

SSL 与 TLS

Eric Rescorla 编著

中国电力出版社

书 名：SSL与TLS
作 者：Eric Rescorla
译 者：崔凯等
出版社：中国电力出版社
出版日期：2002
字 数：556.0 开 本：16
ISBN 7-5083-1093-4/TP.356
定 价：40.00

序　　言

安全套接层协议SSL (Secure Socket Layer) 是世界上部署最为广泛的安全协议。每一种商业浏览器和服务器都在其内部使用SSL来支持安全的Web交易。当你使用“安全的”Web页面进行联机采购（2000年此类交易的价值大概有200亿美圆）时，几乎可以肯定，你是在使用SSL。

尽管SSL最常见的用途是保证Web通信的安全，但实际上它也是一种相当通用的协议，适用于保护种类繁多的各种通信数据的安全。其中的一些，如文件传输(FTP)、远程对象存取(RMI、CORBA、IIOP)、E-mail传输(SMTP)、远程终端服务(Telnet)以及目录存取(LDAP)业已成为使用SSL或其后继TLS (Transport Layer Security) 来保障安全的一部分应用。

保证所有这些协议安全所耗费的精力使我们吸取了许多重要的教训。首先，要想很好地使用SSL/TLS保证一种协议的安全，就要求对SSL/TLS的工作原理有着相当扎实的理解。我们不可能简单的将SSL/TLS当作黑箱对待，指望它能够在使用时神奇地提供所需要的安全。

其次，尽管每种应用稍有不同，但似乎对每种想保证其安全的应用来说都有一组共同的安全问题。例如，我们通常要设法找出某种让一种应用协议中不安全与安全版本和平共处的方法。尽管对这些问题来说没有一致的解决方案，但是专门研究安全的团体还是正在着手开发一组使用SSL/TLS来解决此类问题的公共技术。

我们常常可以稍加修改就能将这些技术应用于一种新的应用协议中。从本质上讲，我们已开发了一套用于保证协议安全的设计模式(design pattern)。保证系统安全的很大一部分工作就是识别出与正在使用的系统最为匹配的模式，然后再采用相应的技术。

本书的意图就是针对这两方面的需求进行讲解。在读完这本书之后，你应当了解即便不是所有也是绝大多数使用SSL/TLS设计安全系统所需的知识。你将会了解足以理解SSL/TLS所能提供以及所不能提供的各种安全特性的知识。此外，还会熟悉使用SSL/TLS的常见设计模式，并随时可以在新的情况下应用这些模式。

本书所提供的内容

本书适用于任何想要理解和使用SSL/TLS的读者。

对设计者来说，本书不但提供了已经付诸使用的技术库，还提供了使用SSL/TLS来设计系统的有关信息。

对于使用SSL/TLS编程的程序员来说，本书提供了有关函数库底层的工作机理，以及你所调用的函数实际完成的工作内容。理解这些细节对于获得可接受及可预料的应用性能非常关键。

对于SSL/TLS的实现者来说，本书可以作为标准之外的辅助资料，起到解疑释惑的作用。

面向的读者群

本书假定你对TCP/IP协议的工作原理有着基本的了解。那些对TCP/IP不熟的读者最好还是能够参考一本讲解TCP/IP的好书。TCP/IP Illustrated, 第一卷[Stevens 1994]是一本不错的选择。RFC791[Postel1991a]、RFC792[Postel1991b], 以及RFC793[Postel1991c]提供了有关TCP/IP的终极参考。尽管无须深刻理解TCP/IP也能理解本书中的一些内容, 但是不理解TCP的行为就很难明白大量与性能有关的讨论。

由于SSL/TLS是一种密码协议 (cryptographic protocol), 所以要想正确地理解有关内容则需要熟悉密码学算法 (cryptographic algorithm), 其中包括公用密钥加密算法 (public key cryptography)、对称加密算法 (symmetric cryptography) 以及摘要 (digest) 算法。第一章将介绍密码学与通信, 但由于篇幅的限制, 无法提供完整的描述。我们试图含盖理解SSL/TLS所必需的所有加密算法细节。不过, 有兴趣从更广的层面上理解加密算法知识的读者应当参阅一本有关密码学的教程, 如 [Schneier1996a] 或 [Kaufman1995]。

本书的结构

本书是分成两个部分来写的, 这与我们前面所描述的两个目标: 理解协议以及如何使用是一致的。前半部分, 从第1到第6章主要讲述SSL和TLS。我们主要关心的是SSL和TLS工作的技术细节, 并分开讨论它们的安全与性能属性。

而在本书的后半部分, 从第7章到第11章, 讲述了如何使用SSL/TLS来保证安全地应用协议和系统实现。首先讲述使用SSL/TLS的一般性指导, 然后讨论几种已经使用SSL/TLS来保障安全的协议。

第1章——与安全有关的概念, 提供了对密码学与通信安全的介绍, 并着眼于它在SSL/TLS中的应用。如果你已经熟悉通信安全的相关知识, 那么就可以跳过这一章。反之, 就应当仔细阅读本章, 以免到后边不知所云。

第2章——SSL介绍, 粗略概括了SSL/TLS的历史以及它所提供的各种安全特性。此外, 还提供了在编写本书时使用SSL/TLS来保证安全的各种协议的现况。

第3章——SSL基础, 讲述了最常用的SSL/TLS操作模式 (operational mode)。我们从头到尾描述了整个SSL/TLS的连接过程。本章可以让你很好地理解SSL/TLS的实际工作原理。一旦理解了本章的内容, 你就能够轻而易举地理解其他操作模式。

第4章——高级SSL, 讲述了其余主要的操作模式。讲解了会话恢复 (session resumption)、客户端认证 (client authentication), 以及几种当前只在SSL/TLS中才被采用的算法, 如DH/DSS和Kerberos。

第5章——SSL的安全, 描述了SSL所提供的安全裨益, 以及所不能提供的一些个好处 (这些内容甚至更为重要)。前面的章节主要将重点放在工作原理上, 而本章则将重点放在为保证使用SSL/TLS的系统安全所要完成的工作上。

第6章——SSL的性能, 描述了基于TLS系统的性能剖析。众所周知, 安全措施对系统提出了很高的性能要求, 但是理解这种影响仅限于协议特定部分的人却不多。我们将讨论这些

问题，并着眼于在获得更高性能的同时维持良好的安全性。

第7章——使用SSL进行设计，是有关使用SSL/TLS来保证应用协议安全的指南。我们将重点放在识别所需的安全属性上，并深刻理解满足这些属性的设计技术。

第8章——进行SSL编程，讨论了编写使用SSL/TLS的软件所需的常见编程套路（programming idiom）。我们提供了完整的采用OpenSSL和PureTLS工具箱，用C和Java语言编写了的范例程序。

第9章——SSL上的HTTP，讲述了开创SSL的应用。SSL起先是由Netscape设计用来与HTTP一起工作的，我们在这里讲述了完成此类工作的传统方式，同时也讲解了当前建议的替代方式。

第10章——TLS上的SMTP，讲述了使用TLS来保证简单邮件传输协议（SMTP）安全的内容，SMTP是用来传输E-mail的协议。SMTP与TLS并不相称，而本章举例说明了SSL与TLS的一些限制。

第11章——各种方案的对比，描述了其他保证应用安全的方案。SSL/TLS并不总是最好的解决方案，了解何时不去使用它也是了解如何使用某种协议所需的。本章试图带你领略一下其他的选择。我们讨论了除SSL/TLS之外的其他方案：IPSEC、S-HTTP和S/MIME。

如何阅读本书

本书适合各种具有不同技术能力和需求的读者。你可以阅读任何自己感兴趣的章节，也可以根据自己的需要，将重点放在特定的章节上。

协议设计人员

如果你是在设计一种新的应用层协议或是使用SSL/TLS来保证一种现有协议的安全，就应当阅读头一部分第1~6章的内容，以便对SSL/TLS的工作原理有个大致了解。然后再仔细阅读第7章有关SSL/TLS设计原则的指南。如果你不打算实现自己的设计，就可以跳过第8章，但是一定要阅读第9章和第10章。从中你能看到现实世界中的一些例子，了解在实际运用中应当怎样和不应当怎样使用SSL/TLS。在开始设计之前，还应该阅读第11章的内容以确信SSL/TLS适合你的设计，以及有没有其他更好的安全协议可供使用。

应用程序员

如果你使用现有的SSL/TLS工具箱编写应用，就可以放心读完第一部分1至6章的内容。你还应当阅读每章之后的总结，这些章节概括性地讨论了SSL/TLS及其实现技术。这些内容将会提供理解SSL/TLS完成各项工作的足够信息。你应当仔细阅读第7章和第8章，要特别注意第8章所讨论的编程技术。如果你是在SSL上实现HTTP或SMTP的话，还应当阅读与这些协议有关的章节。

SSL/TLS实现者

如果你是在从头实现SSL/TLS，就应当阅读整本书的内容。如果你已经熟悉密码学的话，

就可以跳过第1章。然而，如果对密码学没有具体的了解，则应当通读整章的内容。你应当特别注意第2到第6章的内容，这些章节提供了对SSL/TLS的具体描述，以及创建快速而安全的实现所需要的各种实现技术。

仅仅出于好奇

如果你只是想对SSL/TLS有所了解，则可以随意挑选书中的章节阅读。但如果事先不了解有关密码学的知识，就应该阅读第1章的全部内容。然后再阅读第2到第6章以了解SSL/TLS的工作原理。接着就可以或多或少地阅读其余自己感兴趣的章节。要想了解SSL/TLS与其他安全协议之间的比较的话，第11章或许值得一读。

SSL/TLS的版本

至此，你可能已经厌倦了看到SSL/TLS这个字眼。我们一直使用它来避免谈及我们所意指的确切版本。当前有两个版本的SSL被广泛部署：SSL版本2(SSLv2)和SSL版本3(SSLv3)。TLS是SSLv3的一种变体，由因特网工程任务组(IETF)在1999年加以标准化。除了从名字可以想到的内容之外，SSLv2与SSLv3是两种截然不同的协议，而TLS与SSLv3极其相似。SSLv2实质上已经过时，而在编写这本书的当刻，还没有真正地广泛部署TLS。总而言之，我们将使用SSL这个字眼来互换的指代SSLv3/TLS。当意指某种协议时，我们会具体指明。在个别谈论SSL版本2的实例中，我们将会使用SSLv2。

排版约定

本书包含许多真实SSL或TLS会话的网络跟踪信息。在展示此类跟踪信息时，我们使用等宽字体来显示程序输出(CONSTRUCTED)，使用斜体表示之后插入的注释(Comment)。在网络跟踪信息中显示以十六进制表达协议数据的地方，我们使用等宽粗体(01 02 03)来显示。在以明文来显示加密数据的地方，我们将使用等宽斜体来显示(data)。

正文中，从各项标准(如，因特网RFC)和协议结构定义节选的内容以sans serif字体来显示(helvetica)，而图示中使用可读性好的Times字体。代码片段以等宽字体(int)来显示。在个别特殊情况下，需要折行显示较长的行，这种情况下，我们将在折行的行尾使用符号↓来表示下面是该行的接续文本。

历史文献注解和旁白将会用这种较小的字体缩进来显示。

网络跟踪信息

本书中的网络跟踪信息全都源于真实的会话(session)，大部分都是在作者家里的以太网上捕获的。使用了各种各样的客户端和服务器程序，其中包括OpenSSL、Netscape Navigator、Internet Explorer和qmail。这些跟踪信息是用tcpdump程序捕获，并存储在磁盘上的。文中显示的跟踪信息是用作者编写的SSL解码软件包ssldump产生的。你可以从<http://www.tcpdump.org/>获得tcpdump，从<http://www.rfm.com/ssldump/>处获得ssldump。

源代码

本书包含了许多源代码片段。第8章和第9章的源代码是由作者编写的，©1999-2000。不管出于什么目的，你都可以免费使用和拷贝，但是并不提供任何种类的担保。你可以从作者的Web站点<http://www.rfm.com/ssldump/>获得机器可读的版本。

第5章中的Java源代码是PureTLS Java SSL/TLS实现的一部分。你可以从<http://www.rfm.com/puretls>得到。该源代码具有下列版权。

Copyright (C) 1999, Claymore Systems, Inc.
All Rights Reserved.

ekr@rfm.com Tue May 18 09:43:47 1999

This package is a SSLv3/TLS implementation written by Eric Rescorla
<ekr@rfm.com> and licensed by Claymore Systems, Inc.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. All advertising materials mentioning features or use of this software must display the following acknowledgement:
This product includes software developed by Claymore Systems, Inc.
4. Neither the name of Claymore Systems, Inc. nor the name of Eric Rescorla may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ''AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

附录A中SSL会话缓存的例子出自Ralf S. Engelschall (rse's) 的mod_ssl软件包，你可以从<http://www.modssl.org/>获得该软件包，这里使用的版本是2.6.1-1.3.12。它受下列版权限制。

=====

Copyright (C) 1998-1999 Ralf S. Engelschall. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgment:

"This product includes software developed by
Ralf S. Engelschall <rse@engelschall.com> for use in the
mod_ssl project (<http://www.modssl.org/>)."

4. The names "mod_ssl" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact rse@engelschall.com.

5. Products derived from this software may not be called "mod_ssl" nor may "mod_ssl" appear in their names without prior written permission of Ralf S. Engelschall.

6. Redistributions of any form whatsoever must retain the following acknowledgment:

"This product includes software developed by
Ralf S. Engelschall <rse@engelschall.com> for use in the
mod_ssl project (<http://www.modssl.org/>)."

THIS SOFTWARE IS PROVIDED BY RALF S. ENGELSCHALL ''AS IS'' AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL RALF S. ENGELSCHALL OR HIS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT

NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.

致谢

没有许多人的协助是不可能编写图书的，更不要说是一本技术书籍。作为一种惯例，我想在此感谢其中的一些个人。

我的技术审阅人员不但使我诚实，而且使我的写作尽可能的清晰。Joshua Ball、Joe Balsama、Douglas Barnes、Debasish Biswas、Andrew Brown、Robert Bruen、Megan Conklin、Russ Housley、Paul Kocher、Brian Korver、Chris Kostick、Marcus Leech、Robert Lynch、Joerg Meyer、D.Jay Newman、Tim Newsham、Stacey O'Rourke、Radia Perlman、Mark Schertler、Win Treese、Tom Weinstein和Tom Woo，这些人都参加了对手稿的各个章节审阅的工作。

许多人时常、甚至在不知道的情况下，慷慨地解答了我在写作中遇到的技术问题。我要特别感谢John Banes、Steve Bellovin、Burt Kaliski、Paul Kocher、Bodo Moeller、Dan Simon 和Robert Zuccherato，感谢你们填补了我的知识空白。尤其要感谢Terence Spies，是他就整个手稿提出了宝贵的批评，并解答了大量有关微软SSL实现的技术问题。

本书大量使用OpenSSL来产生SSL通信演示。如果没有Eric Young和Tim Hudson在创建SSLeay中的艰苦工作，以及OpenSSL团队在Eric Young为寻求更好的发展而离开后对OpenSSL进行的维护和改进，OpenSSL就不会存在。

来自Alchemy/Nokia的Brian Korver和Stacey O'Rourke在收集第6章的一些性能数据时提供了大量帮助。他们不但允许我使用他们的机器和网络，还亲切地以各种晦涩的方式重新对环境进行配置，以便我能够特定应用情景。

尽管从未谋面，现已去世的W.Richard Stevens仍然对本书有着巨大的贡献。使用网络跟踪信息来演示协议的思想就是从Stevens的那套优秀的TCP/IP Illustrated中汲取的。贯穿本书，我努力模仿他清晰易读的风格（只获得了有限的成功）。当然，没有出版社就不可能出版任何东西，而我非常高兴能够与Addison-Wesley一起工作。我特别要感谢Mary Hart，是她提议这个项目，并忍受着漫无天日的拖延，而本书也从薄薄的200页激增并突破400页大关。我还要感谢我的出品经理Kathy Glidden，是她在耐心地回答我无休止的排版问题。

尽管这些人没有直接对本书的写作做出贡献，但我还是要感谢Allan Schiffman、Marty Tenenbaum和Jay Weber。是Jay和Marty在我只有天分而没有经验时给了我机会。从我认识他的八年来，Allan传授给了我不可估量的计算机科学知识。同样，假如没有我的父母教育我如何思考，我也根本不会取得这些成绩。

写书是一项艰苦的任务，在此过程中，是Jennifer Gates扮演了保证我头脑清醒的主要角色。在过去的几年中，她和她的丈夫Lee许多次提供了超出义务之外的友善和友谊。

另一种使自己持续保持头脑清晰的因素就是三项全能。Kevin Joyce和Kyle Welch提供了宝贵的保持我这种习惯的建议和动力。

最后我还要感谢Lisa Dusseault和Kevin Dick。Lisa和Kevin他们两个阅读整个手稿，并帮我将初稿转变成易读的篇章。没有他们，我极有可能根本无法完成这些工作，而且文章也要比现在糟糕的多。

照排版是由笔者使用James Clark的Groff软件包制作而成的。欢迎读者给我电子邮件提出批评和建议。

Mountain View, CA Eric Rescorla

September 2000 ekr@rtfm.com

目 录

序 言

第 1 章 与安全有关的概念	1
1.1 介绍	1
1.2 因特网威胁模型	1
1.3 角色	2
1.4 安全目标	2
1.5 必要的装备	5
1.6 组合起来使用	12
1.7 简单的安全消息系统	13
1.8 简单的安全通道	14
1.9 出口形式	19
1.10 实际的加密算法	20
1.11 对称加密：序列密码	21
1.12 对称加密：分组密码	22
1.13 摘要算法	26
1.14 密钥的确立	26
1.15 数字签名	29
1.16 MAC	31
1.17 密钥长度	31
1.18 总结	33

第 2 章 SSL 介绍	34
2.1 简介	34
2.2 标准与标准化组织	34
2.3 SSL 概述	35
2.4 SSL/TLS 的设计目标	35
2.5 SSL 与 TCP/IP 族	36
2.6 SSL 的历史	37
2.7 用于 Web 的 SSL	40
2.8 在 SSL 上构建一切	41
2.9 获得 SSL	41
2.10 总结	43

第 3 章 SSL 基础	44
3.1 介绍	44
3.2 SSL 概述	44
3.3 握手	44
3.4 SSL 记录协议	47
3.5 各种消息协同工作	48
3.6 一次真实的连接	49
3.7 其他的连接细节	50
3.8 SSL 规范语言	52
3.9 握手消息结构	54
3.10 握手消息	55
3.11 密钥导出	65
3.12 记录协议	69
3.13 警示与关闭	71
3.14 总结	73
第 4 章 高级 SSL	74
4.1 介绍	74
4.2 会话恢复	74
4.3 客户端认证	75
4.4 临时 RSA	76
4.5 再握手	77
4.6 服务器网关加密	77
4.7 DSS 与 DH	79
4.8 椭圆曲线加密套件	80
4.9 Kerberos	80
4.10 FORTEZZA	81
4.11 小结	82
4.12 会话恢复细节	82
4.13 客户端认证细节	84
4.14 临时 RSA 的细节	87
4.15 SGC 的细节	89
4.16 DH/DSS 的细节	97
4.17 FORTEZZA 的细节	99
4.18 错误警示 (Error Alert)	101
4.19 SSLv2 的向后兼容性	106
4.20 总结	108

第 5 章 SSL 的安全.....	109
5.1 介绍.....	109
5.2 SSL 都提供了什么	109
5.3 保护 master_secret.....	109
5.4 保护服务器的私用密钥	110
5.5 使用良好的随机性	110
5.6 检查证书链.....	111
5.7 算法的选择.....	111
5.8 小结.....	112
5.9 攻破 master_secret.....	112
5.10 在内存中保护秘密	114
5.11 保证服务器私用密钥的安全	115
5.12 随机数生成.....	121
5.13 证书链的验证.....	122
5.14 部分攻破.....	126
5.15 已知的攻击	130
5.16 计时密码分析	130
5.17 百万消息攻击	131
5.18 小-子组攻击 (Small-Subgroup Attack)	133
5.19 降级使用出口模式	134
5.20 总结	135
第 6 章 SSL 的性能.....	136
6.1 介绍.....	136
6.2 SSL 速度慢	136
6.3 性能法则.....	136
6.4 加密的开销昂贵	139
6.5 会话恢复.....	140
6.6 握手算法与密钥选择	141
6.7 批量数据传输	142
6.8 基本的 SSL 性能法则	142
6.9 小结	143
6.10 握手的时间分配	143
6.11 普通 RSA 模式	144
6.12 带有客户端认证的 RSA	146
6.13 临时 RSA	147
6.14 DSS/DHE	149

6.15 具有客户端认证的 DSS/DHE	151
6.16 DH 性能的改进	152
6.17 记录处理.....	154
6.18 Java.....	155
6.19 重负下的 SSL 服务器	157
6.20 硬件加速.....	159
6.21 串联硬件加速器.....	160
6.22 网络延迟.....	163
6.23 Nagle 算法.....	164
6.24 握手缓冲.....	166
6.25 高级 SSL 性能法则	168
6.26 总结	168
第 7 章 使用 SSL 进行设计	170
7.1 介绍	170
7.2 了解要保证什么的安全	170
7.3 客户端认证选项	171
7.4 引用完整性	172
7.5 不适合的任务	173
7.6 协议的选择	174
7.7 减少握手的开销	176
7.8 设计策略	176
7.9 小结	176
7.10 独立端口	177
7.11 磋商升级	178
7.12 降级攻击	178
7.13 引用完整性	180
7.14 用户名/口令认证	182
7.15 SSL 客户端认证	183
7.16 相互用户名/口令认证	184
7.17 再握手	187
7.18 二级通道	188
7.19 关闭	189
7.20 总结	191
第 8 章 SSL 编程	192
8.1 介绍	192

8.2	SSL 的实现	192
8.3	范例程序	192
8.4	上下文环境的初始化	194
8.5	客户端连接	199
8.6	服务器接受请求	204
8.7	简单的 I/O 处理	205
8.8	使用线程实现多路 I/O	208
8.9	使用 select() 实现多路 I/O	212
8.10	关闭	219
8.11	会话恢复	221
8.12	缺少什么？	223
8.13	总结	224
第 9 章 SSL 上的 HTTP		225
9.1	介绍	225
9.2	保护 Web 的安全	225
9.3	HTTP	227
9.4	HTML	228
9.5	URL	231
9.6	HTTP 的连接行为	232
9.7	代理	232
9.8	虚拟主机	233
9.9	协议选择	234
9.10	客户端认证	234
9.11	引用完整性	234
9.12	HTTPS	235
9.13	HTTPS 概述	235
9.14	URL 与引用完整性	238
9.15	连接关闭	243
9.16	代理	244
9.17	虚拟主机	247
9.18	客户端认证	249
9.19	Referrer	253
9.20	替换攻击	254
9.21	升级	254
9.22	编程问题	257
9.23	代理 CONNECT	257

9.24	处理多个客户端.....	261
9.25	总结.....	265
第 10 章	TLS 上的 SMTP.....	266
10.1	介绍.....	266
10.2	因特网邮件的安全.....	266
10.3	因特网消息传递概述.....	268
10.4	SMTP	268
10.5	RFC822 和 MIME.....	271
10.6	E-mail 地址	273
10.7	邮件中继.....	273
10.8	虚拟主机.....	276
10.9	MX 记录.....	276
10.10	客户端邮件存取.....	277
10.11	协议的选择.....	277
10.12	客户端认证.....	277
10.13	引用完整性.....	278
10.14	连接语义.....	278
10.15	STARTTLS	278
10.16	STARTTLS 概述	278
10.17	连接关闭.....	282
10.19	虚拟主机.....	283
10.20	安全指示器.....	284
10.21	经过认证的中继.....	285
10.22	源发者认证.....	285
10.23	引用完整性的细节	286
10.24	为什么不使用 CONNECT	288
10.25	STARTTLS 有什么好处	289
10.26	编程问题.....	290
10.27	实现 STARTTLS	290
10.28	服务器的启动.....	291
10.29	总结	292
第 11 章	各种方案的对比.....	293
11.1	介绍.....	293
11.2	端到端的论述.....	293
11.3	端到端的论述与 SMTP.....	294

11.4	其他协议.....	295
11.5	IPsec	295
11.6	安全关联.....	296
11.7	ISAKMP 和 IKE	296
11.8	AH 和 ESP	298
11.9	协同工作: IPsec	299
11.10	IPsec 与 SSL 的对比.....	300
11.11	安全 HTTP	302
11.12	CMS	303
11.13	消息格式.....	304
11.14	加密选项.....	304
11.15	协调工作: S-HTTP	306
11.16	S-HTTP 与 HTTPS 的对比.....	309
11.17	S/MIME.....	311
11.18	S/MIME 的基本格式.....	311
11.19	只进行签名	312
11.20	算法的选择	313
11.21	协调工作: S/MIME.....	314
11.22	实现障碍	316
11.23	S/MIME 与 SMTP/TLS 的对比.....	316
11.24	选择合适的解决方案	317
11.25	总结	318
附录 A 范例代码		320
A.1	第 8 章	320
A.2	第 9 章	337
附录 B SSLv2.....		357
B.1	介绍	357
B.2	SSLv2 概述.....	357
B.3	缺少的功能	359
B.4	安全问题	360
B.5	PCT	361
B.6	有关 SSLv1 的情况.....	363
参考文献		365

与安全有关的概念

1.1 介绍

本章的目的是提供有关通信安全和密码学的基本介绍。通信安全是一个复杂的话题，存在许多讨论此类问题的优秀书籍。这里，并不打算详细讨论这个话题，而是教给你足以理解本书的余下部分所用概念和术语的知识。那些已经对密码学和通信安全有所了解的读者可以自由选择是否完全跳过这一章。

开始先来讲解所关心的各种威胁，以及所能提供的各种各样的安全服务。接着再对加密算法以及如何组合使用它们来提供这些安全服务进行粗略的描述。最后，讨论各种算法的一些细节，这些细节与讨论它们在 SSL/TLS 中的应用时有关。

1.2 因特网威胁模型

第一件事就是定义威胁模型（thread model）。威胁模型描述了攻击者可望拥有的资源以及可望采用的攻击。几乎每一种安全系统都受制于某种威胁。为了说明这一点，设想将资料保管在一个根本无法打开的保险柜中。这样一切都很好，不会出什么问题。但是如果有人在你的办公室中安装了一台摄像机的话，他们就可以在你每次取出资料阅读的时候看到你的机密信息，所以说保险柜并没有带来多少安全。

因此，当在定义威胁模型的时候，关心的不止是定义所担心的各种攻击，还要定义我们不准备关心的攻击。若忽视了这个重要步骤，则常常会因为设计者为了试图分辨出每一种可能的威胁而使工作彻底陷入僵局。重要的是分清哪些是现实的威胁，哪些可以寄希望于现有的工具而加以阻止。

因特网安全协议的设计者们通常或多或少都要面临一些共同的威胁模型。首先假定协议所运行的实际端系统（end system）是安全的。在其中一个端系统处于攻击者的控制之下时，去防止各种攻击，即便可能也会是极其困难的。针对这种假定有两点需要澄清。第一，破坏任何单一的端系统不应当使所有人的安全遭到破坏。不应当存在一损俱损的情况（single point of failure）。举例来说，如果一个攻击者攻陷了系统 A，那么所有 B 与 A 之

间的通信都会遭到破坏，但是 B 与 C 之间的通信应当是安全的。如果我们必须要有一个一损俱损的节点的话，那么就必须有可能强化安全使其不受攻击的侵害。第二，攻击者有可能会控制系统，企图冒充合法的端系统。我们所假定的一切就是用户能够期望自己的机器没有遭到破坏。

此外，假定攻击者在某种程度上完全控制了任意两台机器之间通信通道。攻击者当然可以将具有任意地址信息（发送者或接受者）的数据包注入到网络中，而且可以读取网络上的任何数据包，以及选择删除任意的数据包。必须假定你所接收的任何数据包都存在潜在源于攻击者的可能，而任何你所发送的数据包都有可能在传输过程中遭到修改。依赖于向网络中写入数据包的攻击被称做主动攻击（active attack），而只涉及从网络中读取数据包的攻击被称之为被动攻击（passive attack）。

这种攻击者可以修改网络通信数据的假定存在一种显而易见的推论，就是攻击者可以通过去除所有相关的数据包来切断任两台机器之间的所有通信。这是一种拒绝服务攻击（denial-of-service）形式。另一种形式就是迫使你耗费大量的 CPU 资源来响应网络连接。协议设计者习惯上并不关心拒绝服务攻击，这可不是因为此类攻击并不重要，而是要想加以阻止简直是太难了。

威胁模型的一项最为重要的功能就是使得保障安全的代价切合实际，物有所值。采用的安全措施应当以实现它们的花费不超过预期的风险为准。无法做出正确的判断就会极易导致这样一种情形，判断不出可接受的风险，也就设计不出可以接受的系统。

风险计算的部分工作就是评估攻击者实施指定攻击所花费的努力，而每阻止一种攻击类型通常都会增加开销。没有任何一种安全系统可以阻挡任何攻击。安全模型的功能就是让设计者判断哪些攻击值得阻止。

然而，准确估计所必须的安全需要对攻击者能力的准确估算。如果一种原先以为不切实际的攻击现在被发现实施起来很简单，那么就会存在一个暴露安全缺陷的窗口，而人们就此调整自身的安全模型和实现以弥补相应的缺陷。

1.3 角 色

为了更容易理解本章讨论的各种例子，我们将重复使用相同的名字来代表各种角色。按照惯例，将通信的双方分别称做 Alice 和 Bob，这里沿用原先 RSA 论文[Rivest1979]中的名字。攻击者还是被称做“攻击者（the attacker）”。

1.4 安 全 目 标

大多数人在谈及安全时，就好像安全是协议的一项孤立的整体特性，但正如前面所讨论的那样，事实显然并非如此。根据攻击者所具备的具体技能，他会对我们的数据安全构成各种不同的危害。通信安全由许多不同的但却相互关联的特性构成。具体对其怎样进行划分则有赖于所谈论的对象。发现最为有用的划分就是将其分成以下三个主要类别：保密性（confidentiality）、消息完整性（message integrity）和端点认证（endpoint authentication）。

保密性

大多数人思考安全的时候，都是在考虑保密性。保密性的意思就是对无关的听众保密，通常这些听众为窃听者。当政府监听你的电话的时候，就会对你的保密性构成威胁。（同时，这也是一种被动攻击，除非联邦调查人员开始试图在线路上模仿你的声音。）

显然，如果你有秘密的话，就会希望没有别人知道这些秘密，所以你最少也需要保密性。当在电影中看到间谍们走进洗手间放开所有的水龙头来扰乱窃听者的时候，他们所寻求的特性就是保密性。

消息完整性

第二个首要目标就是消息完整性。这里的基本思想就是，我们想要确信自己所收到的消息就是发送者发送的消息。在基于纸质的系统中会自动带有一定的消息完整性。当你收到一封用钢笔写的信时，由于钢笔印记很难从纸张中去除，所以你完全可以肯定没有抹掉的词语。然而，攻击者可以很容易地在信中增加一些笔记来彻底改变消息的原意。

另一方面，在电子世界中，由于所有的位看上去都是相似的，所以在传输过程中摆弄起消息来简直就是小菜一碟。你只需将消息从线路中去除，拷贝你想要的部分，随意增加什么数据进去，然后就可以产生一条由你挑选的新消息，而接收者却一无所知。这与攻击者拿到你写的信，买一些新的信纸，然后把消息修改后重新拷贝纸上是一样的。只是采用电子方式则要容易得多，因为所有位都是相似的。

端点认证

所关心的第三个特性就是端点认证。通过它所要达到的意图就是要知道通信中的某个端点（通常为发送者）就是我们所指的那个端点。没有端点认证，要提供保密性和消息完整性就非常困难。例如，如果我们收到了一份来自 Alice 的消息，但无法确定该消息是由 Alice 而不是攻击者发送的话，那么消息完整性特性对我们来说就不会有任何意义。与之类似，当我们向 Bob 发送一份机密消息，假如我们实际上是将机密消息发送给了攻击者的话，对我们来说也就没有什么意义可言了。

注意，端点认证可以以非对称的方式提供。当你给某人打电话时，你可以确信接电话的人是谁——或者实际上至少也是处在你所拨打的电话号码位置的人。另一方面，如果接电话者没有主叫识别，那么他们也不会知道是谁在给自己打电话。给某人打电话就是一个接收方认证的例子，这里你知道接电话的人是谁（要想突破电话网的安全是困难的，但也不是不可能的），但是对方却不知道发送方是谁。

从另一个角度来讲，现金就是一个发送端认证的例子。一张美圆现钞就像是政府签名的消息。政府并不知道是谁拿到了给定的钞票，但是你却可以深信钞票实际上是由 US Mint 印刷的，原因就是货币很难伪造。

一个实际的例子

为了使所有这些内容再清楚一点，让我们来看一个提供所有这三种特性的系统的真实例子，看看现实中的各种安全特性是如何相互作用来提供每种特性的。从一个没有任何安全特性的系统开始，然后对其进行修改，直到获得所有想要的安全特性为止。

你在邮件中收到了一张寄给你的明信片，表面上看来它是 Alice 寄的。你对这张明信片及其内容能了解多少呢？不多。随便一个窃听者（eavesdropper）都可以阅读这张明信片，

而且还能修改它。

一种潜在的改善措施就是让 Alice 写一封信来代替明信片，并把这封信放在信封里。这样一来这种状况会有所改进吗？实际上作用不大。通过检查信封，你可以确信没有人摆弄过它（信可能由熏蒸法开启，但是你可以使用不受此类方法影响的信封），所以你或许会认为自己实现了保密性与信件内容的消息完整性。但事实并非必定如此。攻击者可以打开原来的信封，阅读信件的内容或是对其进行修改，然后再重新在一个新的信封中封好。所以实际上你的消息没有任何保密性可言。由于没有发送者认证，消息完整性和保密性都毁于一旦。

从 Alice 的角度来说，情况甚至更糟。就她所知，一旦邮递员从她的门口将信收走，消息就被打开了。我们稍后再来说明如何解决这种问题。

传统的确定信件在传递过程中没有被打开的方法就是在信封的口盖处加盖蜡印。虽然打开信封并重新加盖这些蜡印是可能的，但是我们此刻假定不可能做到。另一种更为现代的做法就是让发送者在口盖处签名。假定你认识印章或者发送者的签名，这样大体就可以确信信封在到达时未经开启，因为签名很难伪造。

然而，只有在我们有办法知道谁会发给我们这份消息的时候才能这么确信。要是攻击者打开信封，然后重新用他自己的信封封好，并在口盖处签上自己的名字，会怎样呢？如果我们以前并不知道这是 Alice 的真实签名的话，就不能检测出这种攻击。

为了明白这种攻击的用途所在，设想信封里包含了一张购买某种物品的定单，其中填有 Alice 的信用卡号码。攻击者将消息中的发货地址改成自己的地址，但却保留 Alice 的信用卡号，并重新将信封封好。这样货物就会投递给攻击者而账却记在 Alice 头上。使用到目前为止所讨论的这些技术，要想阻止此类攻击是非常困难的。

到目前为止所谈论的各种机制——信封、印章等等都是预防篡改的 (tamper evident)，接收方可以断定内容被篡改但却无法阻止它的发生。阻止此类攻击最简便的方法就是使用抗篡改的 (tamper resistant) 包裹（我们经常使用 tamper-proof（防篡改）这个字眼，但那样又似乎过于乐观），设想 Alice 不是发送一封信，而是寄给我们一个包含那封信的保险箱。保险箱惟一的两把钥匙位于 Alice 和我们的手中。

这差不多使整个问题得到了解决。因为你有惟一的另一把钥匙，而你知道信不是自己发的，那么就可以充分相信是 Alice 发的，这样就实现了发送者认证。因为除了你和 Alice 之外再没有人能打开这个保险箱，于是就可以知道在传递过程中信没有被打开、阅读或修改，因此也实现了保密性和消息完整性。类似的，Alice 知道除你之外没有人能够打开保险箱，所以她就可以确信自己实现了接收者认证、保密性和消息完整性。

这种方案有（至少）一个严重的问题。你不得不与你想要与之通信的每一方交换一对钥匙中的一把。如果你想一想典型的企业每天要发送多少消息的话，就会发现这种方法非常麻烦。

这个例子对于两种目的来说非常有用。首先，它演示了我们有望提供的各种安全服务。同样重要的是它演示了威胁模型的概念。因为四处邮寄保险箱相当昂贵，所以这并不是一种非常切实可行的方案。而人们通常会采用较低的安全级别，只管假定邮政服务是可以信赖的。对于需要特别保密的信息来说，他们或许会交由专人办理，但那是他们通常关心事了。能够打开邮件并任意篡改内容的攻击者恰恰并非标准商业威胁模型的一部分。为每封邮寄的信函提供高安全的代价简直太高了，以至于没有公司能够负担的起。

1.5 必要的装备

注意到此为止我们甚至还没有提及密码学，现在就要讲到它。密码学（Cryptology）是一种设计各种用以提供安全的算法的理论。密码术（Cryptography）研究使用这些算法来保证系统和协议的安全。本节将概括描述各种可用的加密算法类型，下一节将讨论如何使用它们来提供安全服务。

● 加密

从概念上来讲，最易于理解的算法就是加密（Encryption）算法。其思想是简单的：加密算法接收一些数据（称做明文）并在密钥（key）的控制下将其转换为密文（ciphertext）。密文看上去就像随机数据，不知道密钥就无法收集与明文有关的各种有用信息（长度信息可能除外）。密钥通常只是一个简短的随机字符串，通常约有 8~24 个字节。图 1.1 中描述了这些元素之间的相互关系。



图 1.1 加密与解密

加密算法应被视为纯粹是提供保密性。根据所使用的具体加密算法的不同，篡改密文对明文的确切影响也有所不同。但是这种篡改通常很难被发现。因此，当我们接收到一条只经加密的消息时，我们无法知道它是否被篡改过。

好的加密算法完全应当由可能的密钥数量决定其安全与否。最快的攻击应该是穷举搜索（exhaustive search）：在给定密文的情况下，每次尝试一个密钥直到产生大致正确的解密输出为止。算法的安全应当只依赖于密钥的保密，算法的保密不应当是安全性的必要条件。

攻击者在知道密文所对应明文情况下的攻击称做已知明文（known plaintext）攻击。攻击者在不知道明文情况下的攻击被称做只有密文（ciphertext only）攻击。即便是攻击者不知道明文的具体内容，他也有可能了解有关它的一些情况，这可以让他实施一种只有密文攻击。举例来说，攻击者或许知道明文为 ASCII 码，在这种情况下，任何包含非 ASCII 的解密输出都必定使用了错误的密钥。

由于发送者与接收者使用同一个密钥（这个密钥必须保密），我们刚才所描述的有时被称之为秘密密钥加密（secret key cryptography），以与公用密钥加密（public key cryptography）形成对照，我们稍后再来讲述有关后者的知识。

加密算法设计是一个非常活跃的领域，没有成百上千也有几十种可用的算法。最为流行的算法包括数据加密标准（DES）[NIST1993a]，Triple-DES（三重 DES）[ANSI1985]，RC2[Rivest1998]和 RC4（还没有出版任何正式的 RC4 规范，请参阅[Schneier1996a]来了解有关的信息）。

● 消息摘要

消息摘要（Message Digest）是一种函数，它接收一个任意长度消息为输入，并产生一个表示消息特征的定长字符串。消息摘要最重要的属性就是不可逆性（irreversibility）。给定

摘要值，要想计算出它所对应的消息应当是极其困难的。通过一个记数变元就可以很容易的表明摘要值，并没有提供足够产生原始消息的数据。可能的（任意长度）消息远比定长特征值长，从而使得几乎不可能存在函数的逆。然而，要想让一种摘要是安全的，就必须难以生成摘要值为该值的任何消息。我们所讲的困难就是指为了找到匹配的消息文本，你需要搜索与摘要值尺寸成比例的消息空间。

消息摘要值的第二个重要属性就是要想产生具有相同摘要值的两条消息 M 和 M' 应当是困难的。该属性被称做抗冲突性（collision resistance）。实际上，任何抵御冲突发生的消息摘要的强度只有摘要值的一半，因此一个 128 位的摘要值避免发生冲突的强度只有 64 位，也就是说需要大约 2^{64} 次操作才会产生一次冲突。因此，在选择摘要值长度时的限定性因素通常是抵御冲突的强度，而不是抵御可逆性的强度。

消息摘要的首要用途就是用于计算数字签名（digital signature）和信息验证码（MAC），我们将在本章的后面讲到这两项功能。不过，现在至少有一种显而易见的利用简单摘要值的用途：使用它，无需提供保密信息就能证明你拥有相应的保密信息。假设你发明了一种新东西，想在不告诉人们这是什么东西的条件下表明你是第一个发明它的人。你就可以记下保密信息，计算出它的摘要值，然后再将摘要值刊登在报纸的分类广告上。于是，如果有人对你作为第一发明人表示怀疑的话，你就可以将原始文本拿给他们看，而他们就可以独立的对摘要值进行验证。

使用最广泛的消息摘要算法为消息摘要 5 (MD5) [Rivest1992] 和安全散列算法 1 (SHA-1) [NIST1994a]。散列算法这个字眼经常用做消息摘要的同义词，因为摘要算法表面上与普通的散列算法非常相似。我将在本书中互换使用散列与摘要这两个字眼。

MAC (信息验证码)

假设 Alice 与 Bob 共享一个密钥，Alice 想给 Bob 发送一条消息，而 Bob 将会知道它是 Alice 发送的。如果她加密的话，那就非常简单，只需将他们共享的密钥用做加密密钥即可。但是如我们所讲的，这种方法并不能提供任何消息未被篡改的真正保证，只能保证消息来自于 Alice。我们需要一种新的工具，即信息验证码（message authentication codes, MAC）。MAC 类似于摘要算法，但是它在计算的时候还要采用一个密钥，因此 MAC 同时依赖于所使用的密钥以及要计算其 MAC 的信息。实际上 MAC 通常是根据摘要算法构造得出的。

尽管存在许多基于各种摘要算法来构造 MAC 的尝试，但是因特网安全团体似乎就一种称做 HMAC[Krawczyk1997] 的构造方法达成了一致，这种方法描述了如何基于满足某种合理假定摘要来创建具有可证明的安全特性的 MAC。SSLv3 中使用的是一种 HMAC 的变种，而真正的 HMAC 在 TLS 中使用。

密钥管理的问题

现在有了 MAC 这项新增的工具箱，就有了与本章开头所讨论的邮寄保险柜等价的电子系统。Alice 拿到我们的消息，使用共享密钥对信息进行加密，添加一个也是基于该密钥构造的 MAC 并将其发送给 Bob。她知道只有 Bob 能够阅读这条信息，因为 Bob 与其共享解密时所需要的密钥。类似的，Bob 知道发送这条消息的只有 Alice，因为只有 Alice 才具有创建消息上的 MAC 时所需要的密钥。这样，Bob 就可以知道是 Alice 发送的信息，而且还未被篡改。

那么，我们就有了所需要的一切，是吗？不。尽管这些消息比保险柜轻，邮寄也更为便宜，但是与每个人进行通信，仍然存在与其共享密钥的问题。周围有这么多密钥需要处理非常不方便。但更重要的是，这意味着为了进行密钥交换，你实际上必须要与每一个与之通信的人会面。这为通过因特网购买商品设置了障碍，除非你个人已经与供应商碰过面。这里面不便之处就是密钥管理的问题。

KDC（密钥分发中心）

针对密钥管理问题最流行的解决方案就是公用密钥加密（public key cryptography, PKC），将在下一节讲述相关的内容。不过也存在一种只使用到目前为止所讨论的工具来解决密钥管理问题的措施。基本思想就是利用受信任的第三方，我们委托它对与我们通信的各方进行认证。这种第三方通常是由网络上某处一台安全的机器来实现的。这台机器被称做密钥分发中心（key distribution center, KDC）。每个需要保密通信安全的个人都与 KDC 共享一个密钥。当 Alice 想要与 Bob 进行通信时，她就给 KDC 发送消息，该消息以其与 KDC 共享的密钥加以保护，请求与 Bob 进行通信。KDC 产生了一个新的用于 Alice 与 Bob 之间进行通信的加密密钥，再将其放在一条称做许可证（ticket）的消息中返回。

一条许可证消息由两条消息组成。第一条消息是给 Alice 的，其中带有新密钥。第二条消息是给 Bob 的，用 Bob 的密钥进行加密，其中也包含新密钥。Alice 将许可证中 Bob 的那一部分转交给 Bob，现在 Alice 与 Bob 共享密钥。这种协议的基础版本是由 Needham 和 Schroeder[Needham 1978]发明的，但是部署最为广泛的变种为 Kerberos，在 MIT 及其他地方大量应用于认证与加密（请参见[Miller1987]）。

这种方案有两种主要缺点。首先，KDC 必须总是处于联机状态，因为如果它下线的话，就无法完成通信初始化。其次 KDC 能够读取任何两方之间传递的数据。它还能够伪造两方之间的通信。更糟的是，如果 KDC 被攻克的话，任何两个 KDC 用户之间的通信均将遭难。这可不行。尽管如此，对于封闭系统来说，人们对这种类型的协议还颇有兴趣。

公用密钥加密

1976 年，Stanford 有几个非常聪明的人，想出了一种更好的解决密钥管理问题的方法。在一篇名为“密码术新动向”[Diffie1976]的论文中，Whitfield Diffie 和 Martin Hellman 提出了现在称之为公用密钥加密的方案。基本思想就是设计一种在加密和解密时使用不同密钥的函数。你公开自己的加密密钥（公用密钥），但解密密钥（私用密钥）要保密。（由于公用与私用密钥的不同，公用密钥加密有时被称做非对称加密，而共享密钥加密有时被称做对称加密）。这意味着无需碰面，任何人都能够给你发送保密信息。这在消除需要预先共享密钥的不便之处的同时也解决了其中的保密性的问题。

事实上，公用密钥加密在 1970~1974 年间还由一家英国情报机构，Communications-Electronics Security Group (CESG) 发明，他们将其称之为无密码加密（non-secret encryption, NSE）。有趣的是，所发明的技术还包括 Diffie-Hellman 和一种 RSA 的变种。不管怎样，它们暗示了这些方法的基础作用。不过，成果被封存，而学术团体各自独立发明了这两种技术。在情报团体中早先存在这样的技术的情况在 1998 年才揭示的。[Ellis1987]详细描述了 NSE 的历史。[Ellis1970]中演示了 NSE 存在的可能性。[Cocks1973,Williamson1974,Williamson1976]描述了特定的技术。

事实表明 PKC 针对问题中的认证部分也有一套解决方案。你的私用密钥可以用来创建某种称作数字签名的东西，它与 MAC 之间的关系如同公用密钥加密与秘密密钥加密之间的关系一样。你使用私用密钥对消息进行签名，而接收方使用你的公用密钥来验证你的签名。注意，数字签名具有一项 MAC 所不具备的重要属性：不可抵赖性（nonrepudiation）。发送方和接收方都可以产生 MAC，但是只有签名者才能够产生签名。这样，接收者就可以证明发送方对消息进行了签名而发送方无法抵赖。

证书

尽管提供了我们解决问题所需要的工具，不幸的是，到目前为止我们所拥有的工具并不能完全解决密钥管理问题。要明白哪里存在问题，最简单的方法就是请问各方是如何得到彼此的公用密钥的。如果这些密钥是以电子形式发表的，或是通信各方通过交换得到的话，那么攻击者就能在这些密钥传递给接收者的过程中进行篡改。当两方打算进行通信时，攻击者截获他们的密钥，并代之再将自己的密钥发送给每一方。这样每一方都会按照攻击者的要求来进行加密，而攻击者根据真正的接收者重新进行加密，如图 1.2 所示，这被称作中间人攻击（man-in-the-middle attack）。然而，如果将密钥以物理方式印刷出来，则很不方便。解决方案（还是）就是通过称之为证书授予权（certificate authority, CA）的第三方。CA 发布以其私用密钥签名的目录。在实际应用中，CA 不是对目录进行签名，而是对包含密钥属主及其公用密钥的单一信息进行签名。这些信息一般被称作证书（Certificate），证书授予权因而得名。证书的主要标准为 X.509[ITU1988a]，它是在 RFC2459[Housley1999a] 中为因特网编写的。

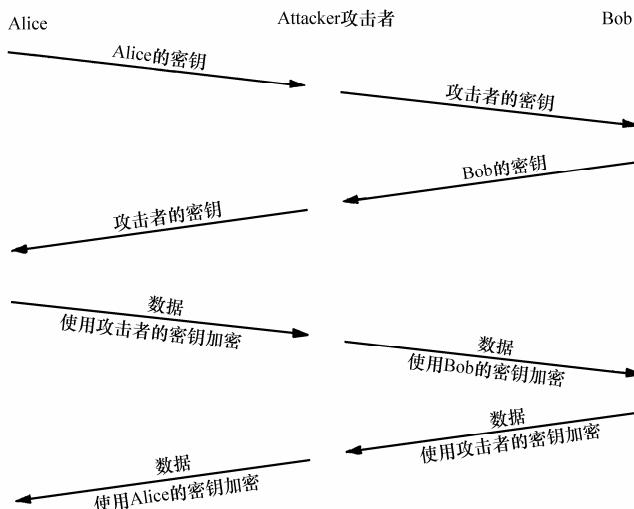


图 1.2 中间人攻击

CA 的公用密钥以某种物理形式发布，不过 CA 并不多而且不经常变换密钥，因此这在实际应用中不成问题。通常将 CA 密钥编译到需要使用它们的软件中，这样就能与软件一起发行。当软件以 CD-ROM 或软盘方式发布时就非常有意义。但有时软件只是下载得到的，在这种情况下就又回到了刚开始时遇到的问题——即人们做事常常并不理智。

图 1.3 描述了一份公用密钥证书展开后的视图。这里的细节并不重要，但请注意其中的



基本结构：证书包含颁发者名称（issuer name）（证书签名者的名字，这里就是“Secure Server...”），主体名称（subject name）（证书所担保的密钥的持有者，在这里就是“www.amazon.com...”），主体公用密钥（subject public key）（即密钥本身），一组控制信息，诸如有效期限、序列号以及对整个数据对象的签名。涉及证书的公用密钥解决方案仍然要包含受信的第三方（即 CA），但是它们的确修正了我们前面所描述的基于 KDC 系统的主要问题。由于同一个证书可以用来向任何人证明其公用密钥，所以 CA 没有必要为了让 Alice 与 Bob 进行通信，而始终处于联机状态。同时因为 CA 无法存取任何人的私用密钥，所以它也不能读取任何信息。

```
version: v1
serial number: 2A 17 EF 73 97 07 74 7B E2 4B FB
signature
    algorithm: md5WithRSAEncryption
issuer:
    C=US
    O=RSA Data Security, Inc.
    OU=Secure Server Certification Authority
validity
    not before: Sat Jan 28 02:21:56 1995
    not after: Thu Feb 15 02:21:55 1996
subject:
    C=US
    ST=Washington
    L=Seattle
    O=Amazon.com, Inc.
    OU=Software
    CN=www.amazon.com
subject public key
    algorithm: rsaEncryption
    modulus
        bit length: 1024
        value:
            00 C8 1B 8B FA 40 C3 5B E3 46 3F 17 10 56 19 64 C4 F4 F9
            CC AE CA F7 0B 02 1C C3 2D 27 60 91 16 CO A1 23 8B CA 90
            77 31 25 CA D9 DE BO 87 F5 25 C9 12 7A 95 DF DC 6C E4 1C
            C3 31 9F 77 BE 69 3E 9F BB 35 BF F3 3D BA 7A 72 DA 5D 0C
            60 91 29 F8 89 67 50 5C 32 46 63 F2 FF 42 9D 24 F2 DC 6F
            E5 CA D3 CD 3A AB 9D 5F A9 4D BO 82 91 E3 D3 EA AA EF 78
            8A C1 06 B6 6D EA 56 B8 7E 68 5D AF 4D 85 AF
    public exponent:
        bit length: 2
        value: 03
signature
    value:
        03 43 60 4B 5B 4B F1 78 56 BF B4 9B 81 E6 EE 0D 19 1B 4E 43 BD
        D9 C7 62 62 55 32 C7 15 A4 33 3A CA 0E 60 E5 FE D7 53 94 C6 AC
        17 D0 CE 7B 11 27 0C 3B 26 19 6D 35 55 4C D8 26 F4 5F FO 90 0D
```

```
90 7F FC 39 47 FE EE B4 72 92 93 BF 93 7F 5C 56 38 10 F5 E5 58
B5 6C 3E E0 B4 55 8D 74 BE 84 F1 53 67 49 5B 14 12 E6 A7 59 A9
97 9E 6C E4 59 A6 8F 4E 7E B5 D9 2D 80 3F 38 3C 4C 11 A7 37
```

图 1.3 公用密钥证书

DN (标识名)

证书中的主体名称及颁发者名称为 X.500 标识名(Distinguished Names, DN)[ITU1988b]。标识名的目的就是为每个网络实体提供一个惟一的名字。为了达到这一目的, DN 具有一种分层的结构。一个 DN 由一系列 RDN (relative distinguished name, 相对标识名) 构成。RDN 就像 DNS 的组成部分那样, 每个 RDN 在其他一些实体的名字空间中指定实体的名字。因此, 最上层的 RDN 必须是全局范围内惟一的, 而其下的每个 RDN 只需在上一个 RDN 作用域中保持惟一即可。RDN 也有结构, 每个 RDN 由一系列属性-值断言(attribute-value assertion, AVA) 构成。基本上说, AVA 就是键/值对。图 1.4 描述了一个 DN 范例。

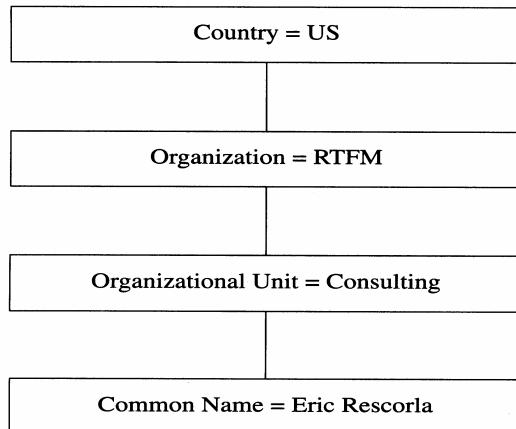


图 1.4 DN 范例

图 1.4 中的每个方框代表一个 RDN。于是顶层 RDN 包含一个 AVA, 其中的属性为 Country, 值为 US, 以此类推。注意, 尽管从理论上来讲, 每个 RDN 可以有多个 AVA, 但在实际应用中, 几乎总是只有一个。与之类似, 尽管从理论上讲, 任何 AVA 都可以在任何层次上出现, 但在实际应用中, 它们却总是从最不具体(如国家)到最具体的(普通名称——一般对应你的名字或 email 地址)。标识名时常以文本形式来表述, 其中将属性名书写为缩写形式。图 1.4 中的名字会被表示为:

C=US, O=RTFM, OU=Consulting, CN=Eric Rescorla

扩展

证书包含一组标准值, 但常常需要在其中安放其他一些信息。X.509 版本 3 提供了一系列任意的扩展(Extension)。扩展就是类型-值对。尽管 X.509 允许私有扩展, 但还是提供了一些标准扩展。三种对我们来说是重要的扩展为:

subjectAltName 其中包含了这个用户其他的名称。这些可能是其他的 DN, 但也可能是

其他的名字形式，其中包括 dNSName (DNS 主机名) 和 emailAddress (email 地址)。

keyUsage 中包含了该密钥可接受用途的屏蔽位，这些用途包括签名、加密等等。

extendedKeyUsage 允许包含一组任意对象标识 (OID) 的列表 (请参见 ASN.1, BER 和 DER 一节)，这个列表具体描述该密钥的用途。

证书的撤消

考虑这样一种情况，用户的密钥由于丢失或爆光而被泄露。我们想要以某种方式告诉其他用户这个公用密钥以及验证它的证书已经不再受信。处理这种问题最常见的方法就是使用某种称作证书撤消列表 (Certificate Revocation List, CRL，发音为 “krill”) 的东西。

CRL 是一个经过签名的、标有日期的、包含所有已撤消证书的列表，撤消意味着不再被视为是有效的。假设某个 CA 发布 CRL，对证书进行验证则要求获得适当的 (最新的) CRL，并检查该证书未在列表中出现。

不幸的是，由于两种原因，CRL 工作的并不是很好。第一，需要定期的发布 CRL，这意味着在发布 CRL 时与下一次发布 CRL 之间存在安全隐患空挡。密钥或许已经被攻破，但是用户却无法得知，因为包含它的 CRL 还没有发布。第二，CRL 的分发存在问题。CRL 可能变的越来越大，而任何一种协议都要捎带上它，或是从 CA 获取，而这就又要求 CA 是联机的。

如果 CA 是联机的话，它还可以提供联机证书状况 (online certificate status) 报告。这样完全可以提供最新的撤消状况信息。用户可以向 CA 请求得到给定证书的状况，而 CA 则提供经过签名的应答。我们可以使用联机证书状况协议 (Online Certificate Status Protocol, OCSP) [Myers1999] 来完成这项工作。然而，对应答进行签名的要求会给服务器 CPU 带来很高的负载。显然，这并不是我们想要的。存在一些减轻此类负载的优化措施 [Kocher1996a]，但是并没有被广泛部署。

ASN.1、BER 和 DER

ASN.1 属于那种让人 “感到很不舒服” 而有时又必须了解的东西。有许多基本的安全工具，特别是 X.509 证书，最终都是以 ASN.1 来定义的。ASN.1 是作为 OSI (Open Standards Interconnect, 国际电信联盟开放标准互联) 成果的一部分而定义的，它用作一种 OSI 协议的描述语言。基本思想就是为描述数据结构而创建一种允许通过机器来产生数据编码解码器 (一般称之为 codec) 的系统。

这在当时似乎是一种不错的想法：你有了一种可以用来将数据格式描述为结构化类型的语言，这非常类似于 C 中的 struct 以及 Java 中的类。这种语言被称作抽象语法记法 1 (ASN.1)。你还描述如何将以这种语言书写的结构映射为线路上的数据编码。它使你能编写一种完成两项工作的编译器：

- (1) 产生从 ASN.1 结构到任何一种你所使用语言的映射；
- (2) 自动为结构生成译码器。

不幸的是，ASN.1 复杂而且违背正常的思维，从而使得不仅开发编译器相当困难，就连编写 ASN.1 本身也都很困难。更糟的是，尽管按理说，多就意味着好，但设计 ASN.1 的人为在线路上的数据传输格式定义了不是一种，而是多达四组不同的编码规则。且每种编码只是为

了一组稍有不同的目标提供服务，结果是一团糟。

将会与我们有关的两种 ASN.1 编码规则为基本编码规则（Basic Encoding Rules, BER）和辨识编码规则（Distinguished Encoding Rules, DER）。鉴于 BER 对任意给定数据段可以有多种编码方式，DER 是 BER 的子集，它选取并坚持使用其中的一种方式，这样你就可以确信任何 codec 对给定的任何结构都以同样的方式进行编码。BER 的优势就在于，以 BER 编码的数据通常要比以 DER 编码的效率要高。然而，如果你对数据进行数字签名，想让语义上等价的信息总以相同的方式进行编码，那么就需要使用 DER。

我们甚至不打算费事地去具体讲解如何读懂 ASN.1。为了掌握要领，实际上你所需要知道的就是它与 C 语言中的结构类似，只不过是将类型定义的位置颠倒一下：名字在前，数据类型在后。图 1.5 描述了一个 ASN.1 结构的范例。它所描述的意思就是 Foo 是一个包含两个元素，bar 和 mumble 的序列。bar 的类型为 INTEGER，而 mumble 的类型为 BIT STRING。

```
Foo ::= SEQUENCE{
    bar INTEGER,
    mumble BIT STRING
}
```

图 1.5 ASN.1 结构范例

这个例子提供的信息足以让你对 ASN.1 结构表达的含义有整体上的把握。如果你对细节感兴趣的话，最好是参考一下[RSA1993a]。实际标准位于[ITU1988c]和[ITU1988d]中。

就本书目的而言，你实际只需再了解一件东西：OID (object identifiers, 对象标识符)。OID 为分配给任何种类对象的全局惟一的字节字符串 (byte string)，如算法、密钥用途等等。OID 空间是邦联性 (federated) 的——ISO 将 OID 空间的各区段分配给各种不同的实体，这些实体还可以进一步针对其他实体对空间进行划分。这样就存在大量的组织可以分配 OID，你如果想分配私有且惟一的 OID，还可以获得自己的 OID 区段 (称做 arc)。

1.6 组合起来使用

现在已经有了足够多的工具来构建一些简单的安全系统。我们将要讲述的系统听起来很复杂，但是需要明白的重要一点是，所要讨论的几乎每一种通信安全技术都是简单地根据四种技术：加密、摘要计算、公用密钥加密和数字签名中的一种来构造的。这些技术常被称做安全原语 (security primitive)，组合使用这些原语可以构建出更为复杂的结构。

举例来说，公用密钥方式要比共享密钥方式慢得多，因此我们可以方便的组合使用这两种技术，使用公用密钥来完成共享密钥的交换。为了加密一条消息，Alice 产生一个随机的共享密钥（该密钥的不同称呼有会话密钥、信息加密密钥、内容加密密钥、数据加密密钥）并用 Bob 的公用密钥对其进行加密，这个公用密钥是从 Bob 的证书中获得的。然后她使用这个会话密钥通过对称加密算法对消息进行加密。这种组合使用公用密钥加密和对称加密的方法提供了对消息的快速加密，同时又具有基于证书的密钥管理的好处。

与之类似，数字签名算法非常慢，且只适用于短小的信息。但是与消息摘要组合使用，

就可以高效的对大块信息进行签名。为了对消息进行签名，Alice 计算出消息的摘要值并使用其私用密钥对摘要值进行签名。组合使用信息摘要和数字签名提供了消息完整性和发送方认证而不必共享密钥。

1.7 简单的安全消息系统

先从设计一种简单的、适合发送保密 E-mail 信息的安全消息系统开始。大量此类协议已经被设计出来，其中包括 S/MIME[Dusse1998]，PGP[Atkins1996] 和 PEM[Linn1993,Kent1993,Balenson1993,Kaliski1993]。但所有这些协议都遵循相同的基本模型，下面就来讲讲这种模型。

图 1.6 中所描述的发送过程与前一节所讲述的内容非常接近。

- (1) Alice 计算出消息摘要。
- (2) 她对消息摘要进行签名并将产生的数字签名与其证书一起附在消息上。
- (3) 她产生一个随机会话密钥，使用它来加密经过签名的消息，即证书和签名。
- (4) 最后，她用 Bob 的公用密钥对会话密钥进行加密并将处理过的会话密钥附到消息上。

现在，她就可以把这条消息发送给 Bob 了。

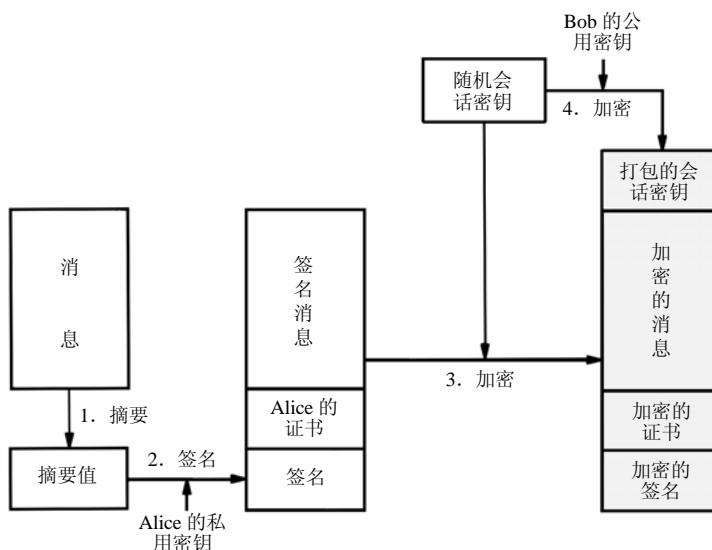


图 1.6 消息发送过程

图 1.7 描述了接收过程，使用与发送过程相反的步骤。

- (1) Bob 使用其私用密钥对会话密钥解密。
- (2) 他使用会话密钥对消息，即证书和数字签名进行解密。
- (3) 他自己计算出消息的摘要值。
- (4) 他对 Alice 的证书进行验证并取出 Alice 的公用密钥。
- (5) 他使用 Alice 的公用密钥来验证 Alice 的数字签名。

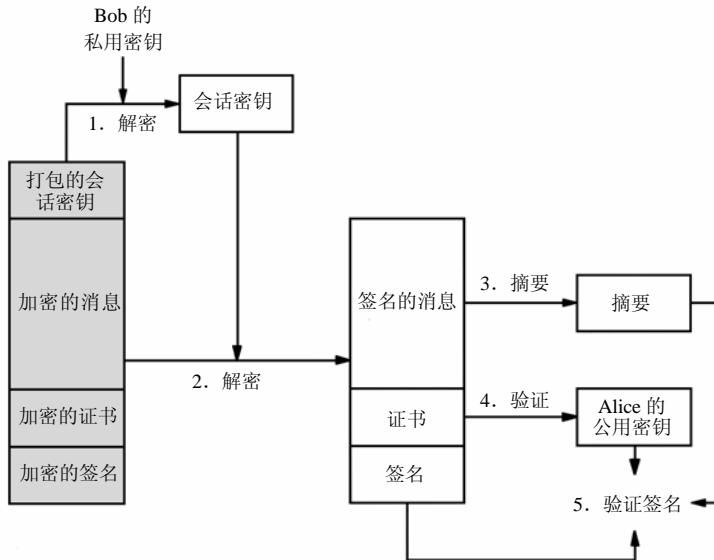


图 1.7 消息接收过程

1.8 简单的安全通道

我们刚才所讲的系统在发送单条消息的情况下工作得很好，如 E-mail，但是如果我们需要的是一条可以在其上传输任意信息的通信通道的话，这种系统的不足之处立刻就显现出来了。我们希望能建立一组可以在整个连接中使用的密钥。这样就可以不必为每个数据包动用开销昂贵的公用密钥操作，这对于交互式应用来说犹其重要，这种应用中的每次击键都有可能产生一个数据包。

我们想向交互式协议中增加的第二项改进就是实现证书发现 (certificate discovery) 的能力。在前一节我们所讲述的消息传递协议中，Alice 需要能够在给 Bob 发送一条消息之前获得 Bob 的证书。这种操作要么通过查阅目录，要么让 Bob 发送一条未加密的包含其证书的消息来实现。在交互式协议中，Bob 可以把给 Alice 发送证书作为他第一件要做的事情。

这种交互式系统的典型设计具有握手阶段，Alice 和 Bob 在这个环节中对对方进行认证并建立一组密钥。然后就可以进入数据传输阶段，在此阶段他们使用那些密钥来完成对其感兴趣的数据的实际传输。本节余下的篇幅将构建一种完成此类工作的简单协议。我们称之为玩具安全协议 (Toy Security Protocol, TSP)。

我们协议中的基本步骤为：

握手 (Handshake)。Alice 和 Bob 使用他们的证书和私用密钥来对对方进行认证并交换共享密钥。

导出密钥 (Key derivation)。Alice 和 Bob 使用达成一致的共享密钥导出一组加密密钥，以用于对传输进行保护。

数据传输 (Data transfer)。将要传输的数据分割成一系列的记录 (record)，并对每条记

录单独加以保护。这样使得数据一准备好就可以进行传输，一旦接收就可以进行处理。

关闭连接（Connection closure）。使用特殊的、经过保护的关闭消息，来安全地关闭连接。这样可以阻止攻击者伪造关闭操作而截断正在传输的数据。

简单的握手

Alice 所要做的第一件事情就是给 Bob 发送一条消息，告诉她已经作好了通信准备。消息中的内容没有加密，就是一句简单的“Hello”。而 Bob 则以他的证书作为应答。至此，我们已经实现了证书发现，于是就可以像先前那样开始发送独立的信息了。但是还有一种更好的办法，我们实际上想要做的就是安排 Alice 和 Bob 共享一个单一的密钥，我们称之为密钥（master secret, MS）。此后，Alice 知道只有 Bob 了解 MS，而对 Bob 来说也是如此。然后我们使用 MS 创建一组用于加密数据的密钥。

第一步是让 Alice 产生一个随机数作为 MS。完成这项工作实际上需要使用一种密码术安全的随机数生成器，这东西要比听起来麻烦得多。目前虽然存在许多用于此目的技术，但是几近艰涩，我们在这里不予讨论。一旦 Alice 拥有了主密钥，她只需使用 Bob 公用密钥对其加密来产生 EMS（encrypted master secret，加密的主密钥）。现在，我们已经完成了工作的头半部分。Alice 知道只有 Bob 和她拥有 MS。如果 Bob 并不在意她的身份的话，她尽可将 EMS 发送给 Bob，继而就可以开始进行通信了。这种方式称做单向认证（见图 1.8）。如果 Bob 经营的是因特网商店，且只要求 Alice 的信用卡号码的话，这种认证方式就很有用了。

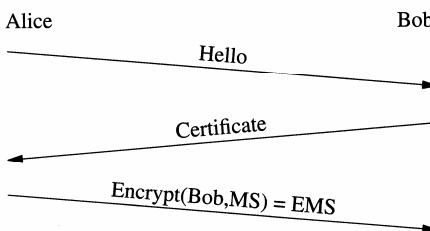


图 1.8 完成单向认证的握手

另一方面，如果 Bob 在意 Alice 的身份的话，我们就需要给该协议再增加一项功能，以让其获得可靠的信息。如果 Alice 使用其私用密钥对 EMS 签名，那么 Bob 就能够对签名进行验证，以确信该 EMS 确实是 Alice 刚刚发送的。

这种新的协议还包含一种新的微妙之处：Bob 将一个称做 nonce 的随机数连同他的证书一起发送给 Alice（见图 1.9）。这样做是为了阻止攻击者重新传送 Alice 发送给 Bob 的所有消息，而使 Bob 确信他正在加入一出新的会话。这种攻击被称做重放攻击（replay attack）。我们在制作密钥的时候就要用到这个 nonce，以确保本次握手所产生的密钥与其他握手所产生的密钥不同。

简单的数据传输协议

在握手结束后，Alice 和 Bob 就会共享一个主密钥，但是那又怎么样呢？这项练习的目的是让 Alice 和 Bob 传输数据，而不是随机数。为了达到那个目标，我们需要一种完全不同

的协议，以完成此类安排的工作。本节的剩余部分就要讲解如何构建此类简单的协议。

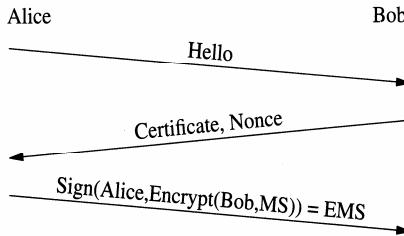


图 1.9 实现相互认证的握手

制作一些密钥

我们所要做的第一件事就是制作一些密钥。一般来说，使用同一个密钥来完成不止一种类型的加密操作被认为是糟糕的思想。我们后面在讨论序列密码（stream cipher）时会看到一种不这么做的原因，不过姑且先将其当作一种更保险的做法。设想攻击者搞懂了如何去对付一种算法（比如说加密算法），如果你使用同一个密钥来完成 MAC 和加密工作，而那种加密又已经被攻破的话，那么通信就会完全处于一种不安全的境地。如果使用不同的密钥，那么即便是加密被攻破，MAC 也仍然是安全的。

在这种情况下，我们需要四个密钥，一对用于完成在每个方向上的加密，一对用于每个方向上的 MAC。为了做到这一点，我们需要用到一种尚未见到过的工具，这种工具通常被称作密钥导出函数（key derivation function, KDF）。KDF 接收主密钥以及（通常如此）其他一些随机数，并从中创建出相应的密钥。这里，KDF 接收主密钥和 Bob 所传送的 nonce。通常使用消息摘要来构建 KDF，我们此处先跳过具体的细节。

为了方便起见，我们将这些密钥标名如下：

E_{cs} ——用于将数据从客户端发送到服务器端的加密密钥

M_{cs} ——用于从客户端发送到服务器端数据的 MAC 密钥

E_{sc} ——用于将数据从服务器端发送到客户端的加密密钥

M_{sc} ——用于从服务器端发送到客户端数据的 MAC 密钥

数据记录

下一项任务就是描述如何对数据打包。如果我们是在像 TCP 这样的可靠协议上工作的话，或许可以设想成在向网络传输数据时对恒定的数据流进行加密。但是那样的话，把 MAC 放到哪儿呢？如果我们只是简单地放在最后进行传输，那么直到处理了所有的数据时才能获得消息完整性。这种方法可不怎么样，因为如果不知道数据是否被篡改的话，就不能够安全地对数据进行操作。

该问题的解决办法就是将数据分割成一系列的记录，每条记录都携带有自己的 MAC。在读取一条记录并检查其 MAC 时就可以知道数据完好与否，以执行下面的操作。我们马上要讲述的内容就是完成这项工作最简单的记录格式。

当我们从线路上将数据读取过来时，需要能够辩识出哪些数据字节是加密数据，哪些是 MAC。如果我们愿意在每条记录中使用固定量的数据的话，就能够知道哪些字节是什么内容了，但是这样工作的并不理想：在某些情况下，我们需要每次传输一个字节，而具有单字

节长的记录就意味着在传输时将数据扩展大约 20 倍。如果我们想要成批传输大量数据的话，这确实不能接受。

由于定长记录效率如此之低，所以我们需要使用变长记录。这意味着需要使用一个长度字段来告诉我们记录数据的截止位置。这个字段必须位于记录的前面，以便知道还要读取多少数据。MAC 可以在前面也可以在后面，但是通常是跟在后面，以允许在计算得出后追加到消息后面（这是一种编程方便的考虑）。这样我们就得到了图 1.10 所示的记录格式。

长 度	数 据	MAC
-----	-----	-----

图 1.10 一种简单的记录格式

因此，如果客户端想要加密长度为 L 的数据块 D 的话，就要经过以下过程：

- (1) 使用 M_{cs} 计算出数据的 MAC。我们称之为 MAC M 。
- (2) 使用 E_{cs} 加密 D ，我们称之为 C 。在这里我们假定对数据进行加密不会扩展数据。正如我们在第 1.13 节所见到的，事实有时并非如此。

(3) 传输 $L||C||M$ ($||$ 代表连接)。

服务器端读取数据则要经历相反的过程：

(1) 从线路上读取 L ，我们现在知道 D 为 L 字节长。

(2) 从线路上读取 C 和 M 。

(3) 使用 E_{cs} 对 C 进行解密从而得到 D 。

(4) 使用 M_{cs} 计算得出 D 的 M' 。

(5) 如果 $M'=M$ ，那么一切正常，对记录进行处理。否则报告错误。

换言之，MAC 计算就如图 1.11 所示。这里，根据我们发送数据的方向， x 要么是 cs 要么是 sc 。

发送：

$$M = MAC(M_x, D)$$

接收：

$$M' = MAC(M_x, D)$$

比较 M 和 M'

图 1.11 MAC 计算

序号

不幸的是，这种简单的协议存在一种安全漏洞。由于记录并没有按照传输的顺序进行标注，因而攻击者可以从线路上拿掉一条记录，转而再发送给接收者（记住，这被称做重放攻击）。要想了解重放攻击存在的理由，考虑消息为一笔金融交易的例子，攻击者想安排两次支付。如果没有应对重放攻击的保护措施，攻击者就能够简单地通过重放消息来请求支付。

当前的协议还可以让攻击者删除记录或重新对记录进行排序。大家很容易就能想象出一些重新排序及重放攻击非常有用的情况来，我们留此作为读者的练习。为了克服这种问题，我们需要使用序号（sequence number）。每一方所传送的第一条记录都被编号为 1，第二条记录为 2 等等。当你收到一条记录后，可通过检查其序号，看是否是你所期望的那条记录。

如果不是，就报错。

当然，序号必须要成为 MAC 输入的一部分，以防攻击者对序号进行修改。最简单的方法就是将序号添加到 D 的前面，然后再进行加密和计算 MAC。不过，如果我们是在可靠协议（如 TCP）上进行通信，那么序号是隐含的，甚至就不必再传了。消息总是依序接收的，因此我们只需在各方维护一个计数器，并使用它来作为 MAC 计算的一部分：

$$M = \text{MAC}(M_x, \text{Sequence}||D)$$

注意，序号并不能阻止攻击者重放 Alice 的所有消息。然而，由于 Bob 每次同 Alice 握手都生成一个新的 nonce，如果攻击者尝试这种攻击的话，Bob 会产生一组不同的密钥（由于其 nonce 是不同的），于是当他对数据记录进行解密时，所得到的不是明文而是垃圾信息，而且不会通过 MAC 检测。

控制信息

还存在一个安全问题需要处理。要知道攻击者可以很容易的伪造数据包。而 TCP 连接的关闭只不过是一个数据包而已，因此攻击者可以很容易的伪造连接关闭。也就是说攻击者能够实施截断攻击（truncation attack），在这种攻击中，攻击者使一方（或双方）确信数据就这么多了，但却比实际的要少。为了克服这种问题，我们需要某种让 Alice 告诉 Bob（或 Bob 告诉 Alice）她已完成数据发送的方法。那样，如果攻击者伪造 TCP 关闭的话，受侵害的一方就可以知道出了什么问题，因为数据结束消息还没有到来。

有多种可以用来阻止这种攻击的技术。举例来说，我们可以用长度为 0 的记录来表示工作完成。然而，总的来说，如果我们能够有某种传输不作为数据流一部分的控制信息的地方就再好不过了。一种简单的实现这种任务的方法就是根据每条记录运载的数据，即普通数据还是控制信息，规定其类型。我们可以通过为每个记录增加一个类型字段来做到这一点，而且还要将这个字段作为计算 MAC 时的一项加以保护，如图 1.12 所示。

长 度	序 号	类 型	数 �据	MAC
-----	-----	-----	------	-----

图 1.12 改进的记录格式

这样就可以使用以下方式计算 MAC：

$$M = \text{MAC}(M_x, \text{Sequence}||\text{Type}||D)$$

当然，如果我们不去定义某些类型的话，类型字段也不会带来什么好处。我们将采用一种简单的方法。如果类型字段为 0，就表示将记录当作普通数据对待。如果为 1，则表示控制信息，就需要由协议对记录的数据部分进行扫描以决定下一步的工作。我们需要采取一种简单的约定，即所有的控制数据都由一个简单的数字构成。我们保留 0 来指示正在关闭连接，而使用非 0 数字来表示错误，如图 1.13 所示。

总结

机灵的读者或许已经猜出来了，我们现在复述了 SSL 绝大多数的核心功能。实现了握手、密钥交换、相互认证以及保密数据传输。还有什么遗漏的吗？

首先，TSP 并不完整。我们并没有说明实现它的具体实现细节。其中的部分细节并不那

么重要。例如，长度字段有多长？两个字节，三个，还是可变的？它统管记录长度的最大值，因此并非不重要，但主要是要足够大——我们必须要达成一致。其他的一些具体细节则稍稍艰涩一些。我们究竟打算使用什么样的算法来完成密钥交换、认证等功能？不管怎样，在协议能够工作之前，我们都需要提供大量的具体信息。

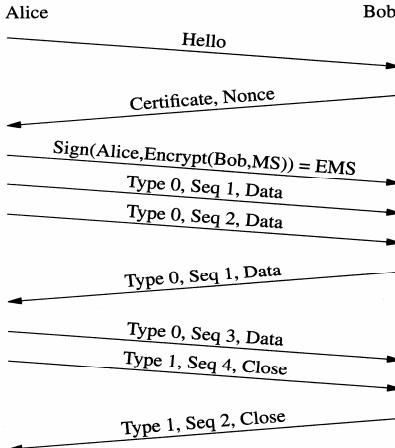


图 1.13 完整的玩具安全协议（TSP）连接

TSP 在具体细节上是不充分的，但更重要的是，它缺少某些想要的功能。它所没有的最重要的一项内容就是磋商（negotiation）功能，我们需要为每种核心任务提供多种算法选择。那样才能在一种算法被攻破的情况下，很容易地切换至其他的算法。我们还希望能够选择是否对客户端进行认证或是采取匿名的方式。尽管如此，它也是一种有趣而富有教育意义的玩具，涵盖了大部分有关的思想。

1.9 出口形式

由于众多计算机软件都是在美国编写的，美国的出口形式对通信安全系统的设计有着戏剧性的影响。美国的出口政策在历史上就由 NSA（国家安全局）驱使。NSA 具有保证政府通信安全和窃听他人通信的职责。出口政策的具体实施总是留给其他部门来处理，原先是国务院（State Department），而现在是商贸部（Commerce Department）的出口管理局（BXA）负责，但是所有重要的决定均由 NSA 做出。

石器时代

直到 1998 年 9 月以前，出口政策的总体思想简单明确。出口用于认证的加密技术是合法的，但是保密性受到严格限制。如果你的产品中包含有任何保密性功能，就必须在出口前经过复查。从未颁布过有关什么可以通过复查的规定，但是一般理解的限制是 40 位强度的加密和 512 位强度的密钥交换。

如果你满足了这些条件，就可以得到称做商品出口权（commodities jurisdiction）的东西，这基本上表示你可以向商贸部申请不必再经过认可的出口软件一般性许可（general

license)。你如果想在大范围内大量销售自己的软件或是提供网络下载的话，这就是你需要办理的批准手续。尽管有此划分，你仍然不能将自己的软件出口到几个受到禁运的国家：古巴、伊朗、伊拉克、利比亚、北韩、苏丹和叙利亚。像法国这样的国家也对加密有着严格的限制。

NSA 还对几种加密算法 (RC2 和 RC4) 有特殊的审批过程。如果你使用其中的一种 (当然是在 40 位模式下) 完成密钥交换的强度不超过 512 位，则复查就会进行得很快，而且几乎总会获得批准。就其他算法而言也可能获得批准，而 IBM 就获批了一种称做 CDMF[Johnson1993]的密钥缩短的 DES。但是一般来说，如果你使用 RC2 或 RC4 的话批准就会顺利一些。大家普遍猜测，NSA 已经对 RC2 和 RC4 进行了透彻的分析而且有可能知道如何迅速加以破解，要么通过密码分析攻击要么通过定制的硬件。不管怎样，NSA 使出口 RC2 和 RC4 变得容易，而出口其他的加密算法则很难。这也是在如此众多的系统中都能见到 RC2 和 RC4 的原因。在普遍要求采用弱强度的规定中有一个知名的例外。如果你出口的系统完全用于金融交易处理 (这基本上意味着只由银行来使用)，那么就可以使用强度更高的加密产品。

● 中世纪

在 1998 年 9 月，加密法规经过修订，可以允许出口 56 位加密和 1024 位的密钥交换。出口产品仍然需要经过复查，但是它们至少可以使用有一定强度的算法。然而，当 DES 被认为乃属弱强度的情况下，仍然不允许真正高强度算法的出口。

● 现代

1998 年放松管制以后，有许多传言称，出口控制将被彻底废除了。在 2000 年 1 月，这些控制大大的放松了。可公开获得的源代码 (“开源” 软件) 简单地粘贴到网上。“零售” 软件仍然要经过一次性技术复查 (不清楚这样做究竟是什么原因)，但是在实际操作中，BXA 已经允许许多具有高强度加密的商用产品的出口。因此，在实际应用中，现在从美国出口高强度加密是合法的，但前面所提到的那七个受禁运的国家除外。

尽管有了这些自由，出口控制的影响仍然存在。许多协议都包含设计用于在可出口环境中工作的功能，SSL 也不例外。首先，它不但包含高强度算法还包含一些弱强度算法，以便 SSL 的实现能够出口。其次，它包含了许多诸如采用临时/短期密钥的 RSA 以及服务器网关加密 (将在第 4 章讨论这两样东西)，这样的迂回措施，设计这些措施用来尽可能地保证安全，同时又是可出口的。

1.10 实际的加密算法

到目前为止，我们所讲的内容似乎让你觉得，给定类型的所有加密算法多少是一个样子。可惜，事实并非如此。例如，每种数字签名算法都稍有不同，而每种密钥交换算法和对称加密算法也是如此。这些细节实际与安全有关，将在第 5 章看到这一点。将在这一章的剩余部分对大多数流行算法进行概括性地描述，以便在后面讲到它们时大家能够知道。

如果你对具体的细节不感兴趣，完全可以跳过这一节，并阅读末尾部分的总结。当我们在本书的其他章节谈到安全问题时，即使不理解算法的细节也应当能明白讨论的意思。

1.11 对称加密：序列密码

对称密码有两个主要的变种，即序列密码（stream cipher）和分组密码（block cipher）。序列密码是最容易理解的类型，所以我们先来讲讲有关它的知识。序列密码的工作方式非常简单。你有一个产生数据序列的函数，每次一个字节。这种数据被称做密钥序列（keystream）。函数的输入是加密密钥，它控制具体产生的密钥序列。没有这个密钥，就无法对密钥序列进行预测。提取密钥序列的每个字节，并将其与明文字节组合产生密文字节。具体是怎样将这两种数据掺和在一起并不重要，不过最流行的方式是使用异或操作（XOR）：

$$\begin{aligned} C[i] &= KS[i] \oplus M[i] \\ M[i] &= KS[i] \oplus C[i] \end{aligned}$$

在本章的余下部分将一直使用这个概念（notion），因此有必要借此机会解释一下。 $C[i]$ 表示密文的第 i 个单元（这里的单元就是指字节）， $KS[i]$ 表示密钥序列的第 i 个单元，而 $M[i]$ 表示消息的第 i 个单元。 \oplus 是表示异或的符号。所以这个式子的意思就是取密钥序列的第 i 个字节，将其与消息的第 i 个字节进行异或而得出密文的第 i 个字节： $A \oplus B \oplus A = B$ 。所以我们可以简单地将密钥与密文进行异或来获得解密的明文。

不幸的是，加密与解密的对称性使得序列密码具有某些危险的安全特征。其中最糟糕的就是绝对不能重复使用密钥序列中的同一段来加密两条不同的消息。试想我们使用同一个密钥对两条消息 M 和 M' 进行加密。如果攻击者知道 M 的话，他就可以很容易地通过计算 $M \oplus C$ 来得出 KS 。一旦攻击者知道了 KS ，他就能在给定 C' 的情况下计算出 M' ：

$$M' = (M \oplus C) \oplus C'$$

令人吃惊的是，这种情况经常发生，因为许多通信格式都包含有大量可预测和重复的数据，这可以让攻击者猜出给定消息中的很大一部分信息。如果使用相同的密钥对另一条消息进行加密的话，攻击者就能利用这些知识对那些消息进行解密。

更糟的是，攻击者甚至在不知道任何一条消息的明文的条件下实施攻击。对两条密文执行异或就会完全抵消密钥的作用：

$$C \oplus C' = M \oplus M'$$

由于在给定它们的异或的情况下存在众所周知的用于解密两条明文的方法，存在这样两条密文的后果是灾难性的。如果你打算使用序列密钥对多项文本进行加密，就必须使用分离的密钥或密钥序列中的不同部分。

即便密钥序列只被使用一次，对使用序列密码加密的数据进行篡改也颇为容易。设想你作为攻击者知道消息 M 和密文 C 的内容，想改变消息以使其解密为一个新消息 M' 的话，就可以轻而易举地计算出导致这种结果的密文 C' ：

$$C' = C \oplus (M \oplus M')$$

即便你不知道明文，也能在其中进行可预测的修改。因为每个密文位都一对一地与一个明文位对应。结果，通过改变密文中对应的位，就可以对任何明文位进行反转。所以，如果

你使用序列密码，就必须结合使用一种高强度的 MAC。

RC4

惟一受到广泛关注和应用的序列密码就是 RC4。RC4 由 Ron Rivest 设计，并在 RSA 数据安全公司（RSADSI）的产品中作为一种专有密码使用了相当长的时间。1994 年有人匿名向计算机服务邮件列表中投递了一份号称 RC4 加密算法（cipher）的代码。接续的测试论证了该算法与 RSA 的 RC4 实现兼容，大家广泛认为该算法要么是通过对 RC4 代码进行反汇编的逆向工程（Reverse Engineer）所得，要么就是有人泄露了 RSA 的源代码。由于 RSADSI 拥有 RC4 名称的商标权，该加密算法一般被称为 Alleged RC4 或 Arcfour。

RC4 是一种密钥长度可变的算法，其密钥长度可以在 8~2048 位之间。不管密钥有多长，密钥都被扩展为一张固定尺寸的内部状态表，所以无论使用什么长度的密钥，该算法都运行得一样快。SSL 和 TLS 总是使用密钥长度为 128 位（16 个字节）的 RC4。RC4 的速度非常快，一台奔腾 II/400 的机器可获得 45MB/s 的速度。

1.12 对称加密：分组密码

对称加密的另一种主要类型就是分组密码（Block Cipher）。分组密码可以被看成一张巨大的查找表，以字节分组（通常为 8 或 16）的形式来对要加密的数据进行处理，而每个可能的明文分组对应表中的一行信息。使用密钥选择表中的一列。于是，要加密一个分组，就要找到与密钥对应的列，查找这张表来找到与待加密分组对应的行，那个位置的条目就是密文。当然，这样一张表有可能大得难以管理，因此在实际应用中，通常由一个函数来替你完成这种计算，基本思想就是用一种函数来模仿某种随机的排列表。

换一种方式来看，分组密码就是带有两个变量（即密钥和输入分组）的函数。已知两个函数 E（用于加密）和 D（用于解密），M 为明文，C 为密文。

$$\begin{aligned} C &= E(K, M) \\ M &= D(K, C) \end{aligned}$$

我们目前所掌握的技术能够对大约 8 个字节进行加密。但我们想要加密的消息通常要比这个长得，所以必须要找到一种可以完成此项工作的方法。最显而易见的方法就是使用一种称做电子代码簿（Electronic Codebook, ECB）的模式。简单来讲，我们将消息分割成分组大小的数据块，并使用我们的加密算法对各块进行加密：

$$\begin{aligned} C[i] &= E(K, M[i]) \\ M[i] &= D(K, C[i]) \end{aligned}$$

这种方式很直观，但却存在明显的不足。试想如果两个分组 $M[j]$ 和 $M[k]$ 完全相同，那么 $C[j]$ 与 $C[k]$ 就会完全一样。如果消息中存在一种频繁出现的模式，攻击者就能检测到它，并对明文有所了解。我们可不需要这样的特性。

密码分组链接（Cipher Block Chaining, CBC）解决了这种问题。在 CBC 模式中，对每个明文分组的加密依赖于前一密文分组的密文。这是通过在加密前将前一个密文分组与明文进行异或来实现的：

$$\begin{aligned} C[i] &= E(K, M[i] \oplus C[i-1]) \\ M[i] &= D(K, C[i] \oplus C[i-1])) \end{aligned}$$

在 CBC 模式中，即便我们拥有两个相同的分组，它们也不太可能加密产生相同的结果。这是因为前一个密文分组是不同的。考虑 $M[j]$ 与 $M[k]$ 相同的情况， $M[j-1]$ 与 $M[k-1]$ 是不同的。在那种情况下， $C[j-1] \neq C[k-1]$ ，于是 $C[j-1] \neq C[k]$ 。

惟一遗留的问题就是怎么处理第一个密文分组。既然不存在可以进行异或的前导密文分组，我们就不能沿用通常的做法。当然也可以使用全为 0 的分组，但是那样的话攻击者就能够判定两条消息的第一个密文分组是否相同。为了解决这个问题，我们生成某种称做 IV（初始化向量）的随机分组。在加密前，使用 IV 与消息的第一个密文分组进行异或。通常这个 IV 与消息一起发送，不过还可以根据双方所共享的某个值来产生。IV 不一定要保密，但是它们应当是全新的，即对每条消息都不相同。

还有其他大量用于操作分组密码的模式，其中包括 OFB（Output Feedback，输出回馈）和 CFB（Cipher Feedback，密码回馈）。不过，CBC 是最流行的一种，而且是惟一与 SSL 相关的一种。

CBC 模式的分组密码总是会使数据稍微扩大一些，而序列密码却不会这样。发生这种情况的原因就是数据输入必须为分组大小的整数倍。由于大多数数据的尺寸均不是分组大小的整数倍，因而就需要对最后一个分组进行填充，使其与分组边界对齐。这里有些棘手，因为在进行解密的时候，你需要去除所有的填充，而其他内容都不能动。通常所采用的方式就是使用值等于所增加的填充字节数的字节来进行填充。如果消息确实结束于分组边界的位置，那么就需要附加一整块填充内容。在对大尺寸消息进行解密时，这种数据扩展不成问题，但是如果每次加密一个字节，就会在产生每个加密分组时产生大量的数据冗余，这样就会浪费网络带宽。

CBC 模式的分组密码要比序列密码更容易安全地使用。只要改变 IV，即使使用同一个密钥来完成不同数据的加密也是安全可靠的。即便不改变 IV，所有攻击者所了解的顶多也就是两条消息是一样的，而重复使用密钥则会完全攻破序列密码。

在经受完整性攻击时，分组密码也工作的好一些。攻击者无法通过改变密文来对明文进行可预测的修改。对密文的任何修改都会同时破坏掉这个分组及下一个分组。这条规则的一个重要例外就是第一个分组。如果 IV 包含在消息中而又没有施加完整性保护，那么攻击者就可以通过修改 IV 而对第一个分组进行可预测的修改。由于 IV 是异或进去的，所以攻击者就可以通过反转 IV 中的同一位来反转他所选择的任何一位。总的来说，当用分组密码进行加密时，使用 MAC 仍然很重要。不过，攻击者无法像对以序列密码方式加密的数据那样，对分组密码加密的数据实施精确地攻击。

当使用 CBC 模式时，你需要留心 CBC 翻转（rollover）的问题。试想两个数据分组 $M[i]$ 和 $M[j]$ 加密为相同的值 C。如果 $M[i+1]=M[j+1]$ ，那么 $C[i+1]=C[j+1]$ 。因此攻击者就会知道 $M[i+1]=M[j+1]$ 。一般而言，分组大小为 X 位的密码每 $2^{x/2}$ (DES 是 2^{32}) 个分组就会有两个密文值产生冲突（一样）。所以对于给定的密钥，不要加密超过这个数据量的数据就非常重要。

DES

DES（数据加密标准）至今仍是应用最为广泛的对称式加密算法。DES 是在 19 世纪 70

年代为了响应 NBS（国家标准局），就是现在的国家标准与技术协会（NIST）的提议由 IBM 设计的。尽管 IBM 拥有 DES 的专利，但他们还是达成一致，使 DES 可供自由使用并且在 [NIST1993a] 中进行了标准化。

DES 是一种具有 56 位密钥的 64 位分组密码。意思就是，数据是以 8 字节大小的分组进行加密的，且密钥空间为 56 位。DES 密钥实际上是 64 位长，不过每个字节的低位用做检测传输与密钥打包与拆包（wrapping/unwrapping）错误的奇偶校验位。在实际应用中，由于 DES 密钥通常以公用密钥技术进行打包，所以奇偶校验是多余的，通常忽略奇偶校验位。

当然可以肯定 DES 是一种经受最广泛分析的公开加密算法。在当初起草时，据悉 NSA 协助进行了设计，而其中的一些设计决定既不透明也没有加以解释。开发 DES 的 IBM 密码学工作者对他们所了解的攻击三缄其口，他们显然了解潜在的、这个世界其他人所不清楚的弱点。自然，曾有人对 DES 是蓄意被弱化的表示关注。

早期关注的焦点是算法中一种称做 S-盒的特定部分，而且在很长一段时间内，都有人担心 S-盒被设计成为 NSA 提供了密码分析后门。在 1990~1993 年，Eli Biham 和 Adi Shamir 通过使用他们的差分密码分析（differential cryptanalysis）技术来研究 DES[Biham1991a、Biham1991b、Biham1993a、Biham1993b]，清楚地表明 DES 的 S-盒经过了抵御差分攻击的优化。只有极为较真的人仍然对 S-盒的设计抱有疑虑。

目前还没有发现好的针对 DES 的分析攻击。不过现在 DES 相当的脆弱，基本原因是由密钥太短（56 位），而计算机的速度却越来越快所造成的。1997 年，人们通过分布式计算（在因特网上大量的计算机之间分配搜索空间），使用穷举搜索恢复了一条 DES 密钥。在 1998 年的深度攻击（Deep Crack）活动中，一台定制的 DES 搜索机在 56 小时内恢复了一条 DES 密钥。而在编写这本书的时候，这项记录少于 24 小时。DES 现在只适用于低价值或短生存期的信息。

3DES

由于 DES 承受了如此猛烈的攻击，当密钥长度显得太短的时候，一种非常诱人的可能就是简单地对数据进行多遍加密，即一种称之为超级加密（superencryption）的过程。不幸的是，仅仅应用两遍 DES（2DES）的结果并不比 DES 安全多少。如果你有 2^{56} 块可以使用的内存的话，一种名为中位会面（meet-in-the-middle）的攻击就可以让你用与攻破 DES 相同的时间攻破 2DES。因此，人们不得不对数据进行 3DES（三重加密）。3DES 的有效长度为 112 位，该长度就是你原本天真地期望 2DES 所具有的长度。

实际上，进行中位会面攻击可以在时间与内存之间加以权衡。如果允许使用更多的 CPU 时间，则可以降低对内存的需求。然而，时间-内存的乘积一直会是 2^{112} （参见[Menezes1996]）。

我们刚才所讲的并没有完全说明 3DES 是如何工作的，并不是所有三项操作均必须为加密操作。3DES 最流行的版本使用称做加密-解密-加密（EDE）的模式。意思就是说使用密钥 1 进行加密，使用密钥 2 进行解密，接着使用密钥 3 再进行加密。这样的结果与加密-加密-加密（EEE）的方式一样安全，而且还有一个主要的好处：如果你将三个密钥设为同一个的话，就等同于单次 DES，也就意味着可以使 3DES 的硬件与 DES 硬件进行互操作。这一种通常被引用为 3DES-EDE。

3DES（并不奇怪）大概要比 DES 慢 3 倍。DES 在起初使用时就不怎么快，因而在性能

关键的应用中人们经常谨慎的使用 3DES。而且使用 192 位的密钥资料（168 位的密钥和 24 位的奇偶校验位）却获得 112 位强度的安全对许多人来说好像不怎么舒服。

有可能使用带有两个密钥（两次加密操作使用一个密钥，而解密操作使用第二个密钥）的 3DES，但是需要 $O(t)$ 空间和 $2^{120-\lg(t)}$ 次操作的双密钥 3DES 存在一种已知的弱点，意思很清楚。这种攻击并不真的可行，无法利用这弱点来对三密钥 3DES 实施攻击。因此，保守措施就是使用三密钥 3DES。

RC2

RC2 是一种由 Ron Rivest 发明的分组密码，他还发明了 RC4。与 RC4 一样，RC2 也是 RSADSI 的商业秘密，同样它最后也是由不知名的人发布出来。之后，Ron Rivest 发布了一份描述 RC2 的 RFC。但由于 RSADSI 享有 RC2 名称的商标权，该 RFC 将其指代为 RC2(r)。

RC2 是采用一种经过变形的变长密码，它还具有一种可变的有效密钥长度。因此，你可以使用 64 位密钥，但是却可以通过 2^{40} 次操作予以攻破，这可使其便于出口。与 DES 类似，它具有 64 位分组。在 SSL 中，RC2 总是使用 128 位密钥和 128 位有效密钥长度。

AES

正如前面所提到的，尽管 DES 没有严重的已知密码弱点，但是密钥长度太短了，早已过了该进行更换的时间。1997 年 NIST 招标高级加密标准（Advanced Encryption Standard，AES）。AES 要具有至少 128 位的分组大小，以及 128、192 和 256 位等三种密钥长度，这使得其强度足以满足可预见未来的要求。同时，AES 算法在软件上应当是快速的，而几乎所有提交的算法都比 DES 快。

在写这本书的时候，NIST 已经将最终的候选者削减为 5 个，即 MARS[Burwick1999]、Serpent[Anderson1999]，Twofish[Schneier1998]，Rijndael[Daemen1999] 和 RC6[Rivest1995]。对所提交算法的一项要求就是如果被选中则该算法必须在特许免费（Royalty-free）的基础上供大家使用，所以使用 AES 将会是免费的。

加密算法	密钥长度	速度（兆字节/秒）
DES-CBC	56	9
3DES-CBC	168	3
RC2-CBC	Variable	.9
RC2-CBC	Variable	3*
RC4	variable	45

*较为快速的 RC2 基准测试（Benchmark）是在一台运行 Windows 2000[Dai2000] 的赛扬 450 机器上完成的，其余的基准测试则是使用 OpenSSL 得出的结果。

图 1.14 一些常见的密码算法（OpenSSL、FreeBSD、Pentium II 400）

总结

图 1.14 总结了我们所讨论的对称算法。正如你所见到的，RC4 的速度最快，而 3DES 和 RC2 相当慢。尽管 RC2 和 RC4 具有可变长尺寸的密钥，但是不管使用多长的密钥，其运行速度也是一样的。与之相比，3DES 要比 DES 慢得多，不过也更安全。现在 128 位密码的出口已不受限制，除了与 40 位系统保持兼容外，实在没有什么理由使用 RC2，因为不管从

安全上还是性能上来说它相对于 3DES 均没有优势可言。

1.13 摘要算法

就协议设计而言，所有的摘要算法都非常相似，惟一的区别就是输出尺寸的不同。两种最流行的算法为：由 Ron Rivest 设计的 MD5，以及由 NIST（据说是 NSA 的帮助下）设计的 SHA-1。

MD5 和 SHA-1 都是从一个共同的祖先导出的。此外，Rivest 还设计了 MD4。SHA-1 基于 MD4，但是具有 160 位的输出，因而强度要高一些。由于称为生日悖论 (Birthday Paradox) 的原因，找到两个具有相同摘要值的消息 (冲突) 的难度大约为密钥空间的平方根。因此，也就是说需要 2^{64} 次操作才能找到 MD5 的一次冲突，而对于 SHA-1 则是 2^{80} 次操作。近来，一些迹象表明可能存在一种比蛮力攻击更容易的，找到 MD5 中冲突的方法[Dobbertin1996]。所以保守的设计都使用 SHA-1。

生日悖论这个名字源于一个数学派对游戏。如果房间里有超过 23 个随机挑选的人，则具有相同生日的两个人的概率将超过 50%。该结果使大多数人都感到吃惊，因为他们一直以为要达到这个概率大约需要 120 人。这里的关键点就是我们计算的是两个人具有相同生日的概率，而不是具有给定生日的两个人的概率，已经划归给某人的生日空间在每次增加一个人的时候都会被耗掉。就散列冲突而言则是在更大范围内的类似效果，即我们所感兴趣的是某两条消息具有相同摘要的概率。

一种类似 MD4 的称做 MD2 的老算法有时用来对非常“古老”的证书进行签名。现代系统都不会使用它了。过去也有一种更老版本的 SHA，但是经过修订后更正了一个弱点，创建了 SHA-1。图 1.15 将 MD5 和 SHA-1 进行了比较。

摘要	输出 (位)	速度 (MB/s)
MD5	128	65
SHA-1	160	31

图 1.15 摘要算法的比较 (OpenSSL、FreeBSD、Pentium II 400)

1.14 密钥的确立

公用密钥加密的两个首要用途就是密钥的确立 (Key Establishment) 和数字签名。本节讲述密钥的确立，下一节讲述数字签名。实际上存在两种类型的密钥确立。在密钥交换 (key exchange) 或密钥传输 (key transport) 中，一方产生了一个对称密钥并使用另一方的公用密钥对其进行加密。在密钥磋商 (key agreement) 中，双方协作产生一个共享密钥。我们将要讲到两种算法，RSA 和 Diffie-Hellman(DH)。

RSA 可以用做一种密钥传输算法，DH 是一种密钥磋商算法。

RSA

当人们想起公用密钥加密的时候，RSA 是大多数人所想到的公用密钥算法。该算法在

1977 年由 Ron Rivest、Adi Shamir 和 Len Adleman（即 RSA 的由来）[Rivest1979]发明。从高层上讲，RSA 非常简单。每个用户都有一个公用密钥和一个私用密钥。公用密钥可以自由分发，而私用密钥则必须保密。

RSA 公用密钥实际上是由两个数字组成的，即模数（modulus）（通常以字母 n 来表示）和公共指数（public exponent）（通常以字母 e 来表示）。模数为两个非常大的素数（习惯以字母 p 和 q 来表示）的乘积，而 p 和 q 也需要保密。RSA 的安全性就是基于对 n 进行因数分解以得到 p 和 q 的难度。

私用密钥为另一个数字，通常被称做 d ，且只有在知道 p 、 q 和 e 的情况下才能计算得到。当我们谈论 RSA 密钥长度的时候，实际上是在谈论模数的长度。RSA 的公共指数 (e) 必须与 e 和 $(p-1)(q-1)$ 互素。为了方便起见，通常挑选几个小素数中的一个作为 e （常常为 3,17 或 65537）。使用一个小的 e ，可以使得公用密钥的操作更快一些。选定了 e ，就可以像下面这样计算得到 d ：

$$d = e^{-1} \bmod((p-1)(q-1))$$

要想使用 RSA 和公用密钥(e,n)加密消息 M ，就计算 $C = M^e$ 。 $\bmod n$ 的意思是指计算取模 n ，即取 M 的 e 次幂，然后再除以 n 得到余数。对该消息进行解密则要求有对应的私用密钥。如 $2^5 \bmod 10 = 2$ 和 $2^5 \bmod 7 = 4$ 。要对消息进行解密就计算 $M = C^d$ ，姑且先认为是这么工作的。

这样解释是在假定消息是一个数字，而实际上，消息往往是一组字节串。你需要使用一些将字符串转换为数字的规则，这个数字多少应该与 n 的大小相当（但不能比 n 更大）。这么要求的原因是我们想要确保 $M^e > n$ 。如果不是这样，那么 $C = m^e \bmod n = M^e$ （即 $\bmod n$ 没起什么作用）。攻击者就能够轻而易举地通过取 C 的第 e 次方根来恢复 M 。另一方面，如果 $M^e > n$ 的话，就不可能存在这种攻击。使 M 与 n 的大小大致相同就可以确保 $M^e > n$ 。

PKCS #1[RSA1993b]中规定如何完成此类转换的标准过程。我们在这里进行一下总结，因为在 SSL 上应用它的时候，事实表明该过程存在安全问题。PKCS#1 版本 2[Kaliski1998a]解决了这些安全问题，但是当前还没有被广泛使用。

要做的第一件事就是计算出格式化加密分组所需的大小。如果 n 是一个 L 位的数字，那么加密分组就有 $L/8$ 字节长（只入不舍）。第一个（高位）字节总是 0，而第二个字节设置为分组类型。对加密来说，这是 2，对签名来说就是 1。这样就可以确保所形成的数字比 n 稍小一些。

消息存放于加密分组的低位字节中，并在之前安放一个 0 字节。（图 1.16 以图片的形式描述了这种填充）其间以填充字节填充。对于分组类型 1 来说，填充字节的值是 255，对于分组类型 2，填充数据则由非零随机数组成。这里的思想就是，当对一个消息重复进行签名的时候，签名应当是一样的。但是，如果是对同一个消息反复进行加密的话，加密分组则应当是不同的。当恢复加密分组时，通过从第三个字节开始一直向右推进到一个 0 字节为止，就能够找到数据的起始位置。0 字节是最后一个填充字节。

至少要有 8 个填充字节才能使得加密可以阻止明文猜测攻击。没有填充，攻击者就可以使用公用密钥尝试加密明文，来查看是否产生给定的密文。有了填充，攻击者就必须针对每种明文尝试 2^{64} 种不同的填充组合，这会使此类攻击困难得多。

一旦完成了加密分组的格式化工作，就可以通过将第一个字节作为最高有效位，最后一个字节作为最低有效位来将其转换为一个整数。这与 big-endian 字节顺序是相同的。

00	01	ff ff ff ff ff ...	00	Data
分组类型 1				
00	02	伪随机非零字节	00	Data
分组类型 2				

图 1.16 PKCS-1 的填充

RSA 在美国享有专利，不过该项专利已于 2000 年 9 月 20 号到期[Rivest1983]，专利由 RSADSI 拥有。尽管有专利权的存在，但 RSA 还是公用密钥加密事实上的标准，而且几乎每一家使用公用密钥的公司都授权使用它。

为了使用 RSA 来完成密钥的传输，需要产生一个随机的会话密钥。（在恰当地填充后）使用接收方的公用密钥进行加密。然后接收方再对消息进行解密，去掉填充字节。最终通过这种方式来共享会话密钥。

DH

Diffie-Hellman (DH) 是首个公开发表的公用密钥算法[Diffie-1976]。它是一种密钥磋商算法，而不是密钥交换算法。发送方不是产生一个密钥并针对接收方进行加密，而是由发送方与接收方共同 (collectively) 产生一个对他们来说是私有的密钥。发送与接收双方共同拥有这个密钥对。要想计算出磋商后的密钥，发送方组合使用其私用密钥和接收方的公用密钥。接收方组合使用其私用密钥和发送方的公用密钥。DH 公用密钥经常被称作份额 (share)，因为它是一方在密钥磋商中所站的份额。DH 也是使用模幂运算，但是模数 (modulus) 是一个大素数 (通常被称做 p)。这个模数是众所周知的，而且要在发送方与接收方之间共享。发送方与接收方还必须共享另外一个称作发生器 (g) 的数字。选取 g ，使得对于任何 $Z < p$ 的值都存在一个使 $g^w \bmod p = Z$ 的值 W 。因此它能“产生”从 1 到 $p-1$ 的所有数字。要想产生一个密钥，就产生一个比 p 小的随机数 (X)，并计算出 $Y = g^X \bmod p$ 。 X 为私用密钥，而 Y 是公用密钥。我们将使用 X_s 和 Y_s 来指代发送方的密钥，使用 X_r 和 Y_r 来指代接收方的密钥。

当要计算共享密钥 (ZZ) 时，发送方计算：

$$ZZ = Y_r^{X_s} \bmod p = (g^{X_r})^{X_s} \bmod p = g^{X_r \cdot X_s} \bmod p$$

接收方计算：

$$ZZ = Y_s^{X_r} \bmod p = (g^{X_s})^{X_r} \bmod p = g^{X_s \cdot X_r} \bmod p$$

很容易就能知道这些数字是相同的，发送方与接收方因此共享同一个数字。要想说明攻击者无法计算得出 ZZ 则困难一些。实际上，没人知道攻击者无法计算出 ZZ 。然而，普遍认为要想做到这一点要求拥有 X_s 或 X_r ，这就要求在给定 Y 时计算得出 X ，这是一种称之为离散对数 (discrete logarithm) 的问题。当前还没有高效计算离散对数的方法。

RSA 与 DH 之间最为重要的操作差别就是，在 DH 中所有通信方必须共享 p 和 q (一般被称为组参数或组) 这常常意味着一群人全部要使用同一个组。然而，让接收方简单的随机

产生他自己的组是可能的。在这种情况下，他必须安排将 g 和 p 传送给发送方，这经常是在其证书中实现的。发送方在接收方组中产生一个临时密钥，只用于加密传送给接收方的信息。这被称做（ephemeral-static DH）临时-静态 DH。如果接收方也产生一个用于此类传输的密钥的话，就被称做（ephemeral-ephemeral）临时-临时 DH。

临时-临时模式的优点是，如果各方在传输完成时删除了自己的密钥，那么即便在此之后有人攻陷了他们的计算机，攻击者也无法读取任何旧有的通信信息，因为机器上不存在任何私用密钥。如果其中一个密钥是静态的，攻击者就有可能使用那个密钥打入这台计算机并恢复它。这种属性被称做完美向前保密（Perfect Forward Secrecy, PFS）。

注意，在 DH 中与在 RSA 中不同，接收方必须要知道发送方的公用密钥才能对消息进行解密。DH 密钥的强度有赖于两个值，即 p 的大小和 X 的大小。经常将 X 选取为与 p 相当的大小，所以只有 p 的大小是有关的。然而，出于对性能的考虑，可以将 X 选取为比 p 小得多的值。在那种情况下，必须大致将 X 选取为你想从 ZZ 产生的对称密钥长度的两倍。例如，如果磋商使用 DES 密钥，则 X 就至少要比 112 位长。注意，这并不意味着 X 必须大于 2^{112} ，只不过必须要在至少为 2^{112} 的区间内随机的选取。将 X 选取为能够像 p 一样抵御攻击的长度也是令人满意的（请参见 Menezes1996）。由于对 X 进行攻击要比攻击 p 困难，所以 X 仍然可以比 p 小许多。对于 1024 位的 p 来说，选用 160 位的 X 就相当不错了。注意，这句话暗指 1024 位的 DH 密钥实在是太短了，不足以 3DES 密钥提供充分的安全。DH 密钥是整个过程中最为虚弱的一环。类似的说明也适用于使用 1024 位保护的 3DES 密钥。

1.15 数字签名

RSA

除了将公用密钥与私用密钥的地位交换一下之外，使用 RSA 来完成数字签名与使用它来完成密钥传送几乎是完全一样的。要进行签名，就要计算出消息的摘要值并使用其私用密钥进行“加密”。接收方对摘要值进行“解密”，并将其与根据消息独立计算得出的摘要值进行比较来进行验证。如果匹配，则签名就是有效的。

RSA 签名中惟一较为难懂的部分就是分组格式化。它不仅仅是对消息摘要值进行加密，而是加密一个 DER 编码的 DigestInfo 结构[RSA1993b]。DigestInfo 结构包括以算法标识符来表示的算法（如 MD5 或 SHA-1），该结构是一系列特定的字节加上摘要本身的内容。（参见图 1.17）。

```
DigestInfo ::= SEQUENCE{
    digestAlgorithm DigestAlgorithmIdentifier,
    digest Digest }
```

```
DigestAlgorithmIdentifier ::= AlgorithmIdentifier
```

```
Digest ::= OCTET STRING
```

图 1.17 DigestInfo 结构

要想明白为什么要这么做，就请考虑如果直接对摘要进行签名会发生怎样的情况。设想存在广泛应用于签名的摘要算法 $H1$ 。现在假设 $H1$ 遭受了灾难性地破坏——即有可能产生摘要为给定值的随机消息。现在进一步假设我们使用另一种不同的算法 $H2$ 对消息 M 进行签名，这种算法的强度仍然很好。而攻击者可以使用 $H1(M')=H2(M)$ 的特性产生一个新的消息 M' 。如果他将我们的签名附加于他新产生的消息 M' （将其标注为用现已攻破的散列算法 $H1$ 进行签名）上，该签名仍然能够通过验证！于是，这位攻击者就成功的产生了显然是由我们签名但却由他来选择的消息。将标识符包括在签名中就可以阻止这种攻击。

● DSS

数字签名算法 (Digital Signature Algorithm, DSA) 由 NSA 发明并在 1991 年由 NIST 制订。当时的思想是要有一种用于数字签名而不是密钥确立的算法。这种算法原本是为了替换 RSA 的，而 RSA 可以同时用于两种目的。这样做就会满足了 NSA 要求的有广泛可用的认证技术，但又对加密技术进行限制的目标。DSA 被标准化为一项称作数字签名标准 (Digital Signature Standard, DSS) 的联邦信息处理标准 (FIPS-186)，现在 DSA 作为 DSS 受到广泛认同。

DSS 基于与 DH 一样的密码数学基础，即素数域中的模幂运算。我们这里不再费神对其进行描述，因为它相当复杂，而且有关细节也无关紧要。需要知道的重要一点是，密钥多少与 DH 的相同，但有一个显著的差异：构造 p (大素数) 时要使 $p-1$ 可以被另一个称作 q 的素数除尽，这样可以使该算法更快一些。

DSS 签名由两个称做 r 和 s 的大数 (160 位) 组成。与 RSA 不同，DSS 只能与 SHA-1 一起使用，而 RSA 则可以与任何摘要算法一起使用 (实际上，可以与任何 160 位的摘要算法一起使用，只是标准要求使用 SHA-1)。DSA 签名不包括摘要标识符，所以允许使用多种摘要算法就会使 DSA 遭受我们前面所描述的替换攻击。

要知道的重要一点是，DSS 对签名进行验证的总体方式与 RSA 不同。对于 RSA，你从签名中恢复消息摘要并将其与计算得出的消息摘要进行比较。DSS 则需要基于消息摘要和签名进行计算并返回是或非的答案。使用 DSS 没有办法恢复发送方计算得出的消息摘要，这使得将基于 RSA 的协议及实现在转换为 DSS 时相当混乱。

DSS 的专利状况并不清楚。NIST 声称它是免费的，但是 RSADSI 曾经声称它由 Claus Schnorr 的专利 [Schnorr1991] 所涵盖，而 RSADSI 又碰巧拥有这项专利的专有许可权。

● 总结

图 1.18 使用 Wei Dai 的 Crypto++ 在一台运行 Windows 2000 Beta 3 的赛扬 450 机器上总结了 RSA、DH 和 DSS 的性能。首先注意到的是，RSA 的公用密钥操作 (加密和签名验证) 要比私用密钥操作 (解密和签名) 快许多。这并不是 RSA 特有的属性，不过还是这样的好。这实际上是人们选取 RSA 密钥的方式的属性，但由于它可以使公用密钥操作快许多，所以几乎每个人都这样选取密钥。

DSS 的性能则更为对称，对于短小的密钥其私用和公用密钥操作要比 RSA 的私用密钥操作慢，但对于大密钥则要快些。因为 DH 对于双方来说都是一样的，其性能通常是对称的 (尽管我们将在第 6 章看到一个特殊的例外)。一般来说，如果你担心性能的话，RSA 就是一种更好的选择。不过要是真的对性能抱有疑虑的话，你或许可以考虑购买加密硬件，因为在那种情况下速度差异就显得无关紧要了。

算法	密钥长度	毫秒/操作
加密	512	.2
解密	512	4
加密	1024	1
解密	1024	27
磋商	512	5
磋商	1024	19
签名	512	4
验证	512	.3
签名	1024	27
验证	1024	.7
签名	512	4
带有预算算的签名	512	2
验证	512	5
签名	1024	15
带有预算算的签名	1024	5
验证	1024	18

图 1.18

1.16 MAC

我们将要使用的惟一一种 MAC 算法是 HMAC[Krawczyk1997]，它使用散列算法来构造具有可证明安全特性的 MAC。在 1996 年引入 HMAC 之前，协议设计者使用各种专门的 MAC 结构，但是对其安全从没有十足的把握。惟一的例外是一种基于 DES 的称做 DES-MAC 的算法[ANSI1986]。该算法经受了广泛的分析，但是速度相当慢。HMAC 在大多数应用中很快取代了其他所有的算法。

HMAC 使用嵌套的密钥控制摘要。也就是说，我们先计算出输入的密钥和数据摘要，然后再使用该摘要值作为另一个密钥控制摘要的输入。实际的算法为：

$$HMAC(K, M) = H(K \oplus opad || H(K \oplus ipad || M))$$

H = the digest algorithm we're using 我们使用的摘要算法

$ipad$ = a string consisting of byte 0x36 由字节 0x36 组成的字符串

$opad$ = a string consisting of the byte 0x5c 由字节 0x5c 组成的字符串

K 对于所有的标准信息摘要均为 64 字节长。如果你的 K 短于 64 个字节，则自动用零字节在其右部进行填充。

1.17 密 钥 长 度

我们谈论某种算法强度的最常见的一种方式就是谈论其密钥的长度。这样所得出的结果

颇具欺骗性，因为密钥长度并不意味着一切。尽管如此，它还是一种有用的衡量方法，而且是最经常使用的一种。我们在这里试图阐明的是，对密码进行攻击所需要的计算资源。而密钥长度提供了有关这种攻击难度的上限，它衡量了以蛮力攻击方式攻破它的难度，简单来说就是攻破系统所需耗费的计算量。

我们需要弄清楚的重要一点是，计算机的速度越来越快，所以随着攻击加密算法的计算机性能的提高，这些算法的强度会逐渐削弱。一般估计计算机的速度每 18 个月就要翻一翻。这种规律被称做摩尔定律，以纪念首先注意到这种现象的 Gordon Moore。大家普遍以为这条性能提升的规律最终会放慢，但是许多断言何时会放慢的预测后来都被证明是错误的，预测速度不久后就会慢下来显然是不可靠的。

实际上，Moore 所观察到的是芯片上部件的数量每 18 个月翻一翻，但是摩尔定律演化成一组指代计算机性能趋势的经验法则。

● 对称算法

攻击一种设计良好的对称密码的惟一方式就是尝试每个密钥，直到找到能行的那个。在这种情况下，密钥的长度直接控制着密钥的强度。40 位的密钥长度允许有 2^{40} 个不同的密钥，也就需要 2^{40} 次密码学操作才能搜索完所有可能的密钥。56 位的密钥需要 2^{56} 次操作，以此类推。

这里需要明白的最重要的一点是随着密钥长度的增加，密码强度会呈指数增长，而不是线性增长。一个 56 位的密码大约比 40 位的密码强上 65000 (2^{16}) 倍：需要花费多 2^{16} 倍的资源进行攻击。因此，如果 40 位的密码使用 X 台机器耗时 Y 可以攻破的话，那么 56 位的密码就得需要 aX 台机器花费 bY 时间才能攻破，而 ab 的乘积必定等于 2^{16} 。

1996 年一个称做“七人团体”（Matt Blaze、Whitfield Diffie、Ron Rivest、Bruce Schneier、Tsutomu Shimomura、Eric Thompson 和 Michael Weiner）的由密码学家组成专家小组推断 90 位的长度应当足够提供 20 年的安全保证[Blaze1996]。对于生命期更短的信息来说，75 或 80 位就够了。任何短于 70 位的密码都是值得怀疑的，而任何短于 56 位的密码都能够遭受商业攻击者的攻击。几种广泛使用的密码都具有 128 位的密钥，这显然是太长了，而新的高级加密标准（AES）就具有 128 位的密钥。

知道密钥长度并不是密码强度惟一的决定因素是非常重要的。许多具有长密钥的密码都被新的分析技术攻破。例如，LUCIFER——DES 的祖先，具有 128 位的密钥，但却能够被差分密码分析攻击。尽管如此，如果你使用的加密算法的密钥过短，那么肯定会有麻烦。

● 非对称算法

非对称算法的密钥长度与对称算法的密钥长度并没有直接的可比性。由于引领非对称加密算法的数学理论的原因，密钥一般都相当长。对诸如 RSA 和 Diffie-Hellman 这样的密码来说，密钥长度通常介于 512 和 2048 位之间。但这并不必定意味着 512 位的 RSA 要比 128 位的 RC4 强度更高，相反其强度可能要弱得多。

不幸的是，我们无法确定会弱多少。对于对称密码来说，蛮力攻击的概念相当清晰：即查找每一个密钥直至找到正确的那个。对于非对称密码来说就不是这样了。拿 RSA 来说，已知的最好攻击就是对 RSA 模进行因数分解。开发出一种新的因数分解算法或仅仅对当前算法的改进就会自动削弱所有现有密钥的强度。

至此，我们对 RSA 的强度有了一个大致地了解。RSADSI 举办有一项因数分解比赛，他们公布大量的“挑战数字”并为对其成功进行分解的人颁发现金奖励。到目前为止，公开知道被分解的最大 RSA 模数为 RSA-155，一个 155 位(512 位)的数字。在 1999 年的 Eurocrypt 上，Adi Shamir 描述了一种称做“(Twinkle)”的 RSA 密钥破译器的设计，它能够对 512 位的数字进行分解[Shamir1999]。这些结果表明使用当前技术，512 位的 RSA 密钥正处于要被攻破的边缘。一般来说，768 或 1024 位的长度是最少可接受的密钥长度。DH 和 DSA 密钥与等长度的 RSA 密钥具有同等的强度。目前的信息表明 1024 位的非对称密钥大约与 80 位的对称密钥强度一致。

1.18 总 结

本章提供了有关通信安全的基本介绍。对这些资料的了解对于理解本书的剩余部分是必要的。

通信安全有三个主要的目标：保密性、消息完整性和端点认证。保密性意味着你所发送的数据要保密。消息完整性意味着当消息在传输过程中遭到篡改时能够检测出来。端点认证的意思是说，你可以确信自己是在与正确的个人进行交谈。

网络是不受信任的。因特网安全的一项基本假定就是攻击者有可能控制网络。为了在这样的环境中保证安全，我们使用密码学技术。

本章涉及了四种基础类型的算法：对称密码、消息摘要、公用密钥密码和数字签名。其他更复杂的体系都是基于这四种原语构建的。每种类型都有许多不同的算法。

通常将公用密钥系统与对称密码结合起来使用。公用密钥密码学提供了无法从对称密码学所获得的能力，但由于效率低下而无法单独使用。

对于非交互式（消息传递）应用来说，我们产生自主式（self-contained）的消息。使用公用密钥加密来保护用来加密消息的对称密钥，并与数字签名函数配合，使用消息摘要来提供消息完整性。

对于交互式应用来说，我们通过握手来确立密钥，然后再使用对称算法。在握手期间使用公用密钥加密对双方进行认证，并实现密钥交换，然后通过对称密钥加密技术使用那些密钥来保护单条数据记录。

2

SSL 介绍

2.1 简介

本章介绍 SSL 与 TLS 以及本书余下内容的背景知识。首先对 SSL 的工作原理进行全局性的概括描述，然后讲述 SSL 及其变种的历史，最终谈及 TLS。此外，我们还会讲到各个版本的实现与使用情况。

接着，简要地概括描述结合超文本传输协议（HyperText Transfer Protocol, HTTP）使用 SSL 的知识，HTTP 是万维网（World Wide Web）基础协议，也是与 SSL 匹配最佳的协议。最后，讲解在 SSL 上部署其他非 Web 协议的问题。

2.2 标准与标准化组织

我们在本书中讨论的大多数协议都是由因特网工程任务组（Internet Engineering Task Force(IETF)）制定的。大家一眼就能够认出 IETF 文档，因为这些文档的名字都具有 RFC ##### 的形式（RFC 2246 是有关 TLS 的文档）。注意，并不是所有的 IETF RFC 都是标准，其中一些只是以存档形式发布的信息资料文档。这使厂商能以稳定和公开的方式描述自己的协议。

IETF 标准系列(Standards Track)文档的制定其实要经过三个阶段，即建议标准(Proposed Standard)、标准草案(Draft Standard) 和标准(Standard)，在此期间可以随时对文档进行改动。然而，文档的制定过程如此漫长，所以任何处于建议或以后阶段的文档均被实现者当作标准来对待，我们基本上也是这样来做的。

IETF 文档具备两种有用的特性，它们是其他标准化组织制定的文档所不具备的。首先，这些文档通常是由实现者或其亲密同盟撰写的，所以通常是针对当时相关问题而提出的。其次，IETF 文档是免费的，大家可以从全球的 Web 或 FTP 站点上下载得到。主站点位于 <http://www.ietf.org/>。

我们要接触的另两类标准分别由美国国家标准协会（ANSI）和国际电信联盟（ITU）制定。ANSI 标准的名称通常具有类似 X9.42（这是 Diffie-Hellman 密钥交换标准）的形式，ITU 标准具有类似 X.509（这是有关证书的标准）形式。知道哪些文档代表什么标准并不重要，

除非你打算购买其中的文档，这两种组织制定的文档都不是免费的。

注：再怎么强调免费文档的重要性都不为过。IETF 协议的实现者不愿意或不能为 IETF 协议所依赖的非免费协议文档付费是相当常见的情况。结果，非免费标准的拷贝或概要私下流传，经常造成这些标准（经常是些非常复杂的标准）的实现往往并不标准。

由于这个以及其他文化上的原因，IETF 的工作人员常常对 ANSI 和 ITU（尤其是对 ITU）报有成见，不过它们的许多文档涵盖了许多重要方面，IETF 文档以引用的形式集成进来，所以了解这些情况是很重要的。

2.3 SSL 概述

安全套接层（Secure Socket Layer，SSL）是一种在两台机器之间提供安全通道的协议。它具有保护传输数据以及识别通信机器的功能。安全通道是透明的，意思就是说它对传输的数据不加变更。客户与服务器之间的数据是经过加密的，一端写入的数据完全是另一端读取的内容。透明性使得几乎所有基于 TCP 的协议稍加改动就可以在 SSL 上运行，非常方便。

SSL 历经多次修订，从开始的版本 1 到 IETF 最终所采纳的传输层安全（Transport Layer Security，TLS）标准。所有 IETF 版本之前的 SSL 版本都是由网景通信公司（Netscape Communications）的工程师们设计的。版本 1 从未经过广泛的部署，所以讨论将从版本 2（SSLv2）开始。附录 B 中包含了有关版本 1 的概要性讨论。

2.4 SSL/TLS 的设计目标

SSLv2 的设计目标

SSL 原先是为 Web 设计的。网景公司的意图是针对其所有的通信安全问题，包括 Web、电子邮件及新闻组通信提供一种单一的解决方案。因为他们当时所面临的问题只是 Web 通信，所以 SSL 就是为了满足 Web 通信的需要而设计的。也就是说 SSL 必须能够与 Web 所使用的主要协议——超文本传输协议（HTTP）[Fielding1999]一起很好的工作。

最初在 1994 年设计 SSLv2 的时候，人们最为关心的 Web 安全问题就是如何在不泄露给实施攻击的第三方的条件下，将信息从客户端传送给服务器。典型的应用案例就是在联机采购时提交信用卡号码。因此，首要的设计目标就是为客户与服务器之间的传输提供保密性。

信用卡应用还引发了第二与第三个设计目标。由于信用卡号的内在价值，所以只将其透漏给你打算与其做交易的人这一点就非常重要，而且最好要是那个可证实其身份的人。这要求顾客（Web 浏览器）能够确信自己是在将信用卡号发送给恰当的经销商（Web 服务器）。另一方面，由于信用卡号码是将账记在某个人头上所需的所有身份信息，所以服务器端没有了解用户是谁的必要。于是第二个设计目标就是：服务器认证。SSL 还可以选择提供客户认证。

因为顾客经常需要与从未打过交道的经销商做交易，所以这种协议应能自动处理这样的情况，因此尽可能少地给用户增添麻烦对这种协议来说就非常重要。对该种设计目标顺理成章。

由于网景公司想以 SSL 作为一种统一的安全解决方案，所以除 HTTP 之外，它对其他协议来说也必须能工作。设计者们注意到大多数流行的因特网协议都是在 TCP 连接上运行的，所以可以使用一种提供安全、透明通道的协议来为所有这些协议提供安全。透明性或许是 SSL 成功的最重要原因，因为它使得只需基于 SSL 运行就能相对简单地为几乎所有的协议提供某种程度的安全。事实确实如此，在引入了 SSL 之后，网景公司不仅着手使用它来保护 HTTP 的传输安全，还保护了基于网络新闻传输协议（Network News Transfer Protocol, NNTP）的新闻组新闻的传输安全。

不幸的是，并不是所有的协议都需要同样的安全特性，所以“单单”在 SSL 上运行协议的结果不管从安全上还是从性能角度考虑常常不如人意。本书的主要目的就是为你提供充足的、设计更快更安全系统的知识，而不只是生硬地使用 SSL。

SSLv3 的设计目标

SSLv2 的广泛流行大体保证了 SSLv2 原先所有的设计目标得以在 SSLv3 中保留下来 [Freier1996]。SSLv3 设计者的首要工作就是修正 SSLv2 中的多处安全问题（请参见附录 B 来了解更多有关这方面的信息）。也就是说，设计一种更加强壮而且更易分析的系统。需要特别说明的是，SSLv3 设计者明显的需求就是设计一种安全磋商多种加密算法的机制。结果，SSLv3 比 SSLv2 支持的算法数量多得多。最终，这些要求使 SSLv3 与 SSLv2 完全不同，只保留了一些基本的协议特色。

2.5 SSL 与 TCP/IP 族

所有版本的 SSL 与 TLS 都采用相同的基本策略：它们在两个通信程序之间提供一条在其间传递任意应用数据的安全通道。从理论上讲，SSL 连接非常类似于“保密的”TCP 连接。从安全套接层这个名字就可以看出，目的就是为了让 SSL 连接像以 TCP 连接起来的套接字那样。

让 SSL 语义模仿 TCP 语义的首要目的就是易于应用程序员的使用。依照这个目标，大多数 SSL 实现都提供了有意模仿最为流行的网络 API、Berkeley sockets 的应用编程接口（application programming interface, API）。图 2.1 描述了调用典型 SSL API（OpenSSL）以及对应 Unix API 的例子。

Sockets API	OpenSSL
int socket(int, int, int)	SSL *SSL_new(SSL_CTX *)
int connect(int, const struct sockaddr *, int)	int SSL_connect(SSL *)
ssize_t write(int, const void *, size_t)	int SSL_write(SSL *, char *, int)
ssize_t read(int, void *, size_t)	int SSL_read(SSL *, char *, int)

图 2.1 Sockets API 与 SSL API 的比较

在理想状况下，从一个应用程序员的角度来看，可以将所有的 socket 调用替换成 SSL 调用而大体上获得所需的安全特性。甚至有可能与提供普通 socket 调用库的安全版本（除此之外与普通库没什么差别）进行链接来使应用获得安全特性。不幸的是，尽管这在某种程度上是可能的，但是 SSL 的语义与 TCP 的语义并不严格匹配，这就会导致一些问题或混淆，正如将在第 8 章所见到的。图 2.2 描述了协议栈中 SSL 的位置，它正好位于应用层的下面，

TCP 的上面。

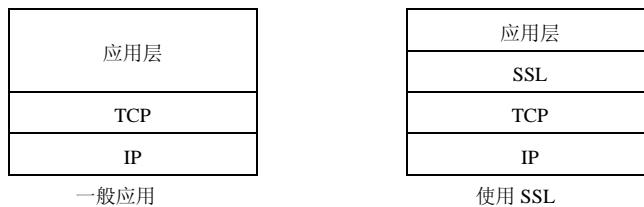


图 2.2 协议栈中 SSL 的位置

SSL 假定其下层的数据包发送机制是可靠的。写入网络的数据将依顺序发送给另一端的程序，不会出现丢包或重复发送的情况。从理论上讲，有许多传输协议都能提供此种服务，但在实际应用中，SSL 几乎只是在 TCP 上运行，它不能在 UDP 或直接在 IP 上运行。

SSL 依赖于可靠传输协议发送数据的特点一直是过去争论的焦点，而且已经存在有两种去除这种依赖性的尝试。总的来说，这种特点需要处理确认和重发超时问题，以便重发丢失的消息。微软的 STLP 和无线应用论坛的 WTLS[WAP1999a]均为意图在数据报传输层如 UDP 上正确工作的 SSL 变种。

2.6 SSL 的历史

图 2.3 描述了各种 SSL 变种的谱系树，最早版本（SSLv2）位于该树的顶端，而最新的版本（WTLS）位于树的底部。原先的 SSLv2 规范于 1994 年 11 月首次公开发表，在此后的不久的 1995 年 3 月就在 Netscape Navigator 1.1 中进行了部署。主要设计由一位网景公司的雇员 Kipp Hickman 完成，公开评论很少。

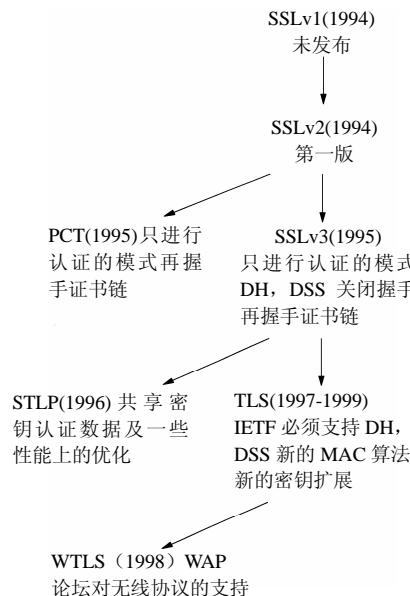


图 2.3 SSL 变种的谱系树

原来的 Netscape Navigator SSLv2 的实现很糟糕。它以当天的时间以及其他几种来源作为其伪随机数发生器的种子。Wagner 和 Goldberg[Goldberg1996]展示了如何在一小时内攻破 Navigator 1.1 的 SSL 连接的方法。

由于在开发 SSL 的厂商投入很少，随着其他厂商着手实现 SSL，为了修正该协议中的问题开发了一些不太兼容的独立变种。其中最重要的就是 1995 年 10 月微软发表的保密通信技术（Private Communications Technology, PCT）[Benaloh1995]。

● PCT

PCT 的作者（Josh Benaloh、Butler Lampson、Daniel Simon、Terence Spies 和 Bennet Yee）有着丰富的密码学经验，他们将自己的知识运用到该协议中。总的来说，PCT 有着比 SSL 更为成熟的安全特性，尽管与 SSL 是向后兼容的。

虽然 SSLv2 具有版本号，但是网景公司控制着版本号空间，而且从没有很好地规定当服务器看到比其所支持的更新的版本号时的行为。因此，一种实现是通过在其密码列表中发送一项特殊的 PCT_SSL_COMPAT 密码来表明它所使用的是 PCT。SSLv3 针对版本升级的问题进行了更好的定义，然而，仍然没有办法标示对具体新特性的支持，也就是说没有好的增添实验性特性的方法。TLS 和 SSLv3 的实现偶尔会用到这项功能。

PCT 完成了三项重大的修改。首先，PCT 包含了一种非加密操作模式，只提供数据认证。其次，限制了密钥扩展变换。由于美国的出口限制，SSLv2 包含了一种出口模式，即加密安全限制在 40 位。不幸的是，SSLv2 使用同一个密钥来完成加密和认证工作，所以认证安全也被限制在 40 位。PCT 允许弱强度加密与高强度认证并存。第三，通过减少所需的往返次数改进了性能。要想更多地了解有关 SSLv2 的内容，请参见附录 B。

● SSLv3

网景公司内部流通有一份流产的“SSLv2.1”，这是第一个作为对 SSLv2 修改的草案。网景公司最终还是决定推倒一切从头再来。他们雇佣了 Paul Kocher，一位知名的安全顾问，与 Allan Freier 和 Phil Karlton 一起开发了一种新版的 SSL，称之为 SSL 版本 3(SSLv3)。SSLv3 是在 1995 年末发布的。

PCT 在一定程度上保留了 SSLv2 的整体风格，它使用同样的规格描述语言，以及一些相同的消息。但是 SSLv3 发明了一种全新的规格描述语言，以及一种全新的记录类型和数据编码。它还处理了一些 PCT 处理的问题，包括增加了只进行认证的模式，以及完全重写了密钥扩展变换。

此外，SSLv3 还增加了一些 PCT 中所没有的新特性，其中包括：多种新的加密算法（数字签名标准（Digital Signature Standard, DSS）、Diffie-Hellman（DH）以及国家安全局的 FORTEZZA 加密令牌），支持防止对数据流进行截断攻击的关闭握手。对于 SSLv2 来说，如果攻击者伪造 TCP 连接关闭，则要想知道所传输的数据比实际的少是不可能的。SSLv3 的关闭握手可以检测出这种攻击。与 PCT 一样，SSLv3 与 SSLv2 向后兼容。

SSL 的向后兼容性一向颇受限制。兼容性通常所表达的含义就是，即便使用不同的版本，只要 SSL 客户与服务器端使用的公共版本号一样，它们也能可靠地进行交互。并不是说前一版本的消息在更新的版本中总是有效，实际上，自打开始就不是这种情况。

TLS

1996 年 5 月，因特网工程任务组（IETF）特许传输层安全（TLS）工作组对一种类似 SSL 的协议进行标准化。尽管官方工作组负责人另有其说，但是这件事还是被广泛理解为一种调和微软与网景公司方案的尝试。尽管也起草了其他一些方案，但实质上没有获得任何支持。这项计划在 1996 年末完成。

微软制作了一份称做安全传输层协议的提案（Submission），它是对 SSLv3 的修改，增加了一些微软认为关键的特性。其中主要的内容是对数据报（如 UDP）的支持，以及使用共享密钥支持客户端认证。尽管在加密的 SSL 连接上传送共享密码是完全可能的，但是这样意味着对于出口情况来说，只能用 40 位的密钥对密码进行加密。STLP 集成了一种强度高得多的基于共享密码的客户端认证，并整合了某些性能上的适当改进，同时还改进了密码的可扩展性，以及允许 TCP “客户端” 成为 STLP “服务器”。

1996 年末，主要的参与者（以及几个次要的参与者）在 Palo Alto 的一次由密码学家 Bruce Schneier 主持的小组会议中碰面，IETF 工作组一年只在 IETF 会议上碰三次面，但是当工作特别紧急的时候，有时也会进行非正式会面。这次就是这样的一种会面。

很快结果就出来了，除了几处显而易见的（非常微小的）不足之处外，几乎没有人支持对 SSLv3 进行任何改动。特别是存在大量的反对意见——大部分都是以向后兼容的名义，反对微软所提议的这种改动。主要公开讨论的问题似乎是新协议的名称是什么，因为使用 SSL 或 PCT 就太像在说微软或是网景公司赢得了胜利。然而不管起什么名字，新协议也只是对 SSLv3 稍加修整的结果。

最终，经过几次工作组会面之后，TLS 工作组决定将协议命名为 TLS，所有各方都不怎么赞同使用这个名字。工作组以安全的名义对文档进行了微小的改动，从而使得密钥扩展和消息认证计算与 SSLv3 完全不兼容，破坏了大部分的向后兼容性。

至此，TLS 中最有争议的改变就是决定要求实现支持 DH、DSS 和三重 DES（3DES）。由于两种原因，这样做会出现问题。首先，网景公司只实现了用 RSA 来完成认证和密钥交换。由于网景公司在业界占据了浏览器中的大多数，与网景公司的 SSL 实现进行互操作实际上是至少要满足的标准要求。由于与老式的客户端或服务器进行互操作是市场所需要的，这就意味着实现者必须同时实现 RSA 和 DH/DSS。

然而，当时更不方便的是需要增加 3DES。那时，美国出口法规总体上禁止出口强度超过 40 位的密码技术。因此，出口包含 3DES 的 TLS 实现是不合法的。结果就造成了要么实现是与 TLS 兼容的，要么是可出口的，但二者不可得兼。

要想理解为什么会进行这些有争议的改动，就必须理解 IETF 是如何工作的。IETF 组织成许多面向特定任务（如，开发 TLS）工作的工作组。每个工作组是一个或多个整体领域的一部分。当前存在八个领域：应用、通用标准、因特网、操作和管理、路由、安全、传输和通用服务。每个领域有一个或多个领域指导，他们负责对那个领域执行监管。由领域指导共同组成 IESG（因特网工程指导组），在任何文档成为因特网档案文档，即请求评议文档（RFC）之前，必须经过指导组的批准。

1995 年 4 月，在 Massachusetts 州的 Danvers 会议上，IESG 采纳了一项称做 Danvers Doctrine 的条款，声明 IETF 要设计体现良好的工程原则的协议，而不管现在的出口状况如何。当时，这项动议暗指至少要对 DES 提供支持，而过了一段时间就成了 3DES。此外，可

能的话, IETF 还将长期偏向于采用不受干扰(即免费的)的算法。当 Merkle-Hellman 专利(涵盖了所有的公用密钥加密)于 1998 年到期时, RSA 仍然受专利保护, IESG 开始敦促工作组采用免费的公用密钥算法。最后,许多 IETF 成员都感觉到让协议拥有一组强制性的选项以确保任何两种实现均能进行交互是一种很好的行事。

当 TLS 工作组在 1997 年末完成工作的时候,将文档发送给 IESG,然后 IESG 将其返还并指示增加一种强制性的加密算法——用于认证的 DSS,用于密钥磋商的 DH,以及用于加密的 3DES,于是就解决了上面所提到的三个问题。邮件讨论组随即出现了大量的讨论,特别是网景公司抵制一般意义上的强制性算法,尤其是 3DES。经过了 IESG 与工作组很长一段时间的僵局之后,勉强达成一致,对文档进行了合适的改变之后再次发回。

不幸的是,与此同时,出现了另一个障碍。执行 X.509 证书标准化工作的 IETF 公用密钥信息基础设施(Public Key Infrastructure, PKIX)工作组陷入停顿。TLS 依赖于证书,因此也就依赖于 PKIX,而 IETF 规定禁止协议推进的进度超出它所依赖的协议。PKIX 的最终完成比预期花费了相当长的时间,增添了额外的延迟。1999 年 1 月(两年之后)TLS 最终以 RFC 2246[Dierks1999]的形式发表。在编写这本书的时候,其部署并不显著。Internet Explorer 支持 TLS,但是 Netscape Navigator 并不支持,且大多数工具箱都不支持 TLS。

2.7 用于 Web 的 SSL

SSL 的首要用途就是保护使用 HTTP 的 Web 通信。将在第 9 章针对这一话题进行广泛地讨论。这里,只对一种典型的应用提供粗略地介绍,以解释在设计 SSL 时所做出的一些选择。

过程很简单。当在 HTTP 中建立了 TCP 连接后,客户端先发送一个请求,服务器随即回应一个文档。而在使用 SSL 的时候,客户端先创建一个 TCP 连接,并在其上建立一条 SSL 通道,然后再在 SSL 通道上发送同样的请求,而服务器则以相同的方式沿 SSL 连接予以响应。

对普通的 HTTP 服务器来说,SSL 握手就像是垃圾信息,因为并不是所有的服务器都是支持 SSL,所以为了使该过程能够正确的工作,客户端需要某种了解服务器已准备好接受 SSL 连接的方法。使用以 https 而不是 http 开头的 Web 地址(技术上被称做统一资源定位符(URL))来指示应当使用 SSL。例如:

```
https://secure.example.com
```

因此,这种在 SSL 上运行 HTTP 的组合常常被称做 HTTPS。

如果一切顺利,服务器与客户端能够进行通信的话,通常会以某种用户界面提供用户正在使用安全特性的反馈。在老版本的 Netscape 中,工具条下方会显示一条蓝色的光棒并会在浏览器的左下角出现一把钥匙。在 Internet Explorer 中,在屏幕的右下角会出现一把锁,所有这些都表示当前页面是用 SSL 来获取的。

尽管几乎所有现代商业服务器都实现了 SSL,但与普通服务器相比而言,网景公司在刚开始时就让其服务器(Netscape Commerce Server)具有了 SSL 的能力,并征收相当数目的额外收费。此外,由于专利的原因,美国的免费服务器都不支持使用 RSA 的 SSL(但是由于 RSA 是事实上的标准,这也就意味着它们根本就不支持 SSL)。因此即便有一种实现了

SSL 的服务器，你还是选择不去激活其安全特性。与纯粹的 HTTP 比起来，HTTPS 会给服务器造成高得多的负载，而且获取所需要的证书也相当不方便。此外，该证书也不便宜。在编写这本书的时候，最大的 CA，Verisign 一份证书要征收超过 300 美元。

2000 年 1 月，Netcraft 的调查[Netcraft2000]发现大约有 1,500,000 万台服务器运行 SSL，但是其中只有 60,000 台由众所周知的第三方进行认证。其余的都是自签名或由私有 CA 颁发的。客户端需要进行特殊的配置才能连接上这些服务器。

2.8 在 SSL 上构建一切

由于许多协议都在 TCP 上运行，而 SSL 连接与 TCP 连接非常相似，所以通过在 SSL 上附加现有协议来保证其安全是一项非常吸引人的设计决策。除了 SSL 之上的 HTTP 和 NNTP（常常被称做 SNEWS）以外，很快就出现了许多使用 SSL 来保护主要因特网协议的提议，其中包括 SMTP[Hoffman1999a]、Telnet[Boe1999]和 FTP[Ford-Hutchinson2000]。许多厂商还使用 SSL 来保护他们的专有协议。

为了接纳不使用 SSL 的客户端的连接，服务器通常必须做好接受受保护的和未受保护版本的应用协议。上一段中提到的所有协议都使用两种基本策略中的一种：即设置不同的端口或采用磋商升级技术（*upward negotiation*）。这两种策略都是既有优势又有不足，我们将在第 7 章讨论相关的内容。

在分设端口策略中，协议设计者简单地为协议分配一个不同的众所周知的端口，而服务器实现让服务器同时在原来的端口和新的安全端口上进行监听。任何到达安全端口的连接都会在协议通信开始之前自动执行 SSL 磋商。HTTPS 就使用这种策略，有关内容将放在第 9 章讨论。图 2.4 列举了一些为常用协议分配的端口号。

名称	端口	用途
ftps-data	989/tcp	ftp protocol, data, over TLS/SSL
ftps-data	989/udp	ftp protocol, data, over TLS/SSL
ftps	990/tcp	ftp protocol, control, over TLS/SSL
ftps	990/udp	ftp protocol, control, over TLS/SSL
nntps	563/tcp	nntp protocol over TLS/SSL
nntps	563/udp	uutp protocol over TLS/SSL

图 2.4 为 SSL 化的协议分配的端口号

当使用升级磋商策略的时候，协议设计者通过修改应用协议来支持一种用于表示一方想要升级为 SSL 的消息。如果另一方同意的话，就会开始一次 SSL 握手。一旦握手完成，应用消息就恢复在新建的 SSL 通道上的传送。基于 TLS 的 SMTP 就使用这种策略，有关内容将在第 10 章讨论。

2.9 获得 SSL

就目前而言，获得 SSL 实现最简便的方式就是通过下载得到。位于 <http://www>.

openssl.org/的 OpenSSL 提供了一种高质量的、免费的、实现 SSLv2、SSLv3 和 TLS 的源代码，并提供了世界范围内的下载。OpenSSL 基于 Eric Young 的 SSLeay 库，该库于 1995 年首次发表。代码大体上是用 ANSIC 写成的，并实现了广泛的移植。OpenSSL 授权在 BSD 风格的许可证下使用，因此可免费用于商业或非商业用途。过去，由于专利的原因，在美国国内不能在 RSA 方式下使用 OpenSSL。然而，因为 RSA 专利已于 2000 年 9 月到期，所以现在在美国使用也是合法的。

下列厂商销售 C/C++ SSL/TLS 工具箱。

- Certicom (<http://www.certicom.com>)
- Netscape Communications (<http://home.netscape.com/>)
- RSA Security (<http://www.rsasecurity.com/>)
- SPYRUS/Terisa Systems (<http://www.spyrus.com/>)

网景公司第一个实现了 SSL。RSA 雇佣了 Eric Young，而他们的工具箱就是基于 SSLeay 的。而 Consensus 和 SPYRUS/Terisa 的实现均是独立开发的。

全揭密：作者在 Terisa 工作了三年，是 Terisa SSL 产品的主要开发人员。

与之类似，最容易获得 Java 实现的方法也是下载。大家可从 <http://www.rfm.com/puretls> 获得作者免费的 Java SSLv3/TLS 实现，PureTLS。下列厂商也拥有 Java 实现。

- Baltimore (<http://www.baltimore.com/>)
- Certicom (<http://www.certicom.com/>)
- Phaos Technology Corporation (<http://www.phaos.com/>)
- Sun (<http://www.javasoft.com/>)

在编写这本书的时候，Sun 的产品可特许免费用于“非商业”用途，但只可获得二进制形式。

Netscape 与 Internet Explorer 都实现了 SSL，网景公司的 Web 服务器和微软公司的 IIS 也都实现了 SSL。微软还允许程序员调用它们的 SSL 实现（SChannel）。但那只是 Windows 程序员的选择。存在几种免费的、针对免费 Apache HTTP 服务器的 SSL 源代码实现。

- ApacheSSL (<http://www.apachessl.com>)
- mod_ssl (<http://www.modssl.org>)

存在几种基于 Apache 的商业实现。

- Raven (<http://www.covalent.net>)
- Stronghold (<http://www.c2.net>)

实现内容

当前几乎每种可获得的 SSL 实现都支持带有 RSA 的 SSLv3。而明显的例外就是那些已经安装的 Web 服务器（通常是些老版本的 Netscape Commerce Server），它们只支持 SSLv2。尽管一些新近的实现省略了对 SSLv2 的支持，但是为了向后兼容性，大多数实现都支持 SSLv2。大多数工具箱都支持 TLS 协议所做的修改，但并不都支持 DSS 和 DH。微软的实现也支持 PCT 和 TLS，而且与 Windows2000 一起发行的 IIS 和 IE 版本还支持 DSS 和 DH。在写这本书的时候，Netscape 的产品仍然只支持 SSLv3 而不对 DSS 和 DH 提供支持。

为了最大限度的兼容性，对 SSLv2 和 SSLv3 提供支持是重要的。实际上不存在只支持



TLS 的实现，所以对 TLS 的支持不是必需的，尽管能支持它还是最好，因为这是一项 IETF 标准。

2.10 总 结

本章对 SSL 及其在网络世界中的地位进行了概括性的描述，提供了我们将要在本书余下部分展开讨论的上下文信息。SSL 和 TLS 在 TCP 上提供一种通用的通道安全机制。任何可以在 TCP 上承载的协议都能够使用 SSL 或 TLS 加以保护。SSL 起先是由 Netscape 设计的。SSLv2 和 SSLv3 基本上是仅由 Netscape 把持的产品，但 TLS 是 IETF 工作组从 SSLv3 着手推出的标准。SSL 提供服务器认证、加密和消息完整性。可选择支持加密的客户端认证。通常把 SSL API 设计成模仿普通网络 API 的样子。在 C 语言中，SSL API 通常模仿 Berkeley 套接字，而在 Java 中模仿 Java 套接字抽象，从而使得可以非常方便地将非安全应用转换为 SSL 使能的应用。SSL 广泛应用于 HTTP 数据传输。然而，现在有越来越多的协议使用 SSL/TLS 加以保护，TLS 正在缓慢地取代 SSL。存在大量可用的 SSL 与 TLS 实现，用 C 语言和 Java 语言编写的免费实现，以及多种商业实现。

3

SSL 基础

3.1 介绍

本章提供了对最常见情况，即使用 RSA 只对服务器进行认证的 SSL 操作的具体描述。把本章的标题取做 SSL 基础，意思是说这里所讲的是 SSL 的基本模式，而不是深入讨论协议的内容（后面再进行讨论）。本章只涉及 SSL 最简单的模式。下一章则要讨论几种更复杂的模式。

本章组织成两个主要部分。第一部分对 SSL 的工作原理以及各个组成部分是如何拼接的进行概括性地描述。将会泛泛地讲述每种协议消息及其一般的用途。第二部分详细剖析一个 SSL 连接范例。我们使用一种自动工具来产生会话跟踪信息，然后具体检查每一种协议消息，看看各个字段是如何相互配合，共同完成协议工作的。如果你将其当作实现指南来读或是想具体理解其中的机理，就应当阅读这两部分。如果你主要的兴趣只是如何使用 SSL，那么跳过第二部分也无妨。

3.2 SSL 概述

SSL 的基本设计看起来与在第一章所设计的玩具安全协议相当类似。连接分为两个阶段，即握手和数据传输阶段。握手阶段对服务器进行认证并确立用于保护数据传输的加密密钥。必须在传输任何应用数据之前完成握手。一旦握手完成，数据就被分成一系列经过保护的记录进行传输。

3.3 握手

SSL 握手有三个目的。第一，客户端与服务器需要就一组用于保护数据的算法达成一致。第二，它们需要确立一组由那些算法所使用的加密密钥。第三，握手还可以选择对客户端进行认证，这一点将在第 4 章看到。我们先不讨论具体的协议消息，从整体上讨论握手过程。然后再讲述如何将这一过程分解为具体的消息。

整个过程工作如下（请参见图 3.1 的图示）：

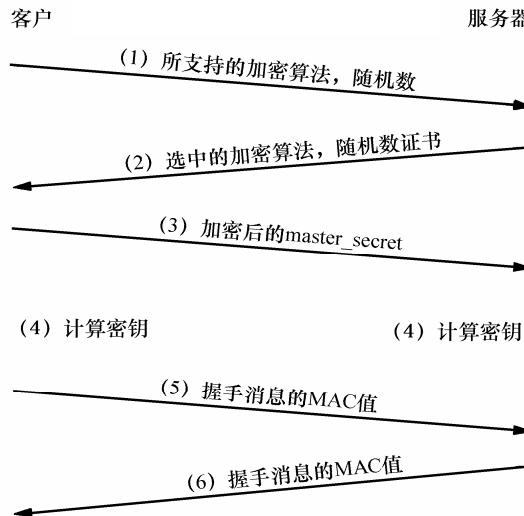


图 3.1 SSL 握手概述

(1) 客户端将它所支持的算法列表连同一个密钥产生过程用作输入的随机数发送给服务器。

(2) 服务器根据从列表的内容中选择一种加密算法，并将其连同一份包含服务器公用密钥的证书发回给客户端。该证书还包含了用于认证目的的服务器标识，服务器同时还提供了一个作为密钥产生过程部分输入的随机数。

(3) 客户端对服务器的证书进行验证，并抽取服务器的公用密钥。然后，再产生一个称做 pre_master_secret 的随机密码串，并使用服务器的公用密钥对其进行加密。最后，客户端将加密后的信息发送给服务器。

(4) 客户端与服务器端根据 pre_master_secret 以及客户端与服务器的随机数值独立计算出加密和 MAC 密钥。

(5) 客户端将所有握手消息的 MAC 值发送给服务器。

(6) 服务器将所有握手消息的 MAC 值发给客户端。

那么，该过程达到了怎样的效果呢？还记得我们的两个目标是什么吗？第一，就一组算法达成一致。第二，确立一组加密密钥。第一和第二步实现了第一个目标。客户端告诉服务器它所支持的算法，而服务器选择其中的一种算法。当客户端收到了服务器在第二步所发的消息时，它也会知道这种算法，所以双方现在就都知道要使用什么算法了。

第二个目标，确立一组加密密钥是通过第二和第三步来实现的。在第 2 步服务器向客户端提供其证书，这样就可以允许客户端给服务器传送密码。经过第 3 步后，客户端与服务器端就都知道了 pre_master_secret。客户端知道 pre_master_secret 是因为这是它产生的，而服务器则是通过解密而得到 pre_master_secret 的。

注意，第 3 步是握手过程中的关键一步。所有要被保护的数据都依赖于 pre_master_secret 的安全。原理非常简单：客户端使用服务器的公用密钥（从证书中抽取的）来加密共享密钥，

而服务器使用其私用密钥对共享密钥进行解密。握手的剩余步骤主要用于确保这种交换过程的安全进行。然后在第4步，客户端与服务器分别使用相同的密钥导出函数（key derivation function, KDF）（请参见3.11节）来产生master_secret，最后再次通过KDF使用master_secret来产生加密密钥。

第5与第6步用以防止握手本身遭受篡改。设想一个攻击者想要控制客户端与服务器所使用的算法。客户端提供多种算法的情况相当常见，某些强度弱而某些强度强，以便能够与仅支持弱强度算法的服务器进行通信。攻击者可以删除客户端在第1步所提供的所有高强度算法，于是就迫使服务器选择一种弱强度的算法。第5步与第6步的MAC交换就能阻止这种攻击，因为客户端的MAC是根据原始消息计算得出的，而服务器的MAC是根据攻击者修改过的消息计算得出的，这样经过检查就会发现不匹配。由于客户端与服务器所提供的随机数为密钥产生过程的输入，所以握手不会受到重放攻击的影响。这些消息是首个在新的加密算法与密钥下加密的消息。

因此，在此过程结束时，客户端与服务器已就使用的加密算法达成一致，并拥有了一组与那些算法一起使用的密钥。更重要的是，它们可以确信攻击者没有干扰握手过程，所以磋商过程反映了双方的真实意图。

握手消息

刚才所描述的每一步都需要通过一条或多条握手消息来实现。在此先简要地描述哪些消息与哪几步相对应，然后再在这一章的后面详细描述每条消息的内容。图3.2描述了各条消息。

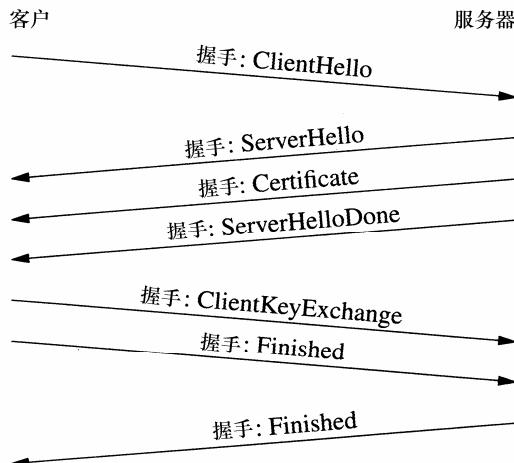


图3.2 SSL握手消息

第1步对应一条单一的握手消息，ClientHello。

第2步对应一系列SSL握手消息，服务器发送的第一条件消息为ServerHello，其中包含了它所选择的算法，接着再在Certificate消息中发送其证书。最后，服务器发送ServerHelloDone消息以表示这一握手阶段的完成。需要ServerHelloDone的原因是一些更为复杂的握手变种还要在Certificate之后发送其他一些消息。当客户端接收到ServerHelloDone消息时，它就知道不会再有其他类似的消息过来了，于是就可以继续它这一方的握手。

第 3 步对应 ClientKeyExchange 消息。

第 5 与第 6 步对应 Finished 消息。该消息是第一条使用刚刚磋商过的算法加以保护的消息。为了防止握手过程遭到篡改，该消息的内容是前一阶段所有握手消息的 MAC 值。然而，由于 Finished 消息是以磋商好的算法加以保护的，所以也要与新磋商的 MAC 密钥一起计算消息本身的 MAC 值。

注意，图示省略了两条 ChangeCipherSpec 消息。以后再来讲解这两条消息，在这一阶段的讨论中它们并不重要。

3.4 SSL 记录协议

到目前为止我们设法做到的就是对服务器进行认证并共享一些密钥资料。记住，将 SSL 放在首位的原因就是能够交换经过加密和认证的数据。握手的目的就是建立起使发送和接收受保护数据成为可能所需要的共享状态。在 SSL 中，实际的数据传输是使用 SSL 记录协议来实现的。

SSL 记录协议是通过将数据流分割成一系列的片段并加以传输来工作的，其中对每个片段单独进行保护和传输。在接收方，对每条记录单独进行解密和验证。这种方案使得数据一经准备好就可以从连接的一端传送到另一端，并在接收到的即刻加以处理。

在传输片段之前，必须防止其遭到攻击。可以通过计算数据的 MAC 来提供完整性保护。MAC 与片段一起进行传输，并由接收实现加以验证。将 MAC 附加到片段的尾部，并对数据与 MAC 整合在一起的内容进行加密，以形成经过加密的负载（payload）。最后给负载装上头信息。头信息与经过加密的负载的连结称作记录（record），记录就是实际传输的内容。图 3.3 描述了传输过程。

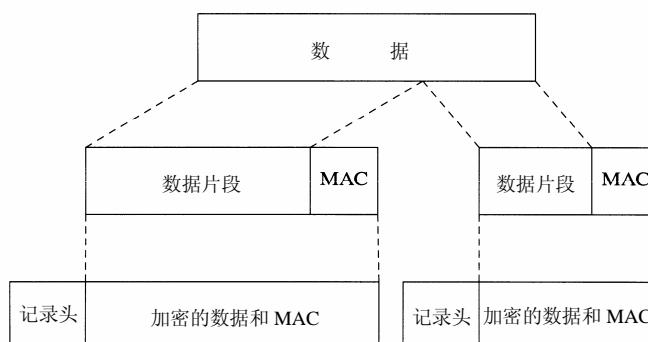


图 3.3 SSL 数据的分段与保护

记录头信息

记录头信息的工作就是为接收实现提供（receiving implementation）对记录进行解释所必需的信息。在实际应用中，它是指三种信息：内容类型、长度和 SSL 版本。长度字段可以让接收方知道他要从线路上读取多少字节才能对消息进行处理，版本号只是一项确保每一方使用所磋商的版本的冗余性检查。

内容类型字段标识消息类型。正如我们在第1章所讨论的，若能在同一条受保护的通道上传输各种类型的信息则是非常方便的。尤其是在我们想要发送受保护的、表示错误与连接关闭的消息时。内容类型字段使得实现能够将这种管理信息与传送给高层应用的数据区分开来。

内容类型

SSL支持四种内容类型:application_data、alert、handshake和change_cipher_spec。使用SSL的软件发送和接收的所有数据都是以application_data类型来发送的。其他三种内容类型用于对通信进行管理，如完成握手和报告错误等。

内容类型alert主要用于报告各种类型的错误。大多数alert(警示)用于报告握手中出现的问题，但是也有一些指示在对记录试图进行解密或认证时发生的错误。alert消息的其他用途是指示连接将要关闭，这对于阻止1.8节所讨论的截断攻击是必需的。

内容类型handshake(不足为奇)用于承载握手消息。即便是最初形成连接的握手消息也是通过记录层以handshake类型的记录来承载的。由于加密密钥还未确立，这些初始的消息并未经过加密或认证，但是其他处理过程是一样的。有可能在现有的连接上初始化一次新的握手(请参见第4.5节)，在那种情况下，新的握手记录就像其他的数据一样，要经过加密和认证。

change_cipher_spec消息有着特殊的用途，它表示记录加密及认证的改变。一旦握手商定了一组新的密钥，就发送change_cipher_spec来指示此刻将启用新的密钥。3.10节将会更详细地讨论这方面的内容。

3.5 各种消息协同工作

正如我们所看到的，SSL是一种分层协议，它由一个记录层以及记录层上承载的不同消息类型组成。而该记录层又会由某种可靠的传输协议如TCP来承载。图3.4描述了该协议的结构。

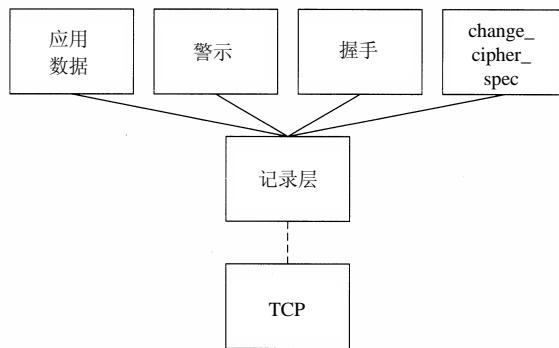


图3.4 SSL协议结构

本章余下部分详细查看了一次具体的SSL连接，它讲述了与本章第一部分类似的内容，但是在更深的层次上进行描述的。由于这里所讲的主要是特定握手消息的内容，因此，如

果你对实现 SSL、分析协议或设计 SSL 变种感兴趣的话，就应该接着读下去。否则直接跳到讲述其他握手模式的第 4 章也无妨。首先，我们讨论这个连接的总体特性，然后再对单条消息进行具体地讨论。

跟前面一样，我们先讲解握手过程，讲述如何通过加密套件方式进行算法的磋商，以及如何使用服务器密钥来为连接确立用于创建加密密钥的共享 `pre_master_secret`。然后再讲述 SSL 记录协议用以保护通信数据的加密变换。

3.6 一次真实的连接

图 3.5 展示了一张我们将要研究的过程图，与此类图示一样，以从左到右的方向来表示从客户端（speedy）发往服务器（romeo）的消息，以从右到左的方向来表示从服务器发往客户端的消息。

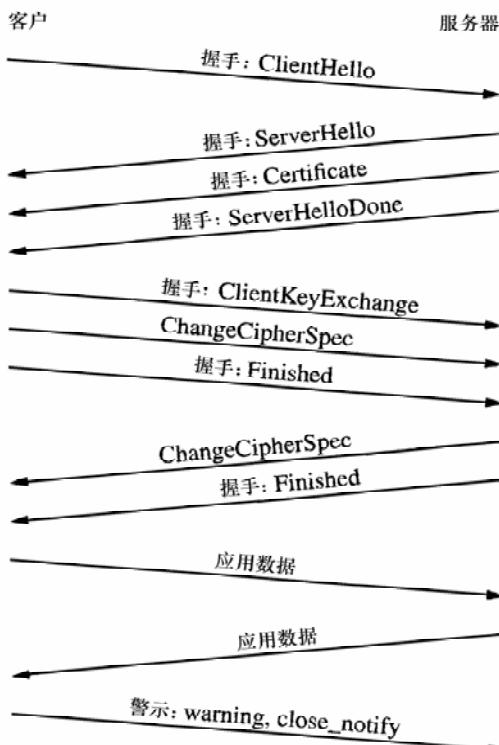


图 3.5 SSL 消息的分时序列

发送的第一条消息为 ClientHello，其中包含了客户端所推荐的加密参数，其中包括它准备使用的加密算法。此外，还包括一个在密钥产生过程中使用的随机值。服务器以三条消息进行响应：首先发送选择加密与压缩算法的 ServerHello，这条消息包含一个从服务器过来的随机值。

SSL 包含压缩的原因就是加密数据对于任何实用目的来说都是随机的，因此也就是无法压缩的，所以在大多数 modem 中，链路层压缩对于 SSL 数据来说不起任何作用。如果想在

传输前对数据进行压缩的话，就必须放在加密之前完成。不幸的是，出于对知识产权的考虑，不管是 SSLv3 还是 TLS 都没有定义任何压缩算法。因此，SSL 几乎从不使用压缩。但有例外，OpenSSL 的确支持压缩，所以如果双方都是基于 OpenSSL 并且进行恰当的配置，那么就能够使用压缩。

接下来，服务器发送 Certificate 消息，其中包含服务器的公用密钥——这里是 RSA 密钥。最后，服务器发送表示握手阶段不再有任何消息的 ServerHelloDone。

客户端发送一条 ClientKeyExchange 消息，其中包含了一个随机产生的用服务器的 RSA 密钥加密的密钥。这条消息后面跟有一条指示客户端在此之后发送的所有消息都将使用刚刚商定的密码进行加密的 ChangeCipherSpec 消息。Finished 消息包含了对整个连接过程的校验，这样服务器就能够判断要使用的加密算法是否是安全商定的。

一旦服务器接收到了客户端的 Finished 消息，它就会发送自己的 Change_Cipher_Spec 和 Finished 消息，于是连接就准备好进行应用数据的传输了。

SSLv3 规范允许一方在发送了 Finished 消息后就开始发送应用消息。然而，如果连接遭受攻击的话，这样做就会使客户端的数据产生安全隐患。如果攻击者对握手进行了修改从而商定使用一种强度较弱的加密套件的话，那么客户端在收到服务器的 Finished 消息之前就不能检测到这种攻击。为了改变这种状况，在 TLS 中，客户端必须等到其收到服务器的 Finished 消息之后，才可以开始发送数据。注意，如果服务器选择的是客户端加密算法中最为安全的一种而不是企图去检测这种攻击的话，这就不会成为一种安全问题。最安全的方法还是等待服务器另一端的 Finished 消息。

接下去的两条消息分别是客户端和服务器发送的应用数据。在这个例子中，这些是在客户和服务器程序中键入数据的结果。但在实际的传输事务中，是应用协议数据发送的时间。

最后，客户端关闭连接，不过它会先发送一条 close_notify alter 来表示连接即将关闭。随后是一条 TCP FIN（这里没有显示出来），服务器使用它自己的 TCP FIN 加以响应，FIN 表示主机的 TCP 实现已经关闭了它那一端的连接，并且不会再发送任何数据。

注意，服务器并没有发送 close_notify。SSL 规范在是否需要 close_notify 这一点上有些不清不楚，但是在理想的状况下，服务器与客户端均需要发送 close_notify。

3.7 其他的连接细节

为了理解我们将要察看的协议消息，要是有一种能够将各种消息拆分开来显示的软件就会非常方便。我们使用 ssldump 来实现这种目的。

图 3.6 显示了一次简单的 SSL 连接中所有消息的 ssldump 跟踪信息。稍后会详细讲述每条消息的内容，但是这里先给出整个连接的跟踪信息，让你有种感性的认识。

由于这是本书展示的第一份 ssldump 跟踪信息，所以抽点儿时间对输出作以讲解。第一行简单指示 SSL 会话将要使用的下层 TCP 连接的建立。这里是测试网络上两台机器 romeo 与 speedy 之间的连接。首先打印的是初始化 TCP 连接（即调用 connect() 的主机）的主机名（romeo）。接着再打印接收 TCP 连接的主机名（speedy）（即调用 accept() 的主机）。在普通的 SSL 操作中，初始化连接的主机扮演客户端的角色，而接受连接的主机扮演服务器的角色。



色。

每个标号的行代表一条 SSL 记录。因此，这个连接由 12 条不同的记录组成。第一个数字为序列中进行处理的记录号，其后跟有两个时间戳记，第一个是自从 TCP 连接建立以来的时间，第二个是从上一条记录以来的时间。这两种时间均以秒来表示，精确到十进制数字小数点后四位。下一个是表示记录发送方向的指示：C>S 表示从客户端到服务器（这里就是从 `romeo` 到 `speedy`），S>C 表示从服务器到客户端（从 `speedy` 到 `romeo`）。接下来是记录类型，它们与前一节描述的层次之一相对应。

```

New TCP connection: speedy(3266) <-> romeo(4433)
1 0.0456 (0.0456) C>S Handshake
    ClientHello
        Version 3.1
        cipher suites
            TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
            TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
            TLS_RSA_WITH_3DES_EDE_CBC_SHA
            TLS_RSA_WITH_IDEA_CBC_SHA
            TLS_RSA_WITH_RC4_128_SHA
            TLS_RSA_WITH_RC4_128_MD5
            TLS_DHE_RSA_WITH_DES_CBC_SHA
            TLS_DHE_DSS_WITH_DES_CBC_SHA
            TLS_RSA_WITH_DES_CBC_SHA
            TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
            TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
            TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
            TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
            TLS_RSA_EXPORT_WITH_RC4_40_MD5
        compression methods
            NULL
2 0.0461 (0.0004) S>C Handshake
    ServerHello
        session_id[32]=
            74 6d 09 76 1d 2a c9 02 4a a1 a3 4e
            27 5c 18 63 8a d9 4a 59 f9 c3 14 a5
            c4 b3 a4 f6 61 ef f5 cd
        cipherSuite          TLS_RSA_WITH_DES_CBC_SHA
        compressionMethod   NULL
3 0.0461 (0.0000) S>C Handshake
    Certificate
4 0.0461 (0.0000) S>C Handshake
    ServerHelloDone
5 0.2766 (0.2304) C>S Handshake
    ClientKeyExchange
        EncryptedPreMasterSecret[64]=
            17 4d 00 32 bc b2 af 95 09 0a 45 24
            97 d8 34 dc 73 20 4d 00 91 a5 0d ed
            c3 f0 b4 f5 32 6f 13 cc ea 41 00 5e
            bb 05 f1 b7 e2 c4 fa 1b 40 c1 2a f3

```

```
00 20 83 43 2d 8c 2a 53 3b 33 cc 5f
0b be db a2
6 0.2766 (0.0000) C>S ChangeCipherSpec
7 0.2766 (0.0000) C>S Handshake
    Finished
8 0.2810 (0.0044) S>C ChangeCipherSpec
9 0.2810 (0.0000) S>C Handshake
    Finished
10 1.0560 (0.7749) C>S application_data
11 6.3681 (5.3121) S>C application_data
12 7.3495 (0.9813) C>S Alert
    level      warning
    value      close_notify
Client FIN
Server FIN
```

图 3.6 一次简单的 SSL 连接

对于某些记录类型 ssldump 会进一步解码，正如接下来缩进的行所显示的那样。特别是，如果记录类型为握手消息，那么 ssldump 就要完成相当数量的解码工作。下一行是握手类型，而接续的行包含了对一些相关字段的解码。究竟要反应出多少信息是由命令行标志来决定的，这里的跟踪信息是使用只显示适当数量细节的设置来产生的。这个会话的目的是描述消息流程，因此在跟踪信息中省略了一些字段。我们将在本章的后面具体对每种消息进行讨论，并讲解每种消息中的所有字段。

例如，第一条记录是握手消息。下一行是一条从 romeo 到 speedy 的 ClientHello 消息。它提供了多组加密套件，范围从使用 3DES 的临时(ephemeral)Diffie-Hellman 到使用 RC4-40 的 RSA，但只有一种压缩方法 NULL。

因为编写 ssldump 部分是为了产生本书中的跟踪信息，所以这些跟踪信息大体上是程序的直接输出并偶尔以斜体增加一些注释。这里所显示的消息跟踪是使用 OpenSSL 的 s_client 和 s_server 程序来产生的，该程序是简单的命令行形式的客户与服务器程序，而加密数据以定宽斜体显示。

3.8 SSL 规范语言

以规范语言的形式来描述协议是一种相当常见的做法，这种语言描述协议消息的各个字段以及在线路上的传递方式。使用规范语言的目的是为了提供一种没有歧义的、精简的协议描述。此类语言的一个例子就是 ASN.1。

TLS 与 SSLv3 都是使用同一种规范语言来描述它们各自的消息的。尽管其语法与 C 语言的类型定义有些粗略的相似，但是这种语言是为 SSLv3 发明的，它是规范中惟一的定义语言。

TLS 规范将语法描述成“随意定义的”，尽管规范中的大部分都显而易见，但是往往未经介绍就多次引入新的结构。曾经有一次，作者企图为这种语言编写一种语法驱动的解析器（使用 YACC），目的是为了机械地生成解码器，但最终由于遇到问题而放弃了这个项目。

● 基本类型

规范使用五种基本类型，`opaque`、`uint8`、`uint16`、`uint24` 和 `uint32`，分别对应无符号的 8-、16-、24- 和 32-位整数，并在线路上以 1、2、3 或 4 字节序列加以表示。所有的数字都以“网络字节顺序”——高位字节在前表示——所以用十六进制字节 `00 00 00 01` 表示 `uint32` 的数字 1。结构/*和*/与 C 语言一样都是用来包裹注释的。

● 向量

向量（vector）就是具有给定类型的元素序列。向量有两种类型，定长或变长。定长向量以 `[]` 来表示，而使用 `<>` 表示变长向量。需要指出的重要一点是，所有的长度均是以字节而不是元素个数为单位的。因此声明 `uint16 foo[4]` 就是指两个 16 位的整数，而不是四个。这样就可以允许解码器具有分层的结构，它可以将结构当做不透明的字符串（因为它知道它们的长度）看待，并将其传送给另一层加以解析。

可以通过指定长度的上限与下限，或是只指定上限来表示变长向量。例如：

```
opaque stuff<1..20> /* A string of between 1 and 20 bytes */  
uint32 numbers<16> /* Up to 4 32 bit integers */
```

定长向量在线路上只是简单地将元素逐个连结起来（第一个元素最先出现）进行编码。变长向量的编码方式与之类似，只是要在前面缀以长度字段。长度字段是一个大到足以编码向量长度上限的整数类型。例如，`opaque vec<300>` 就需要一个 `uint16` 类型的长度字段。

● 枚举类型

枚举类型就是假定只有一系列特定值的字段，且每个值都有名字。例如：

```
enum{red(1), blue(2), green(3)}colors;
```

指定一个名为 `colors` 的类型，它们可以接收值 `red`、`blue` 和 `green`，在线路上用整数 1、2 和 3 来表示。当在线路上对枚举量进行编码时，用一个大到足以容纳其最大值的整数类型来表示，因此使用一个字节的 `uint8` 来表示 `colors`。通过包含一个未命名的值，可以为一个枚举量显式地指定最大尺寸：

```
enum{warning(1), fatal(2), (255)}AlertLevel;
```

作为一种特殊情况，可以定义不包含任何值的枚举量。规范中偶尔使用它来指代内部实现的状态值。显然，这种 `enum` 绝不会在线路上进行编码，因为不存在如何对这些值进行编码的定义。

```
enum{high, low}security;
```

● 结构

可以使用 `struct` 定义来构造结构（structure）化类型，这与 C 语言的 `struct` 语法非常相似：

```
struct{  
    type1 field1,  
    type2 field2,  
    type3 field3,
```

...

```
}name;
```

可以通过将各个字段的编码连结起来，并按照它们在 struct 定义中出现的先后顺序在线路上对 struct 进行编码。struct 的定义还可以嵌套，如果一个 struct 直接包含于另一个 struct 中，则可以省略这个 struct 的名字字段。

变体类型

可以将结构定义成依赖外部信息而变化的变体（Variant）类型。使用一种表面上看起来类似于C语言中的 switch 语句的结构来进行定义，但它实际上更像C语言中的 union 或 ASN.1 的 CHOICE。

```
select(type){  
    case value1:Type1  
    case value2:Type2  
    ...  
}name;
```

控制 select 内容的类型总是枚举类型，下面是一个例子：

```
struct{  
    select(KeyExchangeAlgorithm){  
        case rsa:EncryptedPreMasterSecret;  
        case diffie_hellman:DiffieHellmanClientPublicValue;  
    }exchange_keys;  
}ClientKeyExchange;  
  
enum{rsa, diffie_hellman}KeyExchangeAlgorithm;
```

这个例子根据 KeyExchangeAlgorithm 的值选择性地定义 ClientKeyExchange 结构。如果 KeyExchangeAlgorithm 是 rsa，那么值的类型就是 DiffieHellmanClientPublicValue。注意，消息中没有任何地方表示 KeyExchangeAlgorithm 值，而只是系统上下文的一部分。然而一些变体也指代消息中的其他字段。

3.9 握手消息结构

每种 SSL 握手消息都由一个简单的头信息以及依赖于消息类型的消息体组成。图 3.7 描述了消息的定义。

头信息为 4 字节长，由一个字节的类型字段和 3 个字节的长度字段组成。长度字段表示剩余握手消息的长度（不包括类型与长度字段），余下的消息内容完全有赖于 msg_type 字段。严格来讲，长度字段并不是必需的，因为可以从各组成部分来判定握手消息的结尾。然而，提供长度字段可以使协议的实现要容易得多。实现可以在一层上对连续的握手消息进行拆分，然后在另一层中进行处理。注意，从握手消息到 SSL 记录的映射并不是固定的。单条



记录中可以出现多条握手消息。从理论上讲，单一的一条握手消息可以跨越多条记录，但在实际应用中却不会出现这样的情况。

```

struct {
    HandshakeType msg_type;
    uint24 length;
    select (HandshakeType) {
        case hello_request:    HelloRequest;
        case client_hello:     ClientHello;
        case server_hello:     ServerHello;
        case certificate:      Certificate;
        case server_key_exchange: ServerKeyExchange;
        case certificate_request: CertificateRequest;
        case server_hello_done: ServerHelloDone;
        case certificate_verify: CertificateVerify;
        case client_key_exchange: ClientKeyExchange;
        case finished:         Finished;
    } body;
} Handshake;

enum{
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange(12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    finished(20),(255)
} HandshakeType;

```

图 3.7 SSL 握手消息

3.10 握手消息

创建 SSL 连接的第一步就是握手，由它来确立剩余连接过程中的算法和密钥资料。本节详细描述了我们早先展示的 SSL 握手。我们先针对每条握手消息展示相关的规格描述语言以及扩展的握手消息跟踪信息，然后再解释每个字段的含义和值。

可以采取两种方法来阅读本节的内容。如果你想切实地理解所发生的情况，那么就阅读每一种规格描述，与 ssldump 输出的跟踪信息进行比照，并以对应的文字作为指导。如果你只是想获得总体上的了解，只需简单地阅读文字，并以结构定义和协议跟踪信息作为指导即可。



ClientHello

```

struct {
    ProtocolVersion client_version;
    Random random;
    SessionID session_id;
}

```

```
CipherSuite cipher_suites<2..2^16-1>;
CompressionMethod compression_methods<1..2^8-1>;
} ClientHello;

struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;

struct {
    uint32 gmt_unix_time;
    opaque random_bytes[28];
} Random;

opaque SessionID<0..32>;
uint8 CipherSuite[2];
enum {null(0),(255)} CompressionMethod;

ClientHello
Version 3.1
random[32]=
38 f3 cb de 80 4c b4 79 0a 07 9f b3
51 ba b8 62 69 e3 8f bf ce c7 ff 25
3c 3b 84 16 38 b2 5e f7
cipher suites
    TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
    TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
    TLS_RSA_WITH_3DES_EDE_CBC_SHA
    TLS_RSA_WITH_IDEA_CBC_SHA
    TLS_RSA_WITH_RC4_128_SHA
    TLS_RSA_WITH_RC4_128_MD5
    TLS_DHE_RSA_WITH_DES_CBC_SHA
    TLS_DHE_DSS_WITH_DES_CBC_SHA
    TLS_RSA_WITH_DES_CBC_SHA
    TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
    TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
    TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
    TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
    TLS_RSA_EXPORT_WITH_RC4_40_MD5
compression methods
    NULL
```

ClientHello 消息是发送的第一条握手消息。该消息的主要目的就是让客户端传达有关连接参数的首选项。在发送了 ClientHello 之后，客户端就等待服务器发送其 hello 消息。在 SSLv3 和 TLS 中，客户端提供它可以接受的参数，而服务器最终选定其中的一组。三种可进行磋商的参数分别是：由 client_version 字段表示的版本，由 cipher_suite 表示的加密算法以及由 compression_methods 表示的压缩算法。图 3.8 列出了主要的 TLS 加密套件（cipher suite）。

加密套件	认证算法	密钥交换算法	加密算法	摘要算法	编号
TLS_RSA_WITH_NULL_MD5	RSA	RSA	NULL	MD5	0x0001
TLS_RSA_WITH_NULL_SHA	RSA	RSA	NULL	SHA	0x0002
TLS_RSA_EXPORT_WITH_RC4_40_MD5	RSA	RSA_EXPORT	RC4_40	MD5	0x0003
TLS_RSA_WITH_RC4_128_MD5	RSA	RSA	RC4_128	MD5	0x0004
TLS_RSA_WITH_RC4_128_SHA	RSA	RSA	RC4_128	SHA	0x0005
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5	RSA	RSA_EXPORT	RC2_40_CBC	MD5	0x0006
TLS_RSA_WITH_IDEA_CBC_SHA	RSA	RSA	IDEA_CBC	SHA	0x0007
TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	RSA_EXPORT	DES40_CBC	SHA	0x0008
TLS_RSA_WITH DES_CBC_SHA	RSA	RSA	DES_CBC	SHA	0x0009
TLS_RSA_WITH_3DES_EDE_CBC_SHA	RSA	RSA	3DES_EDE_CBC	SHA	0x000A
TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA	RSA	DH_DSS_EXPORT	DES_40_CBC	SHA	0x000B
TLS_DH_DSS_WITH DES_CBC_SHA	DSS	DH	DES_CBC	SHA	0x000C
TLS_DH_DSS_WITH_3DES_EDE_CBC_SHA	DSS	DH	3DES_EDE_CBC	SHA	0x000D
TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SA	RSA	DH_EXPORT	DES_40_CBC	SHA	0x000E
TLS_DH_RSA_WITH DES_CBC_SHA	RSA	DH	DES_CBC	SHA	0x000F
TLS_DH_RSA_WITH_3DES_EDE_CBC_SHA	RSA	DH	3DES_EDE_CBC	SHA	0x0010
TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	DSS	DHE_EXPORT	DES_40_CBC	SHA	0x0011
TLS_DHE_DSS_WITH DES_CBC_SHA	DSS	DHE	DES_CBC	SHA	0x0012
TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	DSS	DHE	3DES_EDE_CBC	SHA	0x0013
TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	RSA	DHE_EXPORT	DES_40_CBC	SHA	0x0014
TLS_DHE_RSA_WITH DES_CBC_SHA	RSA	DHE	DES_CBC	SHA	0x0015
TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	RSA	DHE	3DESDED_CBC	SHA	0x0016
TLS_HD_anon_EXPORT_WITH_RC4_40_MD5	-	DH_EXPORT	RC4_40	MD5	0x0017
TLS_DH_anon_WITH_RC4_128_MD5	-	DH	RC4_128	MD5	0x0018
TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	-	DH	DES_40_CBC	SHA	0x0019
TLS_DH_anon_WITH DES_CBC_SHA	-	DH	DES_CBC	SHA	0x001A
TLS_DH_anon_WITH_3DES_EDE_CBC_SHA	-	DH	3DES_EDE_CBC	SHA	0x001B
TLS_DHS_DSS_EXPORT1024_WITH DES_CBC_SHA†	RSA	RSA	DES_CBC	SHA	0x0062
TLS_DHE_DSS_EXPORT1024_WITH DES_CBC_SHA†	RSA	RSA	DES_CBC	SHA	0x0063
TLS_RSA_EXPORT1024_WITH RC4_56_SHA†	RSA	RSA	RC4_56	SHA	0x0064
TLS_DHE_DSS_EXPORT1024_WITH RC4_56_SHA†	RSA	RSA	RC4_56	SHA	0x0065
TLS_DHE_DSS_WITH RC4_128_SHA†	RSA	RSA	RC4_56	SHA	0x0066

图 3.8 可用的 TLS 加密套件

client_version 字段应当包含客户端准备接受最高 SSL 版本号。对于 SSLv3 来说，这就是 major=3, minor=0。对 TLS 来说就是 major=3, minor=1。在这里的跟踪信息中，客户端表示使用 TLS。一般来讲，都期望实现使用所有较小的版本。SSL 没有提供表示不使用先前版本的设施。意思就是说如果客户端知道（举例来说）版本 2.0 是不安全的，也没有办法告

诉服务器去使用 3.0 或更高的版本。你不得不简单地与服务器进行尝试性接触，并在服务器选择版本 2 时终止连接。

一次连接的所有加密选项都被捆绑成各种 CipherSuite（加密套件），由任意选取的两字节常量来表示。CipherSuite 指定服务器的认证算法、密钥交换算法、批量加密算法和摘要（消息完整性）算法。这些套件是按照客户端的选择倾向性按照递减的顺序排列的。图 3.8 展示了一组 TLS 加密套件的列表。

以†标注的加密套件不是 TLS 标准的一部分，它们是为了响应美国有关放宽允许 1024 位密钥磋商与 56 位对称加密的出口政策在[Banes1999]中引入的。这些加密套件允许 1024 位的 RSA，以及结合 DES 或 56 位 RC4 的 DH。[Banes1999]还增加了对带有 RC4-128 的 DSS/DH 的支持，这在 TLS 标准中是没有的。尽管这些算法广为实现，但美国出口政策的全面放开使得这些算法不再有存在的必要，因此是否要对其进行标准化还不清楚。注意，由于这些算法被标注为可出口的，所以它们要经过第 3.11 节所描述的出口密钥导出过程这一环。

最后一项可磋商的参数为压缩算法，压缩算法由单字节常量来表示。在实际应用中，主要出于专利的考虑，没有为 SSL 定义任何压缩算法。因此唯一支持的算法就是强制性的 NULL 算法，该算法不改变任何数据。一些专有实现实现了一些私有的压缩算法，OpenSSL 也是这样。

ClientHello 中的第二个参数是一个 32 字节的值（称做 random），它被用做密钥产生过程的一项输入。该值的前四个字节为产生消息时的时间（从 Unix 创世纪，即格林威治 1970 年，1 月 1 日午夜 12 点以来的秒数），而其他 28 个字节应当是随机产生的。注意，即便头 4 个字节不是时间，协议也会工作正常。客户端与服务器的时钟不必同步。随机数据的目的就是确保即便使用同一个 pre_master_secret，所产生的加密和 MAC 密钥也是不同的。这样就可以阻止重放攻击。

ClientHello 还包含一个 session_id 参数。该会话 ID 是一个可长达 32 字节的字节字符串，由客户端用来指示它希望重复使用前一次连接时的加密密钥资料，而不是再产生新的资料。通常这样做速度会快得多，因为公用密钥操作的计算开销昂贵。在这里的跟踪信息中，由于客户与服务器先前没有可以恢复使用的会话，所以 session_id 字段为 0 字节长。将在第 4 章看到一个恢复使用会话的例子。

serverHello

```
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
}ServerHello;

ServerHello
SSL version 3.1
random [32] =
38 f3 cb d1 33 63 1c c7 2e 8c 56 43
9e fb 20 70 cc 4b 16 06 4d 5a 8b 15
```

```

e3 9f 0d 47 39 16 5f 5c
session_id[32]=
74 6d 09 76 1d 2a c9 02 4a a1 a3 4e
27 5c 18 63 8a d9 4a 59 f9 c3 14 a5
c4 b3 a4 f6 61 ef f5 cd
cipherSuite           TLS_RSA_WITH_DES_CBC_SHA
compressionMethod     NULL

```

ServerHello 消息由服务器用来从客户端提供的各种选项中进行选择。这条消息中的 server_version、cipher_suite 和 compression_method 字段就是要在 SSL 连接中使用的版本、加密算法和压缩算法。

在这里的跟踪信息中，客户端与服务器均采用 TLS，所以选择的版本就是 3.1。服务器选择的加密套件为 TLS_RSA_WITH_DES_CBC_SHA：即使用 DES-CBC 完成消息加密，以及用 SHA 来完成消息摘要的 RSA 密钥交换。注意，服务器在选择客户端希望使用的加密套件时完全不受约束，即便它支持某种套件也可以不予选择。SSL 规范中并没有规定如何从客户端支持的密码中进行选择。服务器可以尊重客户端的首选项也可以强行使用自己的偏好。然而，通常的做法是让服务器选择客户端所支持的，并且是最想使用的加密算法。这对于客户端实现者来说有些不太方便，因为他们无法预测服务器会选中哪一种加密算法。而且这还意味着直到连接过程开始以前，他们都无法给用户提供由于建立给定连接而可能造成的后果的反馈。

最后，服务器选中了惟一可用的压缩算法——NULL。此外，服务器还提供了一个随机值。用它连同客户端的随机值，以及 pre_master_secret 一起产生连接中使用的密钥资料。这样，即便客户端出现问题并在两次握手使用同一个随机值，也可以确保最终的加密密钥是不同的。而且还可以防止我们在第 1.8 节中所讨论的握手重放攻击企图。

通常情况下，服务器会提供一个可由客户端恢复会话使用的 session_id。如果服务器不想恢复会话的话，就可以提供 0 长度的会话 ID。这里服务器提供了一个这样的 ID，以指示它准备恢复本次会话，而会话 ID 究竟是怎么产生的则由实现来决定。尽管会话 ID 常常是完全随机的，但还是具有某种结构，以便服务器更容易地在其高速缓存中查找相应的会话。

Certificate

```

struct {
    ASN.1Cert certificate_list<1 ..2^24-1>;
} Certificate;

opaque ASN.1Cert<2^24-1>;

Certificate
    Subject
        C=AU
        ST=Queensland
        O=CryptSoft Pry Ltd
        CN=Server test cert (512 bit)
Issuer
    C=AU
    ST=Queensland

```

```
O=CryptSoft Pty Ltd
CN=Test CA (1024 bit)
Serial          04
certificate[493]=
30 82 01 e9 30 82 01 52 02 01 04 30
0d 06 09 2a 86 48 86 f7 0d 01 01 04
05 00 30 5b 31 0b 30 09 06 03 55 04
06 13 02 41 55 31 13 30 11 06 03 55
04 08 13 0a 51 75 65 65 6e 73 6c 61
6e 64 31 1a 30 18 06 03 55 04 0a 13
11 43 72 79 70 74 53 6f 66 74 20 50
74 79 20 4c 74 64 31 1b 30 19 06 03
55 04 03 13 12 54 65 73 74 20 43 41
20 28 31 30 32 34 20 62 69 74 29 30
1e 17 0d 39 38 30 36 32 39 32 33 35
32 34 30 5a 17 0d 30 30 30 36 32 38
32 33 35 32 34 30 5a 30 63 31 0b 30
09 06 03 55 04 06 13 02 41 55 31 13
30 11 06 03 55 04 08 13 0a 51 75 65
65 6e 73 6c 61 6e 64 31 1a 30 18 06
03 55 04 0a 13 11 43 72 79 70 74 53
6f 66 74 20 50 74 79 20 4c 74 64 31
23 30 21 06 03 55 04 03 13 1a 53 65
72 76 65 72 20 74 65 73 74 20 63 65
72 74 20 28 35 31 32 20 62 69 74 29
30 5c 30 0d 06 09 2a 86 48 86 f7 0d
01 01 01 05 00 03 4b 00 30 48 02 41
00 9f b3 c3 84 27 95 ff 12 31 52 0f
15 ef 46 11 c4 ad 80 e6 36 5b 0f dd
80 d7 61 8d e0 fc 72 45 09 34 fe 55
66 45 43 4c 68 97 6a fe a8 a0 a5 df
5f 78 ff ee d7 64 b8 3f 04 cb 6f ff
2a fe fe b9 ed 02 03 01 00 01 30 0d
06 09 2a 86 48 86 f7 0d 01 01 04 05
00 03 81 83 00 95 be f7 e4 19 27 b6
18 78 03 15 f9 8e b9 ae 08 b2 36 fd
25 58 48 99 63 00 4a 23 82 96 46 65
30 44 83 26 3b 2c ce 0f fa f9 df d6
fb c4 eb 6c e6 e1 6b 3a 65 f7 91 62
bd 70 55 b9 c6 e3 f5 db 9d 87 b0 0e
21 9b b5 87 53 00 3e 5c a4 9d cf 54
77 cd 7a bf 3d c5 7a 30 78 aa a5 28
69 78 e7 96 4a c8 80 46 eb fe e9 fb
8d 24 bc e9 63 9e d2 14 61 c0 79 09
15 41 9c 3d 97 fd 34 3d b6 12 d7 3e
01
```

Certificate 消息就是 X.509 证书序列。证书依序提供，第一个证书是服务器所有的证书，下一个证书（如果有的话）包含证明服务器证书的密钥，以此类推。在所有常见的 SSL 和

TLS 加密套件中，服务器必须发送其证书（存在几种很少使用的加密套件，其中服务器是匿名的）。证书中的密钥要么用来加密 pre_master_secret，要么用来对 ServerKeyExchange 进行验证，这要依赖于所选择的加密套件。

这里，服务器只发送了一个证书，Server Test Cert (512 位)。它假定客户端已经具有 Test CA 的 CA 证书 (1024 位)。我们在这里所展示的是对最重要的证书信息的总结。你可以在第 1 章找到更详细的、有关 X.509 证书结构的信息。

● ServerHelloDone

```
struct { } ServerHelloDone;
```

是一条空消息，它表示服务器已经发送了在此阶段所要发送的全部信息。这条消息是必需的，因为 Certificate 消息之后还可以有一些可选的消息。我们将在第 4 章看到那种消息的一些例子。

● ClientKeyExchange

```
struct {
    select (KeyExchangeAlgorithm) {
        case rsa: EncryptedPreMasterSecret;
        case diffie_hellman: DiffieHellmanClientPublicKey;
    } exchange_keys;
} ClientKeyExchange;

struct {
    ProtocolVersion client_version;
    opaque random[46];
} PreMasterSecret;

struct {
    public-key-encrypted PreMasterSecret pre_master_secret;
} EncryptedPreMasterSecret;

enum { implicit, explicit } PublicValueEncoding;

struct {
    select (PublicValueEncoding) {
        case implicit: struct {};
        case explicit: opaque DH_Yc<1..2^16-1>;
    } dh_public;
} DiffieHellmanClientPublicKey;

ClientKeyExchange
    EncryptedPreMasterSecret [ 64 ] =
        17 4d 00 32 bc b2 af 95 09 0a 45 24
        97 d8 34 dc 73 20 4d 00 91 a5 0d ed
        c3 f0 b4 f5 32 6f 13 cc ea 41 00 5e
        bb 05 f1 b7 e2 c4 fa 1b 40 c1 2a f3
        00 20 83 43 2d 8c 2a 53 3b 33 cc 5f
        0b bc db a2
```

ClientKeyExchange 消息提供创建 pre_master_secret 时客户端所提供的资料。当使用 RSA 密钥交换时，这就是指客户端产生一个 PreMasterSecret 结构并用服务器的 RSA 密钥对其进行加密。pre_master_secret 是一个 48 字节的值，其中由两个版本号字节以及跟在后面的 46 个随机产生的字节。使用一种加密学理论上安全的 RNG（请参见第 5 章来了解有关 RNG 的信息）来选择这些字节非常关键。RFC1750[Eastlake1994]提供了有关产生随机数的指南。Wagner 和 Goldberg 对 Netscape SSLv2 实现实施[Goldberg1996]的成功攻击就是基于 Netscape 的 RNG 中的缺陷。将在第 5 章提供有关产生高强度随机数的指导。

Wagner 和 Goldberg 对 Netscape 的 RNG 进行反汇编并观察发现 RNG 以当天的时间和进程 ID 为种子。因为这两种数字在数量上相当有限，因此可以相当容易地搜索整个空间并恢复 SSL 会话密钥。该过程在当时要花费一个小时，而现在则要更快一些。

EncryptedPreMasterSecret 结构中的 public-key-encrypted 运算符表示要使用接收方的公用密钥来对 pre_master_secret 字段进行加密。然后再将加密后的密钥表示为一个变长向量：

Opaque encrypted_data<0..2¹⁶⁻¹>

ClientKeyExchange 消息是 SSLv3 与 TLS 之间存在差异的、仅有的两条消息之一。这种差异要归咎于 Netscape SSLv3 实现中存在的一个 bug，而这个错误也由其他的 SSLv3 实现厂商效仿。当使用 RSA 来完成加密的时候，加密数据的长度是冗余的，因为 EncryptedPreMaster 的长度隐含地由 ClientKeyExchange 消息和服务器 RSA 密钥的长度来决定。因此，Netscape 的 SSLv3 实现在对 EncryptedPreMasterSecret 进行编码的时候省去了长度字节，而其他大多数 SSLv3 实现厂商也沿用他们（不正确）的做法。也就是说，大多数 SSLv3 实现均违反了 SSLv3 规范。TLS 规范中的一项实现说明指出大多数 SSLv3 实现不符合规范的要求，因而要求 TLS 实现应正确地将长度字节包括在内。我们所看到的跟踪信息源于 TLS 模式下的一种实现，因此包括长度字节。

ChangeCipherSpec

```
struct {
    ChangeCipherSpecType type;
} ChangeCipherSpec;

enum { change_cipher_spec(1), (255) } ChangeCipherSpecType;
```

ChangeCipherSpec 消息指示发送实现已经切换至新磋商好的算法和密钥资料，而未来发送的消息将使用那些算法加以保护。ChangeCipherSpec 消息并不算是握手过程中的一部分，相反而是有自己的内容类型，在这一点上它是惟一的。规范中作为一项性能改进对其描述如下：

注意，为了协助避免管道延迟，ChangeCipherSpec 只是一种独立的 TLS 协议内容类型，而不是一条 TLS 握手消息。

然而，作者只知道一种实现（Netscape 的）以某种有趣的方式利用了这项功能。Netscape 使用它来确保 ChangeCipherSpec 消息与 Finished 消息分开进行传输。由于 ChangeCipherSpec 消息无法加密，而 Finished 消息则必须进行加密，所以它们不能在同一条记录中进行传输。而使之处于不同的层次正是为了保证这一点。

这项技巧只有在你的实现企图在同一条记录中发送多条握手消息时才会有用。正如将在第 6 章所看到的，有时在同一 TCP 段中发送多条握手消息会提升性能，在同一条记录中进行传送则是一种实现方式。然而，许多种实现却是选择在同一 TCP 段中传送多条记录，其效果大多是相同的。对于这种实现来说，这项协议功能很不方便，它使握手状态机复杂化。

一种典型的实现是让握手状态机只读取握手类型的数据。不过还是需要知道 ChangeCipherSpec 已被接收，而且必须在记录层处理过程中独立传输（out of band）此种消息。如果 ChangeCipherSpec 消息是一种握手消息的话，握手层只会简单将其从网络上读取过来。而记录层的代码需要特殊的支持才能识别出 ChangeCipherSpec 消息，接着要么告知握手层，要么根据握手是否处于良好的状态而抛出一个错误。

ChangeCipherSpec 消息只由一个单一类型字节构成，它主要是占位性质的，因为该字节当前只能具有一个单一的值 1。目前还没有公开起草来扩充这条消息用途的建议。

Finished

```
SSLv3:
struct {
    opaque md5_hash[16];
    opaque sha_hash[20];
} Finished;

TLS:
struct {
    opaque verify_data[12];
} Finished;

Finished
verify_data [12] =
03 3b 69 b7 26 e3 0e 23 fc 03 79 27
```

Finished 消息是第一条使用新的加密参数进行加密的消息，它使实现能够确认没有任何消息被攻击者篡改过。

尽管 SSLv3 和 TLS Finished 消息的具体细节不同，但总体概念是相似的。每一方都向另一方发送磋商后的主密码与连结起来的握手消息的摘要信息，而另一方将其与本地计算得出的摘要进行比照。如果不同，则表示握手被篡改了。SSLv3 Finished 消息是根据流行的加密消息认证函数 HMAC (Krawczyk1997) 的模式来建立的，它使用嵌套散列创建具有可证明的、特定安全特征的结构。图 3.9 展示了 SSLv3 中的计算。由于加号表示连接，因此第一遍调用 MD5 (最内层) 的输入就是将所有握手消息、Sender 常量、主密码以及一些填充字节 (pad1 为字节 0x36 重复 48 次所形成的字符串) 的连结。第二遍调用 MD5 的输入为主密码、一些填充字节 (pad2 为字节 0x5c 重复 48 次所形成的字符串) 以及第一次 MD5 计算输出的连结。SHA-1 除了填充字节重复 40 次而不是 48 次之外也是同样的。Sender 常量的作用是确保客户端与服务器产生不同的 Finished 消息，这样就可以阻止攻击者将 Finished 消息返发给发送端。

尽管 SSLv3 Finished 消息是基于 HMAC 的，但是为了可供 SSL 使用还是进行了修改。特别的，鉴于在 HMAC 中密钥总是第一项输入内容，由于主密码只有在握手过程完成一半时才可用，所以握手消息是内层摘要的第一项输入。因此，HMAC 中的一些安全特征就可

能不再适用。HMAC 设计者们的言论是在 TLS 中改变这种计算的主要原因。

```
enum { client(0x434C4E54), server(0x53525652) } Sender;

md5_hash = MD5(master_secret + pad2 +
    M D5(handshake_messages +
        Sender + master_secret + pad1));
sha_hash = SHA-1(master_secret + pad2 +
    SHA-1 (handshake_messages +
        Sender + master_secret + pad1));
```

图 3.9 SSLv3 Finished 消息的计算

TLS 广泛使用一种基于 HMAC 的 PRF (伪随机函数) 来完成许多包括 Finished 消息 (参见图 3.10) 在内的加密计算。(我们将在密钥导出一节看到更多有关 PRF 的内容)。使用 PRF 来创建 verify_data 的值。输入为主密码，代表完成标签的 ASCII 字符串 (“client” 或 “server”) 以及握手消息的 MD5 和 SHA-1 摘要的连结。[0..11] 指示 PRF 输出的头 12 个字节用于创建 verify_data。

```
verify_data =
    PRF(master_secret, finished_label, MD5(handshake_messages) +
        SHA-1(handshake_messages)) [0..11];
```

图 3.10 TLS Finished 的计算

注意，在创建 Finished 消息时同时用到了 MD5 和 SHA-1。这样做的理由是为了提供附加的安全，以应付摘要算法的缺陷。这样，攻击者就必须同时攻破 MD5 和 SHA-1 才能伪造出 Finished 消息。

编程提示：处理 Finished 消息

注意，尽管 Finished 消息必须包含到目前为止所有握手消息的摘要，但它并不包括其自身的摘要。然而，第二条 Finished 消息必须包含第一条的摘要。这样就给实现者提出了一道有趣的编程问题。过于简化的实现有可能会像图 3.11 中的代码那样来编写握手处理程序。

```
receive_handshake_message(context,message) {
    digest_handshake_message(context,message);
    switch(message.type){
        case ClientHello:
            handle_client_hello(context,message);
            break;
        ...
        case Finished:
            handle_finished(context,message);
            break;
    }
}
```

图 3.11 最初的 Finished 握手处理程序

不幸的是，当接收方计算 Finished 的值时，会将刚刚收到的 Finished 消息作为摘要计算的一部分，因此这些值是不会匹配的。然而，服务器不能简单地停止进行摘要计算，因为它需要将第一条 Finished 消息的摘要包括在响应客户端 Finished 消息的应答中。所以它要么维

护两组摘要，要么在处理 Finished 消息之前再创建一份摘要对象的拷贝，如图 3.12 所示。

```

receive_handshake_message(context,message) {
    if(message.type==Finished){
        digests=copy_digests(context);
    }
    digest_handshake_message(context,message)

    switch(message.type){
        case ClientHello:
            handle_client_hello(context,message)
            break;
        ...
        case Finished:
            handle_finished(context,message,digests);
            break;
    }
}

```

图 3.12 修改后的 Finished 握手处理程序

3.11 密 钥 导 出

一旦交换了 pre_master_secret，每一种实现就都需要将其扩展成独特的加密密钥，用以完成加密、认证等任务。我们使用一种密钥导出函数来实现这种扩展。SSLv3 与 TLS 的密钥导出函数是相似的，但是在所使用的具体加密变换上有所不同。我们将在最后根据 SSLv3 过程描述这种差异的时候讲述有关 TLS 密钥导出函数的知识。你可以参考图 3.13 有关该过程的图示。

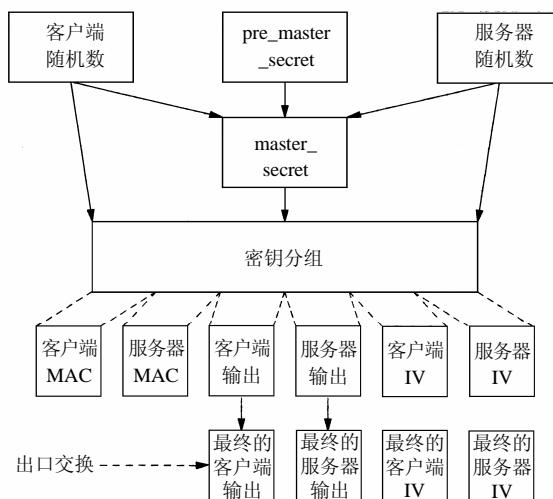


图 3.13 密钥导出

PRF

整个 TLS 密钥导出过程都使用 PRF 作为密钥导出函数。其中的关键就是将 pre_master_secret 扩展成另一个称做 master_secret 的密码，然后又将 master_secret 扩展为各种加密和 MAC 算法所要用到的加密密钥。就第 1.8 节中的密钥导出函数而言，客户端与服务器各自独立地使用同一个密钥导出函数，根据共享 master_secret 来产生密钥。PRF 接受三个变元，即一个密码（应当是随机的）、一个固定的 ASCII 字符串（标签）以及一个种子（seed）（应当是随机的，但却是公开的）、形式为 PRF(secret, label, seed)。使用标签的目的是为了使你在可以使用相同加密资料的情况下，能够简单地通过使用针对每个密钥不同的标签来产生不同的密钥。简单地取标签的摘要作为对应的 ASCII 字符串。

PRF 产生包含伪随机字节的、任意长度的字符串，我们使用 TLS 向量记法来表示。因此，PRF()[0..9] 就表示 PRF 输出的头 10 个字节。PRF 的计算相当复杂，除了实现者感兴趣外，没什么有趣的地方。在此，仅仅为了叙述全面而加以论述，因此你可以自由选择跳过这些内容。考试是不会考的。

首先将密码部分分成两半，S1 和 S2。S1 是密码的头一半，S2 是后一半。如果原始密码长度为奇数，那么 S1 的最后一个字节就与 S2 的第一个字节相同。

每一半都用做一个扩展函数 P_hash 的密码成分，这是一种基于 HMAC 的函数。P_hash 通过如图 3.14 所示的过程产生一个任意长度的字节字符串。

$$\begin{aligned} \text{P_hash}(\text{secret}, \text{seed}) = & \text{HMAC_hash}(\text{secret}, \text{A}(1) + \text{seed}) + \\ & \text{HMAC_hash}(\text{secret}, \text{A}(2) + \text{seed}) + \\ & \text{HMAC_hash}(\text{secret}, \text{A}(3) + \text{seed}) + \dots \end{aligned}$$

A() is defined as:

$$\begin{aligned} \text{A}(0) &= \text{seed} \\ \text{A}(i) &= \text{HMAC_hash}(\text{secret}, \text{A}(i-1)) \end{aligned}$$

HMAC_hash means HMAC using a given hash algorithm, e.g.,
HMAC_MD5 means HMAC using MD5.

图 3.14 P_hash 函数

现在这些函数已经定义好了，我们通过将 P_MD5 与 P_SHA-1 进行异或来构造出 PRF：

$$\begin{aligned} \text{PRF}(\text{secret}, \text{label}, \text{seed}) = & \text{P_MD5}(\text{S1}, \text{label} + \text{seed}) \oplus \\ & \text{P_SHA-1}(\text{S2}, \text{label} + \text{seed}); \end{aligned}$$

因为 MD5 摘要值 16 字节长而 SHA-1 摘要值 20 字节长，所以 P_MD5 与 P_SHA-1 无法对齐。因此如果你想要产生一个 80 字节的输出的话，就不得不执行 5 遍 P_MD5 和 4 遍 P_SHA-1。

密钥导出函数的三个非固定输入为 pre_master_secret、client_random 和 server_random。pre_master_secret 只是一个密码字节字符串，在使用 RSA 密钥交换时，它是一个由客户端产生的 48 字节的字符串，并用服务器的公用密钥对其进行加密。pre_master_secret 是这一过程中唯一的密码信息来源。其他两项输入，client_random 和 server_random 分别为源于 ClientHello 和 ServerHello 消息的随机值。

密钥导出函数的第一步就是将 pre_master_secret 变换成 master_secret。这是通过将伪随机函数 PRF 作用于 pre_master_secret、client_random 和 server_random 来实现的：

```
master_secret = PRF(pre_master_secret, "master secret",
                     client_random + server_random)[0..47];
```

现在，使用以 master_secret 为参数的 PRF 来为我们所需要的所有算法提供足够的密钥资料。根据所使用的加密套件，或许会需要多达 6 个值，即一个加密密钥、一个 MAC 密钥，以及分别用于客户端与服务器的初始化向量（IV）。首先将所有需要的值的长度加起来，并使用 PRF 来产生那么多的字节。例如，如果使用的是 DES 和 MD5，那么每个加密密钥需要 8 个字节，每个 HMAC-MD5 密钥 16 个字节，而每个 IV 需要 8 个字节，总共 48 字节：

```
key_block = PRF(master_secret, "key expansion",
                 server_random + client_random)
```

注意，这里 client_random 和 server_random 的顺序与 master_secret 计算中的顺序不同。

一旦拥有了密钥分组，就可以按照下面的顺序抽取密钥：client_write_MAC_secret、server_write_MAC_secret、client_write_key、server_write_key、client_write_IV 及 server_write_IV。

可以出口的算法

由于设计 SSLv3 和 TLS 时美国的出口限制，可以出口的加密算法都需要将加密密钥缩短为使用不超过 2^{40} 次操作即可攻破的长度。为了做到这一点，我们首先产生一个短的过渡密钥，然后再使用 PRF 重新对其进行扩展。这种方法可以阻止一种预算攻击，其中攻击者简单地为 2^{40} 个密钥构造一张大表。服务器与客户端的随机值作为可以阻止这种攻击的“变形因子”（salt）。虽然攻击者仍然可以通过尝试所有的中间密钥实施攻击，但是无论从时间上还是存储空间上都行不通。

自从设计 SSL/TLS 以来，出口形式已经发生了变化。40 位的密钥实际上难以保证数据安全。对大多数中等装备的商业攻击者来说，攻击 40 位的密钥不在话下。1998 年，建造出来一种专攻 DES 密钥搜索的机器[Gilmore1998]，它表明对于专心致志的攻击者而言 56 位的密钥是可以被攻破的。美国政府对此做出反应，决定可以出口具有 56 位密钥的加密系统了。现已对 TLS 规范进行了修改，以包含 56 位的可出口加密算法。这些加密算法还允许 1024 位而不是 512 位的 RSA 密钥。在 2000 年 1 月份，美国政府取消了所有出口限制。然而，仍然存在许多只具有可出口模式的老软件，知道这些内容非常重要。出口管理局（Bureau of Export Administration）的 Web 站点有了这些内容，你可以通过下面的网址访问到它们：

<http://www.bxa.doc.gov/Encryption/Default.htm/>

同时还要求（同样是由于出口的原因）使用非保密的 IV（正常情况下 SSL IV 部分是根据保密数据产生的，因此也就是保密的），所以我们单纯从公开的 client_random 和 server_random 值来导出这个值。iv_block 的头半部分用来产生客户端用以加密数据的 IV，而后半部分则是服务器用以加密数据的 IV（参见图 3.15）。

```
final_client_write_key =  
    PRF(client_write_key, "client write key", client_random + server_random)  
final_server_write_key =  
    PRF(server_write_key, "server write key", client_random + server_random)  
iv_block = PRF(O, "IV block", client_random + server_random)
```

图 3.15 出口密钥资料的导出

SSLv3 的密钥导出

SSLv3 的密钥导出与 TLS 的密钥导出类似，只不过将 PRF 替换为一系列基于 MD5 和 SHA-1 组合的扩展函数。使用诸如“A”、“BB”等常量确保每个摘要的输出都是不同的，即便保密数据相同也是如此。图 3.16 描述了这个过程。

```
master_secret =  
    MD5(pre_master_secret + SHA-1 ("A" + pre_master_secret +  
        client_random + server_random)) +  
    MD5(pre_master_secret + SHA-1 ("BB" + pre_master_secret +  
        client_random + server_random)) +  
    MD5(pre_master_secret + SHA-1 ("CCC" + pre_master_secret +  
        client_random + server_random))  
  
key_block =  
    MD5(master_secret + SHA-1 ("A" + master_secret +  
        server_random + client_random)) +  
    MD5(master_secret + SHA-1 ("BB" + master_secret +  
        server_random + client_random)) +  
    MD5(master_secret + SHA-1 ("CCC" + master_secret +  
        server_random + client_random)) +  
    ...
```

图 3.16 SSLv3 的密钥导出

注意，client_random 和 server_random 的顺序对于这两种计算是颠倒的。

对 TLS 来说，当它与可出口加密算法一起使用时，SSLv3 推迟（postprocess）密钥资料的处理以削弱其强度，如图 3.17 所示。

```
final_client_write_key =  
    MD5(client_write_key + client_random + server_random);  
final_server_write_key =  
    MD5(server_write_key + server_random + client_random);  
client_write_IV:  
    MD5(client_random + server_random);  
server_write_IV =  
    MD5(server_random + client_random);
```

图 3.17 SSLv3 出口密钥的导出

注意，尽管 SSLv3 的密钥导出针对 TLS 进行了修改，但是 SSLv3 的密钥导出中并不存在已知的弱点。

3.12 记录协议

SSL 记录协议是一种相当简单的封装协议。它由一系列经过加密、受完整性保护的记录组成。每条记录都由一个短小的头信息块和一个加密的数据块构成，而数据块本身又包含内容与 MAC。图 3.18 描述了一个记录范例。头信息以白色来表示，而经过加密的负载(payload)则用深色调来表示。按照解密后的样子绘制出加密负载的各部分内容。

这里所展示的记录是以一种诸如 DES 的分组密码进行加密的，并使用 MD5 来产生 MAC。因此，需要对记录进行填充以适应 DES 的分组长度。注意，尽管加密数据必须是分组长度的倍数，但是记录本身并不沿分组边界对齐。

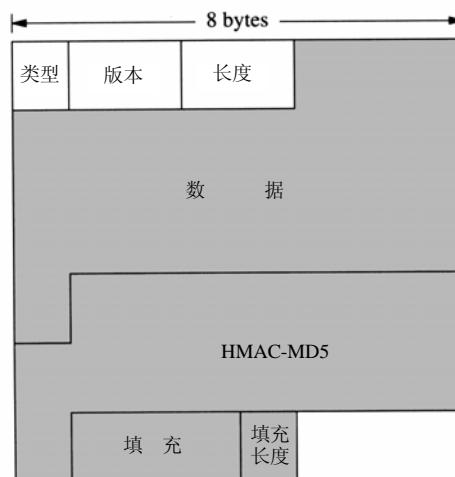


图 3.18 SSL 记录

在这里的跟踪信息中，我们让客户端与服务器每一方都写一条记录。根据 ServerHello 消息的内容，加密套件为 TLS_RSA_WITH_DES_CBC_SHA。

SSL 数据被分割成一系列的片段。协议定义指出了片段的最大尺寸，但却没有指定如何去判断什么才是片段的边界，不过它们通常是与应用数据写调用一一对应的。然而，无须保留调用 SSL_write()（或与之等价的）的边界。因此，记录层可以将任何给定的写操作分割成多条记录或反方向将多次调用得到的输入合并成一条单一的记录。最大的数据片段尺寸为 2^{14} 字节，因此更大的写操作必须被划分成该尺寸或更小尺寸的片段。注意，长度字段的长度足以表示长达 2^{16} 字节的记录。而版本较老的 Internet Explorer 确实不正确的产生过超过 2^{14} 字节的记录。

```
TLS MAC = HMAC_hash (MAC_write_secret, seq_num + type + version +
length + content)
```

一旦数据经过分片，就要经过压缩——或是在定义压缩算法的条件下进行压缩。由于没有定义压缩算法，所以就不用进行压缩了。然后，根据经过压缩的数据片段计算出 MAC 值，并与数据连结在一起。该 MAC 值涵盖了数据与头信息，还包括有一个用以阻止重放与再排

序攻击的序号。图 3.19 描述了这种 MAC 算法。

```
SSLv3 MAC=hash(MAC_write_secret + pad_2 +
    hash(MAC_write_secret + pad_1 + seq_num + content_type +
        length + content));
```

图 3.19 MAC 计算

`seq_num` 是一个 64 位的序号，对于客户端与服务器来说它是独立的（即客户端与服务器所发送的第一条消息的序号为 0）。这里的内容类型为图 3.20 中列出的内容类型之一，而版本就是 SSL/TLS 的版本号。长度为内容的长度（即记录中数据的长度），它与记录头信息中的长度不同，因为那里的长度包括 MAC 和可能的填充内容。

TLS MAC 是标准的 HMAC。SSLv3 MAC 基于 HMAC 的早期草案。`pad_1` 对 MD5 来说是将字节 0x36 重复 48 遍，对 SHA-1 来说则是重复 40 遍。`pad_2` 是将字节 0x5c 重复同样的遍数。使用填充的目的是确保 MAC 密钥与填充加在一起可以充满消息摘要第一块的整块空间。由于 MD5 和 SHA-1 分组为 64 字节长，所以一个 16 字节的密钥就意味着需要 48 字节的填充。注意，对于 SHA-1（20 字节的密钥）来说，填充应当为 44 个字节，但是规范中为 40 字节。这是 SSLv3 规范中的一处错误，但却成为 Netscape 实现的一部分，因此从未对规范进行改动。我们又一次看到了 Netscape 的实现构成了 SSLv3 的基础。

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 length;
} RecordHeader;

enum {
    change_cipher_spec(20), alert(21), handshake(22),
    application_data(23), (255)
} ContentType;

struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;
```

图 3.20 记录头信息

序号用来阻止各种对消息流的攻击，其中包括重放攻击（即攻击者重发记录），和再排序攻击（即攻击者交换一条或多条记录的顺序）。注意，尽管使用序号可以检测出这种攻击，但由于序号并不在记录中，所以无法对其加以纠正。因此，SSL 必须运行于一种能够提供可靠传输的传输协议之上（即必须按照传输的顺序将数据发送给接收方，不存在重复或丢失）。TCP 满足这些要求（数据可以以任何顺序到达，但是核心将确保应用看到的将是传输时的顺序）而 UDP 不是这样，因此 SSL 不能成功地在 UDP 上运行，因为任何记录都有可能丢失或顺序错乱，这看上去就像针对 SSL 实现实施的主动攻击。

注意，TLS 规范所描述的记录多少有些不同，它没有将记录划分成头信息和主体。使用这种表达形式为的是便于实现，而且通常也是更典型的网络协议的表达形式。

如图 3.4 所示，不同的类型值对应可在 TLS 记录层上传递的不同数据类型。我们前面已经见到了长度与版本字段。头信息中的长度是记录中的所有数据的总长，其中包括 MAC 和填充。

序列密码

序列密码无需任何填充，因此 MAC 只是简单地追加到数据的后面，而整个数据块均被加密。由于序列密码的密钥流不能重复使用，所以必须在前后记录之间保留密码加密状态。逻辑上讲，就好像所有要加密的数据都是以密码一遍加密完成的。

分组密码

分组密码的输入长度必须为分组大小（通常为 8 个字节）的倍数。由于输入数据可以具有任意尺寸，所以数据通常不沿分组边界对齐，于是就有必要增加一定数量的填充字节。最多所需要的填充字节数为 7，但是可以增加多达 255 个字节，以隐藏明文的长度使之免于被嗅探到。所使用的填充字节数在填充长度字节中记载。填充字节值必须与填充长度字节中的值相同。

例如，如果分组长度为 8 个字节，数据长度为 16 个字节，MAC 长度为 16 字节，那么填充以前就是 33 个字节，因为长度字节必须包括在结尾部分。至少需要 7 个填充字节，但是也可以使用 15、23 等。假定使用 7 字节的最小填充长度，那么最后的 8 个字节就是 07 07 07 07 07 07 07 07（7 个填充字节加上填充长度字节），即数据总长度为 40 个字节。注意，这种填充结构与标准的分组密码填充并不兼容（请参见[RSA1993c]），后者没有使用填充长度字节，只记录了填充中填充字节的总数。在这里的例子中，标准算法给出的填充为 08 08 08 08 08 08 08 08。

所有当前的分组密码都使用密码分组链接（CBC）模式。链接在前后记录之间必须是连续的：记录 X 的最后一个密码分组被用做记录 X+1 的 IV。

空密码

作为一种特殊情况，空密码就是简单地将数据未加转换地、从输入传递到输出，不带填充。开始的加密套件 TLS_NULL_WITH_NULL_NULL 甚至连 MAC 也没有附加，只是直接将明文放到记录中。注意，TLS_NULL_WITH_NULL_NULL 只有在连接开始的握手期间才可用。磋商使用此种套件则是非法的。

3.13 警示与关闭

警示协议设计用来允许一方向另一方报告例外条件。警示消息非常简单，其中包含严重程度和相应的描述。一旦收到 fatal 级别的警示消息，实现就必须终止连接而且不能再重用该会话，或者忽略该警告消息。然而实现也可以选择视警告为致命的。图 3.21 描述了 TLS 警示消息的结构。SSL 警示消息的结构与之相同，只是定义的警示少些。请参见第 4 章的内容来了解所有警示的具体内容。

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    unexpected_message(10),
    bad_record_mac(20),
    decryption_failed(21),
    record_overflow(22),
    decompression_failure(30),
    handshake_failure(40),
    no_certificate(41), SSLv3 only
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied (49),
    decode_error(50),
    decrypt_error(51 ),
    export_restriction (60),
    protocol_version(70),
    insufficient_security(71 ),
    internal_error(80),
    user_cancelled(90),
    no_renegotiation(100),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

图 3.21 TLS 警示消息

关闭

`close_notify` 警示用来指示发送方已经发送了所有要在该连接上发送的数据。这条警示的目的就是用来阻止截断攻击，即攻击者在发送方还未发送完数据之前插入一条 TCP FIN 消息，迫使接收方认为已经接收了所有的数据。因此，除非接收方收到一条 `close_notify` 警示，否则就无法肯定还有没有更多的数据要来。如果一种实现在没有预先收到 `close_notify` 的情况下收到了 TCP FIN，它就必须将该会话置为不可恢复的。

在实际应用中，许多协议包含有它们自己的数据结束标志，一种实现在收到一条这样的信息时简单地发送一条 `close_notify`，并立即关闭连接而不再等待 `close_notify` 的到来，这是一种相当常见的做法。这种行为被 SSL 和 TLS 规范所默许。这就是我们在本章开头提供的跟踪信息中发生的情况，其中客户端提供一条 `close_notify` 但服务器却没有。正常情况下，

关闭警示是以警告级别发送的。

其他警示

其他的警示均也用于指示各种类型的错误。尽管从理论上讲，实现在遇到非致命警示时可以继续，但在实际应用中，即便是这样的警示也表示另一方不能满足一方请求的内容，因此仍会导致连接的关闭。TLS 规范提供了所有可用警示的完整列表，我们将在第 4 章具体讨论每条警示的内容。

图 3.22 描述了一个警示终止连接的例子。客户端被配置成要求 DSS/DH，而服务器只支持 RSA。因此服务器通过 handshake_failure 警示拒绝连接。注意，服务器在发送完警示之后立刻关闭了连接。

```
New TCP connection: speedy(1085)  <-> romeo(4433)
I 0.0168 (0.0168) C>S Handshake
    ClientHello
        Version 3.0
        cipher suites
            TLS_DHE_DSS_WITH_DES_CBC_SHA
        compression methods
            NULL
2 0.0171 (0.0002) S>C Alert
    level      fatal
    value      handshake_failure
Server FIN
Client FIN
```

图 3.22 因警示而终止连接

3.14 总结

本章详细讨论了 SSL 最基本的模式，即使用 RSA 只进行服务器认证的模式。它为我们未来所要讨论的模式提供了背景知识，所有这些模式都与基本模式共享许多种特性。

SSL 握手在客户端与服务器之间磋商加密算法并确立密钥资料。客户端提供可供选择的加密算法，而服务器选择要用的算法。服务器使用 Certificate 消息提供其公用密钥。

pre_master_secret 以服务器的公用密钥进行加密，而使用密钥导出函数将 pre_master_secret 转换为供连接使用的加密密钥。

使用 Finished 消息来为整个握手提供完整性保护。Finished 消息包含了所有握手消息的消息摘要，从而阻止攻击者对握手进行篡改。

数据流被分割成记录，而每条记录又单独进行加密和计算 MAC，这样就同时提供了保密性和消息完整性。计算 MAC 时要依赖序号来阻止重放和再排序攻击。

通过警示来指示错误。警示是一种不同于主数据流的特殊类型的记录，使用它们来指示握手错误或计算 MAC 时发现的问题和相关描述。

使用 close_notify 警示来指示连接结束。这样就可以阻止攻击者在其下的传输层上伪造关闭而实施截断攻击。

4

高级 SSL

4.1 介绍

前一章讲述了简单类型的 SSL 连接，即使用 RSA 只对服务器进行认证的 SSL 连接。本章展示了其余的主要操作模式。跟前一章一样，本章也分成两个部分。第一部分概括性地描述各种模式，第二部分则对每条消息进行详尽地描述。大多数读者只完整地读完整第一部分而跳过第二部分，那些希望考验自己的读者还应当完整地读完第二部分。

这两部分内容均按照同样的顺序进行讲解。我们首先讨论会话恢复，它允许不用执行完整的握手过程就能形成一条新的 SSL 连接。接着再讲述客户端认证，它允许客户端使用其公用密钥和数字证书对自身进行认证。

由于美国出口法规限制从美国出口的应用所使用的密钥大小，所以 SSL 有两种设计，用来在保持可出口的情况下提供最高程度安全的模式：临时 RSA（ephemeral RSA）和服务器网关加密（Server Gated Cryptograph）。尽管这两种模式的存在已经不再是必需的了，但它们仍在广泛使用。我们将讲述这些模式都是如何工作的，以及它们所带来的好处。此外，还将讲述 TLS 标准的 DH/DSS 密钥交换，以及应用较少的使用椭圆曲线——Kerberos 和 FORTEZZA 的加密套件。最后详细讨论所有的 SSL 警示以及 SSLv2 的向后兼容性。

注意，尽管我们展示的消息流程里包含握手中的所有消息，但是我们将只对新的或与基本握手不同的消息进行详细地描述。

4.2 会话恢复

正如我们将在第 6 章见到的，整个 SSL 握手的开销可能非常巨大，无论从 CPU 时间还是执行所需要的往返次数上讲均是如此。为了减少这种性能开销，在 SSL 中集成进了一种会话恢复机制。如果客户端与服务器已经通信过一次，则它们就可以跳过整个握手阶段而直接进行数据传输。

握手中开销最昂贵的部分就是确立 `pre_master_secret`，它通常要求使用（除了使用 Kerberos 的情况）公用密钥加密。而经过恢复的握手允许新的连接使用上一次握手中确立的

`pre_master_secret`。这就避免了公用密钥加密所需的计算开销昂贵的操作。

会话与连接的差异

SSL 区分连接与会话。连接代表一种特定的通信通道（通常映射为 TCP 连接），以及密钥、加密选择和序号状态等内容。会话则是一种虚拟的结构，它代表磋商好的算法和 `pre_master_secret`。每次当给定的客户端与服务器经过完整的密钥交换并确立新的 `master_secret` 时就会创建一个会话。

一个给定的会话可以与多条连接关联。尽管给定会话中的所有连接均共享同一个 `master_secret`，但是每个连接又都有它自己的加密密钥，MAC 密钥和 IV。从安全上考虑，这是绝对必要的，因为重复使用对称密钥资料是极端危险的。恢复（resumption）允许根据共同的 `master_secret` 来产生一组新的对称密钥，因为这些密钥依赖于对每次连接来说都是全新的随机值，而新的随机值与原来的 `master_secret` 一起产生新的密钥。

如何工作

当客户端与服务器第一次进行交互时，它们创建一个新的连接和一个新的会话。如果服务器准备恢复会话的话，就会在 `ServerHello` 消息中给客户端一个 `session_id`，并将 `master_secret` 缓存起来供以后引用。当这个客户端初始化一条与服务器的新连接时，它就会在其 `ClientHello` 消息中使用 `session_id`。而服务器通过在其 `ServerHello` 中使用相同的 `session_id` 来同意恢复会话。此刻，就会跳过余下的握手部分，而使用保存的 `master_secret` 来产生所有的加密密钥。图 4.1 描述了该过程的流程。

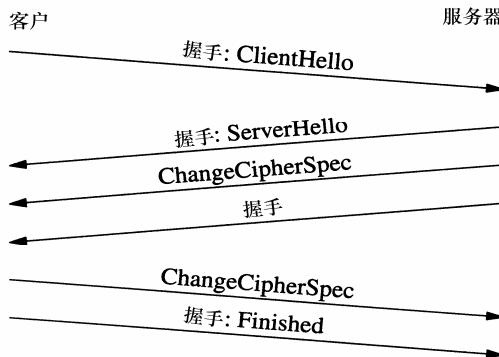


图 4.1 恢复 SSL 会话的流程

4.3 客户端认证

到目前为止，我们讨论的所有握手均是只对服务器进行认证。SSL 还提供了一种对客户端进行加密认证的机制。这在服务器想要限制只有某些授权客户端才能存取的服务时非常有用，它可以使用客户端认证来做到这一点。这里的理想就是客户端使用其私用密钥对某些内容进行签名，这样就证明它拥有与数字证书对应的私用密钥。客户端认证是通过服务器给客户端发送一条 `CertificateRequest` 消息来进行初始化的。而客户端通过发送一条 `Certificate` 消

息（与服务器用来传送其证书的消息一样）和一条 CertificateVerify 消息予以应答。CertificateVerify 消息是一个使用与其传输的证书所关联的私用密钥来签名的字符串。客户端认证总是由服务器进行初始化，不存在由客户端单方提出认证的机制。图 4.2 描述了这一过程的流程。

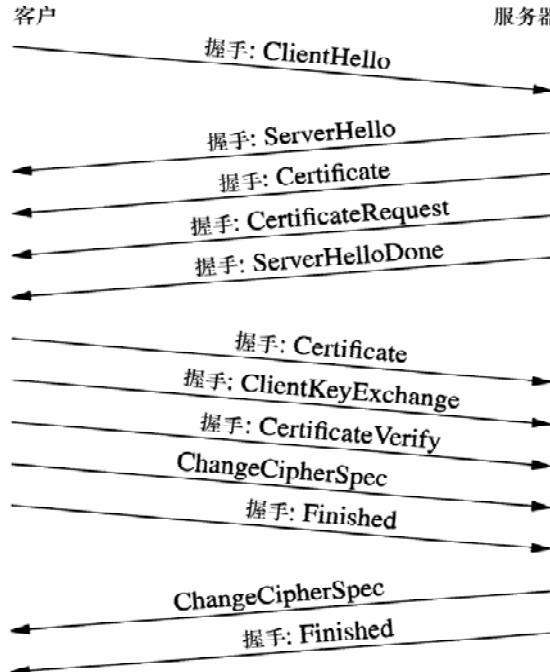


图 4.2 客户端认证的流程

4.4 临时 RSA

当设计 SSL 的时候，美国出口法规将可出口应用 RSA 加密密钥长度的限制为 512 位。不幸的是，512 位长的永久 RSA 密钥确实是攻击的活跃目标。因此，希望同时能够与国内以及可出口客户端进行通信的服务器就要具有两种密钥交换密钥，一种是 1024 位的而另一种是 512 位的，并假定尽可能使用高强度加密。

如果想以尽可能高的安全性同时与出口及国内客户端进行通信的话，SSLv2 就会迫使服务器具有两个经过认证的密钥，而 SSLv3 和 TLS 集成进了一种称做临时 RSA (ephemeral RSA) 的功能，它允许可出口客户端与国内服务器之间进行通信时使用一种永久性的高强度密钥。具体操作是，服务器产生一个 512 位的临时密钥，并使用其高强度密钥进行签名。当与国内客户端进行通信时，它简单地使用我们在第 3 章所讨论的高强度密钥。

握手中临时 RSA 模式与正常 RSA 模式之间的惟一区别就是是否出现一条新的消息：ServerKeyExchange。服务器使用 ServerKeyExchange 消息来传输经过签名的 RSA 密钥。当客户端收到 ServerKeyExchange 消息时，它就会验证临时密钥上的服务器签名并使用它来打包 pre_master_secret，这跟我们在第 3 章所描述的完全一样。图 4.3 描述了该过程的流程。

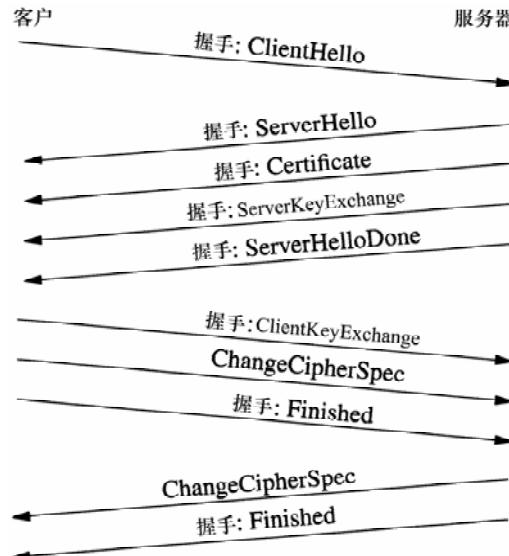


图 4.3 临时 RSA 的流程

4.5 再握手

一旦建立了 SSL 连接，就可能需要进行一次再握手（Rehandshake）。再握手只是在当前受保护的连接之上所进行的一次新的 SSL 握手，因此传输过程中的握手消息是经过加密的。一旦这次新的握手完成，就使用新的会话状态来保护数据。

```
struct{}HelloRequest;
```

客户端可以简单地通过发送一条 ClientHello 消息来初始化一次新的握手。如果服务器愿意初始化一次再握手的话，就必须发送空的 HelloRequest 握手消息。客户端使用 ClientHello 消息响应 HelloRequest 后，握手正常进行。没有让客户端或服务器磋商进行再握手的要求。不想进行再握手的实现可以简单地忽略这条消息或发送一条 no_renegotiation 警示。下一节描述了一个由客户端初始化再握手的例子。第 8 章描述一个由服务器初始化再握手的例子。

4.6 服务器网关加密

即便在放宽出口限制之前，美国出口法规也提供了一项允许特定金融交易使用高强度加密的例外。为了照顾到这一点，许多 SSL 实现都包括一种功能，允许低强度的客户端在检测出它所交谈的对象是特殊服务器时，升级使用高强度加密。这项功能有着各种各样的称呼，它被 Microsoft 称做服务器网关加密（Server Gated Cryptography，SGC），被 Netscape 称为 Step-Up。实际上 SGC 与 Step-Up 是 SSL 的不同变种。然而在此处，它们之间的差异无关紧要，所以我们在余下的这一节简单地将它们统称为 SGC。

SGC 依赖于服务器具有的、一种特殊的表明其能够从事与可出口客户端进行交互的高强度加密的证书。该证书以两种方式加以标注。首先它必须是由少数几个可信赖的 CA 颁发，

当前只有 Verisign 和 Thawte 为获准的颁发者。对于非 SGC 的情况，只要是让用户信任的 CA 即可。然而，在这种情况下，这些 CA 需要是美国政府所信赖的，不会给非金融服务器颁发具有 SGC 能力证书的那些机构。其次，这种证书中包含一个表明证书拥有者具有 SGC 能力（SGC capable）的扩展。

注意，这就意味着具有 SGC 能力的客户端实际上必须含有所有完成高强度加密的代码，而是否使用这些代码依赖于代码中的简单测试。实际上，由于削减密钥长度的阶段是在握手中发生的，所以即便是只出口的客户端也必须包含对称加密算法的高强度版本。因而在实际应用中，浏览器包含了所有高强度加密。在 Netscape 浏览器中，由一张编译进代码中的表来控制可用的算法。1998 年，发行了一种称做 Fortify 的程序，它就利用这一点，通过对二进制文件中的这张表作补丁从而打开可出口浏览器中的高强度加密。

SGC 是通过使用 SSL 的再握手功能来工作的。在第一次握手中，客户端只提供低强度的密码，一旦客户端见到了服务器的证书并验证 SGC 是适当的，它就会初始化第二次握手并在 ClientHello 中提供高强度的密码。第二次握手的消息是在当前经过保护的会话上进行传输的。图 4.4 描述了一次 Netscape 客户端完成 Step-Up 的流程。

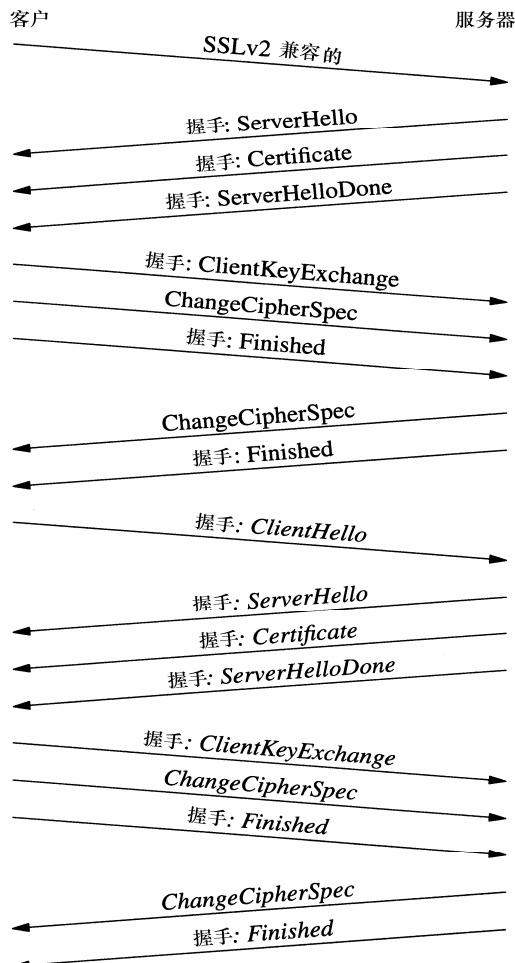


图 4.4 Step-Up 流程

4.7 DSS 与 DH

尽管 RSA 是目前 SSLv2 和 SSLv3 实现所支持的占主导地位的公用密钥算法，但 SSLv3 还支持许多基于其他算法的加密套件，特别是数字签名标准 (DSS) 和 Diffie-Hellman (DH)。正如我们在第二章中所讨论的，对这些加密算法的支持在 TLS 中是强制性的。这里的主要思想就是避免使用受专利保护的算法，尤其是 RSA。Diffie-Hellman 上的专利保护已于 1997 年到期。DSS 的专利情况目前还不清楚，RSA 数据安全公司声称 Schnorr 专利（美国专利 4995082）[Schnorr1991] 涵盖了 DSS，但是美国国家标准与技术委员会则另有说词。考虑到所有这些情况，IETF 决定全当不了解此事而批准使用 DH/DSS。自从 RSA 专利到期以来，这种动机就不那么迫切了，但仍然维持对 DH/DSS 的支持。

与 RSA 不同，DH 只能用于密钥磋商，DSS 只能用于数字签名，而 RSA 既可用于密钥交换又可用于签名。因此为了获得完整的解决方案，通常将 DH 与 DSS 放在一起使用。在这样一种系统中，通常也使用 DSS 对证书进行签名。

在 SSL 中存在两种主要的使用 DH 和 DSS 的方法。最常见的就是使用临时 DH(ephemeral DH) 密钥，它与临时 RSA 密钥相似。而在这种情况下，服务器产生一个临时的 DH 密钥并使用其 DSS 密钥进行签名，然后再将签名后的密钥放在 ServerKeyExchange 消息中进行传输。客户端使用 DH 密钥来完成密钥磋商，当然也可以使用长期 DH 密钥。在后一种情况下，服务器将具有一份包含其 DH 密钥的证书（用 DSS 进行签名），而客户端使用该证书中的密钥完成密钥磋商。

无论客户端如何获得服务器的 DH 密钥，密钥磋商过程都是一样的。客户端需要一个位于同一组中的 DH 密钥。正常情况下它不会有这样的密钥，在这种情况下就得产生一个，然后再将这个 DH 密钥放在 ClientKeyExchange 消息中进行传输。此刻，客户端与服务器每一方都具有它们自己的私用密钥和对方的公用密钥，于是就可以独立计算出 DH 的共享密码，并将这个共享密码用做 pre_master_secret。从此以后，余下的连接进程与 RSA 完全相同。图 4.5 描述了一次使用临时 DH/DSS 握手的流程。

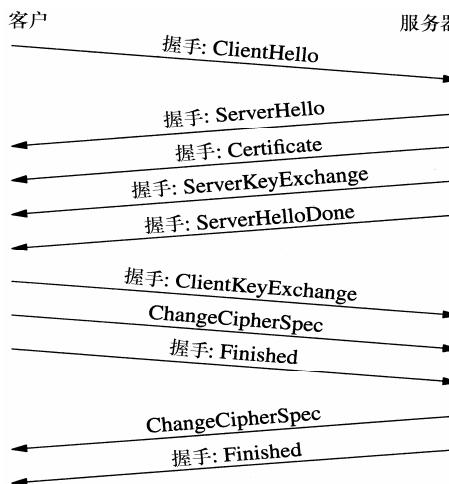


图 4.5 DSS/临时 DH 的流程

注意，DH/DSS 的加密套件号与 RSA 的不同，即便对称算法相同也是如此。而且静态 DH/DSS 的编号也与临时 DH/DSS 的不同。这样就使得当把这些加密算法指定为一个单一的加密套件时就会产生一种不如意的结果：无法保证对称算法支持与公用密钥算法支持是独立的（orthogonal）。例如，RFC2246 指定了五种可以与 RSA 密钥交换一起使用的对称加密算法（DES、3DES、RC2、RC4 和 IDEA），但只指定了两种与 DH/DSS 一起使用的算法。

造成这种局面的部分原因是由于工作组一方抵制使用诸如 RC4 和 IDEA 等专有算法，而另一部分则是出于简单地省略。为了给 DH/DSS 提供整套的对称加密算法，必须采取无可争辩的行动，所以 WG 只劳神对那些获得众多支持而且反对最少的加密算法进行标准化。

4.8 椭圆曲线加密套件

近来人们对 EC（椭圆曲线）加密算法有着相当的兴趣。粗略地讲，EC 加密算法将 DH 和 DSS 中的素数域替换为由一条椭圆曲线上的点所构成的域。在这样的群组中离散对数问题要更难一些，所以密钥就可以更短，而公用密钥操作也就更快。这对于像智能卡一类内存与处理能力都非常有限的设备来说尤其有用，但是即便是在快速的服务器上，常规公用密钥加密系统中的模幂运算的开销也相当高。意思就是让一种 EC 系统来取代 DH 和 DSS。

在编写这本书的时候，已经发行了一份有关增加几种基于 EC 加密套件的因特网草案，但还没有对此类加密套件进行标准化。对这样的加密算法进行标准化的一个主要障碍就是知识产权问题相当模糊。至少有 Certicom 和 Apple 同时拥有 EC 加密算法高效实现的重要专利，而且还存在对加密算法本身受到专利制约的担心。

4.9 Kerberos

Kerberos[Miller1987]是一种流行的在 MIT 开发的基于对称密钥的认证系统。总体思想就是存在一个由系统中所有实体信任的 TGS 也称做授予许可证服务器（中央服务器）。每个实体都与 TGS 共享一个对称密钥。当一个客户端想要与服务器进行通信时，它从 TGS 请求一个许可证。该许可证以目标实体的共享密钥来加密并包含请求方的认证信息，以及下一步两个实体之间进行通信时所使用的会话密钥。接着请求方会将许可证发送给目标实体。

RFC2712[Medvinsky1999]描述了如何与 TLS 一起使用 Kerberos。ClientKeyExchange 包含许可证和经过加密的 pre_master_secret。使用在许可证中找到的共享密钥来加密 pre_master_secret。服务器从许可证中取出共享密钥，并对 pre_master_secret 进行解密。从那一刻开始，握手就像往常一样进行下去。Kerberos 加密套件并没有经过广泛地部署。图 4.6 描述了 RFC2712 中定义的 Kerberos 加密套件。

加密套件	认证算方法	密钥交换算法	加密算法	摘要算法	编号
TLS_KRB5_WITH_DES_CBC_SHA	Kerberos	Kerberos	DES_CBC	SHA	0x001e
TLS_KRB5_WITH_3DES_EDE_CBC_SHA	Kerberos	Kerberos	3DES_DDE_CBC	SHA	0x001f
TLS_KRB5_WITH_RC4_128_SHA	Kerberos	Kerberos	RC4_128	SHA	0x0020
TLS_KRB5_WITH_IDEA_CBC_SHA	Kerberos	Kerberos	IDEA_CBC	SHA	0x0021
TLS_KRB5_WITH_3DES_EDE_CBC_MD5	Kerberos	Kerberos	3DES_EDE_CBC	MD5	0x0022
TLS_KRB5_WITH_DES_CBC_SHA	Kerberos	Kerberos	DES_CBC	SHA	0x0023
TLS_KRB5_WITH_RC4_128_MD5	Kerberos	Kerberos	RC4_128	MD5	0x0024
TLS_KRB5_WITH_IDEA_CBC_MD5	Kerberos	Kerberos	IDEA_CBC	MD5	0x0025
TLS_KRB5_EXPORT_WITH_DES_CBC_40_SHA	Kerberos	Kerberos	DES_40_CBC	SHA	0x0026
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_SHA	Kerberos	Kerberos	RC2_40_CBC	SHA	0x0027
TLS_KRB5_EXPORT_WITH_RC4_40_SHA	Kerberos	Kerberos	RC4_40	SHA	0x0028
TLS_KRB5_EXPORT_WITH_DES_CBC_40_MD5	Kerberos	Kerberos	DES_40_CBC	MD5	0x0029
TLS_KRB5_EXPORT_WITH_RC2_CBC_40_MD5	Kerberos	Kerberos	RC2_40_CBC	MD5	0x002A
TLS_KRB5_EXPORT_WITH_RC4_40_MD5	Kerberos	Kerberos	RC4_40	MD5	0x002B

图 4.6 Kerberos 加密套件

4.10 FORTEZZA

FORTEZZA 卡是一种美国政府设计的 PCMCIA 形式的加密令牌。它使用 DSA 和 SHA 来实现签名与消息摘要，但使用 NSA 设计的密钥磋商和加密算法。密钥磋商算法是一种称做密钥交换算法（KEA）的 Diffie-Hellman 变种，而加密算法是一种称做 SKIPJACK 的分组加密算法。

FORTEZZA 原先由 NSA 设计用来在提供高强度加密的同时允许 NSA 对该通信进行侦听。这些目标之间的矛盾通过在该设备中集成进一种密钥契约功能而得以解决。每张卡都有它自己的密钥，且由 NSA 以契约的形式掌握着。这样，当 NSA 想要对通信进行解密时，它就可以恢复用于加密通信的那张卡的密钥，但是加密的保密性足以挫败其他的攻击者。

FORTEZZA 加密套件要求对客户端进行认证，这可以以两种方式进行。客户端可以具有一个 KEA 证书，也可以具有一个 DSA 证书并使用其 DSA 密钥对 KEA 共享资料进行加密。在任何一种情况下，客户端都具有一个提供给服务器的证书。图 4.7 描述了一次 FORTEZZA 握手。

尽管服务器的 KEA 密钥位于其证书中，但服务器仍要发送一条 ServerKeyExchange 消息，用于传递用做 KEA 密钥磋商过程部分的随机值。同样，由于客户端必须经过认证，所以服务器还要发送一条 CertificateRequest 消息。然而，由于客户端认证是通过密钥磋商完成的，所以没有发送 CertificateVerify 消息。

在最初发行 FORTEZZA 卡的时候，SKIPJACK 和 KEA 都是被禁的。SSLv3 中规定了对 FORTEZZA 的支持，但是在 TLS 中却被删除，这是因为 IETF 不打算对没有经过充分论述的

算法进行标准化。Netscape 的确出品 FORTEZZA 使能的 Navigator 和 Netscape 服务器版本，但在美国政府之外从未见到过广泛的应用。而且将 FORTEZZA 与 SSL 一起使用引发了一些有趣的技术挑战，我们将在第 4.17 节详细讨论。

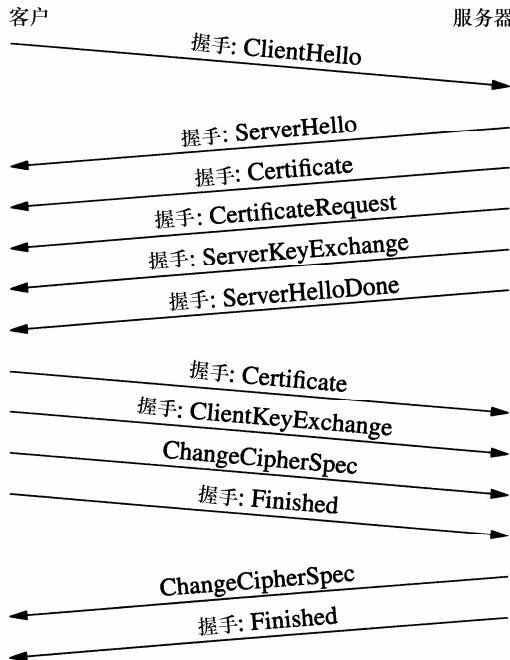


图 4.7 FORTEZZA 握手

4.11 小结

正如我们所看到的，除了只对服务器进行认证的基本模式之外，SSL 还支持其他具有特殊属性的模式，且这些模式都要求使用与 RSA 模式所要求的不同的握手。本章的下半部分将详细讨论这些模式，描述每条新消息的内容以及如何协同提供新的握手功能。在学习这些内容时需要记住的重要一点就是除了 FORTEZZA 以外，所有这些均是对握手进行简单修改的变种。不管你使用什么样的握手变种，其中利用 master_secret 的密钥导出及记录层的数据保护都是相同的。

4.12 会话恢复细节

会话恢复是通过 ClientHello 和 ServerHello 的 session_id 字段来实现的。在任何客户端与服务器执行初始握手期间，因为客户端没有缓存任何会话，所以 session_id 字段不存在。如果服务器打算允许恢复的话——服务器几乎总是愿意的——它就会产生一个 session_id，并在 ServerHello 的 session_id 字段中返回。在握手完成时，客户端与服务器均将会话信息（密钥

资料与磋商好的加密算法)保存在某个以后恢复会话时可以获得的地方。

当客户端想要恢复会话时,它就在ClientHello中使用合适的session_id。如果服务器愿意恢复这个会话,它就会在ServerHello中以相同的session_id予以响应,然后再跳过其余握手过程发送一条ChangeCipherSpec消息。使用存储的master_secret和新的客户端与服务器随机值重新计算key_block,这样所有的密钥均不相同。

图4.8描述了一次TLS会话恢复的详细握手消息。注意,ClientHello中的session_id字段与ServerHello中的session_id字段是匹配的。如果服务器不想恢复会话,则会简单地产生一个新的session_id并继续握手过程,就像没有企图进行恢复一样。发生这种情况的原因通常是服务器不再记得起客户端想要恢复的会话,这要么是因为它维护的缓冲大小有限,要么就是因为会话超时了。

在恢复会话时,要求客户端至少提供原先磋商好的加密套件。如果服务器同意恢复,它就必须选择该加密套件。在这种情况下,客户端提供一整套加密套件。如果服务器不选择恢复会话的话,它就可以从提供的加密套件中任意选择一种。

```
New TCP connection: romeo(1300)  <-> speedy(4433)
I 0.0022 (0.0022) C>S Handshake
ClientHello
resume [32]=
6d f3 02 8a 44 a7 42 94 14 7b 59 ad
8f 00 32 71 e9 a5 d1 bc 3f c0 23 0c
50 fa 4f 9f 27 cf 45 c4
cipher suites
    TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
    TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
    TLS_RSA_WITH_3DES_EDE_CBC_SHA
    TLS_RSA_WITH_IDEA_CBC_SHA
    TLS_RSA_WITH_RC4_128_SHA
    TLS_RSA_WITH_RC4_128_MD5
    TLS_DHE_RSA_WITH_DES_CBC_SHA
    TLS_DHE_DSS_WITH_DES_CBC_SHA
    TLS_RSA_WITH_DES_CBC_SHA
    TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
    TLS_DHE_DSA_EXPORT_WITH_DES40_CBC_SHA
    TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
    TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
    TLS_RSA_EXPORT_WITH_RC4_40_MD5
compression methods
NULL
2 0.0288 (0.0266) S>C Handshake
ServerHello
session_id[32] =
6d f3 02 8a 44 a7 42 94 14 7b 59 ad
8f 00 32 71 e9 a5 d1 bc 3f c0 23 0c
50 fa 4f 9f 27 cf 45 c4
cipherSuite          TLS_RSA_WITH_RC4_128_SHA
compressionMethod   NULL
```

```
3 0.0288 (0.0000) S>C ChangeCipherSpec
4 0.0288 (0.0000) S>C Handshake: Finished
5 0.0293 (0.0005) C>S ChangeCipherSpec
6 0.0293 (0.0000) C>S Handshake: Finished
```

图 4.8 恢复的 TLS 会话握手消息

会话 ID 的查找

服务器与客户端使用不同的信息来取得会话。当客户端初始化一条 SSL 连接时，惟一可用的信息就是服务器的主机名（在没有使用主机名的情况下为 IP 地址）和端口号，因此必须使用这些信息来查找会话 ID。然而，当服务器收到为某个客户端恢复会话的请求时，`session_id` 就会变成可用的，于是就可以用其来查找会话。既然查找是基于 `session_id` 的，所以即便与客户端进行连接的 IP 地址和端口与初始握手时所使用的不同也能够工作。

由于连续选择初始 TCP 端口的原因，客户端的端口号几乎总是与原先连接使用的不同。通常来说，对于每次连接，客户端的 IP 地址都是一样的。然而由于许多客户端的 IP 地址都是经 DHCP 或类似的机制动态分配的，所以一个客户端试图使用与原先创建会话时不同的 IP 地址来恢复会话也是极有可能的。服务器对此应当予以支持。

尽管允许使用不同 IP 地址的客户端恢复会话看起来存在危险，但实际上却是安全的。首先，攻击者伪造源于给定 IP 地址的消息非常容易，因此对 IP 地址进行检查补益甚微。更重要的是，服务器在握手终了时会检查客户端的 `Finished` 消息。为了产生正确的 `Finished` 消息，客户端不但必须能够存取新的加密与 MAC 密钥（为了加密和计算消息的 MAC 值），而且还要能够存取 `master_secret` 才能计算出 `Finished` 的散列值。由于一个掌握 `master_secret` 的攻击者可以轻易取代一个现有连接，因此这样是同等安全的。注意，为了减小给定 `master_secret` 被攻破的机率，服务器通常在一段时间（通常是一天）之后就会使会话过期。

4.13 客户端认证细节

正如我们所看到的，SSL 客户端认证使用了两条我们以前从未见过的新消息，以及一条旧消息的新套路。第一条新消息是 `CertificateRequest`，服务器使用它来请求客户端认证。客户端通过发送用于传递其证书的 `Certificate` 消息（我们以前见服务器使用过这条消息）以及对自身进行认证的 `CertificateVerify` 消息予以响应。

CertificateRequest

```
struct {
    ClientCertificateType certificate_types<1..2^8-1>;
    DistinguishedName certificateAuthorities<3..2^16-1>;
} CertificateRequest;
enum{
    rsa_sign(1),dss_sign(2),rsa_fixed_dh(3),dss_fixed_dh(4),
    (255)
}ClientCertificateType;

opaque DistinguishedName<1..2^16-1>;
```



```

CertificateRequest
  certificate_types          rsa_sign
  certificate_types          dss_sign
  certificate_authority
    C=AU
    ST=Queensland
    O=CryptSoft Pty Ltd
    CN=Server test cert (512 bit)
  certificate_authority
    C=US
    O=AT&T Bell Laboratories
    OU=Prototype Research CA
  certificate_authority
    C=US
    O=RSA Data Security, Inc.
    OU=Commercial Certification Authority
  certificate_authority
    C=US
    O=RSA Data Security, Inc.
    OU=Secure Server Certification Authority
  certificate_authority
    C=ZA
    ST=Western Cape
    L=Cape Town
    O=Thawte Consulting cc
    OU=Certification Services Division
    CN=Thawte Server CA
    Email=server-certs@thawte.com
  certificate_authority
    C=ZA
    ST=Western Cape
    L=Cape Town
    O=Thawte Consulting cc
    OU=Certification Services Division
    CN=Thawte Premium Server CA
    Email=premium-server@thawte.com

```

CertificateRequest 消息指示客户端应当进行客户端认证，并就服务器愿意接受的认证类型提供指导。允许服务器指定其应当使用的加密算法以及服务器所信任的对客户端认证进行签名的认证机构（authority）列表。

certificate_types 字段是由单字节值构成的数组，其中每个字节描述一种获准的签名算法。在这里，服务器指定了 **rsa_sign** 和 **dss_sign**，表示它准备接受以 RSA 或 DSS 签名的应答。注意，该值还指定了用以对证书进行签名的算法。

其他两个可能的值，**rsa_fixed_dh** 和 **dss_fixed_dh** 并不涉及签名，只表示客户端与服务器将使用固定的 Diffie-Hellman 密钥来磋商 **pre_master_secret**。如果选择了这些方法，客户端就不发送 **CertificateVerify** 消息，因为连接是由这种根据 **pre_master_secret** 来产生合适密钥的。

能力隐含地进行认证的。请参见第 4.16 节来了解有关固定 DH 的解释。

`certificateAuthorities` 字段指定包含认证机构 (CA) 的列表，服务器愿意接受这些机构签发的证书，使用 CA 标识名的 BER 编码版本来指定这些 CA。请参见第 1 章了解更多有关标识名与证书的知识。需要指出的重要一点是，如果使用证书链的话，`CertificateRequest` 中指定的 CA 名称就不必是签发客户证书的 CA，而可以是其上级 CA 中的一个。

● Certificate

```
struct{
    ASN.1Cert certificate_list<1..2^4-1>;
}Certificate;

opaque ASN.1Cert<2^24-1>;
```

我们以前见过 `Certificate` 消息，当时是服务器用来向客户端提供其证书的。而这里则由客户端发送用于进行客户端认证的证书。尽管证书是不一样的，但这条消息却具有与服务器的 `Certificate` 消息完全相同的结构。

● CertificateVerify

```
select (SignatureAlgorithm){
    case anonymous: struct{};
    case rsa:
        digitally-signed struct{
            opaque md5_hash[16]
            opaque sha_hash[20];
        };
    case dsa:
        digitally-signed struct{
            opaque sha_hash[20];
        };
}Signature;

struct{
    Signature signature;
}CertificateVerify;

CertificateVerify
Signature[64]=
5a a7 6a 48 54 17 c9 a3 09 87 90 37
42 17 45 c7 bf de 24 0f 4d 73 9b a8
69 a6 c7 f3 a0 41 c5 26 fc 6a 48 0d
e1 02 4f 9d 1b d2 17 55 dc 4e 55 11
6f 89 7c ed 68 59 70 db b4 e7 fe 11
d7 2f f6 83
```

`CertificateVerify` 完成所有实际的客户端认证工作。它包含了用客户端私用密钥进行签名的握手消息的摘要（该私用密钥与客户端 `Certificate` 消息中的证书对应），这就是数字签名操作的意思。因为只有与证书对应的私用密钥的持有者才能对消息进行签名，所以服务器可以断定证书所命名的实体就是正在连接的实体。

在 SSLv3 中，我们按照下面这样计算握手消息的散列值：

```
CertificateVerify.signature.md5_hash =
MD5(master_secret + pad2 +
MD5(handshake_messages + master_secret +
pad1));
```

```
CertificateVerify.signature.sha_hash =
SHA-1(master_secret + pad2 +
SHA-1(handshake_messages + master_secret +
pad1));
```

在消息中包括 master_secret 实际上并没有增加多少安全性，因为另一方的握手消息已经隐含了这种信息。TLS 使用一种简单的握手消息摘要。

```
CertificateVerify.signature.md5_hash = MD5(handshake_messages);
CertificateVerify.signature.sha_hash = SHA-1(handshake_messages);
```

签名计算的细节依赖于所使用的签名算法。在这里的握手中，客户端只使用 RSA 进行签名，因此签名对象就是到目前为止握手消息的 MD5 和 SHA-1 摘要的连结(concatenation)。由于摘要涵盖了服务器的随机值，所以服务器可以确信只要在 ServerHello 中使用一个新的随机值，签名就会是全新的（即不是重放的结果）。

需要指出的重要一点是，这种形式的 RSA 签名与通常在[RSA1993b]中所描述的 RSA 签名不同，后者只包含一个摘要。在 PKCS#1 中，通常只计算 DER 编码的 DigestInfo 结构上的签名，该结构也包含摘要的算法标识符。由于作为 SSL 协议的一部分，MD5 和 SHA-1 摘要都是强制性的，所以摘要算法标识符不是必需的（同时由于不存在 MD5 和 SHA-1 组合的标识符，所以这也是问题所在）。结果连结起来的散列值并没有在 DigestInfo 进行编码，而是直接进行了签名。但是对于硬件实现或多合一的 PKI 工具箱来说就会出现问题，它们经常将 DigestInfo 编码作为签名的一部分来完成。

失败的情况

如果客户端没有合适的证书，那么它就会发送一条不包含证书的 Certificate 消息。此时，服务器可以使用其他的认证方式（例如，在应用层提示输入口令），或干脆就不进行任何认证而继续连接工作。如果服务器不愿意继续下去的话，它就会发送一条致命的 handshake_failure 警示。

4.14 临时 RSA 的细节

```
struct{
    select(KeyExchangeAlgorithm){
        case diffie_hellman:
            ServerDHParams params;
            Signature signed_params;
        case rsa:
            ServerRSAParams params;
            Signature signed_params;
```

```
};

}ServerKeyExchange;

struct{
    opaque RSA_modulus<l..2^16-1>;
    opaque RSA_exponent<l..2^16-1 >;
}ServerRSAParams;

select(SignatureAlgorithm){
    case anonymous:struct{};
    case rsa:
        digitally-signed struct{
            opaque md5_hash[16];
            opaque sha_hash[20];
        };
    case dsa:
        digitally-signed struct{
            opaque sha_hash[20];
        };
}
}Signature;

enum{rsa,diffie_hellman}KeyExchangeAlgorithm;
enum{anonymous,rsa,dsa}SignatureAlgorithm;

ServerKeyExchange
params
    RSA_modulus[64]=
        c9 61 5e 58 08 6a 5c f0 76 a4 5d 6e
        ae 99 51 f9 ae c6 2c 8a e5 34 1e 9d
        a6 cb 68 b4 d6 47 b2 01 d4 5d 9b 32
        3c 49 2a 90 ac c2 41 cd b4 20 4f 54
        c2 a0 26 e4 b8 ee 69 61 04 23 64 72
        f2 40 e3 9f
    RSA_exponent[3]=
        01 00 01
    signature[128]=
        09 6d 34 ee 8c 2f 14 e5 52 05 26 25
        4c 34 72 35 ef 79 if 07 c1 71 82 cd
        3d f6 7c 67 05 ff eb 02 bc da bf 8e
        3a e7 c8 37 14 5c ca 05 4a 69 5f 09
        f3 b0 fa 53 33 50 ab 41 e8 63 6b f5
        5a 5d c1 5d f0 09 6d aa bd d8 0b e2
        bb 13 0d e6 09 f8 91 11 b1 ec 25 58
        1f 7e b3 56 81 4e c0 03 71 a4 95 e4
        be 1f 2e 10 75 80 75 1a 65 ed c4 70
        09 0b 19 64 3f dd f0 42 4a 32 73 36
        82 b1 2b 24 2c a4 6b 3a
```

ServerKeyExchange 消息包含一份签名版本的服务器临时公用密钥。由于这条消息也用于临时 DH，所以规范同时允许 RSA 和 DH 密钥。我们将会在第 4.16 节看到一个临时 DH

的例子。该消息以服务器的长期密钥来签名，这里就是 RSA 密钥。允许的算法为 RSA 签名、DSS 或 anonymous，匿名就意味着不对消息签名。注意，匿名模式没有提供任何应对主动攻击的安全保证。攻击者可以轻而易举地通过使用中间人攻击而伪装成客户端的服务器。

尽管规范允许调和使用密钥交换与签名算法的加密套件，但是常见的组合还是 RSA 密钥交换与 RSA 签名（就像本节开始所描述的那样），以及 DH 密钥交换与 DSS 签名（TLS 实现必须实现 DH/DSS）。虽然定义了使用 DH 密钥交换与 RSA 签名的加密套件，但是并不常用。甚至没有定义使用 RSA 密钥交换与 DSS 签名的加密套件。

RSA 密钥在消息中以一对代表 RSA 模与指数的字节字符串来表示。按照因特网的惯例，大整数以“网络”字节顺序映射为字节字符串，即头一个字节代表整数的最高有效字节。RSA 模的尺寸是一个重要的数字，它为 64 字节长，与 512 位的密钥相一致。正如我们在第 1 章所见到的，出于性能上的考虑，标准的做法是使用一个非常小的公用指数，而这里选用的是 65537。请参见第 1 章来了解更多有关 RSA 密钥的知识。

消息中展示的 RSA 签名是使用像 CertificateVerify 消息中那样的一对散列计算得出的，但是该散列的输入数据有所不同。ServerKeyExchange 消息选用的只有要进行签名的参数以及客户端与服务器的随机值。随机值用来阻止在新握手过程中重放前一次的 ServerKey Exchange 消息：

```
md5_hash =  
    MD5(client_random + server_random + ServerParams);  
  
sha_hash =  
    SHA(client_random + server_random + ServerParams);
```

由于产生 RSA 密钥的计算开销昂贵，所以 SSL 服务器实现通常使用同一个临时密钥来服务于大量客户端。这样不像为每个客户端产生一个全新的密钥那样安全，因为大量交易（常常是一天的量）都是以 512 位的密钥加以保护的。然而，为每个交易产生一个全新密钥的开销几乎总是无法容忍的。

使用临时 RSA 有两个条件：第一，使用的加密套件必须是出口加密套件中的一种（即密钥交换算法为 RSA_EXPORT）。第二，签名密钥必须不长于 512 位，否则就必须使用正常的 RSA。

在我们对临时 RSA 兴趣倍致之前，值得粗略计算一下它能够增加多少安全性。典型密钥的生命周期为一年或两年，而服务器每天至多要产生一个新的临时 RSA 密钥（尽管对于现代的 CPU 能力来说可以更频繁一些，就算每几分钟一次）经常仅仅在服务器启动的时候产生，而服务器可以一连几周不用重起。因此，如果你想要读取给定服务器的所有信息数据，对于典型的应用模式来说，临时 RSA 将会增加不到一千倍的工作量。另一方面，如果想要读取特定消息的话，且知道该条消息的发送时间，那么临时 RSA 根本就不会给你的工作增加任何难度。

4.15 SGC 的细节

SGC/Step-Up 所面临的困难就是客户端必须在知道服务器是一个普通 SSL 服务器还是有

SGC/Step-Up 能力的服务器之前发送它的 ClientHello。因此，它就必须提供只能出口的加密套件。否则，正常的服务器就有可能选择客户端无法使用的高强度加密套件。只有在客户端接收到服务器的 Certificate 消息时，它才能检测到服务器具有 SGC/Step-Up 能力。此刻，SGC 与 Step-Up 的不同之处就表现出来了。我们先来讨论 Step-Up。

Step-Up

服务器通过在 extendKeyUsage 扩展中提供一个特殊的值来指示客户端它愿意执行 Step-Up。当客户端看到这个值时，就知道可以 Step-Up 了。浏览器完成它正在执行的 SSL 握手工作（商定一种出口加密套件），然后立刻再次进行握手来商定一种高强度的加密套件。图 4.9 描述了一次 Step-Up 连接过程。

```
New TCP connection: romeo(4290) <-> romeo(443)
1 0.0004 (0.0004) C>S SSLv2 compatible client hello
    Version 3.0
    cipher suites
        SSL2_CK_RC4_EXPORT40
        SSL2_CK_RC2_EXPORT40
        TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
        TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
        TLS_RSA_EXPORT_WITH_RC4_40_MD5
        TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
2 0.0058 (0.0053) S>C      Handshake
    ServerHello
        session_id[32] =
            5e ae 37 80 c5 30 ff 35 59 65 2a 8c
            9b a1 8a 73 ad 2d 88 30 59 9a f9 58
            45 ac 8d a3 85 5f ca e9
        cipherSuite          TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
        compressionMethod    NULL
3 0.0058 (0.0000) S>C      Handshake
    Certificate
        Subject
            C=US
            O=RTFM, Inc.
            CN=romeo.rtfm.com
    Issuer
        C=US
        ST=California
        O=RTFM, Inc.
        CN=Step-Up CA
    Serial      03
    Extensions
        Extension: X509v3 Subject Alternative Name
            <EMPTY>
        Extension: X509v3 Basic Constraints
            CA:FALSE, pathlen:0
        Extension: Netscape Comment
            mod_ssl generated custom server certificate
        Extension: Netscape Cert Type
```



```

SSL Server
Extension: X509v3 Extended Key Usage
    Netscape Server Gated Crypto
4 0.0058 (0.0000) S>C Handshake
    ServerHelloDone
5 0.0248 (0.0189) C>S Handshake
    ClientKeyExchange
        EncryptedPreMasterSecret[128]=
            al 40 7c 99 2f 40 4d 01 dd b0 0a 7b
            f8 8e ee e3 1d f1 ed 35 04 ea 56 5f
            1a 3f 62 75 cf 6e bc b9 58 cf ad 33
            ba be 30 3c 63 d0 85 ea a7 a4 24 e2
            b5 dd d1 21 03 el 87 bd cb cf 56 54
            8d ed 02 f9 67 d7 bc 9a 73 cb 51 a6
            c2 d6 c2 12 b6 96 06 45 db a3 ed d8
            40 0c ea 11 22 09 61 7c 98 85 8f ed
            ca 4a 51 52 bd e9 14 0a b3 5d 04 be
            58 79 1e 51 cd fd dc ae 66 23 b8 4f
            1b ea 68 10 eb 8a e7 87
6 0.2199 (0.1951) C>S ChangeCipherSpec
7 0.2199 (0.0000) C>S Handshake
    Finished
8 0.2223 (0.0023) S>C ChangeCipherSpec
9 0.2223 (0.0000) S>C Handshake
    Finished
First handshake completed.
10 0.2231 (0.0008) C>S Handshake
    ClientHello
        Version 3.0
        cipher suites
            TLS_RSA_WITH_RC4_128_MD5
            valueunknown: 0xffeONetscapeproprietary cipher suite
            TLS_RSA_WITH_3DES_EDE_CBC_SHA
            TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
            TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
            TLS_RSA_EXPORT_WITH_RC4_40_MD5
            TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
        compression methods
            NULL
11 0.2271 (0.0039) S>C Handshake
    ServerHello
        session_id[32] =
            de 59 fc ef 7f b5 59 e0 92 f9 31 20
            da a3 f1 82 bf 78 ba 53 8b b6 8a a9
            d6 69 82 af 55 8b 99 27
        cipherSuite          TLS_RSA_WITH_RC4_128_MD5
        compressionMethod   NULL
12 0.2271 (0.0000) S>C Handshake
    Certificate
        Subject
            C=US

```

```
O=RTFM, Inc.  
CN=romeo.rfm.com  
Issuer  
C=US  
ST=California  
O=RTFM, Inc.  
CN=Step-Up CA  
Serial          03  
Extensions  
Extension: x509v3 Subject Alternative Name  
<EMPTY>  
Extension: X509v3 Basic Constraints  
CA: FALSE, pathlen:0  
Extension: Netscape Comment  
mod_ssl generated custom server certificate  
Extension: Netscape Cert Type  
SSL Server  
Extension: X509v3 Extended Key Usage  
Netscape Server Gated Crypto  
13 0.2271 (0.0000) S>C Handshake  
ServerHelloDone  
14 0.2381 (0.0110) C>S Handshake  
ClientKeyExchange  
EncryptedPreMasterSecret[128] =  
98 29 0c 5f 72 b0 46 03 fd 3d ed 62  
c6 fc ec d3 e3 73 d3 f5 c8 a0 60 0a  
f7 94 de 18 0a 9c 3c db db 7c f4 9c  
34 65 90 dd i1 fd a2 d3 ae 50 f1 d6  
e8 22 79 72 fe 99 b8 e1 03 a4 4a 64  
22 4d e8 a5 58 d5 80 22 d5 da d4 dc  
fc 42 a0 f3 f3 50 a6 56 4f f4 ae 69  
37 7b 9c 8c 53 c8 d7 7a ac 91 2d 7b  
48 8b 10 e5 b0 55 65 45 99 3e ca 69  
f7 c5 9e ae 0b fe f1 36 7b 4c ic 6d  
7e 4d 42 96 6b 21 55 d4  
15 0.4199 (0.1818) C>S ChangeCipherSpec  
16 0.4199 (0.0000) C>S Handshake  
Finished  
17 0.4215 (0.0015) S>C ChangeCipherSpec  
18 0.4215 (0.0000) S>C Handshake  
Finished
```

图 4.9 一次使用 Step-Up 的连接

图 4.9 展示了一个 Netscape 客户端与一个使用 mod_ssl 的 Apache 服务器之间的 Step-Up 连接。注意，在初始消息中，客户端发送了一条 SSLv2 兼容的只提供出口加密套件的 hello 消息。它必须在初始连接时完成这项工作，因为它还不知道服务器具有 Step-Up 能力。对于以后的连接来说，如果它记得服务器具有 Step-Up 证书的话，它就可以立即提供高强度的加密套件。

第 3 条记录中所描述的服务器证书包含 Step-Up 扩展（这里显示的是 Netscape Server

Gated Crypto)，客户端可凭此知道它能完成 Step-Up。然而，客户端与服务器已经商定了一组加密套件，因此这种信息并不是立即有用，因为 SSL 中没有办法让客户端在此刻改变自己的想法。客户端可以挂断连接并重新连接，但这并不是 Step-Up 工作的方式。客户端继续完成握手。

一旦握手完成，客户端就会在新建的加密通道上初始化一次再握手。图 4.9 的剩余部分就描述了再握手过程。在第 10 条记录中，客户端发送了一条新的 ClientHello。这次它不但提供了弱强度的还提供了高强度的加密套件，服务器自然就接受一个。从那一刻开始，握手就像通常一样完成了。

注意，客户端并不试图恢复会话，大多数出口加密套件使用临时 RSA，在这种情况下恢复是不可能的，因为我们需要使用服务器 1024 位的 RSA 密钥来确立一个新的 master_secret。然而，新（仍未标准化）的 EXPORT1024 加密套件使用长 RSA 密钥来完成密钥交换，因此从技术上讲，这种会话是可以恢复的。然而，客户端并不提供这种功能。因此就必须执行全新的 RSA 密钥交换。因此，Step-Up 握手（至少）是普通握手计算开销的两倍。

这里的跟踪信息也展示了再握手的功能。在连接过程中的任意时刻，SSL 实现都可以初始化新的握手。新握手构建于当前受保护的通道之上。这样，第二次握手就是完全加密的。我们能够读到的原因就是我们已经向 ssldump 提供了服务器的私用密钥，而它已自动替我们对数据进行解密。所有加密的数据都以定长 italic 字体来显示，用以表示我们已经对其进行了解密。在握手终了，所有通信都切换到新磋商的连接之上。

仔细观察图 4.9 就会发现服务器的证书是以作者的本地 CA 签名的。然而，我们先前说过，Step-Up 证书必须由特定的受信 CA 签署。作者的 CA 当然不是其中之一！我们利用了 Netscape 实现中的一项弱点。这些 CA 保存在一个数据库中，而能够颁发 Step-Up 证书与否只是数据库的一条属性而已。通过改变本地数据库中的属性，我们就能颁发 Step-Up 证书。虽然可以连接到合法的 Step-Up 服务器上来产生这些跟踪信息，但是那样的话我们就不会有服务器的私用密钥，也就不能对通信数据进行解密了。

SGC

在前一节我们看到 Step-Up 需要两个密钥交换阶段。这样不仅开销昂贵而且第一个阶段并没有真正增加多少安全性。Microsoft 注意到了这一点，于是 SGC 只使用一次密钥交换。图 4.10 描述了 IE 与 IIS 之间的 SGC 连接。

```
New TCP connection: swagger(1897) <-> 151(443)
1 0.3679 (0.3679) C>S SSLv2 compatible client hello
    Version 3.0
    cipher suites
        TLS_RSA_EXPORT_WITH_RC4_40_MD5
        TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
        TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
        SSL2_CK_RC4_EXPORT40
        SSL2_CK_RC2_EXPORT40
2 0.7508 (0.3829) S>C    Handshake
    ServerHello
    session_id[32] =
```

```
22 00 00 00 4b 66 29 fe 02 a7 f3 cc
56 38 a6 4e 41 ad d8 fd 55 08 75 5d
fe e3 41 02 fe 4e 27 62
cipherSuite           TLS_RSA_EXPORT_WITH_RC4_40_MD5
compressionMethod     NULL
Certificate
Subject
C=IT
ST=Milano
L=Assago
O=BANCO AMBROSIANO VENETO
OU=INNOVAZIONE TECNOLOGICA
OU=Terms of use at www.verisign.com
RPA (c)99
CN=HB.AMBRO.IT
Issuer
O=VeriSign Trust Network
OU=VeriSign, Inc.
OU=VeriSign International Server CA - Class 3
OU=www.verisign.com
CPS Incorp. by Ref. LIABILITY LTD. (c)97 VeriSign
Serial      69 5a a3 c0 d8 8e bf ac 6b 64 ca cc
ce ec 1b 59
Extensions
Extension: X509v3 Basic Constraints
CA:FALSE
Extension: 2.5.29.3
value omitted
Extension: Netscape Cert Type
SSL Server
Extension: X509v3 Extended Key Usage
Netscape Server Gated Crypto, Microsoft Server Gated Crypto
Extension: 2.16.840.1.113733.1.6.7
value omitted
Subject
O=VeriSign Trust Network
OU=VeriSign, Inc.
OU=VeriSign International Server CA - Class 3
OU=www.verisign.com
CPS Incorp. by Ref. LIABILITY LTD. (c)97 VeriSign
Issuer
C=US
O=VeriSign, Inc.
OU=Class 3 Public Primary Certification Authority
Serial      23 6c 97 1e 2b c6 0d 0b f9 74 60 de
f1 08 C3 c3
Extensions
Extension: X509v3 Basic Constraints
CA:TRUE, pathlen:0
Extension: X509v3 Key Usage
Certificate Sign, CRL Sign
```



```

Extension: Netscape Cert Type
    SSL CA, S/MIME CA
Extension: X509v3 Extended Key Usage
    2.16.840.1.113733.1.8.1, Netscape Server Gated Crypto
Extension: X509v3 Certificate Policies
    Policy: 2.16.840.1.113733.1.7.1.1
    CPS: https://www.verisign.com/CPS
    User Notice:
        Organization: VeriSign, Inc.
    Number: 1
    Explicit Text: VeriSign's Certification Practice Statement, ↴
        www.verisign.com/CPS, governs this certificate & is ↴
        incorporated by reference herein. SOME WARRANTIES DISCLAIMED ↴
        & LIABILITY LTD. (c)1997 VeriSign
ServerKeyExchange
ServerHelloDone

```

客户端中断第一次握手，然后在同一个连接上初始化一次新的握手并提供高强度加密套件。

```

3 0.7599 (0.0090) C>S      Handshake
    ClientHello
        Version 3.0
        cipher suites
            TLS_RSA_WITH_RC4_128_MD5
            TLS_RSA_WITH_RC4_128_SHA
            value unknown: 0x80 Microsoft proprietary cipher suite
            value unknown: 0x81 Microsoft proprietary cipher suite
            value unknown: 0x80 Microsoft proprietary cipher suite
            TLS_RSA_WITH_DES_CBC_SHA
            TLS_RSA_EXPORT_WITH_RC4_40_MD5
            TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
            TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
        compression methods
            NULL
4 1.1402 (0.3802) S>C      Handshake
    ServerHello
        session_id[32]=
            22 00 00 00 4b 66 29 fe 02 a7 f3 cc
            56 38 a6 4e 41 ad d8 fd 55 08 75 5d
            fe e3 41 02 fe 4e 27 62
        cipherSuite          TLS_RSA_WITH_RC4_128_MD5
        compressionMethod    NULL
    Certificate
        Subject
            C=IT
            ST=Milano
            L=Assago
            O=BANCO AMBROSIANO VENETO
            OU=INNOVAZIONE TECNOLOGICA
            OU=Terms of use at www.verisign.com

```

RPA (c)99
CN=HB.AMBRO.IT

Issuer
O=VeriSign Trust Network
OU=VeriSign, Inc.
OU=VeriSign International Server CA - Class 3
OU=www.verisign.com
CPS Incorp. by Ref. LIABILITY LTD.(c)97 VeriSign
Serial **69 5a a3 c0 d8 8e bf ac 6b 64 ca cc
ce ec 1b 59**

Extensions
Extension: X509v3 Basic Constraints
CA:FALSE
Extension: 2.5.29.3
value omitted
Extension: Netscape Cert Type
SSL Server
Extension: X509v3 Extended Key Usage
Netscape Server Gated Crypto, Microsoft Server Gated Crypto
Extension: 2.16.840.1.113733.1.6.7
Value omitted

Subject
O=VeriSign Trust Network
OU=VeriSign, Inc.
OU=VeriSign International Server CA - Class 3
OU=www.verisign.com
CPS Incorp. by Ref. LIABILITY LTD.(c)97 VeriSign

Issuer
C=US
O=VeriSign, Inc.
OU=Class 3 Public Primary Certification Authority
Serial **23 6c 97 1e 2b c6 0d 0b f9 74 60 de
f1 08 c3 c3**

Extensions
Extension: X509v3 Basic Constraints
CA:TRUE, pathlen:0
Extension: X509v3 Key Usage
Certificate Sign, CRL Sign
Extension: Netscape Cert Type
SSL CA, S/MIME CA
Extension: X509v3 Extended Key Usage
2.16.840.1.113733.1.8.1, Netscape Server Gated Crypto
Extension: X509v3 Certificate Policies
Policy: 2.16.840.1.113733.1.7.1.1
CPS: <https://www.verisign.com/CPS>
User Notice:
Organization: VeriSign, Inc.
Number: 1
Explicit Text: VeriSign's Certification Practice Statement,
www.verisign.com/CPS, governs this certificate & is
incorporated by reference herein. SOME WARRANTIES DISCLAIMED



```

& LIABILITY LTD. (c)1997 VeriSign
ServerHelloDone
5 1.1459 (0.0057) C>S Handshake
ClientKeyExchange
EncryptedPreMasterSecret[128]=
7d 4f fb ae fb 3e 5c 5b 9f 72 82 86
04 c7 7e 5e 72 e0 81 d3 8a 1a a0 10
97 36 d3 ab 0a 8b d8 e3 ba 83 1b 1c
8e 82 73 34 0a 4e 74 ec 0c 57 08 f7
ce 61 13 cf 8b c2 7b 5e 8a 14 29 28
94 5f 8a ff 93 42 59 93 12 ee 6d d4
d6 15 c2 90 c1 b1 df 2a 73 1e fb 11
bc e1 83 81 cd c7 4e ec 2c 1c e2 ba
14 ab fa 7c 80 b1 e5 8e 52 9b 55 bf
b7 84 b3 60 98 67 58 29 cd 50 ab 4d
2e 3f 21 09 4a 16 a0 57
6 1.1459 (0.0000) C>S ChangeCipherSpec
7 1.1459 (0.0000) C>S Handshake Finished
8 1.4361 (0.2901) S>C ChangeCipherSpec
9 1.4361 (0.0000) S>C Handshake Finished

```

图 4.10 使用 SGC 的连接

握手中的头两条记录多少与图 4.10 中的头两条相同。客户端提供出口加密套件而服务器提供表示它具有 SGC 能力的证书。(注意，尽管 IIS 在一条单一的消息中传输所有握手消息而不是像 OpenSSL 那样在分离的消息中传输，但这只是习惯问题，并不存在真正的这样或那样做的技术原因)。然而，IE 并不是执行完整的连接，而是立刻在第 3 条记录中用一条提供高强度加密套件的 ClientHello 进行响应。此刻，原来的握手终止了，而开始一次全新的握手。注意，IE 还提供了专有加密套件 0x0080 和 0x0081。它们最初源于 Microsoft Money 中的一个项目，它是 SGC 的前身[Banes2000]。

这看似一种聪明的技巧，但它还违反了 SSL 规范，该规范没有规定在握手当中发送 ClientHello 的条款。因此，为了保持互操作性。Microsoft 不得不使用一种不同的 Netscape 扩展以表示服务器不但允许使用 Step-Up，而且还能完成 SGC。注意，图 4.10 中的证书同时包含了 Netscape 和 Microsoft 的 SGC 扩展。因而在与这种服务器交互时，只有 Step-Up 能力的客户端就执行 Step-Up，而具有 SGC 能力的客户端将执行 SGC。OpenSSL 只在 OpenSSL0.9.5 中增加了对 SGC 的支持。

4.16 DH/DSS 的细节

```

struct{
    opaque DH_p<1 ..2^16-1>;
    opaque DH_g<1 ..2^16-1>;
    opaque DH_Ys<1..2^16-1 >;
} ServerDHParams;

```

```

ServerKeyExchange
params

```

```
DH_p[64] =
da 58 3c 16 d9 85 22 89 d0 e4 af 75
6f 4c ca 92 dd 4b e5 33 b8 04 fb 0f
ed 94 ef 9c 8a 44 03 ed 57 46 50 d3
69 99 db 29 d7 76 27 6b a2 d3 d4 12
e2 18 f4 dd 1e 08 4c f6 d8 00 3e 7c
47 74 e8 33
DH_g [1] =
02
DH_Ys [64] =
98 5a ee fc ce ac cf f1 05 cf 08 07
63 18 dd 50 53 66 a5 b8 0b 88 4d 7e
7d ea 11 3e 2a 99 63 e8 92 7a 56 cb
f1 36 74 97 36 4a f0 3e 4e 29 3e a2
e2 53 36 d8 9c a0 40 aa 8c fc eb c0
93 b6 c3 e8
signature[47]=
30 2d 02 14 78 bb 87 40 13 e4 8d e9
73 16 4e 0c dd 1c 9e a8 bd 58 99 a1
02 15 00 95 10 42 e1 cb b9 1d 26 34
d4 5f b1 0d b8 66 ba 8c 61 20 c3
```

临时 DH 与 DSS 在概念上与临时 RSA 非常相似，只是其消息的细节有所不同。

DH/DSS ServerKeyExchange 消息包含以 DSS 密钥签名的 Diffie-Hellman 参数。提供的参数有素数模 (p)、组 (g) 发生器以及服务器的公用密钥 (Y_s)。由于 DSS 需要 160 位的输入，所以仅在 sha_hash 值上计算 DSS 签名。大家记得 DSS 签名由两个分别称做 r 和 s 的大整数构成。这些内容在消息中的 SSL 表示是在 ANSI X9.57[ANSI1995]中定义的 ASN.1 结构的 DER 编码。

```
Dss-Sig-Value ::= SEQUENCE {
    r      INTEGER,
    s      INTEGER
}
```

注意：SSLv3 规范中关于如何对 DSS 签名进行编码的描述不够具体，Netscape 对它的诠释与其他大多数厂商不同。Netscape 没有对 r 和 s 进行 DER 编码，而只是将其连接起来放在一个 40 字节的字段中。这样，尽管 Netscape 为客户端认证实现了 DSS，但是它无法与其他实现进行交互。尽管就什么是“正确的”有着广泛的共识，但 Netscape 拒绝修改他们的实现，并声称它符合 SSLv3 规范及并引用了自己软件的装机量。TLS 规范澄清了这个问题，所有的 TLS 实现都必须对 DSS 签名进行 DER 编码。

ClientKeyExchange

```
struct{
    select(KeyExchangeAlgorithm){
        case rsa:EncryptedPreMasterSecret;
        case diffie_hellman:DiffieHellmanClientPublicValue;
    }exchange_keys;
} ClientKeyExchange;
```

```

struct{
    select(PublicValueEncoding){
        case implicit:struct{};
        case explicit:opaque DH_Yc<1..2^16-1>;
    }dh_public;
}DiffieHellmanClientPublicKey;

ClientKeyExchange
DiffieHellmanClientPublicKey[64]=
  14 ea e2 18 c1 69 b3 60 fc ea c7 54
  f7 18 db b9 47 c7 cf 95 80 2a 32 b7
  0c 07 11 ab 7a 9d dc 0a 1c 82 a1 35
  23 1f 90 71 2a 94 6d d8 86 b4 e2 84
  e9 a6 a2 00 5e bb 82 09 a3 8a ba f2
  e8 29 87 61

```

由于 Diffie-Hellman 要求客户端与服务器共享一组共同的参数 (ServerKeyExchange 中的 g 和 p)，所以客户端需要在 ClientKeyExchange 消息中发送其公用密钥 (Y_c)。对于大整数来说，按照通常的方式来表示 Y_c ，即高位字节在前。

● 固定 DH 密钥

因为客户端具有 DH 密钥，所以可以让客户端拥有一个包含将密钥与客户身份绑定在一起的证书。使用 `rsa_fixsed_dh` 和 `dss_fixsed_dh` 证书类型，可以用该证书来完成客户端认证。这些证书是包含分别以 RSA 和 DSS 签名的 DH 密钥的证书。客户端发送其证书以响应 `CertificateRequest`，但是并不发送 `CertificateVerify`，而是通过能够用私用密钥产生 `pre_master_secret` 的能力来证明其拥有相应的私用密钥。由于客户端的公用密钥位于 `Certificate` 消息中，所以在 `ClientKeyExchange` 消息中没有公用密钥 (`DiffieHellmanClientPublicKey` 结构的隐含选择)。

与之类似，也可以让服务器拥有一个包含在证书中的静态 DH 密钥。通过能够重新产生 `pre_master_secret` 来证明服务器的身份。这两种操作模式都不像使用 DSS 的临时 DH 模式那样常见。

4.17 FORTEZZA 的细节

[Freier1996] 中所描述的 FORTEZZA 支持并不能工作。显然编写时就没有进行测试。[Relyea1996] 中描述了成功操作所需的各种改动。

FORTEZZA 命令接口具有许多使其难以与 SSL 一起使用的属性，其中最重要的一个就是这种卡无法使用任意的密钥。因此，该密钥要么由卡上的随机数发生器产生，要么从另一张卡上接收。所以，不可能使用标准的 SSL 密钥产生过程来得到 FORTEZZA 加密密钥。

第二个重要限制就是卡必须产生 IV，这种要求是由契约特性所强加的。契约化的会话由某种称做法律强制存取字段 (LEAF) 的内容来承载。LEAF 包含用以加密实际通信数据的会话密钥，而会话密钥包裹在卡密钥中。为了确保 LEAF 与密文一起运输，它被嵌入在 IV 中，这就导致 24 字节的 IV。这种卡会自动从提供的 IV 中抽取真正的 8 字节 IV。

1997 年，NSA 屈于有关契约问题的压力将 LEAF 删除。然而，出于兼容性的考虑，24 字节的 IV 与一个哑 LEAF 并列放置。直到 1998 年 6 月 23 日 SKIPJACK 和 KEA 才被解密(declassified)。

```
struct{
    opaque r_s[128];
}ServerFortezzaParams;

struct{
    opaque y_c<0..128>;
    opaque r_c[128];
    opaque y_signature[40];
    opaque wrapped_client_write_key[12];
    opaque wrapped_server_write_key[12];
    opaque client_write_iv[24];
    opaque server_write_iv[24];
    opaque master_secret_iv[24];
    block-ciphered opaque encrypted_pre_master_secret[48];
} FortezzaKeys;
```

ServerFortezzaParams 结构用来运输提供给客户端的服务器随机值 `r_s`。FortezzaKeys 在 ClientKeyExchange 消息中运输，它包含服务器导出与客户端一样的密钥所需的一切信息。

如果 `y_c` 值没有在客户端的证书中，则包含在客户端的 KEA 公用密钥里。如果是后者，那么这个字段就是空的。`r_c` 值包含客户端的随机值，而 `y_signature` 包含客户端在 `y_c` 上的签名。规范并没有清楚地规定当客户端的 KEA 公用密钥位于其证书中时 `y_signature` 的值在 KEA 中，客户端与服务器的公用密钥和随机值一起产生用于加密其他密钥的令牌加密密钥 (TEK)。

`wrapped_client_write_key` 和 `wrapped_server_write_key` 包含以 KEA 产生的共享密钥进行包裹的加密密钥。`client_write_iv` 和 `server_write_iv` 包含这些密钥的 IV (其中嵌有 LEAF)。最终，FortezzaKeys 结构包含使用共享密钥以 `master_secret_iv` 中给定的 IV 来加密的 `pre_master_secret`，而 `pre_master_secret` 只用来产生 MAC 密钥。

FORTEZZA 命令接口还施加了另一种限制：IV 要由发送方产生并由接收方读取。这样，就可以在解密而不是加密的时候向卡上加载一个 IV。然而，密钥与 IV 都是由客户端产生的。结果就是由服务器加密的数据的第一个分组本质上使用的是随机的 IV——这一点客户端是无法知道的。为了同步 CBC 状态，服务器首先传递一个哑加密分组 (dummy encrypted block)，这个分组将会被客户端丢弃。从那一刻起，加密状态就是同步的了。

注意，对哑分组的加密意味着服务器完全没有必要使用 `server_write_iv`。[Relyea1996] 要求服务器实现应当加载解密模式的 IV 来验证 IV，然后再转变为加密模式。对于互操作性来说，没有要求这一步。其目的就是为了验证 IV 与密钥正确匹配，并以此来防止非正常客户端使用假 IV 绕过契约条款的限制。

不能使用随机密钥还意味着通常的会话恢复将无法工作，因为没有办法使用新产生的密钥。[Relea1996] 描述了两种选择：

- 根本就不进行会话恢复。
- 恢复但继续执行加密和解密，就像数据是前一会话的延续一样。注意，如果采用了

这种方法，就不可能在给定的会话上同时有两条连接，而正常情况下是可以的。这对于像 HTTPS 这样的系统就很成问题，该系统使用多个并发的连接。

[Relea1996]只定义了两种 FORTEZZA 加密套件，一种具有 SKIPJACK，而另一种具有 NULL 加密。图 4.11 描述这些内容。

加密套件	认证算法	密钥交换算法	加密算法	摘要算法	编号
SSL_FORTEZZA_DMS_WITH_NULL_SHA	DSA	KEA	NULL	SHA	0x001c
SSL_FORTEZZA_DMS_WITH_FORTEZZA_CBC_SHA	DSA	KEA	SKIPJACK	SHA	0x001d

图 4.11 FORTEZZA 加密套件

4.18 错误警示 (Error Alert)

第 3 章，我们从一般意义上讨论了有关警示的内容，但除了 `close_notify` 之外没有讨论具体的警示案例。其他的警示也都表示某种错误，本节详细讨论这些警示。

大家还记得警示可以有警告 (warning) 或致命 (fatal) 等级。fatal 警示总会终止连接，而实现在碰到 warning 警示时继续执行。虽然从理论上讲，实现可以在碰到非致命警示时继续执行，但是在实际应用中，即便是 warning 警示也表示一方请求了另一方无法满足的东西，因此通常也会导致连接关闭。

规范要求一些警示是致命的，而另一些要么是致命要么是警告性质的。一种收到应当是 fatal 的但却被不正确的标为 warning 的警示的实现仍然应当将其当作 fatal 对待。在实际应用中，由于许多实现都将所有警示当作 fatal 看待。除非愿意让另一方终止连接，否则实现就应当限制自己发送 warning 警示。

本节详细描述了每种 SSL 错误警示的知识。尽管其中的一些警示有可能表明一种正在进行中的攻击，但是如果你是在调试 SSL 实现的话，则极有可能表示软件中的错误。我们将会在合适的地方指明这些警示可能暗示的常见实现错误。阅读这一节的内容可以帮助你判定在实现收到或产生这些警示时发生的情况。请参见图 4.12 中所有警示列表。

unexpected_message

`unexpected_message` 警示宣告接收到了一条不恰当的消息。这条警示几乎总是表示其中一种实现不正确并产生了不恰当的消息或发送消息的顺序不正确。在调试一种 SSL 实现时会经常看到这条消息，但是绝不应该在工作系统中见到。作为一种特殊情况，没有 SGC 能力的服务器有可能给企图进行 SGC 连接的客户端发送 `unexpected_message`。必须总是以 fatal 级别来发送这条警示。

bad_record_mac

如果实现接收到一条具有不正确 MAC 的记录，那么它就会发送 `bad_record_mac` 警示。由于这有可能表示一种攻击，因此必须按照 fatal 来处理这条警示。

如果你在测试自己的实现时收到了一条 `bad_record_alert` 警示，那就极有可能表示你的 MAC 代码有错误。然而，如果这条警示是在响应 `Finished` 消息时发送的，那也有可能表示

密钥导出代码中存在错误。即便加密密钥也不正确，解密看上去也是成功的，但 MAC 是错误的。实际上，即便 MAC 密钥正确而加密密钥不对，这种错误也很有可能在 MAC 检查中出现（请参见 `decryption_error` 警示了解有关解密错误的讨论）。

警示名称	级别	SSL 版本	编号
close_notify	either	either	0
unexpected_message	fatal	either	10
bad_record_mac	fatal	either	20
decryption_failed	fatal	TLS	21
record_overflow	fatal	TLS	22
decompression_failure	fatal	either	30
handshake_failure	fatal	either	40
no_certificate	either	SSLv3	41
bad_certificate	either	either	42
unsupported_certificate	either	either	43
certificate_revoked	either	either	44
certificate_expired	either	either	45
certificate_unknown	either	either	46
illegal1_parameter	fatal	either	47
unknown_ca	fatal	TLS	48
access_denied	fatal	TLS	49
decode_error	fatal	TLS	50
decrypt_error	either	TLS	51
export_restriction	fatal	TLS	60
protocol_version	fatal	TLS	70
insufficient_security	fatal	TLS	71
internal_error	fatal	TLS	80
user_cancelled	fatal	TLS	90
no_renegotiation	warning	TLS	100

图 4.12 所有的 SSL/TLS 警示

最后，一些 SSLv3 实现使用 `bad_record_mac` 消息来指示验证 `Finished` 消息本身时的错误。在 TLS 中，使用 `decrypt_error` 警示来报告验证 `Finished` 的错误。因此，在调试 `bad_record_mac` 错误时，重要的是考虑除 MAC 计算以外的其他原因。

➊ `decryption_failed`

`decryption_failed` 报告无法对密文进行解密，这种错误必须是 `fatal` 的。该错误其实只与分组加密有关。RC4 解密不管解密密钥怎样都会成功，而结果却是错误的。

分组加密可能因多种方式而导致失败。首先，分组加密输入长度有可能不是分组长度的倍数，这显然是一种消息格式化错误。还有就是在对数据进行解密之后，由于不正确的填充

或使用错误的加密密钥，填充检查也会发现错误。

如果使用的是错误的加密密钥，那么填充字节实质上将会是随机的。然而表面上仍有可能是正确的，尤其在实现只检查长度字节，而不检查填充就把它们去掉时就是这样。（这种过程是可靠的，因为不管怎样数据还是受 MAC 的保护）注意，填充实际上不必与原来的填充一样，只要一致就行。

在这里，接收方收到的明文是不正确的，但只有在检查 MAC 的时候才能发现这一点，同时将会收到一个 `bad_record_mac` 警示。显然，由于 RC4 解密表面上看起来总是成功的，所以 RC4 密钥资料中的错误总是在 MAC 检查时才发现。

record_overflow

这条警示表示记录比 SSL 规范所允许的大，这条消息必须是 fatal 的。在实际应用中，这条消息几乎总是表示 SSL 实现中存在错误，因为如果是攻击的话就会在 MAC 检查时被发现。注意，Internet Explorer 的早期版本会不正确地产生长记录，因此在与 IE 交互的时候，看到并不可疑的实现产生这样的警示并不稀罕。这条警示对于 TLSv1 来说是新增的。

decompression_failure

这条警示表示无法对记录执行解压缩，它总是致命的（fatal）。然而由于几乎从不使用压缩，所以很少在实际应用中见到这种错误。

handshake_failure

`handshake_failure` 警示是 SSLv3 中表示握手出现问题的通用警示。TLS 引入了许多细化地表示究竟出了什么问题的警示。在 TLS 中，只有在无法就常用加密算法无法达成一致时才会使用 `handshake_failure` 警示。

在调试 `handshake_failure` 警示时，你需要查明哪个阶段的握手出现了问题，并据此来判定发生的情况。如果你在同时发送了一系列消息之后收到 `handshake_failure` 警示的话，则逐条发送消息并使用网络嗅探器检查每一条消息之后的警示会有所帮助。这条警示必须是致命的（fatal）。

no_certificate

`no_certificate` 警示是一种仅限于 SSLv3 的警示，发送它来表示“没有可用的证书”。在 TLS 中，它被 `certificate_unknown` 取代，这条消息本质上用于同样的目的。

bad_certificate

`bad_certificate` 警示表示证书损坏、签名不正确，或其他类似的错误。从理论上讲，这种警示可以是一种 warning。实现可以选择不管损坏的证书，但是作为警告来发送这样的警示。然而，在实际应用中，这种警示实际上总是致命的（fatal）。这样的话也同样适用于后面的四种警示。有趣的是，Internet Explorer 根本就没有使用这种警示，而是简单地将连接关闭。我们将会在第 8 章看到这样的一个例子。

unsupported_certificate

这条警示表示实现收到了一份不受支持的类型的证书。由于 SSL 只支持 X.509 证书，所以这条警示意味着使用了不支持的算法。由于用于对证书签名的算法并没有在加密套件中

指出，所以有可能存在使用 DSA 密钥对 RSA 密钥进行签名的证书。不支持 DSA 的实现就会发送 `unsupported_certificate` 错误。然而，在实际应用中，这样加以混合的证书类型几乎没有出现过，而 `unsupported_certificate` 错误则表示其中一种实现中的软件错误。

revoked_certificate

这条警示表示发送方收到了一份撤消的证书。现实中的大多数 SSL 实现当前都不支持撤消，因此这条警示几乎不会出现。

certificate_expired

这条警示表示某个证书过期了。实现不应当发送过期的证书，但是有时会出现配置错误，结果就会出现这种警示。

certificate_unknown

这条警示是 TLSv1 中引入的通用证书错误。任何不属于前面类型或下面所描述的 `unknown_ca` 类型的证书错误都会产生一条 `certificate_unknown`。SSLv3 实现会就那样的情况产生 `bad_certificate` 警示。

illegal_parameter

`illegal_parameter` 警示表示握手字段中的一个越界或与其他字段不一致。握手显然无法完成，因此这条错误是致命的（fatal）。

unknown_ca

当实现收到由它不认识的 CA 或无法找到其证书的 CA 签名的证书时就会发送 `unknown_ca` 警示。这条警示必须是致命的（fatal）。注意，这里的处理方式与处理其他证书错误不一样，其他错误可以作为警告发送。我对这种不一致一无所知。在我看来，最好是安静地接受证书或通过警示加以拒绝。注意这里的“安静”只是指 SSL 消息。实现有可能在告之用户或向日志设施发送错误消息之后接受损坏的证书。`unknown_ca` 消息是 TLS 新增的。

SSLv3 实现代之以发送 `no_certificate` 警示。

access_denied

这条警示意味着尽管接收到了一份有效的证书，但证书中的身份没有通过存取控制检查，于是连接会被拒绝，这条警示总是致命的（fatal）。我不知道有产生这种警示的实现。

decode_error

这条警示表示无法对消息进行解码，因为其中的一个字段越界或长度不正确。在大多数情况下，这表示某种实现错误。这条警示总是致命的（fatal）。`decode_error` 是在 TLS 中引入的，还不清楚 SSLv3 中使用什么。OpenSSL 与 PureTLS 均使用 `handshake_failure`，从理论上推测大多数此种错误都在握手时发生。

decrypt_error

`decrypt_error` 警示表示其中的一个握手加密操作失败。可能的情况有无法解密 ClientKeyExchange、验证签名或检查 Finished 消息。这条警示总是致命的（fatal）。`decrypt_error` 是在 TLSv1 中引入的，而 SSLv3 实现通常使用 `handshake_failure` 来代替。注意，如果你在响应 Finished 时收到了一条 `decrypt_error`，那么就意味着你的加密或 MAC 密钥是正确的，

但 Finished 计算有误。

export_restriction

这条警示表示其中一种实现试图违反出口限制。RFC2246 中提供的例子就是试图在 RSA_EXPORT 密钥交换中使用 1024 位的临时密钥。这是 TLSv1 中新增的警示，并不多见。注意，这并不是当只支持高强度加密的实现试图与只支持低强度的实现交互时才会见到的警示。那种情况下会是 insufficient_security 或 handshake_failure。该警示总是致命的 (fatal)。

protocol_version

protocol_version 警示由服务器发送，那么表示客户端使用了无法识别的协议版本。由于客户端只能使用它所支持的最高版本，所以服务器只有在客户端请求比其支持的更老的版本时才会发送这个警示。例如，如果客户端只提供 SSLv3，而服务器只提供 TLS，那么服务器就会产生一条 protocol_version 警示。由于本质上所有的 TLS 实现也都是 SSLv3 实现，所以只有在一方被显式的配置为只支持 TLS 或软件中存在错误时才能见到这种警示。配置为只支持 TLS 没有任何意义。

注意，如果服务器只支持比客户端提供的更老的版本的话，它就不会发送 protocol_version 警示。相反，它将简单地在 ServerHello 中回应自己的版本号。如果客户端不想支持服务器的版本选择，它就得发送一条警示（一般是 protocol_version，但是 SSLv3 和 TLS 规范并没有规定）。protocol_version 警示总是致命的 (fatal)，它是在 TLS 中被加入的，而 SSLv3 实现通常使用 handshake_failure。

insufficient_security

insufficient_security 警示用来表示客户端所提供的加密套件比服务器所要求的弱。这条警示是 TLS 新增的，SSLv3 实现使用 handshake_failure。由于 insufficient_security 用于警示磋商加密套件失败，所以它总是致命的 (fatal)。

internal_error

internal_error 警示用来表示传输实现遇到了某种诸如内存分配或硬件错误的内部错误。这条警示是 TLS 新增的，因此 SSLv3 实现需要使用某种其他的错误。internal_error 警示总是致命的 (fatal)。

use_cancelled

当用户出于某种原因取消握手时就会使用 user_cancelled 警示。这条警示是 TLS 新增的。RFC2246 说它应当属于警告级别的警示，这有些令人迷惑。如果用户取消握手的话，连接就不会成功，所以这条警示应当是致命的 (fatal)。

no_renegotiation

正如我们将要在第 4 章见到的，SSL 允许客户端或服务器在任何时刻初始化新的握手，而不止是在握手期间。客户端通过发送一条新的 ClientHello 来初始化再磋商 (renegotiation)，而服务器通过发送 HelloRequest 消息来初始化再磋商。RFC2246 中说客户端可以安静地拒绝再磋商或发送 no_negotiation 警示。RFC2246 没有明确规定服务器应当如何响应不需要的 ClientHello，但推測也是发送一条 no_negotiation 警示。

no_negotiation 警示是 TLS 新增的，SSLv3 实现只是简单地忽略不需要的再磋商请求。由于现有的大多数 TLS 实现实际上都是升级后的 SSLv3 实现，所以它们通常安静地忽略再磋商而不是发送 no_negotiation 警示。在任何情况下，no_negotiation 警示都是警告性质的。

4.19 SSLv2 的向后兼容性

当发明 SSLv3 的时候，SSLv2 已经广泛部署。因此 SSLv3 的要求之一就是让 SSLv3 客户端和服务器自动与 SSLv2 的客户端和服务器进行交互。然而，消息格式是完全不同的（请参见附录 B 有关 SSLv2 的讨论）。服务器可以直接了当地自动检测出与之进行交互的是 SSLv3 还是 SSLv2 客户端。SSLv2 与 SSLv3 的长度字节位置不同，当作为 SSLv2 握手消息进行解释时，SSLv3 记录看上去出奇的长。因此，服务器可以等待客户端的第一条消息并进行适当地响应。

然而，客户端必须先说话，因而也就无法进行自动检测。所以，SSLv3 规范包含支持以一种与 SSLv2 向后兼容的格式发送 ClientHello。本质上，向后兼容的 ClientHello 将所有的 SSLv3 字段映射成 SSLv2 CLIENT-HELLO 中的 SSLv2 字段。因此，当服务器收到向后兼容的 CLIENT-HELLO 时，它要么将其解释为一条 SSLv2 消息并继续完成 SSLv2 握手，要么将字段转换为 SSLv3 的值并完成 SSLv3 握手。图 4.13 描述了 SSLv2 CLIENT-HELLO。

```
char MSG-CLIENT-HELLO
char CLIENT-VERSION-MSB
char CLIENT-VERSION-LSB
char CIPHER-SPECS-LENGTH-MSB
char CIPHER-SPECS-LENGTH-LSB
char SESSION-ID-LENGTH-MSB
char SESSION-ID-LENGTH-LSB
char CHALLENGE-LENGTH-MSB
char CHALLENGE-LENGTH-LSB
char CIPHER-SPECS-DATA[(MSB<<8)ILSB]
char SESSION-ID-DATA[(MSB<<8)ILSB]
char CHALLENGE-DATA[(MSB<<8)ILSB]
```

图 4.13 SSLv2 CLIENT-HELLO

CLIENT-HELLO 是以[Hickman1995]的 SSLv2 规范语言来描述的。与 SSLv3 不同，SSLv2 将所有的对象长度都放在消息中的某个部分，而将所有对象都放在另一部分中。而且长度的高位与低位字节单独表示。因此，通过像下面这样来计算 CIPHER-SPECS-DATA 字段的长度。

(CIPHER-SPECS-LENGTH-MSB<<8)| CIPHER-SPECS-LENGTH-LSB

所有的 SSLv3 握手参数都能够映射成 SSLv2 值。

● 版本 (Version)

SSLv2 的版本号为 2 (MSB=0,LSB=2)。为了表示 SLv3 或 TLS 兼容性，我们分别使用 SSLv3 或 TLS 版本字段中的 0x300 和 0x301 来表示 SSLv3 和 TLS。

随机数 (Random)

SSLv2 的 CHALLENGE 值用做 SSLv2 密钥产生过程的一部分，它在很大程度上与 SSLv3 所使用 ClientRandom 的方式相同。它是随机产生的，因此可以用做 ClientRandom 值。然而允许它是 16 和 32 字节之间的值，而 SSLv3/TLS 的 ClientRandom 则必须是 32 字节的。我们通过在其左侧填充 0 来将 CHALLENGE 转换成 ClientRandom 值。

会话 ID

由于在恢复 SSLv3 会话的时候，必须要使用 SSLv3 的 CipherHello，所以 SESSION-ID 字段应当总是空的且长度为 0。

加密套件

SSLv2 的 CIPHER-SPECS 与 SSLv3 的 CipherSuites 相似。它们指定加密和摘要算法。（SSLv2 只支持 RSA，因此它们不执行密钥交换和签名）因此，在向后兼容模式下，CLIENT-HELLO 应当同时包含客户端所支持的 SSLv2 和 SSLv3/TLS 加密算法。SSLv2 的 CIPHER-SPEC 值为 3 个字节，因此表示两字节的 SSLv3 值时会带有一个为 0 的首字节。

当使用向后兼容握手的时候还可以使用 SSLv2 加密算法。其中的一些加密算法与 SSLv3/TLS 的加密算法对应，但是有一些不是这样。例如，SSLv2 有一种 RC2-128 模式。SSLv3 没有定义这样的加密套件。然而，如果两个 SSLv3 客户端使用 SSLv2 兼容握手，那么就可以磋商使用它。所有这些加密套件都使用 RSA 完成签名和密钥交换。图 4.11 列出了这些加密算法。

加密套件	加密算法	摘要算法	编号
SSL_CK_RC4_128_WITH_MD5	RC4_128	MD5	0x010080
SSL_CK_RC4_128_EXPORT40_WITH_MD5	RC4_40	MD5	0x020080
SSL_CK_RC2_128_CBC_WITH_MD5	RC4_128_CBC	MD5	0x030080
SSL_CK_RC2_128_CBC_EXPORT40_WITH_MD5	RC4_40_CBC	MD5	0x040080
SSL_CK_IDEA_128_CBC_WTTH_MD5	IDEA_CBC	MD5	0x050080
SSL_CK_DES_64_WTTH_MD5	DES_CBC	MD5	0x060080
SSL_CK_DES_192_EDE3_CBC_WITH_MD5	DES_EDE3_CBC	MD5	0x070080

图 4.14 SSLv2 兼容的加密套件

回滚保护

SSL 采用 PKCS#1 填充，用 RSA 来对数据进行加密。然而为了防止攻击者采用主动攻击，强行磋商使用 SSLv2，当使用 SSLv2 的时候，具有 SSLv2 和 TLS 能力的客户端实现必须使用值 03 作为后 8 个字节的填充。服务器通过检查这个值来确信它的确是与只具有 SSLv2 的客户端进行交互（这种客户端会用随机数来填充这 8 个字节）。注意，这种不寻常的填充在执行 SSLv3 握手时不会出现而且不必进行检查。所有这些工作的目的都是为了阻止攻击者迫使客户/服务器从 SSLv3/TLS 降级使用（强度更弱的）SSLv2。

兼容性

为了让向后兼容握手正确地工作，只具有 SSLv2 功能的服务器必须在 CLIENT-HELLO

中收到更高版本的信息时简单地商定使用 SSLv2。不幸的是，SSLv2 规范没有清楚地规定服务器应当如何处理版本号比其支持的更高的 CLIENT-HELLO 消息。更糟糕的是，Netscape 的 SSLREF 参考实现简单地拒绝具有更高版本号的连接。因此，无法保证所有的 SSLv2 服务器都会正确响应向后兼容的握手，尽管绝大多数都是可以的。

4.20 总 结

本章完成了对 SSL 的详细描述。与我们在第 3 章所讨论的那种传统的仅限于对服务器进行认证的 RSA 模式相比，这里的每一种模式都提供了某种程度的优势。

会话恢复允许客户/服务器对在连接之间重复使用密钥资料，这使得它们之间的后续握手更为快捷。

客户端认证使用客户端的公用密钥向服务器证明其身份。SSL 连接传统使用匿名的客户端，通过使用客户端认证，客户端也可以像服务器一样使用证书进行认证。临时 RSA 允许服务器有一个长的（大于 512 字节）RSA 密钥，但通过产生一个临时的 512 位密钥，并使用长期密钥进行签名仍然能够与需要 512 位 RSA 密钥的可出口客户端进行通信。

服务器网关加密允许可出口客户端使用高强度加密与特别设计的服务器进行交互。这些服务器必须具有表示它们拥有 SGC 资格的特别证书。

DSS 与 DH 加密套件使得在美国实现不侵犯 RSA 专利的 SSL 成为可能。TLS 中强制对 DSS 和 DH 提供支持。

椭圆曲线加密算法比 DH/DSS 或 RSA 都更为迅捷，但 IETF 还没有将其作为 TLS 的一部分加以标准化。

Kerberos 密钥可以用来在客户端与服务器之间交换密钥资料，并保持剩余的 SSL 部分不变。这使得在对 SSL 连接进行认证时引入 Kerberos 设施成为可能。

SSLv3 与 SSLv2 向后兼容。SSLv3 的 ClientHello 可以映射为 SSLv2 的 CLIENT-HELLO，于是就允许使用 SSLv2 的 SSLv3 代理与只有 SSLv2 功能的代理进行通信。

5

SSL 的安全

5.1 介绍

这一章讨论 SSL 的安全属性。本章分为三个部分。第一部分提供安全使用 SSL 的一般规则。理解这些内容需要对密码术（cryptography）以及前两章的头半部分有相当地理解。这里的主要内容就是解释 SSL 的安全属性以及如何安全地使用它。

第二部分更具体地讨论了第一部分所讨论的安全威胁。如果在编写系统的时候不注意，就很容易做出具有互操作性但却隐藏有安全缺陷的实现。第一部分提供一般性的指导，而这一部分描述如果没有遵循这些指导而可能遭受的攻击，并对各种常用的实现技术进行了讨论。

最后一部分详细讲述了一些更为奇异的安全威胁。尽管不存在十分有效的针对 SSL 本身的攻击，但是许多 SSL 实现却遭受过成功的攻击，因此为了确保实现不受这些威胁的影响，要求细心留意某些方面。这一部分的内容技术性相当强，但是只阅读本章前面的三分之二就能让你对 SSL 的安全属性有一个良好地理解。

5.2 SSL 都提供了什么

SSL 提供通道级别的安全，意思就是说连接的两端知道所传输的数据是保密的，而且没有被篡改。几乎总是要对服务器进行认证，这样客户端才能知道连接的另一端是谁。也可以要求客户端使用其证书来对客户端进行认证。SSL 还提供了针对例外情况的安全通知功能，其中包括错误警示和连接关闭。

所有这些保护都依赖于某些对系统的假定：我们假定已经正确产生了密钥资料而且被安全地保管着。若没有谨慎地遵循某些步骤就会使安全大打折扣。

5.3 保护 master_secret

几乎协议的所有安全都依赖于 master_secret 的保密。如果某个会话的 master_secret 被攻破的话，这个会话就会完全暴露于攻击之下。由于用以保护数据的所有加密密钥都是通过

master_secret 来产生的，所以了解 master_secret 的攻击者就能够实施大量的攻击。

第 5.9 节讨论了泄露 master_secret 所造成的具体后果，不过简言之就是，如果攻击者拥有 master_secret，那么还是以明文方式来传输数据得了。这样的攻击者能够读取所有的消息数据，并能够在大多数情况下伪造任何一方的通信数据而不会被检测出来。这可不行。

5.4 保护服务器的私用密钥

一种可以让攻击者获得 master_secret 的显而易见的方法就是取得服务器的私用密钥。为了阻止这种情况的发生，我们必须做好服务器私用密钥的保密工作（奇怪吧！），这听上去似乎不言自明，但是这是一条最常违反的规则，而且要想真正做到这一点也是令人吃惊得难。

总的来说，拥有服务器私用密钥的攻击者会对系统安全造成严重威胁。如果服务器使用我们在第 3 章所讨论的普通静态 RSA 模式，那么拿到服务器私用密钥的攻击者就能恢复 master_secret，读取他所能捕获的任何通信数据而不会被检测出来，而且还能随意冒充服务器。如果服务器使用临时模式的话，攻击者就无法被动地读取通信数据，但却能够实施冒充服务器的主动攻击。不管是哪种情况，攻破服务器的私用密钥都是非常危险的。

保护私用密钥要求安全地存储私用密钥。大多数实现都对磁盘上的私用密钥进行加密，而且要提供口令（passphrase）才能对其进行解密。这要求用户/管理员产生并记住一条高强度的口令，而事实上要想做到这一点却是异常困难的。其他实现在受保护的硬件中存储密钥。这两种方案在启动服务器时都要求管理员的介入，从而使得在遭遇系统崩溃或电力故障时无法实现无人看管的重新启动。

5.5 使用良好的随机性

几乎所有的加密协议都要求使用高强度的随机数才能安全的运转，SSL 也不例外。RNG（随机数发生器）主要用来产生密钥，但也可以用于其他加密操作中的关键环节。如果攻击者能够预测出你的 RNG 将要输出的数字，那么就能够猜出你的密钥并彻底攻破协议。

SSL 在许多地方使用随机数。首先，服务器的私用密钥（以及可选的客户端的私用密钥）需要是随机产生的。其次，客户端需要产生随机数据来完成密钥交换。如果使用的是 RSA，那么客户端就必须生成 pre_master_secret。如果使用的是 DH，那么就要产生一个临时私用密钥。第三，如果使用 DSA 进行签名，那么就必须为每个签名产生一个随机数。最后，客户端与服务器都需要产生握手随机值，但是实际上那些随机值是无需安全产生的，因为它们是公开的。

因此，客户端与服务器都要能够产生高强度的随机数。如果服务器产生了弱强度的随机数，那么攻击者就能猜测出服务器的私用密钥，从而导致 pre_master_secret 的泄露。如果客户端产生了弱强度的随机数，那么攻击者要么能够直接猜测出 pre_master_secret（如果是在 RSA 模式下），要么能够猜测出客户端的临时 DH 私用密钥，继而猜出 pre_master_secret（如果是在 DH 模式下）。对于上述的任何一种情况，协议都会被彻底攻破。

大多数系统使用基于 PRNG（伪随机数发生器）的软件。PRNG 就是一种提供不可预测

(尽管不是随机的) 数字序列的算法。为了产生高质量的随机数, 这些 PRNG 必须以一些随机数据为种子 (seed)。种子数据通常是通过网络通信采样、系统内部变量, 或用户输入计时等内容来搜集的, 同时也存在基于物理随机性的硬件随机数发生器。请参见第 5.12 节来了解有关产生高强度随机性的指导。

5.6 检查证书链

让我们假定你的 SSL 实现现在工作正常, 而你从服务器收到了一份证书。因为该证书由一个你不认识的认证机构 (CA) 签名, 所以它无法提供任何有关服务器身份的可靠证明。尽管服务器通过证书声明了自己的身份, 但是那只不过是份证书而已, 因为你无法验证这种断言是正确的。你仍然可以建立到服务器的安全连接, 但却易于遭受一种主动攻击 (中间人攻击), 其中攻击者对你冒充成服务器, 对服务器冒充成你。要想避免这种攻击, 知道服务器的身份是绝对必要的。

没有证书的服务器显然是未经证实的, 但是存在更多微妙的方式, 通过这些的方式, 一个看似良好的证书链提供的安全性却达不到你所期望的程度。SSL 实现则有助于防止遭受这些攻击, 但是拥有完备的安全通常还要有应用的介入。

最重要的应对措施就是检查证书中提供的身份与应用期望的身份是否匹配。设想你正试图连接到一台由 Bob 操作的服务器, 而服务器向你提供了一份包含 Alice 名字的证书。证书是有效的, 但却是不正确的, 所以需要与期望连接的另一端的身份对照来检查证书中的身份。

对身份的正确检查要求与应用层协议进行交互, 暗指要接受用户的确认, 只有实际用户才掌握与期望的用户身份有关的所有信息。另一种需要通知用户的情况就是过期证书的问题。在某些情况下, 单位或许有禁止使用过期证书的政策, 但是对于其他一些情况则全靠用户来做出这样的决定。我们将在第 7、9、10 章详细讨论证书链与应用层协议的交互情况。

5.7 算法的选择

SSL 支持各种各样的加密套件, 这些加密套件指定一组供连接使用的算法。这些算法的强度从非常弱的可出口型 (如 40 位模式的 RC4) 到强度非常之高的 (希望是这样) 如 3DES。此外, 非对称密钥对可以使用一定范围长度的密钥。SSL 连接的安全自然而然就全部有赖于它所使用的加密算法的安全, 因此应选择与你的数据价值相当的加密套件。

为了做到这一点, 就要根据你的应用程序所能支持的对加密套件集合加以限制。由于客户端与服务器必须就共同的加密套件达成一致才能进行通信, 所以这样将确保你能获得相应级别的安全。然而, 一种可能的副作用就是服务器与客户端可能没有共同的算法集合, 因而也就无法进行通信。因此, 通常最好的策略是允许提供超过可接受的最低安全级别的所有加密套件, 而不只是专选支持最高强度的加密算法。

我们推荐使用至少具有 768 位密钥的 RSA 或 DH/DSS。使用 3DES 或 RC4-128 进行加密, 如果你想获得最好的安全性就使用 3DES, 而若想获得最高的性能就使用 RC4-128, 并使用 SHA-1 来完成消息认证。请参见第 1 章来了解与这些算法有关的性能数据。

5.8 小结

我们已经讲述了许多重要的安全使用 SSL 的一般性规则。这应当使你对获得良好的安全性而需要采取的各种措施的有了一些认识。但这并不是说所列的这张单子是全面的——还有不计其数的方法可以把事情搞糟，大都是不注意自己的所做所为——但是它涵盖了许多容易出错的重要问题。作者个人曾目睹至少有一种实现违反了前面所描述的每条规则。

5.9 攻破 master_secret

攻破 master_secret 是灾难性的。让我们重新抽些时间看一看密钥的导出，看看攻破 master_secret（或 pre_master_secret）是怎么导致整个系统被攻破的。图 5.1 描述了密钥导出过程。保密信息以暗色调来表示，而其他信息都是公开的。

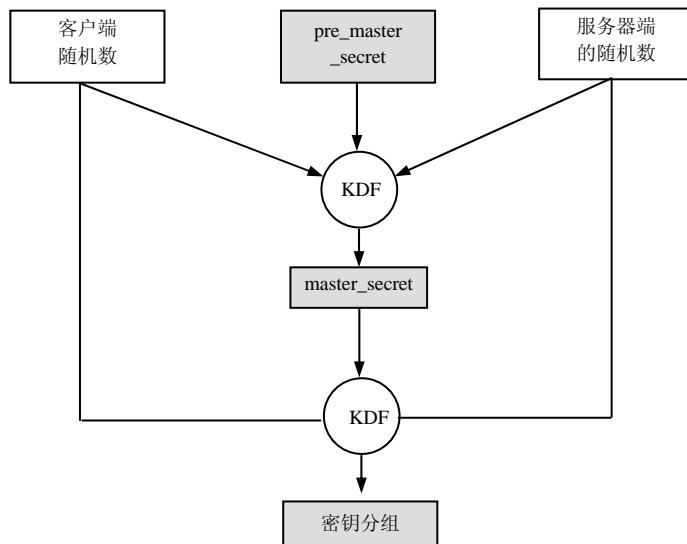


图 5.1 SSL 密钥导出

大家还记得密钥导出分两个阶段进行。首先，将 pre_master_secret 转换为 master_secret。KDF（密钥导出函数）接收三个参数，它们对于每次握手来说都不相同：client_random、server_random 和 pre_master_secret。然而，大家还记得 client_random 和 server_random 是在商定 master_secret 之前交换的，所以在握手时它们必定是以明文传输。因此，能够嗅探通信数据的攻击者就已经掌握了这些信息。唯一一条用来产生 master_secret 的保密信息就是 pre_master_secret。

第二阶段是将 master_secret 转换为独立的加密密钥。与以前一样，KDF 接受三项输入：client_random、server_random 和 master_secret。同样，只有 master_secret 是保密的。因此知道 master_secret 的攻击者就能计算出所有用来保护连接的加密密钥。与之类似，了解

pre_master_secret 的攻击者就能计算出 master_secret，继而计算出各种密钥。

因此，如果攻击者确实恢复了 master_secret，那么后果会非常糟糕。在这样的攻击者看来，你还是以明文来传输消息得了。

● 保密性攻击

正如我们以前所说的，了解 master_secret 的攻击者能够计算出供连接使用的所有加密密钥，其中自然包括加密数据用的加密密钥。回想一下我们的威胁模型，它假定攻击者能够读取在网络上传输的任何数据。当然，在我们使用 SSL 的时候，这些数据是加密的，但这对拥有 master_secret 的攻击者来说则不成问题。这样的攻击者拥有加密密钥，于是就能轻而易举地对每条捕获的记录进行解密。

存在一个非常小的技术问题：如果攻击者没有将每条记录都捕获下来的话，则需要做一些工作才能对随后的记录进行解密。大家还记得每条记录都像是连续数据流中的一部分而被加密的。对于序列密码来讲，每条记录使用密钥序列中的不同部分。为了解密一条记录，攻击者需要了解已经加密的字节数。然而，即便攻击者错过了一条记录，通常也能根据 TCP 序列号正确地猜测出其错过的字节数，稍加搜索就能找到密钥序列中正好适合那一部分内容。

对于分组加密来说，问题就稍微复杂一点。每条记录的 IV 都是前一条记录的最后一个加密分组。如果攻击者错过了那条记录，那么他就没有办法对记录的第一个加密分组进行解密，但仍然能够对记录中剩余的部分以及任何跟在那条记录之后的记录进行解密。

注意，这种攻击完全是被动的。如果攻击者打算介入网络通信的话，那么可以通过安排重传来更容易地绕过错失记录的问题。另一方面，如果攻击者愿意实施主动攻击的话，他还能够实施某些重要的完整性攻击，我们将在下一节进行讨论这个问题。

● 完整性攻击

一个拥有 master_secret 的攻击者当然能够计算出 MAC 密钥。与解密密钥结合在一起，这样的攻击者就能够实施类型广泛的完整性攻击。我们将在这里讲述几种攻击类型，但并不完整。

最简单的攻击形式就是接管连接。攻击者在要攻击的连接上伪造一条关闭消息，并发送给其中的一方（通常使用 TCP RST）然后再冒充那一方。攻击者可以伪造出令人信服的针对他所选择的任何通信的数据包。这种攻击技术并不高超，但却很有效。注意，这种攻击并不要求对底层网络的精确控制：实现 TCP 劫持（Hijacking）的技巧众所周知（请参见[Joncheray 1995]）。

如果攻击者对网络有着良好地控制，那么实施各种更为高深的攻击就是可能的。例如，攻击者在保持连接的同时可以随意增加和删除数据包。那么就有可能改变任何记录，甚至插入或删除记录。究竟需要怎样的能力则依赖于攻击者想要对通信数据进行何种修改以及通信双方正在使用什么样的加密算法。

如果攻击者修改的通信数据是以序列密码加密的，那么攻击者就能简单地通过对记录进行解密，改变消息文本，产生一个新的 MAC，重新加密并发送出去而对单条记录进行局部地修改。注意，攻击者无法改变记录的长度，否则对于密钥序列来说，所有将来的记录都无法同步。

另一方面，如果攻击者愿意截获并修改所有的后继记录的话，他可以对任何想要修改的记录进行任意改动。存在多种中间形式的攻击，它们要求对记录的修改较少，但是目前要清楚的是，了解 master_secret 的攻击者可以让通信数据变成任何所需要的样子。

可怕吧？

此刻我们希望使你相信让攻击者掌握 master_secret 是灾难性的，因此必须对其严加保护。令人欣慰的是，编写正确的 SSL 实现使得恢复 master_secret 即便是可能的也是非常困难的。下面几节讲述几种容易导致 master_secret 恢复的错误。

5.10 在内存中保护秘密

除非 SSL 实现大部分驻留在硬件中，否则 master_secret 将会存留在主机的主存储器中。这就意味着任何可以读取 SSL 进程存储空间的攻击者都能够读取 master_secret。因此，不可能面对掌握机器管理特权的攻击者而保护 SSL 连接。然而，某些错误行为有可能潜在的允许非管理员存取 master_secret 的错误。

磁盘存储

总的来讲，实现绝对应当避免将保密数据写到磁盘上。然而，在某些情况下（请参见第 9 章），这样做是必须的。对于这样的情况，实现应当小心设置文件的权限，以使其他用户无法读取这些数据。此外，实现还应当确保在其退出前删除磁盘文件。在那些具有恢复删除功能的操作系统上，程序应当确保将文件抹去且无法恢复。一种好的措施就是执行三遍覆盖操作来擦除数据：第一遍使用全零，第二遍使用随机数，第三遍再使用全零。这对于像 Windows95 和 98 这样的单用户操作系统来说是尤其需要注意的，这些系统的权限不能提供任何保护，而后来的用户能够读取整个磁盘。

内存加锁

对于使用虚拟内存的操作系统来说，最好确保包含 master_secret（以及其他保密数据）的内存永远不会被交换到磁盘上。如果交换到了磁盘上，后来的用户或系统管理员或许就能够读取数据，并因此而恢复保密数据。许多操作系统都提供用于锁住敏感数据的过程调用，这样这些数据就不会被交换到磁盘上了。然而，在某些情况下，这些过程实际上只是一些不起作用的占位函数（stub）。通过参阅操作系统文档可以了解有关这个主题的更多信息。

注意，内存加锁对于能够从驻留在内存中的进程读取敏感数据的系统管理员来说不能提供任何防护。它只能防止管理员在事后对数据进行恢复。

核心转储（Core Dump）

在许多操作系统上（尤其是 UNIX），当应用产生内存错误时，就会向磁盘上输出一个核心映像（core image）。该映像包含进程的整个存储状态，其中包括任何保密信息。因此能够读取这些文件的人也就可以恢复任何信息。程序员或许希望加装例外处理程序，以便检测出应用何时要进行核心转储并提前将敏感信息清零。

5.11 保证服务器私用密钥的安全

服务器的私用密钥是握手安全的关键。对于静态密钥模式，使用服务器的私用密钥直接对 `pre_master_secret` 进行保护。对于临时密钥模式，则使用它来对临时密钥进行认证，然后再用临时密钥保护 `pre_master_secret`。因此，尽管服务器的私用密钥被攻破总是一件糟糕的事情，但是如果使用了静态密钥则情况就会更糟，因为这将危害到所有以该密钥加密的通信数据。

静态密钥

最容易受到损害的情形就是服务器使用静态密钥对来确立 `master_secret`。最常见的情况就是我们在第 3 章所讲的情况。服务器具有一对 RSA 密钥，公用密钥位于它的证书中。客户端使用公用密钥来加密 `master_secret`，这样只有服务器才能够读取它。然而，掌握私用密钥的攻击者可以轻而易举地对 `master_secret` 进行解密，并实施我们前面所描述的所有攻击。

在这种情况下，攻破服务器的静态私用密钥就会攻破使用该密钥而确立的每个 SSL 会话，因为攻击者可以脱机（offline）选择时间来恢复 `master_secret`。换言之，就是攻击者可以实施一种纯粹的被动攻击。编写出接受服务器的私用密钥和事先记录的消息跟踪信息并输出明文的程序只是小事一桩。实际上，第 4 章 `ssldump` 的跟踪信息就是使用这种技术产生的。

虽然静态密钥大都与 RSA 一起使用，但 SSL 还有一些静态 Diffie-Hellman 模式，其中静态 DH 密钥位于服务器证书中。这些模式同样易受这种类型的攻击。然而，SSL 中大多数使用 DH 的地方都使用临时 DH。下一节将讨论使用临时密钥（请见第 4 章的描述）时，攻破私用密钥所造成的后果。

临时密钥

考虑服务器的私用密钥没有用于密钥确立而只用于签名的情况。服务器产生一个临时密钥对，使用它来完成密钥的确立并使用其静态密钥对向客户端验证临时公用密钥。

当使用临时密钥的时候，实际上存在两种攻破私用密钥的类型，每种都会导致不同的后果。静态密钥（用于认证）或临时密钥（用于密钥的确立）均可被攻破。首先考虑静态密钥被攻破的情况。了解静态密钥对解密通信数据来说没有任何价值，因为根本就没有使用那个密钥来加密数据，数据使用临时密钥加密。然而，在恢复临时密钥时，静态密钥也没有什么用。

不过恢复静态认证密钥并不是毫无用途的。拥有服务器静态认证密钥的攻击者可以在握手阶段接管任何连接（冒充服务器）。然后，再简单地扮做服务器或对客户端冒充成服务器和对服务器冒充成客户端来实施中间人攻击，并在它们之间传递数据。图 5.2 展示了这一过程。

图 5.2 在展示这种中间人攻击时，去除了一些不相关的信息。攻击者将大部分握手消息直接从客户端拷贝到服务器或是从服务器拷贝到客户端，但是将 `ServerKeyExchange` 消息更改为他自己的临时密钥而不是服务器的临时密钥的签名版本。他使用攻破的私用密钥对其进行签名。然后对客户端过来的所有通信数据进行解密并在重新加密后转发给服务器，同时做出他所想要的各种改动。这里有一处很微妙的地方，就是攻击者必须产生一个新的 `Finished`

消息，因为由于临时密钥的改变，客户端所看到的握手散列值与服务器所见到的不同。

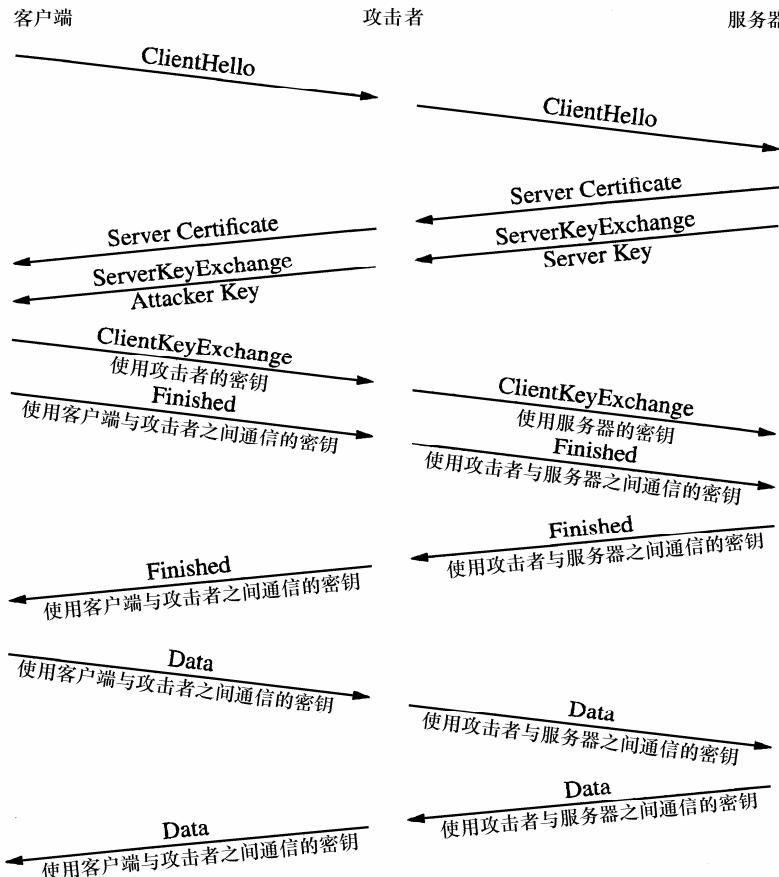


图 5.2 使用已攻破的签名密钥实施的攻击

所以，攻破静态认证密钥可以允许攻击者对任何处于握手而不是之后阶段的给定会话进行主动攻击。需要指出的一点是，如果使用了客户端认证的话，攻击者就无法实施这种形式的攻击。因为客户端的签名是对所有握手消息的签名。攻击者发送给客户端的消息与服务器认为其发送给客户端的消息不同。因此，服务器的签名检查就会失败。

即便服务器机器彻底遭到攻破而且服务器的私用密钥被恢复，这些信息仍无法用来攻击任何已经建立的和关闭的会话。这种属性被称做 PFS（完美向前保密）。注意，静态密钥确立的密钥没有这种属性。然而，只有在通信密钥本身没有存储在别的地方时才能获得 PFS。例如，如果密钥位于会话缓存中，那么攻击者仍然能够对连接进行解密。定期删除无用的或过期的密钥资料对 PFS 来说至关重要。

与之对照，如果攻击者攻破了服务器临时密钥中的一个，那么他就能用它来攻击使用这个密钥保护的连接。且对于这些连接，攻击者能够恢复 master_secret 并实施 5.9 节所描述的攻击。

有多少连接受到影响完全依赖于服务器的行为。当使用临时 RSA 的时候，攻破的连接数会非常巨大。正如我们在第 4 章所讨论的，产生 RSA 密钥开销昂贵，因而服务器经常在启动时产生一个临时 RSA 密钥，并一直用到数天或数周后直至服务器关闭为止。所以，即

使只有一个临时密钥遭到攻破，被攻破的通信数据数量也还是非常巨大的。由于目前只有可出口 RSA 加密密钥 (<512 位) 中使用临时 RSA，而我们知道这样大小的密钥正处于即将被攻破的边缘，所以这是让人揪心的地方。

当使用临时 DH 时，频繁地重新产生临时密钥相当常见。在这种情况下，攻破的会话并不会对其他任何会话产生影响。

● 私用密钥的存储

大多数 SSL 服务器都是在通用计算机上运行的。与大多数服务器程序类似，它们使用计算机的文件系统来存储每次运行期间的持久性数据，这通常包括服务器的私用密钥。如果服务器机器是安全的并能够抵抗攻击，而且系统只允许受信用户使用，那么这样也算是一种相当稳妥的措施。

然而，要想拥有一台完全可信的服务器却是非常困难的。许多单位都无法为 SSL 服务器设置专门的机器，而且即便可以，机器也有可能被攻破或盗取。为了防止这些意外事件的发生，常见的措施就是对磁盘上的私用密钥进行加密。

此刻，敏锐的读者会纳闷，这样做怎么会使情况有所改善呢？我们对私用密钥进行了加密，而现在我们又需要保护加密私用密钥的密钥。当然，我们可以对那个密钥也进行加密，但是那样一来我们又需要保护新的加密密钥，以此类推。因此，只进行加密并不是解决问题的办法。

当然，我们可以强迫用户用脑子记住那个密钥，但是要记住 128 个随机字节的工作量确实太大了，大多数用户都不会愿意接受这项工作。通常的做法是使用口令 (password) 或通行码 (passphrase) 来产生我们用于加密私用密钥的加密密钥。口令很容易记，我们使用特殊的密钥导出函数根据口令来产生密钥。

● 基于口令的加密

存在大量将通行码转换为加密密钥的方法。然而，强度高的方法都有着一些共同的特性。这些函数之所以必须要与普通的密钥导出函数不同是因为文本的熵 (entropy) 通常很低。也就是说，给定部分单词或短语，余下的内容并不是完全不可预测的（而加密密钥通常的构造使得了解密钥的部分内容不会提供任何有关其余内容的信息）。多种对付这种问题的技术被设计出来。

鼓励用户选择比其所导出的密钥还长的通行码。例如，128 位的密钥可以用 16 个字符来表示，但是通行码通常为 50 个字符或更多。这既使得它们易于记忆又提供了某种程度的安全。尽管如此，通行码或许还是比密钥更容易猜测的东西，尤其是用户经常选择简短的口令。为了使得即便选择了不好的口令也能提供一定的保护，基于通行码的密钥导出函数通常使用两种称做成分添加 (salting) 和迭代 (iteration) 的技巧。

成分添加涉及以一个公共的随机值作为密钥导出函数的部分输入。这个值被称做“添加剂”，添加剂与密钥文件一起存储。即便两个用户使用相同的口令，他们的加密密钥也不会相同，因为他们将使用不同的添加剂值。这样可以阻止两种攻击。第一种，攻击者或许会构造一张包含所有常用口令的表，然后通过搜索这张表来攻破口令。第二种，攻击者可以搜集一系列的口令文件，并对其进行搜索，这样只执行一遍密钥导出过程。当使用了成分添加，这两种攻击均不会得逞。

迭代会减缓密钥导出函数的速度。基于口令的密钥导出函数被设计成极其缓慢的那种，以此来增大搜索口令空间的开销。如果对其私用密钥解密花费半秒钟时间的话，用户是不会注意到的，但是如果对数以千万计口令中的每条进行检查都花费一秒钟的话，就会给攻击者造成极大的负担。典型的函数被设计成具有一个基本的原语，可以让该原语作用（迭代）任意多次，其中以一个阶段的输出作为下一阶段的输入。每次迭代都要花费一些时间，这样就会减慢整个过程。迭代计数也是与加密文件一起存储的。

大家最熟悉的基于口令加密的算法为 RSA 的 PKCS#5[RSA1993d]，该算法同时包含成分添加和迭代功能。不幸的是，PKCS #5 至多只能产生 160 位的密钥，这对 3DES 来说毫无用处。当 Microsoft 设计 PFX 密钥存储标准（也就是现在的 PKCS#12[RSA1999a]）时，他们设计了一种新的可以产生任意长度密钥的函数。从那以后，RSA 便更新了 PKCS#5 以支持任意长度的密钥[RSA1999b]。

图 5.3 描述了 PKCS#5 版本 1 中用到的成分添加与迭代过程的图示。该过程分两个阶段进行。第一，计算口令与添加剂连结的散列来产生 T_1 。重复计算该值的散列（count -1 次）来产生 T_n ，它就是函数的输出， T_n 等于 $\text{Hash}(T_{n-1})$ 。显然这种转换无法产生比散列函数的输出还长的输出，这也是为什么不用它来产生长于 160 位 SHA-1 输出的密钥的原因。PKCS#12 与 PKCS#5 版本 2 的转换在本质上是相似的但并不相同，它们都可以产生任意长度的密钥。

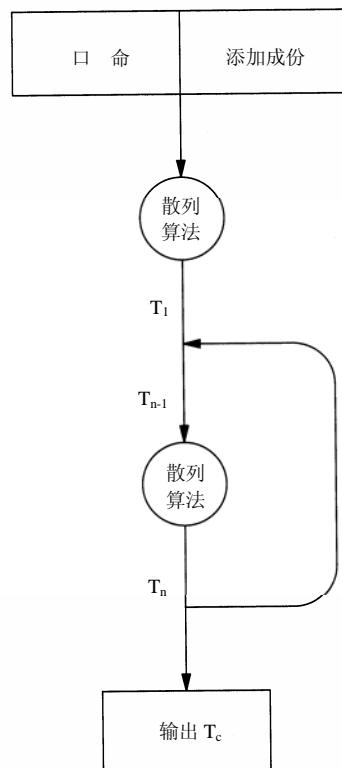


图 5.3 PKCS#5 转换

无存储密钥的恢复

设想你准备运行 SSL 的设备没有为用户提供专门的持续性存储器，且该设备没有在磁盘上留有用于存储加密的或其他形式私用密钥的地方。如果我们能够从通行码来导出私用密钥的话，则该设备还是可以拥有私用密钥的。

最简单的就是使用诸如 DH 或 DSA 等离散对数系统时的情况。由于私用密钥 X 是一个任意数字，所以有可能使用我们刚刚讨论的基于口令的 KDF 根据通行码来产生它。组参数 g 、 p 和 q 对所有设备用户来说都可以是公共的，因而可以在制作时将其烧录到设备上的持续存储器中。

这种技术对于像 RSA 那样的系统来说则不够方便，那样系统中的密钥产生过程开销昂贵，但还是可以用通行码来给产生 RSA 密钥的 PRNG 提供种子。只要通行码相同，每次都会产生相同的 RSA 密钥。

这种方案主要对于那些给定用户而有可能在任意指定时刻使用任意数量不同设备的情况非常有用，例如公共查询机或共享终端。因为证书在网上可以以明文存储，所以用户可以径直走到机器跟前输入自己的口令。

硬件加密

另一种存储私用密钥的方法就是使用硬件设备。这种设备既包含供私用密钥使用的持续性存储器也包含能够执行加密计算的处理器。私用密钥在设备上产生，绝不会跑到设备外边，而所有的私用密钥操作都在设备上完成。

硬件加密设备有两种基本的类型，持续性的和可插拔的。持续性设备与机器时刻相连，要么作为系统总线上的插卡要么作为通过 SCSI、并口或其他接口插在后面的部件。而可插拔设备通常被包装成 PCMCIA 卡或智能卡（信用卡大小的设备）。可插拔设备具有显著的优点，可以将其从机器中拿走并锁起来以提供附加的安全。

任何类型的设备都可以被偷走，因此通常使用口令或 PIN（个人标识码）来对私用密钥进行保护。除非用户使用正确的 PIN 登录，否则设备就无法正常工作。有时就像在磁盘上一样对私用密钥进行加密，而有时则是简单地通过编程来禁止密钥的使用，除非输入了正确的 PIN 才行。此外，这种设备通常还被设计成在数次（次数很少）不正确地尝试后将自身清零，而清零即意味着删除所有敏感的内部状态信息。

硬件设备常常集成一种防篡改功能以提供附加的安全。这种设备被设计成能够检测出打开外壳并直接存取硬件的企图。如果检测出了这种企图，设备就会将自己的内存清零。其设计思想就是即便物理地占有这个设备，但如果没合适的 PIN 的话，也是徒劳的。FIPS-140-1[NIST1994b]提供了有关这种防篡改类别的标准。

因此，硬件设备与软件密钥文件比起来有两种主要的优势：由于在几次失败的登录企图后插卡会清零，所以攻击者就不能简单地尽其所能针对密钥文件尝试所有可能的 PIN，因而选择一个非常好的 PIN 也不是必须的（这样很好，因为选择良好的 PIN 也是很困难的）。

第二，即便攻击者掌握了 PIN，他也需要对设备的物理存取。假定攻击者进入服务器机器并猜出了 PIN，但是你的密钥是在硬件中的。他可以使用服务器对通信数据进行解密，但是一旦你发现了这种入侵，就可以拔掉设备或改变 PIN，而攻击也就到此为止。然而对软件

来说，如果攻击者掌握私用密钥的话，你就不得不产生一个全新的密钥对。固定设备通常被包装成加密加速器（并以这种形式来营销）。它们意图能够比主机系统更快地执行加密操作。结果，它们通常增强了安全而又使性能获得很大的提升。而可插拔设备一般都非常缓慢（一般为每秒几次 RSA 操作），因此也就着实不能在服务器环境中使用，尽管它们可以为客户端的密钥资料提供便利的安全和方便的包装形式。

一些密码加速器既有固定部件也有可插拔部件。这增加了安全，因为没有可插拔密钥的话，设备就无法使用，而该设备同时又是一台普通的固定设备。

通行码的录入（passphrase）

你现在或许已经注意到了，加密磁盘文件与硬件设备通常都要求操作员输入一个通行码才能启动服务器。这是整个系统安全的虚弱一环。要能让服务器系统在没有用户介入的情况下顺利完成重新启动是再好不过的了，但是要求输入通行码才能启动安全服务器使这种愿望化为泡影。

只存在两种可行的方案：打破安全性或打破无人插手的重新启动。要想打破无人插手的重起很容易：要求操作员输入通行码才能启动服务器。这意味着机器可以正确地重起，但是需要操作员在场才能启动安全服务器。

与之类似，打破安全性也很容易。你只要将通行码放置在系统启动脚本的某个地方并安排在启动服务器的时候将其传递给服务器即可。这里明显的不利之处就是任何可以读取那个文件的人（例如，系统管理员或攻击者）都能够读取口令。通常这些文件设置有非常严格的权限，但是大多数操作系统都允许管理员读取任何文件。

至今还不存在可以绕过这两种选择的方法。你只有选择其一并接受随之而来的后果。

生物统计学（Biometrics）

诸如指纹识别和视网膜扫描等生物认证措施经常被用做安全的存取控制手段。虽然对这些措施是否能够安全地加以实现存有怀疑，但它们还是广泛应用在对安全环境和计算机的存取上。

然而，单独使用生物统计技术来保护私用密钥资料是不合适的。到目前为止我们所讨论的措施都是根据用户输入的数据对私用密钥进行加密。由于要用这些数据来创建加密密钥而且加密算法也被专门设计成密钥中的微小错误都会导致在解密时产生大量错误，所以输入数据必须每次都是一样的。

可是生物统计学不能很容易地应用于这种目的，因为每次度量的特征都不一样。因此，要想根据每次生物统计信息重新创建相同的加密密钥即便可能也将是非常困难的。所以，生物统计系统不得不以明文来保存密钥，只是简单地使用生物统计技术对存取进行控制。如果包含密钥的设备被攻破，则这种措施就不可取了，因为攻击者无须对密钥进行解密。而如果密钥保存在磁盘上的某个地方，那么就将是灾难性的了。

这条一般性的规则有两个例外。第一，如果密钥存储在受信的防篡改硬件中，那么生物统计信息就是安全的，因为硬件会在其被恢复之前将密钥清零。这要求你的防篡改技术是非常可信的！第二，如果将通行码与生物统计结合起来使用，那么仍然可以使用通行码对密钥进行加密，而将生物统计用做第二位的存取控制措施。在这种情况下，生物统计技术就增加了宝贵的安全性。

概要

理解自身安全模型对于最终使用何种技术至关重要。例如，如果你的系统具有高强度的物理安全，你就没有必要使用同样高强度的安全来保护密钥。与之对照，如果你担心自己的机器会被盗取或由非授权用户进行存取，那么在硬件中保护密钥来限制遭到破坏的代价则至关重要。本节提供了评估各种技术的信息，但是你必须从中选择适合自己安全模型的一种。

5.12 随机数生成

SSL 实现的安全完全依赖于它所产生的随机数的质量。客户端与服务器都需要拥有高强度的随机数生成能力。虽然什么是“高”强度的随机数并不清楚，但它不是什么却相当明确：大多数编程语言所提供的“随机”数发生器不能胜任这项工作，需要使用特殊的工具才行。

有两种产生随机数据的通用方法：通过某种随机或几乎是随机的物理过程可以获得真正的随机性，而使用能产生看似随机数据序列的算法可以获得伪随机性。一般来讲，大多数机器上都没有良好的硬件随机数发生器，因此程序员们几乎都屈从于使用算法形式的 PRNG（伪随机数发生器）。

不管选择何种策略，我们至少需要下列属性。

序列应当是无偏的（unbiased）。0 与 1 位的数量平均起来应当是一样的。

输出应当是互不相关的（decorrelated）。对字节序列中某一部分输出值的了解不应当有助于了解任何其他部分的值。

PRNG（伪随机数发生器）

PRNG 的总体思想是，你有某种产生无法预测的字节序列的函数。从理论上讲，这种字节序列与由硬件设备（如，抛硬币）产生的字节序列无法区分的。

当然，实际情况并不是这样，因为所有的 PRNG 最终都会重复字节序列。PRNG 的输出是 PRNG 内部状态的函数。由于其内部状态不能比它们所运行的计算机的内存大，所以在输出了那么多的字节序列之后，就必须重复上一阶段的内容。在实际应用中，PRNG 的内部状态只要大到能够在不出现状态重复的情况下产生本质上无穷多个字节的大小就足够了。256 位的内部状态足以轻易达到这种目标。

PRNG 通常基于摘要或加密算法。[Kelsey1999]中描述了一个不错的范例，里面有许多众所周知的 PRNG，但我们在此不进行具体讨论。然而，它们都有一个共同点——需要结合一些保密数据来采集种子，这些数据用于建立起内部状态。知道了种子数据就等同于了解了 PRNG 的内部状态，攻击可以据此者对字节序列进行完全的预测。使用 PRNG 的主要问题就是如何正确地获取种子。

一般来讲，将 PRNG 构造成能够从大量低质量的随机数据区中抽取种子并从中分离出随机性来。例如，考虑诸如几天来一只股票价格这样的数据。如果股票星期一处在 85，那么星期二就非常有可能位于 80 到 90 之间的某个位置，所以这 10 个数字并不算是随机。一种好的 PRNG 可以根据这种数据抽取种子并从中汲取出所有好的随机性，从而使得搜集随机数据的工作多少容易一些。

一种收集种子数据的典型策略就是观察某些相当易变的系统状态，如网络活动、进程

表或屏幕等，并从它们的输入中提取出高强度的随机性。在一种正在运行的系统上，这对产生用以保护单一会话的密钥来说估计已经足够好了。多种操作系统（至少包括 Windows2000、*BSD 和 Linux）都提供了一种内核设备来收集系统事件，并持续在后台提取 PRNG 的种子。

对于高安全应用来讲，如私用密钥生成，实现常常要求用户输入并度量各次击键或鼠标运动之间的时间。RFC 1750[Eastlake1994]提供了一些有关收集种子数据的指导。

尽管大多数 SSL 实现都提供某种从系统数据抽取 PRNG 种子的方法，但是它们通常期望应用来提供大部分的种子数据并以自己的种子数据作为最后一关。即便你使用的是工具箱，也常常需要自己动手抽取 PRNG 种子。

需要指出的一点是，高强度的 PRNG 在其获取的种子质量十分低劣时也能产生看似随机的输出。虽然有一些有用的、用于判断你所使用的 PRNG 是否为高强度的统计测试，但它们在判定是否获取了高质量的种子方面却没有什么用处。



硬件随机数发生器

其他流行的策略是使用硬件随机源。某些物理设备几乎能够在量子机械级别执行随机的动作，因而可以用它产生随机数据。最显而易见的例子就是放射衰变，它以随机的（但平均是可预测的）间隔产生稳定的信号序列。我们可以使用信号之间的时间间隔（或给定时间窗口范围内的密度）来产生随机数据。

虽然放射衰变是随机的，但它并不是无偏的（unbiased），这是其因为输出结果都聚集在某个平均值周围。通常需要一些处理才能确保输出是无偏的（unbiased）。

设想你有一个随机数发生器，它是随机的，但却以某种简单的形式有所偏向，就象没有平衡好的硬币，60% 返回正面，40% 返回反面。John von Neumann[Neumann1951]提出了一种纠偏的简单方法。成对地收集输出，抛弃任何完全相同（如，{正面，正面}或{反面，反面}）的一对。而对于任何不同的对，输出第一个元素（如{正面，反面}成为正面）。由于对第一或第二的选择是任意的，因而很容易就能看出即便发生器是偏心的，这种技巧也能产生平衡的输出序列，因为{正面，反面}序列的概率与{反面，正面}的概率相同。

做到这一点的最简单的方法就是使用硬件随机数发生器来抽取 PRNG 的种子。它在克服偏心问题的同时，也确保了在你需要随机数时，不必再等硬件设备完成工作。

最近，Intel 公司给他们的奔腾 III 处理器增加了一种基于硬件的随机数发生器。RNG 在很大程度上是按照上面描述的那样工作的，在芯片中内建了一种基于热噪声的随机数设备。数据通过基于 SHA-1 的混合函数进行后处理。初始评估[Kocher1999]表明该系统是安全的，前提当然是运行的机器没有被试图伪造 RNG 输出的攻击者攻破。

5.13 证书链的验证

我们已经明确地指出，一份无法验证的证书跟没有证书没什么两样。然而，仅构造出到达某个根结点的证书链并不足以对证书进行验证。为了确信你是在与自己认为的实体进行通信，还要进行一些其他检查。

服务器身份

正如我们先前所讨论的，服务器提供的证书必须包含与期望的服务器身份相关联的身份。期望的身份不能在线路上进行传输，但必须是在初始化连接时客户端已经知道的。

在大多数情况下，当客户端想要与 SSL 服务器连接的时候，目的地是通过 DNS 域名来指定的（请参见[Mockapetris1987a,Mockapetris1987b]），如 foo.example.com 或是通过包含有域名的统一资源定位符（URL）[Berners-Lee1994]来指定。因此，所期望的服务器身份就是域名，而且必须与证书中的身份相匹配。

不幸的是，证书中的标识名（DN）并不真的与域名兼容。大家回想一下第一章，标识名的结构是一系列的属性-值对，而域名实际上就是一个单一的字符串。此外，域名没有 DN 属性。结果，大多数 CA 都使用 Common Name 属性来表示域名。这种方案要求客户端与服务器都要知道这种约定才能正确工作。

如果你使用的是 X509v3 证书，由于这种证书允许有扩展，所以就是另一回事了。版本 3 的证书可以包含一个 subjectAltName 扩展，其中又包含证书主题的另一种名称形式。dNSName 就是这样一种形式，使用它来表示域名。允许主题的 DN 字段为空而仅使用 subjectAltName 来指示身份。

需要指出的重要一点是，我们不使用服务器的 IP 地址来表示其身份。这样做的原因是客户端通常不是从受信的来源获得 IP 地址的。相反，域名虽是受信的但却是使用 DNS（一种称作名字解析的过程）来进行查找的，这种过程很容易被攻破，如[Bellovin1995]中所描述的那样。因此，无法安全地将 IP 地址与证书中的 IP 地址进行比对。

不幸的是，原始 BSD 套接字 API 要求客户端使用 IP 地址来指定要连接的主机（从而迫使它们自己去完成名字解析）。许多 SSL 实现都模仿这种行为。由于 SSL 实现并不了解域名，所以它无法根据证书进行检查，而这种检查又必须在应用层完成。图 5.4 提供了使用 PureTLS 来完成这一过程的例子。我们将在第 7 章讨论安全协议的设计时更多地谈到服务器身份检查的问题。

```
public static SSLSocket connect(SSLContext ctx, String host, int port)
    throws IOException {
    // Connect to the remote host
    InetAddress hostAddr=InetAddress.getByName(host);
    SSLSocket s=new SSLSocket(ctx,hostAddr,port);

    // Check the certificate chain
    Vector certChain=s.getCertificateChain();

    // Length check
    if(certChain.size()>2)
        throw new IOException("Certificate chain too long");

    // Hostname check (using Common Name)
    Certificate cert=(Certificate)certChain.lastElement();
    String commonName=dnToCommonName(cert.getSubjectName());
    if(!commonName.equals(host))
        throw new IOException("Host name does not match commonName");

    return s;
}
```

图 5.4 Java SSL 身份检查范例

客户端的身份

显然，只有在使用客户端认证的时候才需要对客户端身份进行验证。对于其他情况，客户端完全是匿名的。SSL 其实不会挑剔客户端的身份，应用层应该知道如何对其进行解释。也就是说，剩下的有关证书链验证的讨论对客户端证书链来说与对服务器同样有效。

选择你的根（Root）

所有验证的证书链最终都必须终止于一份根证书。这份根证书要么由应用开发人员要么由用户（或其系统管理员）在应用中进行配置。例如，当前大多数流行的浏览器都在应用中配置有 25 个以上的根证书。

这些根只有是受信的才能准确认证顾客的身份，顾客的证书由这些根来签名。如果不是这样，那么系统就会变成不可信的。例如，一个根给一个攻击者提供了包含 Alice 名字的证书，那么攻击者就能够在你与其服务器进行连接时冒充 Alice。

为了评估一个根是否可接受，需要评估它们对用户进行认证时所持有的认真程度（它们是否要求可靠的物理 ID，等等）以及它们对保护自己的密钥所持的认真程度。Netscape 和 Microsoft 都没有把它们用于判定将哪些根编译到其浏览器中的规定公布于众。然而，Microsoft 与 Netscape 都允许用户安装他们自己的根。这通常是通过让用户解析一个具有将其标注为根证书的指定内容类型的 Web 页面来完成的。除非小心行事，否则这种做法就会引入一种安全隐患。必须通过受信的连接并从受信的地点获得根，否则攻击者就可以说服用户安装一个假根。一种常见的做法是脱机分发根的指纹信息——即根密钥的摘要，然后再允许用户在安装的时候对根进行校验。

证书链的深度

出于管理的考虑，经常使用包含不止一个证书的证书链是方便的。这样就可以允许分布式的证书颁发。例如，设想 Widgets 公司想要给位于 10 个地点的 20000 名雇员颁发证书，让每个地点颁发自己的证书或许是一种方便的办法。

一种完成这项工作的方法就是让所有 Widgets 证书都拥有一个单一的根证书，这个根为每个地点的 CA 证书签名，而所在地的 CA 又会为每个雇员的证书签名。图 5.5 描述了使用这种模型的证书的层次结构。

在这个例子中，Widgets 根 CA 对两个地点的 CA 进行认证，一个是 Palo Alto 办事处，另一个是 Seattle 办事处。每个办事处都有两个雇员，他们由其本地的 CA 进行认证。Alice 和 Bob 都在 Palo Alto 工作，因而由 Palo Alto CA 认证。Charlie 与 Dave 均在 Seattle 工作，由 Seattle CA 进行认证。

现在需要区分两类不同的实体：那些我们可以信任能为他人提供担保的，以及那些我们所不信任的。例如，在上面的例子中，Widgets Palo Alto CA 证书的意思就是说“根 CA 确定这个密钥属于 Widgets Palo Alto CA，而该 CA 可以颁发证书”。然而，Alice 的证书意思是说“根 CA 确定这个密钥属于 Alice”。信任 Widget Palo Alto CA 对证书进行签名，但是不相信 Alice 执行这样的任务。如果允许 Alice 对证书进行签名，那么她自己就能够创建如图 5.6 所示的证书链并冒充 Bob。

这两种证书分类之间的差异可以是隐含的也可以是显式的。在 X.509v3 证书之前，惟一的选择就是通过控制证书链的深度使其为隐含的。也就是说，它们会声明从它们的根开始的

证书链不会长于 n 个证书。在当前的这个例子中， n 是 2：当地 CA（由根来签名）和用户。

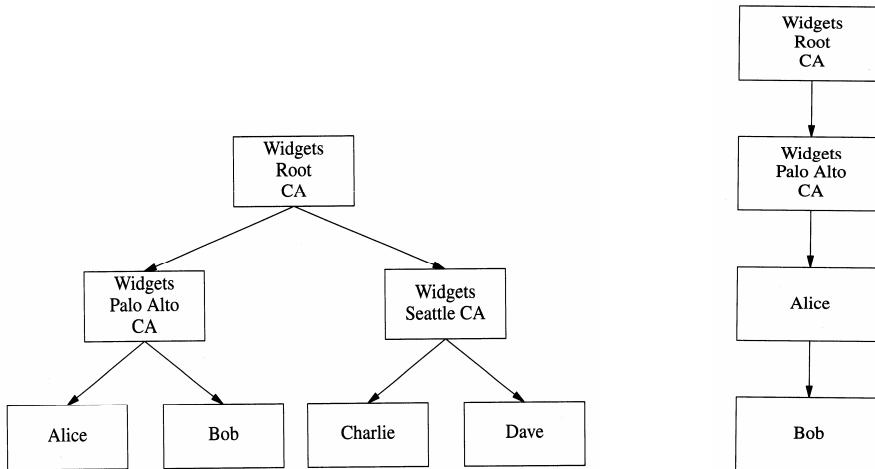


图 5.5 证书层次范例

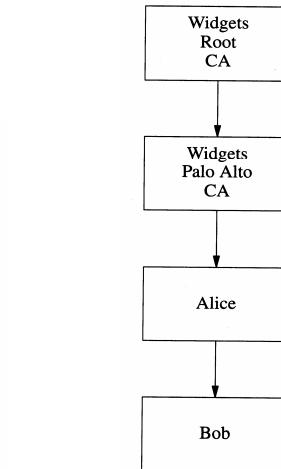


图 5.6 通过扩展层次结构来进行冒充

这种方案的主要问题就是它不允许实现遵循一种全局性的策略。一个 CA 可能允许深度为 2，而另一个可能允许深度为 1。若将实现的最大深度设置为 2 就会使得由第二个 CA 认证的个人能够实施冒充攻击，而将深度设置为 1 又无法验证由第一个 CA 颁发的证书链。在实际应用中，几乎所有的 CA 都使用只有一层深度的证书链，可以很容易的增加下一级 CA，来作为使用更长证书链的 CA 的根。

更新的解决方案是用 X.509v3 证书扩展来指示证书是否对应一个 CA。尤其是 PKIX ([Housley1999a]中所描述的 IETF X.509 证书资料) 要求在所有的 CA 证书中必须提供基本约束 (Basic Constraint) 扩展。基本约束扩展指定证书是否属于 CA，以及证书链可以从那条路径延展的深度。

不幸的是，当前许多可用的证书和 CA 都不遵循 PKIX，而且存在大量不遵循 PKIX 的传统证书。因此，如果你的系统是封闭系统，就应当尽量使用基本约束扩展，不然的话，使用硬性编码的长度约束很容易让你陷入困境。遵循 PKIX 的软件将会拒绝本是合法的 CA，因为他们不具备基本约束。

密钥用法扩展

X.509 密钥用法扩展限制密钥的使用目的。它定义了 8 种密钥用法值，它们通过构成位屏蔽 (bitmask) 来指定允许的操作。这些值是：digitalSignature、nonRepudiation、keyEncipherment、dataEncipherment、keyAgreement、keyCertSign、cRLSign、encipherOnly 和 decipherOnly。没有浏览器要求 keyUsage 扩展必须存在，但是 Netscape 和 Microsoft 都检查它是否存在。图 5.7 描述了各种类型的证书在完成每种类型的操作时所要求的比特位。

用途	Netscape	Microsoft
SSL 服务器	keyEncipherment	keyEncipherment
SSL 客户端	digitalSignature	KeyEncipherment
证书签名	keyCertSign	keyCertSign

图 5.7 X.509 扩展的使用

注意，Microsoft 和 Netscape 设置用来指示执行客户端认证的比特位是不同的。究竟使用什么比特位来表示客户端认证的困惑源于这样一个事实，即 SSL 客户端认证涉及对客户端从未见过的字符串进行签名。可以证明，客户端的密钥被用于对密钥交换进行认证，因此 keyEncipherment 更为合适。而 Netscape 采用了自然一些的方案，使用 digitalSignature。

更混乱的是，Netscape 还定义一个它自己的证书用法扩展：netscape-cert-type。有关 Netscape 证书情况的文档可以在[Netscape1995a]中找到。netscape-cert-type 扩展是一种位字符串。相关的值有：SSL Client(bit 0)、SSL Server(bit 1)和 SSL CA(bit 5)，此外还有几个用于 S/MIME 证书和某些 Netscape 称做对象签名的东西。虽然还不清楚当这两种扩展同时存在而且产生冲突时的行为。但在实际应用中，不太可能发生这样的情况。

根 CA 的证书被编译到浏览器中。中介 CA 的证书应当包含 basicConstraints。Netscape 还接纳在 netscape-cert-type 中的某位进行设置的 CA 为中介 CA。而许多其他的实现，包括比较老的 OpenSSL 版本远没有这么挑剔，它们可以接受任何证书为 CA，只要它们能够构成一条链。作为一种规定，现代实现应当手工限制证书链的长度或者坚持在证书中使用表示认证实体为 CA 的指示。

并不是所有的证书都包含 keyUsage 扩展，大多数浏览器会接受客户端或服务器的证书但不接受中间 CA 的。

其他的证书扩展

Netscape 还定义了其他的证书扩展，如图 5.8 所示。这些扩展在[Netscape1999a]中描述。其中的大多数扩展都是验证器能够通过解析来获取有关证书信息的 URL，如颁发所依据的政策以及撤消状态等。这些扩展并没有被广泛使用，因为撤消实际上并不常见。还有一个 netscape-ssl-server-name 扩展，可以使用这个字段代替 Common Name 来标识证书所属的服务器。因此存在三种将证书的名字安置于证书中的方法：Common Name、subjectAltName 扩展以及 netscape-ssl-server-name 扩展。一般来讲，应当避免使用 netscape-ssl-server-name，因为它根本就不标准。通过使用 Common Name 来获得最大程度的兼容性，或只对于新设计的系统使用 subjectAltName。

扩展	目的/用途
netscape-base-url	证书中 URL 的基地址——这样别的 URL 就可以相对它书写的短一些
netscape-revocation-url	包含指向返回撤消信息的 URL
netscape-ca-revocation-url	用于检查 CA 证书的撤消状态
netscape-cert-renewal-url	证书拥有者可用来复用证书的 URL
netscape-ca-policy-url	文本形成的证书颁发所依据的策略
netscape-ssl-server-name	SSL 服务器名
netscape-comment	显示给用户的说明

图 5.8 其他的 Netscape 证书扩展

5.14 部 分 攻 破

SSL 的安全完全依赖于所使用的加密套件。每个加密套件指定四种算法：数字签名算法、

密钥确立算法、数据加密算法和消息摘要算法。虽然不是正交地指定这些算法，但一般来讲每种类型的算法都存在多种选择，这些选择在强度上存在差异。此外，对于数字签名和密钥确立算法，可以使用多种密钥尺寸，而且每种密钥尺寸均代表一种不同的安全级别。很可能存在这样一种情况，一种算法被攻破了而其他的算法仍是安全的。这一节考虑遭遇这种破坏时对整个系统安全的影响。

当谈到算法强度时，我们需要考虑两种因素。第一种，我们可以基于针对该算法最熟知的攻击来指定一个该算法强度的上限。依据这种度量方式，具有 512 位密钥的 RC2-40、RC4-40、DES、RSA、DH 以及 DSS 都已经处于危险的弱强度之列。也就是说，通过一定努力，攻击者有可能恢复这些算法中的某个密钥。

第二种因素就是该种算法在将来某个时候遭到彻底攻破的可能性。也就是说，有可能不管密钥的长度，花费少量的精力，就能恢复任意数量的密钥。其他的算法发生过这样的情况。SSL 中所使用的大多数算法都经历了相当广泛地分析，因此普遍相信人们对这些算法的安全属性有一定了解。不管怎样，一种算法遭遇灾难性的破解仍然是一种最糟糕的情况。

数字签名算法

SSL 使用数字签名来实现三种用途：

- (1) 对证书进行认证。
- (2) 对临时密钥签名。
- (3) 对于客户端认证的 CertificateVerify 进行签名。

彻底攻破一种数字签名算法就会攻破上述所有的用途，并使协议完全处于主动攻击之下。这种情况要比攻破单个服务器的私用密钥还要糟，因为攻破签名算法就意味着所有使用那种算法签名的证书也存在安全隐患。即，任何接受被攻破算法的 SSL 连接都处于主动攻击之下。

对算法的部分攻破也是可能的。从某种意义上讲，具有 512 位或更少密钥的 RSA 和 DSS 已经出现过这种情况，因为恢复这样的密钥对于设备精良和专注的攻击者来说是能力之内的事。这种损害的结果则与攻破了哪个密钥有关。

注意，由于不正确地使用，DSA 密钥也可被攻破。DSA 计算要求为每个签名产生一个随机量 k。这个数字必须保密，如果泄露或者压根就不是随机生成的，那么攻击者可以彻底攻破 DSA 密钥。

攻破 CA 的密钥将会导致所有信任那个 CA 的代理均被攻破。尽管与 CA 已经颁发的证书对应的单个私用密钥仍然是安全的，但是这样丝毫没有影响，因为攻击者可以实施一种主动攻击，只需通过使用合适的 CA 密钥给自己签发一份证书就能冒充任何客户端或服务器。

攻破一个服务器的密钥将使服务器处于主动攻击之下。我们在这一章的前面已经讲述了这种情况。

攻破客户端的密钥可以让攻击者冒充这个客户端。注意并不能使攻击者接管客户端已经创建的连接，但却允许他初始化自己的连接，就像那个客户端一样完成客户端认证。

密钥确立算法

彻底攻破一种密钥确立算法意味着攻击者可以恢复任何以那种算法加密的连接的

master_secret，也就能够实施 5.9 节所提到的任何一种攻击。与之类似，如果密钥确立密钥（key establishment key）被攻破了（即便算法是高强度的），那么只要会话使用该算法所确立的密钥，攻击者就能恢复该会话 master_secret，并实施在掌握 master_secret 情况下的任何一种攻击。当密钥为静态密钥时，情况尤其糟糕，因为那意味着到给定服务器的所有通信都被攻破了。图 5.9 总结了泄露各种密钥所带来的后果。

密钥种类	攻破后的后果
CA	攻击者能够冒充任何信任这个 CA 的代理
密钥确立（静态密钥）	彻底攻破到服务器的所有通信
服务器认证（临时密钥）	攻击者能够在握手时对任何会话实施主动攻击
密钥确立（临时 RSA）	彻底攻破所有使用这个密钥交换的数据（通常是成百上千的连接）
密钥确立（临时 DH）	彻底攻破所有使用这个密钥交换的数据（通常为单个连接）
客户端签名密钥	攻击者能够冒充该客户端

图 5.9 对私用密钥被攻破的后果的总结

撤消

正如我们在第 1 章所看到的，当私用密钥被攻破时，某些 CA 会发布 CRL（证书撤消列表），并可以根据列表来检查某个证书是否已经被撤消。那么问题就是如何将 CRL 发给最终用户呢？一些协议，如 S/MIME 允许 CRL 和证书在消息中一起传送，而 SSL 不允许这样。代理必须以某种未定义的形式获得 CRL，这常常意味着什么也不做，它使得攻破密钥对 SSL 来说尤其成问题。

在 Goldber 和 Wagner [Goldberg1996] 指出 Netscape 的 PRNG 抽取种子的弱点之后，已经使用那种 PRNG 产生其私用密钥的服务器都不得不重新产生密钥对。但由于 CRL 并没有被广泛部署，因而使得四处大量流传着包含弱强度密钥的证书，除了等着它们过期以外，没什么好办法能让它们停止使用。

加密算法

在 SSL 中，算法强度最高的要数加密算法。SSL 最强支持 3DES，它的有效密钥长度为 112 位，承受着最知名的各种攻击。此外，DES 比起其他任何公开加密算法都经受了多得多的分析，所以不大可能再发现针对 3DES 的好的攻击。SSL 最弱支持 40 位的 RC4，拥有几台高端 PC 的攻击者就能将其攻破。

如果攻击者设法恢复了通道一方的加密密钥（如 client_write_key），那么就可以读取使用那个密钥加密的所有通信数据（从客户端发往服务器的通信数据），但是攻击范围仅限制在数据的保密性上。

client_write_key 和 server_write_key 是根据 master_secret 独立导出的，因此掌握了 client_write_key 不足以恢复 server_write_key。与之类似，你无法从任何给定的加密密钥追溯到 master_secret，因为 master_secret 要比它产生的加密密钥大得多，也就是说 master_secret 与加密密钥之间是一种多对一的关系。

类似的，保密性与消息完整性是独立提供的。即便攻击者攻破了加密环节，他也只能实施被动的嗅探攻击。也就是说，他能够恢复加密数据，但不能使用这种信息伪造通信数据。要想做到那一点，他就得攻破 MAC 算法。

攻破加密密钥的影响还会受到偶尔再握手的限制。即便没有进行新的密钥交换，仅仅刷新随机值也会产生新的加密密钥，从而迫使攻击者再次对密钥进行攻击才能读取新的通信数据。注意，由于 CBC 反转的原因，如果你使用分组加密的话，则每 2^{23} 个分组左右进行一次再握手是非常重要的。注意在再握手的时候执行新的密钥交换没有什么意义，因为大多数对密钥交换算法的攻击都会破坏整个私用密钥而不仅仅是一次交换而已。

摘要算法

攻破摘要算法包含有几个意思。一般来讲，由于消息摘要会破坏信息（消息空间比摘要空间大得多），所以给定摘要不可能恢复消息。然而，目前还不知道是否能够产生摘要值为给定值的消息，尽管就当前使用的摘要算法而言，还不知道如何做到这一点。如果存在此种算法的话，那么就会彻底攻破这种摘要算法。

若能够产生具有给定摘要的消息，则后果就是非常糟糕的，因为这样就有可能伪造证书。攻击者开始先提供一个具有有效签名的证书并使用他选择的摘要值为同一个值的名字和公用密钥产生一个新的证书。

设想你使用一种安全的摘要算法对证书进行签名，但是双方都接受一种有问题的摘要来对消息进行保护。这不会必然导致系统被攻破。首先，SSL 使用组合的摘要算法（既有 SHA-1 又有 MD5）来保护握手，所以攻击者需要同时攻破这两种算法才能对握手实施主动攻击。因此，即便没有了消息完整性，你或许还有保密性。

此外，据信 HMAC 是不受某些形式的底层摘要攻击影响的[Krawczyk1996]，因此攻破部分摘要算法甚至不会产生消息完整性的问题。例如，摘要中的冲突并不会导致基于那种摘要的 HMAC 出现缺陷。

还有另外一种情况值得一提。由于 DSS 签名只是在 SHA-1 摘要上计算得出的，因而如果 SHA-1 有问题的话，那么使用 DSS 签名的任何临时密钥交换都会暴露于主动攻击之下，因为攻击者可以伪造 ServerKeyExchange。图 5.10 总结了攻破对称算法的影响。

算法	攻破的后果
任何摘要算法	攻击者能够使用那种算法代替 CA 来造证书
SHA-1	攻击者能够对任何使用 DSA 保护的内容实施主动攻击
任何加密算法	丧失保密性，完整性不受影响

图 5.10 对攻破对称算法影响的总结

最薄弱环节原则 (The Weakest Link Principle)

尽管 SSL 提供了对握手的检查以阻止中间人降级攻击，但这并没有提供全面的保护。因为所有的安全都是基于 master_secret，SSL 实现不会比它所支持的最弱的密钥确立机制更安全。设想一个客户端支持 RSA，但是愿意接受使用 512 位 RSA 密钥签名的证书。它还愿意接受具有更长密钥的证书，但是并不区分这两种证书。

尽管该客户端愿意接受更长的密钥，但因为客户端愿意接受使用较弱密钥签名的握手，

所以一个能够攻破 512 位 RSA 密钥的攻击者就能攻击该客户端建立的连接。一旦攻击者能够攻破一方接受的密钥确立（或签名）算法，他就能够实施中间人攻击。注意，如果要求客户端认证的话，攻击者还需要伪造出服务器将要接受的签名。

加密算法不必有这种担心。握手中的检查可以防止攻击者降低握手的级别来支持较弱的加密算法，只要密钥确立机制是高强度的就行。

5.15 已知的攻击

尽管还没有已知的针对 SSL 本身的好攻击，但却存在多种针对特定实现的有效攻击。下面几节的内容将讲述这些攻击以及所需的应对措施。

尤其值得一提的是计时密码分析 (timing cryptanalysis) 和百万消息攻击 (million message attack)。计时密码分析是 1996 年，在 SSLv2 被广泛部署和 SSLv3 已经发表之后开发的。百万消息攻击是在 SSLv3 被广泛部署和发表 TLS 之后开发的。据说还没有针对生产服务器使用过这两种攻击，而且甚至不知道是否可行，但是细心的实现者至少应当知道到它们的存在以及适当的应对措施。

5.16 计时密码分析

1996 年，Paul Kocher 发表了一种他称做计时密码分析的技术。根据所使用的密钥和数据，这种技术依赖于对完成密码操作所花费的不同时间的观察。这种攻击实际上是一种通用技术，必须针对目标密码系统加以剪裁。Kocher 在他的原始论文中提供了 RSA、DH 和 DSS 的例子。

攻击概述

Kocher 在他的论文中强调了针对公用密钥密码系统的计时攻击。为了实施计时攻击，攻击者需要测量被攻击方使用其私用密钥的时间。有了足够多的采样（对于 1024 位的 RSA 密钥大概要有 2500 次采样），就能恢复整个密钥。这种攻击的细节超出了本书的范围。一言以蔽之就是 Kocher 描述了如何针对 RSA、DH 和 DSS 实施计时攻击。

自然，由于存在其他系统活动干扰计时，所以测量中会存在一些噪声。Kocher 描述了如何滤掉度量中的噪声，但是这种技术要求更多的采样。为了使效率达到最大化，攻击者应当能够获取指定操作的具体时间，并使增加的噪声尽可能的少。

最方便的情况就是攻击者与被攻击者同处一台机器，而且能够直接对被攻击者进行度量。这种优越的环境是很难找到的，但是像 SSL 这样的网络协议经常可以让攻击者通过观察网络通信来判定同样的信息。

可应用性

大家回想一下，服务器的私用密钥对攻击来说是最有价值的密钥。它暗中遭受计时分析吗？是的。考虑第一个静态 RSA 的情况。收到 ClientKeyExchange 与发送服务器的 ChangeCipherSpec 消息之间的时间主要由三个任务瓜分：对 pre_master_secret 进行解密，使

用密钥导出函数创建密钥以及处理客户端的 Finished 消息。

因为密钥导出与 Finished 验证所花费的时间大都是恒定的——更重要的是，相对于服务器的私用密钥操作来说是随机的——可以将其当作噪声来处理，并通过足够多的采样将其过滤掉。分离出所有这些问题之后，攻击者就会对私用密钥操作花费的时间有所了解并使用这种信息来攻击服务器的私用密钥。

临时密钥资料也有类似的弱点。尽管攻击者无法取得足够多的采样来攻击临时密钥，但是他可以攻击静态签名密钥。服务器收到 ClientHello 和发送 ServerKeyExchange 之间的时间是由 ServerKeyExchange 中计算签名来决定的。如果使用的是 DH 而且为每个会话产生一个新的 DH 密钥，那么这就提供了一种随机噪声的来源，但是这种东西可以被滤去。临时 RSA 通常针对每个会话重复使用相同的临时密钥，从而使得这种模式尤其易受攻击。

注意，这种攻击不要求攻击者是主动的，只要通过监视被攻击者的网络连接就能搜集所有这些信息。如果攻击者实施主动攻击的话，采样就可以更少而且可以选择被攻击方解密的密文。这种计时攻击确实要求攻击者对被攻击方的 SSL 实现有相当具体地了解，但是做到这一点并不难，通过下载源代码（如 OpenSSL）或对二进制文件进行逆向工程分析都可以达到此目的。

目前还没有出现有关针对运行服务器实施恶意计时攻击的报道。不管怎样，许多 SSL 实现都包含了增加这种攻击难度的应对措施。

● 应对措施

最明显的措施就是简单地以随机时间减慢所有的操作。但出于性能的考虑，这并不可取，而且效果也不怎么样。这完全是因为噪声是随机的，如果攻击者搜集了足够多的采样，他就可使用基本的信号处理技术滤去噪声。

另一种方法就是让所有的操作都花费恒定的时间。这在硬件中是可行的，但在软件中实际上却无法做到，而且它也具有不希望的性能影响。

最流行的做法就是使用至盲（blinding）技术。被攻击者将数据转换成以这样一种方式对其进行签名（或解密），即让私用密钥现在所操作的数据对攻击者来说一无所知。然后被攻击者再对结果执行反至盲（unblind）来获得原来在没有使用至盲技术时获得的结果。

5.17 百万消息攻击

1998 年，Daniel Bleichenbacher 发表了一种被称之为百万消息攻击的攻击[Bleichenbacher 1998]。这种攻击实际上是针对使用 PKCS#1 的 RSA 的，但是所有使用 RSA 的 SSL 版本都有可能受到它的影响。这种攻击的结果允许攻击者恢复给定会话的 pre_master_secret。我们假设攻击者捕获了包含相关加密 pre_master_secret 的 ClientKeyExchange 消息，他通过发送一系列选择的密文给服务器来做到这一点。也就是说，他产生一系列的消息并观察服务器的响应。根据这些信息，他就能恢复 pre_master_secret。

● 攻击概述

攻击像下面这样进行。攻击者基于加密的 pre_master_secret 产生一系列消息，然后选择

一系列整数 s 并计算：

$$c' = cs^e \bmod n$$

这里 c 是加密的 pre_master_secret，而 e 是 RSA 公共指数。

然后攻击者使用被攻击者的服务器作为启发器（oracle）：使用这些消息对其进行刺探并检查响应。攻击者与服务器初始化握手并在其 ServerKeyExchange 消息中发送 c' 。当服务器对 c' 解密时，他就恢复一条新的消息 m' 。可以从数学上证明 m' 就是 ms ，这里的 m 就是原来的消息。然而，由于服务器没有给我们发送 m' ，这种信息并不是直接有用。然而，仍然可以使用它来攻击 m 。

这种攻击的关键是攻击者需要能够用被攻击者的服务器作为获取有关 m' 信息的启发器，他可以用这些信息获得 m 的精确描述。这种攻击所使用的信息就是 m 是否具有正确的 PKCS#1 格式。

回想一下 PKCS#1（参见第 1 章的描述），它所包含的头三个字节分别是一个 0 和一个分组类型指示符。对于正常格式化的公用密钥加密数据来说，这些字节是 00 02。大约 2^{16} 个 s 值中会有一个产生正常格式化的 m' 。图 5.11 展示了一个 TLS 连接整个填充后的 pre_master_secret，其中包括版本号。

00	02	随机非 0 字节	00	03	01	pre_master_secret 的剩余部分（46 个字节）
----	----	----------	----	----	----	---------------------------------

图 5.11 TLS 格式的 pre_master_secret

攻击通过找到一系列产生符合 PKCS#1 明文的 s 值来进行。可以使用这些值缩小原来消息 m 的范围，直到完全判定为止。做到这一点的具体细节超出了本书的范围，但一言以蔽之就是大概要需要 2^{20} 条消息（这个数字大约是一百万，也就是百万消息攻击名称的由来）。

可应用性

这种攻击有赖于攻击者能否判断出正确与不正确格式消息之间的差别。对攻击者来说方便的是，在发现这种攻击以前编写的实现会在遇到不正确格式化的消息时发送一条警示（然后关闭连接）。从而使得对于这样的服务器可以非常容易地实施攻击。

注意，SSL 格式的 pre_master_secret 包含有三种而不是一种可验证的信息。除了 PKCS#1 格式以外，pre_master_secret 必须是 48 个字节长而且头两个字节必须为版本号。随机挑选的消息使所有这三种信息都正确的概率低于 2^{-40} ，因此一种完全相同的方式拒绝所有这些错误的实现不会遭受这样的攻击。

上面那句话的关键词是“以完全相同的方式”。根据实验判定，一些实现会拒绝所有这些情况但行为却各不相同。这些实现仍然受制于这种攻击。

还没有已知的针对工作服务器使用这种攻击的例子，但是 Bleichenbacher 说他已经对自己的一台服务器进行了试验而且成功地恢复了 pre_master_secret[Bleichenbacher1999]。

应对措施

最广泛采用的应对措施（而且是 RFC-2246 中推荐的措施）就是进行所有三项检查但不发送警示。如果检测到了一种错误，实现就使用随机数来填充 pre_master_secret 并像往常一样处理。

样继续进行握手。当收到客户端的 Finished 消息并解密为垃圾信息时，服务器会抛出一个警示，这种行为就在 m' 格式化正确但 `pre_master_secret` 错误的情况下所表现出的行为。

注意，将 `pre_master_secret` 设置为常量是不行的。那样的话，攻击者就可以使用那个值作为 `pre_master_secret` 并检查 Finished 消息是否是可接受的。

一种更好的解决办法是将 PKCS#1 替换为一种不同的，对明文遭受的任何破坏都敏感的填充算法。如果以这种方式填充明文，那么这种攻击使用的所有刺探消息都会自动被拒，因为它们没有通过完整性检查。PKCS#1 版本 2[kaliski1998a] 中已经包含了最广泛接受的范例，最优非对称加密填充（Optimal Asymmetric Encryption Padding）[Bellare1995]。不幸的是，OAEP 填充与 PKCS#1 版本 1 填充不兼容，因此让 SSL 转移到 PKCS#1 版本 2 上的动力不大。

5.18 小-子组攻击 (Small-Subgroup Attack)

在一些特有的条件下，攻击者有可能攻破用于大量密钥交换的静态 Diffie-Hellman 密钥，这种攻击通常被称做小-子组攻击[Lim1997]。

攻击概述

这种攻击源于一种改善 DH 性能的优化。回想一下 DH 共享密码的计算： $ZZ = Y^x \bmod p$ 。 X 越小，计算的速度越快。选择太小的 X 会使安全受到损害（ X 应当约为你想生成的最大密钥的两倍），但是比 1024 位（ p 典型的大小）小很多的 X 仍然相当的安全。

ANSI X9.42 标准 [ANSI1998] 中推荐的选择为创建 p 并使得 $p = jq + 1$ ，这里的 q 是一个大素数，然后将 X 选为比 q 小的数。必须将 g 选为产生数量为 q 的组。

在这种情况下，如果攻击者所选公用密钥合适的话，就能获得有关 X 的信息。选择使 $Y < q$ 的 Y 后，如果能够说服被攻击者给他发送 ZZ 的话，他就能获得有关 X 的信息。通过足够多的采样，他就可以恢复 X 的全部内容。这至少要尝试一百次。实际上让被攻击者返回 ZZ 并不是必须的，仅简单地返回有关解密成功或失败的信息就足够了。

可应用性

小的子组只对数量非常少的，使用 Diffie-Hellman 密钥来加密多条事务并使用小指数(X)的系统来说才构成问题。因为几乎从不使用静态 DH，这实际上只是指那些使用临时 DH 密钥来保护事务的实现，且在这些实现中也只是其中的一些使用小 X 。

应对措施

存在多种可用的应对措施。最简单的就是对每个事务都使用新的密钥来保护。这还有提供 PFS 的优点。

即使必须使用静态 DH 密钥，也有几种可用的应对措施。最简单的就是选择长指数，这样就可以避免该安全缺陷。选择使得 $p-1=2qj$ 的 p 会减少风险，以至于只泄露了私用密钥的单个位，这对于大多数用途来说已经足够了。作为一种特殊情况，如果 $j=1$ 那么 p 就被称做高强度素数。[Kaliski1998b] 中描述了一种更为复杂的称做相容余因子取幂（compatible cofactor exponentiation）的技术。

5.19 降级使用出口模式

即便双方都支持 1024 位的 RSA 密钥，一个可以分解 512 位 RSA 密钥的攻击者也能够将客户/服务器对降级为 512 位的临时 RSA 模式。下面的攻击在 [Moeller1998] 中进行了描述，这种攻击要求双方还要支持出口模式而且攻击者能够分解服务器的临时 RSA 密钥。

攻击概述

如图 5.12 中的描述，这种攻击要求攻击者作为中间人。他截获客户端的 ClientHello 并去掉除出口加密算法以外的所有内容并发送一条包含其临时 RSA 密钥的 ServerKeyExchange 消息，握手嗅探 ClientKeyExchange。因为他能够分解服务器的临时 RSA 密钥，所以能够恢复 master_secret。然后攻击者伪造新的 Finished 消息，一个发给服务器以反映修改过的 ClientHello，一个发给客户端以反映原来的 ClientHello。一旦握手完成，服务器直接地嗅探通信数据就行了。由于他已经恢复了 master_secret，所以连接被彻底攻破。

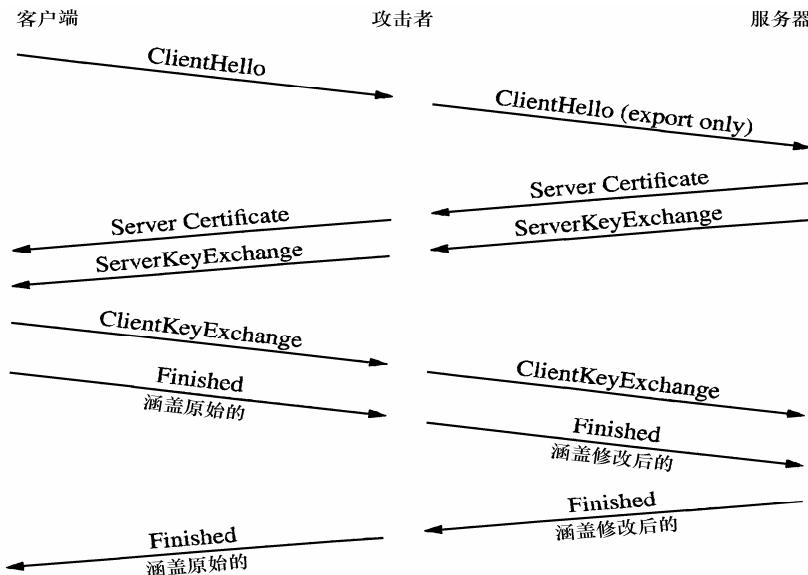


图 5.12 降级到 512 位的临时 RSA

可应用性

这种攻击要求客户端与服务器支持高强度的加密套件，但愿意协商使用出口加密套件。如果双方都只支持高强度的加密套件，那么这种攻击就不会成功。如果任何一方只支持出口加密套件，那么它们还是会协商使用出口算法。此外，攻击者必须能够在生命期（lifetime）内分解临时 RSA 密钥。因此，频繁重新产生临时 RSA 密钥的服务器是安全的。然而，随着质因分解技术的进步，安全的生命期将变得越来越短。

应对措施

最简单的应对措施就是修改协议，使 ServerKeyExchange 上的签名覆盖握手更多的部分，

包括 ClientHello。这样就可以让客户端检测出攻击者对 ClientHello 的修改。这项改动还可以加强 SSL 抵御类似基于降级密钥交换算法的攻击。

然而，不改动协议也有可能防止这种攻击。临时 RSA 加密套件早就过时了。它们已经被替换为 EXPORT_1024 加密套件，而且现在还可以出口高强度的加密套件。因此，除了保持与老实现的互操作性之外，不再有任何支持它们的真正理由。

在大多数情况下，用户可以安全地将自己的浏览器配置为只支持高强度的加密套件。几乎所有的服务器都支持 1024 位的 RSA，因此这样做不会产生互操作问题。服务器管理员可以通过频繁地重新产生临时 RSA 密钥来保护自己，从而实现在生命期内无法对其进行分解。

一种类似的攻击

[Schneier1996b] 中描述了一种类似的攻击。如果客户端支持 DH 和 RSA 而服务器支持 DH，那么攻击者就能篡改 ServerHello，使客户端相信服务器的临时 DH 密钥实际上是 RSA 密钥。如果客户端不小心的话，它就会将 DH 的素数 p 解释为 RSA 的模，而将 DH 的底解释为 RSA 的指数。根据这种 ClientKeyExchange 能很容易地恢复 pre_master_secret。

只有客户端相当大意时，这种攻击才会成功。首先，服务器想要使用一个比 512 位还长的 DH 模，这种情况下，客户端在看到那个“模”有可能超过 512 位时应当终止握手。为了履行出口约束，这项检查是 SSL 所要求的。其次，既然包含了服务器的公用密钥 Y ，客户端就必须不管 ServerKeyExchange 包含有比其所应当包含的还要多的数据的事实。所言的实质就是客户端与服务器应当仔细检查所有的协议值。

5.20 总 结

本章提供了有关 SSL 安全属性的描述。我们从一些安全使用 SSL 的一般指导原则说起，然后具体描述了各种潜在的攻击以及如何加以应对。

master_secret 就是一切，攻破了 master_secret 就攻破了整个协议。了解 master_secret 的攻击者能够读取通信数据并随意伪造消息。

保护服务器的公用密钥。在普通 RSA 模式和静态 DH 模式下，攻破服务器的公用密钥就会导致 master_secret 的攻破。即便使用的是临时模式，拥有服务器公用密钥的攻击者也能冒充服务器并实时地对通信进行攻击。

良好的随机性是根本。如果任一方都没有使用安全的随机数发生器，那些协议就有危险。在产生诸如公用密钥这样的长期密码的时候，好的随机数尤其重要。

只拥有一份证书并不算行。如果你关心另一方的身份的话，不仅要检查它具有的有效证书，还要检查证书与所期望的另一方的身份一致。

使用足够好的算法来保护你的数据。SSL 支持具有各种密钥长度的各种各样的算法。选择一种攻破它所花费的代价远高于数据价值的算法。

没有已知良好的针对 SSL 的攻击。然而，存在有多种草拟的针对特定实现的攻击。细心地实现就能防止这些攻击。

6

SSL 的性能

6.1 介绍

本章讨论 SSL 的性能属性。与前面的章节一样，我们将讨论划分为两个主要部分。第一部分从总体上讨论 SSL 的性能属性，广泛观察各种环境下各种模式的性能。本章的第二部分则更详细地描述了同样的内容，通过使用带有测量数据的网络跟踪信息以及插入性能剖析代码的实现精确描述了运行时间的占有情况。

6.2 SSL 速度慢

最常听到的来自服务器管理员有关 SSL 的抱怨就是速度慢。这种批评并不是无中生有，根据所用协议、服务器硬件，还有网络环境的不同，SSL 连接可比普通 TCP 连接慢上 2 到 100 倍。

这种性能开销直接转变为运做成本的增加。因特网服务器通常即便是在使用非保密协议的时候也负载沉重。运算负荷大的安全协议必然意味着要购买和操作更多的服务器才能维持同等的服务质量。当原来的服务只在一台服务器上运行时，这种转变尤其困难。如果安全考虑促使要去操作多台机器的话，就意味着要安排服务器之间的数据共享，而对于原来的情况来说则是不必要的。

不幸的是，某种程度的性能衰减在很大程度上是无法避免的。加密运算开销昂贵，而简单将数据发送到网络上与先加密再进行发送之间的差别经常不可小觑。然而，经过仔细的系统设计和配置，将 SSL 的性能影响降至最小经常是可能的。

6.3 性能法则

在讨论 SSL 性能之前，我们首先讨论一些通用的系统性能法则，并特别强调与网络协议有关的内容。这提供了理解 SSL 各个部分对协议整体性能所带来的影响的知识。

● Amdahl 定律

性能调校最基本的法则就是[Hennessey1996]中讲述的 Amdahl 定律。粗略地说就是，优化所带来的速度提升等于提速的改进乘以提速代码耗费 CPU 时间所站的比例，如图 6.1 所示：

$$\text{整体加速比} = \frac{\text{改进前执行任务的时间}}{\text{改进后执行任务的时间}} = \frac{1}{(1 - \text{改进部分所占执行时间的比例}) + \frac{\text{改进部分所占执行时间的比例}}{\text{改进部分的加速比}}}$$

图 6.1 Amdahl 定律

因此，判定系统的那些部分耗费的时间最多就非常重要，因为它们代表首要的优化目标。一旦知道了耗费时间的地方，我们就有了清晰的策略：努力改进系统中这些部分的性能，要么使之运行得更快要么就少用它们。

作为一项推论，应当清楚的是，即便对于开销非常昂贵的操作也不值得提高它们的速度，除非它们频繁发生。就网络协议而言，意思就是说我们首要关心的是随同每个事务或协议消息发生操作。我们不考虑系统启动和关闭所耗费的时间，尽管这些时间有可能等同于几百次事务所花费的时间，因为在服务器的生命期中它们微不足道。

● 90/10 法则

如果系统中的所有部分均耗费等量的时间，Amdahl 定律就没什么用处。在这样的系统里，我们得对系统中的大部分内容进行优化才能看到性能改进。这种情况并不常见，相反，大多数系统的时间都花费在执行少数几项任务之上。

一条有用的经验法则，90/10 法则指出，一个程序 90% 的执行时间花在 10% 的代码中。将 90/10 法则与 Amdahl 定律结合起来，我们可以发现，对系统的那一部分内容进行优化可以极大提升系统的性能。而关注其他的 90% 几乎没有任何补益。

该法则的一个特例就是称做“瓶颈”的东西。当系统执行许多种不同的操作，但在所有这些操作都需要通过中间某个环节或完成某项操作的时候就会出现瓶颈问题。每当你发现某一点之前出现拥塞而过了这一点又变得通畅时就应怀疑在该处出现了瓶颈。

● I/O 开销

程序性能调校通常是在剖析器（判断程序各部分耗费 CPU 时间的软件）和测量软件的帮助下完成的。这样就可以让调校代码的人识别出耗费大量 CPU 时间的代码部分。总体上执行大量输入/输出，尤其是网络协议的系统增加了另一项可以进行调校的内容：I/O 开销。必须在其与 CPU 开销之间进行权衡。

考虑网络上两台机器之间传输文件的问题。在大多数情况下，限制传输性能的因素并不是传输和接收程序给 CPU 造成的负担，而是两台机器之间的网络性能。

在每秒 28.8 千位的（Kb/s）的传输通道上传输一个 10MB 的文件大约要花费 3000 秒。只要任一方的机器都满足最低的处理能力标准（实际上任何可用的商用机器都可以完成这样的工作），传输的速度就是恒定的，因为限制因素为通道的通信速率。那么在这种情况下，为了减少传输时间，在传输之前或传输过程中花些 CPU 处理能力来对数据进行压缩是值得的。几乎任何机器压缩数据的速度都要比 28.8Kb/s 快上许多。

编写这本书时所使用的膝上电脑大约每秒钟可以压缩 1MB。典型文本文件的压缩比例

为 2~4 之间，因此加密时进行压缩将会把传输时间削减至 750 到 1500 秒之间，将有效数据传输速率提升至大约 100Kb/s——这是相当大的改进。注意，瓶颈仍然是网络，机器压缩数据的速度可以轻而易举地超过网络传输的速度。实际上压缩速度如此之快以至于几乎所有的商业调制解调器都自动在传输时进行压缩。

现在考虑同样的两台机器以快速以太网进行连接的情况。这样的网络可以大约传输 100MB/s。在这种情况下，传输同样的文件大约要 1 秒种。与之相比，传输前压缩文件大约要花费 10 秒种。于是这样情况就会更糟：现在的瓶颈是 CPU 而不是网络。

正如我们所见到的，我们可以在 I/O 带宽与 CPU 之间权衡使用。这样一种系统的最优设计与配置有赖于相对可用的 CPU 与 I/O 带宽。如果你的系统部署在各式各样的环境中，那么在决定使用何种方案之前侦测一下所需的 I/O 与 CPU 就非常有益了。

延迟与通量

“决不要低估装满磁带的旅行车的容量。”

——Dr. Warren Jackson

大家回想一下前一节中有关数据传输的问题。这一次让我们考虑一个 100MB 的文件。若使用压缩的话，则在 28.8Kb/s 的通道上传输这个文件大约要花费 3 个小时。对于这么大的文件，直接将其写到磁带上（花费大约 10 分钟即可完成）并在机器之间传递（或邮递）会更有效率。

如果邮寄的话，将文件从一台机器转到另一台机器要花费 2 到 5 天时间。与 3 个小时相比看上去可不怎么样，但是设想我们要传输许多这样的文件，通过调制解调器，我们一天可以传输大概 8 个文件。如果邮寄的话，迁移的速度可以与我们向磁带上拷贝的速度一样快：大约每天 100 个文件。不错，第一个文件要呆几天才能到，但是随后就会以恒定的速率到达。

这个例子演示了延迟与通量之间的差异。延迟就是从开始到结束，处理一个给定的事务所花费的时间。通量是一段时间之内能够维持处理的事务总量。如你所见，它们并非总是相关的。

为了更清楚地说明这种差异，假定我们通过联邦快递而不是普通邮递邮寄磁带。这样就将延迟缩短为一个晚上的时间，然而这对通量并未造成影响。作为对比，如果我们再购买一台机器来写磁带的话，我们就能够在不影响延迟的情况下增加通量。

稍加考虑，就能够轻易地列举出通量与延迟组合所产生的每一种情况的例子。图 6.2 列举了这样一些例子：

	低延迟	高延迟
低通量	调制解调器	传呼机
高通量	局域网	卫星通讯

图 6.2 各种类型的延迟/通量比较

服务器与客户端的比较

就一般规律而言，网络系统都是客户端多服务器少。在这样的系统中，客户端速度减慢一般表现为延迟的增加，而服务器端速度减慢则表现为延迟的增加与通量的减少。

回想一下我们熟悉的网络文件传输的例子。现在稍微做一下改动，我们有一台服务于大

量客户端的服务器。该服务器具有 T1 级的因特网连接（大约能够承载 1.5Mb/s 的流量），每个客户端使用一台 28.8Kb/s 的调制解调器。在这样的环境中，服务器大约可以一次为 50 个客户端提供服务并保持其网络连接满负荷运行。与前面一样，传送一个 1MB 大小的文件需要大约 300 秒时间。这样，我们的通量大致就是 1.5Mb/s，而延迟为 300 秒。

现在设想将客户端增加一倍。服务器对其同等对待，因此客户端现在所获得的数据量是先前的一半。这使延迟增加一倍，达到 600 秒。然而，请注意，尽管服务器为每个客户端服务需要两倍的时间，但是我们同时服务的客户端数目却是先前的两倍，因此通量并未受到影响。

接着设想我们将客户端的网络连接速度削减一半，即 14.4Kb/s。同样，延迟加倍至 600 秒，但是以这种速度我们可以服务两倍的客户端，因此通量同样不受影响。

最后，考虑将服务器的因特网连接速度削减一半的影响。如果我们维持客户端数目不变，那么就能以前一半的速率为其提供服务，从而使延迟增加一倍而通量削减一半。当瓶颈是 CPU 而不是 I/O 的时候，也会出现这样的效果。慢客户端可以将服务器资源留给其他客户端，而慢服务器只会使系统整体速度下降。

注意，CPU 与带宽并不是服务器惟一匮乏的资源。每个客户端都消耗一些服务器资源，尤其是内存，因此当客户端达到相当数量时，服务器为它们同时提供服务就会超载，即便有充足的 CPU 和带宽可用也一样会超载。

6.4 加密的开销昂贵

在本章第一部分，我们肯定了 SSL 速度慢的事实。在第二部分，我们引入了理解影响性能因素所必需的概念。这一节，我们开始将这种一般性的知识运用到 SSL 上。正如在一节所暗示的，SSL 之所以慢的首要原因就是加密，尤其是公用密钥加密所需的大数运算极为消耗 CPU 资源。

使用 gprof(1) 对 OpenSSL 进行剖析的结果相当清楚。考虑使用 TLS_RSA_WITH_3DES_EDE_CBC_SHA 的简单 SSL 连接。客户端与服务器上的绝大多数时间都花费在加密处理上。实际上，绝大多数时间都花在一种操作上，即 RSA 解密上。

值得将 SSL 连接的两个阶段，握手与数据传输分开来考虑。握手在每次连接时只发生一次，但开销相对昂贵。每条单独的数据记录的开销则相对小一些，但是对于涉及大量数据传输的连接来说，数据传输阶段的开销最终会大大超过握手的开销。尽管如此，数据传输的开销很大程度上是由于加密所造成的。

客户端与服务器之间握手的开销差别显著，但数据传输阶段是对称的。因此，本节的剩余部分将讨论客户端与服务器的握手，然后再讨论数据传输阶段。

服务器握手

握手在服务器一端的性能剖析相对简单一些。超过一半的 CPU 时间都花在一种操作上：对 pre_master_secret 的 RSA 私用密钥解密。排在第二位的较为悬殊，即计算 master_secret 以及根据 pre_master_secret 的密钥处理。

客户端握手

客户端的情况远没那么直白。客户端的操作比起服务器端的操作要快上许多。因此在许

多情况下，客户端的大部分时间都花在等待服务器的消息上。

一旦去除这种等待时间，我们就会发现两种开销最大的操作，即验证服务器证书和加密 `pre_master_secret`。尽管这两种操作均涉及 RSA 操作，但是它们是公用密钥操作，因此大约要比服务器执行的私用密钥操作快上一个数量级。所以，客户端主要承受两方面计算负担：RSA 计算与对服务器证书进行 ASN.1 解码。

瓶颈

显然，在这种情况下，服务器的性能限制会从整体上限制系统的性能。如果使用客户端认证的话，这种状况就会有所改变。那样的话，客户端多少会像服务器一样完成 RSA 私用密钥操作，而这就不会成为客户端的主导性开销。服务器几乎也就不会招致附加的性能开销。

同时值得指出的是，这种特定的工作划分其实只适用于 RSA。DSA 操作在客户端与服务器之间的计算更为对称，而且我们发现，在使用 DSS/DH 加密套件的时候，客户端与服务器之间的计算负载几乎是对称的。

然而，在大多数生产系统中，服务器仍然是瓶颈所在。首先，客户端认证和 DSS/DH 很少被使用。更重要的是，正如我们前面所讨论的，服务器是由许多客户端来存取的，而任何客户端只打开少量的连接。因此，即便是非常快速的服务器也会很快达到无法满足客户端速度的工作量。

数据传输

在数据传输阶段，存在两种相关的加密操作，即记录加密和记录的 MAC 计算。它们共同占据了数据传输的大部分开销。这些操作的相对份量有赖于具体选择什么样的算法和实现细节。如果选用了一种快速的加密算法（如 RC4），那么 HMAC 就会成为主要开销。如果使用了像 3DES 这样慢一些的加密算法，那么加密与 MAC 计算大概会平摊这部分开销，而记录的组装与拆分所占用的时间微不足道。

6.5 会话恢复

正如我们在前一节所讨论的，SSL 握手中的性能瓶颈为与握手本身有关的公用密钥加密操作。这些操作中的大多数都与交换会话密钥有关。对 `master_secret` 的加密和解密显然是直接相关的。然而要记住，验证服务器证书的唯一必要就是为了使用它来完成密钥交换。

因此可以断定，如果我们能够去除这些操作，那么握手的速度就会得到显著地提升。正如我们在第 4 章所讨论的，会话恢复就提供了那样行事的机会。由于恢复的会话使用前一次连接的 `master_secret`，所以我们可以省去昂贵的公用密钥计算。

不出所料，这使性能获得了极大的改进。在我们的测试记录中，使用 512 位的 RSA 密钥，在握手中可以得到大约 20 倍的性能改进。使用更长的密钥可以观察到甚至更大的改进。更重要的是，这种改进的结果几乎彻底减轻了服务器的负担。因此，在一种大多数会话都要恢复的环境中，我们有望看到通量上的相应改善。

显然，会话恢复可以极大地改善性能。然而，并不是在所有情况下都合适。相反，只有合理数量的客户端在合理的时间段内重新连接时才合适。

会话恢复的开销

会话恢复的开销几乎完全由服务器来承担。正如我们在前面指出的那样，客户端只与数量非常少的几个服务器进行交互，因此维护高速缓存的开销极小。然而，服务器需要为大量客户端保留缓存条目，因此负载也就相应地更多一些。

会话恢复主要消耗两种资源：存储可恢复会话的内存，以及检查缓存时花费的 CPU 时间。显然在会话恢复频繁发生的环境中，花费这些资源是值得的。但是在从不恢复会话的环境中，就是浪费。

会话缓存中表示可恢复会话的状态信息大约需要 50-100 字节的空间。这对于单个会话来说并不算多，但对于每分钟处理几百次事务的服务器来说，意味着一小时就是多少兆字节的数据。显然，即使恢复有所回报，对缓存的大小加以限制并相当频繁地使条目超时也是值得的。

存取缓存也需要时间。与之有关的问题就是必须在多个线程或进程之间共享缓存。这就意味着要对缓存进行加锁，读取内容后再进行解锁。加锁与解锁是相对耗时的操作。如果需要在进程之间共享缓存的话（许多 UNIX 服务器都是这样），每次对缓存的读写都要求上下文环境切换至核心状态，这样开销就更大。此外，如果缓存条目得以恢复使用的话，这种时间就可以忽略不计。但是，如果从未恢复会话，那么耗费的 CPU 时间就是不必要的负担。

这些讨论的意图就是对客户端来讲会话恢复几乎总是值得的。然而，对服务器来讲，只有在相当短的时间内存在大量重复的业务时才是值得的。即便在这种情况下，看看可否将协议转换成简单的保持连接打开，而不是去使用那种开销极小的握手恢复也是值得的。

6.6 握手算法与密钥选择

正如我们在 6.4 节所讨论的，SSL 连接的性能特性极大地依赖于所使用的算法。在握手阶段，这会极大影响握手的开销，同时也可能改变客户端与服务器上的相对负载。

RSA 与 DSA 的比较

在部署 SSL 系统的时候，签名算法的选择常常有赖于对知识产权或兼容性的考虑。然而，在性能是主要因素的情况下，RSA 是优于 DSS/DH 的选择。一般来讲，在用于进行验证的时候，RSA 要比 DSA 快得多，而在用于签名时速度相当，请参见图 6.3。然而这种开销主要是由客户端负担的，因此它会对延迟而不是通量造成影响。

算法	密钥长度	每秒签名操作	每秒验证操作
RSA	512	342	3287
DSA	512	331	273
RSA	1024	62	1078
DSA	1024	112	94
RSA	2048	10	320
DSA	2048	34	27

图 6.3 数字签名的性能 (Pentium II 400/OpenSSL)

然而，DSA 加密套件几乎总是运行于临时模式。这要求双方同时花费大量额外的工作来产生，签名和验证临时 DH 密钥。这会极大地降低通量。在实际应用中，DSA 加密套件

往往要比具有可比长度的 RSA 加密套件慢上 2 至 10 倍。

速度慢还是强度高，由你选择

随着密钥尺寸的增大，公用密钥算法的性能急剧下降。1024 位的 RSA 要比 512 位的 RSA 大约慢 4 倍，而具有更长密钥的 DSA 性能也会类似被削弱。因此，选择私用密钥的长度要在安全与握手性能上加以权衡。正如我们在第 1 章所讨论的，512 位对于珍贵的数据来讲肯定太短了。然而，768 位的密钥对于大多数商业事务来说很可能强度已经足够而且要比 1024 位的密钥快上许多。

临时 RSA

大家回想一下，当使用具有高强度位 (>512) 密钥和出口加密算法的 RSA 时，标准要求你使用 512 位的临时 RSA 密钥。从本质上讲，这给每一方都增加了一次 512 位的 RSA 操作（客户端为公用密钥操作，服务器端为私用密钥操作）。这对通量和延迟来讲都有一定的负面影响。

注意，双方仍要完成即便在不使用临时 RSA 时也要完成的所有操作。这些操作速度越慢，使用临时 RSA 所占影响的比例就越少。对于 1024 位的密钥来说，可望减慢 10%~25%。注意，这并不是安全或性能上的折中，因为 512 位的临时密钥强度既低，速度又比 1024 位的静态 RSA 慢：512 位的密钥要比 1024 位的密钥强度低许多，而且由于大量的事务都使用同样的 512 位密钥，所以 PFS（完美向前保密）也没有什么补益。

6.7 批量数据传输

算法的选择

大家或许还记得在处理记录时，加密运算耗费了大量时间。所选的 MAC 与加密算法会对系统性能造成影响，这一点并不奇怪。简而言之，使用 RC4 来获得最佳的性能，使用 3DES 来获得最好的安全性。RC4 要比 3DES 快大约 10 倍。随着 MD5 逐渐停止使用，摘要选择并没有什么灵活性。在大多数现实情况下，SHA-1 的速度足够快了，所以一般来讲它是一种好的选择。在现代系统中，使用 SHA-1 和 RC4 足以使大多数网络满负荷运作。

最优的记录大小

记录计算中令人吃惊计算量都对传输数据大小不敏感。不管是加密算法还是向网络上输出都存在大量的固定开销。因此采用小记录来传输数据就会导致糟糕的性能。在我们的测试系统上，最优尺寸看起来大概是 1024 字节。超出这一点，使用更大分组带来的好处就变小。如果进行批量数据传输，那么缓冲数据直至能够加密相当大的记录是值得的。

6.8 基本的 SSL 性能法则

非对称算法的选择。使用 RSA。

私用密钥的尺寸。使用最短的你认为安全的私用密钥，768 对于大多数应用来说足够好了。

对称算法的选择。使用 RC4 来获得最佳性能，使用 3DES 来获得最好的保密性。出口版本在使保密性打折扣的同时并没有使性能有所改善。

摘要算法的选择。MD5 相对 SHA-1 只提升了 40% 的性能。在大多数情况下，SHA-1 足够快了，要想保证安全的话就应坚持使用 SHA-1。

会话恢复。客户端总应使用会话恢复。服务器应当只在客户端在 5 到 10 分钟内就重新连接时使用会话恢复。

记录尺寸。尽可能使用最大的分组发送数据。

6.9 小结

获得良好性能的指导思想就是尽可能完成少量的操作。意思就是说将耗时操作减至最少，并尽可能加快这些操作的速度。正如我们所见到的，SSL 中的耗时操作绝大部分都是加密运算。会话恢复允许我们去除一定数量的此种操作，而选择合适的算法则会使我们把仍然不得不完成的操作的耗时减至最少。

本章第一部分的意图是为服务器或客户端管理员提供足够多的信息，让他们从 SSL 获得良好的性能。本章的剩余部分则是面向程序员的，目的是提供对 SSL 性能特性更完整地描述。我们重新更具体地研究了第一部分中的一些内容，并就 SSL 与 TCP 进行交互的一些关键环节进行了讨论。

6.10 握手的时间分配

以下几节详细描述了多种不同的 SSL 握手。这些章节中描述的数据是使用一种特殊改写的 OpenSSL 0.9.4 版本来收集的。测量器记录了产生、处理每条握手消息所花费的时间。

测试环境

该测试环境是一台运行 FreeBSD 的 300MHz Pentium。客户端与服务器是 OpenSSL s_client 和 s_server 程序修改后的版本。客户端与服务器均在同一系统上运行，因此网络延迟微乎其微。测量器并没有记录实际发送和接收网络消息或进程间上下文切换所花费的时间。

序列化

正如我们在第 3 和第 4 章所讨论的，SSL 消息的顺序完全由标准来描述。而且在特定时刻，服务器必须等待客户端的消息，反之亦然。OpenSSL 为了提高性能对网络输出进行了缓冲。在现代操作系统上，网络传输需要上下文切换至核心态，这样开销昂贵。因此产生许多消息并一次传输完毕常常会使计算成本低一些。传统的缓冲方案使用固定大小的缓冲区并在缓冲区满时将数据发送出去。就 SSL 而言，在等待应答之前传输最后一条消息时，刷新缓冲区是必须的。不然实现就会出现死锁。

图 6.4 在一次简单 RSA 握手（与我们在第 3 章展示的相同）中描述了这种行为。ServerHello、Certificate 和 ServerHelloDone 都具有相同的时间戳记，意味着它们是被同时传

送到网络上的。与之类似，客户端的 ClientKeyExchange 和 ChangeCipherSpec 也是同时进行传输的。

```
New TCP connection: speedy(3266) <-> romeo(4433)
1 0.0456 (0.0456) C>S Handshake ClientHello
2 0.0461 (0.0004) S>C Handshake ServerHello
3 0.0461 (0.0000) S>C Handshake Certificate
4 0.0461 (0.0000) S>C Handshake ServerHelloDone
5 0.2766 (0.2304) C>S Handshake ClientKeyExchange
6 0.2766 (0.0000) C>S ChangeCipherSpec
7 0.2766 (0.0000) C>S Handshake Finished
8 0.2810 (0.0044) S>C ChangeCipherSpec
9 0.2810 (0.0000) S>C Handshake Finished
10 1.0560 (0.7749) C>S application_data
11 6.3681 (5.3121) S>C application_data
12 7.3495 (0.9813) C>S Alert
Client FIN
Server FIN
```

图 6.4 缓冲 SSL 握手消息

然而，当以这种方式缓冲消息时，有可能出现一方等待已经由另一方输出的消息，但该消息仍然位于输出缓冲区中的情况。因此客户端与服务器中的处理有时是顺序发生而有时却是并行发生的，这要依赖于握手消息是否完全充满了缓冲区。如果缓冲区满了，消息就会被立即发送出去，在这种情况下，另一方的主机可以在产生下一条消息的同时并行处理这条消息。如果缓冲区没有满，那么就只有在发送方停止等待应答时才将消息发送出去，这种情况下，依序对消息进行处理。

用来产生这些跟踪信息的 OpenSSL 版本将缓冲区的大小从 1024 字节增加至 4096 字节，以便所有的消息都是按照顺序来处理的。这使得握手跟踪信息更加易读，而且可以让我们展示在不受上下文切换影响下使用的 CPU 时间。OpenSSL 中选用 1024 字节是随意的，其他的实现可以根本不使用缓冲。

在某些情况下，这种类型的缓冲会改善延迟，但是对于某些情况则会使情况更糟。我们将在 6.23 节讨论 Nagle 算法的时候讨论这个问题。

6.11 普通 RSA 模式

我们将要研究的第一种模式就是只对服务器进行认证的 RSA 模式。图 6.5 描述了处理各种消息的时间分配。划分为客户端处理的时间位于左边一栏，而划分给服务器的时间位于右边一栏，使用缩进来指示操作计量。所有的时间均以毫秒为单位。因此，读取服务器证书耗时 8.11 毫秒。这包括三次验证，每次耗时 2 毫秒。

一般来讲，RSA 操作有两种类型：私用密钥操作（数字签名和私用密钥解密）和公用密钥操作（签名验证和公用密钥加密）。这种区别非常重要，因为私用密钥操作比公用密钥操作慢许多。然而，所有使用给定密钥长度的私用密钥操作耗费的时间都差不多，使用给定

密钥长度的公用密钥操作也是如此。我们将在讨论 RSA 的过程当中使用这些术语，不再区分每项操作的具体类型。

正如根据 90/10 法则以及第 6.4 节中的讨论而期望得到的，大部分 CPU 时间都花在了三条消息的处理上：在客户端，处理服务器的 Certificate 消息以及产生 ClientKeyExchange 消息；在服务器端，处理客户端的 ClientKeyExchange 消息。其余的消息总共才花费了不到 1 毫秒的 CPU 时间。下面我们来依次讨论这三条重要的消息。

我们在初期收集到这些跟踪信息时，大部分时间都花在创建 ClientHello 消息上。这是由于 OpenSSL PRNG 一次性（按需）生成种子数据（seeding）的开销。在现实环境中，这是一种启动开销，因此我们将获取种子的操作移到了一段在握手之前调用的代码中。

Certificate (客户端)

Certificate 消息中包含服务器的证书链。为了对其进行处理，客户端需要解析和验证每个证书。主要思想就是执行 RSA 验证。在这里，证书链为三个证书长，因此需要三次验证。每次验证大约花费 2 毫秒，而其余的时间大部分都花在对 ASN.1 的语法分析上。

Client	Server
0.07 Write client_hello	0.15 Read client_hello 0.07 write server_hello 0.20 Write certificate 0.00 Write server_hello_done
0.05 Read server_hello	
8.11 Read certificate	
2.47 verify	
1.95 verify	
2.25 verify	
0.00 Read server_hello_done	
2.54 Write client_key_exchange	
2.26 encrypt_premaster	
0.15 Write finished	31.01 Read client_key_exchange 30.79 decrypt_premaster 0.00 Read finished 0.12 Write finished
0.00 Read finished	

```

Client: 10.95      Server: 31.59

```

图 6.5 只对服务器进行认证的 RSA 模式的时间分配（毫秒）

这里的证书链特别长。正常的证书链只有一个证书长（直接由根签发的证书），只需要一次验证。在这种情况下，处理这条消息所花费的时间大致与处理 ClientKeyExchange 的时间相同。还要注意这里的限制因素是 CA 密钥的长度，而不是服务器的密钥。在此阶段实际上还没有处理服务器的密钥。

作为一种服务器优化措施，可以把证书验证的结果缓存起来，以保存公用密钥验证阶段。这在证书链短的时候就没有什么意义，因为人们所碰到的服务器证书个数有可能非常多而无

法存储。然而，当使用长链的时候，就像在这种握手中一样，缓存 CA 证书常常是值得的。

s_client 默认提供对 OpenSSL 证书处理代码的回调。该回调向 stderr 输出许多证书调试信息。我们不得不切断这个回调，因为 I/O 会耗费大量的 CPU 时间，它们把测量结果搞砸的。

● ClientKeyExchange（客户端）

大家还记得第 4 章中讲到 ClientKeyExchange 包含以服务器的公用密钥加密的 pre_master_secret。这里的主要操作是 RSA 加密，因此限制性因素就是服务器公用密钥的长度。

● ClientKeyExchange（服务器）

并不奇怪，这里的主要操作是针对 pre_master_secret 的服务器 RSA 解密，它实际上花费了划给这条消息的所有时间。注意，该项操作花费的时间直接与服务器 RSA 密钥的尺寸有关。密钥越长，解密时间就越长。注意，这里的时间要比客户端完成的加密和验证时间慢得多。

注意，由于解密是一项服务器操作，因此它是 SSL 通量中的瓶颈。解密的开销有赖于服务器选择的密钥长度，从而使得服务器的密钥长度是该模式中最重要的性能限制因素。

6.12 带有客户端认证的 RSA

具有客户端认证的 RSA 模式其实就是带有一些附加步骤的普通 RSA 模式。从性能的角度来看，这种模式完成所有先前要在服务器上完成的工作而且还要在客户端执行这些操作。因此，客户端认证模式大大增加了客户端的负担，而服务器上的负载仅有微量地增加。图 6.6 描述了测量客户端认证的跟踪信息。从现在开始，跟踪信息中与以前相比发生改变的部分都用粗体来显示。

我们在普通模式下完成的所有操作仍花费大致等量的 CPU 时间。新增的耗时操作有客户端 CertificateVerify 消息的产生和服务器端对客户端 Certificate 和 CertificateVerify 消息的验证。

CertificateVerify 消息由一条 RSA 签名的消息组成。这里开销昂贵的步骤显然是 RSA 签名。前一节有关处理 ClientKeyExchange 消息的论述同样在这里适用。

在服务器端，对 Certificate 消息的处理与客户端对服务器 Certificate 消息的处理完全一样。也就是说，主要的开销是对数字签名进行验证。最后，服务器需要处理 CertificateVerify 消息。这主要包括一项公用密钥操作，即检查客户端的数字签名。

尽管客户端认证模式的总体计算开销大约是只进行服务器认证的两倍，但是这种开销不成比例地由客户端分担了。这里，服务器分担的开销大约是客户端的 20%。结果使用客户端认证极大地增加了延迟，但对通量的影响要小得多。总体上客户端认证增加了如下的内容：

- +1 客户端 RSA 私用密钥操作
- +1 服务器端 RSA 公用密钥操作
- +每个客户端证书都增加一次服务器端的 RSA 公用密钥操作

Client	Server
0.07 Write client_hello	0.15 Read client_hello

```

        0.17 Write server_hello
        0.49 Write certificate
0.06 Write certificate_request
        0.00 Write server_hello_done

0.05 Read server_hello
9.01 Read certificate
2.44   verify
1.94   verify
2.24   verify
0.18 Read certificate_request
0.00 Read server_hello_done
0.42 Write certificate
2.61 Write client_key_exchange
2.33   encrypt_premaster
30.56 Write certificate_verify
0.12 Write finished
          8.65 Read certificate
          2.38   verify
          1.97   verify
          2.24   verify
          30.72 Read client_key_exchange
          30.51   decrypt_premaster
2.23 Read certificate_verify
          0.00 Read finished
          0.14 Write finished

0.00 Read finished

Client: 43.075      Server: 42.66

```

图 6.6 RSA 客户端认证模式的时间划分（毫秒）

6.13 临时 RSA

正如我们在第 4 章中所讨论的，临时 RSA 使用一个短小的临时 RSA 密钥来加密 pre_master_secret。这同时有效地给客户端与服务器增加了一次短小的 RSA 操作。然而，正如我们将要在这一节看到的，这是通过增加某些操作并去掉其他的操作而实现的。为了做到这一点，我们需要查看每一条在临时模式下内容发生变化的消息。图 6.7 描述了一次使用临时 RSA 握手的跟踪信息。

Client	Server
0.07 Write client_hello	0.15 Read client_hello
	0.12 Write server_hello
	0.49 Write certificate
	30.45 Write server_key_exchange
	30.43 sign
	0.00 Write server_hello_done

```
0.05 Read server_hello
8.57 Read certificate
2.34   verify
1.94   verify
2.23   verify
2.23 Read server_key_exchange
2.20   verify
0.00 Read server_hello_done
1.12 Write client_key_exchange
0.86   encrypt_premaster
0.11 Write finished
5.81 Read client_key_exchange
      5.59   decrypt_premaster
0.00 Read finished
0.11 Write finished
0.00 Read finished
```

Client: 12.2

Server: 37.17

图 6.7 临时 RSA 模式的时间划分（毫秒）

ServerKeyExchange（服务器）

在普通 RSA 握手中这条消息甚至就没有出现过，因此不管包含什么样的加密处理，它都是新的，而且我们将会看到，它是一种非常耗时的操作。原因就是我们使用服务器的私用密钥对 ServerKeyExchange 进行签名，该密钥一般都很长（1024 位或更多）。

注意，那并不包括产生临时 RSA 私用密钥的时间。它在平台上的花费要花费 200-400 毫秒时间，开销太昂贵了以至于无法为每一个事务都完成这样的操作。通常只是一天左右产生一次临时密钥。因此，我们不将其计入每个事务的开销。

到目前为止增加的操作有：

+1 服务器端的长 RSA 私用密钥操作

ServerKeyExchange（客户端）

在客户端，我们需要处理服务器的 ServerKeyExchange 消息。这里惟一有趣的处理就是验证消息上服务器的签名。这给客户端增加了一项长公用密钥操作。

到目前为止增加的操作有：

+1 服务器端的长 RSA 私用密钥操作

+1 客户端的长 RSA 公用密钥操作

ClientKeyExchange（客户端）

普通 RSA 模式下也有这条消息，但是那里客户端执行的是长 RSA 加密。这里客户端则是简单完成 512 位的加密，速度极快。因此，我们增加了一次 512 位的操作并去掉了一次长 RSA 操作。

到目前为止的情况是：

+1 服务器端的长私用密钥操作

+1 客户端的 512 位 RSA 公用密钥操作

ClientKeyExchange（服务器）

正如我们上面所讲的，我们通过这条消息使用短 RSA 操作代替短操作。这显然也适用于服务器端。于是，最终的变化是：

+1 客户端的 512 位 RSA 公用密钥操作

+1 服务器端的 512 位 RSA 私用密钥操作

因为 512 位的操作大约比 1024 位的操作快 4 倍，所以这种差别并不是很大。在这里的平台上，客户端大约需要 2 毫秒，而服务器端为 6 毫秒。

6.14 DSS/DHE

下面的一系列跟踪信息都是使用 DSS/DHE 模式的握手。DSA 签名密钥与临时 DH 密钥均为 1024 位长。同以前一样，客户端与服务器位于同一台机器上，我们对缓冲区进行调整以使握手顺序进行。

图 6.8 描述了一次普通 DSS/DHE 握手的时间划分。这些消息与临时 RSA 的一样，但是处理时间大为不同。这是由两种因素造成的：首先，DSS 与 DHE 的性能特性与 RSA 相比有很大不同。其次，双方在连接期间均产生新的临时 DH 密钥。

Client	Server
0.07 Write client_hello	0.18 Read client_hello
	0.11 Write server_hello
	0.26 Write certificate
	113.90 Write server_key_exchange
	96.67 generate_keys
	17.15 sign
	0.00 Write server_hello_done
0.06 Read server_hello	
0.48 Read certificate	
21.24 Read server_key_exchange	
21.19 verify	
0.00 Read server_hello_done	
196.95 Write client_key_exchange	
96.82 generate_keys	
99.86 key_agree	
0.18 Write finished	
	100.86 Read client_key_exchange
	100.66 key_agree
	0.00 Read finished
	0.15 Write finished
0.01 Read finished	

Client: 219.02 Server: 215.51

图 6.8 DSS/DHE 模式的时间划分

我们立刻可以发现两点：第一，DSS/DHE 握手要比 RSA 握手慢得多，我们大约花费了 8 倍的 CPU 时间。第二，附加握手开销的大部分都由客户端承担。尽管如此，图 6.8 表示在使用 DSS/DHE 的时候，服务器耗费的 CPU 时间要比使用 RSA 时慢得多（大约 5 至 7 倍）。

ServerKeyExchange (服务器)

大家还记得在临时 RSA ServerKeyExchange 中，几乎所有时间都花在使用服务器的永久密钥给消息签名上。在这里的握手中，我们增加了一些新的内容：服务器直接产生 DH 密钥，每个连接一个。产生 ServerKeyExchange 消息的主要开销就是产生这个密钥。实际上，1024 位的 DSA 签名大约比同等的 RSA 签名快两倍。

有必要更详细地对 DH 密钥的产生过程进行研究。大家还记得第 1 章中 DH（与 RSA 不同）依赖于一个众所周知的组 (p 和 g)。私用密钥就是一个随机数 X 。公用密钥 (Y) 为 $g^X \bmod p$ 。我们这里所谈论的密钥产生就是产生 (X, Y) 对而不是那个组。

组的产生极为缓慢，因为它需要制作一个 1024 位的素数 (p)。而且由于经常建议那个素数具有某些特殊的属性，从而使得花费的时间会更长一些。在测试机上，OpenSSL 产生一个 1024 位的 DH 组要花费 10 分钟。因此一般是在启动服务器或（更好的方式）是在刚开始安装服务器的时候产生组。

Certificate (客户端)

DSA 的客户端证书处理与 RSA 的相同，惟一的区别就是算法。正如我们前面所讨论的，DSA 用于验证是比 RSA 慢很多（当使用 1024 位的密钥长度时慢大约 10 倍）。所以在使用 DSA 的时候，客户端对 Certificate 消息的处理要慢得多。

ServerKeyExchange (客户端)

与 Certificate 消息一样，对于 DSS/DHE 来说，客户端处理 ServerKeyExchange 消息与 RSA 的不同仅仅是将 RSA 替换成 DSA。因此，处理这条消息之所以慢纯粹是由于 DSA 验证慢得多。

ClientKeyExchange (客户端)

在这里的跟踪信息中，对 ClientKeyExchange 消息的处理实际上由两种操作组成，每一种大约耗费一半的 CPU 时间：产生客户端的临时 DH 密钥以及计算 DH 共享密钥 (ZZ)。产生客户端的临时 DH 密钥与产生服务器的临时 DH 密钥完全相同，而且我们可以看出在客户端分配给这项操作的时间与在服务器端的大致相同。

我们以前见过 ZZ 计算。大家还记得我们是这么计算的：

$$ZZ = Y_s^X \bmod p = (g^{X_s})^{X_c} \bmod p = g^{X_s \cdot X_c} \bmod p$$

正如你从跟踪信息中所看到的，这项操作花费与产生客户端的 DH 密钥大致相同的时间。

注意，我们没有利用可使双方并行推进的机会。产生 ClientKeyExchange 并不依赖于知道 ZZ 。 ZZ 从不在线路上进行传输，使用它纯粹是为了计算对称消息密钥。在发送 ClientKeyExchange 之前计算 ZZ 只会推迟消息传输，增加延迟。

一种更好的方案就是先发送 ClientKeyExchange，然后再计算 ZZ 。那样一来客户端就能够在传输 ClientKeyExchange 或服务器自己计算 ZZ 的过程中计算 ZZ 。注意，这种改动主要

影响延迟。除非服务器负载很轻，否则对通量就没有太大的影响。

ClientKeyExchange（服务器）

服务器端对 ClientKeyExchange 的处理就是客户端处理的两步。我们使用服务器的私用密钥及客户端的公用密钥计算出 ZZ。这些计算大致也花费相同的时间，这并不奇怪。

6.15 具有客户端认证的 DSS/DHE

我们所将讲到的最后一一种握手就是具有客户端认证的 DSS/DHE。图 6.9 显示，与 RSA 客户端认证一样，与非客户端认证模式相比惟一显著的差异就是增加了几条新消息：客户端 Certificate 和 CertificateVerify。

Client	Server
0.07 Write client_hello	
	0.17 Read client_hello
	0.12 Write server_hello
	0.26 Write certificate
	113.96 Write server_key_exchange
	96.63 generate_keys
	17.26 sign
	0.02 Write certificate_request
	0.00 Write server_hello_done
0.06 Read server_hello	
0.46 Read certificate	
21.36 Read server_key_exchange	
21.30 verify	
0.01 Read certificate_request	
0.00 Read server_hello_done	
0.23 Write certificate	
193.21 Write client_key_exchange	
96.87 generate_keys	
96.07 key_agree	
17.06 Write certificate_verify	
0.18 Write finished	
	0.42 Read certificate
	95.98 Read client_key_exchange
	95.79 key_agree
	21.00 Read certificate_verify
	0.00 Read finished
	0.15 Write finished
0.01 Read finished	
Client: 232.69	Server: 232.13

图 6.9 DSS/DHE 客户端认证模式的时间划分（毫秒）

客户端惟一有分量的操作就是 CertificateVerify 上的 DSA 签名。服务器对客户端 Certificate 消息的处理主要由证书验证主导。因为这是一份 DSS 证书，所以验证相当缓慢。

与之类似，服务器对 CertificateVerify 消息的处理由 DSA 验证主导，注意，对 DSA 来说（与 RSA 不同）客户端认证的主要开销由服务器承担，因为 DSA 验证速度很慢。

6.16 DH 性能的改进

正如我们以前所见到的，DHE 的性能比起 RSA 来要糟糕许多。大家会问为什么还要使用这种模式。直到最近，存在两种原因：首先它是免费的。RSA 在美国仍受专利保护，而 DSA 则不受。其次，它支持一种新功能，PFS（完美向前保密）。由于 RSA 密钥产生的昂贵开销，对 RSA 来说 PFS 是不现实的，但对 DHE 来说却是可行的（尽管开销也昂贵）。现在 RSA 专利失效了，惟一的理由就是具有 PFS。PFS 有时是很有价值的，因此有必要考虑一下我们究竟能不能改善 DHE 的性能。答案是肯定的。

● 使用较小的私用密钥

大家或许还记得，所有的 DH 操作都需要将某个数字还原为 $X \bmod p$ 。 X 的值越大，这项操作的速度就越慢。一种常用的方法是简单地使用与 p 大小接近的 X 。更具体地讲就是在区间 $(2, p)$ 中选择 X 。 X 有可能比 p 小得多，但由于 X 是在这个区间上随机选择的，它大致是 p 的长度。然而，如果特意选择了一个较小的 X ，那么操作的速度就会快得多。

只有在特定的情况下才允许优化。不然就有可能极大地削弱密钥交换的安全性。回想一下第 1 章，为了使用短 X 、 g 与 p 必须具有特殊属性。

然而，客户端无法简单地判定服务器是否恰当地产生了 g 与 p ，因为服务器只在 ServerKeyExchange 中将 g 与 p 发送给它。因而，客户端就不能安全地使用短 X ，也就不能获得任何性能提升。在大多数情况下，只有服务器的性能才是要紧的，但如果客户端是一种像掌上电脑或蜂窝电话这样一种计算能力非常有限的设备，则 DH 操作给客户端施加的负担就可能是巨大的。

为了演示这种优化，我们修改了 OpenSSL，让它在 $(2, p)$ 上选择 X ，在仍然使用 1024 位的 p 的情况下，选择一个 256 位的 X 。这种简单地改动使服务器端 DH 操作的速度大约提升了 4 倍，请见图 6.10 所示。

大家可能还记得我们在第 1 章说过 X 的长度应当选取为所需密钥资料长度的两倍，然而在实际应用中 DH 密钥的强度还受 p 的尺寸的限制。DH 密钥的强度由 p 的强度与 X 的强度中最小的一个来决定。一个 160 位的 X 等价于 1024 位的 p 。对 1024 位的素数来说，256 位的 X 完全够长了（参见[Menezes1996]）。

```
Client           Server
0.07 Write client_hello
                  0.17 Read client_hello
                  0.12 Write server_hello
                  0.26 Write certificate
43.14 Write server_key_exchange
25.93   generate_keys
17.14   sign
                  0.00 Write server_hello_done
0.06 Read server_hello
```

```
0.47 Read certificate
21.26 Read server_key_exchange
21.21 verify
0.00 Read server_hello_done
192.66 Write client_key_exchange
96.60 generate_keys
95.79 key_agree
0.18 Write finished
25.57 Read client_key_exchange
25.38 key_agree
0.01 Read finished
0.15 Write finished
0.01 Read finished
```

Client: 214.75 Server: 69.46

图 6.10 短 DH 密钥的性能影响

重用临时密钥

通过使用更短一些的私用密钥(X)，在没有对安全造成任何显著影响的同时改善了性能。然而通过放弃使用 PFS 还有可能进一步改善性能。回想一下，服务器计算负载中有大约三分之一都是由于产生临时 DH 密钥所消耗的。重用那个密钥就会相应地减少工作负载。然而不幸的是，服务器仍然得完成 ServerKeyExchange 上的签名工作，因为签名也是根据握手中的部分来计算的。

重用临时密钥只会减少服务器而不是客户端的负载。因为组是由服务器决定的，在大多数情况下，每个服务器的组是不同的，因此客户端仍要为每个服务器产生一个新的密钥。尽管如此，这种优化还是同时改善了通量和延迟，因为它使服务器运行的更快了。

有些人或许会争论：为了获得这些性能上的改进，我们不得不放弃一定程度的安全。然而，要知道 RSA 模式根本就不提供 PFS，所以尽管这种类型的重用比为每个连接使用一个新的密钥弱一些，但是其强度是与 RSA 相当的。作为一种变通措施，可以周期性地重新产生服务器的 DH 密钥。每 20 或 30 个连接产生一次将会提供部分的 PFS，而对性能的影响则可忽略。注意，你如果使用短 X 并重用于多个连接的话，就必须留心阻止我们在第 5 章所讲的子组攻击 (subgroup attacks)。

为 DH 密钥的产生执行预算算

即便我们不打算重用临时密钥，也有可能极大地改善密钥的生成。为了做到这一点，需要预先计算出一张包含产生器 g 的幂的表。因为每次公用密钥计算都需要还原 g 的对数，这种优化就可以减少从 X 计算得出 Y 所花费的时间。Wei Dai [Dai2000] 报告称其与原来的 DH 密钥产生相比，预算算的 DH 密钥产生速度大约提高 3 倍。作为一种特殊情况，如果底是 2，则通过简单地使用左移就能极快地计算出指数 (exponential)。

静态 DH

最后一种选择就是使用静态 DH 密钥。大家可能还记得第 4 章，可以将服务器的 DH 密钥放在证书里，就像通常对待 RSA 密钥那样。这就完全与 ServerKeyExchange 方法结合在一

起了，只在服务器上留有计算 ZZ 的工作。连同短 X 一起，这种优化将服务器端的负载减轻至大致与 RSA 相当的水平。客户端的负载仍然要高得多，因为 DSA 验证太慢了。注意，如果使用静态 DH 的话，必须要非常仔细的确保无法实施小-子组攻击。

静态 DH 主要的部署问题就是它需要静态 DH 证书，该证书甚至还没有 DSA 证书的应用普遍。

6.17 记录处理

算法的选择

图 6.11 描述了 OpenSSL 在一台 Pentium 300 机器上组合使用各种加密和 MAC 算法的性能。这些数字是由计算 SSL 记录但并不将其发送到网络上来产生的。这样，它们度量的只是记录产生的开销，而不包括网络开销。我们通过它们来表示你所期望的各种行为。显然，每种系统的性能和实现都是不一样的。

简而言之，使用 MD5 的 RC4 提供最佳的性能，而且通常要比使用 SHA 的 3DES 大约快上 8 倍。本文的意图就是，你如果需要不错的保密性与高性能，那么就应当使用 RC4。用 MD5 来代替 SHA-1 可以有 40% 的改善。

加密/散列算法	KB/秒
RC4-MD5	15034
RC4-SHA	10831
DES-CBC-SHA	4758
DEC-CBC3-SHA	2068

图 6.11 SSL 记录处理速度 (Pentium300)

注意，最慢的加密算法（3DES/SHA）的处理速度也超过 2MB/s。这种速度快得足以充盈 10MB 的以太网。在一台不错的机器上，RC4-MD5 的速度快得足以充盈 100MB 的以太网。在我们的测试机上，即便是使用 SHA-1 的 RC4 也产生了超过每秒 80MB 的流量。在实际应用中，这种速度足以充盈 100MB 的网络。即便对最快的网络来说，如果使用 RC4 的话，一台速度快的机器很容易就能完成 SSL 记录的处理，即便是 SHA-1 也是如此。

需要指出的重要一点是使用每种加密算法的出口变种不会有任何性能上的优势。SSL 总是使用同一版本的加密算法并将缩短的密钥扩展为全长密钥。结果，出口版本具有与高强度版本完全一样的数据传输速率。对公用密钥加密算法来说情况就有所不同，出口版本的速度明显更快。

最佳记录尺寸

SSLv3 与 TLS 都使用 HMAC（SSLv3 使用的是它的一个变种，但是性能特性几乎是一样的）正如我们在第 1 章所讨论的，HMAC 由一对嵌套的摘要组成。第一个摘要涵盖密钥与数据。第二个摘要含盖密钥以及第一个摘要的输出。密钥总是作为分离的摘要分组来处理的。

因此，内层摘要依赖于消息的长度，但外层的摘要则不是这样。一次给摘要传递 64 字节的数据。对于多达 64 字节的摘要输入，需要处理 4 个摘要分组。对于随后的每 64 个字节，

就会再需要一个摘要分组。关键就在于 MAC，存在大量固定的开销。

在加密记录时，需要同时对 MAC 和数据进行加密。如果使用的是 SHA，就意味着需要加密至少 20 个字节，同样也存在不少固定的消息开销。因此，为了减少开销 (overhead) 的影响，尽可能加密大尺寸的负载 (payload) 是值得的。图 6.12 非常清晰地描述了这一点。

图 6.12 的数据是在使用 3DES-CBC 和 SHA 情况下采集的，但是使用任何加密套件均能绘制出类似的图例。为了清楚地显示小尺寸记录时的效果，我们在 X 轴上使用了测量标尺 (log scale)。

正如图 6.12 所示，使用小记录尺寸时的性能非常糟糕，并且渐进变得越来越好，直到大约 600 字节那一点时增长逐渐减缓。这一点的确切位置当然依赖于实现、操作系统和编译器等，但是基本意思是明显的。使用小于 32 字节的记录尺寸的性能极差。要想获得最佳性能，就要尽可能使用 1024 字节或更大的记录尺寸。

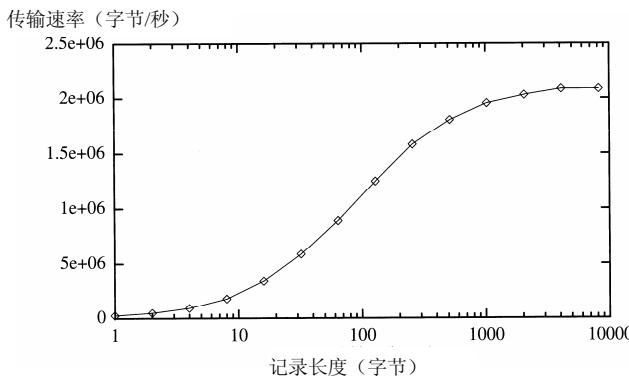


图 6.12 各种消息尺寸的性能 (3DES/SHA-1)

6.18 Java

Java 的性能状况比起 C 来要糟糕得多。Java 代码总体上是缓慢的，而加密代码尤其耗费 CPU 时间，因此也就特别慢。在我们的测试平台上，OpenSSL 执行 SHA-1 的速度大约是 22MB/s，而我所知道的最快的 Java 代码在 Pentium II/450 上执行的速度也就是大约 2.3MB/s。

操作系统

不幸的是，很难就此状况有一个清晰的认识，因为执行 Java 加密代码的虚拟机 (VM)，是独立于操作系统的。DES 或许是最常见的算法实现，其速度从运行在 FreeBSD 上的 Netscape Navigator 的 37KB/s 到运行在 Windows NT 上的 Internet Explorer 的 2.3MB/s 不等。图 6.13 描述了 DES 在各种平台上的性能。

注意，Windows 实现的性能比 FreeBSD 实现好非常多。Windows 是 Sun 在其上发布官方 JDK 平台，而 FreeBSD 的移植版本不是官方的。显然对 Windows 实现的调校花费了多得多的精力。然而，即便是在 Windows 上，IE 与 Navigator 之间的差别也是非常明显的。

处理器	操作系统	虚拟机	速度 (KB/秒)
Pentium II (400)	FreeBSD 3.4	Netscape 4.7	36
Pentium II (400)	FreeBSD 3.4	JDK 1.1.8	105
Pentium II (450)	Windows NT	JDK 1.2.2	2368
Pentium II (450)	Windows NT	Netscape 4.61	1856
Pentium II (450)	Windows NT	Internet Explorer 5	2334

图 6.13 各种平台上的 Java DES 性能

本机代码加速

由于这一过程中的瓶颈是 Java 加密原语，一种高效而直接的，可以极大提高性能的方法就是将 Java 原语替换为快速的本机代码。JNI (Java 本机接口) 提供了一种让 Java 与本机例程（通常是用 C 编写的）接口的可移植方式。

GoNative Provider 公司（请见 <http://www.rtfm.com/puretls/gonative.html>）实现了一种完成此项工作的方案：JDK (Java 开发箱) 提供了一种称做 JCA (Java 加密体系) 的通用加密供应接口。使用 JCA 的程序调用 JCA 来获得实现指定算法的特定实例（类对象）。供应商通知 JCA 它所实现的算法，而 JCA 会自动安排使用恰当的供应商。

结合使用 JCA 与 JNI 使得这种形式的加速非常容易。GoNative Provider 就是一个 JCA 供应商，它作为连接 OpenSSL 的桥梁（使用 JNI）。可以将其作为供应商安装在给定的机器上，而应用无须任何特殊的努力就能让所有的应用透明地加以存取。这种方式实现的加密算法要比标准的 Java 算法快得多，而且只比用 C 写成的同等代码慢 10% 左右。注意，算法越快，JNI 所带来的开销越显著。图 6.14 描述了某些具有代表性的算法的性能属性（平台：Pentium300，FreeBSD）。

算法	OpenSSL	Java	GoNative Provider
DES(KB/S)	6792	31	5165
3DES(KB/S)	2392	10	2142
RC4(KB/S)	32363	132	13872
SHA-1(KB/S)	22838	93	10014
DSA(sign/S)	60	7	48
DSA(verify/S)	49	4	40

图 6.14 Java 和 C 的相对性能

从这张表可以观察出调用 JNI 本身存在显著的开销，这也是 GoNative Provider 与 C 之间的 3DES 性能相差无几的原因。作为此项观察的一项推论，如果你想通过 JNI 加速获得最好的结果的话，使用大记录尺寸尤为重要。

在传递给 C 代码之前，缓冲尽可能多的数据也很重要。例如，设想你给 DES 提供了一大块数据。重要的是要作为一个分组传递给 DES 代码，而不是对每个 8 字节分组调用一次 DES。图 6.15 描述了就 GoNative Provider 而言，输入分组的大小对 DES 加密的影响。注意，图 6.15 只描述了 DES 的性能，而没有描述包含 MAC 计算的 SSL 记录加密的性能。然而，对于消息摘要与 MAC 而言，也存在类似的分组尺寸影响。

不幸的是，所有这些调校的最终结果仍没有纯 C 代码快，但是差的没那么多了。根据

所选算法的不同，使用 GoNative Provider 模块的 PureTLS 的速度大约是 OpenSSL 的一半。

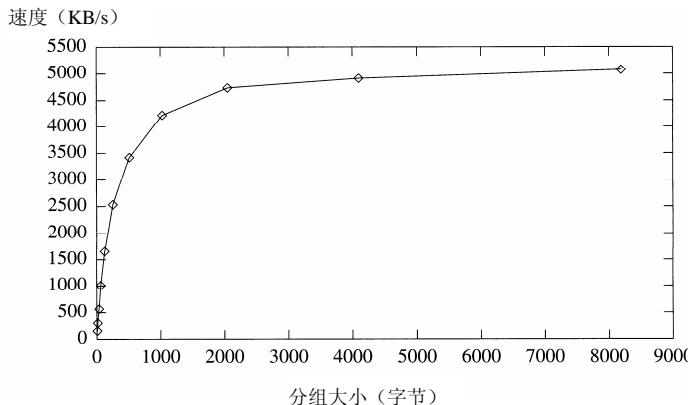


图 6.15 JNI 加密分组尺寸的影响

6.19 重负下的 SSL 服务器

在重负下 SSL 所需的加密计算成为服务器性能的瓶颈并不奇怪。因此，能够处理给定负载的服务器在通过 SSL 来传输那些交易时，很快就会超载。为了仿真出大量客户端的影响，我们使用了负载生成器，让一台客户端机器（或多台机器）连接到服务器并发送请求。每台机器在给定时间能够产生大量的请求，这样就仿真出大量用户连接到指定站点的情形，这里的客户端是一台运行 FreeBSD4.3 的 Pentium II/400。

可以使用任何 SSL 协议来完成这项试验，但是为了方便，我们使用基于 SSL 的 HTTP。我们所选用的通信搭配类型仿真了当许多人几乎同时决定请求同一个页面时的 Web 站点行为。Abbott 和 Keung [Abbott1988] 在他们的论文中指出这是其顾客最为关心的情形。

为了产生这种通信混杂，我们安排每个客户端以不同的到达时间连接到服务器。使用平均 3 秒的泊松过程来产生这些到达时间。这样在仿真开始之后，所有的客户端大约都在 0 到 10 秒之间的某个时间进行连接。

图 6.16 描述了在一台普通的 HTTP 服务器上，这样一种仿真的结果。该服务器是带有 mod_ssl 2.6.5 的 Apache1.3.12，在一台装有 Linux2.2.12-5.0（红帽子）的 Pentium II/650 上运行。标为“arrivals”的条形图代表在每秒内仿真客户端连接的次数。因此，在仿真的头一秒钟内初始有 23 条连接。标为“queue”的线形图代表在这一时间窗口内处于活动状态的客户端的数目。因此在 1.5 秒进行连接并在 2.5 秒断开的客户端对 1-2 和 2-3 时间窗口间的队列进行增减。

当我们研究图 6.16 的时候，可以看出队列尺寸与客户端的到达紧密一致。发生这种情况的原因就是服务器多少是在请求到来时进行处理的。这是服务器成功处理负载的良好表示。

将这种结果与图 6.17 进行比较，图 6.17 描述了具有 550 个仿真 HTTP/SSL 客户端时的服务器行为。加密套件是 TLS_RSA_WITH_3DES_EDE_CBC_SHA，客户端被设置为从

不恢复连接，因此每次握手都需要一次 RSA 操作。刚开始，队列尺寸与客户端的到达时间紧密一致，但是随着仿真期间客户端数量的增多，队列尺寸开始陡增。注意，尽管我们的客户端比服务器慢好多，但仍然能够完全使服务器满负荷工作。这是对异步 SSL 的良好演示。

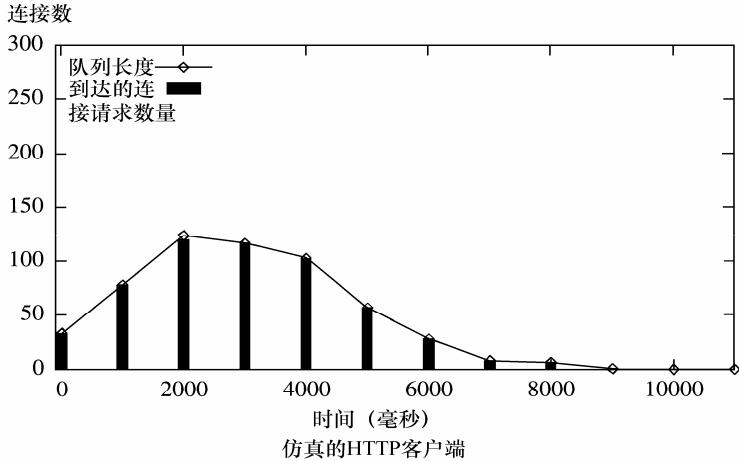


图 6.16 550 个仿真的 HTTP 客户端

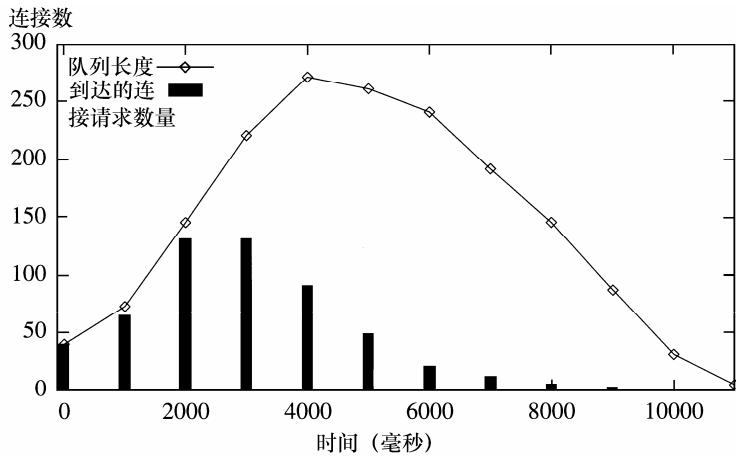


图 6.17 550 个仿真的 HTTP/SSL 客户端

这里所发生的情况就是服务器 CPU 被已经在执行的 SSL 握手所充盈。这样，更多的客户端到来却没有被成功处理，从而产生了未处理客户端的积压。随着新到连接的减少，服务器能检查排队的未处理连接，于是队列尺寸会缓慢减小。然而，服务器只有在到达数目适当时才能清空队列。

服务器无法持续处理图 6.17 中所描述的突发性负载。图 6.18 就描述了这种情况，其中服务器处于大约每秒 75 个连接的持续负载下。注意，队列尺寸随时间呈线性增长，表明服务器无法跟上当前的负载。

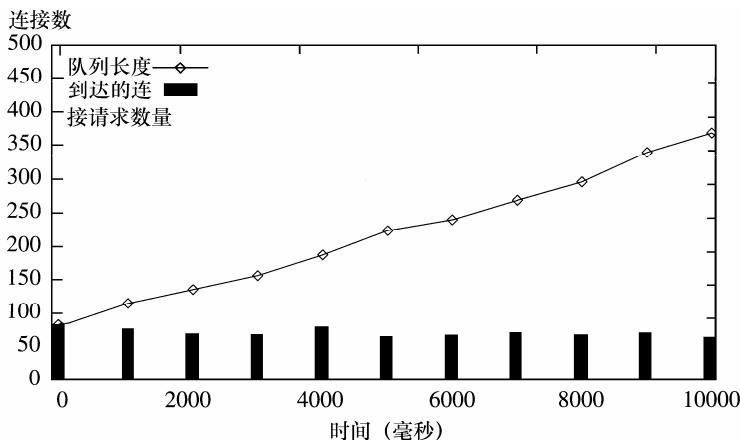


图 6.18 处于持续高负载下的服务器

6.20 硬件加速

由于 SSL 性能中的瓶颈为服务器 CPU，因此自然就会考虑通过将开销昂贵的加密操作转移到分离的处理器上来加速 SSL 服务器。硬件加速不但可以增加服务器所能处理的连接数目，还能留出 CPU 时间执行其他任务，如解释用来运行许多 Web 站点的 CGI 脚本。

硬件加速器通常包装成可以插入服务器机器中的板卡，在 Intel 体系的机器上，常常是通过 PCI 总线，还可以通过以太网连接或 SCSI 总线来连接加速器。有不少提供加速器的厂商，但是最著名的是 Rainbow(<http://www.rainbow.com/>)和 nCipher(<http://www.ncipher.com/>)。

图 6.17 描述了同一台用 Rainbow SSL 加速器加速的服务器在处于先前的连接情况下的状况，该加速器每秒能够执行 200 次 RSA 操作。我们同样使用 550 仿真客户端。然而，使用加速器后，队列尺寸又一次紧密地与到达时间一致。服务器能够处理这些负载，因此也就没有形成积压。

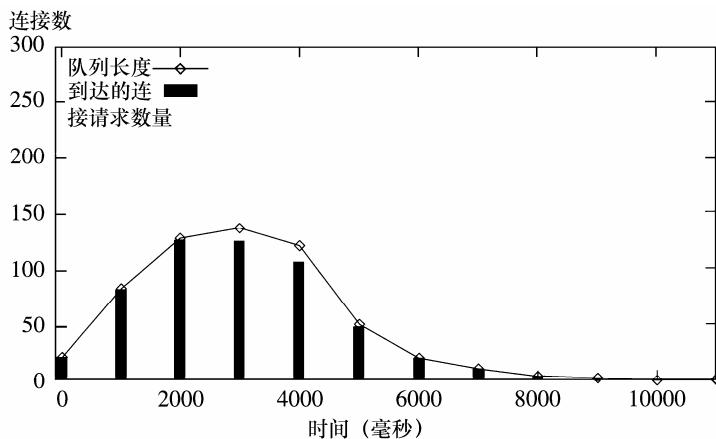


图 6.19 550 HTTP/SSL 客户端（硬件加速）

由于 SSL 握手中开销最昂贵的计算就是 RSA 计算，对 RSA 加速最有意义。Rainbow 和 nCipher 宣称他们的高端产品通过加速 RSA 计算，每秒可以处理多达 1000 次 SSL 握手。只要服务器还算可以，这种加速在很大程度上是独立于服务器 CPU 的。对比一下我们的 Pentium III/650 系统，它能够每秒完成 95 次 RSA 操作。但在实际应用中，它所能处理的每秒连接次数非常低，因为现实情况下 CPU 除了完成加密之外还得运行服务器。因此，75 仿真的客户端就足以使服务器超载。

加速器还能够用来去掉对称操作。在大多数情况下，只去掉异步操作就足够了，但是如果网络连接非常快，则对加密以及消息摘要进行加速也是值得的。Keung [Keung] 报告称，在给一台处于 10MB 以太网上的 200MHz Pentium 加装加速器之后，吞吐量几乎增加一倍。注意，对于现在更快的处理器来说，效果不会有那么显著。尽管如此，如果服务器处理大量保密通信的话，考虑用加速器来增加通量是值得的。如果你已经对异步加密进行了加速，对对称加密进行加速肯定是值得的。RC4 已经足够快了，因此或许根本就不值得对它进行加速。

通常来说，加速器生产商还提供让流行的服务器使用其加速器的驱动程序。一种流行的做法是发布补丁程序或经过修改地使用硬件加速器的 OpenSSL 版本。这样就可以对任何使用基于 OpenSSL 的应用进行加速。

6.21 串联硬件加速器

另一种硬件加速 SSL 的方法就是使用分离的串联加速器。该加速器是一种位于客户端与服务器之间的硬件设备。它监听来自于客户端的 SSL 连接并接受握手，然后初始化一条到服务器的普通 TCP 连接。这样就缓解了服务器执行 SSL 的必要。为了使这种方案能够工作，加速器需要拥有服务器的密钥资料。

串联加速器的主要好处就是它不需要对服务器进行任何改动。而硬件加速卡必须插在服务器中，所以带来了不稳定的可能。串联加速的主要不足就是客户端与服务器之间的联系是断开的。这使得服务器很难基于 SSL 连接的安全属性做出存取控制决定，因为没有办法将这些信息从加速器传递给服务器。

有多个厂商生产串联加速器，其中包括 iPivot/Intel(<http://www.ipivot.com/>)、F5(<http://www.f5.com/>)和 Network Alchemy/Nokia(<http://www.network-alchemy.com/>)。

配置

考虑将串联加速器插入到一个具有工作服务器的环境中。我们现在必须安排客户端连接要以加速器而不是服务器为终点。一种方案就是给加速器分配自己的 IP 地址，然后再调整 DNS 记录让其指向加速器而不是服务器。另外一种方案就是给服务器分配一个新的 IP 地址而不改动 DNS。注意，尽管服务器与加速器可能位于同一个网段上，但由于其间的通信是未加密的，所以那个网段必须是安全的。

一种更为流行的方案就是使用透明配置。在这样一种配置中，加速器必须位于客户端与服务器之间在物理上分开的网络中。加速器被配置为像服务器一样截获一类特定的连接。然而，它允许所有其他连接简单地通过，就像个网桥一样。透明方案具有显著的优势，它不要求对地址分配或 DNS 做任何改动。你只要将加速器插入到网络中并将服务器插入到加速器

中就可以了，如图 6.20 所示。

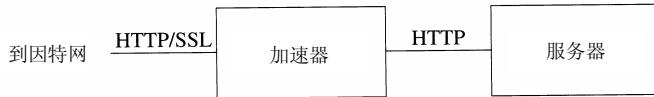


图 6.20 串联 SSL 加速器

多重加速器

一个非常繁忙的服务器的流量是一台加速器无法处理的，即便是硬件加密也不行。为了处理高负载，管理员或许希望使用多个加速器。这种方案的附加好处就是，如果一个加速器坏掉的话，服务器仍然能够处理请求，尽管它所能服务的请求总数会少一些。存在两种使用多重加速器的方式：“链式”和“集群”。

链式加速器

首个提供串联加速器的厂商是 iPivot，后来该公司连同他们的 Commerce Accelerator 一起由 Intel 收购。iPivot 的机器只是一种透明代理。为了使用多台机器，只需将它们顺序串联起来即可，如图 6.21 所示。

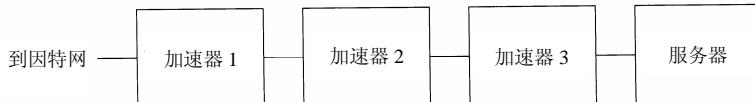


图 6.21 链式 SSL 加速器

如果机器上的负载太高，则这台机器就会简单地将通信原封不动传送给链中的下一台机器，该过程持续到所有客户端都得到处理为止。可以将链中的最后一台机器配置为在负载过高时拒绝连接或将其传送给服务器。服务器应当被配置为支持 SSL 以便处理这样的请求。

iPivot 的硬件装备有一种“bypass”功能。当机器失效时，bypass 功能就会起作用，而所有的通信就会直接穿过那台机器，就像它不存在一样。这样，即便机器发生了相当严重的故障，服务器也能够连接到网络上为客户端提供服务。然而，由于必须将此台机器配置成原始的 HTTP 连接，使用 bypass 功能就会导致服务器接受 HTTP 连接，而服务器却以为这是经过解密后的 HTTPS 连接。这显然成为一种安全问题。

集群加速器

考虑有多台链式加速器中的一台失效时的情况。尽管余下的加速器仍能处理新的连接，但所有在那台机器失效时以它为终点的 SSL 连接均会丢失。当然，客户端与服务器都不会知道发生了这样的情况，因此当它们试图发送信息时，就会得到 TCP RST。集群加速器的思想就是允许在失败的情况下单个连接可以在多台机器间进行切换，结果就不会丢失通信信息。

揭秘：作者以签约形式在 Network Alchemy 的集群 SSL 加速器上完成了大量的开发工作。说这些的好处就是我熟悉这种设备的工作原理，而且能在它发布之前讲解有关的内容。不利之处就是存在着公私利益冲突。我曾试图客观地描述这种方案的利与弊，但是了解了这些实际情况之后，读者就能对我的讲解有更客观的判断。

预计 2001 年一季度出品的 Network Alchemy/Nokia 集群 SSL 加速器企图解决活动连接的容错问题。如图 6.22 所示，我们不是将多台机器连成链而是平行地连接在一起。这种配置就称做集群。

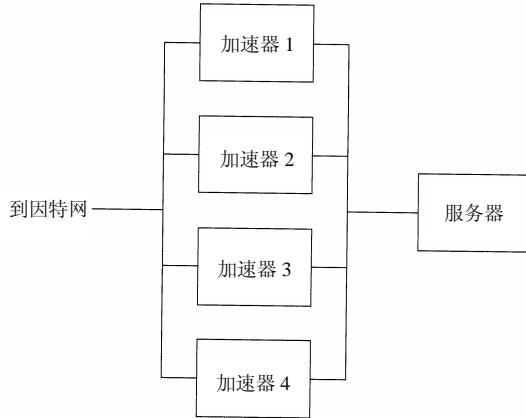


图 6.22 集群 SSL 加速器

整个集群具有一个虚拟的以太网 MAC 地址，这样其行为就像是一个单一的硬件。集群中的每台机器都监听发往这个地址的所有通信数据。集群自动在机器之间分派处理每部分通信的责任。如果一台机器坏掉的话，其通信会自动由其他机器拾起来。

为了具有容错功能，集群机器自动将处理连接机器上的连接状态传播到集群中其余机器上。这样，如果其中的一台机器失效，则指定处理这个连接的新机器就已经掌握了所有的连接状态，于是就能够从别的机器丢掉的地方重新捡起来。注意，如果机器在失效时正在处理给定连接的通信，那个 IP 包将会被丢弃。然而，当重发这个包的时候，目前正处理这个连接的机器就会自动进行处理。而所有的客户端和服务器看到的只是等待重传时的一些延迟。

注意，尽管集群方案能够完善地处理单台机器失效的情况，但却不能很好处理灾难性的事故。正如我们在前一节所看到的，如果所有的 iPivot 机器都失效的话，网络所表现出来的行为就好像它们都不存在一样。由于我们想要让另一台机器在一台机器失效时重新捡起这个负载，所以当单个集群机器失效时，它们就会处于一种断开的状态。这样如果集群中的所有机器都失效时，服务器将会拒绝服务。

然而，集群中所有机器都失效的几率一般来说相当低。最可能的情况就是某种电力问题将所有的机器都给当掉了。为了避免这种情况的发生，各个集群成员必须位于分离的回路上，以避免单一失效点。与之对应，在 iPivot 的解决方案中，每台机器都是单一失效点。尽管 iPivot 机器被设计成是失效安全的，但是某些可以预见的问题（如内部线路断裂）仍然会让整个系统停掉，这种情况下，旁路功能 bypass 也不行。

集群方案还有其他几个优点。由于集群自动分享工作负载，因此负载可以更容易而且更平均地在集群成员之间分配，要想使用链式方案实现这一点则更为困难。集群还可以被配置成一个单一的单元，不必将每台机器都配置成拥有所有的 SSL 信息。对集群进行部分配置，而 SSL 配置就会自动进行传播（propagate）。

6.22 网络延迟

到目前为止，我们的注意力仅集中在 SSL 如何使用处理器上，而忽略了其与网络的交互。然而，SSL 连接所运行的网络特性对其性能特性有着重要的影响。本章的剩余部分将集中讨论这种交互。

尽管从理论上讲，SSL 可以在任何面向连接的协议上运行，但在实际应用中，几乎总是在 TCP/IP 上运行的。因此，我们将把注意力几乎完全放在 TCP 网络上运行 SSL 的性能特性上，尽管其中一些内容也适用于其他类型的网络。

我们到目前为止所收集的数据源都是运行在单台机器上并直接通过内核进行通信的程序，而未经过任何通信网络。与真正的网络不同的是，其速度要快得多。通量非常高，且延迟非常低。然而在真实的网络中，通量会低一些而延迟则会高一些。

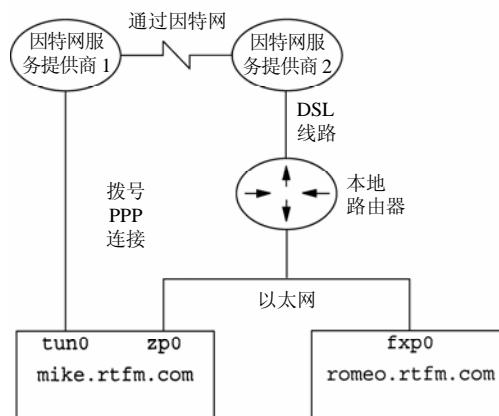


图 6.23 慢速的测试网络

为了演示较慢网络中的效果，我们从图 6.23 所示的网络配置中采集了许多跟踪信息。该网络有两台测试机器，romeo.rtfm.com 和 mike.rtfm.com。romeo 与网络正常连接，但是 mike 被特别配置成双宿主（dual homed）的机器。它有两个 IP 接口，主要的网络连接是接口 tun0 上的 PPP 拨号连接。mike 与本地网络之间的通信必须要经过 PPP 接口。

然而，mike 也直接通过以太网接口 zp0 与局域网相连。在正常情况下，mike 与 romeo 之间的通信将会通过这个接口。然而，我们故意将 zp0 配置成私有网络以强迫通信经过 PPP。

这种配置的意义就在于 mike 可以被配置成同时捕获 tun0 和 zp0 上的数据包。这样，它就能两次看到每个数据包——即在 PPP 接口上以及在以太网上出现时。这就可以让我们直接观察给定数据包在 mike 和 romeo 之间通过网络传输时花费的时间。例如，当从 romeo 发送一个数据包给 mike 时，该数据包首先从 romeo 传递给以太网上的本地路由器。与 mike 的 zp0 接口连接的包嗅探器就会捕获这个数据包。该数据包然后又通过 DSL 线路传递给一个 ISP，通过因特网传递给另一个 ISP，最后通过 PPP 连接传递给 mike。这时，与 tun0 连接的包嗅探器就会捕获这个数据包。

图 6.24 描述了 mike（客户端）与 romeo（服务器）之间完成的一次简单 SSL 握手，加

密套件还是使用 RSA 完成密钥交换。与前面的时间表不同，这里绘制的是全尺寸的。每个箭头指示的垂直位移表示网络传输的时间。每个箭头左边和右边的线描述从发送到收到的实际时间。因此，ClientHello 从 mike 到 romeo 花费了 112 毫秒。

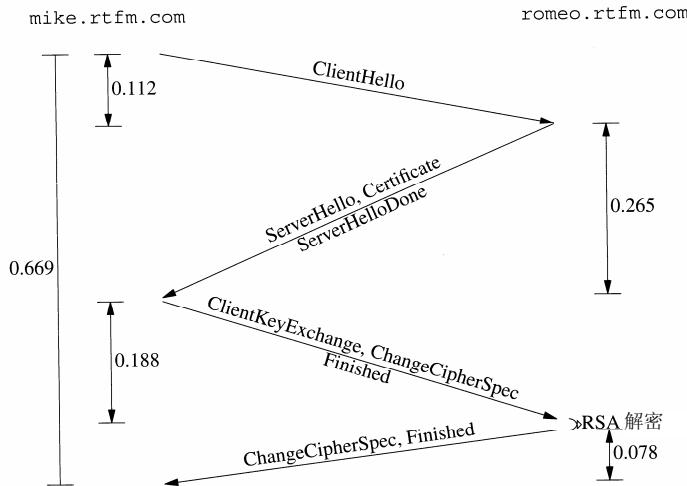


图 6.24 高延迟连接上的 SSL 握手

注意，握手的总时间增加了很多。这里，从开始到完成的时间是 669 毫秒。而在快速网络上类似的握手可能需要 100 毫秒。然而，绝大部分的时间都只是网络延迟。数据包一经接收就迅速得到处理。惟一一个具有可以察觉到的影响的操作就是 RSA 解密，它耗费大约 20 毫秒——这可是握手总时间中不起眼的一部分。

有趣的一点是，各种握手消息的传递时间大为不同。合在一起的 ServerHello、Certificate 和 ServerHelloDone 花费了 265 毫秒，而服务器的 ChangeCipherSpec 和 Finished 只花费了 78 毫秒。乍一看这种结果令人吃惊，但是非常容易解释：系统中的大多数延迟不是由 ISP 之间的因特网引入的，而是由 mike 与第一个 ISP 之间的 PPP 连接所导致的。在慢速的调制解调器链路上，延迟由传输时间控制。数据包越大，传送花费的时间就越长，延迟也就越高。Certificate 消息很大，因此引入的延迟最高。

注意，尽管增加的网络延迟使单个握手慢了下来，但总体上并不降低服务器的通量。只要服务器的网络连接还没有饱和，在服务器等待一个客户端的响应时，它还能够处理其他的客户端。然而这并不是说高握手延迟不重要，因为在用户看来就是性能差，所以如果减少延迟的话，就能够改善用户的感觉。

6.23 Nagle 算法

在一定情况下，SSL 与各种 TCP 拥塞控制方法之间的交互非常糟糕。这种交互可以导致 SSL 握手在客户端与服务器都空闲的情况下发生停滞。罪魁祸首就是在 [Nagle1984] 中所描述的 Nagle 算法。

Nagle 算法为的是减少 tinygram：即线路上的小数据包。问题就出在 TCP 包头上，TCP

段最小包头尺寸为 40 个字节。因此，在每段发送一个字符的时候，就必须在网络上传输 41 个字节，这样很快就会导致拥塞。在正常情况下，只要程序调用 `write()`（或等价的调用），TCP 栈就会简单地将数据立刻发送出去。然而，如果程序以小数据段（比如用户击键）调用 `write()`，而核心对每一次写操作都发送一个数据包的话，网络上就会塞满这样的数据包。

Nagle 算法使用了一种简单的方法来阻止这种情况的发生。当存在有已经发送但还未应答的未完成（outstanding）数据时，TCP 实现就不再发送小数据包。而是将数据加以缓存并在收到 ACK 时将整个缓冲区的内容发送出去。在正常情况下，这样工作的很好，但是对 SSL 来说就会出问题，我们马上就会看到这一点。

这种问题造成的第二个影响就是一种称做延迟 ACK（delayed ACK）的技巧。TCP 实现不是在一收到数据包就产生一个 ACK，而是将 ACK 附着在它已经发送出来的数据段上面（piggyback）。为了利用这一点，TCP 实现将会等待至多 200 毫秒后才将数据写出去。如果在滑动窗口内中没有数据，那么到时也会将 ACK 输出到网络上。

这种交互的一个很好的例子就是恢复 SSL 会话时发生的情况。图 6.25 描述了这样一种连接的网络跟踪。注意，尽管客户端与服务器处于同一台机器上，数据仍然要流经 TCP 栈，于是我们也能够观察到 Nagle 算法的行为。这里需要注意到的重要一点是，客户端发送 `Finished` 消息（第 6 行）与第一条数据记录（第 7 行）之间有 148 毫秒的延迟。这可要比准备这条记录所花的时间长得多。

```
New TCP connection: localhost(2830) <-> localhost(4433)
1 0.0003 (0.0003) C>S Handshake ClientHello
2 0.0028 (0.0024) S>C Handshake ServerHello
3 0.0028 (0.0000) S>C ChangeCipherSpec
4 0.0028 (0.0000) S>C Finished
5 0.0039 (0.0011) C>S ChangeCipherSpec
6 0.0039 (0.0000) C>S Finished
    TCP ACK arrives at 0.1516
7 0.1517 (0.14777) C>S application_data
8 0.1530 (0.0014) S>C application_data
```

图 6.25 The Nagle algorithm and session resumption

这里发生的情况是客户端 TCP 栈在等待 `Finished` 消息被响应之后才发送第一条应用数据记录（这是由于 Nagle 算法的原因）。由于服务器实现了延迟 ACK，所以服务器 TCP 栈没有发送 ACK。但是服务器没有输出的原因是它在等待客户端发送其第一条应用数据记录：于是出现了死锁。

当服务器的延迟 ACK 定时器触发时，死锁最终在 0.1516 秒解除。服务器还是发送了 ACK（第 6 和第 7 行之间），即便它不能采用捎带的方式。客户端在 0.1517 秒收到那个 ACK，并刷新自己的缓冲区，将自己的第一条应用数据记录发送出去。（注意，为了清楚地显示这一点，我们没有选择在数据包上显示捎带的 ACK。）

禁止 Nagle 算法

我们可以禁止 Nagle 算法（套接字 API 使用 `TCP_NODELAY` 选项）。如果我们禁止了 Nagle 算法，数据就会在提交给 TCP 栈的当刻被发送到网络上（当然要求 TCP 滑动窗口中

还有地方), 这样避免在图 6.25 中见到的那种死锁情况。图 6.26 描述了在客户端禁止 Nagle 算法的情况下, 同样的 SSL 握手。如你所见, 客户端在发送完 Finished 消息之后立刻将应用数据发送出去, 这极大缩短了完成握手所花费的时间。

```
New TCP connection: localhost(2830) <-> localhost(4433)
1 0.0006 (0.0060) C>S Handshake ClientHello
2 0.0016 (0.0010) S>C Handshake ServerHello
3 0.0016 (0.0000) S>C ChangeCipherSpec
4 0.0016 (0.0000) S>C Finished
5 0.0028 (0.0012) C>S ChangeCipherSpec
6 0.0028 (0.0000) C>S Finished
7 0.0046 (0.0018) C>S application_data
8 0.0059 (0.0013) S>C application_data
```

图 6.26 禁止 Nagle 算法情况下的会话恢复

恰当的进行处理

此刻显然要问的问题就是, 对于 SSL 到底值不值得使用 Nagle 算法。一般来讲, 答案或许是不。Nagle 算法的目的是为了减少传输小数据报所带来的 TCP 开销。

然而, SSL 记录几乎从不是小数据包。由于加上了头信息和 MAC, 一条 SSL 数据记录的最小绝对尺寸为 22 字节。因此, 如果 SSL 实现每条记录打包一个字节的话, 在到达 TCP 栈之前就已经扩展了 20 倍。而且, 正如我们前面所见到的, 每次处理一个字节的记录从加密计算上讲开销非常昂贵。

因此, 如果我们想要解决 SSL 的 tinygram 问题, 我们就必须安排在 SSL 实现将数据打包成记录之前对其进行缓冲。究竟需要什么东西要依赖于 SSL 在什么样的协议上运行, 我们将在第 7 章看到更多有关它们的内容。

6.24 握 手 缓 冲

正如我们在第 6.10 节所讨论的, 一些实现对 SSL 握手的各部分进行缓冲。OpenSSL 使用 1024 字节的缓冲区, 大部分时间都工作得相当好, 但是存在几处不尽人意的地方, 而采取另一种策略则会极大的降低延迟。考虑图 6.27 所描述的 DSS/DHE 握手情况。这次握手是在 10MB 以太网上完成的, 因此传输延迟非常低。然而, 我们仍然受到 Nagle 算法的影响。ServerHello、Certificate 和 ServerKeyExchange 消息无法容纳在单个 1024 字节的缓冲区中。因此, ServerKeyExchange 被分成两部分。第一部分与 Certificate 一起传输。第二部分则缓冲在缓冲区中并在 ServerHelloDone 之后刷新缓冲区时发送给操作系统。

然而, 由于缓冲区尺寸为 1024 字节, 第一段包含 ServerHello、Certificate 和部分 ServerHelloDone, 也是 1024 字节。这比以太网的最大段尺寸 (MSS) 还小, 因此为了 Nagle 算法, 这被看作是小数据包。这样, 直到前一段被响应之前不会发送包含 ServerKeyExchange 其余部分的下一段。正如我们在前一节所描述的, 客户端推迟发送 ACK 152 毫秒。在此期间, 客户端与服务器都是空闲的。一旦服务器收到 ACK, 它就会将 ServerKeyExchange 和 ServerHelloDone 的剩余部分发送出去。

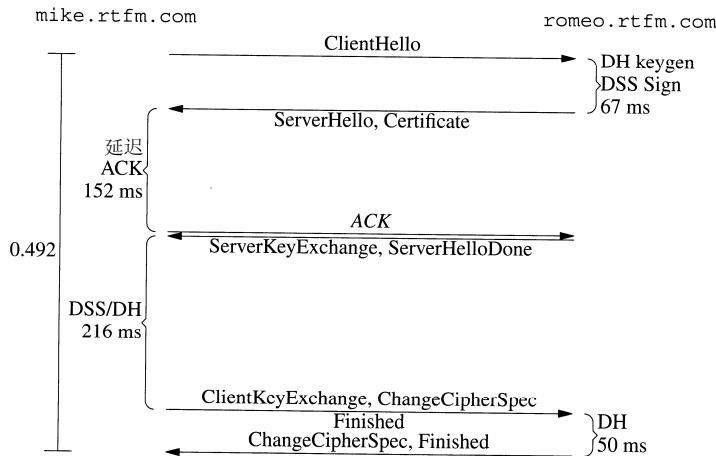


图 6.27 使用 OpenSSL 正常缓冲区的 DSS/DHE

OpenSSL 的多项功能共同导致握手手中延迟的增加。首先，OpenSSL 的缓冲区尺寸小而加上 Nagle 算法的影响会产生一段完全空闲的时期，服务器在等待 ACK 而客户端执行延迟的 ACK。解决这个问题的简单方法就是增加缓冲区的尺寸。图 6.28 描述了同样的握手，只不过缓冲区已经增加至 4096 字节。

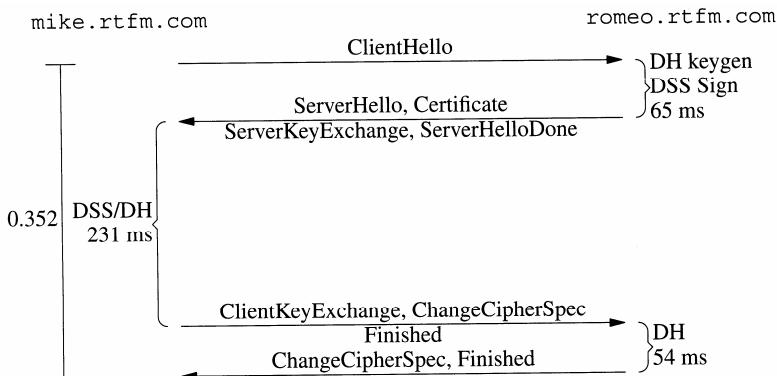


图 6.28 使用扩大了的缓冲区的 DSS/DHE

现在我们增大了缓冲区的尺寸，所有的服务器消息都可以容纳于同一个缓冲区中。这样，它们就会被同时发送出去。这消除了服务器等待客户端延迟 ACK 的条件，也就避免了我们先前所见到的临时性死锁。结果，这次握手完成花费了 352 毫秒，而不是使用较小缓冲区时的 492 毫秒。

注意，任何使用固定尺寸的缓冲区都是拙劣的解决方案。无论我们选用多大的缓冲区尺寸，证书仍有可能大到将其充满。一种更雅致的解决方案就是使用足以容纳其大小的变长缓冲区，该缓冲区可以自动调整大小并在传输完 ServerHelloDone 之后手工刷新缓冲区。注意，如果数据太大的话仍有可能在不止一个的 TCP 段中进行传输。然而，这样不会触发 Nagle 算法，因为除了最后一段，所有的都是全尺寸的。

除了增加缓冲区以外，还有一种方法就是将 Nagle 算法完全停掉，这与增加缓冲区尺寸有着相同效果。这样确实会稍微增加服务器的负担，因为要调用 write()两次而不是一次。然而，SSL 握手其余部分的高昂开销使得只多一次环境切换的开销相比起来微不足道。

增加并行性

另一项导致延迟的因素就是客户端与服务器之间的握手缺乏并行性。实质上，服务器在握手第一阶段的所有时间都花在产生 ServerKeyExchange 消息上。只有在产生 ServerKeyExchange 之后才从服务器发送任何消息。即便是在使用 1024 字节的缓冲区时，也是 ServerKeyExchange 充满了缓冲区并迫使将 ServerHello 和 Certificate 发送出去。

在产生 ServerKeyExchange 期间，客户端完全处于空闲状态。然而，存在一项它可以在期间完成的任务：验证服务器的证书。大家还记得第 6.14 节 DSA 验证耗费大量的 CPU 时间。如果在服务器产生 ServerKeyExchange 的时候，由客户端完成这项验证的话，我们就能够减少延迟。为了做到这一点，必须安排服务器在产生 ServerKeyExchange 之前发送 ServerHello 和 Certificate。简单的方案就是修改服务器使之逐条发送每条消息。而复杂的方案就是在写出 Certificate 消息之后刷新输出缓冲区。在任何一种情况下都需要关掉 Nagle 算法。

在对服务器进行恰当地修改后，我们可以预期客户端完全会在其收到 ServerKeyExchange 之前验证服务器的证书。这样，延迟应当减掉验证服务器证书所花费的时间。修改后的客户端经过实验显示有我们所期望的 20 毫秒的改进。

DSS/DHE 是一种极端的情况，客户端与服务器都要求完成大量的处理，于是并行带来的性能补益是巨大的。然而，它演示了一种通用的法则：为了获得最好的性能，SSL 实现者需要非常仔细地了解实现与网络之间的交互。尤其是当一方在传输的数据需要另一方完成大量的处理时，应当尽可能早地将数据传送到网络上以达到最大限度的并行性，继而也就减少了延迟。一条好的经验法则就是在进入任何耗时的操作之前刷新网络缓冲区。

6.25 高级 SSL 性能法则

避免使用临时密钥资料。临时 DH 会降低服务器的速度，除非需要 PFS(完美向前保密)，否则就不要使用它。即便使用固定 DH 密钥的 DHE 模式也是一种改进。

使用短 DH 指数。256 位的 DH 指数会极大的改善性能，而在安全上只稍稍减弱一点。

Nagle 算法。总是关闭 Nagle 算法。

握手缓冲。在进入任何耗时的操作之前将数据传送出。注意缓冲区刷新与 Nagle 之间的相互影响。

6.26 总 结

本章描述了 SSL 的性能属性。我们从总体观察一些控制 SSL 性能的因素开始，然后使用加装测量功能的软件和网络跟踪信息对它们的细节进行了详细地描述。

大多数时间都花在了加密上。加密操作是 SSL 性能中的主导因素。协议状态机和 I/O 对系统性能的影响可忽略不记。

握手开销昂贵。对于不传输大量数据的会话，SSL 握手耗费了其中大部分的 CPU 时间。

会话恢复可以减少握手的开销。恢复会话的握手比普通会话握手快很多。如果客户/服务器对反复进行通信，恢复会话就会物有所值。

服务器性能比客户端性能更重要。慢速的服务器会增加延迟减少通量。慢速客户端主要会增加延迟。

使用大记录。处理任何记录都要耗费一定最少数量的加密与网络开销。记录尺寸越大，带来的开销的次数就越少。

注意算法的选择。想要获得美好的握手性能就使用 RSA。想要快速传输数据就使用 RC4。摘要算法的性能所起的作用不大。

如果你需要可以接受的性能，就使用 C。Java 密码运算性能糟糕透了。通过使用 JNI 调用 C 实现可以极大地改善 Java 实现的性能。

如果你想要高性能，就使用加密硬件。对于高负载的服务器，硬件加密引擎可以极大地改善性能。

7

使用 SSL 进行设计

“首先，搞清楚你要做什么（这在大多数情况下是一个好的建议，在此尤其适用）”

——NNTP 安装指南

7.1 介绍

到目前为止，我们的注意力都集中在 SSL 自身的机制上。然而，SSL 本身并没有什么用处，它只是用来承载应用层协议的。是这些应用层的协议来为用户完成各种服务的。SSL 的工作就是尽可能多地将安全功能抽象出来，允许这些协议安全地执行它们的任务。SSL 以一种通用方式来处理安全机制，应用层协议就在其上运行，利用它所提供的各种安全服务。

然而，在实际应用中，在 SSL 连接上简单地承载应用数据往往会导致事与愿违的安全与性能瑕。应用层协议（和协议设计者）需要对 SSL 有足够了解后才能在其上安全运行。本章涵盖了为完成这项任务所要了解的知识。与往常一样，我们将内容分成两个部分，第一部分概括性地描述基本概念，第二部分则更详细地讲述其中的一些技术性更强的内容，同时介绍在开始一带而过的部分要点。

本章中的所有内容都是针对从应用层角度来保护应用协议安全的问题。内容面向应用协议设计者和系统设计师。第 8 章和本章配套，从程序员的角度讲述了这些技术，并提供了使用常见 SSL 实现的范例代码。

7.2 了解要保证什么的安全

需要搞清楚的第一件事就是，你想提供什么样的安全服务。为了做到那一点，需要评估自己应用的安全模型。如果你已忘记了什么是安全模型，请重新阅读一下第 1 章的内容。了解了安全模型后，接着搞清楚你想保护的系统弱点都有哪些。你的目标就是保证所有可以经济地保证其安全的弱点的安全。

作为一种提示和指引，请回忆一下我们在第 1 章所讨论的三种安全服务：保密性、消息完整性和端认证。并不是所有的应用都需要所有这些服务（尽管一个什么也不需要的应用不值得保证其安全），在某种程度上可以独立提供这些服务。因此，重要的是识别出你所需要

的服务，而这个过程的一部分就是了解提供这些服务的开销。服务的开销越低，提供它所需要的论证就越少。

没有必要让 SSL 来提供所有的安全服务。例如，让系统使用用户名和口令来提供客户端认证相当常见。尽管口令加密是通过 SSL 提供的，但应用协议负责口令的传输、接收和检查。现在需要识别出都有哪些服务最好由 SSL 来提供，而哪些最好由应用协议来提供。

● 保密性

在大多数应用中保密性都是有用的。然而，正如我们在第 6 章所见到的，它们都带有一定的性能开销，特别是在 RC4 不可用的情况下。此外，还有一些不需要保密性的应用。一个好例子就是自由软件下载。当你下载软件的时候，想知道下载过程中没有被人修改，但由于任何人都可以下载它，因此对内容进行保密没有任何价值。

● 消息完整性

消息完整性是 SSL 要求的部分。此外，若没有消息完整性，则提供任何安全服务都是困难的。因此，对是否使用 SSL 的消息完整性功能实际上不存在疑问。

● 服务器认证

惟一不需要服务器认证的 SSL 模式为匿名 DH 模式。然而，对于任何 SSL 模式而言，使用自签名的证书（因此是无法验证的）当然也可以获得匿名连接的效果。这样并不会令人满意，因为这使连接处于中间人攻击的威胁之下。因此，总的来说，几乎所有的 SSL 应用都要求服务器证书。

也存在某种特殊情况，中间人攻击可被应用层协议加以阻止。我们将在第 7.16 节讨论这样一种技术，这些技术通常要求应用层协议与 SSL 之间紧密集成。

● 客户端认证

正如我们所讲到的，使用 SSL 往往将你锁定在使用基于证书的服务器认证模式。然而，证书并不是客户端认证的惟一选项，许多应用协议集成了它们自己的客户端认证机制，它们能够在 SSL 上运行。尽管当以明文的形式在网上运行这些机制时有可能不安全，但是在 SSL 上运行时则要安全得多。下一节我们讨论其中几种比较简单的客户端认证选项。

● 经验法则

正如我们以前所谈到的，消息完整性不是可有可无的东西，所以在此别无选择。一般来讲，最好提供保密性。许多 SSL 实现都不支持只进行认证的模式，所以为了满足互操作性，你需要提供保密性。此外，它符合用户的期望。一般来讲，客户端认证对用户来说是令人讨厌的东西，所以除非你确实有一些需要限制特定用户才能存取的数据，否则就应避免使用客户端认证。

7.3 客户端认证选项

● 用户名/口令

最传统的进行客户端认证的方法就是使用用户名/口令。用户将用户名和口令提供给网

络客户端，而它则将其提供给用户要存取的服务器。服务器根据口令数据库检查用户口令。如果口令检查通过，那么网络客户端就会被认证为用户。

这种方法通常与 ACL（存取控制列表）机制结合起来使用。一个 ACL 就是一张包含用户和每个用户所具有的权限的列表。服务器上的每种资源都有一个与之关联的 ACL，当客户端请求存取一种资源的时候，服务器就检查与那个客户端关联的用户在相应的 ACL 中是否具有该权限。

用户名/口令方法受制于一种简单的被动攻击：攻击者从线路上嗅探口令，然后初始化一次新的会话，将嗅探到的用户名/口令对作为自己的用户名/口令来提供。SSL 可以阻止这种攻击，因为口令是在经过加密的通道上发送的。然而，攻击者仍有可能通过反复猜测口令发动主动攻击，以期猜对口令。

用户名/口令的变种

由于如此多的系统都设计有用户名/口令安全系统，所以设计了许多复杂的模式用以在维持通用用户名/口令模型的同时改善安全。最简单的就是诸如 S/Key 或 SecureID 卡这样的一次性口令模式。总的来讲，这些模式都是设计用来在对传输口令的连接进行任意的主动和被动攻击时口令的安全。因为 SSL 能够阻止这些攻击，这些模式通常在使用 SSL 的环境中没有什么优势。然而，在一种既有非安全连接又有安全连接的环境中，这些模式还是有用的。在 [Jablon1996] 和 [Wul1998] 中描述了这种模式的两个例子。

SSL 客户端认证

与服务器认证一样，SSL 中的客户端认证依赖拥有一份数字证书。然而，由于客户端相对服务器的巨大比例，客户端认证所提出的操作性挑战要比服务器认证大得多。对大量客户授权常常需要与 CA 进行特殊磋商。为了避免这种安排，许多组织都选择运行它们自己的证书审批机构，但这同样也会造成操作负担。

基于客户端认证的一个附加问题就是将证书映射为用户标识，只要基于证书中的身份来查找 ACL 条目就可以将 ACL 与证书一起使用。当结合使用基于证书的认证以及用户名/口令类型的认证时就需要特别留意，因为系统必须将两种类型的身份信息映射成一种共同的用户格式。

经验法则

在大多数情况下，提供用户名/口令认证更容易一些。口令可以更容易地与大多数基础结构整合在一起，而且用户也更容易理解。作为一种例外，使用证书的自动客户端常常也一样易于管理，尤其是在不需要存取控制的情况下。这种客户端的一个很好例子就是邮件服务器，它运行时无须外部干预而且在转发消息时能对自身进行认证。

7.4 引用完整性

服务器身份

正如我们在第 5 章所讨论的，要确保 SSL 安全连接的安全就必须验证服务器的身份。没有这项检查，主动攻击者就能够针对该连接实施中间人攻击。此外，不仅要对服务器的证

书进行验证，还要验证这个服务器就是你期望与之交互的服务器。不然，具有有效证书的服务器就有可能拦截另一服务器验证的连接。

自然，将所期望的服务器身份与服务器的实际身份进行比较就要求掌握所期望的关于服务器身份的信息。因此不管使用什么样的应用协议，这些协议都必须让你对有关服务器身份的有所预期。

一般来讲，这种信息以服务器域名的形式出现，必须将其与第 5 章所描述的证书进行比对。该域名通常要么由用户直接提供（如，Telnet 请求中的主机名），要么间接地通过 URL 提供。在这两种手段都不存在的情况下，情况就要困难得多。我们将在第 7.16 节考虑一种该情况下的可行方案。

安全属性

由于安全的原因，客户端需要验证连接属性是否符合其安全期望。这些期望的具体细节依赖于客户端所掌握的服务器的引用类型。例如，对于像 Telnet 那样的协议来说，客户端可能只有服务器的域名。然而，对于像 HTTP 这样的协议，客户端常常在 URL 中指示该协议应当是 SSL 上运行的 HTTP 协议（URL 以 https://开头）。

如果引用指示使用 SSL，那么客户端需要检查实际上使用的就是 SSL。如果有域名可用，客户端就需要检查服务器证书中的域名是否与引用的域名匹配。如果不知什么原因没能对服务器进行认证，那么就有可能遭受中间人攻击。

需要指出的重要一点是，安全连接的实施一般应在客户端完成。如果客户端不强制使用安全连接的话，攻击者就能够成为中间人，与客户端用非安全连接进行连接而又初始化一条到服务器的安全连接。这样，服务器就会认为客户端使用的是安全连接，而无法判别这条安全连接是到攻击者的还是到客户端的。这种类型的攻击会使服务器推行的安全属性出现问题。如果服务器要求 SSL 客户端认证的话，就不会出现这种情况。如果使用了客户端认证，攻击者可以连接到服务器，但却不能冒充客户端。

经验法则

总的说来，你需要提供这样一种引用①具体标识出客户端希望与之连接的服务器；②指示要求使用 SSL。需要能够自动将服务器的身份与服务器的证书进行比较。应能通过域名来识别服务器，而这种信息也应当出现在证书中。

7.5 不适合的任务

尽管 SSL 对大量的安全任务来说都是有效的，但是还存在相当数量的不适合使用 SSL 的工作。大家知道 SSL 在两台机器之间提供一条安全通道，而这条通道对在其上传输的数据是透明的，这就产生了 SSL 用途的一些限制。这一节就讲述那些限制。第 11 章简要描述了其他一些可以填补这些缺憾的技术。

不可抵赖性（Nonrepudiation）

无法使用 SSL 来完成的一种常见任务就是提供数据的不可抵赖性。考虑进行联机购物的情况。交易是在 SSL 连接上完成的，因此你可以确信自己是在与正确的商家进行交互。

商家通常会发给你一份收据，因为这种数据是在 SSL 连接上传送的，所以你可以确信它没有被篡改。然而，你无法拿给任何别的人去看，以及让他人在 SSL 之外对内容进行验证。因此，如果商家声称你伪造收据的话，你也没有办法证明。

● 端到端的安全

由于 SSL 握手是交互性的，所以就不可能向无法同其确立网络连接的机器发送加密或认证数据。这样，如果你的机器位于防火墙后面，若是想与另一端建立网络连接的话，就必须在防火墙上开个洞。另一种办法就是建立到防火墙的 SSL 连接，并让防火墙建立从防火墙到那一端机器的 SSL 连接。这种方案有一个主要的缺点，那就是防火墙现在可以读取你的所有通信数据。

总的来说，SSL 在任何涉及以这种先存储对象并保持其加密属性的不变，而后又加以恢复、验证的方式，来保护对象的任务时都表现得十分拙劣。如果你有这样的应用，就应当仔细考虑这种不适合使用 SSL 的可能性。更好的解决方案就是使用像 S/MIME 这样的消息安全协议，我们将在第 11 章讨论这种类型的解决方案。

7.6 协议的选择

能够同时运行同一种应用协议的安全与非安全版本是一种相当常见的愿望。如果对于给定的协议，这种能力符合人们的愿望，那么就必须有某种区分使用原始协议（非安全的）与安全化的协议版本连接的方法。

对于并行使用安全与非安全连接，我们有两个要求：第一，必须能够明确区分安全与非安全连接。第二，没有安全意识的客户端与服务器必须能够正确地与有安全意识的客户端和服务器进行互操作。

这里需要对第二条稍做解释。自然，对安全一无所知的代理无法与期望安全的代理安全地进行互操作。然而，一种代理企图同一种没有安全意识的代理切磋安全一定不能打破没有安全意识的代理。显然抛出一个错误是允许的，但它应当是一种能够被理解的错误。

一种方案就是让服务器基于客户端发送的头几个字节自动进行检测。这种方案有赖于 SSL ClientHello 与应用层协议存在足够数量的服务器能够辨识的差异。对于大多数协议来说均不是这回事儿。

此外，即便这两种协议存在差异，这种方案还要求服务器能够自动检测出这种差异。这项要求使得在将 SSL 改制成一种应用层协议时非常困难，因为已有服务器不会知道 SSL 握手的事，因而会返回错误或停滞等待，若是后者则更糟。尽管在设计一种新协议时，自动检测或许是可接受的（但却是不雅致的），但在保护现有协议的安全时则是完全不可接受的。

存在两种常用的协议选择方案：独立端口（separate port）以及磋商升级（upward negotiation）。

● 独立端口

大多数因特网协议都有着称做众所周知端口（well-known port）的端口。例如，Telnet 连接站用端口 23，而 SMTP（电子邮件）为连接站用端口 25。请参见 [IANA] 来了解包含当

前所有以已分配端口号的列表。IANA（因特网编号分配机构）控制着这些端口号的注册，以此来确保不会有两种注册的协议共用同一个端口号。显然，如果两种不同的协议在同一个端口上产生冲突，我们就会遇到上一节所讨论的同一种类型的问题。

采用独立端口策略意味着我们要给协议的安全版本分配一个与非安全版本不同的 TCP 端口。服务器只要通过查看客户端连接到哪个 TCP 端口上就能知道这条连接是安全的还是不安全的。然后它就会知道期望的是 SSL ClientHello 还是立刻开始应用层协议。独立端口策略的主要问题就是它的伸缩性不怎么好。可用的端口数非常多（大约有 65000 个）但是却不是无限的。如果我们为每种协议都定义两个端口，那么就会加倍耗尽端口号空间。此外，这样会开创一个危险的先例。当我们想要增加其他一些新功能时，我们还要为其定义新的端口号吗？我们还需要定义这项功能的非安全的版本和安全版本吗？这种潜在的组合爆炸的影响十分巨大。

磋商升级

通常，当我们想要给一种现有协议增加新的协议功能时，我们会找到某种以向后兼容形式在现有协议中引入它们的方法。只有在对协议进行大的改动时，我们才使用新端口。例如，在 SSL 中，我们通过定义新的加密套件号及其相关的语义就能增加新的加密套件。协议会自动判定这些加密算法是否可用。

许多其他的协议都允许这种类型的扩展。因此，自然可以使用这些机制来引入安全性。在一种典型的此类机制中，一方会表示它支持 SSL，而另一方将会表示它想要使用这种协议。接着就会进行握手，而其余的应用层协议将会在受保护的连接上运行。

图 7.1 以一种简化形式展示了一种典型的此类磋商。第一条消息中，客户端表示它作好了使用 TLS 的准备。第二条消息中，服务器同意使用 TLS 并告诉客户端继续进行。下一条消息中，客户端发送其 ClientHello。客户端与服务器完成握手，接着客户端在新建的 TLS 连接上发送第一条应用层协议消息。这条消息用斜体来显示，以表明它是加密的。

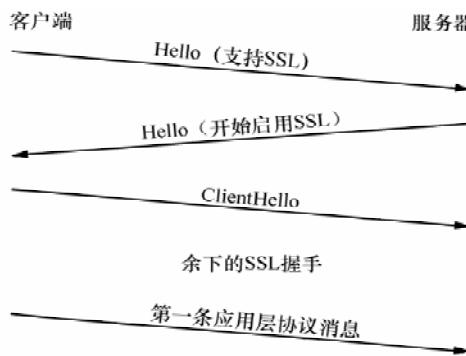


图 7.1 磋商升级为 TLS

磋商升级要求协议已经具有某种磋商扩展机制。这是一种常见但绝非普遍存在的功能。如果协议没有这样一种机制，那么就得安插这样一种机制。因为切换到 SSL 是在正常协议交换期间发生的，必须加倍小心才能确保足够的安全。在 SSL 会话磋商完成之前不应当传输任何敏感数据。客户端与服务器也需要配置成在另一方拒绝切换至 SSL 的情况下能够正确地（安全地）操作。

经验法则

总的说来，设计与实现一种独立端口的策略要容易得多。如果你设计的协议是供内部大量使用的，或是使用的是一种你不能更改（并不少见）的协议，那么就使用独立的端口。惟一当真值得花费精力设计和实现磋商升级策略的情况就是在设计是为了大范围的标准化时。

7.7 减少握手的开销

大家还记得第 6 章中大多数 SSL 连接的主要性能开销就是握手。作为应用协议设计者，为减少开销所能做的主要工作就是将协议所需的新的握手次数减至最少。

如果你是在设计一种新协议，减少新连接是第一位的考虑。然而，即便你是在保护现有协议，思想上有这种考虑也是重要的。一些协议（如 HTTP 和 SMTP）能够在同一个协议连接上处理多道事务。这总体来讲是一种好的想法，但是对于 SSL 来说却尤其重要。

7.8 设计策略

到目前为止，我们已经讨论了使用 SSL 来保护协议的主要设计考虑。本节对你应当历经的设计步骤以及步骤的执行顺序进行一个宽泛地描述。

识别你的威胁模型。任何安全设计的第一步就是搞清你试图抵御的安全威胁类型。需要搞清楚你的数据所具有的敏感形式。这一过程直接导向下一步工作。

搞清楚你要保护什么的安全。首先搞清楚你希望提供的安全服务有哪些，以摘要的形式记录下来。接着弄清楚其中的哪些可以使用 SSL 来提供以及哪些得通过 SSL 以外的东西来提供。现在就是考虑 SSL 是否适合你的应用的时候了。

挑选一种协议选择策略。你需要做出的第一项设计决定就是怎样选择安全或非安全模式。正如我们在前面所讨论的，独立端口策略更容易实现，但存在一些伸缩性的问题。

决定如何（是否）提供客户端认证。如果你的威胁模型要求对用户进行认证，那么就需要使用某种客户端认证。如果你的应用已经使用了用户名/口令认证，你或许想重用这种认证模式。即便选择使用证书，你或许也想利用用户名/口令来进行存取控制。

识别服务器并提供引用完整性。下一步就是搞清楚如何对服务器进行认证。几乎可以肯定你会使用证书来完成这项工作，但是重要任务是搞清楚如何将引用内容与证书中的身份进行比较。这要求识别出你所拥有的引用类型并安排证书依样拷贝。

优化连接语义。最后，你想要查看一下各种性能考虑。在协议一层，你所做的大部分工作都是使连接的次数保持为最少，而且会话尽可能地是可恢复的。此外，还要确保关闭操作能正确执行，因为这样既减少了再握手又提供了抵御截断攻击的安全保证。

7.9 小结

本章的第一部分讨论了使用 SSL 保护协议安全的一般性策略。总的来讲，有可能通过

很少的努力就能完成基本的保护协议安全的工作，但是存在几个应当考虑的重要细节。正如我们所见到的，没有考虑这些细节就会留下重大的安全漏洞，而没有考虑其他的一些细节则会导致糟糕的性能。

第一部分的目的就是提供对所需任务的概括描述。本章的第二部分详细讨论了执行这些任务的各种策略。其中我们要讨论的大多数策略都已经在一种或多种协议以及部分标准 SSL 设计工具箱中实现了有效的部署。

7.10 独立端口

最流行的操作并发安全与非安全服务器的方法就是针对每种协议变体使用独立的端口。独立端口策略的实现非常简单，只需给协议的安全变体分配一个新的端口，并安排服务器同时在非安全端口和安全端口上监听就是了。这可以实现为让一个服务器进程同时监听两个端口或是每个端口由一个进程进行监听。没有必要区分 SSL 通信数据与普通的通信数据，因为任何从安全端口上过来的信息肯定都是 SSL 形式的。任何从安全端口上过来的非安全通信数据都会导致出错，对于从非安全端口上过来的任何安全通信数据来说也是一样。

惟一棘手的地方就是如何安排客户端正确地完成工作。首先，客户端必须知道如何使用 SSL。其次，它必须知道要连接到安全端口上去。如果该安全端口是众所周知的，那么选择哪个端口就很容易。但如果不是这样，就得另外通知客户端。最后，客户端还需要知道如果它不能就安全连接达成一致时又该如何行事。要它回过头去使用非安全连接还是报告失败？我们将在本章后面讨论引用完整性的时候讲到这个问题。

这种策略的主要优势就是它不要求对应用层协议做任何真正修改。你需要在 SSL 上描述协议的行为，但是所有的协议命令都不变。此外，现存的非安全实现可以像往常一样工作。不会有安全与非安全实现之间交互而带来的风险，因为它们根本就不会进行交互。

独立端口策略的主要缺点就是它所消耗 TCP 的端口的速度是磋商升级策略的两倍。由于许多协议设计者都喜欢用小于 1024 的数字来表示众所周知的端口号。在 UNIX 机器上，只有有效用户 ID 为 0（根用户）的进程才能打开端口号小于 1024 的端口。因此收到从小于 1024 的端口发出的数据包就被认为暗指发送方具有根用户权限。类似的，连接到服务器的低编号端口上就意味着该服务器是由机器所有者正式认可的，因为只有根用户拥有的进程才能在那种端口上监听。

显然，面对因特网威胁模型，这是一种荒唐可笑的鉴别措施。伪造低编号端口的数据包简直就是小事一桩。而且其他的操作系统并不强加这种低端口号的约定，因而无法区分你是否是在与具有与前面所提到的这样一种操作系统的机器进行交互。尽管如此，仍有许多设计者使用倾向于低编号端口，所以他们担心这些端口会被耗尽。

这种策略的另一个问题就是它要求对某些防火墙进行修改。许多包过滤防火墙都被设置成除了一些特定的端口外不允许任何 TCP 连接。必须对这种防火墙进行修改才能支持这种策略。最后，如果你是在为 IETF 标准设计协议，就应当谨慎地使用独立端口策略。在编写这本书的时候，IESG 强烈建议协议设计者使用磋商升级。

7.11 碰商升级

碰商升级策略对安全连接与非安全连接都使用同一个端口。这种方法的明显优势就是不再需要分配额外的端口，从而消除了与之有关的问题。

此外，碰商升级还增加了一项重要的新功能：自动发现。不管客户端还是服务器都能提供安全，而且如果另一方接受的话，就能自动进行碰商。与之对照，独立端口策略多少要求客户端知道安全是可用的。这项功能增添了很强的抵御被动嗅探攻击的能力。然而，如果没有仔细地加以实现，协议就会受制于主动降级攻击，在后面我们会看到这一点。

碰商升级策略要求对客户端和服务器上的代码都进行大的改动。如果协议没有增加扩展机制，就必须增加一种。如果有这种机制，则必须增加对碰商安全的支持。这在第一次设计时并不总能正确工作。HTTP 中的机制称做升级（Upgrade），在 TLS 使用之前根本就没有使用过它，因此遇到了一些问题，我们将在第 8 章对此进行讨论。

其次，定义失败握手的语义很重要。仅仅是客户端与服务器都支持 SSL 并不意味着它们都支持兼容的加密套件或密钥资料。如果 SSL 握手失败了，那么应用就可以断开连接并以非安全方式重新连接，或不重新进行连接而以非安全连接继续执行，或者只是报告错误。第二种方式要求确信另一方的网络缓冲区处于明确定义的状态，这样 SSL 数据才不会被不经意地当作应用协议数据来处理。

碰商升级也存在一定的性能开销。当在独立的端口上运行安全与非安全协议时，第一个数据包就是 ClientHello。然而，如果采用了碰商升级，那么客户端与服务器必须先确定是否允许 SSL。这要花费一个来回的时间才能确定（如果你在第一条协议消息上捎带 SSL ClientHello 的话或许要不了一个来回）。

最后，尽管支持碰商升级策略无须对包过滤防火墙进行修改，但对应用层防火墙代理的影响却是巨大的。由于应用层代理必须能够识别碰商升级命令而不予介入，这常常要求的不是重新对代理进行简单配置，而是要对代码进行修改。这是一种相当大的部署障碍。

7.12 降 级 攻 击

这两种协议选择方法都易于受到主动的降级攻击。在这种攻击中，攻击者说服客户端或服务器使用 SSL 是不可能的，这样就迫使各方屈从于通过非安全通道进行通信，前提是它们无论怎样都要进行通信。需要指出的重要一点是，所有这些攻击都不是通过对 SSL 握手本身实施降级来工作的。那样的攻击由 Finished 消息中的摘要所阻止。相反，这种攻击是通过表面看起来无法碰商使用 SSL 连接来工作的。因此，不存在防止这种攻击发生的 SSL 协议措施。

独立端口

针对独立端口碰商而实施的最简单的攻击就是使服务器看起来根本不在相应的端口上进行监听。对于任何能够向网络中注入数据包的攻击者来说要想做到这一点没有丝毫困难。攻击者只管等待客户端发送 TCP SYN 打开连接，然后伪造一个 RST 作为应答。客户端没有



办法检测出这个 RST 来自于攻击者而不是合法的服务器，因此 TCP 栈会向客户端代码返回一个错误。

在正常情况下，这只不过是一种简单的拒绝服务攻击，不值得大惊小怪。然而，客户端的（或用户的）行为有可能使情况糟糕得多。许多安全服务器的 Web 页面都说明了这种问题。这样的服务器想要能够支持 SSL，但并不想拒绝非 SSL 交易。例如 amazon.com 的 Web 服务器就包含下列文字：

（如果当你试图使用我们的安全服务器时收到错误信息，那么就使用我们的标准服务器签到。如果你选择的是安全服务器，那么你所输入的信息将会被加密。译者注）

如果用户率真地遵循这些指导而点击了标准服务器链接，那么攻击者通过在用户尝试安全链接时模仿一条错误信息就能轻而易举地迫使任何连接指向非安全模式。如果客户端自动回退使用非安全模式的话，情况甚至会更糟。不管是哪一种情况，攻击者所得到的都要比拒绝服务攻击多得多。它是一种主动的保密性与完整性攻击。

自然，这并不是攻击者促使看起来无法创建 SSL 连接的惟一方式。其他的方法有伪造不含加密套件的 ClientHello 和声称无法进行磋商的服务器警示。

● 磋商升级

针对使用磋商升级策略的协议实施降级攻击与对使用独立端口策略的稍有不同，但是原理是一样的。攻击者伪造来自于服务器（或客户端）的消息，表明它不愿意使用 SSL。这意味着攻击者必须能够接管 TCP 连接，但是要做到这一点并不困难。图 7.2 在此展示了协议消息的流程。

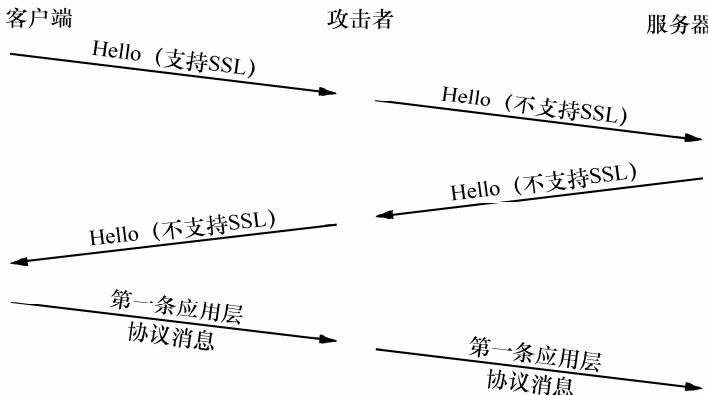


图 7.2 针对磋商升级的降级攻击

同图 7.1 中所示的工作中的磋商升级范例一样，客户端试图形成一条到服务器的 TCP 连接。然而，这一次它形成了一条到攻击者的连接。在图 7.2 中，我们展示了攻击者连接到服务器并冒充客户端，但是在许多情况下，攻击者只能冒充服务器。客户端请求 SSL，但攻击者拒绝了这种请求。接下来所发生的情况依赖于客户端是如何配置的。

在许多情况下，如果无法商定使用 SSL，客户端会被配置成没有 SSL 也继续进行下去。如果一切顺利，我们会有机会商定加密算法，但是如果商定的话也不用担心。在只有部分而不是全部服务器支持 SSL 的环境中，客户端对任何服务器都没有预先了解，这是惟一一种明智而且具有互操作性的配置方式。

然而，如果按照这种方式进行配置，客户端就完全处于该种降级攻击之下。正如我们在图 7.2 中所展示的，它会使用普通交易继续进行下去。这样，伺机（opportunistic）加密提供了优秀的抵御被动攻击的保护，但对主动攻击来说却没有什么保护。解决该问题的一种部分解决方案（在 RFC2487 [Hoffman1999a] 中推荐的）就是记住那些你曾商定使用安全连接的服务器，对于那些要坚持使用安全连接。这在你第一次与某个新服务器打交道时没有任何帮助，但是它会防止针对你已经联系过的服务器的降级攻击。

即便服务器要求 SSL，攻击者常常也能通过对客户端冒充服务器并作为代理转发客户端的请求。只要客户端不要求使用 SSL，这种方法就能工作。服务器能够成功推行 SSL 的惟一一种情况就是要求 SSL 和客户端认证。在那种情况下，攻击者无法冒充客户端。然而，这要求所有的客户端都支持 SSL 而且拥有证书。其结果会导致绝大多数的客户端都将无法进行连接，这常常是不可接受的。

应对措施

正如我们所看到的，这两种协议选择策略都受制于降级攻击。对于独立端口策略而言，只要用户稍不注意就会彻底丧失安全。合法服务器作为一种对出错的响应，鼓励采取手工降级处理的事实使这种攻击甚至更加容易。

当使用磋商升级策略的时候，这种情况甚至更糟。在处于攻击之下时，磋商升级的主要优点——自动升级——就会成为自动降级的缺点。为了进行互操作，引用内容无须包含要求安全的指示，因为根本就不引入这样引用的情况是十分常见的。结果就没有办法自动强制使用安全连接，从而使协议完全处于降级攻击之下。

针对这种类型的攻击要求有两种应对措施，一种是技术性的而另一种是社会性的。技术性措施直截了当。引用服务器的引用必须包含这样一个事实，即它是一个安全站点而且至少还有一些所期望的有关你能够磋商使用的连接类型。SSL 会对磋商过程提供保护。如果能够连接到站点并验证证书，那么就可以确信你没有遭受降级攻击。

社会性的问题就要难一些。必须教育用户和程序员在安全连接失败的情况下不要进行非安全连接。如果你看到一个示意安全的引用，几乎可以肯定以非安全方式引用它是不安全的。很难教授用户掌握这种行为，但是如果这样的话，系统安全简直就不值一提。

7.13 引用完整性

引用完整性的目标就是确保你所连接的服务器与用户（或替用户代劳的程序）意图连接的服务器一样。因为几种原因，这是一项困难的任务。

首先，服务器提供的身份（证书中的标识名）并不是用户友好的。更重要的是，用户也不怎么愿意留意他们所连接的服务器的身份，因此必须要自动完成这一过程。许多使软件界面更加友好的企图结果都会内在的造成软件难以对身份表示进行解析。例如，让软件来匹配域名非常容易。然而，随着软件逐渐将域名隐藏起来以支持“因特网关键词”，即 Web 页面的名字或用户的个人姓名。那么将这些名字形式映射为域名则要困难得多。

引用完整性最突出的要求就是应用层所期望的身份与服务器证书所展示的应用层之间存在一条信任链。最好的情况就是只包含一个环节：证书中的身份与应用层的身份相匹配。

不幸的是，这只有在某些情况下是可能的。

如果不是那样，应用层身份就必须能够安全的与某种其他的身份形式相关联。这种身份形式又必须依次与第三种身份形式关联。这一过程持续下去直到身份能够与服务器证书进行比对为止。自然，如果这条链很长，那么就必须能够自动进行验证，因为用户根本就不会花时间做这种验证。我们需要关心的另一种主要形式就是 IP 地址（即原始网络地址）、alternate DNS 名以及可供人识别的（human-readable）名字。

IP 地址

正如我们在第 5 章所讨论的，可验证身份链的要求就是为什么证书一般不包含 IP 地址的原因。总的来讲，DNS 不提供安全的将名字映射成 IP 地址的方法。Secure DNS [Eastlake1999] 提供了一些解决该问题的保障，但是其部署目前还远未普及。

作为一种特殊情况，如果应用层身份是以 IP 地址的方式来表达的，那么证书包含 IP 地址就是可接受的，因为可以安全地对它们进行比较。然而，一般来说，因为几种原因，这是一种糟糕的做法。第一，IP 地址通常是通过使用 nslookup 或类似工具解析 DNS 域名得到的。我们刚刚说过，那是不安全的。其次，服务供应商变更或配置发生改变都会导致地址的重新编排，因此它们不会有域名那么持久。很难让用户知道什么时候这些地址重新编排是合法的，以及什么时候不合法，因此很容易诱导用户接受盗用地址的连接（spoofed connection）。

Alternate DNS 名

将域名映射为 IP 地址并不是 DNS 中惟一一种不安全的情形。由于诸多原因，将单个 DNS 名映射为多个其他的 DNS 名相当常见。两种最常见的情况就是 CNAME 和 MX 记录。CNAME 通常用来在对外界公布的名字与实际机器名之间提供一层间接，MX 记录用于 E-mail 路由，我们将在第 10 章讨论 SMTP 时详细讨论它们的特殊问题。

CNAME 记录两种最常见的用途就是简单别名和负载平衡。设想你运做一家已经大到想拥有自己的 Web 站点的公司，但还没有大到由自己来运行这样一个 Web 站点的地步，于是你将这项工作转包给你的 ISP 来做。尽管如此，你还是想让自己的顾客能够通过你的公司名，如 www.example.com 来引用公司的 Web 站点。但实际上这台机器是由你的 ISP 来运做的，而它已经有了一个名字：web1.isp.com。CNAME 可以在这两个名字之间提供别名。

CNAME 代表规范名（canonical name）。意思就是说 web1.isp.com 为机器的规范名而 CNAME 记录包含了规范名。

现在，当顾客在他们的浏览器中输入 www.example.com 时，就会自动检测出它实际是要与 web1.isp.com 进行通信，于是就查找它的 IP 地址并连接到那台机器上。如果 ISP 决定对这个 Web 站点进行搬迁，那么只需改变 CNAME 的目标即可。更棒的是，如果你决定自己来管理这个 Web 站点的话，只要将 CNAME 改为指向你建站的机器即可，如 server.example.com。

设想你现在的负载变得非常高，以至于单台机器再不能处理得过来。这里，CNAME 也能帮上忙，因为你可以安排名字服务器每次返回不同的 CNAME，这样就可以在多台服务器之间共同分担负载。

每当我们使用这种形式的间接时，就需要小心考虑证书中出现的名字应该是哪个。从安

全的角度来看，答案很明显，但是从操作的角度来看，却又很难。因为 DNS 查找是不受信的，因此不可能创建从名字 www.example.com 到任何由 CNAME 所指的名字的信任链。因此，证书中出现的地址必须是 www.example.com。

不幸的是，在证书中使用原名（CNAME）存在操作上的问题。在 ISP 运行你的 Web 服务器的情况下，这会要求它们取得多个证书（每个证书针对一位顾客）或具有多个别名（多个 Common Name 或 subjectAltName）的单个证书。在运行多个服务器的情况下，你需要在所有这些机器之间共享你的证书（以及私用密钥！），从而增添了安全风险。

在 Web 环境中，一种常见的解决这种问题的办法就是拥有一个只能通过 HTTP 来存取的起始页面，并让它使用一个与安全服务器的证书匹配的 URL 指向安全服务器。所以在这个例子中，实际将你转到安全服务器的链接有可能是一个指向 https://secure1.isp.com 的链接，这种 URL 与用户期望之间的不匹配有它自己的安全问题，我们将在下一节花部分篇幅进行讲解，并在第 9 章讨论 SSL 上的 HTTP 时更具体地讲到这个问题。

可供人识别的名字

要考虑的最后一形式的身份就是可供人识别的名字。许多应用协议和客户端都向用户提供有意义的引用，而对软件来说几乎毫无意义。例如，E-mail 客户端常常显示的是用户名而不是他们的 E-mail 地址。类似的，在 Web 浏览器中查看数据的用户常常查看文档的标题或链接内容而不是底层的 URL。

由于这种数据并不是安全地与服务器证书绑定在一起，于是它就形成了一种不充分的信任基础。偶尔，一种实现能够存取可受信的数据库，该数据库将这些名字映射成网络名。（然而，即便是安全 DNS 也不能完成这种工作）只有在这种情况下这些名字才是受信的。不然，就需要给用户提供查看经过验证的，未转换的身份的机会。只有那样才能提供充分的安全。

7.14 用户名/口令认证

我们在前面所阐述的对用户（和客户端）进行认证提出了一个难题。从安全的角度来看，理想的情况就是为我们希望认证的每个客户端部署证书。然而，这在许多情况下又会提出部署问题，因此重要的是要有其他的选择。

用户名/口令

最显而易见的（而且常用的）方式就是口令。正如我们先前所阐述的，在 SSL 上传输口令消除了口令的主要问题——被动的嗅探攻击——但是仍然处在其他一些形式更为难（但仍然是可能的）的攻击之下。

有三种主要的攻击类型值得考虑。首先要注意通道安全依赖于客户端形成到恰当的服务器的安全连接。这就是说客户端必须仔细检查服务器的证书。如果没有进行这种检查，这种通道就会受制于中间人攻击，而攻击者就能够恢复用户的口令。

第二种攻击就是口令猜测攻击。我们在第 5 章说过，用户经常选择低质量的口令（请参见 [Klein]）像用户名、名字、出生日期等内容。这样的口令很容易被猜到，因此攻击者可以简单地初始化到服务器的连接，并依次尝试每一个容易猜到的口令。这样的攻击并不是对任何给定的用户都能成功，但是如果用户的基数比较大，那么至少有一个用户选择劣质口令的

概率就会很高。存在多种能有效对付口令猜测的措施，其中包括前摄口令检查（proactive password checking）、限制尝试次数以及放慢每个口令检查的速度（以使猜测更加耗时），我们在此将不予以讨论。

口令的最后一个弱点就是它们是可传输的。当你向服务器证明自己身份的时候，需要向它提供自己的口令。如果你在多台机器上使用相同的口令（许多用户都这样），那么其中的某个服务器就能够对另一个服务器冒充成你，如你所见，尽管在 SSL 上而不是以明文形式来传递口令要好一些，但是它们仍然存在相当多的弱点。

7.15 SSL 客户端认证

另一种明显的解决方案就是基于证书的 SSL 客户端认证（SSL 没有任何客户端认证）。这种方法没有我们所描述的基于口令认证的弱点。特别的，攻击者没有办法使用客户端认证信息来冒充客户端——尽管存在攻击者诱骗客户端替攻击者制作认证请求的可能。

我们已经说过，基于证书的客户端认证的主要问题是操作上的。需要为每个能够认证的客户端颁发证书。此外，在大多数情况下仍然需要在服务器端维护一张用户列表，二者均是为了支持存取控制和用户删除。

证书的颁发

颁发证书的主要问题就是在颁发之前要对用户进行认证。由于一些众所周知的原因，如果你确实想将一个人的实际身份映射为证书，那么就不能联机颁发证书，而是要对他们进行实际审查。然而在大多数情况下，采用一种相对较弱的认证形式就够了。例如，可以要求用户提供一个信用卡号码。

一旦对用户进行了认证，就需要安排只有那个用户才能获得具有那个用户名的证书。证书颁发的一种直白的解决方案就是通过某些带外机制（out-of-band）颁发临时口令。例如，对公司来说，你可以只需打个电话把口令告诉用户，这样既能提供某种形式的认证又能提供口令安全。然后，在提出证书请求时一并提供已知的口令（自然要在 SSL 连接下加密）。注意，如果你原来就有某种用户名/口令类型的设施，那么就可以简单地使用那些口令来实现到证书的过渡。

存取控制

在某些情况下，只知道用户有证书是不够的，但是在大多数情况下，管理员想要能够为不同类别的用户提供不同级别的存取。有多种关于在证书中嵌入这种信息的方法描述，[Blaze1999] 和 [Ellison1999] 描述了两种这样的系统。然而在大多数情况下，维护描述每个证书持有者获准执行的动作的存取控制列表（ACL）较为简单一些。这种方案的主要优点就是可以只需修改 ACL 就能改变权限而不用给客户端办法任何新的证书。

这种方案的优点是它允许你拥有一种混合证书与口令的设施。为了实现这一点，你需要做的一切就是将通过证书认证的用户名映射为用户名/口令设施中的用户。

图 7.3 描述了一个简单的使用这种方案的系统。实际上有两种类型的列表：用户列表（其中还存储口令）和 ACL。一般来讲系统中每个对象都有一个 ACL，但是为了简单起见，我们只描述了一个。在这种基础设施中，Alice 可以通过口令或证书进行认证，Bob 只能通过

证书来认证，而 Charlie 只能通过口令来认证。ACL 条目指向用户列表中的条目。

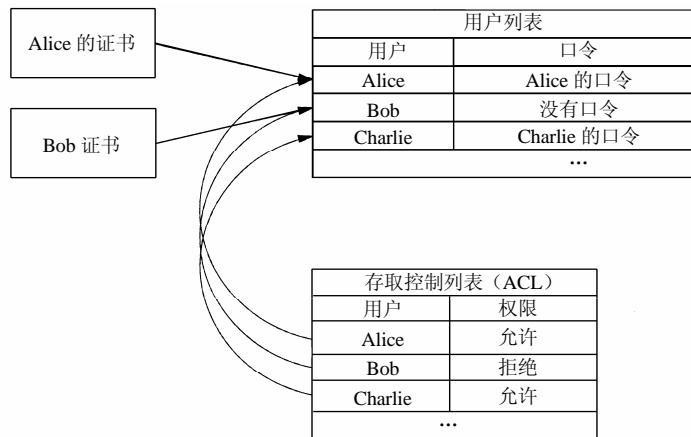


图 7.3 混合使用口令与证书的体系结构

注意，你可以允许这样的用户通过任一种方法进行认证或使用认证来标识用户不允许使用口令进行认证。这种方案的主要问题就是需要在两个地点对用户数据，即存取控制列表和 CA 进行维护。这样，要想创建一个用户你就必须颁发一个证书并修改用户列表。自然，这些数据库有可能不同步，从而产生维护问题。

撤消

在上面所描述的双料设施中，删除用户很简单。删除存取控制列表中的条目即可，别的什么也不做。用户可以进行认证，但由于他们的证书不会映射为任何用户，所以也就没有权限。ACL 条目就变成了悬挂引用，可以在以后某个方便的时候通过垃圾收集回收。如果你不使用 ACL（或类似的技术），那么另一种选择就是 CRL（证书撤消列表）。然而，正如我们在第 1 章所讨论的，CRL 也会引发大量的操作问题，SSL 中缺乏对 CRL 的直接支持更加重了这一问题。

主机对主机通信

存在一种特殊的情况，在这种情况下使用基于证书的认证带来的开销最小：这就是主机到主机的通道。设想你想要在两台机器之间建立一条安全隧道（tunnel）（用于虚拟专用网）。在这种情况下，你所要关心的全部问题就是另一端的身份。这里可以使用与完成服务器身份检查相同的过程来完成客户端认证。

此外，如果你所感兴趣的是对连接源的机器进行认证，那么就不必担心存取控制问题（因此也就不需要任何 ACL）。撤消仍然存在一定的问题，但是由于大多数实现都在涉及服务器的时候将其忽略，所以如果在涉及客户端的时候忽略它也不会造成太大的差别。

7.16 相互用户名/口令认证

到目前为止，我们一直假定服务器是通过证书来认证的。在第 7.1 节讲过，这对于使用

普通的方法来完成安全口令传输来说是绝对必要的。然而，SSL 不允许服务器为匿名的操作模式：即加密套件 `SSL_DH_anon`。在大多数情况下，这些加密套件完全受制于一种主动攻击。然而，可以组合使用口令和匿名加密套件来提供具有一定安全的协议。

中间人攻击

正如在第 1 章所讨论的，DH 的主要弱点就是主动的中间人攻击。在普通的 SSL 操作模式下，通过让服务器使用其静态签名密钥对其 DH 密钥进行签名就能阻止这种攻击。这样可以阻止攻击者实施这种攻击。然而，如果我们使用匿名模式，服务器的 DH 密钥就不会再被签名，于是攻击又成为可能的。剩下的唯一一种认证就是客户端与服务器之间共享口令。我们必须通过某种方式利用它来构造受信连接。

一种解决方案就是简单地使用口令直接对 `ServerKeyExchange` 消息进行签名。显然，这是一种 MAC 而不是数字签名，但是效果是一样的。这种方案的主要缺点就是要求对 SSL 进行修改，因为 MAC 并不是对 `ServerKeyExchange` 签名定义的方法之一。历史上有人抵制直接向 SSL 中增加这种口令机制。因此，就制定了在应用层解决这种问题的建议。

一种行不通的方案

考虑传统的 DH 中间人攻击，如图 7.4 所示。攻击者截获从客户端和服务器发来的 DH 公用密钥并将其替换为自己的公用密钥。这样就能够同时与客户端和服务器就共享密钥 (ZZ) 达成一致。注意，尽管攻击者能够同时跟客户端与服务器计算出成对的密钥，但是因为客户端与服务器私用密钥不同，所以那些 ZZ 并不相同。因为 SSL 使用 ZZ 作为 `pre_master_secret`，客户端与服务器将会计算出不同的 `pre_master_secret`。这看起来好像能防止中间人攻击。

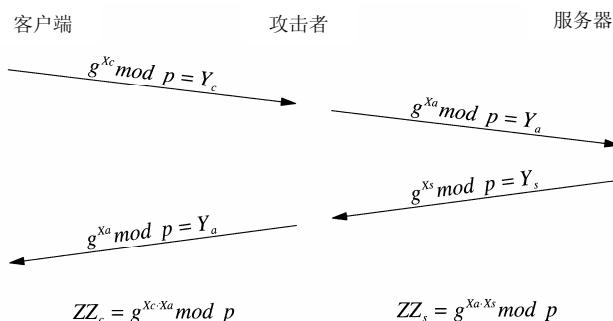


图 7.4 经典的 DH 中间人攻击

一种可行的方案就是使用口令计算出 `master_secret` 之上的 MAC（这等同于在 `PreMasterSecret` 之上计算一个）。然后双方交换 MAC，如果它们不一致，就会知道自己遭到攻击。攻击者不知道口令，因此也就不能用它计算出新的 MAC，即便他知道双方的 ZZ 也不行。

虽然这种方案可以阻止普通的中间人攻击，但是还有一些它无法制胜。事实上（通过挑选合适的伪造密钥）有可能迫使双方的 ZZ （同时由客户端和服务器计算得出）相同而且属于非常小的某个集合中的一个值。于是 ZZ 上的签名就会相同，而攻击者只需在客户端与服务器之间转发它们即可。

图 7.5 展示了这种类型的攻击。考虑情况 $p=2q+1$, 这里 q 也是素数而 R 是一个相对较小的数。在这种情况下, $g^q=-1$ 。如果攻击者将传输中的每个公用密钥 (X) 替换为 X^q , 我们就能够使结果的共享密钥 (ZZ) 成为 $g^{2X_s X_y \cdot q}$ 。注意, ZZ 在任一方都是一样的。这非常有用, 因为:

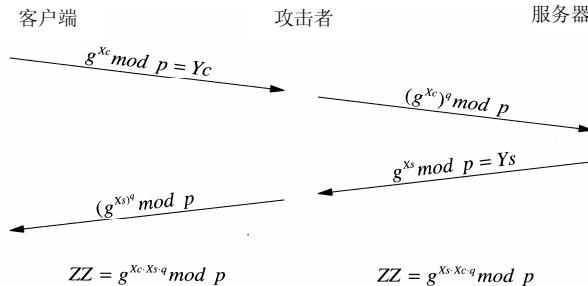


图 7.5 一种更为复杂的中间人攻击

$$g^{X_s \cdot X_y \cdot q} = (g^q)^{X_s \cdot X_y} = (-1)^{X_s \cdot X_y} = 1 \text{ or } -1$$

攻击者已经使 ZZ 成为 1 或 -1, 只需对它们两个进行尝试即可得到答案。这种攻击可以扩展至 $p \neq 2q + 1$ 的 DH 组。

● 一种可行的方案

我们已经看到了计算 PreMasterSecret 的 MAC 不行。虽然计算 ServerKeyExchange 的 MAC 能行, 但是它要求修改 SSL, 这一点就成问题。然而, 我们可以间接地通过计算 Finished 消息计算 DH 公用密钥的 MAC。因为 Finished 消息包含公用密钥的摘要信息, 继而有可以计算出公用密钥自身的 MAC。由于客户端与服务器见到的公用密钥不同, 因而可以检测出这种攻击。这是 STS 协议中所用到的过程的变体, 详细内容参见 [Deffie1992] 中的描述。这里所描述的特定变体是在 OpenSSL 邮件列表上由 Bodo Moeller 首先提出的。

图 7.6 描述了这一过程。在 SSL 连接建立起来后, 应用发送 Finished 消息的 MAC 作为其第一条应用数据。注意, 每一方都要计算它自己 Finished 消息的 MAC。这种方法允许不用证书就能构造一条安全通道, 这是通过共享口令来初始化的。此外, 客户端与服务器都知道另一方知道口令, 从而提供了相互认证。这在某种程度上要比使用证书的用户名/口令优越一些, 因为要强迫服务器证明它知道口令。注意, SSL 没有磋商这种技术的机制。客户端与服务器必须通过某种 SSL 以外的机制达成一致。

● 主动字典攻击

尽管上面介绍的方法很好, 但是它并没有对口令提供全面的保护。攻击者有可能实施针对连接的主动攻击并有可能的恢复口令。攻击者可以通过实施普通的中间人攻击来与客户端和服务器共享密码。然后再收集从客户端或服务器过来的计算了 MAC 的 Finished 消息并利用该信息来实施这种攻击。

这种攻击是我们在第 5 章所讨论的字典搜索攻击的简单变体。攻击者知道 Finished 消息的内容但不知道用来计算 MAC 的口令。因此他将尝试各种潜在的口令直至找到能够产生正确 MAC 的那个为止。

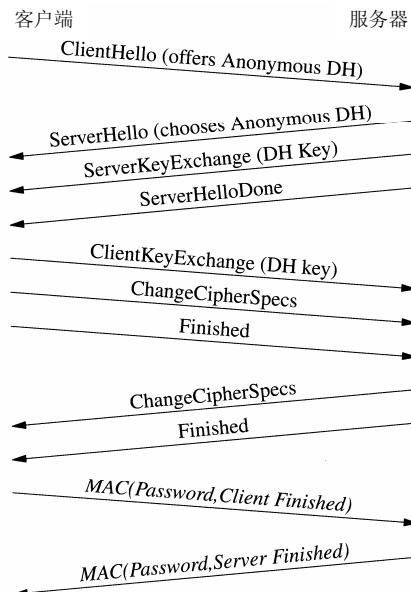


图 7.6 基于口令的相互认证

注意，这种攻击远比对正规的用户名/口令认证实施的口令猜测攻击优越，因为攻击大部分可以脱机进行，这种攻击惟一主动的部分就是收集 Finished 消息。与之对照，普通的口令猜测攻击要求服务器与每次企图都进行交互。

7.17 再握手

正如我们在第 4 章所讨论的，一旦完成了 SSL 握手之后就可以初始化新的 SSL 握手。如果你的服务器被配置成不同类型的协议请求，要求不同类型的安全就非常有用。因为 SSL 连接是在输出应用数据的第一个字节之前磋商好的，所以没有办法知道在第一次握手时给连接施加怎样的安全属性。考虑到这种新信息，有时在连接期间初始化第二次握手是一种有用的策略。

客户端认证

一种执行再磋商（renegotiation）的常见原因就是要求客户端认证。许多环境允许任何客户端存取某些资源，但要求认证后才能存取其他资源。要求对所有连接实施客户端认证就会将一些用户排除在外，因而不令人满意的。然而，对于其他连接来说则是需要。

一种方案就是使用某种独立端口方案的变种，对于要求客户端认证的连接使用不同的端口。然而，更雅致的方法就是在用户请求存取保护资源的时候允许客户端连接并请求客户端认证。许多 Web 服务器都使用这样的策略。

再握手的主要问题就是安排在正确的时间进行。SSL 规范没有提供关于何时进行再握手的实际指导，只提到了下列内容：

如果客户端不希望再次执行会话磋商，则可以忽略这条消息，或者如果愿意的话，客户

端可以使用 no_renegotiation 警示予以响应。由于握手消息特意拥有比应用数据更高的传输优先权，所以能在没收到客户端的几条记录之前就开始进行磋商。

在客户端认证的情况下，情况相当简单。服务器只是想确保在向客户端提供任何受限服务之前对其进行认证。因此，服务器应当发送一条 HelloRequest，然后等待客户端开始握手。任何在发送了 HelloRequest 和握手完成期间服务器收到的请求都应当排队等待，并在客户端认证通过后予以处理。自然，如果认证失败，连接就会被关闭或请求被拒绝。

● 加密套件升级

再磋商的另一种用途就是升级为强度更高的加密套件。就客户端认证而言，服务器或许发现客户端请求的是比普通通信数据更敏感的信息。正常情况下，服务器可以选择磋商使用 DES 或 RC4，并在传输这种敏感通信数据的时候升级为 3DES。再磋商可以用于这种目的，也可以使用它以新的临时 DH 密钥磋商一条新连接来获得 PFS（完美向前保密）。

在这种情况下，时间选择（timing）有可能要困难得多。如果你所关心的只是服务器传输数据的保密性，那么服务器就可以简单地发送 HelloRequest 并像客户端认证时那样进行等待。然而，如果客户端传输的数据必须是保密的，情况就会变得更为复杂，因为没有办法迫使客户端立刻进行再磋商。服务器所能做的唯一一件事就是在再磋商之前，当客户端开始发送敏感数据的时候终止连接。因为客户端最多只能传送 TCP 滑动窗口大小的内容，即便服务器在刚收到第一个敏感字节时就发送出错信息，可能已经有相当数量的敏感数据也可通过安全性较低的加密套件传送过来了。

● 替换密钥资料

实现者经常关心的一个 SSL 问题就是：他们是否应当通过再磋商来重新构造密钥资料，这通常是为了阻止针对长生存期的连接而进行的高强度攻击的。如果确实正在传输大量的数据，那么就必须不时地产生新密钥来防止 CBC 反转。从应用的角度来看，选择什么时间并不关键，因为要考虑这种因素的环境通常移动的都是多少兆字节的数据，在再磋商之前传送的 1K 左右的字节并不算什么。

● 客户端的行为

前面几节的讨论建议了几种服务器传送 HelloRequest 消息然后终止等待新连接的情况。结果，收到 HelloRequest 消息的客户端几乎立刻会尝试初始化一次新的握手。当然，如果它们已经发送了所有的通信数据而且服务器空闲的话，他们会立刻完成这项工作。没有这么做就有可能产生死锁。

7.18 二级通道

有些协议设计成在不止一条 TCP 连接上运行，FTP [Postel1985] 就提供了一个很好的例证。用于客户端初始化到服务器的连接并登录的连接称作控制连接（control connection），各种命令都是在这条连接上发出的。每次在客户端与服务器之间传输文件都是在第二条称作数据连接（data connection）的连接上进行的。

通常，数据连接由服务器创建。客户端先选择一个临时端口并通过控制连接将其告诉服

务器，随后服务器打开一条到这个端口的连接并使用从服务器到客户端的第二条连接传输数据。

这里的情况让人有些迷惑，因为是服务器在主动地打开连接：即与传统 TCP 客户端的行为一样。哪一端是 TLS 客户端呢？SSL 上的 FTP 从没有被标准化，但是已经公布了几种草案，它们在这个问题上有一定的作用（参见 [Ford-Hutchinson2000]）。客户端总是按照 TLS 客户端来行事，而不管是谁主动打开连接。

这种选择使情况稍稍简单一些，因为一些实现只有在创建会话的同一模式之中才进行会话恢复（即如果就第一个会话而言它们是 TLS 客户端，那么对于恢复的会话来说它们也必须是客户端）。更重要的是，如果你打算获得互操作性，就必须做出这样的选择。

7.19 关闭

许多协议都使用 TCP 连接关闭来指示在这个方向上不再有数据过来。从安全的角度来看，这样做存在问题，因为 TCP FIN 伪造起来相当容易。为了解决这个问题，SSL 提供了它自己的连接关闭机制：`close_notify` 警示。然而，这并不是惟一安全的方法。如果应用层协议有自己表示连接将要关闭的指示——并且是在 SSL 上传输的话——那么从安全的角度来看 `close_notify` 就是多余的。

除了协议要求的原因之外，所有协议都应当执行完整的 SSL 关闭握手。此外，如果你不这样做，就应当从技术上禁止恢复连接——尽管许多实现都违反了这条规则。也就是说，重要的是要在你的协议设计中指定没有完成关闭握手的后果。尤其是某些情况意味着安全风险而某些则不是。

不完整的关闭

许多协议都包含有指示一方知道不再会有数据从另一方过来的状态。考虑使用 FTP 传输数据的情况。数据通道是单向的，因此当发送端完成时就意味着根本不会再传输更多的数据。

在这样的情况下，许多实现都在发送 `close_notify` 后立即关闭连接。这会产生一种 RFC 2818 [Rescorla2000] 称之为不完整关闭（incomplete close）的情况。当连接的另一端收到 `close_notify` 时，它会像 TLS 规范第 7.2.1 节所要求的那样自己也发送一条 `close_notify` 警示予以响应。然而，由于连接已经被发送第一个 `close_notify` 的那一方关闭，因此响应将是一个 TCP RST 段。

我们这里所使用的术语“已被关闭”非常不严格。TCP 支持半关闭，即一方完成数据的发送但是仍然准备接收数据。在套接字 API 中，这是使用 `shutdown()` 调用来实现的，并将 `how` 设置为 1。大多数其他的 API 都提供了类似的功能。

然而，大多数应用都是简单地使用 `close()`，它同时表示系统不再从另一方接收任何数据。这就是为什么会产生 RST 段的原因：服务器在其已经关闭连接之后又接收到了通信数据。

图 7.7 描述了一种典型的情况。服务器知道连接已经结束，并产生一个 `close_notify`，然后再调用 `close()` 完全关闭连接（在此过程中产生一个 FIN）。客户端没有办法知道服务器不准备接收更多的数据（这顶多就是半关闭）于是发送 `close_notify` 作为响应，后面跟有它自

己的 FIN。因为服务器一端的套接字已经被关闭，服务器就使用 RST 段来响应。另一种收到 RST 的方法就是当客户端得到 close_notify 时更多的数据已经输出了。注意，RFC2246 要求实现一旦收到 close_notify 时就立刻停止传输数据，但是数据或许已经位于网络缓冲区或线路上了。

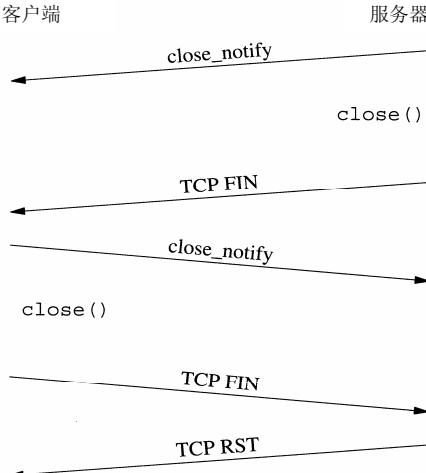


图 7.7 不完整关闭

不完整关闭并不一定就代表安全问题。为了明白这一点，考虑连接双方的情况。服务器知道将不会有更多的数据了（或者至少不再有它感兴趣的数据了），不然它就不会第一个发送 close_notify。如果不知什么原因存在更多的数据，这就代表应用协议而不是 SSL 连接存在问题。不管什么应用数据导致实现相信不再有更多的数据，对这些数据都应当像保护 close_notify 一样安全地加以保护。

现在从连接的客户端来考虑。客户端知道这样一个事实，即服务器不再有更多的数据发送过来，因为它相应的收到了一个 close_notify。于是相应的也产生一个 close_notify 并遭到拒绝。

过早关闭 (Premature Close)

第二种要考虑的关闭失败类型就是过早关闭。在过早关闭情况中，一方在没有发送 close_notify 的情况下关闭了 TCP 连接（即发送一个 FIN）。与不完整关闭不同的是，提前关闭极有可能代表安全威胁。close_notify 的存在就是为了阻止截断攻击。在没有见到 close_notify 的情况下，接收方无法区分发送方产生的 FIN 段和攻击者伪造的 FIN 段，因此很有可能是攻击者正在实施截断攻击。

应该对这两种情况加以区分。对于第一种情况，应用协议有自己的数据结束标志。如果在收到这些标志之前收到了 FIN，则接收方就假定发送端存在实现错误，估计是这个错误导致了过早发送数据结束标志或是省略了 close_notify 的发送。只要数据结束标志是在 SSL 上发送的，就无法伪造它们。

与之对照，如果协议不包含数据终结标志——或者包含但没有在 FIN 之前收到——那么系统就极有可能正处于截断攻击之下。不可能将这种情况与发送端的实现错误区分开来。在这种情况下，应用必须将其当作出错对待。任何情况下都不应该恢复以过早关闭结束的会话，



SSL 规范中明确禁止这种行为。

注意，没有恢复不正常关闭的连接实际上不会增添任何安全价值。过早关闭或许表示连接处于某种攻击之下，但是它并不表示有任何密钥资料已被攻破。

图 7.8 总结了关闭问题的正确行为。

各种情况	是否报错	是否允许恢复会话
不完整的关闭	No	Yes
在收到数据结束标志后发生的过早关闭	No	No
还未收到数据结束标志时就发生的过早关闭	Yes	No

图 7.8 处理关闭问题的正确的行为

7.20 总 结

本章讨论了使用 SSL 应用协议的设计。虽然采用简单的方案可能获得某种程度的安全，但是要想获得最高程度的安全则要注意几点重要而棘手的细节。

并不是所有的安全服务都需要通过 SSL 来提供。尽管 SSL 对于保护种类繁多的协议都非常有用，但是存在一些它无法充分发挥作用的服务。特别的，它无法为静态数据提供安全。有时将 SSL 与其他安全措施结合起来使用是非常有用的。

两种协议选择机制都非常有价值。独立端口更容易实现，但是磋商升级更雅致一些，而且拥有非常棒的对付被动攻击的安全性。

使用证书对服务器进行认证。我们讨论了一种特殊情况，其中可以使用口令来对服务器进行认证，但是总的来说最好还是坚持使用服务器证书。

树立客户端的安全期望。为了阻止主动攻击，客户端需要知道能从服务器得到何等类型的安全。也就是说引用必须包含所期望的服务器身份及某些希望服务器启用安全指示。

证书比口令要麻烦一些，但是强度更高。基于证书的客户端认证存在大量部署问题，但是比起即便是在 SSL 上传输口令的方法要安全得多。

定义再握手的语义。再握手是一种非常有价值的工具，但是有关再握手的时间选择缺乏明确规范，因而可能会导致竞争状况（race condition）的发生。设计者需要清楚地说明允许再握手的应用协议状态。

关闭问题看似简单。由于某些协议有自己的数据结束标志，从而使得 SSL 的 close_notify 没有存在的必要。而有些协议却没有，又使得 close_notify 对于阻止截断攻击来说至关重要。设计者需要仔细检查特定协议的关闭行为。

8

SSL 编程

8.1 介绍

在前一章，从协议角度对 SSL 进行了集中讲解。对于协议设计者和系统设计者而言，这是一种恰当的分析层次。然而，即便是最好的协议设计，如果无法实现的话，也是无用的。所以，本章从实现的角度集中讲解 SSL。

在第 7 章看到了一系列在 SSL 使用中经常碰到的问题，其中的大部分都可以用某种众所周知的设计模式加以处理。与之类似，开发 SSL 的实现者也需要完成许多同样的实现任务，不管他们使用什么样的应用协议。本章就是要讲述这些任务以及在应用过程中被证明是成功的编程套路（idiom）。

8.2 SSL 的实现

由于大多数语言都有 SSL 实现，所以大部分程序员都使用工具箱而不是亲自实现 SSL，假定你已有这种工具箱。尽管每种 SSL 工具箱都有其自身的应用编程接口（API），但针对每种语言而设计的接口通常都具有某种相同的风格，这样一来，即便你使用不同的工具箱，为一种工具箱编写的代码也是有启发的。

这里选择了两种免费的 SSL 工具箱，一种用 C 编写，一种用 Java 编写。对 C 语言来说，选择 OpenSSL，它源于 Eric Young 的 SSLeay 工具箱。对 Java 而言，我们选择 PureTLS，它由本书的作者编写。这两种工具箱都具有相当典型的针对各自语言的 API。此外，这两种工具箱均可以从因特网下载得到。具体描述请参见第 2 章。

8.3 范例程序

学习编程技巧的最好方法就是写程序。就这一点来说，将用 C 和 Java 编写两种程序——一个客户端和一个服务器——总共是四个程序。目的只是演示编程技巧，而不是提供使用 PureTLS 或 OpenSSL 编程的详细指南，因此当基于 OpenSSL 和 PureTLS 的代码在很大程

度上相同时，只编写一个版本。然而，由于 C 与 Java 的套路存在极大差别，所以我们会同时展示用两种语言编写的例子。

● 平台问题

C 程序是在 FreeBSD 上用 OpenSSL 0.9.4 开发的，它们同样能够与 OpenSSL 0.9.5 一起工作。我们只使用了最基本的 OpenSSL 函数，这样对新近和未来版本的 OpenSSL 来说也能工作。这些代码在 FreeBSD 和其他 UNIX 及类 UNIX 的操作系统中无须修改就能编译通过，而在 Windows 上要进行一些修改才能编译。

Java 程序是在 FreeBSD 上使用 JDK1.1.8 和 PureTLS 0.9b1 开发的。已经使用 JDK1.2.2 在 Windows NT 上进行了测试，对于未来的 PureTLS 版本也应当能够编译运行。

出于几种原因，特意选择强调 UNIX 而不是 Windows。首先，OpenSSL 的前身 SSLeay 是为 UNIX 设计的，因此 OpenSSL 编程接口更符合的是 UNIX 接口的风格，而不是 Windows 的。此外，即便是在非 UNIX 系统中，网络 API 通常也是基于套接字的，所以编写基于套接字的网络代码适用范围更为广泛。虽然 Windows 的网络接口 Winsock 也是基于套接字的，但却不跟它一样。

第二种原因就是 Microsoft 为 SSL 提供了自己的 API：SChannel，详细内容在 [Microsoft 2000] 中有所描述。SChannel 接口与其他 SSL 实现所提供的接口有相当大的差异，因此展示使用这种接口的例子程序对于非 Windows 程序员的作用就很有限。相反，使用 OpenSSL 编写的代码具有广泛的适用性，我们展示的 UNIX 代码具有充分的一般性，且不用怎么费劲就将其能移植到 Windows 上。

最后一个原因就是简单。在笔者看来，UNIX 程序的 API 更为干净，从而可以更简单地阐述 SSL 的问题，特定于操作的问题最少。

● 客户端程序

客户端是一种类型最为简单的交互式客户端。它先初始化一条到另一端服务器的 SSL 连接，然后再简单地在用户与服务器之间转发数据。换言之，它读取来自键盘的数据，并将数据通过 SSL 连接发送给服务器。类似的，它通过 SSL 连接读取从服务器过来的数据并将其输出到屏幕上。

虽然简单，但这种程序在调试时却很有用。许多因特网协议（SMTP、HTTP 和 IMAP 等）都是由简单的 ASCII 命令和响应构成的。因此可以使用一种简单的客户端直接与服务器联系来进行调试。UNIX 的 telnet 程序——就是打开一条简单的 TCP 连接——常常用于这种类型的服务器调试。

该客户端确实有一点特别之处：在许多系统上，数据都是以逐行的模式发送给服务器的。UNIX 终端驱动程序（以及其他一些程序）默认将键盘数据缓冲到用户按下回车键为止。当然，也可以改变这种行为，但是这么做对演示来说没有任何用处。

● 服务器程序

服务器程序就是一个简单的 echo 服务器。等待客户端的 TCP 连接，并在接受连接后初始化一条 SSL 连接。一旦连接建立，它就会从客户端读取数据，然后再将其按原样发送回去。最终当客户端关闭连接时，服务器就会关闭它这一端的连接。

显然，这并不是一种有用的服务——尽管标准 UNIX 系统确实带有这样一种以非安全方式完成此类工作的 echo 服务器——但是这种程序可以让我们演示网络服务器的实质性功能：等待连接，服务连接以及后续的清理工作。

运行中的程序

图 8.1 描述了客户端与服务器程序之间的交互。首先用户在键盘上键入数据。数据发送到服务器。服务器简单地将数据原封不动地返回给客户端。最后，客户端将数据打印到屏幕上。

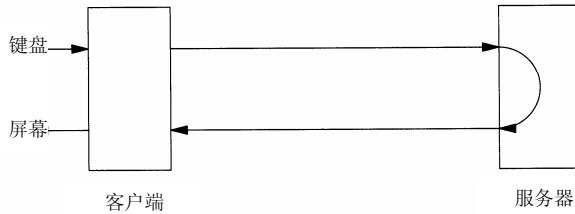


图 8.1 客户端/服务器交互

图 8.2 描述了 Java 客户端的运行演示（和 C 客户端的输出几乎完全一样）

[63] java SClient	执行程序
line 1	键入的内容
line 1	客户端程序的打印输出
line 2	键入的内容
line 2	客户端程序的打印输出
[64]	按 CTRL-D 结束程序

图 8.2 SClient 的实例运行演示

程序展示

本章只是通过展示源代码的片段来演示我们所讨论的技巧。附录 A 包含了所有例子程序的完整源代码列表，以及各种源文件之间的关系路径。可以通过检查列表中的头一行和最后一行来判断给定片段出自哪个源文件。

8.4 上下文环境的初始化

编写使用 SSL 程序要完成的第一项任务就是建立系统所使用的上下文环境。大多数实现都是通过使用一种由程序员初始化的上下文对象来完成这项工作的。然后再利用这个上下文对象来为每个新 SSL 连接创建一个新的连接对象，以及 SSL 握手和读写操作是通过这些连接对象来完成的。

这种方案有两个优点。第一，上下文对象允许对许多结构只进行一次初始化，从而节省了时间。在大多数应用中，每个 SSL 连接会使用相同的密钥资料、根列表等等。我们不针对每个连接重新加载这些资料，而是在程序启动时将其加载到上下文对象中。当我们想要创

建一个连接时，可以指示那个连接使用这个上下文对象。

第二，单一上下文对象允许多个 SSL 连接共享数据。最明显的例子就是 SSL 会话恢复。会话数据可以保存在上下文对象中，这样只需创建一个具有相同上下文对象的新连接就能自动恢复一个会话。

客户端初始化

总体上讲，客户端必须完成以下初始化任务。

加载 CA。几乎对每个 SSL 连接，服务器都是通过提供其证书来标识自己的身份。显然，为了对证书进行验证，客户端需要拥有一系列包含它所信任的对服务器证书进行签名的 CA 列表。该列表通常存储在磁盘某个位置的文件中并由客户端进行加载。

加载客户端密钥资料。如果客户端拥有用以完成客户端认证的密钥，那么就需要加载它们以及与之关联的证书。正如在第 5 章所讨论的，这些信息通常要么位于磁盘文件中，要么位于某种硬件设备中。对于其中的任何一种情况，程序都有可能需要提供口令才能存取这些密钥。

为随机数发生器抽取种子数据。由于每种应用环境都是不同的，使用工具箱的程序员常常很难产生良好的随机种子数据。例如，屏幕或鼠标常常是一种好的随机来源，但是某些环境既没有屏幕又没有鼠标，于是许多工具箱都把随机数种子留给应用程序员来产生。这通常意味着你要找到系统上的某种随机来源并将其输出交给工具箱。

设置允许的加密套件。大多数工具箱都具有某种准备在默认情况下磋商的加密套件集合。这或许并不适合你的应用，所以应当做好改变它们的准备。许多工具箱都允许你在设置上下文的时候或是针对每个连接这么做。

完成这些任务所需要的确切操作数量有赖于你正在使用的特定工具箱。例如，SPYRUS 的 TLSGold 仅用一个操作来完成从一个单一的文件中加载所有密钥资料，而 OpenSSL 则要求多项不同的加载操作。

服务器初始化

总体上讲，服务器初始化是客户端初始化的超集。尽管从技术上讲，不要求客户端认证的服务器有可能不包含其信任 CA 的列表，但是在初始化时检查自己的证书是一种良好的习惯，这样可以防止数据损坏和用户错误。因此对服务器来说，加载 CA 列表也是个好办法。另外还需要某些特定于服务器的初始化工作要做。

设置 DH 组。如果你计划使用临时 DH 模式，那么服务器就需要加载所使用的 DH 组。由于产生 DH 组速度太慢，所以一般都是在启动时完成这项工作的。你常常需要在磁盘上保存一个长期的 DH 组，然后在这一刻加载它，而不是每次启动时产生一个新组。

设置临时 RSA 密钥。类似的，如果你计划使用临时 RSA（如果你使用的是长 RSA 密钥并支持出口加密套件的话就极有可能要使用临时 RSA），或许会在此刻加载一个临时 RSA 密钥。许多工具箱都会在需要时自动产生一个，但是我们所说的这种方法避免了以后所带来的性能开销。

图 8.3 所示的 Demo.java 文件中包含了我们在使用 PureTLS 时要用到的所有通用初始化代码。此外，它还包含一些全局定义，用于表示像正在使用的端口号这样的内容。我们的客户端与服务器程序均为 Demo 的子类。

```
Demo.java
1 import COM.claymoresystems.sslg.*;
2 import COM.claymoresystems.ptls.*;

3 public class Demo {
4     public static final String host= "localhost";
5     public static final int port= 4433;
6     public static final String root= "root.pem";
7     public static final String random= "random.pem";

8     static SSLContext createSSLContext(String keyfile,String password){
9         SSLContext ctx=new SSLContext();

10    try {
11        ctx.loadRootCertificates(root);
12        ctx.loadEAYKeyFile(keyfile,password);
13        ctx.useRandomnessFile(random,password);
14    } catch (Exception e){
15        throw new InternalError(e.toString());
16    }

17    return ctx;
18 }
19 }
```

Demo.java

图 8.3 Demo 源代码

导入其他的软件包

1~2 PureTLS 的公共接口位于两个 Java 软件包中。COM.claymoresystems.sslg 包含接口，而 COM.claymoresystems.ptls 包含相应的实现。

设置常量

4~7 在实际程序中，诸如文件位置、主机名和端口号这样的变量一般位于配置文件中或是由用户输入。在这里直接将其硬性编写在代码中。

初始化上下文环境

9~18 PureTLS 使用 SSLContext 对象存储其上下文。一旦创建了上下文对象，就需要加载根证书和要使用的密钥文件。注意，因为该文件在磁盘上是加密的，所以 PureTLS 要求只有在提供了口令后才能加载其随机文件。这样就可以阻止能够读取随机文件的攻击者重新产生随机序列。

C 初始化

在图 8.4 中，OpenSSL 的 initialize_ctx() 与 PureTLS 的 Demo 类的功用是一样的。该函数要比 Demo 长得多，部分是因为对每个调用的错误检查：对 PureTLS 来说，Java 的例外机制替我们完成了这些工作。然而，OpenSSL 的设置多少更复杂些，从而使我们的例子代码也就较为复杂。



common.c

```
38     SSL_CTX *initialize_ctx(keyfile,password)
39         char *keyfile;
40         char *password;
41         {
42             SSL_METHOD *meth;
43             SSL_CTX *ctx;
44
45             if(!bio_err){
46                 /* Global system initialization*/
47                 SSL_library_init();
48                 SSL_load_error_strings();
49
50             /* An error write context */
51             bio_err=BIO_new_fp(stderr,BIO_NOCLOSE);
52         }
53
54             /* Set up a SIGPIPE handler */
55             signal(SIGPIPE,sigpipe_handle);
56
57             /* Create our context*/
58             meth=SSLv3_method();
59             ctx=SSL_CTX_new(meth);
60
61             /* Load our keys and certificates*/
62             if(!(SSL_CTX_use_certificate_file(ctx,keyfile,SSL_
63 FILETYPE_PEM)))
64                 berr_exit("Couldn't read certificate file");
65
66             pass=password;
67             SSL_CTX_set_default_passwd_cb(ctx,password_cb);
68             if(!(SSL_CTX_use_PrivateKey_file(ctx,keyfile,SSL_FILETYPE
69 _PEM)))
70                 berr_exit("Couldn't read key file");
71
72             /* Load the CAs we trust*/
73             if(!(SSL_CTX_load_verify_locations(ctx,CA_LIST, 0)))
74                 berr_exit("Couldn't read CA list");
75             SSL_CTX_set_verify_deph(ctx, 1);
76
77             /* Load randomness */
78             if(!(RAND_load_file(RANDOM,1024*1024)))
79                 berr_exit("Couldn't load randomness");
80
81             return ctx;
82         }
```

common.c

图 8.4 initialize_ctx():OpenSSL 的初始化

初始化全局数据

44~52 甚至在我们创建 SSL 上下文之前，就需要使用 `SSL_library_init()` 来初始化 OpenSSL 函数库本身。剩下的代码通过初始化错误列表来显示有助于记忆的错误字符串而不是错误代码。

初始化上下文

53~69 OpenSSL 将其 SSL 上下文变量称做 `SSL_CTX`（代表 SSL context）。初始化代码相当直接，但有几点值得评述。首先，注意 `SSL_CTX_new()` 参数 `meth` 的使用。这个参数有两方面的用途。第一，它允许我们设置该上下文准备磋商的 SSL 版本。

更重要的是，这种技巧允许我们只将需要的代码段链接进来。每种 `method` 都引用各种实现这种方法的函数和目标文件。在大多数系统中，链接器只将那些由 `main` 直接或间接引用的库函数链接进来（也就是说只有那些由 `main` 引用的函数或由 `main` 所引用的函数引用的函数，等等）。因此，如果你只指定 `SSLv3_method()` 的话，则 `SSLv2` 和 `TLS` 函数就不会被链接到最终的目标文件中，从而减少了二进制文件的尺寸。要想编写能够与任何 SSL 版本一起工作的客户端或服务器，则需要使用 `SSLv23_method()`。

同样请注意第 59~69 行中口令回调的使用。在 PureTLS 中，口令是直接通过 API 调用提供给函数库的。在 OpenSSL 0.9.4 中，函数库通过回调从用户请求口令。在这种情况下，我们将默认回调（它会通过终端提示用户输入口令）重新设置为简单返回我们硬性编写到代码中的口令的调用。在以后的 OpenSSL 版本中，可能会让程序员直接通过 API 调用将口令传递进来。

公共 C 代码

我们的演示程序还使用了其他几个公共例程和变量，可以在 `common.c` 中找到。这些函数的函数原型位于 `common.h` 中，由我们的其他源文件所包含，而 `common.h` 还包含许多我们在其他源文件中需要的包含文件。此外，它包含了许多表示端口号和根文件等硬性设定的常量。我们在图 8.5 中重新整理了 `common.h` 文件，以便于阅读。

common.h

```
1      #ifndef _common_h
2      #define _common_h

3      #include <stdio.h>
4      #include <stdlib.h>
5      #include <errno.h>
6      #include <sys/types.h>
7      #include <sys/socket.h>
8      #include <netinet/in.h>
9      #include <netinet/tcp.h>
10     #include <netdb.h>
11     #include <fcntl.h>
12     #include <signal.h>

13    #include <openssl/ssl.h>
```



```

14      #define CA_LIST "root.pem"
15      #define HOST "localhost"
16      #define PORT 4433
17      #define BUFSIZZ 1024

18      extern BIO *bio_err;
19      int berr_exit (char *string);
20      int err_exit(char *string);

21      SSL_CTX *initialize_ctx(char *keyfile, char *password);
22      void destroy_ctx(SSL_CTX *ctx);

23      #endif

```

common.h

图 8.5 common.h

服务器初始化

服务器可能还需要客户端所不需要的额外初始化步骤。正如先前所讨论的，需要完成临时密钥计算的服务器或许需要加载要用的 DH 组或 RSA 密钥。同样，在 OpenSSL 中，你需要显式地为服务器激活会话恢复，这在第 23~24 行实现。图 8.6 显示了 OpenSSL 所需要的附加服务器的初始化步骤。

```

19      /* Build our SSL context*/
20      ctx=initialize_ctx(KEYFILE,PASSWORD);
21      load_dh_params(ctx,DHFILE);
22      generate_eph_rsa_key(ctx);

23      SSL_CTX_set_session_id_context(ctx,(void*)&s_server_session_id_context,
24          sizeof s_server_session_id_context);

```

sserver.c

图 8.6 OpenSSL 服务器初始化

8.5 客户端连接

一旦客户端初始化了 SSL 上下文环境，就准备好连接到服务器了。SSL 工具箱 API 通常以普通的网络 API 为模型，因此 C API 与 Berkeley 的套接字 API 非常接近，也具有一系列与套接字调用类似的 SSL 调用。Java API 通常为 java.net.Socket 的子类——尽管 Sun 的 JSSE [JavaSoft1999] 采取了另一种截然不同的方式，在此不予说明。

Java 客户端连接

PureTLS 实现了一个从 java.net.Socket 导出的 SSLSocket 类。因此，可以非常容易地用与建立普通套接字连接类似的代码来创建一个简单 SSL 连接。需要费些周折的地方就是需

要将服务器证书中的标识名与我们想要连接的主机名进行核对，如图 8.7 所示。

```
Client.java
```

```
13     private static String dnToCommonName(DistinguishedName dN)
14         throws IOException {
15     Vector dn=(Vector)dN.getName();
16     Vector rdn=(Vector)dn.lastElement();
17
18     if(rdn.size()!=1)
19         throw new IOException
20             ("DN forms with multiple AVAs per RDN are unacceptable");
21
22     String[] ava=(String [])rdn.firstElement()
23
24     if(ava.length!=2)
25         throw new IOException("Bogus AVA array")
26
27     if(!ava[0].equals("CN"))
28         throw new IOException("CN must be most local AVA");
29
30     return ava[1];
31 }
32
33 public static SSLSocket connect(SSLContext ctx, String host, int port)
34     throws IOException {
35     // Connect to the remote host
36     SSLSocket s=new SSLSocket(ctx,host,port)
37
38     // Check the certificate chain
39     Vector certChain=s.getCertificateChain()
40
41     // Length check
42     if(certChain.size()>2)
43         throw new IOException("Certificate chain too long");
44
45     // Hostname check (using Common Name)
46     Certificate cert=(Certificate)certChain.lastElement();
47     String commonName=dnToCommonName(cert.getSubjectName());
48     if(!commonName.equals(host))
49         throw new IOException("Host name does not match commonName");
50
51     return s;
52 }
```

```
Client.java
```

图 8.7 PureTLS 的客户端连接

抽取 Common Name

13~26 为了检查服务器的证书，我们需要从 DN 中抽取 Common Name。这项工作由 dnToCommonName()负责完成。大家或许还记得标识名是一系列 RDN（相对标识名，relative

distinguished name) 序列。RDN 本身是包含属性值断言 (AVA) 的列表，AVA 就是一些属性 - 值对。在 PureTLS 中，DN 被表示为 Vector。它们的内容 RDN 也是 Vector，AVA 本身则表示为包含 Strings 的数组，其中第 0 个元素为属性，第 1 个元素为值。因此，这段代码就是从 DN 中的第一个 RDN (最近的) 中取的 CN AVA。

连接到服务器

30 连接本身并不复杂。在 Java 中，当你创建一个新的 Socket 时就执行连接工作。Socket 提供一个构造器，它会自动完成域名解析，所以甚至不用查找服务器的 IP 地址。

检查服务器证书

31~42 该函数的大部分功能都是将服务器的域名与服务器的证书进行核对。注意，由于使用域名来创建套接字，所以从理论上讲 PureTLS 可以完成这项检查，然而这样会降低程序员的灵活性。这里所描述的检查针对极少出现情况。注意，我们检查证书链的长度而不是依赖基本约束扩展。在理想状况下，如果基本约束普遍可用的话，PureTLS 就会替我们进行检查（实际上它不会这样）。

C 客户端连接

OpenSSL API 比起 PureTLS API 来要更为复杂难用。这部分反映了 Berkeley 套接字比 Java 网络 API 复杂得多的事实，此外还会看到 OpenSSL API 要比 PureTLS API 更加灵活。

不像 PureTLS，OpenSSL 要求在客户端与服务器之间创建一条 TCP 连接，然后再用 TCP 套接字创建 SSL 套接字，我们使用函数 `tcp_connect()` 来完成这项工作。该函数（如图 8.8 所示）对于任何曾经搞过 TCP 编程的人来说都不陌生。使用 `gethostbyname()` 解析服务器的 IP 地址，接着再使用 `socket()` 和 `connect()`。

client.c

```
3     int tcp_connect()
4     {
5         struct hostent *hp;
6         struct sockaddr_in addr;
7         int sock;
8
9         if(!!(hp=gethostbyname(HOST)))
10            berr_exit("Couldn't resolve host");
11         memset(&addr,0,sizeof(addr));
12         addr.sin_addr=*(struct in_addr*)hp->h_addr_list[0];
13         addr.sin_family=AF_INET;
14         addr.sin_port=htons(PORT);
15
16         if((sock=socket(AF_INET,SOCK_STREAM, IPPROTO_TCP))<0)
17             err_exit("Couldn't create socket");
18         if(connect(sock,(struct sockaddr *)&addr,sizeof(addr))<0)
19             err_exit("Couldn't connect socket");
20         return sock;
21     }
```

client.c

图 8.8 `tcp_connect()` 函数

把 TCP 连接与 SSL 连接分开既有好处也有坏处。主要缺点显而易见：这样使得编程更难。然而，像 `tcp_connect()` 这样的函数是相当常见的 C 语言术语，任何有能力的网络程序员大概都编写过许多遍。此外，如果你正在开发的是已有的网络应用，那么就已经存在这样的代码，而且集成这种风格的 SSL API 相应会容易一些。

然而，将 TCP 连接与 SSL 握手分开就意味着 SSL API 并不了解服务器可望拥有的 DNS 名或端口号。也就是说，这样的工具箱（甚至从原则上）不能将服务器的证书与 DNS 名进行核对，应用代码必须进行检查。以后还会看到，没有服务器域名，客户端就无法知道它企图恢复的会话，于是就得要求程序员来负责客户端的会话恢复。

要求程序员完成 TCP 连接确实有许多灵活性。因为磋商升级策略要求应用协议能够存取套接字，以便磋商过渡为 SSL，而如果想完成磋商升级就要使用这种类型的 API。此外，正如将在第 9 章见到的，某些代理形式要求应用协议在 SSL 握手前能够存取套接字。

● C 客户端 SSL 握手

图 8.9 中所示的 `sclient.c` 的 `main()` 函数负责连接到服务器并完成 SSL 握手工作。

```
----- sclient.c -----  
10      int main(argc,argv)  
11          int argc;  
12          char **argv;  
13          {  
14              SSL_CTX *ctx;  
15              SSL *ssl;  
16              BIO *sbio;  
17              int sock;  
  
18          /* Build our SSL context */  
19          ctx=initialize_ctx(KEYFILE,PASSWORD);  
  
20          /* Connect the TCP socket */  
21          sock=tcp_connect();  
  
22          /* Connect the SSL socket */  
23          ssl=SSL_new(ctx);  
24          sbio=BIO_new_socket(sock,BIO_NOCLOSE);  
25          SSL_set_bio(ssl,sbio,sbio);  
26          if(SSL_connect(ssl)<=0)  
27              berr_exit("SSL connect error");  
28          check_cert_chain(ssl,HOST);  
  
29          /* read and write */  
30          read_write(ssi,sock);  
  
31          destroy_ctx(ctx);  
32      }
```

----- sclient.c -----

图 8.9 `sclient.c` 的 `main()` 函数

● 创建 SSL_CTX

18~19 我们需要做的第一件事就是使用 `initialize_ctx` 构造 `SSL_CTX` 对象。客户端不需要进一步的初始化。

● 连接到服务器

20~21 使用 `tcp_connect()` 函数连接到服务器。

● SSL 握手

22~28 一旦创建了到服务器的 TCP 连接，就需要打开 SSL。OpenSSL 使用 `SSL` 对象来表示一条 SSL 连接。这一阶段最需要注意的问题是不要直接将 `SSL` 对象附加给套接字。应该先创建一个使用该套接字的 `BIO` 对象，然后再将 `SSL` 对象附加到 `BIO` 上。OpenSSL 使用 `BIO` 对象来提供一层 I/O 抽象。只要你的对象满足 `BIO` 接口，底层的 I/O 设备是什么就无关紧要了。

这层抽象提供了甚至比简单的将 TCP 连接与 SSL 握手分开还多的灵活性：对 OpenSSL 来说，完全有可能在根本就不是套接字的设备上实现 SSL 握手——只要你有合适的 `BIO` 对象就行。例如，OpenSSL 中的一个测试程序就是纯粹通过内存缓冲区将 SSL 客户端与服务器连接起来的。一种更为实际的用途就是支持无法通过套接字存取的协议。例如，你可以在串行线路上运行 SSL。

一旦建立了 SSL 连接，就可以通过熟悉的方式检查证书链：将 Common Name 与主机名进行核对。图 8.10 中所示的 `check_cert_chain()` 函数与图 8.7 中所示的证书检查类似。但是要注意，我们在这里并没有检查证书链的长度。相反，只要使用 `SSL_CTX_set_verify_depth()` 设置了最大链长度，OpenSSL 就会自动完成长度检查，就像在 `initialize_ctx()` 中所做的一样。

client.c

```

25     void check_cert_chain(ssl,host)
26         SSL *ssl;
27         char *host;
28     {
29         X509 *peer;
30         char peer_CN[256];

31         if(SSL_get_verify_result(ssl)!=X509_V_OK)
32             berr_exit("Certificate doesn't verify");

33         /*Check the cert chain. The chain length
34             is automatically checked by OpenSSL when we
35             set the verify depth in the ctx

36             All we need to do here is check that the CN
37             matches
38         */
39         /*Check the common name*/
40         peer=SSL_get_peer_certificate(ssl);
41         X509_NAME_get_text_by_NID(X509_get_subject_name(peer),
42             NID_commonName, peer_CN, 256);

```

```
43         if(strcasecmp(peer_CN,host))  
44             err_exit("Common name doesn't match host name");  
45     }  
----- client.c
```

图 8.10 check_cert_chain()函数

8.6 服务器接受请求

SSL 连接服务器一端的代码在概念上与客户端类似。和以前一样，工具箱 API 仿效编写它们所套路言的编程术语。因此，Java 代码使用 ServerSocket 的子类，而 C 代码使用 accept() 和 SSL_accept()。

Java 服务器接受请求

java.net 使用一个称作 ServerSocket 的类作为服务器接受连接。ServerSocket.accept() 返回一个 Socket。不管 Socket 是用 ServerSocket.accept() 还是 new Socket() 创建的，Socket 的行为完全相同。PureTLS 使用称作 SSLSocket 的类效仿这种方式。图 8.11 显示了使用 PureTLS 接受请求的代码。

```
----- Server.java  
23         SSLSocket listen=new SSLSocket(ctx,port);  
24         while(true){  
25             SSLSocket s=(SSLSocket)listen.accept();  
26             // Process this connection in a new thread  
27             ReadWrite rw=new ReadWrite(s,s.getInputStream(),  
28                 s.getOutputStream());  
29             rw.start();  
30         }  
----- Server.java
```

图 8.11 PureTLS 服务器接受请求循环

代码相当简单。在创建了 SSLSocket 之后，就进入无限循环，调用 accept()，然后为产生的套接字提供服务。accept() 的返回值是一个 Socket。实际上，即便服务器对 SSL 一无所知，也可能仅仅通过给它传递一个 SSLSocket 而不是 ServerSocket 来使该服务器具有 SSL 能力，之所以能行就是因为 PureTLS 类是 java.net 类的导出类。

然而，在实际应用中，大多数应用都想使用特定于 SSLSocket 的方法，所以它们将 accept() 的返回值通过强制类型转换 (cast) 转换为 SSLSocket 类型，正如我们在第 25 行所做的那样。举例来说，如果我们想要知道磋商加密套件的话，这种转换就是必须的。在第 25 行，因为 accept() 返回一个 Socket，所以将 accept() 的返回值转换为一个 SSLSocket。若想按照 Socket 的方法编写所有代码，就不要执行这种强制类型转换，而完全依照 Socket 的方法来操作。



C 服务器接受请求

与 C 客户端接受请求一样, OpenSSL 要求程序员执行 TCP accept。有了 TCP 套接字, 就需要创建 BIO 和 SSL 对象, 与我们进行连接时所做的工作完全一样, 不是调用 SSL_connect(), 而是调用 SSL_accept(), 如图 8.12 所示。

```

90         sock=tcp_listen();
91
92         while(1){
93             if((s=accept(sock,0,0))<0)
94                 err_exit("Problem accepting");
95
96             sbio=BIO_new_socket(s,BIO_NOCLOSE);
97             ssl=SSL_new(ctx);
98             SSL_set_bio(ssl,sbio,sbio);
99
100            echo(ssi,s);
}

```

图 8.12 OpenSSL 服务器接受请求循环

C 代码与 Java 代码之间的显著差别就是: sserver 一次只能处理一个客户端。相反, Server (Java 服务器) 能够同时处理任意数量的客户端。图 8.11 中的第 29 行暗示了这种差异, 用来自客户端与服务器之间 echo 数据的 ReadWrite 类是从 Thread 导出的。当调用 start() 的时候, 它就会产生一个新的线程来处理这个新客户端, 让原来的线程处理新客户端。

可以通过在 sserver 中调用 fork() 来获得类似的效果。然而, 要想得到在 Java 中得到的结果, 得做一些改动。将在第 8.8 节中更详细地讨论这个话题。

8.7 简单的 I/O 处理

一旦确立了 SSL 连接, 就想在其上传送数据。原则上讲, 可以通过使用与在 TCP 套接字上传递数据的类似风格做到这一点。在实际应用中, 使用 SSL 会引入一些需要处理的微妙问题。将从一个简单的任务着手, 然后逐渐建立起更为复杂的应用。

在这些程序中, SSL I/O 最简单的例子就是服务器端。图 8.13 重新绘制了图 8.1 中服务器一端的图示。如你所见, 服务器的工作非常简单: 从客户端读取数据并原封不动地发送回客户端。如果没有可发送给客户端的数据, 就什么也不做。图 8.14 描述了有关的代码。

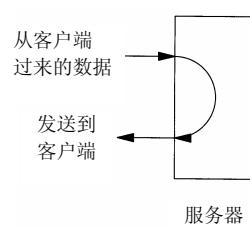


图 8.13 服务器 I/O

echo.c

```
3      void echo(ssl,s)
4          SSL *ssl;
5          int s;
6          {
7              char buf[BUFSIZZ];
8              int r,len,offset;
9
10             while(1){
11                 /* First read data */
12                 r=SSL_read(ssl,buf,BUFSIZZ);
13                 switch(SSL_get_error(ssl,r)){
14                     case SSL_ERROR_NONE:
15                         len=r;
16                         break;
17                     case SSL_ERROR_ZERO_RETURN:
18                         goto end;
19                     default:
20                         berr_exit("SSL read problem");
21                 }
22
23                 /* NOW keep writing until we've written everything*/
24                 offset=0;
25
26                 while(len){
27                     r=SSL_write(ssl,buf+offset,len);
28                     switch(SSL_get_error(ssl,r)){
29                         case SSL_ERROR_NONE:
30                             len-=r;
31                             offset+=r
32                             break;
33                         default:
34                             berr_exit ("SSL write problem");
35                     }
36                 }
37             end:
38                 SSL_shutdown(ssl)
39                 SSL_free(ssl);
40                 close(s);
41             }
```

echo.c

图 8.14 echo():一个简单的服务器 echo 例程

任何编写过 TCP 服务器的人都应当熟悉 echo()的实现。它在很大程度上就是将对 read() 和 write() 调用替换为 SSL_read() 和 SSL_write() 调用的标准 TCP 读/写循环。它很好地演示了 OpenSSL 如何使我们免于考虑 SSL 所引入的新的复杂性的；我们将在第 8.9 节考虑那些问题。

在变员 `ssl` 中将用来读取数据的 SSL 对象传递给 `echo()`。还在变员 `s` 中传递了 `ssl` 所依附的套接字。在完成大多数功能时我们都用不着 `s`，但是在最后关闭套接字连接的时候需要用到它。

● 读取数据

10~20 在循环中我们所做的第一件事就是从 SSL 连接的客户端读取一些数据。通过选择适当大小的缓冲区并调用 `SSL_read()` 可以很容易地完成这项工作。注意，这里缓冲区的尺寸并不重要。`SSL_read()` 的语义与 `read()` 的类似，都是返回可用的数据，即便可用的数据少于请求的数量。另一方面，如果没有可用的数据，那么调用 `read()` 就会阻塞，该调用会一直等到有数据可用并返回那些数据。

`BUFSIZZ` 的选择基本上是性能上的折中。这种折中与从普通套接字中简单读取数据时的情况颇为不同。对于那种情况，每次对 `read()` 的调用都要求一次到核心的上下文切换。因为上下文切换开销昂贵，所以程序员们都试图使用大缓冲区来减少这种开销。然而，当使用 SSL 的时候，调用 `read()` 的次数——也就是上下文切换的次数在很大程度上是由写入数据的记录数量而不是对 `SSL_read()` 的调用次数来决定的。

例如，如果客户端写一条 1000 字节的记录，而我们以 1 个字节的段长度调用 `SSL_read()`，那么第一次调用 `SSL_read()` 就会将记录读进来，而剩下的调用只是从 SSL 缓冲区中将其读取过来。因此在使用 SSL 而不是普通的套接字时，缓冲区尺寸的选择并不那么重要。

注意，当数据是以一系列小记录写入时，你或许希望能够用一条 `read()` 仅调用一次就将所有的记录都读取进来。OpenSSL 提供了一种激活这种行为的标志 `SSL_CTRL_SET_READ_AHEAD`。

注意第 12 行中针对 `SSL_get_error()` 返回值的 `switch` 的使用。普通套接字约定任何负数（通常为 -1）表示失败，然后通过检测 `errno` 来判断到底发生了什么情况。显然，`errno` 在这里无法工作，因为那样只是显示系统错误，而我们则想根据 SSL 的错误行事。使用 `errno` 还要求细心地编程才能保证是线程安全的（*thread safe*）。

OpenSSL 提供了 `SSL_get_error()` 调用来替代 `errno`。这个调用让我们检查返回值并弄清是否有错误发生以及是什么错误。在这种情况下，我们只关心三种情况中的一种：

返回值为正数。在这种情况下，我们已经读取了一些数据，将 `len` 值设置为那个长度以便以后写出。

返回值为零。这并不意味着没有可用的数据。如果是没有可用的数据的话，正如上面所讨论的，调用就会阻塞。它的意思是说套接字被关闭，将不会有任何可读取的数据，因此退出循环。

返回值是某个负数。这指示某种错误，但是需要使用 `SSL_get_error()` 来查明究竟发生了什么错误。由于这个程序没有错误处理，因此我们简单地默认使用 `berr_exit()`，它会打印出一条错误信息并退出。复杂一些的程序会有更多的 `switch` 分支来判定具体的错误类型并采取相应的动作。

注意，由于我们对所有错误的处理都是一样的，因此可以通过检查返回值是正、是负还是零来执行相应的动作。不错，另一种很好的术语就是只在值为负数时进入 `switch` 语句，直接就地处理非错误的情况。还要注意的是 `SSL_get_error()` 是特定于 OpenSSL 的功能。而别的工具箱有的选择直接使用返回值来指示错误值。

写数据

21~33 在抵达这一段代码的时候，肯定已经从客户端读取了一些数据。那些数据保存在 `buf` 中（从位置 0 开始）并且长度为 `len`。这里代码的工作就是将数据原封不动地返回给客户端（即，将其写到 `ssl`）。

乍一看来，好像不明白外围 `while` 循环的意思。毕竟当我们从网络上读取数据时没有这样的循环。道理很简单：我们想要将所有的数据都写到网络上。`SSL_write()` 的语义只保证会写出一些数据，而并不非得是所有的数据才可，返回值会告诉我们实际写出的数量。因此，如果所有的数据没有在一次调用中写出，那么就要循环操作直到所有的数据都被写出为止。

然而，我们想要输出我们还没有输出的数据，而不是重复输出已经输出的数据。为了实现这一点，我们需要维护一个指向已输出数据结尾的指针（也就是指向我们需要输出数据的起始位置）。因此我们保留变量 `offset`。在循环开始的时候，因为我们还没有输出任何数据，所以它被设置为 0。然而，一旦写出数据，我们就将其增加到已输出数据的长度并相应减少 `len`。最终，当 `len` 为 0 的时候（这时 `offset` 指向缓冲区的结尾）就退出循环。

为什么将所有数据写出的操作这么重要？考虑用户键入一些内容并等待服务器响应的情况。服务器开始输出数据但只成功写出了部分内容。没有第 23 行的循环，服务器就会跳转到第 9 行并企图从客户端读取数据。然而客户端还在等待服务器的输出，因此不会有任何可读的数据，服务器就会在 `SSL_read()` 中阻塞。由于客户端在等待服务器的输出，而服务器在等待客户端的输出，于是就发生了死锁——尽管这样，服务器还是可以输出数据。

当然，脱离这种状况的方法就是在等待客户端之前要完成自己所有数据的输出。尽管在这种情况下，从根本上讲必须先输出自己所有的数据，但是还存在除循环执行 `SSL_write()` 之外的其他方法可以安排。我们将在第 8.8 节中看到另一种解决该问题的方案。

在实际应用中，对 `SSL_write()` 的调用应当总是在将所有数据输出之后才返回。在默认情况下，OpenSSL 坚持这种行为。然而，如果设置了 `SSL_MODE_ENABLE_PARTIAL_WRITE` 标志，它就会在输出一条记录之后返回，即便输出缓冲区中有不止一条记录的空间也是如此。因此，如果设置这个标志位而且处理的是大于 1024 字节的分组，就很有可能看到部分输出。

与读取数据时一样，使用 `SSL_get_error()` 来处理返回值。然而，在这里只关心两种情况：返回值为正。在这种情况下，我们输出了一些数据。

返回值非正。在这种情况下，出现了某种错误。注意，我们应该不会见到返回值为 0 的情况，如果不能再输出数据的话就会阻塞。如果套接字被关闭，就应当看到一种实际的错误。

关闭连接

36~39 当我们到达循环底部的时候，就会知道已经读取了客户端要发送的所有数据，于是准备关闭套接字。我们调用 `SSL_shutdown()` 来发送 `close_notify`，然后再释放 SSL 结构并关闭套接字。

8.8 使用线程实现多路 I/O

在前一节见到的简单 I/O 模式对服务器来说工作得挺好，因为客户端是它惟一的输入。

然而，这种方案对客户端来说有所欠缺。考虑图 8.15，它展示了客户端的输入与输出。客户端必须同时监听两种输入源，即键盘和服务器。键盘与服务器的输入看上去似乎是异步的，也就是说他们可以以任何顺序出现。这就意味着一种简单的读写模式，就像服务器的那样，从根本上来讲是不够的。

甚至对我们这种简单的服务器也可以很容易看出这一点。考虑与我们在服务器中所使用的近似的 I/O 处理模式，图 8.16 描述了它的伪代码。

```

1   while(1){
2       read(keyboard,buffer);
3       write(server,buffer);
4
5       read(server,buffer);
6       write(screen,buffer);
7   }

```

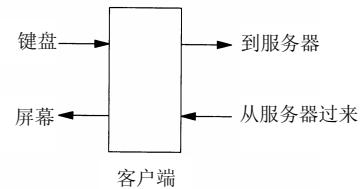


图 8.15 客户端 I/O

图 8.16 有问题的客户端 I/O 处理模式

现在设想我们有一个慢速服务器。用户在键盘上键入了一些内容，客户端将其发送给服务器。然而，服务器速度慢，花费了一些时间才响应。与此同时，用户键入了更多的数据——但是客户端不能读也不能写，因为它在第 4 行给卡住了，仍在等待从服务器读取数据。最后，服务器将数据输出，于是客户端开始往屏幕上输出内容。这使客户端解放出来读取客户端的数据并将其发送给服务器。显然这种行为不怎么地道。我们本可以向服务器写数据，但是由于服务器缓慢却写不成。另一方面，这不会直接影响用户的体验，因为我们的协议太简单了——真是没有美感。考虑一个由这种行为而导致严重问题的例子：远程登录。

考虑远程登录到一台机器并请求目录列表的情况。你的请求只有一行信息（UNIX 上的 ls），但是响应却是许多行。一般来讲，输出这些行需要不止一次写操作。因此也就很可能要花不止一次的读操作才能将其从服务器读取过来。然而，如果使用图 8.16 中的 I/O 模式，那么就会遇到问题。

我们从用户读取命令以及第一块服务器响应的内容，但是之后就遇到了死锁。客户端在第 2 行等待用户键入内容，但是用户却在等待剩余的目录列表，出现死锁了。大家发现这就是在第 8.7 节所见到的造成死锁的两方，而解决办法也是一样：我们需要从服务器将所有可用的数据都读取过来。

然而，我们先前所采用的这种解决方案行不通。大家应该还记得，当在没有数据的时候调用 SSL_read() 就会阻塞等待服务器的输出。我们不能那样做，因为这样会在服务器数据的结尾处被卡住，而我们实际需要的是从键盘读取数据。可以采取类似的论断来说明安排读写操作的顺序也不行。相反，我们需要某种可以确保从键盘或服务器读取数据的方法，而依据就是看哪一个准备好了。

解决这种问题的传统方法都是依赖于你所使用的语言。在 Java 这样的语言中，该方法就是使用线程。而在 C 这类的语言中，对线程的支持不存在或不标准，因而开发了更为复杂

的机制。本节将展示一个用 Java 编写的多线程解决方案，下一节再展示一个使用 select() 以 C 来编写的解决方案。

图 8.17 描述了 Java 客户端的 main() 方法。12~14 行就是我们以前见过的调用创建上下文环境与套接字的代码，因此不再多讲。重点是该函数剩余的部分。

```
Sclient.java
11     public static void main(String []args)
12         throws IOException {
13             SSLContextctx=createSSLContext(keyfile,password);
14             SSLSockets=connect(ctx,host,port);

15             // This thread reads from the console and writes to the server
16             ReadWrite c2s=new ReadWrite(s,System.in,s.getOutputStream());
17             c2s.start();

18             // This thread reads from the server and writes to the console
19             ReadWriteWithCancel s2c=new ReadWriteWithCancel(s,
20                     s.getInputStream(),System.out,c2s);
21             s2c.start();
22             s2c.setPriority(Thread.MAX_PRIORITY);
23             s2c.join();
24 }
```

```
Sclient.java
```

图 8.17 Java 客户端的 main()

启动线程

15~23 该客户端的总体结构简单。不再使用像图 8.16 中所描述的单读/写循环，而是使用两个那样的循环，且每个循环都在独立的线程中工作。每个循环都是在没有数据可读的情况下阻塞，然而却不会导致死锁，因为线程调度器会自动运行有数据可读或可写的线程。

我们通过创建 `ReadWrite` 对象实例来完成这项工作（`ReadWriteWithCancel` 是 `ReadWrite` 的子类，使用它来获得连接关闭期间的正确行为。将在第 8.10 节进行进一步的考察。就目前而言，姑且把 `ReadWriteWithCancel` 当作 `ReadWrite`）。`ReadWrite` 是 `Thread` 的子类，因此当调用 `start()` 的时候，它会自动启动一个新的线程。

第 22 行不是必需的。它绕过了 JDK1.1.8 线程调度器中的一个问题。没有这一行，`c2s` 线程就会使 `s2c` 线程饿死。这种技巧在 JDK1.2.X 中就不是必需的了。第 23 行的 `join()` 使该线程等待 `s2c` 线程完成。这种情况下这并不是绝对必需的，但是如果想要知道到服务器的连接何时终止的话就是必需的了。

图 8.18 描述了与 `ReadWrite` 有关的部分。我们省略了关闭代码，因为将会在第 8.10 节再来研究这个问题。构造器只是创建对象并将变元赋值给实例变量。注意，我们所提供的变元包括一个 `InputStream` 和一个 `OutputStream`。它们都是标准的 Java I/O 类，且是 `PureTLS` 的子类。因而 `PureTLS` 完全封装了这些代码，可以像使用标准套接字一样使用它！



ReadWrite.java

```
8  public class ReadWrite extends Thread{
9      protected SSLSocket s;
10     protected InputStream in;
11     protected OutputStream out;
12
13     /** Create a ReadWrite object
14
15         @param s the socket we're using
16         @param in the stream to read from
17         @param out the stream to write to
18
19     */
20     public ReadWrite(SSLSocket s,InputStream in,OutputStream out){
21         this.s=s;
22         this.in=in;
23         this.out=out;
24     }
25
26     /** Copy data from in to out.*/
27     public void run() {
28         byte[] buf=new byte[1024];
29         int read;
30
31         try {
32             while(true){
33                 // Check for thread termination
34                 if(isInterrupted())
35                     break;
36
37                 // Read data in
38                 read=in.read(buf);
39
40                 // Exit if there is no more data available
41                 if(read==-1)
42                     break;
43
44                 // Write the data out
45                 out.write(buf,0,read);
46             }
47
48             } catch (IOException e){
49                 // run() can't throw IOException
50                 throw new InternalError(e.toString());
51             }
52
53             // Finalize
54             onEOD();
55         }
```

ReadWrite.java

图 8.18 ReadWrite I/O 处理

读操作与写操作

22~45 由于 `ReadWrite` 是 `Thread` 的导出子类，因此它自动拥有一定的行为。特别的，它继承了一个称做 `start()` 的方法。`start()` 方法自动创建一个新线程并调用 `run()`，这也是代码为什么会在 `run()` 方法中完成 `ReadWrite` 工作的原因。

`run()` 方法本身很简单。我们只管从 `InputStream(in)` 读取内容并将其写到 `OutputStream(out)` 中。与 OpenSSL 不同，PureTLS 保证在调用 `write()` 时，只有在所有的数据都被发送到网络上后才会返回。这样一来，我们甚至不需要像图 8.8 中那样围绕 `write()` 的循环。显然，如果整个程序在 `write()` 上阻塞的话，那么就将是灾难性的。但因为我们是在自己的线程中操作，所以根本就不会出现那样的问题。

PureTLS 处理网络连接问题与 OpenSSL 不同。标准的 Java 方式就是让网络错误通过例外来投递，而 PureTLS 就遵循这种模式。因此，当在第 32~35 行读取数据的时候，我们不需要检查读错误。我们所要做的就是区分是成功读取还是遇到了文件结尾。这很容易，因为 `read()` 返回读取的数据长度或是在数据结束时返回 -1。如果我们读得了一些数据，就会到达第 37 行的写代码位置。否则就将退出循环并关闭连接。

以前说过，网络错误是通过抛出 `IOException` 来处理的。在这个程序中，简单地捕获这些错误，将它们当作致命错误对待，并抛出一个 `InternalError` 作为响应。在现实世界的程序中，需要试图从非致命错误中判别出致命错误并把它报告给用户。

8.9 使用 `select()` 实现多路 I/O

尽管多线程方案对 Java 来说能工作得很好，但是对 C 来说则远不能令人满意。线程支持并不是 C 标准的一部分，因此有赖于平台来提供选择的线程机制。Windows 提供了相当完善的线程支持，而该线程是那个平台上首选的多路编程方法。

不幸的是类似的情况在 UNIX 上要糟糕得多。UNIX 的线程支持没有标准化，而更遭的是 UNIX 上许多基于 C 的函数库都不是线程安全的。如果你想在 UNIX 上获得可移植的代码，就必须避免使用线程。这就是说如果你需要多路 I/O (multiplex I/O) 的话，就要使用 `select()`。尽管 `select()` 这种套路是 UNIX 中常用的技术套路 (idiom)，但是 `select()` 与 SSL 的交互远没有那么干净利落，而且需要理解一些我们目前只是有所暗示的微妙之处。

读操作

我们所面临的基本问题就是 SSL 是一种面向记录的协议。因此，即便我们只想从 SSL 连接读取一个字节，也仍需将包含那个字节的整个记录读取到内存中。不读取整个记录，SSL 实现就无法检查记录的 MAC，也就不能安全地将数据移交给程序员。不幸的是，这种行为与 `select()` 的交互是一团糟，如图 8.19 所示。

图 8.19 的左半部分描述了当机器收到一条记录，但这条记录仍然在网络缓冲区中等待的情形。箭头代表读指针，它被设置于缓冲区的开头。底部一行代表 SSL 实现解码的数据但是还没有被程序读取 (SSL 缓冲区)。这个缓

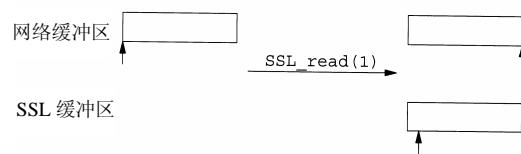


图 8.19 读操作与 SSL 的交互

冲区现在是空的，因此我们没有画那个矩形。如果程序在此刻调用 `select()`，它就会立即返回，指示调用 `read()` 将会成功。当程序员调用 `SSL_read()` 请求一个字节，这就到了图的右半部分所描述的情形。

我们以前说过，甚至给程序传送一个字节，SSL 实现都必须读取整条记录。一般来说，应用不知道记录的尺寸，因此它读取的尺寸与记录不相匹配。这样右上角的矩形描述了记录指针已经移动到了记录的末尾。我们已经读取了网络缓冲区中的所有数据。当实现对记录进行解密和验证的时候，它将数据放到 SSL 缓冲区中。然后再将程序所要的那个字节在 `SSL_read()` 中发送过去。我们在右下角描述 SSL 缓冲区。读指针指向缓冲区中的某个位置，指示存在一些数据可供读取，但是一些已经被读取过了。

考虑程序员在此刻调用 `select()` 发生的情况。`select()` 只关心网络缓冲区的内容，但那个缓冲区是空的。因此就 `select()` 而言，没有数据可读。根据传给它的具体参数不同，它要么返回称没有数据可读，要么等待有更多的网络数据变成可用的。对于任何一种情况，我们都不能读取 SSL 缓冲区中的数据。注意，如果又到达了一个记录，那么 `select()` 就会指示套接字处于可读状态，我们就有机会读取更多的数据。

因此，`select()` 有关是否有 SSL 数据可读的指示并不可靠。我们需要某种判断 SSL 缓冲区状态的办法。这种方法不能由操作系统提供，因为其无法存取 SSL 缓冲区。它必须由工具箱来提供。OpenSSL 就提供了这样一个函数。函数 `SSL_pending()` 告诉我们给定套接字的 SSL 缓冲区中是否有数据。图 8.20 描述了 `SSL_pending()` 的行为。

```
read_write.c
```

```

43             /* NOW check if there's data to read */
44             if(FD_ISSET(sock,&readfds)){
45                 do {
46                     r=SSL_read(ssl,s2c,BUFSIZZ);
47
48                     switch(SSL_get_error(ssl,r)){
49                         case SSL_ERROR_NONE:
50                             fwrite(s2c,l,r,stdout);
51                             break;
52                         case SSL_ERROR_ZERO_RETURN
53                             /* End of data */
54                             if(!shutdown_wait)
55                                 SSL_shutdown(ssl);
56                             goto end;
57                             break;
58                         case SSL_ERROR_WANT_READ:
59                             break;
60                         default:
61                             berr_exit("SSL read problem");
62                     }
63                 } while (SSL_pending(ssl));
64             }

```

```
read_write.c
```

图 8.20 使用 `SSL_pending` 读取数据

这段代码的逻辑相当直白。早先已经调用了 `select()`，并将变量 `readfds` 设置为可读的套接字。如果 SSL 套接字处于可读状态，则继续执行并试图填满缓冲区。一旦读取了一些数据，就将它们写到屏幕上。然后再使用 `SSL_pending()` 检查该记录是否比我们的缓冲区还长。如果是，就循环读取更多的数据。

注意，我们给 `switch` 语句增加了一个新的分支：对 `SSL_ERROR_WANT_READ` 的检查。在这里我们已经将套接字设置为非阻塞操作。我们在第 8.7 节讲过，如果在网络缓冲区为空时调用 `read()`，它就会阻塞到它不空为止。将套接字设置成非阻塞模式就会使其立刻返回，意思就是说它原本要阻塞的。

要想理解为什么要这样做，请考虑如果一条 SSL 记录分两块到达时的情形。当第一块到达时，`select()` 将会通知有数据可读。然而，需要读取整条记录才能返回任何数据，因此这次属于误报。企图读取所有那些数据就会阻塞，会导致我们以前曾努力避免的那种死锁。相反，将套接字设置成非阻塞的并捕获错误，OpenSSL 将这种错误翻译成 `SSL_ERROR_WANT_READ`。

写操作

当我们向网络中写数据的时候，就不得不再次面对在执行读操作时遇到的不一致性。同样，问题也是要么全读取过来，要么什么也不读，这是 SSL 传输的本质。为了简单起见，让我们来考虑网络缓冲区几乎要满，程序企图完成一次中等尺寸，如 1k，的写操作时的情况。图 8.21 描述了这种情况。



图 8.21 写操作与 SSL 的交互

同样，图的左半部分代表初始状态。程序要在缓冲区中写出 1K 的数据，写指针位于缓冲区的开头。SSL 缓冲区是空的，而网络缓冲区半满（阴影表示充满的区域）。写指针位于开头，故意模糊了 TCP 缓冲区与 TCP 窗口尺寸之间的差别，因为在这里它们并没有什么关联。一言以蔽之就是程序可以安全地写出 512 个字节而不会阻塞。

现在，程序以 1024 字节的分组调用 `SSL_write()`。工具箱没有办法知道它可以安全地写出多少字节，因此它只是将缓冲区格式化为单条记录，因此将程序缓冲区中的写指针移动到缓冲区的末尾。我们可以忽略 SSL 头信息和 MAC 对数据造成的微小扩展，只是像写到网络中的数据为 1024 字节那样行事。

现在，当工具箱调用 `write()` 时又会发生什么情况呢？它成功地写出了 512 字节，但是当它企图向记录尾部写数据时却得到了一条“would block”错误。结果，SSL 缓冲区中的写指针移动到了中间——指示那半数据已经写到了网络上。网络缓冲区的阴影指示它是全满的，网络写指针并没有移动。



现在需要担心两个问题：第一，工具箱如何将这种情况反映给用户。第二，当网络缓冲区中有空间可用时，用户如何安排 SSL 缓冲区清空。内核有可能会自动清空网络缓冲区，因此不必操心那样的安排。可以使用 `select()` 来查看何时网络缓冲区中有更多的空间可用，而我们就可以刷新 SSL 缓冲区。至少存在两种处理这种情况的策略。我们先讲解 OpenSSL 使用的方法，然后再描述另外一种策略。

● OpenSSL 写操作的处理

一旦 OpenSSL 从网络收到一个要阻塞的错误，它就停下来并将那个错误一直传给应用。注意，这并不是说它会扔掉 SSL 缓冲区中的数据。不可能这样做，因为或许有部分记录已经发送出去了。

为了清空这个缓冲区，程序员必须再次以它第一次调用的同一个缓冲区来调用 `SSL_write()`（也可允许对缓冲区进行扩展，但是起始位置必须相同）。OpenSSL 自动记住缓冲区写指针原来的位置，且只将写指针之后的数据写出。图 8.22 描述了执行这种动作的程序。

```

read_write.c
 66         /* Check for input on the console*/
 67         if(FD_ISSET(fileno(stdin),&readfds)){
 68             c2sl=read(fileno(stdin),c2s,BUFSIZZ);
 69             if(c2sl==0){
 70                 shutdown_wait=1;
 71                 if(!SSL_shutdown(ssl))
 72                     return;
 73             }
 74             c2s_offset=0;
 75         }

 76         /* If we've got data to write then try to write it*/
 77         if(c2sl && FD_ISSET(sock,&writefds)){
 78             r=SSL_write(ssl,c2s+c2s_offset,c2sl);

 79             switch(SSL_get_error(ssl,r)){
 80                 /* We wrote something*/
 81                 case SSL_ERROR_NONE:
 82                     c2sl-=r;
 83                     c2s_offset+=r;
 84                     break;

 85                 /* We would have blocked */
 86                 case SSL_ERROR_WANT_WRITE:
 87                     break;

 88                 /* Some other error */
 89                 default:
 90                     berr_exit("SSL write problem");
 91             }
 92         }

```

read_write.c

图 8.22 使用 OpenSSL 完成客户端到服务器的写操作

从主控读取数据

66~75 我们需要做的第一件事就是写出一些数据。因此查看主控是否可读，如果可读，就将那里的数据读到缓冲区 c2s 中（至多读 BUFSIZZ 字节），将长度信息放置在变量 c2sl 中。

向网络写数据

76~92 如果 c2sl 非 0 而且网络缓冲区是空的（或者至少部分是空的），那么就将数据写到网络上。与通常一样，我们以 c2s 调用 SSL_write()。跟以前一样，如果我们只写出了部分而不是所有的数据，就只需增加 c2s_offset，并减少 c2sl。

这里新举指就是检查 SSL_ERROR_WANT_WRITE。这种错误指示 SSL 缓冲区中有未刷新的数据，正如上面所描述的，需要以同样的缓冲区再次调用 SSL_write()，因此保持 c2sl 和 c2s_offset 不变。这样在下次调用 SSL_write()时它会自动处理相同的数据。

OpenSSL 实际提供了一个称做 SSL_MODE_ACCEPT_MOVING_WRITE_BUFFER 的标志，它允许你在要阻塞错误之后以不同的缓冲区来调用 SSL_write()。然而，这只是让你分配一个新的具有相同内容的缓冲区，SSL_write()仍然要移动到相同的写指针位置才寻找新数据。

另一种非阻塞策略

SPYRUS 的 TLSGold 工具箱（一种商业版的基于 C 的工具箱）实现了一种稍有不同的处理非阻塞 I/O 的策略。只要工具箱能够完整地对程序的写缓冲区进行编码，它返回成功信息——即便它不能彻底刷新 SSL 缓冲区。它惟一返回指示要阻塞的情况就是 SSL 缓冲区中已经有数据而且不能将它们全部刷新，这种行为与 UNIX 套接字的语义更为相符。

然而，由于调用者没有表示数据实际发往 SSL 缓冲区而不是直接写到网络上的指示，所以它必须采取正面的行动。TLSGold 提供了一种 SSL_IsMoreWriteData() 调用，它对写操作来说完成的功能与我们所见到的 SSL_pending() 针对读操作所完成的功能相同。因此，当使用 TLSGold 编程的时候，程序员需要检查 TSW_SSL_IsMoreWriteData() 并在网络可写的时候刷新缓冲区。通过使用长度为 0 缓冲区调用 TSW_SSL_Write() 刷新写缓冲区。

一套完整的基于 select() 的解决方案

图 8.23 描述了 read_write() 的完整代码。我们以前已经见过了其中的大部分的片段，不过值得从整体上研究一下这个函数。

read_write.c

```
8     void read_write(ssi,sock)
9         SSL *ssl;
10        {
11            int width;
12            int r,c2sl=0,c2s_offset=0;
13            fd_set readfds,writefds;
14            int shutdown_wait=0;
15            char C2s[BUFSIZZ],s2c[BUFSIZZ];
16            int ofcmode;
```



```
17     /*First we make the socket nonblocking*/
18     ofcmode=fcntl(sock,F_GETFL,0);
19     ofcmode|=O_NDELAY;
20     if(fcntl(sock,F_SETFL,ofcmode))
21         err_exit("Couldn't make socket nonblocking.");
22
23     width=sock+1;
24
25     while(1){
26         FD_ZERO(&readfds);
27         FD_ZERO(&writefds);
28
29         FD_SET(sock,&readfds);
30
31         /*If we've still got data to write then don't try to read*/
32         if(c2sl)
33             FD_SET(sock,&writefds);
34         else
35             FD_SET(fileno(stdin),&readfds);
36
37         r=select(width,&readfds,&writefds,0,0)
38         if(r==0)
39             continue;
40
41         /* Now check if there's data to read */
42         if(FD_ISSET(sock,&readfds)){
43             do {
44                 r=SSL_read(ssl,s2c,BUFSIZZ);
45
46                 switch(SSL_get_error(ssl,r)){
47                     case SSL_ERROR_NONE:
48                         fwrite(s2c,l,r,stdout);
49                         break;
50                     case SSL_ERROR_ZERO_RETURN:
51                         /* End of data */
52                         if(!shutdown_wait)
53                             SSL_shutdown(ssl);
54                         goto end;
55                         break;
56                     case SSL_ERROR_WANT_READ:
57                         break;
58                     default:
59                         berr_exit("SSL read problem");
60                 }
61             }
62         }
63     }
64 }
```

```
54         } while (SSL_ending(ssl));
55     }

56     /* Check for input on the console*/
57     if(FD_ISSET(fileno(stdin),&readfds)){
58         c2sl=read(fileno(stdin),c2s,BUFSIZZ);
59         if(c2sl==0){
60             shutdown_wait=1;
61             if(SSL_shutdown(ssl))
62                 return;
63         }
64         c2s_offset=0;
65     }

66     /* If we've got data to write then try to write it*/
67     if(c2sl && FD_ISSET(sock,&writefds)){
68         r=SSL_write(ssl,c2s+c2s_offset,c2sl);

69         switch(SSL_get_error(ssl,r)){
70             /* We wrote something*/
71             case SSL_ERROR_NONE:
72                 c2sl-=r;
73                 c2s_offset+=r;
74                 break;

75             /* We would have blocked */
76             case SSL_ERROR_WANT_WRITE:
77                 break;

78             /* Some other error */
79             default:
80                 bert_exit("SSL write problem");
81         }
82     }
83 }

84     Figure8.23(continue)

85     }
86 end:
87     SSL_free(ssl);
88     close(sock);
89     return;
90 }
```

read_write.c

图 8.23 read_write()函数

设置非阻塞模式

17~21 我们所要做的第一件事就是将套接字置为非阻塞的，这是标准的 UNIX 代码。

设置 select() 的位屏蔽

24~34 我们总是想随时做好从服务器读取数据的准备，但是却想对写往服务器的数据施加流动（flow control）控制。就这一点而言，我们总是查询服务器套接字的可读性，但是在查询键盘上的数据和查询可向服务器写出数据之间来回切换。如果 c2s1 非 0，那么我们就有数据写给服务器，然后告诉 select() 查询服务器套接字是否可写。不然，就得等待键盘可读。

注意，如果愿意话，我们可以一直等待键盘上有数据可用。然而，这样就意味着，当有数据要写给服务器的同时，键盘上可能会有数据过来。那么我们就得将这些新数据追加到 c2s 缓冲区的后面，从而使代码复杂化。我们确实想只在有数据可写的时候才查询服务器套接字是否可写，而在大多数时间都没有数据可读或可写，因此服务器套接字在大多数时间都是可写的。这样，当我们监听可写的服务器套接字时就会不断地进出 select() 循环，这可不是我们想要的行为。相反，我们想让程序一直睡到有事情可做为止。

前面已经看到了该函数的其他部分，但有一处低效的地方值得指出：一旦我们从键盘读取了数据，甚至在将其写给服务器之前就得再次进入 select()。在大多数情况下，服务器是可写的，因此立即执行写操作能稍微快一些。然而，这样又会使代码复杂化且只算是一种微不足道的优化。

8.10 关闭

我们已经阅读了读写数据的代码，现在准备研究一下关闭操作。我们主要需关心两个方面。第一，需要确保程序不但发送而且还接收了一个 close_notify——如果没有收到 close_notify 的话要抛出一个错误。第二，需要确保能正确处理非完整关闭。首先，我们查看一下基于 OpenSSL 的关闭代码，如图 8.23 所示。

OpenSSL 关闭

我们已经见过了服务器上的关闭行为，所以现在先查看一下客户端上的关闭行为。在两种情况下客户端需要关闭连接。第一，客户端通过键盘收到了一个数据结束标志。第二，客户端从服务器收到了一个 close_notify（如果客户端收到了一个 TCP FIN 而没有收到 close_notify，它就会在 SSL_read() 的末尾因出错而退出）。服务器能够终止的惟一方式就是在其收到用户数据结束标志。

用户数据结束

59~63 如果我们从键盘收到了一个数据结束标志，那么就需要初始化关闭过程。OpenSSL 只有一个完成此项工作的函数，SSL_shutdown()。如果它能够完成关闭过程就返回 1，否则就返回 0。惟一有可能返回 0 的情况就是它还没有收到 close_notify。因此，如果我们得到了一个返回值 0，那么就必须等待一个 close_notify。我们将 shutdown_wait 设置为 1 并返回 select() 循环。

等待关闭

43~47 这一段代码同时处理收到服务器冒然提出的 (unsolicited) close_notify 的情况，以及对我们的 close_notify 延迟作出响应的情况。如果服务器套接字是可读的，但得到的返回值为 0，则表示已经收到了一个 close_notify（因为纯粹的 TCP FIN 会产生错误）。如果这是一个冒然提出的 close_notify，那么 shutdown_wait 将会是 0，而且我们需要使用 SSL_shutdown() 来发送 close_notify。另一方面，如果我们已经发送了 close_notify，那么就可以简单的跳出循环。

在遇到未请求 close_notify 时，有可能另一方已经完全关闭了连接。在这种情况下，我们的 close_notify 就会触发一个 TCP RST，它会通过发出 SIGPIPE 信号在发送端表现出来。由于对 SIGPIPE 的默认响应是关闭进程，所以我们必须安排忽略它。common.c 中的第 52 行对 signal() 的调用（如图 8.4）就是通过建立起一个哑处理句柄来负责处理这件事情。另一种方案就是设置忽略该信号，并潜在地进行一些处理——如关闭连接。

清除对象

84~88 清除工作包括释放 SSL 对象和关闭后来的套接字。

PureTLS 关闭

与 OpenSSL 不同，PureTLS 提供了三种处理 SSL 关闭过程的方法。close 的行为与 SSL_shutdown() 非常像，但是 sendClose() 和 recvClose() 能够更精确地控制所发生情况。范例代码使用的是这些调用，而不是 close()。

在第 8.8 节的讨论中省略的 ReadWrite.onEOD 实现如图 8.24 所述。

```
----- Readwrite
56     protected void onEOD(){
57         try {
58             s.sendClose();
59         } catch (IOException e){
60             ; // Ignore broken pipe
61         }
62     }
----- Readwrite
```

图 8.24 ReadWrite.onEOD()

onEOD() 是在 ReadWrite.run() 结尾处调用的。通常这发生在线程从键盘收到了数据结束标志的时候。在那种情况下，线程使用 sendClose() 发送一个 close_notify 并退出。然而，还可以在从服务器收到冒然提出的 close_notify 时调用 ReadWrite.onEOD()。这一过程是通过类 ReadWriteWithCancel 初始化的，如图 8.25 所示。

```
----- ReadWriteWithCancel.java
11  public class ReadWriteWithCancel extends ReadWrite {
12      protected ReadWrite cancel;
13
14      public ReadWriteWithCancel(SSLocket s, InputStream in, OutputStream out,
15          ReadWrite cancel){
16          super(s, in, out);
```

```

16         this.cancel=cancel;
17     }

18     protected void onEOD (){
19         if(cancel.isAlive ()) {
20             cancel.interrupt ();
21             try {
22                 cancel.join();
23             } catch (InterruptedException e) {
24                 throw new InternalError(e.toString());
25             }
26         }
27     }

```

ReadWriteWithCancel.java

图 8.25 ReadWriteWithCancel 类

ReadWriteWithCancel 与 ReadWrite 不同，它替换了 onEOD()。第 8.8 节中讲到的监听服务器线程就是 ReadWriteWithCancel 的实例。当那个线程看到服务器发来的数据结束标志时，它就知道自己收到了一个 close_notify（纯粹的 TCP FIN 会导致抛出例外 IOException）。它通过触发线程向服务器写数据来发送自己的 close_notify 对其进行响应，这样就确保了在同一时刻不会有两个线程向同一个套接字写数据的情形，当 ReadWriteWithCancel 试图自己调用 sendClose() 时就会发生这种情况。

信号报告是通过调用 cancel.interrupt() 完成的。ReadWrite 使用 interrupted() 调用测试中断。如果 interrupted() 返回 true，那么读/写循环就会退出而 onEOD() 就会被调用。与此同时，服务器到客户端的线程等待客户端到服务器的线程退出，然后自己再退出。

如果另一方发送了一个冒然提出的（unsolicited）close_notify，我们就需要注意在发送我们的 close_notify 时收到 RST 的情况，这也是将 sendClose() 包裹在 try/catch 中的原因。RST 会触发一个 IOException，我们只是简单地将其忽略。

8.11 会话恢复

本章所要讲述的最后一个主题是会话恢复（Session Resumption）。正如在第 4.10 节所讨论的，当客户端拥有服务器的域名时，就可以进行自动会话恢复，PureTLS 做到了这一点。图 8.26 描述了一个在客户端完成会话恢复的 PureTLS 范例类。由于我们一直在使用的 PureTLS 服务器自动进行循环，所以演示会话恢复无须特殊的支持。

Java 会话恢复

Rclient.java

```

6   public class RClient extends Client {
7       public static void main(String []args)
8           throws IOException {
9               SSLContext ctx=createSSLContext(keyfile,password);
10              /* Connect and close*/

```



```
11         SSLSocket s=connect(ctx,host,port);
12         s.close();

13         /* Now reconnect: resumption happens automatically*/
14         s=connect(ctx,host,port);
15         s.close();
16     }
17 }
```

—— Rclient.java

图 8.26 使用 PureTLS 恢复的会话

注意，我们在这里没做什么特殊的工作就激活了会话恢复，而以前所使用的 SClient 类却无法做到这一点。

● C 会话恢复

OpenSSL 需要一些附加的支持才能完成会话恢复。前面已经见到了服务器上所要求的初始化代码而且不需要增加任何新的代码。然而，我们确实要在客户端做一些附加的工作。图 8.27 描述了一个执行会话恢复的客户端。这里关键的代码就是第 27 和 34 行。使用 SSL_get_session()从第一个 SSL 对象获得会话数据，然后使用 SSL_set_session()将数据赋给第二个 SSL 对象。从那以后，我们就能够简单地再次调用 SSL_connect()了。

```
6      int main(argc,argv)
7      int argc;
8      char **argv;
9      {
10         SSL_CTX *ctx;
11         SSL *ssl;
12         BIO *sbio;
13         SSL_SESSION *sess;
14         int sock;

15         /* Build our SSL context*/
16         ctx=initialize_ctx(KEYFILE,PASSWORD);

17         /* Connect the TCP socket*/
18         sock=tcp_connect();

19         /* Connect the SSL socket */
20         ssl=SSL_new(ctx);
21         sbio=BIO_new_socket(sock,BIO_NOCLOSE);
22         SSL_set_bio{ssl,sbio,sbio};
23         if(SSL_connect(ssl)<=0)
24             berr_exit("SSL connect error (first connect)");
25         check_cert_chain(ssl,HOST);

26         /* Now hang up and reconnect */
27         sess=SSL_get_session(ssl); /*Collect the session*/
```



```
28         SSL_shutdown(ssl);
29         close(sock);

30         sock=tcp_connect();
31         ssl=SSL_new(ctx);
32         sbio=BIO_new_socket(sock,BIO_NOCLOSE);
33         SSL_set_bio(ssl,sbio,sbio);
34         SSL_set_session(ssl, sess); /*And resume it*/
35         if(SSL_connect(ssl)<=0)
36             berr_exit("SSL connect error (second connect)");
37         check_cert_chain(ssl,HOST);

38         /*Now close everything down again*/
39         SSL_shutdown(ssl);
40         close(sock);
41         destroy_ctx(ctx);
42     }
```

rclient.c

图 8.27 使用 OpenSSL 恢复会话

当然这种代码并不适合生产性的应用。你只能在创建会话的服务器上恢复会话，而且这种代码不区别对待各种服务器——因为服务器名是硬性编码的。在现实应用中，你会需要某种将主机名/端口对映射为 SESSION 对象的查找表。

最后还要注意，即便没有将主机名和端口号传递给 SSL 工具箱，也仍可能自动恢复会话。工具箱可以通过在套接字上调用 `get_peername()` 来得到远端主机名和端口号。然而，由于 OpenSSL 的 BIO 抽象将 SSL 代码与套接字隔离开来，因此这对于 OpenSSL 来说是不可能的。TLSGold 通过使 `getpeername()` 函数作为其 I/O 抽象的一部分解决了这种问题。然而 OpenSSL 并没有这么做，所以应用需要显式地处理恢复。

8.12 缺少什么？

本章已经讲了这么多，仍然省略了 SSL 编程的某些方面。本节将对这些内容进行概括性地描述。

更好的证书检查

正如在第 7 章所讨论的，一种更为复杂的将服务器证书与服务器主机名加以核对的方案就是使用 X.509 `subjectAltName` 扩展。为了进行这种检查，你需要从证书中抽取出这一扩展，然后再根据主机名进行检查。这并不很难但却是复杂的，而且与 SSL 工具箱的证书处理 API 关系紧密，这也是我们在这里省略它的原因。此外，最好能够将主机名与证书中的通配名进行核查，我们在这里也没有讲如何去做。

/dev/random

越来越多的 UNIX 系统都支持一种称做 `/dev/random` 的设备。从本质上讲，`/dev/random` 就像连续的随机字节源。你只管打开设备并从中读取即可。`/dev/random` 通过检查各种系统

变量、定时器和中断并将其混合成伪随机数发生器，来产生高质量的随机输出来工作的。在有/dev/random 的系统上，这是一种有用的随机源，它可以代替或补充 OpenSSL 或 PureTLS 的随机文件。Windows 2000 也有一种类似的设施。出于可移植性的原因，我们没有描述如何存取这些设施，但是它们确实值得一提。

多进程操作

这些例子中讲得最多的就是如何让服务器支持不止一个客户端的方法。我们曾经提到过，基于 PureTLS 的服务器启动新的线程来为客户端实际提供服务。但是 SSL accept 则是在主线程中完成的，将它用在高流量的服务器上无法令人满意的。基于 OpenSSL 的服务器同一时刻根本不能处理多于一个的客户端。Web 应用中经常要求干净利落地处理这种情况——多条低开销的并发连接在这里非常重要——将在下一章讨论 TLS 上的 HTTP 时重新拾起这个话题。

更好的错误处理

注意，上述的这些应用只是通过出错退出来处理错误。但是在现实应用中则需要能够识别错误的类型或将其汇报给用户或一些记账日志，而不仅仅是退出而已。

8.13 总 结

本章讨论了编写使用 SSL 应用的编程技术。我们提供了两组例子程序，一组使用 OpenSSL 用 C 写成，另一种使用 PureTLS 用 Java 写成。这些例子演示了许多使用 SSL 的应用中的常见技术套路。

编程语言举足轻重。正如我们所见到的，PureTLS 和 OpenSSL 模仿编写它们的语言的网络 API。此外，最适合的 I/O 模式有赖于你所使用的编程语言的能力。

上下文对象允许一次性初始化。许多初始化过程如密钥文件的加载和随机数生成都是耗时的，为了避免对每个 SSL 连接都重复完成这种工作，我们初始化一个上下文对象并使用那个对象来创建新的连接。

单源 I/O 很简单。对于简单的服务器情况，我们常常只需从 SSL 连接读取数据并将其直接输出——这几乎与使用套接字编程完全一样。

多路 I/O 是复杂的。当我们需要在多种输入源之间执行多路传输时，要么使用线程，要么使用 select() 来为所有的 I/O 提供服务而避免出现“饥饿现象”。这要求仔细认真地处理记录分帧（framing）以及关闭行为。

SSL 记录分帧会产生问题。每次都必须从网络中读取或写出 SSL 数据，且一次只能对一条记录进行操作。因为应用经常想读取或写出的数据尺寸与记录的不同，所以 select() 常常给出错误的有关可用数据的回答。在这种情况下，程序员必须使用工具箱特定的机制来判断 SSL 读写缓冲区的状态。

必须仔细地处理关闭。重要的是能够处理本地初始化的和冒然提出的关闭。对于本地初始化的关闭来说，我们绝不能简单地关闭连接，而是要等到收到 close_notify 为止。对于冒然提出的关闭，程序员必须发送一条 close_notify，但是应准备好在对方以 RST 应答时进行恢复。

会话恢复 API 的差别迥异。PureTLS 会在合适的地方自动恢复会话，OpenSSL 要求用户的介入，而其他一些基于 C 的工具箱也可以自动完成恢复。

9

SSL 上的 HTTP

9.1 介绍

HTTP（超文本传输协议，Hypertext Transfer Protocol）提供了一种非常适宜使用 SSL 来保护其安全的协议的例子。它是第一个使用 SSL 的协议，而且到目前为止仍然是最重要的使用 SSL 来保护其安全的协议。几乎可以在每一种 Web 浏览器与服务器中见到的标准处理方式都是使用独立端口策略，而 IETF 已经标准化了一种用于在 HTTP 中升级至 TLS 的磋商升级（upward negotiation）技术。

本章从讨论高层次上的 Web 安全问题开始，其中包括对基本 Web 技术的介绍。然后讨论传统处理 HTTP 与 SSL 的方案（HTTPS，在 RFC2818 中描述）以及它与这些 Web 技术进行怎样的交互。接着再讨论更新的 HTTP 升级（HTTP Upgrade）技术（在 RFC2817 中描述），以及与 HTTPS 的比较。最后讨论一些在使用 SSL 与 HTTP 时经常碰到的编程问题。

9.2 保护 Web 的安全

Web 安全应用的原型就是向 Web 服务器提交信用卡信息。用户使用浏览器浏览 Web 站点并在虚拟购物车中存放选购的物品。一般来说，完成这一步不涉及任何安全操作，因为我们假定这种信息不是敏感信息——尽管当你购买一些令人尴尬的物品时这些信息或许是敏感的。

当顾客准备结账时，安全需求应运而生。顾客需要向服务器提供自己的信用卡号码才能进行结算。由于任何具有信用卡号码的人都可以充当该用户并将账记在该用户名下，显然我们需要为信用卡信息提供保密。此外，用户还需要确信他将信用卡信息提交给了恰当的服务器而不是偷盗其信用卡的假服务器。

在结账时，服务器将用户转移到一个安全页面，用户在那里可以键入其信用卡号码和有效期。在安全提交该信息之后，用户需要从服务器得到定单被接收的确认。这种信息同样要安全地传递给顾客，以阻止攻击者悄无声息地阻止定单的提交。

对于这种应用，我们的安全需求非常简单。我们需要提供客户端与服务器之间数据传递

的保密性，而且用户需要确信其客户端连接的是正确的服务器。其他的应用或许有更高级的安全需求，但是任何 Web 安全协议都需要满足这些基本要求。

● 基本技术

在讨论安全之前，重要的是理解构成万维网（World Wide Web）基础设施的基本技术。我们需要考虑的三种技术为 HTTP、HTML 和 URL。

● HTTP

HTTP（超文本传输协议，HyperText Transfer Protocol）是 Web 的基本传输协议。Web 是一种客户/服务器系统，因而需要存在某种在服务器与客户端之间传输数据的机制，HTTP 就提供了这样的机制。大多数 Web 浏览器都能够使用其他协议，如 FTP，但是绝大多数 Web 信息都是通过 HTTP 来传输的。

● HTML

HTML（超文本置标语言，HyperText Markup Language）是 Web 的基本文档格式。HTML 本质上是一种改进版本的 ASCII 文本。它所提供的两项最重要的功能就是对文档施加结构来表示段落、折行等内容的能力以及提供链接（link）的能力。链接允许用户通过点击从一个文档跳转到另一个文档。

● URL

URL（统一资源定位符，Uniform Resource Locators）提供链接中所使用的引用。每个 HTML 链接都有一个与之相关的告诉浏览器在用户点击该链接时执行相应动作的 URL。从理论上讲，URL 能够描述可以通过任何协议获取的数据，但是在实际应用中，它们大多是指通过 HTTP 获取数据。

● 现行要考虑的问题

现实的 Web 环境还有几项功能还无法从到目前为止我们所提出的简单客户/服务器模型中显露出来的。在我们考虑 Web 安全问题的时候，这三种功能实际上有着特殊的干系，它们是连接行为、代理和虚拟主机。

● 连接行为（Connection Behavior）

大多数 Web 页面都包含一些内嵌的图片，必须通过单独的 HTTP 请求逐个获取。作为一种性能上的改进，大多数客户端都是并发地获取这些图片。因此，获取任何页面实际上都牵扯到一系列的请求。为了确保合理的性能，需要确保将单个请求的开销保持为最小。

● 代理（Proxy）

数目巨大的 Web 交易都是从大型的内部网环境引发的。这些内部网往往通过防火墙与因特网隔离，只有通过代理才能存取因特网资源。成功的 Web 安全解决方案必须能够通过代理进行数据传输。

● 虚拟主机（Virtual Host）

由一个单一的服务器（如一个 ISP 的服务器）来负责多个组织 Web 站点的情况相当常见。举例来说，由 ISP 提供 Web 设施并为那些太小而没有自己经营账户的商家提供信用卡清算就相当常见。自然，尽管多个商家的服务器实际上是在同一台服务器上运行，但是每个

商家仍然想表面上拥有自己的 Web 服务器。一种称作虚拟主机的技术可以完成这项工作，但却无法与安全技术配合地良好。

安全上的考虑

一旦理解了我们想要完成的交易类型、Web 协议及其所需的工作环境，就该考虑如何来处理第 7 章所讨论的重要问题：协议选择、客户端认证、引用完整性和连接语义。

9.3 HTTP

本节提供了在[Fielding1999]中所描述的有关 HTTP 的简明概要。我们并不打算提供全面的描述。有许多信息来源都对这些知识进行了透彻地描述，其中包括[Stevens1994]。相反，我们的目的是讲述足够多的与 HTTP 有关的信息，以便能对使用 SSL 来保证 HTTP 安全所表达的含义进行充分论述。因此，我们将重点放在与安全和 SSL 关系最为密切的细节上面。

从概念上讲，HTTP 是一种简单的协议。HTTP 交互的基本单位是请求/响应对。客户端打开一条到服务器的 TCP 连接并将请求发送出去。而服务器将响应发回，服务器使用表示长度的头信息（header）或仅仅通过关闭连接来指示响应的结尾。

请求

HTTP 请求包括三个部分：请求信息行（request line）、头信息（header）以及可选的信息体（body）。请求信息行只有一行信息。头信息是一系列以冒号分隔的键-值对，信息体则为任意的数据，头信息与信息体之间由一个空行分隔。图 9.1 描述了一个请求范例。

```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.7 [en] (X11; U; FreeBSD 3.4-STABLE i386)
Host: www.rtfm.com
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
(blank line)
```

图 9.1 一个 HTTP 请求

图 9.1 中所展示的请求仅由请求信息行和头信息组成。第一行（以 GET 开始）为请求信息行，其余部分为头信息。

HTTP 请求信息行的格式为：

Method Request-URI HTTP-Version

HTTP/1.1 的 RFC 定义了七种请求方法：OPTIONS、GET、HEAD、POST、PUT、DELETE 和 TRACE。在 WEBDAV [Goland1999]中还定义了其他一些方法，但我们不必关心各种请求方法之间的差异。两种最常见的方法是 GET 和 POST，它们之间惟一有关的差别就是 POST 可以有一个消息体，而 GET 没有。

Request-URI 应当被看作我们企图存取的资源名。总的来说它看上去就像 UNIX 路径名，如 /foo/bar/baz，其末尾还可以有一系列参数。最后，HTTP-Version 一般为 HTTP/1.0 或 HTTP/1.1。HTTP/1.1 是 IETF 进行标准化的版本，不过我们可以很大程度上忽略这两个版本之间的差异。

可能存在大量的头信息字段，但是其中的大多数都与 SSL 没什么关系。图 9.1 中的请求就有一条与之相关的头信息行。头信息 Connection 指示客户端想让服务器在发送完响应之后保持连接打开。因此，keep-alive 影响 SSL 的关闭过程。我们将在第 9.15 节进一步讨论这种影响。

客户端传递给服务器的所有信息都位于客户端请求中。因此，如果要使数据保密的话就必须确保客户端请求是加密的。注意，这必然包括请求信息行的保密，因为客户端获取的资源标识本身有可能是敏感的。显然，若要提供这种服务则要求客户端知道它是在与恰当的服务器进行交互。

响应

HTTP 响应与请求的格式非常近似，惟一的差别就是请求信息行被替换为一个指示服务器如何处理请求的状态行。状态行的格式为：

HTTP-Version Status-Code Reason-Phrase

HTTP-Version 与请求中的一样。Status-Code 是一个表明服务器执行动作的数字编码。一般来讲，200 表示成功的请求，而其他的数字则表示失败（300 系列的编码用来表示其他一些未对请求提供服务的非出错情况）。在第 9.21 节讨论升级问题时，我们会见到多种状态编码。Reason-Phrase 是对所发生情况的简单文本描述。对于成功的事务，它通常是 OK。图 9.2 描述了一个成功的 HTTP 响应——它是对图 9.1 中请求的响应。

图 9.2 中的响应演示了许多重要的信息。首先，请注意状态行 HTTP/1.1 200 OK，它表示请求成功。同时请注意下面三个头信息字段：

```
Content-Length: 1650
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
```

头信息字段 Keep-Alive 表示服务器不会在发送完响应之后关闭连接。意思就是说客户端必须根据响应的内容来判断响应何时结束。头信息 Content-Length 告诉客户端响应体中的字节数。Content-Type 行告诉我们消息体的类型为 text/html，它表示消息体是 HTML 文档。

显然，如果要想真正保证 HTTP 事务的安全，就必须为响应提供保密性。这不仅是指确保攻击者无法查看其内容，而且还要确保攻击者不能冒充成服务器来给客户端发送数据或是篡改传输过程中的数据。

9.4 HTML

HTML（超文本置标语言，HyperText Markup Language）就是使用一系列给文档增加结构的标记（称作标签）来修饰的 ASCII 文本。例如，标签 <P> 表示开始一个新的段落。Web 浏览器包含 HTML 解析器，它以接收到的 HTML 作为输入，产生格式化的页面输出。注意，

由于大多数标签描述的都是结构而不是布局，所以存在多种对给定页面进行格式化的方法。

关于 HTML 我们惟一需要关心的就是其中包含的链接（link）。链接就是页面中指向另一内容的引用。那部分内容可以依次由 Web 浏览器进行获取，通常使用 HTTP，但有时也使用其他协议来存取该内容。

```
HTTP/1.1 200 OK
Date: Sat, 15 Jan 2000 05:15:54 GMT
Server: Apache/1.3.1 (UNIX)
Last-Modified: Tue, 22 Jun 1999 19:25:14 GMT
ETag: "2a99d-672-376fe31a"
Accept-Ranges: bytes
Content-Length: 1650
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<HTML>
<HEAD>
<TITLE>RTFM</TITLE>
</HEAD>
<!-- Background white, links blue (unvisited), navy (visited), red (active) -->
<BODY
  BGCOLOR="#FFFFFF"
  TEXT="#000000"
  LINK="#0000FF"
  VLINK="#000080"
  ALINK="#FF0000">
<
<CENTER>
<A HREF="contact.html">
<IMG SRC="rtfm.gif" BORDER=0></A>
</CENTER>
Deleted text
RTFM in cooperation with Claymore Systems is releasing a free
Java SSLv3/TLS implementation: <A HREF="/puretls">PureTLS</A>.
<P>
&nbsp;
<CENTER>

<A HREF="contact.html">Contact Us</A>
</BODY>
</HTML>
```

图 9.2 一个 HTTP 响应

锚点（Anchor）

大家最为熟悉的链接类型就是被称作锚点（anchor）的那种。它是被标记为与给定引用对应的一段 HTML。在与该段 HTML 对应的屏幕部分上进行点击就会使浏览器取得指定的

内容。当你在浏览器中各种加重显示并带有下划线的链接上点击时，就是在使用锚点。图 9.2 所示的 HTML 页面就包含一个锚点：

```
<A HREF="/puretls">PureTLS</A>
```

屏幕上显示的是带有下划线的 PureTLS。在带有下划线的部分上点击就会使浏览器通过对链接进行解析来获取由/puretls(在从它那儿获取最初文档的 Web 服务器上: www.rfm.com)指向的文档。HREF=“puretls”指定链接的目标。连接的目标(这里就是字符串/puretls)称作 URL(统一资源定位符)。我们将在第 9.5 节讨论 URL 的语法。链接本身是由(起始标签)和(终止标签)括起来的区域来定义的。

插图 (Inline Image)

另一种链接类型就是被称作插图的类型。与锚点不同，浏览器自动获取插图并将其放在 HTML 中出现图片标签位置的 Web 页面中，而锚点则要等到用户在其上点击时才进行获取。(由于 HTML 页面的布局部分是由浏览器决定的，位置的概念相当模糊。)

你在 Web 页面中看到所有图片都是某种类型的插图，即便是动画图片也是由一系列插图实现的。图 9.2 中所示的 HTML 页面中包含一个指向一幅插图的引用：

```
<IMG SRC="rtfm.gif" BORDER=0>
```

IMG 标签本质上与 A 标签类似，只不过它使用 SRC 字段而不是 HREF 字段来承载指向引用内容的 URL。所以，文件 rtfm.gif 中存储的图片(RTFM 的标识图案)将会在这个页面中显示。该标签还包含一个 BORDER=0 属性，它表示显示图片时不带任何边界。

注意，插图的安全属性或许与引用它的页面的安全属性不同。由于图片常被用做表达页面内容的一部分，所以浏览器应当清楚地表明其安全属性是否与所嵌页面的安全属性有所不同。

表单

我们要考虑的最后一种类型的链接就是一般被称作 Web 表单的东西。尽管 Web 表单的实现是复杂的，但是其思想却非常简单而且大家都很熟悉。Web 页面显示一系列诸如文本字段和下拉菜单这样的用户界面组件。有一个称作发送按钮的特殊按钮，该按钮有时还被标注为“submit”。

用户以某种方式与组件进行交互，比如说选择他所拥有的信用卡类型以及想用来付费的信用卡号和有效期。在完成这些工作之后，用户点击发送按钮。此刻魔力发生了：浏览器接收所有组件的当前值并构建一个承载这些值的字符串。然后向服务器发送一个请求(发给与表单关联的 URL)将组件的值通过这个请求发送过去。

可以为表单(实际上是所有类型的链接)标注一个代表在解析链接时所使用的 HTTP 方法的方法类型。与表单有关的两种方法为 POST 和 GET，它们之间存在简单而重要的差别：在 GET 方法中，表单中字段的值附加于请求信息行中的 URI 上。而在 POST 方法中，值是在消息体中发送的。

动态内容

Web 页面可以包含各种类型的动态内容：由 Web 客户端执行的代码。这种代码可以直接位于 HTML 中也可以通过与插图类似的链接来引用。代码可以用多种语言编写，其中包括

括 Java、JavaScript 和 VBScript。还有可能让链接引用加载到 Web 浏览器内存空间中的二进制程序，这被称作插件（plug-in）。

9.5 URL

URL 是 World Wide Web 中基本的寻址形式。URL 就是提供单一的标识任何可通过网络存取的资源的短字符串。URL 为种类繁多的各种不同的存取方法提供了统一接口。URL 可以引用那些通过 HTTP、FTP 和 Gopher 存取的文档，甚至还可以指导浏览器创建邮件消息。这免除了用户熟悉各种不同规范的必要，而在此之前则要求使用特定的存取方法才行（例如，匿名 FTP）。

URL 可以非常复杂，对于不同的存取方法来说区别非常大（请参见[Berners-Lee1998]来了解完整的信息），然而我们关心的所有 URL 都有着共同的形式：

```
<scheme>://<host>[:<port>]/<path>[?<query>]
```

scheme 对应存取资源的协议。所以对 HTTP 来说就是 http，对 FTP 来说就是 ftp。host 和 port 字段指定要连接的服务器。包裹 port 的方括号表示它是可选的。协议通常定义默认的端口号，但是允许通过任何端口进行存取。HTTP 的默认端口为 80。

URL 的 path 部分提供指定所在服务器上的资源名（或位置）。路径常常与 UNIX 文件名类似（/foo/bar/baz）。从理论上讲，与文件名看上去类似并不意味着其各部分与目录、文件等对应，但是在实际应用中却常常是这样的。在这种情况下，服务器将会提供与资源对应的文件。

最后，URL 有可能在末尾放置一项以?界定的查询（query），该查询特定于资源并“由资源来解释”。查询最常用在资源不是文件而是动态产生资源的程序情况下。在这种情况下，使用查询部分来提供程序输入。例如，ldap: URL 对应 LDAP（轻量目录存取协议）目录服务器的一个条目可以使用查询信息来标识客户端感兴趣的属性、查询范围，以及其他查询参数。

一个例子

简单的 URL 类似图 9.3 描述的这样：

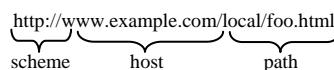


图 9.3 一个简单的 URL

这个 URL 所描述的资源描述如下：使用 HTTP 连接到机器 www.example.com 的端口 80 上，请求资源/local/foo.html。

还可以使用像/local/foo.html 这样的相对 URL。在这种情况下会假定模式和主机与获取包含 URL 的文档所使用的相同。

URI 与 URL 的对比

HTTP 请求信息行包含的是一个 URI。我们刚才已经见过了什么是 URL，但是 URI 是

什么东西呢？URI 代表统一资源标识符，URI 是 URL 的超集。一个 URI 就是一个指向给定资源的短字符串。URL 是包含有关如何获取资源的具体指示的 URI。可以有惟一标识资源但不提供如何获取它的指示的 URI。此种类型的 URI 被称作 URN（统一资源名）。

一般来讲，你可能见到的 URI 都是 URL，因为它们描述了如何使用 HTTP、FTP 等来获取资源。可以想像，HTTP 请求信息行可以包含非 URL 的 URI，所以 RFC2616 使用 URI 这个术语。就本章剩余部分而言，我们将在谈论 HTTP 请求中出现的 token 时使用 URI 这个术语，并在谈论链接中出现的 token 时使用 URL。请参见[W3C2000]来了解有关这个主题的更多信息。

9.6 HTTP 的连接行为

考虑一个典型的 Web 页面，该页面大概有 5 到 10 个插图。正如我们在前一节所讨论的，每个插图都需要一个单独的 HTTP 请求/响应对。由于大多数比较老的 Web 浏览器和服务器都是通过关闭连接来指示响应的结尾，所以获取一个单一的页面潜在需要多达 11 个单独的 TCP 连接。

像 Mosaic 这样非常老式的浏览器顺序建立所有这些 TCP 连接，但是该过程使得性能结果非常糟。于是 Netscape 很快就引入了同时打开多个 HTTP 的连接技术，每个连接对应一幅图片，连接数的上限可以配置。

1994 年 Spero [Spero1994] 说明了为何使用多个连接（无论是顺序还是并行）与 TCP 进行作用会导致糟糕的性能。首先，每个连接的初始化都需要一轮三次握手（1.5 倍的往返开销）。更糟的是，TCP 的慢启动（请参见 [Jacobsen1988] 中的描述）会在客户端请求长度超过服务器的最大段尺寸时造成严重的延迟。

为了避免这些问题，[Padmanabhan1995] 和 [Mongui1995] 建议在发送页面之后保持 TCP 连接打开，这种方法最终演变为我们在第 9.1 节所见到的 Connection: Keep-Alive 头信息。尽管持续性连接减少了浏览器建立连接的数目，但是许多浏览器仍然在加载带有大量图片的页面时初始化许多连接，以期并行地加载所有图片。

使用大量并行连接是 SSL 会话恢复功能的主要动机。因为即便是简单的页面也会在几秒钟的会话中产生大量的连接，所以通过会话恢复，SSL 的性能可以得到极大地改善。

9.7 代理

HTTP 代理是一种位于客户端与服务器之间的程序。它接受客户端的请求，然后再把自己的请求发送给服务器。代理之所以有用主要有两个原因：为多个客户端实现高速缓存以及作为一种通过防火墙的方式。我们所面对的首要问题就是确保使用 SSL 的 HTTP 能够在代理的环境中操作。本节提供了一些有关代理如何工作的背景知识。

浏览器与代理之间流通的协议信息几乎与浏览器与服务器之间流通的协议信息一样，然而又并非完全相同。我们所要关心的主要差别就是请求信息行上的差异。问题就在于标准的请求信息行中没有包含服务器身份，其中包含的只是服务器上资源地址。因此，在与代理进

行交互时，客户端使用完全限定的 URL，其中不但包含资源路径还包含有主机名。对于图 9.1 中所示的请求，请求信息行如下所示：

```
GET http://www.rfm.com/HTTP/1.0
```

在与服务器进行交互的时候，代理会把主机名去掉并像客户端发送请求给服务器一样将请求发送给服务器。在与代理的交互过程中存在许多微妙之处，一些头信息字段是供代理使用的，而代理应当在将请求发送给服务器时对其进行处理（和删除）。其他一些是供服务器使用的，代理则应当忽略它们。我们将在第 9.21 节讲述升级问题时见到演示这种差异的例子。

● 缓存代理

缓存代理背后的思想非常简单。许多 Web 页面是静态的而且存取频繁，如果将这些页面缓存在客户端的机器上，就可以极大地改善服务器的负载和性能。大多数浏览器目前在硬盘或内存中缓存文件，但这只有在客户端重复获取相同的数据时才会有用。即便如此，这种优化的好处也是令人吃惊的，考虑一幅诸如菜单条或标志的图片在整个站点中反复使用的情况。

在一个许多用户都有可能获取相同页面的环境中，一种更为主动的策略会物有所值。如果在代理处完成缓存，那么一旦有用户获取了某个页面，那么所有的用户都将受益。自然，浏览器都必须被配置为与缓存而不是直接与服务器建立连接。这种配置可以手工完成，但是许多浏览器还提供了自动配置功能，它允许 IS 管理员为所有的内部客户端配置代理。

主要困扰缓存代理设计与实现的问题就是如何保持缓存的信息是最新的。有必要检测出那些动态而无须缓存的内容以及定期检查某个文档是否已经在服务器上更新。HTTP 包含处理这种问题的复杂机制，但是它们在很大程度上与 SSL 无关，因为从客户端到服务器的信息是加密的，因此不管怎么缓存都是不可能的。

● 防火墙代理

防火墙代理的目的是限制整个内部（受保护的）网与因特网之间的通信。尽管一些防火墙允许内部网与因特网之间建立任意的 TCP 连接，但大部分都不行。这样的防火墙通常依赖应用层网关来激活必需的（通常是受限的）服务。

防火墙代理的主要设计目标当然就是安全。就此而言，这些代理通常非常简单而且被设计成没有错误的。尽管防火墙 HTTP 代理也可以是缓存代理，但是通常这样的代理不含缓存。SSL/代理交互的主要问题就是确保 SSL 之上的 HTTP 信息能够安全地不受代理破坏地通过代理。

9.8 虚拟主机

考虑顾客的 Web 站点由因特网服务提供商 (ISP, Internet Service Provider) 管理的情况。尽管一家 ISP 可能有许多顾客，但它只想使用几台机器。这就必须要有在同一台服务器机器上运行许多 Web 服务器的能力。

自然，每个顾客都希望能有自己独立的 Web 地址。即便这些 Web 站点都是在一台服务

器 `www.provider.com` 上提供的，也希望它们的客户能够通过 `http://www.customer.com/` 来访问它们的主页。因为只有一个 Web 服务器，所以这台服务器将会替多个虚拟服务器接收请求。现在的问题是如何区分给定的请求意图发往哪个虚拟服务器。Request-URI 只指出了服务器上的资源路径，因此必须有其他的办法才行。

现代 Web 客户端使用 Host 头信息来提供它们认为要连接的 Web 服务器的主机名。这就允许 Web 服务器辨别出同一台物理服务器上的多个虚拟服务器，显然设计良好的 SSL 上的 HTTP 应当确保不会破坏虚拟主机。

9.9 协议选择

当第一个 SSL 上的 HTTP 被设计出来的时候，还没有发明 SSL 的磋商升级策略，而且 HTTP 服务器的一项重要设计考虑就是它们是无状态的（stateless）——每个请求/应答对都是独立的。因此，一种要求通过 HTTP 引擎几个来回的协议模型（我们将会看到这是升级所要求的）就很难被接受。SSL 背后的设计思想就是你可以简单地将套接字调用替换为 SSL 调用而别的代码一点都不用动。如果尽量减少对其余代码的影响是压倒一切的设计目标（常常是这样的），那么独立端口方案就是惟一一种可行的策略。

9.10 客户端认证

基于证书的客户端认证对于安全表单提交而言是完全没有必要的。在实际应用中，大多数 Web 站点都选择根本不对用户进行认证。要进行客户端认证的 Web 站点通常希望使用外部的身份认证措施（诸如信用卡号码）。因此，基于证书的客户端认证对 Web 应用来说并不是一种优先选择的措施。

尽管 SSLv2 中的客户端认证并没有得到广泛支持，但是大多数 SSLv3 和 TLS 实现确实都支持客户端认证。在某些保密（内部网）环境中，基于证书的认证非常有用而且很实用。因此，Web 安全方案也应当能够支持客户端证书。

HTTP 无须提供任何特殊的协议支持就可以支持使用证书（和 SSL 相反），因为服务器只需在 SSL 握手中请求客户端进行客户端认证。然而，因为 SSL 握手是在服务器知道客户端企图存取的是哪种资源之前发生的，因此倾向于采取一种孤注一掷的方式。我们将在第 9.18 节看到，最令人满意的方法还是能够让服务器在引用中指明它期望客户端认证，这样就能够允许客户端在握手中予以提供。

9.11 引用完整性

Web 有一个非常明确的引用模型：资源是通过 URL 来标识的。一种显而易见的支持 SSL 的 HTTP 设计方案就是简单地将 URL 引用与服务器身份进行匹配。这要求在 CA 颁发的证书中包含服务器的主机名，这就是 HTTP 设计者们选择的方案。

更为复杂的方案就是让引用本身包含附加所期望的服务器身份的指示。可以将这种信息

安置在 anchor 中（就像 Secure HTTP 所做的那样 [Rescorla1999a]，请参见第 11 章）或是放在 URL 的某个地方。这种方案的主要缺点就是它会使得这种键入 URL 的经常性操作变得困难得多。然而它的主要优势是，就服务器所能拥有的证书种类而言，可以有非常高的灵活性。我们将会在第 9.17 节讨论虚拟服务器的时候看到一个展示这种灵活性的例子。

连接语义

SSL 上的 HTTP（即支持 SSL 的 HTTP）本质上限制使用 HTTP 的连接语义。这显然存在问题，因为 SSL 握手的开销远比 TCP 握手的开销昂贵得多。减轻握手负担是快速恢复机制动机。不幸的是，恢复机制破坏了 HTTP 的无状态模型。事实上，我们将在第 9.24 节看到，它给服务器施加了不小的负担，因为实现恢复要求进程间沟通会话状态。

如果我们把 SSL 看作是在没有参考 HTTP 的情况下设计的，而后又将其应用于 HTTP，那么这种负担看起来就像一种通用安全协议与 HTTP 交互而导致的不幸后果。然而，实际情况却是 SSL 在很大程度上是为 HTTP 设计的，而且如果设计者考虑到这些问题的话，做些修改来减轻状态负担原本是可能的。我们将在第 11 章讨论安全 HTTP 的时候讨论几个与此有关的技巧。不管怎样，在设计 SSL 的时候，由于第 9.6 节所讨论的 TCP 性能因素和在 UNIX 系统上难以接受的性能影响，这种无状态交互模型已经成为解决问题的出路。

9.12 HTTPS

现在我们已经从设计的角度探讨了 Web 安全的问题，为理解 HTTPS，这种站主导地位的 HTTP 与 SSL 解决方案作好准备。我们还准备评估 HTTPS 在应对我们所讨论的各种挑战时实际处理的有多棒。

我们先通过查看一种使用 HTTPS 发送的简单请求来演示一下它的基本功能。尽管 HTTPS 背后的基本思想非常简单，但一些技术要点仍需要仔细定义：连接关闭行为和使用 URL 来提供引用完整性。一旦清楚地定义了 HTTPS，就可以考虑它与实际网络环境中所面对的各种复杂特性代理与虚拟主机之间的影响。我们还要考虑在真实的 Web 环境中，判定何时要求基于证书的客户端认证的问题。

尽管 HTTPS 本质上是安全的，但也发现了许多种攻击和限制。我们在第 9.19 和 9.20 节讲述这些攻击以及避免这些攻击的技术。最后考虑 HTTP 升级，这是除 HTTPS 以外另一种解决方案。HTTP 升级用以改正 HTTPS 一些已知的问题，但是我们将会看到，它自身也引入了一些问题（HTTPVupgrade）。

9.13 HTTPS 概述

HTTP 是第一种使用 SSL 来保护其安全的应用层协议。1995 年，首个公开发表的 SSL 上的 HTTP 实现出现在 Netscape Navigator 2 中。那时只有独立端口策略可用，Netscape 不光在 HTTP 上使用了这种策略，对 NNTP 和 SMTP 也使用了该策略。通过 SSL 上的 HTTP 来获取页面的 URL 以 `https://` 开头，以便将其与 HTTP URL 区分开来，这种方案很快就以 HTTPS

成名。

HTTPS 代表“HTTP Secure”。这种表达似乎有些僵硬，但 Netscape 不能使用 shttp://，因为 Secure HTTP（请参见第 11 章）已经使用了该协议名来表示其 URL 模式。

虽然 SSL 使能的 HTTP 实现部署广泛，但是 Netscape 以及其他团体都没有发布 SSL 上 HTTP 的标准，直到 1998 年 IETF 才开始考虑这个问题。实际上，普遍认为只存在一种清晰透彻的解决方法，而具体的（并不算少）细节像“口头文学”一样四处传播。最终在 RFC2818[Rescorla2000] 记录了 HTTPS 的内容。

HTTPS 这种方案非常简单：客户端连接到服务器，磋商一条 SSL 连接，然后在 SSL 应用数据通道上传输它的 HTTP 数据。这种表述听起来非常简单而原则上也确是如此。为了创建一种可以互操作的（尽管并不一定安全）实现，另一种你需要掌握的知识就是端口号。因为我们使用独立端口的策略，所以 HTTPS 需要它自己的端口。IANA 给 HTTPS 分配端口 443，所以 HTTPS 连接默认使用该端口。当然，也可以在 URL 中指定不同的端口号。

图 9.4 描述了一个 Netscape4.7 客户端与运行 mod_ssl 2.4.10 的 Apache 1.3.9 服务器之间的 HTTPS 请求活动。所有初始的通信数据（记录 0-记录 9）都是 SSL 握手。这是一种相当普通的 SSLv3 握手，只不过客户端是使用 SSLv2 向后兼容握手来连接的。即便对于现代 Web 浏览器，这仍然是最常见的情况。还要注意的一点是，客户端只提供了 SSLv3 而没有提供 TLS。在编写这本书的时候，Netscape 中还没有支持 TLS。

```
New TCP connection: romeo(4577) <-> romeo(443)
I 948676151.6444 (0.0005) C>S SSLv2 compatible client hello
    Version 3.0
    cipher suites
        TLS_RSA_WITH_RC4_128_MD5
        value unknown: 0xffe0 Proprietary Netscape cipher suite
        TLS_RSA_WITH_3DES_EDE_CBC_SHA
        value unknown: 0xffe1 Proprietary Netscape cipher suite
        TLS_RSA_WITH_DES_CBC_SHA
        TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
        TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
        TLS_RSA_EXPORT_WITH_RC4_40_MD5
        TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
2 948676151.6495 (0.0051) S>C      Handshake
    ServerHello
        Session_id[32]=
            15 07 d3 46 a9 40 bc bc 6f 54 f9 60
            40 d0 bf 2f 08 3e 1e 4e f4 1d 7c 52
            31 46 14 20 ad 95 5b 04
        cipherSuite          TLS_RSA_WITH_RC4_128_MD5
        compressionMethod    NULL
3 948676151.6495 (0.0000) S>C      Handshake
    Certificate
4 948676151.6495 (0.0000) S>C      Handshake
    ServerHelloDone
5 948676151.6637 (0.0141) C>S      Handshake
```



```

ClientKeyExchange
6 948676151.6900 (0.0262) C>S ChangeCipherSpec
7 948676151.6900 (0.0000) C>S Handshake
    Finished
8 948676151.6921 (0.0020) S>C ChangeCipherSpec
9 948676151.6921 (0.0000) S>C Handshake
    Finished
10 948676151.6933 (0.0012) C>S application_data
    data: 284 bytes
-----
GET /tmp.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.7 [en] (X11; U; FreeBSD 3.4-STABLE i386)
Host: romeo
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,* ,utf-8

-----
11 948676151.7013 (0.0079) S>C application data
    data: 395 bytes
-----
HTTP/1.1 200 OK
Date: Mon, 24 Jan 2000 01:09:11 GMT
Server: Apache/1.3.9 (Unix) mod_ssl/2.4.10 OpenSSL/0.9.4
Last-Modified: Sun, 23 Jan 2000 23:08:11 GMT
ETag: "58820-79-388b89db"
Accept-Ranges: bytes
Content-Length: 121
Connection: close
Content-Type: text/html

<HTML>
<HEAD>
    <TITLE>Test</TITLE>
</HEAD>
<BODY>
<H1>
    Test Page
</H1>
    This page is just a sample.
<P>
</BODY>
</HTML>
-----
12 948676151.7063 (0.0050) S>C Alert
    level          warning

```

```
        value      close_notify
Server FIN
13 948676151.8052 (0.0989) C>S Alert
    level      warning
    value      close_notify
Client FIN
```

图 9.4 使用 HTTPS 的请求

记录 10 包含了 HTTP 请求。该请求本质上与我们在图 9.1 中描述的一样，只不过它针对的是不同的 URL。注意，在磋商好 SSL 连接之前客户端绝对不会发送数据。我们很快就会看到在从 HTTP 系统转向 HTTPS 时，这种性质怎么会产生问题。

记录 11 包含 HTTP 响应。注意 HTTP 响应完全装在一条记录中。当然，如果 Web 页面比较长的话，就有可能需要跨站多条记录。除了页面本身不同以外，这种响应基本上与我们在图 9.2 中描述的响应类似。

惟一重要的不同之处就是服务器发送的不是 Connection: keep-alive 而是 Connection: close 头信息，指示在传完这个页面之后它会关闭连接。我们已经将服务器配置成不使用保持连接，以便描述关闭行为。

正如我们所期望的，服务器发送它的 close_notify 并跟着发送 TCP FIN。从技术上讲，服务器有可能暂不发送 FIN，直到它收到客户端的 close_notify 之后为止。SSL 规范就这一点没有规定。然而在实际应用中，Netscape 的早期版本在收到 FIN 之前是不会响应 close_notify 的，因此立刻发送 FIN 是一种良好的行为。

9.14 URL 与引用完整性

HTTP 连接的大部分工作都是通过解析某种 URL 开始的。因为 HTTPS 使用独立端口策略，所以 URL 必须指示使用的协议为 HTTPS 而不是 HTTP。只需使用协议标识符 https 来替代协议标识符 http 即可：

```
https://www.example.com/example.html
```

降级攻击

使用 https 协议标识符可以防止降级攻击。客户端期望初始化到服务器的 SSL 连接——实际上没有办法让服务器（或攻击者）指示其不使用 SSL。攻击者只能导致错误的产生。当然，我们先前讨论过，在 HTTPS 连接失败的情况下，用户有可能被诱导使用 HTTP 连接进行重试。

然而，https 只提供了一种信息：即客户端应该期望使用安全措施。这并未提供任何有关期望何种安全属性的指示。对 SSLv2 来说，这会构成严重问题，因为 SSLv2 没有抵御攻击者将连接降级为较弱算法的保护措施。然而，SSLv3 和 TLS 的却提供了算法降级保护。因为 SSLv3 和 TLS 还提供了针对降级为 SSLv2 的保护措施，所以只要服务器使用 SSLv3，就无法将磋商降级为较弱的算法。

只要对服务器进行了正确的认证，就可以安全地完成握手。因此，只要客户端不接收匿

名 DH 并像下一节所描述的那样对服务器的身份进行了充分检查，HTTPS（使用 SSLv3）就可以安全地抵御降级攻击。然而，如果没有对服务器进行正确认证，那么攻击者就可以实施中间人攻击。

端点认证

HTTPS 规范提供了一种颇为复杂的用来检查服务器身份的算法。其意图就是要包罗各种特殊的情况。在大多数情况下，可以从 URL 或某些类似的来源获得服务器的期望主机名。如果客户端知道服务器的期望主机名，那么它就必须根据服务器的证书对其进行检查。

另外，客户端有可能知道的不是主机名而是有关服务器证书的特定信息，在这种情况下，就要求它对那些信息进行检查。最后，尽管不进行任何检查是明确允许的，但是实际上绝不应当这样做，因为这会使连接处于主动攻击之下。

图 9.5 描述了主机名检查过程的流程图。如果存在具有类型 dNSName 的 subjectAltName 扩展，那么就必须对其进行检查。目前还没有颁发此种类型证书的 CA，但是将来的 CA 会这么做，因为这种方案比使用 Common Name 利落得多——Common Name 方案要求 CA 与用户之间就 Common Name 的语义的默认达成一致，而 dNSName 的语义是清晰的。

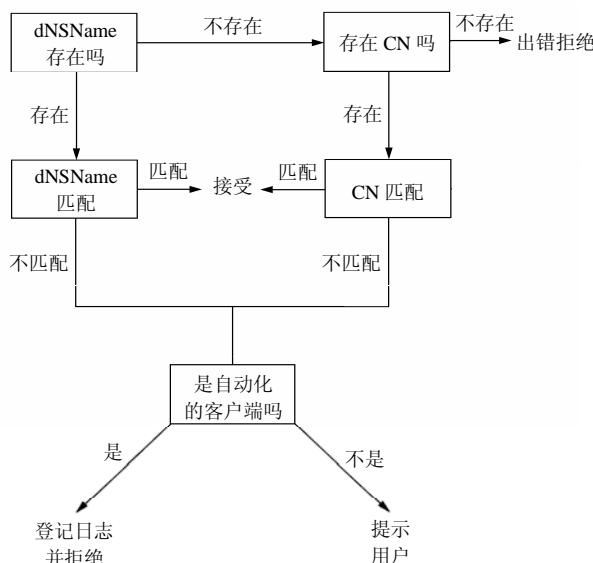


图 9.5 证书检查算法

然而，如果不存在 dNSName，就必须检查 Common Name。因为一个 DN 可能有两个 RDN，它们均指定 Common Name 值，因此规范明确要求要对最具体的 RDN 进行检查。没有提供显式地处理有关一个 RDN 包含有两个 Common Name 值的规则。明显的选择就是如果其中任一个名字匹配的话就接受这个证书。

RFC2818 也允许通配风格的 DN。名字中的部分内容可由“*”来代替。“*”可以匹配任何字符串。这样，如果 Common Name 为 *.example.com，那么这个证书对 foo.example.com 和 bar.example.com 来说都是可行的。通配只有在由同一个组织运行多个虚拟服务器的时候

才能真正很好地工作。你不能使用它来表示（举例来说）www.domain.com 和 www.anotherdomain.com，因为同时匹配这两个名字的通配符还可以匹配并非由证书持有者拥有的其他域名。

失败行为

对证书与服务器的期望身份之间不匹配的直接反应就是简单地终止连接。不幸的是，这种做法很容易引起用户的烦感。尽管这种不匹配原则上反映了一种主动攻击，但在实际应用中它更有可能代表服务器的配置错误。

Netscape Navigator 传统上通过向用户弹出描述错误，并提供关闭连接还是继续执行的对话框来响应这种不匹配（对于无法验证的证书也是如此）。老版本的 **Internet Explorer** 只是简单地关闭连接而拒绝继续执行。然而新版的 **IE** 允许用户继续执行。

HTTP 规范提供了另外几种处理不匹配的解决方案。诸如浏览器这样的客户端至少要将这种不匹配告知用户。它可以提供继续或简单终止连接的选择，自动化的客户端必须登记这种错误而且终止连接。

尽管人们知道在遇到证书不匹配的情况继续执行是不安全的，无法阻挡主动攻击，但是要记住的重要一点是，用户在使用 **HTTPS** 存取失败的情况下会企图使用 **HTTP** 来刷新页面。对于企图尝试非安全存取的情况，接受错误的证书还算是可以的，因为问题可能只是由于配置错误所造成的。例如，服务器管理员或许在没有得到新证书的情况下改变了服务器的 DNS 名。图 9.6 描述了各种可能的情况。如你所见，接受错误的证书绝不会比让用户通过 **HTTP** 重试还糟糕（有时甚至更好一些）。

客户端行为			
	失败	接受证书	使用HTTP重试
配置错误	无法进行通信	通信正常	数据以明文形式进行传输
攻击所致	攻击被阻止	导致成功的攻击	数据以明文形式进行传输

图 9.6 各种错误处理策略的结果

在证书处理失败的情况下判定如何行事要求检查你的安全策略以及用户的愿望。简单地接受（Implementation）认为是错误的证书是绝对不合适的。然而，在某些情况下，允许用户接受证书还算是比较好的解决办法。当证书的有效性验证失败时，**Netscape** 和 **Internet Explorer** 均会弹出对话框来允许用户继续对连接进行操作。

依照用户的选择

正如上面所讨论的，**Netscape** 和新近版本的 **IE** 都允许用户在证书不匹配的情况下继续执行。从用户的角度来看，这些浏览器的行为本质上是相同的。然而，从线路上传输数据的角度来看它们的行为颇为不同。

图 9.7 描述了一个 **Netscape** 客户端使用无法接受的证书连接到服务器上的情况。正如你所期望的，客户端等待服务器发送 Certificate 消息，然后保持连接打开并提示用户接受证书还是取消连接。在图 9.7 所示的跟踪信息中，用户选择接受证书，所以握手继续进行。

注意，根据证书具体的出错情况（不正确的主机名、未知的 CA、过期或所有三种情况兼有之），用户有可能要点击一系列的对话框才能接受证书并继续进行。这些操作要花费一些时间，服务器会因此而必须设置足够长的超时值，以便在此过程中不会超时。

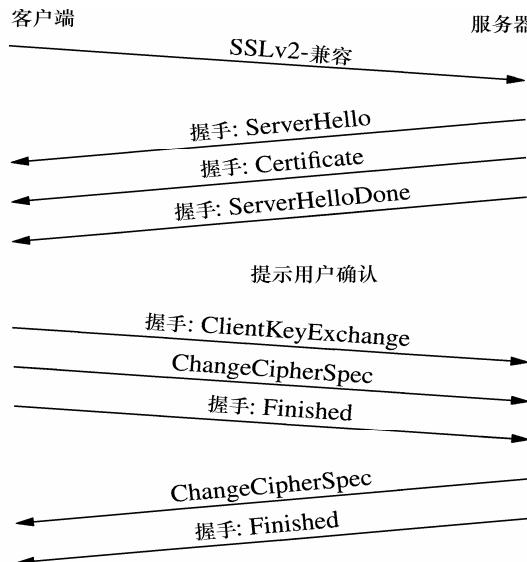


图 9.7 Netscape: 用户接受错误的证书

图 9.8 描述了当用户选择取消握手时 Netscape 的行为。当用户在对话框中按下取消按钮后，客户端给服务器发送一条 `bad_certificate` 警示并终止连接。



图 9.8 Netscape: 用户拒绝错误的证书

IE 的行为多少不那么显而易见。当 IE 收到一个错误的证书时，它会先完成握手，然后再关闭连接，如图 9.9 所示。一旦连接被关闭，IE 就提醒用户接受还是拒绝证书。如果拒绝证书，那么连接仍然处于关闭状态。如果接受证书，IE 就会初始化新的到服务器的连接。注意，IE 将恢复 SSL 会话，这样可以避免新完整握手的开销。

Netscape 的行为是你阅读规范时所期望的行为。然而，尽管 IE 的行为令人吃惊，但实际上并非是禁止。IE 这种行为的主要优势就是它在用户决定是否接受证书期间不消耗服务器的资源。因为服务器套接字多少算是一种有限资源，所以这种行为是一种对重负载服务器来说是一种十分必要的优化。然而，由于频繁使用的服务器应当具有由众所周知的 CA 颁发的证书，所以这种情况在实际中并没有那么严重。

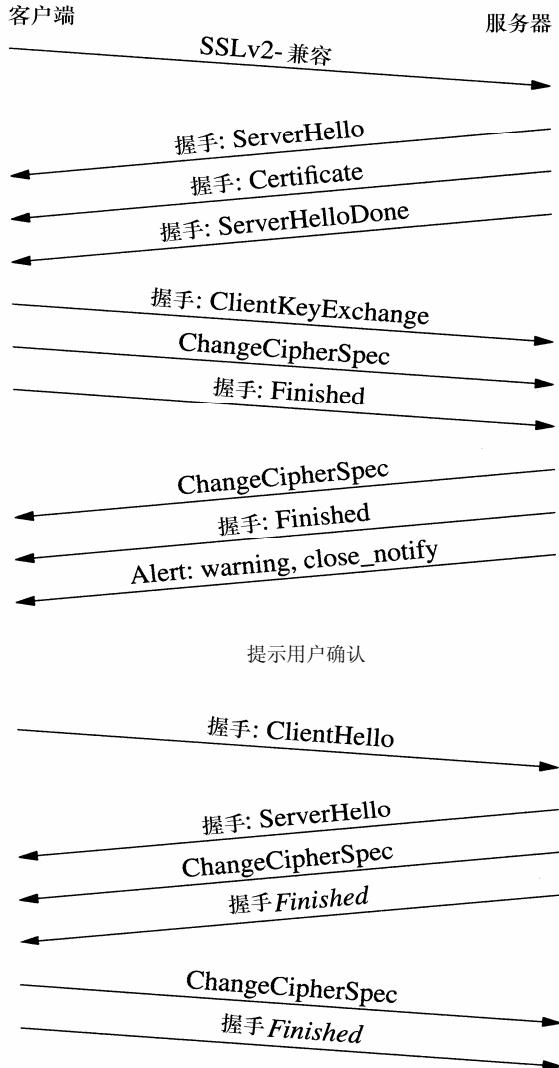


图 9.9 IE: 用户接受错误的证书

IE 的方案有两个不利之处：第一，服务器没有获得任何客户端不使用其证书的反馈。如果服务器管理员实际阅读日志文件的话，`bad_certificate` 警示会提醒它存在问题。其次，如果服务器不使用会话缓冲或在重新连接之前客户端会话超时的话，就需要进行一次新的完整握手，这对服务器和客户端来说都将是（指开销）。

由于回答接受/拒绝对话框会令用户烦感，所以 Netscape 和 IE 都将用户接受证书的结果缓存起来。Netscape 还允许用户指定永久性地接受某个证书，这样就不用再烦扰用户了。该功能到底是一个 bug 还是一种功能要由读者来判断了。

引用源

要指出的重要一点是引用完整性依赖引用信息被安全地传达给客户端。如果引用信息是以一种本身可以信任的方式传达的，那么前面所提到的属性仍然有效。例如，引用信息本身

可能是通过 HTTPS 连接来传达的。然而，对于引用信息通过不安全链接传达的情况来说，就有可能通过改变包含引用信息的页面而实施主动攻击。我们将会在第 9.20 节见到这样的攻击。不存在阻止这种形式的攻击的引用完整性检查。

客户端身份

在一些非 Web 环境中，“客户端”实际上是另一个主机。如果服务器知道这种情况，就应当完成我们上面所描述的同类型检查。

9.15 连接关闭

最后，我们必须考虑关闭的问题。正确的关闭行为总结起来很简单：无论何时哪一方想要关闭连接，都应当发送一条 close_notify 并选择等待一个 close_notify。TLS 与 HTTPS RFC 都要求这样的行为。我们需要关心的问题是在没有收到 close_notify 而收到了 TCP FIN 的情况下该如何行事。在某些情况下，这种行为明显代表一种可能的攻击。

但在其他情况下，则有可能只是一种编程错误。为了避免记录大量的可疑错误，我们希望能够对这些情况加以区分。

会话恢复

大家还记得我们在第 7 章讨论过，会话恢复的有关规则是清楚的。SSL 实现绝不应当在已经收到提前关闭指示的情况下恢复会话。

这样做就是赤裸裸地违反 SSL 规范。然而发送自己的 close_notify，而后不等待另一方的 close_notify 而关闭连接是允许的。在这种情况下，SSL 实现可以恢复会话。

错误处理

我们主要关心客户端的行为。首先，HTTP 语义暗示错误对客户端来说要比对服务器更为严重。服务器在请求期间收到数据结束标志，不管它是正常的 SSL 关闭还是 TCP FIN，其本质上的行为是一样的：放弃处理请求。其次，当服务器碰到错误时，通常只是将其登记在记账日志中。因此服务器编写人员尽可以安全地将任何非正常信息在日志中进行登记。

与之对照，客户端常常直接通过对话框以及类似的机制将错误告知用户。因此重要的是避免不报告合理错误（因为这样就向用户隐藏了重要的信息），而报告非合理错误（non-error）（因为这种数量巨大的信息会烦扰用户并增加用户忽略合理错误的机会）。

作为一种通用规则，我们想要在两类错误之间加以区分：

可能的编程错误。HTTP 有关连接关闭的通信指示与 SSL 行为不匹配。对于收到的消息，我们希望能够对那种没有指示存在安全问题的坏消息不进行报告。

截断攻击。客户端收到一条有可能不完整的响应。

可能的编程错误

考虑这样一种情况，客户端期望连接关闭，但接着收到提前关闭指示。举例来说，客户端从服务器收到一条包含头信息 Connection: close 的响应。客户端读取由 Content-Length 头信息指定的数据量。因此客户端期望连接立即关闭。现在客户端只收到了一个 FIN 而不曾收到一个 close_notify。有可能是攻击者伪造了 FIN，但更有可能是服务器出错而不正确地关闭

了连接。我们知道客户端已经收到了服务器所要发送的所有数据，因此不必担心页面被损坏，客户端可不用报告错误而直接进行显示。

一类稍微复杂些的情况涉及客户端没有期望会从服务器发来通信数据，但也没有期望服务器关闭连接。设想客户端收到了一条包含头信息 Content-Length 的响应并读取了所有的数据。然而，客户端相信自己是在使用持续连接，因此不知道服务器要关闭连接。（HTTP 允许服务器在不通知客户端的情况下关闭连接）因此，如果客户端在读完了整个响应之后收到了提前关闭指示，那么就有可能是攻击者在实施拒绝服务攻击，但是同样更有可能是服务器的毛病。对于任何一种情况，客户端知道它实际收到的响应是完整的并能够加以显示。

最后，考虑在期望之前出现 close_notify 的情况，例如在读取了头信息 Content-Length 指定的多少字节之前。这明显是一种服务器的编程错误，因为攻击者既不能伪造 close_notify 也不能改变头信息 Content-Length 而不被检测出来。

● 截断攻击

大家还记得较老版本的 HTTP 使用连接关闭来指示数据结束。因此，如果客户端期望从服务器接收更多的数据但是却收到了提前关闭指示，那么它就必须报告错误。首先，考虑不存在头信息 Content-Length 的情况。由于除了连接关闭之外客户端没有表示响应何时完成的指示，所以收到 close_notify 是知道整个响应都被接收的关键。

类似的，如果存在头信息 Content-Length，但是却在读取整个响应之前出现了提前关闭，那么就必须将其当作出错对待。当然有可能只是服务器错误地计算了长度信息，但是没有办法将这种错误与攻击区分开来。注意，我们对这种情况的处理与接收到全部长度的处理不同，对于那种情况，我们知道自己已经收到了全部的响应而任何可能的攻击只会阻止我们接收下一个响应。

对于任何一种有可能存在截断攻击以及响应有可能被缩短的情况，客户端都应当警告用户，通常是通过打印错误信息或弹出表示连接被意外关闭而且数据有可能被截断的对话框。特别较针的客户端有可能根本拒绝显示响应的任何内容。

9.16 代理

HTTPS 与代理的交互不怎么协调。主要问题就是 HTTPS 的语义暗含客户端要与目标服务器磋商 SSL 会话。一旦客户端完成了这项工作，所有在客户端与服务器之间传输的数据都会被加密，而缓存代理将无法工作。把这种行为看作是一项功能是合理的，因为我们企图在客户端与服务器之间传输的数据不管怎样都应当是保密的。然而，即便对于没有施加保密性的情况（即，只保持消息完整性的加密套件）不可能进行缓存。

● CONNECT

一种更严重的问题就是通常的 HTTP 代理语义不能与 HTTPS 一起工作。大家应该还记得代理要检查客户端请求才能决定要连接的服务器。HTTPS 要求客户端在加密通道上传输请求，因此这种方式显然不能工作。这是一种比无法进行缓存还要严重的问题，因为代理是穿过某些防火墙的惟一方法。因此，如果没有特殊的 support，HTTPS 无法与那些防火墙兼容。

这种特殊支持以一种新的代理方法的形式出现：CONNECT，该文献记录在 RFC2817



[Khare2000] 中。CONNECT 方法指示代理初始化一条到指定远端服务器的 TCP 连接，然后再在客户端与服务器之间传递数据而不对内容进行检查和变更。最后，客户端将 SSL 数据传送给代理就像代理是服务器一样。图 9.10 描述了从这样一种连接开始一直到传送 ClientHello 的流程。注意，客户端将指定的服务器主机与端口信息放在请求的 Request-URI 字段中提供给代理。

```
New TCP connection: romeo(2577) <-> romeo(80)
1 949442170.3636 (0.0002) C>S
data: 37 bytes
-----
CONNECT www.rfcm.com:443 HTTP/1.0
-----
2 949442170.3686 (0.0052) S>C
data: 102 bytes
-----
HTTP/1.0 200 Connection established
Proxy-agent: Apache/1.3.9 (Unix) mod_ssl/2.4.10 OpenSSL/0.9.4
-----
3 949442170.4403 (0.0769) C>S      Handshake
ClientHello
Version 3.1
cipher suites
    TLS_DHE_DSS_WITH_RC4_128_SHA
    TLS_DHE_DSS_WITH_RC2_56_CBC_SHA
    TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
    TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
    TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
    TLS_RSA_EXPORT1024_WITH_RC2_CBC_56_MD5
    TLS_RSA_EXPORT1024_WITH_RC4_56_MD5
    TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
    TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
    TLS_RSA_WITH_3DES_EDE_CBC_SHA
    TLS_RSA_WITH_IDEA_CBC_SHA
    TLS_RSA_WITH_RC4_128_SHA
    TLS_RSA_WITH_RC4_128_MD5
    TLS_DHE_RSA_WITH_DES_CBC_SHA
    TLS_DHE_DSS_WITH_DES_CBC_SHA
    TLS_RSA_WITH_DES_CBC_SHA
    TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
    TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
    TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
    TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
    TLS_RSA_EXPORT_WITH_RC4_40_MD5
compression methods
NULL
```

图 9.10 使用 CONNECT 代理

从 HTTPS 用户或程序员的角度来看，这种方案似乎工作得很好。然而，从防火墙管理员的角度来看，结果则并没有这么积极。防火墙管理员常常关心的是控制穿越防火墙的各种通信信息。例如，他们有可能想要屏蔽特定类型的 Web 内容。因为使用 CONNECT 传递的信息应该是不受检查的——而且从理论上讲是不可读的，因为它是加密的——这种类型的屏蔽现在是不切实际的。

能够使用 CONNECT 穿过防火墙的协议不仅限于 HTTP 上的 SSL。尽管从原则上讲，防火墙有可能判定这种通信类型是否为 SSL，但在实际应用中，这样做非常困难。因此 CONNECT 方法允许客户端在防火墙上打开一个出口并通过那个出口传递任意的通信数据。虽然管理员能够限制客户端连接的主机与端口号，但是这种强制措施收效甚微。

中间人代理

我们已经见过，总的来讲代理不能查看客户端与服务器之间的通信数据。然而，有可能通过让代理对连接实施中间人攻击来安排代理做到这一点。代理接受客户端的请求，但没有对数据进行代理而是磋商一对 SSL 连接、一条与客户端相连，一条与服务器相连。然后再在客户端与服务器之间传递数据，但是数据对代理来说是明文形式。图 9.11 描述了这一过程。

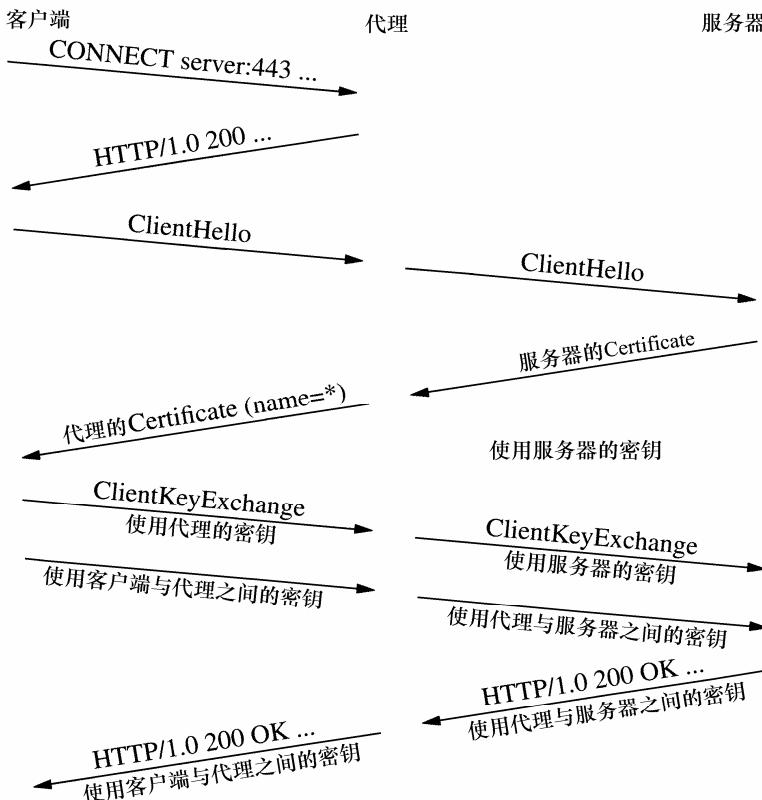


图 9.11 中间人代理的活动

但是这样怎么可能呢？SSL 应该能够抵御中间人攻击的，要做到这一点，需要客户端的协作。代理有一个非常特别的证书，该证书的 Common Name 为 “*”。因为主机名匹配是通配性质的，所以不管客户端与哪个主机进行连接它都会接受这个证书。显然，不存在合法的

会颁发这种证书的 CA，因为它可以让持有人冒充任何服务器。为了绕过这种限制，管理员需要运行自己的 CA 并将其安装到浏览器中。因为不处于管理员控制之下的浏览器不会在其受信 CA 列表中有这个 CA，所以它们不会被这种危险的 CA 所愚弄。

注意，RFC2818 禁止颁发这样的证书，因为“*”只能匹配一个单一的域名成分。然而，正如我们将要在第 9.17 节中所看到的，Netscape 允许“*”匹配任何内容。为了让这种技巧能够与符合 RFC2818 的浏览器一起工作，你需要有一个包含一系列模式的证书：“*”、“*.*”、“*.*.*”…

注意，这种类型的代理具有严重缺陷。如果代理被攻破，那么所有客户端的通信均会被攻破。由于代理位于防火墙之外（或是防火墙的一部分），所以它就成了一个很好的攻击对象。同样因为客户端并不直接与服务器相连，所以 SSL 客户端认证也就不再能够工作。

加密套件的转换

由于旧的美国出口法规的原因，客户端与服务器之间存在一种古怪的不对等。两种最为流行的浏览器，Netscape 和 Internet Explorer 都是由美国公司出品的，因此要受到严格的美国出口法规的限制。因为下载更为安全的“国内”版本很难，许多美国人都使用“出口”浏览器。因此，存在大量可用的浏览器，但却只有弱强度的加密算法。

目前最流行的 Web 服务器为 Apache。尽管 Apache 是在美国编写的，但存在几种可以很容易获得的用于 Apache 的 SSL/TLS 补丁程序，这些程序是在美国之外写成的。同时由于 Netscape 的服务器过去不是自由软件，要得到“国内”的 Netscape 服务器并不比获得“出口”服务器难多少。因此美国内所销售的拥有高强度加密的 Netscape 服务器占有相当高的百分比。总的来讲，大多数服务器都支持高强度加密。

C2 Net 利用这种状况提供了一种称做 SafePassage 的中间人代理。SafePassage 在客户端所运行的同一台机器上运行。它允许“出口”客户端使用弱强度加密与其进行连接，而它自己则使用高强度加密连接到服务器。由于 SafePassage 是在美国之外制作的，因此外国人可以取得一个拷贝来升级自己的浏览器。这样，弱强度加密只用来加密从没有离开机器的数据。

对 SafePassage 的需求随着 Fortify 的引入而减少，Fortify 是一种 Netscape 的补丁程序，它将“出口”版本升级为支持高强度加密套件的版本。而最终，当美国政府在 2000 年 1 月放开出口限制的时候，这种需求就消失了。不管怎样，SafePassage 是第一个中间人代理的例子，因此是一种具有创新意义的产品。SPYRUS 也使用了一种类似的方案将普通的基于 RSA 的加密套件转换为使用美国政府 FORTEZZA 卡的加密套件。

9.17 虚拟主机

HTTPS 与虚拟主机的交互非常糟糕，问题又一次出在 SSL 连接是在传送任何 HTTP 数据之前建立的。服务器平常使用请求中的头信息 Host 来判定正在存取的虚拟主机是哪一个。由于请求是在握手之后发生的，所以服务器必须在没有头信息 Host 的指导下磋商 SSL 连接。

考虑给客户端提供哪个服务器证书的情况。如果服务器实际位于独立的机器上，那么我们就期望它们拥有不同的证书，并在通常情况下期望每个虚拟服务器拥有它自己的证书。然而，要知道提供哪个证书则要求知道客户端企图连接的服务器虚拟服务器是哪个。如果虚拟

服务器有不同的安全策略而且想要磋商使用不同的加密套件，那么也要做出类似的考虑。

然而，还存在另一种实现与 SSL 保持一致的虚拟服务器的方法。有可能将机器配置成单个网络接口拥有多个 IP 地址的（称作别名）的形式。每个虚拟服务器分配有自己的别名。这样当服务器接受连接时，它就会查看自己接受的连接所在的 IP 地址并使用它来判断客户企图连接的虚拟服务器是哪个。

正如我们在 9.11 节所暗示的，另一种方案就是让 anchor 来指定服务器的证书，或许是在 URL 中的某个位置指定。那么服务器就能为所有的虚拟服务器使用同一个证书，同时仍保持引用完整性。

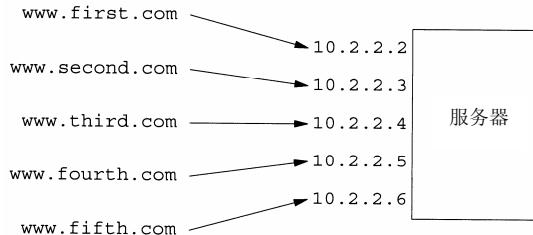


图 9.12 使用 IP 别名的虚拟主机

图 9.12 描述了包含 5 个虚拟服务器的虚拟服务器配置范例。DNS 名 www.first.com 分配地址 10.2.2.2，名字 www.second.com 分配地址 10.2.2.3 等等。注意，这种技术与使用 CNAME 记录截然不同。尽管多个 Web 地址指向同一个服务器，但是它们并不指向同一个 DNS A 记录，而是指向具有不同 IP 地址的不同 A 记录。

这种情况可以被认为是 SSL 中的一种设计缺陷。让客户端在 ClientHello 中指明它试图连接的服务器的 DNS 名会相当直接，这样就可以允许服务器完成相应的动作。这种改变不会对保密性造成破坏，因为攻击者可以很容易地根据其他公开的网络数据（IP 头信息以及握手的其他部分）来推测出客户端要连接的服务器是哪个。这是 TLS 的下一版本（如果有下一版本的话）所建议的几项改动之一。

注意，这项对 SSL 的改动不会使客户端摆脱必须要根据期望的身份对服务器证书进行检查的负担。不进行该项检查就有可能引入安全隐患。这项改动的惟一目的就是作为给服务器握手前的暗示。

● 多个名字

另一种稍微逊色一点的解决虚拟主机问题的方案就是允许一份证书为多个主机提供服务。RFC2818 规定了如何使用通配符来完成这项工作。请参见第 9.14 节来了解更为具体的信息。

Netscape 支持通配。实际上 Netscape 的通配方案要比 RFC2818 中所描述的灵活得多。它支持：

- *——任何内容
- ?——任何单个的字符
- 字符串结尾
- [abc]——a、b 或 c
- [A-Z]——A-Z 之间的任何字符

[^ab]——除 a 或 b 之外的任何一个字符

~——不匹配后面的模式

(pattern1|pattern2)——要么匹配 pattern1 要么匹配 pattern2

为了能在模式中指定特殊的字符, Netscape 还使用\作为换码字符。然而, 由于无论什么情况, 域名中都不会出现特殊字符, 所以着实没有这个必要。Netscape 的通配方案在 [Netscape1995a] 中有所描述。

除了通配之外, Internet Explorer 还支持另一种方案。每个证书都允许有多个 Common Name。这样, 在证书中可以包含 foo.example.com 和 bar.example.com 的名字。这种方案的优点是可以为多个不相关的域颁发证书。然而, 不利之处是即便为证书增加了单个新的虚拟主机, 也必须重新颁发。与之对照, 只要能够与模式匹配, 通配就允许证书匹配任意数量的虚拟主机。

9.18 客户端认证

最后, 考虑服务器是否要求客户端使用证书进行认证的判定问题。一些服务器对于通过 HTTPS 获取的所有页面都要求进行客户端认证, 但是更为常见的原因是只对某些页面才要求客户端认证。很难使用 HTTPS 顺利地支持这种策略。

问题还是 SSL 握手必须要在向服务器传送 HTTP 请求之前完成。因而服务器无法知道请求的是什么资源(也就不知道是否要求客户端认证), 直到过了想要要求客户端认证的时机才能知道这些信息。

一种解决办法就是让服务器总是请求客户端认证, 但是没有认证也允许客户端进行连接。那么如果某个请求要求客户端认证而客户端没有提供的话, 就会在那一刻抛出一个错误。尽管从技术上讲行得通, 但是却对用户界面有不良影响: 即当进行认证的时候, 总是向用户弹出使用那个证书的对话框。因此总是要求客户端认证会给所有用户造成不便——即便那些无须进行认证的用户也一样。通常需要对这种方案所造成的不便深思熟虑之后才能否定这种方案。

常用的解决方案就是让服务器与所有的客户端磋商一条普通的 SSL 连接。然后在收到了请求后, 服务器再决定是否需要客户端认证。如果不需要, 服务器就简单地为该请求提供服务。如果需要, 服务器则使用 HelloRequest 请求一次再握手。在第二次握手中, 服务器请求客户端认证。图 9.13 描述了发生这种情况的连接。

```
New TCP connection: romeo(4569) <-> romeo(443)
I 948675468.0084 (0.0005) C>S SSLv2 compatible client hello
  Version 3.0
  cipher suites
    TLS_RSA_WITH_RC4_128_MD5
    value unknown: 0xffff Proprietary Netscape cipher suite
    TLS_RSA_WITH_3DES_EDE_CBC_SHA
    value unknown: 0xffff Proprietary Netscape cipher suite
    TLS_RSA_WITH_DES_CBC_SHA
    TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
```

```
TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
TLS_RSA_EXPORT_WITH_RC4_40_MD5
TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
2 948675468.0139 (0.0054) S>C Handshake
    ServerHello
    session_id[32]=
        22 0f e0 2c 1e 63 a3 b2 cd 68 6a af
        6c f5 bc ff 0d 77 7d d4 dc bd e5 3f
        80 1e da 33 3c 79 f4 0d
    cipherSuite          TLS_RSA_WITH_RC4_128_MD5
    compressionMethod    NULL
3 948675468.0139 (0.0000) S>C Handshake
    Certificate
4 948675468.0139 (0.0000) S>C Handshake
    ServerHelloDone
5 948675468.0281 (0.0142) C>S Handshake
    ClientKeyExchange
6 948675468.0796 (0.0515) C>S ChangeCipherSpec
7 948675468.0796 (0.0000) C>S Handshake
    Finished
8 948675468.0818 (0.0021) S>C ChangeCipherSpec
9 948675468.0818 (0.0000) S>C Handshake
    Finished
10 948675468.0830 (0.0012) C>S application_data
    data: 291 bytes
-----
GET /secure/tmp.html HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.7 [en] (X11; U; FreeBSD 3.4-STABLE i386)
Host: romeo
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,* ,utf -8
-----
11 948675468.0892 (0.0062) S>C Handshake
    HelloRequest
12 948675468.0912 (0.0019) C>S Handshake
    ClientHello
    Version 3.0
    cipher suites
        TLS_RSA_WITH_RC4_128_MD5
        value unknown: 0xffe0 Proprietary Netscape cipher suite
        TLS_RSA_WITH_3DES_EDE_CBC_SHA
        value unknown: 0xffffel proprietary Netscape cipher suite
        TLS_RSA_WITH_DES_CBC_SHA
        TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
        TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
        TLS_RSA_EXPORT_WITH_RC4_40_MD5
```



```

TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
compression methods
NULL
13 948675468.0928 (0.0016) S>C Handshake
ServerHello
session_id[32]=
20 60 4a f0 84 cb b4 7c 7e 9b af d0
3c fe 70 4c 47 58 96 18 2a 02 89 39
94 50 4b a2 77 b0 2c e8
cipherSuite           TLS_RSA_WITH_RC4_128_MD5
compressionMethod     NULL
14 948675468.0928 (0.0000) S>C Handshake
Certificate
15 948675468.1005 (0.0077) S>C Handshake
CertificateRequest
certificate_types      rsa_sign
certificate_types      dss_sign
certificate_authority
30 49 31 0b 30 09 06 03 55 04 06 13
02 55 53 31 13 30 11 06 03 55 04 0a
13 0a 52 54 46 4d 2c 20 49 6a 63 2a
31 13 30 11 06 03 55 04 0b 13 0a 43
6f 6e 73 75 6c 74 69 6e 67 31 10 30
0e 06 03 55 04 03 13 07 54 65 73 74
20 43 41
16 948675468.1005 (0.0000) S>C Handshake
ServerHelloDone
17 948675471.2692 (3.1687) C>S Handshake
Certificate
ClientKeyExchange
EncryptedPreMasterSecret[128] =
CertificateVerify
Signature[128] =
77 db fe 35 67 0d fa 1d 7d ea 2e 70
ae 8f b4 a8 6f 26 91 df 81 1b 8b c6
e1 7a 94 67 ed 9c ad be be 1a 71 74
6e 1b b1 ae c1 9e 26 81 d7 6c 30 ae
67 54 b8 12 f5 cf 0a e2 71 81 aa 0e
8a 14 ee 76 de 39 44 33 9b 6e fd e7
19 51 73 43 67 28 5d bf d3 74 a2 b8
4a f1 32 0a 02 c8 27 b7 bd eb 79 38
ca 3f 91 c7 95 46 b0 c3 32 ff 07 0d
7b 54 28 30 c3 f4 67 f1 f1 58 e9 7c
61 4c a7 28 51 c5 ad 92
18 948675471.2997 (0.0304) C>S ChangeCipherSpec
19 948675471.4797 (0.1800) C>S Handshake
Finished
20 948675471.4811 (0.0013) S>C ChangeCipherSpec
21 948675471.4811 (0.0000) S>C Handshake

```

```
Finished
22 948675471.4848 (0.0037) S>C application_data
data: 423 bytes
-----
HTTP/1.1 200 OK
Date: Mon, 24 Jan 2000 00:57:48 GMT
Server: Apache/1.3.9 (Unix) mod_ssl/2.4.10 OpenSSL/0.9.4
Last-Modified: Sun, 23 Jan 2000 23:08:52 GMT
ETag: "8e469-95-388bSa04"
Accept-Ranges: bytes
Content-Length: 149
Connection: close
Content-Type: text/html

<HTML>
<HEAD>
<TITLE>Test</TITLE>
</HEAD>
<BODY>
<H1>
Client Auth Test Page
</H1>
This page must be fetched with client auth.
<P>
</BODY>
</HTML >
-----
23 948675471.4895 (0.0047) S>C Alert
level      warning
value      close_notify
Server FIN
24 948675471.5060 (0.0164) C>S Alert
level      warning
value      close_notify
Client FIN
```

图 9.13 再握手要求客户端认证

在图 9.13 中，当客户端第一次连接时，服务器完成一次普通的 SSL 握手，跟我们以前见到的握手一样。一旦握手完成，客户端就在记录 10 中发送请求。这同样与图 9.4 中的请求一样，只不过我们请求的是一种不同的资源/secure/tmp.html 而不是/tmp.hrml。

此刻，服务器检查它的存取控制信息并判定请求的资源要求客户端认证，而目前的连接没有进行客户端认证。这样，服务器就需要请求一次再握手才能迫使客户端使用其证书进行认证。这是通过在记录 11 中发送 HelloRequest 来完成的。

HelloRequest 使客户端初始化一次新的握手。尽管从技术上讲并不要求客户端立刻完成此项工作，但是直到服务器给它发送响应为止，客户端都不能做任何事情。服务器发送 HelloRequest 很明显地表示了服务器在等待新握手后才能进行响应，因此在实际应用中客户端会立刻初始化一次新的握手，在记录 12 中发送 ClientHello。

至此，握手就像我们在第 4 章所见到的 SSL 客户端认证握手。然而，要注意的是，整个过程是在第一次握手建立的加密通道上进行的。最后，当握手完成时，记录 18-22，服务器在记录 22 中将其响应发送回来（使用新磋商的会话）。

对性能的影响

注意，当客户端发送其第二个 ClientHello 的时候，它不提供恢复会话。尽管它能够这样做，但是会话恢复与客户端认证不兼容，因此不管怎样，服务器都必须拒绝恢复会话。结果，客户端与服务器需要完成一次全新的握手。这指明了再磋商方案的主要缺点：为了建立连接，它使服务器的工作量增加一倍——而 HTTPS 已经比纯粹的 HTTP 慢很多了（假设密钥为 RSA 密钥，那么客户端的大部分工作都是完成签名，因此相比第一次握手的简单客户端认证多出来的额外握手的额外开销倒不是多么严重）。

其他解决方案

显然这种问题表示 SSL 的行为与常见的 HTTP 套路存在不一致。真想称之为一种 SSL 的设计错误，但是重要的是要知道 SSL 是一种通用协议而 SSL 的固有特性就是必须在传送应用数据的第一段内容之前磋商好加密密钥。要知道请求中有可能包含敏感信息，因此必须对其进行保护。

不管怎样，还是有可能改善这种境况的。首先，SSL 可以让你在恢复的会话上完成客户端认证。这在技术上有相当强的可行性，只是稍微改动一下 SSL 消息的行为而已。有各种可能的改动方案，但是其中一种就是即便在会话恢复的情况下也使用 ServerHelloDone，那么服务器就有机会发送一条 CertificateRequest。这不会导致性能上的问题，因为整组服务器消息都可以在一条握手记录中传输，即一个 TCP 段。

我们不能使用与虚拟服务器一起使用的方案，在那种方案中我们指示客户端想要连接哪个服务器。对于这种问题，客户端必须在线路上以明文形式传输 Request-URI。但是 Request-URI 本身有可能是敏感信息而攻击者未必能够通过嗅探来推导出来。因此无法安全地使用这种方案。

另一种方案就是在 URL 中包含要求客户端认证的指示。这也要求 SSL 具有一种由客户端单方提出客户端认证的机制。然后客户端就可以根据 URL 来提出请求。这种方案存在一种问题，如果服务器改变了存取控制设置，那么它就需要安排对所有相关的引用信息进行改动。然而，如果客户端使用过时的 URL 进行连接的话，总是可以执行再磋商。目前还没有如何完成此项任务的标准，但是要设计一个也是相当简单的。

9.19 Referrer

1997 年，Daniel Klein 观察到头信息 Referrer 可以使一种针对 HTTPS 连接的被动嗅探攻击成为可能。头信息 Referrer 只包含了当用户点击引用当前页面的连接时用户所查看的页面。Referrer 提供了一种方便的让站点查看都有哪些其他的站点指向它的方法，这对于收集各种统计信息而言非常有用。

这种攻击是 HTTP GET 与 referrer 之间进行交互的结果。大家还记得当表单使用 GET 的时候，变元都被放置在 Request-URI 的结尾。这样当用户点击提交的表单所产生的页面上的

链接时，提供给表单的变元就会出现在头信息 Referrer 中。如果链接是 HTTP 链接，那么这些变元就会在下一个请求中以明文进行传输。然而，即便链接是 HTTPS 链接但是它指向用户提交表单之外的站点，那么那个站点就会在头信息 Referrer 中获得这些变元。

如果其中的一个表单字段泄露的是用户的信用卡号码的话，那么可以理解，用户会不安的。应对这种攻击的一般方法是使用 HTTP POST 来提交表单。POST 将变元放在消息体中，这样就可以轻易地挫败这种攻击。

9.20 替换攻击

第 7 章引用完整性要求能够可靠地获取引用信息，基于这个原则我们描述了几种不同的攻击。总体思想就是对包含第一个指向 HTTPS 页面的引用的 Web 页面实施中间人攻击。然后攻击者将 HTTPS URL 替换为一个指向他自己站点的引用。由于 URL 与攻击者的证书相匹配，所以能彻底绕过客户端的引用完整性检查。

当然用户可以（尽管不大可能）检查浏览器显示的 URL，看看是否是所期望的内容。为了应对这种行为，攻击者可以获得一个名字与受害者的 Web 站点看上去相像的 Web 站点（如用 www.foos0ft.com 来代替 www.foosoft.com。只有非常细心的用户才会注意到这种替换）。

依照用户的选择

正如我们先前所讨论的那样，即便对于引用信息不与服务器证书匹配的情况，许多浏览器都给用户提供继续完成连接的机会。我们可以很容易地看出如何利用这种行为来实施攻击。攻击者只需安排使用一份用户有可能认为不合适的证书（例如，上一节所说的 foos0ft 替换），然后接管客户端的 HTTPS 连接。在许多情况下，用户会允许连接继续进行下去，在这种情况下攻击就会成功。

9.21 升 级

RFC2817 [Khare2000] 定义了一种使用 HTTP/1.1 头信息 Upgrade 来完成从 HTTP 到 TLS 上的 HTTPS 的磋商升级的方法。这种方案意图提供一种除流行的 HTTPS 方案之外的另一种方案，并且不要求分配独立的端口。在实际应用中，实质上还没有在 Web 环境中部署这种升级方案。对这种升级方案的大部分兴趣似乎都是对其他使用 HTTP 的协议来说的，部分是因为 IESG 现已抵制对使用独立端口来支持 TLS 的基于 HTTP 的协议的标准化。

客户端请求的升级

可以理解的最简单的情况就是客户端请求升级。客户端简单地在其请求中包含头信息 Upgrade: TLS/1.0，如图 9.14 所示。注意，这个请求包含头信息 Host。如此，服务器就知道客户端打算与之交互的是哪一个主机而且虚拟主机也能在没有多个 IP 地址的情况下顺利工作。注意记号 TLS/1.0 的套路。没有任何 SSL 记号经过标准化，所以不存在正式的升级为 SSL 的方法。

```
GET /foo.bar HTTP/1.1
Host: www.example.com
Upgrade: TLS/1.0
Connection: Upgrade
(blank line)
```

图 9.14 升级请求

如果服务器愿意进行升级，它就会发送一个包含 101 响应码的哑响应(dummy response)，如图 9.15 所示。在发送完哑响应之后，它会立刻开始 TLS 磋商。当客户端收到响应时，又会发送一条 ClientHello。只有在 TLS 连接磋商好之后，服务器才会将对客户端起初请求的响应发送过去。注意服务器的 Upgrade 头信息同时包含 TLS/1.0 和 HTTP/1.1 记号。这些信号是以“从底向上”的顺序来传递的。这样，服务器是在说它将使用 TLS 上的 HTTP。然而，服务器也可以选择不进行升级并简单地响应客户端的消息。

```
HTTP/1.1 101 Switching Protocols
Upgrade: TLS/1.0, HTTP/1.1
Connection: Upgrade
(blank line)
```

图 9.15 接受升级请求的响应

这种方案的一个明显的缺点就是客户端原先的请求是以明文来发送的。这样存在问题，因为客户端的请求有可能包含敏感信息。RFC2817 提供了一种称作强制升级的机制。意思就是客户端给服务器发送一个哑请求，这样实际的请求就不会被泄露了。所使用的请求是一个 OPTIONS 请求，它请求关于服务器的一般信息。规范对此并没有清楚地规定，但是即便对于强制升级，服务器也有可能拒绝。对于那种情况，客户端有机会关闭连接而不会有任何敏感信息传送出去。

服务器请求的升级

服务器也有机会请求 TLS 升级。服务器可以以一种非正式的方式通过在普通的响应中包含 Upgrade 头信息来告知自己支持 TLS。然而，在某些情况下，服务器或许想要强制推行使用 TLS。它可以通过以 426 Upgrade Required 错误代码来响应请求要求 TLS，如图 9.16 所示：

```
HTTP/1.1 426 Upgrade Required
Upgrade: TLS/1.0, HTTP/1.1
Connection: Upgrade
(blank line)
```

图 9.16 强制升级宣传

响应代码 426 本身并不初始化 TLS 握手，而是一种指示客户端必须使用 TLS 才能获取该 URL 的错误。当客户端收到 Upgrade Required 时，它必须使用上面所描述的方法初始化升级。注意，强制升级(advertisement)只有在仅有服务器发送的信息是敏感的情况下才有用。客户端已经通过明文通道传输了自己的数据，因此如果请求包含敏感数据，那么损害就已经造成了。

HTTP 升级确实允许在 HTTP 连接之上磋商 TLS 上的 HTTP。从那种意义上讲，可以认为是成功的。然而，这种方案存在许多问题，从而极大地削弱了它的魅力。首先，它与代理的交互甚至比 HTTPS 还糟。其次它并不提供指示应当采用升级的引用，从而会产生安全和性能问题。

与代理的交互

RFC2616 [Fielding1999] 将头信息 `Upgrade` 定义为一种 `hop-by-hop` 头信息，意思就是说代理行为就像头信息是发给自己的那样。这样做的目的是，即便服务器还没有升级，客户端也应当能够与代理磋商更好的协议（如从 HTTP/1.0 到 HTTP1.1）。

然而，由于客户端想要与服务器磋商一条端到端的 TLS 连接，所以当升级至 TLS 的时候，头信息 `Upgrade` 的 `hop-by-hop` 本质就会产生问题。为了绕过这个问题，客户端需要先使用 `CONNECT` 建立一条到服务器的隧道（tunnel），就像使用 HTTPS 时所做的那样。但这样会更糟，因为它意味着在没有创建隧道的情况下根本就不能升级为 TLS，这极大削弱了代理的价值。更好的解决方案就是使用一个新的端到端的头信息来向客户端宣传支持 TLS。代理可以识别这个头信息（尽管不对它进行修改）并在磋商好使用 TLS 的情况下进入透明传输模式。

没有引用信息

HTTP 升级的另一个问题就是没有办法在引用（reference）中告诉客户端它们能够在连接到服务器的时候进行升级。实际上，规范特别指出 https 的行为完全没有改变：它仍然是指 HTTPS。

省略这项内容造成许多不幸的后果。第一个就是损失了 HTTPS 所提供的起码的引用完整性保护。使用升级的话，客户端甚至无法检测出已经或是没有磋商好使用 TLS。如果客户端必须要使用 TLS，即使在伺机模式下，它惟一的选择也是对每次连接都提出升级。我们先前讲过，这意味着客户端对任何代理都要使用 `CONNECT`，而代理就彻底成了低效率的代理。

更糟的是，除非客户端愿意接受最糟情况下的性能表现，否则客户端的请求就会彻底暴露在外。大家还记得客户端的请求会包含任何表单提交内容的情况，那么考虑一下提交的表单请求信用卡号码的情况。此外，客户端软件无法分辨何时请求中包含有敏感信息，因为一些站点将这些信息直接放在 URL 或浏览器 cookies 中。

因此，为了获得最高级别的安全，客户端总是需要先尝试 `OPTIONS` 请求。然而在不支持 TLS 的常见情况下，这就意味着多了一个没有任何裨益的请求/响应来回。即便在升级成功的情况下，HTTPS 也比升级快，因为它不需要那种哑请求。幸好只会在浏览器第一次连接到一个站点的时候才引入这种开销，如果服务器不支持安全的话——客户端可以将服务器不升级的事实加以缓存。

最后，缺少引用标识使得 HTTP 升级受制于一种降级攻击，在这种攻击中，攻击者将升级记号删除，从而迫使客户端与服务器磋商使用 HTTP。对于 HTTPS 来说就没有这种攻击的可能，因为客户端在解析 `https://URL` 的时候只准备使用 HTTPS。

所有这些问题都可以通过引入一种新的指示期望使用升级 HTTP/TLS 的 URL 方法而轻易得到解决。然而，如果因为这个目的而取代 https 是不明智的，因为那样客户端就无法区分要求 HTTP 升级的服务器与 HTTPS 服务器。



9.22 编程问题

第 8 章讨论了与 SSL 有关的通用编程问题，但是每种协议都是不同的，而使用 HTTPS 编程会提出一些具体问题。本章的剩余部分将讲述其中的两个问题：实现第 9.16 节所讨论的 HTTP CONNECT 方法以及实现对大量 HTTPS 连接的支持，这是任何活跃的服务器都要面对的现实。

9.23 代理 CONNECT

我们在第 8 章所描述的范例程序都是直接与服务器进行连接。正如我们在第 9.16 节所讨论的，HTTPS 客户端常常需要通过代理而不是直接与服务器进行连接。这需要程序员完成一些额外的工作。图 9.17 描述了我们的新版客户端通过代理而不是直接与服务器连接的情景。注意，这个客户端只包含通过代理与服务器连接的代码。现实世界中的客户端应当能够根据设置选项来进行切换。

程序员注释：许多防火墙都只允许 443 端口上的 CONNECT 方法。出于编程的方便，我们使用端口 4433，因为你无须具有根权限就能在端口 4433 上运行服务器。然而，如果你想要尝试一下代理的话，就要在端口 443 上运行测试服务器。

pclient.c

```

1  /* A proxy-capable SSL client.

2  This is just like sclient but it only supports proxies
3  */
4  #include <string.h>
5  #include "common.h"
6  #include "client.h"
7  #include "read_write.h"

8  #define PROXY "localhost"
9  #define PROXY_PORT 8080
10 #define REAL_HOST "localhost"
11 #define REAL_PORT 4433

12 int writestr(sock,str)
13     int sock;
14     char *str;
15     {
16         int len=strlen(str);
17         int r,wrote=0;

18         while(len){
19             r=write(sock,str,len);
20             if(r<=0)

```

```
21         err_exit("Write error");
22         len-=r;
23         str+=r;
24         wrote+=r;
25     }

26     return (wrote);

27 }

28     int readline(sock,buf,len)
29     {
30         int sock;
31         char *buf;
32         int len;
33         {
34             int n,r;
35             char *ptr=buf;

36             for(n=0;n<len;n++) {
37                 r=read(sock,ptr,1);
38                 if(r<=0)
39                     err_exit("Read error");
40                 if (*ptr=='\n') {
41                     *ptr=0;

42                     /* Strip off the CR if it's there */
43                     if (buf[n-1]==''
44                         buf[n-1]=0;
45                     n--;
46                 }

47                 return(n);
48             }
49         }

50         err_exit("Buffer too short");
51     }

52     int proxy_connect(){
53         struct hostent *hp;
54         struct sockaddr_in addr;
55         int sock;
56         BIO *sbio;
57         char buf[1024];
58         char *protocol, *response_code;

59         /* Connect to the proxy, not the host */
```



```

60         if(!(hp=gethostbyname(PROXY)))
61             berr_exit("Couldn't resolve host");
62         memset(&addr,0,sizeof(addr));
63         addr.sin_addr=*(struct in_addr*)hp->h_addr_list[0];
64         addr.sin_family=AF_INET;
65         addr.sin_port=htons(PROXY_PORT);

66         if((sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))<0)
67             err_exit("Couldn't create socket");
68         if(connect(sock,(struct sockaddr *)&addr,sizeof(addr))<0)
69             err_exit("Couldn't connect socket");

70     /* Now that we're connected, do the proxy request */
71     sprintf(buf,"CONNECT %s:%d HTTP/1.0\r\n\r\n",REAL_HOST,REAL_PORT);
72     writestr(sock,buf);
73
74 
```

Figure 9.17 (continued)

```

75     /* And read the response*/
76     if(readline(sock,buf,sizeof(buf))==0, )
77         err_exit("Empty response from proxy");

78     if((protocol=strtok(buf," " ))<0)
79         err_exit("Couldn't parse server response: getting protocol");
80     if(strncmp(protocol,"HTTP",4))
81         err_exit("Unrecognized protocol");
82     if((response_code=strtok(0," " ))<0)
83         err_exit("Couldn't parse server response: getting response code");
84     if(strcmp(response_code,"200"))
85         err_exit("Received error from proxy server");

86     /* Look for the blank line that signals end of header*/
87     while(readline(sock,buf,sizeof(buf))>0) {
88         ;
89     }

90     return(sock);
91 }

92     int main(argc,argv)
93     int argc;
94     char **argv;
95     {
96         SSL_CTX *ctx;
97         SSL *ssl;
98         BIO *sbio;
99         int sock;

100        /* Build our SSL context*/ 
```

```
101     ctx=initialize_ctx(KEYFILE,PASSWORD);  
  
102     /* Connect the TCP socket */  
103     sock=proxy_connect();  
  
104     /* Connect the SSL socket */  
105     ssl=SSL_new(ctx);  
106     sbio=BIO_new_socket(sock,BIO_NOCLOSE);  
107     SSL_set_bio(ssl,sbio,sbio);  
108     if(SSL_connect(ssl)<=0)  
109         berr_exit("SSL connect error");  
110     check_cert_chain(ssl,HOST);  
111     /* read and write */  
112     read_write(ssi,sock);  
  
113     destroy_ctx(ctx);  
114 }
```

pclient.c

图 9.17 支持代理的客户端

写函数

12~27 我们支持代理的客户端需要能够将 CONNECT 请求发送给代理服务器。我们将写代码封装在 writestr() 函数中，这个函数简单地接受任意长度的输入字符串并将其输出到指定的套接字中。

读函数

28~51 我们还需要能够读取代理服务器的响应。响应应当是一系列以回车 (0x0d) 和换行 (0x0a) 分隔的 HTTP 头信息。readline() 函数简单地从网络上将一行信息读取到缓冲区中。这个函数只适用于演示目的。它每次仅从网络中读取一个字节，效率非常低。此外，如果缓冲区太短而不足以包含一行信息的话就会出现灾难性的错误。

连接到代理

52~69 函数 proxy_connect() 意图替换我们在第 8 章所使用的函数 tcp_connect()。开头部分只是重复 tcp_connect() 中的连接代码。然而，这一次不是连接到服务器而是连接到代理服务器。

写请求

70~72 客户端需要完成的第一件事就是构造 CONNECT 请求，其中包含了我们想让代理连接的实际主机和端口。字符串结尾的 “\r\n\r\n” 以空行来指示 HTTP 头信息的结束。然后，我们只需使用 writestr() 将请求发送给代理。

读请求

73~88 一旦将请求发送给了代理，就需要读取返回的响应以确认连接代理成功。我们对响应的第一行（状态行）信息进行解析，确保协议是 HTTP 并且状态码为 200。对剩余的

头信息，我们只是读取后丢掉。同样，代理响应的结尾以一个空行来标识。注意，我们必须将所有的头信息行都读取过来，这样当 SSL 引擎查找 ServerHello 的时候才不会读到任何代理响应。

main()

90~112 我们的 main() 函数与 scclient.c 中的 main() 一样，只不过它在第 101 行调用的是 proxy_connect() 而不是 connect()。一旦建立了通道，客户端就可以完全不理睬代理的存在。

注意，为了使用代理，客户端必须在执行 SSL 握手之前直接读写 TCP 套接字。对于 OpenSSL 来说这样做是可以的，因为 API 要求程序员在开始 SSL 会话之前首先执行 connect()。然而，如果 connect() 是与 SSL 握手集成在一起的，就像在 PureTLS 中那样，这样做就是不可能的。

这个问题在 PureTLS 已知 bug 列表上列出，所以幸运的话，在你读这本书的时候，作者已经解决了这个问题。

9.24 处理多个客户端

真正的服务器需要能够同时处理多个客户端。意思就是说能够在那些客户端之间完成多路读写操作，为那些做好准备的客户端提供服务而忽略那些还未准备好的客户端。此外，它还需要及时地进行处理，不能只为一个客户端提供服务而使其他客户端处于饥饿状态。在 HTTPS 中这尤其成问题，这里任何给定的客户端都可能使用独立的连接来请求数据。

可以使用我们在第 8 章所用的基于 select() 的多路 I/O 技术来完成这项工作，但是并不十分方便。实际上这样编写程序非常困难，部分原因是由于它要求显式地承载大量的状态信息。

要想理解这一点，考虑读取 HTTP 请求的问题。服务器读取数据，但可能只是请求部分的内容是可用的。服务器不能等待剩余的内容，因为它必须为其他的客户端提供服务。因此，它必须将已经读取的数据存储起来，并在客户端最终将那些数据发送过来的时候完成对剩余数据的读取。这要求为那个客户端创建一个显式的上下文对象，并通过某种方式将其与客户端套接字关联起来。这虽不困难，但也不方便。

第二个问题是服务器需要非常小心，不要给单个操作分配过多的时间。设想某个请求要求完成某种操作——如数据库查询——要花费 5 秒时间。显然发送请求的客户端必须等待操作完成，但是同样重要的是服务器不能由于它在为这个特定的客户端提供服务而使其他的客户端全都处于饥饿状态。相反，它必须暂时脱离这个操作并为其他客户端提供服务，然后再恢复操作。这同样表示执行到哪里都要显式的保存操作状态。

此外，程序员必须不断留意各种操作可望花费的时间，以免服务器在完成耗时的操作时错过了为其他客户端提供服务。在使用第三方软件库时这尤其成问题，因为它们或许没有提供暂时脱离 API 调用的功能。

多进程服务器

并不令人吃惊，大多数程序员都选择了一种不同的方案来为多个客户端提供服务：为每

一个客户端连接分配一个自己的执行线索。在 UNIX 系统上，这通常是指一个进程，在 Windows（或 Java）上是指线程。然后调度器负责在服务器进程/线程之间分配 CPU 时间来为每个客户端提供服务并确保每个客户端都得到合理的份额。

这种服务器的传统设计是让一个单一的进程/线程监听客户端的连接。当它接收到一个新连接时，产生一个新的进程/线程来处理该连接。图 9.18 描述了这种服务器的伪代码。

```
server_process() {
    server_socket=create_socket();

    for(;;){
        client=accept(server_socket);
        if(pid=fork()){
            /* Parent process */
            continue;
        }
        else {
            /* Child process */
            serve_request(client);
            exit(0);
        }
    }
}
```

图 9.18 简单的服务器

主服务器进程不停地执行循环。对于循环的每次迭代，它都调用 `accept()`。如果来了一个新的客户端连接，`accept()`就返回与那个新连接对应的套接字。否则进程被置于睡眠状态直到一个新的客户端到来为止。一旦创建了一个新连接，服务器就创建一个新进程对它进行处理。在 UNIX 上，这是通过系统调用 `fork()` 来实现的。在 Windows 上，我们使用 `CreateThread()` 创建一个新线程进行处理，父进程返回循环的顶部并再次调用 `accept()`。

在 UNIX 上，需要父进程使用 `close()` 关闭与客户端连接的套接字以释放服务器资源。任何给定的进程在任何时刻都只允许拥有有限数目的打开文件描述符，如果不关闭套接字，父进程会很快将资源耗尽。这对子进程中的套接字没有影响。

子进程或线程只是为客户端提供服务。因为调度器会自动中断它来为服务器控制的其他线程提供服务，所以子进程/线程只需专注于处理一个客户端。特别的，它可以安全地使用阻塞 I/O 或执行耗时的操作而不用担心暂时脱离那些操作而为其他客户端提供服务的问题。

使用 SSL 的多进程服务器

当为这些多进程服务器中的一个增加 SSL 的时候，我们必须非常小心。大家倾向于将所有的套接字调用均替换为 SSL 套接字调用。这样不行，SSL 握手必须在子进程而不是父进程中执行，否则服务器一次就只能执行一个 SSL 握手。握手代表一种严重的瓶颈制约，尤其当客户端与服务器之间的往返时间很长时更是如此。对于 HTTP 来说，这种瓶颈尤其恼人，因为连接的数量太大了。

为了避免这种瓶颈，服务器需要在父进程中完成 TCP `accept()`，而在子进程中完成 SSL



握手。对于 OpenSSL 来说，要做到这一点相当直接，因为 `SSL_handshake()` 有它自己的调用。图 9.19 描述了我们的 OpenSSL 服务器程序中一种以这种方式执行的主循环变体。

```

24         while(1){
25             if((s=accept(sock,0,0))<0)
26                 err_exit("Problem accepting");
27
28             if(pid=fork()){
29                 close(s);
30             }
31             else {
32                 sbio=BIO_new_socket(s,BIO_NOCLOSE);
33                 ssl=SSL_new(ctx);
34                 SSL_set_bio(ssl,sbio,sbio);
35
36                 if((r=SSL_accept(ssl)<=0))
37                     berr_exit("SSL accept error");
38             }

```

mserver.c

图 9.19 一个简单的基于 OpenSSL 的多进程服务器

这里惟一没见过的代码就是第 27 行对 `fork()` 的调用。它创建了一个新进程（UNIX 上），该进程能够独立执行 SSL 握手。这种代码与 Windows 上的类似，只不过在 Windows 上调用的是 `CreateThread()`，而不是 `fork()`。

由于 PureTLS 模仿的是 Java 的套接字 API，`accept()` 与 SSL 握手都是在同一个 API 调用中发生的，因此没有办法安排在子线程中执行握手。这是 PureTLS 中缺少的一项功能，更好的方案是允许程序员告诉 `accept()` 只创建套接字，然后再单独完成 SSL 握手。

SSL 会话缓存

每当我们使用多进程或多线程服务器时，都需要在会话缓冲策略中提供相应的设施。会话数据是惟一的，因为必须在所有不同的控制线索之间进行共享，而且可以由任何进程/线程进行更新，但并不是在不停地更新。在两种情况下必须对会话缓存数据进行更新。第一，每当创建新会话的时候，都必须将其增加到缓存当中。第二，如果某个会话被标为不可恢复的（如，由于收到了警示），那么就必须从缓存中删去。我们需要确保正常的并发控制，以便同时更新不会导致缓存数据的破坏。

如果我们使用的是线程而不是进程，这些工作就会相当简单。因为所有线程都共享内存，所以只需使用一种内存中的会话缓存表示，然而仍需要在读写的时候对缓存数据进行加锁。至于如何进行加锁则有赖于所使用的系统和线程软件包。Java 提供了可能是完成这项工作的最简单的方法。标注为 `synchronized` 的方法在调用时将会串行地执行。图 9.20 描述了 PureTLS 内部缓存实现的代码片段，完全具备同步处理能力。

```
----- SSLContext.java
438     protected synchronized void storeSession(String key, SSLSessionData sd) {
439         SSLDebug.debug(SSLDebug.DEBUG_STATE, "Storing session under key" + key);
440         session_cache.put(key, (Object)sd);
441     }
442
443     protected synchronized SSLSessionData findSession(String key) {
444         SSLDebug.debug(SSLDebug.DEBUG_STATE, "Trying to recover session
445         using key" + key) ~
446         Object obj=session_cache.get(key);
447
448         if(obj==null)
449             return null;
450
451         return (SSLSessionData)obj;
452     }
453
454     protected synchronized void destroySession(String sessionLookupKey) {
455         session_cache.remove(sessionLookupKey);
456     }
----- SSLContext.java
```

图 9.20 PureTLS synchronized 会话缓存代码

PureTLS 使用散列表 (session_cache) 来存储会话数据，它是 SSLContext 类的实例变量。该代码片段描述了存取缓存至少所需要的三种方法：storeSession() 创建一个条目，findSession() 查找一个条目而 destroySession() 从缓存中删除一个条目。

多进程服务器的会话缓存

如果你的服务器使用多进程而不是多线程，那么情况就会变得困难得多。因为进程并不共享内存，所以不可能简单地使用一个指针来指向某种可由所有进程存取的公共结构。相反，服务器需要依赖操作系统服务来承载这些会话数据。

存在多种可以用来完成此项工作的技术。大多数工具箱都没有提供在进程之间共享会话数据的支持，因此通常需要基于特定的服务器来解决问题。一种方案就是使用一个会话服务器，该会话服务器在内存中存储了所有的会话数据。SSL 服务器通过进程间通信机制来存取会话服务器，通常使用某种类型的套接字。这种方案的主要缺点就是它需要创建某种截然不同的服务器程序来完成这项工作，以及设计 SSL 服务器与会话服务器之间的通信协议。要想确保实现只有授权服务器进程才能获得存取权限的存取控制也是困难的。

另一种方案就是使用共享内存。许多操作系统都提供了允许进程共享内存的方法。一旦分配了共享内存段，就可以像存取普通内存一样来存取数据。不幸的是，这种技术并没有听起来那么有用，因为没有好的从共享内存池中进行分配的办法，所以进程需要分配一块单一的大段，然后静态地寻址内部的内容。更糟的是，共享内存存取不易于移植。

最常用的方案就是简单地在磁盘文件中存储数据。然后，每个服务器进程都能够打开那个文件并从中读写内容。可以使用标准的诸如 flock() 加锁例程来提供同步与并发控制。大家可能会认为在磁盘上存储数据要比共享内存慢得多，但是要记住，大多数操作系统都有磁盘缓存，而这些数据很有可能被放在了这样的缓存中。

这种方案的一个变种就是使用一种简单的 UNIX 键-值数据库（DBM 或其变种）来代替平直文件。这允许程序员简单地创建新会话记录和删除记录，而不用操心在文件中进行存储。如果使用了这样的函数库，就必须小心地在每次写操作后刷新库缓冲区，因为存储在缓冲区中的数据还没有写到磁盘上。

OpenSSL 根本就没有提供对这种会话缓冲的支持，但确实提供了让程序员使用他们自己的会话缓冲代码的钩子函数（hooks）。ApacheSSL（基于 OpenSSL）使用会话服务器方案，mod_ssl（也是基于 OpenSSL）能够支持基于磁盘数据库的方案或共享内存方案，SPYRUS TLSGold 工具箱使用了没有数据库支持的基于文件的解决方案。这些方案的代码都非常复杂，因而我们在此不予描述，但是附录 A 描述了 mod_ssl 的会话缓存代码。

高级服务器配置

早期的 SSL 服务器像我们在本节前面所描述的那样运行。然而，人们发现 UNIX 上的 fork()开销巨大。当 Netscape 开发自己的服务器时，他们增加一项预先启动许多子服务器并安排将客户端分配给那些服务器而不是为每个客户端创建一个新进程的功能。大多数基于 UNIX 的服务器，其中包括流行的免费 Apache 服务器都是这么做的。

后来 Netscape 服务器也是在同一个服务器进程中操作多个线程。然而，他们仍继续使用多服务器进程。因此，尽管这些新配置结构代表了 HTTP 服务器行为的改善，但是它们仍然要求多个进程能够共享会话数据。

9.25 总 结

SSL 设计用来与 HTTP 一起使用，因此我们期望它们之间能够相对协调地一起工作。总的来说是这样的，无处不在的 HTTPS 客户端和服务器就见证了这一点。然而，当我们具体查看它们的交互时，可以看出在许多情况下都使用了笨拙的迂回策略来弥补 SSL 与 HTTPS 的不足。

HTTPS 是占主导地位的用于保护 HTTP 安全的解决方案。它的实现极其简单，这无疑对它的快速部署起到了推动作用。尽管存在许多问题，但是尚且不存在任何真正取代它的理由。

HTTPS 使用主机名来实现引用完整性。HTTPS URL 包含了服务器的期望主机名。客户端应当将主机名与服务器证书中的名字进行核查。没有规定证书中的名字与主机名不匹配的条款。

使用 HTTPS 会破坏代理。当使用 HTTPS 的时候，代理缓存机制彻底无效。此外，代理要求对 CONNECT 方法的特殊支持才能传递 HTTPS。

服务器在对不同的资源请求不同的握手时会遇到麻烦。因为 SSL 握手是在服务器见到请求之前发生的，所以服务器没有办法根据请求的资源对握手提出条件。这会导致开销昂贵的再磋商。

HTTP（HTTP Upgrade）升级解决了一些问题而又引入了其他的问题。HTTP 升级有助于减少端口消耗而且能够与虚拟主机一起正常工作。不幸的是，它与代理的交互甚至比 HTTPS 还要糟糕而且所提供的引用完整性要弱得多。

会话缓存非常关键。因为如此多的 Web 页面都要求使用多条 HTTPS 连接来进行存取，所以 SSL 会话缓存极其重要。因为 Web 服务器经常以多进程来运行，所以服务器必须在进程之间共享缓存数据，从而产生了实现问题。

10

TLS 上的 SMTP

10.1 介绍

我们要考虑的第二种协议就是在 RFC821 [Postel1982] 中描述的 SMTP (Simple Mail Transfer Protocol, 简单邮件传输协议)。SMTP 是用来传输因特网电子邮件 (E-mail) 的基本协议。可以说 E-mail 是独一无二的最重要的因特网应用，所以保护 E-mail 安全的要求尤为迫切。因此 TLS 上 SMTP 的标准(RFC2487 [Hoffman1999a])基本上是与 TLS 的标准 RFC2246 同时发表的。

跟 HTTP 那一章一样，从描述因特网邮件和 SMTP 开始。然后考虑 RFC2487 所使用的解决方案。将会看到使用 TLS 为邮件提供即使是基本的安全服务都非常困难，这并不是 RFC2847 作者的问题。实际上，SMTP 出现这种有趣情况的原因恰恰是因为其安全需求与 TLS 所能提供的服务之间极不协调。最后，简要讨论使用支持 TLS 的 SMTP 过程中涉及的编程问题。

10.2 因特网邮件的安全

想必大多数用户都熟悉 E-mail 的发送过程。用户使用一种可以让他输入收件人和书写信函的程序。在书写完毕之后，他告诉程序将邮件发送出去。通常发送者把邮件发送给一个本地邮件服务器，该邮件服务器再把邮件投递给接收者。用户或许还希望了解消息到达接收者时并未受到破坏。

与之类似，接收者也有一个本地邮件服务器。发送者的邮件服务器将邮件投递给接收者的邮件服务器。然后，接收者的邮件程序从邮件服务器获取邮件并显示给接收者看。接收者的安全需求是消息认证和消息完整性，他想知道是谁发送的消息以及这条消息在传输过程中没有被篡改。

● 基本技术

为了理解电子邮件是如何工作的——以及任何潜在的安全解决方案——重要的是要理解 E-mail 基础设施中所使用的基本技术。需要考虑的三种主要技术为 SMTP、RFC822、MIME

消息格式以及 E-mail 地址。

SMTP

SMTP 是 E-mail 的基本传输协议。它的工作就是在发送者（客户端程序或邮件服务器）与接收邮件服务器之间传输消息。在网络早期存在其他一些完成这项工作的协议，但是现在 SMTP 是惟一主要的邮件投递协议。

RFC822 与 MIME

SMTP 只提供消息传输。实际的消息规格说明在 RFC822[Crocker1982] 中讲述。然而，RFC822 只规定了 ASCII 文本消息的格式并最终增加了多用途因特网邮件扩展（MIME）[Freed1996a、Freed1996b、Morre1996、Freed1996c 和 Freed1996d] 来支持其他形式的内容。

E-mail 地址

为了给某人发送邮件，不但要知道接收方的名字还要知道他们的 E-mail 地址。这种地址不仅告诉发送者与哪个服务器进行连接，还告诉接收服务器在取得邮件时应该怎么做。

现实的考虑

我们还需要关心一些前面没有明显提到的邮件环境特性。其中的四种特性实际上与安全有着特殊关系：邮件中继（mail relaying）、虚拟主机（virtual host）、MX 记录（MX record）和客户端邮件存取（client mail access）。

邮件中继

邮件不是直接在两个服务器之间进行传输的，在途中经过一系列邮件中继的情况相当常见。显著的安全问题就是在通过这些中继时保持安全。

虚拟主机

与 Web 服务器一样，ISP 为人们的个人域或小的公司域维护邮件服务器。这样可以允许每个用户甚至在没有直接与因特网相连的情况下也能拥有自己的域和 E-mail 地址。我们这里所面临的问题与处理 HTTPS 时一样：即要确保接收服务器向发送者提供恰当的证书。

MX 记录

使用邮件中继可以给当前没有连接到因特网上的机器发送邮件。发送者只需连接到正确的中继并依靠这个中继来投递邮件。邮件交换器（Mail Exchanger，MX）记录用来标识要连接的中继。

客户端邮件存取

当过去大多数邮件都是在 UNIX 机器上读取的时候，邮件服务器只是将邮件存放到磁盘上，用户的客户端从磁盘上读取邮件。在现代网络环境中，大多数邮件客户端都安装在 PC 上，而且不能直接存取邮件服务器的文件系统。那些客户端使用各种网络协议来存取邮件服务器并收取邮件。

安全上的考虑

一旦我们理解了因特网邮件环境，就可以考虑如何提供相应的安全服务。跟以前一样，需要考虑第 7 章所讨论的重要问题：协议选择、客户端认证、引用完整性以及连接语义。

10.3 因特网消息传递概述

任何因特网邮件系统至少都由两种类型的实体构成：客户端（称作邮件用户代理 mail user agents, (MUA)）和服务器（MTA (mail transport agents, 邮件传输代理)）。邮件客户端就是用户与之交互的程序，它显示邮件并允许用户书写和发送新邮件。然而，大多数客户端都与邮件接收或邮件发送没有任何关系。这些工作都是通过 MTA 来完成的。

早先讲过，邮件不是直接从发送者到达接收者的。相反，它沿途要经过一系列服务器。图 10.1 描述了一个简单地投递一封 E-mail 的邮件系统。位于左边的发送客户端给右边的接收客户端发送一条消息。用户书写一条消息，然后他的邮件客户端使用 SMTP 将其传送给其本地的邮件服务器，本地邮件服务器再将消息传送给接收者的本地邮件服务器（也是使用 SMTP）。

实际上，让邮件客户端以某种除 SMTP 以外的其他方法将邮件传递给第一个步跳 (hop) 服务器相当常见。在 UNIX 系统上，邮件客户端常常将 MTA 作为一个程序来调用（通常为 sendmail）。当 MTA 为 Microsoft Exchange 的时候，客户端常常使用邮件应用 MAPI (Mail Application Programming Interface, 编程接口) 来与 MTA 进行交互。

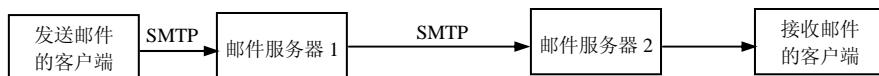


图 10.1 一种简单的邮件系统

从接收者的本地邮件服务器到接收者这最后一个环节不是通过 SMTP 来实现的。存在许多种读取邮件的协议，其中包括邮局协议 (POP) [Myers1996] 和因特网邮件存取协议 (Internet Mail Access Protocol, IMAP) (在 [Crispin1996] 中定义)。在许多 UNIX 系统上，邮件服务器都位于接收者所使用的同一台机器上，接收者可以简单地从磁盘上读取邮件消息，甚至可以使用 HTTP 来读取邮件。不管是哪一种情况，这一环都不使用 SMTP。

考虑接收者的邮件服务器暂时脱机会发生什么情况。发送者的本地邮件服务器就会临时保留这封邮件并频繁地尝试发送，直到将消息投递出去或超时为止。类似的，为了让邮件投递到他的本地服务器，接收者根本不需要处于联机状态。本地邮件服务器只需存储邮件并等待接收方将其取走即可。这样，因特网邮件系统就可以以一种 HTTP 无法胜任的方式恰当的处理断开的连接。为了适应没有永久因特网连接的用户，这显然是必须的。这种风格的消息传递就被称作存储转发 (store-and-forward)。

10.4 SMTP

与 HTTP 一样，SMTP 概念上是简单的，但在实际操作中多少有些复杂。发送代理通过 TCP 端口 25 连接到接收代理。然后发送与接收代理通过一系列请求与响应确定发送与接收参数，其中包括发送者与接收者。最后一旦确立了所有的参数，发送代理就将消息传送给服务器。

所有通信都按部就班地完成。发送代理发送一条命令，接收代理发送一条响应。命令由单行信息构成，以 CRLF 来终止。而响应由三位数的数字（应答码）以及跟在后面的解释文本组成。响应可以有多行，除最后一行外，每行都在应答码后面带有一个连字符，用来表示其后面还有信息行。图 10.2 描述了一个简单的 SMTP 范例。

```
New TCP connection: romeo(3094) <-> speedy(25)
I 949464306.6034 (0.0966) S>C
data: 101 bytes
-----
220 speedy.rfm.com ESMTP SendWhale 8.9.1/8.6.4 ready at 28.8K Tue, 1 ↴
Feb 2000 20:05:03 -0800 (PST)
-----

2 949464306.6036 (0.0969) C>S
data: 21 bytes
-----
EHLO romeo.rfm.com
-----

3 949464306.6089 (0.1022) S>C
data: 173 bytes
-----
250-speedy.rfm.com Hello romeo.rfm.com [216.98.239.227], pleased to ↴
meet you
250-EXPN
250-VERB
250-8BITMIME
250-SIZE
250-DSN
250-ONEX
250-ETRN
250-XUSR
250 HELP
-----

4 949464306.6094 (0.1026) C>S
data: 41 bytes
-----
MAIL From:<ekr@romeo.rfm.com> SIZE=224
-----

5 949464306.8004 (0.2937) S>C
data: 39 bytes
-----
250 <ekr@romeo.rfm.com>... Sender ok
-----

6 949464306.8005 (0.2938) C>S
data: 24 bytes
```

RCPT To:<ekr@rtfm.com>

7 949464306.8302 (0.3234) S>C
data: 36 bytes

250 <ekr@rtfm.com>... Recipient ok

8 949464306.8303 (0.3236) C>S
data: 6 bytes

DATA

9 949464306.8570 (0.3502) S>C
data: 50 bytes

354 Enter mail, end with "." on a line by itself

10 949464306.8578 (0.3510) C>S
data: 445 bytes

Received: from romeo.rtfm.com (localhost [127.0.0.1]) by romeo.rtfm.com (8.9.3/8.6.4) with ESMTP id UAA46227 for <ekr@rtfm.com>; Tue, 1 Feb 2000 20:05:06 -0800 (PST)

Message-ID: <200002020405.UAA46227@romeo.rtfm.com>

To: ekr@rtfm.com

Subject: Test message

Mime-Version: 1.0 (generated by tm-edit 7.108)

Content-Type: text/plain; charset=US-ASCII

Date: Tue, 01 Feb 2000 20:05:06 -0800

From: Eric Rescorla <ekr@rtfm.com>

This is a test

11 949464306.9991 (0.4924) C>S
data: 3 bytes

.

12 949464307.0654 (0.5587) S>C
data: 44 bytes

250 UAA09843 Message accepted for delivery



```

-----  

I3 949464307.0864 (0.5797) C>S  

data: 6 bytes  

-----  

QUIT  

-----  

I4 949464307.1127 (0.6059) S>C  

data: 40 bytes  

-----  

221 speedy.rfm.com closing connection  

-----  

Client FIN  

Server FIN

```

图 10.2 一个简单的 SMTP 范例

当发送者第一次与接收者进行连接时，接收者会发送一条带有标识信息的欢迎信息，其中通常包括主机名和软件版本号。第 1 段显示服务器为 Sendmail 8.9.1。

发送方发送的第一条命令应当是 HELO 或 EHLO。RFC821 中所描述的传统 SMTP 使用 HELO 命令，而 RFC1425 引入了许多 SMTP 扩展并使用 EHLO 来表示发送方遵从那种规范。其中的任何一个命令都应当包含发送方的主机名（就是它所认为的）。在这个范例中，发送方的主机名为 romeo.rfm.com。

接收方使用它自己的主机名以及一系列它所支持的扩展来响应 EHLO。因此，可以看出这个服务器支持许多扩展，其中包括 EXPN、VERB 和 8BITMIME。

连接的余下部分也以类似的方式进行。发送方使用 MAIL 命令标识发送消息的用户，使用 RCPT 标识消息的接收者。在这两种情况中，接收方服务器都是用 250 状态码来响应，用于表示一切顺利。

一旦发送方与接收方受到承认，发送代理就可以使用 DATA 命令来发送实际消息。接收方以 354 来响应，表示发送方应当继续发送消息。实际的消息出现在第 10 段中。SMTP 使用只包含单个点（即.CRLF）的一行信息来表示数据结束，可以在第 11 段看到那种信息。在第 12 段中，接收方表示它接受了消息并准备进行投递。

此刻，发送代理就可以反复执行 MAIL、RCPT、DATA 序列来发送另一条消息。然而，这里的跟踪信息显示没有其他要发送的消息，因此发送代理使用 QUIT 关闭连接。

10.5 RFC 822 和 MIME

RFC 822 风格的消息非常简单。它们由许多头（header）信息行组成，后面还跟有一个消息体。头信息行只不过是在头信息的名字后面跟有一个冒号和值，就像下面这样：

From: ekr@rfm.com

消息体内容在很大程度上是格式随意的。RFC 822 限制消息体数据只能为 ASCII 字符，

但是 MIME 使二进制数据成为可能。图 10.3 描述了一个简单地从作者发送给作者自己的范例邮件消息。注意，两行长的 Received 头信息行分成多行放置。如果一个头信息行以一个空白符开头，那么它就会自动连结到前一行的末尾。

```
From ekr@Network-Alchemy.COM Tue Feb 8 09:25:10 2000
Received: from Hydrogen.Network-Alchemy.COM (Hydrogen.Network-Alchemy.COM
[199.46.17.130]) by speedy.rfm.com (8.9.1/8.6.4) with ESMTP id JAA04954
for <ekr@rtfm.com>; Tue, 8 Feb 2000 09:25:09 -0800 (PST)
Received: (from ekr@localhost) by Hydrogen.Network-Alchemy.COM
(8.8.7/8.8.8) id JAA05750 for ekr@rtfm.com; Tue, 8 Feb 2000 09:23:52
-0800 (PST) (envelope-from ekr)
Date: Tue, 8 Feb 2000 09:23:52 -0800 (PST)
From: Eric Rescorla <ekr@Network-Alchemy.COM>
Message-Id: <200002081723.JAA05750@Hydrogen.Network-Alchemy.COM>
To: ekr@rtfm.com
Subject: Test message
```

How about this?

图 10.3 一封 E-mail 消息

这种格式应当同所讨论的 HTTP 请求与应答的格式非常相似。这并不奇怪，因为 HTTP 消息详尽仿效的是 E-mail 消息。大家请看消息中的 Received 行，SMTP 服务器沿邮件路径给消息增加了新的头信息行。实际上，只有最后两个头信息行 To 和 Subject 是由客户端的 MUA 创建的。随着每个服务器对消息的处理，它有可能将自己的头信息行增加到消息前面。

这样，当客户端的 MUA 将消息传递给它位于 hydrogen.network-alchemy.com 的本地 MTA 时，服务器自动增加 Date 字段来表示它发送消息的时间，From 字段指示发送方是谁，Message-Id 字段提供这条消息的跟踪编号（tracking number）。注意，对 From 字段的处理并不一致，一些邮件客户端会自己产生这种信息而服务器通常不会改变它。

消息顶部的 From 字段以及第一个 Received 标题是由位于 speedy.rfm.com 的接收者的邮件服务器增加的。这一行只是由某些 UNIX 发件程序作为“mbox”存储格式的一部分而增加的。

Received 信息行

Received 行提供了记录消息通过邮件系统采用的路径。每个服务器都在处理邮件时将自己的 Received 行增加到消息的前面。

通过检查这些标题，我们可以看出原始消息是由位于机器 hydrogen.network-alchemy.com 上的用户 ekr 发送的。该用户直接调用邮件服务器而不是通过 SMTP 与邮件服务器连接（实际上用户使用的是 Mail 命令，这个命令会调用 Sendmail）。hydrogen.network-alchmey.com 直接与 speedy.rfm.com (rfm.com 域的邮件服务器) 进行连接并投递消息。而 speedy 将消息写到磁盘上，准备让用户来读取。

发送方的身份

如你所见，在消息中存在两种表示发送方是谁的指示。消息顶部的 From 行不是头信息行，它没有冒号，且是由 Sendmail 在将数据写到磁盘上之前增加的。其他的发件程序可能



使用不同的格式。这一行可以被认为表示传输者身份（transport identity）的。该身份就是发送邮件的 SMTP 代理在 MAIL 命令中所使用的身份。消息中的 From 行应当被看作消息身份（message identity），它是消息的发送者在消息中设置的身份标识，而 SMTP 根本就不关心这种身份。

这里，发送代理是作为程序而不是通过网络来调用的，因此它能够验证用户的身份并产生恰当的 MAIL 命令。与之对照，用户可以编写一个新的 From 标题并简单地让发送代理承载这种信息。因此，From 标题是不受信的。我们将会在后面看到，MAIL 命令的内容也是不可信的。

10.6 E-mail 地址

因特网邮件所使用的身份形式为 E-mail 地址。使用它来描述如何到达某个人（一种引用）以及作为一种发送方身份的指示。因此，若想理解如何才能安全地完成该工作，就需要熟悉地址是如何工作的。

因特网邮件寻址在某些情况下极端复杂。特别的，地址可以包含各种各样的路由信息，这些信息告诉服务器沿途将邮件发送给下面的那个服务器。在使用 DNS 和 MX 记录以前，经常使用在“叹号路径”中包含的路由信息来指定地址，如下所示：

```
foo!bar!ekr
```

上面的地址表示给主机 foo 发送邮件，而 foo 应当将邮件发给主机 bar 上的用户 ekr。幸运的是，在现代因特网环境中，这种东西大部分都消失了，地址可以简单地描述如下：

```
local-part@domain
```

domain 可以被简地认为是目标机器的 DNS 域名。我们将在第 10.9 节讨论邮件中继和 MX 记录的时候看到，严格来讲这样是不正确的，但却是一种好的近似说法。

可以广泛地认为 local-part 是邮箱。同样，严格来讲这也是不正确的，但它也是一种有益的近似。它当然有可能不与任何实际用户对应。例如，它有可能对应服务器上运行的邮件列表，或指向另一个用户的别名。例如，机器一般都有 postmaster 地址，但是通常没有 postmaster 用户，而是作为指向管理员账户的别名。图 10.4 描述了一个 E-mail 地址范例，包含指向 local-part 和 domain 的指针，它指向位于 example.com 域的 postmaster 邮箱。

```
postmaster@example.com
  _____   _____
  |       |   |
postmaster   example.com
  local-part    domain
```

图 10.4 一个 E-mail 地址

10.7 邮件中继

在我们到目前为止所讨论的例子中，消息都是直接从发送者的本地邮件服务器发送到接收者的本地邮件服务器。然而在许多情况下，其间会有许多其他的服务器。由于网络分段，特别是防火墙的原因，这种情况经常出现。

在第 7 章已讲过，许多防火墙都只允许代理的形式通过。这就意味着防火墙需要有某种

类型的 SMTP 代理来代理传输数据。与 HTTP 不同，SMTP 代理是双向的。它从外界接收消息并发送给内部的 SMTP 服务器，同时从内部将消息传送给外面的服务器。因此在发送方与接收方均位于防火墙之后的环境中，任何消息都至少要通过四个 SMTP 服务器：发送者的本地服务器、发送者的防火墙、接收者的防火墙以及接收者的本地服务器。

每次使用邮件中继时所要面对的安全问题是确保恶意中继不能读取、修改和伪造邮件。对于 HTTPS 来说，我们通过简单地将中继隔离于回路之外而解决了这个问题。CONNECT 方法允许我们在客户端与服务器之间创建端到端的连接。因为 SMTP 是存储转发性质的，所以这种方案是不可行的。如果使用 SSL，必须安排只通过受信的中继进行连接。

组织服务器

公司有多台具有组织服务器（organizational server）的情况也相当常见，这样就可以在不同的服务器上管理同一个公司的不同分支机构。例如，会计可以在一台服务器上，而工程在另一台服务器上。工程与会计服务器为了满足各自不同的需要甚至可以运行在不同的操作系统之上，工程师在带有 Sendmail 的 UNIX 上工作，而会计在带有 Exchange 的 NT 上工作。

另一种使用组织服务器的原因就是用于分担负载。E-mail 常常占据相当数量的磁盘空间，而一个大型的组织可以选择使用许多具有较小磁盘空间的机器而不是使用具有大量磁盘空间的单台机器。这种方案所带来的好处就是可以避免整个公司存在单点失效（Single point of failure）。

然而，大家经常希望将这种复杂性隐藏起来，这样用户就可以只有一个邮箱，能够从任何一台他们实际使用的组织服务器上进行存取。要实现这种功能，所有的邮件都必须经过某种中心服务器，由它来将邮件分发给各个组织服务器。图 10.5 描述了这样一种设置。

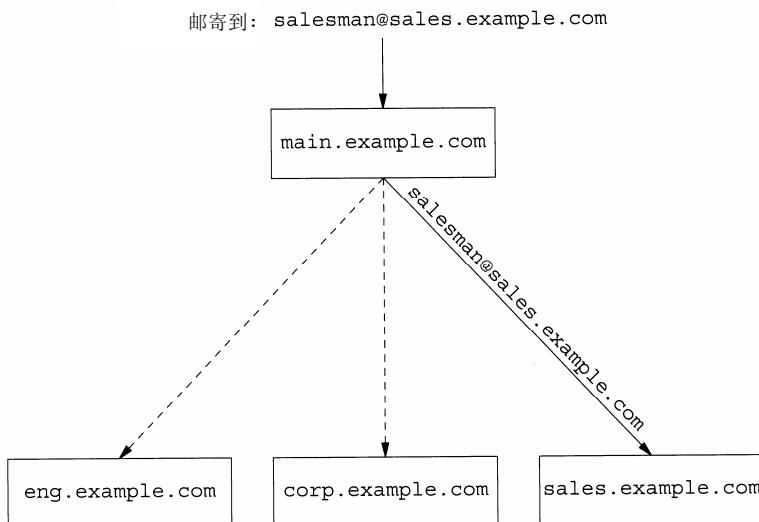


图 10.5 组织服务器

在图 10.5 中，我们为 ExampleCo(example.com 域)公司设置单台中心服务器。ExampleCo 有三个分支，工程、公司管理和销售，每个分支都有其自己的 SMTP 服务器。当一封发给 salesman 的消息过来时，它首先到达 main.example.com，它判定应该将邮件中继给

`sales.example.com`。

让所有的组织服务器都拥有同一个域的缺点就是要求在服务器之间同步用户名。为了避免这种工作，一些公司让组织服务器为独立的域提供服务。这种系统中的典型 E-mail 地址为 `foo@corp.example.com`。这样的组织仍然需要选择使用单个服务器来从外界收取所有的邮件，这纯粹是出于管理和防火墙的考虑。

聪明主机

在组织具有自己的邮件服务器的环境中，最好使用单台服务器来处理传送给外界的所有通信。组织服务器被简单配置成将所有的消息都中继到主服务器上去。这台主服务器经常被称作聪明主机（Smart Host）。

使用聪明主机的优势就是所有复杂的邮件配置都可以在单个管理节点上完成。组织服务器配置起来非常简单，因为路由非常简单：它们总是路由到聪明主机。在现代因特网环境中，邮件路由一般都相当简单，因此并不像过去那样成问题。然而，智明主机仍然相当常见，因为它们允许从单一节点管理邮件的投递。

图 10.6 描述了我们在图 10.5 中所展示的同样的服务器结构，但是这一次是从 `eng.example.com` 的用户给地址 `recipient@rtfm.com` 发送一条消息。机器 `main.example.com` 为智明主机，因此 `eng` 将消息发送给 `main`，而 `main` 安排将消息发送给 `rtfm.com`。

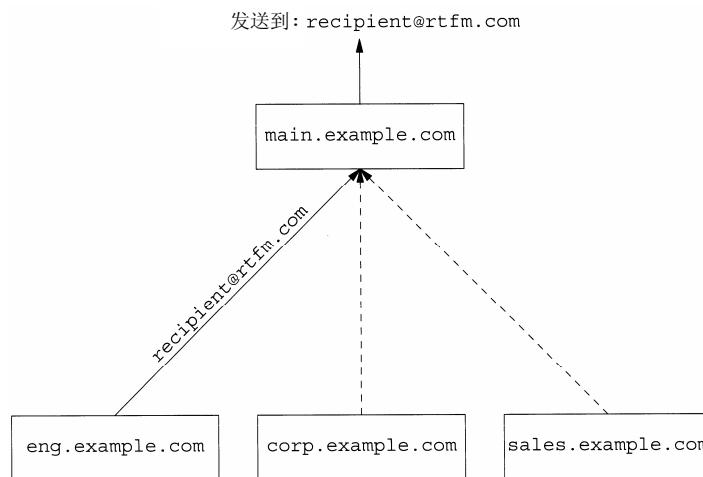


图 10.6 智能主机

开放中继

在因特网流行之前，没有全天候到网络其他部分连接的组织极为常见。它们偶尔连接两台邮件服务器，这些服务器接受网络邮件并将邮件中继到外面的世界。由于历史的原因和善意的行为，人们一般简单地将自己的发件程序配置成接受来自任何主机且发往任何主机的邮件通信，并尽力将它们投递出去。这种配置称作开放中继（Open Relay）。

现在这种配置已经变得非常稀少了（尽管还存在一些，尤其是在美国之外），但是许多邮件代理仍然被配置为开放中继。近来由于滥发邮件（spam）（未经请求的大量商业 E-mail），即垃圾邮件的增多，这种配置越发成为一种问题。

一种常见的用于阻止滥发邮件的策略就是简单地拒绝从滥发者网络过来的 SMTP 连接。许多组织都维护包含已知滥发者的清单，你可以将发件程序配置成阻止这些清单上的发件者。滥发者通过无辜的开放中继发送邮件加以应对。管理员所面临的难处就是选择拒绝源于合法机器的邮件还是选择接受通过那些机器中继过来的滥发邮件。那些选择拒绝的管理员可以订阅开放中继邮件列表，就像他们订阅已知滥发者列表一样。

这里的安全问题就是允许为授权机器提供中继——即便这些机器不是你的网络的一部分——但是禁止为非授权机器提供中继。SSL 允许我们对那些希望以我们的服务器为中继的发送者提供高强度认证，并只为那些认证为授权机器的发送者中继邮件。

10.8 虚 拟 主 机

正如先前所指出的，ISP 的顾客希望拥有自己 E-mail 域的情况相当常见。这些顾客常常通过拨号线路进行连接，因此他们没有与因特网的全时连接。在这种情况下，ISP 替他们接收邮件而顾客则安排远程读取邮件。因为 SMTP RCPT 命令标识邮件接受者的域名，因此支持使用 SMTP 的虚拟主机很容易。然而，与 HTTP 一样，需要关心的是当给这种组合增添全时也能工作的一样好。

10.9 MX 记 录

到目前为止，我们避免谈及为了将邮件投递给某个接收者，服务器是怎么知道连接到哪个服务器的问题。邮件路由从概念上讲是简单的，但是有一些需要考虑的微妙之处。

一般来讲，邮件服务器由两种类型，笨（dumb）服务器与聪明服务器构成。笨服务器能够识别目的为其邮箱的邮件并进行投递，而将别的消息简单地转交给聪明主机来投递。因为笨服务器不存在路由问题，所以只要将它们配置为知道聪明主机在哪儿就行了。

有人或许认为聪明主机的行为也相当简单：查找 E-mail 地址的域名部分并连接到给定的服务器。然而请考虑虚拟主机的情况。具体来说，假定我们有单一的一台 ISP 服务器 mail.exampleisp.com，它管理虚拟域 examplecustomer.com。一种方法就是让一个 DNS A 记录从 examplecustomer.com 指向 mail.exampleisp.com 的 IP 地址。这暗指其他诸如 Telnet 的服务对 examplecustomer.com 来说也是可用的，而事实上那台机器根本就不在网络上。

更好的方案就是使用特别为邮件设计的记录，即 MX (Mail Exchanger，邮件交换器) 记录。一条 MX 记录告诉发送者 MX 记录的目标为那个请求域的邮件中继。可以有多条 MX 记录，如图 10.7 所示。

```
microsoft.com    preference = 10, mail exchanger = mail2.microsoft.com
microsoft.com    preference = 10, mail exchanger = mail3.microsoft.com
microsoft.com    preference = 10, mail exchanger = mail4.microsoft.com
microsoft.com    preference = 10, mail exchanger = mail5.microsoft.com
microsoft.com    preference = 10, mail exchanger = mail1.microsoft.com
```

图 10.7 microsoft.com 的 MX 记录

图 10.7 描述了 microsoft.com 域的 MX 记录，这是通过 nslookup 获取的。这里有五条记录，从机器 mail1~mail5。注意，每条记录都有一个优先值（preference）。可以用它来指示发送方应当依次尝试这些邮件服务器。因此，可以优先使用某些 MX 记录将通信数据导向特定的链接。如果使用 A 记录的话，则要想做到这一点是不可能的，因为它们没有优先值。

不幸的是，这种邮件中继对安全提出了一种挑战。由于我们不能信任 DNS，所以并不知道获得的 MX 记录是不是伪造的。因此就像 CNAME 那样，当进行连接时，我们需要根据服务器的证书检查接收者的域（如 microsoft.com），而不是邮件交换器（如 mail1.microsoft.com）的名字。在后面我们将会看到，这样做提出了管理上的问题。

10.10 客户端邮件存取

尽管许多客户端与它们的本地邮件服务器处于同一台机器上，可以从磁盘上读取数据，但许多都是通过诸如 POP 或 IMAP 这样的网络协议来存取邮件的。显然，必须建立起这些连接。保护这些协议的安全超出了本章的范围，但要指出的一点是，除非客户端安全地获取邮件，否则保护邮件服务器之间的链接毫无用处。自然，客户端/服务器通信常常是通过 SSL 来保护的，具体内容参见 [Newman1999] 中的描述。与往常一样，每种协议都有一些在使用 SSL 时必须应对的微妙之处。在复习完第 7 章的指导之后，研究一下这些标准是有指导意义的。

10.11 协议的选择

正如我们在图 10.2 中所看到的，SMTP 已经具有一种扩展机制。因此，使用这种扩展来实现升级磋商策略非常自然。此外，这种方案更符合系统管理员的口味。由于 SMTP 通信数据几乎总是需要通过防火墙才能进行投递，因此不用打开另一个端口就能对 SMTP 通信数据加以保护是非常有吸引力的。

10.12 客户端认证

在邮件环境中，我们实际关心两种客户端认证：认证中继（authenticated relaying）和源发者认证（originator authentication）。认证中继的目的是允许中继但限制服务为特定的发送服务器。源发者认证的目的是给接收者提供有关是谁发送消息的信息。正如我们在第 10.5 节所指出的，这是 E-mail 的一项重要需求，接收者希望能够对发送者进行认证。

不幸的是，SMTP 的本质使得源发者认证非常困难。因为发送服务器通常不是由客户端而是由一个系统进程来控制的，所以不可能掌握用户的密钥资料。因此，我们可能提供的最好服务就是对发送服务器进行认证。发送服务器当然能够断言客户端的身份，但是接收者必须信任发送服务器。我们将在第 10.23 节看到，当考虑客户端与服务器之间可能的中继时，情况甚至变得更糟。

10.13 引用完整性

正如我们在先前所讲到的，因特网邮件使用 E-mail 地址作为引用内容。当关注引用完整性时，我们关心的是将邮件投递给正确的接收者。但不幸的是，接收邮件服务器有可能是某个不在用户直接控制下的邮件服务器。用户的机器甚至在邮件投递时根本就不在网络上。与源发者认证一样，我们立刻被迫屈从于使用某种不太如愿的措施。在最好的情况下，引用完整性可以意味着我们将邮件投递给了正确的服务器，而且我们信任该服务器能够将邮件安全的投递给正确的用户。

10.14 连接语义

由于发送给典型邮件服务器的邮件消息数量比发送给典型 Web 服务器的 HTTP 请求数量要少得多，所以我们不必过分担心会话恢复问题。我们仍然要操心如何阻止截断攻击，但是 SMTP 面向命令的本质在这里符合我们的愿望。何时应当关闭连接以及何时不应当是相当明了的。

10.15 STARTTLS

现在我们从设计的角度理解了 E-mail 的安全问题，接下来准备讲述 STARTTLS。标准的解决 SSL 上 SMTP 的方法在 RFC 2487 [Hoffman1999a] 中描述。此外我们还准备评估 STARTTLS，满足在本章前面部分所讨论的需求的效果如何。

我们从研究一个使用 STARTTLS 的简单事务开始。这可以让我们看到如何使用 STARTTLS 扩展从 SMTP 升级至 TLS 上的 SMTP。和往常一样，我们将讨论如何处理连接关闭。

一旦对 RFC 2487 所规定的行为有了一个清晰地认识，我们就可以研究它与现实世界中的一些需求的交互情况。特别的，我们将查看要求使用 TLS 来传输消息的可能性以及 STARTTLS 与虚拟主机之间的影响。

最后，我们还将讨论 STARTTLS 实际所能提供的安全保护。我们将研究用来描述消息路径中每一环使用何种保护的安全指示器（security indicators）的问题。此外还将考虑 SMTP 提供的认证中继、源发者认证以及引用完整性的问题。

10.16 STARTTLS 概述

RFC 2487 大约是在 TLS 规范（RFC 2246）发表的同时发表的。尽管 SSL 上 SMTP 的应用已经有一段时间了，但 RFC 2487 还是编纂了它的使用规则。独立端口策略（SMTPLS）的应用时间不长，还没有广泛地部署，随着 STARTTLS 的出现而逐渐被废弃。

STARTTLS 使用 SMTP 扩展机制，在 RFC 1869 [Klensin1995] 中记述的。服务器提供 STARTTLS 作为它的扩展（extension）之一，而客户端可以使用 STARTTLS 命令来调用它。之后客户端与服务器进入 TLS 握手阶段。一旦握手完成，它们就使用 EHLO 像什么都没有发生过一样而重新开始。

图 10.8 中的跟踪信息描述了使用 STARTTLS 执行升级的 SMTP 连接的开始部分。这些信息是用流行的、支持 STARTTLS 的补丁程序之后的邮件程序 qmail 来产生的，不过也可由其他邮件程序产生类似的跟踪信息。

```
New TCP connection: romeo(2515) <-> mike(25)
I 950072479.5489 (0.0225) S>C
data: 25 bytes
-----
220 mike.rfm.com ESMTP
-----

2 950072479.5490 (0.0226) C>S
data: 21 bytes
-----
EHLO romeo.rfm.com
-----

3 950072479.5495 (0.0231) S>C
data: 63 bytes
-----
250-mike.rfm.com
250-PIPELINING
250-STARTTLS
250 8BITMIME
-----

4 950072479.5496 (0.0232) C>S
data: 10 bytes
-----
STARTTLS
-----

5 950072479.5536 (0.0272) S>C
data: 19 bytes
-----
220 ready for tls
-----

6 950072479.6269 (0.1004) C>S SSLv2 compatible client hello
Version 3.1
cipher suites
    TLS_DHE_DSS_WITH_RC4_128_SHA
    TLS_DHE_DSS_WITH_RC2_56_CBC_SHA
    TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
    TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
```

Cipher suite list trimmed

7 950072479.7353 (0.1084) S>C Handshake
ServerHello
session_id[32] =
73 bf d5 04 ac 45 c1 7e a6 04 82 d0
ae 5c ed db f8 8f b4 2b f5 9e d0 6c
52 08 87 20 16 0e 4d 54
cipherSuite TLS_RSA_WITH_3DES_EDE_CBC_SHA
compressionMethod NULL
8 950072479.7353 (0.0000) S>C Handshake
Certificate
9 950072479.7353 (0.0000) S>C Handshake
ServerHelloDone
10 950072479.7380 (0.0027) C>S Handshake
ClientKeyExchange
11 950072479.7380 (0.0000) C>S ChangeCipherSpec
12 950072479.7380 (0.0000) C>S Handshake
Finished
13 950072479.7702 (0.0321) S>C ChangeCipherSpec
14 950072479.7702 (0.0000) S>C Handshake
Finished

Client restarts SMTP session with a new EHLO

15 950072479.7707 (0.0004) C>S application_data
data: 21 bytes

EHLO romeo.rtfm.com

16 950072479.7714 (0.0007) s>c application_data
data: 49 bytes

250-mike.rtfm.com
250-PIPELINING
250 8BITMIME

17 950072479.7716 (0.0002) C>S application_data
data: 32 bytes

MAIL FROM:<ekr@romeo.rtfm.com>

18 950072479.7723 (0.0006) S>C application_data
data: 8 bytes

250 ok

19 950072479.7724 (0.0001) C>S application_data
data: 29 bytes

RCPT TO:<ekr@mike.rtfm.com>



```
20 950072479.7731 (0.0006) S>C      application_data
    data: 8 bytes
-----
250 ok
-----
21 950072479.7732 (0.0001) C>S      application_data
    data: 6 bytes
-----
DATA
-----
22 950072479.7749 (0.0016) S>C      application_data
    data: 14 bytes
-----
354 go ahead
-----
Client transmits actual message
23 950072479.7752 (0.0003) C>S      application_data
    data: 244 bytes
-----
Received: (qmail 21198 invoked by uid 556); 9 Feb 2000 05:01:19 -0000
Date: 9 Feb 2000 05:01:19 -0000
Message-ID: <20000209050119.21197.qmail@romeo.rfm.com>
From: ekr@romeo.rfm.com
To: ekr@mike.rfm.com
Subject: test

test message
-----
24 950072479.8403 (0.0650) S>C      application_data
    data: 27 bytes
-----
250 ok 950048042 qp 25454
-----
Client closes connection
25 950072479.8404 (0.0001) C>S      application_data
    data: 6 bytes
-----
QUIT
-----
Client FIN
26 950072479.8412 (0.0007) S>C      application data
    data: 19 bytes
-----
221 mike.rfm.com
-----
Server FIN
```

图 10.8 使用 STARTTLS 的升级

与先前一样，在第 1 段，服务器发送 220 响应来标识自己的身份，客户端随后发送 EHLO 命令。当客户端在响应这条命令时提供 STARTTLS 扩展，指示它做好了处理 STARTTLS 命令的准备（第 3 段）。

在我们原来的 SMTP 跟踪信息（图 10.2）中，客户端会在此刻发出 MAIL 命令开始发送消息。然而，它在这里发出 STARTTLS 命令初始化 TLS 握手，见第 4 段。

在第 5 段中，服务器使用 220 ready for tls 进行响应，指示客户端应当使用 ClientHello 继续执行。注意，可以在响应 EHLO 的时候宣传支持 STARTTLS，但在实际请求它的时候加以拒绝。实际上，这恰恰是不带证书配置 qmail 时 qmail 的做法。

第 6 段包含客户端的 ClientHello。注意客户端实际提出一种 SSLv2 向后兼容的握手。与 HTTP 升级不同，RFC 2487 在服务器与客户端可望具体支持何种版本的 SSL 和 TLS 上并不十分明确。使用 TLS 这个字眼暗含它们支持 TLS 而不是 SSLv2 或 SSLv3。然而大多数当前可用的 STARTTLS 实现都基于同时提供 SSLv2 和 SSLv3 支持的工具箱，因此它们通常都提供 SSLv2。

握手的其余部分（从记录 7~记录 16）以通常的方式结束，但有一项令人吃惊的特性就是服务器从不请求客户端认证，这是 STARTTLS 实现中相当常见的特征。因此，接收服务器无法采用密码学的方法来识别服务器的身份。实际上标准暗含它不应当采用这样的手段。将在第 10.18 节看到这样做的原因。

一旦握手完成，客户端就通过新磋商好的 TLS 通道加密发送其 EHLO，而邮件投递像普通邮件投递一样进行，只不过是经过 TLS 加密的。

10.17 连接关闭

注意，客户端在图 10.8 中仅简单地关闭连接而没有发送 close_notify。它只是发送 QUIT 命令然后关闭它这一端的连接。类似的，服务器发送响应 221 然后关闭它这一端的连接。大家应该还记得 close_notify 的目的是阻止截断攻击。在这里我们要问的问题是，这样做有危险吗？

显然这条连接未受攻击，有可能受到攻击的吗？答案是否定的。交换 QUIT 与 221 明显表示双方想要关闭连接。这些消息是通过 TLS 发送的，因此攻击者无法对其进行篡改。在第 7 章我们讲过，如果应用协议有其自己的数据结束标志，则不需要 close_notify。这里就是这样一种情况。

其他的情况

SMTP 有一种方便的属性，双方总能够知道该哪一方说话以及一条命令或响应的时间。因此很容易判定在协议消息中或协议消息之间是否发生了非期望的关闭。

由于邮件服务器在碰到错误的时候会（超过一定时限之后）自动重试，对大多数非期望关闭的恰当响应都只是进行登记并继续执行。然而，重复出现的过早关闭模式有可能代表一种拒绝服务攻击，应当对其进行调查。

一般来说，任何代理都不应当在只有部分消息的情况下执行相应的动作。因此，如果一个代理只收到了部分命令或 E-mail 消息，则应当不予处理。类似的，发送者不应当假定邮

件已经被接收者接收除非切实见到了指示传输邮件被接受的 250 响应才行。

恢复

正如 SSL 规范所要求的，代理一定不能恢复没有使用 close_notify 关闭的会话。因此，图 10.8 中所描述的会话无法恢复。对 SMTP 来说，会话恢复并不像 HTTP 那么重要，因为发送者通常不是在短时间内反复重新连接的。尽管如此，正确的实现应当发送一个 close_notify。

10.18 要求使用 TLS

大家可能希望将自己的邮件服务器配置成要求使用 TLS 来传输消息。从技术上讲可以对外出通信和入内通信单独进行配置。如果你并不关心是谁给你发送邮件但想确保你传输的邮件是安全传送的则可以选择这样来做。与之对应，你可能选择只接受经过认证的来源发来的邮件但可以将邮件传送给任何人。然而不幸的是，对入内或外出通信要求使用 TLS 实际上并不是大家所期盼的。

外出通信

要求使用 TLS 传输邮件是容易的。只要拒绝连接到不提供 STARTTLS 扩展的服务器即可。然而，由于大多数服务器都不支持 STARTTLS，这也就意味着你企图发送的绝大多数消息都会被弹回（bounce）。人们可能希望由用户来指定要求使用 TLS 进行传输的特殊消息，但是没有完成这项任务的标准做法，因为没有标准的将消息标记为以 TLS 方式加以传输的方法，所以无法要求中继链中的其他 MTA 使用 TLS。

入内通信

RFC 2487 支持对所有入内通信要求使用 TLS，接收服务器简单地使用下面的内容来响应发送方的命令（除 EHLO、NOOP、STARTTLS 和 QUIT 以外）：

530 Must issue a STARTTLS command first

虽然 RFC 2487 显式地禁止公用服务器进行这样的配置，但有一个例外：除非是从适当经过 TLS 认证通道过来的通信数据，否则服务器均可以拒绝进行中继（请参见第 10.21 节）。这条规则的意图是阻止管理员以这样一种打破互操作性的方式对他们的系统进行配置。要求使用 TLS 会将 E-mail 与因特网的大部分通信隔离开来。而且还将使得推行安全的工作变得非常困难，因为它禁止接收者要求对邮件发送者进行认证。

10.19 虚拟主机

在第 9 章见过升级磋商（即升级）方案可以使虚拟主机与 HTTP 一起正常工作。或许我们还期望它们能够与 STARTTLS 一起正常工作。不幸的是，由于规范中的一点疏漏，它们始终无法一起正常工作。在 HTTP 升级中，客户端使用 Host 头信息来指示它希望连接的虚拟主机。类似的，我们期望发送者在头几段中的某个地方标识主机。

然而，在先于 TLS 握手的 TCP 段中（1~5），发送者没有标识接收者域（它标识虚拟主机）的地方。这只有在发送者发出 RCPT 命令时才行，由于 TLS 连接已经磋商好了，所以

为时已晚。

恰当解决这种问题的方案就是让发送者以 STARTTLS 命令变元的形式来提供接收者的域。它不应当提供 local-part，因为那样是不必要的，而且我们还想保守这个秘密。如果这样做，虚拟主机就能正常工作。注意，这对 STARTTLS 方案来说不成问题，它只是规范的一个错误，而且已经建议在 RFC 2487 推进到草案标准时加以修正。

10.20 安全指示器

显然，用户想要能够判传输消息时所使用的安全属性。因为消息常常是被写到磁盘上然后才由客户端读取的，所以安放这种信息的逻辑位置就是在消息的头信息中。这也省去了我们修改邮件阅读协议以承载这种信息的麻烦。

图 10.9 描述了一个例子，它与我们在图 10.8 中见到的消息一样，都是于后存储在磁盘上的。注意倒数第二个 Received 头信息，它的部分内容为 DES-CBC3-SHA encrypted SMTP。如果发送者使用过客户端认证，则这个头信息还有可能包含发送者用来认证的证书或其他一些等价的有关发送者身份的指示。

```
From ekr@romeo.rfm.com Tue Feb 08 22:14:02 2000
Return-Path: <ekr@romeo.rfm.com>
Delivered-To: ekr@mike.rfm.com
Received: (qmail 25454 invoked from network); 8 Feb 2000 22:14:02 -0000
Received: from romeo.rfm.com (216.98.239.227)
        by mike.rfm.com with DES-CBC3-SHA encrypted SMTP; 8 Feb 2000 22:14:02 -0000
Received: (qmail 21198 invoked by uid 556); 9 Feb 2000 05:01:19 -0000
Date: 9 Feb 2000 05:01:19 -0000
Message-ID: <20000209050119.21197.qmail@romeo.rfm.com>
From: ekr@romeo.rfm.com
To: ekr@mike.rfm.com
Subject: test

test message
```

图 10.9 使用 STARTTLS 发送的消息

对指示器的解释

没有究竟如何在头信息中安放这种安全信息的标准。RFC 822 提供了一些有关头信息中这种信息出现的位置的指导（在 with 关键词之后），除此之外实现可以按照自己的需要来安排这些信息。类似的，客户端程序能够尝试对它们进行解释，但有可能在无法解释的情况下最后显示给用户自己看。注意，在正常情况下，许多客户端甚至不向用户显示 Received 头信息，除非他们要求这样。

最后一个步跳

重要的是要认识到头信息是不以任何方式进行加密认证的。因此，最好的情况下，用户可以直接掌握有关最后一组 Received 头信息的信息：那些由本地邮件服务器在最后一个步跳 (hop) 上创建的信息。接收方必须假定其本地邮件服务器是受信的，否则就无安全可言。



因此，他相信其邮件服务器没有创建虚假的头信息。然而任何其他的头信息都有可能被沿途的任何代理所篡改。

因此为了对从发送者到接收者整个路径下来的消息的安全属性确信无疑，连接必须信任沿途的所有主机。此外，必须对那些主机进行加密认证。不然，攻击者就可能冒充你信任的发送者。因为 Received 字段的格式不是固定的，所以无法自动进行这样的评估，必须由用户完成这样的工作。

10.21 经过认证的中继

STARTTLS 基于证书的客户端认证存在两种可能的用法 (*authenticated relaying*)：认证中继以及源发者认证 (*originator authentication*)。本节讨论认证中继，下一节讨论源发者认证。

认证中继的目的是允许服务器为其识别的服务器提供中继服务而拒绝那些它不认识的服务器。正如我们在 10.7 节所讨论的，开放中继常常被垃圾邮件者所滥用。尽管如此，一些站点还是需要中继邮件。如果你用于中继邮件的唯一一台机器与你的邮件服务器都处于防火墙后面，那么往往可以使用基于 IP 的认证。然而，现实常常不是这种情况，要么是因为没有防火墙，要么是你的邮件服务器需要为网络上其他地方的机器中继邮件。

在这种情况下，使用基于证书的客户端认证来限制中继服务是相当合适的。有两个值得考虑的有趣情况。第一种情况，发送者已经使用 TLS 进行了连接。第二种情况是发送者没有使用 TLS 进行连接。对于任何一种情况，服务器都需要检查 RCPT 命令来查看一下是否请求中继。

如果发送者使用 TLS 连接，那么服务器需要根据允许请求中继的服务器列表来检查它的证书。如果没有证书，服务器就需要请求再握手。这与我们在第 9 章 HTTP 服务器请求再握手促使客户端进行认证的情况是类似的。

RFC 2487 就发送者压根没有使用 TLS 时应该怎么做没有清晰地规定，但是估计接收服务器应当发送 530 响应，指示发送者必须使用 TLS，然后在 TLS 握手时请求客户端认证。

10.22 源发者认证

客户端认证的另一种用途就是提供源发者认证。源发者认证的目的就是向接收者提供发送者身份的担保。显然，这对接收者来说是颇有价值的，因为他们不想被伪造的 E-mail 所欺骗。不幸的是，源发者认证与 STARTTLS 一起工作时非常糟。

由于 RFC 2487 禁止要求使用 TLS，显然不可能再要求通过 TLS 进行客户端认证。因此确实不可能强制推行任何种类的源发者认证。然而，服务器有可能选择在磋商 TLS 的时候请求客户端认证。在那样的情况下，如果发送服务器使用 STARTTLS 并有证书的话，接收服务器至少能知道是谁在最后的步跳上发送数据。如果接收者着实幸运，则每个单一的步跳都使用 TLS，而且每个发送者都对接收者执行客户端认证。在这种情况下，用户就有希望判定原始发送者是谁。

然而，为了做到这一点，用户将不得不检查 Received 头信息来判定处理消息的每个主机都是受信的并应位于消息路径当中。这样尤为困难，因为不可能每个接收服务器都将每个发送者的整个证书链添加到头信息当中。因此最终的接收者无法判断原始发送者使用的证书，以及对路径中的每个服务器执行有效性验证直至其信任的 CA。

至今为止在大多数情况下，没有好的使用 SMTP 和 TLS 来完成源发者认证的方法。这种结论并不是因为 STARTTLS 细节的限制，它是 TLS 与 SMTP 服务模型交互的一般性问题。即便对各种头信息进行了标准化，从技术上可以自动地将合法的发送者与非法的发送者区分开来，也仍然要求信任沿途的每个主机，这是一种根本的不合理要求。因此，接收者无法保证发送者就是它所声称的发送者，即使消息沿整个路径都是加密的。

10.23 引用完整性的细节

引用完整性的情况稍微好一些。在第 10.2 节讲过，发送者希望消息能被投递给接收者而且在整个传输过程中进行保护。不幸的是，SMTP 和 STARTTLS 的机制使其几乎是不可能。

安全引用

我们面临的第一个问题就是 E-mail 地址不包含任何可以安全地给他们发送邮件的指示。为什么不包含这样的指示是显而易见的：这样的指示能表示什么呢？接收者对连接发送方服务器一无所知，他或许了解最多的还是位于连接的这一方的服务器。

从用户 `foo@corp.example.com` 的角度考虑图 10.10 中的网络设置。他知道当服务器查找 `example.com` 域时，将会得到一个指向 `mail.example.com` 的 MX 记录。他还知道 `mail.example.com` 和 `corp.example.com` 支持 TLS。因此知道可以安全地将邮件投递到最后一个步跳。如果连接到 `mail` 的服务器也具有 TLS 能力，那么它就能够安全地与 `mail` 进行连接。但是他可以确信的顶多如此。

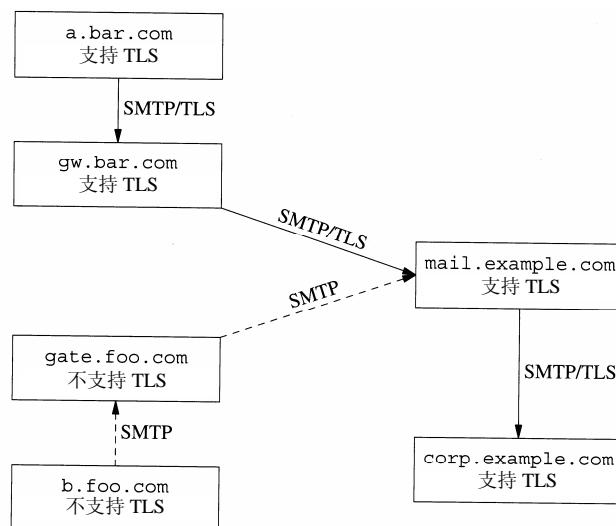


图 10.10 具有安全与非安全服务器的网络

事实上，两个不同的发送者会得到两组不同的安全属性。位于 a.bar.com 的用户能够将自己的邮件通过加密链接一直安全发送到 corp.example.com。与之对照，从 b.foo.com 发送的邮件经过非安全通道一直到达 mail.example.com，只有最后一个到 corp.example.com 的步跳是加密的。接收者没有合理地预测任何给定发送者行为的办法。

显然，我们可以使用本质上表明：“如果你是 TLS 使能的（enabled），就应当使用 TLS 给我发送邮件”这样的弱强度指示。然而这样必定导致不同形式的 E-mail 地址，因为我们目前的形式没有安排这样一种指示的位置。这似乎是用高开销带来小收益。

强制推行安全

设想发送者有一些外部的信息可以影响接收者使其愿意通过 TLS 连接接收 E-mail。而且，发送者知道他的本地服务器具有 TLS 能力。那么他肯定可以将 SMTP 代理配置为仅限使用 TLS 收发邮件。然而，这样的所带来的好处并不多。

同样，考虑图 10.10 中的网络。位于 a.bar.com 的用户想要给 foo@corp.example.com 发送邮件。他知道他具有安全能力而且接收者也是如此，因此他将自己的代理配置成执行 TLS，它可以一直推行到 gw.bar.com。然而，gw.bar.com 不知道与 mail.example.com 磋商使用 TLS，因此主动攻击者可以将连接降级为普通的 SMTP。所要做的仅是篡改 EHLO 响应，并删除 STARTTLS 提议。而连接的其余部分可以保持不便。

显然，如果我们想发送邮件的话，就需要某种指示沿途的每个服务器都使用 TLS 的办法。可以通过使用新的邮件头信息或（更可能的情况）使用提供给 STARTTLS 扩展的一些变元。当然这样一种扩展意味着如果客户端与服务器之间的任何服务器不使用 TLS，那么邮件将被简单地弹回。但有时牺牲互操作性是我们获得安全所要付出的代价，因此也可能是值得的。然而，我们很快将会看到情况并不是这样。

中继还是安全

不幸的是，SMTP 提供中继的可能多少破坏了任何沿消息路径推行 TLS 的企图。同样考虑我们在图 10.10 中发送的消息，在 DNS 欺骗（spoofing）攻击下，攻击者会伪造 corp.example.com 的 MX 记录。通常它们会读取到类似下面的信息。

```
corp.example.com preference=10, mail exchange=mail.example.com
```

攻击者将其替换为：

```
corp.example.com preference=10, mail exchange=mail.attacker.com
```

现在，当 a.bar.com 发送消息的时候，我们得到图 10.11 中所示的路径（假定攻击者费神投递消息而不是将其丢弃），注意投递过程中的每个步跳仍然使用 TLS 进行传输。此外，MX 记录的内容与攻击服务器的证书匹配。我们在链接中简单地增加一个中继。显然，只要求使用 TLS 是不够的。我们需要服务器强制核查向其传送信息的中继的身份。这并不奇怪，因为我们在第 7 章说过，只有有效证书对引用完整性来说是不够的。

我们所面临的问题是，图 10.11 中所示的路径没有任何根本的错误。在这种情况下，多余的中继为攻击者，但是这种拓扑结构相对常见。正如在第 10.7 节所讨论过的，对于从组织 A 发到组织 B 的邮件路由经过第三方组织的情况并不稀罕。因此，我们根据 RCPT 命令中的 E-mail 地址对证书中的主机名进行检查。

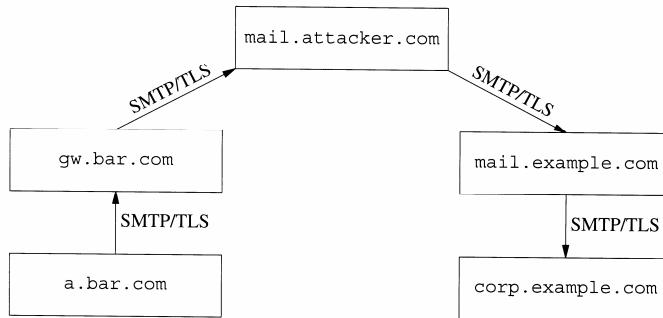


图 10.11 伪造的邮件交换器

惟一已知可行的方法就是对于每个合法中继给定的主机都要有一个以那个主机的 domain 作为其身份的证书。尽管这是安全的，但最终将会成为配置灾难，因为必须为每一个这样的中继执行手工配置。更糟的是，除非细致地实施撤消工作，否则攻破任何这样的服务器就会永远攻破到目标域的所有通信数据。如果使用安全 DNS，那么就能知道 MX 记录是正确的，然后我们就可以根据服务器的证书对 MX 记录中的名字进行检查。然而，安全 DNS 的部署到目前为止还非常少，因此这并不是一种切实的选择。

至今为止，我们得出的结论是 STARTTLS 不能给发送者提供任何真正的有关其发送消息的安全属性的担保。即便他愿意限制自己只通过 TLS 发送邮件，也不能确信消息将会安全地投递给指定的接收者，更不能确信邮件投递没有遭到破坏。这种问题与特定的 STARTTLS 细节没有任何关系，它与使用 TLS 来保护步跳连接的安全有着内在的联系。步跳之间的安全并不能保证两个端点之间的安全。

10.24 为什么不使用 CONNECT

显然，在处理 SMTP 时我们所面临的问题与存在的中继有关。HTTPS 本质上是通过绕过像代理这样的中间节点来工作的。CONNECT 方法告诉代理不要处理 HTTPS 连接。有了客户端与服务器之间端到端的通道，HTTPS 就能够确保提供这种 STARTTLS 所不能保证的安全属性。

这提出了一个问题，是否有可能使用某种 CONNECT 的变体以隧道形式通过 SMTP 中继，就像我们以隧道形式通过 Web 代理那样。不幸的是，这种做法无论从技术上还是从管理上讲都存在问题。

技术问题

这种方案所面临的技术问题是这会彻底打破为断开的服务器提供中继。正如我们在第 10.3 节所讲的，一些组织只是偶尔才连接到网络上存取邮件，这样的网络使用中继来替它们存储邮件。显然不可能通过隧道连接到没有与网络连接的服务器。

我们可以通过将最后的中继当作接收者的本地服务器来暂时解决这种问题。但我们得为接收者的域颁发恰当的证书并让它与发送者磋商使用 TLS 连接。由于除此之外根本不让这样的网络收取加密邮件，所以这或许是可以接受的。

邮件服务器压根就不是设计以这种方式来工作的。它期望能够将消息入队并在空闲时进行处理。这一点是重要的，因为这可以使它强壮得足以抵御暂时性的网络失效。而使用隧道通过服务器则需要对软件进行大的体系改动。此外，它还侵害了因特网邮件环境重要的稳固不变性。即便双方同时都不在网络上，邮件也是可投递的，这就是存储转发的意思。

最后考虑对发送客户端的影响。在当前的环境中，邮件是“发送后不用管”的。客户端可以将邮件发送给本地邮件服务器而用户可以彻底与网络断开，但仍然相信他的邮件可以被投递。如果客户端打算以隧道形式到达最终的服务器，那么该服务器必须是联机的——而且即便接收者的服务器暂时脱机也要安排重试。用户频繁连接上网，但每次只有几分钟时间，所以这种结果实在不可接受。

● 管理问题

更为严重的问题是管理问题。与 Web 服务器不同，邮件服务器常常位于防火墙后面。HTTP CONNECT 只允许客户端从防火墙后面以隧道形式出去。我们考虑的这种方案将允许发送者通过隧道形式进来。可以理解，允许加密的连接连接到位于防火墙后面的邮件服务器会使管理员紧张，这是因为占主导地位的 UNIX 邮件程序 Sendmail 的安全问题实在是臭名昭著。

10.25 STARTTLS 有什么好处

在第 10.2 节我们提出了几种因特网邮件安全协议最有价值的属性。我们想让发送者确信他们的邮件被安全地投递给接收者。想让接收者能够对发送者进行认证并知道消息在传输过程中没有被篡改。在第 9 章我们已经见过，HTTPS 满足所有这些目标，然而我们刚刚见到 STARTTLS 不能提供这些属性中的任何一种。自然，问题出来了：STARTTLS 有什么好处呢？

● 被动攻击

我们所描述的针对 STARTTLS 的攻击都依赖于攻击者对网络传输进行篡改，一般是通过扮做中间人或通过污染发送者的 DNS 缓存来实现的。本质上它们依赖于攻击者假冒通信的一方，然后颠覆我们的安全保护。虽然这是一种需要担心的合乎常理的攻击形式，但是实施起来也比较困难的。此外，攻击者需要处于联机状态并传送数据，这使得追踪起来更加容易。因此合理的是考虑 STARTTLS 针对被动攻击提供什么类型的保护。

如果我们只对伺机(opportunistic)加密感兴趣，也就是忽略主动攻击的话，那么 STARTTLS 是有用的。因为发送者不必再担心连接到错误的网关，所有的基于中继替换的攻击都消失了，而降级攻击也不存在了。我们实际所要关心的就是服务器是否支持 STARTTLS。如果消息传输路径中所有服务器都支持它，那么就能够保密地投递这个消息。当然，仍然需要信任网关没有读取我们的邮件。根据它们是否属于第三方，这可能是也可能不是一种合理的假设。

我们甚至无须使用“强制推行 TLS”标志。服务器仍然能够在合适的地方自动进行升级。然而，可以引入一种标志，在服务器不能安全投递的情况下弹回邮件。

注意，由于我们不关心主动攻击，所以证书所带来的好处非常少。大家还记得证书的主要目的就是阻止主动攻击者将合法接收者的密钥替换为其自己的密钥。由于我们已经裁定那种攻击超出了范围，所以证书的作用不大。我们可以彻底放弃使用证书而只使用匿名 DH。从某种意义上讲，这是甘心情愿且非做不可的事，因为我们在第 10.8 节看到检查一份证书

是否代表合法的接收者对 SMTP 来说可是难题之一。

没有发送者认证

尽管我们可以拥有保密性，但是讨论认证或消息完整性毫无意义。认证的要点就是在面临主动攻击时对消息提供担保。由于我们已经裁定这种攻击超出范围，所以认证与消息完整性压根就无关紧要。

10.26 编程问题

实现 TLS 上的 SMTP 实际上不要求掌握我们以前所见到的任何技术用法，但是 SMTP 有几处小的我们在讨论 HTTPS 时还没有涉及的问题。本章余下的内容将描述其中两个问题：实现 STARTTLS 扩展以及服务器启动。这些问题不难，所以不需要范例代码，但是它们的重要性足以引起重视。

10.27 实现 STARTTLS

从概念上讲，实现 STARTTLS 与实现 HTTPS 的 CONNECT 方法是相同的。对于这两种情况，客户端都需要以明文形式完成应用层握手，然后再切换到 SSL/TLS。服务器代码与客户端代码类似。然而，STARTTLS 比 CONNECT 稍微复杂一些。HTTPS 客户端知道它们要通过代理而且 CONNECT 的处理非常简单，客户端无须实际的 HTTP 引擎就可以编写。然而，由于 STARTTLS 是一种升级磋商策略，所以它通常直接与 SMTP 引擎交缠在一起。

状态

我们需要关心的主要问题是 SMTP 引擎的状态。RFC 2487 相当清楚地表明，在 TLS 握手之前宣传和磋商的任何扩展在握手完成时就不再适用于该连接。因此实现必须准备在 TLS 握手完成时抛弃所有状态和再磋商扩展。这种要求有两个原因。第一，攻击者有可能干预了扩展宣传（extension advertisement）。第二，服务器有可能愿意在使用 TLS 时提供不同的服务。

网络存取

出于效率的考虑，SMTP 实现想要对网络读写提供缓冲是相当常见的。一些实现使用 stdio，而另一些使用自己的网络缓冲代码。无论哪一种情况，SSL 实现都需要能够向网络中写入和从网络中读取数据。大多数 SSL 实现都提供了某种抽象的网络 API，它允许应用提供自己的 I/O 例程。OpenSSL 使用 BIO 对象来完成这种工作，如图 10.12 所示。

```
26     ssl=SSL_new(ctx);  
27     sbio=BIO_new_socket(sock,BIO_NOCLOSE);  
28     SSL_set_bio(ssl,sbio,sbio);
```

sclinet.c

sclinet.c

图 10.12 OpenSSL BIO 初始化



这种抽象允许用户提供自己的 I/O 例程，只要它们能够表示为一个 BIO 对象就行。从理论上讲，PureTLS 可以通过让用户提供 InputStream 和 OutputStream 对象来完成相同的工作。然而，正如我们先前所讨论的，PureTLS 中的这种功能现在还不可用。注意，如果 SMTP 实现中的缓冲想要每次读取一行信息，那么就必须在 TLS 握手之前禁止这种功能，因为 TLS 数据不是面向行的。

如果 SSL 工具箱不提供可替换的网络 API，那么几乎可以肯定它是直接从套接字读写信息的。对于那种情况，SMTP 实现需要给工具箱提供一个 raw 套接字。对于 STARTTLS 来说这相对容易一些，因为命令和响应的长度总是已知的，SSL 握手的开始也是如此。因此没有必要担心 SSL 握手的某些部分会滞留在 SMTP 读/写缓冲区中。

10.28 服务器的启动

以与 Web 服务器相同的方式来运行邮件服务器相当常见。我们有一个单一的服务器进程，它接受客户端的连接然后创建新的控制线程来处理它们。我们在第 8 章已经看到如何编写这类程序。然而，UNIX 系统上的许多邮件服务器配置都不相同。UNIX 有一种称作 inetd 的程序。Inetd 是超级服务器，它在多个端口上监听连接并且有一个配置文件告诉它在处理每种连接时运行哪一种程序。通常这种方案对于负载轻的服务器来说工作良好，但是在使用 SSL 时就会出现问题。

在第 8 章中我们看到 SSL 代理通常需要一次性初始化一个包含密钥资料、随机值等内容的 SSL 上下文对象，所有后续的连接都将使用这个上下文。显然，如果我们通过 inetd 运行邮件服务器，则必须在每次收到 E-mail 的时候创建上下文对象。这提出了两个问题：首先我们必须安排服务器取得它的密钥资料。其次，必须确保初始化是快速的。

密钥资料

正如我们在第 5 章所见到的，保护密钥资料的标准方法是使用通行码进行加密。如果我们运行单个像 HTTPS 服务器这样的服务器进程，则只需在系统启动时提示用户输入通行码。由于这给无人看管的启动造成了困难，所以多少有些不方便，但并非是不可行的。然而，如果是从 inetd 启动的话，那么这种方案就不再可行了。

在这种情况下，常用的方法就是简单地以明文形式将私用密钥留在磁盘上，使用文件选项和存取控制列表加以保护。稍微复杂些的解决方案就是在机器启动时，启动一个单一的服务器进程。当启动进程时，提示用户输入口令。邮件服务器程序随后联系该服务器来获得密钥资料。对服务器程序的存取则通过文件权限或 ACL 来控制。

这种方案的好处就是密钥资料在磁盘上仍然是加密的。因此，如果机器被关闭的话，需要通行码才能进行重起，进而提供了某种抵御服务器盗窃的保护。显然，如果密钥保存在硬件中，则系统就不会受制于这些攻击。

快速初始化

大家应该还记得使用上下文对象的主要动机就是它只执行一次耗时的初始化。因此，如果我们有耗时的初始化阶段，那么就需要通过某种方式加以回避。启动时常见的耗时工作就是产生临时密钥以及随机种子的抽取。可以预先计算出这种数据并存储在文件中。正如我们

在第 9 章所看到的，我们的服务器可以在启动时从磁盘上读取这种数据。

10.29 总 结

尽管 SSL 设计者想让 SSL 成为一种通用的安全层，但是 SSL 不是特别设计与 SMTP 一起工作的。然而，对邮件安全的迫切要求引发了使用 SSL 来保护 SMTP 的尝试，尽管 SMTP 模型与 SSL 所提供的服务之间存在一些严重的不匹配。这一章我们探讨了支持 TLS 的 SMTP 的标准化套路，并对这种方案中的一些问题进行了研究。

E-mail 是一种存储转发系统。发送代理使用 SMTP 将消息发送给路径中的下一个服务器，而该服务器再尽力将消息投递出去。

RFC 2487 规定了一种使用 TLS 保护单个 SMTP 链接的方法。它引入了 STARTTLS 扩展，客户端与服务器可以用它来从普通的 SMTP 连接磋商升级为 TLS 上的 SMTP。

中继会产生问题。由于单独的 SMTP 链接是要保护的对象，所以客户端与服务器之间的每个中继都必须是受信的。此外，由于中继发现（discovery）是通过 DNS 完成的，所以要想阻止攻击者冒充中继很困难。

发送者不能决定安全属性。由于 E-mail 地址不包含有关传输通道安全的信息，所以发送者无法知道他的消息是以保密形式发送的。

接收者无法要求发送者认证。RFC 2487 禁止要求使用 TLS 接收邮件。即便使用 TLS，存在的中继也会使得判断原始发送者非常困难。

STARTTLS 提供抵御被动攻击的保护。如果发送和接收服务器均具有 TLS 能力，那么它们将默认磋商使用 TLS 连接，因此也就可以防止嗅探。

各种方案的对比

“如果你只有一把锤子，则会把任何问题都看成钉子。”

——Abraham Maslow

11.1 介绍

前面的章节重点放在 SSL 上。我们已经讨论了 SSL 的设计目标、工作原理以及套路。此外还研究了两种使用 SSL 来保护的应用层协议：HTTP 和 SMTP。

现在你应当对 SSL 是如何工作的，以及能够完整什么样的工作有了相当理解。最后一章，我们将从更广的角度查看一下这些内容。正如我们在第 9 章和第 10 章所看到的，某些安全任务很难用 SSL 来实现。在许多情况下，其他的安全技术却能够更为妥善而且更容易地完成这种工作。因此要想挑选出最好的解决方案，重要的是对其他的技术有所了解。人们使用 SSL 来保护某些应用往往只是因为 SSL 是他们惟一熟悉的解决方案。

本章考虑三种其他的安全协议。首先要考虑的是 IPsec，它为 IP 包提供安全。与 SSL 不同，IPsec 工作在较低的层次上，但却提供了许多相同的安全服务。它也有一些相同的限制。

我们要考虑的第二种方案就是对象（object）安全。对象安全不是对通道进行加密而是在明文通道上发送经过保护的对象。因为需要保护单个协议对象的安全，所以对象安全协议通常是特定于应用的。我们将研究安全 HTTP（Secure HTTP），它为 HTTP 事务提供安全和 S/MIME，且为因特网邮件消息提供安全。

11.2 端到端的论述

这一章将对协议栈的多个层次进行研究。我们将会看到，在协议栈中所处的层次越高——离应用层越近，越能提供更好的安全服务。这就是端到端论述的一个实例，Saltzer、Reed 和 Clark [Saltzer1984] 对此有精彩地陈述：

只有在通信系统末端的应用对情况有所了解并协助完成工作的条件下，才可以彻底且正确地

实现所要实现的功能。因此将那种功能作为通信系统的一种特性来提供是不可能的（有时通信系统所提供的不完整的功能版本对于提升性能是有用的）。

[Saltzer1984] 是计算机科学中的伟大文献之一，是任何对理解网络感兴趣的人们的重要读物。这篇论述可以总结如下：任何通信系统都要涉及各种中间环节，如网络设备、计算机以及对所涉及的通信系统的整个上下文环境一无所知的程序。因此这些中间环节没有能力保证数据被正确处理。[Voydock1983] 在安全的上下文中具体阐述了这一观点。

这种论述给人以明显直观的感觉，它与我们处理 HTTP 和 SMTP 时的经验相符。在开始引入代理和中继之前，HTTPS 和 SMTP/TLS 还工作得好。为了使 HTTPS 能够与代理协调工作，我们不得不以隧道的形式通过代理——本质上就是创建端到端的通道——而 SMTP/TLS 的安全被中继搞得一团糟。

11.3 端到端的论述与 SMTP

通过例子最容易理解端到端的论述。考虑 TLS 上 SMTP 的情况，位于通信系统末端的应用为发送者和接收者所使用的 E-mail 程序。然而，SMTP/TLS 是在较低层次上提供安全的：即邮件中继之间的邮件传输层。因此，它没有能力提供正确的安全服务器。各个方面都反映了这一点，不过我们只考虑其中的两种情况：最后步跳（last-hop）的投递与中继。

● 最后步跳的投递（last-Hop delivery）

大家还记得将邮件投递给最终用户不是通过 SMTP 来完成的，而是通过其他一些诸如 IMAP、POP 或干脆就是从磁盘上读取邮件的方法来实现的。这就意味着不可能存在能够确保本地服务器与邮件读取客户端之间数据完整性与保密性的 SMTP/TLS 方法，因为服务器与邮件阅读器之间的通信通道有可能被攻破。

即便我们能够以某种方式保护协议的安全，比如说通过在 SSL 上运行该协议，本地服务器仍有可能被攻破，从而使消息受到损害。且没有能预防这种问题的链接层安全措施。与之对照，如果安全是由邮件程序以端到端的方式来提供的，那么我们就能够阻止这种攻击。

● 中继

在处理中继时我们所面临的问题也是类似的。SMTP/TLS 提供链接层次（link-level）的安全。中继之间的链接是受保护的，但是任何给定的链接都有可能被攻破。更糟的是，SMTP/TLS 没有办法指示下一个步跳必须进行加密——甚至下一个步跳是谁它都不清楚——因此攻击者可以将自己替换为中间人或对 SMTP 连接实施降级。同样，如果安全是以端到端的形式提供的，那么就可以阻止这种攻击。

● 一种端到端的解决方案

解决方案就是提供端到端的安全。因为接收者与本地服务器之间没有 SMTP 连接，所以无法像使用 HTTP CONNECT 那样建立一条 SMTP 隧道。正确的方案就是创建安全对象并通过普通的 SMTP 进行传送。我们将在第 11.17 节至 11.21 节看到，这就是 S/MIME 所使用的方法。

11.4 其他协议

我们将考虑三种协议：IPsec、Secure HTTP 和 S/MIME。每种协议都能完成 SSL 所完成的一部分工作。IPsec 是一种通用 IP 安全解决方案，可以用它来为任何在 IP 上运行的应用提供通用安全，就像可以使用 SSL 来为任何在面向连接的传输层上运行的应用提供安全一样。安全 HTTP（Secure HTTP）和 S/MIME 均为对象安全协议，设计用于特定的应用层协议。安全 HTTP 通过将每个请求和响应当作单个受保护的对象来为 HTTP 请求与响应提供安全。S/MIME 为单个 E-mail 消息提供安全。

图 11.1 描述了协议栈中各种协议之间的关系，图中阴影部分。底部是 IP 协议，我们通过它来承载所有的数据。IPsec 只是一组对 IP 的扩展，这些扩展增加了安全服务。因此任何在 IP 上运行的（也就是在 TCP 和 UDP 上运行）应用都能够很容易地在 IPsec 上运行。注意，尽管 UDP 只是 IPsec 上面的一层，不过它也能在 IP 上运行。

尽管从理论上讲，SSL 可以在任何面向连接的传输层上运行，但是在实际应用中，它几乎总是在 TCP 上运行的。通常建立在 TCP 层上的协议层，然而也可以建立在 SSL 层上。图 11.1 描述了三种协议：SMTP、HTTP 和安全 HTTP。注意，尽管我们描述的这些协议都位于 SSL 之上，不过 HTTP 与 SMTP 也能够直接在 TCP 上运行。

在图 11.1 的顶部是 RFC 822 和 MIME 消息，它们位于 SMTP 之上。S/MIME 消息是一种特定类型的 MIME 消息，负责提供加密安全。这样就能在 SMTP 之上像运输 RFC 822/MIME 消息一样来运输 S/MIME 消息了。

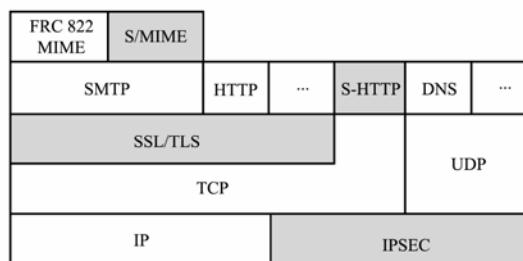


图 11.1 Where security fits in the protocol stack

11.5 IPsec

IPsec 是一种用来代表多种不同技术的总称，这些技术由 IETF IP 安全工作组进行标准化，分别在 RFC 2408 中记述：因特网安全关联与密钥管理协议（ISAKMP）[Maughan1998]；在 RFC 2409 中记述：因特网密钥交换（IKE）[Harkins1998]；在 RFC 2402 中记述：认证头信息（AH）[Kent1998a]；以及在 RFC 2406 中记述：封装安全负载（ESP）[Kent1998b]。这些技术共同为 IP 通信提供安全。RFC 2401[Kent1998c]描述了 IPsec 的体系结构。

与 SSL 类似，IPsec 也有一种由 ISAKMP 和 IKE 来提供的密钥交换与参数管理设施以及由 AH 和 ESP 提供的数据保护设施，这些设施之间的结合成分由 SA（安全关联）来提供。

ISAKMP 与 IKE 用来确立 AH 和 ESP 保护数据用的 SA。与 SSL 不同，密钥管理与通信保护功能是彻底分开的。也可以使用不带 IKE 的 AH 和 ESP，只要你有其他办法来确立 SA 就行。从理论上讲，能用 IKE 来为非 IPsec 协议交换密钥，但在实际应用中还没出现过这种情况。

我们的方案

为了比较 IPsec 与 SSL，我们首先需要掌握 IPsec 的基本工作原理。因此先从高层讨论 IPsec。先描述三种主要的部件：SA、ISAKMP/IKE 和 AH/ESP。然后再看看它们是如何来共同保护 IP 通信的。最后我们将对使用 IPsec 的安全通信与使用 SSL 的安全通信进行比较，看看每种方案的有利与不利之处。

11.6 安全关联

IPsec SA 与 SSL 会话之间有着松散的对应关系。SA 是指一组磋商好的算法和密钥资料，用于在两台主机之间传输数据。安全关联与 SSL 会话不同，它是单向的。为了让一对主机能够安全地进行通信，必须有两个 SA，且每个方向一个。每个 SA 都有一个 SPI (security parameter index, 安全参数索引)：用来标识 SA 的 32 位的值。SPI 有可能不是全局唯一的，但是每个 SPI、源 IP 地址以及协议 (AH 或 ESP) 三元组必须是唯一的。特意将 SPI 取得非常短是因为要在每个受保护的包中传输这种数据，较长的 SPI 会消耗过多的网络带宽。

11.7 ISAKMP 和 IKE

在使用 AH 或 ESP 发送任何数据之前，需要确立一个 SA。IPsec 体系结构文档 (RFC 2401) 提供了两种确立 SA 的方法：手工密钥设置和 ISAKMP/IKE。手工密钥设置的意思就是手工在两台主机之间设置密钥和算法。ISAKMP 和 IKE 是以一种与 SSL 握手类似的方式提供自动的密钥与会话管理。

ISAKMP 为磋商安全关联和密钥提供了一种通用框架。它描述了握手的各个阶段和消息，以及如何磋商算法。然而，它并没有提供任何具体的密钥交换方法。IKE 提供了基于 Diffie-Hellman 的密钥交换方法。

IKE 密钥交换基于 STS[Diffie1992]、Oakley[Orman1998] 和 SKEME[Krawczyk1995]。双方交换 Diffie-Hellman 份额 (公用密钥)，并使用共享密钥来导出通信加密和消息认证密钥。可以使用数字签名、公用密钥加密或共享密码来对 DH 份额进行认证。注意，由于我们使用的是 DH，所以可以自动获得 PFS (perfect forward secrecy, 完美向前安全) (请参见第 5 章有关 PFS 的讨论)。IKE 定义了许多众所周知的 DH 组 (group)。实现可以使用私有组，但是大多数实现都使用其中的一个公共组。与之对照，SSL 让服务器来选择组。

身份保护

IKE 所提供的独一无二的功能就是为交换各方提供身份保护。在主模式下，DH 份额是以明文形式来交换的，但是证书与份额的认证数据却是加密的。图 11.2 描述了这种模式的

执行过程。

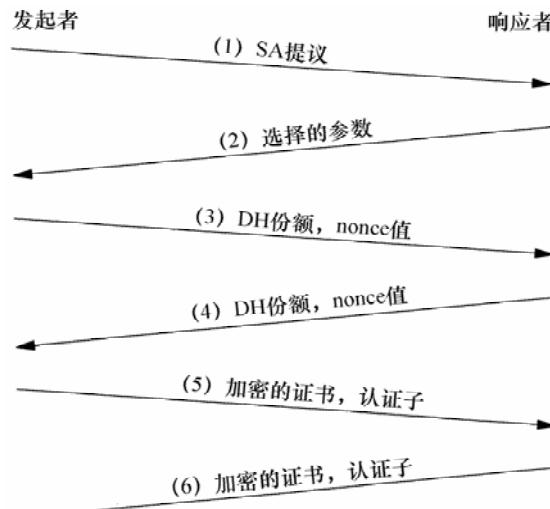


图 11.2 IKE 主模式

IKE 不区分客户端与服务器，因为 IP 层不存在这种差别。相反，第一个数据包的发送者被称作发起者，而第二个则被称作响应者。左边描述的是发起者，右边描述的是响应者。握手过程中发生一系列交换 (exchange)，发起者与响应者每方都只发送一条消息，因此主模式包含三次交换。

在第一次交换中，我们看到发起者发送一条它愿意支持的算法建议，响应者从集合中选择一种。在第二次交换中，它们每一方发送一个 nonce（与 SSL 中随机值类似的随机字符串）以及它们各自的 DH 份额。在第三次交换中，它们基于 DH 份额以及交换的其他信息交换签名。

握手的最终结果与使用 DH 和客户端认证的 SSL 大致相同，客户端与服务器进行相互认证并共享一组磋商好的密钥和算法。IKE 中没有办法像 SSL 只对服务器进行认证那样只进行单方认证。

在主模式中使用三次交换的原因就是防止各方的身份被他人嗅探到。身份保护主要在一个主机有多个身份的情况下并且想让攻击者无法分辨某个特定握手使用的是哪个身份时有用。首先，通过执行一次匿名 DH 交换，然后在加密的通道上继续执行其余交换，IKE 可以阻止被动攻击者判断通信各方的身份。尽管在 SSL 中通过使用匿名 DH 交换并加跟普通交换也可以提供这种服务，但远没有 IKE 的主模式妥善，因为它要求多个密钥磋商阶段。然而，如果这种类型的保护不是所要求的，那么所有交换就都可以合并成一次交换，这被称作 IKE 的进取模式 (Aggressive Mode)。

两阶段操作 (Two-Phase Operation)

SSL 握手作用于单个连接，而 IKE 则用来为两台机器之间的所有通信确立参数。不同类型的通信极有可能需要不同类型的保护。例如，Telnet 会话可能需要加密而 DNS 服务只需要认证。

为了满足这些不同的需求，IKE 使用一种两阶段握手。两台机器之间的第一次 IKE 握手只用来确立一个 ISAKMP SA，而这个 SA 只用来传输下一步的 IKE 通信数据。因此为了实际传输

正常的 IP 数据，你需要执行另一次 IKE 握手来为想要保护的特定类型的通信数据确立 SA。

IKE 提供了一种特殊的快速模式 (Quick Mode) 来完成这些新的握手。快速模式与 SSL 会话恢复类似，可以用来在无需执行新的密钥交换的情况下从原先的 SA 产生新的密钥资料。另外，如果新的 SA 需要 PFS 的话，也可以执行一次新的 DH 密钥磋商。认证阶段仍然会跳过去。由于交换是在经过认证的通道上进行的，所以也隐含地对这些通信进行了认证。快速模式建立一对 SA，每个方向上一个。

ISAKMP 传输

SSL 中，握手是在用来传输数据的 TCP 连接上于带内 (in-band) 发生的。一般来说，由于 ISAKMP 意图在两台机器之间而不是在单一的连接之上确立通信参数，因此采用带内传输显然不行。相反，指派 UDP 端口 500 来传输 ISAKMP 通信数据。

注意，使用 UDP 的结果无法保证数据包从发送方投递到接收方。因此，任何 ISAKMP 实现都必须维护重传定时器。如果给定的消息没有被响应，那么实现就必须重新传输这条消息。这增加了一些 SSL 所不要求的编程复杂性。

一种使用 UDP 的严重缺点就是 UDP 与防火墙的交互，许多防火墙都完全阻塞 UDP。因此，如果你的 IPsec 主机位于防火墙后面，那么就需要对防火墙进行配置，允许在端口 500 上通过或在其上通过代理来传输 UDP 数据。这样做要求说服你的防火墙管理员，而要想说服他们常常是非常困难的，因为防火墙上的漏洞，尤其是进入的 UDP 通信被认为是危险的。

11.8 AH 和 ESP

SSL 具有统一的单一记录格式，而 IPsec 则有两种——AH 和 ESP——格式，它们分别完成稍有区别的工作。AH 主要用于消息认证和反重放 (anti-replay)。ESP 用于通信数据加密、可选的消息认证及反重放 (anti-replay)。

AH

AH 背后的思想就是提供一种新的针对 IP 数据以及尽可能多的 IP 头信息进行认证的 IP 头信息，它通过计算数据以及大多数头信息的 MAC 来完成这项工作。由于有些头信息会在传输过程中发生改变，因此将它们排除在 MAC 计算之外。图 11.3 描述了一个使用 AH 来保护的数据包，其中显示了在增加 AH 前后的状态。要计算 MAC 的域用阴影来表示 (注意，尽管整个头信息都带有阴影，但是只有部分头信息计算了 MAC)。

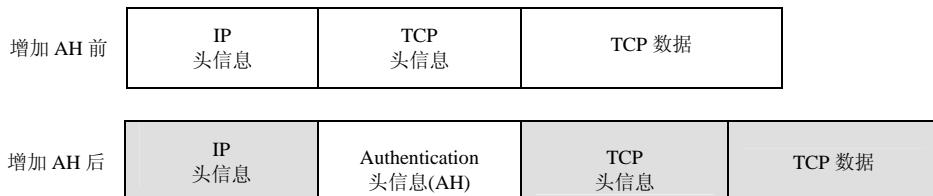


图 11.3 使用 AH 进行保护

AH 头信息还包含一个序号，MAC 计算也将它包括在内。这个数字用来提供重放 (replay)

检测。注意，工作正常的网络有时会有重放现象，这是中间路由器重传的原因。重放检测的目的就是为了确保主机的 TCP 和 UDP 层只看到每个数据包一次。阻止重放主要对 UDP 来说，因为 TCP 自动处理和拒绝重放的数据包。

ESP

ESP 提供加密和可选的对 IP 包负载（payload）的认证。与 AH 不同，没有对头信息提供任何保护，只对消息的负载提供了保护。ESP 能够提供加密、认证或二者兼有。图 11.4 描述了施加 ESP 前后的数据包，数据包受到保护的部分用阴影表示。



图 11.4 使用 ESP 进行保护

隧道模式（Tunnel Mode）

注意，由于 ESP 没有为头信息提供保护，所以攻击者有可能看到（或改变）源或目的地址。AH 与 ESP 都提供了一种称做隧道模式（tunnel mode）（我们刚才所讨论的比较简单的模式为传输模式（transport mode））的模式，其中对整个 IP 数据包进行保护。从根本上讲，这是通过将数据包包裹在另一个使用 AH 或 ESP 的数据包中来实现的。图 11.5 描述了在对一个数据包施加 ESP 前后的状态。



图 11.5 隧道模式中的 ESP

总体而言，传输模式应当用于两台主机之间的数据通信，因为像在隧道中那样重复头信息是不必要的而且还浪费带宽。发送方与接收方的地址无需经过加密验证，因为它们隐含于 SA 中。隧道模式主要用于创建 VPN（虚拟专用网）。可以将两台具有 IPsec 能力的路由器配置成对其间的所有通信数据进行加密，这种配置就像网络直接经由一条专线进行连接，这也是 VPN 字眼的由来。应当使用隧道模式是因为 IP 地址有可能是路由器后面的任何机器的 IP 地址。

11.9 协同工作：IPsec

让我们来考虑当一台主机希望向另一台它从没有与之通信过的主机传输数据包时所发

生的情况。由于只有在保护数据的 SA 建立起来后才能进行传输，所以主机会持有这个数据包直到完成建立所需 SA 所必须的 IKE 交换为止。一旦磋商好了 SA，主机就施加保护并传输这个数据包。

在收到一个数据包时，主机首先需要查看 SPI（在 AH 或 ESP 头信息中）并判断它是否与已知的 SA 对应。如果对应，它就执行所施保护的逆操作并将数据传送给 IP 栈。注意，所有这些行为对应用程序来说都可以是透明的。主机只需使用普通的网络调用，就会自动对数据施加保护和去保护。

策略

到目前为止，我们跳过了有关主机如何决定 IKE 应当磋商使用何种保护的问题。一般来讲有两种机制。一，系统管理员能够为主机设置策略。这些策略描述基于目的主机、端口和协议磋商使用何种类型的保护，因此应用无需对其自己的代码进行改动就能默认获取某种程度的安全。

在某些情况下，应用希望直接控制安全服务。为了利用这样一点，操作系统需要给套接字 API 提供允许应用控制套接字所磋商的安全的扩展。当然，必须对应用进行改动才能利用这些扩展。

当我们针对不同类型的通信有多种策略时，尽管我们有给定主机的 SA，但仍有可能使用不正确的保护类型。对于那种情况，IKE 必须磋商一个新的 SA。类似的，主机有可能收到一个具有与数据包类型不匹配的 SA 的数据包。对于这样的情况，IP 栈必须拒绝这种数据，并常常会产生某种类型的 ICMP 错误。在收到这样一种错误时，发送方应当磋商一个恰当的 IPsec SA。

11.10 IPsec 与 SSL 的对比

注意，尽管存在多种差别，IPsec 与 SSL 从概念上讲还是非常相似的。每种协议都有磋商密钥、参数的握手阶段还有保护通信的数据传输阶段。当在传输模式下使用 ESP 时，IPsec 的行为与 SSL 最为接近，它为通信数据但不包括任何 IP 头信息提供认证。然而，使用 IPsec，因为 TCP 消息是经过保护的，因此可以安全地使用 TCP FIN 来关闭连接，没有必要使用特殊的 IPsec 关闭消息。

在许多地方都更可将 IPsec 当成 SSL。可以用 SSL 来保护任何在 TCP 上传输的通信数据的安全，也可以使用 IPsec 来保护任何 IP 上运行的通信数据的安全，其中包括 UDP。SSL 要求只将套接字调用替换为 SSL 调用，不用对应用进行任何改动就能增加 IPsec 支持。即便应用想要成为 IPsec 感知的（aware），它仍然能够使用基本的套接字调用而不必像 SSL 那样由于 SSL 记录分帧的原因而操心处理不寻常的 I/O 行为，这对于大部分的网络代码都源于第三方的情况来说特别的原因。使用 SSL 要求改变所有这些代码，但 IPsec 本质上什么都不用改。

尽管 IPsec 不要求改动应用，但是它要求改动操作系统。执行 AH 和 ESP 的代码必须是 IP 栈的一部分。在大多数系统中，栈是位于操作系统内核中的。可以在用户空间中实现 ISAKMP/IKE 代码，但是必须能够为 SA 创建内核表条目，这样 AH/ESP 代码才能进行存取。因此，激活 IPsec 意味着安装一个全新的操作系统。

端点认证

与 SSL 不同，ISAKMP 要求对双方进行认证。这样是不方便的，因为在许多交互中，客户端的身份都是不重要的。然而，ISAKMP 允许基于共享密码的认证而 SSL 却不支持。注意，这些差异只不过是各种协议的设计选择，它们并不是将安全安置于传输层或 IP 层的固有结果。

中间节点

我们已经见过 HTTPS 可以通过代理以隧道形式传输 HTTPS，但是 TLS 上的 SMTP 必须直接与中继进行连接并将邮件投送给它们，这种差别是邮件投递的存储转发语义所造成的。对 IPsec 来说，以隧道形式通过中间节点进行传输甚至不算是一种选择。

当我们以隧道方式通过代理传输 HTTPS 的时候，仍得建立到代理的 TCP 连接。因此，代理必须能够读取 TCP 通信——它所不能读取的是应用层通信。因为 IPsec 在 IP 层提供保护，因此这种划分不再可行。最可能的方法就是给中间节点提供 IPsec。当涉及应用层的中间节点时，IPsec 只能提供链接保护，而 SSL 有时却能够提供端到端的保护。

处理中间节点的另一种方法就是让以 IPsec 保护的连接完全绕过中间节点并直接与目标主机相连。然而，一般来说这样做是不行的。防火墙代理会在那里对存取进行控制，而打通一个漏洞——即便是 IPsec——也违反了这一目的。此外，对于 SMTP 来说，中继是邮件系统的必要部分，因而不能绕过去。除了投递给中继以外，发送主机并不知道将邮件投递到哪里。

虚拟主机

在第 9 章和第 10 章，我们看到可以不用 IP 别名（aliasing）就能与 SSL 一起使用虚拟主机，前提就是要使用指示目标主机身份的升级磋商策略。然而，对于 IPsec 来说，由于在确立 IPsec SA 之前不会传输应用层协议数据，因此也就不存在这种升级磋商。所以，IP 别名是惟一处理虚拟主机的方法。

NAT

当有路由器执行网络地址转换（NAT）时，IPsec 就会彻底无效。这种技术允许大量的计算机使用少量的 IP 地址。每台机器都分配有其自己的私有 IP 地址，在传送时，路由器自动将私有 IP 编号转换成其中一个已分配的公共 IP 编号并在接收时再将公共编号自动转换为私有编号。

自然，由于 IPsec 依赖确定到特定主机的 SA，且这些主机由 IP 地址标识，所以如果你想从 NAT 路由器后面的主机上运行 IPsec，NAT 就会产生严重的问题（尽管如果只有少量主机使用 IPsec 的话，可以对路由器编程使其具有固定的 IP 地址）。与之对照，SSL 完全不受 NAT 的影响。存在广泛的（常常是尖刻的）有关 NAT 是好是坏的争论（[Rekhter1994、Lear1994、Rekhter1996]），因此根据你所持有的观点，这要么是一种 bug，要么是一种功能。不管怎样，NAT 部署广泛，所以在 NAT 路由器后面不能部署 IPsec 就很成问题。

什么是最重要的

反对 IPsec 的压倒多数的理由就是需要对 IP 栈进行改动。反对这种理由的理由是，为了保护应用的安全而无须对应用进行改动。在实际应用中，由于大多数应用实际上都不需要安全，并且改动操作系统非常不便，因此还是使用基于应用的 SSL 较好一些。然而，IPv6 要

求安全支持，因此如果 IPv6 最终在系统中广泛部署的话，可是个给 IPv4 增加安全的好时机。同样，Windows2000 内带对 IPsec 的支持，从而使 IPsec 对于 Windows2000 的应用来说是一种非常诱人的选择。

IPsec 最常见的用途就是创建 VPN，它十分胜任这项工作。对单个路由器进行升级相当直接，而且具有 IPsec 能力的路由器彼彼皆是。在你的网络上安装这样一种路由器可以方便地建立起到远程网络的安全连接。因此从一个网络到另一个网络的任何通信数据都会自动由网关路由器进行加密，这是构建 VPN 最直接的方案。

如果我们假定 IPsec 将来能够在最终用户的机器上广泛部署，那么可望会广泛应用于伺机（opportunistic）安全。正如我们在 TLS 上的 SMTP 中所看到的，在甚至没有对主动攻击进行任何防护的情况下，伺机磋商安全是非常有用的。这提供了在几乎没有配置或管理开销的情况下防止被动嗅探攻击。因此，虽然很难将 HTTPS 替换为 HTTP/IPsec，但是将 SMTP/TLS 替换为 SMTP/IPsec 却是相当诱人的。

11.11 安全 HTTP

S-HTTP (Secure HTTP) [Rescorla1999a] 提供了一种用来保护以 HTTP 发送消息的安全的语法。与 HTTPS 不同，它在很大程度上忽略了单条 HTTP 消息的内容和边界，而 S-HTTP 把每个 HTTP 请求和响应都当成单一的单元分别加以保护。这样就使得 S-HTTP 能对客户端与服务器之间不同的消息施加不同的保护，同时提供消息层的数字签名和不可抵赖性。

S-HTTP 由两个主要部分构成，即用于封装和保护单条消息的消息格式，以及允许客户端与服务器表达它们有关如何对数据进行保护及提供密钥资料的磋商语法。

全揭密：我曾是 S-HTTP 的主要设计者之一，并为一家销售 Secure HTTP 工具箱的公司工作了几年。这使我对 Secure HTTP 和 SSL 的相对优势有比较全面的看法，同时也带来了大家对我的倾向性的怀疑。不管怎样，我已尽可能客观地展示了它们优点和缺点。

● 消息格式

S-HTTP 消息格式基于加密消息语法 (CMS)，在 RFC 2630 中有所描述[Housley1999b]。CMS 是 PKCS #7[RSA1993c] 的变种，它设计与 Secure MIME (S/MIME) 一起使用。一般来讲，S-HTTP 消息看起来像是 HTTP 请求和响应，但主体是 CMS 消息。受保护的主体包含客户端或服务器所发送的实际 HTTP 请求和响应。因此，头信息与消息体是受保护的。

● 加密选项

S-HTTP 中的一项慎重的设计目标就是要与 HTTP 一样的消息模型。因此，虽然 SSL 要求客户端与服务器在完成握手时经过多个回合，但 S-HTTP 却不要求这样。相反，包含 S-HTTP 链接的 Web 页面还包含磋商信息。这种信息足以让客户端通过对链接进行解析而不用先与服务器交互就能判断对某个请求施加何种保护。类似的，S-HTTP 客户端在它们的请求头信息中放置磋商信息，这种信息足以告诉服务器应如何对响应进行保护。

● 我们的方案

我们的意图是将在第 9 章所讨论的 HTTPS 与 S-HTTP 进行比较。为了做到这一点，我

们需要理解 S-HTTP 工作的有多么好才能欣赏这种差异。因此，我们开始先对 S-HTTP 进行概括性的描述。首先讨论 CMS，接着讨论 S-HTTP 消息的格式。然后，讨论磋商与消息传递如何协调工作来保护 HTTP 数据传输的安全，并展示一个相关的例子。

一旦对 S-HTTP 有了扎实地理解，我们就可以讨论它相对 HTTPS 的优点和缺点。总地来讲，S-HTTP 面向消息的特性提供了比 HTTPS 更强的灵活性。它能够提供消息级别的签名和不可抵赖性，而 HTTPS 则不行。它还可以更为妥善地与代理和虚拟主机进行交互。S-HTTP 的主要缺点就是它实现起来要复杂得多，无论在客户端还是服务器上均是如此。

11.12 CMS

CMS 是使用加密来保护安全的一种相当典型的协议，它可以为任何内容提供加密和签名。每条 CMS 消息都有一个描述所施加的加密增强形式的类型。CMS 消息可以递归地加以封装以便施加多种类型的增强防护（enhancement）。CMS 定义了六种基本类型，即 Data、SignedData、EnvelopedData、EncryptedData 和 DigestedData。然而 S-HTTP 只使用 SignedData 和 EnvelopedData，因此我们只对这些类型进行讨论。

SignedData

SignedData 消息由一些内容以及一个或多个对内容的签名组成。与往常一样，先计算内容的摘要，然后用发送方的公用密钥对这个摘要进行签名。这种消息可以——通常如此——包含相应的对签名消息使用的密钥进行授权的证书，消息还可以包含检查证书有效性所需的 CRL。

还可以有无内容的 SignedData 消息，这被称作拆离的签名（detached signature），它表示对某些必须外定的数据的签名。

EnvelopedData

EnvelopedData 消息包含加密数据。数据是以随机产生的对称内容加密密钥（CEK）来加密的。加密的内容封装在一个包含一个或多个 RecipientInfo 分组的包裹中。每个 RecipientInfo 分组都包含为给定接收者加密的 CEK。

可以用三种方法中的任意一种对 CEK 进行加密。如果使用 RSA 的话，就使用接收方的公用密钥进行加密。如果使用 DH，则使用发送方与接收方成对的（pairwise）DH 共享密码进行加密。注意，你不必直接使用成对的（pairwise）DH 共享密码对消息进行加密，

因为这要求为每个接收者都重新对整个消息进行加密。使用成对密码对 CEK 进行加密可以让发送方只为每个接收者产生一个新的包裹（wrapped）CEK。最后可以只用由发送方与接收方所共享的对称密钥加密密钥（KEK）对 CEK 进行加密。如何共享这样密钥超出了 CMS 的范围。然而，S-HTTP 提供了一种确立这种密钥的方法，我们将在第 11.14 节进行讨论。

签名与加密

CMS 不提供同时提供签名与加密的内容类型。要想同时施加这两种增强措施（enhancement），则必须采取递归增强的内容。我们对数据进行签名来产生 SignedData，然

后再使用 SignedData 作为加密过程的输入来产生 EnvelopedData。要想读取这样一条消息只需反转这一过程即可。

11.13 消息格式

与 HTTPS 不同，S-HTTP 消息使用与 HTTP 消息相同的端口。这样设计是为了服务器可以轻易分辨出 S-HTTP 请求并在处理请求之前将增强措施去掉。为了做到这一点，S-HTTP 使用一种特殊的请求方法：Secure。因为 Request-URI 有可能包含敏感信息，所以它被替换成“*”。图 11.6 描述了一种典型的 S-HTTP 请求。

```
Secure * Secure-HTTP/1.4
Content-Type: message/http
Content-Privacy-Domain: CMS
```

下面删除了二进制的 CMS 消息

图 11.6 一条 S-HTTP 消息

Content-Privacy-Domain: CMS 行指示（删除的）请求体为 CMS 消息。这是必须的，因为 S-HTTP 还支持另外一种称作 MIME 对象安全服务（MOSS）的消息安全机制（尽管不那么流行）。当对 CMS 消息进行解包时，它的内容就是在没有使用 S-HTTP 的情况下客户端本要发送的 HTTP 请求（这就是 Content-Type: message/http 的意思）。

S-HTTP 响应遵循与 S-HTTP 请求类似的格式。由于响应的成功或失败信息是潜在敏感的，所以我们将整个 HTTP 响应包裹起来，状态行总是显示为：

Secure-HTTP/1.4 200 OK

S-HTTP 原先是用 PKCS#7 来设计的，然后又改编为 CMS。尽管 CMS 支持使用 MAC 的对称消息认证，但 PKCS#7 却不支持。因此 S-HTTP 还提供了一种允许将 MAC 追加到消息上的 MAC-Info 头信息行。

11.14 加密选项

正如我们在第 9 章所讨论的，可以通过模式 https 将 HTTPS URL 与 HTTP URL 区分开来。然而，这是提供给客户端的惟一信息。最终使用哪种加密套件的决定是在 SSL 握手过程中做出的。S-HTTP 也有它自己的模式：shttp。然而，其中的大部分信息都是在加密选项中给出的，这些选项指示接收方期望发送方在发送消息时使用的变换种类。

服务器的选项通常放置在相关的 HTML 锚点（anchor）中。RFC 2659[Rescorla1999b] 描述了这一过程。客户端的选项放置在内层头信息中——即请求的 HTTP 头信息而不是 S-HTTP 头信息中。当客户端解析 URL 时，它首先找到对应的选项并使用它们来判定发送消息的类型。当服务器发送响应时，它首先在客户端请求的 HTTP 头信息中检查客户端的选项。

S-HTTP 没有任何真正意义上的磋商，而是发送方将它自己的首选项与接收方的首选项加以合并来提供一组可接受的增强措施。然后它再将这些增强措施施加给消息并将消息发送

出去。除了加密选项的位置以外，客户端与服务器在这方面的行为是相同的。

加密选项包含两种不同种类的信息：密钥资料和磋商头信息。密钥资料包含密钥、证书等内容，在保护消息时使用。磋商头信息包含使用何种保护的首选项，具体包括算法和密钥长度。

密钥资料

密钥资料选项允许一方给另一方发送用来响应该条消息的密钥资料。因此，服务器可以在 HTML 页面的链接中包括自己的证书，这样客户端就能使用证书来对那个链接的请求进行加密。S-HTTP 还允许使用 Key-Assign 行的对称加密密钥。因为 RSA 加密开销昂贵，所以任何客户端和服务器对最好只执行一次这样的计算。因此，如果客户端给服务器加密信息，它就可以给服务器提供一个对称密钥，然后当服务器响应时再用它作为 KEK 对 CEK 进行加密。

磋商头信息

在 SSL 中，所有可能的增强措施选项都整合在一起：密钥交换算法、认证算法、加密算法和消息认证算法都是由加密套件来定义的。与之对照，S-HTTP 有着更多的选项，它们多少是以正交（独立的）方式来指定的。

S-HTTP 允许磋商使用三种不同的保密增强措施：对内容的数字签名（sign），对内容的加密（encrypt）以及对内容的对称消息认证（auth）。这些增强措施可以以任意的组合来提供，尽管同时提供 auth 和 sign 看起来有些重复。

对于其中的每一种增强措施，可以磋商算法。此外，可以独立于内容加密算法来磋商密钥交换算法（公用密钥还是对称密钥），还可以在施加数字签名的情况下提供有关发送方使用何种密钥进行签名的信息。

最后，磋商头信息可以包含有关代理在发送时将使用哪种增强措施的信息。因此，服务器可以告诉客户端可望在对请求的响应中包含什么种类的消息。然后客户端应当将服务器的真实响应与加密选项中所保证的行为进行比较。图 11.7 描述了服务器所发送的单个磋商头信息的例子。

```
SHTTP-Symmetric-Content-Algorithms: orig-optional=DES=CBC, DES-EDE3-CBC;
Recv-require=DES=EDE3-CBC
```

图 11.7 磋商头信息

图 11.7 中的头信息说明必须使用 3DES（recv-require=DES-EDE3-CBC）来解析这个链接，但是那个服务器愿意使用 DES 或 3DES（orig-optional=DES-CBC,DES-EDE3-CBC）进行响应。

S-HTTP 定义了 8 种这样的磋商条目，因此加密选项有可能相当长。为了解决这种问题，RFC 2660 提供了默认值。如果一个选项的值与默认值匹配，那么就可以省略这个选项。

总而言之，S-HTTP 允许磋商下列参数：

- 消息格式——CMS 或 MOSS
- 证书类型——X509、PKCS7 包裹的
- 密钥交换算法——DH、RSA、Inband、Outband

- 签名算法——RSA、DSS
- 消息摘要算法——MD5、SHA-1、HMAC
- 对称内容加密算法——DES、3DES、DESX、CDMF、IDEA
- 对称密钥加密算法——DES、DES、DESX、CDMF、IDEA
- 保密增强措施——签名、加密、认证或组合

注意，没有与 CMS 一起使用 DES、DESX 或 CDMF 的规范。然而，实际上只存在一种显而易见的插入新的内容加密算法的方法。通过宣传（advertising）对这些算法的支持，S-HTTP 实现隐含表（implementation）对这种方案的支持。

11.15 协调工作：S-HTTP

最容易明白 S-HTTP 工作原理的方法就是查看一个例子。图 11.8 描述了一个带有 Secure-HTTP 链接的页面。客户端是如何得到这个页面的并不重要，通过 HTTP 获取的。

```
<CERTS FMT=PKCS-7>
Certificate deleted
</CERTS>

<A DN="CN=Test Server, O=RTFM, Inc., C=US"
CRYPTOPTS="
    SHTTP-Privacy-Enhancements: recv-required=encrypt;
    SHTTP-Key-Exchange-Algorithms: recv-required=RSA;
    SHTTP-Symmetric-Content-Algorithms: recv-required=DES-EDE3-CBC"
HREF="shttp://www.rtfm.com/test.html.>
Click here to dereference</A>
```

图 11.8 带有 S-HTTP 链接的页面

元素 CERTS 包含以 base64 编码的 PKCS#7 证书链，我们把它从这个例子中删掉了。锚点包含有关的加密选项。DN 字段包含服务器的标识名。注意，服务器的证书与标识名是独立提供的，而 CERTS 元素只是建议性质的。DN 属性是用来找到服务器密钥的字段。

CRYPTOPTS 属性包含磋商头信息。这里，服务器要求客户端进行加密，使用 RSA 完成密钥交换以及 3DES 来完成对称加密。最后，HREF 属性包含资源的 URL，和往常一样，但使用的是 shttp 模式。

当客户端解析链接的时候，它对请求进行加密，依照要求使用 RSA 完成密钥交换并用 3DES 完成加密。此外，它还使用 Key-Assign 头信息给服务器提供对称密钥。图 11.9 描述了加密的请求。这个请求（以及本章往下描述的所有消息）在 crypto-vision 中描述。我们不是以大二进制对象的形式来显示加密的内容，而是展示了当你对加密内容进行解密后所看到的内容。内容以定宽斜体来显示，以便与消息的其他部分区分开来。

```
Secure * Secure-HTTP/1.4
Content-Type: message/http
Content-Privacy-Domain: CMS
```



图 11.9 S-HTTP 请求

注意，头信息本来应当位于 HTTP 请求中，诸如 Security-Scheme 和 User-Agent 这样的信息位于内层的加密请求中。同样，这些头信息是潜在敏感的，因此必须对其进行保护。

最后四行头信息为 S-HTTP 碰商头信息，Key-Assign 行提供带内（inband）密钥。这个密钥可以用来让服务器使用加密消息进行响应而不用再执行一遍 RSA 操作。它也可以使客户端不用掌握私用密钥就能接收加密消息，就像使用 SSL 时那样。客户端应当为每个请求产生一个崭新的密钥，且这个密钥的标签（label）为 1。

最后三行头信息告诉服务器必须使用 3DES 对响应进行加密，它必须使用在请求中传送过来的带内（inband）密钥包裹 CEK，使用 3DES。它还必须提供整个消息上的 MAC。

最后，图 11.10 描述了服务器对这条消息的响应，同样也在 crypto-vision 中描述。就像先前所指导的，服务器使用密钥 `inband:1` 作为 CEK 对发送给客户端的消息进行加密。这隐藏在 CMS 包裹（wrapper）中。服务器还使用 `inband:1` 来计算 MAC-Info 行中出现的 HMAC 值。注意，真实的 HTTP 头信息是加密并且计算 MAC 的。它们潜在包含敏感数据，所以必须加以保护。因此一个 S-HTTP 响应的 S-HTTP 状态行为 OK 而内部的 HTTP 状态行指示出错的情况是可能的。

```
Secure-HTTP/1.4 200 OK
Content-Type: message/http
MAC-Info:31ff8122,rsa-sha-hmac,
          a51d612e5f3d0fbb3d8837ca351fce879e5e3a6,inband:1
Content-Privacy-Domain: CMS

HTTP/1.0 200 OK
Security-Scheme: S-HTTP/1.4
Content-Type: text/html

Congratulations, you've won.
```

图 11.10 S-HTTP 响应

客户端认证

S-HTTP 通过对请求消息进行认证来对客户端进行认证。存在两种对消息进行认证的选择：即消息上的数字签名和 MAC。数字签名提供不可抵赖性，而 MAC 只提供快速的完整性和发送方认证检查。数字签名将消息的发送者与发送者的证书绑定在一起。MAC 可以让

接收方知道消息是由掌握 MAC 密钥的某个人发送的。

引用完整性

保持 S-HTTP 的引用完整性非常简单。由于包含引用的页面还包含加密请求所需的加密信息，所以客户端可以简单地使用该信息。因此不要求服务器的证书实际与服务器的 DNS 名匹配，所要求的就是检查服务器的证书验证通过。客户端是从证书来了解服务器的身份的。

由于每一种消息都是单独进行认证的，因此我们需要提供某种将引用与服务器的响应绑定在一起的方法，否则攻击者就能够向客户端重放老的响应。完成这种工作的方法就是让客户端使用 Key-Assign 来提供一个密钥，然后要求服务器使用那个密钥对其响应进行加密或计算 MAC。要知道密钥就要能够对原来的请求进行解密，即要知道服务器的私用密钥。由此就响应将与请求绑定在了一起。

自动选项生成

S-HTTP 最著名的功能就是任何给定的请求，即便是那些请求同一站点资源的请求，都可能使用不同组的加密增强措施。尽管这提供了极大的灵活性，但也给管理造成了不小的负担。即便使用默认情况，S-HTTP 碰商头信息都难以用手工编写。更糟的是。将选项编写到 HTML 中就意味着如果你的策略发生改变，就不得不对所有指向受影响资源的页面进行编辑。

一种更好的方法就是让服务器自动创建选项。这样做要求服务器使用能够识别链接的原始 HTML 解析器。当发现链接时，服务器接着找到与资源相关的存取控制策略并自动沿那个链接产生合适的选项。不幸的是，早期的 S-HTTP 服务器并不支持这种功能，从而使得管理起来非常麻烦。

注意图 11.9 中的 Security-Scheme 头信息字段，这是 S-HTTP 新增的 HTTP 头信息。它允许客户端告诉服务器其具有 S-HTTP 能力。这允许服务器只重写发往具有 S-HTTP 能力的浏览器的页面。尽管 S-HTTP 对 HTML 的修改当在兼容 HTML 的浏览器中查看时应当是不可见的，然而一些浏览器还是将其作为页面的一部分显示出来。这种头信息可以阻止包含 S-HTTP 链接的页面当在不支持 S-HTTP 的浏览器中显示时包含那些无关的信息。

无状态操作

我们已经描述了客户端如何使用 Key_Assign 头信息来为服务器提供用于服务器响应的对称加密密钥。显然，服务器能够在其加密选项中完成同样的工作，但是这通常要求服务器记住密钥。由于我们必须应对多个服务器进程，所以这就意味着要安排在这些进程之间沟通密钥。我们在使用 SSL 会话缓冲时也遇到了类似的问题。

然而，对于 S-HTTP 来说可以不创建服务器状态就处理这种问题，Key-Assing 头信息允许发送者给密钥分配一个任意的名字。我们可以利用这种名字来避免存储临时密钥。所有的服务器进程都共享一个单一的主 CEK。然后，当创建与 Key-Assing 一起使用的新的临时密钥时，我们安排从标签产生那个密钥。最简单的做到这一点的方法就是使用主 CEK 对密钥进行加密并使用那个结果作为标签。

$$\text{Label} = E(\text{MasterCEK}, \text{Key})$$

另外我们可以随机产生标签并使用它来产生密钥，就像下面这样：

$$\text{Key} = \text{HMAC}(\text{MasterCEK}, \text{Label})$$

我们可能会设想在 SSL 中通过使用加密的主密码作为会话 ID 来应用这种技巧。不幸的是，由于两种原因这样做是不行的。首先，会话 ID 需要与主密码一般长，但是它只是 32 个字节，而主密码为 48 个字节。其次，SSL 要求我们不能恢复非正常关闭的会话。这就要求我们能够使会话无效，而它要求进程间通信。注意，使会话无效并不能增加多少安全性，而且许多实现根本就不这么做。然而主要的障碍就是会话 ID 太短了。

11.16 S-HTTP 与 HTTPS 的对比

一般来讲，在离应用层协议越近的地方提供安全，越能提供具有更强安全与更好灵活性的服务。然而这种灵活性常常是以复杂性和实现花费为代价的。原则上讲，获取 S-HTTP 使能的 Web 服务器上每一种资源都可能要求不同的加密增强措施。在实际应用中，很少需要这么强的灵活性，然而需要三至五种策略还是相当常见的。对于这么多的策略，连 HTTPS 都很难处理，通常需要屈就于像虚拟主机或多端口（每个端口针对一种策略）这样的变通措施。正如我们在第 9 章所看到的，如果一个 HTTPS Web 站点只有一部分内容要求客户端认证，那么存取那些内容就会要求一次完整的再握手。对 S-HTTP 来说，却只是改变相应链接选项那么简单。

然而，S-HTTP 的灵活性有着高昂的代价。必须做出大量的决定对管理员来说是可怕的。使用良好的服务器设计和用户界面可以将这种管理负担减小到最少，但这要比编写 HTTPS 使能的服务器需要的编程工作多得多。

不可抵赖性

无法通过 HTTPS 获得的一种 S-HTTP 功能就是不可抵赖性。由于 SSL 对各种 HTTP 消息的边界一无所知并且使用 MAC 来实现消息认证，所以没有办法提供发出某个给定请求或响应的证明。与之对照，由于 S-HTTP 提供整个消息的签名，所以对 S-HTTP 来说，这只是对恰当的消息进行签名的事。通过设置合适的选项让客户端和服务器请求这样的服务也同样简单。

S-HTTP 的签名功能还可以用来使服务器能够缓存预签名的文档。这允许服务器执行一次签名并将同样的文档反复提交给多个客户端，这对于需要完整性保护但无须保密的静态对象来说是非常有用的。

代理

在最简单的情况下，S-HTTP 代理的行为与 HTTPS CONNECT 代理的行为极为相像。然而，由于 S-HTTP 的消息结构对代理来说是透明的，一种更为复杂的代理可以利用这种信息。例如，代理可能注意到消息边界并关闭空闲的连接。一种真正复杂的代理可以对使用 S-HTTP 获取的静态、非加密文档进行缓存。这样，S-HTTP 代理就有可能对诸如程序或长期数据文件这样的签名内容进行缓存。

虚拟主机

S-HTTP 与虚拟主机的交互是简单的。因为 S-HTTP 在加密选项中提供服务器的 DN，

所以没有必要针对不同的虚拟主机使用不同的证书。即使需要这样做，服务器也能够很容易地对消息进行解密，因为 CMS 消息清楚地指出了接收方的 DN。一旦对消息进行了解密，服务器就能够存取 Host 头信息并执行相应的动作。

● 用户体验

这种将加密选项放置在锚点中的策略的一个缺点就是 URL 立刻变得不足以用来获取资源，客户端还需要从锚点中了解加密选项。因此，用户可以直接在浏览器中键入 https:URL，但直接键入 shttp:URL 则没有多大的用处。RFC 2660 描述了一种浏览器可以用来向服务器索要通用加密选项的变通措施，但是由于服务器不知道请求的是什么资源，所以在这种情况下 S-HTTP 并不比 HTTPS 灵活多少。

当最初引入 S-HTTP 和 HTTPS 的时候，shttp:URL 的不足似乎是主要缺点。用户已经习惯直接在浏览器中键入 URL，而且如果需要对整个 Web 站点进行保护，则假定人们会键入安全的 URL 进行访问。然而事实上大多数 Web 站点都只对站点中非常有限的一部分内容使用安全并使用 https:链接将非安全的部分转换为安全的。因此，由于 URL 几乎总是出现在 HTML 页面中，所以我们可以在页面中安置加密选项。

● 实现的难易程度

实现 S-HTTP 和 SSL 的难度大致相当，而将其与现有软件集成的困难程度却不同。我们已经提到 S-HTTP 要求服务器的强力支持才能重写页面来插入加密选项，而 HTTPS 不需要丝毫这样的工作。然而，注意这句话的意思是说 Web 设计者需要显式地编辑页面才能使资源可以通过 HTTPS 存取。使用一种重写策略，就像 S-HTTP 所应当使用的那种，管理员可以只改变存取控制设置，而服务器将会相应的自动将页面转换为安全存取。

S-HTTP 比 HTTPS 还要求更多的客户端支持。主要需要对 HTML 解析器进行修改才能允许抽取加密选项，HTTPS 不需要丝毫这样的工作。还因为 S-HTTP 允许更多的操作模式，用户界面应当精确地反映对给定消息进行了何种类型的处理，而 HTTPS 可以更直观简单地指示安全或不安全。

这是一个 bug 还是一项功能则依赖于你所持有的观点。对 S-HTTP 来说，之所以不足以说是安全或不安全是因为 S-HTTP 具有远为灵活的安全模型。因此，如果你觉得 S-HTTP 所增加的附加功能值得拥有的话，你就需要某种将其表达给用户的方法。另一方面，如果你认为这种功能是不必要的，那么它们只会增加额外的用户界面处理负担。

总地来说，实现 S-HTTP 比 HTTPS 要求更多的编程工作。此外，这种编程工作无法隔离在一种工具箱中，而是要求与浏览器和服务器紧密集成。这种差别并不奇怪，因为 S-HTTP 远比 HTTPS 与 HTML 和 HTTP 结合得更为紧密。

● 什么是最重要的

从技术优势上考虑，S-HTTP 更强的灵活性以及与 Web 的紧密集成既是一种负担也是一种优势。它允许 S-HTTP 提供 HTTPS 所不能提供的功能，如不可抵赖性和预签名数据。它还意味着 S-HTTP 与虚拟主机和代理的交互远为干净利落。然而，它也意味着实现 S-HTTP 要比实现 HTTPS 困难得多。

此外，实现 HTTPS 的微妙之处在于使其做到彻底的安全。实现 S-HTTP 的微妙之处在于使其具有可用性。因此，没有经验的 HTTPS 实现可能无法正确地处理关闭和引用完整性。

没有经验的 S-HTTP 实现有可能要求管理员以一种令人讨厌的方式手工编辑 HTML。由于用户常常把方便性看得比安全性更重，所以没有经验的 S-HTTP 实现要比没有经验的 HTTPS 实现有价值得多。

在 1995 和 1996 年，人们对 HTTPS 或 S-HTTP 是否会成为占主导地位的 Web 安全协议持有很大怀疑。而现在则不再有疑虑了。HTTPS 是无可争辩的胜利者，不管它们各自的技术优势如何，每种主要的浏览器和服务器都实现了 HTTPS，而没有实现 S-HTTP。因此，实际上无法选择使用 S-HTTP 还是 HTTPS。

11.17 S/MIME

S/MIME 为 E-mail 消息提供面向消息的安全服务。在 Web 出现以前，E-mail 曾是——而且可以说现在仍是——最重要的网络服务。因此进行了许多次尝试对其进行标准化的工作。光是 IETF 自己就有不少于四种的 E-mail 安全标准：PEM、MOSS、OpenPGP 和 S/MIME。所有这些协议本质上都采用相同的方法。它们将 E-mail 消息当作单个对象并为那个对象提供安全服务，这些标准之间的差别在很大程度上都是具体消息格式和信任模型之间的差别。

S/MIME 版本 2[Dusse1998]原先是由 RSA 实验室开发的，使用 RSA 的 PKCS #7[RSA1993c]加密消息格式。PKCS #7 支持仅限于密钥交换的 RSA，因此 S/MIMEv2 只支持 RSA。在组建 IETF S/MIME 工作组对 S/MIMEv3 进行标准化时[Ramsdell1999]，其中的一个主要目标就是增加对其他算法的支持。因此，对 PKCS #7 进行修订而创建了 CMS[Housley1999b]，它还支持其他的密钥交换和签名算法。

我们的方案

我们的意图是比较通过 SMTP/TLS 来保护 E-mail 传输安全以及使用 S/MIME 来保护 E-mail 传输安全之间的差别。跟 S-HTTP 一样，我们先展示足够的细节来解释 S/MIME 的工作原理。我们已经在第 11.12 节讲述了 CMS，但是 S/MIME 中存在一些使用 CMS 的微妙之处，特别是在对数据进行签名而不加密的情况下。最后讲述 S/MIME 的算法选择。因为 E-mail 是存储转发性质的，所以确保发送者知道使用哪一种算法需要某些特殊的考虑。

一旦掌握了 S/MIME 的知识，就可以讨论它相对于 TLS 上 SMTP 的优势与不足。正如我们在第 10 章所讨论的，TLS 上的 SMTP 实际上不能真正胜任保护 E-mail 安全的工作。然而，S/MIME 却能够完成得很好。TLS 上的 SMTP 唯一真正的优势就是不要求那么多的客户端支持。

11.18 S/MIME 的基本格式

MIME 允许 E-mail 消息通过使用 Content-Type 行来承载任何内容。S/MIME 使用 application/pkcs7-mime 内容类型指示内容类型为 CMS 封装的 MIME 消息。也就是说，当对 CMS 主体进行解包时，它本身包含一个 MIME 消息体。图 11.11 展示了这样一个例子，同样也是在 crypto-vision 中描述的。

```
From: ekr@rtfm.com
Subject:Test message
Content-Type: application/pkcs7-mime; smime-type=enveloped-data

Content-Type: text/plain

This is an encrypted message.
```

图 11.11 加密的 S/MIME 消息

注意，解密部分中所出现的消息的 `Subject` 行。因此，`Subject` 行没有以任何形式进行加密保护。实际上，普通的邮件头信息都没有加以保护。如果要对其进行保护，则必须出现在内层内容的头信息部分。

11.19 只进行签名

图 11.11 描述了一条加密消息。最简单的准备一条签名消息（signed message）的办法除了使用 CMS `SignedData` 类型外，跟以前所做的工作完全一样。当然，参数 `smime-type` 现在是 `signed-data`。这种方案简单但却有重大缺陷。消息不能由没有 S/MIME 能力的客户端阅读。根据定义，加密的消息要求使用具有加密能力的客户端来阅读。对于精心划分结构的签名消息而言则不是这种情况。

如果对消息进行签名但不加密，那么最好没有 S/MIME 能力的接收者也能阅读。这样就可以鼓励发送者不管接收者是否具有 S/MIME 能力都进行签名。人们可以给多个接收者发送签名消息——当其中一些能够对消息进行验证时就是一种有用的功能，但是必须能够被所有的接收者读取。

然而，只是使用 `application/pkcs7-mime` 将消息表达为 CMS `SignedData` 就达不到这种目的。大多数 MIME 阅读器只是将具有不可识别类型的内容保存在磁盘上。此外，即便阅读器试图显示数据，由于 CMS 为二进制格式，要想将 CMS 包裹体与数据区分开来也是非常不方便的，标准工具常常压根就不显示二进制数据。

Multipart/Signed

解决这种问题的方案就是利用 CMS 拆离的签名（detached signature）。我们还需要使用不同的包裹方法（wrapping method），`multipart/signed`。除了支持任意的数据内容类型外，通过使用 `multipart` 类型，MIME 还允许单个消息中存在多个内容类型。`multipart/signed` 消息包含两个部分。第一部分是在没有 S/MIME 包裹情况下出现的实际对象，第二部分是消息上的 CMS 拆离签名。图 11.12 描述了这样一个例子。

--boundaryYYY 行指示各主体部分之间的隔断。注意，任何 MIME 兼容的代理都能够发现第一部分主体内容为 `text/plain` 类型并将其显示给用户。它可以忽略第二主体部分或是提出将其保存到磁盘上的请求。对于任何一种情况，用户都能够阅读签名的消息，而签名不会碍事。即便用户没有 MIME 兼容的邮件程序，如果签名数据与二进制形式的 CMS 包裹体分开的话，它也有更大的机会看到消息内容。

```

Content-Type: multipart/signed; protocol="application/pkcs7-signature,,";
micalg=shal; boundary=boundaryYYY

--boundaryYYY
Content-Type: text/plain

This is a clear-signed message.
--boundaryYYY
Content-Type: application/pkcs7-signature
Content-Transfer-Encoding: base64

Signature deleted
--boundaryYYY--

```

图 11.12 multipart/signed 消息

S/MIME 与 S-HTTP

S/MIME 封装本质上提供与 S-HTTP 的消息格式相同的功能。S-HTTP 提供自己的消息格式的原因就是它是在 S/MIME 之前设计的。如果 S-HTTP 是在现在设计的话，S-HTTP 磋商则很有可能使用 S/MIME 封装，而且这会是最好的选择。

11.20 算法的选择

由于 E-mail 的存储转发本质不可能让发送者与接收者磋商算法和密钥。所以原则上，发送者与接收者有可能以前从未通信过。发送者只需要掌握接收者的公用密钥，就可以从目录服务中获得。

然而，在发送第一条消息时，发送者有机会向接收者传达它希望接收者应答时所使用的增强措施种类。为了确保出口与国内客户端的互操作性，这一点特别重要。因此，S/MIME 签名消息可以包含一种指示签名者首选项的 capability（能力）属性。

能力

SMIMECapabilities 属性用来指示发送者支持的加密算法。它只是一个发送者希望宣传自己所支持的算法列表，它至少可以包含下列算法类型：

- 签名算法
- 对称加密算法
- 密钥加密算法（指示如何加密 CEK 的算法）

能力属性是一种相当通用的宣传用户首选项的机制。算法故意以优先选择的顺序列出，这样用户不但能够描述它所支持的算法，而且还能描述它希望使用的优先顺序。使用 preferSignedData 能力，甚至可以表示你希望人们对给你发送的消息进行签名。实现应当忽略它们不能识别的标识符，将来可以对这种集合进行扩充。

算法选择

我们实际必须关心的唯一算法选择就是加密算法。密钥确立算法由接收者密钥定义，因

此这里别无选择，除非接收者有多个密钥，但是未必会出现这种情况，在这种情况下发送者可以任选一种。从理论上讲有可能存在使用何种摘要算法的问题，但在实际应用中普遍支持的几乎全是 SHA-1。然而，在加密算法支持方面存在相当大的波动。特别的，存在大量可出口的只支持 RC2-40 的客户端。由于明显的原因，重要的是使用接收者所支持的算法对你的消息进行加密。RFC 2633 中详细描述了三种情况。

已知能力。在发送者知道接收者能力的情况下，情况很简单。它应当选择接收者所支持的一种算法。出于关照，RFC 2633 上讲应当使用接收者最满意的加密算法。然而，发送者有可能选择使用一种接收者不太喜欢但发送者更喜欢的加密算法。如果存在目录服务的话，发送者很有可能在获取证书的时候取得接收者的能力。

未知能力，知道使用加密。在接收者从发送者收到加密消息但对能力一无所知的情况下，使用接收者在消息中使用的同一种加密算法是相当保险的。

未知能力。S/MIME v3 要求支持 3DES，因此如果发送者知道接收者支持 S/MIME v3 的话，最可靠的就是使用 3DES 来发送。然而，S/MIME v2 要求支持 RC2-40，而许多出口客户端只支持那一种算法，因此使用 RC2-40 可以获得绝对最大的可互操作性（要付出巨大的安全代价）。一般来讲，RC2-40 是弱强度的，最好使用 3DES。

能力发现

这种规则的结果就是通过几封 E-mail 交换，各方形成对各自能力的理解，接着进行某种类型的算法磋商。考虑 Alice 和 Bob 两方交互的情况。他们都有 RSA 密钥并支持 3DES，但是不知道对方的能力。他们发送的头三条消息在图 11.13 中描述如下。



图 11.13 使用 S/MIME 进行能力发现

Alice 想要绝对确信 Bob 能够读取她发送的第一条消息，因此她使用 RC2-40 来发送这条消息（她还可以以明文形式来发送这条消息）。她支持 3DES，因此还包含了指示她支持这种能力的能力属性。这样，当 Bob 回答时，它使用 3DES 进行加密。现在 Bob 甚至不用包含让 Alice 了解它支持 3DES 的能力属性——而只管使用这种能力即可——所以 Alice 最终的消息也使用 3DES。

11.21 协调工作：S/MIME

现在我们看到了 S/MIME 的各种组成成分，让我们查看一下它们是如何提供一致的安全

服务集合的。设想 Alice 想要给 Bob 发送一条消息。她已经有了 Bob 的证书以及他的能力，估计是通过某种目录服务获得的。

● 端点识别

Alice 已经从目录中获取了 Bob 的证书并进行了验证，但是她怎么知道它就是 Bob 的证书呢？大家应该还记得我们在 SSL 中经常遇到的将证书与 DNS 名进行比较的问题。这里的解决方案也是类似的：证书可以包含 E-mail 地址，要么作为标识名的一部分，要么位于 subjectAltName 扩展的 emailAddress 值中。这样，Bob 的证书就是在其中一个位置包含 Bob 的 E-mail 地址的那个。

● 消息发送

有了 Bob 的证书，产生只能由 Bob 读取的消息非常简单。Alice 使用随机产生的 CEK 对消息进行加密并用 Bob 的公用密钥跟往常一样加密 CEK。她可以选择在加密之前对消息进行签名，然后再将消息通过任意的通道传输给 Bob。此刻使用普通的非加密 SMTP 完全是安全的。

● 发送者认证

一旦 Bob 收到了消息，他就能够完成两件事情：读取消息以及对消息和发送者进行认证。读取消息非常简单。与往常一样，他使用自己的私用密钥对 CEK 进行解密，然后再使用 CEK 对消息进行解密。自然，所有这些工作都是由他的 MUA 自动完成的。

对消息签名进行验证同样直接了当。首先 Bob 验证消息的加密身份。然而，正如我们在第 10 章所看到的，E-mail 消息在 From 头信息中也包含非加密的消息发送者指示。因为头信息是发送者控制的，接收者的 MUA 必须验证它是否与用来对消息进行签名的证书相匹配。如果不匹配，则 MUA 必须通过某种方式通知用户，以阻止其受到欺骗。

● 多个签名者

在某些情况下，最好让消息由多个签名者进行签名。考虑消息实际是由两个人发送的情况。在普通信笺里，他们两个都会签上自己的名字。类似的，对于数字签名的消息来说，我们也想让两个人进行签名。这对 S/MIME 来说是小事一桩，即使签名者完全位于不同的网络和计算机上，因为 CMS 支持同一条消息上的多个签名。

● 多个接收者

甚至比多个签名者还常见的情况就是多个接收者。想将一条消息发送给多个接收者相当常见，同时对所有这些接收者都进行加密。这在 S/MIME 中非常高效。一方只需使用单个 CEK 对消息进行加密，然后再在独立的 RecipientInfo 结构中针对每个接收者逐个对 CEK 进行加密。这样只需对消息加密一次，既节省了 CPU 时间又节省了带宽。对于大规模的邮件列表而言，S/MIME 甚至包含了一种允许整个邮件列表共享单个对称邮件列表密钥的功能，CEK 就是使用这个密钥来加密的。

● 收据 (Receipt)

S/MIME 所提供的一种很好的功能就是提供了收据 (receipt) 的能力，参见 RFC 2634 [Hoffman1999b] 中的描述。接收者允许发送者确信接收者收到了消息而且未受损坏。发送者可以使用一种属性来指示他想收到接收者的收据 (receipt)。当接收者收到消息时，他产生

一个包含收据的签响应。注意，发送方不能强迫接收方产生收据，它要求接收者的配合。尽管接收者的软件应当自动产生收据，但是可以绕过这项功能。因此，尽管收据可以证明接收者收到了邮件，但是缺少收据并不能证明他没有收到。

11.22 实现障碍

若想使用 S/MIME 就要求有三样东西：一个具有 S/MIME 能力的客户端、一份证书以及对他人证书的存取。具有 S/MIME 能力的客户端越来越普及。实现 S/MIME 不比 SSL 困难，而许多支持 SSL 的浏览器现在都支持 S/MIME，其中包括 Netscape Communicator 和 Microsoft Outlook Express。在写这本书的时候，OpenSSL 甚至就有某种程度的 S/MIME 支持。然而，证书的需求以及对证书的存取更成问题。

证书

普遍可从商业 CA 获得证书。然而，S/MIME 与我们到目前为止所讨论的任何一种系统都存在根本的不同，它要求最终用户——而不只是服务器——拥有证书。这是一种相当大的入门障碍，因为用户数目远远超过服务器数目。因此，至今证书的部署都很缓慢。

证书部署的一个问题就是证书需要对用户的 E-mail 地址提供担保。这会提出一种社会问题，因为还不清楚谁能够为声称用户拥有某个 E-mail 地址进行担保。显然不能相信用户。从理论上讲，E-mail 地址所在域的所有者应当对用户的身份提供担保，但这要求系统管理员的参与，而这是非常不方便的。

一种强度较弱但相当常见的方案就是让 CA 强迫用户证明他能够收取发给定 E-mail 地址的邮件。典型的 CA 给用户发送一封包含随机字符串的电子邮件，用户必须将这封邮件发回才能获得证书。这种协议显然受制于各种各样的攻击，其中包括简单的嗅探攻击，但由于对于给定的用户只完成一遍这样的工作，所以也并非毫无价值。不管怎样，更好的证书颁发过程有可能需要以安全 DNS 为基础。

证书发现

第二种部署问题就是从他人获取证书。除非你有他的证书，否则不可能给别人发送加密邮件。在理想环境中，可以从目录服务获得证书，但是尽管已经部署了一些这样的目录服务，但它还远未普及。

在缺少全球性目录服务的情况下，最好的策略就是为每一封发送的消息进行签名。这样，任何收到你的消息的人都会拥有你的证书并能够给你发送加密邮件。当你收到一条签名消息时，你可以记住 E-mail 地址到证书的映射。这样你就能够给任何你曾经收到他的 E-mail 的人发送加密 E-mail 了。一段时间之后，大多数你与之通信过的人都会拥有你的证书并给你发送加密 E-mail。直到目录服务普及时，本地缓存有可能是我们目前所能采用的最好办法。

11.23 S/MIME 与 SMTP/TLS 的对比

在第 10 章中我们看到 SMTP/TLS 在保护 E-mail 安全时并不十分擅长。与之对照，

S/MIME 却完成得非常出色。这在很大程度上取决于在恰当的抽象层面上（即发送者与接收者的 MUA）处理问题。

● 端到端的安全

S/MIME 提供端到端的安全。消息由发送者进行加密和签名，并由接收者进行解密和验证。它们以受保护的形式通过网络。与之对照，使用 SMTP/TLS，我们经常无法保护邮件客户端与本地服务器之间的“最后一步”一环，因为这些链接根本就不使用 SMTP，而是使用其他一些诸如 MAPI 或 POP 的协议。对 S/MIME 来说这就不成问题，因为加密消息只不过是 MIME 数据而已，可以使用任何传输方式。

● 不可抵赖性

SSL 与 TLS 根本不能提供不可抵赖性，因为它们使用 MAC 来实现消息完整性。由于 S/MIME 使用数字签名并对整个消息进行签名，经过签名的 S/MIME 消息能够从一个接收者转发给另一个，而签名是完整并且是可验证的。

● 中继

即便发送者与接收者的本地服务器都支持 SMTP/TLS，如果邮件沿途必须穿过非 TLS 传输，那么消息也不会受到保护。与之对照，S/MIME 不需要任何邮件服务器具有安全能力，它只要求最终的用户程序对 S/MIME 支持。

S/MIME 并不要求中继提供某种类型的特殊支持，但它们一定不能破坏内容。因为签名是针对特定字节表达的，如果中继破坏了字节序列的话，那么签名将不再能进行验证。SMTP 兼容的中继不会产生问题，但是邮件经常通过某些其他类型的中继。在某种程度上，S/MIME 能够抵御已知的转换，但是中继仍然有可能破坏消息而无法修复。然而，这样的中继肯定不会是 SMTP/TLS 兼容的，所以在这种情况下 S/MIME 的情况不比 SMTP/TLS 更糟。

● 虚拟主机

S/MIME 根本不要求邮件服务器提供任何特殊的 support，因此虚拟主机工作得相当好。由于发送者和接收者的证书是与他们的地址而不是与任何特定服务器绑定在一起的，所以用户可以轻易地从一个 ISP 移到另一个 ISP，而不用改变任何保密信息。

● 什么是最重要的

一般来讲，S/MIME 是一种远比 SMTP/TLS 优越的解决方案。使用 S/MIME 的唯一障碍就是部署障碍。然而，具有 S/MIME 能力的邮件客户端越来越流行，现在，证书虽然没有普及但也是可以得到的。SMTP/TLS 与 S/MIME 相比的优势就是在传输过程中，它伺机保护 email 的能力。然而，这种需求最终可以使用 IPsec 来更好地满足。

11.24 选择合适的解决方案

现在，我们已经见到了几种解决方案，掌握了足够多的信息，可以得出一些更一般性的有关何时适合使用 SSL，何时最好在较高或较低的层次上处理问题的结论。

一般来讲，在通信模型简单的情况下，SSL 可以工作得很好。通信模型的简单性表现在

几个方面，但是我们总的意思是指交互应当尽可能类似于两端之间直接的 TCP 连接。环境越复杂，使用 SSL 来提供充分的安全就越困难。

● 直接连接

使用 SSL 来提供充分的安全多少会要求客户端与服务器之间拥有直接连接。我们在处理 SMTP 时碰到的许多问题都是因为没有直接连接才造成的。使用 HTTPS 之所以能够得到适当的结果，只是因为我们能够通过说服代理为我们提供隧道来模仿出直接连接。

当没有直接连接的时候，对象安全方案就会更为适合。S-HTTP 与 SSL 相比可以更好地与代理进行交互，而 S/MIME 对邮件中继来说是完全透明的。总地来说，只要你能够避免对象被中间节点损坏，对象安全协议就是一种更好的选择。

● 仅限于 TCP

考虑诸如通过 UDP 进行通信的 DNS 协议。由于 SSL 要求有 TCP 通道，所以不可能使用 SSL 来保护它。然而，由于 IPsec 位于协议栈中的较低层次上，所以它能够为 UDP 数据提供安全。类似的，诸如 DNSSEC（在 RFC 2535 [Eastlake1999] 中描述）这样保护实际 DNS 数据的对象安全协议就能够与 UDP 一起很好的工作。在这种情况下，DNSSEC 是一种更好的解决方案，因为它不要求建立 IPsec 通道就能执行名字解析。

● 通信各方只有两个

由于 SSL 连接本质上是一对一的，所以在通信方只有两个的情况下，SSL 工作得最好。对象安全倾向于更为妥善地支持多对多的通信。在我们使用 SSL 模仿多对多的交互时，它通常远比对应的对象安全解决方案笨拙得多。

考虑给大量接收者发送消息的情况。如果使用 SSL，我们需要在每次给新的接收者发送消息时重新进行加密。如果使用对象解决方案，则只需针对每个接收者重新对 CEK 进行打包，这既节省了 CPU 时间又节省了带宽。如果只对消息进行签名，我们甚至不需要重新加密 CEK，只需将同样的消息发送给多个接收者。当我们反复传输相同的签名内容时，这种能力尤为重要。我们可以脱机对内容进行签名，然后再将预签名的文件放到某个非安全服务器上。这样即便服务器被攻破，攻击者也无法创建伪造的发行版。

● 简单安全服务

当所提供的安全服务简单时 SSL 工作得最好。由于它对传输的对象一无所知，所以无法提供像不可抵赖性、时间戳记或返回收据这样的服务。这些服务要求使用对象安全协议，而这种协议本质上是一种应用层协议。

11.25 总 结

SSL 是一种有用而灵活的安全工具，但是它并不是我们工具箱中惟一的工具。为了知道怎样才能选择恰当的协议，重要的是对其他选择的理解。本章勘察了其他三种安全协议——用于保护任意网络数据安全的 IPsec，保护 HTTP 事务安全的安全 HTTP 以及保护 E-mail 消息安全的 S/MIME——以期提供这样的观察角度。

端到端的论述要求正确的应用安全要由应用来提供。因为 HTTP 和 E-mail 是应用层协



议,所以它们的安全必须在应用层来提供。HTTPS 在很大程度上符合这种情况,但 SMTP/TLS 不是这样,这是导致其众多缺点的原因所在。

高层安全协议更为灵活。与端到端论述一致,较高层次的协议能够更切实地提供应用所需要的安全。特别的,对象安全协议能够提供诸如不可抵赖性这样的高级特性。一般来讲,如果不太麻烦的话,最好使用一种较高层次的安全机制。

较低层次的安全协议更具一般性。我们越将安全置于协议栈中的较低层次,它所能适合的通信种类就越多。因此, S-HTTP 和 S/MIME 只能保护一种服务的安全,而 SSL 能够保护任何建立在 TCP 之上的通信的安全, IPsec 能够保护任何 TCP/IP 通信的安全。

IPsec 为所有网络数据提供安全。与 SSL 不同, IPsec 可以为两台主机之间的任何通信提供安全。这就是说,一旦安装了 IPsec, 所有的应用都获得了某种程度的安全。然而,由于 IPsec 在协议栈中比 SSL 所处的层次要低,所以它对通信通道中的中间节点的干扰更为敏感。

Secure-HTTP 为 HTTP 事务提供安全。S-HTTP 将每个 HTTP 请求或响应当作单个消息,这使得我们能更灵活地磋商安全属性以及消息的对象安全。

S/MIME 为 E-mail 消息提供安全。对单条消息逐一进行保护。这样可以实现端到端的消息保护而且也与 E-mail 的存储转发模型结合得很好。它还可以实现真正的发送者与接收者认证。

SSL 可以与实时 TCP 服务一起很好的工作。如果通信各方之间使用 TCP 进行连接,那么 SSL 就能工作得很好。如果它们使用除 TCP 之外的其他方式,那么就应当使用 IPsec。如果它们并不直接连接,那么应用层安全协议则更为适合。SSL 是提供较好保护的应用层安全与提供更具一般性的 IP 安全之间的一种折中。

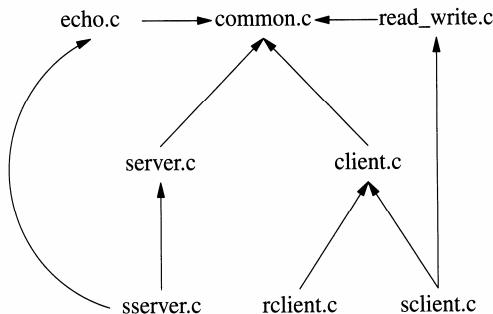
A

范例代码

A.1 第 8 章

本节包含第 8 章完整的例子程序的源代码版本。你还可以从作者的 Web 站点 <http://www.rfm.com/sslbook/examples> 下载机器可读的版本。

● A.1.1 C 语言编写的范例



上面的这张图描述了例子中各个 C 源代码文件的依赖关系。

common.c 提供一组常用的工具函数，其中包括用于客户端与服务器的公共初始化函数以及错误退出例程。

server.c 提供服务器工具函数，主要用于初始化，但还有用于封装创建与绑定套接字的 tcplisten()。

client.c 提供封装 connect() 与验证例程 check_cert_chain() 的包裹函数 tcp_connect()。

echo.c 包含主服务器的响应循环。

sserver.c 包含服务器主程序，它安排在一个套接字上进行监听，接着循环执行 accept() 和 echo()。

read_write.c 包含客户端基于 select() 的多路 I/O 处理。

sclient.c 是我们的客户端主程序。它连接到服务器，然后再用 read_write() 来回移动数据。

rclient.c 是一个占位 (stub) 客户端，它演示了如何使用 OpenSSL 完成会话恢复。



common.h

```
1      #ifndef _common_h
2      #define _common_h

3      #include <stdio.h>
4      #include <stdlib.h>
5      #include <errno.h>
6      #include <sys/types.h>
7      #include <sys/socket.h>
8      #include <netinet/in.h>
9      #include <netinet/tcp.h>
10     #include <netdb.h>
11     #include <fcntl.h>
12     #include <signal.h>

13     #include <openssl/ssl.h>

14     #define CA_LIST "root.pem"
15     #define HOST "localhost"
16     #define RANDOM "random.pem"
17     #define PORT 4433
18     #define BUFSIZZ 1024

19     extern BIO *bio_err;
20     int berr_exit (char *string);
21     int err_exit(char *string);

22     SSL_CTX *initialize_ctx(char *keyfile, char *password);
23     void destroy_ctx(SSL_CTX *ctx);

24     #endif
```

common.h

common.c

```
1      #include "common. h"

2      BIO *bio_err=0;
3      static char *pass;
4      static int password_cb(char *buf,int num, int rwflag,void *userdata);
5      static void sigpipe_handle(int x);

6      /* A simple error and exit routine*/
7      int err_exit(string)
8          char *string;
9          {
10              fprintf(stderr,"%s\n",string);
11              exit(0);
12          }
```

```
13     /* Print SSL errors and exit*/
14     int berr_exit(string)
15         char *string;
16     {
17         BIO printf(bio_err,"%s\n",string);
18         ERR_print_errors(bio_err);
19         exit(0);
20     }

21     /*The password code is not thread safe*/
22     static int password_cb(char *buf,int num,int rwflag,void *userdata)
23     {
24         if(num<strlen(pass)+1)
25             return(0);

26         strcpy(buf,pass);
27         return(strlen(pass));
28     }

29     static void sigpipe_handle(int x){
30     }

31     SSL_CTX *initialize_ctx(keyfile,password)
32         char *keyfile;
33         char *password;
34     {
35         SSL_METHOD *meth;
36         SSL_CTX *ctx;
37     if(!bio_err){
38         /* Global system initialization*/
39         SSL_library_init();
40         SSL_load_error_strings();

41         /* An error write context */
42         bio_err=BIO_new_fp(stderr,BIO_NOCLOSE);
43     }

44     /* Set up a SIGPIPE handler */
45     signal(SIGPIPE,sigpipe_handle);

46     /* Create our context*/
47     meth=SSLv3_method();
48     ctx=SSL_CTX_new(meth);

49     /* Load our keys and certificates*/
50     if(!(SSL_CTX_use_certificate_file(ctx,keyfile,SSL_FILETYPE_PEM)))
51         berr_exit("Couldn't read certificate file");

52     pass=password;
```



范例代码

```
53     SSL_CTX_set_default_passwd_cb(ctx,password_cb);
54     if(!(SSL_CTX_use_PrivateKey_file(ctx,keyfile,SSL_FILETYPE_PEM)))
55         berr_exit("Couldn't read key file");
56
57     /* Load the CAs we trust*/
58     if(!(SSL_CTX_load_verify_locations(ctx,CA_LIST, 0)))
59         berr_exit("Couldn't read CA list");
60     SSL_CTX_set_verify_depth(ctx,1);
61
62     /* Load randomness */
63     if(!(RAND_load_file(RANDOM,1024*1024)))
64         berr_exit("Couldn't load randomness");
65
66     return ctx;
67 }
```

----- common.c

```
1 #ifndef _echo_h
2 #define _echo_h
3
4 void echo(SSL *ssl,int sock);
5
6 #endif
```

----- echo.h

```
1 #include "common.h"
2
3 void echo(ssl,s)
4     SSL *ssl;
5     int s;
6     {
7         char buf[BUFSIZZ];
8         int r,len,offset;
9
10        while(1){
11            /* First read data */
12            r=SSL_read(ssl,buf,BUFSIZZ);
13            switch(SSL_get_error(ssl,r)){
14                case SSL_ERROR_NONE:
15                    len=r;
16                    break;
```

```
15         case SSL_ERROR_ZERO_RETURN:
16             goto end;
17         default:
18             berr_exit("SSL read problem");
19     }

20     /* Now keep writing until we've written everything*/
21     offset=0;

22     while(len){
23         r=SSL_write(ssl,buf+offset,len);
24         switch(SSL_get_error(ssl,r)){
25             case SSL_ERROR_NONE:
26                 len-=r;
27                 offset+=r;
28                 break;
29             default:
30                 berr_exit("SSL write problem");
31         }
32     }
33 }
34 end:
35     SSL_shutdown(ssl);
36     SSL_free(ssl);
37     close(s);
38 }
```

echo.c

----- read_write.h

```
1 #ifndef _read_write_h
2 #define _read_write_h

3 void read_write(SSL *ssl,int sock);

4 #endif
```

----- read_write.h

----- read_write.c

```
1 #include "common.h"

2 /* Read from the keyboard and write to the server
   Read from the server and write to the keyboard

4     we use select() to multiplex
5 */
6 void read_write(ssl,sock)
7     SSL *ssl;
8     {
9         int width;
```

```
10     int r,c2sl=0,c2s_offset=0;
11     fd_set readfds,writefds;
12     int shutdown_wait=0;
13     char c2s[BUFSIZZ],s2c[BUFSIZZ];
14     int ofcmode;

15     /*First we make the socket nonblocking*/
16     ofcmode=fcntl(sock,F_GETFL,0);
17     ofcmode|=O_NDELAY;
18     if(fcntl(sock,F_SETFL,ofcmode))
19         err_exit("Couldn't make socket nonblocking");

20     width=sock+1;
21     while(1){
22         FD_ZERO(&readfds);
23         FD_ZERO(&writefds);

24         FD_SET(sock,&readfds);

25         /*If we've still got data to write then don't try to read*/
26         if(c2sl)
27             FD_SET(sock,&writefds);
28         else
29             FD_SET(fileno(stdin),&readfds);

30         r=select(width,&readfds,&writefds,0,0);
31         if(r==0)
32             continue;

33         /* Now check if there's data to read */
34         if(FD_ISSET(sock,&readfds)){
35             do {
36                 r=SSL_read(ssl,s2c,BUFSIZZ);

37                 switch(SSL_get_error(ssl,r)){
38                     case SSL_ERROR_NONE:
39                         fwrite(s2c,1,r,stdout);
40                         break;
41                     case SSL_ERROR_ZERO_RETURN:
42                         /* End of data */
43                         if(!shutdown_wait)
44                             SSL_shutdown(ssl);
45                         goto end;
46                         break;
47                     case SSL_ERROR_WANT_READ:
48                         break;
49                     default:
50                         berr_exit{"SSL read problem"};
51                 }
52             }
53         }
54     }
55 }
```

```
52             } while (SSL_pending(ssl));
53         }

54         /* Check for input on the console*/
55         if(FD_ISSET(fileno(stdin),&readfds)){
56             c2sl=read(fileno(stdin),c2s,BUFSIZZ);
57             if(c2sl==0){
58                 shutdown_wait=1;
59                 if(SSL_shutdown(ssl))
60                     return;
61             }
62             c2s_offset=0;
63         }

64         /* If we've got data to write then try to write it*/
65         if(c2sl && FD_ISSET(sock,&writefds)){
66             r=SSL_write(ssl,c2s+c2s_offset,c2sl);

67             switch(SSL_get_error(ssl,r)){
68                 /* We wrote something*/
69                 case SSL_ERROR_NONE:
70                     c2sl-=r;
71                     c2s_offset+=r;
72                     break;

73                     /* We would have blocked */
74                     case SSL_ERROR_WANT_WRITE:
75                         break;

76                     /* Some other error */
77                     default:
78                         berr_exit("SSL write problem");
79                 }
80             }

81         }
82     end:
83         SSL_free(ssl);
84         close(sock);
85         return;
86     )
```

read_write.c

server.h

```
1     #ifndef _server_h
2     #define _server_h

3     #define KEYFILE "server.pem"
4     #define PASSWORD "password"
```

```
5      #define DHFILE "dh1024.pem"  
  
6      int tcp_listen(void);  
7      void load_dh_params(SSL_CTX *ctx,char *file);  
8      void generate_eph_rsa_key(SSL_CTX *ctx);  
  
9      #endif
```

server.h

```
1      #include "common.h"  
2      #include "server.h"  
  
3      int tcp_listen()  
4      {  
5          int sock;  
6          struct sockaddr_in sin;  
7          int val=1;  
  
8          if((sock=socket(AF_INET,SOCK_STREAM,0))<0)  
9              err_exit("Couldn't make socket");  
  
10         memset(&sin,0,sizeof(sin));  
11         sin.sin_addr.s_addr=INADDR_ANY;  
12         sin.sin_family=AF_INET;  
13         sin.sin_port=htons(PORT);  
14         setsockopt(sock,SOL_SOCKET,SO_REUSEADDR,&val,sizeof(val));  
  
15         if(bind(sock,(struct sockaddr *)&sin,sizeof(sin))<0)  
16             berr_exit("Couldn't bind");  
17         listen(sock,5);  
  
18         return(sock);  
19     }  
  
20     void load_dh_params(ctx, file)  
21     {  
22         SSL_CTX *ctx;  
23         char *file;  
24         {  
25             DH *ret=0  
26             BIO *bio;  
  
27             if ((bio=BIO_new_file(file,"r")) == NULL)  
28                 berr_exit("Couldn't open DH file");  
  
29             ret=PEM_read_bio_DHparams(bio,NULL,NULL,NULL);  
30             BIO_free(bio);  
31             if(SSL_CTX_set_tmp_dh(ctx,ret)<0)  
32                 berr_exit("Couldn't set DH parameters");
```

```
32      }
33  void generate_eph_rsa_key(CTX)
34  SSL_CTX *ctx;
35  {
36  RSA *rsa;
37
38  rsa=RSA_generate_key(512,RSA_F4,NULL,NULL);
39  if (!SSL_CTX_set_tmp_rsa(ctx,rsa))
40  berr_exit("Couldn't set RSA key");
41  RSA_free(rsa);
42 }
```

server.c

```
1 #ifndef_client_h
2 #define_client_h
3
4 #define KEYFILE "client.pem"
5 #define PASSWORD "password"
6
7 int tcp_connect(void);
8 void check_cert_chain(SSL *ssi,char *host);
9
10#endif
```

client.h

```
1 #include "common.h"
2
3 int tcp_connect()
4 {
5     struct hostent *hp;
6     struct sockaddr_in addr;
7     int sock;
8
9     if (!(hp=gethostbyname(HOST)))
10         berr_exit("Couldn't resolve host");
11     memset(&addr,0,sizeof(addr));
12     addr.sin_addr=*(struct in_addr*)hp->h_addr_list[0];
13     addr.sin_family=AF_INET;
14     addr.sin_port=htons(PORT);
15
16     if ((sock=socket(AF_INET,SOCK_STREAM, IPPROTO_TCP))<0)
17         err_exit("Couldn't create socket");
18     if (connect(sock,(struct sockaddr *)&addr,sizeof(addr))<0)
19         err_exit("Couldn't connect socket");
```



```

17         return sock;
18     }

19     /* Check that the common name matches the host name*/
20     void check_cert_chain(ssl,host)
21         SSL *ssl;
22         char *host;
23     {
24         X509 *peer;
25         char peer_CN[256];

26         if(SSL_get_verify_result(ssl)!=X509_V_OK)
27             berr_exit("Certificate doesn't verify");

28         /*Check the cert chain. The chain length
29          is automatically checked by OpenSSL when we
30          set the verify depth in the ctx

31         All we need to do here is check that the CN
32         matches
33     */
34     /*Check the common name*/
35     peer=SSL_get_peer_certificate(ssl);
36     X509_NAME_get_text_by_NID(X509_get_subject_name(peer),
37         NID_commonName, peer_CN, 256);
38     if(strcasecmp(peer_CN,host))
39         err_exit("Common name doesn't match host name");
40     }

```

client.c

```

1      /* A simple SSL echo server */
2      #include "common.h"
3      #include "server.h"
4      #include "echo.h"

5      static int s_server_session_id_context = 1;
6      int main(argc,argv)
7          int argc;
8          char **argv;
9      {
10         int sock,s;
11         BIO *sbio;
12         SSL_CTX *ctx;
13         SSL *ssl;
14         int r;

15         /* Build our SSL context*/

```

```
16     ctx=initialize_ctx(KEYFILE,PASSWORD);
17     load_dh_params(ctx,DHFILE);
18     generate_eph_rsa_key(ctx);

19     SSL_CTX_set_session_id_context(ctx,(void*)&s_server_session_id_context
20                                     sizeof s_server_session_id_context);

21     sock=tcp_listen();

22     while(1){
23         if((s=accept(sock,0,0))<0)
24             err_exit("Problem accepting");

25         sbio=BIO_new_socket(s,BIO_NOCLOSE);
26         ssl=SSL_new(ctx);
27         SSL_set_bio(ssl,sbio,sbio);

28         if((r=SSL_accept(ssl)<=0))
29             berr_exit("SSL accept error");

30         echo(ssl,s);
31     }
32     destroy_ctx(ctx);
33     exit(0);
34 }
```

sserver.c

sclient.c

```
1     /* A simple SSL client.

2         It connects and then forwards data from/to the terminal
3         to/from the server
4     */
5     #include "common.h"
6     #include "client.h"
7     #include "read_write.h"
8     int main(argc,argv)
9         int argc;
10        char **argv;
11    {
12        SSL_CTX *ctx;
13        SSL *ssl;
14        BIO *sbio;
15        int sock;

16        /* Build our SSL context*/
17        ctx=initialize_ctx(KEYFILE,PASSWORD)

18        /* Connect the TCP socket*/
```



```
19         sock=tcp_connect();  
  
20         /* Connect the SSL socket */  
21         ssl=SSL_new(ctx);  
22         sbio=BIO_new_socket(sock,BIO_NOCLOSE);  
23         SSL_set_bio(ssl,sbio,sbio);  
24         if(SSL_connect(ssl)<=0)  
25             berr_exit("SSL connect error");  
26         check_cert_chain(ssl,HOST);  
  
27         /* read and write */  
28         read_write(ssl,sock);  
  
29         destroy_ctx(ctx);  
30     }
```

sclient.c

```
1      /* SSL client demonstrating session resumption */  
2      #include "common.h"  
3      #include "client.h"  
4      #include "read_write.h"  
  
5      int main(argc,argv)  
6          int argc;  
7          char **argv;  
8          {  
9              SSL_CTX *ctx;  
10             SSL *ssl;  
11             BIO *sbio;  
12             SSL_SESSION *sess;  
13             int sock;  
  
14             /* Build our SSL context*/  
15             ctx=initialize_ctx(KEYFILE,PASSWORD)  
  
16             /* Connect the TCP socket*/  
17             sock=tcp_connect();  
  
18             /* Connect the SSL socket */  
19             ssl=SSL_new(ctx);  
20             sbio=BIO_new_socket(sock,BIO_NOCLOSE );  
21             SSL_set_bio(ssl,sbio,sbio);  
22             if(SSL_connect(ssl)<=0)  
23                 berr_exit("SSL connect error (first connect)");  
24             check_cert_chain(ssl,HOST);  
  
25             /* Now hang up and reconnect */  
26             sess=SSL_get_session(ssl); /*Collect the session*/
```

```

27     SSL_shutdown(ssl);
28     close(sock);

29     sock=tcp_connect();
30     ssl=SSL_new(ctx);
31     sbio=BIO_new_socket(sock,BIO_NOCLOSE);
32     SSL_set_bio(ssl,sbio,sbio);
33     SSL_set_session(ssl,sses); /*And resume it*/
34     if(SSL_connect(ssl)<=0)
35         berr_exit("SSL connect error (second connect)");
36     check_cert_chain(ssl,HOST);

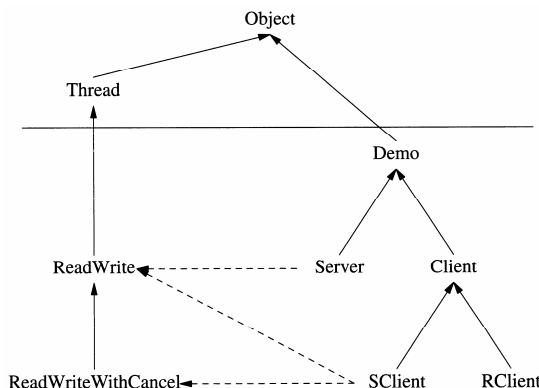
37     /*Now close everything down again*/
38     SSL_shutdown(ssl);
39     close(sock);
40     destroy_ctx(ctx);
41 }

```

rclient.c



A.1.2 Java 编写的范例



上图展示了例子中 Java 类的类关系图。横线上面的类为标准 Java 类，横线下面的类是例子中的类，虚线表示非继承性依赖。

因此，Server 继承 Demo，但只是使用了 ReadWrite。

Demo 提供了一组常量定义，以及常用的初始化例程。

ReadWrite 提供一个将数据从 InputStream 拷贝到 OutputStream 的线程实例。当收到 InputStream 上的数据结束标志时，它会沿 OutputStream 发送过去一个 close_notify。

ReadWriteWithCancel 是专门化的 ReadWrite，它会在收到数据结束标志时通知另一个线程。

Server 是我们的服务器程序。它创建一个 SSLSocket，接着在 accept() 和 ReadWrite 之间进行循环。

Client 提供公共的客户端功能。特别的，它包含通用连接函数以及服务器证书检查。



SClient 是我们的主客户端程序。它连接到服务器并使用 ReadWrite 和 ReadWriteWithCancel 来回拷贝服务器的数据。

RClient 是一个占位 (stub) 客户端，它演示了使用 PureTLS 的会话恢复。

——ReadWrite.java

```
1      import COM.claymoresystems.sslg.*;
2      import COM.claymoresystems.ptls.*;
3      import java.io.*;
4
5      /** This class simply copies anything from the in InputStream
6          to the out OutputStream
7      */
8      public class ReadWrite extends Thread {
9          protected SSLSocket s;
10         protected InputStream in;
11         protected OutputStream out;
12
13         /** Create a ReadWrite object
14             @param s the socket we're using
15             @param in the stream to read from
16             @param out the stream to write to
17         */
18         public ReadWrite(SSLocket s,InputStream in,OutputStream out){
19             this.s=s;
20             this.in=in;
21             this.out=out;
22         }
23
24         /** Copy data from in to out.*/
25         public void run() {
26             byte[] buf=new byte[1024];
27             int read;
28
29             try {
30                 while(true){
31                     // Check for thread termination
32                     if(isInterrupted())
33                         break;
34
35                     // Read data in
36                     read=in.read(buf)
37
38                     // Exit if there is no more data available
39                     if(read==-1)
40                         break;
41                     // Write the data out
42                     out.write(buf,0,read);
43                 }
44             } catch (IOException e) {
45                 e.printStackTrace();
46             }
47         }
48     }
```

```
38         } catch (IOException e){
39             // run() can't throw IOException
40             throw new InternalError(e.toString());
41         }
42
43         // Finalize
44         onEOD();
45     }
46
47     protected void onEOD(){
48         try {
49             s.sendClose();
50         } catch (IOException e){
51             ; // Ignore broken pipe
52         }
53     }
54 }
```

— ReadWrite.java

```
— ReadWriteWithCancel.java
1  import COM.claymoresystems.sslg.*;
2  import COM.claymoresystems.ptls.*;
3  import java.io.*;
4
5  /** This class is a simple extension of ReadWrite.
6   * When it receives an end of data on its in socket it
7   * sends an interrupt to the 'cancel' thread.
8 */
9  public class ReadWriteWithCancel extends ReadWrite {
10     protected ReadWrite cancel;
11
12     public ReadWriteWithCancel(SSLocket s,InputStream in,OutputStream out,
13                               ReadWrite cancel){
14         super(s,in,out);
15         this.cancel=cancel;
16     }
17
18     protected void onEOD(){
19         if(cancel.isAlive()){
20             cancel.interrupt();
21             try {
22                 cancel.join();
23             } catch (InterruptedException e){
24                 throw new InternalError(e.toString());
25             }
26         }
27     }
28 }
```



25 }

ReadWriteWithCancel.java

----- Demo.java

```
1 import COM.claymoresystems.sslg.*;
2 import COM.claymoresystems.ptls.*;

3 public class Demo {
4     public static final String host= "localhost";
5     public static final int port= 4433;
6     public static final String root= "root.pem";
7     public static final String random= "random.pem";

8     static SSLContext createSSLContext(String keyfile,String password){
9         SSLContext ctx=new SSLContext();

10    try {
11        ctx.loadRootCertificates(root);
12        ctx.loadEAYKeyFile(keyfile,password);
13        ctx.useRandomnessFile(random,password);
14    } catch (Exception e){
15        throw new InternalError(e.toString());
16    }

17    return Ctx;
18 }
19 }
```

----- Demo.java

----- Server.java

```
1 /** This class is a simple SSL echo server */
2 import COM.claymoresystems.sslg.*;
3 import COM.claymoresystems.ptls.*;
4 import java.io.*;

5 public class Server extends Demo {
6     protected static final String keyfile="server.pem";
7     protected static final String password="password";
8     protected static final String dh="dh1024.pem";
9     protected static boolean requireclientauth=true;

10    protected SSLSocket sock;
11    protected InputStream in;
12    protected OutputStream out;

13    public static void main(String []args)
14        throws IOException {
15        SSLContext ctx=createSSLContext(keyfile,password);
```

```
16          // Load our DH group
17          ctx.loadDHParams(dh);

18          SSLServerSocket listen=new SSLServerSocket(ctx,port);

19          while(true){
20              SSLSocket s=(SSLSocket)listen.accept();

21                  // Process this connection in a new thread
22                  ReadWrite rw=new ReadWrite(s,s.getInputStream(),
23                      s.getOutputStream());
24                  rw.start();
25          }
26      }
27 }
```

Server.java

```
1      /* A simple SSL client.

2          It connects and then forwards data from/to the terminal
3          to/from the server
4      */
5      import COM.claymoresystems.sslg.*;
6      import COM.claymoresystems.ptls.*;
7      import java.io.*;
8      public class SClient extends Client {
9          public static void main(String []args)
10             throws IOException, InterruptedException {
11                 SSLContext ctx=createSSLContext(keyfile,password);
12                 SSLSocket s=connect(ctx,host,port);

13                 // This thread reads from the console and writes to the server
14                 ReadWrite c2s=new ReadWrite(s,System.in,s.getOutputStream());
15                 c2s.start();

16                 // This thread reads from the server and writes to the console
17                 ReadWriteWithCancel s2c=new ReadWriteWithCancel(s,
18                     s.getInputStream(),System.out,c2s);
19                 s2c.start();
20                 s2c.setPriority(Thread.MAX_PRIORITY);
21                 s2c.join();
22         }
23     }
```

SClient.java

```
1      /* SSL client demonstrating session resumption */
2      import COM.claymoresystems.sslg.*;
```

RClient.java

```

3      import COM.claymoresystems.ptls.*;
4      import java.io.*;

5      public class RClient extends Client {
6          public static void main(String []args)
7              throws IOException {
8                  SSLContext ctx=createSSLContext(keyfile,password);

9                  /* Connect and close*/
10                 SSLSocket s=connect(ctx,host,port);
11                 s.close();

12                 /* Now reconnect: resumption happens automatically*/
13                 s=connect(ctx,host,port);
14                 s.close();
15             }
16         }

```

—— RClient.java

A.2 第 9 章

这一节包含了第 9 章例子程序的完整源代码版本：pclient.c 和 mserver.c，此外，它还包含用于执行进程间会话缓冲的 mod_ssl 部分。

A.2.1 HTTPS 范例

第 9 章所展示的程序演示了 HTTPS 通常所用到的各种 SSL 专业用语。它们与第 8 章的程序一样建立在相同的公共框架之上。大家同样可以从作者的 Web 站点 <http://www.rfm.com/sslbook/examples> 下载这些程序的机器可读版本。

pclient 是具有代理能力的客户端。它使用 HTTP CONNECT 方法连接到代理，然后再以隧道方式连接到服务器。

mserver 是能够同时以多个进程来处理多个客户端的 SSL 服务器。这本质上与用 Java 编写的 Server 完成的工作相同。

```

1      /* A proxy-capable SSL client.

2          This is just like sclient but it only supports proxies
3          */
4          #include <string.h>
5          #include "common.h"
6          #include "client.h"
7          #include "read_write.h"

8          #define PROXY "localhost"
9          #define PROXY_PORT 8080

```

```
10     #define REAL_HOST "localhost"
11     #define REAL_PORT 4433

12     int writestr(sock,str)
13         int sock;
14         char *str;
15     {
16         int len=strlen(str);
17         int r,wrote=0;

18         while(len){
19             r=write(sock,str,len);
20             if(r<=0)
21                 err_exit("Write error");
22             len-=r;
23             str+=r;
24             wrote+=r;
25         }

26         return (wrote);
27     }

28     int readline(sock,buf,len)
29         int sock;
30         char *buf;
31         int len;
32     {
33         int n,r;
34         char *ptr=buf;

35         for(n=0;n<len;n++){
36             r=read(sock,ptr,1);

37             if(r<=0)
38                 err_exit("Read error");

39             if(*ptr=='\n'){
40                 *ptr=0;

41                 /* Strip off the CR if it's there */
42                 if(buf[n-1]=='\r'){
43                     buf[n-1]=0;
44                     n--;
45                 }

46                 return(n);
47             }

48             *ptr++;
}
```

```
49         }
50         err_exit("Buffer too short");
51     }

52     int proxy_connect(){
53         struct hostent *hp;
54         struct sockaddr_in addr;
55         int sock;
56         BIO *sbio;
57         char buf[1024];
58         char *protocol, *response_code;

59         /* Connect to the proxy, not the host */
60         if(!(hp=gethostbyname(PROXY)))
61             berr_exit("Couldn't resolve host");
62         memset(&addr,0,sizeof(addr));
63         addr.sin_addr=*(struct in_addr*)hp->h_addr_list[0];
64         addr.sin_family=AF_INET;
65         addr.sin_port=htons(PROXY_PORT);

66         if((sock=socket(AF_INET,SOCK_STREAM,IPPROTO_TCP))<0)
67             err_exit("Couldn't create socket");
68         if(connect(sock,(struct sockaddr *)&addr,sizeof(addr))<0)
69             err_exit("Couldn't connect socket");

70         /* Now that we're connected, do the proxy request */
71         sprintf(buf,"CONNECT %s:%d HTTP/1.0\r\n\r\n",REAL_HOST,REAL_PORT);
72         writestr(sock,buf);

73         /* And read the response*/
74         if(readline(sock,buf,sizeof(buf))==0)
75             err_exit("Empty response from proxy");

76         if((protocol=strtok(buf," " ))<0)
77             err_exit("Couldn't parse server response: getting protocol");
78         if(strncmp(protocol,"HTTP",4))
79             err_exit("Unrecognized protocol");
80         if((response_code=strtok(0," " ))<0)
81             err_exit("Couldn't parse server response: getting response code");
82         if(strcmp(response_code,"200"))
83             err_exit("Received error from proxy server");

84         /* Look for the blank line that signals end of header*/
85         while(readline(sock,buf,sizeof(buf))>0) {
86             ;
87         }

88         return(sock);
```

```
89      }
90      int main(argc,argv)
91          int argc;
92          char **argv;
93          {
94              SSL_CTX *ctx;
95              SSL *ssl;
96              BIO *sbio;
97              int sock;

98          /* Build our SSL context*/
99          ctx=initialize_ctx(KEYFILE,PASSWORD);

100         /* Connect the TCP socket*/
101         sock=proxy_connect();

102         /* Connect the SSL socket */
103         ssl=SSL_new(ctx);
104         sbio=BIO_new_socket(sock,BIO_NOCLOSE);
105         SSL_set_bio(ssl,sbio,sbio);
106         if(SSL_connect(ssl)<=0)
107             berr_exit("SSL connect error");
108         check_cert_chain(ssl,HOST);

109         /* read and write */
110         read_write(ssl,sock);

111         destroy_ctx(ctx);
112     }
```

pclient.c

```
1      /* A multiprocess SSL server */
2      #include "common.h"
3      #include "server.h"
4      #include "echo.h"

5      int main(argc,argv)
6          int argc;
7          char **argv;
8          {
9              int sock,s;
10             BIO *sbio;
11             SSL_CTX *ctx;
12             SSL *ssl;
13             int r;
14             pid_t pid;
```

mserver.c

```

15         /* Build our SSL context*/
16         ctx=initialize_ctx(KEYFILE,PASSWORD);
17         load_dh_params(ctx,DHFILE);
18         generate_eph_rsa_key(ctx);

19         sock=tcp_listen();

20         while(1){
21             if((s=accept(sock,0,0))<0)
22                 err_exit("Problem accepting");

23             if(pid=fork()){
24                 close(s);
25             }
26             else {
27                 sbio=BIO_new_socket(s,BIO_NOCLOSE);
28                 ssl=SSL_new(ctx);
29                 SSL_set_bio(ssl,sbio,sbio);

30                 if((r=SSL_accept(ssl)<=0))
31                     berr_exit("SSL accept error");

32                 echo(ssl,s);
33             }
34         }
35         destroy_ctx(ctx);
36         exit(0);
37     }

```

mserver.c

● A.2.2 mod_ssl 会话恢复

mod_ssl 是流行的 Apache Web 服务器的一个模块，它使用 OpenSSL 来给 Apache 增加 SSL/TLS 支持。Apache 主要用于 UNIX 系统，因此它使用多进程来处理并发的多个客户端。正如我们在第 9 章所讨论的，我们需要特殊的支持才能在这些进程之间共享会话。本节的剩余部分就来研究一些这样的代码。

OpenSSL 提供它自己的会话缓冲代码，它在单个程序中工作。为了适应具有更多复杂需求的应用程序的需要，它提供了许多可以让程序提供自己会话处理代码的钩子函数（hook）。我们首先研究一下 mod_ssl 安装的钩子函数。

接着考虑 mod_ssl 中实际的会话恢复代码。实际上，mod_ssl 提供了两种会话缓存支持。第一（可移植性更好）种版本使用 dbm 将会话 id 保存到磁盘上。第二（速度更快）种版本使用共享内存段。mod_ssl 的钩子调用通用函数，而这些函数分别导向各种不同的实现。

● 钩子函数

```

167     /*
168      * This callback function is executed by OpenSSL whenever a new SSL_SESSION is

```

```
169     * added to the internal OpenSSL session cache. We use this hook to spread the
170     * SSL_SESSION also to the inter-process disk-cache to make share it with our
171     * other Apache pre-forked server processes.
172     */
173 int ssl_callback_NewSessionCacheEntry(SSL *ssl, SSL_SESSION *pNew)
174 {
175     conn_rec *conn;
176     server_rec *s;
177     SSLSrvConfigRec *sc;
178     long t;
179     BOOL rc;

180     /*
181     * Get Apache context back through OpenSSL context
182     */
183     conn = (conn_rec *)SSL_get_app_data(ssl);
184     s    = conn->server;
185     sc   = mySrvConfig(s);

186     /*
187     * Set the timeout also for the internal OpenSSL cache, because this way
188     * our inter-process cache is consulted only when it's really necessary.
189     */
190     t = sc->nSessionCacheTimeout;
191     SSL_set_timeout(pNew, t);

192     /*
193     * Store the SSL_SESSION in the inter-process cache with the
194     * same expire time, so it expires automatically there, too.
195     */
196     t = (SSL_get_time(pNew) + sc->nSessionCacheTimeout);
197     rc = ssl_scache_store(s, pNew, t);

198     /*
199     * Log this cache operation
200     */
201     ssl_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache: "
202             "request=SET status=%s id=%s timeout=%ds (session caching)",
203             rc == TRUE ? "OK" : "BAD",
204             ssl_scache_id2sz(pNew->session_id, pNew->session_id_length),
205             t-time(NULL));

206     /*
207     * return 0 which means to OpenSSL that the pNew is still
208     * valid and was not freed by us with SSL_SESSION_free().
209     */
210     return 0;
211 }
212 */


```

```
213 * This callback function is executed by OpenSSL whenever a
214 * SSL_SESSION is looked up in the internal OpenSSL cache and it
215 * was not found. We use this to lookup the SSL_SESSION in the
216 * inter-process disk-cache where it was perhaps stored by one
217 * of our other Apache pre-forked server processes.
218 */
219 SSL_SESSION *ssl_callback_GetSessionCacheEntry(
220     SSL *ssl, unsigned char *id, int idlen, int *pCopy)
221 {
222     conn_rec *conn;
223     server_rec *s;
224     SSL_SESSION *pSession;

225     /*
226      * Get Apache context back through OpenSSL context
227      */
228     conn = (conn_rec *)SSL_get_app_data(ssl);
229     s = conn->server;
230     /*
231      * Try to retrieve the SSL_SESSION from the inter-process cache
232      */
233     pSession = ssl_scache_retrieve(s, id, idlen);

234     /*
235      * Log this cache operation
236      */
237     if (pSession != NULL)
238         ssi_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache: "
239                 "request=GET status=FOUND id=%s (session reuse)",
240                 ssl_scache_id2sz(id, idlen));
241     else
242         ssi_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache: "
243                 "request=GET status=MISSED id=%s (session renewal)",
244                 ssl_scache_id2sz(id, idlen));

245     /*
246      * Return NULL or the retrieved SSL_SESSION. But indicate (by
247      * setting pCopy to 0) that the reference count on the
248      * SSL_SESSION should not be incremented by the SSL library,
249      * because we will no longer hold a reference to it ourself.
250      */
251     *pCopy = 0;
252     return pSession;
253 }

254 /*
255  * This callback function is executed by OpenSSL whenever a
256  * SSL_SESSION is removed from the the internal OpenSSL cache.
257  * We use this to remove the SSL_SESSION in the inter-process
```

```
258     * disk-cache, too.
259     */
260 void ssl_callback_DelSessionCacheEntry(
261     SSL_CTX *ctx, SSL_SESSION *pSession)
262 {
263     server_rec *s;

264     /*
265      * Get Apache context back through OpenSSL context
266      */
267     s = (server_rec *)SSL_CTX_get_app_data(ctx);
268     if (s == NULL) /* on server shutdown Apache is already gone */
269         return;
270     /*
271      * Remove the SSL_SESSION from the inter-process cache
272      */
273     ssl_scache_remove(s, pSession);

274     /*
275      * Log this cache operation
276      */
277     ssi_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache: "
278             "request=REM status=OK id=%s (session dead)",
279             ssl_scache_id2sz(pSession->session_id,
280             pSession->session_id_length));

281     return;
282 }
```

ssl_engine_kernel.c

OpenSSL 提供了三种会话缓存钩子：会话创建、会话获取以及会话删除。mod_ssl 钩子函数本质上是我们在下一节讨论的缓存函数的包裹函数。它们实际上所完成的工作就是调用会话缓存函数并登记执行结果。

会话恢复

173~211 会话创建代码是惟一复杂的钩子函数。mod_ssl 的进程间缓存意图只有在 OpenSSL 内部缓存无效的情况下才使用。因此如果创建会话的进程与在其中恢复会话的是同一个进程，我们就不必使用进程间缓存来恢复会话。然而，由于必须在两个缓存中创建条目，所以当创建新会话的时候总是调用 `ssl_callback_NewSessionCacheEntry()`。它完成三项工作，设置内部缓存的超时值，使用 `ssl_scache_store()` 创建进程间缓存条目及登记结果。

会话获取

219~253 每当客户端请求会话恢复而 OpenSSL 无法在内部缓存中找到这个会话时就要调用 `ssl_callback_GetSessionCacheEntry()`，它调用 `ssl_scache_retrieve()` 来取得缓存条目并登记结果。

会话删除

260~282 每当会话变成无效时，比如在我们捕获一个警示或提前关闭时，都要调用 `ssl_callback_DeleteSessionCacheEntry()`。它只是调用 `ssl_scache_remove()` 来删除会话并登记结

果。

通用会话缓存代码

```
ssl_engine_scache.c
108     BOOL ssl_scache_store(server_rec *s, SSL_SESSION *pSession, int timeout)
109     {
110         SSLModConfigRec *mc = myModConfig( );
111         ssl_scinfo_t SCI;
112         UCHAR buf[MAX_SESSION_DER];
113         UCHAR *b;
114         BOOL rc = FALSE;

115         /* add the key */
116         SCI.ucaKey = pSession->session_id;
117         SCI.nKey = pSession->session_id_length;

118         /* transform the session into a data stream */
119         SCI.ucaData = b = buf;
120         SCI.nData = i2d_SSL_SESSION(pSession, &b);
121         SCI.tExpiresAt = timeout;

122         /* and store it... */
123         if (mc->nSessionCacheMode == SSL_SCMODE_DBM)
124             rc = ssl_scache_dbm_store(s, &SCI);
125         else if (mc->nSessionCacheMode == SSL_SCMODE_SHM)
126             rc = ssl_scache_shm_store(s, &SCI);

127 #ifdef SSL_VENDOR
128     ap_hook_use("ap::mod_ssl::vendor::scache_store.,
129                 AP_HOOK_SIG3(void,ptr,ptr), AP_HOOK_ALL, s, &SCI);
130 #endif

131         /* allow the regular expiring to occur */
132         ssl_scache_expire(s, time (NULL));

133     return rc;
134 }

135 SSL_SESSION *ssl_scache_retrieve(server_rec *s, UCHAR *id, int idlen)
136 {
137     SSLModConfigRec *mc = myModConfig();
138     SSL_SESSION *pSession = NULL;
139     ssl_scinfo_t SCI;
140     time_t tNow;

141     /* determine current time */
142     tNow = time(NULL);

143     /* allow the regular expiring to occur */
```

```
144     ssl_scache_expire(s, tNow);  
  
145     /* create cache query */  
146     SCI.ucaKey      = id;  
147     SCI.nKey       = idlen;  
148     SCI.ucaData    = NULL;  
149     SCI.nData      = 0;  
150     SCI.tExpiresAt = 0;  
  
151     /* perform cache query */  
152     if (mc->nSessionCacheMode == SSL_SCMODE_DBM)  
153         ssl_scache_dbm_retrieve(s, &SCI);  
154     else if (mc->nSessionCacheMode == SSL_SCMODE_SHM)  
155         ssl_scache_shm_retrieve(s, &SCI);  
  
156 #ifdef SSL_VENDOR  
157     ap_hook_use("ap::mod_ssl::vendor::scache_retrieve",  
158                 AP_HOOK_SIG3(void,ptr,ptr), AP_HOOK_ALL, s, &SCI);  
159 #endif  
  
160     /* return immediately if not found */  
161     if (SCI.ucaData == NULL)  
162         return NULL;  
  
163     /* check for expire time */  
164     if (SCI.tExpiresAt <= tNow) {  
165         if (mc->nSessionCacheMode == SSL_SCMODE_DBM)  
166             ssl_scache_dbm_remove(s, &SCI);  
167         else if (mc->nSessionCacheMode == SSL_SCMODE_SHM)  
168             ssl_scache_shm_remove(s, &SCI);  
169 #ifdef SSL_VENDOR  
170         ap_hook_use("ap::mod_ssl::vendor::scache_remove",  
171                     AP_HOOK_SIG3(void,ptr,ptr), AP_HOOK_ALL, s, &SCI);  
172 #endif  
173         return NULL;  
174     }  
  
175     /* extract result and return it */  
176     pSession = d2i_SSL_SESSION(NULL, &SCI.ucaData, SCI.nData);  
177     return pSession;  
178 }  
179 void ssl_scache_remove(server_rec *s, SSL_SESSION *pSession)  
180 {  
181     SSLModConfigRec *mc = myModConfig();  
182     ssl_scinfo_t SCI;  
  
183     /* create cache query */  
184     SCI.ucaKey      = pSession->session_id;  
185     SCI.nKey       = pSession->session_id_length;
```

```
186     SCI.ucaData      = NULL;
187     SCI.nData        = 0;
188     SCI.tExpiresAt  = 0;

189     /* perform remove */
190     if (mc->nSessionCacheMode == SSL_SCMODE_DBM)
191         ssl_scache_dbm_remove(s, &SCI);
192     else if (mc->nSessionCacheMode == SSL_SCMODE_SHM)
193         ssl_scache_shm_remove(s, &SCI);

194 #ifdef SSL_VENDOR
195     ap_hook_use("ap::mod_ssl::vendor::scache_remove.,
196                 AP_HOOK_SIG3(void,ptr,ptr), AP_HOOK_ALL, s, &SCI);
197 #endif

198     return;
199 }

200 void ssl_scache_expire(server_rec *s, time_t now)
201 {
202     SSLModConfigRec *mc = myModConfig();
203     SSLSrvConfigRec *sc = mySrvConfig(s);
204     static time_t last = 0;

205     /*
206      * make sure the expiration for still not-accessed session
207      * cache entries is done only from time to time
208      */
209     if (now < last+sc->nSessionCacheTimeout)
210         return;
211     last = now;

212     /*
213      * Now perform the expiration
214      */
215     if (mc->nSessionCacheMode == SSL_SCMODE_DBM)
216         ssl_scache_dbm_expire(s, now);
217     else if (mc->nSessionCacheMode == SSL_SCMODE_SHM)
218         ssl_scache_shm_expire(s, now);

219     return;
220 }
```

ssl_engine_scache.c

mod_ssl 会话缓存代码分为通用代码与底层的进程间共享实现代码。特别的，通用代码安排在恰当的时候执行超时并确保到期会话无法恢复。除此之外，所有实质性的工作都由底层的进程间共享代码来完成。

会话存储

108~134 ssl_scache_store()安排会话对象的序列化，接着再调用合适的会话缓存方法

来实际共享会话，最后调用会话到期函数。

会话获取

135~178 `ssl_scache_retrieve()`首先调用合适的会话缓存方法来加载会话对象。如果找到了对象，它就接着检查是否到期。如果会话到期了，则自动将其从缓存中删除，否则执行对象的逆序列化并将其返回。

会话删除

179~199 `ssl_scache_remove()`只是调用底层的缓存方法来删除缓存条目。

缓存到期

200-245 `ssl_scache_expire()`决不会直接由 `mod_ssl` 会话缓存钩子函数来调用。然而，不时检查一遍会话缓存并删除旧有的条目是必须的，否则会话缓存将会无限制的增长。相反，每当创建新的会话对象时会自动调用这个函数。`mod_ssl` 允许管理员设置一定时间间隔之后到期一次，因此大多数对 `ssl_scache_expire()` 的调用都是在那个时段内发生的并在第 210 行返回。否则 `ssl_scache_expire()` 调用合适的到期方法。

DBM 会话缓存

```
ssl_engine_scache.c
368     BOOL ssl_scache_dbm_store(server_rec *s, ssl_scinfo_t *SCI)
369     {
370         SSLModConfigRec *mc = myModConfig();
371         DBM *dbm;
372         datum dbmkey;
373         datum dbmval;

374         /* be careful: do not try to store too much bytes in a DBM file! */
375 #ifdef SSL_USE_SDBM
376         if ((SCI->nKey + SCI->nData) >= PAIRMAX)
377             return FALSE;
378 #else
379         if ((SCI->nKey + SCI->nData) >= 950 /* at least less than approx. 1KB */)
380             return FALSE;
381 #endif

382         /* create DBM key */
383         dbmkey.dptr = (char *) (SCI->ucaKey);
384         dbmkey.dszie = SCI->nKey;

385         /* create DBM value */
386         dbmval.dszie = sizeof(time_t) + SCI->nData;
387         dbmval.dptr = (char *) malloc(dbmval.dszie);
388         if (dbmval.dptr == NULL)
389             return FALSE;
390         memcpy((char *) dbmval.dptr, &SCI->tExpiresAt, sizeof(time_t));
391         memcpy((char *) dbmval.dptr+sizeof(time_t), SCI->ucaData, SCI->nData);

392         /* and store it to the DBM file */
393         ssl_mutex_on(s);
```

```
394     if ((dbm = ssl_dbm_open(mc->szSessionCacheDataFile,
395                             O_RDWR, SSL_DBM_FILE_MODE)) == NULL) {
396         ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
397                 "Cannot open SSLSessionCache DBM file '%s' for writing (store)",
398                 mc->szSessionCacheDataFile);
399         ssl_mutex_off(s);
400         free(dbmval.dptr);
401         return FALSE;
402     }
403     if (ssl_dbm_store(dbm, dbmkey, dbmval, DBM_INSERT) < 0) {
404         ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
405                 "Cannot store SSL session to DBM file '%s'",
406                 mc->szSessionCacheDataFile);
407         ssl_dbm_close(dbm);
408         ssl_mutex_off(s);
409         free(dbmval.dptr);
410         return FALSE;
411     }
412     ssl_dbm_close(dbm);
413     ssl_mutex_off(s);

414     /* free temporary buffers */
415     free(dbmval.dptr);

416     return TRUE;
417 }

418 void ssl_scache_dbm_retrieve(server_rec *s, ssl_scinfo_t *SCI)
419 {
420     SSLModConfigRec *mc = myModConfig();
421     DBM *dbm;
422     datum dbmkey;
423     datum dbmval;

424     /* initialize result */
425     SCI->ucaData      = NULL;
426     SCI->nData        = 0;
427     SCI->tExpiresAt   = 0;

428     /* create DBM key and values */
429     dbmkey.dptr = (char*)(SCI->ucaKey);
430     dbmkey.dszie = SCI->nKey;

431     /* and fetch it from the DBM file */
432     ssl_mutex_on(s);
433     if ((dbm = ssl_dbm_open(mc->szSessionCacheDataFile,
434                             O_RDONLY, SSL_DBM_FILE_MODE)) == NULL) {
435         ssi_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
436                 "Cannot open SSLSessionCache DBM file '%s' for reading (fetch)"
```

```
437             mc->szSessionCacheDataFile);
438         ssl_mutex_off(s);
439         return;
440     }
441     dbmval = ssl_dbm_fetch(dbm, dbmkey);
442     ssl_dbm_close(dbm);
443     ssl_mutex_off(s);
444     /* immediately return if not found */
445     if (dbmval.dptr == NULL || dbmval.dsize <= sizeof(time_t))
446         return;
447
448     /* copy over the information to the SCI */
449     SCI->nData    = dbmval.dsize-sizeof(time_t);
450     SCI->ucaData = (UCHAR *)malloc(SCI->nData);
451     if (SCI->ucaData == NULL) {
452         SCI->nData = 0;
453         return;
454     }
455     memcpy(SCI->ucaData, (char *)dbmval.dptr+sizeof(time_t), SCI->nData);
456     memcpy(&SCI->tExpiresAt, dbmval.dptr, sizeof(time_t));
457
458     return;
459 }
460 void ssl_scache_dbm_remove(server_rec *s, ssl_scinfo_t *SCI)
461 {
462     SSLModConfigRec *mc = myModConfig();
463     DBM *dbm;
464     datum dbmkey;
465
466     /* create DBM key and values */
467     dbmkey.dptr = (char *)(SCI->ucaKey);
468     dbmkey.dsize = SCI->nKey;
469
470     /* and delete it from the DBM file */
471     ssl_mutex_on(s);
472     if ((dbm = ssl_dbm_open(mc->szSessionCacheDataFile,
473                             O_RDWR, SSL_DBM_FILE_MODE)) == NULL) {
474         ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
475                 "Cannot open SSLSessionCache DBM file '%s' for writing (delete)",
476                 mc->szSessionCacheDataFile);
477         ssl_mutex_off(s);
478         return;
479     }
480     ssl_dbm_delete(dbm, dbmkey);
481     ssl_dbm_close(dbm);
482     ssl_mutex_off(s);
483
484     return;
```

```
480     }
481
482     void ssl_scache_dbm_expire(server_rec *s, time_t tNow)
483     {
484         SSLModConfigRec *mc = myModConfig();
485         DBM *dbm;
486         datum dbmkey;
487         datum dbmval;
488         pool *p;
489         time_t tExpiresAt;
490         int nElements = 0;
491         int nDeleted = 0;
492         int bDelete;
493         datum *keylist;
494         int keyidx;
495         int i;
496
497         /*
498          * Here we have to be very carefully: Not all DBM libraries are
499          * smart enough to allow one to iterate over the elements and at the
500          * same time delete expired ones. Some of them get totally crazy
501          * while others have no problems. So we have to do it the slower but
502          * more safe way: we first iterate over all elements and remember
503          * those which have to be expired. Then in a second pass we delete
504          * all those expired elements. Additionally we reopen the DBM file
505          * to be really safe in state.
506         */
507
508 #define KEYMAX 1024
509
510         ssl_mutex_on(s);
511         for (;;) {
512             /* allocate the key array in a memory sub pool */
513             if ((p = ap_make_sub_pool(NULL)) == NULL)
514                 break;
515             if ((keylist = ap_palloc(p, sizeof(dbmkey)*KEYMAX)) == NULL) {
516                 ap_destroy_pool(p);
517                 break;
518             }
519
520             /* pass 1: scan DBM database */
521             keyidx = 0;
522             if ((dbm = ssl_dbm_open(mc->szSessionCacheDataFile,
523                                     O_RDWR, SSL_DBM_FILE_MODE)) == NULL) {
524                 ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
525                         "Cannot open SSLSessionCache DBM file '%s' for scanning",
526                         mc->szSessionCacheDataFile);
527                 ap_destroy_pool(p);
528                 break;
529             }
```

```
524     }
525     dbmkey = ssl_dbm_firstkey(dbm);
526     while (dbmkey.dptr != NULL) {
527         nElements++;
528         bDelete = FALSE;
529         dbmval = ssl_dbm_fetch(dbm, dbmkey);
530         if (dbmval.dsize <= sizeof(time_t) || dbmval.dptr == NULL)
531             bDelete = TRUE;
532         else {
533             memcpy(&tExpiresAt, dbmval.dptr, sizeof(time_t));
534             if (tExpiresAt <= tNow)
535                 bDelete = TRUE;
536         }
537         if (bDelete) {
538             if ((keylist[keyidx].dptr = ap_palloc(p, dbmkey.dsize)) != NULL) {
539                 memcpy(keylist[keyidx].dptr, dbmkey.dptr, dbmkey.dsize);
540                 keylist[keyidx].dsize = dbmkey.dsize;
541                 keyidx++;
542                 if (keyidx == KEYMAX)
543                     break;
544             }
545         }
546         dbmkey = ssl_dbm_nextkey(dbm);
547     }
548     ssl_dbm_close(dbm);

549     /* pass 2: delete expired elements */
550     if ((dbm = ssl_dbm_open(mc->szSessionCacheDataFile,
551                             O_RDWR, SSL_DBM_FILE_MODE)) == NULL) {
552         ssl_log(s, SSL_LOG_ERROR|SSL_ADD_ERRNO,
553                 "Cannot re-open SSLSessionCache DBM file '%s' for expiring",
554                 mc->szSessionCacheDataFile);
555         ap_destroy_pool(p);
556         break;
557     }
558     for (i = 0; i < keyidx; i++) {
559         ssl_dbm_delete(dbm, keylist[i]);
560         nDeleted++;
561     }
562     ssl_dbm_close(dbm);

563     /* destroy temporary pool */
564     ap_destroy_pool(p);
565     if (keyidx < KEYMAX)
566         break;
567     }
568     ssl_mutex_off(s);

569     ssl_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache (DBM) Expiry: "
```

```
570     "old: %d, new: %d, removed: %d", nElements, nElements-nDeleted, nDeleted);
571     return;
572 }
```

ssl_engine_scache.c

DBM 是标准的 UNIX 软件包，它提供了一种简单的键-值数据库。它允许以一种一对一种的方式创建任意的查找键与数据值对。它本质上是一种巨大的、自伸缩的散列表。然而，数据是在磁盘文件中存储的，因此可以在进程之间进行共享。

DBM 维护快速查找所有键值的索引。不幸的是，**DBM** 起初不是针对并发存取来设计的。因此，它没有办法检测出数据库已被更新而重新加载索引。惟一重新加载索引的时候是打开数据库的时候。类似的，数据库的部分内容可以进行缓冲而且只有在关闭它的时候才将缓冲数据写到磁盘上。因此，当多个进程使用数据库时，必须在每次想要存取数据时打开和关闭数据库。**mod_ssl** DBM 代码中大多数复杂的地方都与这种安排有关的。

会话创建

368~417 第一件工作就是创建 **DBM** 键和值。键就是会话 ID，值包含时间与序列化会话对象。由于值必须是单一的字节字符串，所以必须将它们压缩在一起，这是在第 385-391 行完成的。一旦准备好了要存储的键和值，就必须对数据库进行加锁以便其他进程不能执行写操作。**mod_ssl** 具有一种用来实现这种用途的进程间互斥锁（mutex）（第 393 行）。一旦对数据库进行加锁，**mod_ssl** 就能打开数据库并存储键-值对。当完成了所有这些工作之后，**mod_ssl** 关闭数据库并释放互斥锁。

会话获取

418~457 会话获取只是会话存储的逆操作。注意，不区分用于读写互斥锁。一般来说两个进程同时读取多个相同的 **DBM** 文件是安全的。在一种会话恢复比会话创建多得多的活跃系统中，使用一个读锁和一个写锁是值得的。这样，多个读取进程就能够并发的而不是相互阻塞的读取数据库。一旦获取了会话，必须将到期时间与会话对象展开并返回给调用者（第 448-453 行）。

会话删除

458~480 **ssl_scache_dbm_remove** 遵循一种大家所熟知的模式，先对数据库加锁然后执行删除，接着再对数据库进行解锁。

缓存到期

481~572 会话删除是棘手的。从理论上讲，我们想线性地扫描数据库并同时删除到期的元素。不幸的是，许多老式的 **DBM** 实现都会在同时执行扫描与删除的时候破坏数据库。因此分两步来执行到期工作。第一遍（第 515-548 行）扫描数据库并创建包含所有到期会话的列表。在第二遍（第 549-567 行）删除列表上的每个条目。由于这样做是低效的，所以不会经常完成这样的工作。

共享内存会话缓存

DBM 会话缓存非常慢。每次缓存的存储与获取都要求对数据库依次进行加锁，打开两个文件（索引与数据文件），从磁盘读取数据，关闭文件并对数据库进行解锁。这给进程增加了很大的开销。因此，**mod_ssl** 有一种快速的基于共享内存的解决方案。总体思想就是在内存而不是磁盘上创建存储用的键-值数据库。这种内存可以位于共享内存段中，以便多个

进程来存取。

```
----- ssl_session_cache.c
752     BOOL ssl_scache_shm_store(server_rec *s, ssl_scinfo_t *SCI)
753     {
754         SSLModConfigRec *mc = myModConfig();
755         void *vp;
756
757         ssl_mutex_on(s);
758         if (table_insert_kd(mc->tSessionCacheDataTable,
759                             SCI->ucaKey, SCI->nKey,
760                             NULL, sizeof(time_t)+SCI->nData,
761                             NULL, &vp, 1) != TABLE_ERROR_NONE) {
762             ssl_mutex_off(s);
763             return FALSE;
764         }
765         memcpy(vp, &SCI->tExpiresAt, sizeof(time_t));
766         memcpy((char *)vp+sizeof(time_t), SCI->ucaData, SCI->nData);
767         ssl_mutex_off(s);
768         return TRUE;
769     }
770
771     void ssl_scache_shm_retrieve(server_rec *s, ssl_scinfo_t *SCI)
772     {
773         SSLModConfigRec *mc = myModConfig();
774         void *vp;
775         int n;
776
777         /* initialize result */
778         SCI->ucaData = NULL;
779         SCI->nData = 0;
780         SCI->tExpiresAt = 0;
781
782         /* lookup key in table */
783         ssl_mutex_on(s);
784         if (table_retrieve(mc->tSessionCacheDataTable,
785                            SCI->ucaKey, SCI->nKey,
786                            &vp, &n) != TABLE_ERROR_NONE) {
787             ssl_mutex_off(s);
788             return;
789         }
790
791         /* copy over the information to the SCI */
792         SCI->nData = n-sizeof(time_t);
793         SCI->ucaData = (UCHAR *)malloc(SCI->nData)
794         if (SCI->ucaData == NULL) {
795             SCI->nData = 0;
796             ssl_mutex_off(s);
797             return;
798         }
```

```
794     memcpy(&SCI->tExpiresAt, vp, sizeof(time_t) );
795     memcpy(SCI->ucaData, (char *)vp+sizeof(time_t), SCI->nData);
796     ssl_mutex_off(s);

797     return;
798 }

799 void ssl_scache_shm_remove(server_rec *s, ssl_scinfo_t *SCI)
800 {
801     SSLModConfigRec *mc = myModConfig();

802     /* remove value under key in table */
803     ssl_mutex_on(s);
804     table_delete(mc->tSessionCacheDataTable,
805                  SCI->ucaKey, SCI->nKey, NULL, NULL);
806     ssl_mutex_off(s);
807     return;
808 }

809 void ssl_scache_shm_expire(server_rec *s, time_t tNow)
810 {
811     SSLModConfigRec *mc = myModConfig();
812     table_linear_t iterator;
813     time_t tExpiresAt;
814     void *vpKey;
815     void *vpKeyThis;
816     void *vpData;
817     int nKey;
818     int nKeyThis;
819     int nData;
820     int nElements = 0;
821     iht nDeleted = 0;
822     int bDelete;
823     int rc;

824     ssl_mutex_on(s);
825     if (table_first_r(mc->tSessionCacheDataTable, &iterator,
826                       &vpKey, &nKey, &vpData, &nData) == TABLE_ERROR_NONE) {
827         do {
828             bDelete = FALSE;
829             nElements++;
830             if (nData < sizeof(time_t) || vpData == NULL)
831                 bDelete = TRUE;
832             else {
833                 memcpy(&tExpiresAt, vpData, sizeof(time_t));
834                 if (tExpiresAt <= tNow)
835                     bDelete = TRUE;
836             }
837             vpKeyThis = vpKey;
```

```
838     nKeyThis = nKey;
839     rc = table_next_r(mc->tSessionCacheDataTable, &iterator,
840                         &vpKey, &nKey, &vpData, &nData);
841     if (bDelete) {
842         table_delete(mc->tSessionCacheDataTable,
843                     vpKeyThis, nKeyThis, NULL, NULL);
844         nDeleted++;
845     }
846 } while (rc == TABLE_ERROR_NONE);
847 }
848 ssl_mutex_off(s);
849 ssl_log(s, SSL_LOG_TRACE, "Inter-Process Session Cache (SHM) Expiry: "
850 "old: %d, new: %d, removed: %d", nElements, nElements-nDeleted, nDeleted);
851 return;
852 }
```

ssl_session_cache.c

共享内存会话缓存代码与基于 DBM 的代码极为相似。从编程的角度来看，主要差别就是它使用散列表（table_insert_kd() 和相关函数）而不是 DBM。然而，由于 DBM API 本质上为散列表，所以主要的差别就是调用哪个函数的问题。

会话存储

752~769 与 DBM 代码一样，我们首先使用 ssl_mutex_on() 对缓存进行加锁，然后再存储数据。这里的主要差别就是我们实际上不必像使用 DBM 那样打开数据库。显然，这是一种巨大的性能改进。

会话获取

769~898 同上

会话删除

799~808 同上

缓存到期

809~852 缓存到期要比 DBM 简单得多。因为散列表实现能够正确处理扫描过程中的删除，所以可以一遍完成到期工作。除此之外，这种代码与 DBM 的代码相当相似。



SSLv2

B.1 介绍

本书的重点放在 SSLv3 和 TLS 上。然而，大多数 SSLv3 实现也是 SSLv2 实现，而且一些重要的 Web 站点仍然只支持 SSLv2。因此这个附录提供了 SSLv2 的介绍，该协议在 [Hickman1995] 中有所描述。我们从 SSLv2 的工作原理开始讲起。然而，在讨论 SSLv2 时我们不会像讲解 SSLv3 那么详细，只是像讨论 IPSEC、S-HTTP 和 S/MIME 那样多少停留在同样的抽象层次上。

在设计 SSLv3 时，Netscape 雇佣了专业安全人员，其中就包括著名的密码学家 Taher Elgamal。SSLv3 本身的设计是在安全顾问（Paul Kocher）的协助下完成的。然而，在设计 SSLv2 时，Netscape（当时的 Mosaic Communication）没有雇用任何安全专家。在缺乏真正专业知识的情况下，这项设计任务交由一位没有知名的安全文凭或经验的雇员 Kipp Hickman 来完成，因而 SSLv2 含有多种安全缺陷并不奇怪，这最引出了 SSLv3 的设计。

我们下一项任务就是探讨 SSLv2 的缺陷都表现在哪些地方。首先，SSLv2 缺少几种特性。其次，它有许多安全缺陷。尽管对简单的信用卡提交来说所有这些无妨大碍，不过问题仍相当严重。我们将要对这些问题进行讨论。我们已经看到这些问题是如何在 SSLv3 中得到解决的，但是了解一下 Microsoft 在 PCT（Private Communications Technology，私有通信技术）中如何解决这些问题也是具有意义的。最后我们将简要讨论一下 SSLv1 的有关情况。

B.2 SSLv2 概述

SSLv2 使用与 SSLv3 一样的模型。有确立密钥和磋商算法的握手阶段。在握手完成之后，使用这些密钥和算法来加密应用通信数据。事实上，SSLv2 握手中的许多消息都有着与 SSLv3 握手类似的名称和字段值（这无疑是 SSLv3 设计者的主意）。图 B.1 描述了一个 SSLv2 握手范例。

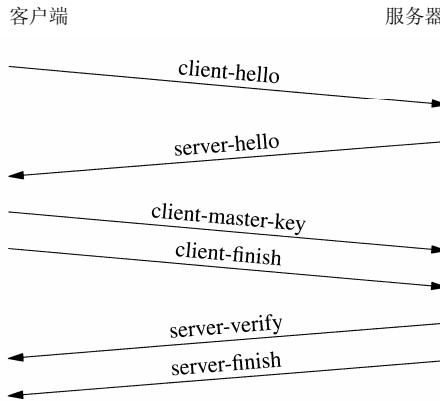


图 B.1 SSLv2 握手

CLIENT-HELLO

在 CLIENT-HELLO 消息中，客户端发送包含它所支持的加密算法的列表、一个随机的 CHALLENGE 以及一个可能的会话 ID。CHALLENGE 与 SSLv3 中随机值的角色大致相同：它为会话密钥提供新鲜因子以使客户端能够检测重放攻击。与 SSLv3 一样，会话 ID 用于会话恢复。

SERVER-HELLO

在 SERVER-HELLO 消息中，服务器提供它所支持的客户端加密算法的子集和证书。与 SSLv3 不同，最后是客户端选择使用哪种加密算法。然而，服务器可以删除任何它不支持的加密算法，服务器还提供了与 SSLv3 中的随机值功用相同的 CONNECTION-ID。

CLIENT-MASTER-KEY

CLIENT-MASTER-KEY 提供以服务器的公用密钥加密的主密钥。为了处理可出口加密算法的情况，将主密钥切分成一段加密的部分以及一段明文部分。因此，对于 40 位的 RC4 而言，加密的是 40 位而 88 位都是以明文形式存在的。然后使用摘要函数将主密钥转换为加密和 MAC 密钥。如果加密算法要求的话，这条消息还提供 IV。IV 以明文形式进行传输。

CLIENT-FINISH

CLIENT-FINISH 消息是第一条加密的消息。它包含 CONNECTION-ID 并证明客户端知道加密密钥。

SERVER-VERIFY

SERVER-VERIFY 消息包含加密的提问 (challenge)，这可以证明服务器知道加密密钥。

SERVER-FINISH

SERVER-FINISH 消息包含会话 ID。如果客户端想要恢复会话的话，客户端就使用这个东西。

会话恢复

SSLv2 还有一种会话恢复模式。如果服务器识别了客户端的会话 ID，那么它就在 SERVER-HELLO 中设置一个标志。当恢复会话时，客户端只需省略 CLIENT-MASTER-KEY 消息，而客户端与服务器会重复使用主密钥。

● 客户端认证

SSLv2 的客户端认证相当简单。服务器有一条附加的称作 REQUEST-CERTIFICATE 的消息，服务器使用它来请求客户端认证。消息包含一个用来产生由客户端签名的数据的提问 (challenge)。客户端使用 CLIENT-CERTIFICATE 消息予以响应，该响应包含客户端的证书和提问 (challenge)，客户端加密密钥和服务器加密密钥上的签名。

● 数据传输

尽管细节不同，SSLv2 数据传输与 SSLv3 数据传输类似。数据被分割成一系列记录。对每条记录单独计算 MAC 并加密。MAC 不像 HMAC，它是一种临时选用的函数，但是没有任何已知的严重安全弱点。

B.3 缺少的功能

SSLv2 缺少几种看起来非常有价值的功能。注意，没有一种功能是绝对必须的，但是缺少这些功能很不方便，所以在设计 SSLv3 的时候增加了进来。

● 证书链

SSLv2 允许客户端和服务器每一方只发送一份证书。因此，这种证书必须直接由根来签署。正如我们在第 5 章所讨论的，如果有一个根 CA 为多个 CA 签名，而这些 CA 本身又来为用户提供担保是相当方便的。在 SSLv2 中这是不可能的。SSLv3 允许客户端和服务器有任意长度的证书链。

● 出口与国内客户端使用相同的 RSA 密钥

在设计 SSLv2 的时候，美国出口法规禁止可出口软件中的密钥交换使用长度超过 512 的密钥。然而，1024-和 512-位的密钥已经是中等程度到高安全应用的标准。因此如果你想让服务器处理高强度的（128-位）密码，那么明显是想用 1024-位的 RSA 密钥。然而，由于密钥还要用于密钥交换，所以不能使用同样的密钥来为出口客户端提供服务。相反，你的服务器需要使用两个密钥。

然而，当 Netscape 首先在 Navigator 中交付 SSLv2 时，他们忽略了出口客户端中的长密钥测试。可出口 Navigator 支持 1024-位的 RSA。尽管如此，也不知是什么原因获得了出口许可。目前还不清楚（至少作者不知道）NSA 是否在审查 Navigator 的时候不知道这个问题，或者只是忽视了这个问题。不管什么情况，这开创了一个先例而其他几个厂商也获得了类似的许可。

然而，当设计 SSLv3 的时候，有情况显示 NSA 要收紧约束。因此 SSLv3 通过使用长 RSA 密钥对 512-位的临时密钥进行签名来符合法规的要求。使用这种技巧，不但只有一个密钥而且可以符合法规的要求。

B.4 安全问题

出口消息认证

SSLv2 的消息认证使用与加密相同的密钥。因此，如果我们在出口模式中使用 RC4 的话，加密与 MAC 密钥基于只有 40 位的保密数据，而且还有 40 位的熵。因此在出口模式中，实施完整性攻击并不比实施保密性攻击困难。攻击者可以简单地穷举搜索所有 40 位的保密主密钥，穷举搜索 40 位的密钥空间对于具有中等实力的攻击者来说不在话下。

与之对照，SSLv3 使用巨大的主密钥并在密钥导出阶段降低熵。你可以穷举搜索加密密钥，但是这无法追溯到主密钥。因此即便加密强度不高——或者没有，MAC 密钥也是高强度的。在 SSLv3 中即便使用弱强度的加密算法，实施完整性攻击也是困难的。

弱强度 MAC

尽管没有已知的针对 SSLv2 MAC 的有效攻击，但是它肯定没有 HMAC 那么健壮。特别的，它主要依赖于 MD5，没有办法使用 SHA-1。近来人们对 MD5 [Dobbertin1996] 的强度有些担忧，因此这一点颇不如人愿，尽管不是致命的问题。至少对这种问题来说多少可以原谅设计者。对 MD5 的攻击和 HMAC 的开发是在 SSLv2 发表之后发生的。然而，不管是什情况，无法磋商使用 SHA 都缺乏必要的灵活性。

降级

SSLv2 没有任何保护握手的措施。因此，攻击者有可能强迫各方磋商使用弱强度的加密算法，即便它们都支持更高强度的密码计算也是如此。SSLv3 使用 Finished 消息解决了这种问题，该消息包含整个握手上的消息摘要。

截断攻击

SSLv2 只使用 TCP 连接关闭来指示数据结束。这意味着有可能受制于截断攻击。攻击者可以简单地伪造 TCP FIN，而接收者无法辨别出它是否是合法的 (legitimate) 数据结束标志。SSLv3 通过使用显式地关闭警示解决了这个问题。

客户端认证传输

PCT 规范 [Benaloh1995] 中描述了一种微妙但却严重的针对 SSLv2 客户端认证的攻击。考虑部分内容可以通过普通 SSL 连接存取，而部分内容要求客户端认证的 Web 站点的保护问题。因为 SSLv2 消息认证在使用出口密码计算的时候如此虚弱，所以服务器会要求在对受保护的 Web 站点存取时使用高强度的加密计算，否则攻击者就可以攻破连接并冒充客户端。

设想客户端已经使用出口加密计算连接到服务器的情况。攻击者通过穷举搜索 40 位的密钥空间攻破了这个连接，因此知道了主密钥和读/写密钥。最终，客户端有可能重新连接到服务器来发送另一个请求，接下来他就可以执行图 B.2 中所示的攻击。

攻击者拦截客户端与服务器之间的 TCP 连接并读取客户端的 CLIENT-HELLO。然后打开他自己的到服务器的连接，提出使用 128 位的加密计算但却使用客户端的提问 (challenge)。它将自己的 SERVER-HELLO 发送给客户端同意恢复连接，然后使用原先连接中的同一个主密钥给服务器发送自己的 CLIENT-MASTER-KEY 消息，但是这一次使用以



服务器的 RSA 密钥加密的整个密钥。此刻，客户端攻击者与服务器攻击者的主密钥均与原来连接中的主密钥相同，因为客户端恢复了那个连接。

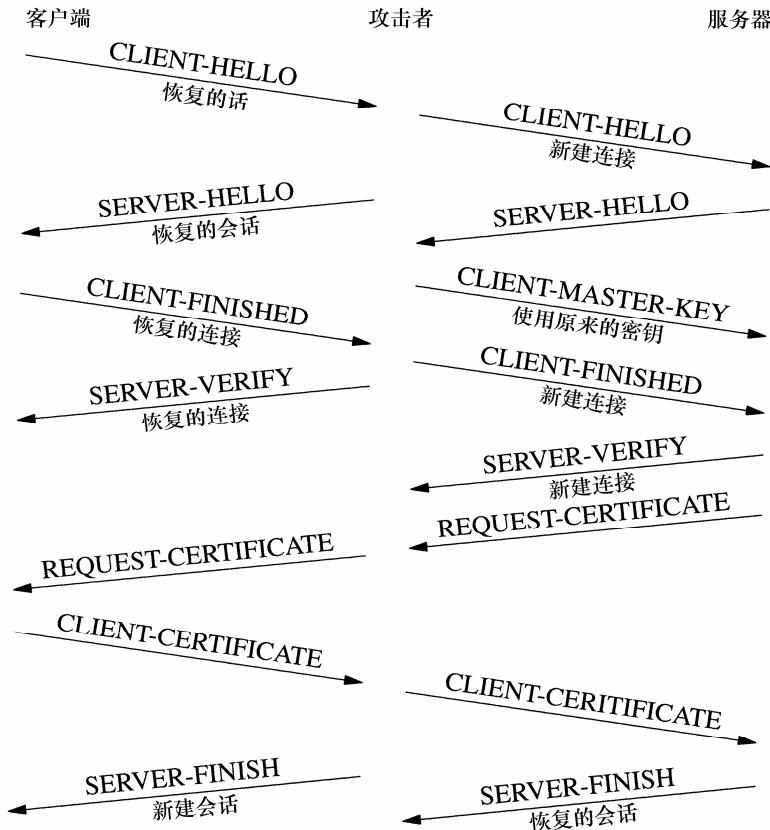


图 B.2 SSLv2 认证传输攻击

服务器使用 REQUEST-CERTIFICATE 消息要求客户端认证，攻击者将这条消息转发给客户端。客户端使用 CLIENT-CERTIFICATE 消息来响应。大家还记得签名数据由客户端写密钥，服务器写密钥以及提问组成。所有这些在客户端攻击者连接以及服务器攻击者连接中都是相同的。因此，攻击者只需将 CLIENT-CERTIFICATE 消息转发给服务器，他就具有了在服务器看起来像是使用高强度加密计算实现客户端认证的连接——只不过它实际上是到攻击者的连接。这种攻击在 SSLv3 和 TLS 中是不可能的，因为在那里认证密钥与加密密钥是分开的。

B.5 PCT

设计自己的协议是 Microsoft 对 SSLv2 中安全错误的回答，这是一种称作私人通信技术（PCT）的 SSLv2 变种。PCT 修正了 SSLv2 中的大多数严重的安全缺陷。特别的，它更正了降级与客户端认证缺陷。它还多少对握手进行了简化，因此减少了为建立连接所需的往返次数。最后它还集成了更多灵活的加密套件选择。

verify-prelude

SSLv3 使用 Finished 消息来阻止主动攻击者的降级攻击。PCT 使用 CLIENT-MASTER-KEY 消息中的 verify-prelude 字段来提供相同的保护。verify-prelude 包含客户端与服务器 hello 消息的 MAC。因此，如果连接被降级的话，服务器能够检测出来。因为 verify-prelude 是与密钥交换一起发送的，所以如果发生任何篡改的话，服务器可以立即拒绝连接——这一点与 SSLv3 不同，它是在整个连接必须完成时才能检测出这种攻击。

加强的消息完整性

PCT 使消息认证与安全成为正交的 (orthogonal, 即互不相干的)。与 SSLv3 一样，传送一个巨大的主密钥并使用这个密钥独立产生 MAC 和加密密钥。即便使用 40-位的加密密钥，MAC 的强度也是极其有力的。因此，即便加密密钥被攻破，消息完整性仍然可以完好无损。

改进的客户端认证

针对 SSLv2 的认证传输攻击依赖于在不管使用什么加密套件的情况下 CLIENT-CERTIFICATE 消息都是相同的。PCT 通过用客户端证书对 verify-prelude 字段进行签名来克服这种攻击。因此这个字段有赖于加密套件，所以传输攻击不再有效。此外，传输攻击有赖于能够恢复使用 40-位加密套件连接的主密码。对 PCT 来说就不再可能，因为即便使用出口加密，主密钥也是够长的。

正交加密套件的磋商

虽然 SSLv2、SSLv3 和 TLS 都有单一的定义所使用加密参数的加密套件，但 PCT 单独指定加密、散列、证书和密钥交换算法。此外，加密算法的规格说明本身是正交的 (orthogonal, 即互不相干的)，因此可以独立于密钥长度来指定加密算法。这种方案的好处就是不必为每种需要的算法组合都注册一个加密套件 (cipher suite)。

反对这种方案的 SSLv3 设计者们所持有的意见是，正交的指定算法会允许构造出不合适或不安全的组合。我并不觉得这种理由多么有说服力，即便加密套件浑为一体，仍有可能创建出不合适的组合（如 4096-位的 RSA 密钥与 DES）。与之对照，我们已经在前面的章节中看到无法在 SSLv3 中正交地指定加密套件会致使无法注册一些有用的组合 (DH/DSS 和 RC4)。

向后兼容

与 SSLv3 一样，当初也非常希望 PCT 可以与 SSLv2 向后兼容的。因此 PCT CLIENT-HELLO 必须表示为 SSLv2 CLIENT-HELLO。然而，由于 Microsoft 没有掌握版本号空间的控制，所以他们不能像 Netscape 那样使用版本 3。相反，迫使 PCT 使用一种迂回措施：设计者指定一种新的指示客户端支持 PCT 的加密套件。这样，看到这个加密套件的 SSLv2 服务器因为无法识别会继续使用 SSLv2，而 PCT 服务器能够执行 PCT 握手。

第二种向后兼容的问题是对 PCT 正交算法选择的支持。设计者使用的方案是将算法全部表示为 SSLv2 加密套件，然后使用位屏蔽来区分各个分类。图 B.3 描述了一个 PCT 兼容的 CLIENT-HELLO 的一部分列印输出 (dump)。

这个 CLIENT-HELLO 实际上是与 SSLv2、PCT 和 SSLv3/TLS 兼容的。因此 SSLv3/TLS 加密套件（表示为 SSLv2 加密套件）也在这个消息中出现。第 1 行指示客户端支持 PCT。

第 2-3 行指示它支持 X.509 证书链和纯 X.509 证书。第 4-5 行指示支持 MD5 和 SHA。第 6 行指示支持 RSA 密钥交换。

要特别注意第 10-11 行和 21-22 行。PCT 有关加密和 MAC 算法的规格说明为四字节长，因此必须将其表示为两个 SSLv2 加密套件，头半部分在第一个中出现，第二部分在下一个中出现。因此第 10-11 行指定支持 128 位 MAC 的 128 位 RC4。这种迂回方案是需要的，因为正交 PCT 加密套件在 SSLv2 所提供的 3 字节空间中装不下。

● 什么是最重要的

在引入 PCT 的时候，Netscape 是当时占主导地位的 Web 浏览器，因此 SSLv3 远比 PCT 更为重要。所以除 Microsoft 之外，没有任何主要的厂商实现了 PCT。尽管 Microsoft 的 Internet Explorer 最终成为占主导地位的浏览器，但是击败 Netscape 发生得太晚了，不足以保全 PCT。尽管可以辩称 PCT 相比 SSLv3/TLS 囊括了更幽雅的设计选择，但是面对部署远为广泛的 SSLv3 而言，它却没有包含任何重要的足以证明使用它才更为合理的功能。

```

1   8f 80 01  PCT_SSL_COMPAT | PCT_VERSION_1
2   80 00 03  PCT_SSL_CERT_TYPE | PCT1_CERT_X509_CHAIN
3   80 00 01  PCT_SSL_CERT_TYPE | PCT1_CERT_X509
4   81 00 01  PCT_SSL_HASH_TYPE | PCT1_HASH_MD5
5   81 00 03  PCT_SSL_HASH_TYPE | PCT1_HASH_SHA
6   82 00 01  PCT_SSL_EXCH_TYPE | PCT1_EXCH_RSA_PKCS1
7   00 00 04  TLS_RSA_WITH_RC4_128_MD5
8   00 00 05  TLS_RSA_WITH_RC4_128_SHA
9   00 00 0a  TLS_RSA_WITH_3DES_EDE_CBC_SHA
10  83 00 04  PCT_SSL_CIPHER_TYPE_1ST_HALF | PCT1_CIPHER_RC4
11  84 80 40  PCT_SSL_CIPHER_TYPE_2ND_HALF | PCT1_ENC_BITS_128 | PCT1_MAC_BITS_128
12  01 00 80  SSL2_RC4_128_WITH_MD5
13  07 00 c0  SSL2_DES_192_EDE3_CBC_WITH_MD5
14  03 00 80  SSL2_RC2_128_CBC_WITH_MD5
15  00 00 09  TLS_RSA_WITH_DES_CBC_SHA
16  06 00 40  SSL2_DES_64_CBC_WITH_MD5
17  00 00 64  TLS_RSA_EXPORT1024_WITH_RC4_56_SHA
18  00 00 62  TLS_RSA_EXPORT1024_WITH_DES_CBC_SHA
19  00 00 03  TLS_RSA_EXPORT_WITH_RC4_40_MD5
20  00 00 06  TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
21  83 00 04  PCT_SSL_CIPHER_TYPE_1ST_HALF | PCT1_CIPHER_RC4
22  84 28 40  PCT_SSL_CIPHER_TYPE_2ND_HALF | PCT1_ENC_BITS_40 | PCT1_MAC_BITS_128
23  02 00 80  SSL2_RC4_128_EXPORT40_WITH_MD5
24  04 00 80  SSL2_RC2_128_CBC_EXPORT40_WITH_MD5
25  00 00 13  TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA
26  00 00 12  TLS_DHE_DSS_WITH_DES_CBC_SHA
27  00 00 63  TLS_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA

```

图 B.3 PCT CLIENT-HELLO

B.6 有关 SSLv1 的情况

此刻你或许在想“我听说过有关 SSLv3 和 SSLv2 的历史，但是 SSLv1 的情况又是怎样

呢？”Netscape 从没有真正交付过 SSL 版本 1 的规范或公开实现。我们刚刚看到 SSLv2 包含了许多中等程度的安全缺陷，可以通过细致的工作加以避免。流传供审阅的 SSLv1 草案除了具有上述这些安全缺陷还包含一种非常严重的安全缺陷：弱强度的完整性保护。

最早的 SSLv1 草案根本就没有任何完整性保护。几乎难以相信的是它也使用 RC4，攻击者能够在明文中施加可预测的改动。此外，它没有任何序号，因此完全受制于重放攻击。

后来的版本增加了序号和校验码（checksum），但使用 CRC（循环冗余校验）作为校验码。不幸的是 CRC 不是不可逆的或无冲突的，而 MD5 却是这样，因此它在使用 RC4 时无法阻止消息完整性攻击。最后，就在发行 SSLv2 之前，设计者们搞清了有关完整性攻击的知识并将 MAC 更改为 MD5，从而解决了这一问题。对于这些改动，在很大程度上应当感谢 Martin Abadi，是他向 Netscape 指出了存在的问题。

参考文献

所有的 RFC 都可以从 Web 上免费下载, 请参见 2.2 节的描述。

[Abbott1988] Abbott, S., and Keung, S., *CryptoSwift(ver.2)Performance on Netscape Enterprise Server* (April, 1988).

<http://isglabs.rainbow.com/isglabs/NS351-CSv2-NT-perf/NS351-CSv2.html>

Describes performance behavior of a commercial (though somewhat outdated) server with and without hardware acceleration.

[Anderson1999] Anderson, R., Biham, A., and Knudsen, L., *Serpent: A Proposal for the Advanced Encryption Standard*(1999).

<http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Serpent/Serpent.pdf>

[ANSI1985] ANSI, "American National Standard for Financial Institution Key Management(wholesale)," ANSI X9.17 (1985).

Describes 3DES.

[ANSI1986] ANSI, "American National Standard for Financial Institution Message Authentication (Wholesale)," ANSI X9.9(Revised)(1986).

One of many descriptions of DES-CBC MAC.

[ANSI1995] ANSI, "Public key cryptography for the financial services industry-Certificate management," ANSI X9.57 (1995).

[ANSI1998] ANSI, "Agreement of Symmetric Keys Using Diffie-Hellman and MQV Algo-rithms," ANSI X9.42 draft (1998).

[Atkins1996] Atkins, D., Stallings, W., and Zimmermann, P., "PGP Message Exchange For-mats," RFC 1991 (August 1996).

[Balenson1993] Balenson, D., "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers," RFC 1423 (February 1993).

[Banes1999] Banes, J., and Harrington, R., "56-bit Export Cipher Suites for TLS," draft-ietf-tls-56-bit-ciphersuites-00.txt(April 1999).

[Banes2000] Banes, J., *Personal communication.* (2000).

[Bellare1995] Bellare, M., and Rogaway, P., "Optimal asymmetric encryption padding,"*Advances in Cryptology –Eurocrypt 94 Proceedings*, Springer-Verlag, Berlin (1995).

[Bellovin1995] Bellovin, S.M., "Using the Domain Name System for System Break-ins" in *5th USENIX UNIX*

- Security Symposium*, p. 199-208, USENIX, Salt Lake City, UT (June 5-7, 1995).
- [Benaloh1995] Benaloh, J., Lampson, B., Simon, D., Spies, T., and Yee, B., "Private Communication Technology Protocol," draft-microsoft-PCT-01.txt (September 1995). The second PCT draft
- [Berners-Lee1994] Berners-Lee, T., Masinter, L., and McCahill, M., "Uniform Resource Locators," RFC 1738 (December 1994).
- [Berners-Lee1998] Berners-Lee, T., Fielding, R., and Masinter, L., "Uniform Resource Identifiers (URI)," RFC 2396 (August 1998).
- [Biham1991a] Biham, E., Shamir, A., and Differential Cryptanalysis of DES-like Cryptosystems, *Advances in Cryptology--CRYPTO '90 Proceedings*, p. 2-21, Springer-Verlag, Berlin (1991).
- [Biham1991b] Biham, E., Shamir, A., and Differential Cryptanalysis of DES-like Cryptosystems, *Journal of Cryptology*, 4, 1, p. 3-72 (1991).
- [Biham1993a] Biham, E., and Shamir, A., "Differential Analysis of the Full 16-Round DES," *Advances in Cryptology--CRYPTO '92 Proceedings*, p. 487-496, Springer-Verlag, Berlin (1993).
- [Biham1993b] Biham, E., and Shamir, A., *Differential Analysis of the Data Encryption Standard*, Springer-Verlag, New York, N.Y. (1993).
- [Blaze1996] Blaze, M., Diffie, W., Rivest, R., Schneier, B., Shimomura, T., Thompson, E., and Weiner, M., *Minimal Key Lengths for Symmetric Ciphers to Provide Adequate Commercial Security* (January 1996).
- [Blaze1999] Blaze, M., Feigenbaum, J., Ionnidis, J., and Keromytis, A., "The Keynote Trust Management System, Version 2," RFC 2704 (September 1999).
- [Bleichenbacher1998] Bleichenbacher, D., "Chosen Ciphertext Attacks against Protocols Based on RSA Encryption Standard PKCS #1," *Advances in Cryptology -- CRYPTO 98*, p.1-12(1998).
- [Bleichenbacher1999] Bleichenbacher, D., *Personal communication*. (1999).
- [Boe1999] Boe, M., "TLS-based Telnet Security," draft-ietf-tn3270-telnet-tls-03.txt (October, 1999). Telnet over TLS has never been standardized, but this document describes the best current practice.
- [Burwick1999] Burwick, C., Coppersmith, D., D'Avignon, E., Genarro, R., Halevi, S., Jutla, C., Matyas, S.M. Jr., O'Connor, L., Peyravian, M., Safford, D., and Zunic, N., *MARS, a Candidate Cipher for AES*, IBM Corporation (September 1999).
<http://csrc.nist.gov/encryption/aes/round2/AESAlgs/MARS/mars.pdf>
- [Cocks1973] Cocks, C., "A Note on Non-Secret Encryption," CESG Report, CESG (1973).
<http://www.cesg.gov.uk/about/nsecret/notense.htm>
Describes CESG's invention of what we call RSA.
- [Crispin1996] Crispin, M., "Intemet Message Access Protocol--Version 4rev1," RFC 2060 (December 1996).
- [Crocker1982] Crocker, D., "Standard for the Format of ARPA Internet Text Messages," RFC 822 (August 1982).
The basic standard for internet email messages.
- [Daemen1999] Daemen, J., and Rijmen, V., *AES Proposal: Rijndael* (March 1999).
<http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Rijndael/Rijndael.pdf>
- [Dai2000] Dai, W., *Crypto++ 3.1 Benchmarks* (2000).

<http://www.eskimo.com/~weidai/benchmarks.html>

Wei Dai's benchmarks for his Crypto++ package.

[Dierks1999] Dierks, T., and Allen, C., "The TLS Protocol Version 1.0," RFC 2246 (January 1999).

[Diffie1976] Diffie, W., and Hellman, M.E., "New directions in cryptography," *IEEE Transactions on Information Theory*, 22, 6, p. 655-654 (1976).

[Diffie1992] Diffie, W., van Oorschot, P.C., and Wiener, M.J., "Authentication and Authenticated Key Exchanges," *Designs, Codes, and Cryptography*, 2, p. 102-125 (1992).

Describes the Station to Station protocol (STS), a form of authenticated DH upon which IKE is based.

[Dobbertin1996] Dobbertin, H., "The Status of MD5 After a Recent Attack," *CryptoBytes*, 2, 2, RSA Laboratories (Summer 1996).

[Dusse1998] Dusse, S., Hossman, P., Ramsdell, B., Lundblade, L., and Repka, L., "S/MIME Version 2 Message Specification," RFC 2311 (March 1998).

[Eastlake1994] Eastlake, D. 3rd., Crocker, S., and Schiller, J., "Randomness Recommendations for Security," RFC 1750 (December 1994).

[Eastlake1999] Eastlake, D., 3rd., "Domain Name System Security Extensions," RFC 2535 (March 1999).

[Ellis1970] Ellis, J.H., *The Possibility of Non-Secret Encryption*, CESG (1970).

<http://www.cesg.gov.uk/about/nsecret/possnse.htm>

[Ellis1987] Ellis, J.H., *The Story of Non-Secret Encryption*, CESG (1987).

<http://www.cesg.gov.uk/about/nsecret/ellis.htm>

[Ellison1999] Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., and Ylonen, T., "SPKI Certificate Theory," RFC 2693 (September 1999).

[Fielding1999] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and Berners-Lee, T., "Hypertext Transfer Protocol," RFC 2616 (June 1999).

[Ford-Hutchinson2000] Ford-Hutchinson, Paul, Carpenter, M., Hudson, T., Murray, E., and Wiegand, V., "Securing FTP with TLS," draft-murray-auth-ftp-ssl-05.txt (January 2000).

FTP over TLS has never been standardized. This document describes the de facto standard.

[Freed1996a] Freed, N., and Borenstein, N., "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies," RFC 2045 (November 1996).

[Freed1996b] Freed, N., and Borenstein, N., "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types," RFC 2046 (November 1996).

[Freed1996c] Freed, N., Klensin, J., and Postel, J., "Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures," RFC 2048 (November, 1996).

[Freed1996d] Freed, N., and Borenstein, N., "Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples," RFC 2049 (November 1996).

[Freier1996] Freier, A.O., Carlton, P., and Kocher, P.C., *The SSL Protocol Version 3.0* (November 1996).

<http://home.netscape.com/eng/ss13/draft302.txt>

The last published draft of SSLv3. Note that SSLv3 was never published as an IETF RFC of any kind.

This document combined with Netscape's implementation represent the de facto standard.

[Gilmore1998] Gilmore, J. (Ed.), *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip*

Design, O'Reilly & Associates (May 1998).

Describes how to build a dedicated hardware DES search engine.

[Goland1999] Goland, Y., Whitehead, E., Faizi, A., Carter, S., and Jensen, D., "HTTP Extensions for Distributed Authoring--WEBDAV," RFC 2518 (February 1999).

[Goldberg1996] Goldberg, I., and Wagner, D., "Randomness and the Netscape Browser," *Dr.Dobb's Journal* (January 1996).

The paper that described the weakness in Netscape PRNG seeding.

[Harkins1998] Harkins, D., Carrel, D., and The Internet Key Exchange (IKE), RFC 2409 (November 1998).

[Hennessey1996] Hennessey, J., Goldberg, D., and Patterson, D.A., *Computer Architecture: A Quantitative Approach, 2ed.*, Morgan Kaufmann (January 1996).

The standard book on processor design by some of the inventors of RISC.

[Hickman1995] Hickman, K., *The SSL Protocol* (February 1995).

http://www.netscape.com/eng/security/SSL_2.html

The specification for SSLv2.

[Hoffman1999a] Hoffman, P., "SMTP Service Extension for Secure SMTP over TLS," RFC 2487 (January 1999).

The standard for SMTP over TLS.

[Hoffman1999b] Hoffman, P., "Enhanced Security Services for S/MIME," RFC 2634 (June 1999).

[Housley1999a] Housley, R., Ford, W., Polk, W., and Solo, D., "Internet X.509 Public Key Infrastructure Certificate and CRL Profile," RFC 2459 (January 1999).

[Housley1999b] Housley, R., "Cryptographic Message Syntax," RFC 2630 (June 1999).

CMS is a variant of PKCS #7 that has been extended to support Diffie-Hellman and DSA.

[IANA] IANA, *Well Known Ports*.

<http://www.iana.org/numbers.html>

A list of all the assigned well known ports.

[ITU1988a] ITU, "The Directory--Authentication Framework," ITU Recommendation X.509 (1988).

[ITU1988b] ITU, "The Directory--Models," ITU Recommendation X.500 (1988).

[ITU1988c] ITU, "Specification of Abstract Syntax Notation One (ASN. 1)," ITU Recommendation X.208 (1988).

[ITU1988d] ITU, "Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN. 1)," ITU Recommendation X.209 (1988).

[Jablon1996] Jablon, D., "Strong Password-only Authenticate Key Exchange," *ACM Computer Communications Review*, 26, 5 (October 1996).

[Jacobsen1988] Jacobsen, V., "Congestion Avoidance and Control," *Computer Communication Review*, 18, 4, p. 314-329 (August 1988).

[JavaSoft1999] JavaSoft, *Java Secure Socket Extension (JSSE 1.0)* (1999).

<http://java.sun.com/products/jsse/>

Sun's free "non-commercial" quality implementation of SSL/TLS.

[Johnson1993] Johnson, D.B., Matyas, S.M., Le, A.V., and Wilkins, J.D., "Design of the Commercial Data Masking Facility Data Privacy Algorithm" in 1st ACM Conference on

- Computer and Communications Security*, p. 93-96, ACM Press (1993).
- [Joncheray1995] Joncheray, L., "A Simple Active Attack Against TCP" in *5th USENIX UNIX Security Symposium*, p. 7-19, USENIX, Salt Lake City, UT (June 5-7, 1995).
- [Kaliski1993] Kaliski, B., "Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services," RFC 1424 (February 1993).
- [Kaliski1998a] Kaliski, B., and Staddon, J., "PKCS #1: RSA Cryptography Specifications Version 2.0," RFC 2437 (October 1998).
- [Kaliski1998b] Kaliski, B.S., Jr., "Compatible cofactor multiplication for Diffie-Hellman primitives," *Electronics Letters*, 34, 25, p. 2396-2397 (December 1998).
- [Kaufman1995] Kaufman, C., Perlman, R., and Speciner, M., *Network Security: Private Communications in a Public World*, Prentice-Hall, Englewood Cliffs, NJ (1995).
An excellent introduction to cryptography and network security. Focuses on the application of cryptography to communications security.
- [Kelsey1999] Kelsey, J., Schneier, B., and Ferguson, N., "Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator" in *Sixth Annual Work shop on Selected Areas in Cryptography*, Springer-Verlag, Berlin (August 1999).
- [Kent1993] Kent, S., "Privacy Enhancement for Internet Electronic Mail: Part II: CertificateBased Key Management," RFC 1422 (February 1993).
- [Kent1998a] Kent, S., and Atkinson, R., "IP Authentication Header," RFC 2402 (November, 1998).
- [Kent1998b] Kent, S., Atkinson, R., and RFC 2406, *IP Encapsulating Security Payload (ESP)* (November 1998).
- [Kent1998c] Kent, S., and Atkinson, R., "Security Architecture for the Internet Protocol," RFC 2401 (November 1998).
- [Keung] Keung, S., *Cryptoswift performance under SSL with file transfer* (19XX).
<http://isglabs.rainbow.com/isglabs/SSLperformance/SSL+file%20performance.html>
- [Khare2000] Khare, R., and Lawrence, S., "Upgrading to TLS Within HTTP/1.1," RFC 2817 (May 2000).
- [Klein1990] Klein, D.V., "Foiling the Cracker": A Survey of and Improvements to Password Security (1990).
A good description of how easy it is to crack passwords when users choose them.
- [Klensin1995] Klensin, J., Freed, N., Rose, M., Stefferud, E., and Crocker, D., "SMTP Service Extensions," RFC 1869 (November 1995).
- [Kocher1996a] Kocher, P., *A Quick Introduction to Revocation Trees* (1996).
http://www.valicert.com/pdf/Certificate_revocation_trees.pdf
- [Kocher1996b] Kocher, P., *Timing Attacks on Implementation of Diffie-Hellman, RSA, DSS, and Other Systems* (1996).
- [Kocher1999] Kocher, P., and Jun, B., *The Intel Random Number Generator* (April 1999).
This paper describes an analysis of the Intel hardware PRNG in the Pentium III.
- [Krawczyk1995] Krawczyk, H., *SKEME: A Versatile Secure Key Exchange Mechanism forInternet* (August 1995).
- [Krawczyk1996] Krawczyk, H., *Personal communication*. (1996).

HMAC is believed to be immune to Dobbertin's attacks on MD5.

[Krawczyk1997] Krawczyk, H., Bellare, M., and Canetti, R., "HMAC: Keyed-Hashing for Message Authentication," RFC 2104 (February 1997).

[Lear1994] Lear, E., Fair, E., and Kessler, T., "Network 10 Considered Harmful (Some Practices Shouldn't be Codified)," RFC 1627 (June 1994).

The RFC that launched an extremely acrimonious debate over NAT.

[Lim1997] Lim, C.H., and Lee, P.J., "A key recovery attack on discrete log-based schemes using a prime order subgroup" in *Advances in Cryptology--Crypto 97*, p. 249-263, Springer-Verlag, Berlin (1997).

[Linn1993] Linn, J., "Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures," RFC 1421 (February 1993).

[Maughan1998] Maughan, D., Schertler, M., Schneider, M., and Turner, J., "Internet Security Association and Key Management Protocol (ISAKMP)," RFC 2408 (November 1998).

[Medvinsky1999] Medvinsky, A., and Hur, M., "Addition of Kerberos Cipher Suites to Transport Layer Security (TLS)," RFC 2712 (October 1999).

[Menezes1996] Menezes, A.J., van Oorschot, P.C., and Vanstone, S.A., *Handbook of Applied Cryptography*, CRC Press, Boca Raton, FL (1996).

Not a useful introduction but an extremely useful technical reference to cryptography.

[Microsoft2000] Microsoft, *Security Support Provider Interface* (2000).

http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/~secpapi/portalsspi_1545.htm

Documents Microsoft's built-in security services including SChannel.

[Miller1987] Miller, S., Neumann, B., Schiller, J., and Saltzer, J., "Kerberos Authentication and Authorization System," *Project Athena Technical Plan*, MIT Project Athena (December 1987).

[Mockapetris1987a] Mockapetris, P.V., "Domain Names--Concepts and Facilities," RFC 1034 (November 1987).

[Mockapetris1987b] Mockapetris, P.V., "Domain Names--Implementation and Specification," RFC 1035 (November 1987).

[Moeller1998] Moeller, B., "Export-PKC attacks on SSL 3.0/TLS 1.0," *Message to IETF-TLS mailing list* (October 1998).

<http://www.ietf-tls/mail-archive/msg01671.html>

[Mogul1995] Mogul, Jeffrey C., "The Case for Persistent-Connection HTTP," Research Report 95/4 (May 1995).

<http://www.research.digital.com/abstracts/95.4.html>

An early paper showing the benefits of HTTP retained connections.

[Moore1996] Moore, K., "MIME (Multipurpose Internet Mail Extensions) Part Three: Message Header Extensions for Non-ASCII," RFC 2047 (November 1996).

[Myers1996] Myers, J., and Rose, M., "Post Office Protocol--Version 3," RFC 1939 (May 1996).

[Myers1999] Myers, M., Ankney, R., Malpani, A., Galperin, S., and Adams, C., "X.509 Internet Public Key Infrastructure Online Certificate Status Protocol--OCSP," RFC 2560 (June 1999).

[Nagle1984] Nagle, J., "Congestion Control in IP/TCP Internetworks," RFC 0896 (Jan 1984).

- [Needham1978] Needham, R.M., and Schroeder, M.D., "Using Encryption for Authentication in Large Networks of Computers," *Communications of the ACM*, 21, p. 993-999 (December 1978).
- [Netcraft2000] Netcraft, *Netcraft Secure Web Server Survey* (January 2000). Netcraft kindly provided me with a copy of this survey.
- [Netscape 1995a] Netscape Communications Corp, *SSL 2.0 Certificate Usage* (1995).
http://www.netscape.com/cng/security/ssi_2.0_certificate.html
Describes Netscape's wildcarding technique.
- [Netscape1999a] Netscape Communications Corp., *Netscape Certificate Extensions, Communicator 4.0 Version (1999)*.
http://www.netscape.com/eng/security/comm4tcert_exts.html
- [Neumann1951] von Neumann, J., "Various Techniques Used in Connection with Random Digits," *Applied Mathematics Series*, 12, p. 36-38, U.S. National Bureau of Standards (1951).
- [Newman1999] Newman, C., "Using TLS with IMAP, POP3 and ACAP," RFC 2595 (June 1999).
Provides upward negotiation mechanisms for IMAP, POP and ACAP.
- [NIST1993a] National Institute of Standards and Technology (NIST), "Data Encryption Standard," FIPS PUB 46-2, U.S. Department of Commerce (December 1993).
This is the reissued DES document. It's essentially identical to the document published in 1977.
- [NIST1994a] National Institute of Standards and Technology (NIST), and Secure Hash Standard, FIPS PUB 180-1, U.S. Department of Commerce (May 1994).
The revised SHA draft that describes SHA-1.
- [NIST1994b] National Institute of Standards and Technology, "Security Requirements for Cryptographic Modules," FIPS PUB 140-1, U.S. Department of Commerce (January 1994).
Describes four levels of secure modules, ranging from Level 1 (requiring approved algorithms but running on general purpose computers) to Level 4 (full tamperproofing.)
- [Orman1998] Orman, H., "The OAKLEY Key Determination Protocol," RFC 2412 (November 1998).
- [Padmanabhan1995] Padmanabhan, V.N., "Improving World Wide Web Latency," UCB/CSD 95-875, Computer Science Division, University of California, Berkeley (May 1995).
- [Postel1982] Postel, J., "Simple Mail Transfer Protocol," RFC 821 (August 1982).
The base standard for Intemet mail transport.
- [Postel1985] Postel, J., and Reynolds, J.K., "File Transfer Protocol," RFC 959 (October 1985).
- [Postel 1991a] Postel, J., "Intemet Protocol," RFC 791 (September 1991).
The IETF standard for IP.
- [Postel1991b] Postel, J., "Internet Control Message Protocol," RFC 792 (September 1991).
- [Postel 1991c] Postel, J., "Transmission Control Protocol," RFC 793 (September 1991).
The IETF standard for TCP.
- [Ramsdell1999] Ramsdell, B., "S/MIME Version 3 Message Specification," RFC 2633 (June 1999).
- [Rekhter1994] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G.J., and RFC 1597, *Address Allocation for Private Internets* (March 1994).
The original RFC describing NAT and Network 10.

[Rekhter1996] Rekhter, Y., Moskowitz, B., Karrenberg, D., de Groot, G.J., and Lear, E., RFC 1918 (February 1996).

This document represents a truce of sorts in the NAT debate.

[Relyea1996] Relyea, B., *Appendix A--SSL Protocol Version 3.0 Specification Errata for Fortezza Implementations* (November, 1996).

http://www.armadillo.huntsville.al.us/Fortezza_docs/ssl_fortezza.pdf

Describes how to make SSLv3 work with FORTEZZA.

[Rescorla1999a] Rescorla, E., and Schiffman, A., "The Secure HyperText Transfer Protocol," RFC 2660 (August 1999).

[Rescorla1999b] Rescorla, E., and Schiffman, A., "Security Extensions for HTML," RFC 2659 (August 1999).

[Rescorla2000] Rescorla, E., "HTTP over TLS," RFC 2818 (May 2000).

[Rivest1979] Rivest, R.L., Shamir, A., and Adelman, L.M., "On Digital Signatures and PublicKey Cryptosystems," Technical Report, MIT/LCS/TR-212, MIT Laboratory for Computer Science (January 1979).

[Rivest1983] Rivest, R., Shamir, A., and Adleman, L.M., "Cryptographic communications system and method," US Patent 4405829 (September 1983).

The RSA Patent.

[Rivest1992] Rivest, R., "The MD5 Message-Digest Algorithm," RFC 1321 (April 1992).

[Rivest1995] Rivest, R., Robshaw, M.J.B., Sidney, R., and Yin, Y.L., *The RC6TM Block Cipher* (August 1995).

<http://csrc.nist.gov/encryption/aes/round2/AESAlgs/RE6/cipher.pdf>

[Rivest1998] Rivest, R., "A Description of the RC2(r) Encryption Algorithm," RFC 2268 (January 1998).

[RSA1993a] Kalish, B.S., Jr., "A Layman's Guide to a Subset of ASN. 1, BER, and DER," Technical Note, RSA Laboratories (November 1993).

Provides a readable introduction to ASN. 1, BER, and DER.

[RSA1993b] RSA Laboratories, "RSA Encryption Standard," PKCS #1 (November 1993).

[RSA1993c] RSA Laboratories, "Cryptographic Message Syntax Version 1.5," PKCS #7 (November 1993).

[RSA1993d] RSA Laboratories, "Password Based Encryption Standard," PKCS #5 (November 1993).

[RSA1999a] RSA Laboratories, "Personal Information Exchange Syntax," PKCS #12 (June 1999).

[RSA1999b] RSA Laboratories, "Password Based Encryption Standard," PKCS #5v2.0 (March 1999).

[Saltzer1984] Saltzer, J.H., Reed, D.P., and Clark, D.D., "End-to-End Arguments in System Design," *ACM Transactions in Computer Systems*, 2, 4, p. 277-288 (November 1984).

The classic description of the end-to-end argument, a basic networking design principle.

[Schneier1996a] Schneier, B., *Applied Cryptography*, 2ed., John Wiley & Sons, New York, N.Y. (1996).

The standard text on cryptography.

[Schneier1996b] Schneier, B., and Wagner, D., "Analysis of the SSL 3.0 Protocol," *The Second USENIX Workshop on Electronic Commerce Proceedings*, p. 29-40, USENIX Press (November 1996).

[Schneier1998] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., and Ferguson, N., *Twofish: A 128-Bit Block Cipher* (June 1998).

<http://csrc.nist.gov/encryption/aes/round2/AESAlgs/Twofish/J>



Twofish.pdf

- [Schnorr1991] Schnorr, K., "Method for Identifying Subscribers and for Generating and Verifying Electronic Signatures in a Data Exchange System," US Patent 4995082 (Feb 1991).
- [Shamir1999] Shamir, A., "Factoring Large Numbers with the TWINKLE Device," Eurocrypt '99 *Rump Session* (1999).
- [Spero 1994] Spero, S., *Analysis of HTTP Performance Problems* (1994).
<http://sunsite.unc.edu/mdma-release/http-prob.html>
- [Stevens1994] Stevens, W.R., TCP/IP Illustrated, Volume 1: The Protocols, Addison-Wesley, Reading, MA (1994).
The classic book on TCP/IP
- [Voydock1983] Voydock, V., and Kent, S.T., "Security mechanisms in high-level network protocols," *ACM Computing Surveys*, 15, p. 135-171 (1983).
An early survey of various mechanisms for providing cryptographic security in networks. Voydock and Kent discuss many of the same issues that we've seen in this chapter in the context of early network protocols. Kent has been a key figure in the design of a number of important public and classified security protocols, including IPsec and PEM.
- [W3C2000] W3C, *Naming and Addressing: URIs, URLs* (2000).
<http://www.w3.org/Addressing/>
A good guide to the relationship between URIs and URLs.
- [WAP1999a] Wireless Application Protocol Forum, *WAP WTLS* (Nov 1999).
- [Williamson1974] Williamson, M., *Non-Secret Encryption Using a Finite Field*, CESG (1974).
<http://www.cesg.gov.uk/about/nsecret/secenc.htm>
Describes CESG's invention of a system which is essentially Diffie-Hellman.
- [Williamson1976] Williamson, M., *Thoughts on Cheaper Non Secret Encryption*, CESG (1976).
<http://www.cesg.gov.uk/about/nsecret/cheapnse.htm>
- [Wu1998] Wu, T., "The Secure Remote Password Protocol," *Proceedings of the 1998 Internet Society Network and Distributed Systems Security Symposium*, p. 97-111 (March 1998).