有这些表时 e_phoff 和 e_shoff 才出现，否则被初始化为 0。e_ flags 字段包含与处理器相关的标志；不过 IA32 架构并没有指定标志，因此这个字段将为（也应当是）0。e_ehsize 字段保存的是 ELF 头部的大小，e_phentsize 和 e_shentsize 分别确定程序头部表和区段头部表里一个条目的大小（用字节表示）。e_phnum 和 e_shnum 字段表示程序头部表和区段头部表里的条目数量；因此，如果要计算程序头部表的大小，把 e_phentsize 与 e_phnum 字段相乘即可。再说一次，如果这些表不存在，这些字段将被初始化为 0。最后，在 ELF 头部的结尾处是 e_shstrndx 字段，如果适用的话，它保存了区段头部表里区段名字符串表的索引。

ELF 区段头部表是区段头部结构的一个数组，每一个结构的格式如图 3-9 所示，且非常直观。这是说，在这个数组里肯定有保存特殊值的索引；这些特殊的索引如表 3-3 所示。

表3-3　ELF区段头部表数组的特殊索引

名　字	索引/值	名　字	索引/值
SHN_UNDEF	0	SHN_ABS	0xFFF1
SHN_LORESERVE	0xFF00	SHN_COMMON	0xFFF2
SHN_LOPROC	0xFF00	SHN_HIRESERVE	0xFFFF
SHN_HIPROC	0xFF1F		

SHN_UNDEF 标明未定义、不存在或无意义的区段引用。需要注意的是，尽管它未被定义，但区段头部表（如果存在的话）在索引 0 处总会包含 SHN_UNDEF 条目。因此，如果 e_shnum 字段声明有 10 个字段，其实是 9 个再加上 SHN_UNDEF 条目。SHN_LORESERVE 指定保留的索引范围的下界。SHN_LOPROC 和 SHN_HIPROC 指定为处理器特性条目保留的条目范围。SHN_ABS 指定相关符号的绝对值，这基本上意味着符号引用不受重定位的影响。SHN_COMMON 指定通用符号（例如 C 语言里未分配的外部变量），最后的 SHN_HIRESERVE 指定保留的索引范围的上界。ELF 文件里每一个区段都有一个区段头部描述它；尽管存在空区段和区段本身不重叠的现象，但区段描述是连续的。区段头部的元素如图 3-11 所示。

更多资源请点击访问稀.酷客(www.ckook.com)

偏移量	大小	字段名
0	2	**sh_name**
2	2	**sh_type**
4	2	**sh_flags**
6	4	**sh_addr**
10	4	**sh_offset**
14	2	**sh_size**
16	2	**sh_link**
18	2	**sh_info**
20	2	sh_addralign
22	2	**sh_entsize**

黑体表示已讨论过

图 3-11 区段头部

sh_name 是指向区段头部字符串表（section header string table）的索引，它指定区段名。sh_type 确定区段内容及语义的类型。具体定义的类型如表 3-4 所示。

表3-4 区段中内容和语义的类型

名 字	值	名 字	值
SHT_NULL	0	SHT_NOBITS	8
SHT_PROGBITS	1	SHT_REL	9
SHT_SYMTAB	2	SHT_SHLIB	10
SHT_STRTAB	3	SHT_DYNSYM	11
SHT_RELA	4	SHT_LOPROC	0x70000000
SHT_HASH	5	SHT_HIPROC	0x7FFFFFFF
SHT_DYNAMIC	6	SHT_LOUSER	0x80000000
SHT_NOTE	7	SHT_HIUSER	0xFFFFFFFF

更多资源请点击访问稀.酷客(www.ckook.com)

SHT_NULL 值表示区段头部未激活，没有与之相关联的区段。SHT_PROGBITS 表示区段中保存的数据由程序本身定义，只有程序才知道它的格式。SHT_SYMTAB 和 SHT_DYNSYM 区段定义了符号表；一个应用程序可能只有一个 SHT_SYMTAB 区段（尽管可能一个都没有），它包含完整的符号表，而 SHT_DYNSYM 区段是动态链接使用符号的最小集；我们不准备详细介绍这两个区段，因为你不太可能逆向分析真正包含符号的文件；不过我们希望读者能读一读 ELF 规范。SHT_STRTAB 是保存字符串表的区段；一个文件可以有多个字符串表区段，我们稍后会详细介绍这些区段。SHT_RELA 区段包含带有显式加数的重定位条目。SHT_HASH 区段包含一个符号哈希表，目前只允许每个对象有一个这种类型的区段；这个区段是动态链接中所有对象所必需的。SHT_DYNAMIC 类型用于动态链接。SHT_NOTE 确定这个区段保存以某种方式标记这个文件的信息；为简洁起见我们就不介绍这个区段了。SHT_NOBITS 表示这个区段不占用这个文件里的空间，然而它与 SHT_PROGBITS 区段类似；最知名的 SHT_PROGBITS 区段是.BSS。SHT_REL 保存没有显式加数的重定位条目，SHT_SHLIB 区段被保留，但有未指定的语义。SHT_LOPROC 和 SHT_HIPROC 在区段中定义一段范围，保留给处理器专用的语义使用；而 SHT_LOUSER 和 SHT_HIUSER 除了是为应用程序保留的之外，与它们一样。

在 sh_type 字段之后是 sh_flags 字段，它用位表示区段的各种属性。表 3-5 列举了这些属性。

表3-5 属 性

名 字	值	名 字	值
SHF_WRITE	0x1	SHF_EXECINSTR	0x4
SHF_ALLOC	0x2	SHF_MASKPROC	0xF0000000

SHF_WRITE 标志表示这个区段是可写的。SHF_ALLOC 表示这个区段在运行时应当驻留在内存里。SHF_EXECINSTR 指示这个区段包含可执行的指令。最后，SHF_MASPROC 指示这个区段是留给处理器专用的。

接下来是 sh_addr 字段；它提供了这个区段应该从哪里开始的首选 VA。sh_offset 除了提供从文件开头到区段开头的文件偏移量外，与 sh_addr 类似。sh_size 是区段的大小，除非它是 SHT_NOBITS 类型，尽管 SHT_NOBITS 区段也可以有非零的 sh_size；它不会占用物理文件中的空间。sh_link 和 sh_info 成员是两个相关联的值，需要根据区段的类型进行解释。在 SHT_DYNAMIC 区段的例子里，sh_link 成员指出区段使用的字符串表的区段头部表索引，有一个为 0 的 sh_info 字段。对于 SHT_HASH，sh_link 成员保存着适用于哈希表的符号表的区段头部表索引，sh_info 也是 0。要看到底有多少个值，请参考 ELF 规范。最后，sh_entsize 成员适用于有固定大小的表在它们里面的区段；例如，对一个带符号表的区段，这个条目可以指示这个符号表的长度。

现在我们对各区段有了一些了解，表 3-6 以标准的方式描述常见的区段，如区段的类型、属性及简短描述。应该注意的是，名字前带 “.” 的区段（例如.bss）是保留给系统使用的。

更多资源请点击访问稀.酷客(www.ckook.com)

表3-6 通用区段

区段名	类 型	属 性	描 述
.bss	SHT_NOBITS	SHF_ALLOC SHF_WRITE	包括未初始化数据，载入时初始化为0，一般是全局作用域内的
comment	SHT_PROGBITS	n/a	包含版本控制信息
.data	SHT_PROGBITS	SHF_ALLOC SHF_WRITE	包含已初始化的为应用程序存储映像建立的数据，一般是全局作用域的
.data1	SHT_PROGBITS	SHF_ALLOC SHF_WRITE	同上
debug	SHT_PROGBITS	n/a	用于符号调试的无硬性规定的内容
.dynamic	SHT_DYNAMIC	SHF_ALLOC SHF_WRITE (处理器专用)	包含动态链接信息，SHF_WRITE比特位是否为1取决于处理器
.dynstr	SHT_STRTAB	SHF_ALLOC	包含动态链接必需的字符串
.dynsym	SHT_DYNSYM	SHF_ALLOC	同上
.fini	SHT_PROGBITS	SHF_ALLOCSHF_EXECINSTR	包含结束应用程序的可执行指令，如destructor
.got	SHT_PROGBITS		稍后详细讲述
.hash	SHT_HASH	SHF_ALLOC	包含一个符号哈希表
.init	SHT_PROGBITS	SHF_ALLOCSHF_EXECINSTR	与ex.中的.fini相反，包含一个构建符
.interp	SHT_PROGBITS	SHF_ALLOC	包含一个程序解释器的路径名，如果文件有可加载区段，就设置此属性
.line	SHT_PROGBITS	n/a	包含调试用的行信息
.note	SHT_NOTE	n/a	由实现用于标记需要标记的可执行程序
.plt	SHT_PROGBITS		稍后详细讲述
.rel<name>	SHT_REL	SHF_ALLOC	包含重定位信息。如果有一个可加载区段，那么SHF_ALLOC就被置位。根据传统的定义，<name>是指要重定位区段的名字，如.rel.text
.rela<name>	SHT_RELA	SHF_ALLOC	包含重定位信息。如果有一个可加载区段，那么SHF_ALLOC就被置位。根据传统的定义，<name> 是指要重定位区段的名字，如.rala.text
.rodata	SHT_PROGBITS	SHF_ALLOC	包含的是只读数据，如常量字符串
.rodata1	SHT_PROGBITS	SHF_ALLOC	同上
.shstrtab	SHT_STRTAB	n/a	包含区段名
.strtab	SHT_STRTAB	SHF_ALLOC	包含字符串，通常是符号表条目名。如果有可加载区段，则SHF_ALLOC被设置
.symtab	SHT_SYMTAB	SHF_ALLOC	包含一个符号表（不在本章叙述）。如果有可加载区段，则SHF_ALLOC被设置
.text	SHT_PROGBITS	SHF_ALLOCSHF_EXECINSTR	组成程序的可执行指令

程序头部表是程序头部结构的一个数组，这些结构定义了段（segment），以及怎样将二进制载入 OS（Operating System，操作系统）。表的大小及其包含的条目数量是在 ELF 头部中指定的。

更多资源请点击访问稀.酷客(www.ckook.com)

有一些段是追加的，而另一些有助于进程映像。就像 ELF 文件里的其他部分（不包括 ELF 头部本身）那样，没有段的特定顺序，也没有到程序头部表的特定偏移量，这些都由 ELF 头部单独定义。在图 3-12 的右边，是 ELF_32Phdr 结构的成员及顺序的图解。P_type 字段表示描述的是什么类型的段，并告诉系统怎样解释它的内容。定义的值如表 3-7 所示。

偏移量	大小	字段名
0	4	**p_type**
4	4	**p_offset**
8	4	**p_vaddr**
12	4	p_paddr
16	4	**p_filesz**
20	4	**p_memsz**
24	4	**p_flags**
32	4	p_align

黑体表示已讨论过

图 3-12　程序头部结构

表3-7　定义的值

名　字	值	名　字	值
PT_NULL	0	PT_SHLIB	5
PT_LOAD	1	PT_PHDR	6
PT_DYNAMIC	2	PT_LOPROC	0x700000000
PT_INTERP	3	PT_HIPROC	0x7FFFFFFF
PT_NOTE	4		

类型为 PT_NULL 的段未被使用，它的其他成员的值是未经定义的（因此应当被忽略）。定义另一个 NULL 类型的段，其理由是允许段被定义但在实现时会忽略。类型为 PT_LOAD 的段优先载入位于 p_vaddr 地址处的内存。从文件被映射的内存基址的位于 p_offset 的第一个 p_ filesz 字节被载入内存。如果 p_memsz 大于 p_filesz，这些字节也会被映射到段里并填充零，当 p_ filesz 成员大于 p_memsz 时，它是无效的。如果 PT_INTERP 段存在的话，那么它必须位于 PT_LOAD 段之前，作为程序解释器的路径名，这个段在文件里只能出现一次（如果希望这个文件是有效的）。类型为 PT_DYNAMIC 的段与动态链接相关。相比较而言，PT_NOTE 不是很重要，但它允许与应用

更多资源请点击访问稀.酷客(www.ckook.com)

程序交互检查一致性（也就是说 GLIBC 版本）。PT_SHLIB 被定义但保留给以后使用，包含 PT_SHLIB 段的文件不遵守 ABI。PT_PHDR 段指定程序头部表的大小和位置，此规范适用于物理文件及内存里的映像。就像 PT_INTERP 类型的段，它在文件里只能出现一次，而且如果出现的话，必须位于 PT_LOAD 段之前。最后，PT_LOPROC 和 PT_HIPROC 被保留，用于处理器指定的功能。

根据前面的描述，我们可以猜测：p_offset 成员指定从文件头开始的段偏移量。p_vaddr 成员指定段的首选 VA。p_filesz 和 p_memsz 元素分别表示段在物理文件和内存中的大小。最后，p_flags 指定段的属性，它有三种可能的值：PF_R、PF_W 和 PF_X，分别对应读、写和执行。

现在，我们对段有了一些基本的了解，接下来可以简单讨论一下可执行映像、共享库映像或采用 ASLR 的映像之间的差异。为了加载可执行映像，在建立映像时必须包括每个段使用的地址，这个地址由段的 p_vaddr 成员指定。如果地址被改变的话，将导致对这个映像的绝对引用被中止。ASLR 映像和共享库映像可以通过使用所谓的 PIC（Position Independent Code，位置无关代码）绕过此限制。PIC 隐含的意思是，不再采用绝对引用指向某段数据，而是改为相对引用。例如，在传统代码里你可能访问一个在地址 XYZ 处的变量，在 PIC 里你可以通过另外的手段引用它，也就是说相对于当前的位置。我们来看一个例子，应用程序使用共享库来访问通常使用的函数，例如常见的标准 C 函数库，会使用一系列的中介，在 ELF 的例子里，它们是：GOT（Global Offset Table，全局偏移表），即.got；_DYNAMIC（_Dynamic Segment/Section，动态段/区段），即.dynamic；PLT（Procedure Linkage Table，过程链接表），即.plt。这三个段是整体关联的，它们也是替代老标准例如 a.out 的主要理由——标准动态链接。.dynamic 段出现在每一个采用动态链接的可执行映像里；这个段由符号_DYNAMIC 所引用，它是一个由结构组成的数组，如图 3-13 所示。

偏移量	大小	字段名
0	4	**d_tag**
4	4	**d_val**
4	4	**d_ptr**

黑体表示已讨论过

图 3-13 动态结构

动态结构包括两个值，一个是 tag，另一个是 union。Tag 决定了如何解释 union。d_val 成员包含一个整数，它有多种解释，而 d_ptr 成员包含 VA。大家都知道，编译时的 VA 和运行时的 VA 可能不一样，重定位区段并不包含_DYNAMIC 数组的重定位。在表 3-8 里，你将看到一些定义的 d_tag 类型，无论它们是可选的还是强制的。这并不是一个完整的列表，我只挑了一些和

更多资源请点击访问稀.酷客(www.ckook.com)

我们相关的，再次建议有兴趣的读者阅读 ELF 规范，从而获取更多的细节及完整的解释。

表3-8　定义的d_tag类型

名　字	值	d_val或d_ptr?	是否可执行	共享目标
DT_NULL	0	忽略	强制	强制
DT_PLTRELSZ	2	d_val	可选	可选
DT_PLTGOT	3	d_ptr	可选	可选
DT_FINI	13	d_ptr	可选	可选
DT_PLTREL	20	d_val	可选	可选
DT_JMPREL	23	d_ptr	可选	可选

DT_NULL 元素作为标记_DYNAMIC 数组结束的标志，是必须存在的。除它之外，这个数组内并没有固定的顺序。DT_PLTRELSZ 元素保存着与 PLT 有联系的重定位条目的总大小。如果存在 DT_ JMPREL 元素，那么对应的 DT_PLTRELSZ 条目也必须存在。DT_PLTGOT 条目保存着与 GOT 或 PLT 关联的地址，将在后面对此作详细介绍。DT_FINI 元素保存着结束函数或析构函数的地址，这对黑客来说可能会有用处，因此我们对它也有兴趣，稍后会大致了解一下它。最后，DT_JMPREL 条目包含一个指向仅仅与 PLT 相关的重定位条目的指针。分开这些重定位允许链接程序（linker）使用被称为后期连接(lazy binding)的连接形式，从而在映像初始化过程中忽略它们。后期连接在真正用到这些符号时再重定位。例如，思考下面的 C 程序：

```
#include <stdio.h>
#include <stdlib.h>
int
main(void)
{
        unsigned int cnt;
        for (cnt = 0; cnt < 2; cnt++)
         printf("cnt: %u\n", cnt);
        exit(EXIT_SUCCESS);
}
```

在这个应用程序中，有 2 个（可见的）标准库函数，即 printf()和 exit()。如果采用后期连接，直到最后需要时才会解析这两个符号。第一次调用 printf()时解析这个符号，这会导致重定位开销，而不是在初始化时；不过，当第二次调用 printf()时，因为这个符号已经被解析过了，也就不会产生开销了。然而，解析 exit()仍会产生开销。这样做的好处是提高了速度和效率，因为并不是每个符号都会被解析。此外，它还使动态模板加载更容易处理。不过，它也有一些缺点——有问题的程序将更容易被利用。近年来，流行向称作 relro 的加固后的 GCC 工具链传递一个标志，它会在程序初始化时做所有的重定位工作，然后在运行时禁止向这些段写入数据。这意味着攻击者将不能利用一些含糊的指针算法或缓冲区溢出向这些区段写入数据，之后也不能把执行流重定向到这些区段。尽管具体情形取决于架构体系，但一般会标记这些区段：.init/.ctors、.fini/.dtors、PLT、GOT 和.dynamic。这个方法正变得越来越常见，作

更多资源请点击访问稀.酷客(www.ckook.com)

为一名逆向工程师，随着时间的推移，你碰到后期连接的机会可能会越来越少。

在 IA32 平台上，可通过符号_GLOBAL_OFFSET_TABLE（一个地址的数组）访问 GOT。这些地址被绝对引用，也允许 PIC 对此的相对引用。因此，PIC 代码将获得 GOT 的地址，并从它的相对引用提取绝对引用。符号_GLOBAL_OFFSET_TABLE 不需要指向.got 段的开头，因此正索引和负索引都是有效的。在加载映像时，动态链接器遍历重定位，寻找指定类型的条目，用它们的绝对地址替换它们在 GOT 中的条目，从而有效避开静态链接器的限制。GOT 的第一个元素是一个特别的条目，包含了_DYNAMIC 结构的地址；这允许动态链接器通过在_DYNAMIC 结构里找出自己来处理这个 GOT，而不必依赖任何重定位信息。此外，在 IA32 上第二个和第三个条目也被保留，用于保存特殊的值。GOT 重定位位置无关地址到绝对位置，而 PLT 也做同样的工作，只不过是为函数而做。PLT 在需要时确定函数的绝对地址并更新 GOT。PLT 的实现多种多样，主要取决于它是否采用了 PIC 编译。一个非 PIC 条目看起来像下面这样：

```
PLT
.PLT0
        push    address_of_GOT+0x04
        jmp     [address_of_GOT+0x08]
        nop
        nop
        nop
        nop
.PLT1:
        jmp     [name1_in_GOT]
        push    offset
        jmp     [.PLT0+$]
.PLT2:
        jmp     [name2_in_GOT]
        push    offset
        jmp     [.PLT0+$]
        . . .
```

而 PIC PLT 看起来可能像下面这样：

```
PLT
  .PLT0
        push    [ebx+0x04]
        jmp     [ebx+0x08]
        nop
        nop
        nop
        nop
.PLT1
        jmp     [ebx+name1]
        push    offset
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
        jmp     [.PLT0+$]
.PLT2
        jmp     [ebx+name2]
        push    offset
        jmp     [.PLT0+$]
```

所有这些都要综合考虑上下文，为了解析动态引用，链接器及应用程序将按以下步骤协同工作。

(1) 在创建映像的基础上，把 GOT 中的第二个和第三个条目设成像下面定义的值。

(2) 如果 PLT 是 PIC，那么 GOT 的地址必须驻留在 ebx 寄存器里。调用函数负责把这个地址放入寄存器。

(3) 假设应用程序尝试调用 name1（可以在标签.PLT1 处找到它）。

(4) 位于标签处的第一条指令是跳转到 GOT，它最初包含跟在到 GOT 的跳转指令后的 push 和 jmp 指令的地址。

(5) 之后，应用程序压入重定位条目的地址，在这个例子里就是名为 offset 的变量。这个偏移量将指定在前面跳转时使用的与符号表索引一起的 GOT 条目，在这种情形是 name1。

(6) 之后，应用程序跳转到.PLT0，并把 GOT 第二个元素的地址压入栈，为了识别给动态链接器一个字来引用，然后把控制权交给第三个 GOT 条目，这将把控制权交给动态链接器。

(7) 动态链接器展开栈并检索识别信息，找出符号的绝对地址，并把它存入相关的 GOT 条目，然后把控制权交给被请求的函数。

(8) 对这个函数的远调用将跳过压入 offset，作为 GOT 条目被修改的结果，它将跳到 .PLT0。

就像你看到的那样，动态链接通过间接和抽象的方式来完成。应用程序预先完全不知道将调用什么地址，它依次调用 PLT，PLT 再依次跳转至 GOT。如果这个地址至此还没有解析成功，控制权将被交回 PLT，PLT 压入重定位条目，并跳转到 PLT 里的第一个条目，然后把控制权交给动态链接器。

注意

前面提到，一些高级特性（例如要求关闭后期连接，从而使重定位发生在初始化阶段，而不是在运行时）在不久前已经实现了。如果你努力思考过，那么现在应该知道 GOT/PLT 怎样工作了，也可能了解了其中缘由。如果作为一个攻击者，我可以覆盖一个 GOT 条目，那么真正的问题是应用程序在 shellcode 获得控制权之前再调用那个函数。在 www.milw0rm.com/papers/3 可以找到它的白皮书。

与此相类似，通过覆盖.dtors 里的数据（一般包含了在.fini 里的函数的地址）可以攻击给定映像的析构函数。通过覆盖那里的地址，应用程序在执行时将有可能调用流氓函数（rogue function）。Juan M. Bello Rivas 公布了此技术，在 http://synnergy.net/downloads/papers/dtors.txt 可以找到他写的白皮书。

更多资源请点击访问稀.酷客(www.ckook.com)

3.4 小结

我们一起简要学习了 PE 及 ELF 文件格式，你对这些格式有了基本的认识，希望你在手工分析它们时能用得上。你应该可以确定 PE 的输入表及输出表（从而了解怎样重建打包 PE 里的输入区段），并真正理解发生在 ELF 文件里的动态链接是怎么回事。对这两种格式，你应该熟悉它们的成员及结构，并对链接器及加载器怎样处理这两种格式有一些了解。作为贯穿本章的建议，我们鼓励大家阅读它们的规范，因为我们在本章只强调了我们认为比较重要的内容，你可能会发现一些有趣的或有用的概念并没有被提及。你可以在 www.muppetlabs.com/~breadbox/software/ELF.txt 或在搜索引擎中输入“ELF specification”找到 ELF 规范，在 www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx 可以找到 PE 的规范。

更多资源请点击访问稀.酷客(www.ckook.com)

第4章

实战1

本章内容：

- 理解执行流
- 跟踪函数
- 恢复硬编码的密码
- 寻找有问题的函数
- 回溯执行
- 了解缓冲区溢出

☑ 快速跟踪并找出解决方案
☑ 常见问题

更多资源请点击访问稀.酷客(www.ckook.com)

4.1 导言

本章，我们将不再讲述 IDA 的基本知识，而是直接讨论如何灵活运用我们的已有知识。对于具有安全、汇编和编程方面的常识的普通的计算机从业人员及专业安全技术人员来说，这是一个很好的起点。下面就通过对首个二进制程序的逆向分析，了解此程序有什么功能，然后我们将这些知识应用到安全产业的惯例中。具体来说，我们将看看能否找到程序在启动时要求的密码，然后把该知识应用于在二进制程序中寻找漏洞。用这两种方法，我们最终将明白利用应用程序所需要的步骤。

在 Syngress 网站上可以下载本章用到的示例代码和二进制文件的压缩包，文件名为：StaticPasswordOverflow.zip。

4.2 跟踪执行流

逆向分析二进制程序的第一步是确定程序做些什么，以及是怎么做的。让我们开始此任务，一步步地跟踪我们的程序，并对二进制程序中的一般操作做一些注释。首先，直接跳到第一块有用的代码段。我个人喜欢用记事本或笔记簿等记下分析过程中的地址及一切可能有用的事项，从而保持自己清晰的思路。在逆向分析开始时，你并不知道什么时候可能就会用到这些记录，对我而言，从键盘输入要比在纸上写慢得多。此外，在纸上画图要更容易一些。

```
.text:00401270 ; int __cdecl main(int argc,const char **argv,const char *envp)
.text:00401270 Dst    = byte ptr -80h
.text:00401270 argc   = dword ptr 8
.text:00401270 argv   = dword ptr 0Ch
.text:00401270 envp   = dword ptr 10h
.text:00401270        push    ebp
.text:00401271        mov     ebp, esp
.text:00401273        sub     esp, 80h
.text:00401279        push    offset aReverseEnginee
.text:0040127E        call    sub_401554
.text:00401283        add     esp, 4
.text:00401286        push    offset aPleaseProvideT
.text:0040128B        call    sub_401554
.text:00401290        add     esp, 4
.text:00401293        push    80h     ; Size
.text:00401298        push    0       ; Val
.text:0040129A        lea     eax, [ebp+Dst]
.text:0040129D        push    eax     ; Dst
.text:0040129E        call    _memset
.text:004012A3        add     esp, 0Ch
.text:004012A6        lea     ecx, [ebp+Dst]
.text:004012A9        push    ecx
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
.text:004012AA          push    offset a127s ; "%127s"
.text:004012AF          call    _scanf
.text:004012B4          add     esp, 8
.text:004012B7          lea     edx, [ebp+Dst]
.text:004012BA          push    edx ; Str2
.text:004012BB          call    sub_4011C0
.text:004012C0          add     esp, 4
.text:004012C3          movsx   eax, al
.text:004012C6          test    eax, eax
.text:004012C8          jge     short loc_4012D9
.text:004012CA          push    offset aYouFailed_
.text:004012CF          call    sub_401554
.text:004012D4          add     esp, 4
.text:004012D7          jmp     short loc_4012E6
.text:004012D9 loc_4012D9: ; CODE XREF: _main+58
.text:004012D9          push    offset aYouWon_Goodbye
.text:004012DE          call    sub_401554
.text:004012E3          add     esp, 4
..text:004012E6 loc_4012E6: ; CODE XREF: _main+67
.text:004012E6          mov     eax, 1
.text:004012EB          mov     esp, ebp
.text:004012ED          pop     ebp
.text:004012EE          retn
.text:004012EE _main endp
```

大致看一下我们就能发现，main 函数基本没做什么事情。它包含了一些调用及条件语句。通过观察这些语句里的字符串，我们似乎可以假定代码里有判断成功或失败的语句，因为字符串中包含“YouFailed”和“YouWon”。我们可以把 IDA 切换到图表模式，看这些条件语句是怎样工作的。但我们首要的任务是理解整个函数是怎样工作的，这样到了后面我们就不会感到太迷惑了。

```
.text:00401279          push offset aReverseEnginee
.text:0040127E          call sub_401554
.text:00401283          add  esp, 4
.text:00401286          push offset aPleaseProvideT
.text:0040128B          call sub_401554
```

这里我们发现，程序在设好栈后，向栈空间压入一个静态字符串并调用了一个函数。看一下压入的字符串，可以假设它是某种启动时的提示语。不过，IDA 好像并不能确定这个二进制程序调用了什么函数。所以我们继续往下看，跟踪 call 指令看它会跳到哪里。选中 call 指令并按回车键跳转到那个位置。

```
.text:00401554 ; int printf(const char *,...)
.text:00401554 _printf        proc near ; CODE XREF: sub_401000+65
.text:00401554 ; sub_401000+C0
```

更多资源请点击访问稀.酷客(www.ckook.com)

啊哈！看起来好像 IDA 并不想认出这个函数是干什么的；它是一个静态编译的 printf。我们可以认为这个函数没有做任何多余的操作，因此把它标记成 print 函数并继续。按 Backspace 键回到进入点并继续。

注意　一些函数可能并不像它看上去的样子，应该仔细审视一下这样明显命名的静态函数，确认其名实相符。因为坏家伙可能会用这样的办法蒙骗我们。

因为我们已经知道这个例程是干什么用的，所以应该在 IDA 中把它重命名，这样一来就不用担心后面会把它和其他函数弄混了。在这个指令名上单击，并按下 N 键，IDA 会弹出一个更改名字的对话框。为方便起见，我们把这个函数重命名为 printf（它本来也应该是这个名字）。在完成这个工作后，我们发现这段代码其实是作为一个启动过程打印字符串的。继续往下看：

```
.text:00401290         add     esp, 4
.text:00401293         push    80h      ; Size
.text:00401298         push    0        ; Val
.text:0040129A         lea     eax, [ebp+Dst]
.text:0040129D         push    eax      ; Dst
.text:0040129E         call    _memset
.text:004012A3         add     esp, 0Ch
.text:004012A6         lea     ecx, [ebp+Dst]
.text:004012A9         push    ecx
.text:004012AA         push    offset a127s ; "%127s"
.text:004012AF         call    _scanf
```

逐步跟踪这些指令，一眼就能看到它调用 memset()函数填充一个缓冲区，然后用 scanf()把数据读入缓冲区。特别要提到的，我们看到 memset()函数调用向 Dst 栈缓冲区的前面 0x80（十进制是 128）个字节填入 0x00 或 NULL。因为 memset()调用前需要压入值，我们可以在那儿看到下面 4 条指令：

```
.text:00401293         push     80h    ; Size
.text:00401298         push     0      ; Val
.text:0040129A         lea      eax,  [ebp+Dst]
.text:0040129D         push     eax    ; Dst
```

我们看到此二进制程序压入了表示大小的 0x80，表示数值的 0，最后压入的是指向[ebp+Dst]地址的指针，也就是我们的栈变量。最后，我们看到 scanf()调用也使用了同样的操作。特别值得一提的是，载入指向 Dst 缓冲区的指针的指令又被执行了一次，这导致 scanf()的结果被保存在了这个缓冲区里。我们也看到这个 scanf()调用用格式化串%127s 正确地填充了缓冲区，因此，只会向这个缓冲区写入 127 个字节。

更多资源请点击访问稀.酷客(www.ckook.com)

如果你不习惯十六进制的表示形式，在 IDA 里你可以在数值上右击，然后查看或选择自己习惯的数制。尽管 IDA 默认用十六进制表示形式，你可以在数值上点击并按下 H 键，把它转换成标准的十进制表示形式。

暂时回到这个函数初始化栈的部分，我们需要确保这些大小对应这个变量在处理后的真正大小。我们看到 IDA 已经确定了这个函数确实有一个变量，它的长度为 0x80 个字节。此外，我们还看到栈初始化调用执行这个函数，因此确定这是这个栈变量的硬设置大小。

```
.text:00401270        Dst     = byte ptr -80h
 ……
.text:00401273        sub     esp, 80h
```

现在，我们解决了这些乏味的代码，而且我们也理解了它们的作用。我们继续看代码中那些有趣的条件语句，那似乎是所有“魔力”产生的地方。如果切换到图形视图（按空格键），你可以看到条件跳转正好发生在 sprintf()调用之后，如图 4-1 所示。此外，你可以看到小图表窗口，这对有许多条件跳转的较大函数来说是非常有用的。

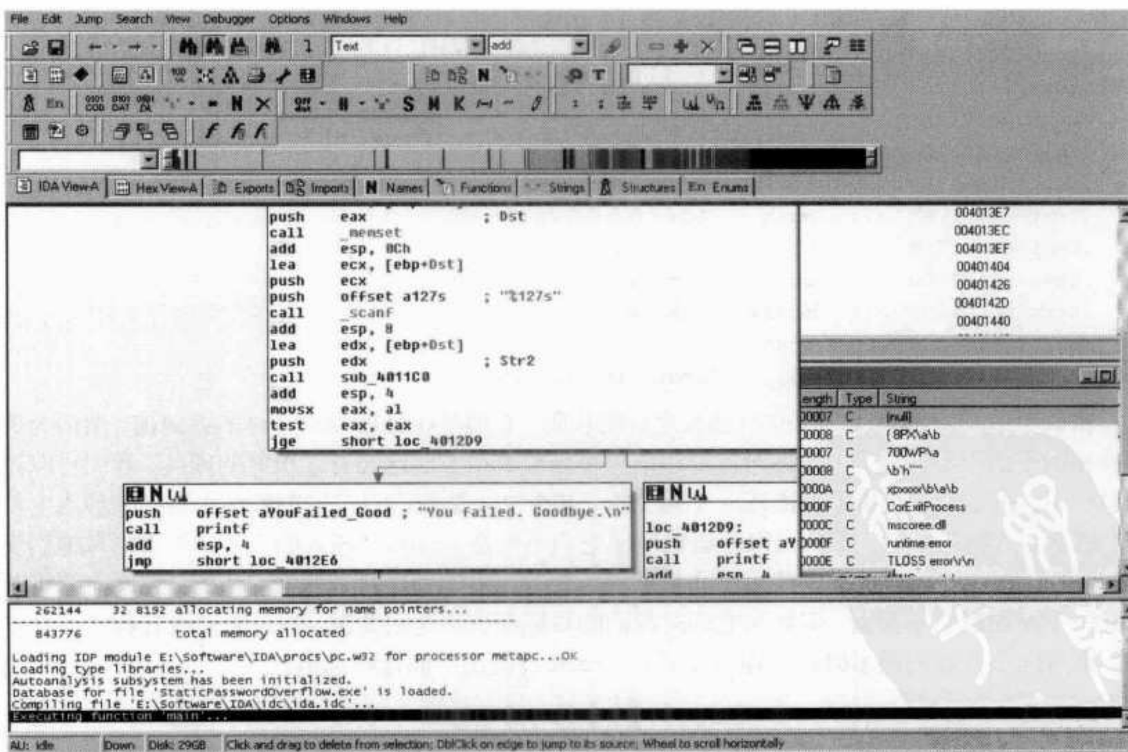


图 4-1 图形视图

更多资源请点击访问稀.酷客(www.ckook.com)

逆向分析二进制程序

跳过这个函数可能的设置部分，我们可以看到这个函数在条件跳转（已成为我们的目标）之前调用了一些子例程。特别值得一提的是，我们看到在条件语句之前，子例程正调用另外的在二进制程序内的静态例程，然后立即转换到栈，执行这个条件语句。

> **工具和陷阱**
>
> **在IDA Pro中的二进制子例程**
>
> 分析二进制程序里的似乎还不可认别的子例程时要特别仔细。尽管通常可以假设它们是可执行程序本身的一部分，IDA Pro将用这种方式识别许多静态编译的库函数。
>
> 在可能时尽量导入调试符号，一旦你理解函数的作用就把它们标上标签。ELF Linux二进制文件在这方面是臭名昭著的，无数个小时可能就浪费在跟踪静态编译的基础库或GOT表上。

```
.text:004012A9          push    ecx
.text:004012AA          push    offset a127s    ; "%127s"
.text:004012AF          call    _scanf
.text:004012B4          add     esp, 8
.text:004012B7          lea     edx, [ebp+Dst]
.text:004012BA          push    edx     ; Str2
.text:004012BB          call    sub_4011C0
.text:004012C0          add     esp, 4
.text:004012C3          movsx   eax, al
.text:004012C6          test    eax, eax
.text:004012C8          jge     short loc_4012D9
```

我们可以扫一眼调用 scanf()函数之后的代码，它似乎在处理 scanf()的返回值，然后为调用神秘的子例程准备栈。假定你对这个栈结构及函数调用方法已经有了很好的理解，我们可以再重温一下。在 scanf()调用前刚压入了两个值，其中之一是我们在二进制程序中看到的静态字符串。综合各方面的资料，我们可以轻松推测出它们不仅是 scanf()压入的变量，而且也和他们表示的一样。这在逆向分析那些使用了非常见库函数的二进制程序时是非常有用的。在逆向分析前先查一下库函数总不会错，但如果它们对我们的目标而言不是很关键，假设它们像宣称的那样工作，则可以节省我们很多时间。你可以看到 scanf()就像下面这样构造：

```
int scanf(const char *format, …);
```

注意到这点，我们现在看到真正被加载并传递的首先是指向缓冲区的指针，然后是格式化串（记住，变量被“反向”压入栈)。接下来，栈被移动了8个字节，然后指向我们的缓冲区[ebp+Dst]

更多资源请点击访问稀.酷客(www.ckook.com)

的指针被载入 edx 并压入栈。在这之后，神秘的子例程就被调用了。现在可以假设这个函数对被 scanf()所填充的缓冲区执行了一些神秘处理，并返回一个随后被用于比较的整型值。从紧跟在调用之后的指令：movsx eax, al 和 test eax, eax 我们基本上就可以推断出这点。这些指令告诉这部分代码从被调用的子例程里获取返回值，调用 sub_4011C0，如果大于或等于的话将跳转（调用的返回值通常保存在 eax 寄存器里）。

嗯，回顾一下，我们现在知道这个神秘子例程对我们的输入值执行了一些操作，并提供控制我们的条件跳转的值。我们离目标又近了一步！现在，如果我们能知道这个例程正执行哪些操作，以及怎样给它提供合适的值，我们就能控制这个条件跳转。供参考，我们把调用语句标为 input_process，然后通过双击它的名字把 IDA pro 视图切换到这个函数。

处理子例程

现在，我们知道这个函数控制着我们的成败，让我们仔细单步跟踪这些代码，以理解那里到底可能会发生什么。另外，查看下面的代码，为了真正理解发生的事情，可能需要跳过一些代码。不是所有的逆向工程都是从头分析到结尾的，有时候直接分析结束部分反而可以加速我们的理解过程。

```
.text:004011C0 ; int __cdeclinput_process(char *Str2)
.text:004011C0 input_processproc near ; CODE XREF: _main+4B
.text:004011C0 Dst    = byte ptr -80h
.text:004011C0 var_7F= byte ptr -7Fh
.text:004011C0 var_7E= byte ptr -7Eh
.text:004011C0 var_7D= byte ptr -7Dh
.text:004011C0 var_7C= byte ptr -7Ch
.text:004011C0 var_7B= byte ptr -7Bh
.text:004011C0 var_7A= byte ptr -7Ah
.text:004011C0 var_79= byte ptr -79h
.text:004011C0 var_78= byte ptr -78h
.text:004011C0 var_77= byte ptr -77h
.text:004011C0 var_76= byte ptr -76h
.text:004011C0 var_75= byte ptr -75h
.text:004011C0 var_74= byte ptr -74h
.text:004011C0 var_73= byte ptr -73h
.text:004011C0 var_72= byte ptr -72h
.text:004011C0 var_71= byte ptr -71h
.text:004011C0 var_70= byte ptr -70h
.text:004011C0 Str2   = dword ptr 8
.text:004011C0         push    ebp
.text:004011C1         mov     ebp, esp
.text:004011C3         sub     esp, 80h
.text:004011C9         push    80h     ; Size
.text:004011CE         push    0       ; Val
.text:004011D0         lea     eax, [ebp+Dst]
.text:004011D3         push    eax ; Dst
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
.text:004011D4          call    _memset
.text:004011D9          add     esp, 0Ch
.text:004011DC          mov     [ebp+var_70], 0
.text:004011E0          mov     [ebp+var_75], 73h
.text:004011E4          mov     [ebp+Dst], 74h
.text:004011E8          mov     [ebp+var_76], 73h
.text:004011EC          mov     [ebp+var_7F], 68h
.text:004011F0          mov     [ebp+var_7A], 6Dh
.text:004011F4          mov     [ebp+var_7C], 69h
.text:004011F8          mov     [ebp+var_7B], 73h
.text:004011FC          mov     [ebp+var_71], 64h
.text:00401200          mov     [ebp+var_74], 77h
.text:00401204          mov     [ebp+var_7E], 69h
.text:00401208          mov     [ebp+var_7D], 73h
.text:0040120C          mov     [ebp+var_78], 70h
.text:00401210          mov     [ebp+var_73], 6Fh
.text:00401214          mov     [ebp+var_72], 72h
.text:00401218          mov     [ebp+var_79], 79h
.text:0040121C          mov     [ebp+var_77], 61h
.text:00401220          mov     ecx, [ebp+Str2]
.text:00401223          push    ecx     ; Str2
.text:00401224          lea     edx, [ebp+Dst]
.text:00401227          push    edx     ; Str1
.text:00401228          call    _strcmp
.text:0040122D          add     esp, 8
.text:00401230          test    eax, eax
.text:00401232          jz      short loc_401247
.text:00401234          push    offset aInvalidPasswor ;
"\n******* INVALID PASSWORD *  ******\n"
.text:00401239          call    printf
.text:0040123E          add     esp, 4
.text:00401241          or      al, 0FFh
.text:00401243          jmp     short loc_40125D
.text:00401245          jmp     short loc_40125D
.text:00401247 loc_401247: ; CODE XREF: input_process+72
.text:00401247          mov     eax, [ebp+Str2]
.text:0040124A          push    eax
.text:0040124B          push    offset aSIsCorrect_ ; "%s is correct.\n\n"
.text:00401250          call    printf
.text:00401255          add     esp, 8
.text:00401258          call    sub_401000
.text:0040125D loc_40125D: ; CODE XREF: input_process+83
.text:0040125D ; input_process+85
.text:0040125D          mov     esp, ebp
.text:0040125F          pop     ebp
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
.text:00401260          retn
.text:00401260 input_process endp
```

我们看到这个函数比进入点大一些，但似乎 IDA Pro 在一定范围内提供了帮助。浏览一下代码，立即可以发现函数的一些重要部分，我们应当对它们加以考虑。在地址 00401228 处，是 strcmp()调用。

```
.text:00401220          mov     ecx, [ebp+Str2]
.text:00401223          push    ecx      ; Str2
.text:00401224          lea     edx, [ebp+Dst]
.text:00401227          push    edx      ; Str1
.text:00401228          call    _strcmp
```

看起来 strcmp()调用并不只是比较字符串，它也使用保存 scanf()输入内容的缓冲区 Ebp+Dst。这看起来像我们的目标：在我们输入值之后是字符串比较，紧接着是决定成功与否的条件语句。但它和什么作比较呢？让我们往回退一点查看代码。在这个字符串比较操作里，我们可以看到这个函数正在比较 ecx 和 edx 里的指针，这两个指针分别是 ebp+Str2 和 ebp+Dst，在具体确定 Str2 是什么之前，为备查把它改名为 StrPassword。

在进行比较之前，有一小块 mov 指令正在操作一段连续的内存空间，那恰好在我们以前分配的缓冲区里。memset()函数分配了 128 字节的栈缓冲区（即 0x80 字节），并把它们置为 NULL。

```
.text:004011C0          push    ebp
.text:004011C1          mov     ebp, esp
.text:004011C3          sub     esp, 80h
.text:004011C9          push    80h     ; Size
.text:004011CE          push    0       ; Val
.text:004011D0          lea     eax, [ebp+Dst]
.text:004011D3          push    eax     ; Dst
.text:004011D4          call    _memset
```

缓冲区准备妥当后，它被连续用这些静态变量进行填充。仔细看一下，会发现这些值彼此之间也颇为接近；此外，它们都在 ASCII 范围内，这能讲得通，因为它们被用在字符串比较中。现在，让我们弄明白这些值到底是什么！可惜，在静态分析二进制程序时，我们不能像在调试器里那样弹出窗口然后等待将填入的值，因此，最好手动剥离这些数据并确定它们是什么。

提示

大多数逆向工程专家在分析二进制程序时会频繁使用铅笔和纸。在纸上作笔记、画草图、记数据等要比使用计算机键盘的方式快得多。试着在手边放好纸和笔，以便随时记下地址、数值等事项。好记性不如烂笔头。

不过在我们深入前，在 IDA 解释这段代码时出现了一个小反常现象。在所有的 mov 指令中，IDA 确定有一个 mov 正向我们的输入缓冲区写入数据。

```
.text: 004011E4mov      [ebp+Dst], 74h
```

更多资源请点击访问稀.酷客(www.ckook.com)

这是怎么回事？往回看一下 IDA 已经确定这个函数具有的变量，我们可以看到除了这个反常之处外，它试图为每一单条的 mov 指令自动分配一个不同的变量引用。一瞥之下，这使这个二进制程序看上去复杂且具有迷惑性。由于 IDA Pro 反汇编函数的方式，有时很难从代码中确定真正的内存结构。此外，比较好的经验规则是不要完全依赖 IDA Pro 分发的变量和参数；相反，只在需要时作为参考。此处的异常是由于 IDA 这样的问题，它在分发变量时没有顾及栈对齐调整。因此，当 IDA 确定 ebp+dst 是什么后，可能会把它当作 var_80（如果 IDA 这样命名的话）。我们知道，在调用函数前会把指针压入栈，因为这个原因，IDA 在这个操作过程中没有适当地计算固有的栈移位，因此导致这段代码被曲解。知道栈是怎样构造的，我们就可以还原它本来的面目，看起来如下所示：

[建立的缓冲区]

[指向我们建立的输入字符串的指针]

[栈指针保存及返回地址]

[真实的输入字符串]

因此，这条指令（IDA 认为它会覆盖我们的指针）实际上是写四个字符数组的第一个字节，也就是密码字符串的第一个字符。现在，我们注意到了这个小问题，就可以把它适当考虑进来了；我们可以坐下来弄清这个字符数组包含的内容和密码。参见表 4-1。

注意

在静态分析二进制程序过程中，碰到函数时在纸上画出栈结构是不错的主意。特别是碰到复杂的函数时，我们并不能完全依赖 IDA。这就是为什么在静态分析过程中，理解指令执行结果及栈结构非常重要的原因所在。虽然不能测试、检验我们的假定，但必须推断可执行文件每一步的执行情况。

表4-1　口令字符串分析

位　　置	值	字　　符	位　　置	值	字　　符
70	0	(NULL)	79	79	y
71	64	d	7A	6D	m
72	72	r	7B	73	s
73	6F	o	7C	69	i
74	77	w	7D	73	s
75	73	s	7E	69	i
76	73	s	7F	68	h
77	61	a	80	74	t
78	70	p			

我们看到和输入缓冲区做比较的实际上是一个有效的 ASCII 字符串。因为这是一个普通的 x86 二进制程序，所以此字符串在代码里是以逆序的形式出现的。因此，我们的密码是“thisismy-password”。回头再来看这段代码，会看到这个函数为了确定我们是否提供了正确的密码，其中

更多资源请点击访问稀.酷客(www.ckook.com)

的条件语句用 strcmp()的结果作为判断的依据。在确定应用程序做什么，以及它怎样保护自己之后，我们算是跨过第一个障碍了。

继续看这段代码，我们发现在返回之前，它用了一个条件跳转检查正确的密码，紧接着调用一个外部函数。更有趣的是，这个被调用函数（sub_40100）的返回值作为这个函数的返回值被传递了。

```
.text:00401255          add    esp, 8
.text:00401258          call   sub_401000
.text:0040125Dloc_40125D: ; CODE XREF: input_process+83
.text:0040125D ; input_process+85 ❑j
.text:0040125D          mov    esp, ebp
.text:0040125F          pop    ebp
.text:00401260          retn
```

此话怎讲？在调用 sub_401000 函数前，这个栈实际上移动了 8 个字节。确切地说，ESP 增加了 8。这个移动使两个函数的返回值的栈位置重合了，所以这个值被传递了。

往回走两步，我们现在知道这个二进制程序的一些新信息。第一，为了继续执行，我们需要一个密码。第二，如果输入正确的密码，程序将处理第二个函数，即最后返回给进入点主函数的返回值。为了得到一个成功的响应，我们还需要满足第三个函数内的条件。我们把这个调用重命名为 SecondCheck，然后双击它，就可以查看它的内容。

```
.text:00401000 SecondCheck proc near ; CODE XREF: input_process+98
.text:00401000 Dst    = byte ptr -4D0h
.text:00401000 var_450      = byte ptr -450h
.text:00401000 Dest   = byte ptr -400h
.text:00401000         push    ebp
.text:00401001         mov     ebp, esp
.text:00401003         sub     esp, 4D0h
.text:00401009         push    esi
.text:0040100A         push    edi
.text:0040100B         mov     ecx, 13h
.text:00401010         mov     esi, offset aPleaseSelectAn ; "Please select an option
from the follow"…
.text:00401015         lea     edi, [ebp+var_450]
.text:0040101B         rep     movsd
.text:0040101D         movsw
.text:0040101F         movsb
.text:00401020         push    80h      ; Size
.text:00401025         push    0        ; Val
.text:00401027         lea     eax, [ebp+Dst]
.text:0040102D         push    eax      ; Dst
.text:0040102E         call    _memset
.text:00401033         add     esp, 0Ch
.text:00401036         push    80h      ; Size
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
.text:0040103B          push    0 ;  Val
.text:0040103D          lea     ecx, [ebp+Dest]
.text:00401043          push    ecx ; Dst
.text:00401044          call    _memset
.text:00401049          add     esp, 0Ch
.text:0040104C loc_40104C: ; CODE XREF: SecondCheck+1AB
.text:0040104C          mov     edx, 1
.text:00401051          test    edx, edx
.text:00401053          jz      loc_4011B0
.text:00401059          lea     eax, [ebp+var_450]
.text:0040105F          push    eax
.text:00401060          push    offset aS ; "%s"
.text:00401065          call    printf
.text:0040106A          add     esp, 8
.text:0040106D          lea     ecx, [ebp+Dst]
.text:00401073          push    ecx
.text:00401074          push    offset a127s_0 ; "%127s"
.text:00401079          call    _scanf
.text:0040107E          add     esp, 8
.text:00401081          push    80h ; Count
.text:00401086          lea     edx, [ebp+Dst]
.text:0040108C          push    edx ; Source
.text:0040108D          lea     eax, [ebp+Dest]
.text:00401093          push    eax ; Dest
.text:00401094          call    _strncat
.text:00401099          add     esp, 0Ch
.text:0040109C          push    offset Str2 ; "Exit"
.text:004010A1          lea     ecx, [ebp+Dst]
.text:004010A7          push    ecx ; Str1
.text:004010A8          call    _strcmp
.text:004010AD          add     esp, 8
.text:004010B0          test    eax, eax
.text:004010B2          jnz     short loc_4010CD
.text:004010B4          lea     edx, [ebp+Dst]
.text:004010BA          push    edx
.text:004010BB          push    offset aOperationSComp ; "Operation: %s: Completed\n"
.text:004010C0          call    printf
.text:004010C5          add     esp, 8
.text:004010C8          jmp     loc_401195
.text:004010CD loc_4010CD: ; CODE XREF: SecondCheck+B2
.text:004010CD          push    offset aSelect ; "Select"
.text:004010D2          lea     eax, [ebp+Dst]
.text:004010D8          push    eax ; Str1
.text:004010D9          call    _strcmp
.text:004010DE          add     esp, 8
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
.text:004010E1          test    eax, eax
.text:004010E3          jnz     short loc_4010FE
.text:004010E5          lea     ecx, [ebp+Dst]
.text:004010EB          push    ecx
.text:004010EC          push    offset aOperationSCo_0 ; "Operation: %s: Completed\n"
.text:004010F1          call    printf
.text:004010F6          add     esp, 8
.text:004010F9          jmp     loc_401195
.text:004010FE loc_4010FE: ; CODE XREF: SecondCheck+E3
.text:004010FE          push    offset aDrop ; "Drop"
.text:00401103          lea     edx, [ebp+Dst]
.text:00401109          push    edx ; Str1
.text:0040110A          call    _strcmp
.text:0040110F          add     esp, 8
.text:00401112          test    eax, eax
.text:00401114          jnz     short loc_40112C
.text:00401116          lea     eax, [ebp+Dst]
.text:0040111C          push    eax
.text:0040111D          push    offset aOperationSCo_1 ; "Operation: %s: Completed\n"
.text:00401122          call    printf
.text:00401127          add     esp, 8
.text:0040112A          jmp     short loc_401195
.text:0040112C loc_40112C: ; CODE XREF: SecondCheck+114
.text:0040112C          push    offset aCreate ; "Create"
.text:00401131          lea     ecx, [ebp+Dst]
.text:00401137          push    ecx ; Str1
.text:00401138          call    _strcmp
.text:0040113D          add     esp, 8
.text:00401140          test    eax, eax
.text:00401142          jnz     short loc_40115A
.text:00401144          lea     edx, [ebp+Dst]
.text:0040114A          push    edx
.text:0040114B          push    offset aOperationSCo_2 ; "Operation: %s: Completed\n"
.text:00401150          call    printf
.text:00401155          add     esp, 8
.text:00401158          jmp     short loc_401195
.text:0040115A loc_40115A: ; CODE XREF: SecondCheck+142
.text:0040115A          push    offset aExit_0 ;        "Exit"
.text:0040115F          lea     eax, [ebp+Dst]
.text:00401165          push    eax ; Str1
.text:00401166          call    _strcmp
.text:0040116B          add     esp, 8
.text:0040116E          test    eax, eax
.text:00401170          jnz     short loc_401188
.text:00401172          lea     ecx, [ebp+Dst]
```

4

更多资源请点击访问稀.酷客(www.ckook.com)

```
.text:00401178        push    ecx
.text:00401179        push    offset aOperationSCo_3 ; "Operation: %s: Completed\n"
.text:0040117E        call    printf
.text:00401183        add     esp, 8
.text:00401186        jmp     short loc_401195
.text:00401188 loc_401188: ; CODE XREF: SecondCheck+170
.text:00401188        push    offset aInvalidCommand ; "Invalid command failure.
Please try aga"…
.text:0040118D        call    printf
.text:00401192        add     esp, 4
.text:00401195 loc_401195: ; CODE XREF: SecondCheck+C8, SecondCheck+F9
.text:00401195        push    80h     ; Size
.text:0040119A        push    0       ; Val
.text:0040119C        lea     edx, [ebp+Dst]
.text:004011A2        push    edx     ; Dst
.text:004011A3        call    _memset
.text:004011A8        add     esp, 0Ch
.text:004011AB        jmp     loc_40104C
.text:004011B0 loc_4011B0: ; CODE XREF: SecondCheck+53
.text:004011B0        mov     al, 1
.text:004011B2        pop     edi
.text:004011B3        pop     esi
.text:004011B4        mov     esp, ebp
.text:004011B6        pop     ebp
.text:004011B7        retn
.text:004011B7 SecondCheck endp
```

如你所见，这个函数比我们已经见过的都要大。虽然我们到目前为止掌握的技能可以搞定它，但碰到这么复杂的函数，最好还是稍微改进一下分析方法。分析复杂的函数时，最好先从全局掌握函数调用和各种条件语句，然后再深入运算过程。因此，我们准备先弄清楚这个函数里的一系列函数调用。这里我们想进入图形视图，看 IDA Pro 为我们确定的条件跳转结构，这样，我们就可以了解更多关于这些函数间调用关系的信息，如表 4-2 所示。

表4-2 函 数

函数名	描 述
memset()	用零来填充函数中的栈缓冲区
printf()	输出命令请求头部文本
scanf()	在第一个参数表示的缓冲区中接收并存储用户输入
strncat()	从第一个缓冲区中接收并复制数据到第二个缓冲区
strcmp()	比较用户预先输入的数据和一个静态命令字符串
printf()	输出经过格式化之后的结果

如你所见，这个函数除了存在多个条件语句外，它的结构并不像看起来那样复杂。不过，看

更多资源请点击访问稀.酷客(www.ckook.com)

一下图形视图，再看一下由不同条件指定的 printf 输出内容，明显可以看出这个方法是一种命令解析引擎：它用 scanf 获得命令，解析并检查输入的内容，然后输出适当的结果。另外仔细看时，通过图形视图我们发现这实际上是一个无限循环。尽管我们通过分析二进制程序里纷杂的跳转语句也可以得出这样的结论，但 IDA Pro 以流程图的方式把这个函数从结束到开始的总体连接显示出来，更容易查看。实际上，这明显是一个简单的解析命令字符串的循环。(参见图 4-2)。

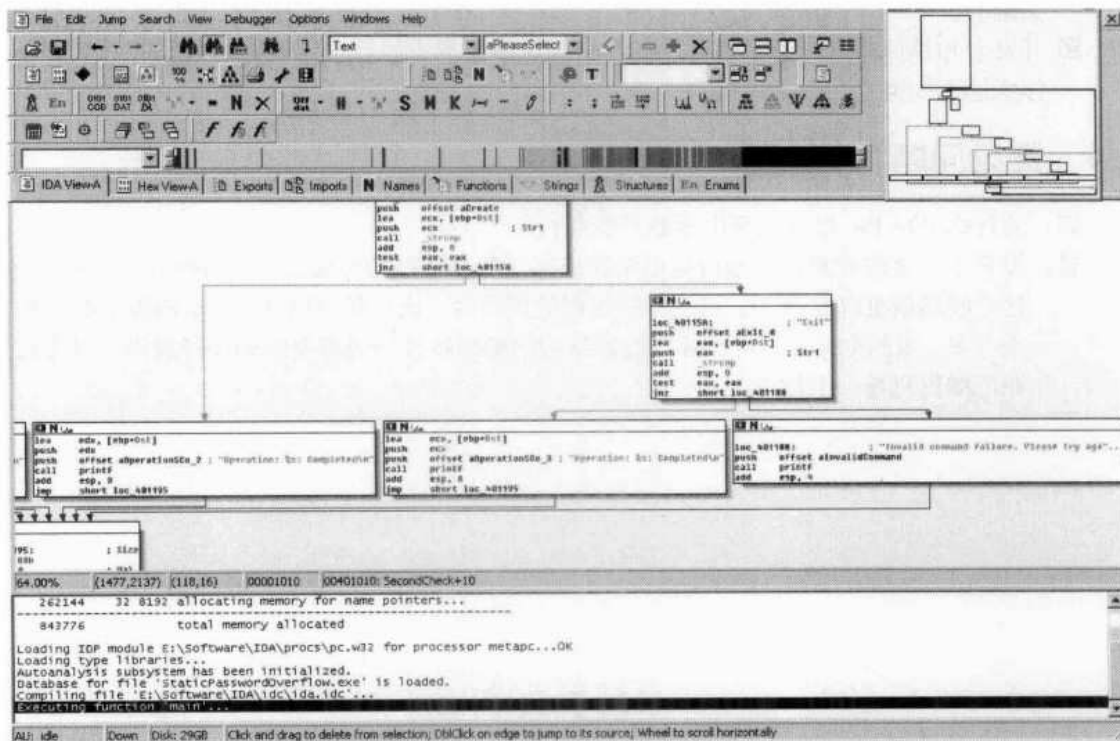


图 4-2 在图形视图中读 print()输出

4.3 快速跟踪并找出解决方案

理解执行流

☑ IDA 有许多工具和视图，比如说图形视图，它可以帮助我们快速了解二进制程序的实际操作。

☑ 在深入剖析整块代码前，要审查一个或一组函数的执行流。

☑ 从长远看，熟悉各种数制会有一定的帮助，特别是十六进制。心算要比用计算器节省很多时间。

☑ 编译器在编译时会对二进制程序做一些稀奇古怪的事情，有时会让人费解或不得要领，但在大多数优化情形下，这些差异还是很小的。

更多资源请点击访问稀.酷客(www.ckook.com)

恢复硬编码的密码

- ☑ 识别主要的条件语句，例如在多种环境里，比较的结果真或假比硬编码的口令要有用得多。然而，要留意优化器和迷惑技术，它们会滥用跳转和比较指令，而这会使可执行文件非常难懂。
- ☑ 在识别怎样恢复口令或绕过二进制程序里的检查时，找出控制真/假语句的代码块是非常关键的。
- ☑ 开发者和编译器有时为了提升性能，会尝试多变的方法。一定要提防代码的恶意部分，你永远不知道它们会做什么。

4.4　常见问题

问：为什么 IDA Pro 很难识别出函数的参数？

答：没有上下文的静态代码分析是很难完成的任务。特别是再加上类型转换和优化编译器，这个问题就更加复杂了。自己检查例程的调用者，比依赖 IDA Pro 识别函数的参数要安全得多。虽然如此，但对于那些控制结构的编译器代码仍需要时刻保持警惕，因为它们在汇编里只是一组数值。

更多资源请点击访问稀.酷客(www.ckook.com)

第 5 章

调试

本章内容：

- 调试的基础知识
- 使用 IDA Pro 进行调试
- 调试技术在逆向工程中的应用
- 堆和栈的访问和修改
- 其他调试器

☑ 小结

更多资源请点击访问稀.酷客(www.ckook.com)

5.1 导言

调试是指在软件中找出bug的行为。通常情况下，这是在程序中出现bug时由开发者来完成的。调试有多种形式，菜鸟程序员通常把输出作为调试的基本形式。比如说在C语言里，输出可以由printf语句完成。

关于调试有一个广为流传的故事，它与真正的“小虫”（bug）有关。用bug表示编程错误起源于Admiral Grace Murray Hopper。当时有一只小蛾子钻进了哈佛大学马克II计算机的继电器中，除去小蛾子的过程后来就被说成调试了。

调试器本身也是程序，它运行并监视其他程序的执行。调试器可以控制并更改目标程序的执行，当然也可以监视并更改内存及变量。

5.2 调试的基础知识

调试器是逆向工具包里最基本的工具。可以用它执行运行时分析，从而加速程序理解及逆向工程。某些任务用调试器做很容易，例如可以实际观察而不必猜测调用链。

在调试时跟踪间接调用很容易，例如通过寄存器调用就是间接调用。IDA Pro的静态分析以受限方式跟踪间接调用，这时不建立交叉引用。

我们可以通过调试来监视、观察并指导逆向工程。我们并不想分析整个程序，只想了解感兴趣的部分。

工具和陷阱

用户模式调试器和内核模式调试器

用户模式调试器针对进程操作。因为它们本身也是标准的进程，所以仅限于可以访问的内存。用户模式调试器不能访问内核内存，因此不能调试内核模式下的代码。内核模式代码可以是操作系统、模块和驱动程序。

利用驱动程序的攻击正变得日益流行。攻击内核可以绕过许多现代防护技术。一个例子是Broadcom无线网卡驱动侦测响应SSID溢出(CVE-2006-5882)，使用一个特别定制的探测器在内核里执行任意代码。因为这个攻击所处的位置比防火墙还要靠近底层，所以防火墙不会处理这些包。

把rootkits写成驱动的形式已经有很多年的历史了。一些恶意软件采用rootkit的策略，把自己嵌入内核。DRM软件通常有一个包含漏洞的驱动程序，参见CVE-2007-5587，我们已经发现它被利用的案例了。

用户模式调试器的例子有IDA Pro和Ollydbg等。

更多资源请点击访问稀.酷客(www.ckook.com)

> 在 Windows 环境下，事实上的内核模式调试器是 Compuware 公司开发的，包含在 DriverStudio 里的 SoftICE。不过可惜的是 SoftICE 已停止开发且不再被支持了。所幸的是微软自己的调试工具已经有了长足的进步[①]。

5.2.1 断点

断点是让调试器里的程序停在我们选择的地方。执行停止后，控制权就会传给调试器。断点有两种形式：硬件断点和软件断点。硬件断点，就像名字表达的那样，需要 CPU 提供硬件支持。

1. 硬件断点

IA-32 系列处理器支持 4 种硬件断点。硬件断点使用特殊的调试寄存器。这些寄存器包含断点地址、控制信息及断点类型。

断点地址保存在调试寄存器 D0 至 D3 中。为了设置断点，需要有一个大小字段。大小可以是 1、2 或 4 字节。执行中断使用的大小为 1 字节。在 64 位 CPU 上，大小已经扩展至 8 字节。有多种条件可以触发断点：

- 执行中断；
- 内存访问中断（读内存或写内存）；
- 仅写内存时中断；
- I/O 端口访问中断（很少被使用，很多调试器没有这个选项）。

2. 软件断点

软件断点只能中止执行。软件断点由于缺乏硬件支持，一般都是模拟出来的。软件断点用触发调试器的指令替换原来的指令。在 IA-32 处理器里，这条新指令通常是 INT 3（0xCC）。调试器必须记录原来的指令。

当软件断点被触发时，INT 3 指令把控制权传给调试器。调试器在内部表里查找断点，并用原来的指令替换 INT 3，然后恢复指令指针，使保存的指令成为下一条要执行的指令。整个过程对用户来说是不可见的；调试器将显示包含原来的指令的反汇编。

3. 使用断点

调试器使用软件断点要多于硬件断点。最主要的理由是软件断点没有数量上的限制。有些反调试技术会通过计算代码的校验和判断是否有指令被修改了。我们将在第 6 章详细介绍反调试技术。

硬件断点可以设在内存地址上，这点与软件断点不一样。我们可以通过中断内存访问寻找使用的表或内存崩溃。

① 国人开发的 Syser Debugger 也值得尝试。——译者注

更多资源请点击访问稀酷客(www.ckook.com)

5.2.2　单步

单步是指每次执行一条指令，然后把控制权交给调试器的过程。IA-32 系列处理器在硬件上直接支持单步。通过一次执行一条指令，我们可以仔细监视代码的特定部分。不过不太可能用这个方法调试整个程序。单步通常用于理解特定部分的代码。

从 CPU 的角度看，调试器设置 EFLAGS 寄存器里的 TF（Trap Flag，陷阱标志）位。一条指令执行完后就会生成调试异常，调试器会把这个调试异常当作中断（INT 0x01）捕获。

注意　大多数调试器都提供 step（单步）命令。通常把他们称为步入（step into）和步过（step over）。从用户的角度来看，它们之间唯一的不同是出现在某些指令上，其实就是 call 和 rep 指令上。当碰到 call 时，步入命令将跟随 call，而步过命令将在 call 之后的指令上中断。这通常由设置断点来完成，而不是单步执行到返回。

5.2.3　监视

我们需要跟踪变量。在源代码级调试里，通常以变量的位置（地址）命名变量。在汇编级调试里，通常用内存位置命名变量。编译器有时会把变量优化进寄存器。

监视是显示变量或表达式的一种方法。每当控制权传给调试器时（诸如断点或单步执行）它们就会被更新。监视的对象可以是一个变量（例如 loop_counter），或一个表达式（例如 packet[offset*4]）。

5.2.4　异常

程序员用异常捕获错误。下面的伪代码就是一个异常的例子：

```
_try()
{
        Open(file)
}
_except()
{
        Printerror
}
```

调试器可以中止异常，也可以把它传给应用程序处理。异常并不意味着一定出现了问题。很多时候程序会使用定制的异常。定制异常也是一种常见的反调试技术。

在 Windows 里，异常 0xc000000005 表示访问违例。这意味着进程企图访问没有被映射的地址。你可能看过像下面这样的提示：

更多资源请点击访问稀.酷客(www.ckook.com)

```
:
Exception C0000005 (ACCESS_VIOLATION reading [41414141])
```

地址 0x41414141 没被映射到进程的内存空间里，看起来很像是用 A 覆盖数据的一部分。我们希望出现访问违例时调试器会停下来。

5.2.5 跟踪

跟踪是执行程序并记录相关信息的过程。UNIX 下的 strace 命令可以运行一个可执行程序，并截获所有的系统调用（包括传递的参数）。

```
user@redbull:~$ strace ls
execve("/bin/ls", ["ls"], [/* 31 vars */]) = 0
brk(0) = 0x805c000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0xb7eec000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
```

跟踪还可以在调试器里进行，记录各个层次的细节。指令级跟踪是最详细的，通常是在执行每条指令后，把寄存器的值记录下来。当然，并不总是需要记这么详细，而且指令级跟踪也会非常地慢。

函数跟踪可以通过在所有函数的入口上设置断点或单步执行到 call 来实现。当一个函数被调用时，执行被停止。调试器记录下函数的参数，并有选择地记录诸如寄存器和调用者之类的数据。然后恢复执行。

虽然函数跟踪比指令跟踪更快一些，但有时它提供的信息并不够用。在理想情况下，我们只想跟踪基本块。基本块是一系列的顺序指令，它们连续执行而没有分支。这类跟踪有助于确定为什么会出现分支，以及如果需要，为了跳到某个分支该如何修改输入。虽然基本块跟踪比指令跟踪快很多，但速度仍比较慢。从 P6 系列的处理器开始，Intel 对跟踪分支提供了硬件支持。较新的处理器在这方面提供了更多的功能，但它们都使用 MSR 寄存器。

Intel 在 *Intel® 64 and IA-32 Architectures System Programming Guide* 的第 18 章介绍了 Last Branch Recording。关于这方面的最新研究成果已经公开了，其验证工具也可以找到，见 www.openrce.org/blog/view/535/Branch_Tracing_with_Intel_MSR_Registers。

5.3 使用 IDA Pro 进行调试

IDA Pro 从 4.50 版开始提供了内置的调试器。这个调试器是以插件的形式实现的，这是 IDA Pro 超强扩展性的有力证明。

IDA Pro 不仅可以调试本地系统，也可以通过网络操作本地及远程系统。调试客户端允许 IDA 调试其他的运行不同操作系统甚至不同 CPU 的机器。虽然提供了认证，但实际经验告诉我们，

更多资源请点击访问稀.酷客(www.ckook.com)

调试最好还是在局域网上进行。

IDA Pro 支持如下调试环境：

- 本地 WIN32 系统；
- 远程 WIN32 系统；
- 远程 WIN64 系统；
- 远程 LINUX 系统（仅支持 x86）；
- 远程 OSX 系统（仅支持 x86）；
- 远程 WinCE 系统（仅支持 ARM）。

注意

在 IDA Pro 的 GUI 里可以改变寄存器的值，但不能改变内存位置。

IDC 是 IDA Pro 提供的脚本语言，可以改变内存位置。内存位置包括数据和可执行的代码。像 `PatchByte()`、`PatchWord()`和 `PatchDword()`这样的函数都可用于这个目的。

如果被分析的二进制程序匹配前面所列的目标系统，调试器菜单选项将变为可用状态。从调试器菜单选项里可以设置调试器，如图 5-1 所示。

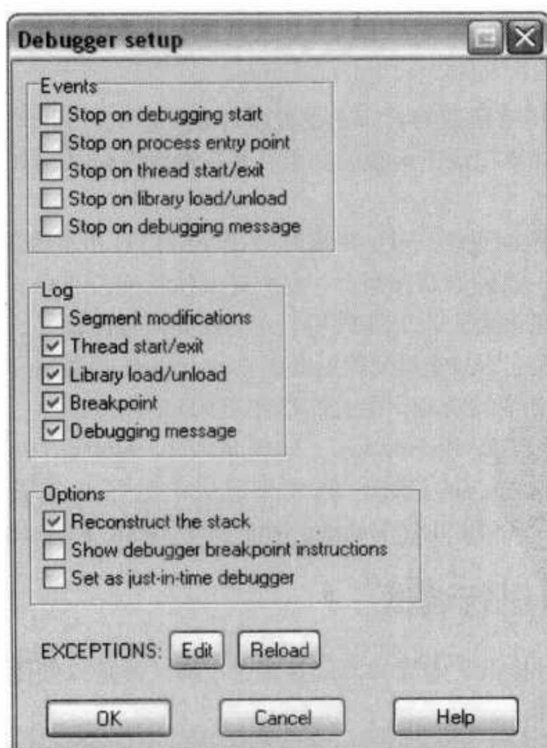


图 5-1　IDA 调试器设置选项

更多资源请点击访问稀.酷客(www.ckook.com)

一些值得的关注的选项是：

- Stop on debugging start——在碰到带有 TLS 区段的 PE 二进制文件时，在进入点之前停下；
- Stop on entry point——在罗列的进入点上停下。这时一些初始化可能已经完成了；
- Set as just-in-time debugger——Windows 允许把调试器设为在程序崩溃时默认使用；
- Exceptions——这个选项控制 IDA 怎样处理异常，不论它们是否传给应用程序。

工具和陷阱

编写属于自己的IDA Pro调试器客户端

你可以用 IDA SDK 编写插件。SDK 里面包括了 Linux 调试器插件/客户端的源代码。这个插件的源代码显示了与 IDA 协同工作所需要的接口。

我们可以借鉴这些源代码，为现在还不支持的操作系统或架构编写插件/客户端。

5.4 调试技术在逆向工程中的应用

为了演示 IDA Pro 的调试功能，我们将以 Netcat 为例。Netcat 是一个网络工具，它的名字源于 network 与 UNIX 命令 cat 的结合。你可以通过网络向/从其他的程序用管道输送数据，这是大家为什么把它称为“TCP/IP 瑞士军刀”的原因所在。

在 2004 年 12 月，Windows 1.1 版的 Netcat 被报有漏洞（www.vulnwatch.org/netcat/netcat-111.txt）。我们将分析这个有漏洞的版本（http://packetstormsecurity.org/UNIX/netcat/nc11nt.zip）。漏洞在 1.11版上修复了（www.vulnwatch.org/netcat/）。

注意 Netcat 是一个网络工具，提供远程访问的功能。一些反病毒厂商把它归为黑客工具，请留意这一点，并采取必要的防范措施。

供应商的建议描述了在使用“-e”选项时会导致远程缓冲区溢出。漏洞存在于源文件 dosexec.c 的 SessionWriteShellThreadFn 函数里。下面的代码段是我们从有漏洞的函数中选取的。

```
static VOID
SessionWriteShellThreadFn(LPVOID Parameter)
{
    PSESSION_DATA Session = Parameter;
    BYTE RecvBuffer[1];
    BYTE Buffer[BUFFER_SIZE];
    BYTE EchoBuffer[5];
    DWORD BytesWritten;
```

更多资源请点击访问稀酷客(www.ckook.com)

```
    DWORD BufferCnt, EchoCnt;
    DWORD TossCnt = 0;
    BOOL PrevWasFF = FALSE;
    BufferCnt = 0;
    //Loop, reading one byte at a time from the socket.
    while (recv(Session->ClientSocket, RecvBuffer, sizeof(RecvBuffer), 0) != 0)
        {
            EchoCnt = 0;
            Buffer[BufferCnt++] = EchoBuffer[EchoCnt++] = RecvBuffer[0];
            if (RecvBuffer[0] == '\r')
                Buffer[BufferCnt++] = EchoBuffer[EchoCnt++] = '\n';

            //Trap exit as it causes problems
            if (strnicmp(Buffer, "exit\r\n", 6) == 0)
                ExitThread(0);
            //
            //If we got a CR, it's time to send what we've buffered up down to the
            //shell process.
            if (RecvBuffer[0] == '\n' || RecvBuffer[0] == '\r') {
                if (! WriteFile(Session->WritePipeHandle, Buffer, BufferCnt,
                &BytesWritten, NULL))
                {
                  break;
                }
                BufferCnt = 0;
            }
      }
      ExitThread(0);
}
```

这个函数包含一个有两个可能的退出点的接收循环。第一个退出条件是缓冲区收到包含exit\r\n的字符串。第二个退出条件是接收的字节中有\n或\r，并且调用WriteFile失败。

注意　在Windows下，Netcat发送\n表示新行。终止命令是exit\r\n。为了终止新行需要发送\r\n。

在IDA Pro里打开nc.exe后，调试器菜单项是可用状态。如果可执行格式是调试器支持的，调试器菜单将变得可见。

需要配置处理选项，特别是需要配置命令行参数，因为这个漏洞只有在使用“-e”选项时才会出现。为了使用这个选项，我们还必须使用-l（listening，侦听）以及-p（端口号）选项。-e选项执行一个程序，并把传送通过网络输入的数据。我们不需要被执行的程序做任何事情，因此我们选用more程序。图5-2显示了使用我们的命令行参数的典型的处理选项。

更多资源请点击访问稀.酷客(www.ckook.com)

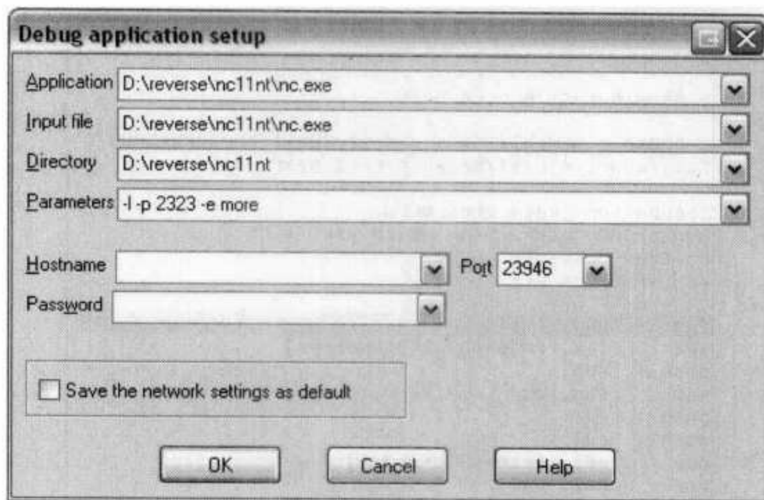


图 5-2 调试器应用程序设置

注意

如果我们调试的是 dll，设置和上面有些区别。dll 应该输在 Input file 框里，使用这个 dll 的应用程序应该输在 Application 框里。

这种类型的设置在处理 IE 时很常见。iexplore.exe 所做的工作其实非常少，真正的重活都交由 dll 处理了。

调试器的热键如下：

- F9 启动调试器/继续处理（如果已经在调试）；
- F2 设置/移除断点；
- F7 步入；
- F8 步过；
- CTRL+F7 运行到返回指令；
- F4 运行到光标所在位置；
- CTRL+F2 终止进程。

我们在输入表中查找 WriteFile 时发现了 `SessionWriteShellThreadFn` 函数。根据交叉引用，我们确定这个函数的地址是`.text:00401520`。我们随即在这个函数的开头设一个断点。图 5-3 显示了这个函数的图形视图。为了便于阅读，我们把这个函数改名为 `SessionWriteShell-ThreadFn`，并把栈变量 buf 改成 `RecvBuffer`。

按下 F9 启动调试器。不同的窗口将重新排列。调试器用传递的参数运行 nc.exe。因为还没有外来的连接，我们设置的断点还没有被触发。

更多资源请点击访问稀.酷客(www.ckook.com)

```
; Attributes: noreturn

; DWORD __stdcall SessionWriteShellThreadFn(LPVOID)
SessionWriteShellThreadFn proc near

RecvBuffer= byte ptr -0CDh
NumberOfBytesWritten= dword ptr -0CCh
Buffer= byte ptr -0C8h
arg_0= dword ptr  4

sub     esp, 0D0h
lea     eax, [esp+0D0h+RecvBuffer]
push    ebp
mov     ebp, ds:__imp_recv
push    esi
push    edi
mov     edi, [esp+0DCh+arg_0]
xor     esi, esi
push    esi             ; flags
push    1               ; len
mov     ecx, [edi+0Ch]
push    eax             ; buf
push    ecx             ; s
call    ebp ; __imp_recv
test    eax, eax
jz      short loc_4015BC
```

图 5-3　SessionWriteShellThreadFn 函数

运行 cmd.exe“壳”，我们将用另一个 Netcat 连接被调试的 Netcat。为了与被调试的 Netcat 区分开，我们把这个 Netcat 称为 Netcat 客户端。使用下面的命令行：

```
nc localhost 2323
```

这时，我们设置的断点将被触发。寄存器窗口将包含类似于图 5-4 所示的数值。栈窗口将与图 5-5 所示类似。

图 5-4　寄存器窗口

更多资源请点击访问稀.酷客(www.ckook.com)

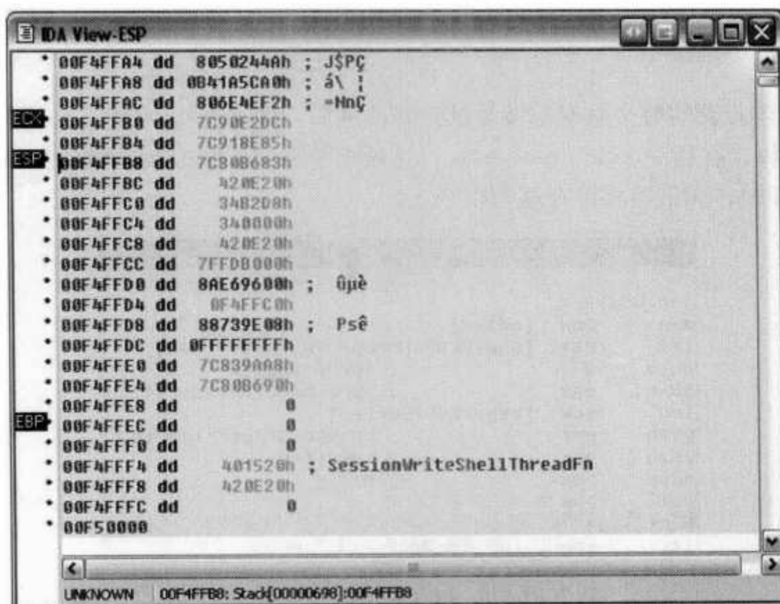


图 5-5 栈窗口

寄存器窗口的左边显示寄存器的值，右边是这些值的解释。可以在值框里按下鼠标右键或者直接在值框里输入来改变寄存器的值。在寄存器和大多数地址上按下鼠标右键会出现新视图。这个视图显示的可以是汇编或十六进制格式。

用 F8 键单步（步过）调试可以避免进入真正的 recv 调用。recv 调用在接到数据前一直被阻塞。在 Netcat 客户端输入 **hello \n**。recv 调用将返回，因为现在它收到了数据。我们可以继续单步调试。

- 图 5-6 中的基本块做下面这些事情。
- 从 recv 调用读取单个字节，并把它放入 al 寄存器里。
- 把这个字节写入缓冲区。
- 递增缓冲区计数器（在寄存器 esi 中）。

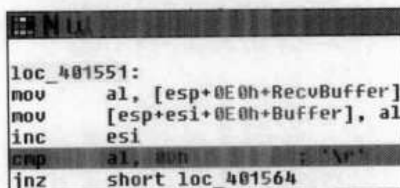


图 5-6 从缓冲区里面读写数据

在图 5-6 显示的基本块中，我们可以在指令 cmp al,0x0d 上设置另一个断点。按 F9（继续）运行这个程序到另一个断点被触发。执行将停在我们刚设的断点上。注意 esi 的值在每个字符后

更多资源请点击访问稀.酷客(www.ckook.com)

都会递增。这个字符在寄存器 al 中仍可以看到。在这个断点上右击并选择 disable breakpoint（禁用断点）。

来自 Netcat 客户端的整个命令都被复制到缓冲区后，将调用 WriteFile。图 5-7 显示了包含这个调用的基本块。在指令 text eax,eax（这条指令在 WriteFile 调用之后）上设置断点。当我们继续时（F9），调试器将停在这条指令上。

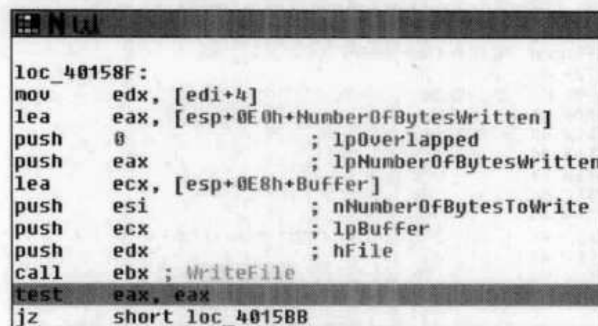


图 5-7　包含调用的基本块

我们知道这个缓冲区的大小，按照图 5-7 的显示它是 0xc8。不过栈看起来和预计的不一样，因为这个函数是被 CreateThread()调用的。

在如下图 5-8 显示的基本块中，我们可以在指令 cmp al,0x0d 上设置一个条件断点。为了设置条件断点，在禁用的断点上右击并选择 Edit breakpoint（编辑断点）。在图 5-8 的 Condition 框里输入 esi==0xc8 || esi==0xcc。在到达这个缓冲区最后位置并覆盖接下来的 DWORD 时将触发这个断点。

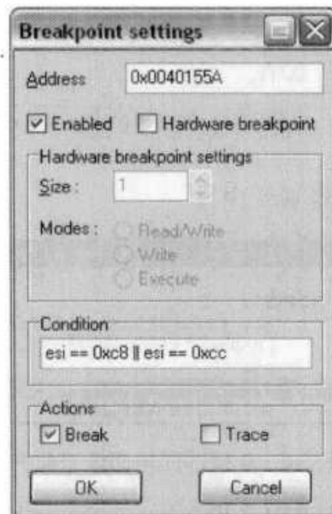


图 5-8　条件断点

更多资源请点击访问稀.酷客(www.ckook.com)

警告 Condition 框接受一个做比较的 IDC 语句。人们常犯的错误是使用单等号=（赋值），其实那里应该是双等号==（比较）。这在 C 代码中是一个经典的 bug。

我们需要发送更多的数据。从栈视图看 272 个字节将超出这页的尾部。注意，其他类型的机器或操作系统的内存布局可能不一样。发送数据最简便的方法是在文本编辑器里输好字符串，然后把它粘贴到 Netcat 客户端。

断点被触发后，栈看起来像图 5-9 那样。这时页面还剩下非常小的空间，但仍有数据需要读入。接下来的断点在栈第一次被破坏时触发。当允许这个程序继续执行时，我们会看到警告信息，然后出现如图 5-10 所示的异常。

```
IDA View-ESP
* 00F4FF78 dd  41414141h ; AAAA
* 00F4FF7C dd  41414141h ; AAAA
* 00F4FF80 dd  41414141h ; AAAA
* 00F4FF84 dd  41414141h ; AAAA
* 00F4FF88 dd  41414141h ; AAAA
* 00F4FF8C dd  41414141h ; AAAA
* 00F4FF90 dd  41414141h ; AAAA
* 00F4FF94 dd  41414141h ; AAAA
* 00F4FF98 dd  41414141h ; AAAA
* 00F4FF9C dd  41414141h ; AAAA
* 00F4FFA0 dd  41414141h ; AAAA
* 00F4FFA4 dd  41414141h ; AAAA
* 00F4FFA8 dd  41414141h ; AAAA
* 00F4FFAC dd  41414141h ; AAAA
* 00F4FFB0 dd  41414141h ; AAAA
* 00F4FFB4 dd  41414141h ; AAAA
* 00F4FFB8 dd  7C80B683h
* 00F4FFBC dd    420E20h
* 00F4FFC0 dd    34B208h
* 00F4FFC4 dd    340000h
* 00F4FFC8 dd    420E20h
* 00F4FFCC dd  7FFDC000h
* 00F4FFD0 dd  8AE67600h ;  vµè
* 00F4FFD4 dd   0F4FFC0h
* 00F4FFD8 dd  88F32D48h ; H-=ê
* 00F4FFDC dd 0FFFFFFFFh
* 00F4FFE0 dd  7C839AA8h
* 00F4FFE4 dd  7C80B690h
* 00F4FFE8 dd          0
* 00F4FFEC dd          0
* 00F4FFF0 dd          0
* 00F4FFF4 dd    401520h ; SessionWriteShellThreadFn
* 00F4FFF8 dd    420E20h
* 00F4FFFC dd          0
* 00F50000
* 10000000 ; ================================================
  10000000
UNKNOWN    00F4FF8C: Stack[0000135C]:00F4FF8C
```

图 5-9 栈

更多资源请点击访问稀.酷客(www.ckook.com)

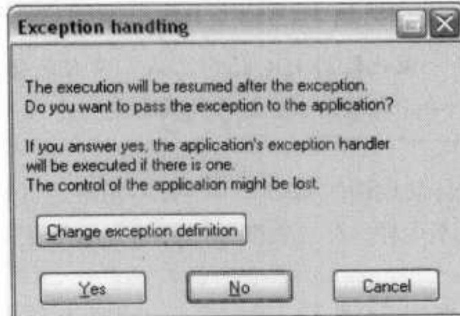


图 5-10 异常

此前异常被配置成在触发时停止程序。我们可以进入 Change exception definition（更改异常定义），并选择 Pass to application（传送给应用程序）。参见图 5-11。

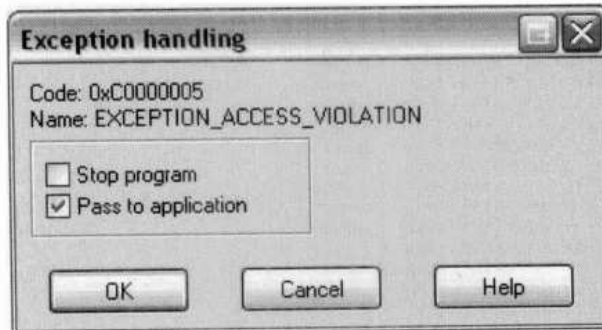


图 5-11 处理选项

当这个异常出现时，程序将停止。IDA Pro 在日志窗口里已记录下这个异常：

nc.exe: The instruction at 0x401555 referenced memory at 0xF50000. The memory could not be written (0x00401555 -> 00F50000)

```
Debugger: Thread terminated: id=00001660 (exit code = 0xC0000005).
Debugger: Thread terminated: id=000017AC (exit code = 0xC0000005).
Debugger: Process terminated (exit code = C0000005h).
```

5.5 堆和栈的访问和修改

检测内存崩溃对逆向工程师来说非常重要。栈和堆溢出就是覆盖和崩溃内存的攻击。

我们可以用调试器检测内存崩溃，也可以采用手动的方式，不过某些检测方式用脚本或插件会更好一些。

从 Visual Studio 2003 开始，微软为编译器提供了栈 cookie（用 GS 命令行切换）。在函数入口处，栈 cookie 被放到栈上。这个 cookie 是从全局安全 cookie——__security_cookie 计算而来的，

更多资源请点击访问稀.酷客(www.ckook.com)

并把它与 esp 寄存器做 xor 运算。在函数退出过程中，再把栈 cookie 与 esp 寄存器做 xor 运算。得到的结果应该是__security_cookie，这个值传递给__security_check_cookie()函数。如果传来的值与__security_ cookie 匹配，那么__security_check_cookie()返回，允许原来的函数继续执行。更详细的解释参见http://uninfomed.org/index.cgi?v=7&a=2&p=1。

这种防护方法的原理是，如果栈已崩溃 cookie 检查将失败。如果检查失败，进程将被终止，其退出代码为 0xc0000409。为了捕获这个栈崩溃，我们可以在__security_check_cookie()里设置断点，如图 5-12 所示。另一个选择是直接在__report_gsfailure()函数上设置断点。__security_check_cookie()函数一般被静态编译，其地址视二进制程序的具体情况而定。

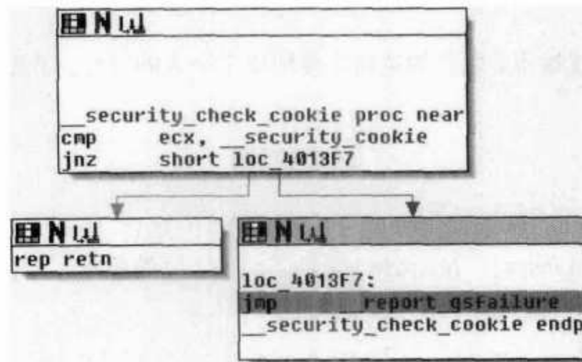


图 5-12 设置一个断点

检测堆崩溃的方法对所用的操作系统依赖更大。Windows XP SP2 、Windows 2003 和 Vista 具有多种内置的堆保护方法。不过，与 GS 栈保护措施不同，堆保护属于操作系统的一部分。现在已经有多种技术可以绕过堆保护，但来自漏洞检查工具 fuzzer 的数据仍非常有可能被这些保护措施拦截。

多重检查及简单的断点可能还不够，有基于调试器的脚本或插件就很理想了。可以钩住堆函数来提供分配的数据，钩住保护函数来报告崩溃情况。对那些没有这样周全的保护函数的系统来说，可以加上包含检查的被钩住的函数。检查会尽可能快地向我们报告崩溃，而不是停止攻击。

警告

调试器可以改变进程的环境与行为。

从调试器启动的进程使用调试堆，这点与正常启动的进程不同。对进程的攻击不会受此影响。当查找堆崩溃时，这些差异非常重要。为了禁用调试堆，把环境变量 _NO_DEBUG_HEAP 设为 1。

```
set _NO_DEBUG_HEAP=1
```

微软的 gflags.exe 实用程序允许设置许多调试选项。gflags.exe 是 Windows 调试工具的一部分（www.microsoft.com/whdc/devtools/debugging/default.mspx）。

更多资源请点击访问稀.酷客(www.ckook.com)

操作系统代码可以有调试器检查。kernel32.Unhandled Exception Filter 会根据是否存在调试器来更改其行为。这个行为最初在 Dave Aitel 的文章 MSRPC Heap Overflow –Part II 中被提及，随后出现在 *The shellcoder's handbook* 中。

5.6 其他调试器

IDA Pro 中的调试器非常有用。你可以用它访问静态分析、重命名的函数，以及已经经历逆向工程的代码部分。像其他的工具一样，每个人都有自己的偏好，对逆向工程师来说，可供选择的调试器可真不少。

每种调试器都有其优缺点，它们的选用主要取决于个人的喜好。下面大致介绍了其他一些常见的调试器。

5.6.1 Windbg

来自微软的 Windows Debugging Tools 是一个调试器的集合（http://www.microsoft.com/whdc/devtools/debugging/default.mspx）。Microsoft 提供了两种不同的版本：32位和 64 位版本。

32 位的 Windows Debugging Tools 可以运行在：

- Windows NT 4.0；
- Windows 2000；
- Windows XP (32 位或者 64 位)；
- Microsoft Windows Server 2003 (32 位或者 64 位)；
- Windows Vista （32 位或者 64 位）；
- Windows Server 2008 （32 位或者 64 位）。

64 位的 Windows Debugging Tools 可以运行在：

- Windows XP (64 位版本)；
- Microsoft Windows Server 2003 (64 位版本)；
- Windows Vista（64 位版本）；
- Windows Server 2008（64 位版本）。

64 位的 Windows Debugging Tools 只适用于调试原生的 64 位应用程序。

尽管包括了其他的调试器，例如 NTSD、CDB 和 KD，但是 Windbg 是你最有可能用到的调试器。Windbg 是一个用户和内核模式调试器，最主要的优势之一是与 Windows 的无缝集成。

5.6.2 Ollydbg

Ollydbg 是一个免费的 win32 用户模式调试器，可从www.ollydbg.be 下载。尽管没有提供源代码，但提供了 SDK。有很多为 Ollydbg编写的插件。

更多资源请点击访问稀.酷客(www.ckook.com)

Ollydbg 在逆向工程师间非常流行，可以说它就是从逆向工程师的角度设计的。它有很多著名的插件，包括脚本、反反调试（anti-anti-debugging）及跟踪，也有大量的 Ollydbg 教程，其涉及内容从基本的逆向分析，到寻找安全漏洞，到破坏软件保护。

当前的正式版本是 1.10，不过作者已经不再对这个版本提供支持了，他正全力开发 2.0 版。这个调试器被报有多个漏洞，包括格式化串漏洞（CVE-2004-0733）。有些加壳程序就利用这些漏洞防止被调试。不过已经有人发布修补这些漏洞的插件了，其中最有名的是 Olly Advanced（www.openrce.org/downloads/details/241/Olly_Advanced）。

5.6.3 immdbg

Immdbg（Immunity Debugger）是一款 Immunity 公司（http://www.immunityinc.com/productsimmdbg.shtml）发布的免费调试器。当你第一次运行 Immdbg 时，会注意到它附带广告，这些广告一般都是安全公司购买用来网罗安全产业人才的。如果你觉得 Immdbg 与 Ollydbg 类似，并不是你看走眼了，因为 Immunity 公司被授权可以访问 Ollydbg 的源代码，他们在此基础上开发出 Immdbg，增加了与开发利用相关的功能。

有了源代码许可，他们可以修复原来的漏洞，并新增了图形工具、命令行及远程调试等功能，最引人注目的新功能是内置 Python 脚本支持。安装包中包含一些演示 Python API 的脚本，Immdbg 论坛里也有用户发布脚本。

5

5.6.4 PaiMei/PyDbg

PaiMei 是一个逆向工程框架（http://paimei.openrce.org/）。它是用 Python 语言编写的，用脚本处理来自 IDA Pro 分析的数据。PaiMei 的关键组件之一是 PyDbg。PyDbg 是一个支持脚本语言的调试器，也是用 Python 写的，可以与 IDAPython（http://d-home.net/idapython/）集成使用。

IDAPython 是 IDA Pro 的插件，支持脚本。它封装了许多 IDC 及 SDK 函数，但遗憾的是，还有很多调试器调用没有被封装（或者说还没来得及封装）。不过，为了调试及把运行时分析与 IDA Pro 静态分析结合起来，IDAPython 可以使用 PyDbg。

5.6.5 GDB

到现在为止讨论的调试器都是运行在 Windows 上的。GDB（GNU Project Debugger）则在大多数 UNIX 系统上可用。GDB 主要是源代码级调试器，不过也可以在汇编级操作。

GDB 使用基于文本的接口，但现在已有多个图形化的前端程序，它们用 MI（Machine Interface，机器接口）与 GDB 通讯。通过使用 MI 可以用脚本语言驱动 GDB。

更多资源请点击访问稀.酷客(www.ckook.com)

工具和陷阱

加壳工具

什么是加壳程序？加壳程序大多运行在Win32环境下，它们压缩可执行程序，运行时再在内存里对它们解压缩。因此，为了分析加过壳的二进制文件，首先需要实际的解压缩后的映像。常见的开源加壳程序是UPX，见http://upx.sourceforge.net。UPX可以压缩/解压缩，也可以把一个加壳的二进制文件复原。大多数加壳程序基本都不是如此设计的，且通常需修改UPX以禁止对称行为。

OEP（Original Entry Point，在原始入口点）被定义之前，二进制程序通常会在调试器或模拟环境中运行。那时把内存及恰当的PE头部做丢弃处理。因为输入表通常被破坏了，因此需要加入输入表。这是提取原始映像的基本方法。时至今日，加壳/脱壳竞赛仍在上演。

除恶意软件外，有些程序也会用到加壳程序。加壳程序的目标是为逆向工程设置障碍，不过也降低了文件特征。我们将在第6章讨论一些反调试/逆向技术。软件保护通常会使用加壳程序，这些软件从共享软件到商业软件包都有。处理DRM的软件通常会使用加壳和/或反调试技术。Skype是一个流行的通话程序，也大量运用了此类技术。为了评估或审计加壳后的软件，脱壳的映像是必不可少的。

5.7 小结

IDA Pro的调试器非常强大，比静态分析更有助于理解程序。这个调试器支持在大多数操作系统上进行本地及远程操作。

和其他调试器相比，IDA Pro调试器的优势是它拥有逆向工程所需要的全部功能，包括重命名函数、表及局部变量。

如果你觉得IDA Pro不顺手，也可以尝试其他调试器。

更多资源请点击访问稀酷客(www.ckook.com)

第 6 章

反逆向技术

本章内容：

- 调试
- 举例阐述
- 混淆技术

☑ 小结

更多资源请点击访问稀.酷客(www.ckook.com)

6.1 导言

反调试技术的出现是可以预料的。在人们开始逆向分析应用程序后，反逆向分析技术的出现只是时间上的问题。有些人努力想使逆向分析变得更难，或根本不可能。反调试像逆向工程或汇编编程一样，也是一种艺术，其根本目的就是阻止人们逆向分析应用程序。不过在大多数情况下，这些努力要么十分乏力，要么困难重重。起初你可能会认为只有恶意软件才会阻止你逆向分析，但实际上它（反调试）无处不在。确实有些合法的工作岗位对拥有反逆向技术的人们虚位以待，特别是在视频游戏行业。在本章，我想全面介绍一下反调试及反反汇编技术。没搞错吧？这些技术可是针对我们的！而且，很多时候它们确实会阻碍我们解决出现的问题。没搞错，知己知彼，方能百战百胜。不过大家一定要记住：只要有足够的时间与动力，逆向工程师将是最后的赢家。

首先，如果我们真的想理解反逆向技术，那么掌握一些调试及反汇编知识将大有裨益。这当然不是说要全面了解，我们将只是大致介绍它怎样工作，从而更好地理解反逆向技术。

6.2 调试

要真正理解调试，了解各种中断及调试寄存器是很有必要的，特别是有关进程状态变化的。*Intel Software Developers Manual* 在这方面依然是最好的参考资料，我知道的与它相比，不过是小巫见大巫。不过，IA-32 平台处理调试基本上是通过二者（中断或调试寄存器）之一实现的。

首先介绍一下调试寄存器。IA-32 平台有 8 个调试寄存器，最多可以监视 4 个地址。访问这些寄存器是通过 mov 指令（调试寄存器暗中充当目的操作数或源操作数）的变体实现的。要知道，访问这些寄存器是需要有 RING0 特权的，这当然是一个限制，但考虑其影响极大，这样做也是有道理的。对每一个断点，有必要具体指定涉及的地址、位置的长度（范围在一个字节与双字之间）、处理程序（当生成一个调试异常时）以及这个断点是否被启用。前 4[①]个调试寄存器，DR0 到 DR3 可以包含 3 个 32 位地址（定义了断点应该出现的地址）。接下来的 2 个调试寄存器 DR4 和 DR5 取决于操作模式可以交替使用。当 DE（Debugging Extension，调试扩展）标志被设在 CR4（Control Register 4，控制寄存器 4）里时，DR4 和 DR5 被保留，试图引用它们时，将引发一个无效操作码异常。如果这个标志没有被设置，DR4 和 DR5 则作为别名替代调试寄存器 6 和 7。

DR6（Debug Register 6，调试寄存器 6），作为调试状态寄存器指示对最后发生的调试异常进行条件检查的结果。用位模式访问 DR6，位 0 到位 3 与前 4 个调试寄存器相关。这些位指示哪个断点条件被满足而生成了一个调试异常。DR6 的位 13 在置位时表示下一条指令引用一个调试寄存器，连同 DR7（我们随后就会介绍）的一部分被使用。位 14 对我们来说或许是最有诱惑力的，它被置位时指示处理器处于单步模式（single-step mode），我们稍后介绍。最后使用的是位 15，它指示当调试陷阱标志被置位时，作为任何切换的结果都会抛出调试异常。最后，我们介绍

① 原文为 3，根据上下文应该为 4。

更多资源请点击访问稀.酷客(www.ckook.com)

调试寄存器 7，也就 DR7。所有的黑客都对这个寄存器垂涎欲滴，它也被称为调试控制寄存器，像 DR6 一样被理解为位字段。这个寄存器的第一个字节对应断点是否是活动的，如果是活动的则对应它的作用域。位 0、2、4 和 6 确定调试寄存器是否被启用，位 1、3、5 和 7 对应同样的断点，但基于全局作用域。在这个例子里，作用域被定义为在任务切换时断点是否保持，是否针对全局启用的断点对所有的任务都是可用的。根据 Intel 的手册，现在的处理器不支持位 8 和 9。不过在以前，可以用它们确定引起断点事件的精确的指令。接下来是位 13，这是一个比较有意思的位，它允许在访问调试寄存器本身之前中断。最后是位 16 到 31，这些位确定什么类型的访问会引起断点，以及所在地址的数据长度。当 CR4 中的 DE 标志被置位时，位 16 到 17、20 到 21、24 到 25、28 到 29 被解释成下面的意思：

```
00 - Break on execution
01 - Break on write
10 - Break on I/O read or writes
11 - Break on read and writes but not instruction fetches
```

当 DE 标志没有置位时，除了 10 的值没有定义外，其他的解释同上面一样。位 18 到 19、22 到 23、26 到 27、30 到 31 对应各种断点的长度，值 00 表示长度为 1，值 01 表示 2 字节长度。在 32 位平台上 10 未被定义，在 64 位平台上它表示 8 字节长度。你可能已经猜到了，11 表示长度为 4 字节。

现在，把所有这些位与 DR0 至 DR3 联系起来似乎比较困难，但实际并不是这样的。每个 2 位组合对应 DR0 至 DR3 范围里特定的连续的寄存器。这些长度必须对齐某些边界（取决于它们的大小）——例如，16 位需要以字为界，32 位以双字为界。这由处理器通过掩码地址的低位相关地址来强制执行，因此，一个未对齐的地址将不会产生预期的结果。如果访问起始地址加上它的长度范围内的任意地址，将产生异常，通过使用 2 个断点有效地允许未对齐的断点；每个断点被适当地对齐，在它们两个之间包含了讨论的长度。这里还有一点值得一提。当断点访问类型只是执行时，应当把长度设为 00，其他的值导致未定义的行为。

注意

有趣的是，调试寄存器本身并没有受到很多关注。不过，私下里还是有很多利用它们的 rootkits 和后门。比如说，可以在指向进程结构的指针上设置一个全局访问断点，在进程的链表里隐藏进程。当访问这个地址时，将出现调试异常，并重定向到它们的处理程序并执行任意多的任务，包括返回列表里下一个进程的地址。

更糟的是，它们可以启用 DR7 里的 GD 标志，并导致访问调试寄存器本身并引发异常，阻挠审查寄存器来检查它们当前的配置。

我们在描述寄存器的过程中一直提到调试异常，但还没有做深入介绍。IA-32 处理器在 IDT（在第 2 章介绍过）中包含指定的中断向量。处理器用两个中断向量处理这些异常，分别是中断 1 和中断 3，对应调试和断点异常。调试异常，即 INT 1，由多个事件生成（还应该查询 DR6 及

更多资源请点击访问稀.酷客(www.ckook.com)

DR7，精确确定出现的是什么类型的事件）。在异常的进程里，有两个常见的类——故障和陷阱。从本质上讲，它们的差异是在处理程序得到控制时，生成中断的指令是否已被执行。在故障里，控制权首先交给处理程序，但在陷阱里，执行控制在引起异常的指令被生成后传给处理程序。关于这两个类，根据条件还可以细分：指令断点、数据和I/O断点、一般检测、单步和任务切换条件。指令断点、一般检测和所谓的任务切换条件属于故障类；而数据和I/O断点和单步条件属于陷阱类。

指令断点类条件是最高优先级的异常，当企图执行DR0至DR3中提及的地址上的指令时出现。我们说这些异常有最高优先级，意味着它们优先接受服务；不过，也有这样的情况，这些事件可能根本不会被触发，我们稍后介绍。现在，如果你回忆起对EFLAGs寄存器的描述，可能还记得有一个恢复标志。问题是，因为这个异常属于陷阱类异常，所以有可能重新抛出调试异常。这就是恢复标志开始发挥作用的地方，因为它将防止调试异常的循环。深入研究时我还将对此做简短介绍。另一个故障类条件是一般检测条件；在DR7里相关位置位时抛出它，保护对调试寄存器的存取。

数据和I/O断点是陷阱类条件，对DR0至DR3范围里地址进行数据存取时生成。数据存取本质上是任意条件（但不是执行尝试）。引起陷阱类的指令被执行后，IA-32处理器在陷阱类事件出现时包含一个小动作。例如，假如你在地址X（它的值是0）上设置一个写断点，然后应用程序修改地址X，把它的值设为1。当异常处理程序接到控制权时，地址X处的值将是1，而不是0。Intel手册建议，与原始值相互影响的应用程序应当在断点时保存它，尽管这样做会产生有趣却与主题无关的同步问题。另外，这些断点（如指令断点）并不总是精确的；例如，反复执行某些SIMD指令可能引起异常被抛出，直到第2个迭代结束。

单步异常也是陷阱类条件，是调试时经常碰到的条件之一。当EFLAG寄存器里的陷阱标志置位时，这些条件被引发。就像其他类型的异常一样，单步异常也有它自己的怪癖。例如，通常来说在执行各种任务的进程里是不会修改陷阱标志的，但有些指令（像软中断和INTO指令）会清除陷阱标志。这实际上意味着，为了维持控制调试器必须模拟这些指令，但并不能直接执行它们，如果它们希望继续产生单步异常的话。最后，如果在新任务TSS里陷阱标志被置位，在任务切换后将产生任务切换异常。这个异常在任务切换之后，但在新任务里第一条指令之前被抛出。

除INT 1外，还有断点异常（INT 3）。断点异常比较有意思，它允许扩充断点以传递调试寄存器支持的数值。不过，这需要修改内存——这是一个命门，我们稍后会谈到怎样利用它。现在，我们只需要知道它存在就可以了。

看到这里，有灵感且富有想象力的读者可能已经开始关注能被检查的条件以及可以避免中断的方法。然而，人们买这本书不是为了培养想象力，而是要学习技术，因此，在本章剩余部分中我们将介绍一些可能发生的难题。我们将探索大多数基于RING3的、规避反汇编和调试的实现，但为了增加一些趣味，也会涉及一些基于RING0的要领。

更多资源请点击访问稀.酷客(www.ckook.com)

6.3 举例阐述

因为本章介绍的主题稍微有些棘手，要想解释清楚是比较困难的，特别是只有一章的篇幅。我们用例子配合说明：我写的一个简单的程序一个无聊的网络 RPC 服务器及客户端。我们准备处理这个（服务器端组件）应用程序，尽可能加大它被逆向的难度。实际上，我们会有两个程序：原来的程序和加工后的程序。我们的想法是：通过对比原程序与具备反逆向能力的新程序逆向分析反 RCE 进程；通过这样的对比，你就能大致了解反逆向的所作所为以及如何应对。本章使用的例子代码可以从 Syngress 网站上下载。我们不关注这个应用程序的客户端，它存在的意义非常简单，就是为你所用进行试验，为我所用检验程序是否正常工作。

不要瞎忙乎，在动手之前先了解一下程序，还是那句话，知己知彼，方能百战百胜。图 6-1 显示了控制流程图的相关部分，选择菜单 **View > Graphs > Control Flow**，或者按下 **F12**（热键）就可以生成这样的流程图。因为整个图太大，在屏幕上没法一次性全部显示出来，但我希望你能从此图了解程序控制流的一些端倪。我强烈建议你把源代码下载下来，编译并仔细检查它。

像你看到的，这明显是以某种形式处理 RPC 的程序，当我们更进一步查看时，一系列的 API 调用显示它是一个服务器端组件。当你检查代码时，会注意到有多个字符串常量，这主要是因为我们并没有想着混淆这个程序的功用等情况。现在，让我们改变它！关于这一点，如果你还有些茫然，就仔细阅读代码吧，你会找到感觉的。本章是介绍反逆向的，重点不是编写一个 RPC 服务器，因此我不会对此介绍得很详细。

6.4 混淆技术

像我们以前提到过，关于这个程序里还有什么正在进行中（或者至少有什么看起来好像在进行中），我们是没有疑惑的。（在匆忙之间，需谨慎判定程序的目的，不过，在这个例子里，我们看到的就是实际的情形。）尽管有很多技术可以用，但我们还是选择一个较简单却很有效的技术来混淆代码。

这类技术很常见，实际上你应当把它们视为通用的例程，它们不像其他类型的安全技术那样会给你带来很多难题。我们这样做，主要是想提高阅读这个代码的门槛。你应当看到过，非常正派的人士称他们自己为“逆向工程师”或“干应急响应工作”，实际并不是这样的。一些人会的可能只是从二进制中提取可读的字符串，一些人可能只能读 API 调用，而另一些人可能只会根据输入区段里的数据进行一些推测。一个经常使用的技巧（特别是处理打包后的二进制文件时）是对这个二进制进行修改，如果某人试图从内存中直接丢弃映像，他只会得到残缺的数据。

考虑这些之后，让我们看图 6-2 里最新修改后的 main 例程。

只看进入点，显示的本地变量数量已表明这个新的二进制程序可能更复杂了。当然，我们也查看这段代码的其他视图，因此，当我们只看到代码前面的一系列指令时这点没那么明显（参见图 6-3）。

更多资源请点击访问稀.酷客(www.ckook.com)

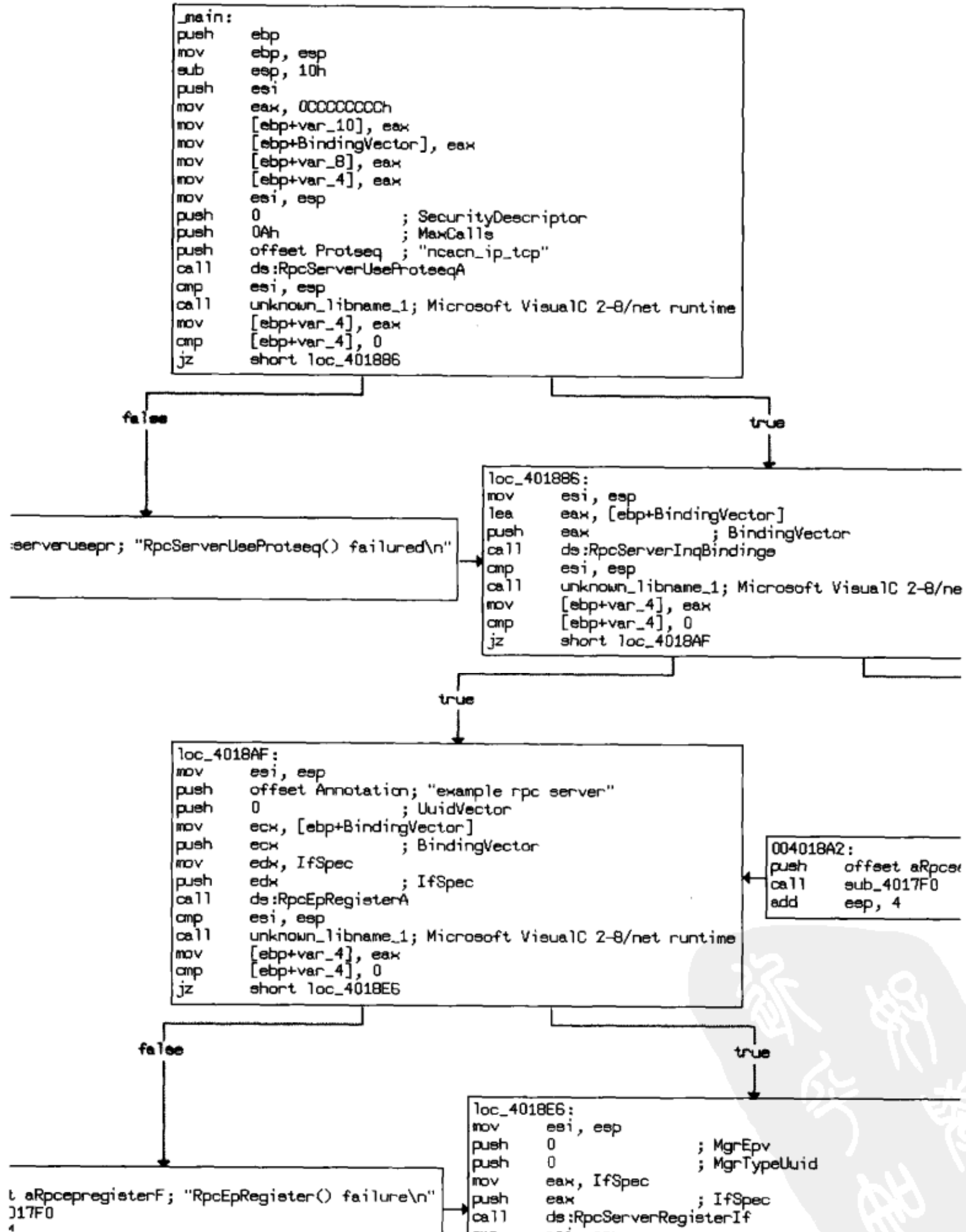


图 6-1 控制流程图

更多资源请点击访问稀.酷客(www.ckook.com)

```
; Attributes: bp-based frame

; int __cdecl main(int argc,const char **argv,const char *envp)
_main proc near

var_234= dword ptr -234h
var_218= dword ptr -218h
var_1F4= dword ptr -1F4h
var_1F0= dword ptr -1F0h
var_1EC= dword ptr -1ECh
var_1E8= dword ptr -1E8h
var_1E4= dword ptr -1E4h
var_1E0= dword ptr -1E0h
var_1DC= dword ptr -1DCh
var_1D8= dword ptr -1D8h
var_1D4= dword ptr -1D4h
var_1D0= dword ptr -1D0h
var_1B4= dword ptr -1B4h
var_194= dword ptr -194h
var_190= dword ptr -190h
var_18C= dword ptr -18Ch
var_16C= dword ptr -16Ch
var_168= dword ptr -168h
var_164= dword ptr -164h
var_160= dword ptr -160h
var_15C= dword ptr -15Ch
var_158= byte ptr -158h
var_154= dword ptr -154h
var_134= dword ptr -134h
var_130= dword ptr -130h
var_12C= dword ptr -12Ch
```

图 6-2 修改后的 main 例程

```
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4
argc= dword ptr  8
argv= dword ptr  0Ch
envp= dword ptr  10h
arg_34= dword ptr  3Ch
arg_70= dword ptr  78h

push    ebp
mov     ebp, esp
sub     esp, 348h
mov     eax, dword_404000
xor     eax, ebp
mov     [ebp+var_C], eax
push    ebx
push    esi
push    edi
mov     eax, ds:dword_403118
mov     [ebp+var_6C], eax
mov     ecx, ds:dword_40311C
mov     [ebp+var_68], ecx
mov     dx, ds:word_403120
mov     [ebp+var_64], dx
mov     al, ds:byte_403122
mov     [ebp+var_62], al
mov     ecx, ds:dword_403124
mov     [ebp+var_C4], ecx
mov     edx, ds:dword_403128
mov     [ebp+var_C0], edx
mov     eax, ds:dword_40312C
mov     [ebp+var_BC], eax
mov     ecx, ds:dword_403130
mov     [ebp+var_B8], ecx
mov     edx, ds:dword_403134
mov     [ebp+var_B4], edx
mov     al, ds:byte_403138
mov     [ebp+var_B0], al
mov     ecx, ds:dword_40313C
mov     [ebp+var_A4], ecx
mov     edx, ds:dword_403140
mov     [ebp+var_A0], edx
```

图 6-3 前面的一系列指令

6

更多资源请点击访问稀.酷客(www.ckook.com)

像我们提到的，代码持续变化过程中的趋势，一旦开始调用 RPC，就会把一长串的字节从数据段复制到栈上。你可能注意到，栈在下降，而数据索引是增加的，这对于用字符串常量初始化栈变量等事件是很典型的。我们稍后会再次讨论。现在，在深入研究变化之前，我希望你能看到并感受程序里的差别，就像图 6-4 所示的那样。

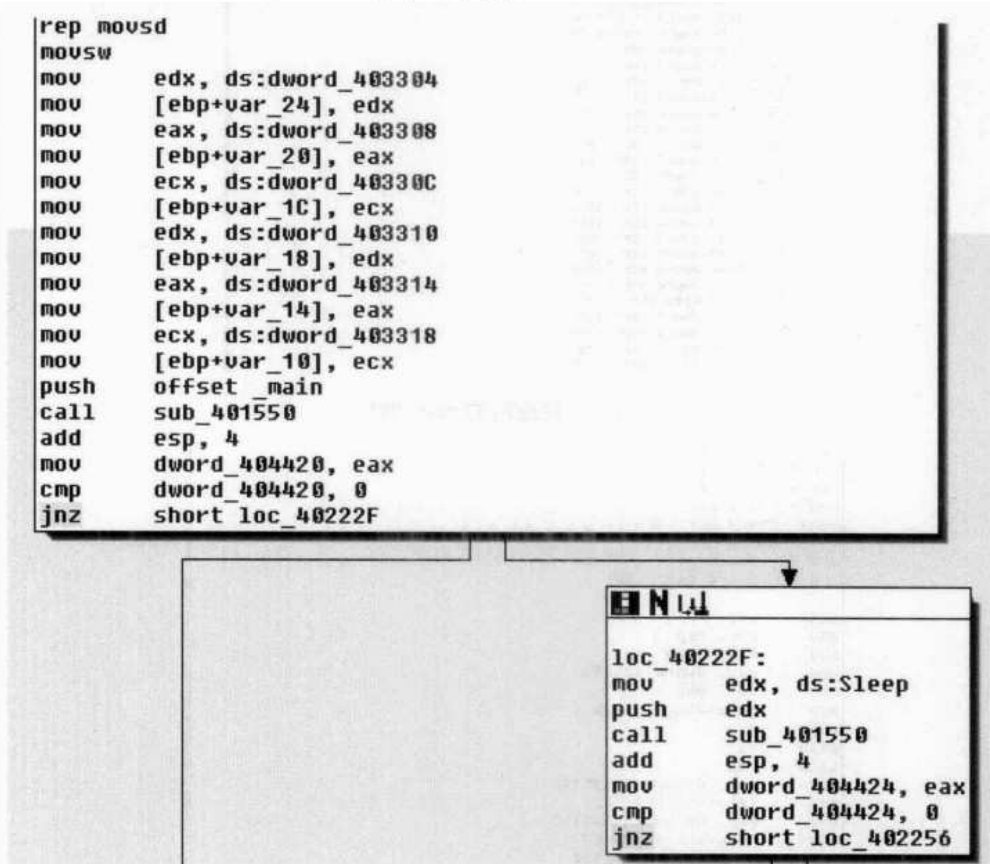


图 6-4 程序中的差别

继续看代码，我们最后会到达第一个“真正的”代码的部分即对函数 sub_40155 的调用。有两个调用指向它，第一个传递 main 函数的地址作为参数，第二个传递 kernel32 函数 Sleep() 的地址。现在我们基本上知道接下来该做什么，让我们看一下这段代码的概略图（图 6-5），就会了解这个程序的结构有多疯狂。

这里我们明白一点，不论最初这段代码显得有多么棘手，它展示的是典型结构，且对其整体结构显示的仍是布尔逻辑。复杂性大都集中在开始部分，我们会看到一系列的 if( ) { if ( ) [...] else [...] } else [...]结构。注意第一个匹配失败分支位于右边，而第二个匹配失败分支位于左边，并继续向下直到进程终止。

更多资源请点击访问稀.酷客(www.ckook.com)

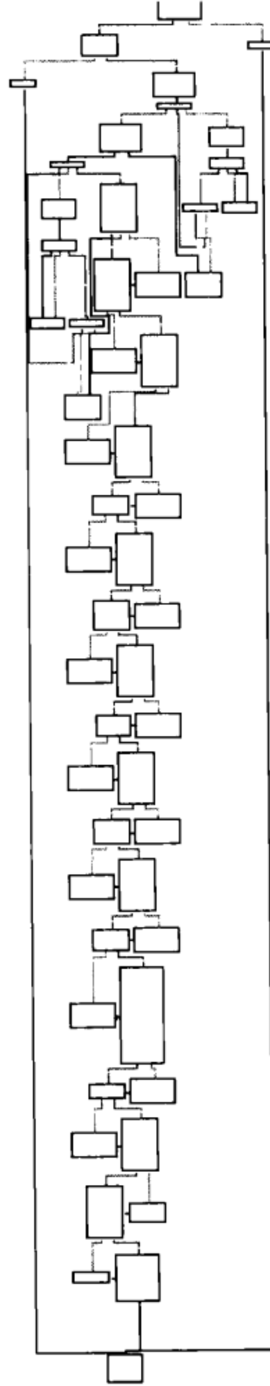


图 6-5 复杂的应用程序结构

更多资源请点击访问稀.酷客(www.ckook.com)

太好了！我们现在对接下来的工作有了一些头绪，让我们回过头再看一下它是怎样变化的，看是否能理解这个以把一长串的数据从数据段复制到栈上开始，纠缠在一起的谜题。在数据的开头，我们有地址 loc_403118，让我们跳到那里看能确定什么（图 6-6）。

```
.rdata:00403118 dword_403118     dd 2D23190Eh            ; DATA XREF: _main+16↑r
.rdata:0040311C dword_40311C     dd 3B021856h            ; DATA XREF: _main+1E↑r
.rdata:00403120 word_403120      dw 233Ch                ; DATA XREF: _main+27↑r
.rdata:00403122 byte_403122      db 0                    ; DATA XREF: _main+32↑r
.rdata:00403123                  align 4
.rdata:00403124 dword_403124     dd 0C23192Eh            ; DATA XREF: _main+3A↑r
.rdata:00403128 dword_403128     dd 3A5A5E47h            ; DATA XREF: _main+46↑r
.rdata:0040312C dword_40312C     dd 3A171A22h            ; DATA XREF: _main+52↑r
.rdata:00403130 dword_403130     dd 2B260F7Eh            ; DATA XREF: _main+5D↑r
.rdata:00403134 dword_403134     dd 1E561F43h            ; DATA XREF: _main+69↑r
.rdata:00403138 byte_403138      db 0                    ; DATA XREF: _main+75↑r
.rdata:00403139                  align 4
.rdata:0040313C dword_40313C     dd 0C23192Eh            ; DATA XREF: _main+80↑r
.rdata:00403140 dword_403140     dd 3A5A5E47h            ; DATA XREF: _main+8C↑r
.rdata:00403144 dword_403144     dd 2E0A0622h            ; DATA XREF: _main+98↑r
.rdata:00403148 dword_403148     dd 3B27146Ch            ; DATA XREF: _main+A3↑r
.rdata:0040314C dword_40314C     dd 2C401459h            ; DATA XREF: _main+AF↑r
.rdata:00403150 byte_403150      db 0                    ; DATA XREF: _main+BB↑r
.rdata:00403151                  align 4
.rdata:00403154 dword_403154     dd 1A23192Eh            ; DATA XREF: _main+C6↑r
.rdata:00403158 dword_403158     dd 38497E52h            ; DATA XREF: _main+D2↑r
.rdata:0040315C dword_40315C     dd 3A103C39h            ; DATA XREF: _main+DE↑r
.rdata:00403160 word_403160      dw 3C5Ch                ; DATA XREF: _main+E9↑r
.rdata:00403162 byte_403162      db 0                    ; DATA XREF: _main+F7↑r
.rdata:00403163                  align 4
.rdata:00403164 dword_403164     dd 0C23192Eh            ; DATA XREF: _main+103↑r
.rdata:00403168 dword_403168     dd 3A5A5E47h            ; DATA XREF: _main+10E↑r
.rdata:0040316C dword_40316C     dd 38011D22h            ; DATA XREF: _main+11A↑r
.rdata:00403170 dword_403170     dd 3A3D0E47h            ; DATA XREF: _main+126↑r
.rdata:00403174 dword_403174     dd 413342h              ; DATA XREF: _main+131↑r
```

图 6-6　地址 loc_403118

当我们检查最初在 main()函数里引用的数据时，明显看出它们不是 ASCII 字符，但它们好像是 NULL 结尾的，这从另一方面支持了用字符串常量初始化局部变量的想法。此外，我们还留意到这些数据有一些重复模式，例如，有很多序列以字节 0x0C2319 开始。这可能是弱加密的迹象，但在代码段与这个数据交互之前，我们并不能真正确定。现在先把这个想法搁置一下，但我们可以推测出，这个程序曾处理大量的 RPC 且有众多字符串常量，现在包含一系列 NULL 结尾的数据（在大多数情况下，开始的三个字节是重复的）。

现在，对这些显然加过密的数据，我们并没有太多可做的（除非我们愿意追查这些数据是在哪里使用的），我们继续顺序查看下面的操作。接下来，会碰到函数 sub_401550（图 6-7）。

当我们检查 sub_401550 的入口时，发现一个比 main()看起来要正常一些的函数。你可能想起来了，在 main()的第一部分，它被调用两次，一次是以 main()的地址为参数，另一次是以指向 Sleep()函数的指针为参数。这段代码的用户部分以 cmp 指令开始，朝着第一个框的结尾测试 arg_0 是否为 0。在编译器生成的代码里需要注意的是，其中有一个异常处理程序的设置部分。

更多资源请点击访问稀.酷客(www.ckook.com)

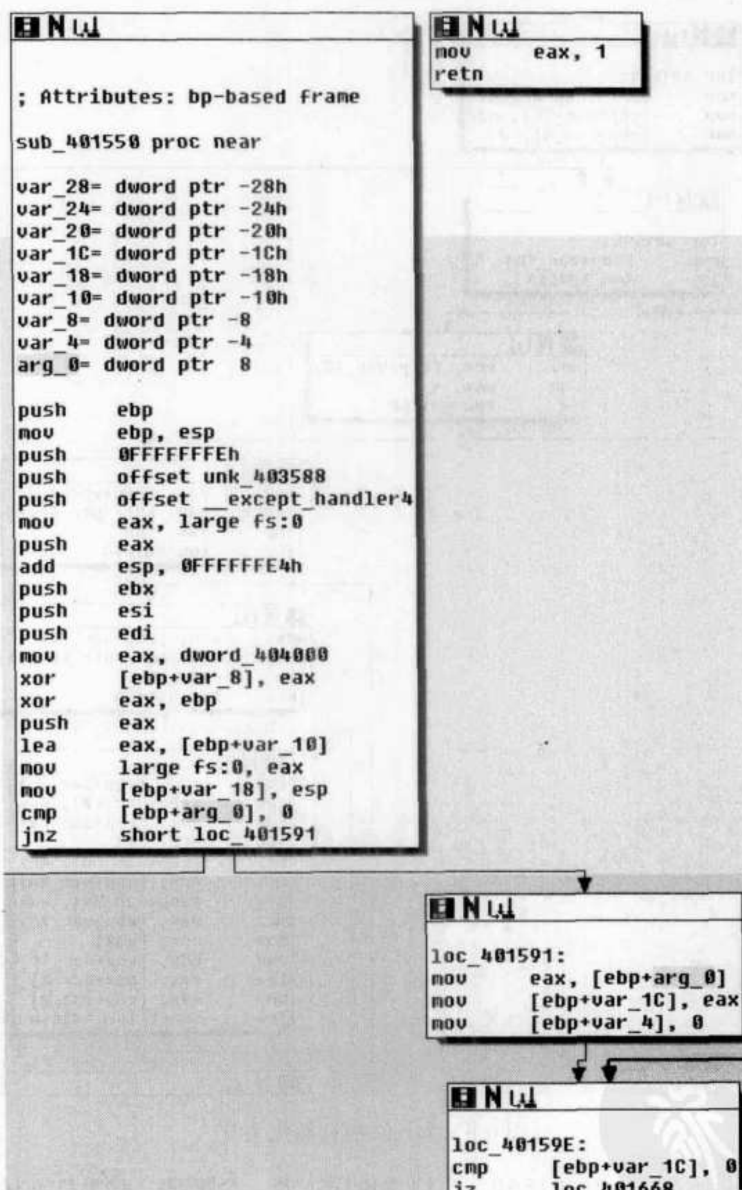


图 6-7 函数 sub_401550

因此，我们看到的第一件“真实的”事情是测试这个参数是否为 0，更确切地说是检查这个指针是否为 NULL，如果是则跳到左边的分支，最后可能是一个异常中止。如果这个指针非 NULL，则跳到右边的分支，在 loc_401591 处我们看到变量 var_1C 被初始化成这个例程参数的值，var_4 被初始化成 0（图 6-8）。

更多资源请点击访问稀.酷客(www.ckook.com)

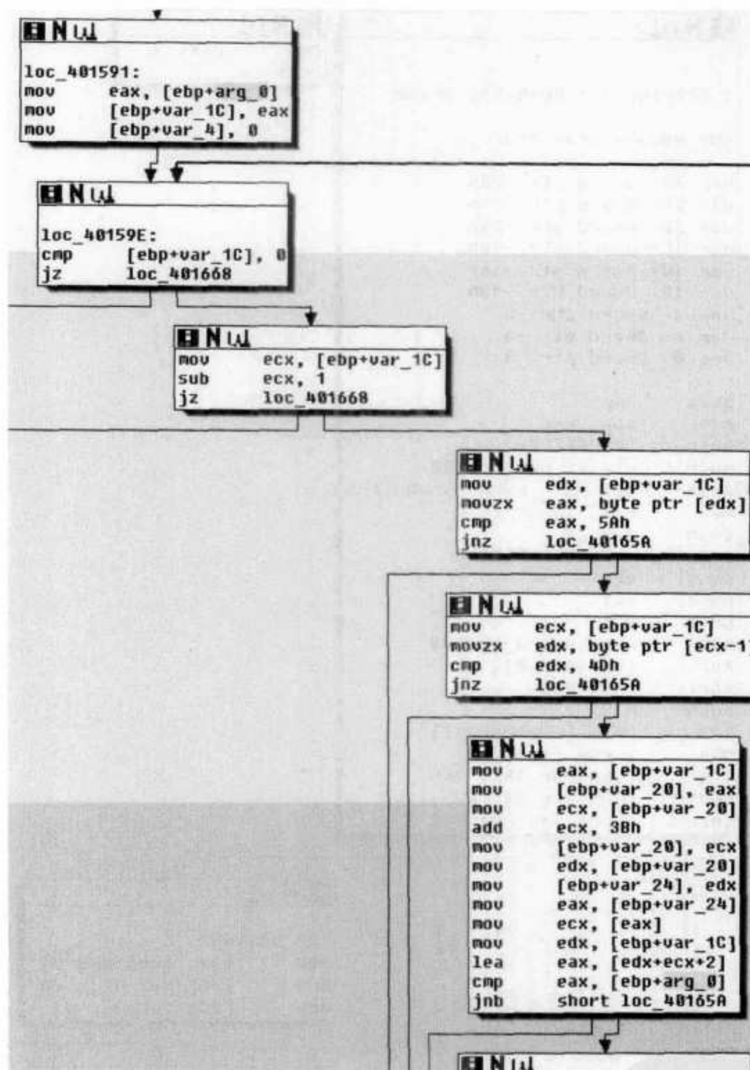


图 6-8　var_4 被初始化为 0

继续向下，我们到达 loc_40159E，它似乎也标志着一个循环。如果你在流程图里整体查看这个函数，这个循环就很明显了；当你看到从右上回指 loc_40159E 的箭头时，应该记住这是环回点（loop back point）。至于功能，我们在这个框里看到的是，再次测试这个例程参数的指针复制是否为 NULL，如果是则转到左分支，否则转向右分支。我们看从这里开始向下的第三个框，var_1C 指针被递减 1，并测试结果是否为 0，如果条件为真再次跳到左边分支。从这里开始的第四个框，我们看到这个指针被解除引用，并被检查其值是否为字节 0x5A，如果是则跳到第五个框，如果不是则跳到左边分支。

更多资源请点击访问稀.酷客(www.ckook.com)

如果匹配（0x5A），将检查它之前的字节是否为 0x4D，这意味着我们正在寻找的是 16 位序列 0x4D5A。你可能想起来了，它正好对应 ASCII 字符 MZ，是 DOS 头部开始的标记。不用再看，我们基本上可以猜测这个函数企图回溯内存以找出文件的开头，这也解释了为什么使用 SHE，因为回溯内存操作在理论上可能碰到坏内存而崩溃。我们继续看例程，基于非常少的事实做假定并不是一个好习惯。

接下来，在最后可见的框里，如果发现匹配 0x4D5A，将把这个指针复制到 var_20，然后递增 0x3B，并把这个新指针复制到 var_24，最后，我们看到接下来的是 var_1C，或者是指向 MZ（var_24（即 var_1C + 0x3B）的值加上 2）的指针。这（纯属偶然地）对应 DOS 头部里 PE 头部的偏移量，而且看上去最后一个框企图找出 PE 头部的地址，然后把指针与 arg_0 作比较并根据结果跳转，可能会跳到另一个异常的退出（参见图 6-9）。

进入下一区段，我们看到同样的主题在继续；在第一个框里，我们看到对 0x50 的比较，接着是检查 0x45，这意味着它正在寻找 0x4550。如果匹配的话，接下来的两个框检查随后的序列是不是 2 个 0 字节，意味着测试完全匹配并请求序列 0x45500000，即 PE\0\0，这显然是 PE 头部的幻数。本区段最后会测试是否为值 0x14C（对应 IMAGE_FILE_MACHINE_I386）。这么一来，这和前面区段中的操作都讲得通了，显示出这个例程的参数是一个指向可执行映像里偏移量的指针。根据这些输入，我们发现这个例程企图回溯内存找出 DOS 头部地址，一旦找到，就用这个信息定位 PE 头部，并对数据进行其他不太重要的校验。

从第二个直到最后的代码段，如果发现完全匹配则跳到左支分支，如果不匹配则跳到 loc_40165A，它接受这个指针，把它递减 1 并重复这个循环。现在我们已经了解了这个例程的主体，让我们再看一下它的分支吧，顺带看看返回值。

在图 6-10 中，从顶部右边开始（loc_40165A）

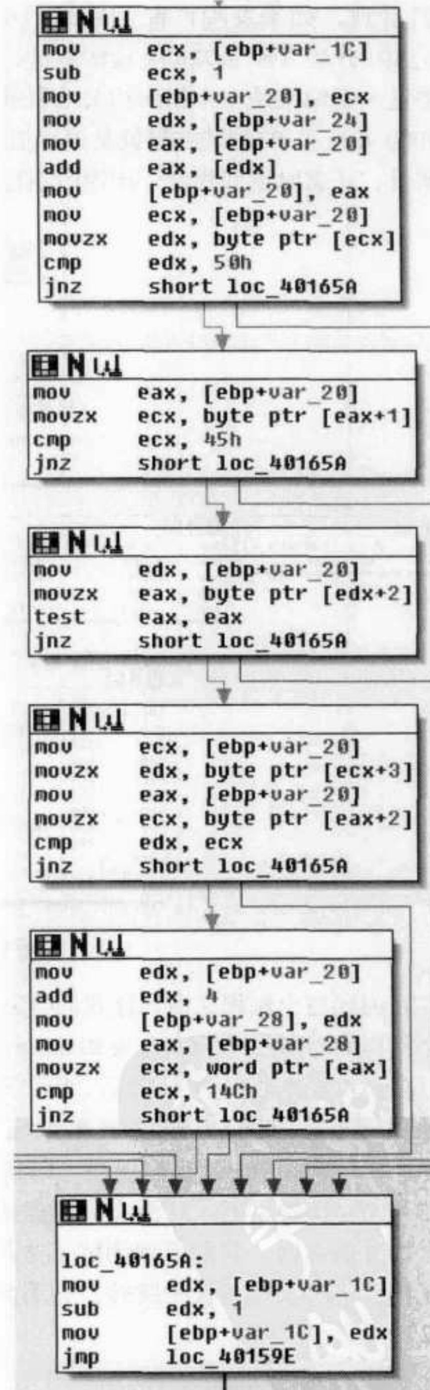


图 6-9 比较是否为 0x4D5A

更多资源请点击访问稀.酷客(www.ckook.com)

我们看到，如果发现匹配则跳到 loc_401668，loc_401668 修改 var_4 并把控制权传给 loc_401694，loc_401694 检索基址，递减 1 后把控制权交给例程的具有清除功能的部分。这意味着这个返回值是一个指向相关映像基址的指针。在最左边我们也看到，如果 arg_0 是 NULL，最初的 arg_0 的测试结果就是返回值。因此我们可以推断出，这个例程根据接受的指针找出它的基址，正常时返回指针，出错时返回 NULL。

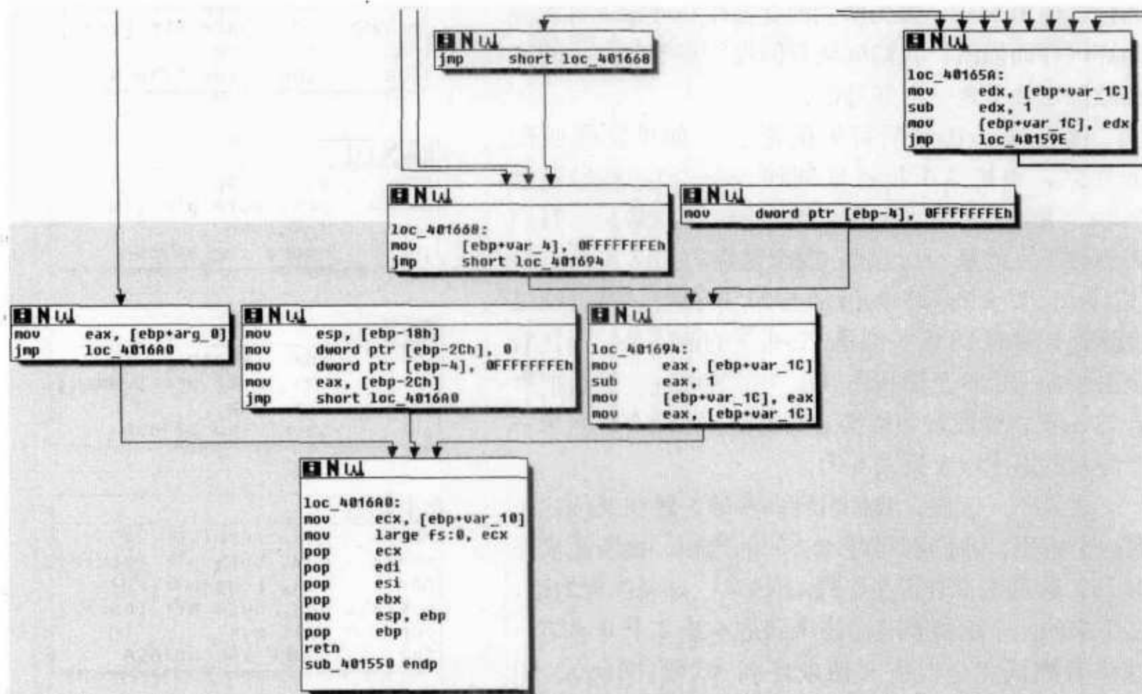


图 6-10　跳到 loc_401668

在回顾这个例程之后，让我们再来看一下 main()，看还能得出什么明显的结论。我们根据这个例程的功能，把它改名为 FindBaseAddr( )（图 6-11）。

在 IDA 中更新这个信息后，我们可以清楚地看到这两个调用分别寻找当前模块和 kernel32 模块的基址。我们更新变量名来反映这点，分别称这两个指针为 ModuleBasePtr 和 Kernel32BasePtr。我们现在可以猜测程序接下来会做什么，代码通过这个方法手动寻找自己的基址不是很常见，更没有必要查找 Kernel32。我们或许可以肯定这是在混淆库调用。随着这个例程的返回，我们看到测试了这两个指针是否为 NULL，如果结果为真跳到左分支。我们现在仍不考虑这些错误时跳转，先看两个对 FindBaseAddr( )的调用都成功的情景（参见图 6-12）。

更多资源请点击访问稀.酷客(www.ckook.com)

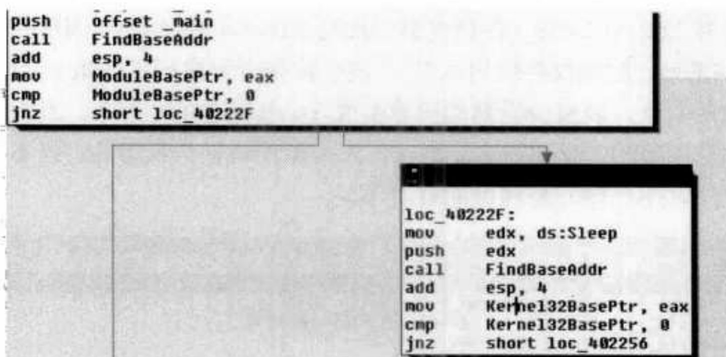


图 6-11 FindBaseAddr( )函数

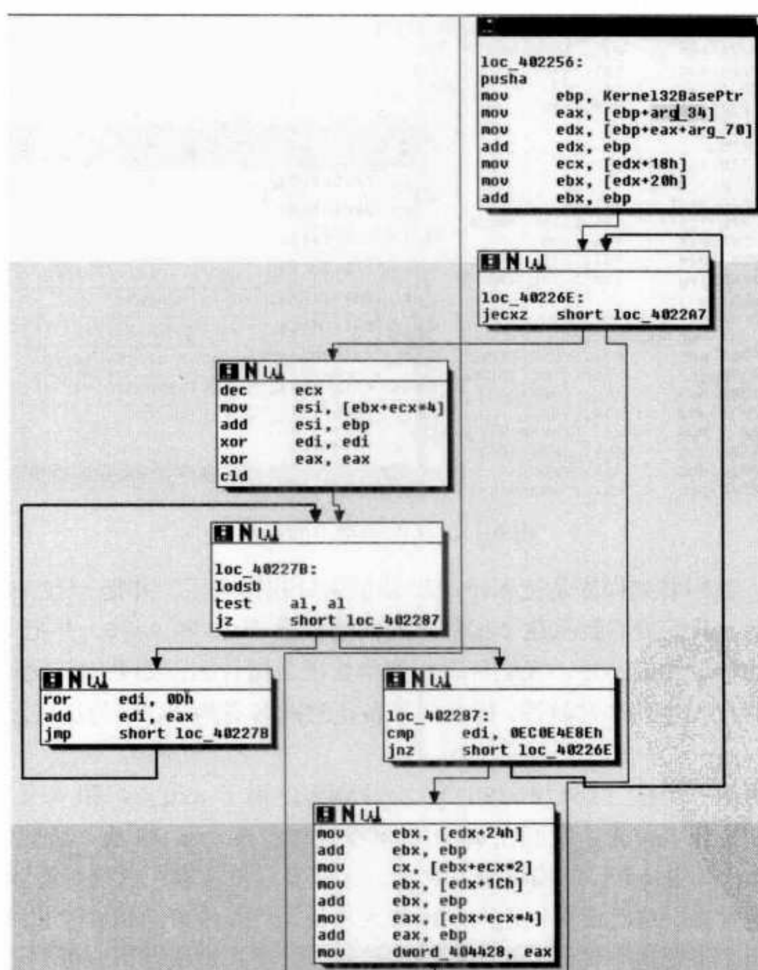


图 6-12 loc_402256

更多资源请点击访问稀.酷客(www.ckook.com)

继续看代码，在 loc_402256（在检查第二次对 FindBaseAddr()调用的返回值之后，如果找到 kernel32.dll 的基址，控制权将转到这里），我们首先看到执行了一条 pusha 指令（图 6-13）。有人可能不同意我的看法，我很少看到编译器生成 pusha 这样的指令，因此当看到这样的代码时，我常常推测它是手写的。我知道这是事实，因为源代码就是我写的；但不限于此，我认为通常情况下都是这样的，不过你的里程可能有所不同。

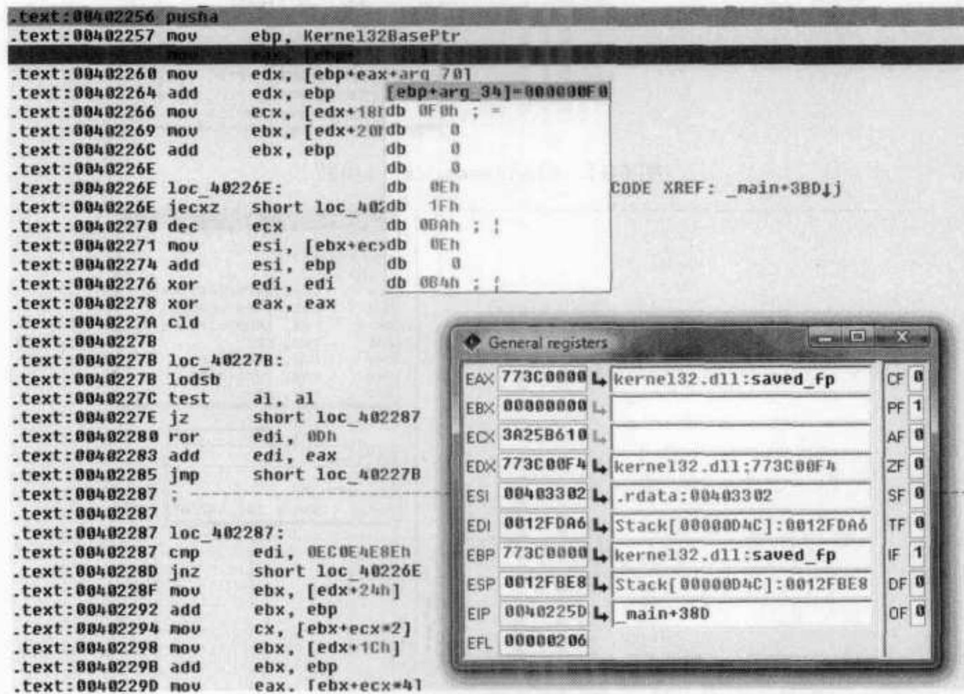


图 6-13 反汇编器下的视图

不管怎么样，我们看到程序是把 Kernel32 基址指针找回来了，并把 arg_34 和 arg_70 与这个基址相加；然后，我们就看到从这个偏移量找回偏移量 0x18 和 0x20，并把 0x20 与基址指针相加。基于看到的参数和偏移量，我们可以猜测函数准备做什么。如果你开发过 Windows 程序，应该对这一系列的代码序列非常熟悉，但我们准备花时间多看两眼，因为这里发生了一些有趣的事情。

如果你看一看第一个框（loc_402256），会注意到使用了 arg_34 和 arg_70。有两点使这个情形看起来有些奇怪：首先，我们在 main()例程里并没有 arg_34 或 arg_70。如果有的话，也是表示成来自 argv 或 envp 的偏移量。其次，这些值只被读取，仅仅静看我们甚至不能肯定这些偏移量到底是什么。这对使用反逆向技术的人来说当然是个好事情；除非跳过这部分，停止静态分析，否则为了了解这个代码段里到底做了什么，需要在调试器里分析它。依我之见，与反汇编器里充当被动/静态的角色相比，这么做可以在反反逆向的活动中占据主动。（参见图 6-13）

更多资源请点击访问稀.酷客(www.ckook.com)

当我们用调试器打开它，会发现所见到的对我们也没有太大帮助。不过，如果我们在 arg_34 上双击并查看内存，就会看得更清楚一些（图 6-14）。

```
kernel32.dll:773C0000 assume cs:ke
kernel32.dll:773C0000 ;org 773C000
kernel32.dll:773C0000 saved_fp db
kernel32.dll:773C0001 db  5Ah ; Z
kernel32.dll:773C0002 db  90h ; É
kernel32.dll:773C0003 db    0
kernel32.dll:773C0004 retaddr db
kernel32.dll:773C0005 db    0
kernel32.dll:773C0006 db    0
kernel32.dll:773C0007 db    0
kernel32.dll:773C0008 db    4
kernel32.dll:773C0009 db    0
kernel32.dll:773C000A db    0
kernel32.dll:773C000B db    0
kernel32.dll:773C000C db 0FFh
kernel32.dll:773C000D db 0FFh
kernel32.dll:773C000E db    0
kernel32.dll:773C000F db    0
kernel32.dll:773C0010 db 0B8h ; +
kernel32.dll:773C0011 db    0
kernel32.dll:773C0012 db    0
kernel32.dll:773C0013 db    0
kernel32.dll:773C0014 db    0
kernel32.dll:773C0015 db    0
kernel32.dll:773C0016 db    0
kernel32.dll:773C0017 db    0
kernel32.dll:773C0018 db  40h ; @
kernel32.dll:773C0019 db    0
kernel32.dll:773C001A db    0
kernel32.dll:773C001B db    0
kernel32.dll:773C001C db    0
kernel32.dll:773C001D db    0
kernel32.dll:773C001E db    0
kernel32.dll:773C001F db    0
kernel32.dll:773C0020 db    0
kernel32.dll:773C0021 db    0
kernel32.dll:773C0022 db    0
kernel32.dll:773C0023 db    0
kernel32.dll:773C0024 db    0
kernel32.dll:773C0025 db    0
kernel32.dll:773C0026 db    0
kernel32.dll:773C0027 db    0
kernel32.dll:773C0028 db    0
kernel32.dll:773C0029 db    0
kernel32.dll:773C002A db    0
kernel32.dll:773C002B db    0
kernel32.dll:773C002C db    0
kernel32.dll:773C002D db    0
kernel32.dll:773C002E db    0
kernel32.dll:773C002F db    0
kernel32.dll:773C0030 db    0
kernel32.dll:773C0031 db    0
kernel32.dll:773C0032 db    0
kernel32.dll:773C0033 db    0
kernel32.dll:773C0034 db    0
kernel32.dll:773C0035 db    0
kernel32.dll:773C0036 db    0
kernel32.dll:773C0037 db    0
kernel32.dll:773C0038 db    0
kernel32.dll:773C0039 db    0
kernel32.dll:773C003A db    0
kernel32.dll:773C003B db    0
kernel32.dll:773C003C db 0F0h ; =
```

图 6-14 arg_34

更多资源请点击访问稀.酷客(www.ckook.com)

当我们看到被 IDA 称为 arg_34 的指针时，会发现它是一个来自 Kernel32.dll 基址的偏移量，特别是偏移量 0x3C，你可能想起来它是 FindBaseAddr()使用的对基址（PE 头部的偏移量在此定义）的偏移量。这很合理，而且比 arg_34 要合理得多。你可能会认为这个方法对混淆来说很不错，有一定的作用，但更多时候只令人厌烦。不过应当注意的是，其他的调试器（如 OllyDBG）可能没有这样的问题。例如，图 6-15 显示了来自 ImmunitySec 调试器（其实就是重新包装过的 OllyDBG，带有 Python 接口）的截屏。

```
00402251  .-E9 1A060000    JMP ersvr.00402870
00402256  > 60             PUSHAD
00402257  . 8B2D 24444000  MOV EBP,DWORD PTR DS:[404424]
0040225D  . 8B45 3C        MOV EAX,DWORD PTR SS:[EBP+3C]
00402260  . 8B5405 78      MOV EDX,DWORD PTR SS:[EBP+EAX+78]
00402264  . 03D5           ADD EDX,EBP
00402266  . 8B4A 18        MOV ECX,DWORD PTR DS:[EDX+18]
00402269  . 8B5A 20        MOV EBX,DWORD PTR DS:[EDX+20]
0040226C  . 03DD           ADD EBX,EBP
0040226E  >-E3 37          ┌JECXZ SHORT ersvr.004022A7
00402270  . 49             │DEC ECX
```

图 6-15　ImmunitySec 调试器

像你看到的，这段代码在 ImmunitySec 调试器里显示正常，可以很容易看清楚其意图。为什么在 IDA 里不能正常显示呢，这是因为这段代码篡改了 EBP 寄存器（一般而言，表明是内联汇编），造成 IDA 无法识别它是偏移量，错误地把它当作函数的参数。另一个可能有显著影响的理由是——尽管篡改 EBP 已经给出充分理由，这实际上是内联的函数。不管怎样，在调试器里我们知道这两条 mov 指令实际上分别从 Kernel32 基址获取偏移量 0x3C 和 0x78。（参见图 6-16）

提示

许多人发现 IDA 的调试器并不好用，因此更喜欢使用其他的调试器。例如，微软发布的 RING0 调试器 WinDBG，缓慢死去的另一个 RING0 调试器 SoftIce（很长时间以来是破解高手的最爱）。SoftIce 因为缺乏支持，随着 Windows 的升级出现了可操作性问题，导致其被越来越多的人抛弃。

另一个调试器是 OllyDBG，它一直是逆向社团的主力武器，主要因为它易于使用、界面直观，并且是免费的。最近这半年，Immunity Sec 购买了访问 OllyDBG 源代码的权利，并使之与 Python 的插件及其他功能相结合并发布。如果你正在开发破解程序，Immunity 调试器将非常有帮助，不仅是因为其可编写脚本的能力，而且因为它附带了有用的脚本和特性，例如堆元数据的识别等。如果你的工作是在 Windows 平台上开发破解程序，而且没用过 Immunity 的调试器，那将是你的一大损失。

无论如何，在 IDA 里修复这个错误表示还是非常容易的；在变量名上右击，会提示不同的表示方式，此时选择第一选项。因此，实际上我们并不需要使用调试器。就这里剩下的代码而言，作者非常熟悉这段代码序列（已经到了下意识的程度），它们是名为 LSD（Last Stage of Delirium）的波兰黑客组织最早公开的，Skape 和 nologin crew 在“Understanding Win32 Shellcode”

更多资源请点击访问稀.酷客(www.ckook.com)

中又对它进行了扩充重申。它的作用是在偏移量 0x3C 处找到 PE 头部，然后把这个偏移量加上基址，再加上 0x78，将指向输出数据目录；接下来就是简单地迭代输出，用 ror 指令把取到的名字做散列处理。如果结果与特定的输出相匹配，将会把这个结果与另一个 4 字节散列做比较。

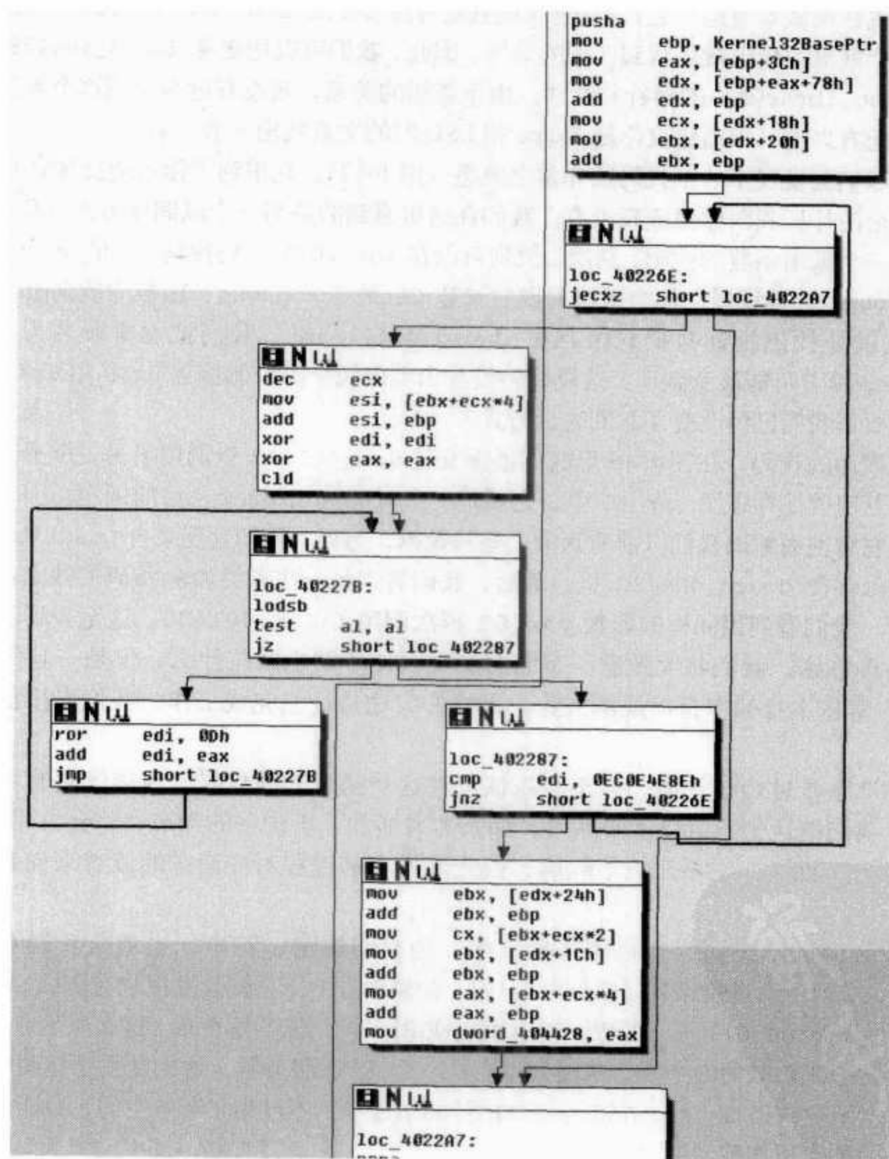


图 6-16 来自 kernel32 基址的偏移量 0x3C 和 0x78

更多资源请点击访问稀酷客(www.ckook.com)

换句话说，这是一个与位置无关的、寻找 DLL 输出的方法，且过程中不依赖其他的 API。通常在 shellcode（壳代码）的“bootstrapping”过程中使用它。一般用它查找像 LoadLibrary( ) 和 GetProcAddress( ) 这样的函数的地址。因此，如果查看 loc_402287 的话，你会看到把 EDI 与常量值 0x0EC0E4E8E 做比较的 cmp 指令。这个常量就是这段代码寻找的散列，如果你在 goolge 中搜索它（或在调试器里运行它），就会发现它是对应 LoadLibraryA( ) 的散列。从底部开始算的倒数第二个框里的代码就是找到匹配的地方，因此，我们可以把变量 dword_404428 重命名成一个指向 LoadLibraryA() 的指针并继续。由于篇幅的关系，我没有过多介绍这个算法的细节，但如果你对此有兴趣，强烈建议你把 Skape 和 LSD 写的文章找出来看一看。

继续，我们会发现下一个代码段非常之熟悉（图 6-17），几乎到了你可能会怀疑我错误地重复粘贴了这个图片！我向你保证我没有。我们在这里看到的是另一个以同样方式遍历 Kernel32，企图寻找另一个输出函数的视图。这次，散列可以在 loc_4022D9 处找到，其值为 0x7C0DFCAA。再次召唤 Google 或调试器，你会很快发现它就是 GetProcAddress() 函数的散列值。因此，这段代码其实就是找出指针并把它保存在 dword_40441C 里，我们把它重命名为 GetProcAddressPtr。读者可能注意到了，这段代码没有出现像较早提取偏移量 0x3C 和 0x78 时显示的怪癖，这主要是我们已经改变了它的表示方式。

紧接着前面的代码，在图 6-18 里我们很快发现 loc_4022F9 里调用了另一个子例程，这个新子例程的返回值保存在了 var_6C 中。之后是一个对 LoadLibraryA() 的调用。因此，我们基本可以假定它解密或解码我们以前看到的一些栈数据。另外，我们发现来自 LoadLibraryA() 的返回值会被保存在 dword_40442C 里。最后，我们看到另一些有关加密/编码的线索：在使用这个字符串后，我们看到用同样的参数 var_6C 再次调用了 sub_4014D0，这至少暗示我们这或许是某种对称加密。我们将大致看一下它的实现以便了解它做些什么，但是一旦你确定它只是某种加密/解密本身的字符串混淆方式，最快的方法是让它完成工作，我们直接把结果复制出来。

但在你准备步过函数之前，你至少应该读过这个函数的源代码以便确保它的功用与你想象中相同；我们把这个留给读者当练习，如果本书增加分析加密的细节，将会占用大量篇幅，反而得不偿失。再说，它平凡但不陈腐，因此弄懂它的过程对受激励的读者来说是非常好的练习机会。

重新载入调试器，让这个程序辛勤工作，为我们解密字符串，也就是我们看到的调用 sub_4014D0 后的十六进制序列（参见图 6-19）。如果你查一下，就会发现它对应以 NULL 结尾的 ASCII 字符串 rpcrt4.dll——在 Windows 里生成 RPC 调用需要这个库。这么做是有道理的，当然，这主要基于我们以前的分析，从而知道它是一个 RPC 服务器。这也是整个谜面中的一大部分，我们基本上能猜出 GetProcAddress() 指针的用途了。不过为了节省空间，也让读者有些事做，我们还是把它留作练习吧。

更多资源请点击访问稀.酷客(www.ckook.com)

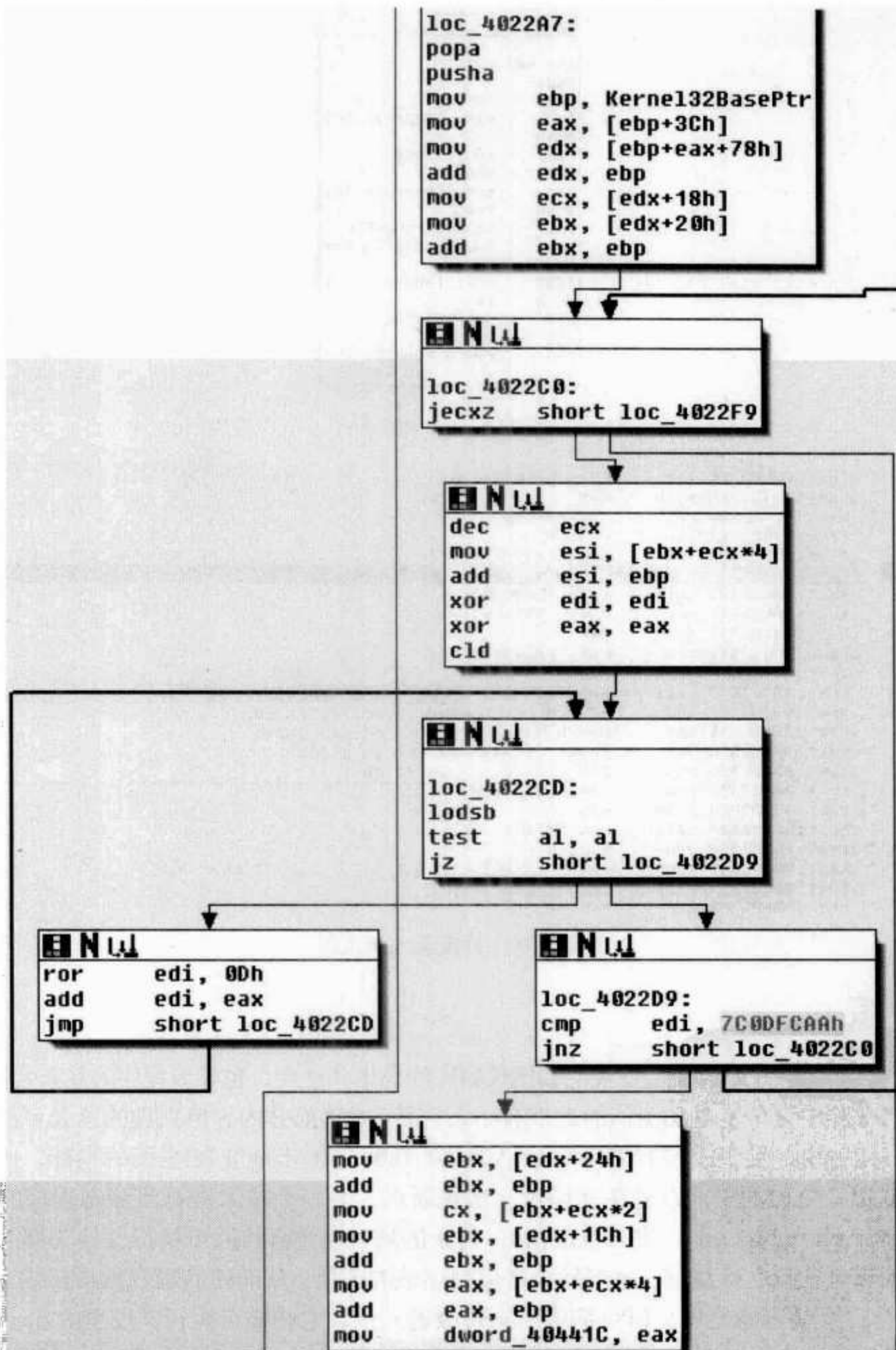


图 6-17 kernel32

6

更多资源请点击访问稀.酷客(www.ckook.com)

```
loc_4022F9:
popa
push    0Ah
lea     eax, [ebp+var_6C]
push    eax
call    sub_4014D0
add     esp, 8
lea     ecx, [ebp+var_6C]
push    ecx
call    LoadLibraryAPtr
mov     dword_40442C, eax
push    0Ah
lea     edx, [ebp+var_6C]
push    edx
call    sub_4014D0
add     esp, 8
cmp     dword_40442C, 0
jnz     short loc_402345
```

图 6-18　loc_4022F9

```
.text:004022FC lea    eax, [ebp+var_6C]
.text:004022FF push   eax
.text:00402300 call   sub_4014D0
.text:00402305 add    esp, 8
.text:00402308 lea    ecx, [ebp+var_6C]
EIP
.text:0040230C call   LoadLibraryAPtr
.text:00402312 mov    dword_40442C, eax
.text:00402317 push   0Ah
.text:00402319 lea    edx, [ebp+var_6C]
.text:0040231C push   edx
.text:0040231D call   sub_4014D0     [ebp+var_6C]=72637072
.text:00402322 add    esp, 8         var_6C dd 72637072h
.text:00402325 cmp    dword_40442C,  var_68 dd 642E3474h
.text:0040232C jnz    short loc_4023 var_64 dw 6C6Ch
.text:0040232E push   17h            var_62 db 0
.text:00402330 lea    eax, [ebp+var_24]
.text:00402333 push   eax
.text:00402334 call   sub_4014D0
.text:00402339 add    esp, 8
.text:0040233C push   eax                                ; char *
.text:0040233D call   sub_401EA0
.text:00402342 add    esp, 4
```

图 6-19　再度载入调试器

6.5　小结

回顾本章描述的技术，我们发现它们使代码库的变化非常大，也导致程序的复杂性快速增长。我们已看到移除字符串常量和类似的数据有多么容易。对这部分内容感兴趣的读者可能想阅读更多的内容，比如说，关于改写 IAT (Import Address Table，输出地址表)里面的指针，或者把系统函数复制到用户分配的空间以避免在函数上出现断点。另一个技术来自写病毒程序的圈子，即 EPO（Entry Point Obscuring，进入点混淆）。传统的病毒通常会修改可执行文件头部的进入点，把自己追加到可执行文件尾部。这当然会产生被感染的迹象，从而使病毒暴露给反病毒软件。结果，导致了 EPO 病毒的出现。EPO 病毒不再修改进入点，它扫描可执行区段里的 jmp 或类似指令，然后修改它，从而把控制权交给位于其他地方的病毒主体。同样的技术可以用于这样一个事实：IA-32 机器有有限的调试支持，这是通过进入系统库函数 5 至 10 字节，而非 OEP 获得的。

更多资源请点击访问稀.酷客(www.ckook.com)

第 7 章

实战2

本章内容：

- 跟踪 Read 事件的执行流
- 确定协议的结构
- 确定协议中是否有未文档化的消息
- 用 IDA 确定处理特殊消息的函数

更多资源请点击访问稀.酷客(www.ckook.com)

7.1 协议问题

现在，有许多厂商都选择不公开所开发软件使用的协议，或者只公开其中的一部分。作为一名逆向分析工程师，你需要找出与之兼容的协议，或分析一个程序中有无可能导致安全问题的隐藏功能。本章，我们将通过对示例程序所使用的协议进行分析，最终还原它的消息结构。

7.2 协议结构

大多数协议是由离散的、可以单独解释的消息组成的消息流。但凡事都有例外。比如，HTTP是由客户端发送的未结构化的请求，以及服务器端回应的未结构化的响应组成的。FTP则使用了基于文本的控制通道，并为每一个被传输的文件单独建立 TCP 会话。但是这些只占到所需要分析的协议的很小一部分。

逆向工程师如果不能访问相应的可执行文件，则可以通过网络上的原始字节逆向分析协议。逆向工程师如果可以接触到可执行文件，但不能运行它，那么最终也可以从可执行文件中提取出协议。但在大多数情况下，我们还是能拿到可执行文件并运行它的。这样就能结合上面所说的两种方法，加快协议分析的速度。逆向工程师可以从网络上的字节中获知协议结构的大致情形，然后再通过分析二进制文件获得客户端或服务器端尚未表现出的特性。如果有足够的时间，我们总可以从二进制文件中获得最准确的协议结构。

7.2.1 分帧与重组

任何一个协议都要知道消息的启始及结束位置。这就是通常所说的分帧（framing）。许多协议都可以与连接的另一端保持同步。如果发送端认为消息有 30 个字节，而接收端认为这个消息只有 20 个字节，那么剩下的 10 字节很可能会被接收端当作新消息，从而破坏消息，最终导致连接被中止。

消息通过互联网传送时，TCP/IP 协议并不能保证把它作为一个整体传递。消息可能会被分成好几个片段(具体的实现由系统完成，程序员并不需要了解细节)。对程序员而言，重要的是一个称之为 `read()`的函数，它返回整个消息或消息的某些片段，但也可能会返回多个消息。应用程序在继续执行前必须保证把整个消息都读进了缓冲区。我想要说大多数程序都能做得很好。但不幸的是，你碰到的许多程序在重组及解析消息方面做得都不好。

大多数小型的或草草写成的程序一般都假设通过互联网传输的消息是一个整体，它们的基本功能模块如图 7-1 所示。

如果消息在传输过程中被分成几个片段，根据函数的实现方法，该程序要么会在解析消息时崩溃，要么就把它作为不完整的消息丢弃掉。不过，许多这类程序的健壮性却令人吃惊。当一个消息被分段后，程序将从套接字分两次读取分段后的消息，然后把它们当作无效的消息丢弃掉，但系统仍会接收下一个消息（如果这个消息没有被分段的话）并正确处理。

更多资源请点击访问稀.酷客(www.ckook.com)

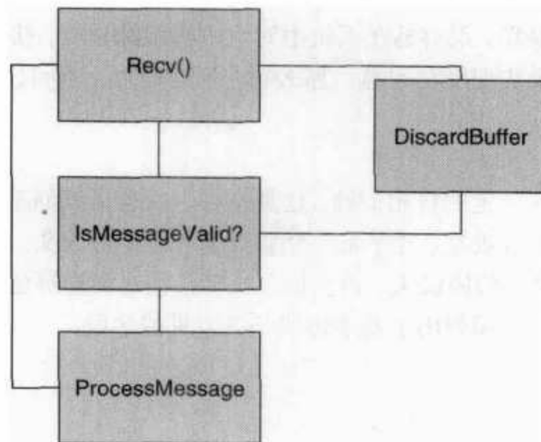


图 7-1 小型程序的网络消息接收和处理基本模块示意图

有个流行的即时通信工具的客户端就是用这种循环实现的。要分辨出这类程序很简单，只需在客户端与服务器端之间引入一个代理并把消息分解成一些小片段。如果这些（分段后的）消息被忽略了，那么你就可以认定这个程序只能处理完整的消息。

较大的商用程序通常会把读取的数据放入读缓冲区，当接收了完整的消息后，它就会把消息从读缓冲区里移走，为接收后续的消息做准备。它们的基本功能模块如图 7-2 所示。

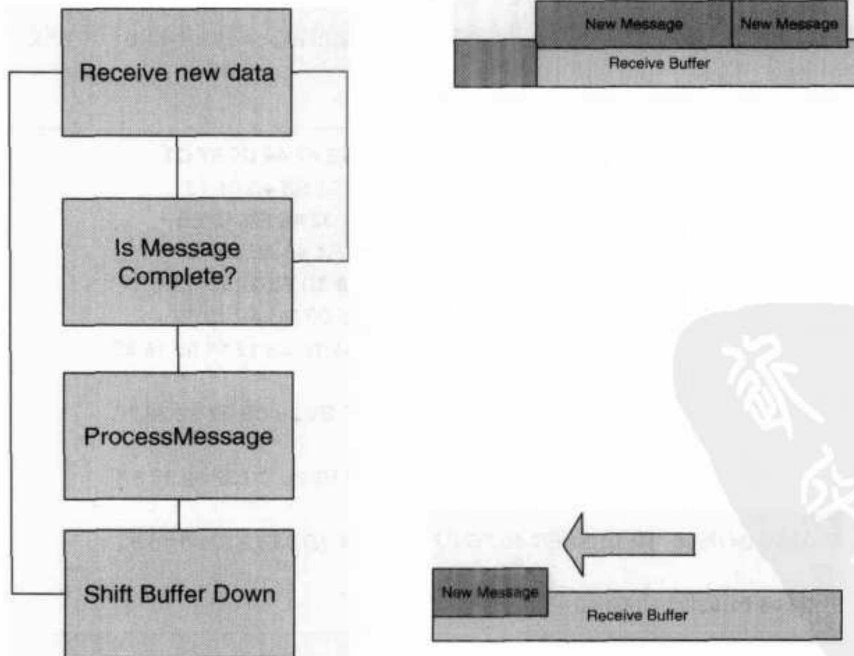


图 7-2 较大的商用软件的网络消息接收和处理基本模块示意图

更多资源请点击访问稀.酷客(www.ckook.com)

这是一个完全正确的循环，很容易在系统中对它进行跟踪分析。接收缓冲区一般作为全局变量分配或在堆上分配，如果实现没有问题，那么它在所有情形下都可以正常工作。

7.2.2 自相似性

协议发送的数据包都有一定的自相似性，这是指在一次会话期间某些操作是重复执行的，因此通过网络传输的字节也会有重复。几乎每一个协议都有一个协议头。网络上的每一个消息都会包含这个协议头。即使是最小的协议头，也会包含长度及消息类型等信息。解析器需要知道当前的消息长度及其解析方法。下面列出了基本协议头里常见的字段：

- 幻数[①]；
- 序列号；
- 时间戳；
- 数据或区段的长度；
- 会话 ID；
- 子消息的数量；
- 错误代码；
- 随机内容。

在分析协议之前，最好抓一些不同类型的数据包看看，以便有一个大致的印象。我总是把抓到的数据包以十六进制的格式打印出来，以便在分析过程中随时做一些注释。下面是本节例子的一些数据包。我用粗体标记了从客户端到服务器端的包，用斜体标记了从服务器端到客户端的包。

DE AD BE EF 00 18 01 00 76 B7 0B 5A 42 DD 54 B9 6B E8 1E 47 44 D9 67 C3
DE AD BE EF 00 18 02 00 C1 06 45 18 90 51 5D 71 44 46 D7 21 B6 4C 01 73
DE AD BE EF 00 18 01 00 45 EB 2E 08 42 68 22 72 60 E9 1B 32 16 FA 45 EB
DE AD BE EF 00 18 02 00 2E 08 42 68 22 72 60 E9 1B 32 16 FA 40 AB 55 AD
DE AD BE EF 00 18 01 00 64 D4 7F 88 7E E1 5A AA 21 46 49 3D E3 22 7E 1E
DE AD BE EF 00 18 02 00 79 18 D2 6C D7 3D C9 61 60 7B 02 00 DC 4F 40 59
DE AD BE EF 02 08 03 01 AF 59 3E 31 ED 45 FD 02 E3 26 A1 1B 08 12 19 05 16 82
59 26 3B 90 77
DE AD BE EF 02 08 03 00 77 0D 33 A4 03 19 4D F1 62 F5 1F B2 20 CB 37 82 25 87
46 0E 6E 8A 56
DE AD BE EF 02 08 04 00 77 0D 33 A4 03 19 4D F1 62 F5 1F B2 20 CB 37 82 25 87
46 0E 6E 8A 56
DE AD BE EF 02 08 04 00 AF 59 3E 31 ED 45 FD 02 E3 26 A1 1B 08 12 19 05 16 82
59 26 3B 90 77
DE AD BE EF 00 08 01 02 00 00 00 00

① 用来识别该消息使用的是不是当前应用程序所能处理的协议。

更多资源请点击访问稀.酷客(www.ckook.com)

乍一看，这么多乱七八糟的数据，其中有多少种不同类型的消息呢？基本头部有多长呢？逐列看一下，我们可以做一些假设。每一个包的前 4 个字节是一样的，都是 DEADBEEF，我们可以假设它是协议头里的幻数。这里最短的包有 12 个字节。我们可以据此推定基本头部比观察到的最短消息要小一些。所以我们可以大致猜测协议的结构，如下所示：

```
struct base_header{
        int MagicNumber;          /*Always 0xDEADBEEF*/
        char unknown1;            /*00,02*/
        char unknown2;            /*08, 18*/
        char unknown3;            /*01, 02, 03, 04*/
        char unknown4;            /*00,01,02*/
        char unknown5;            /*Lots of possibilities*/
        char unknown6;            /*Lots of possibilities*/
        char unknown7;            /*Lots of possibilities*/
        char unknown8;            /*Lots of possibilities*/
}
```

数据包最终会从网络中读取并由程序进行处理。有几个 API 调用用于完成这种任务。当然也有些程序会直接使用系统调用读取数据，而不使用 API 调用，但一般只有恶意软件才这么做。程序中从网上读取数据的地方通常就是我们开始分析的好地方。下面列举的是一些可以从网络套接字里读取数据的常见的 API 调用。

- read/write
- recv/send
- recvfrom,/sendto
- WSARecv/WSASend
- WSARecvFrom/WSASendTo
- ioctl
- ioctlsocket
- WSARecvDisconnect/WSASendDisconnect
- WSARecvEx/WSASendEx
- recvmsg/sendmsg
- WSARecvMsg/WSASendMsg

可执行文件的输入表（如图 7-3 所示）中只有对 WSARecv 的一个引用，所以我们可以把它作为分析这个协议的理想起点。WSARecv 调用把数据读入位于栈上的缓冲区。它一次最多可以读入 0x4000 个字节，如图 7-4 所示。

在数据被完全读入栈缓冲区后，程序立即检查读入的字节数是否小于 8，如果小于 8 个字节，程序将直接跳到退出函数，如图 7-5 所示。

更多资源请点击访问稀酷客(www.ckook.com)

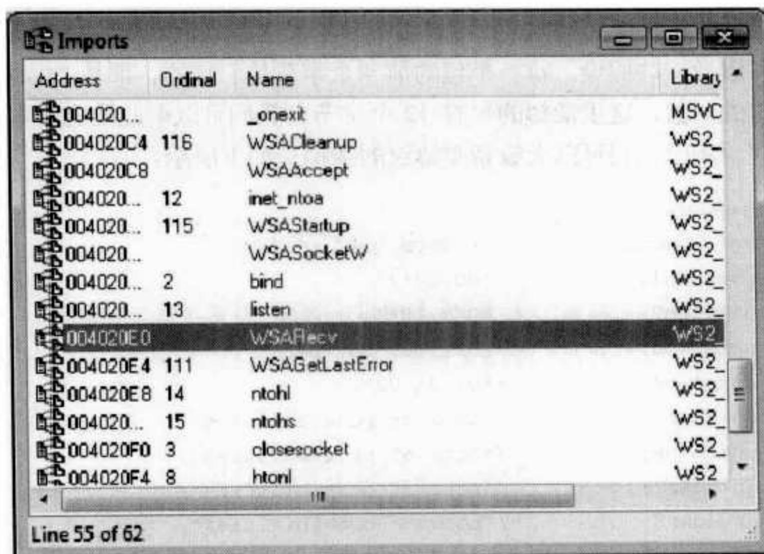


图 7-3　可执行文件的输入表，该表显示此处引用了 WSARecv 这个套接字函数

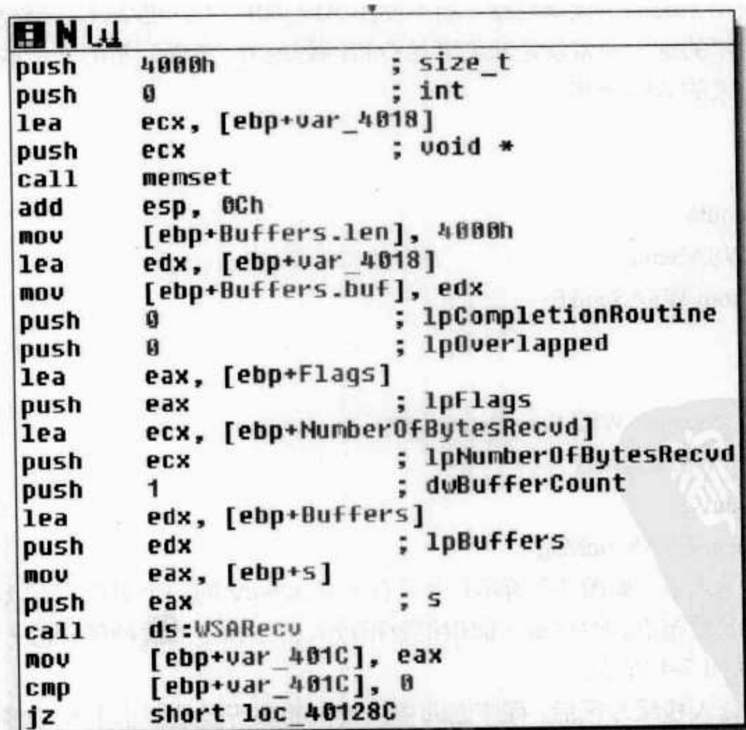


图 7-4　WSARecv 调用

更多资源请点击访问稀.酷客(www.ckook.com)

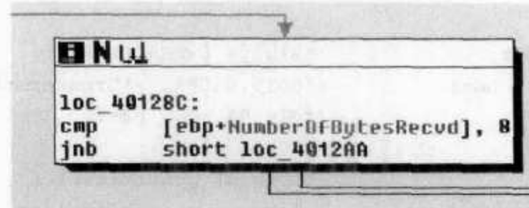


图 7-5 退出函数

（如果读入的字节不少于 8 个字节），程序就会检查它们的前 4 个字节是否为 DEADBEEF。它们与 dump 包（前 4 个字节）中的幻数完全匹配。这些从网络上读取的字节在做比较之前会经由 ntohl()处理。这意味着这个协议是大端字节序（big-endian）。这同样有助于我们确定字段的大小。这个程序在处理 2 字节字段和 4 字节字段之前需要做字节交换，如图 7-6 所示。

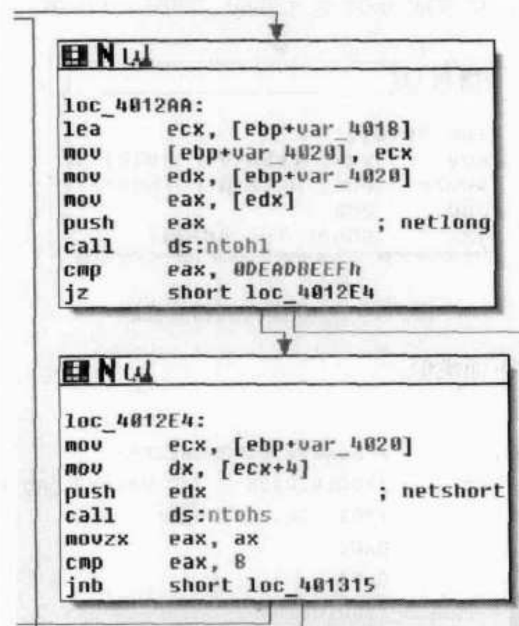


图 7-6 （大端格式到小端格式的）字节转换

如果消息通过了 DEADBEEF 测试，程序将检查接下来的 2 个字节是否长度小于 8。如果小于 8，这个包将被丢弃。我们现在知道跟在幻数后的 2 个字节是一个单元，且总是大于 8[①]。有了从这两个从基本块里获取的信息，就可以修正之前的结构示意图了。这个数字对应包的长度，所以我们可以暂时把这 2 个字节的字段标成长度字段。

① JNB 是 jump not below 的意思，所以这里应该是大于等于 8。

更多资源请点击访问稀.酷客(www.ckook.com)

```
struct base_header{
        int MagicNumber;            /*Always 0xDEADBEEF*/
        unsigned short Len;         /*0018,0208*/ /*Greater than 8*/
        char unknown3;              /*01, 02, 03, 04*/
        char unknown4;              /*00,01,02*/
        char unknown5;              /*Lots of possibilities*/
        char unknown6;              /*Lots of possibilities*/
        char unknown7;              /*Lots of possibilities*/
        char unknown8;              /*Lots of possibilities*/
}
```

在长度检查后，处理函数将把偏移量为7处的一个字节放入寄存器，并对它做一个AND操作。当协议中一个字段被按位做AND或OR操作，且操作数是常量时，几乎可以肯定它就是某种形式的位字段[①]。在这个例子里，如果第3位设为1，进程将停止处理这个数据包。我们完全可以说字节7是一个位字段，且常量0x04是无效的，如图7-7所示。

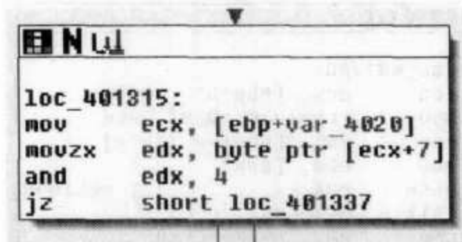


图7-7 AND操作的相关代码

修正后的结构看起来与下面类似：

```
struct base_header{
        int MagicNumber;        /*Always 0xDEADBEEF*/
        unsigned short Len;     /*0018,0208*/ /*Greater than 8*/
        char unknown3;          /*01, 02, 03, 04*/
#define FLAG_NONE               0x00
#define FLAG_INVALID            0x04
        char Flags;             /*00,01,02*/
        char unknown5;          /*Lots of possibilities*/
        char unknown6;          /*Lots of possibilities*/
        char unknown7;          /*Lots of possibilities*/
        char unknown8;          /*Lots of possibilities*/
}
```

从dump的数据包中，我们还看到有另外2个标志被置位。在大多数标志里，空的位字段意味着“什么也不做”，因此可以把它看作占位符，0被定义成FLAG_NONE。这至少使得0x01和

① 也就是说把字段里的比特位当作标志位，每一比特位都有一定的含义。

更多资源请点击访问稀.酷客(www.ckook.com)

0x02 在协议中是有效的标志。

接下来，二进制程序把数据包的字节 6 保存到栈变量里，并把这个栈变量与 0x01、0x03 以及 0x5C 这三个可能的值做比较。在这个 dump 包里，我们看到过 0x01 到 0x04，但只有 0x01 和 0x03 是为服务器指定的包。看起来 0x02 和 0x04 是从服务器到客户端的数据包使用的；0x01、0x03 和 0x5C 是从客户端到服务器的数据包使用的[①]。程序不处理其他的值，如图 7-8 所示。

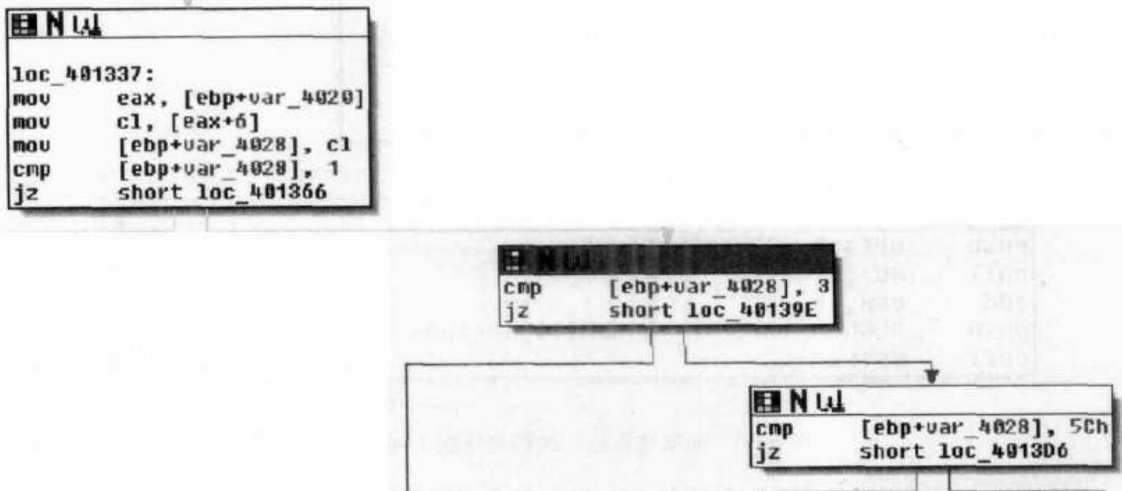


图 7-8 其他值不被处理

根据 unknown3 的值[②]，二进制程序将会调用 3 个函数中的一个。这 3 个函数的原型区别不大，传递给它们的参数都是：第一个参数是指向我们在网上发送的原始数据包的指针，第二个参数是从网上读取的字节数，第三个是读取数据时数据包的套接字句柄。如图 7-9 所示。

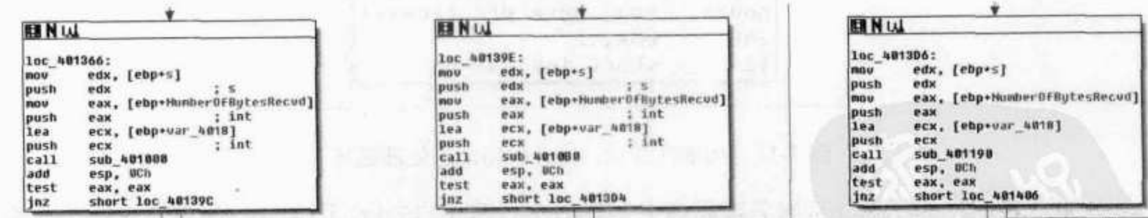


图 7-9 三个被调用函数的原型

① 原文在这里写反了，译文中已经修正了。原文和前面的十六进制 dump 正好相反，因为拿不到示例程序，我没有办法判断现在调试的到底是服务端程序还是客户端程序。不过要是前面的 dump 是正确的话，现在我们调试的应该是服务端程序，而且这一点在下文的那句“至此，我们完成了对示例服务器端程序中处理消息的循环的分析”中可以得到证明。

② 根据其含义，我们在后面把它改为 PacketType。

更多资源请点击访问稀.酷客(www.ckook.com)

后面我们将对这三个函数进行逆向分析。现在，先把它们放一放，把当前这个函数分析完再说。最后的两个比较操作都是与标志字段相比二进制程序用按位 AND 操作对这个字段的 0x01 位进行检查。如果这个位被置为 1，线程将休眠 1 000ms，如图 7-10 所示。

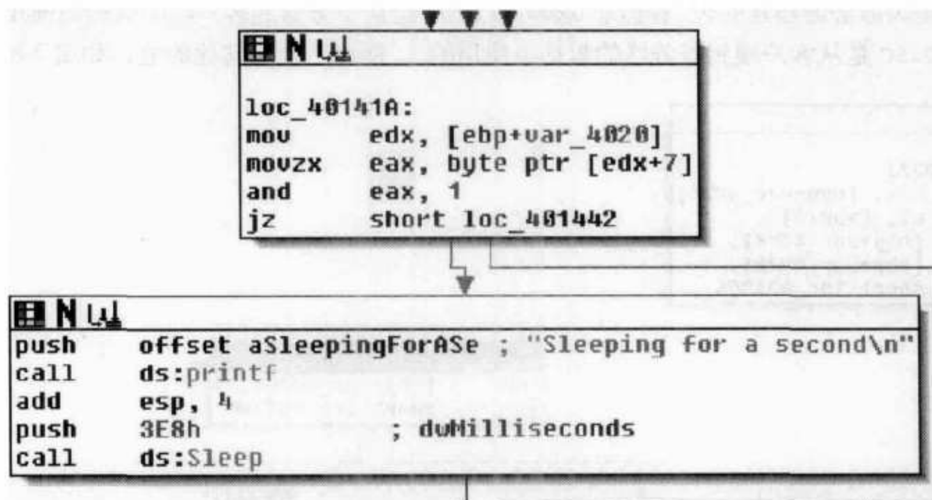


图 7-10　如果被置位，线程将休眠 1 000ms

在处理消息的循环结束之前，程序最后一个检查是对标志字段中的 0x02 位做按位 AND 操作。如果这个位被置为 1，将跳出消息处理循环。如图 7-11 所示：

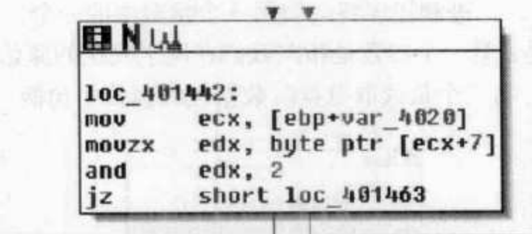


图 7-11　如果被置位，就会跳出消息处理循环

至此，我们完成了对示例服务端程序中处理消息的循环的分析。因为我们没有考虑这个结构中的其他字节，所以基本上可以肯定这些字节不是这个协议基本头部的一部分，而是协议的其他部分。最终的结构如下所示：

```
struct base_header{
        int MagicNumber;          /*Always 0xDEADBEEF*/
        unsigned short Len;       /*0018,0208*/ /*Greater than 8*/
#define PACKET_SERVER_01          0x01
#define PACKET_CLIENT_02          0x02
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
#define PACKET_SERVER_03            0x03
#define PACKET_CLIENT_04            0x04
#define PACKET_SERVER_5C            0x5C
        char PacketType;            /*01, 02, 03, 04*/
#define FLAG_NONE               0x00
#define FLAG_SLEEP                  0x01
#define FLAG_PROCESS_AND_EXIT       0x02
#define FLAG_INVALID                0x04
        char Flags;                 /*00,01,02*/
}
```

对这个简单的协议来说，我们接下来要做的就是确定三个消息类型的作用。服务器端处理 0x01、0x03 及 0x5C 类型的消息。因为我们准备逆向分析整个协议，所以 0x01 消息是个不错的起点。

0x00401000 处的函数处理 0x01 类型的消息。传递给它的三个参数依次为：指向保存整个消息的栈缓冲区的指针；读取的字节数；读取消息处的套接字。

我们首先需要注意的是，这个函数并没有涉及传递给它的缓冲区或长度字段，而是直接跳转到本地栈缓冲区里构造另一个消息。它首先用 memset 把缓冲区置 0，然后将幻数（DEADBEEF）交换字节并把它存入缓冲区，之后再次交换字节并在长度字段填入 0x18，在类型字段填入 0x02，在标志字段填入 0。最后向接下来的 0x10 字节填入随机数。（当然，填入的数据已经全部转换成网络传输所需要的大端格式了），如图 7-12 和图 7-13 所示。

```
push    ebp
mov     ebp, esp
sub     esp, 24h
mov     eax, dword_403000
xor     eax, ebp
mov     [ebp+var_4], eax
push    18h                 ; size_t
push    0                   ; int
lea     eax, [ebp+buf]
push    eax                 ; void *
call    memset
add     esp, 0Ch
push    0DEADBEEFh          ; hostlong
call    ds:htonl
mov     dword ptr [ebp+buf], eax
push    18h                 ; hostshort
call    ds:htons
mov     [ebp+var_18], ax
mov     [ebp+var_15], 0
mov     [ebp+var_16], 2
mov     [ebp+var_24], 0
jmp     short loc_401054
```

图 7-12 传递给函数的三个参数

更多资源请点击访问稀.酷客(www.ckook.com)

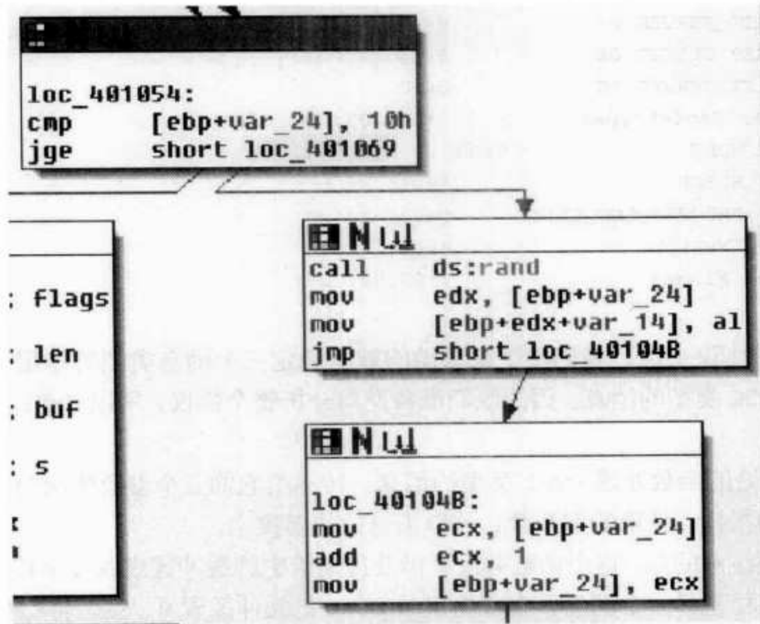


图 7-13　在接下来的 0x10 个字节里填上随机数

最后，把构造好的数据包发送给套接字，如图 7-14 所示。

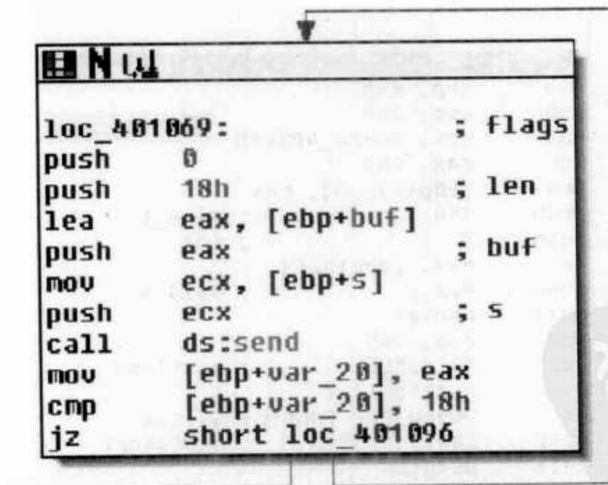


图 7-14　把构造好的数据包发送给套接字

从上面可以得知，这个函数生成预定义的数据包，然后把它返回给调用者，而接收的消息的内容则没用上。我们现在知道，包头部之后的字节被程序忽视了，跟在 0x02 消息之后的字节只是一些随机值。

更多资源请点击访问稀.酷客(www.ckook.com)

```
struct base_header{
        int MagicNumber;                /*Always 0xDEADBEEF*/
        unsigned short Len;             /*0018,0208*/ /*Greater than 8*/
#define PACKET_GETRANDOM                0x01
#define PACKET_RANDOM                   0x02
#define PACKET_SERVER_03                0x03
#define PACKET_CLIENT_04                0x04
#define PACKET_SERVER_5C                0x5C
         char PacketType;               /*01, 02, 03, 04*/
#define FLAG_NONE                       0x00
#define FLAG_SLEEP                      0x01
#define FLAG_PROCESS_AND_EXIT           0x02
#define FLAG_INVALID                    0x04
         char Flags;                    /*00,01,02*/
}
struct packet_get_random{
        struct base_header BaseHeader;
        char Ignored[16];
}
struct packet_random{
        struct base_header BaseHeader;
        char RandomValues[16];
}
```

0x004010B0 处的函数处理下一个消息。传递给它的也是上面提过的三个参数：指向消息缓冲区的指针，消息缓冲区的长度，读取消息处的套接字。

这个函数首先做的是确保消息的长度至少为 520 字节，如图 7-15 所示。

```
push    ebp
mov     ebp, esp
sub     esp, 214h
mov     eax, dword_403000
xor     eax, ebp
mov     [ebp+var_8], eax
cmp     [ebp+ReadBufferLen], 520
jnb     short loc_4010D3
```

图 7-15 确保消息长度至少为 520 个字节

这个数据包的整个处理是在一个基本模块里完成的，如图 7-16 所示。

更多资源请点击访问稀.酷客(www.ckook.com)

```
loc_4010D3:
mov     eax, [ebp+ReadBuffer]
mov     [ebp+var_214], eax
push    208h            ; size_t
push    0               ; int
lea     ecx, [ebp+buf]
push    ecx             ; void *
call    memset
add     esp, 0Ch
push    0DEADBEEFh      ; hostlong
call    ds:htonl
mov     dword ptr [ebp+buf], eax
push    208h            ; hostshort
call    ds:htons
mov     [ebp+var_20C], ax
mov     [ebp+var_209], 0
mov     [ebp+var_20A], 4
push    200h            ; size_t
mov     edx, [ebp+var_214]
add     edx, 8
push    edx             ; void *
lea     eax, [ebp+var_208]
push    eax             ; void *
call    memcpy
add     esp, 0Ch
push    0               ; flags
push    208h            ; len
lea     ecx, [ebp+buf]
push    ecx             ; buf
mov     edx, [ebp+s]
push    edx             ; s
call    ds:send
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 208h
jz      short loc_401177
```

图 7-16　处理数据包的独立的基本模块

这个基本模块用 memset 把栈缓冲区全部置 0。然后把幻数（字 DEADBEEF）交换字节并存入缓冲区的前四个字节。交换字节后它往长度字段填入 0x208，把类型设置为 4，把标志设置为 0。接下来，它把接收到的数据包里的 512 字节读出来放入将要发出的数据包里。

```
struct base_header{
        int MagicNumber;            /*Always 0xDEADBEEF*/
        unsigned short Len;         /*0018,0208*/ /*Greater than 8*/
#define PACKET_GETRANDOM            0x01
#define PACKET_RANDOM               0x02
#define PACKET_ECHO                 0x03
#define PACKET_ECHOREPLY            0x04
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
#define PACKET_SERVER_5C             0x5C
        char PacketType;             /*01, 02, 03, 04, 0x5C*/
#define FLAG_NONE                    0x00
#define FLAG_SLEEP                   0x01
#define FLAG_PROCESS_AND_EXIT        0x02
#define FLAG_INVALID                 0x04
        char Flags;                  /*00,01,02*/
}
/*must be 520 bytes long or longer*/
struct packet_get_echo{
        struct base_header BaseHeader;
        char RandomData[512];
}
struct packet_echoreply{
        struct base_header BaseHeader;
        char EchoData[512];
}
```

0x00401190 处的函数处理最后一个消息（类型 0x5C）。它和上面的函数一样接收同样的三个参数：指向消息的指针，消息的长度，读取消息处的套接字。这个消息也是在一个独立的基本模块里处理的，如图 7-17 所示。

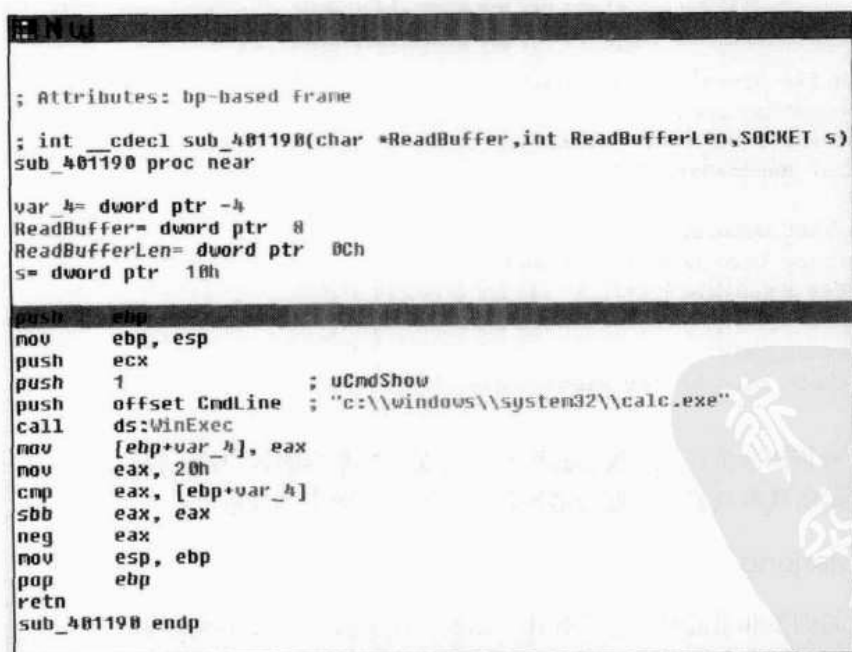


图 7-17 处理最后一个消息的独立的基本模块

更多资源请点击访问稀.酷客(www.ckook.com)

看来这个函数只是在服务器端打开一个计算器。最终的协议可以用下面的结构描述：

```
struct base_header{
        int MagicNumber;                /*Always 0xDEADBEEF*/
        unsigned short Len;             /*0018,0208*/ /*Greater than 8*/
#define PACKET_GETRANDOM                0x01
#define PACKET_RANDOM                   0x02
#define PACKET_ECHO                     0x03
#define PACKET_ECHOREPLY                0x04
#define PACKET_CALC                     0x5C
        char PacketType;                /*01, 02, 03, 04, 0x5C*/
#define FLAG_NONE                       0x00
#define FLAG_SLEEP                      0x01
#define FLAG_PROCESS_AND_EXIT           0x02
#define FLAG_INVALID                    0x04
        char Flags;                     /*00,01,02*/
}
struct packet_get_random{
        struct base_header BaseHeader;
        char Ignored[16];
}
struct packet_random{
        struct base_header BaseHeader;
        char RandomValues[16];
}
/*must be 520 bytes long or longer*/
struct packet_get_echo{
        struct base_header BaseHeader;
        char RandomData[512];
}
struct packet_echoreply{
        struct base_header BaseHeader;
        char EchoData[512];
}
struct packet_calc{
        struct base_header BaseHeader;
}
```

利用上面分析出来的结果，我们甚至可以实现一个兼容的客户端。我们现在可以确认这些就是这个服务器端所具有功能——服务器端没有其他什么隐藏的功能了。

7.2.3 Hit Marking

上面分析的例子非常简单。它只有几个函数，且全部函数都直接处理协议。对这样的例子来说，逆向工程整个程序是可能的，但实际上，逆向工程师极少会有逆向分析整个程序的余地。当程序的调用树如图7-18所示时，可以很容易地把需要的函数分离出来。

更多资源请点击访问稀.酷客(www.ckook.com)

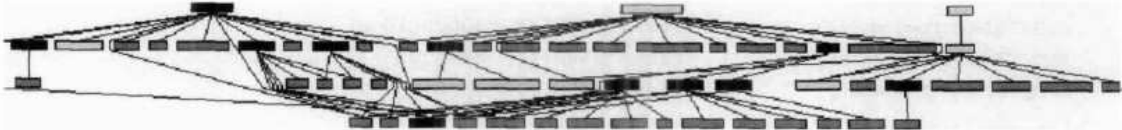


图 7-18 能够帮助我们识别函数功能的函数调用关系图

但是，如果调用树如图 7-19 所示，那我们的麻烦就大了。

图 7-19 难以识别函数功能的函数调用关系图，你有点麻烦了☹

这时，逆向工程师必须回答的第一个问题是，在这幅令人恐怖的图里哪些函数是用于处理消息的，哪些是我们可以忽视的。逆向工程师挑选这些函数的最常用的方法被称为 hit marking。

hit marking 是指记录处理每个消息时经过的判定树的调用路径。在每个函数或基本模块的开始处设置断点，当断点被触发时，记下被访问的函数，并让程序继续运行至下一个函数。在消息被完全处理后，把触发的断点的列表进行排序并去掉重复的函数，最后得到的列表就称为 hit list。通过分析 hit list 里的函数，我们应该可以拼凑出消息的结构。

图 7-20 显示了用 IDA 得到的 hit list（重复的部分还没有被去掉）。

Trace window

Thread	Address	Instruction
00000E40	.text:sub_401BD5	push esi
00000E40	.text:sub_401E1D	xor eax, eax
00000E40	.text:sub_401490	push ebp
00000E40	.text:sub_4016CC	cmp ecx, dword_403000
00000E40	.text:sub_4015A0	push ebp
0000152C	.text:sub_401000	push ebp
0000152C	.text:sub_4016CC	cmp ecx, dword_403000
0000152C	.text:sub_4016CC	cmp ecx, dword_403000

图 7-20 IDA 中的 short trace

遗憾的是，IDA 并没有提供直接生成 hit list 的功能。不过，市面上倒有一些第三方插件及应用程序，它们可以帮助你更好地完成这件事情，但我们不准备花这个钱，而是与 IDA 死磕。在本章的剩余部分里，我将会介绍怎样编写生成 hit list 的 IDA 插件。

为了生成 hit list，首先要做的就是列出程序中所有的函数。如果可以移去不感兴趣的函数，那就尽量去掉它们。为了获得函数列表，用 IDA 打开二进制文件，按 **shift+F3** 打开“function”窗口，在窗口里点击右键并选择“**copy**”（也可以用快捷键“**ctrl+ins**”），IDA 将把函数列表复制到剪贴板。把这个列表粘贴到编辑器里，你就会得到像下面这样的列表：

更多资源请点击访问稀.酷客(www.ckook.com)

```
_IID_ISAXErrorHandler          .text 01001308 00000010 R . . . . . . .
_IID_IXMLDOMDocument2          .text 01001318 00000019 R . . . . T .
_IID_ISchemaElement            .text 01001338 00000009 R . . . . . . .
_IID_ISchemaAttribute          .text 01001348 00000009 R . . . . . . .
_IID_ISchemaModelGroup         .text 01001358 0000000D R . . . . . . .
_IID_ISchemaComplexType        .text 01001368 0000000D R . . . . . . .
_IID_ISchemaType               .text 01001378 0000000D R . . . . . . .
_IID_ISchemaItem               .text 01001388 00000009 R . . . . . . .
_IID_ISAXAttributes            .text 010013A8 00000010 R . . . . . . .
```

在文本编辑器里把除函数地址之外的其他信息全部删除。结果应该像下面这样：

```
01001308
01001318
01001338
01001348
01001358
01001368
01001378
01001388
010013A8
```

IDA 调试器内置了跟踪功能。有两种使用方式：一是直接从菜单中选择 Debug | tracing window，二是在任意指令上点击右键，选择“execution trace”。从理论上讲，你也可以手工处理整个函数列表，为每一个断点加一个 trace。四天后，或许你的老板就会让你卷起铺盖走人。

为了更有效率地为每个函数添加跟踪点，我们需要编写一个高效的 IDA 插件。可以用 C、ruby 或者是 python 编写 IDA 插件。程序设计语言优劣之争了无新意，因此，我不准备蹚这趟浑水。在这里，我选择 python，因为我喜欢它（并不是说 python 比其他语言更好一些）。你可以从 www.d-dome.net/idapython/下载 IDAPython，并按说明安装。

我们设想这个插件可以自动为我们设置所有的跟踪点。看一个简单的例子，用几行 python 代码，你就可以在 0x004011E8 上设置断点。创建一个包含下列内容的.py 文件[①]。

```
#Set a breakpoint at 0x004011E8
from idautils import *
ea=ScreenEA()
ea=0x004011E8
add_bpt(ea, 1, 4)
```

现在，按 Alt+9，在弹出的对话框里选择刚才编写的.py 程序（要是你使用了默认的 Python 文件关联，这个文件的扩展名应该写成.py）。这时，0x004011E8 上应该设置了一个断点。我们可以打开 breakpoints 窗口验证一下（打开 breakpoints 窗口的快捷键是 Ctrl+Alt+B），结果应该如图 7-21 所示。

① 这里给出的代码是错误的，显然作者是随手写的其中第 2 行代码应改为：from idaapi import *。

更多资源请点击访问稀.酷客(www.ckook.com)

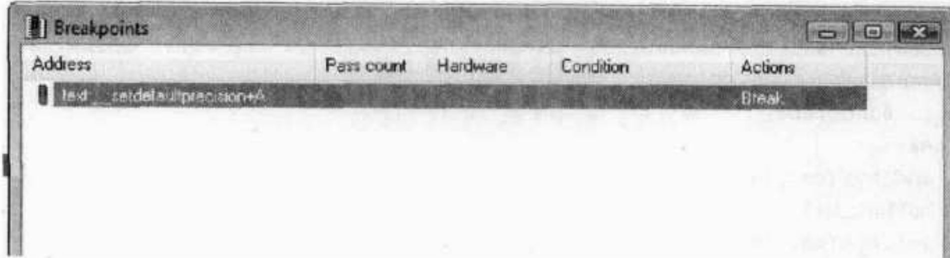


图 7-21 IDA 中的 breakpoints 窗口

OK，这个插件已经帮我们设置断点了，但我们还希望在每次触发断点时它都能创建记录。按 IDA 的术语来说，这应该算是跟踪而不是中断了。实际上，跟踪和中断并不矛盾。你可以设置成：当断点被触发时停止执行(中断)并同时记录这个事件(跟踪)。因为我们想找出处理特定消息的函数，所以要对所有的断点设置“跟踪”，在 IDA 里要做到这一点，只要在相关设置上把断点的标志位去掉，加上跟踪的标志位就可以了。下面给出的代码就是在 0x004011E8 上先设一个断点，然后再把它设为“跟踪”.[①]

```
from idautils import *
ea=ScreenEA()
ea=0x004011E8
add_bpt(ea, 1, 4)
bp=bpt_t()
get_bpt(ea, bp)
bp.flags^=BPT_BRK
bp.flags|=BPT_TRACE
```

我们现在可以在 IDA 里设置跟踪点了。由于之前有了函数地址列表，接下来的任务比较简单，我们只要加一个循环，并且在每个函数起始位置上设置跟踪点就行了[②]。

① 译者在测试这段代码时发现了一点问题，就是虽然 flags 已经改成了 BPT_TRACE，但是运行这段代码之后手工编辑断点，还是会发现这个断点仍然是 break 而不是 trace，而且运行 IDA 调试器的话，代码运行到这个位置会停下而不是记录日志。似乎上面这段代码不起作用。不过不管怎么说，直接调用 IDA 内置的函数就可以解决这个问题。比如你可以把最下面的 4 行代码改为下面这句代码：

```
SetBptAttr(ea,BPTATTR_FLAGS,BPT_TRACE)
```

就可以了。

② 这段代码是有错误的，下面给出我的一个实现：

```
from idaapi import *
from idautils import *
funclist = [0x004011F2, 0x00401200, 0x00401206]
ea=ScreenEA()
for line in funclist:
    ea = line
    add_bpt(ea,1,4)
    SetBptAttr(ea,BPTATTR_FLAGS,BPT_TRACE)
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
from idautils import *
funclist=[0x004011F2, 0x00401200, 0x00401206] /*add all the other addresses here*/
ea=ScreenEA()
for i in funclist:
       ea=i
       add_bpt(ea, 1, 4)
       bp=bpt_t()
       get_bpt(ea, bp)
       bp.flags ^=BPT_BRK
       bp.flags|=BPT_TRACE
```

对于那些必须为每一个函数都标记跟踪点的情况，这个 IDA 插件可以自动找出每个函数引用，而不需要我们手动操作。但实际上，有一些函数你可能想把它们立刻删除，因此，上面这个例子反而比全自动的例子用得更多一些。出于完整性考虑，下面这个 python 脚本自动为每个函数设置跟踪点[①]。

```
from idautils import *
# Loop through all the functions and add a breakpoint
for i in range(get_func_qty() ):
    f=getn_func(i)
    print "Function %s at 0x%x" % (GetFunctionName(f.startEA), f.startEA)
    add_bpt(f.startEA, 1, 4)
#change all the breakpoints to trace-only
for i in range(get_bpt_qty() ):
    b=bpt_t()
    getn_bpt(i, b)
    b.flags ^=BPT_BRK
    b.flags|=BPT_TRACE
    update_bpt(b)
```

现在，每个函数的起始位置上都设了跟踪点，运行一下客户端程序就可以生成 hit list 了。

7.2.4 Hitlist 示例

之前那个例子太简单了，显示不出 hit list 的强大威力。我们准备以较大的程序 Pidgin（www.pidgin.im）为例。Pidgin 是一款开源的网络聊天程序，它支持网络聊天程序所使用的多种协议。因为它是开源的，在练习中碰到麻烦时，可以比较反汇编的结果和程序的二进制源代码。

在这个例子里，我准备生成一份 Pidgin v2.1.1 在 Windows 中的 hit list。Pidgin 以插件的形式

① 下面给出我的一个实现

```
from idaapi import *
for i in range(get_func_qty() ):
    f = getn_func(i)
    print "Function %s at 0x%x" % (GetFunctionName(f.startEA), f.startEA)
    add_bpt(f.startEA, 1, 4)
    SetBptAttr(f.startEA,BPTATTR_FLAGS,BPT_TRACE)
```

更多资源请点击访问稀.酷客(www.ckook.com)

实现聊天协议。我们就挑雅虎通（Yahoo! Instant Messenger）来演示吧。这个协议的逻辑是在libyahoo.dll中实现的。这个DLL的调用树非常庞大和复杂，而IDA中的绘制结果极为复杂几乎非人力所能解释。其调用树如图7-22所示。

图7-22 IDA给出的复杂示意图

在这个例子里，我准备用上一节的脚本把每个函数都标记跟踪点。据统计，这个DLL总共有549个函数。第一次分析时，我们只对与初始化及登录相关的函数感兴趣。用hit list应该可以把需要分析的函数数量从549缩小到一个可控的范围。

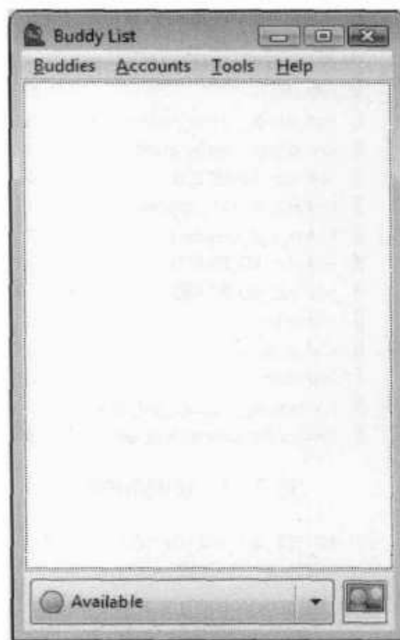


图7-23 登录完成

首先，注册一个雅虎通账号，用Pidgin登录一下，确认一切都工作正常。接下来，在IDA中打开libyahoo.dll，在菜单Debugger→Process Options中设好Pidgin可执行文件所在的路径，以便在执行调试器时运行Pidgin。最后，按下“Alt+9”快捷键，调用上一节给出的python脚本为每个函数标记跟踪点。让IDA执行脚本，我们可以抽空冲杯咖啡了（因为程序在IDA的调试器里运行的比平时慢一些）。当Pidgin底下的那一栏显示“Available”时，表示初始化工作完成了，如图7-23所示。

现在我们应该看一下IDA的跟踪窗口，看看连接Yahoo!时都调用了哪些函数。为了了解协议的登录部分，我们需要分析这些函数，通过跟踪窗口我们大致可以了解工作量的上限。

在跟踪窗口（在菜单栏上选Debugger→Tracing→Trace Windows）中各个函数是以其被调用的顺序列出来的，而断点列表（在菜单栏上选Debugger→Breakpoints→Breakpoint List）更接近真实的hit list一些，它列出了所有的函数以及各个函数被调用的次数，如图7-24所示。

如果我们把没用到的函数删除，将只剩下133个函数需要分析了。这把我们分析的范围一下缩小到原来的24%。而剩下的这些函数中又有很多函数只是用来封装其他函数的（只是调用hit list中其他的函数）。我现在想找一个经常被调用，但也不是被调用太过频繁（比如说被调用了几百次）的函数来分析。大致看一下，就会很快找到图7-25中显示的函数。

一般来说，在分析协议之前，先dump一些数据包以备比较总不会错。下面是我的客户端发给服务器的第一个消息（十六进制表示）。

更多资源请点击访问稀.酷客(www.ckook.com)

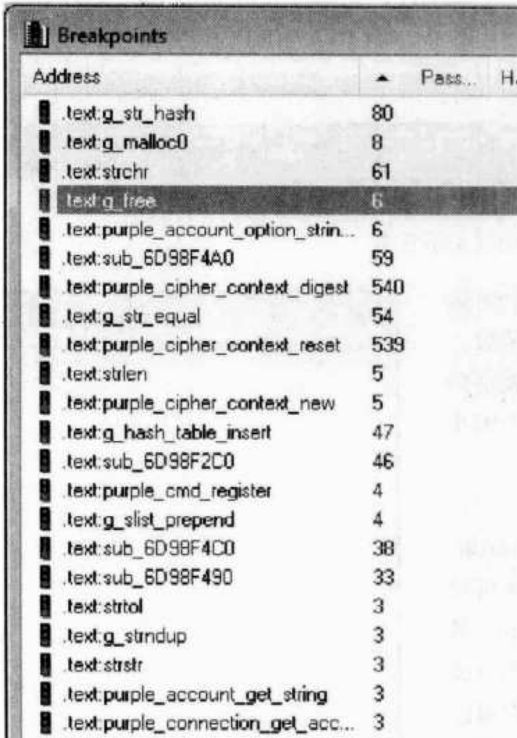


图 7-24　被调用的断点

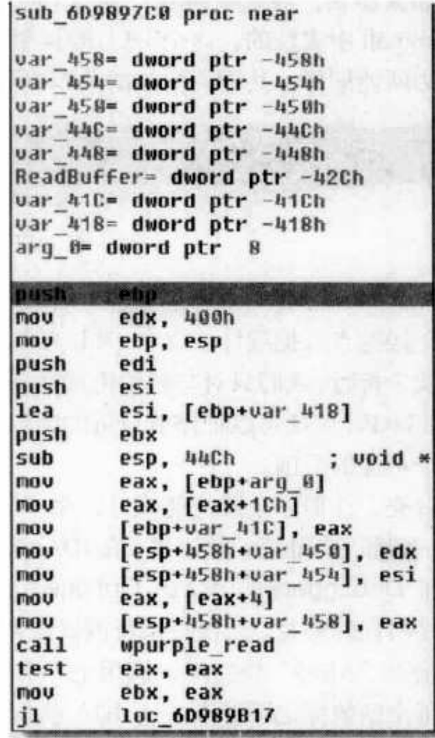


图 7-25　一个经常被调用的函数

```
59 4D 53 47 00 0F 00 00 13 00 57 00 00 00 00 00 00 00 00 31 C0 80 69 64 61 70 6C 75
67 69 6E 31 32 33 34 35 C0 80
```

选中的这个函数一开始就调用了 wpurple_read 这个函数，这让我眼前一亮（因为 wpurple_read 是用来从网上读取数据的）。这个函数判断读入的数据大于 4 个字节之后，就把数据复制到另一个缓冲区，然后进入处理数据包的第一个模块。第一个协议处理模块如图 7-26 所示。

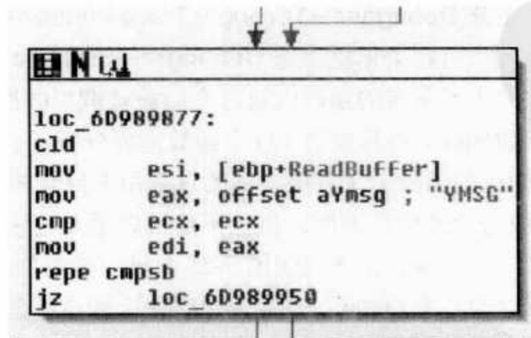


图 7-26　第一个协议处理模块

更多资源请点击访问稀.酷客(www.ckook.com)

在这个模块中，程序检查数据包的前四个字节是不是字符串 YMSG，如果不匹配将退出。由此可见，YMSG 应该是协议中的幻数，如图 7-27 所示。

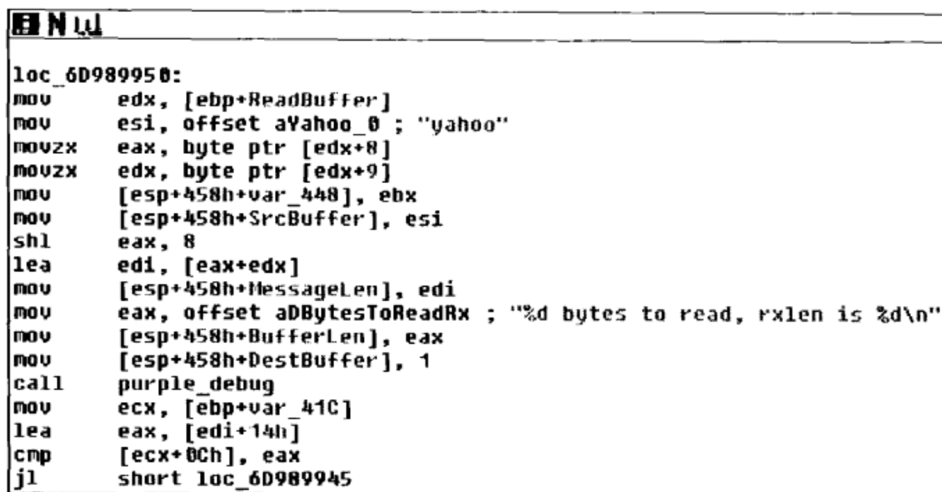


图 7-27　YMSG 幻数

接下来好像是检查长度字段。程序把数据包中的第 8 和第 9 两个字节读出来，把第 8 个字节放在左边并把二者相加，因此，我们猜测长度字段是大端字节序的 short。所以我们可以假设协议的结构如下所示：

```
struct login_packet{
        char Magic[4];            /*Always YMSG*/
        char unknown1;
        char unknown2;
        char unknown3;
        char unknown4;
        unsigned short Len;       /*Big Endian*/
}
```

雅虎通协议中的处理过程及调用的函数数量远大于前一节的例子，但是从二进制文件中提取协议的思路是一致的：程序从网络上读取数据，然后由一系列的函数对数据包进行处理。顺着这个思路，你应该可以把雅虎通协议的其余部分搞清楚。

更多资源请点击访问稀.酷客(www.ckook.com)

第 8 章

高级攻略

本章内容：

- 逆向分析恶意软件

更多资源请点击访问稀.酷客(www.ckook.com)

8.1 导言

通过前几章的学习，你应该可以自己动手做一些分析了。本章，我们将分析一个真实的恶意代码中的一部分。这个恶意软件可是一个真家伙，你在处理它的时候要格外小心，否则可能会严重影响你的计算机和网络。此外还要确认一下你所处的环境是否允许分析这类程序。我强烈建议你在虚拟化软件（例如 VMware）里做这类分析，我特别喜欢 VMware 的快照功能，这一功能可以保存某个时间点的系统状态（还原点），并在需要时把系统恢复到还原点的状态。这样一来，不管是旅游还是工作，我们都可以灵活安排时间，而不用担心有人把电源拔了，或者必须随身带个笔记本电脑。总之，你要牢记：一人做事一人当。在这一意义上，之所以把本章称为“高级攻略”，并不是说本章的内容有多么高深，而是说我们将在本章综合运用前几章介绍的内容，使你对逆向工程有一个更清晰的认识。

注意

像其他事情一样，你的经验越丰富，做起事来就越顺手。因此，一旦掌握基础知识，就应该做大量的练习，增加经验值。当然，前进的道路不可能一帆风顺，在遇到挫折时可以求助互联网，有许多组织或网站会发起各种挑战赛，比如 Honey Net 发起的计算机应急事件响应的系列挑战赛。这不仅比较好玩，而且也是提升经验值的好方法，不过 Honey Net 挑战赛的重点往往并不在逆向工程上。

说到逆向工程，网上有很多不错的站点（比如说国内的看雪学院，http://bbs.pediy.com），其中比较有名的是http://crackmes.de。这个网站上有很多网友提供的供逆向分析的程序。例如，unpackme 提供脱壳的练习，crackme 提供模拟破解商业软件保护机制的程序，等等。它们为不同的操作系统出了难题，而且有不同的难度级别。

第二个要介绍给大家的网站更像是一个论坛（http://community.reverse-engineering.net/），尽管它也有一些 crackme 之类的挑战，但它的亮点在于论坛上活跃的“一群大牛”，你可以获取一些最新的信息并有可能结识这些“大牛”。

第三个网站是 Offensive Computing（http://www.offensivecomputing.com/）。这个网站可以说是一个恶意软件的仓库。在经验达到一定的程度后，你可以去这个网站下载一些恶意软件进行分析，分析这些真实的恶意软件一定会让你的经验值暴涨。

最后要介绍的是 OpenRCE（http://www.openrce.org），它由 TippingPoint 公司一位多才多艺的逆向工程师——Pedram Amini 维护。除了论坛，它还提供插件、脚本之类的工具下载。此外，还提供了像 Win32 Call Chains Database 这样的资料，这些东西可能比你想象的还要有用。

更多资源请点击访问稀.酷客(www.ckook.com)

8.2 逆向分析恶意软件

正如我在前面所提到的，我们要分析的这个程序严格意义上讲其实并不是一个恶意软件，而是一个广告软件（比恶意软件的危害要小一些），但它的确是现实中容易遇到的真实程序，特别是如果你在计算机应急响应小组或类似的单位里工作过，肯定碰到过这类程序。许多人喜欢使用 regmon 或者 filemon 之类的工具[1]，特别是在商业公司里，大家都较劲看谁能先把病毒的行为特征给分析出来时，它们是比较有用的。不过，这并不是逆向工程，我们是逆向工程师而并不是预言家，如果有时间的话，我劝大家不要用这样的工具，你想知道程序正在做些什么，并不是使用这些工具的技术上的理由，除非你并不想逆向分析二进制文件。另一个理由是:这样做是比较危险的，你让程序接管控制权，但并不真正知道它将要做些什么，而你又无访问文件系统、注册表或标准 API 的必要。最后要说明的是，如果碰上正当的应用程序，这个方法将没什么作用，何况这是一本讲逆向工程的书呢。不过话虽如此，知道如何使用系统监视工具也并不是一件坏事。有时老板给的时间实在是太紧了，那么让恶意程序跑一遍，看看它究竟干了些什么，然后写报告交差也许是个不错的选择。你可以在 www.syngress.com/solutions 下载此程序，解压缩的口令是：'!DANGER!INFECTEDMALWARE!DANGER!'。

现在，我们开始查看这个代码，这显然不是一个标准的入口，看起来像某些形式的自修改代码的特征，就是说它被加壳了。在这一区段里，我们至少有三条线索。第一条（也是最明显的）是这个区段的名字是 UPX1，这是用 UPX 打包过的程序的特征。第二条线索是位于我们进入点之前的十六进制数据，这很像是打包并加密过的代码/字符串之类的东西，最后的标记是——这需要一点经验（但不需要很多）——pusha 指令。UPX 在操作之前用 pusha 保存所有的寄存器信息，之后会在一条无条件跳指令之前再用 popa 恢复它们。因此，碰到 popa/jmp 时我们可以休息一下，然后再决定是继续跟下去还是把这些数据转储到硬盘上。我们应该进一步确认接下来的事情是否符合我们的预期。我们可以通过查看和（或）单步调试代码，确认 popa/jmp 组合是否真的会出现。你通常首先查看控制流；如果有必要也可以动态进行。我们在这里不准备这样做，因为真的没必要用 15 页的篇幅罗列 UPX 解压缩应用程序的过程，以确认控制权没有跳转到什么奇怪的地方。还有，我不推荐你在工作机上做这些，强烈建议使用 VMware 或类似的虚拟机。

如图 8-1 所示，我们所预期的正好出现在 loc_40A081[2]，是 popa/jmp 指令。为了使用 IDA 的脱壳插件（unpacker），我们需要知道代码的 OEP。然而，因为恶意软件并没有把 UPX 用于反跟踪的目的，所以 OEP 很明显就在 word_40395E。下面我们快速检查一下这个地址，看它是否对头（参见图 8-2）。

8

① 这两个工具的功能已经合并到 Procmon.exe 里了，你可以去 sysinternals 网站下载这些工具。

② 原文如此，可能是 loc_40A087 的笔误。

更多资源请点击访问稀.酷客(www.ckook.com)

```
UPX1:0040A06A loc_40A06A:                          ; CODE XREF: UPX1:0040A062↑j
UPX1:0040A06A                  mov     ecx, 0AEF24857h
UPX1:0040A06F                  push    ebp
UPX1:0040A070                  call    dword ptr [esi+0A484h]
UPX1:0040A076                  or      eax, eax
UPX1:0040A078                  jz      short loc_40A081
UPX1:0040A07A                  mov     [ebx], eax
UPX1:0040A07C                  add     ebx, 4
UPX1:0040A07F                  jmp     short loc_40A059
UPX1:0040A081 ; ---------------------------------------------------------------
UPX1:0040A081
UPX1:0040A081 loc_40A081:                          ; CODE XREF: UPX1:0040A078↑j
UPX1:0040A081                  call    dword ptr [esi+0A488h]
UPX1:0040A087
UPX1:0040A087 loc_40A087:                          ; CODE XREF: UPX1:0040A040↑j
UPX1:0040A087                  popa
UPX1:0040A088                  jmp     near ptr word_40395E
UPX1:0040A088 ; ---------------------------------------------------------------
```

图 8-1　popa/jmp 指令组合

```
UPX1:00408000          dd 0AF4BF3Ch, 59BF2E05h, 0F72CB9FBh, 37400189h, 0FBFF5060h
UPX1:00408000          dd 15E16F1Bh, 5953426Ch, 0C98309A5h, 6028BFFFh, 0B7BF2B0Eh
UPX1:00408000          dd 0F2C0337Fh, 2BD1F7AEh, 0F78BD1FEh, 11635813h, 7FDBF7CCh
UPX1:00408000          dd 0C14FCA8Bh, 0A5F302E9h, 3E18307h, 242CA4F3h, 0C1B58B99h
UPX1:00408000          dd 272CF72Ah, 0B2AF2A16h, 20BF4F60h, 582A1860h, 7746B363h
UPX1:00408000          dd 7C2CC120h, 3E9B2141h, 5FC8918Ch, 1CEC8111h, 0F68B530Eh
UPX1:00408000          dd 9CEEFB5Bh, 55072824h, 40242D8Bh, 0DB85C43Ah, 17C3840Fh
UPX1:00408000          dd 0FCDED982h, 24B48B00h, 53036A30h, 0A268E456h, 0C2B67B9Bh
UPX1:00408000          dd 8E1B0D0Ch, 8DE562FBh, 0E1980D82h, 0C2182494h, 9212FA6Bh
UPX1:00408000          dd 70C16BAEh, 0C0507222h, 590FB970h, 0BDC2FA1Bh, 1C24848Dh
UPX1:00408000          dd 0BBCCE038h, 8C8B2F9Fh, 51933824h, 532015FFh, 0FDAC10BEh
UPX1:00408000          dd 20868EEh, 0FFE92604h, 92B48DD5h, 16C7D9DDh, 8C8D20EBh
UPX1:00408000          dd 5118041Fh, 16CD7756h, 45C207ECh, 80B73D1Ch, 0B35B2A73h
UPX1:00408000          dd 6468DDB9h, 458688Ah, 6501816h, 6685F058h, 0D005E62Bh
UPX1:00408000          dd 815B5D1Bh, 9B3982C4h, 0CC216FDh, 5256CA00h, 0D8397718h
UPX1:00408000          dd 7E1274EEh, 50130528h, 7C10EB53h, 3E3B3305h, 2C55111h
UPX1:00408000          dd 35CE355Ch, 0A72989Fh, 7150029Ch, 63962CCFh, 1048588h
UPX1:00408000          dd 0E5D0EAEh, 8316CD87h, 254CBF38h, 9EE0DC58h, 68281EF0h
```

图 8-2　检查地址

看这个地址，它指向一个偏移量，那里有一长串的十六进制字符，在进入点没有指向原始文件真正的起始位置时，这正是我们所预期的。因此我们有理由假设所看到的就是 UPX，如果我们能把它脱壳就可以证实了。最近的 IDA 版本（自 4.8 以后）都包括了一个通用的脱壳插件。我们下面简单的介绍一下这个插件。不过就像前几章一样，我们不会对脱壳讨论的太过深入。

在 IDA 里使用插件很简单，通过菜单 Edit→Plugins→Universal PE unpacker 或按下 Alt+1[①]（参见图 8-3）皆可。接下来会弹出一个警告对话框，警告你使用这个插件可能会有一定的风险。如果你选择确定，则会弹出如图 8-4 所示的对话框：

正如你所见到的，知道精确的 OEP 并没有太大必要。从根本上说，这个技术并不是很高级，因为调试器并不知道它正在做什么。它所做的只是监视 Start address 到 End address 这段范围内可

① 此快捷键与具体的配置文件有关，不一定都有效。

更多资源请点击访问稀.酷客(www.ckook.com)

执行的指令。在这个例子里，我根据二进制文件把默认的 End address 参数改了一下，然后点击了 OK。当 IDA 检测到符合必要的条件时，它将弹出对话框提示你（参见图 8-5)。

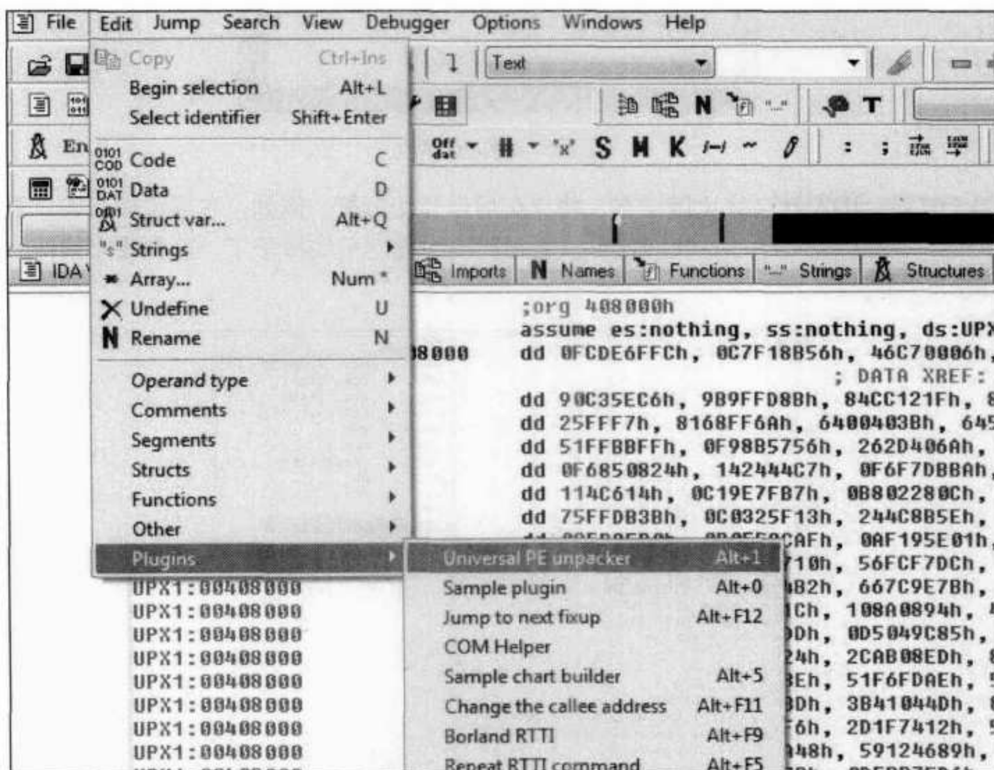


图 8-3 Universal PE Unpacker 插件

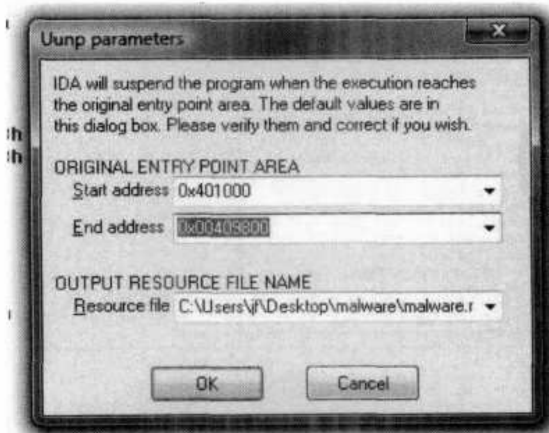


图 8-4 输入脱壳所需的参数

8

更多资源请点击访问稀.酷客(www.ckook.com)

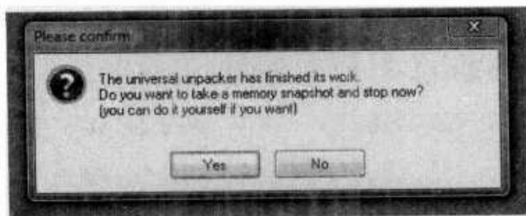


图 8-5 确认对话框

点击 OK 后，IDA 就会终止调试器，并且在后台开始分析。例如，它重建输入表，重新分析代码流，等等。这些完成之后，我们就可以看到代码脱壳之后的样子了，这和我们预计的相差不远。(如图 8-6 所示)

```
; Attributes: bp-based frame

public start
start proc near

var_78= dword ptr -78h
var_74= dword ptr -74h
var_70= dword ptr -70h
var_6C= dword ptr -6Ch
var_68= dword ptr -68h
var_64= dword ptr -64h
var_60= dword ptr -60h
StartupInfo= _STARTUPINFOA ptr -5Ch
var_18= dword ptr -18h
var_14= dword ptr -14h
var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    0FFFFFFFFh
push    offset unk_404538
push    offset _except_handler3
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
sub     esp, 68h
push    ebx
push    esi
push    edi
mov     [ebp+var_18], esp
xor     ebx, ebx
mov     [ebp+var_4], ebx
push    2
call    __set_app_type
pop     ecx
or      dword ptr unk_406454, 0FFFFFFFFh
or      dword ptr unk_406458, 0FFFFFFFFh
call    __p__fmode
mov     ecx, dword ptr unk_406448
mov     [eax], ecx
call    __p__commode
mov     ecx, dword ptr unk_406444
mov     [eax], ecx
mov     eax, _adjust_fdiv
mov     eax, [eax]
mov     dword ptr unk_406450, eax
call    nullsub_1
cmp     dword ptr unk_4062A0, ebx
jnz     short loc_4039E1
```

```
mov     eax, [ebp+var_14]
mov     ecx, [eax]
mov     ecx, [ecx]
mov     [ebp+var_78], ecx
push    eax
push    ecx
call    _XcptFilter
pop     ecx
pop     ecx
retn
start endp ; sp = -8
```

图 8-6 脱壳后的代码

查看代码，我们会看到典型的例程入口。然而，如果注意到对例程的引用（例如 nullsub_1），如果再深入一点，你将会发现如图 8-7 这样的引用。

更多资源请点击访问稀.酷客(www.ckook.com)

```
.idata:00404174                 extrn __imp__941:dword ; DATA XREF: _941↑r
.idata:00404178                 extrn __imp__1576:dword ; DATA XREF: _1576↑r
.idata:0040417C                 extrn __imp__2976:dword ; DATA XREF: _2976↑r
.idata:00404180                 extrn __imp__3081:dword ; DATA XREF: _3081↑r
.idata:00404184                 extrn __imp__2985:dword ; DATA XREF: _2985↑r
.idata:00404188                 extrn __imp__3262:dword ; DATA XREF: _3262↑r
.idata:0040418C                 extrn __imp__3136:dword ; DATA XREF: _3136↑r
.idata:00404190                 extrn __imp__1776:dword ; DATA XREF: _1776↑r
.idata:00404194                 extrn __imp__101_1:dword ; DATA XREF: _101_1↑r
.idata:00404198                 extrn __imp__101_2:dword ; DATA XREF: _101_2↑r
.idata:0040419C                 extrn __imp__101_3:dword ; DATA XREF: _101_3↑r
.idata:004041A0                 extrn __imp__5714:dword ; DATA XREF: _5714↑r
.idata:004041A4                 extrn __imp__5289:dword ; DATA XREF: _5289↑r
.idata:004041A8                 extrn __imp__5307:dword ; DATA XREF: _5307↑r
.idata:004041AC                 extrn __imp__4698:dword ; DATA XREF: _4698↑r
.idata:004041B0                 extrn __imp__4079:dword ; DATA XREF: _4079↑r
.idata:004041B4                 extrn __imp__2725:dword ; DATA XREF: _2725↑r
.idata:004041B8                 extrn __imp__5302:dword ; DATA XREF: _5302↑r
.idata:004041BC                 extrn __imp__5300:dword ; DATA XREF: _5300↑r
.idata:004041C0                 extrn __imp__3346:dword ; DATA XREF: _3346↑r
.idata:004041C4                 extrn __imp__2396:dword ; DATA XREF: _2396↑r
.idata:004041C8                 extrn __imp__5199:dword ; DATA XREF: _5199↑r
.idata:004041CC                 extrn __imp__1089:dword ; DATA XREF: _1089↑r
.idata:004041D0                 extrn __imp__3922:dword ; DATA XREF: _3922↑r
.idata:004041D4                 extrn __imp__5731:dword ; DATA XREF: _5731↑r
.idata:004041D8                 extrn __imp__2512:dword ; DATA XREF: _2512↑r
.idata:004041DC                 extrn __imp__2554:dword ; DATA XREF: _2554↑r
.idata:004041E0                 extrn __imp__4486:dword ; DATA XREF: _4486↑r
.idata:004041E4                 extrn __imp__6375:dword ; DATA XREF: _6375↑r
```

图 8-7　例程引用

看来，IDA 并没有重新分析所有的输入函数。产生这种情况的原因是，没有加载适当的 FLIRT 签名库[①]。这个问题不大，因为我们可以亲自动手。如果你不清楚该怎么做，那就容我再啰嗦两句吧。

要加载 FLIRT 签名库文件，在 IDA 的菜单栏里选择 File→Load File→FLIRT signature file（如图 8-8 所示），这时 IDA 会弹出一个列有当前所有可用的 FLIRT 签名库文件的窗口，如图 8-9 所示。

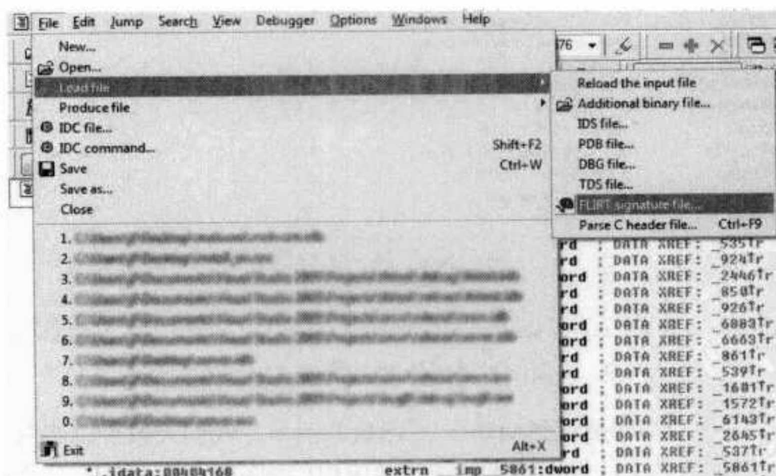


图 8-8　加载 FLIRT 库

① IDA 在默认情况下不会主动加载 FLIRT 签名库，除非它识别出被分析的程序是用哪种编译器编译产生的。由于这个程序是我们刚刚用脱壳插件处理过的，所以现在 IDA 显然没有发现它是用哪个编译器生成的，所以也就不会加载适当的 FLIRT 签名库。

更多资源请点击访问稀酷客(www.ckook.com)

图 8-9　当前所有可用的 FLIRT 签名库文件的清单

看到这个提示窗口，特别是看到编译器/库函数的清单，我们大致可以明白它是做什么用的了。也就是说，FLIRT 签名文件根据给定的编译器应用已知的特征。但我们怎么知道所用的是什么编译器呢？乍一看，这几乎不可能解决，但如果有一些经验，这还是比较容易的。一般情况下，我们可以看二进制文件中的字符串，因为编译器一般会在其中嵌入广告性的字符串（通常会包括编译器名及版本号）。如果没有这些信息，我们也可以根据生成的代码猜测所使用的编译器。查看代码时该找些什么有点超出了本书的范围，但随着时间的推移你会明白怎么做。例如，如果你

更多资源请点击访问稀.酷客(www.ckook.com)

看过 GCC 生成的许多代码，或许会注意到它喜欢在栈上分配超出需要的空间，然后在接下来的一条或两条指令中改正这点，你所需要寻找的就是这些东西。在我们的例子里，这些字符串如图 8-10 所示。

IDA View-A | Hex View-A | Exports | Imports | Names | Functions | Strings

Address	Length	Type	String
"..." UPX0:0...	0000000C	C	wininit.ini
"..." UPX0:0...	00000008	C	%s=%s\r\n
"..." UPX0:0...	00000009	C	Kernel32
"..." UPX0:0...	0000000C	C	MoveFileExA
"..." UPX0:0...	00000005	C	.tmp
"..." UPX0:0...	0000000C	C	Ad_AdSen_20
"..." UPX0:0...	00000017	C	SOFTWARE\\Alexa Toolbar
"..." UPX0:0...	00000017	C	RegisterServiceProcess
"..." UPX0:0...	00000009	C	KERNEL32
"..." UPX0:0...	0000000B	C	\\AdsNT.exe
"..." UPX0:0...	00000006	C	AdsNT
"..." UPX0:0...	00000008	C	Version
"..." UPX0:0...	0000000B	C	\\AdsNT.ini
"..." UPX0:0...	0000000B	C	\\index.htm
"..." UPX0:0...	0000000A	C	Referer:
"..." UPX0:0...	00000006	C	adsnt
"..." UPX0:0...	00000040	C	SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\policies\\Explorer\\R...
"..." UPX0:0...	0000000B	C	adsntB/1.6
"..." UPX0:0...	0000000E	C	AdsNTGroupURL
"..." UPX0:0...	00000009	C	gurl%d%d
"..." UPX0:0...	00000008	C	group%d
"..." UPX0:0...	0000000C	C	showStyle%d
"..." UPX0:0...	0000000F	C	showIEWindow%d
"..." UPX0:0...	00000009	C	weight%d
"..." UPX0:0...	00000009	C	objurl%d
"..." UPX0:0...	00000006	C	url%d
"..." UPX0:0...	00000009	C	height%d
"..." UPX0:0...	00000008	C	width%d
"..." UPX0:0...	00000009	C	toppos%d
"..." UPX0:0...	00000009	C	AdsNTURL
"..." UPX0:0...	0000000A	C	leftpos%d
"..." UPX0:0...	00000006	C	AdNum
"..." .newIID:...	0000000D	C	ADVAPI32.dll
"..." .newIID:...	0000000F	C	RegSetValueExA
"..." .newIID:...	00000011	C	RegQueryValueExA
"..." .newIID:...	0000000E	C	RegOpenKeyExA
"..." .newIID:...	0000000C	C	RegCloseKey
"..." .newIID:...	00000010	C	RegCreateKeyExA
"..." .newIID:...	0000000D	C	kernel32.dll
"..." .newIID:...	00000011	C	GetModuleHandleA
"..." .newIID:...	0000000B	C	GetVersion
"..." .newIID:...	0000000A	C	CopyFileA
"..." .newIID:...	00000012	C	GetShortPathNameA
"..." .newIID:...	0000000D	C	ReleaseMutex
"..." .newIID:...	0000000D	C	GetLastError
"..." .newIID:...	0000000D	C	CreateMutexA
"..." .newIID:...	00000014	C	GetCurrentProcessId

图 8-10 恶意软件中的字符串

更多资源请点击访问稀.酷客(www.ckook.com)

遗憾的是，我们没有看到任何与编译器相关的字符串，所以我们只好去猜了。怎么猜呢？就是逐个加载 FLIRT 签名文件，直到 IDA 能识别出大多数的函数。在我们的例子里，基本上就是这样做的。有许多签名文件需要匹配。一般首选（也是最通用的）MS Visual C++ runtime 签名库，然后再尝试用 MFC 开头的签名库来获得大多数结果。最后，我设法识别出了几乎所有的函数，虽然这个二进制文件有一些问题，但完全是因为它使用的是 MFC 4.2。我们基本上还是全部识别出来了。在靠近进入点函数结尾的部，我们看到结尾部分的调用已经被解析成 WinMain( )，如图 8-11 所示[①]。

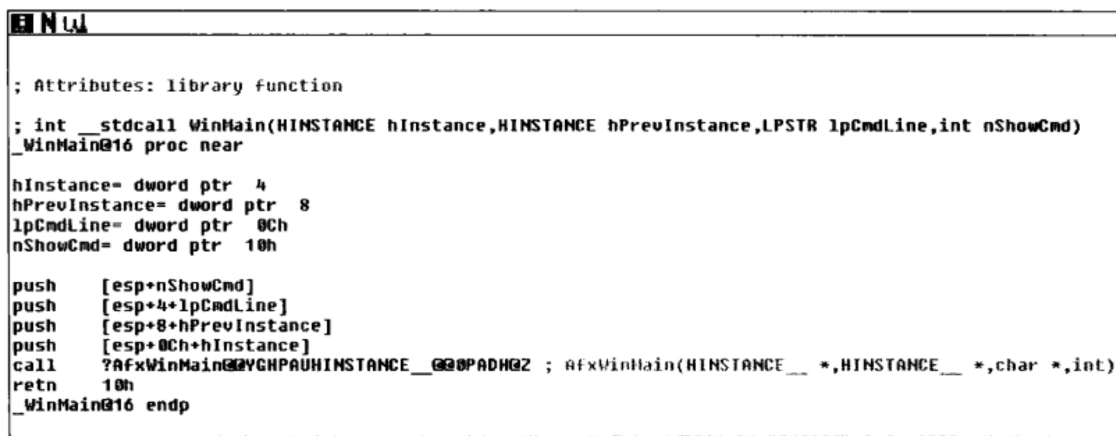


图 8-11　WinMain()例程

比较有意思的是，这个代码使用了 MFC。在我的印象里，曾分析过的恶意软件中只有 2～3 个使用了 MFC。虽然我没有学过太多关于 MFC 的知识，但我的从业经历还算顺畅，因为我们大家都可以保持这种趋势吧。因此，我在这个调用（AfxWinMain）上设置了一个断点，然后步入，老天保佑我们不要碰到大量的 MFC 例行代码（参见图 8-12）。

这个函数比较简单，而我们寻找目标的过程也比较轻松。因为这个恶意软件我也没有分析过，所以并不能完全确定它到底会干些什么，再加上对 AfxWinMain()不太熟，所以我就步过了对其他 MFC API 的内部调用。我注意到这里有两个对 EAX 寄存器的调用，其中之一可能就是我们要找的东西。而其中的第二个调用正是我们要寻找的，且正如我们在前面注意到的，它将控制权交回给应用程序的 sub_401800[②]。现在终止调试器，然在 sub_401800 处进行静态分析。在停止调试器之后，我想看一下 functions 标签（这个标签页位于 names 标签和 strings 标签之间），如果在此没找到，可能是你随手把它关闭了，可以在菜单栏中点击 View→Open Subviews→Functions

① 用 UPX 脱壳后，再用 PEiD 查看，就会发现这个程序是用 VC6.0 编译的，但因为本书是讲 IDA 的，所以就显得有些舍近求远了。

② 如果你熟悉 MFC 的话，特别是对 AfxWinMain()还算了解的话，你一眼就能看出来：这实际上是用户重写的 CXxxApp:: InitInstance()方法，Xxx 表示项目的名称。

更多资源请点击访问稀.酷客(www.ckook.com)

或者直接使用组合键 Shift+F3 重新打开 Functions 标签。在 Functions 标签中选择你要找的函数后，就会出现如图 8-13 所示的界面。

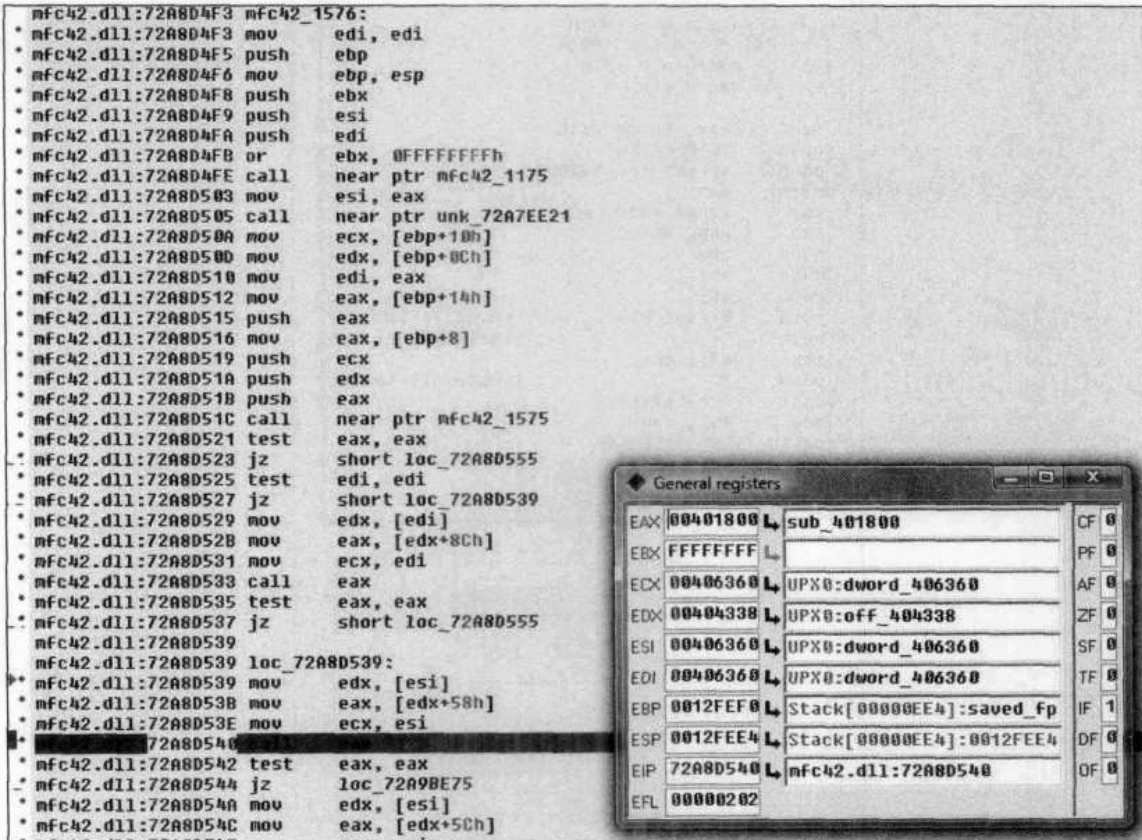


图 8-12 通用寄存器

如我们所见，这个子例程没有接收参数，且本身只有少量的参数。这个过程的 prologue 是很标准的。这个过程里第一段有实际意义的代码是用 Ad_AdSen_20 参数来调用一个 CreateMutexA()函数。CreateMutexA()通常用于两个线程间的同步访问，究竟是不是这样呆会儿就知道了。不过，这个 mutex（互斥量）很可能是恶意程序为了防止它本身多个复制间的双重感染而设置的。另一方面，如果用 google 搜索这个字符串，我们可以找到许多与之相关的信息，这样我们基本上完成了人力反病毒程序这个角色的操作。不过（既然本章是个恶意软件分析攻略），我们假设什么也没找到并继续分析。

8

更多资源请点击访问稀.酷客(www.ckook.com)

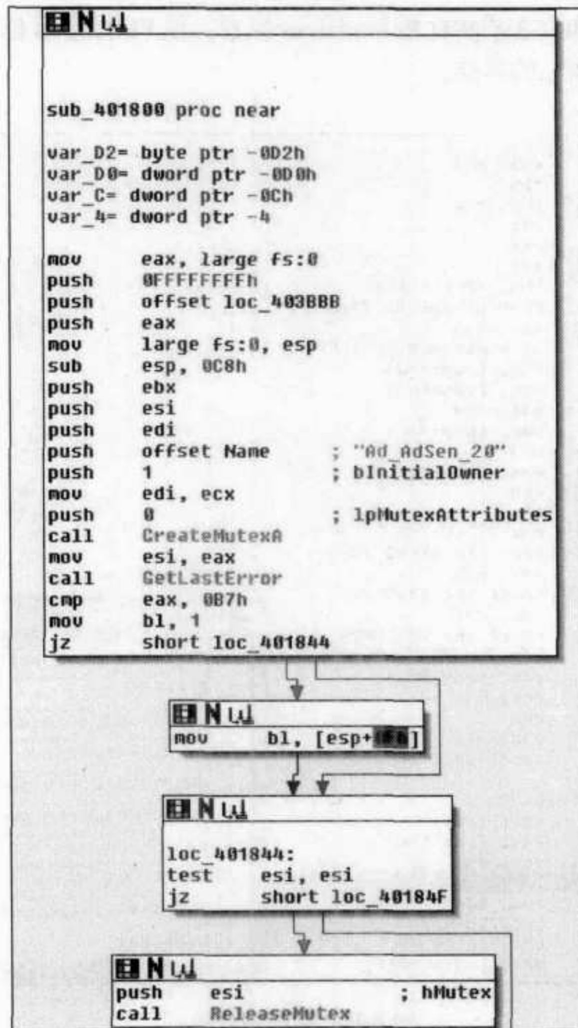


图　8-13

我们看到在调用 CreateMutexA()后，程序把返回值复制到了 ESI 寄存器并调用了 GetLastError()[①]。这个返回值会和 0xB7（即十进制的 183）进行比较，0xB7 代表的是 ERROR_ALREADY_EXISTS。很显然，这是恶意软件防止自己反复进行自我复制时我们希望看到的情况。从这里开始，程序把 1 复制到 EBX 寄存器的低 8 位，然后根据这个返回值进行跳转。如果条件为真（也就是 CreateMutexA()调用失败），我们将会跳到 loc_401844，否则我们将会把栈上的一个值复制到 EBX 的低 8 位。接着，我们测试 ESI 寄存器是否非零（CreateMutexA()出错时会返回 NULL）。如果为非零将跳到 loc_40184F，否则我们将释放 mutex 的所有权。继续看这个函数，

① 原文如此，译者推测是 RtlGetLastWin32Error()函数，后同。

更多资源请点击访问稀.酷客(www.ckook.com)

我们会发现图 8-14 所示的代码。

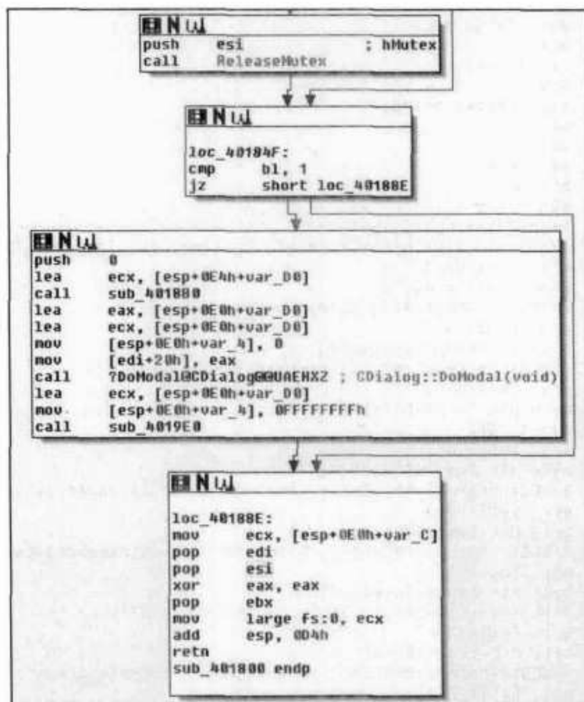


图　8-14

通读这段代码，我们看到可能会调用 `ReleaseMutex()`，接下来，位于 `loc_40184F` 的代码检查 EBX 寄存器的低 8 位中存放的是不是 1。如果你还记得的话，如果调用 `CreateMutexA()` 失败的话，就会往 EBX 中放入 1，因此，`loc_40184F` 处的代码只是检查 mutex 对象是否创建成功或所有权是否给予调用线程了。如果对 `CreateMutexA()` 的调用失败，执行控制将传递给 `loc_40188E`，否则将继续。假设这里的调用并没有失败，那么就会把 0 作为 `sub_4018B0()` 的参数压入栈。我们还注意到，在调用 `sub_4018B0()` 前，保存在 `var_D0` 里的偏移量的地址被放入 ECX 中，在这个值被用到之前，又用不同的偏移量覆盖了这个值。这里有两种解释，第一是使用了 `fastcall` 调用约定，第二是我们调用的是类里的方法。因为第一个参数被压入栈，而不是放在 ECX 中（第二个参数放在 EDX 中），我们可以假设正在调用一个实例方法，并假设 ECX 中保存的是 `this` 指针。现在我们看一下 `sub_4018B0()`，看它做了什么，并验证我们关于 ECX 寄存器的猜测，如图 8-15 所示。[①]

① `sub_4018B0()` 实际上就是 `XxxDlg` 类的构造函数，其中 Xxx 是项目的名称，关于这一点，熟悉 MFC 的朋友可以从下面马上就要调用这个类的 `DoModal()` 方法上看出来。这两条 “`lea ecx, [esp+0E4h+var_D0]`” 指令就是在传递同样的 `this` 指针啊！由于变量 `var_D0` 之前并没有初始化过，接着又会被当成 `DoModal()` 方法的 `this` 指针使用，所以即使不熟悉 MFC 也能猜出 `sub_4018B0()` 这个函数是某个类的构造函数。

更多资源请点击访问稀.酷客(www.ckook.com)

```
sub_4018B0:
push    0FFFFFFFFh
push    offset loc_403C3C
mov     eax, large fs:0
push    eax
mov     large fs:0, esp
push    ecx
mov     eax, [esp+10h+arg_0]
push    esi
push    edi
mov     esi, ecx
push    eax
push    66h
mov     [esp+20h+var_10], esi
call    ??0CDialog@@QAE@IPAVCWnd@@@Z; CDialog::CDialog(uint,CWnd *)
lea     ecx, [esi+60h]
mov     [esp+18h+var_4], 0
call    ??0CString@@QAE@XZ; CString::CString(void)
lea     ecx, [esi+64h]
mov     byte ptr [esp+18h+var_4], 1
call    ??0CString@@QAE@XZ; CString::CString(void)
lea     ecx, [esi+68h]
mov     byte ptr [esp+18h+var_4], 2
call    sub_401000
lea     ecx, [esi+78h]
mov     byte ptr [esp+18h+var_4], 3
call    ??0CStringArray@@QAE@XZ; CStringArray::CStringArray(void)
lea     ecx, [esi+90h]
mov     byte ptr [esp+18h+var_4], 4
call    ??0CStringArray@@QAE@XZ; CStringArray::CStringArray(void)
lea     ecx, [esi+0A4h]
mov     byte ptr [esp+18h+var_4], 5
call    ??0CString@@QAE@XZ; CString::CString(void)
lea     ecx, [esi+0A8h]
mov     byte ptr [esp+18h+var_4], 6
call    ??0CStringArray@@QAE@XZ; CStringArray::CStringArray(void)
lea     edi, [esi+0BCh]
mov     byte ptr [esp+18h+var_4], 7
mov     ecx, edi
call    ??0CString@@QAE@XZ; CString::CString(void)
push    offset Default
mov     ecx, edi
mov     byte ptr [esp+1Ch+var_4], 8
mov     dword ptr [esi], offset off_404460
mov     dword ptr [esi+8Ch], 0
call    ??4CString@@QAEABV0@PBD@Z; CString::operator=(char const *)
mov     ecx, esi
call    sub_4035A0
push    0               ; pvReserved
call    CoInitialize
mov     ecx, [esp+18h+var_C]
mov     eax, esi
pop     edi
pop     esi
mov     large fs:0, ecx
add     esp, 10h
retn    4
```

图 8-15　奇怪的 ECX 寄存器

当我们进入 sub_4018B0()，可以看到这些代码相当标准（我们注意到这里使用了 fastcall 调用约定）。在过程的 prologue 之后，还包括了设置 SEH 记录的指令。在这之后，我们看到 ECX 寄存器被保存起来了，之前压入栈的参数被复制到 EAX 寄存器里。ECX 寄存器的副本被保存在 ESI 寄存器里，this 指针被存入 var_10 里，最后，程序调用 CDialog 类的构造函数。传递给构造函数的两个参数表明我们并不需要特定的父窗口，或者说这个父窗口应当属于主应用程序，

更多资源请点击访问稀.酷客(www.ckook.com)

而这个整数则指定了要使用的 Dialog 模板。这里我们正准备设置一个对话框。老实说，这对一个恶意软件来说有些奇怪。我们将继续关注它，直到水落石出。继续看这个例程，我们看到当前 this 指针的偏移量经过计算后被保存到 ECX 里，然后 var_14 被初始化成 0，接着调用了 CString:: CString(void)的构造函数。如果我们单步步入这个构造函数，将会看到如图 8-16 所示的界面，这有助于我们理解程序接下来的所作所为。

```
mfc42.dll:699FCB77 ; ------------------------------------
mfc42.dll:699FCB77
mfc42.dll:699FCB77 mfc42_540:

mfc42.dll:699FCB79 push    esi
mfc42.dll:699FCB7A mov     esi, ecx
mfc42.dll:699FCB7C call    near ptr unk_699FCAEF
mfc42.dll:699FCB81 mov     eax, esi
mfc42.dll:699FCB83 pop     esi
mfc42.dll:699FCB84 retn
mfc42.dll:699FCB84 ; ------------------------------------
mfc42.dll:699FCB85 db  90h ; É
mfc42.dll:699FCB86 db  90h ; É
mfc42.dll:699FCB87 db  90h ; É
mfc42.dll:699FCB88 db  90h ; É

EAX 0012FE10  Stack[00000740]:0012FE10     CF 0
EBX FFFFFF00                               PF 1
ECX 0012FE70  Stack[00000740]:0012FE70     AF 0
EDX 7703OF34  ntdll.dll:ntdll_KiFastSy     ZF 1
ESI 0012FE10  Stack[00000740]:0012FE10     SF 0
EDI 00406360  UPX0:dword_406360            TF 0
EBP 0012FEF0  Stack[00000740]:saved_fp     IF 1
ESP 0012FDDC  Stack[00000740]:0012FDDC     DF 0
EIP 699FCB77  mfc42.dll:mfc42_540          OF 0
EFL 00000246
```

图 8-16 CString 类的单步构造函数

由此我们知道，在这个上下文里 ECX 是父类方法[①]的 this 指针的偏移量。我们可以清楚地看到，它的副本被放入 ESI，接着调用了一个近偏移量。不过比较有意思的是，我们看到返回值被调用者传递给方法的指针覆盖了。因为知道这是构造函数，我们基本上可以把它视为诸如 new 之类的东西。不过更有可能的是，我们的调用者本身就是构造函数，如果你跟随调用者的帧里的 CDialog::CDialog()调用，可能就会明白几分。你会发现 sub_4018B0()里的递归调用。如果你注意的话还会看到另一个有趣的方面，在这个方法里明显有一些实例变量被实例化了，具体来说就是 4 个 CString 实例和 3 个 CStringArray 实例。不过，因为在进入时 this 指针并没有被修改，所以 CDialog 的实例看上去有些奇怪。这至少暗示我们的类是 CDialog 类的抽象/接口，等等。

看完这个例程，我们发现了一些奇怪的地方。例如，this 指针被存入 var_10，但之后这个变量就再也没有被用到，var_4 的值在每个新对象创建后就会递增，但也没有再被使用，等等。希望你能了解这些由编译器留下的不同类型的线索。此后，我们看到 this 指针所指的数据被修改了——被修改成 off_404460。比较有趣的是指向最后一个 CString 对象的指针被 NULL 指针覆盖了，留下一个没有引用的对象。尽管如此，我们还是要等到析构函数后才能做逆向判断。为 CString 对象调用分配操作等也很容易明白，如果你单步步入，就会发现其实很简单——只是保证字符串空中止，接下来，我们看到 ESI 中的 this 指针副本被复制到了 ECX，并调用了 sub_4035A0()，意味着它是这个类的另一个成员方法。

① 原文如此。这个“父类方法”的讲法有问题，这个 CString 类并不是用户自己写的这个类的子类，而是这个类中的一个成员，这个类应该是继承自 CDialog 的，这一点我前面的译注里已经讲了。

更多资源请点击访问稀.酷客(www.ckook.com)

步入 sub_4035A0()，我们看到的代码如图 8-17 所示，这明显是字符串初始化例程，此外还隐约看出程序企图混淆代码，至少在这里看起来是这样的。这个推测主要是基于调用操作符+=的数量，以及所有的参数都是字符串常量。

```
sub_4035A0 proc near
push    esi
push    edi
mov     edi, ecx
call    sub_403630
lea     esi, [edi+60h]
push    offset dword_40626C
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_406264
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_406260
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_406258
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
lea     esi, [edi+64h]
push    offset dword_406254
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_406250
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_406244
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_40623C
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_406238
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_406230
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
pop     edi
pop     esi
retn
sub_4035A0 endp
```

图 8-17

进入 sub_3035A0()，我们看到一个无框架的（frameless）方法。也就是说，我们没有看到严格的 prologue 或 epilogue 过程，就几乎立即调用了 sub_403630()，可以看到这样生成的方法与 sub_3035A0()的很类似，参见图 8-18。

如果你真正感兴趣，可以单步步入这两个方法分析它们，不过它们所做的只是分别用值“http://www.alxup.com/adsnt/AdsNT.ini”和“http://www.alxup.com/adsnt/AdsNT.exe”初始化 CString。

更多资源请点击访问稀.酷客(www.ckook.com)

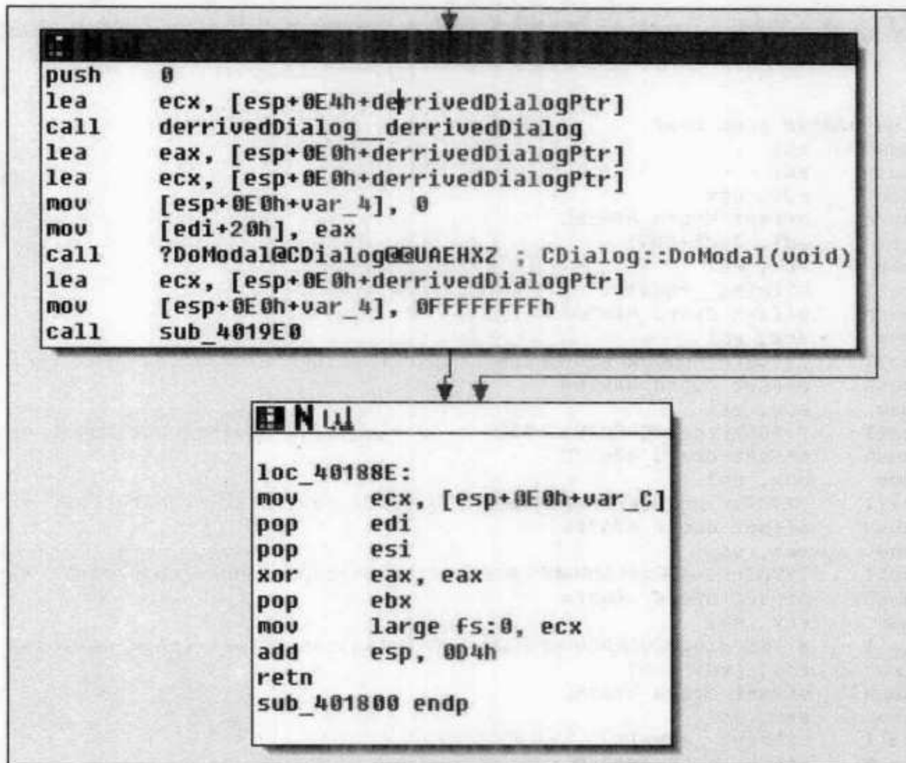
```
sub_403630 proc near
push    esi
push    edi
mov     edi, ecx
push    offset dword_40628C
lea     esi, [edi+60h]
mov     ecx, esi
call    CString__equals
push    offset dword_406284
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_406280
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_40627C
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_406278
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
push    offset dword_406274
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
lea     esi, [edi+64h]
push    offset dword_40628C
mov     ecx, esi
call    CString__equals
push    offset byte_406270
mov     ecx, esi
call    ??YCString@@QAEABV0@PBD@Z ; CString::operator+=(char const *)
pop     edi
pop     esi
retn
sub_403630 endp
```

图 8-18

最后，返回构造方法时我们看到了对 CoInitialize()的调用，而且发现 this 指针被返回给 EAX 里的调用者。据我们所知，我们将调用刚分析过的 derrivedDialog:: derivedDialog()方法。返回调用例程——WinMain()函数，我们会看到如图 8-19 所示的代码。

在对象（我们刚分析过的）创建之后，我们看到指向这个对象的指针的一个副本被存入 EAX 和 ECX，且 var_4 被置为零，然后生成了 this 寄存器的一个拷贝，之后调用了 CDialog::DoModal()。我们走到这一步主要是想通过分析代码学一点关于 MFC 的知识。不过，出现的代码给我们带来了点小问题，因为我们几乎已处在代码的结尾处，但看起来从这里开始的代码都以 MFC 为中心。MSDN 告诉我们 DoModal()将创建我们刚设置的对话框。因此，我们可以预计到会有一系列处理这个对话框的回调函数。让我们重新启动调试器，再次步入这个函数。

更多资源请点击访问稀.酷客(www.ckook.com)


```
push    0
lea     ecx, [esp+0E4h+derrivedDialogPtr]
call    derrivedDialog__derrivedDialog
lea     eax, [esp+0E0h+derrivedDialogPtr]
lea     ecx, [esp+0E0h+derrivedDialogPtr]
mov     [esp+0E0h+var_4], 0
mov     [edi+20h], eax
call    ?DoModal@CDialog@@UAEHXZ ; CDialog::DoModal(void)
lea     ecx, [esp+0E0h+derrivedDialogPtr]
mov     [esp+0E0h+var_4], 0FFFFFFFFh
call    sub_4019E0

loc_40188E:
mov     ecx, [esp+0E0h+var_C]
pop     edi
pop     esi
xor     eax, eax
pop     ebx
mov     large fs:0, ecx
add     esp, 0D4h
retn
sub_401800 endp
```

图　8-19

在单步多条指令后，我们最终到达一个在 MFC 里的调用，它实际上是通过调用 CreateDialogIndirectParamA() 创建窗口。这个 API 调用向合适的对话框过程发送一个 WM_INITDIALOG 消息，因此，它充当了应用程序 GUI 方面的进入点，这正是对这个对话框类型的初始化回调。

如图 8-20 所示，窗口初始化的回调方法是 sub_401AF0()，它是另一个无框架的方法。我们看到的过程初始化中没有多少值得我们关注的。我们应当单步单过 CDialog::OnInitDialog()，以确保没有调用衍生的过程，还要观察标签为 0x00401B08、0x00401B28 和 0x00401B4D 的代码块里对 EDX 的间接调用。不过，一眼望去基本上都是 MFC 代码。

一直执行到如图 8-21 所示的代码。对 CDialog::OnInitDialog()的调用没有执行任何用户定义的代码，我们在随后的屏幕快照里可以清楚地看到这个函数指针。需要着重注意的是，因为我们步过这个代码的方式以及所使用的方法学的模糊性，我在这个二进制文件里定义的所有函数的进入点上及其附近都设了断点。因此，至少从理论上讲如果我们错过什么东西，我们将注意到它并有机会倒回来弄明白我们错过了什么，以及我们是怎样错过的，等等。

看这个函数接下来的代码段，我们将发现如图 8-22 所示的代码，这部分代码有点意思，它可以进一步揭示这个恶意软件的用途。

更多资源请点击访问稀.酷客(www.ckook.com)

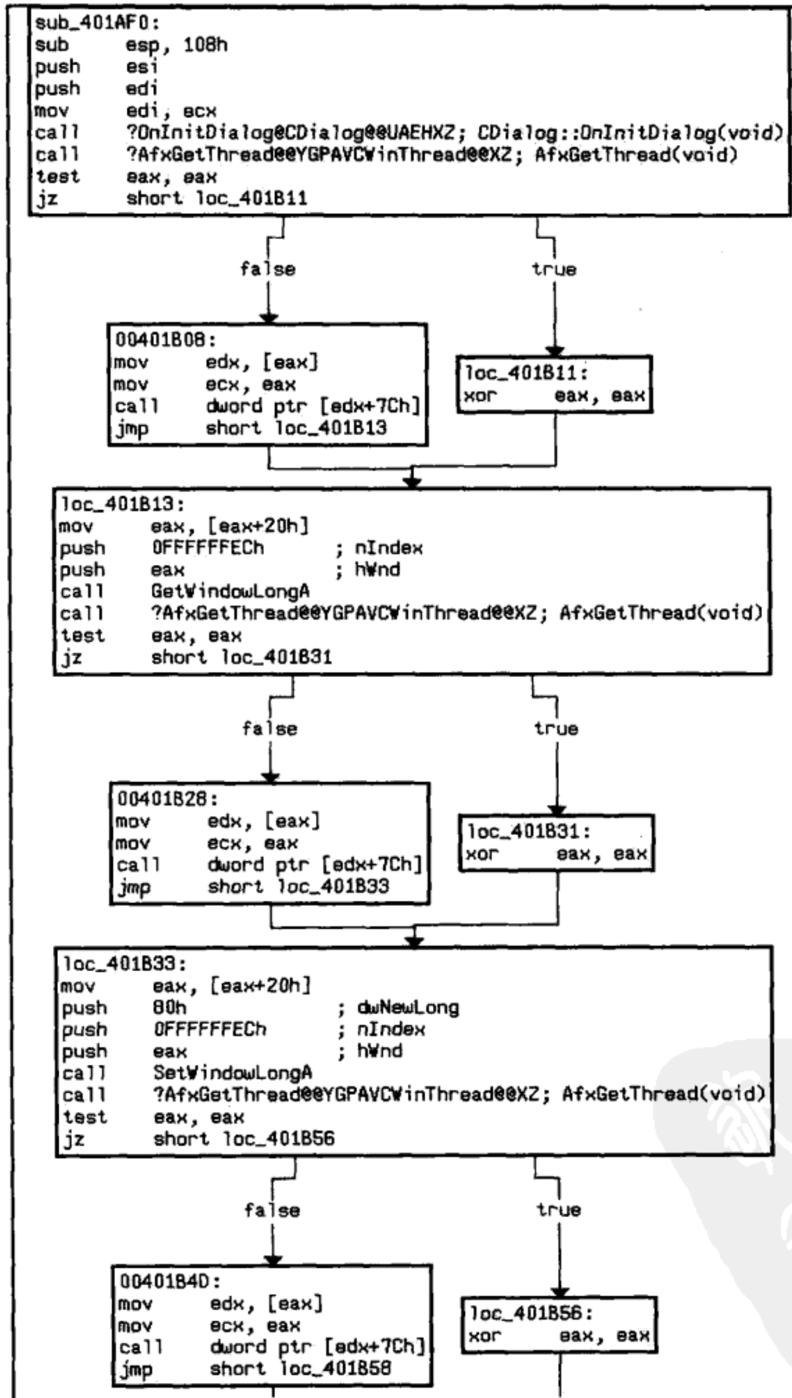


图 8-20 初始化回调函数

更多资源请点击访问稀酷客(www.ckook.com)

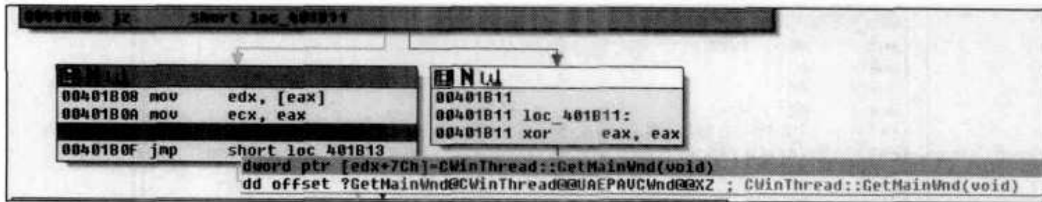


图 8-21

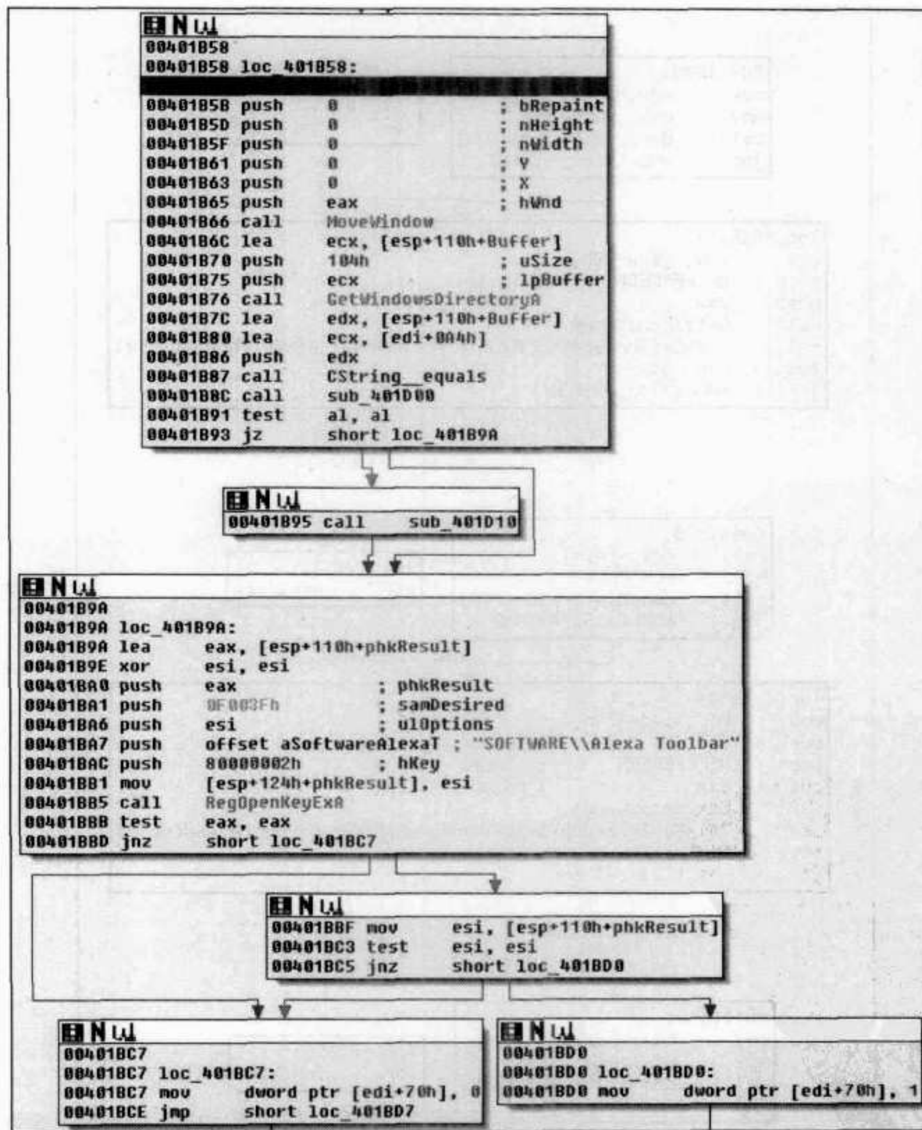


图 8-22

更多资源请点击访问稀.酷客(www.ckook.com)

我们在这里发现了期望的内容。初始化窗口完成的第一件事情就是调用 MoveWindow()，它将重新调整窗口的大小和/或位置（图 8-23）。在这个例子里，它们把窗口的大小及位置都设为 0。接下来，我们通过调用 GetWindowsDirectoryA()获得 windows 目录的路径，并把返回值赋给位于 EDI+0x0A4 的 CString，紧接着调用 sub_401D00()。

```
00401D00
00401D00
00401D00
00401D00 sub_401D00 proc near
[illegible]
00401D00                              ; and information about the operating system platform
00401D06 cmp     eax, 80000000h
00401D0B setnb   al
00401D0E retn
00401D0E sub_401D00 endp
00401D0E
```

图 8-23

就像我们看到的，sub_401D00()是比较简单的过程，它只检查当前操作系统的版本是否低于 Windows NT，然后基于结果修改 AL 寄存器并将结果返回调用者。从那开始，我们测试 EAX 的低 8 位，根据 Windows 的版本决定调用 sub_401D10()还是转到 loc_401B9A()。如果操作系统的版本太低，那么 sub_401D10()所做的就是调用 RegisterServiceProcess()，使应用程序不会在用户退出系统时退出。接着，我们看到它打开“Software\Alexa Toolbar”注册表键。如果调用成功的话，它将把位于 EDI+0x70 的变量设为 1，否则设为 0（图 8-24）。

```
00401BD7
00401BD7 loc_401BD7:                  ; time_t *
00401BD7 push    0
00401BD9 call    time
00401BDF push    eax                  ; unsigned int
00401BE0 call    srand
00401BE6 mov     ecx, [edi+20h]
00401BE9 add     esp, 8
00401BEC push    0                    ; lpTimerFunc
00401BEE push    2BF20h               ; uElapse
00401BF3 push    1                    ; nIDEvent
00401BF5 push    ecx                  ; hWnd
00401BF6 call    SetTimer
00401BFC test    esi, esi
00401BFE mov     uIDEvent, eax
00401C03 jz      short loc_401C0C

00401C05 push    esi                  ; hKey
00401C06 call    RegCloseKey

00401C0C
00401C0C loc_401C0C:
00401C0C pop     edi
00401C0D mov     eax, 1
00401C12 pop     esi
00401C13 add     esp, 108h
00401C19 retn
00401C19 sub_401AF0 endp
00401C19
```

图 8-24

接下来，我们看到把当前时间作为 PRNG（Pseudo-Random Number Generator，伪随机数生成器）的种子，然后调用 SetTimer()API。这里要注意的是，一个 NULL 指针被作为参数传递给回调函数，这意味着当定时器达到 0x2BF20 毫秒（大致为 3 分钟），将向窗口发送 WM_TIMER 消息。再接下来，关闭注册表键值，例程返回。随后，我们将单步更多指令，看程序会在哪里停下

更多资源请点击访问稀.酷客(www.ckook.com)

来，不过我们已经大概知道 3 分钟后，我们就会停在 WM_TIMER 处理程序的过程上（参见 8-25）。

```
mfc42.dll:69C5C860 loc_69C5C860:                      ; CODE XREF: mfc42.dll:69C5C8CB↓j
mfc42.dll:69C5C860 push    0
mfc42.dll:69C5C862 push    0
mfc42.dll:69C5C864 push    0
mfc42.dll:69C5C866 push    0
mfc42.dll:69C5C868 push    ebx
mfc42.dll:69C5C869 call    ds:PeekMessage
mfc42.dll:69C5C86F test    eax, eax
mfc42.dll:69C5C871 jnz     short loc_69C5C8D6
mfc42.dll:69C5C873 cmp     [ebp-4], eax
mfc42.dll:69C5C876 jz      short loc_69C5C88F
mfc42.dll:69C5C878 push    1
mfc42.dll:69C5C87A mov     ecx, esi
mfc42.dll:69C5C87C call    near ptr mfc42_6215
mfc42.dll:69C5C881 mov     ecx, esi
mfc42.dll:69C5C883 call    near ptr unk_69C5C206
mfc42.dll:69C5C888 mov     dword ptr [ebp-4], 0
mfc42.dll:69C5C88F
mfc42.dll:69C5C88F loc_69C5C88F:                      ; CODE XREF: mfc42.dll:69C5C876↑j
mfc42.dll:69C5C88F test    byte ptr [ebp+8], 1
mfc42.dll:69C5C893 jnz     short loc_69C5C8B1
mfc42.dll:69C5C895 mov     eax, [ebp-0Ch]
mfc42.dll:69C5C898 test    eax, eax
mfc42.dll:69C5C89A jz      short loc_69C5C8B1
mfc42.dll:69C5C89C test    edi, edi
mfc42.dll:69C5C89E jnz     short loc_69C5C8B1
mfc42.dll:69C5C8A0 mov     ecx, [esi+20h]
mfc42.dll:69C5C8A3 push    ecx
mfc42.dll:69C5C8A4 push    edi
mfc42.dll:69C5C8A5 push    121h
mfc42.dll:69C5C8AA push    eax
mfc42.dll:69C5C8AB call    ds:SendMessage
mfc42.dll:69C5C8B1
mfc42.dll:69C5C8B1 loc_69C5C8B1:                      ; CODE XREF: mfc42.dll:69C5C893↑j
mfc42.dll:69C5C8B1                                    ; mfc42.dll:69C5C89A↑j ...
mfc42.dll:69C5C8B1 test    byte ptr [ebp+8], 2
mfc42.dll:69C5C8B5 jnz     short loc_69C5C8CD
mfc42.dll:69C5C8B7 push    edi
mfc42.dll:69C5C8B8 push    0
mfc42.dll:69C5C8BA push    36Ah
mfc42.dll:69C5C8BF mov     ecx, esi
mfc42.dll:69C5C8C1 call    near ptr unk_69C3DA04
mfc42.dll:69C5C8C6 add     edi, 1
mfc42.dll:69C5C8C9 test    eax, eax
mfc42.dll:69C5C8CB jnz     short loc_69C5C860
mfc42.dll:69C5C8CD
mfc42.dll:69C5C8CD loc_69C5C8CD:                      ; CODE XREF: mfc42.dll:69C5C8B5↑j
mfc42.dll:69C5C8CD mov     dword ptr [ebp-8], 0
mfc42.dll:69C5C8D4 lea     ebx, [ebx]
mfc42.dll:69C5C8D6
mfc42.dll:69C5C8D6 loc_69C5C8D6:                      ; CODE XREF: mfc42.dll:69C5C85E↑j
mfc42.dll:69C5C8D6                                    ; mfc42.dll:69C5C871↑j ...

mfc42.dll:69C5C8DB mov     edx, [eax]
mfc42.dll:69C5C8DD mov     ecx, eax
mfc42.dll:69C5C8DF mov     eax, [edx+64h]
mfc42.dll:69C5C8E2 call    eax
mfc42.dll:69C5C8E4 test    eax, eax
mfc42.dll:69C5C8E6 jz      short loc_69C5C958
mfc42.dll:69C5C8E8 cmp     dword ptr [ebp-4], 0
```

图 8-25

当我们单步前面的对话框创建代码后，就进入 MFC DLL 里的一段代码了，它如同消息队列循环，处理窗口消息。在调用 PeekMessage()之后，如果队列里有消息，那么控制权将会转到 loc_69C5C8D6，之后会执行 callEAX 指令（也就是调用 CWinThread::PumpMessage()）。最后，在执行完冗长的循环后，我们发现了下一个回调函数——sub_401FC0()，它的定义如图 8-26 所示。

更多资源请点击访问稀.酷客(www.ckook.com)

```
00401FC0
00401FC0
00401FC0
00401FC0 ; int __fastcall sub_401FC0(char *)
00401FC0 sub_401FC0 proc near
00401FC0
00401FC0 var_4= dword ptr -4
00401FC0

00401FC1 push    esi
00401FC2 mov     esi, ecx
00401FC4 push    edi
00401FC5 push    ecx             ; char *
00401FC6 lea     edi, [esi+0A4h]
00401FCC mov     eax, esp
00401FCE mov     [esp+10h+var_4], esp
00401FD2 push    offset aIndex_htm ; "\\index.htm"
00401FD7 push    edi
00401FD8 push    eax
00401FD9 call    ??H@YG?AVCString@@ABV0@PBD@Z ; operator+(CString const &,char const *)
00401FDE lea     ecx, [esi+60h]
00401FE1 call    sub_401F90
00401FE6 push    eax             ; lpszUrl
00401FE7 mov     ecx, esi
00401FE9 call    sub_402B50
00401FEE test    eax, eax
00401FF0 jnz     short loc_40201D
```

```
00401FF2 push    offset aIndex_htm ; "\\index.htm"
00401FF7 lea     ecx, [esp+10h+var_4]
00401FFB push    edi
00401FFC push    ecx
00401FFD call    ??H@YG?AVCString@@ABV0@PBD@Z ; operator+(CString const &,char const *)
00402002 mov     edx, [eax]
00402004 push    edx             ; lpFileName
00402005 call    DeleteFileA
0040200B lea     ecx, [esp+0Ch+var_4]
0040200F call    ??1CString@@QAE@XZ ; CString::~CString(void)
00402014 mov     eax, 1
00402019 pop     edi
0040201A pop     esi
0040201B pop     ecx
0040201C retn
```

```
0040201D
0040201D loc_40201D:
0040201D pop     edi
0040201E xor     eax, eax
00402020 pop     esi
00402021 pop     ecx
00402022 retn
00402022 sub_401FC0 endp
00402022
```

图 8-26

进入函数时，我们看到一些感兴趣的事情。首先完成的事情中其一就是把字符串`\index.htm`与 Windows 目录连在一起，这样就得到了字符串`C:\Windows\index.htm`，接着是对`sub_401F90()`的调用。这只是解除了保存在`ECX`里的指针的引用，并把这个指针复制到`EAX`。然后把指向字符串`http://www.alxup.com/adsnt/AdsNT.ini`的指针与新连接成的字符串作为参数传递给例程`sub_402B50()`，`sub_402B50()`例程的代码如图 8-27 所示。

当我们步入这个新例程，明显看出已经接触到这个恶意软件的核心了。我们现在可以看到它怎样与外部世界联系。在过程 prologue 之后，我们看到对`InternetOpenA()`的调用。在这个例子里，我们唯一感兴趣的参数是将要使用的`User-Agent`，这样的话，我们在网络上就可以把它作为静态的特征加以检测了。比如说，假设你想生成一条 IDS 规则，就可以搜索与内容类似的数据包（可以忽略因 fragmentation/request splitting/weirdo tab placement/urgent data/等原因产生的差异）：

```
GET /adsnt/AdsNT.ini [...]
[...]
User-Agent: adsntB/1.6
[...]
```


更多资源请点击访问稀.酷客(www.ckook.com)

```
00402B50
00402B50
00402B50
00402B50 ; int __stdcall sub_402B50(LPCSTR lpszUrl,char *)
00402B50 sub_402B50 proc near
00402B50
00402B50 dwNumberOfBytesRead= dword ptr -41Ch
00402B50 Buffer= dword ptr -418h
00402B50 var_414= dword ptr -414h
00402B50 dwBufferLength= dword ptr -410h
00402B50 var_40C= dword ptr -40Ch
00402B50 var_C= dword ptr -0Ch
00402B50 var_4= dword ptr -4
00402B50 lpszUrl= dword ptr  4
00402B50 arg_4= dword ptr  8
00402B50

00402B52 push    offset loc_403E18
00402B57 mov     eax, large fs:0
00402B5D push    eax
00402B5E mov     large fs:0, esp
00402B65 sub     esp, 410h
00402B6B push    ebx
00402B6C push    ebp
00402B6D push    edi
00402B6E push    0                  ; dwFlags
00402B70 push    0                  ; lpszProxyBypass
00402B72 push    0                  ; lpszProxy
00402B74 push    0                  ; dwAccessType
00402B76 push    offset szAgent     ; "adsntB/1.6"
00402B7B mov     ebp, 1
00402B80 call    InternetOpenA
00402B86 mov     ebx, eax
00402B88 test    ebx, ebx
00402B8A mov     [esp+428h+var_414], ebx
00402B8E jz      loc_402C62
```

```
00402B94 mov     eax, [esp+428h+lpszUrl]
00402B98 push    0                  ; dwContext
00402B9D push    84000100h          ; dwFlags
00402BA2 push    0                  ; dwHeadersLength
00402BA4 push    0                  ; lpszHeaders
00402BA6 push    eax                ; lpszUrl
00402BA7 push    ebx                ; hInternet
00402BA8 call    InternetOpenUrlA
00402BAE lea     ecx, [esp+428h+dwBufferLength]
00402BB2 push    0                  ; lpdwIndex
00402BB4 lea     edx, [esp+42Ch+Buffer]
00402BB8 push    ecx                ; lpdwBufferLength
00402BB9 mov     edi, eax
00402BBB push    edx                ; lpBuffer
00402BBC push    20000013h          ; dwInfoLevel
00402BC1 push    edi                ; hRequest
00402BC2 mov     [esp+43Ch+dwBufferLength], 20h
00402BCA call    HttpQueryInfoA
00402BD0 cmp     [esp+428h+Buffer], 190h
00402BD8 jnb     loc_402C62
```

图 8-27

InternetOpenA() API 调用会返回一个句柄，我们可以看到它被复制到 EBX 里了。然而，它最终会被复制到 var_414 中（重命名为 inetHandle）。我们还要注意的是，程序会检查这个返回值是否为 NULL。接下来，就会通过调用 InternetOpenUrlA()检索作为参数传入的

更多资源请点击访问稀.酷客(www.ckook.com)

URL（也就是"http://www.alxup.com/adsnt/AdsNT.ini"）。检索完成后，将调用 HttpQueryInfoA() 检查远程网络服务器返回的状态码。然后检查这个值是否为 0x190（也就是十进制的 400），如果状态码不小于 400 话，程序将会跳转。现在有这种可能（我也不是很肯定，因为我需要进一步研究 InternetOpenURL()，但现在时间不允许），如果返回的状态码的长度超过 0x20，我们就会以一个未初始化的变量结束，而这将导致一个 bug。不过在这个例子里不太可能会这样，因为传递给 HttpQueryInfoA()的标志指定了我们想要返回的状态码是整数（0x20000013 等价于 HTTP_QUERY_FLAG_NUMBER | HTTP_QUERY_ STATUS_CODE）。接着往下看这个函数的代码，如图 8-28 所示。

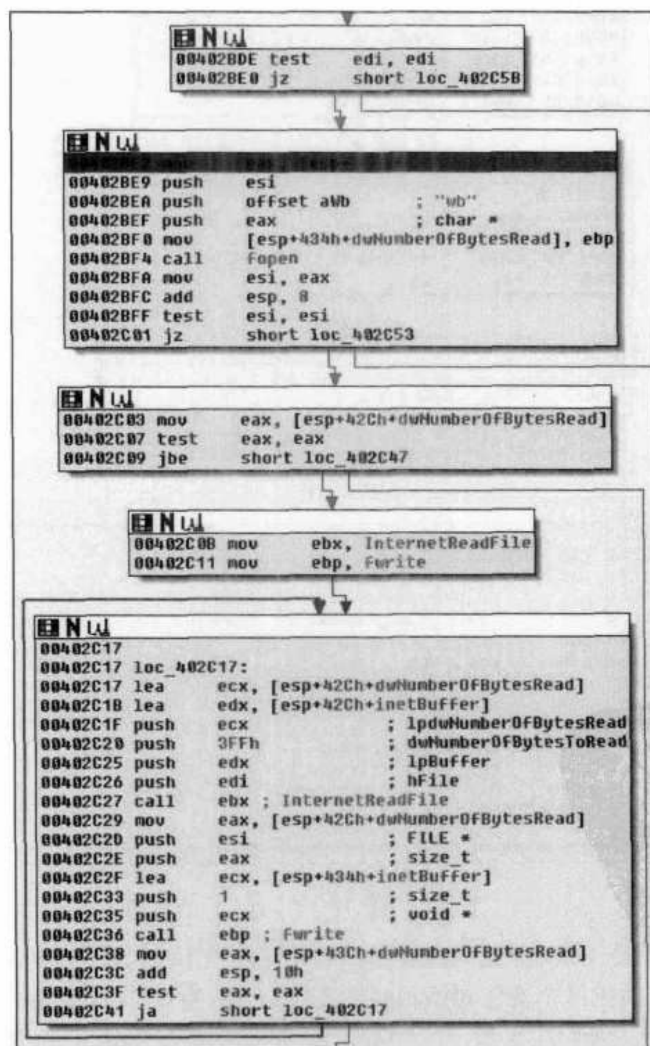


图 8-28

更多资源请点击访问稀.酷客(www.ckook.com)

往下进行，我们看到程序在测试 InternetOpenA()返回的句柄的副本，如果它为 NULL 则跳转，否则往下继续调用 fopen()。在这里，我们看到函数的第二参数是 wb，表示以二进制写操作方式打开文件。接下来检查字节数，如果一切 OK，我们将在 loc_402c17 处结束。这里我们会看到一个简单的循环，每次通过调用 InternetReadFile()从网络上读入 0x3FF 个字节的数据，然后通过调用 fwrite()把数据写到磁盘上。从这个循环开始，我们进入下一段代码（如图 8-29 所示）。

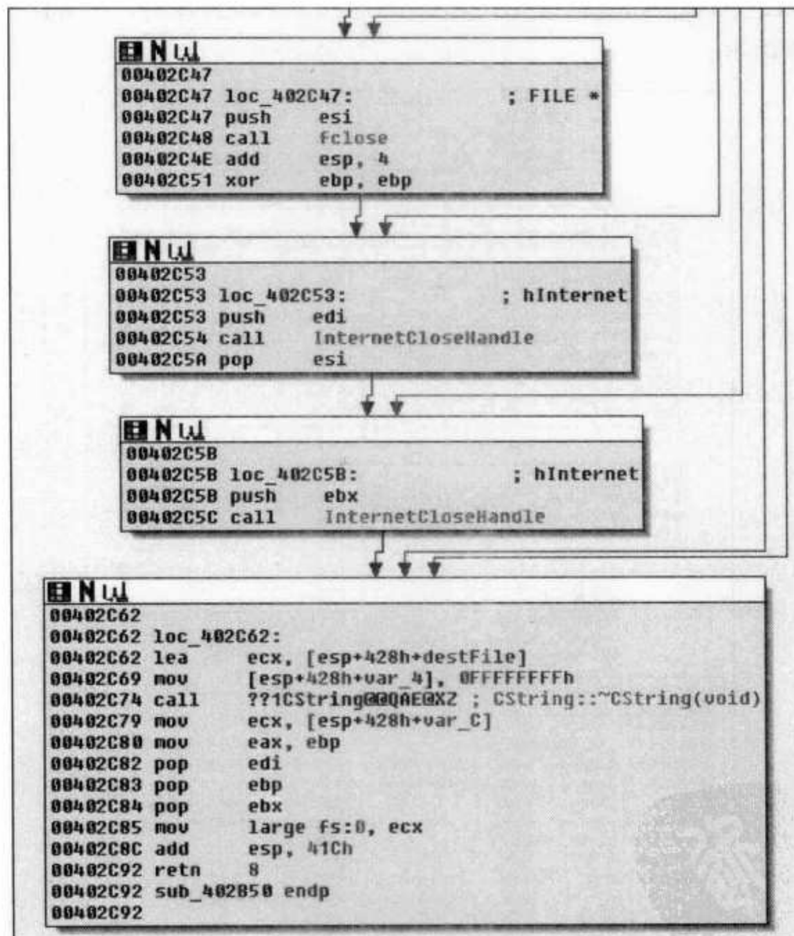


图 8-29

来到这个函数的最后一部分，这些内容和我们预计的差不多。我们已从网上读取文件的内容并把它写入磁盘，现在关闭这个文件句柄和网络连接，然后为 CString 调用析构函数，并把 EBP 里的值作为返回值返回。我们注意到，0x00402C51 的代码会把 EBP 中的值改为 0，所以这个例程中的一个成功的 I/O 周斯返回值 0，失败时则非 0。因此可以说，这个函数需要两个参数：一

更多资源请点击访问稀.酷客(www.ckook.com)

个 URL 和一个路径。这个例程检索 URL 所指的文件，并把它保存到指定的路径。我们把这个函数重命名为 downloadToFile()。需要提一下，如果你的系统是 Vista，文件将会保存在 C:\Users\[username]\AppData\Local\VirtualStore\路径下，而不是 C:\Windows。让我们看看下载的 INI 文件内容是什么。

```
[AdsNT]
Version=100
AdNum=9
[AdsNTURL]
leftpos1=-1000
toppos1=-1000
width1=1
height1=1
url1=http://www.deepdo.com/union/3721/yad.htm
objurl1=http://zzz.yy.xom
weight1=10
showIEWindow1=0
showStyle1=0
group1=0
leftpos2=-1000
toppos2=-1000
width2=1
height2=1
url2=http://talent.deepdo.com
objurl2=http://xx.you.com
weight2=100
showIEWindow2=0
showStyle2=0
group2=0
leftpos3=-1000
toppos3=-1000
width3=1
height3=1
url3=http://www.deepdo.com/union/3721/yad.htm
objurl3=http://xxx.yyy.com/
weight3=10
showIEWindow3=0
showStyle3=0
group3=0
leftpos4=-1000
toppos4=-1000
width4=1
height4=1
url4=http://www.92site.cn/search/135go.jsp
```

8

更多资源请点击访问稀.酷客(www.ckook.com)

```
objurl4=http://xxx.com
weight4=15
showIEWindow4=0
showStyle4=0
group4=0
leftpos5=-1000
toppos5=-1000
width5=1
height5=1
url5=http://www.5isou.cn/
objurl5=http://xxx.xxx.com/
weight5=100
showIEWindow5=0
showStyle5=0
group5=0
leftpos6=-1000
toppos6=-1000
width6=1
height6=1
url6=http://www.deepdo.com/site.htm
objurl6=http://xxx.xxx.com/
weight6=150
showIEWindow6=0
showStyle6=0
group6=0
leftpos7=-1000
toppos7=-1000
width7=1
height7=1
url7=http://www.5isou.cn/calendar/index.htm
objurl7=http://xxx.xxx.com/
weight7=150
showIEWindow7=0
showStyle7=0
group7=0
leftpos8=-1000
toppos8=-1000
width8=1
height8=1
url8=http://www.92site.cn/search/135go.jsp
objurl8=http://xxx.xxx.com/
weight8=150
showIEWindow8=0
showStyle8=0
group8=0
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
leftpos9=-1000
toppos9=-1000
width9=1
height9=1
url9=http://www.iesafe.cn/yahoo/index.htm
objurl9=http://xxx.xxx.com/
weight9=150
showIEWindow9=0
showStyle9=0
group9=0
[AdsNTGroupURL]
gurl21=http://www.deepdo.com/union/iplus/edodo.htm
```

像我们看到的，这个文件里有一些 URL 及一系列的位置和大小，我们基本上可以肯定看到的就是广告软件，位置和大小信息用于指定浏览器打开这些 URL 时窗口的位置和大小。令人讨厌的是，我们准备下载的应用程序可能会下载其他的应用程序，而它们又可能下载其他的，等等。甚至说，这些 URL 可能涉及更多的恶意内容。如果用户顺手点了某个链接，可能会陷入无穷无尽的麻烦，当然并不仅限于 CPU 负载过高这一种情形。这比较有意思，因为如果你看过一些病毒和蠕虫，经常会看到它们在同一主机上和平共处，基于这点考虑，有些病毒会删除它们识别出的其他病毒。广告软件与间谍软件截然相反。我认为虽然动机可能会存在争论，但贪婪导致企业恶意软件千方百计感染计算机而不顾其死活却是有目共睹。闲话少说，言归正传。

回到发起调用的函数，我们分析下面的代码（参见图 8-30）。

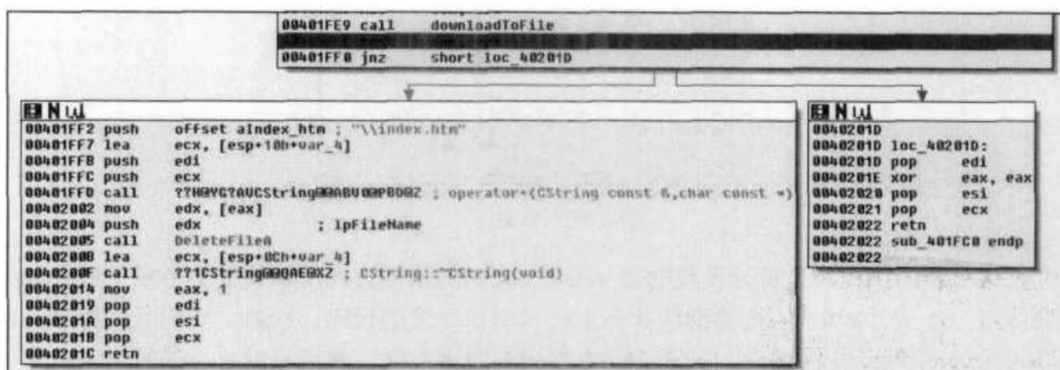


图 8-30

我们看到，在检查过返回值且成功调用 downloadToFile()之后，程序在 0x00401FF2 继续执行（如果调用失败则转至 loc_40201D）。再次构造路径的 CString，并调用 DeleteFileA()删除这个文件。然后，在 CString 的析构函数被调用之后，我们回到调用者（这个应用程序的消息处理代码）。从这里开始我们进入下一个函数——sub_401DF0()，这个函数的定义如图 8-31 所示。

更多资源请点击访问稀.酷客(www.ckook.com)

```
00401DF0
00401DF0
00401DF0
00401DF0 sub_401DF0 proc near
00401DF0
00401DF0 lpExistingFileName= dword ptr -1Ch
00401DF0 var_18= dword ptr -18h
00401DF0 lpFileName= dword ptr -14h
00401DF0 var_10= dword ptr -10h
00401DF0 var_C= dword ptr -0Ch
00401DF0 var_4= dword ptr -4
00401DF0

00401DF2 push    offset loc_403D08
00401DF7 mov     eax, large fs:0
00401DFD push    eax
00401DFE mov     large fs:0, esp
00401E05 sub     esp, 10h
00401E08 push    ebx
00401E09 push    esi
00401E0A mov     esi, ecx
00401E0C push    edi
00401E0D push    offset aAdsnt_ini ; "\\AdsNT.ini"
00401E12 lea     eax, [esp+2Ch+lpFileName]
00401E16 lea     edi, [esi+0A4h]
00401E1C push    edi
00401E1D push    eax
00401E1E call    ??H@YG?AVCString@@ABV0@PBD@Z ; operator+(CString const &,char const *)
00401E23 push    ecx                ; destFile
00401E24 lea     edx, [esp+2Ch+lpFileName]
00401E28 mov     ecx, esp
00401E2A mov     [esp+2Ch+var_10], esp
00401E2E push    edx
00401E2F mov     [esp+30h+var_4], 0
00401E37 call    ??0CString@@QAE@ABV0@@Z ; CString::CString(CString const &)
00401E3C lea     ecx, [esi+60h]
00401E3F mov     byte ptr [esp+2Ch+var_4], 0
00401E44 call    sub_401F90
00401E49 push    eax                ; lpszUrl
00401E4A mov     ecx, esi
00401E4C call    downloadToFile
00401E51 test    eax, eax
00401E53 jnz     short loc_401E78

00401E55 mov     eax, [esp+28h+lpFileName]
00401E59 push    eax                ; lpFileName
00401E5A push    0                  ; nDefault
00401E5C push    offset KeyName     ; "Version"
00401E61 push    offset AppName     ; "AdsNT"
00401E66 call    GetPrivateProfileIntA
00401E6C cmp     dword_40608C, eax
00401E72 jge     loc_401F6C
```

图 8-31

进入这个新的例程时，首先发现的是 AdsNT.ini 的路径被再次构造并随后被下载，如果出错会跳转到 loc_401E78，否则会继续执行 0x00401E55 处的代码。接着，我们看到程序会通过调用 GetPrivateProfileIntA()检查 INI 文件里的版本信息，然后根据检查的结果进行条件跳转（因此，这里是一个简单的版本兼容性检查）。继续往下，我们看到如图 8-32 所示的代码。

从 loc_401E78 开始，我们看到非常熟悉的模式，正在创建一系列的 CString，尽管通过调用 GetTempPathA()检索到的临时文件名里有些弯弯绕。从这个路径我们最终注意到临时文件名后加上了字符串\AdsNT.exe。从这儿起，我们完全可以通过调用 downloadToFile()了解正在发生什么。如果没记错的话，我们在这个例程前刚分析过它，它充当 URL 与磁盘上的本地文件的中间人，获取远程文件并保存它。因此，这个最初创建的、真实的可执行映像最终被下载下来了。

更多资源请点击访问稀.酷客(www.ckook.com)

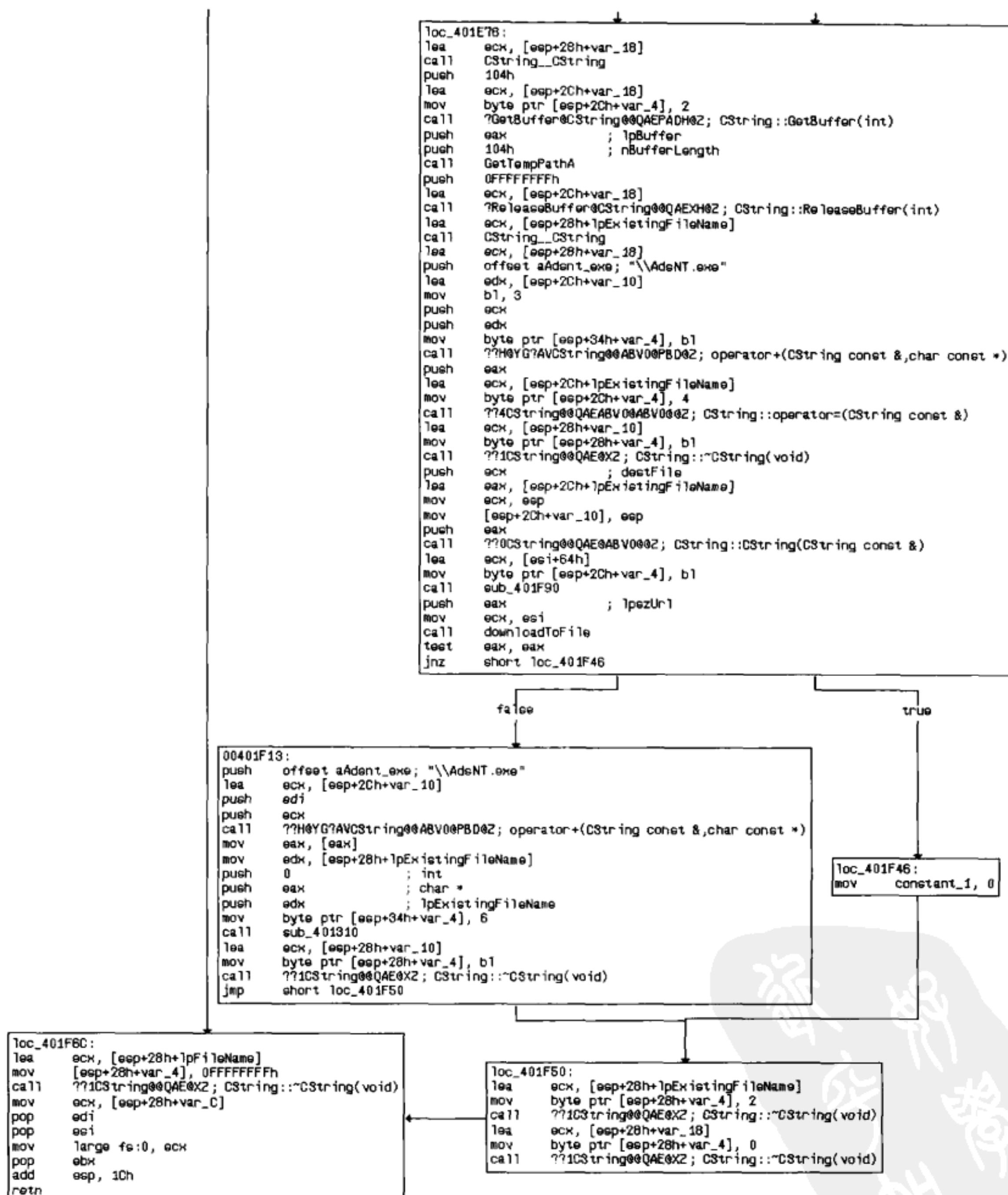


图 8-32

8

更多资源请点击访问稀.酷客(www.ckook.com)

迄今为止，我们可以断言这个应用程序有明确的恶意，因为它明显就是广告软件。它被加壳了，并且企图混淆里面存储的字符串，特别是后者这种迹象尤其应该引起我们的警觉。此外，我们看过它下载的 INI 文件后，看出它包含一些 URL 以及打开这些 URL 时预期的浏览器窗口大小和位置等信息。事情看起来都非常可疑，但仍有许多工作要做。OK，可以把它作为向读者提供的绝佳机会，以实践在本书中学到的知识。你可以在 Syngress 网站下载这个恶意软件，重复我们已经做过的某些步骤（例如脱壳），之后考虑下面的练习。

(1) 从 `0x00401F13` 开始，分析应用程序的剩余部分，看它们操作。

(2) 逆向分析/分析 AdsNT.exe。

(3) 在处理消息队列里，我们跳过了一段代码，它实际上由应用程序定义并充当回调函数，最终调用我们分析过的大多数函数。从调用 `downloadToFile()`（它第一个获取了 INI 文件）的第一个例程的结尾开始，一直查看到例程返回，注意是在应用程序的什么地方转交控制流的，在初始的进入点被实现前，不要回退应用程序。

更多资源请点击访问稀.酷客(www.ckook.com)

第 9 章

IDA脚本编写和插件

本章内容：

- 导言
- IDA 脚本编写基础
- IDC 语法
- 简单脚本示例
- 编写 IDC 脚本
- IDA 插件基础
- 插件语法
- 简单插件示例
- 间接调用插件

☑ 常见问题

更多资源请点击访问稀.酷客(www.ckook.com)

9.1 导言

IDA Pro 使用者众多，在多个逆向工程领域皆有应用。用户包括恶意软件分析员、漏洞研究员、软件逆向者、黑客、固件/硬件逆向者、开发人员、软件保护爱好者等。

IDA Pro 的扩展能力使之成为一个优秀的工具。脚本及插件的编写大幅提升了 IDA Pro 的交互性。IDA 是真正意义上的工具。用户操纵 IDA 完成需要完成的工作。

本章讨论用脚本或插件来扩展 IDA Pro。当我们逆向分析二进制文件时，最后都可以归纳总结出某种模式。这些模式可以是代码模式，也可以是适用自动化的重复任务。

可以用多种方式扩展 IDA。IDC 是其内置的脚本语言。IDC 在结构上与 C 语言类似，它的优势在于已经解释而不需要额外的工具支持。更复杂的任务由插件来完成。Hex-rays 为客户提供 SDK，允许他们用它编写插件。这个 SDK 是用 C++写的，支持多种编译器。现在市面上也有一些第三方的混合解决方案。这些混合解决方案封装了 IDC 函数和一些 SDK 函数。(你可以从 www.syngress.com/solutions 下载本章用到的代码和脚本。)

9.2 IDA 脚本编写基础

IDC 是 IDA 内置的脚本语言，其语法与 C 非常类似，熟悉 C 的人应该可以很快上手。它是解释型语言。

有两种标准的方法执行 IDC。

- 在 IDA 里可以直接执行 IDC 语句。按下 **SHIFT+F2** 会弹出一个对话框。在对话框里输入的语句将被执行。这个对话框适合输入小段代码。通常不在对话框里定义函数。
- 可以载入 IDC 文件。在菜单栏 **File | IDC File** 中，就可以打开一个文件浏览器窗口，你可以在此选择要载入的 IDC 文件。

提示

执行 IDC 表达式的另一个方法是使用可选的命令行。必须在 idagui.cfg 里把命令行选项设为 yes。

```
DISPLAY_COMMAND_LINE=YES // 显示表达式/IDC 命令行
```

9.3 IDC 语法

IDC 脚本语言从 C 借鉴了大量语法。所有的语句以分号结束。类似处还包括许多相同的关键字，包括 `if`、`if else`、`while`、`do while`、`continue`、`break`。本节将介绍 IDC 语法，特别是 IDC 与 C 之间的差别。

对于访问反汇编后的内容，编写脚本比编写插件要容易得多。脚本可以以文件的方式运行，也可以在 IDC 对话框中运行。在我们介绍函数前，本节中的例子都使用对话框方式。即使简单

更多资源请点击访问稀.酷客(www.ckook.com)

的脚本也会加速分析，有助于任务的自动化。为了运行 IDC 脚本，必须把 idb 文件载入 IDA。

9.3.1 输出

从 K&R 的《C 程序设计》开始，大多数程序设计书里所教的第一个例子都是 hello world 程序。从脚本中取出数据给用户是非常重要的。这可以是真正的输出，也可以是出于调试目的的输出。

按 **SHIFT+F2**，或使用菜单（**File | IDC** Command...）打开 IDC 命令窗口，输入：

```
Message("Hello world\n");
```

你看到的对话框应该与图 9-1 相似。对话框里可以输入多个语句，但现在有 Message 语句就足够了。点击 OK 后，hello world 将出现在消息窗口里。

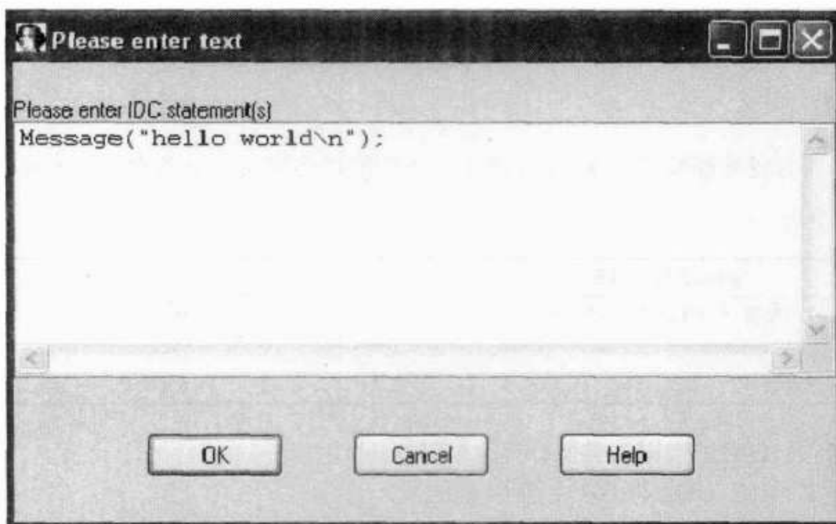


图 9-1 hello world

Message 函数与 C 的 printf 函数类似，也使用格式化串。其原型是：

```
void Message(string format,...);
```

使用 Message 的其他形式：

```
Message("%s\n", "hello world");
Message("%x\n", 0x40100);
Message("%x is the cursor\'s address\n", ScreenEA()));
```

使用格式化串%s 的第一个例子也输出 hello world。使用%x 的第二个例子输出十六进制数值。不过这两个例子是杜撰的（用于说明参数的用法），第三个例子则使用了一个新的 IDC 函数。ScreenEA 函数返回当前光标所处的地址。脚本中经常会看到这个函数。

更多资源请点击访问稀.酷客(www.ckook.com)

Message 不是唯一的输出函数，但是最常用的。另外两个输出函数是 Warning 和 Fatal。它们都使用格式化串，且有同样的函数原型。

Warning 用于向用户报错。它将弹出一个与图 9-2 类似的对话框。Fatal 用得比较少，因为它在退出 IDA 时不保存数据库。

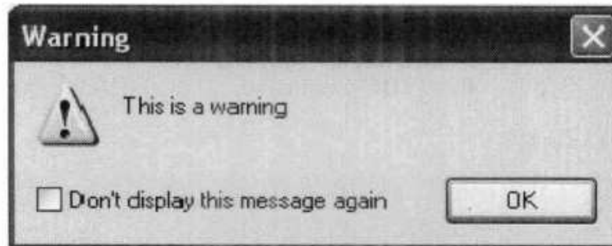


图 9-2　warning 窗口

9.3.2　变量

IDC 中的所有变量都被定义成 auto 类型。下面的语句声明一个名为 counter 的变量：

```
auto counter;
```

auto类型可以表示	例　子
32位整数（在IDA Pro64中为64位）	0x00401000
字符串	"hello world"
浮点数	5.23

变量根据它们包含的数据的类型，有大小限制：

- 整数是 32 位的（IDA Pro 64 中是 64 位的）；
- 字符串最多包含 1023 个字符；
- 浮点变量最多 25 个十进制数位。

auto 变量可以表示不同类型的数据。因此，操作不同类型的数据时会用到转换规则。不过在编写脚本时，类型转换没有在 C 中那么常见。有些函数用于手动执行类型转换：

```
long(expr)
char(expr)
float(expr)
```

变量的声明和赋值必须在不同的语句中进行。

```
auto  currentAddress;
currentAddres=ScreenEA();
```

许多标准的 C 操作符（+、-、/、*、%、<<、>>、++、--）在 IDC 中一样使用。还有一些

更多资源请点击访问稀.酷客(www.ckook.com)

操作符不被支持，包括组合赋值操作符（+=）和逗号操作符（,）。不像 C 那样，加字符串将会连起来。

所有的变量都有局部作用域[①]。这意味着它们仅仅在定义它们的函数里是可用的。对于我们当前的目的，这适用于 IDC 命令窗口。我们在后面将会介绍函数，以及允许全局变量的方法。

9.3.3 条件

大多数标准的 C 条件语句都可用，包括 if、if else 及三元操作符“?:”。C 语言中的 switch 语句是不可用的。下面的代码段演示了 if else 的用法。

```
auto currAddr;
currAddr = ScreenEA();
if (currAddr % 2)
    Message("%x is odd\n", currAddr);
else
    Message("%x is even\n", currAddr);
```

9.3.4 循环

循环可以用 for、while、do while 实现。除了不允许使用逗号运算符外，它们的用法与 C 类似。IDC 不支持 switch 语句，但可以用多个 if、else if 语句来实现类似的功能。下面的代码段演示一个循环，并引入了一些新的 IDC 函数和概念。

```
auto origEA, currEA, funcStart, funcEnd;
origEA = ScreenEA();
funcStart = GetFunctionAttr(origEA, FUNCATTR_START);
funcEnd = GetFunctionAttr(origEA, FUNCATTR_END);
if(funcStart == -1)
    Message("%x is not part of a function\n", origEA);
for(currEA=funcStart; currEA != BADADDR;currEA=NextHead(currEA, funcEnd))
{
    Message("%8x\n", currEA);
}
```

注意

BADADDR 是 IDC 中使用的常量。它代表一个错误或函数返回了无效的结果。在脚本里用它测试结果，有时也用它初始化变量。

一些 IDC 函数在出错时返回-1，BADADDR 的内部表示就是-1。

下面的代码段输出当前函数里每条指令的地址。它引入了两个新的 IDC 函数，即 GetFunctionAttr 和 NextHead。它们的原型是：

① 类似 C 中的命名空间。

更多资源请点击访问稀.酷客(www.ckook.com)

```
long GetFunctionAttr(long ea, long attr);
long NextHead(long ea, long maxea);
```

GetFunctionAttr 允许我们查询函数的某些属性。参数 ea 是函数内的任意地址。参数 attr 是我们比较感兴趣的属性。在这个例子里，我们查找一个给定的地址是否是一个函数的开始/结束地址。如果地址 ea 不在函数内，那么 GetFunctionAttr 返回-1。

NextHead 返回下一条指令或数据项。参数 ea 是开始地址，maxea 是结束地址。如果在给定的地址范围里没有定义指令或数据，那么将返回 BADADDR。在像 IA-32 这样包含变长指令的体系结构里，必须用 NextHead 对指令进行迭代处理。在 RISC 这样设置了指令长度的体系结构里，可以通过递增地址的方式处理，而不需要使用 NextHead 了。因为递增不会检查 IDA 中是否定义了该项，因此我们应该避免这么做。

下面的代码段演示了如何用 while 循环实现同样的功能。

```
auto origEA, currEA, funcStart, funcEnd;
origEA = ScreenEA();
funcStart = GetFunctionAttr(origEA, FUNCATTR_START);
funcEnd = GetFunctionAttr(origEA, FUNCATTR_END);
if (funcStart == -1)
    Message("%x is not part of a function\n", origEA);
currEA = funcStart;
while (currEA != BADADDR)
{
      Message("%8x\n", currEA);
      currEA = NextHead(currEA, funcEnd);
}
```

9.3.5　函数

一旦简单的 IDC 片段不能满足我们的需求，就需要函数了。函数必须在 IDC 文件之内。IDC 中所有函数都被定义成静态的。下面的代码是一个示例函数。

```
static outputCurrentAddress(myString)
{
    auto currAddress;
    currAddress = ScreenEA();
    Message("%x %s\n", currAddress, myString);
    return currAddress;
}
```

IDC 中的函数声明与 C 中的有些差别，主要与类型相关。因为 IDC 只有一种类型——auto，函数的参数或返回值都不需要指定类型。IDA 只接受声明里不带类型的函数。IDC 文件将在 9.4 节介绍。

通常是在一个 IDC 文件中声明函数。在 IDC Command 窗口里输入上面的函数，将产生一个

更多资源请点击访问稀.酷客(www.ckook.com)

错误“Syntax error near static”。这涉及 IDC Command 窗口怎样工作的问题。最早的解决方案由 Willem Jan Hengeveld 提供（见http://www.xs4all.nl/~itsme/projects/disassemblers/ida.html)。

从内部看，对话框的内容保存在名为_idc 的函数里。因此，在对话框里输入一个函数声明实际上是企图在另外一个函数（_idc）里声明一个函数。因此，在声明新函数前，需要先关闭_idc 函数。新函数还必须丢掉右大括号。为了声明 outputCurrentAddress，输入：

```
}
static outputCurrentAddress(myString)
{
    auto currAddress;
    currAddress = ScreenEA();
Message("%x %s\n", currAddress, myString);
return currAddress;
```

我们应该不会看到错误提示了。尽管我们声明了一个函数，但它并不执行。如果我们想执行它，则需要把它作为_idc 函数的一部分。

```
    AddHotkey("Alt-f9", "outputCurrentAddress2");
    outputCurrentAddress2();
}
static outputCurrentAddress2()
{
    auto currAddress;
    currAddress = ScreenEA();
    Message("%x\n", currAddress);
    return currAddress;
```

前面的代码引入一个新 IDC 函数——AddHotKey。这个函数把一个按键与一个 IDC 函数名绑在一起。目标函数不能带参数。增加热键绑定后，接着执行 outputCurrentAddress2。以后就可以通过热键或从命令窗口调用来执行函数 outputCurrentAddress2 了。

局部作用域和全局作用域

作用域是指变量或函数在代码中可见的范围。我们将用函数 outputCurrentAddress 中的变量 currAddress 作为局部作用域的例子。currAddress 变量仅仅在此函数中可见，不可以从其他函数里访问它。

函数声明被放在全局作用域里，因此可以从其他函数中调用此函数。这也包括从命令窗口里调用。一旦函数被定义了，在我们用同样的名字声明另外的函数或在 IDA 会话终止之前，它会一直保留在全局作用域里。关闭 idb 文件将从内存中清除所有的 IDC 函数。

一旦 outputCurrentAddress 被声明，我们就可以在命令框里输入下面的内容调用它。

```
outputCurrentAddress("some string");
```

通过把自己的 IDC 函数库加到 ida.idc 文件里，我们就可以在 IDA 中使用自己的 IDC 函数了。

更多资源请点击访问稀.酷客(www.ckook.com)

这个文件位于 IDA Pro 安装目录下的 idc 目录里。

提示

可以考虑使用自定义的前缀，以避免与其他脚本中的函数产生命名冲突。

9.3.6　全局变量

自动变量只在定义它们的函数中是可见的。我们需要一个方法使数据在整个脚本里可见，这就需要全局变量了。IDC 不直接支持全局变量。不过，可以用数组模拟全局变量。

IDC 内置数组。数组可以包含字符串数据或长整型数据。下面的代码将创建一个数组并定义一些项。

```
auto gArray;
gArray = CreateArray("myGlobals");
```

代码引入一个新的 IDC 函数——CreateArray。CreateArray 函数的原型是：

```
long CreateArray(string name);
```

函数名必须少于 120 个字符。函数在数组创建成功后返回数组 id，如果创建失败则返回-1。下面的代码向数组中加入一些项。

```
SetArrayLong(gArray, 23, 415);
SetArrayString(gArray, 0, "some string data");
```

这些新函数的原型是：

```
/*
arguments:
    id       -     array id
    idx      -     index of an element
    value    -     32bit value to store in the array
    str      -    string to store in array element
returns: 1-ok, 0-failed
*/
success SetArrayLong(long id,long idx,long value);
success SetArrayString(long id,long idx,string str);
```

索引 idx 可以是任意 32 位数。索引值不必连续，因为变量空间只有在赋值时才会分配。前面的例子把值 415 赋给索引 23，把字符串 “some string data” 赋给索引 0。全局数据被赋值后，我们就可以从脚本的其他地方或其他的脚本或命令窗口访问了。

为了访问全局数据，需要引入新的 IDC 函数。为了访问数组的成员需要其 id。下面的代码演示了新 IDC 函数。

```
auto arrayId, strItem, longItem;
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
arrayId = GetArrayId("myGlobals");
strItem = GetArrayElement(AR_STR, id, 0);
longItem = GetArrayElement(AR_LONG, id, 23);
```

这里引入了两个新的 IDC 函数，即 GetArrayId 和 GetArrayElement。这两个函数的原型是：

```
// get array id by its name
// arguments: name - name of existing array.
// returns:        -1 - no such array
//                    otherwise returns id of the array

long GetArrayId(string name);
/* get value of array element
         arguments: tag     - tag of array, specifies one of two
                              array types:
#define AR_LONG 'A' // array of longs
#define AR_STR 'S' // array of strings
                        id      - array id
                        idx     - index of an element
        returns:        value of the specified array element.
                        note that this function may return char or long
                        result. Unexistent array elements give zero as
                        a result.
*/
string or long GetArrayElement(long tag,long id,long idx);
```

GetArrayElement 函数的参数顺序与 SetArray 函数的不一样。在数组被定义后，可以在任何地方使用这些函数。考虑到篇幅的原因，上面的代码段没有做错误检测，但实际应加上错误检测。

有人发布了一些经常用到的 IDC 库。这些库还包括了一些全局变量并做了错误检测。举例来说，由 lallous 编写的 common.idc(http://www.openrce.org/downloads/details/81/Common_Scripts)。除全局变量外，它还包括其他有用的函数。下面的片段是使用 common.idc helper 函数带的全局变量的例子。

在使用 InitGlobalVars 前，我们需要先初始化全局变量，这很可能出现在脚本的 main 函数中。

```
if (InitGlobalVars() == 0)
{
    Message("InitGlobalVars() failed\n");
}
```

一旦初始化完毕，为了读/写全局变量，我们可以访问4个宏定义。下面是使用了和先前相同的命名约定(index、value、string)的宏。

9

更多资源请点击访问稀.酷客(www.ckook.com)

```
SetGlobalVarLong(index, value)
SetGlobalVarString(index, string)
GetGlobalVarLong(index)
GetGlobalVarString(index)
```

使用同前面例子中一样的数据设置数组元素：

```
SetGlobalVarLong(23, 415)
SetGlobalVarString(0, "some string data")
```

访问这些项变得更清晰、简单了。

```
auto strItem, longItem;

strItem = GetGlobalVarString(0)
longItem = GetGlobalVarLong(23)
```

当经常需要一些持久信息时，全局变量显得尤为有用。可以把库加到 idc.idc 里，之后就可以从其他的脚本里访问它们了。我们觉得有用的函数都可以加进去。

提示

Message 应当在最左边包含相关地址。例如：

```
Message("%x breakpoint set\n", bpAddr);
4014c6 breakpoint hit
```

这样一来，就可以通过双击这个地址使 IDA 跳到该地址。

9.4　简单脚本示例

到目前为止，大多数例子都在使用 IDC 命令窗口。命令窗口对于交互式脚本来说非常好，但不久就会发现它比较笨拙了。脚本使我们可以运行 IDC 代码，而不必每次都把它们重新输入到对话窗里。

代码片段和脚本的区别是什么？没有太大区别。所有的代码都必须存在于函数中。甚至命令窗口里的代码也是位于_idc 函数中的。下面是脚本的大致模板。

```
#include <idc.idc>
static some_function()
{
}
static main()
{
}
```

IDC 使用类似于 C 的预处理指令。文件 idc.idc 包含 IDC 函数的原型和常量，且通常所有的脚本都会包含此文件。[①]也可以把这个文件当做文档来读，它包含了与 help 文件相同的信息。IDC

① 因为 IDA 的文档比较缺乏。

更多资源请点击访问稀.酷客(www.ckook.com)

支持#define、#ifdef 和其他命令预处理指令。

由脚本执行 main 函数。如果脚本中没有 main 函数，其他函数将停伫在内存中且仍然可以被调用。

工具和陷阱

搭建IDC开发环境

有很多东西可以使开发 IDC 脚本变得更方便。合适的文本编辑器尤为重要。

文本编辑器最好支持语法高亮。语法高亮使用不同的颜色、字体、字号来表示不同类型的数据。这使我们可以很容易地从其他数据中区分出 IDC 函数调用和关键字。现在许多编辑器都支持语法高亮。你喜欢的文本编辑器可能就有选项来增加新语法。Sebastian Porst为Crimson文本编辑器上传了一个IDC语法文件(http://www.the-interweb.com/serendipity/exit.php?url_id=157&entry_id=26)。

除了选择合适的编辑器外，我还把扩展名为 idc 的文件的默认打开方式改为用此编辑器打开。在 IDA 中，你可以设置喜欢的编辑器来编辑 IDC 脚本。设置选项在 Options| General | Misc | Editor 中。

IDA 默认从最后一次打开的位置起浏览 IDC 文件。这个行为可以通过编辑配置文件来改变。IDA 安装目录下的 cfg 目录中有许多配置文件。我们对 idagui.cfg 文件尤其感兴趣，它包括与 idag.exe 相关的选项。

在开发和使用 IDC 文件时，经常会改变的选项是 OPEN_DEFAULT_IDC_PATH。这个选项缺省值为 NO。把它改为 YES，将始终在 idc 目录中打开 IDC 文件浏览器对话框。修改后需要重启 IDA 更改才能生效。

图 9-3 中的脚本将把函数重设为默认的颜色。覆盖范围工具在跟踪执行时将把基本块着色，类似于图 9-4。其他的时候，用户将对块着色来高亮显示某些代码。不论是跟踪运行还是我们对某些部分不再感兴趣时，都需要重置颜色。

```
#include <idc.idc>
static main(void)
{
    auto origEA, currEA, currColor, funcStart, funcEnd;
    origEA = ScreenEA();
    funcStart = GetFunctionAttr(origEA, FUNCATTR_START);
    funcEnd = GetFunctionAttr(origEA, FUNCATTR_END);

    Message("Welcome to resetColor.idc\n");
    if (funcStart == -1 || funcEnd == -1)
    {
```

图 9-3 重置颜色的 IDC 脚本

9

更多资源请点击访问稀.酷客(www.ckook.com)

```
            Message("*** Error: not in a function **\n");
            return -1;
        }
        Message("[*] Function: %s\n", GetFunctionName(funcStart) );
        Message("[*] start == 0x%x, end == 0x%x\n", funcStart, funcEnd);
        for (currEA = funcStart; currEA != BADADDR; currEA =
    NextHead(currEA, funcEnd) )
        {
            if (SetColor(currEA, CIC_ITEM, DEFCOLOR) == 0)
            {
                Message("*** Error: SetColor failed 0x%x **\n", currEA);
            }
        }
        Refresh();
        Message("resetColor is done\n");
    }
```

图 9-3　（续）

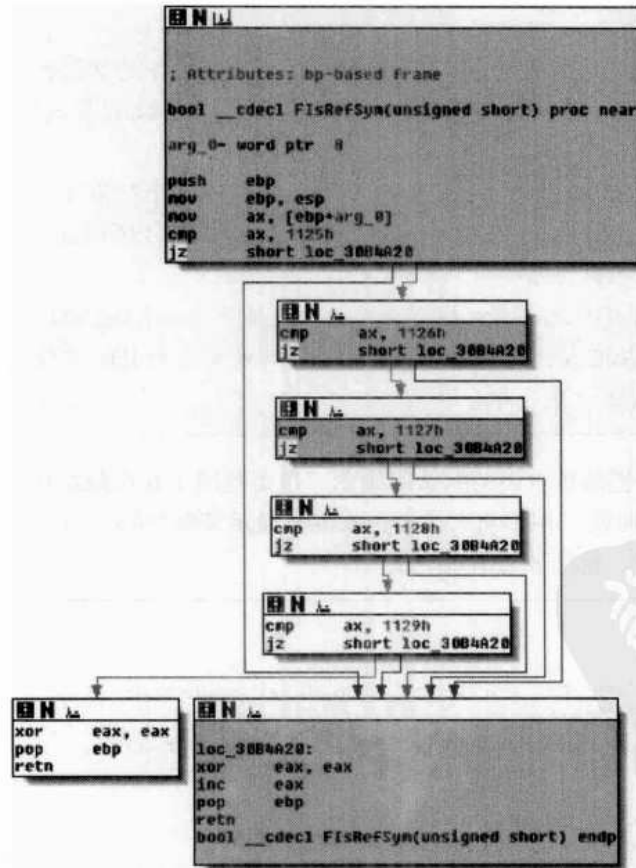


图 9-4　被跟踪的函数

更多资源请点击访问稀.酷客(www.ckook.com)

在你喜欢的文本编辑器（最好支持语法高亮）里输入图 9-3 中的代码。并把它保存成以.idc 结尾的文件。用菜单 **File | IDC file...**选择运行它，则这个脚本将被运行，然后把控制权返回给用户。一个新窗口——**Recent IDC scripts** 将出现，如图 9-5。按左边的按钮将编辑此脚本，按右边的按钮是执行此脚本。这允许我们快速编辑脚本。

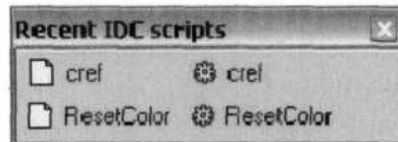


图 9-5 最近的 IDC 脚本

脚本里的代码和前面节中的代码段非常相似。通过 `GetFuncAttr` 调用确定当前函数的地址范围。用循环迭代函数的地址空间，并用一个新的 IDC 函数——`SetColor` 重置颜色。

脚本虽然很简单，但解决紧急问题时很有用。接下来的一节在引入更多 API 及概念时，将继续遵循此思想。

注意

IDA Pro 的帮助文件中包含了 IDC 语言的相关文档。它简单地描述了如语句、表达式和循环等结构。

这个文档也可以充当所有内置 IDC 函数的 API 参考手册。

9.5 编写 IDC 脚本

脚本语言现在很流行，因为它们可以即时提供结果。用户经常要完成的是一些简单的任务，而不是开发很成熟的产品。

脚本能够也应该被用来自动化简单的任务。完整的解决方案，尤其是在逆向工程领域里，看上去正呈增长趋势。脚本与此趋势保持一致。虽然有时候我们是为了特殊的反汇编项目来编写脚本的，但其他时候这些脚本也可以重复使用。

编写脚本以及插件并不必将用户排除在外。这正如 IDA 是一个交互式反汇编器，而脚本是帮助我们进行逆向分析的交互式工具。

9.5.1 用 IDC 解决问题

本节介绍一个怎样用 IDC 解决具体问题的例子。它肯定算不上一个完整的解决方案，但它演示了只用很少的代码和时间可以做些什么。

1．问题

C++使用间接调用调用了许多函数。但 IDA Pro 并没有为这些函数创建交叉引用。

更多资源请点击访问稀.酷客(www.ckook.com)

2. 问题背景

对逆向工程师来说，逆向 C++带来了一些新的挑战。这些挑战，像大多数逆向工程一样，可以在静态或运行时解决。最近，作为 OpenRCE 网站的系列文章，Igor Skochinsky 发表了一些静态分析的研究结果。(http://www.openrce.org/articles/full_view/21) (http://www.openrce.org/articles/full_view/23)，此外还提供了一些 IDC 脚本。另外， IBM ISS 研究机构的 Paul Vincent Sabanal 和 Mark Vincent Yason 发布了一个论文（https://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/Paper/bh-dc-07- Sabanal_Yason-WP.pdf)，讨论一个基于 IDAPython 的内部工具。

与逆向 C++代码相关的一个问题是间接调用。经常会看到类似于图 9-6 中的代码——通过寄存器和偏移量进行调用。看上去 ecx 没有初始化就被使用了。ecx 被传递给函数，代表 this 指针。因为 IDA Pro 并不知道哪个函数正被调用，所以也就不会为它创建交叉引用。

```
; Attributes: bp-based frame

sub_30CBA25 proc near

arg_0= dword ptr  8
arg_4= dword ptr  0Ch
arg_8= dword ptr  10h

push    ebp
mov     ebp, esp
mov     eax, [ecx]
push    esi
mov     esi, [ebp+arg_8]
lea     edx, [ebp+arg_8]
push    edx
push    [ebp+arg_4]
mov     [ebp+arg_8], esi
push    0
push    [ebp+arg_0]
call    dword ptr [eax+14h]
test    eax, eax
jz      short loc_30CBA4E
```

图 9-6　间接调用

如果我们用调试器跟踪执行，就可以认出目标函数。从目标函数检查交叉引用将显示如图 9-7 所示的结果。

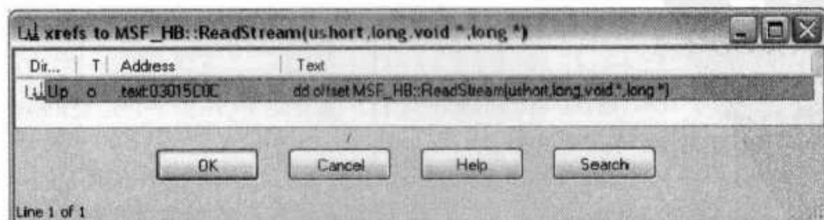


图 9-7　脚本之前 MSF_HB::ReadStream 的交叉引用

更多资源请点击访问稀.酷客(www.ckook.com)

所有的引用都是指针，而不是函数调用。指针是 VTable 的一部分。VTable 是指向特定对象里函数的指针数组。

```
.text:03015BF8 const MSF_HB::'vftable' dd offset
MSF_HB::QueryImplementationVersion(void)
.text:03015BF8; DATA XREF: MSF_HB::MSF_HB(void)+9□o
.text:03015BF8; MSF_HB::~MSF_HB(void)+9□o
.text:03015BFC        dd offset MSF_HB::QueryImplementationVersion(void)
.text:03015C00        dd offset MSF_HB::GetCbPage(void)
.text:03015C04        dd offset MSF_HB::GetCbStream(ushort)
.text:03015C08        dd offset MSF_HB::GetFreeSn(void)
.text:03015C0C        dd offset MSF_HB::ReadStream(ushort,long,void *,long *)
.text:03015C10        dd offset MSF_HB::ReadStream(ushort,void *,long)
.text:03015C14        dd offset MSF_HB::WriteStream(ushort,long,void *,long)
.text:03015C18        dd offset MSF_HB::ReplaceStream(ushort,void *,long)
.text:03015C1C        dd offset MSF_HB::AppendStream(ushort,void *,long)
.text:03015C20        dd offset MSF_HB::TruncateStream(ushort,long)
.text:03015C24        dd offset MSF_HB::DeleteStream(ushort)
.text:03015C28        dd offset MSF_HB::Commit(void)
.text:03015C2C        dd offset MSF_HB::Close(void)
.text:03015C30        dd offset MSF_HB::GetRawBytes(int(*)(void const *,long))
.text:03015C34        dd offset MSF_HB::SnMax(void)
.text:03015C38        dd offset TM::PPdbFrom(void)
.text:03015C3C dd offset MSF_HB::CloseStream(ulong)
```

图 9-8 MSF_HB::'vftable'

当对象的方法被调用时，VTable 被访问，然后用指向相应函数的偏移量进行调用。图 9-6 中的代码使用 VTable 来调用第一个 MSF_HB::ReadStream 函数。

提示

如果使用 Alexander Sotirov 的 Determina PDB 插件(http://www.determina.com/security.research/utilities/index.html)，在微软的二进制里查找 VTable 就很简单了。pdb 包含的符号就包括 VTable 名。可以通过搜索下面的字符串找出所有的 VTable:

```
::' vftable' dd
```

注意，vftable 前面的字符是一个反向单引号，而跟在 vftable 后面的字符是个单引号。

3. 建议解决方案

为了找到调用地址，我们可以通过编写脚本控制调试器并检查栈。在 IDA 5.2 之前，与调试器相关的 IDC 功能非常有限，函数不允许处理任何调试器事件，例如断点。不过，有一个变通方案——使用条件断点。

更多资源请点击访问稀.酷客(www.ckook.com)

下面的脚本在 IDA 5.2 版之前就写好了，因此并不依赖于新函数。随后我们将讨论 5.2 版中的新函数。图 9-9 是编辑断点对话框。条件可以是任意 IDC 语句包括函数调用。

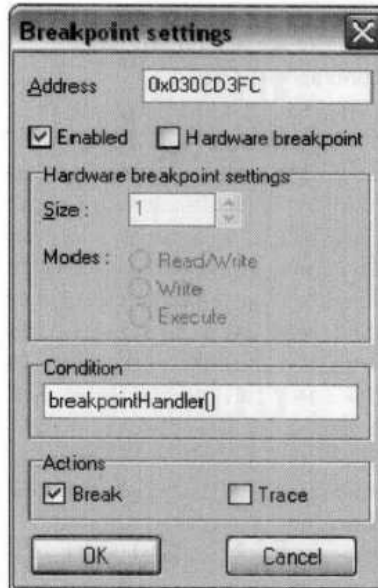


图 9-9　为处理程序设置条件

当断点被触发时将调用这个函数，因此我们就可以在断点中断期间运行代码。如果我们不想停止调试器，函数就只返回 0。函数判断为 `false`（假）时允许继续执行。下面的代码用于在每次断点被触发时记录 EAX 的值。

```
static breakpointHandler()
{
    Message("%x bp hit, EAX == 0x%x\n", EIP, EAX);
    return 0; // don't stop on breakpoint
}
```

为了检查调用者，我们从查看栈开始。在函数调用期间，返回地址被压入栈。我们需要指向调用指令的地址，它是返回地址前一条指令的地址。在图 9-6 中的调用发生后，返回地址指向 `test` 指令而不是 `call`。更新后的 `breakpointHandler` 函数将记录调用者。

```
static breakpointHandler()
{
    auto caller;
    caller = PrevHead(Dword(ESP), (Dword(ESP) - 10));
    Message("%x bp hit, caller == %x\n", EIP, caller);
    return 0; // don't stop on breakpoint
}
```

更多资源请点击访问稀酷客(www.ckook.com)

这里引入了一个新的 IDC 函数——PrevHead。它的原型是：

```
long    PrevHead          (long ea, long minea);
```

PrevHead 搜索前面定义的指令或数据。ea 参数是向后搜索的起始地址，minea 是包含在搜索中的最小地址。breakpointHandler 中的搜索向后查询 10 个字节。寄存器的调用指令一般只有 3 个字节，因此搜索可以找出 call。调用者被确定后，就可以加上交叉引用了。更新后的 breakpointHandler 增加了交叉引用。

```
static breakpointHandler()
{
    auto caller;
    caller = PrevHead(Dword(ESP), (Dword(ESP) - 10));
    AddCodeXref(caller, EIP, XREF_USER | fl _CN);
    return 0; // don't stop on breakpoint
}
```

AddCodeXref 增加交叉引用。其原型是：

```
//        Flow types (combine with XREF_USER!):
#define   fl_CF 16        // Call Far
#define   fl_CN 17        // Call Near
#define   fl_JF 18        // Jump Far
#define   fl_JN 19        // Jump Near
#define   fl_F 21         // Ordinary flow
#define XREF_USER 32      // All user-specified xref types
                          // must be combined with this bit
void AddCodeXref(long From,long To,long flowtype);
```

Message 函数被移走了，因为它会降低断点处理的速度。检查 DLL 中的其他调用会发现交叉引用为 Call Near。breakpointHandler 函数现在完整了（参见图 9-10）。

```
#include <idc.idc>
static breakpointHandler()
{
        auto caller;
        caller = PrevHead(Dword(ESP), (Dword(ESP) - 10));
        AddCodeXref(caller,EIP, XREF_USER | fl _CN);
        return 0; // don't stop on breakpoint
}
static setBPs()
{
        auto currAddr;
        auto vStart;
        auto vEnd;
```

图 9-10　VTable xref 脚本

更多资源请点击访问稀.酷客(www.ckook.com)

```
        auto virFunc;

        Message("setBPs() executed\n");
        vStart = SelStart();
        vEnd = SelEnd();

        Message("start = 0x%x\n",vStart);
        Message("end = 0x%x\n",vEnd);
        if ((vStart == BADADDR) || (vEnd == BADADDR))
        {
              Message("No selection made !!\n");
              return;
        }
        if ((vStart - vEnd) %4 != 0)
        {
        Message("not DWORD aligned\n");
              return;
        }
        for (currAddr = vStart; currAddr < vEnd; currAddr = currAddr + 4)
        {
              virFunc = Dword(currAddr);
              if (GetBptAttr(virFunc, BPTATTR_EA) == -1) // no bpt there yet
              {
                  if (!AddBptEx(virFunc, 0, BPT_SOFT))
                  {
                      Message("AddBptEx() failed 0x%x\n;", virFunc);
                      return;
                  }

                  if (!SetBptCnd(virFunc, "breakpointHandler()"))
                  {
                      Message("SetBptCnd() failed 0x%x\n;", virFunc);
                      return;
                  }
                  Message("BP 0x%x set\n", virFunc);
              }
              else
              {
                  Message("BP already set 0x%x\n", virFunc);
              }
        }
}
static main()
{
    AddHotkey("Alt-f9", "setBPs");
}
```

图9-10　（续）

更多资源请点击访问稀.酷客(www.ckook.com)

在运行这个脚本时，其中的函数被载入内存，main 函数被执行，main 的唯一目的是为 setBPs 函数设置热键。

选择类似于图 9-8 的 VTable，然后按下热键，将调用 setBPs 函数。这个函数的用途是在 VTable 表中找到的目标上设置断点。因为每个地址只能加一个断点，所以函数首先会检查是否已经有断点了。如果没有断点，就新加一个软件断点。随后为这个断点加上条件，其条件就是 breakpointHandler 函数。在这个例子里，我们选择返回 0，而不是在断点上停下。

使用脚本并运行调试器后的结果如图 9-11 所示。它比图 9-7 中的原始版本有了很大改进。

图 9-11　使用脚本后的 MSF_HB::ReadStream 的交叉引用

4. 可能的改进

如果断点设在经常被调用的函数上，可能会降低性能。因此，可以把一个全局变量当做计数器，当计数器达到预设（限制）值，就移除断点。还可以为调用指令增加包含目标地址的注释，这样一来，用户就可以通过双击这个地址直接跳到目标函数。

在获得交叉引用数据后，我们就能分析它以确定它们所代表的类的信息。在 C++中，方法的可见性可以是 public、protected 或者 private。

Public	目标函数最少有一个来自在任何VTable中都找不到的函数的调用
Private	目标函数只能被它自己的VTable中的其他函数调用
Protected	目标函数只能被位于VTable表中的函数调用

9

更多资源请点击访问稀.酷客(www.ckook.com)

如果一个函数位于多个 VTable 的相同偏移处，这些类之间可能存在继承关系。

交叉引用信息与静态分析相结合可以重建对象模型。可以用图形工具描绘这个模型，例如用 UML。

9.5.2　新的 IDC 调试器功能

发布 IDA Pro 新版本时通常会引入新的 IDC 函数，而很少会丢弃函数。新函数反映了新版本中增加的特性。

IDA 5.2 新增了 53 个函数。其中最重要的是增加了调试器对脚本的支持（http://www.hex-rays.com/idapro/scriptable.htm）。现在调试器完全支持 IDC 的脚本编写。我们可以控制调试器的每一方面。这包括响应调试器事件、附加到进程、跟踪。

支持脚本编写的调试器为我们带来许多可能。通常使用 Ollydbg 的 OllyScript 脚本语言或 Immunity Debugger 所使用的 Python 语言编写脚本，把用已知加壳程序处理过的二进制文件脱壳。运行时分析可以反馈给静态分析。

插件环境下的调试器通常要求有一个回调函数来处理事件。IDC 接口需要考虑等待事件的阻塞调用的等待时间。所用的函数是 GetDebuggerEvent，它的原型是：

```
long GetDebuggerEvent(long wfne, long timeout);
```

可以把 timeout（超时）设为-1，表示无限长。wfne 标志如下所示：

```
WFNE_ANY        return the first event
WFNE_SUSP       wait until the process gets suspended
WFNE_SILENT    set: be slient, clear:display modal boxes if necessary
WFNE_CONT      continue from the suspended state
```

在断点引发挂起状态前，我们通常想等待。可能的返回值如下：

```
// debugger event codes
NOTASK          process does not exist
DBG_ERROR       error (e.g. network problems)
DBG_TIMEOUT     timeout
PROCESS_START   New process started
PROCESS_EXIT    Process stopped
THREAD_START    New thread started
THREAD_EXIT     Thread stopped
BREAKPOINT      Breakpoint reached
STEP            One instruction executed
EXCEPTION       Exception
LIBRARY_LOAD    New library loaded
LIBRARY_UNLOAD  Library unloaded
INFORMATION     User-defined information
SYSCALL         Syscall (not used yet)
WINMESSAGE      Window message (not used yet)
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
PROCESS_ATTACH    Attached to running process
PROCESS_DETACH    Detached from process
```

这个新加入 IDC 的功能非常受欢迎，因为它使我们可以很方便地通过脚本控制调试器。稍后将介绍一个与调试器协同工作的插件。

9.5.3 有用的 IDC 函数

本节包含 IDC 函数的一个示例，你可能在其他的脚本和编写新脚本时见过这些函数。这些函数根据功能分成不同的类别，而且我们在此简单地描述了它们的一些用法。

1. 读写内存

有三个函数可以读内存，它们根据所读数据的大小有不同的变体。这些函数是 Byte、Word 和 Dword。

```
long  Byte (long ea);           // get a byte at ea
long  Word (long ea);           // get a word (2 bytes) at ea
long  Dword (long ea);          // get a double-word (4 bytes) at ea
```

函数操作失败时返回-1。为了区分失败和值-1，应该调用宏 hasValue。它的原型是：

```
#define hasValue(F)         ((F & FF_IVL) != 0)         // any defined value?
```

如果值未被定义，这个宏将返回 0。

写内存由 patch 系列函数完成。这些函数用于在静态分析时向 IDB 中写入数据，以及在调试器下向虚拟内存写入数据。实际上，patch 函数是在调试器执行期间修改所调试代码的唯一方法。在调试器的 GUI 中只能修改寄存器及区段。修改代码区段或数据区段则需要这些 IDC 函数或插件。

根据所写数据的大小，这个函数有三个变体。分别是 PatchByte、PatchWord、PatchDword。

```
void    PatchByte     (long ea,long value);    // change a byte
void    PatchWord     (long ea,long value);    // change a word (2 bytes)
void    PatchDword    (long ea,long value);    // change a dword (4 bytes)
```

2. 交叉引用

对于数据和代码有不同类型的交叉引用。

● **代码交叉引用**

代码交叉引用由它们的 flowtype 定义。下面是 flowtype 代码的列表：

```
//      Flow types (combine with XREF_USER!):
#define fl_CF   16          // Call Far
#define fl_CN   17          // Call Near
#define fl_JF   18          // Jump Far
#define fl_JN   19          // Jump Near
#define fl_F    21          // Ordinary flow
#define XREF_USER 32        // All user-specified xref types
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
                    // must be combined with this bit
```

用户所创建的引用都应该与 XREF_USER 结合起来。我们在脚本里使用 fl_CN 调用近 flowtype。还有近跳转和远跳转 flowtype。普通的 flowtype 用于连续的指令之间。

增加和删除代码交叉引用的 IDC 函数：

```
void   AddCodeXref(long From,long To,long flowtype);
long   DelCodeXref(long From,long To,int undef);
```

如果这里是对它的最后一个引用，undef 参数将取消 To 地址的定义。

有两组 IDC 函数可以遍历引用。其差异体现在把普通的 flow 认做是交叉引用。第一组将首先返回普通的 flow。

```
long  Rfirst   (long From);              // Get first code xref from 'From'
long  Rnext    (long From,long current);// Get next code xref from
long  RfirstB  (long To);                // Get first code xref to 'To'
long  RnextB   (long To,long current);   // Get next code xref to 'To'
```

函数由一个 first 和一个 next 组成。一般是在循环中使用这两个函数迭代交叉引用。下面演示了这些函数：

```
auto xfAddr, origAddr;
origAddr = ScreenEA();
xfAddr = RfirstB(origAddr);
while (xfAddr != BADADDR)
{
    Message("%x to %x, type == %d\n", xfAddr, origAddr, XrefType());
    xfAddr = RnextB(origAddr, xfAddr);
}
```

这段代码迭代光标所在地址的所有交叉引用，而且还引入了一个新的 IDC 函数 XrefType。它的原型是：

```
long XrefType(void);    // returns type of the last xref
                        // obtained by [RD]first/next[B0]
                        // functions. Return values
                        // are fl _… or dr_…
```

XrefType 返回最后访问的交叉引用的类型。这个函数也可用于数据交叉引用，我们稍后将简单介绍一下。第二组代码交叉引用函数与第一组相对应。

```
long   Rfirst0  (long From);
long   Rnext0   (long From,long current);
long   RfirstB0(long To);
long   RnextB0  (long To,long current);
```

这些函数不返回普通 flow 的交叉引用。

更多资源请点击访问稀.酷客(www.ckook.com)

- **数据交叉引用**

下面是有效的数据类型：

```
//      Data reference types (combine with XREF_USER!):
#define dr_O   1                  // Offset
#define dr_W   2                  // Write
#define dr_R   3                  // Read
#define dr_T   4                  // Text (names in manual operands)
#define dr_I   5                  // Informational
#define XREF_USER 32           // All user-specified xref types
                               // must be combined with this bit
```

同代码交叉引用一样，用户创建的数据交叉引用应该与 XREF_USER 结合起来。数据交叉引用只有一组对应的函数。

```
long    Dfirst     (long From);     // Get first data xref from 'From'
long    Dnext      (long From,long current);
long    DfirstB    (long To);       // Get first data xref to 'To'
long    DnextB     (long To,long current);
```

3. 数据表示

数据表示函数创建结构、函数、数据，定义代码及其他内容。它们是这样的一些 IDC 函数，即能够完成许多在反汇编过程中用手工完成的任务。下面是这些函数的例子。

```
success    MakeArray(long ea,long nitems);
success    MakeByte(long ea);
long       MakeCode(long ea);
success    MakeData(long ea, long flags, long size, long tid);
success MakeDword(long ea);
success MakeFunction(long start,long end);
success MakeStr(long ea,long endea);
success MakeStructEx(long ea,long size, string strname);
```

4. 注释

要成功地开展逆向工程，注释很关键。注释与适当的命名可以作为我们分析的二进制文件的笔记。有一些设置和读取注释的 IDC 函数。

```
// repeatable, 0 = standard, 1 = repeatable
string CommentEx(long ea, long repeatable);
success MakeComm(long ea,string comment);
success MakeRptCmt(long ea,string comment);
long SetBmaskCmt(long enum_id,long bmask,string cmt,long repeatable);
success SetConstCmt(long const_id,string cmt,long repeatable);
success SetEnumCmt(long enum_id,string cmt,long repeatable);
void SetFunctionCmt(long ea, string cmt, long repeatable);
long SetMemberComment(long id,long member_offset,string comment,long repeatable);
```

9

更多资源请点击访问稀.酷客(www.ckook.com)

```
long SetStrucComment(long id,string comment,long repeatable);

long GetBmaskCmt(long enum_id,long bmask,long repeatable);
string GetConstCmt(long const_id,long repeatable);
string GetEnumCmt(long enum_id,long repeatable);
string GetFunctionCmt(long ea, long repeatable);
string GetMarkComment(long slot);
string GetStrucComment(long id,long repeatable);
```

5. 代码遍历

IDA针对代码和数据有不同类型的容器。其中一些容器包括段、函数和指令或数据头。在脚本中迭代不同的容器和区域是很常见的操作。

一些常见的迭代函数是：

```
long NextAddr(long ea);
long NextFunction(long ea);
long NextHead(long ea, long maxea);
long NextNotTail(long ea);
long NextSeg(long ea);

long PrevAddr(long ea);
long PrevFunction(long ea)
long PrevHead(long ea, long minea);
long PrevNotTail(long ea);
```

下面的代码段演示了一些迭代函数。

```
auto currAddr, func, endSeg,funcName, counter;

currAddr = ScreenEA();
func = SegStart(currAddr);
endSeg = SegEnd(currAddr);

counter = 0;
while (func != BADADDR && func < endSeg)
{
        funcName = GetFunctionName(func);
        if (funcName != " ")
        {
                Message("%x: %s\n", func, funcName);
                counter++;
                }
                func = NextFunction(func);
}
Message ("%d functions in segment: %s\n", counter, SegName(currAddr));
```

这个脚本迭代所有属于当前区段的函数。脚本用GetFunctionName调用来测试一个地址是否在函数中。如果地址不是函数的一部分，这个调用将返回一个空字符串。另外，也可以用GetFunctionFlags。这个脚本打印区段里所有函数的地址及名字，并在结尾打印函数的总数。

更多资源请点击访问稀.酷客(www.ckook.com)

6. 输入输出

迄今为止，实际使用的输入输出函数只有 Message 函数。对不同类型的数据，有不同的 IDC 输入函数可供选择。

```
string    AskStr(string defval,string prompt);
string    AskFile(bool forsave,string mask,string prompt);
long      AskAddr(long defval,string prompt);
long      AskLong(long defval,string prompt);
long      AskSeg(long defval,string prompt);
string    AskIdent(string defval,string prompt);
long      AskYN(long defval,string prompt);
```

前面的函数检索用户输入。下面的代码段演示 AskYN IDC 函数。

```
auto answer;
answer = AskYN(1, "hello");
if (answer == 1)
    Message("YES\n");
else if (answer == 0)
    Message("NO\n");
else
    Message("CANCEL\n");
```

也有用于文件 I/O 的 IDC 函数。这些文件 I/O 函数与 C 语言中的对应者非常类似。

```
long fopen(string file,string mode);
long fseek(long handle,long offset,long origin);
void fclose(long handle);

long fgetc(long handle);
long fprintf(long handle,string format,…);
long fputc(long byte,long handle);
long ftell(long handle);

long writelong(long handle,long dword,long mostfirst);
long writeshort(long handle,long word,long mostfirst);
long writestr(long handle,string str);

long readlong(long handle,long mostfirst);
long readshort(long handle,long mostfirst);
string readstr(long handle);
```

9.6 IDA 插件基础

可以通过模块扩展 IDA Pro，可以为 IDA 开发不同类型的模块。插件也是模块的一种，可以用于扩展 IDA。有时候，术语“插件”被错误地用于表示所有可扩展的模块。

IDA 中有不同类型的模块可用。模块类型依赖于必需的功能。分类如下：

- 插件；

更多资源请点击访问稀.酷客(www.ckook.com)

- 装载器；
- 处理器；
- 调试器。

9.6.1　模块/插件资源

SDK中有许多具有充分源代码的模块/插件。

- Hex-Rays向注册用户提供SDK。在购买IDA时，光盘中会包含SDK，大家也可以在Hex-Rays的网站(http://www.hex-rays.com/idapro/idadown.htm)上下载SDK。尽管说是支持，但Hex-Rays并不为SDK提供正式支持。
- Hex-Rays公告板提供了与插件相关的帮助信息，主要是Ilfak或其他用户提供的(http://www. hex-rays. com/forum/)。
- 与插件开发有关的信息不太多。Steve Micallef写了一部精彩的教程IDA PLUG-IN WRITING IN C/C++ (http://binarypool.com/idapluginwriting/)。
- Hex-Rays在SDK中包括许多带源代码的插件。其中有一个插件，即一个通用的脱壳程序，在Hex-Rays的文章中有介绍(http://www.hex-rays.com/idapro/unpack_pe/unpacking.pdf)。
- Ilfak在他的博客上还提供各种带有完整源代码的插件(http://hexblog.com)。
- 对逆向工程来说，OpenRCE是一个很有价值的资源。其中某部分下载站专门提供IDA Pro的插件下载（http://www.openrce.org/downloads/browse/IDA_Plugins)。

插件是用C++写成的，支持多种编译器和开发环境。SDK为下面的环境提供了导入库：

- Visual C++ (32位和64位)；
- Borland C++ Builder (32位和64位)；
- GCC C++ Compiler；
- Windows (32位和64位)；
- Linux (32位和64位)；
- Mac OSX (32位和64位)。

本章的重点是32位Windows插件，使用Microsoft Visual Studio2005/2008编译器。

注意

针对其他开发环境的说明位于SDK的根目录下。
install_cb.txt包含CBuilder设置说明。
install_mac.txt包含OS X GCC设置说明。
install_linux.txt包含Linux GCC设置说明。

处理器模块增加了对不同CPU和体系结构的支持。处理器模块位于procs目录下。这些模块解释操作码并生成我们在IDA中看到的反汇编。

处理器模块使用下面的文件扩展名。

更多资源请点击访问稀.酷客(www.ckook.com)

- w32 windows
- w64 windows 64
- ilx Linux
- ilx64 Linux 64
- imc OS X
- imc64 OS X 64

加载器与操作系统加载器的操作类似。加载器解析可执行文件、创建区段，确定区段是代码还是数据。IDA 包括多种可执行文件的加载器，包括 PE 和 ELF。

加载器模块使用下列文件扩展名。

- ldw windows
- l64 windows 64
- llx Linux
- llx64 linux 64
- lmc OS X
- lmc64 OS X 64

调试器模块是可以与 IDA 交互操作的完全调试器。不应该把它们与内置调试器模块一起工作的标准插件混为一谈。包含源代码的调试器模块文档位于\plugins\debugger\目录下的 SDK 里。

标准插件包括除处理器或加载器模块以外的所有部分。我们通常就把它们称为插件。这些插件对反汇编进行操作，是最常见的插件类型，我们随后将介绍它们。

标准插件使用下面的文件扩展名。

- plw windows
- p64 windows 64
- plx Linux
- plx64 Linux 64
- pmc OS X
- pmc64 macosx64

9.6.2 IDA Pro SDK 介绍

Hex-Rays 从版本 4.9 开始就冻结 SDK 了。这对我们意味着什么？在以前，不同版本的 SDK 差异很大。对于每个 SDK，插件都需要被重新编译，因为它们的二进制之间不兼容。但现在除了增加的新功能外，我们不再担心不同版本间的 SDK 会出现重大改变。

IDA Pro 光盘中包括 SDK，你也可以直接从 Hex-Rays 网站上下载 SDK（http://www.hex-rays.com/idapro/idadown.htm）。

SDK 是 zip 文件，最新版是 idasdk52。我习惯把不同版本的 SDK 放在不同的目录里，因此我把这个文件解压到\SDK\idasdk52 目录下。

更多资源请点击访问稀.酷客(www.ckook.com)

警告　SDK 可能包含一个使编译出错的 bug。这个 bug 在\include 目录下的 intel.hpp 文件中。其中的一个#include 列表是错误的。把

```
#include "../idaidp.hpp"
```

改为

```
#include "../module/idaidp.hpp"
```

这个 bug 在最新的版本 5.2 中仍然存在。

SDK布局

SDK 包含很多不同的目录。这些目录包括 include 文件、导入库、工具和源代码。下面是较重要目录的一个概览：

include	SDK 所有的头文件
ldr	几个加载器的源代码
libbor.w32	Borland 的 32 位 Windows 平台插件
libbor.w64	Borland 的 64 位 Windows 平台插件
libgcc.w32	GCC 的 32 位 Windows 平台插件
libgcc.w64	GCC 的 64 位 Windows 平台插件
libgcc32.lnx	GCC 的 32 位 Linux 平台插件
libgcc32.mac	GCC 的 32 位 OS X 平台插件
libgcc64.lnx	GCC 的 64 位 Linux 平台插件
libgcc64.mac	GCC 的 64 位 OS X 平台插件
libvc.w32	Visual Studio 的 32 位 Windows 平台插件
libvc.w64	Visual Studio 的 64 位 Windows 平台插件
module	几种处理器模块的源代码
plugins	示例插件和真实插件的源代码

9.7　插件语法

插件是可加载的库、DLL 或其他 IDA 在需要时加载的库。插件必须有确定的结构输出，而结构类型取决于插件类型。本节将介绍标准插件，因为它们是最常见的类型。从现在起，我们所说的插件都是指标准插件。任何与加载器模块或处理器模块相关的细节都会特别标明。

IDA 插件是用 C++写的，它输出插件结构——PLUGIN_t。

```
plugin_t PLUGIN =
{
  IDP_INTERFACE_VERSION,
  plugin_flags,      // plugin flags
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
    init,                // initialize
    term,                // terminate. this pointer may be NULL.
    run,                 // invoke plugin
    comment,             // plugin comment
    help,                // multiline help about the plugin
    wanted_name,         // the preferred short name of the plugin
    wanted_hotkey        // the preferred hotkey to run the plugin
};
```

这个结构包含常量、函数指针、字符串指针。

IDP_INTERFACE_VERSION 是一个在 SDK 文件中定义的常量。在 4.9 版 SDK 冻结前，这个值会随着每个发行版本递增。在 4.9 之后这个值就保持不变了。

Plugin_flags 定义了插件怎样同 IDA 交互操作。不同的标志在 loader.hpp 中都有描述。当调试一个插件时，这个字段通常被设成 0 或 PLUGIN_UNL。

后面的三项，即 init、term、run 是函数指针。

当插件被加载时，将执行 init 函数。它主要是确定这个插件是否可用于当前的数据库。插件可以是针对处理器或文件格式的。另外，一旦 run 被执行，这个函数可以为插件设置环境。

init 函数需要返回下述内容之一。

- **PLUGIN_SKIP** 它通知 IDA 不要加载此插件。当体系结构或文件格式不合适的时候，插件通常返回此值。比如，

```
if (inf.filetype != f_PE)
    return PLUGIN_SKIP; // not a PE file
```

- **PLUGIN_OK** 它通知 IDA 这个插件是合适的，IDA 将在首次使用时装入这个插件。
- **PLUGIN_KEEP** 它通知 IDA 插件是合适的，并且让该插件驻留在内存中。

当 IDA 被终止时，执行 term 函数。这个函数可以用来清除插件生存期占用的资源。许多插件都把这个指针设为 NULL。

run 函数通过运行插件来执行。这个函数可以带参数。参数在位于 plugin 目录下的 plugin.cfg 文件中定义。许多插件都使用 run 函数完成所有必须的工作，另一些插件用 run 设置回调函数。SDK 中的调试功能由回调函数处理。

- Comment 是一个指针，是对插件进行简要描述的 short 型字符串。
- Help 也是一个指向描述插件的字符串的指针。但与 Comment 字符串不同的是，Help 通常是多行描述。
- Wanted_name 是显示在插件列表上的名字，File | Edit | Plugins 可显示插件列表。
- Wanted hotkey 为运行这个插件设置热键。这个热键可以在 plugins.cfg 中覆写。

当前 IDA 没有使用 comment 和 help 字段，但这点在将来可能会发生变化。

9.8 设置开发环境

本节介绍在 Visual Studio 2005 和 Visual Studio 2008 下设置开发环境。用于其他平台的构建

更多资源请点击访问稀.酷客(www.ckook.com)

指令在 SDK 目录下可以找到。

设置开发环境挺乏味的。编写插件最简单的方法是使用 IDA Pro 插件向导，这个向导与 Visual Studio 2005、Visual Studio 2008 兼容它用于合理配置编译器和连接器选项。

IDA Pro 插件向导可在 http://ringzero.net/re 下载。这个向导与下面的环境兼容。

- Visual Studio 2008
- Visual Studio 2005
- Visual C++ 2008 Express Edition
- Visual C++ 2005 Express Edition

工具和陷阱

在Linux下构建插件

在 Linux 下设置合适的构建环境可能有些复杂。下面的 makefile 可以用来构建插件。注意，命令行必须以标签开头。在 all、install 和 clean 标记后面的就是命令行。

```
# Makefile for IDA Pro Plugins under Linux
# Updated version of makefile from Steve Micallef's
# IDA Plugin Writing Tutorial
# http://www.binarypool.com/idapluginwriting/

# Set your plugin name here. PLUGINNAME.plx
PLUGINNAME=myplugin

# Set your IDA install directory
IDABASEDIR=/usr/local/idaadv

# Set your IDA SDK directory
SDKBASEDIR=/usr/local/idaadv/sdk

# Compiles all cpp files in current dir
SRC=$(wildcard *.cpp)
OBJS=$(SRC:.cpp=.o)
CC=g++
LD=g++
CFLAGS=-D__IDP__ -D__PLUGIN__ -c -D__LINUX__ \
    -I$(SDKBASEDIR)/include $(SRC)
LDFLAGS=-shared $(OBJS) -L$(IDABASEDIR) -lida -no-undefined \
    -Wl,-version-script=$(SDKBASEDIR)/plugins/plugin.script

all:
	$(CC) $(CFLAGS)
	$(LD) $(LDFLAGS) -o $(PLUGINNAME).plx

install:
	cp $(PLUGINNAME).plx $(IDABASEDIR)/plugins
clean:
	-rm -f *.plx *.o core

rebuild: clean all
```

更多资源请点击访问稀.酷客(www.ckook.com)

9.9 简单插件示例

设置好开发环境就可以写插件了，我们首先写一个简单的“hello world”插件。这样一来既可以测试一下开发环境，又可以验证 IDA 是否可正确地加载和执行插件。find memcpy 插件将演示一些 IDA API，包括指令解码以及一些 UI 代码。

9.9.1 Hello World 插件

启动 Visual Studio，选择 IDA 插件向导。输入项目名并点击 **OK**。图 9-12 显示了在 Visual C++ 2008 Express 版中的选择项。

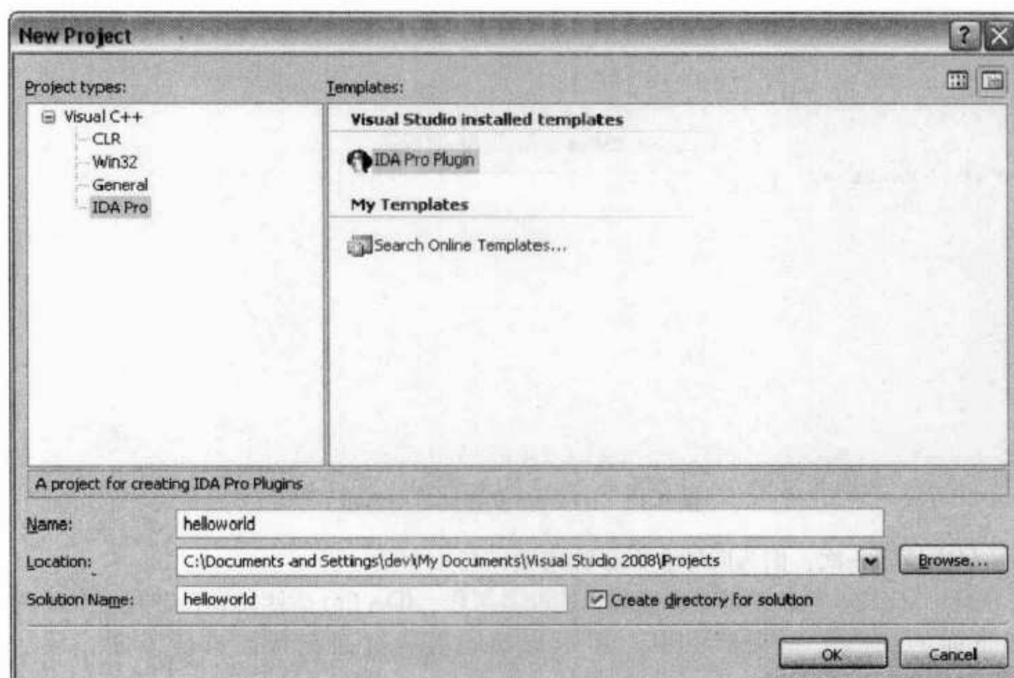


图 9-12 选择 IDA 插件向导

向导如图 9-13 所示。插件类型默认就是插件，也就是我们正在构建的。Name of Author（作者名）字段是可选的，如果有的话将出现在头部注释处。

构建插件需要知道 SDK Path（SDK 路径）。点击按钮弹出文件浏览器对话框。确保选择的是 SDK 目录中的库。

最后一项虽然是可选的，但非常有用，它会在项目属性里创建一个构建后事件，此事件把插件复制到合适的 IDA Pro 目录。点击按钮弹出文件浏览器对话框。确保选择了 IDA Pro 安装目录中的库。

更多资源请点击访问稀.酷客(www.ckook.com)

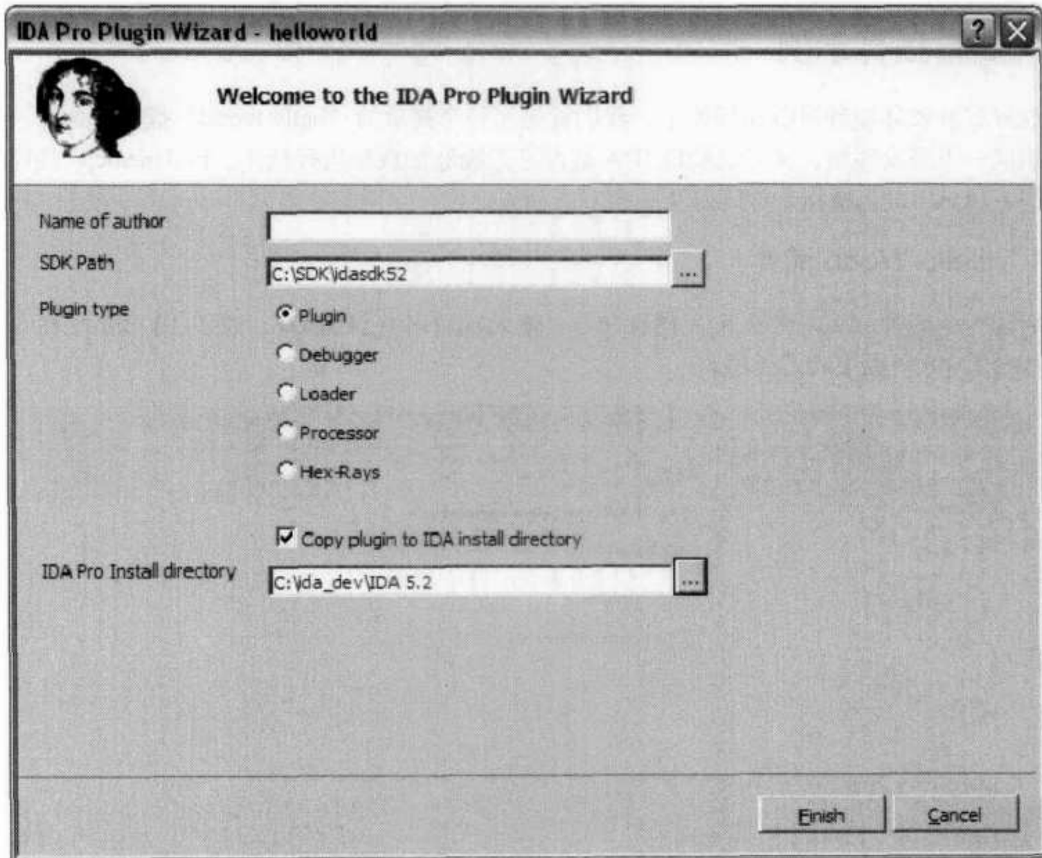


图9-13　IDA Pro插件向导对话框

路径只需填写一次，因为向导会保存该选项。

点Finish（完成）后，向导将完成项目的准备工作。IDA Pro插件向导会带有一个示例模板。可以把下面的代码复制到模板中。用于启动的组合键因配置而有所不同，缺省的是**CTRL+SHIFT+B**。

```
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>

// Determine if the plugin is suitable. Return:
// PLUGIN_SKIP - plugin not suitable, wont be used
// PLUGIN_KEEP       - plugin is suitable. keep in memory
// PLUGIN_OK - plugin is suitable, load when used
int init(void)
{
    return PLUGIN_OK;
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
}
// plugin termination function. Unhook any notification points.
void term(void)
{
    return;
}
// This function is called when the plugin is executed.
// The arg is configured in the plugin.cfg file.
void run(int arg)
{
    msg("Hello world! my address is %a\n", get_screen_ea());
    return;
}
char comment[] = "hello world";
char help[] = "hello world";

// Name of plugin in ( Edit | Plugins )
// An entry in plugins.cfg can overide this field.
char wanted_name[] = "hello world";
// Plugin's hotkey
// An entry in plugins.cfg can overide this field.
char wanted_hotkey[] = " ";
// PLUGIN DESCRIPTION BLOCK
plugin_t PLUGIN =
{
IDP_INTERFACE_VERSION,
PLUGIN_UNL,          // plugin flags
init,                // initialize
term,                // terminate. this pointer may be NULL.
run,                 // invoke plugin
comment,             // comment about the plugin
help,                // multiline help about the plugin
wanted_name,         // the preferred short name of the plugin
wanted_hotkey        // the preferred hotkey to run the plugin
};
```

前面的代码等同于一个 hello world 程序，它实现了 plugin_t 结构所体现的关键功能。插件使用了一个标志——PLUGIN_UNL。这个标志常在调试插件时使用。它被定义成：

```
#define PLUGIN_UNL 0x0008       // Unload the plugin immediately after
                                // calling 'run'.
                                // This flag may be set anytime.
                                // The kernel checks it after each
                                // call to 'run'
                                // The main purpose of this flag is to ease
                                // the debugging of new plugins.
```


更多资源请点击访问稀.酷客(www.ckook.com)

在 run 命令执行后，插件将被卸载，这使我们可以做一些更改、重新编译工作，而且可以将插件复制到插件目录。如果没有设置这个标志，将需要重新启动 IDA，因为它持有打开的插件文件句柄。一个工作区会短暂出现。注意，卸载只有在执行了 run 命令后才发生。如果 init 函数返回 PLUGIN_KEEP，插件就会保留在内存中，直到 run 函数执行后才会被卸载。不过，如果 init 函数返回 PLUGIN_OK，插件只会在第一次使用时加载。

警告　如果 init 函数设置了回调，它必定返回 PLUGIN_KEEP；否则，使用的内存地址可能会变为无效，因为插件可能会在不同的地址加载。

插件的 run 函数输出一条包含“hello world”和当前地址的消息。使用的 IDA API 调用是 get_screen_ea。这个函数等同于在本章前面使用的 IDC 函数 ScreenEA。这个 IDA API 包含了许多与 IDC 函数等价的函数以及 IDC 中没有带的功能。Hello world 插件验证了我们有一个可以正常工作的开发环境。

9.9.2　find memcpy 插件

有了可正常工作的开发环境，我们就可以编写更有用的插件，同时引入一些新的 IDA API 调用。这个插件搜索内联 memcpy（参见图 9-14）。编译器在优化时通常会内联库调用。memcpy 通常会与 strlen 和 strcpy 等 string 函数一起内联。

图 9-14 中的汇编程序使用 movsd 和 movsb 指令复制数据。mov 指令操作 edi 和 esi 寄存器。esi 存储源地址，edi 存储目的地址。movsd 复制 dword（4 个字节），而 movsb 复制单字节。

```
                    ; memcpy (edi, esi, eax)
.text:00418ECC      mov ecx, eax    ; copy eax into ecx
.text:00418ECE      shr ecx, 2      ; shift ecx right by 2 (divide by 4)
.text:00418ECE                      ; ecx = number of dwords to copy
.text:00418ED1      rep movsd       ; copy dwords from esi to edi
.text:00418ED3      mov ecx, eax    ; copy eax into ecx
.text:00418ED5      and ecx, 3      ; and ecx by 3
.text:00418ED5                      ; ecx = number of bytes to copy
.text:00418ED5                      ; (remaining bytes)
.text:00418ED8      rep movsb       ; copy bytes from esi to edi
```

图 9-14　内联 memcpy

rep 是指令前缀，它使 mov 指令重复执行。每次 mov 执行，ecx 就会递减，而一旦 ecx 递减到 0，mov 指令就停止执行。（参见图 9-15）

eax 包含要复制的字节数。shr（右移）指令将 ecx 指令除以 4，计算需复制的 dword 数。rep movsd 指令把 ecx dword 从 esi 复制到 edi。在复制 dword 后，最后可能会剩 0 到 3 个字

更多资源请点击访问稀.酷客(www.ckook.com)

节要复制。and 指令计算剩下的要 rep movsb 复制的字节。

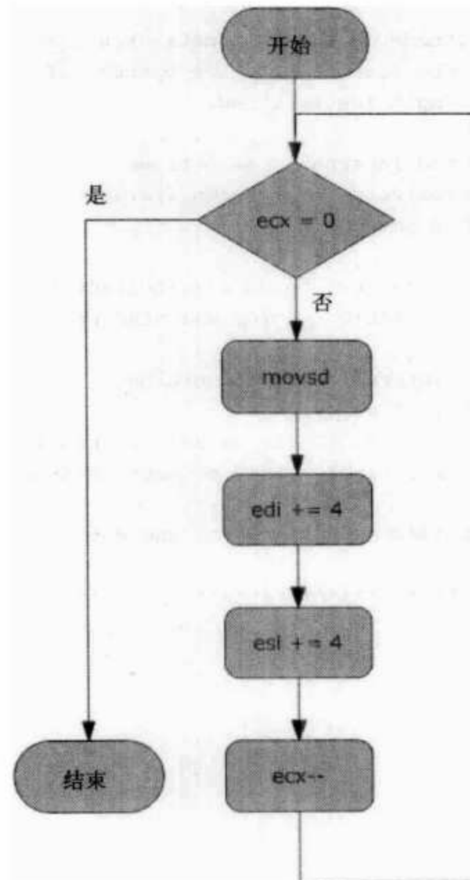


图 9-15 rep movsd 流程图

图 9-16 中的插件描述了一个查找这些代码构造类型的方法。

```
/**********************************************************************
* Find memcpy() IDA Pro plugin
*
* Copyright (c) 2008 Luis Miras
* Licensed under the BSD License
*
* Requirements: The plugin requires x86 processor.
*
* Description: The plugin searches for rep movsd/rep movsb
*              pairs identidying them as memcpy()
*              Single rep movsd and rep movsb instructions
```

图 9-16 Find Memcpy 插件

更多资源请点击访问稀.酷客(www.ckook.com)

```
 *              are also recorded
 *
 * Data structures: a netnode is the main data structure.
 *                  movsobj_t represents the either pairs
 *                  or single instructions.
 *
 * netnodes are implemented internally as B-trees.
 * IDA uses netnodes extensively for its own storage.
 * netnodes are defined in netnode.hpp.
 *
 * netnodes in the plugin: calls - holds all indirect calls
 *                         vtable - holds all vtables
 *
 * netnodes have various internal data structures.
 * The plugin uses 2 types of arrays:
 *    altval - a sparce array of 32 bit values, initially set to 0.
 *    supval - an array of variable sized objects (MAXSPECSIZE)
 *
 * The plugin holds base addresses in altval and movsobj_t objects
 * in supval
 ******************************************************************/
#include <ida.hpp>
#include <idp.hpp>
#include <loader.hpp>
#include <allins.hpp>
#include <intel.hpp>

#define NODE_COUNT -1

struct movsObj {
   ea_t movsDW; // addr of rep movsd. BADADDR if none
   ea_t movsBT; // addr of rep movsd. BADADDR if none
};

typedef movsObj movsobj_t;
static const char* header[] = {"Address", "Type", "Movsd/b distance"};
static const int widths[] = { 16, 25, 25};
char window_title[] = "Inline memcpy" ;
/*********************************************************************
 * Function: processMemcpy
 *
 * This function determines the types of memcpy based on the movsobj_t
 * and calculates distance between rep movsd and rep movsb
 ********************************************************************/
```

图 9-16 (续)

更多资源请点击访问稀.酷客(www.ckook.com)

```
char* processMemcpy(movsobj_t* my_movs, ea_t* movs_distance)
{
    if (my_movs->movsDW == BADADDR)
    {
        *movs_distance = BADADDR;
        return "memcpy movsb only";
    }
    else if (my_movs-> movsBT == BADADDR)
    {
        *movs_distance = BADADDR;
        return "memcpy movsd only";
    }
    else
    {
        *movs_distance = my_movs-> movsBT - my_movs->movsDW;
        return "memcpy()";
    }
}
/**************************************************************************
* Function: description
*
* This is a standard callback in the choose2() SDK call. This function
* fills in all column content for a specific line. Headers names are
* set during the first call to this function, when n == 0.
* arrptr is a char* array to the column content for a line.
*                arrptr[number of columns]
*
* description creates 3 columns based on the header array
**************************************************************************/
void idaapi description(void *obj,ulong n,char * const *arrptr)
{
     netnode *node = (netnode *)obj;
     movsobj_t my_movs;
     char* outstring = NULL;
     ea_t movs_distance;

     if ( n == 0 ) // sets up headers
     {
        for ( int i=0; i < qnumber(header); i++ )
          qstrncpy(arrptr[i], header[i], MAXSTR);
        return;
     }
     // list empty?
     if (!node->altval(NODE_COUNT) )
```

图 9-16 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
            return;
        node->supval(n-1, &my_movs,sizeof(my_movs));
        outstring = processMemcpy(&my_movs, &movs_distance);
        qsnprintf(arrptr[0], MAXSTR, "%08a", node->altval(n-1));
        qsnprintf(arrptr[1], MAXSTR, "%s", outstring);

        if (movs_distance != BADADDR)
        {
           qsnprintf(arrptr[2], MAXSTR, "%02x", movs_distance);
        }
        else
        {
           qsnprintf(arrptr[2], MAXSTR, " ");
        }
        return;
}
/*************************************************************************
* Function: enter
*
* This is a standard callback in the choose2() SDK call. This function
* is called when the user pressed Enter or Double-Clicks on a line in
* the chooser list.
*************************************************************************/
void idaapi enter(void * obj,ulong n)
{
        ea_t addr;
        netnode *node = (netnode *)obj;
        addr = node->altval(n-1);
        jumpto(addr);
        return;
}
/*************************************************************************
* Function: destroy
*
* This is a standard callback in the choose2() SDK call. This function
* is called when the chooser list is being destroyed. Resource cleanup
* is common in this function. The netnode deleted here.
*************************************************************************/
void idaapi destroy(void* obj)
{
        netnode *node = (netnode *)obj;
        node->kill();
        return;
}
```

图 9-16　（续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
/*********************************************************************
* Function: size
*
* This is a standard callback in the choose2() SDK call. This function
* returns the number of lines to be used in the chooser list.
*********************************************************************/
ulong idaapi size(void* obj)
{
      ulong mysize;
      netnode *node = (netnode *)obj;
      mysize = node->altval(NODE_COUNT);
      return mysize;
}
/*********************************************************************
* Function: functionSearch
*
* functionSearch looks through functions for rep movsd and rep movsb
* memcpy is defined as a rep movsd followed by rep movsb
* single rep movsd and movsb are also recorded
*
* last_movs is used to track for rep movsd/rep movsb sets
* the netnode's alval and supval arrays are used
* node->alset contains the base address
* node->supset contains a movsobj_t object
*
*
* memcpy() == movsobj_t with {addr, addr}
* mosvd only == movsobj_t with {addr, BADADDR}
* movsb only == movsobj_t with {BADADDR, addr}
*
* NOTE: this function misses rep movw (66 F3 A5) instructions
*********************************************************************/
void functionSearch(func_t* funcAddr, netnode* node)
{
      movsobj_t my_movs;
      int counter = node->altval(NODE_COUNT);
      ea_t last_movs = BADADDR;
      ea_t addr = funcAddr->startEA;

      while (addr != BADADDR)
      {
         flags_t flags = getFlags(addr);
         if (isHead(flags) && isCode(flags) )
         {
```

图 9-16 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
            // fill cmd, only looking for 2 byte instructions
            if (ua_ana0(addr) == 2)
            {
               if ( (cmd.auxpref & aux_rep) && (cmd.itype == NN_movs))
               {
                  if (cmd.Operands[1].dtyp == dt_dword) // rep movsd
                  {
                     if (last_movs != BADADDR)
                     {
                         // two consecutive rep movsd
                         // set the previous one to movsd only
                         my_movs. movsDW = last_movs;
                         my_movs. movsBT = BADADDR;
                         node->altset(counter, last_movs);
                         node->supset(counter++, &my_movs,sizeof(my_movs));
                     }
                     // found a rep movsd waiting for rep movsb
                     last_movs = cmd.ea;
                 }
                 else if (cmd.Operands[1].dtyp == dt_byte) // rep movsb
                 {
                    if (last_movs == BADADDR)
                    {
                       // rep movsb with no preceding rep movsd
                       my_movs. movsDW = BADADDR;
                       my_movs. movsBT = cmd.ea;
                       node->altset(counter, cmd.ea);
                       node->supset(counter++,&my_movs,sizeof(my_movs));
                    }
                    else // memcpy()
                    {
                       // complete set rep movsd/rep movsb
                       my_movs. movsDW = last_movs;
                       my_movs. movsBT = cmd.ea;
                       node->altset(counter, last_movs);
                       node->supset(counter++, &my_movs,sizeof(my_movs));
                    }
                    last_movs = BADADDR;
                 }
                 else
                 {
                    msg("%x: rep", addr);
                    msg("ERROR !!!\n");
                 }
```

图 9-16 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
                    }
                }
            }
            addr = next_head(addr, funcAddr->endEA);
}
if (last_movs != BADADDR)
{
    // a remaining single rep movsd
    my_movs. movsDW = last_movs;
    my_movs. movsBT = BADADDR;
        node->altset(counter, last_movs);
        node->supset(counter++, &my_movs, sizeof(my_movs));
    }
    node->altset(NODE_COUNT, counter);
    return;
}
/**************************************************************************
* Function: collectData
*
* This function iterates through all functions calling functionSearch
/**************************************************************************
void collectData(netnode* node)
{
     for (uint i = 0; i < get_func_qty(); ++i)
     {
         func_t *f = getn_func(i);
         functionSearch(f, node);
     }
     return;
}
/***************************************************************************
Function: init
*
* init is a plugin_t function. It is executed when the plugin is
* initially loaded by IDA
*************************************************************************/
int init(void)
{
     // plugin only works for x86 executables
     if (ph.id != PLFM_386)
         return PLUGIN_SKIP;
     return PLUGIN_OK;
}
/**************************************************************************
* Function: term
*
```

图 9-16 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
* term is a plugin_t function. It is executed when the plugin is
* unloading. Typically cleanup code is executed here.
* The window is closed to remove the choose2() callbacks
*************************************************************************/
void term(void)
{
     close_chooser(window_title);
     return;
}
/*************************************************************************
* Function: run
*
* run is a plugin_t function. It is executed when the plugin is run.
* This function collects data and and displays results
*
*      arg - defaults to 0. It can be set by a plugins.cfg entry. In this
*            case the arg is used for debugging/development purposes
* ;plugin displayed name    filename      hotkey     arg
* find_memcpy               findMemcpy    Ctrl-F12   0
* find_memcpy_unload        findMemcpy    Shift-F12  415
*
* Thus Shift-F12 runs the plugin with an option that will unload it.
* This allows (edit/recompile/copy) cycles.
*************************************************************************\
void run(int arg)
{
     char node_name[] = "$ inline memcpy";

     if(arg == 415)
     {
          PLUGIN.flags |= PLUGIN_UNL;
          msg("Unloading plugin ...\n");
          return;
     }
     netnode* node = new netnode;
     if(close_chooser(window_title))
     {
          //window existed and is now closed
          msg("window existed and is now closed\n");
     }
     if (node->create(node_name) == 0)
     {
          msg("ERROR: creating netnode %\n", node_name);
          return;
     }
     // set netnode count to 0
```

图9-16　（续）

更多资源请点击访问稀酷客(www.ckook.com)

```
        node->altset(NODE_COUNT, 0);

        // look for memcpys

        collectData(node);
        // create chooser list box
        choose2(false,          // non-modal window
           -1, -1, -1, -1,      // position is determined by Windows
           node,                // object to show
           qnumber(header),     // number of columns
           widths,              // widths of columns
           size,                // function that returns number of lines
           description,         // function that generates a line
           window_title,        // window title
           -1,                  // use the default icon for the window
           0,                   // position the cursor on the first line
           NULL,                // "kill" callback
           NULL,                // "new" callback
           NULL,                // "update" callback
           NULL,                // "edit" callback
           enter,               // function to call when the user pressed Enter
           destroy,             // function to call when the window is closed
           NULL,                // use default popup menu items
           NULL);               // use the same icon for all line
           return;
}
char comment[]    = "findMemcpy - finds inline memcpy";
char help[]       = "findMemcpy\n"
                    "This plugin looks through all functions\n"
                    "for inline memcpy\n";
char wanted_name[] = "findMemcpy";
char wanted_hotkey[] = " ";

/* defines the plugins interface to IDA */
plugin_t PLUGIN =
{
      IDP_INTERFACE_VERSION,
      0,                // plugin flags
      init,             // initialize
      term,             // terminate. this pointer may be NULL.
      run,              // invoke plugin
      comment,          // comment about the plugin
      help,             // multiline help about the plugin
      wanted_name,      // the preferred short name of the plugin
      wanted_hotkey     // the preferred hotkey to run the plugin
};
```

图 9-16 (续)

更多资源请点击访问稀.酷客(www.ckook.com)

编译并运行这个插件。我们不再受限于消息窗口了，findMemcpy 打开一个如图 9-17 所示的选择列表框。构建的功能都在列表框中，列表可以按任意列排序。在某行上点击将会跳到反汇编视图里的内存地址。

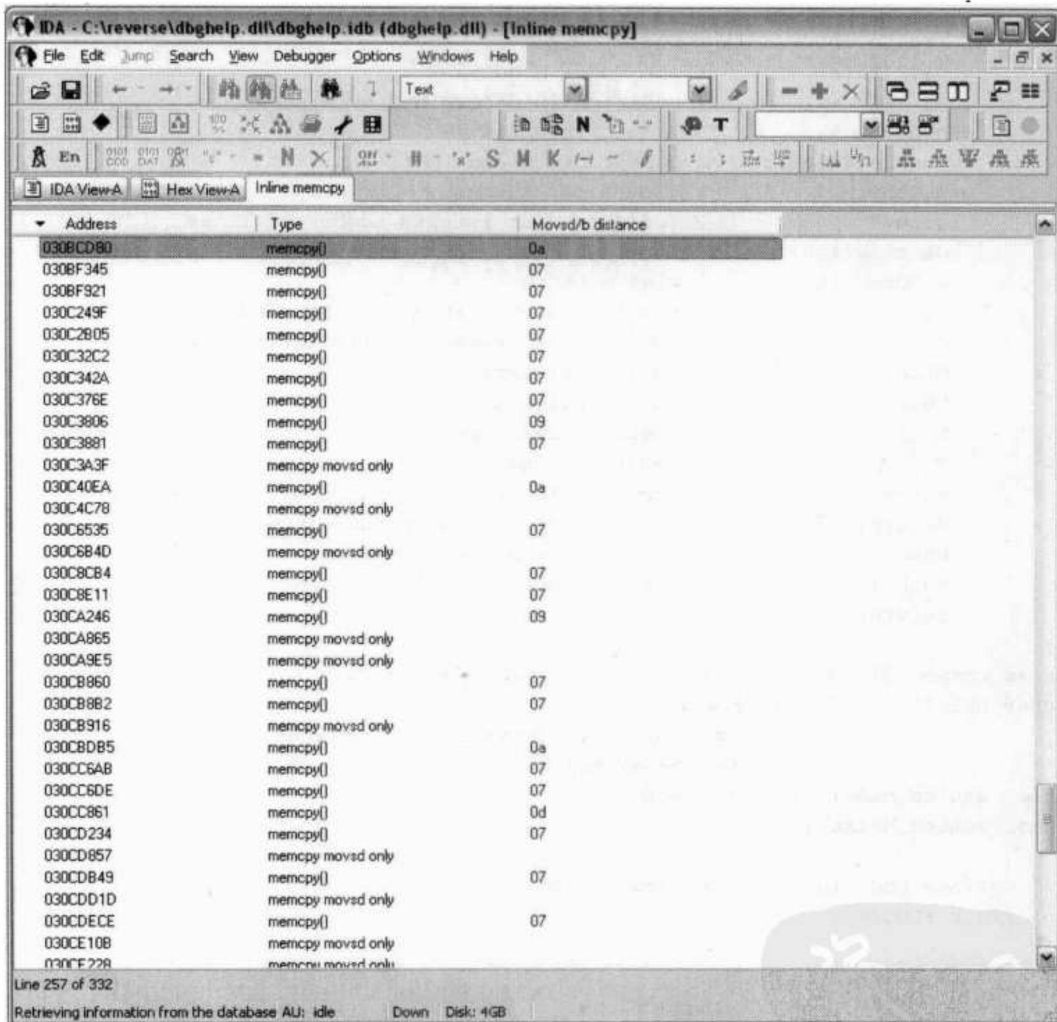


图 9-17 Find Memcpy 结果

这个插件新引入了一些 IDA API 功能，最明显的是列表框。不过，这个插件也引入了 IDA 的一个构建数据类型。IDA 使用 netnode 类存储内部信息。

netnode 是什么？netnode 是在 netnode.hpp 中定义的，其内部实现是一个 B 树。netnode 存储在数据库中，因此可以为 idb 提供永久存储。这个插件去掉了 netnode，因为它不需要永久保存功能。此插件和本章后面要介绍的 indirectCall 插件使用了 netnode 里的两个类型，这两个类

更多资源请点击访问稀.酷客(www.ckook.com)

型是 altval 和 supval。

类 型	描 述
altval	这是一个存储32位值的稀疏数组。alvtal常和地址一起被当做键使用。与键绑定的值就被用做supval数组的索引
supval	这是一个可变大小对象的数组（最大值为以1 024字节计数的MAXSPECSIZE）

这个插件使用 supval 数组保存 movsobj_t 对象。每个 movsobj_t 代表一整个 memcpy 或部分 memcpy。部分的 memcpy 是指单个 rep movsd 或 rep movsb。

```
struct movsObj {
    ea_t movsDW;        // addr of rep movsd. BADADDR if none
    ea_t movsBT;        // addr of rep movsd. BADADDR if none
};
typedef movsObj movsobj_t;
```

如果遇到单个或不匹配的 rep movsd，缺失项的地址就被记录为 BADADDR。altval 被用做一个标准数组，其值是要显示的基址。除单个 rep movsb 外，这个基址是 rep movsd 的地址。

altval 在索引 NODE_COUNT(-1)中保存数组计数。在索引-1 中保存数组计数在其他插件中也很常见。

介绍了数据类型，我们再看看 run 函数，netnode 就是在这个函数里创建的，推荐在 netnode.hpp 中给 netnode 对象名字加上前缀 ‘$’。IDA 中不能使用位置名，因为它 IDA 以位置名命名 netnode。

这个插件使用了带参数的 run 函数。参数在 plugin 目录下的 plugins.cfg 文件中定义。把下面的内容加到 plugins.cfg 末尾：

```
find_memcpy            findMemcpy      Ctrl-F12        0
find_memcpy_unload     findMemcpy      Shift-F12       415
```

因为插件使用了回调函数，它在执行 run 后不能卸载自己。为了卸载插件，需要增加附加选项。如果收到正确的参数，插件标志就会被修改以支持插件卸载。卸载允许我们编译和复制一个新的插件版本，而不必关闭和重启 IDA。在 9.11 节会包含更多的调试技术。

```
if(arg == 415)
{
    PLUGIN.flags |= PLUGIN_UNL;
    msg("Unloading plugin ...\n");
    return;
}
```

1. 收集数据

run 函数的剩余部分会调用两个函数，即 colletData 和 choose2 API 函数，后者创建列表框。collectData 函数调用 functionSearch 来迭代所有函数，但必须引入两个新的 API 调用。新函数在 funcs.hpp 中定义。

更多资源请点击访问稀.酷客(www.ckook.com)

```
// Get pointer to function structure by number
//      n - number of function, is in range 0..get_func_qty()-1
// Returns ptr to a function or NULL
idaman func_t *ida_export getn_func(size_t n);
// Get total number of functions in the program
idaman size_t ida_export get_func_qty(void);
```

真正的工作由 functionSearch 完成了。这个函数看上去与我们在 IDC 中看到的相似。迭代所有区域在脚本和插件中很常见。while 循环迭代函数中定义的所有项。

```
// Get start of next defined item. Return BADADDR if none exist.
// maxea is not included in the search range
idaman ea_t ida_export next_head(ea_t ea, ea_t maxea);
```

第一个 if 语句定义了我们正在寻找的指令。接下来的一个 if 语句包含一个新的 API 调用。

```
// Analyze the specified address and fill 'cmd'
// This function does not modify the database
// Returns the length of the (possible) instruction or 0
idaman int ida_export ua_ana0(ea_t ea);
```

ua_ana0 函数是一系列在 ua.hpp 中定义的分析指令的函数中的一个。ua_ana0 是其中最小的函数，因为它只分析地址，而不修改数据库。分析进入‘cmd’，它保存有指令信息。

```
idaman insn_t ida_export_data cmd; // current instruction
```

insn_t 类型实际上是一个类，它保存普通的和特定于处理器的指令信息。rep movsd 和 rep movsb 都是满足 if 语句的双字节指令。下面的代码行使用不同的 cmd 属性。

```
01 if ((cmd.auxpref & aux_rep) && (cmd.itype == NN_movs) )
02 {
03    if (cmd.Operands[1].dtyp == dt_dword) // rep movsd
04    {
05      // removed for now
06    }
07    else if (cmd.Operands[1].dtyp == dt_byte) // rep movsb
08    {
09      // removed for now
```

第一行查看被分析指令的 auxpref 和 itype 属性。itype 变量是表示指令的助词符。

```
ushort itype;  // instruction code (see ins.hpp)
```

文件 ins.hpp 没有被使用，指令助记符位于 allins.hpp 中。指令助记符存储于一个非常大的枚举结构（enum）中，前两个字符定义处理器。auxref 属性是一个依赖于处理器的字段，因此 aux_rep 是一个位标志（在 intel.hpp 中定义）。故第一行查找 rep movsd 或者 rep movsb。

更多资源请点击访问稀.酷客(www.ckook.com)

注意

rep movsw 怎么样？Intel 有一些奇妙的被称做前缀的东西，可以改变跟在其后的指令的长度。

```
   F3 A5    rep movsd
66 F3 A5    rep movsw
   F3 A4    rep movsb
```

前缀 0x66 将 rep movsd 同 rep movsw 区分开来。这个插件将忽略 movsw 指令。

第三行和第七行的 if 语句查看指令的操作符。insn_t 类包含一个 op_t 对象的数组，这里的对象就是操作符。

```
#define UA_MAXOP      6
   op_t Operands[UA_MAXOP];
```

操作符类提供更多关于指令的细节。你可以使用操作符确定指令在使用哪个寄存器，或指令是否有一个立即数作为偏移量。代码使用的属性是 dtype。

```
#define dt_byte     0        // 8 bit
#define dt_word     1        // 16 bit
#define dt_dword    2        // 32 bit
```

指令现在被充分的解码以确定是 rep movsd 还是 rep movsb。funcSearch 函数里剩余部分的代码用以查找匹配的 rep movsd/rep movsb 对，变量 last_bp 被用来跟踪顺序。任何不匹配的动作也被存储下来。这个插件不执行任何代码分析，这里有可能有跳转连接没被匹配的集合。rep movsd 和 rep movsb 之间的区别也会被核算以标出不符之处。

2．显示数据

IDA 有单列和多列列表框的 API 函数。choose2 函数是一个封装器，它使用预设的选项（如创建一个模式窗口等）调用 choose 函数。这个函数运行良好，因为它相对来说比较容易使用。

下面是 kernwin.hpp 中带注释的函数原型。

```
inline ulong choose2(
  void *obj,                               // object to show
  int width,                               // Max width of lines
  ulong (idaapi*sizer)(void *obj),         // Number of items
  char *(idaapi*getl)(void *obj,           // Description of
  ulong n,char *buf),                      // n-th item (1..n)
                                           // 0-th item if header line
  const char *title,                       // menu title (includes ptr to
                                              help)
  int icon,                                // number of the default icon to
                                           // display
  ulong deflt=1,                           // starting item
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
    chooser_cb_t *del=NULL,                    // cb for "Delete"(may be NULL)
                                               // supports multi-selection
                                               // scenario too
                                               // returns: 1-ok, 0-failed
    void (idaapi*ins)(void *obj)=NULL,         // cb for "New" (may be NULL)
    chooser_cb_t *update=NULL,                 // cb for "Update"(may be NULL)
                                               // update the whole list
                                               // returns the new location of
                                                  item 'n'
  void (idaapi*edit)(void *obj,ulong n)=NULL,  // cb for "Edit"
                                               // (may be NULL)
  void (idaapi*enter)(void * obj,ulong n)=NULL, // cb for non-modal
                                                  "Enter" (may be NULL)
  void (idaapi*destroy)(void *obj)=NULL,       // cb to call when the
                                               window is closed (may be NULL)
  const char * const *popup_names=NULL,        // Default:
                                               // insert, delete, edit, refresh
  int (idaapi*get_icon)(void *obj,ulong n)=NULL); // cb for get_icon
                                               // (may be NULL)
  }
```

下面是查找 memcpy 时对 choose2 的调用。

```
// create chooser list box
choose2(false,                 // non-modal window
  -1, -1, -1, -1,              // position is determined by Windows
  node,                        // object to show
  qnumber(header),             // number of columns
  widths,                      // widths of columns
  size,                        // function that returns number of lines
  description,                 // function that generates a line
  window_title,                // window title
  -1,                          // use the default icon for the window
  0,                           // position the cursor on the first line
  NULL,                        // "kill" callback
  NULL,                        // "new" callback
  NULL,                        // "update" callback
  NULL,                        // "edit" callback
  enter,                       // function to call when the user pressed Enter
  destroy,                     // function to call when the window is closed
  NULL,                        // use default popup menu items
  NULL);                       // use the same icon for all line
```

Find memcpy 插件有许多回调函数被置为 NULL。不过，大部分此类回调函数根本用不上。在列表框中增加新行的做法不太常见。弹出菜单的回调函数在操作列表数据方面要比跳转到单项

更多资源请点击访问稀酷客(www.ckook.com)

的反汇编上更有用。

关键的回调函数是 size、description、enter 和 destroy。

size 回调函数返回显示在列表框中的行数。关于它没有太多可讲的，它只涉及列表项的增加或删除。Find memcpy 的 size 函数原型是：

```
ulong idaapi size(void* obj)
```

description 回调函数向列表框的行中填入数据。列表中的每项都会调用它。传递给这个函数的参数有对象、行号和 arrptr。最后一项是列数据的指针数组。description 函数把它希望显示的文本数据复制到数组。当把 0 传给参数 n 时 description 设置列头部。下面的代码来自 Find memcpy，把列头部复制到 arrptr。

```
static const char* header[] = {"Address", "Type", "Movsd/b distance"};
void idaapi description(void *obj,ulong n,char * const *arrptr)
{
    if ( n == 0 ) // sets up headers
    {
      for ( int i=0; i < qnumber(header); i++ )
        qstrncpy(arrptr[i], header[i], MAXSTR);
      return;
}
```

enter 回调函数通常被用于跳转到一个地址。当用户在选择的列上按回车键或双击时将调用此函数。传递给此函数的参数是对象和行号。

```
void idaapi enter(void * obj, ulong n)
```

选择列被销毁时将调用 destroy 回调函数。destroy 可以执行资源清除，正如在 Find memcpy 例子中的情况一样。

```
void idaapi destroy(void* obj)
{
    netnode *node = (netnode *)obj;
    node->kill();
    return;
}
```

3. 总结

Find memcpy 插件是对 IDA API 的一份简单介绍。下面的插件将使用本节里出现的很多函数并在其基础上构建。

9.10 间接调用插件

我们在 IDC 一节介绍了一个脚本，它通过 VTable 寻找间接调用并为其创建交叉引用。这个解决方案需要知道感兴趣的 VTable 位于何处。与观察 VTable 的目标不同，我们可以反其道而

更多资源请点击访问稀.酷客(www.ckook.com)

行之——寻找所有的调用者。调用者包含任意间接的跳转指令。不过为简洁起见，我们所说的间接调用是指间接调用和跳转。

建议的策略

(1) 与 Find memcpy 插件类似，扫描二进制文件中感兴趣的指令，这里指的是间接调用。

(2) 在所有的间接调用上设置断点。

(3) 插件向调试器新加一个回调函数。

(4) 调试器处理（instruments）二进制文件。

(5) 回调函数记录信息。回调可以选择步入调用目标并记录地址。

(6) 当进程退出时移除断点。

(7) 向用户呈现数据，也可以加上交叉引用。

基于以上建议的策略，需要执行 4 项独立的任务。把任务分开后也可以对代码进行划分，这样方便以后对代码做部分替换。现在并不需要完全可工作的选择列表，在开发过程中写入消息窗口足矣。

- 收集数据
- 询问用户选项
- 实现回调函数
- 将结果显示给用户

本章后面将展示插件，相应的代码和屏幕截图则在接下来的小节中给出。

9.10.1　收集数据

收集数据前，我们需要存储它们的数据结构。在编写插件时，数据容器会发生变化，但 netnode 仍是主数据结构。这个插件使用两个 netnode 和一个 qvector。netnode 在前面的插件中已经介绍过，qvector 也是一个 SDK 数据类型，它是在 pro.h 中定义的，它支持绝大多数的标准向量方法。

名　字	数据类型	描　述	内部类型
calls	Netcode	包含所有发现的间接调用	indirect_t
vtables	Netcode	包含所有发现的VTable	vtable_t
bplist	qvector	调用netnode的索引列表	ulong

netnode 使用 altval 和 supval 数组支持地址查询和对象迭代。altval 稀疏数组是通过地址访问的，它包含的值是一个 supval 数组的索引，而 altval 数组被初始化为 0。因此 supval 索引从 1 开始。下面是一些从 indirectCalls 头部注释中提取的示例代码。

```
// .text:030CC0FB   call   dword ptr [eax+3Ch] ;

indirect_t myObj;

ulong index = calls->altval(0x030CC0FB);

if (index != 0) // indirect call (assume we assigned it earlier)
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
{
    indirect_t myObj = calls->supval(index, &myObj, sizeof(myObj));
    msg("%a -> %a\n", myObj.caller, myObj.target);
}
```

数据的收集是由 findIndirectCalls 函数执行的。不仅是函数，当前区段也会被扫描。尽管并不在严格意义上的函数内，但此函数应该收集定义的间接调用。下面的代码将扫描此类调用。

```
switch (cmd.itype)
{
case NN_callfi:
case NN_callni:
case NN_jmpfi:
case NN_jmpni:
  {
    if (get_first_fcref_from(cmd.ea) == BADADDR &&
      get_first_dref_from(cmd.ea) == BADADDR) //no fwd xref
  {
    indirect_t currcall;
    fillIndirectObj(currcall);
    if (cmd.itype & NNJMPxI) // jmp?
    {
        currcall.flags |= JMPSETFLAG;
    }
    node->altset(cmd.ea, counter); // altval keyed by addr
    node->supset(counter++, &currcall, sizeof(currcall) );
```

这个代码和前面的插件差不多，它分析指令并检查某些助记符。检查代码和数据中是否有来自调用的交叉引用，为了避开跳转表有必要检查数据交叉引用。调用 fillIndirectObj 是了准备 indirect_t 对象。更多的指令解码发生，结果将记入对象。

如果这个调用是一个间接的近调用，将会有更多的处理。其目的是确定调用是否是下面的形式：

```
call  [reg] or call  [reg + offset]
```

前面的调用尤其是带偏移量的调用在寄存器里可能包含一个 VTable 地址。寄存器被提取并同偏移量一起保存在对象中。注意，此信息位于操作数的类型属性里。有了寄存器和偏移量，目标地址就可以算出来了。理论上，所有的调用指令都可以被解码。最后，有一个 test 检查标志，确定调用是否是 jmp。如果调用在运行期间完成，为了设置合适的交叉引用类型需要这么做。

这个函数在从用户获取选项之前完成信息的收集。

9.10.2 用户接口

有多个选项可供用户使用。AskUsingForm_c API 调用创建用户接口。（参见图 9-18）

更多资源请点击访问稀.酷客(www.ckook.com)

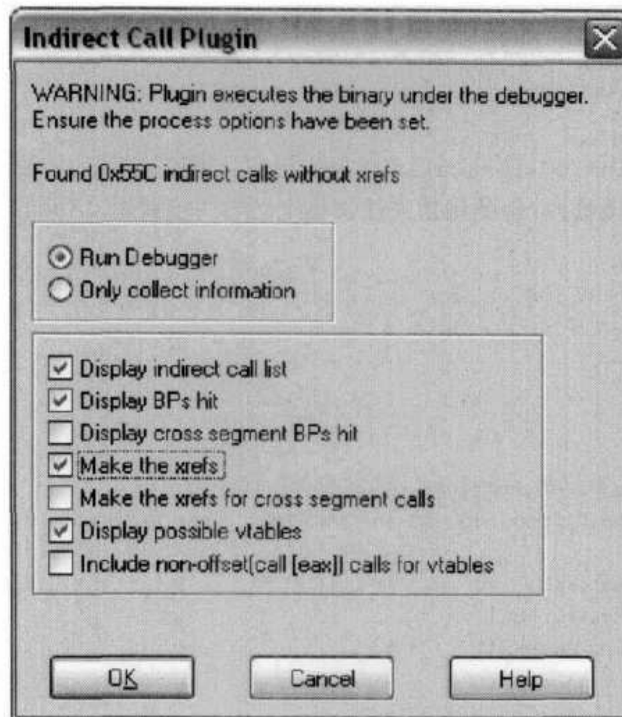


图 9-18　间接调用用户接口

特定角色控制是否会出现复选框或单选按钮。没有多少可用的相关文档，不过有些示例代码。（参见 http:// www.openrce.org/downloads/details/32/User_Interface_Sample_Code。）

选项会被处理，如果用户选择运行调试器，会产生一个新的 API 调用去钩住调试器的通知。

```
if (!hook_to_notification_point(HT_DBG, callback, &gDbgOptions) )
{
    warning("Could not hook to notification point\n");
    register_event(E_HOOKFAIL);
    return;
}
```

下面是函数 retype 及支持的钩子类型。

```
HT_IDP,        // Hook to the processor module.
HT_UI,         // Hook to the user interface.
HT_DBG,        // Hook to the debugger.
HT_IDB,        // Hook to the database events.
idaman bool ida_export hook_to_notification_point(
                              hook_type_t hook_type,
                              hook_cb_t *cb,
                              void *user_data);
```

更多资源请点击访问稀.酷客(www.ckook.com)

第一个参数是钩子类型，第二个参数是接收通知的回调函数，最后一个参数可以是 NULL。传递一个对象起两个作用：一是为了去除钩子必须使用同样的对象；二是把数据传递给回调函数，在这种情况下，传递的 user_data 是全局的。

最后，设置断点，使用 start_process 调用来启动进程。

```
int idaapi start_process(const char *path, const char *args,
                         const char *sdir)
```

如果参数是 NULL，start_process 使用先前在 **Debugger | Process options** 选项中输入的数据。回调函数被设置后，调试器就开始运行。

9.10.3 实现回调

调试器启动后，回调函数就耐心地等待事件。下面是回调的原型。

```
int idaapi callback(void* user_data,int notification_code,va_list va)
```

通知代码描述接收的事件类型。有不同类型的通知，从低层的处理库加载到高层的断点通知。dgb.hpp 文件的 dbg_notification_t 枚举结构中记录了这些通知。回调函数有一个 switch，可以处理三类通知。

1. **dbg_bpt**

dbg_bpt 是断点通知。处理 dbg_bpt 的代码有三种可能的结果。

- 断点地址不是调用 netnode。这是用户设置的一个断点，应该进行相应处理。插件调用 suspend_process 并退出回调函数。

```
suspend_process();
return 0;
```

- 断点由插件设置，不过调用指令不是预解码类型之一。调用者地址存储在 last_bp 中，因为在 step_into 之前目标不会被解析。会产生对 request_del_bp 和 request_setp_into 的调用。不能从通知处理程序中调用 Setp_into 函数。

```
last_bp = from;              // saves the caller address
request_del_bpt(from);       // queue request_del_bpt()
  //
  // From: dbg.hpp request_step_into() AND step into()
  // Type: Asynchronous function - available as Request
  // In Notification handler it is MANDATORY to call
  // Async function in request form
request_step_into();     // queue a request_step_into()
  // request will be run after all notification handlers
run_requests();
break;
```

- 断点由插件设置，调用指令是预解码类型之一。my_indirect 对象包含寄存器号和偏移

更多资源请点击访问稀.酷客(www.ckook.com)

量（可以是0）。用 get_reg_val 读取寄存器值。

寄存器值随后被保存在 vtaddr 中，这被假定是一个 VTable 地址。为了恢复目标地址，需要执行 VTable 查找。不过，在通知处理程序中读内存并不能提供可靠的结果，因为数据库和进程可能不同步。这个问题是在开发这个插件时发现的。invalidate_dbgmem_contents 函数使 IDA 缓存作废并刷新缓存。

在通知处理程序中，在读写内存前要调用 invalidate_dbgmem_contents。另一个可选项是 invalidate_dbgmem_config，它虽然慢点但更全面一些。这两个函数都是在 bytes.hpp 中定义的。

另外两个被调用的函数是 addVTable 和 setTargetXref。假设用户选项允许的话，函数将创建交叉引用和可能的 VTable。

```
  // copy register_t struct in regval
get_reg_val(regname[my_indirect.call_reg], &regval);
  // vtaddr == VTable base addr
vtaddr = (ea_t)regval.ival;
  // flushes IDA's cache
invalidate_dbgmem_contents((ea_t)regval.ival, 0x100 +
                           my_indirect.offset);
  // read target address from table
to = get_long(my_indirect.offset + vtaddr);
my_indirect.target = to;
  //
addVTable(my_dbg,vtaddr, &my_indirect);
setTargetXref(my_dbg, index, &my_indirect);
calls->supset(index, &my_indirect, sizeof(my_indirect));
del_bpt(from);
continue_process();
break;
```

2. **dgb_step_into**

- dbg_step_into 是 step_into 通知，它要么是由用户发起的，要么是由 request_step_into 调用产生的。如果是用户引起的通知，那么会调用 suspend_process。

  当前 EIP 是调用的目标，其地址被复制到对象中。setTargetXref 基于用户的选项增加交叉引用。

```
from = last_bp;
if (from == BADADDR)
{
    suspend_process(); // user caused step_into
    return 0;
}
long index = calls->altval(from); // index into supval
get_reg_val("EIP", &regval); // current EIP is the 'to'
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
to = (ea_t)regval.ival;
indirect_t my_indirect;
calls->supval(index, &my_indirect, sizeof(my_indirect) );
my_indirect.target = to;
    // Add cross reference based on user options and checks
setTargetXref(my_dbg, index, &my_indirect);
    // save completed indirect_t object
calls->supset(index, &my_indirect, sizeof(my_indirect) );
    // reset last_bp and continue the debugger
last_bp = BADADDR;
continue_process();
break;
```

3. **dbg_process_exit**

这个通知标志着被调试的进程终止了。

```
unhook_from_notification_point(HT_DBG, callback, user_data);
requestDelBps(calls);
run_requests();
register_event(E_PROCEXIT);

if (options & DISPLAY_INCALLS)
    createIndirectCallWindow(calls);

if (options & DISPLAY_BPS)
    createCompletedBpWindow(calls, my_dbg->bplist);

if (options & DISPLAY_VTABLES)
    createVTableWindow(my_dbg->vtables);
```

9.10.4 显示结果

VTable 显示的内容包括一个估算的 VTable 大小，这个大小是通过迭代指针和检查引用估算的。显示函数的余下部分同 Find memcpy 插件类似。description 回调引入了两个新函数，它们都处理显示文本。第一个是 get_nice_colored_name，它能构建在 IDA 中列出的地址，例如 **segment:address**。不同的标志指定格式。

```
#define GNCN_NOSEG        0x0001      // ignore the segment prefix
                                      // producing the name
#define GNCN_NOCOLOR      0x0002      // generate an uncolored name
#define GNCN_NOLABEL      0x0004      // don't generate labels
#define GNCN_NOFUNC       0x0008      // don't generate funcname+... expressions
#define GNCN_SEG_FUNC     0x0010      // generate both segment and function names
      (default is to omit segment name if a function name is present)
#define GNCN_SEGNUM     0x0020        // segment part is displayed as aa hex number
#define GNCN_REQFUNC    0x0040        // return 0 if the address does not
                                      // belong to a function
#define GNCN_REQNAME      0x0080      // return 0 if the address can only be
```


更多资源请点击访问稀.酷客(www.ckook.com)

```
                                        // represented as a hex number
// returns: the length of the generated name in bytes
// The resulting name will have color escape characters
// GETN_NOCOLOR was not specified
// (see lines.hpp for color definitions)
idaman ssize_t ida_export get_nice_colored_name(
        ea_t ea,
        char *buf,
        size_t bufsize,
        int flags=0);
```

第二个新函数恢复名字。默认情况下，IDA 使用有缺损的名字，尽管可以改变这个选项。下面的函数产生一个简短的恢复好的名字。

```
inline char *get_short_name(ea_t from, ea_t ea, char *buf, size_t bufsize)
```

两个函数都位于 names.hpp 中。我们尝试用这个插件处理 IE7 中的 jscipt.dll（参见图 9-19 至图 9-23）。

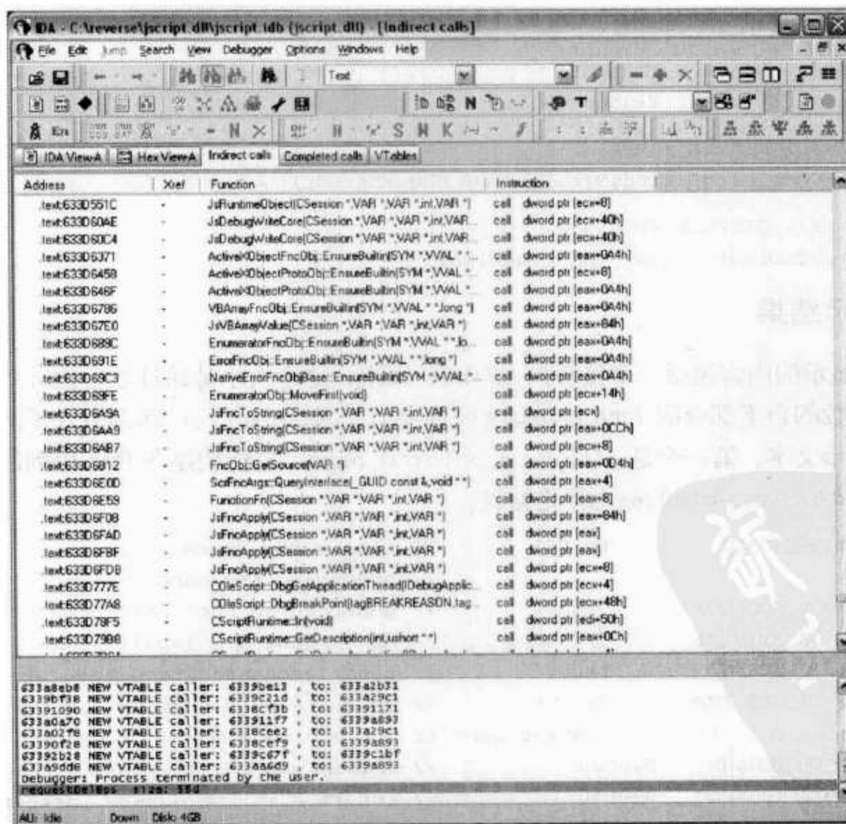


图 9-19　jscript.dll 中的间接调用列表

更多资源请点击访问稀.酷客(www.ckook.com)

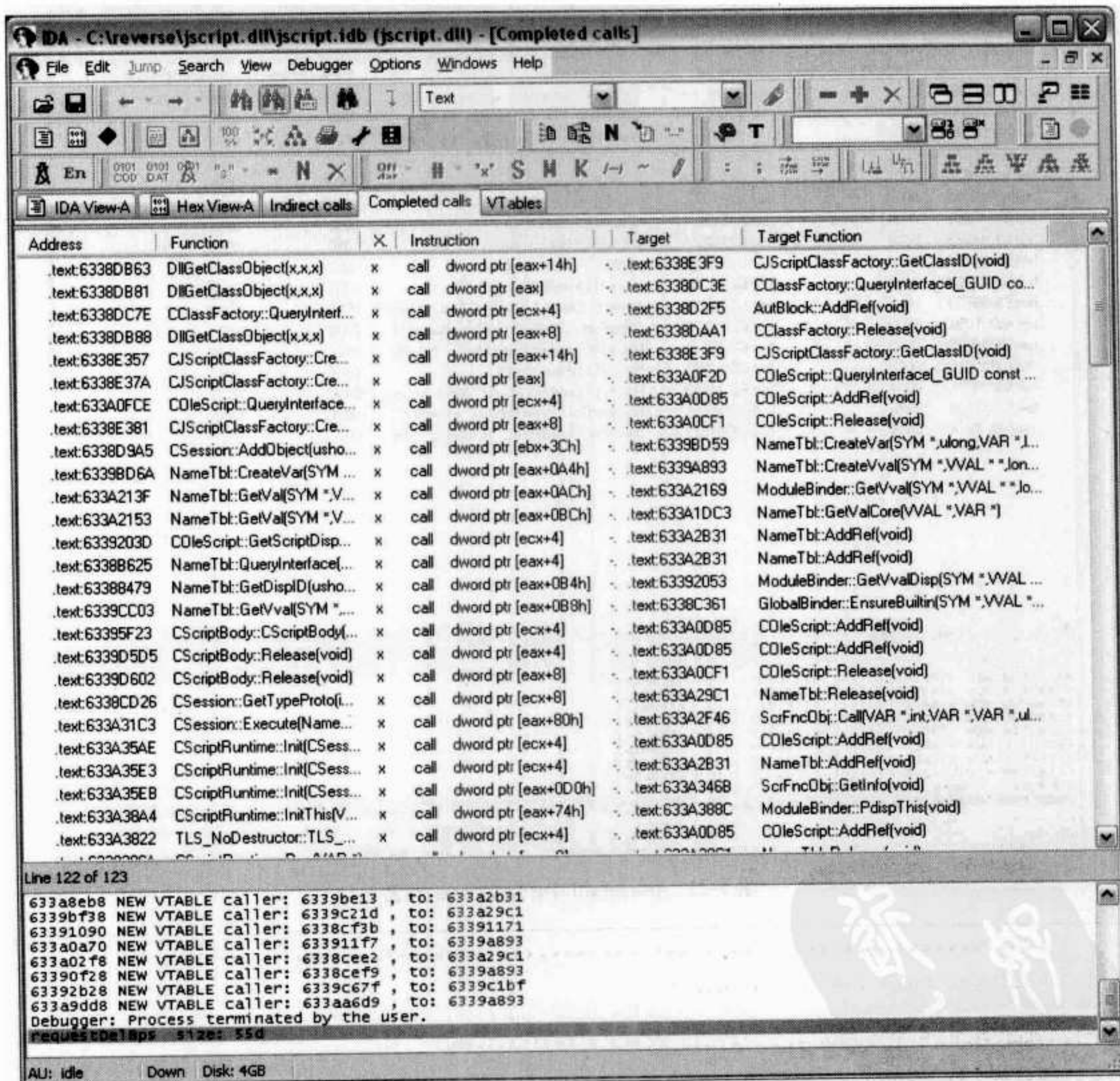


图 9-20 jscript.dll 中完整的调用列表

更多资源请点击访问稀.酷客(www.ckook.com)

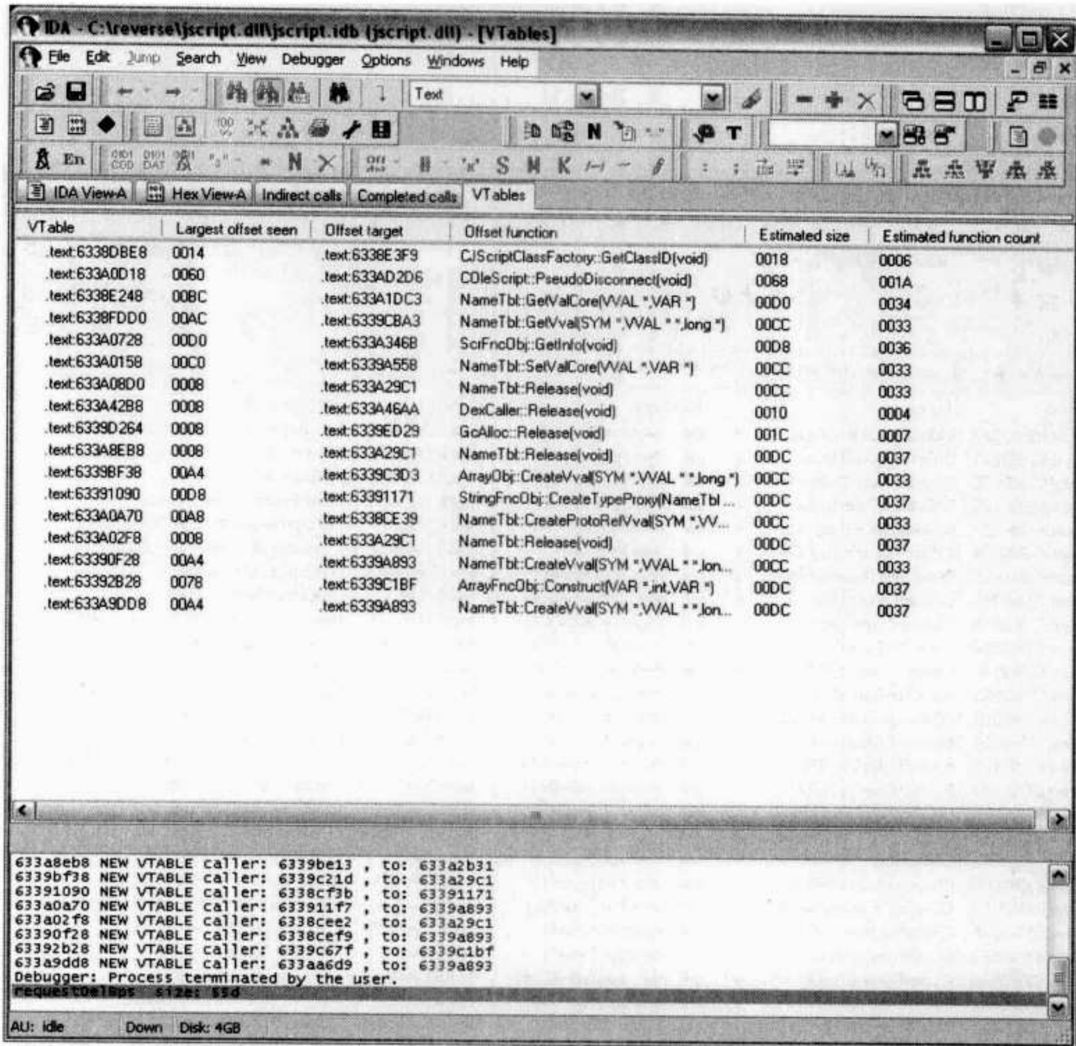


图 9-21　jscript.dll 中的 VTable 列表

```
/******************************************************************************
* Indirect Call IDA Pro plugin
*
* Copyright (c) 2008 Luis Miras
* Licensed under the BSD License
*
******************************************************************************/
#ifndef INDIRECTCALLS_H_
```

图 9-22　间接调用插件 indirectCall.h

更多资源请点击访问稀.酷客(www.ckook.com)

```
#define INDIRECTCALLS_H_

#define NODE_COUNT -1
#define NNJMPxI 0x40
#define CNAMEOPT (GNCN_NOCOLOR | GNCN_NOFUNC | GNCN_NOLABEL)

struct dbgOptions; //fwd declaration
struct indirectCallObj; //fwd declaration
typedef indirectCallObj indirect_t;
typedef qvector<ulong> bphitlist_t;
long vtEstimateSize(ea_t);
void idaapi vtDescription(void *,ulong, char * const *);
void idaapi vtEnter(void * ,ulong);
void idaapi vtDestroy(void*);
void createVTableWindow(netnode* vtables);
void idaapi icDescription(void *,ulong ,char * const *);
void idaapi icEnter(void * ,ulong);
void idaapi icDestroy(void*);
ulong idaapi size(void*);
void createIndirectCallWindow(netnode*);
void idaapi ccDescription(void *,ulong ,char * const *);
void idaapi ccEnter(void* ,ulong);
void idaapi ccDestroy(void*);
ulong idaapi ccSize(void*);
void createCompletedBpWindow(netnode* , bphitlist_t*);
void requestSetBps(netnode*);
void setBps(netnode*);
void requestDelBps(netnode*);
void delBps(netnode*);
void setTargetXref(dbgOptions* , long , indirect_t*);
void addVTable(dbgOptions* , ea_t , indirect_t*);
int idaapi callback(void* , int , va_list);
void fillIndirectObj(indirect_t &);
bool setnodesize(netnode* , long);
long getnodesize(netnode*);
long getobjcount(netnode*);
void findIndirectCalls(segment_t* , netnode*);
void closeListWindows(void);
void register_event(ulong);
void run(int);
int init(void);
void term(void);

struct indirectCallObj
{
```

图 9-22 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
    ea_t caller;     // indirect caller address
    ea_t target;     // target address
    ea_t offset;     // valid for call [reg+offset]
                     // defaults to 0
    short call_reg;  // enum REG
    short flags;     // enum callflags_t
};

struct vtableObj
{
    ea_t baseaddr;         // baseaddr reg in call [reg + off]
    ea_t largestOffset;    // largest off seen in call [reg + off]
};
typedef struct vtableObj vtable_t;
typedef qvector<ulong> bphitlist_t;
struct dbgOptions
{
    netnode* calls;
    netnode* vtables;
    bphitlist_t* bplist;
    ulong options;
};
struct completedbp
{
    netnode* calls;
    bphitlist_t* callindex;
};
typedef completedbp completedbp_t;

enum uioptions_t {
    DISPLAY_INCALLS   = 0x0001,
    DISPLAY_BPS       = 0x0002,
    DISPLAY_XS_BPS    = 0x0004,
    MAKE_XREFS        = 0x0008,
    MAKE_XS_XREFS     = 0x0010,
    DISPLAY_VTABLES   = 0x0020,
    INC_NONOFF_CALLS = 0x0040
};
enum callflags_t {
    JMPSETFLAG = 1,
    XRSETFLAG = 2,
    XSEGFLAG = 4
};
char* regname[] = {"EAX","ECX","EDX","EBX","ESP","EBP","ESI","EDI"};
```

图 9-22 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
enum REG {eax, ecx, edx, ebx, esp, ebp, esi, edi, none = -1};
enum EVENTS
{
  E_START, E_CANCEL, E_OPTIONS, E_HOOKFAIL, E_PROCFAIL,
  E_DWCALL, E_DWXREFS, E_DWVTABLE, E_PROCEXIT
};
// incomplete calls, choose2() list box
char icTitle[] = "Indirect calls" ;
static const char* icHeader[] = {"Address", "Xref","Function", "Instruction"};
static const int icWidths[] = {16, 4, 36, 20};

// completed calls, choose2() list box
char ccTitle[] = "Completed calls" ;
static const char* ccHeader[] = {"Address", "Function", "Xref", "Instruction",
"Xseg","Target", "Target Function"};
static const int ccWidths[] = { 16, 28, 4, 18, 4, 16, 28};

// vtables, choose2() list box
char vtTitle[] = "VTables";
static const char* vtHeader[] = {"VTable ", "Largest offset seen", "Offset
target", "Offset function", "Estimated size", "Estimated function count"};
static const int vtWidths[] = { 16, 16, 16, 28, 16, 20};

// ui string AskUsingForm_c()
const char preformat[] =
"STARTITEM 0\n"
// Help
"HELP\n"
"This plugin searches for indirect calls. For example:\n"
"\n"
"call     dword ptr [eax+14h]\n"
"jmp      eax\n"
"\n"
" "
"Breakpoints are set on all the calls.\n"
"A breakpoint handler will:\n"
" 1. Determine if one of its breakpoints triggered.\n"
" 2. Delete the breakpoint\n"
" 3. Step into the call\n"
" 4. Record both the caller and callee addresses\n"
"\n"
"ENDHELP\n"
// Title
"Indirect Call Plugin\n"
// Dialog Text
"WARNING: Plugin executes the binary under the debugger.\n"
```

图 9-22 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
"Ensure the process options have been set.\n\n"
"Found 0x%a indirect calls without xrefs\n\n"
// Radio Buttons
"<#Runs the debugger#"
"Run Debugger:R>\n"
"<#Collects data on indirect calls#"
"Only collect information:R>>\n"
// Check Boxes
"<# Create indirect call window. #"
"Display indirect call list :C>\n"
"<# Create BP window. #"
"Display BPs hit :C>\n"
"<# Include cross segment BPs in BP window. #"
"Display cross segment BPs hit :C>\n"
"<# automatically create xrefs btwn caller and target. #"
"Make the xrefs :C>\n"
"<# automatically create xrefs btwn caller and target in different segments. #"
"Make the xrefs for cross segment calls:C>\n"
"<# Create a vtable window #"
"Display possible vtables :C>\n\n"
"<# May lead to false positives (not recommended) #"
"Include non-offset(call [eax]) calls for vtables :C>>\n\n";
#endif /* INDIRECTCALLS_H_ */
```

图 9-22　(续)

```
/***************************************************************************
* Indirect Call IDA Pro plugin
*
* Copyright (c) 2008 Luis Miras
* Licensed under the BSD License
*
* Requirements:      This plugin works alongside the IDA Pro debugger.
*                    The plugin requires x86 processor. The plugin "should"
*                    work under the IDA Linux debugger. It has not been
*                    tested.
*
* Description:       The plugin attempt to create cross references for
*                    indirect calls/jmps. For brevity indirect calls/jmp
*                    will be refered only as indirect calls. The plugin
*                    also attempts to identify vtables.
*
* Strategy:          The binary's current segment is scanned for indirect
```

图 9-23　间接调用插件 indirectCall.cpp

更多资源请点击访问稀.酷客(www.ckook.com)

```
 *                      calls. The binary is instrumented under the debugger.
 *                      A breakpoint handler either calculates the target or
 *                      steps into the target. Depending on user options cross
 *                      references will be made and possible vtables listed.
 *
 * Data structures:     netnode and qvector are used. Both are built in IDA
 *                      types, minimizing 3rd party dependencies. netnodes
 *                      allow for persistent data,they are saved in the IDB
 *                      However, in this plugin the netnodes are kill()'ed
 *
 * netnodes are implemented internally as B-trees.
 * IDA uses netnodes extensively for its own storage.
 * netnodes are defined in netnode.hpp.
 *
 * netnodes in the plugin: calls - holds all indirect calls
 *                         vtable - holds all vtables
 *
 * netnodes have various internal data structures.
 * The plugin uses 2 types of arrays:
 *   altval - a sparce array of 32 bit values, initially set to 0.
 *   supval - an array of variable sized objects (MAXSPECSIZE)
 *
 * Addresses are used as keys into altval array. The value at the key
 * is then used as an index into the supval array. The supval array
 * holds an object of variable size.
 *
 * This allows fast lookup using address keys, while being able to
 * iterate through all items using supval.
 *
 * An example:
 *
 * .text:030CC0FB call dword ptr [eax+3Ch]
 *
 * indirect_t myObj;
 * ulong index = calls->altval(0x030CC0FB);
 *
 * if (index != 0) // indirect call (assume we assigned it earlier)
 * {
 *   indirect_t myObj = calls->supval(index, &myObj, sizeof(myObj));
 *   msg("%a -> %a\n", myObj.caller, myObj.target);
 * }
 *
 * the calls netnode holds indirect_t objects
```

图 9-23 (续)

更多资源请点击访问稀.酷客(www.ckook.com)

```
 * the vtables netnode holds vtable_t objects
 * bphitlist_t is a qvector that holds indexes into the calls netnode
 ***************************************************************************/

#include <ida.hpp>
#include <idp.hpp>
#include <dbg.hpp>
#include <loader.hpp>
#include <allins.hpp>
#include <intel.hpp>
#include "indirectCalls.h"

dbgOptions gDbgOptions = {NULL, NULL, NULL, 0};
/****************************************************************************
 * Function: vtEstimateSize
 * Args:      ea_t addr          - base address of a VTable
 * Return:    long               - Estimated VTable length
 *
 * This function attempts to calculate the size of a vtable given its
 * base address. It checks xrefs to determine if still in a vtable
 *
 .text:03010D34 off_3010D34    dd offset sub_308A561
 .text:03010D34
 .text:03010D38                dd offset sub_3082D8D
 .text:03010D3C                dd offset sub_3082DA6
 .text:03010D40                dd offset sub_3091542
 .text:03010D44                dd offset sub_30B9110
 .text:03010D48                dd 75667608h, 6174636Eh, 62h ;
                                                              ; 8 'vfunctab'
 .text:03010D54 off_3010D54    dd offset sub_308A561
 *
 * Sometimes a string is stored at the end of a vtable as in this case.
 * vtEstimateSize doesn't understand anything other than dword ptrs
 ***************************************************************************/
long vtEstimateSize(ea_t addr)
{
  flags_t flags;
  ea_t curraddr = addr;
  ea_t lastaddr = addr;
  bool done = false;

  curraddr = next_head(lastaddr, BADADDR);
  while (!done)
  {
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
    if (curraddr - lastaddr != 4) // DWORD size differences
    {
      done = true;
    }
    flags = getFlags(curraddr);
    if (!done && !isDwrd(flags))
    {
      done = true;
    }
    // a dref_to could suggest the start of a new vtable
    if (!done && get_first_dref_to(curraddr) != BADADDR)
      done = true;
    if (!done)
    {
      lastaddr = curraddr;
      curraddr = next_head(lastaddr, BADADDR);
    }
  }
  return lastaddr - addr + 4;
}
/***************************************************************************
* Function: vtDescription
*
* This is a standard callback in the choose2() SDK call. This function
* fills in all column content for a specific line. Headers names are
* set during the first call to this function, when n == 0.
* arrptr is a char* array to the column content for a line.
*                 arrptr[number of columns]
*
* vtDescription creates 6 columns based on the vtHeader array
***************************************************************************/
void idaapi vtDescription(void *obj,ulong n,char * const *arrptr)
{
  netnode *node = (netnode *)obj;
  vtable_t curr_vtable;
  ea_t target;
  long vtSize;
  if ( n == 0 ) // sets up headers
  {
    for ( int i=0; i < qnumber(vtHeader); i++ )
      qstrncpy(arrptr[i], vtHeader[i], MAXSTR);
    return;
  }
```

图 9-23 (续)

更多资源请点击访问稀.酷客(www.ckook.com)

```
    // Empty netnode
    if (!getobjcount(node) )
      return;
    char buffer[MAXSTR];
    node->supval(n, &curr_vtable, sizeof(curr_vtable));
    vtSize = vtEstimateSize(curr_vtable.baseaddr);
    target = get_long(curr_vtable.largestOffset + curr_vtable.baseaddr);

    get_nice_colored_name(curr_vtable.baseaddr,
                          arrptr[0], MAXSTR, CNAMEOPT);
    qsnprintf(arrptr[1], MAXSTR, "%04a", curr_vtable.largestOffset);
    get_nice_colored_name(target, arrptr[2], MAXSTR, CNAMEOPT);

    get_short_name(BADADDR,target , buffer, MAXSTR); //demangles fname
    qsnprintf(arrptr[3], MAXSTR, "%s", buffer);
    qsnprintf(arrptr[4], MAXSTR, "%04a", vtSize);
    qsnprintf(arrptr[5], MAXSTR, "%04a", vtSize/4);
    return;
  }
  /*************************************************************************
  * Function: vtEnter
  *
  * This is a standard callback in the choose2() SDK call. This function
  * is called when the user pressed Enter or Double-Clicks on a line in
  * the chooser list.
  *************************************************************************/
  void idaapi vtEnter(void * obj,ulong n)
  {
    vtable_t curr_vtable;
    netnode *node = (netnode *)obj;
    node->supval(n, &curr_vtable, sizeof(curr_vtable));
    jumpto(curr_vtable.baseaddr);
    return;
  }
  /*************************************************************************
  * Function: vtDestroy
  *
  * This is a standard callback in the choose2() SDK call. This function
  * is called when the chooser list is being destroyed. Resource cleanup
  * is common in this function. In this case any resource
  * cleanup is handled by register_event().
  *************************************************************************/
  void idaapi vtDestroy(void* obj)
  {
    netnode *node = (netnode *)obj;
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
  msg("\"%s\" window closed\n", vtTitle);
  register_event(E_DWVTABLE);
  return;
}
/**************************************************************************
* Function: createVTableWindow
*
* A wrapper around choose2() API. 'Generic list chooser (n-column)'
* This sets up the callbacks and necessary options.
* NOTE: 1. Cannot free the "object to show" until chooser closes
*       2. Cannot unload plugin until chooser closes,
*          removing callbacks.
**************************************************************************/
void createVTableWindow(netnode* vtables)
{
  choose2(false,              // non-modal window
    -1, -1, -1, -1,           // position is determined by Windows
    vtables,                  // object to show
    qnumber(vtHeader),        // number of columns
    vtWidths,                 // widths of columns
    size,                     // function that returns number of lines
    vtDescription,            // function that generates a line
    vtTitle,                  // window title
    -1,                       // use the default icon for the window
    0,                        // position the cursor on the first line
    NULL,                     // "kill" callback
    NULL,                     // "new" callback
    NULL,                     // "update" callback
    NULL,                     // "edit" callback
    vtEnter,                  // function to call when the user pressed Enter
    vtDestroy,                // function to call when the window is closed
    NULL,                     // use default popup menu items
    NULL);                    // use the same icon for all line
}
/**************************************************************************
* Function: icDescription
*
* This is a standard callback in the choose2() SDK call. This function
* fills in all column content for a specific line. Headers names are
* set during the first call to this function, when n == 0.
* arrptr is a char* array to the column content for a line.
*                 arrptr[number of columns]
*
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
* vtDescription creates 4 columns based on the icHeader array
*************************************************************************/
void idaapi icDescription(void *obj,ulong n,char * const *arrptr)
{
  netnode *node = (netnode *)obj;
  indirect_t curr_indirect;
  if ( n == 0 ) // sets up headers
  {
    for ( int i=0; i < qnumber(icHeader); i++ )
      qstrncpy(arrptr[i], icHeader[i], MAXSTR);
    return;
  }

  // list empty?
  if (!getobjcount(node) )
    return;
  char buffer[MAXSTR];
  node->supval(n, &curr_indirect, sizeof(curr_indirect) );
  func_t* currFunc = get_func(curr_indirect.caller);
  ua_ana0(curr_indirect.caller);
  get_nice_colored_name(curr_indirect.caller,
                        arrptr[0], MAXSTR, CNAMEOPT); // address
  if (curr_indirect.flags & XRSETFLAG)
    qstrncpy(arrptr[1], "x", MAXSTR);
  else
    qstrncpy(arrptr[1], "-", MAXSTR);
  get_short_name(BADADDR, currFunc->startEA, buffer, MAXSTR);
  qsnprintf(arrptr[2], MAXSTR, "%s", buffer);
  generate_disasm_line(cmd.ea, buffer, sizeof(buffer) );
  tag_remove(buffer, buffer, sizeof(buffer) );
  qsnprintf(arrptr[3], MAXSTR, "%s", buffer);
  return;
}
/*************************************************************************
* Function: icEnter
*
* This is a standard callback in the choose2() SDK call. This function
* is called when the user pressed Enter or Double-Clicks on a line in
* the chooser list.
*************************************************************************/
void idaapi icEnter(void * obj,ulong n)
{
  indirect_t curr_indirect;
```

图 9-23　（续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
  netnode *node = (netnode *)obj;

  node->supval(n, &curr_indirect, sizeof(curr_indirect) );
  jumpto(curr_indirect.caller);
  return;
}
/**************************************************************************
* Function: icDestroy
*
* This is a standard callback in the choose2() SDK call. This function
* is called when the chooser list is being destroyed. Resource cleanup
* is common in this function. In this case any resource cleanup is
* handled by register_event().
**************************************************************************/
void idaapi icDestroy(void* obj)
{
  netnode *node = (netnode *)obj;
  msg("\"%s\" window closed\n", icTitle);
  register_event(E_DWCALL);
  return;
}
/**************************************************************************
* Function: size
*
* This is a standard callback in the choose2() SDK call. This function
* returns the number of lines to be used in the chooser list.
**************************************************************************/
ulong idaapi size(void* obj)
{
  netnode *node = (netnode *)obj;
  return getobjcount(node);
}
/**************************************************************************
* Function: createIndirectCallWindow
*
* A wrapper around choose2() API. 'Generic list chooser (n-column)'
* This sets up the callbacks and necessary options.
* NOTE: 1. Cannot free the "object to show" until chooser closes
*       2. Cannot unload plugin until chooser closes,
*          removing callbacks.
**************************************************************************/
void createIndirectCallWindow(netnode* calls)
{
```

图 9-23 (续)

9

更多资源请点击访问稀.酷客(www.ckook.com)

```
  choose2(false,              // non-modal window
    -1, -1, -1, -1,           // position is determined by Windows
    calls,                    // object to show
    qnumber(icHeader),        // number of columns
    icWidths,                 // widths of columns
    size,                     // function that returns number of lines
    icDescription,            // function that generates a line
    icTitle,                  // window title
    -1,                       // use the default icon for the window
    0,                        // position the cursor on the first line
    NULL,                     // "kill" callback
    NULL,                     // "new" callback
    NULL,                     // "update" callback
    NULL,                     // "edit" callback
    icEnter,                  // function to call when the user pressed Enter
    icDestroy,                // function to call when the window is closed
    NULL,                     // use default popup menu items
    NULL);                    // use the same icon for all line
}
/***************************************************************************
* Function: ccDescription
*
* This is a standard callback in the choose2() SDK call. This function
* fills in all column content for a specific line. Headers names are
* set during the first call to this function, when n == 0.
* arg:    arrptr is a char* array to the column content for a line.
*         arrptr[number of columns]
* arg: completedbp_t* is atruct: netnode*     - points to all calls
*                               bphitlist_t - indexes of hit calls
*
* ccDescription creates 7 columns based on the icHeader array
***************************************************************************/
void idaapi ccDescription(void *obj,ulong n,char * const *arrptr)
{
  completedbp_t* cbp = (completedbp_t*)obj;
  indirect_t curr_indirect;
  if ( n == 0 ) // sets up headers
  {
    for ( int i=0; i < qnumber(ccHeader); i++ )
      qstrncpy(arrptr[i], ccHeader[i], MAXSTR);
    return;
  }
```

图9-23　（续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
  bphitlist_t& tmp = *(bphitlist_t*)cbp->callindex;
  ulong index = tmp[n-1];
  if (!tmp.size() ) // only needed if choose2 kill callback used
    return;          // since it removes members
  char buffer[MAXSTR];
  cbp->calls->supval(index, &curr_indirect, sizeof(curr_indirect) );
  func_t* currFunc = get_func(curr_indirect.caller);
  ua_ana0(curr_indirect.caller); //
  // seg.addr
  get_nice_colored_name(curr_indirect.caller, arrptr[0],
                        MAXSTR, CNAMEOPT);
  get_short_name(BADADDR, currFunc->startEA, buffer, MAXSTR);
  qsnprintf(arrptr[1], MAXSTR, "%s", buffer);
  if (curr_indirect.flags & XRSETFLAG)
    qstrncpy(arrptr[2], "x", MAXSTR); // made a cross reference
  else
    qstrncpy(arrptr[2], "-", MAXSTR);
  // get instruction disasm, remove color info
  generate_disasm_line(cmd.ea, buffer, sizeof(buffer));
  tag_remove(buffer, buffer, sizeof(buffer));
  qsnprintf(arrptr[3], MAXSTR, "%s", buffer);
  if (curr_indirect.flags & XSEGFLAG)
    qstrncpy(arrptr[4], "x", MAXSTR); // cross segment reference
  else
    qstrncpy(arrptr[4], "-", MAXSTR);
  get_nice_colored_name(curr_indirect.target,
                        arrptr[5], MAXSTR, CNAMEOPT);
  currFunc = get_func(curr_indirect.target);
    //demangles fname
  get_short_name(BADADDR, currFunc->startEA, buffer, MAXSTR);
  qsnprintf(arrptr[6], MAXSTR, "%s", buffer);
  return;
}
/************************************************************************
* Function: ccEnter
*
* This is a standard callback in the choose2() SDK call. This function
* is called when the user pressed Enter or Double-Clicks on a line in
* the chooser list.
************************************************************************/
void idaapi ccEnter(void * obj,ulong n)
{
  completedbp_t* cbp = (completedbp_t*)obj;
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
    bphitlist_t &tmp = *(bphitlist_t*)cbp->callindex;
    indirect_t curr_indirect;
    ulong index = tmp[n-1];

    cbp->calls->supval(index, &curr_indirect, sizeof(curr_indirect));
    jumpto(curr_indirect.caller);
    return;
}
/***************************************************************************
* Function: ccDestroy
*
* This is a standard callback in the choose2() SDK call. This function
* is called when the chooser list is being destroyed. Resource cleanup
* is common in this function. In this case any resource cleanup is
* handled by register_event().
***************************************************************************/
void idaapi ccDestroy(void* obj)
{
    completedbp_t* cbp = (completedbp_t*)obj;
    msg("\"%s\" window closed\n", ccTitle);
    register_event(E_DWXREFS);
    return;
}
/***************************************************************************
* Function: ccSize
*
* This is a standard callback in the choose2() SDK call. This function
* returns the number of lines to be used in the chooser list.
***************************************************************************/
ulong idaapi ccSize(void* obj)
{
    completedbp_t* cbp = (completedbp_t*)obj;
    return cbp->callindex->size();
}
/***************************************************************************
* Function: createCompletedBpWindow
*
* A wrapper around choose2() API. 'Generic list chooser (n-column)'
* This sets up the callbacks and necessary options.
* NOTE: 1. Cannot free the "object to show" until chooser closes
*       2. Cannot unload plugin until chooser closes,
*          removing callbacks.
***************************************************************************/
void createCompletedBpWindow(netnode* calls, bphitlist_t* bplist)
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
{
  completedbp_t* bp = new completedbp_t;
  bp->calls = calls;
  bp->callindex = bplist;
  choose2(false,               // non-modal window
    -1, -1, -1, -1,            // position is determined by Windows
    bp,                        // object to show
    qnumber(ccHeader),         // number of columns
    ccWidths,                  // widths of columns
    ccSize,                    // function that returns number of lines
    ccDescription,             // function that generates a line
    ccTitle,                   // window title
    -1,                        // use the default icon for the window
    0,                         // position the cursor on the first line
    NULL,                      // "kill" callback
    NULL,                      // "new" callback
    NULL,                      // "update" callback
    NULL,                      // "edit" callback
    ccEnter,                   // function to call when the user pressed Enter
    ccDestroy,                 // function to call when the window is closed
    NULL,                      // use default popup menu items
    NULL);                     // use the same icon for all line
}
/***************************************************************************
* Function: requestSetBps
*
* requests all our breakpoints be set, then run_requests
***************************************************************************/
void requestSetBps(netnode* node)
{
  indirect_t my_indirect;
  long no_calls = getnodesize(node);
  msg("requestSetBps size: %x\n", no_calls);
  for (int i = 1; i < no_calls; ++i)
  {
    node->supval(i, &my_indirect, sizeof(my_indirect));
    request_add_bpt(my_indirect.caller);
  }
  run_requests();
  return;
}
/***************************************************************************
* Function: requestDelBps
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
*
* requests all our breakpoints be deleted, caller calls run_requests
***********************************************************************/
void requestDelBps(netnode* node)
{
  indirect_t my_indirect;
  long no_calls = getnodesize(node);
  msg("requestDelBps size: %x\n", no_calls);
  for (int i = 1; i < no_calls; ++i)
  {
    node->supval(i, &my_indirect, sizeof(my_indirect));
    request_del_bpt(my_indirect.caller);
  }
  return;
}
/***********************************************************************
* Function: setBps
*
* set all our breakpoints
***********************************************************************/
void setBps(netnode* node)
{
  indirect_t my_indirect;
  long no_calls = getnodesize(node);
  msg("setBps size: %x\n", no_calls);
  for (int i = 1; i < no_calls; ++i)
  {
    node->supval(i, &my_indirect, sizeof(my_indirect));
    add_bpt(my_indirect.caller);
  }
  return;
}
/***********************************************************************
* Function: delBps
*
* delete all our breakpoints
***********************************************************************/
void delBps(netnode* node)
{
  indirect_t my_indirect;
  long no_calls = getnodesize(node);
  msg("delBps size: %x\n", no_calls);
  for (int i = 1; i < no_calls; ++i)
```

图9-23　（续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
    {
      node->supval(i, &my_indirect, sizeof(my_indirect));
      del_bpt(my_indirect.caller);
    }
    return;
}
/***************************************************************************
 * Function: setTargetXref
 *
 * This function serves two purposes. First decides whether to add the
 * call to the completed call/bp list. It also can create the cross
 * reference between the caller and the target.
 ***************************************************************************/
void setTargetXref(dbgOptions* myDbg,long index,indirect_t* myIndirect)
{
  bphitlist_t* entry = myDbg->bplist;
  ulong options = myDbg->options;
  ea_t from = myIndirect->caller;
  ea_t to = myIndirect->target;
  short &flags = myIndirect->flags;
  segment_t* from_seg = getseg(from);
  segment_t* to_seg = getseg(to);

  if (from_seg == to_seg)
  {
    if (options & MAKE_XREFS)
    {
      flags |= XRSETFLAG;
      if (flags & JMPSETFLAG)
        add_cref(from, to, (cref_t)(fl_JN | XREF_USER));
      else
        add_cref(from, to, (cref_t)(fl_CN | XREF_USER));
    }
    entry->push_back(index);
  }
  else // cross segment
  {
    if (to_seg != NULL && !(to_seg->is_ephemeral_segm()))
    {
      flags |= XSEGFLAG;
      if (options & MAKE_XS_XREFS)
        {
          flags |= XRSETFLAG;
          if (flags & JMPSETFLAG)
```

图 9-23 （续）

9

更多资源请点击访问稀.酷客(www.ckook.com)

```
            add_cref(from, to, (cref_t)(fl_JF | XREF_USER));
          else
            add_cref(from, to, (cref_t)(fl_CF | XREF_USER));
        }
        if(options & DISPLAY_XS_BPS)
        {
          entry->push_back(index);
        }
      }
    }
}
/***************************************************************************
* Function: addVTable
*
* Determines if vtable is considered valid. A new vtable is added to
* the vtable netnode. If the vtable already exists. The offset is
* checked against the largest offset recorded for the vtable.
***************************************************************************/
void addVTable(dbgOptions* myDbg, ea_t vtaddr, indirect_t* myIndirect)
{
  ea_t from = myIndirect->caller;
  ea_t to = myIndirect->target;
  ea_t offset = myIndirect->offset;

  segment_t* from_seg = getseg(from);
  segment_t* vt_seg = getseg(vtaddr);
  netnode* vtables = myDbg->vtables;
  ulong options = myDbg->options;

  if (offset || (options & INC_NONOFF_CALLS))
  {
    if (from_seg != vt_seg) // only documenting vtables in from_seg
    {
      return;
    }
    if ( (get_first_dref_to(vtaddr) == BADADDR) ||
      (get_first_dref_from(vtaddr) == BADADDR))
    {
      msg("%x to %x , probably jump table, not vtable [%x]\n",
          from, to, vtaddr);
    }
    else // considered a valid vtable
    {
      ulong tmp = vtables->altval(vtaddr);
      if (tmp == 0) // new vtable
```

图 9-23　（续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
        {
          vtable_t my_vtable;
          int vtable_counter = getnodesize(vtables);
          my_vtable.baseaddr = vtaddr;
          my_vtable.largestOffset = myIndirect->offset;
          vtables->altset(vtaddr, vtable_counter);
          vtables->supset(vtable_counter++, &my_vtable,
                          sizeof(my_vtable));
          setnodesize(vtables, vtable_counter);
          msg("%x NEW VTABLE caller: %x , to: %x\n", vtaddr, from, to);
      }
      else // vtable already defined
      {
          vtable_t tmpVtable;
          vtables->supval(tmp, &tmpVtable, sizeof(tmpVtable) );
          // new offset > old offset
          if (myIndirect->offset > tmpVtable.largestOffset)
          {
            tmpVtable.largestOffset = myIndirect->offset;
            vtables->supset(tmp, &tmpVtable, sizeof(tmpVtable));
          }
        }
      }
    }
  }
/***************************************************************************
 * Function: callback
 *
 * The debugger calls this function when handling any HT_DBG events.
 * The dbgOptions structure is passed to this function allowing the use
 * of previously defined data structures and user options.
 *
 * callback handles 3 types of HT_DBG events
 *
 * dbg_bpt - All breakpoints are handled here. The bp address
 *           is checked to be ours. If not the the process is
 *           suspended. Otherwise:
 *           The instruction is call [eax] with or without an
 *           offset OR anything else.
 *
 *           For everything else 'step into' is requested.
 *           The current bp addresses is saved in last_bp
 *           for the step_into handler
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
 *
 *           With the instruction decoded, both the base and
 *           target can be calculated.
 *           addVTable() & setTargetXref() process if
 *           vtables and cross references are made. The
 *           indirect_t obj is saved with updates.
 *           continue_process() is called
 *
 *      dbg_step_into - All step_into events are handled here. last_bp
 *           is checked. For user caused step_into event
 *           suspend_process() is called.
 *           setTargetXref() deltemines if cross references
 *           are made. The indirect_t obj is saved with updates.
 *           continue_process() is called
 *
 *      dbg_process_exit - This event signifies that the debugger is
 *           shutting down. Brealpoints are cleared and depending on
 *           options, up to three chooser list windows are opened.
****************************************************************************/
int idaapi callback(void* user_data,int notification_code,va_list va)
{
  dbgOptions* my_dbg = (dbgOptions*)user_data;
  netnode* calls = my_dbg->calls;
  ulong options = my_dbg->options;
  static ea_t last_bp = BADADDR;
  ea_t from = BADADDR;
  ea_t vtaddr = BADADDR;
  ea_t to = BADADDR;
  regval_t regval;
  switch (notification_code)
  {
  case dbg_bpt:
    {
      va_arg(va, tid_t);
      from = va_arg(va, ea_t);
      long index = calls->altval(from);
      if (index == 0)
      {
        // not one of our breakpoints
        msg("%x not mine options:0x%x", from, options);
        suspend_process();
        return 0;
      }
```

图 9-23　（续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
            indirect_t my_indirect;
            calls->supval(index, &my_indirect, sizeof(my_indirect));
            // check for call [reg] or call [reg + offset]
            if (my_indirect.call_reg == none)
            {
              last_bp = from;
              request_del_bpt(from);
              request_step_into();
              run_requests();
              break;
            }

            get_reg_val(regname[my_indirect.call_reg], &regval);
            vtaddr = (ea_t)regval.ival;
            // flushes possibly stale memory cache
            invalidate_dbgmem_contents((ea_t)regval.ival,
                                       0x100 + my_indirect.offset);
            to = get_long(my_indirect.offset + vtaddr);
            my_indirect.target = to;
            addVTable(my_dbg,vtaddr, &my_indirect);
            setTargetXref(my_dbg, index, &my_indirect);
            // save completed indirect
            calls->supset(index, &my_indirect, sizeof(my_indirect));
            del_bpt(from);
            continue_process();
            break;
          }
      case dbg_step_into:
          {
            from = last_bp;
            if (from == BADADDR)
            {
              msg("not mine\n");
              suspend_process();
              return 0;
            }
            long index = calls->altval(from);
            get_reg_val("EIP", &regval);
            to = (ea_t)regval.ival;
            indirect_t my_indirect;
            calls->supval(index, &my_indirect, sizeof(my_indirect));
            my_indirect.target = to;
            setTargetXref(my_dbg, index, &my_indirect);
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
        // save completed indirect
        calls->supset(index, &my_indirect, sizeof(my_indirect));

        last_bp = BADADDR;
        continue_process();
        break;
      }
    case dbg_process_exit:
      {
        unhook_from_notification_point(HT_DBG, callback, user_data);
        requestDelBps(calls);
        run_requests();
        register_event(E_PROCEXIT);

        if (options & DISPLAY_INCALLS)
        {
          createIndirectCallWindow(calls);
        }
        if (options & DISPLAY_BPS)
        {
          createCompletedBpWindow(calls, my_dbg->bplist);
        }
        if (options & DISPLAY_VTABLES)
        {
          createVTableWindow(my_dbg->vtables);
        }
        break;
      }
    default:
      break;
  }
  return 0;
}
/**************************************************************************
* Function: getnodesize
*
* returns size (including location 0)
**************************************************************************/
long getnodesize(netnode* node)
{
  return node->altval(NODE_COUNT);
}
/**************************************************************************
* Function: getobjcount
*
```

图 9-23　（续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
 * returns number of items in the netnode not counting invalid slot 0
 * see data structure documentation at top of file
 ***********************************************************************/
long getobjcount(netnode* node)
{
  return node->altval(NODE_COUNT)-1;
}
/***********************************************************************
 * Function: setnodesize
 *
 * store netnode size
 ***********************************************************************/
bool setnodesize(netnode* node, long size)
{
  return node->altset(NODE_COUNT, size);
}
/***********************************************************************
 * Function: fillIndirectObj
 *
 * Determines if instruction is call [reg+offset], call [reg], or other
 * Fills in the indirect_t struct.
 ***********************************************************************/
void fillIndirectObj(indirect_t &currcall)
{
  currcall.caller = cmd.ea;
  currcall.target = BADADDR;
  currcall.call_reg = none;
  currcall.offset = 0;
  if (cmd.itype == NN_callni)
  {
    // need a single opcode
    ushort no_operands = 0;
    while(no_operands < UA_MAXOP &&
         cmd.Operands[no_operands].type != o_void)
    {
      no_operands++;
    }
    if (no_operands == 1)
    {
      if (cmd.Operands[0].type == o_phrase)
      {
        currcall.call_reg = cmd.Operands[0].reg;
      }
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
      else if (cmd.Operands[0].type == o_displ)
      {
          currcall.call_reg = cmd.Operands[0].reg;
          currcall.offset = cmd.Operands[0].addr;
        }
      }
    }
    else if (cmd.itype & NNJMPxI) // jmp?
    {
      currcall.flags |= JMPSETFLAG;
    }
  }
/***************************************************************************
* Function: findIndirectCalls
*
* This function through a segment for indirect calls and jmps
* NN_callfi , NN_callni, NN_jmpfi , NN_jmpni
* then it pkgs it in a inidirect_t struct and stores in the netnode
***************************************************************************/
void findIndirectCalls(segment_t* seg, netnode* node)
{
  ea_t addr = seg->startEA;
  ulong counter = getnodesize(node);
  while ( (addr < seg->endEA) && (addr != BADADDR))
  {
    flags_t flags = getFlags(addr);
    if (isHead(flags) && isCode(flags))
    {
      if (ua_ana0(addr) != 0)
      {
        switch (cmd.itype)
        {
        case NN_callfi:
        case NN_callni:
        case NN_jmpfi:
        case NN_jmpni:
          {
            if (get_first_fcref_from(cmd.ea) == BADADDR &&
              get_first_dref_from(cmd.ea) == BADADDR) //no fwd xref
            {
              indirect_t currcall;
              fillIndirectObj(currcall);
              node->altset(cmd.ea, counter); // altval keyed by addr
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
                node->supset(counter++, &currcall, sizeof(currcall));
              }
              break;
            }
          default:
            break;
          }
        }
      }
      addr = next_head(addr, seg->endEA);
    }
    setnodesize(node, counter);
    return;
}

void closeListWindows(void)
{
  close_chooser(icTitle);
  close_chooser(ccTitle);
  close_chooser(vtTitle);
}
/***************************************************************************
* Function: register_event
*
* This function serves as an interface to three semaphores in the form
* of event messages. IDA Pro is single threaded and is non reentrant.
* True concurrency requirements such as mutexes and atomic operations
* are not needed.
*
* The caller reports an event and this function adjusts the semaphores
* and can release resources when needed.
* semaphores are tied to the
*     netnode*       calls   - all indirect calls
*     netnode*       vtables - all vtables
*     bphitlist_t*   bplist  - bp hits, an index list into
*                                       netnode* call
***************************************************************************/
void register_event(ulong rEvent)
{
  static long dbgState = 0;
  static long semcall = 0;
  static long semxref = 0;
  static long semvtable = 0;

  switch (rEvent)
```

图 9-23 (续)

更多资源请点击访问稀.酷客(www.ckook.com)

```
    {
    case E_START:
      {
        closeListWindows();
        if (gDbgOptions.calls)
      {
        gDbgOptions.calls->kill();
      }
      if (gDbgOptions.vtables)
      {
        gDbgOptions.vtables->kill();
      }
      if (gDbgOptions.bplist)
      {
        gDbgOptions.bplist->~qvector();
      }
      semcall = semxref = dbgState = semvtable = 0;
      break;
    }
  case E_CANCEL:
    {
      semcall = semxref = dbgState = semvtable = 0;
      gDbgOptions.calls->kill();
      gDbgOptions.vtables->kill();
      gDbgOptions.bplist->~qvector();
      break;
    }
  case E_OPTIONS:
    {
      if((~gDbgOptions.options) >> 15)
      {
        dbgState++;
        semcall++;
        semxref++;
        semvtable++;
      }
      if (gDbgOptions.options & DISPLAY_INCALLS)
      {
        semcall++;
      }
      if (((gDbgOptions.options & DISPLAY_BPS) >> 1) && dbgState)
      {
        semcall++;
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
        semxref++;
      }
      if (((gDbgOptions.options & DISPLAY_VTABLES) >> 5) && dbgState)
      {
        semvtable++;
      }
      break;
    }
  case E_HOOKFAIL:
    {
      dbgState = semvtable = semxref = 0;
      break;
    }
  case E_PROCFAIL:
    {
      // note: call window may be open
      delBps(gDbgOptions.calls);
      semcall--;
      dbgState = semvtable = semxref = 0;
      unhook_from_notification_point(HT_DBG, callback, &gDbgOptions);
      break;
    }
  case E_DWCALL:
    {
      Semcall--;
      if (!semcall)
      {
        gDbgOptions.calls->kill();
      }
      break;
    }
  case E_DWXREFS:
    {
      Semxref--;
      Semcall--;
      if (!semcall)
      {
        gDbgOptions.calls->kill();
      }
        if(!semxref)
        {
          gDbgOptions.bplist->~qvector();
        }
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
        break;
      }
    case E_DWVTABLE:
      {
        Semvtable--;
        if (!semvtable)
        {
          gDbgOptions.vtables->kill();
        }
        break;
      }
    case E_PROCEXIT:
      {
        dbgState = 0;
        semcall--;
        semvtable--;
        semxref--;
        if(!semxref)
        {
          gDbgOptions.bplist->~qvector();
        }
        if (!semcall)
        {
          gDbgOptions.calls->kill();
        }
        if (!semvtable)
        {
          gDbgOptions.vtables->kill();
        }
        break;
      }
    default:
      {
        msg("ERROR UNKNOWN EVENT\n");
        msg("%s dbg:%d scall:%d sxref:%d svtable:%d \n",
             "ERROR", dbgState, semcall, semxref, semvtable);
        break;
      }
    }
}
/**************************************************************************
* Function: run
*
```

图 9-23 （续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
* run is a plugin_t function. It is executed when the plugin is run.
* This function brings up the UI, collects data and sets the debugger
* callback.
*   arg - defaults to 0. It can be set by a plugins.cfg entry. In this
*         case the arg is used for debugging/development purposes
* ;plugin displayed name    filename        hotkey    arg
* indirectCalls_dbg         indirectCalls   Alt-F8    0
* indirectCalls_unload      indirectCalls   Alt-F9    415
*
* Thus Alt-F9 runs the plugin with an option that will unload it.
* This allows (edit/recompile/copy) cycles.
*************************************************************************/
void run(int arg)
{
  char nodename_calls[] = "$ indirect calls";
  char nodename_vtables[] = "$ vtables";
  ea_t curraddr = get_screen_ea();
  segment_t* my_seg = getseg(curraddr);
  char* format;
  short checkbox = DISPLAY_INCALLS | DISPLAY_BPS | DISPLAY_VTABLES;
  short radiobutton = 0;
  int start_status;

  register_event(E_START);

  if(arg == 415)
  {
    PLUGIN.flags |= PLUGIN_UNL;
    msg("Unloading plugin ...\n");
    return;
  }

  netnode* calls = new netnode;
  netnode* vtables = new netnode;
  bphitlist_t *hitlist = new bphitlist_t;
  if (calls->create(nodename_calls) == 0)
  {
    calls->kill();
    msg("ERROR: creating netnode %s\n", nodename_calls);
    return;
  }
  if (vtables->create(nodename_vtables) == 0)
  {
    msg("ERROR: creating netnode %s\n", nodename_vtables);
    calls->kill();
```

图 9-23 （续）

9

更多资源请点击访问稀.酷客(www.ckook.com)

```
    vtables->kill();
    return;
  }
  calls->altset(NODE_COUNT,1); // position 0 is not used
  vtables->altset(NODE_COUNT,1); // position 0 is not used

  findIndirectCalls(my_seg, calls); // finds jmps/calls

  ulong format_size = sizeof(preformat)+9;
  format = (char*)qalloc(format_size);
  qsnprintf(format, format_size, preformat, getobjcount(calls));

  int ok = AskUsingForm_c(format, &radiobutton, &checkbox); // UI

  gDbgOptions.calls = calls;
  gDbgOptions.vtables = vtables;
  gDbgOptions.bplist = hitlist;
  gDbgOptions.options = checkbox;

  register_event(E_OPTIONS);

  if (!ok)
  {
    msg("user canceled, exiting, unloading\n");
    register_event(E_CANCEL);
    PLUGIN.flags |= PLUGIN_UNL;
    return;
  }
  // debugger closing this window, now only open for non debugger
  if ((checkbox & DISPLAY_INCALLS) && (radiobutton == 1))
  {
    createIndirectCallWindow(calls);
  }
  if (radiobutton == 1)
    return; // only collect data
  // the hook is created here. callback() will receive HT_DBG
  // events only. gDbgOptions is passed to callback()
  // it is global so termination funcs have access
  if (!hook_to_notification_point(HT_DBG, callback, &gDbgOptions))
  {
    warning("Could not hook to notification point\n");
    register_event(E_HOOKFAIL);
    return;
  }

  requestSetBps(calls);
  start_status = start_process(NULL, NULL, NULL);

  if (start_status == 1) // SUCCESS
  {
    msg("process started ...\n");
```

图 9-23　（续）

更多资源请点击访问稀.酷客(www.ckook.com)

```
    return;
  }
  else if (start_status == -1)
  {
    warning("Sorry, could not start the process");
  }
  else
  {
    msg("Process start canceled by user\n");
  }
  register_event(E_PROCFAIL);
  return;
}
/***************************************************************************
* Function: init
*
* init is a plugin_t function. It is executed when the plugin is
* initially loaded by IDA
***************************************************************************/
int init(void)
{
  if (ph.id != PLFM_386) // intel x86
    return PLUGIN_SKIP;
  return PLUGIN_OK;
}
/***************************************************************************
* Function: term
*
* term is a plugin_t function. It is executed when the plugin is
* unloading. Typically cleanup code is executed here.
* The unhook is called as a safety precaution.
* The windows are closed to remove the choose2() callbacks
***************************************************************************/
void term(void)
{
  unhook_from_notification_point(HT_DBG, callback, &gDbgOptions);
  closeListWindows();
  return;
}
char comment[] = "indirectCalls";
char help[] = "This plugin looks\nfor indirect\ncalls\n";
char wanted_name[] = "indirectCalls";
char wanted_hotkey[] = "Alt-F7";
/* defines the plugins interface to IDA */
```

图 9-23 (续)

更多资源请点击访问稀.酷客(www.ckook.com)

```
plugin_t PLUGIN =
{
  IDP_INTERFACE_VERSION,
  0,                  // plugin flags
  init,               // initialize
  term,               // terminate. this pointer may be NULL.
  run,                // invoke plugin
  comment,            // comment about the plugin
  help,               // multiline help about the plugin
  wanted_name,        // the preferred short name of the plugin
  wanted_hotkey       // the preferred hotkey to run the plugin
};
```

图 9-23 （续）

9.11 插件开发和调试策略

本节为编写和调试插件提供一些有用的策略。Visual Studio 调试器相对来讲工作良好而且使用方便，它能附加到或脱离 IDA 进程。

9.11.1 创建一个新的 IDA 开发目录

把 IDA Pro 安装目录复制到一个新位置，保留原有目录。选择一个较短的目录名以减少敲键次数，比如使用：

C:\ida_dev

进入插件目录，在这里是 **C:\ida_dev\plugins**，创建一个名为 **plugin_backup** 的新目录。把插件目录的内容复制到 **plugin_backup** 目录，然后删除所有与当前插件开发无关的插件。例如，如果开发一个 32 位插件，那么所有的 64 位插件都应该被删除。如果你准备开发使用调试器的插件，则需要保留调试器。

移除插件有几个目的。

- 当使用-z 调试选项（稍后简单介绍）时，消息窗口将包含比较少无关调试消息。
- 移除插件可以释放潜在的热键。我们可能想设置多个热键以传递不同的参数给正在开发的插件。
- 若没有不必要插件的初始化过程，将减少启动时间。

9.11.2 编辑配置文件

编辑配置文件后别忘了测试，这意味着移除不必要的热键绑定，增加其他有用的热键。下面的配置位于 idagui.cfg 中。

1. 使用未压缩的数据库

IDA 操作未压缩的数据库时可以加速进程启动和停止。在开发插件时，使用 IDB 文件一定

更多资源请点击访问稀.酷客(www.ckook.com)

要进行备份。为了操作未压缩的数据库，需遵循下面的步骤。

(1) 把 IDB 复制到新目录

(2) 在此目录下创建一个批处理文件。这个文件将执行 IDA 的开发副本。此时支持设置命令行参数。一个示例文件 idadev.bat 可能包含以下内容：

```
C:\ida_dev\idag.exe myidb.idb
```

(3) 运行批处理文件。退出，然后选择 **Don' t pack database**。你会看到一个警告，请选择 **Yes**，IDB 文件将不再在目录下。在这里，也可以改变选项来移除警告。

(4) 在 idagui.cfg 中设置下面的选项：

```
ASK_EXIT_UNPACKED = NO          // Ask confirmation if the user
                                // wants to exit the database without
                                // packing it
ASK_EXIT = NO                   // Ask confirmation if the user
                                // wants to exit
```

(5) 你可以选择为“Abort”分配一个热键。

(6) 在 ida.cfg 中设置下面的选项：

```
PACK_DATABASE = 0               // 0 - don't pack at all
                                // 1 - pack database (store)
                                // 2 - pack database (deflate)
```

使用未压缩的数据库的好处是加快开始和关闭时间。

注意，这些选项只应为 IDA 的开发副本而设，正常使用时不建议这些选项。

2. 启用不保存退出功能

另一个可选的策略是在开发时不保存 IDB 文件。未压缩方法仍会保存文件。如果你的插件崩溃，文件和数据库的状态可能未知。为了启用“操作而不保存”功能，执行如下操作。

(1) 把 IDB 复制到新目录。

(2) 在此目录下创建一个批处理文件。这个文件将执行 IDA 的开发副本。这里同样支持设置命令行参数。一个示例文件 idadev.bat 可能包含如下内容：

```
C:\ida_dev\idag.exe myidb.idb
```

(3) 为“Abort”分配一个热键：

```
"Abort" = "Alt-Z" // Abort IDA, don't save changes
```

执行“Abort”时，会弹出一个确认对话框。看上去没有选项可以避免它，不过你按 **Y** 就可以退出了。

这个策略有它的好处，主要优点是在每次测试循环中都有一个已知的启始 IDB。缺点是延长了启动时间。这个策略使得关闭时间可以被忽略，因为 IDA 并不存储和压缩数据库。总的延迟视 IDB 的大小而定。

9

更多资源请点击访问稀.酷客(www.ckook.com)

3. 插件参数

可以向插件传递参数，这样可以控制和改变插件的行为。plugis.cfg 文件定义热键和传递给插件的参数。

IDA API 不允许插件卸载。许多重要的插件将建立回调函数或钩子并驻留在内存中。这避免更新的重编译的插件副本覆盖当前的插件。你可以选择退出 IDA，但也可以选择使用插件参数。下面的内容来自 plugins.cfg 文件：

```
indirectCalls              indirectCalls    Alt-F8   0
indirectCalls_unload       indirectCalls    Alt-F9   415
```

处理参数的对应代码是：

```
if(arg == 415)
{
  PLUGIN.flags |= PLUGIN_UNL;
  msg(" Unloading plugin ...\n");
  return;
}
```

PLUGIN_UNL 标志可以在任何时间设置，但 IDA 只在 run 函数退出时检查它。用参数 415 调用插件，就会设置 PLGUIN_UNL 标志位。插件应该确保它去除了所有的通知钩子，移除了所有的回调函数。插件在 term 函数中使用前面的代码去除钩子。

除了卸载插件外，参数也可以用于其他用途。可以定义一个参数来设置全局调试标志，允许有多个输出函数。比如，某个 arg 可以把结果输出到消息窗口，而另外一个 arg 可以创建选择列表框。在 IDC 里可以用 RunPlugin 函数发送参数。

```
RunPlugin("indirectCalls", 415);
```

4. 编写脚本以协助插件开发

在编写插件前，先用脚本来测试概念或原型是非常有用的。这时，IDAPython 就非常有用了，因为它封装了许多 API 调用。当然，也可以用 IDC，尽管它缺少许多高级 API，但 IDC 总是可用的（不像 IDAPython 插件还需另行安装）。

在开发和测试间接调用插件期间，使用了 IDC 脚本。这个插件对自己的许多逻辑使用了交叉引用数据。为了测试和验证插件及其建立的理论的健壮性，特编写了一个脚本，下面就是这个脚本。

```
#include <idc.idc>
static decode_xtype(xtype)
{
  if (xtype & XREF_USER)
  {
    Message("XREF_USER");
    xtype = xtype & ~XREF_USER;
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
    }
    if (xtype == fl_CF)
      Message("fl_CF Call Far");
    else if (xtype == fl_CN)
      Message("fl_CN Call Near");
    else if (xtype == fl_JF)
      Message("fl_JF Jump Far");
    else if (xtype == fl_JN)
      Message("fl_JN Jump Near");
    else if (xtype == fl_F)
      Message("fl_F Ordinary flow");
    else if (xtype == dr_O)
      Message("dr_O Offset");
    else if (xtype == dr_W)
      Message("dr_W Write");
    else if (xtype == dr_R)
      Message("dr_R Read" );
    else if (xtype == dr_T)
      Message("dr_T Text (names in manual operands)");
    else if (xtype == dr_I)
      Message("dr_I Informational");
}
static lookup_from_ref(void)
{
   auto from, current_code, current_data, no_cxrefs, no_dxrefs;
   from = ScreenEA();
   no_cxrefs = 0;
   no_dxrefs = 0;
   Message("%x [from] xrefs\n", from);
   current_code = Rfirst0(from);
   while(current_code != BADADDR)
   {
      no_cxrefs++;
      Message(" %x CODE (0x%x) ",current_code, XrefType());
      decode_xtype(XrefType());
      Message("\n");
      current_code = Rnext0(from, current_code);
}
   current_data = Dfirst(from);
   while(current_data != BADADDR)
   {
      no_cxrefs++;
      Message(" %x DATA (0x%x) ",current_data, XrefType());
      decode_xtype(XrefType());
      Message("\n");
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
        current_data = Dnext(from, current_data);
      }
      if ((no_cxrefs + no_dxrefs) == 0)
        Message(" NONE\n");
    }
    static lookup_to_ref(void)
    {
      auto to, current_code, current_data, no_cxrefs, no_dxrefs;
      to = ScreenEA();
      no_cxrefs = 0;
      no_dxrefs = 0;
      Message("%x [to] xrefs\n", to);
      current_code = RfirstB0(to);
      while(current_code != BADADDR)
      {
        no_cxrefs++;
        Message(" %x CODE (0x%x) ",current_code, XrefType());
        decode_xtype(XrefType());
        Message("\n");
        current_code = RnextB0(to, current_code);
        if (current_code != BADADDR && no_cxrefs > 7)
        {
          Message(" TOO MANY (%d) CODE xrefs ...\n", no_cxrefs);
          current_code = BADADDR;
        }
      }
      current_data = DfirstB(to);
      while(current_data != BADADDR)
      {
        no_dxrefs++;
        Message(" %x DATA (0x%x) ",current_data, XrefType());
        decode_xtype(XrefType());
        Message("\n");
        current_data = DnextB(to, current_data);
        if (current_data != BADADDR && no_dxrefs > 7)
        {
          Message(" TOO MANY (%d) DARA xrefs ...\n", no_dxrefs);
          current_data = BADADDR;
        }
      }
      if ((no_cxrefs + no_dxrefs) == 0)
        Message(" NONE\n");
    }
    static main(void)
    {
```

更多资源请点击访问稀.酷客(www.ckook.com)

```
    AddHotkey("Shift-F7", "lookup_to_ref");
    AddHotkey("Shift-F8", "lookup_from_ref");
}
```

这个脚本为 lookup 函数绑定了热键，并在消息窗口列出代码和数据的交叉引用。当 IDA 使用 xref.idc 时，其输出格式不利于快速阅读。下面是 my_xref.idc 输出内容的示例，包括它处理的指令列表。

```
.text:030BDF13      call      dword ptr [eax+18h] ; my_xref.idc
30bdf13 [from] xrefs
    30cba25 CODE (0x13) fl_JN Jump Near
30bdf13 [to] xrefs
    NONE
```

这个脚本由 ida.idc 加载。当一个脚本被包含在 ida.idc 里面时，此脚本的 main 函数并不会被执行，但插件中的函数可用。因此，热键其实是在 ida.idc 内绑定的，正如下面代码所显示的。

```
#include <my_xrefs.idc>

static main(void) {
//
//      This function is executed when IDA is started.
//
//      Add statements to fine-tune your IDA here.
//
  AddHotkey("Shift-F7", "lookup_to_ref");
  AddHotkey("Shift-F8", "lookup_from_ref");
}
```

9.12 加载器

加载器负责识别文件格式和创建适当的区段。分析通常由处理器模块执行。加载器的功能如它名字所示，只是把二进制文件加载到 IDA 中。

SDK 模块目录里有多个带源代码的处理器模块。还有一些公开发布的加载器。NSDLDR 是由 Dennis Elser 编写的 Nintendo DS ROM 文件加载器(http://www.openrce.org/downloads/details/56/NDSLDR)。这个加载器相对比较简单，其代码也易于阅读。

加载器输出 loader.hpp 中定义的 loader_t 结构。

```
struct loader_t
{
  ulong version;          // api version, should be IDP_INTERFACE_VERSION
  ulong flags;            // loader flags
  accept_file;            // checks the input format. Shows up in the
                          // "load file" dialog box
  load_file;              // loads file into database
```

9

更多资源请点击访问稀.酷客(www.ckook.com)

```
    save_file;              // can create output file from database
    move_segm;              // moves segment for relocation or rebasing
    init_loader_options;    // initialize user configurable options
};
```

9.13　处理器模块

处理器模块执行实际的反汇编和二进制分析。因为支持超过 50 个系列的处理器，所以覆盖了绝大多数的主流 CPU。不过，许多嵌入式设备还没有对应的模块。发展更小的设备是为了获得更少的消耗，而且这些设备的结构更简单。这些设备从标准的微控制器到稀有的、能在音频和视频设备上受限运行的芯片应有尽有。不过，如果有固件，就会有人尝试逆向分析它。

SDK 有许多处理器模块的源代码，这些处理器模块涵盖曾经流行的 Atmel AVR 芯片到经典的 z80。对每个主要的结构，多数模块使用同样的文件命名约定，以利模块之间的比较和对比。模块使用的结构在 idp.hpp 中定义。主要的结构是 `processor_t`、`asm_t` 和 `instruct_t`。

或许你不想为解码微小的硅片而写模块，但可以考虑虚拟机。从嵌入式设备到软件保护和 crackme，虚拟机变得越来越流行。不管你是有意为硅片还是想象中的硅片编写模块，Rolf Rolles 的文章 *Defeating HyperUnpackMe2 With an IDA Processor Module* 都是必须要读的，特别是它的附录 B。(http://www.openrce.org/articles/full_view/28)

9.14　第三方脚本插件

我们不必受限于编写 IDC 脚本或用 C++编写完备的插件，第三方脚本编写插件提供了另外的选择，它们通常用 SWIG 封装许多 IDC 和 SDK 函数。

像 Python 和 Ruby 这样的脚本语言使我们有机会访问大型代码库，但或许更重要的是，它们具有更好的内置数据类型。当前有两种脚本语言可供选择，即以 IDAPython 形式出现的 Python 和以 IdaRub 形式出现的 Ruby。

第一个脚本编写插件可能是 IDAPerl，但可惜地是它不能下载了，且没有人再为它提供支持。

9.14.1　IDAPython

IDAPython(http://d-dome.net/idapython)由 Gergely Erdélyi 编写，它是一个非常流行的 IDA 插件。新版 IDAPython 着眼于封装函数的覆盖范围以及增加新的 SDK 函数，其源代码在 Darg 仓库中可以得到。

支持平台

IDAPython 可以运行在 Windows 或 Linux 下，有人反映新的测试版可以在 Mac OS X 下运行。

在 Windows 下安装很简单。Python2.4 或 2.5 下有编译好的 IDAPython 版本。除非你有特别的理由非要使用 2.4 版，否则请安装 2.5 版(http://www.python.org/download/)。

请下载合适版本的插件，我一般用测试版，因为它们封装了更多的函数。测试版位于 http://

更多资源请点击访问稀.酷客(www.ckook.com)

ode.google.com/p/idapython/。

解压缩之后，安装其实就是把文件复制到适当位置的过程，而此处就是把 python 目录复制到 IDA Pro 安装目录，从解压缩后的插件目录把 python.plw 复制到 IDA Pro 的插件目录。这样插件就安装好了，下次 IDA 启动后就可以使用。

可以下载或在线浏览函数参考手册(www.d-dome.net/idapython/reference/)。它是由 epydoc从源代码中生成的。

9.14.2 IDARub

IDARub（http://www.metasploit.com/users/spoonm/idarub/）如它的名字所示，使用 Ruby 作为它的脚本语言。IDARub 由 Spoonm 编写。Ruby 在安全工具编写中也非常流行，最著名的是 MetasploitFramework 工具(www.metasploit.com)。

IDARub 当前版本是 2006 年 8 月 1 日发布的 0.8 版本。因为它是在 SDK 4.9 下编译的，所以它在 IDA 后续版本中仍可工作。不过，SDK 4.9 以后增加的函数就用不上了。

虽然 IDAPython 更流行，且有更多的支持，但 IdaRub 中也有几个独有的特性。比如：

- 远程网络访问
- 控制台
- Sweet 演示

Sebastian Porst 编写了 Rublib（http://www.the-interweb.com/serendipity/index.php?/archives/91-RubLib-0.04.html），它被描述成 IdaRub 的高层 API。当前版本是 0.04，其中包含超过 160 个帮助函数。

9.15 常见问题

问： 我可以编写多线程插件么？

答： IDA Pro 是单线程这一点确切无疑，对数据库的所有访问都必须以串行方式进行。有一些多线程插件的例子。Spoonm 编写的 IdaRub 就创建了一个隐藏的窗口和处理程序。源代码可从这儿得到：http://www.metasploit.com/users/spoonm/idarub。

问： 我的插件把消息输出到消息窗口，但消息窗口好像只能保存 2000 行消息，我可以增加缓冲区的大小吗？

答： 如果设置 IDALOG 环境变量 `set IDALOG=mylog.txt`，IDA 就可以把消息重定向到日志文件。

问： 列表框挺有用的，但我可以使用图形引擎完成输出吗？

答： SDK 有一个示例插件 ugraph，它创建图形视图。在 SDK 的 graph.hpp 文件中包含与图形创建相关的类。

更多资源请点击访问稀.酷客(www.ckook.com)