# 基于OpenSSL的安全WEB服务器实现

## 本章训练目的与要求

Web使用的传输协议是HTTP协议。HTTP采用明文传输，网络传输中的重要数据有被第三方截获的危险。因此，为了保护敏感数据在传送过程中的安全，安全超文本传输协议（Hypertext Transfer Protocol over SSL，HTTPS）应运而生。HTTPS就是经过安全套接字协议层（Secure Socket Layer，SSL）加密后的HTTP，它可以与HTTP信息模型共存并容易与HTTP应用程序相整合。HTTPS使用SSL在发送方将原始数据进行加密，然后在接收方进行解密，加密和解密需要发送方和接收方通过交换共享密钥来实现，因此，所传送的数据不容易被截获和解密。

本章训练目的如下。

①理解HTTPS协议及SSL协议的工作原理。

②掌握使用OpenSSL编程的方法。

③提高网络安全系统设计的能力。

本章训练要求如下。

①运行于Linux平台。

②利用OpenSSL库编写一个Web服务器。

③实现HTTPS协议下最基本的GET命令功能。

## 相关背景知识

### SSL协议介绍

SSL是Netscape公司于1996年提出的安全通信协议，它为网络应用层的通信提供了认证、数据保密和数据完整性的服务，较好的解决了Internet上数据安全传输的问题。设计SSL的主要目的是为网络环境中两个通信应用进程之间提供一个安全通道。

图6-1给出了SSL协议栈的结构示意图。SSL借助TCP协议来提供端到端的安全服务，SSL并不是一个单独的协议，而是两层结构的协议集合，上层包括SSL握手协议、SSL修改密文规约协议和SSL警告协议，下层包括SSL记录协议。

|  |  |  |  |
| --- | --- | --- | --- |
| SSL握手协议 | SSL修改密文规约协议 | SSL警告协议 | HTTP |
| SSL记录协议 | | | |
| TCP | | | |
| IP | | | |

图6‑1 SSL协议栈

#### SSL记录协议

SSL记录协议为通信提供机密性和完整性保护，图6-2给出了该协议的工作流程，具体步骤如下。

1. 接收到应用层数据以后，SSL记录协议首先对其进行分组，分组后数据块的长度不超过214（16384）B；
2. 对数据块进行压缩，压缩过程中不能出现信息的丢失，同时增加的长度不能超过1024B（压缩过程是可选的，现有的SSL3.0和TLS1.0都没有指定压缩算法）；
3. 在压缩后的数据上计算消息认证码MAC，并把MAC附加在数据块之后；
4. 对压缩保温和MAC进行加密，加密可以采用流加密或者块加密方式，如果采用块加密的话，必要时需要进行填充；
5. 为加密后的数据添加SSL记录协议的头部。

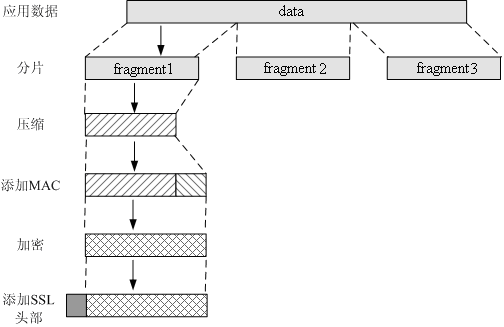


图6‑2 SSL记录协议操作流程

#### SSL握手协议

SSL握手协议对SSL会话状态进行维护，为通信双方建立安全的传输通道，它是SSL协议中最复杂的部分。

当SSL客户端和服务器首次通信时，双方通过握手协议，协商通信协议版本号、选择密码算法、互相认证身份（可选），并使用公钥加密技术通过一系列的交换消息在客户端和服务器之间生成共享的秘密。双方根据这个秘密信息产生数据加密算法和HASH算法的参数等。

握手协议在应用层数据传输之前进行，包含一系列服务器与客户端的报文交换，这些报文都含有3个字段，即消息类型（1B）、消息长度（3B）、消息内容（不少于1B）。

图6-3描述了客户端和服务器建立连接时的报文交换过程，整个交换过程可以分为4个阶段。

1. 发起阶段：

客户端发起client\_hello类型的连接请求，其中包含的参数有客户端所支持的SSL协议的最高版本号、随机码、会话ID、密码套件（cipher suite，包括密钥交换算法和加密、认证算法）、压缩算法等，发出client\_hello消息之后，客户端等待服务器的回应。服务器反馈server\_hello类型的消息，其中版本字段是客户端和服务器都支持的最高版本号，另外还有一个与客户端相互独立的随机码，同时服务器还从客户端所提供的密码套件中确定后面所使用的密钥交换算法、加密/认证算法、压缩算法。

1. 服务器认证和密钥交换：

服务器用certificate消息向客户端发送自己的证书，根据密钥交换算法的不同，server\_key\_exchange消息包含有该算法的相关参数。如果不允许匿名的客户端，则服务器发送certificate\_request消息，会要求客户端提供证书。这一阶段的最后消息是server\_done，这条消息表明服务器的相关报文已经发送完毕，接下来等待客户端的回应。

1. 客户端认证和密钥交换：

收到服务器server\_done类型的消息以后，客户端需要验证服务器提供的证书是合法的，并且server\_hello所包含的参数是可行的。如果服务器需要客户端提供证书的话，客户端会用certificate消息向服务器发送证书。在client\_key\_exchange消息中，根据协商所确定的算法，客户端向服务器发送相应的参数信息。

1. 结束阶段：

客户端和服务器分别向对方发送change\_cipher\_spec和finished类型的消息，至此握手的4个阶段全部结束，双方后续的通信全部采用协商确定的加密/认证算法和密钥进行。

#### SSL修改密文规约协议

修改密文规约协议工作在SSL记录协议之上，是SSL协议集中最简单的，它只包含一种报文格式，报文内容只有一个字节，值为1，用于通知对方消息发出后发送的报文将采用握手协议中协商好的算法、密钥等进行。

#### SSL警告协议

当通信过程中出现错误或异常情况时给出警告或者种植连接，根据错误的严重程度，Alert消息分为两种类型：warning和fatal，其中出现fatal类型的消息将立即关闭连接。

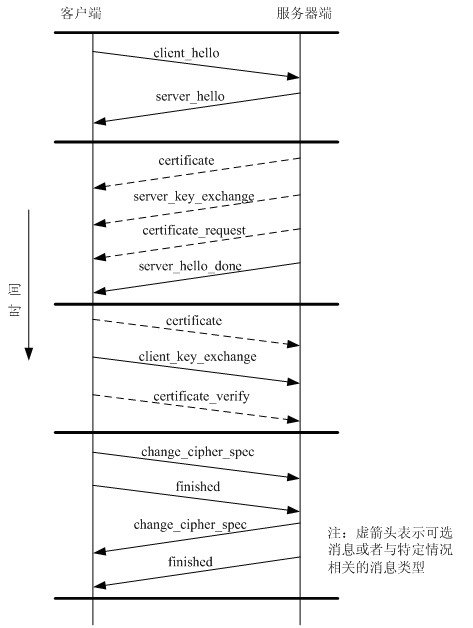


图6-3 SSL握手协议工作过程

### OpenSSL库

OpenSSL库最早发布于1998年，其前身是Eric Young和Tim Hudson开发的SSLeay库，目前最新的版本是0.9.8k。OpenSSL库提供了完全的、免费的SSL协议实现，支持SSL2.0、3.0以及TSL1.0等版本，并且能工作于大部分主流的系统平台上，如UNIX、Linux、Windows等。OpenSSL库支持最常用的对称加密算法、公钥算法、消息摘要算法等，提供了命令行接口和编程的API接口。

#### Linux环境下OpenSSL库的安装

1. 从官方网站<http://www.openssl.org/>下载OpenSSL源代码（openssl-0.9.8k.tar.gz）。
2. 将源代码包解压缩：

# tar zxvf openssl-0.9.8k.tar.gz

1. 设置安装目录并开始安装

# cd openssl-0.9.8k

# ./config --prefix=/usr/local/openssl

# make

# make install

#### OpenSSL命令简介

1. 指令genrsa用于生成RSA私有密钥,其使用格式如下。

openssl genrsa [-out filename] [-passout arg] [-des] [-des3] [-idea] [-f4] [-3] [-rand file(s)] [numbits]

下面对其选项内容进行介绍：

1. -out filename：私有密钥输入文件名，缺省为标准输出。
2. -passout arg：用来把保护key文件的密码输出
3. -des|-des3|-idea ：加密的密钥的加密算法，一般会要输入保护密码，如果这三个中一个也没set， 密钥将不被加密而输入。
4. -F4|-3：使用的公共组件，一种是3， 一种是F4， 我也没弄懂这个option是什么意思。
5. -rand file(s)：产生key的时候用过seed的文件，可以把多个文件用冒号分开一起做seed.
6. numbits：指明产生的参数的长度。必须是本指令的最后一个参数。如果没有指明，则产生512bit长的参数。
7. 指令req用来创建和处理PKCS#10格式的证书或者建立自签名证书,其使用格式如下。

openssl req [-inform PEM|DER] [-outform PEM|DER] [-in filename] [-passin arg] [-out filename] [-passout arg] [-text] [-noout] [-verify] [-modulus] [-new] [-rand file(s)] [-newkey rsa:bits] [-newkey dsa:file] [-nodes] [-key filename] [-keyform PEM|DER] [-keyout filename] [-[md5|sha1|md2|mdc2]] [-config filename] [-x509] [-days n] [-asn1-kludge] [-newhdr] [-extensions section] [-reqexts section]

1. -inform DER|PEM：指定输入的格式是DEM还是DER。 DER格式采用ASN1的DER标准格式。一般用的都是PEM格式，即为base64编码格式。PEM格式的第一行和最后一行指明内容，中间就是经过编码的内容。
2. -outform DER|PEM：与inform类似，不同的是指定输出格式。
3. -in filename ：要处理的CSR的文件名称，当-new和-newkey俩个option没有被set，本option才有效。
4. -out filename：要输出的文件名。
5. -text：将CSR文件里的内容以可读方式打印出来。
6. -noout：不打印CSR文件的编码版本信息。
7. -modulus：将CSR里面的包含的公共米要的系数打印出来。
8. -verify：检验请求文件里的签名信息。
9. -new：产生一个新的CSR， 要求用户输入创建CSR的一些必须的信息。需要的信息定义在config文件中。如果-key没有被set，那么就将根据config文件里的信息先产生一对新的RSA密钥。
10. -rand file(s)：产生key的时候用过seed的文件，可以把多个文件用冒号分开一起做seed。
11. -newkey arg：同时生成新的私有密钥文件和CSR文件，本option是带参数的。如果是产生RSA的私有密钥文件，参数是一个数字，指明私有密钥bit的长度，如果是产生DSA的私有密钥文件，参数是DSA密钥参数文件的文件名。
12. -key filename：参数filename指明私有密钥文件名，允许的格式是PKCS#8。
13. -keyform DER|PEM：指定输入的私有密钥文件的格式是DEM还是DER. DER格式采用ASN1的DER标准格式。一般用的多的都是PEM格式。
14. -days n：如果-x509被设定，那么这个选项的参数指定CA给第三方签证书的有效期，缺省是30天。
15. 指令x509是一个功能很丰富的证书处理工具。可以用来显示证书的内容，转换其格式，给CSR签名等等。其具体格式如下。

openssl x509 [-inform DER|PEM|NET] [-outform DER|PEM|NET] [-keyform DER|PEM][-CAform DER|PEM] [-CAkeyform DER|PEM] [-in filename][-out filename] [-serial] [-hash] [-subject] [-issuer] [-nameopt option] [-email] [-startdate] [-enddate] [-purpose] [-dates] [-modulus] [-fingerprint] [-alias] [-noout] [-trustout] [-clrtrust] [-clrreject] [-addtrust arg] [-addreject arg] [-setalias arg] [-days arg] [-signkey filename][-x509toreq] [-req] [-CA filename] [-CAkey filename] [-CAcreateserial] [-CAserial filename] [-text] [-C] [-md2|-md5|-sha1|-mdc2] [-clrext] [-extfile filename] [-extensions section]

1. -inform DER|PEM|NET：指定输入文件的格式。
2. -outform DER|PEM|NET：指定输出文件格式。
3. -in filename：指定输入文件名。
4. -out filename：指定输出文件名。
5. -md2|-md5|-sha1|-mdc2：指定使用的哈希算法，缺省的是MD5于打印有关的option
6. -text：用文本方式详细打印出该证书的所有细节
7. -noout：不打印出请求的编码版本信息
8. -modulus：打印出公共密钥的系数值
9. -serial：打印出证书的系列号。
10. -hash： 把证书的拥有者名称的哈希值给打印出来。
11. -subject：打印出证书拥有者的名字。
12. -issuer：打印证书颁发者名字。
13. -nameopt option：指定用什么格式打印输出
14. -email：如果有，打印出证书申请者的email地址
15. -startdate：打印证书的起始有效时间
16. -enddate：打印证书的到期时间
17. -dates：把上两个option都打印出来
18. -fingerprint：打印DER格式的证书的DER版本信息。
19. -C：用C代码风格打印结果。
20. -trustout：打印出可以信任的证书。
21. -setalias arg：设置证书别名。
22. -alias：打印证书别名。
23. -clrtrust：清除证书附加项里所有有关用途允许的内容。
24. -purpose：打印出证书附加项里所有有关用途允许和用途禁止的内容。

### 相关数据结构分析

BIO是Openssl库中重要的数据结构。无论即将建立的Openssl连接安全与否，Openssl都使用BIO抽象库(bio.h)来处理包括文件和套接字在内的各种类型的通信。一个BIO对象就是一个I/O接口的抽象，它隐藏了对于一个应用的许多底层I/O操作的细节工作。如果一个应用使用了BIO来进行I/O操作，它可以透明的处理SSL连接、加密连接和文件传输连接。

BIO分为两种类型，一种是Source/Sink类型，一种是Filter类型。前者代表数据源或数据目标，例如套接字BIO和文件BIO。后者的目的就是把数据从一个BIO转换到另外一个BIO或应用接口，在转换过程中，这些数据可以不经过修改就进行转换。例如在加密BIO中，如果进行写操作，数据就会被加密，如果是读操作，数据就会被解密。

#### BIO结构

typedef struct bio\_st BIO;

struct bio\_st

{

BIO\_METHOD \*method; //BIO方法结构，决定BIO类型和行为的重要参数

/\* bio, mode, argp, argi, argl, ret \*/

long (\*callback)(struct bio\_st \*,int,const char \*,int, long,long); //BIO回调函数

char \*cb\_arg; //回调函数都第一个参数

int init; //初始化标识，已初始化则为1

int shutdown; //开关标识，关闭为1

int flags; /\* extra storage \*/

int retry\_reason;

int num;

void \*ptr;

struct bio\_st \*next\_bio; //Filter型BIO所用，代表BIO链的下一节

struct bio\_st \*prev\_bio; //Filter型BIO所用，代表BIO链的上一节

int references;

unsigned long num\_read; //读出的数据长度

unsigned long num\_write; //写入都数据长度

CRYPTO\_EX\_DATA ex\_data;

};

#### 常用BIO相关函数

在BIO的所用成员中，method可以说是最关键的一个成员，它决定了BIO的类型，可以看到，在声明一个新的BIO结构时，总是使用下面的声明：

 BIO\* BIO\_new(BIO\_METHOD \*type);

从源代码可以看出，BIO\_new函数除了给一些初始变量赋值外，主要就是把type中的各个变量赋值给BIO结构中的method成员。

一般来说，上述type参数是以一个类型生成函数的形式提供的，如生成一个mem型的BIO结构，就使用下面的语句：

BIO \*mem = BIO\_new(BIO\_s\_mem());

两种BIO的相关常用函数如下。

1. source/sink型：
2. BIO\_s\_accept():是一个封装了类似TCP/IP socket Accept规则的接口，并且使TCP/IP操作对于BIO接口是透明的。
3. BIO\_s\_bio():封装了一个BIO对，数据从其中一个BIO写入，从另外一个BIO读出。
4. BIO\_s\_connect():是一个封装了类似TCP/IP socket Connect规则的接口，并且使TCP/IP操作对于BIO接口是透明的。
5. BIO\_s\_fd():是一个封装了文件描述符的BIO接口，提供类似文件读写操作的功能。
6. BIO\_s\_file():封装了标准的文件接口的BIO，包括标志的输入输出设备如stdin等。
7. BIO\_s\_mem():封装了内存操作的BIO接口，包括了对内存的读写操作。
8. BIO\_s\_null():返回空的sink型BIO接口，写入这种接口的所有数据读被丢弃，读的时候总是返回EOF。
9. BIO\_s\_socket():封装了socket接口的BIO类型。
10. filter型：
11. BIO\_f\_base64()：封装了base64编码方法的BIO,写的时候进行编码，读的时候解码
12. BIO\_f\_buffer()：封装了缓冲区操作的BIO，写入该接口的数据一般是准备传入下一个BIO接口的，从该接口读出的数据一般也是从另一个BIO传过来的。
13. BIO\_f\_cipher()：封装了加解密方法的BIO，写的时候加密，读的时候解密。
14. BIO\_f\_md()：封装了信息摘要方法BIO，通过该接口读写的数据都是已经经过摘要的。
15. BIO\_f\_null()：一个不作任何事情的BIO，对它的操作都简单传到下一个BIO去了，相当于不存在。
16. BIO\_f\_ssl()：封装了openssl 的SSL协议的BIO类型，也就是为SSL协议增加了一些BIO操作方法。

上述各种类型的函数正是构成BIO强大功能的基本单元，所有这些源文件，基本上都包含于/crypto/bio/目录下扩展名为.c的同名文件中。

#### 通过BIO结构进行IO操作

BIO的基本读写操作函数，包括四个，它们的定义如下（openssl/bio.h）：

  int BIO\_read(BIO \*b, void \*buf, int len);

int BIO\_gets(BIO \*b,char \*buf, int size);

int BIO\_write(BIO \*b, const void \*buf, int len);

int BIO\_puts(BIO \*b,const char \*buf);

1. BIO\_read：

从BIO接口中读出指定数量字节len的数据并存储到buf中。成功就返回真正读出的数据的长度，失败返回0或-1，如果该BIO没有实现本函数则返回-2。

1. BIO\_gets：

该函数从BIO中读取一行长度最大为size的数据。通常情况下，该函数会以最大长度限制读取一行数据，但是也有例外，比如digest型的BIO，该函数会计算并返回整个digest信息。此外，有些BIO可能不支持这个函数。成功就返回真正读出的数据的长度，失败返回0或-1，如果该BIO没有实现本函数则返回-2。需要注意的是，如果相应的BIO不支持这个函数，那么对该函数的调用可能导致BIO链中自动增加一个buffer型的BIO。

1. BIO\_write：

往BIO中写入长度为len的数据。成功就返回真正写入的数据的长度，失败返回0或-1，如果该BIO没有实现本函数则返回-2。

1. BIO\_puts：

往BIO中写入一个以NULL为结束符的字符串，成功就返回真正写入的数据的长度，失败返回0或-1，如果该BIO没有实现本函数则返回-2。

另外，除了这四个基本的IO操作函数以外，还有一个比较重要的IO函数。该函数也是BIO\_ctrl的宏定义函数，其定义如下。

#define BIO\_flush(b) (int)BIO\_ctrl(b,BIO\_CTRL\_FLUSH,0,NULL)

该函数用来将BIO内部缓冲区的数据一次性的都写出去，有些时候，也用于根据EOF查看是否还有数据可以写。调用成功的时候该函数返回1，失败的时候返回0或-1。之所以失败的时候返回0或者-1，是为了标志该操作是否需要稍后以与BIO\_write()相同的方式重试，这时候，应该调用BIO\_should\_retry()函数。

需要注意的是，返回值为0或-1的时候并不一定就是发生了错误。在非阻塞型的source/sink型或其它一些特定类型的BIO中，这仅仅代表目前没有数据可以读取，需要稍后再进行该操作。

阻塞类型的socket使用如select, poll, equivalent等函数检测BIO中是否存在需要被read()函数读取的有效数据，但不建议在阻塞型的接口中使用这些技术，因为这种情况下如果调用BIO\_read()会导致在底层的IO中多次调用read()函数，从而导致端口阻塞。这种情况下，select（或equivalent）应该和非阻塞型的IO一起使用，可以在失败之后重新读取该IO，而不使端口阻塞。

## 实例编程练习

### 编程练习要求

在Linux下利用OpenSSL实现安全的Web Server的具体要求如下。

①在理解HTTPS及SSL的工作原理后，实现安全的Web Server。

②Server能够并发处理多个请求，要求至少能支持Get命令。可以增强Web Server的功能，如支持Head、Post以及Delete命令等。

③书写必要的客户端测试程序用于发送HTTPS请求并显示返回结果，也可以使用一般的Web浏览器测试。

下面介绍示例程序的相关内容，仅供参考。

本程序中，服务器端为命令行程序，客户端为Linux下普通的Web浏览器，这里使用的是FireFox浏览器。服务器端的IP地址为192.168.1.91，开放的端口为22222。

程序执行流程如下。

服务器端：./Server

[root@localhost share]# ./Server

\*\*\*\*\*\*\*\*\*\*\*\*\* Server Starting \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

此时Web服务器已经开启，等待客户端的连接请求。

在客户端，即浏览器中，键入服务器的IP地址和端口，<https://192.168.1.91:22222>，请求返回服务器端默认的主页。

服务器端显示从客户端发送请求到成功将所请求的文件发送回客户端之间所发生事件的日志记录。

[root@localhost share]# ./Server

\*\*\*\*\*\*\*\*\*\*\*\*\* Server Starting \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

IP:192.168.1.100 connecting to socket:872

/index.html

Closing socket:872

IP:192.168.1.100 connecting to socket:776

/index.files/winpcap.css

Closing socket:776

IP:192.168.1.100 connecting to socket:934

IP:192.168.1.100 connecting to socket:796

IP:192.168.1.100 connecting to socket:752

IP:192.168.1.100 connecting to socket:734

IP:192.168.1.100 connecting to socket:712

/index.files/New.gif

Closing socket:934

/index.files/curve.gif

Closing socket:796

/index.files/airpcap.gif

Closing socket:752

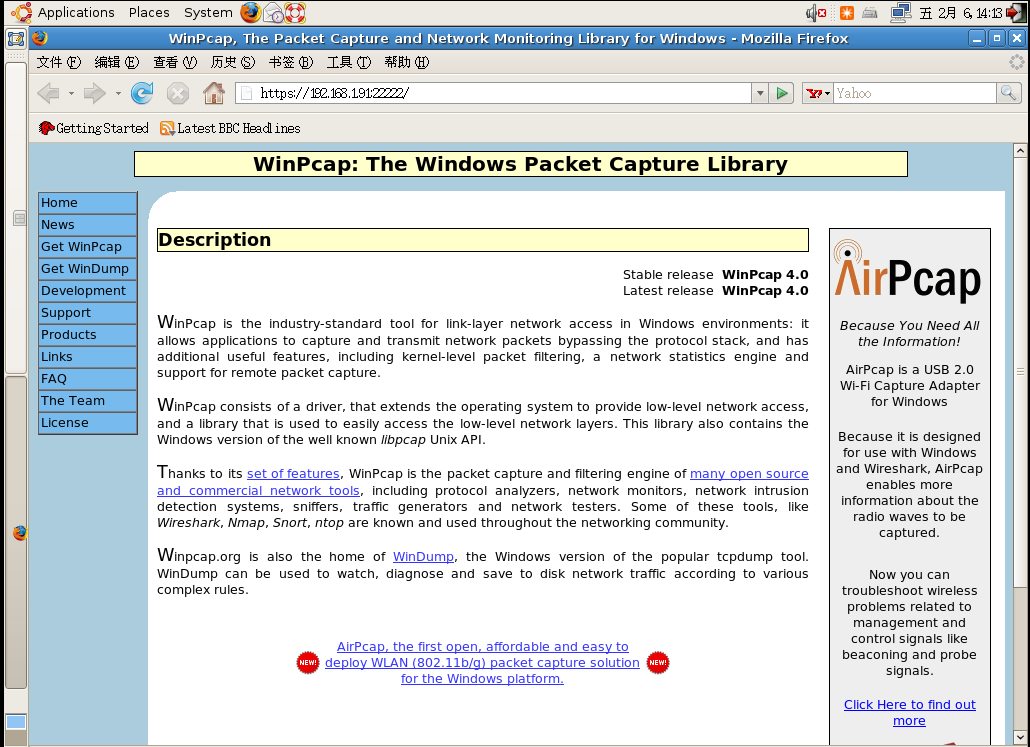
/index.files/cace\_logo.gif

Closing socket:734

/index.files/curvedown.gif

Closing socket:712

此时，客户端已成功收到并显示所请求的网页内容。如图6-4所示。

图6-4 客户端截图

至此，服务器端成功完成一次基于HTTPS的Web响应。

### 编程训练设计与分析

程序流程如图6-5所示。



图6-5 程序整体流程

程序可以分为两部分：初始化模块，即通过编译及函数调用对OPENSSL库进行初始化并且创建上下文环境；WEB服务模块，即基于SSL机制利用OPENSSL库函数实现HTTPS的服务。

#### 初始化模块

在开启HTTPS服务之前，服务器端只需要初始化OPENSSL库和创建上下文环境。在此过程中，会用到如下函数：

SSL\_library\_init(); // 加载OpenSSL将会用到的算法

SSL\_load\_error\_strings(); // 加载错误字符串

SSL\_METHOD \*meth;

SSL\_CTX \*ctx; // SSL\_CTX对象

meth = SSLv23\_method(); // 相应的SSL结构能理解SL2.0、3.0以及TSL1.0

ctx = SSL\_CTX\_new(meth); // 创建一个上下文环境

SSL\_CTX\_use\_certificate\_chain\_file(ctx, “server.pem”); // 指定所使用的证书文件

SSL\_CTX\_set\_default\_passwd\_cb(ctx, password\_cb); // 设置密码回调函数

SSL\_CTX\_use\_PrivateKey\_file(ctx, “server.pem”, SSL\_FILETYPE\_PEM)); // 加载私钥文件

SSL\_CTX\_load\_verify\_locations(ctx, "root.pem", 0); // 加载受信任的CA证书

load\_dh\_params(ctx, "dh1024.pem");

/\*当使用RSA算法鉴别的时候，会有一个临时的DH密钥磋商发生。这样会话数据将用

这个临时的密钥加密，而证书中的密钥作为签名\*/

#### WEB服务模块

WEB服务模块是本程序的核心部分，其执行流程如图6-6所示。

图6-6 HTTPS实现流程

WEB服务模块在类CHttpProtocol中实现。该类中封装了HTTPS中的与本程序相关的操作，主要包括监听线程函数以及客户端线程函数中需要调用的子函数的定义及实现。其中，比较核心的子函数包括SSL分析请求，将响应头部返回给客户端，将文件发送回客户端等等。

下面给出该类定义中的部分代码：

class CHttpProtocol

{

……

CHttpProtocol(void);

~CHttpProtocol(void);

SSL\_CTX \*ctx; //SSL上下文

char \* initialize\_ctx(); //初始化CTX

char \* load\_dh\_params(SSL\_CTX \*ctx, char \*file); //加载CTX参数

int TcpListen(); //TCP监听函数

void StopHttpSrv(); //停止HTTP服务

bool StartHttpSrv(); //开始HTTP服务

static void \* ListenThread(LPVOID param); //监听线程

static void \* ClientThread(LPVOID param); //客户线程

//接收HTTPS请求

bool SSLRecvRequest(SSL \*ssl,BIO \*io, LPBYTE pBuf, DWORD dwBufSize);

int Analyze(PREQUEST pReq, LPBYTE pBuf); //分析HTTP请求

bool SSLSendHeader(PREQUEST pReq, BIO \*io); //发送HTTPS头

bool SSLSendFile(PREQUEST pReq, BIO \*io); //由SSL通道发送文件

bool SSLSendBuffer(PREQUEST pReq, LPBYTE pBuf, DWORD dwBufSize);

……

};

下面对本示例程序中的模块依次进行介绍。

1. 监听线程：用于监听WEB服务连接请求。

使用pthread\_create()创建监听线程，代码如下。

pthread\_t listen\_tid;

pthread\_create(&listen\_tid,NULL,&ListenThread,this);

其中传递给监听线程函数的参数是调用该成员函数的类对象指针，监听函数可以由这个类对象指针调用其需要的信息，如套接字句柄、SSL上下文等等。

在监听线程函数中，服务器端循环等待客户端的连接请求，若有新的请求到达，服务器端将新创建一个客户端线程去处理该客户端的请求。

void \* CHttpProtocol::ListenThread(LPVOID param)

{……

CHttpProtocol \*pHttpProtocol = (CHttpProtocol \*)param;

while(1) // 循环等待,如有客户连接请求,则接受客户机连接要求

{

nLen = sizeof(SockAddr);

// 套接字等待链接,返回对应已接受的客户机连接的套接字

socketClient = accept(pHttpProtocol->m\_listenSocket, (LPSOCKADDR)&SockAddr, &nLen);

if (socketClient == INVALID\_SOCKET)

break;

// 创建client进程，处理request

pthread\_create(&client\_tid,NULL,&ClientThread,pReq);

} //while

……

}

1. 客户端线程：用于处理接收到的客户端请求。

客户端线程由监听线程在accept()调用成功后创建。在客户端线程函数中，首先将根据之前初始化的上下文创建出来的SSL对象绑定在一个socket类型的BIO上面，然后调用SSL\_accept()函数，与客户端进行握手。

下面的代码摘自客户端线程函数：

void \* CHttpProtocol::ClientThread(LPVOID param)

{

……

PREQUEST pReq = (PREQUEST)param;

CHttpProtocol \*pHttpProtocol = (CHttpProtocol \*)pReq->pHttpProtocol;

SOCKET s = pReq->Socket;

sbio = BIO\_new\_socket(s, BIO\_NOCLOSE); // 创建一个socket类型的BIO对象

ssl=SSL\_new(pReq->ssl\_ctx); // 创建一个SSL对象

SSL\_set\_bio(ssl, sbio, sbio); // 把SSL对象绑定到socket类型的BIO上

//连接客户端，在SSL\_accept过程中，将会占用很大的cpu

nRet = SSL\_accept(ssl);

//nRet<=0时发生错误

io = BIO\_new(BIO\_f\_buffer()); //封装了缓冲区操作的BIO

ssl\_bio = BIO\_new(BIO\_f\_ssl());

//封装了SSL协议的BIO类型，也就是为SSL协议增加了一些BIO操作方法

BIO\_set\_ssl(ssl\_bio, ssl, BIO\_CLOSE);

// 把ssl(SSL对象)封装在ssl\_bio(SSL\_BIO对象)中

BIO\_push(io, ssl\_bio);

// 把ssl\_bio封装在一个缓冲的BIO对象中，实现对SSL连接的缓冲读和写

……

//做好上述IO绑定之后，开始接受客户端请求

if (!pHttpProtocol->SSLRecvRequest(ssl,io,buf,sizeof(buf)))

{

// 处理错误

……

}

//HTTPS协议分析

nRet = pHttpProtocol->Analyze(pReq, buf);

if (nRet)

{

// 处理错误

……

}

// 生成并返回头部

if(!pHttpProtocol->SSLSendHeader(pReq,io))

{

// 处理错误

……

}

BIO\_flush(io);

// 向client传送数据

if(pReq->nMethod == METHOD\_GET)

{

if(!pHttpProtocol->SSLSendFile(pReq,io))

{

// 处理错误

……

}

}

//析构操作

……

}

客户端线程的参数定义如下。

typedef struct REQUEST

{

SOCKET Socket; // 请求的socket

int nMethod; // 请求的使用方法：GET或HEAD

DWORD dwRecv; // 收到的字节数

DWORD dwSend; // 发送的字节数

int hFile; // 请求连接的文件

char szFileName[256]; // 文件的相对路径

char postfix[10]; // 存储扩展名

SSL\_CTX \* ssl\_ctx; //SSL上下文

void\* pHttpProtocol; // 指向类CHttpProtocol的指针

……

}REQUEST, \*PREQUEST;

1. 接受客户端请求：接受客户端请求的工作由函数SSLRecvRequest()完成。摘录代码如下。

bool CHttpProtocol::SSLRecvRequest(SSL \*ssl,BIO \*io, LPBYTE pBuf, DWORD dwBufSize)

{ ……

memset(buf, 0, BUFSIZZ); //初始化缓冲区

while(1)

{

r = BIO\_gets(io, buf, BUFSIZZ-1);

switch(SSL\_get\_error(ssl, r))

{

case SSL\_ERROR\_NONE:

memcpy(&pBuf[length], buf, r);

length += r;

break;

default:

break;

}

// 直到读到代表HTTP头部结束的空行

if(!strcmp(buf,"\r\n") || !strcmp(buf,"\n"))

break;

}

// 添加结束符

pBuf[length] = '\0';

return true;

}

该函数首先利用BIO\_gets()函数从io中读取数据，放到事先分配好的缓冲区buf里，并通过对换行符的判断确保获得整个完整的HTTPS头部，以便对缓冲区中的HTTPS请求数据报进行分析。

1. HTTPS协议解析：

下面代码片段摘自SSL请求分析函数，由客户端线程函数调用：

int CHttpProtocol::Analyze(PREQUEST pReq, LPBYTE pBuf)

{

// 分析接收到的信息

char szSeps[] = " \n";

char \*cpToken;

// 判断request的method

cpToken = strtok((char \*)pBuf, szSeps); // 缓存中字符串分解为一组标记串。

if (!strcmp(cpToken, "GET")) // GET命令

{

pReq->nMethod = METHOD\_GET;

}

……

strcpy(pReq->szFileName, m\_strRootDir);

if (strlen(cpToken) > 1)

{

strcat(pReq->szFileName, cpToken); // 把该文件名添加到结尾处形成路径

}

else

{

strcat(pReq->szFileName, "/index.html"); //若无文件名，则默认为index.html

}

……

}

该函数根据HTTPS协议的标准请求格式的头部来进行分解与分析，获取请求的命令，请求的文件等信息。服务器根据这些信息，将相应的响应及文件本身发送回客户端。

1. 发送HTTPS响应：

服务器端会首先根据客户端请求的命令和文件，来判断命令是否符合标准，所请求的文件是否存在，然后组成一个响应发回客户端。人们通常在上网的时候，网页上出现404或400等错误，就是因为这一步请求的文件于服务器上不存在或者命令错误，服务器就会发送一个错误的响应到用户的浏览器上。

下面的代码摘自服务器端的响应函数，该函数由客户端线程函数调用。

bool CHttpProtocol::SSLSendHeader(PREQUEST pReq, BIO \*io)

{

……

char curTime[50];

GetCurrentTime(curTime);

// 取得文件的last-modified时间

char last\_modified[60] = " ";

GetLastModified(pReq->hFile, (char\*)last\_modified);

// 取得文件的类型

char ContentType[50] = " ";

GetContentType(pReq, (char\*)ContentType);

//组成完整的服务器响应

sprintf((char\*)Header, "HTTP/1.1 %s\r\nDate: %s\r\nServer: %s\r\nContent-Type: %s\r\nContent-Length: %d\r\n\r\n",

HTTP\_STATUS\_OK,

curTime, // Date

"Villa Server 192.168.1.91", // Server"My Https Server"

ContentType, // Content-Type

length); // Content-length

……

BIO\_flush(io); //一次性清空缓冲区，全部写入io

return true;

}

1. 释放相关资源

在客户端线程完成请求之后，需要释放相关资源：

SSL\_shutdown(ssl); //关闭SSL连接

SSL\_free(ssl); //释放SSL结构

SSL\_CTX\_free(ssl\_ctx); //释放上下文环境

## 扩展与提高

### 客户端认证

建立SSL连接时，一般要求服务器提供认证证书，由客户端验证通过才可继续建立连接，客户端一般并不被要求提供客户端的认证证书，本Web服务器没有要求客户端提供认证证书。作为扩展，可以为服务器添加客户端认证功能，即进行客户端和服务器双向认证。

采用双向认证方式的SSL连接，既要求服务器提供认证证书，由客户端验证，也要求客户端也提供自己的认证证书，由服务器进行验证。

OpenSSL提供了建立双向认证SSL连接的API，只需要在单向认证的基础上，设置要求对方提供证书的属性，并设置系统可信的证书，OpenSSL提供的相关API如下。

1. 设置可信CA证书文件：

int SSL\_CTX\_load\_verify\_locations(SSL\_CTX \*ctx, const char \* cafile, const char \* cadir);

1. 设置认证模式：

int SSL\_CTX\_set\_verify(SSL\_CTX \*ctx, int mode,

int (\*verify\_callback)(int, X509\_STORE\_CTX \*));

其中mode参数表示认证模式，verify\_callback提供注册认证回调函数的机制，为NULL时使用OpenSSL内置的认证函数。使用双向认证模式需要将mode的值置为： SSL\_VERIFY\_PEER | SSL\_VERIFY\_FAIL\_IF\_NO\_PEER\_CERT。

1. 设置可信证书链深度：

void SSL\_CTX\_set\_verify\_depth(SSL\_CTX \*ctx, int depth);

如果需要验证客户端，则在服务器添加如下代码即可。

SSL\_CTX\_load\_verify\_locations(ctx, RSA\_SERVER\_CA\_CERT/\*客户证书的根CA\*/, NULL);

SSL\_CTX\_set\_verify(ctx, SSL\_VERIFY\_PEER | SSL\_VERIFY\_FAIL\_IF\_NO\_PEER\_CERT, NULL);

SSL\_CTX\_set\_verify\_depth(ctx, 1);

### 基于IPsec的安全通信

利用SSL可以保证Web浏览器和Web服务器间的安全通信，利用PGP（Pretty Good Privacy）及S/MIME（Secure/Multipurpose Internet Mail Extension）可以实现邮件加密，但是这些安全技术都只能用于局部业务，并不能保证TCP/IP整体上的安全通信，因此出现了能够使企业和个人用户在开放的Internet上通用的安全协议—IPSec。

IPSec协议是IETF于1998年11月公布的IP安全标准，目前IPSec被广泛应用于实现端到端的安全、虚拟专用网和安全隧道，它对IPv4是可选的，对于IPv6则是强制必须实施的，是唯一一种可为任何形式的Internet通信提供安全保障的协议，也是易于扩展的、完整的一种基础网络安全方案。IPSec协议是一个协议族，它包含验证头协议（Authentication Header，AH）、封装安全载荷协议（Encapsulation Security Payload，ESP）和Internet密钥交换协议（Internet Key Exchange，IKE）等。

#### IPSec体系结构

IPSec提供了一种标准、健壮且包容广泛的安全机制，可以为IP及其上层协议提供安全保证。其具体保护形式有：数据源验证、无连接数据的完整性验证、数据机密性、抗重播及有限的数据流机密性保证。IPSec是一个协议族，其体系结构如图6-7所示。

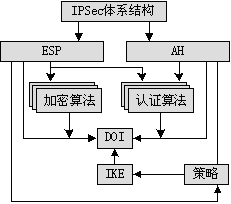


图6-7 IPSec安全体系结构

ESP和AH：ESP是插入IP数据包内的一个协议头，可以为IP提供机密性、数据源验证、抗重播以及数据完整性等安全服务。AH与ESP类似，但它不提供机密性服务。

策略：目前尚未形成标准，它决定两个实体之间是否能够通信以及如何通信，IPSec策略由安全策略数据库（Security Policy Database，SPD）维护。一个SPD条目可能定义了下述几种行为之一：丢弃、绕过、应用。对那些定义了应用行为的SPD条目，它们均会指向一个或一串安全关联（Security Association，SA）。SA定义了对一个特定的IP包作IPSec处理对应的各种安全参数，包括要应用的安全协议（ESP、AH）、算法及密钥、密钥生存期、抗重播窗口等。

IKE：SA可以手工或动态建立，IKE用于动态建立SA，它可以代表IPSec对SA进行协商。

1. ESP：

ESP是属于IPSec的一种协议，协议号为50，用于保护IP数据包的机密性、数据的完整性以及数据源的身份认证，也负责抵抗重播攻击。ESP用一个加密器来提供机密性，一个身份验证器来保证完整性。对于外出的数据包，首先进行的是加密处理，然后是验证处理；对于进入的数据包，操作相反。应用ESP时，在IP报头之后，要保护的数据之前插入一个ESP头，之后插入一个ESP尾。ESP保护的数据包的封装格式如图6-8所示。

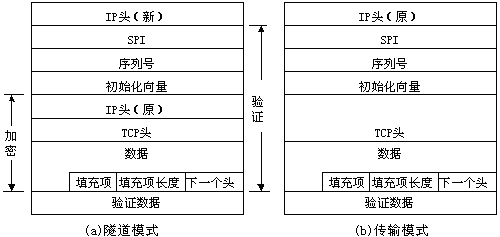


图6-8 ESP报文格式

* 1. 安全参数索引(Security Parameter Index，SPI)：用来和对端的安全设备同步使用加密算法和认证算法。通常接收端使用元组<SPI，目标主机，协议>唯一标识所使用的SA。
  2. 序列号：单向递增的计数器，用来抗重播攻击。
  3. 初始化向量（Initialization Vector，IV）：这是一个可选的32位字段，会出现三种不同情况：加密算法不需要IV、需要隐式的IV和需要显式的IV。
  4. 填充项：0～255字节，用于保证边界正确性。某些加密算法模式对加密数据长度有要求；即使没有要求机密性保证，仍需要把“下一个头”字段靠右排列；同时，这项技术隐藏了原始数据的实际长度。
  5. 填充项长度：给出前面的填充项的长度，置0时表示没有填充。
  6. 下一个头：表明数据类型。在隧道模式下该值为４，表示IP-in-IP；传输模式下，该值表示载荷所属协议的类型，例如TCP协议对应的值为6。
  7. 验证数据：用于容纳数据完整性的验证结果，即完整性验证值（Integrity Check Value，ICV）。

ESP的工作机制如下。发送方先对数据包进行加密，再用信息摘要算法计算验证数据，并作为该IP分组的一部分一起转发出去。在分组的接收端，重新计算该摘要并与原摘要进行比较，验证成功后，进行解密。

1. AH：

AH也是一种IPSec协议，协议号为51，用于为IP提供数据完整性，数据原始身份验证和一些可选的、有限的抗重播服务。AH不提供任何保密性服务，它不加密所保护的数据包，因此AH头比ESP头简单很多。由于不需要填充，不需要填充长度指示器，因此也不存在尾，另外，也不需要初始化向量。图6-9描述了隧道模式下受AH保护的IP报文格式。



图6-9 隧道模式下的AH报文格式

1. 载荷长度：表示以32比特为单位的验证头的长度减去2，并不单指IP包的实际负载长度。
2. 下一个头、SPI和序列号：这几个字段与ESP中的意义相同。
3. 验证数据：用于容纳数据完整性的验证结果（ICV）。计算出ICV之前，该字段置为0。和ESP不同，AH将完全保护扩展到外部IP头的恒有或预计有的字段，因此，在做ICV计算时，可变字段必须全部置为0。可变字段包括IP头中服务类型、分段偏移、TTL、头校验和等。

AH的工作机制如下。利用单向的信息摘要算法，对整个IP分组或上层协议计算一个摘要并作为该IP分组的一部分一起转发出去，在分组的接收端，重新计算该摘要并与原摘要进行比较，如果分组在传输过程中被修改过，则会由于摘要不一致而被丢弃，从而实现数据源鉴别和数据的完整性保护。

1. 安全策略和安全关联：
2. 安全策略（Security Policy，SP）是IPSec中尚未成为标准的一个重要组件，它决定了为一个数据包提供的安全服务，存放在安全策略数据库SPD中。IP包的外出和进入处理都要以安全策略为准，并且，要求策略管理应用能够添加、删除和修改策略，但SPD的具体管理方式由实现方案决定，并未为此专门定义一套统一的标准。

每个SPD中的条目由选择符和策略项组成，根据选择符（selector）对SPD进行检索，得到相应的策略。选择符是从网络层和传输层头内提取出来的，包括源地址、目的地址、传输协议、上层端口等。策略项一般是三种不同的行为之一：丢弃、绕过和应用IPSec。当采取应用IPSec这一行为时，策略会提供一个三元组或四元组，可以称作SAID：<目的地址（外部地址）、安全协议（ESP或AH）、SPI>或<源地址、目的地址、安全协议、SPI >。这个SAID作为检索SA以获得详细安全参数的依据。

1. SA是构成IPSec的基础。SA是两个通信实体经协商建立起来的一种协定。它们决定了用来保护数据包安全的IPSec协议（ESP或AH）、转码方式、密钥以及密钥的有效存在时间等等。任何IPSec实施方案始终会构建一个安全关联数据库（Security Association Database，SAD），由它来维护IPSec协议用来保障数据包安全的SA记录。

SA是单向的。如果两个主机（比如A和B）正在通过ESP进行安全通信，那么主机A 就需要有一个SA，即SA (out)，用来处理往外发的数据包；另外还需要有一个相对应的SA，即SA (in)，用来处理进入的数据包。主机A的SA (out)和主机B的SA ( in )将共享相同的安全参数（比如密钥）。类似地，主机A的SA (in)和主机B的SA (out )也会共享同样的安全参数。

每个SAD中的条目也可以看成由两部分组成：SAID和SA记录。SA记录中主要包括序号计数器、算法、密钥、SA的TTL、IPSec模式等。

1. 在包的处理过程中，SPD和SAD这两个数据库需要联合使用。外出处理时先检索SPD，进入处理时先检索SAD。

对一个外出包而言，根据提取的选择符检索SPD，它会命中某个SP。根据策略，包可能被丢弃、直接传送或应用IPSec安全服务；当需要IPSec安全服务时，SP返回一个SAID，用SAID检索SAD，最终命中SA。根据SA指明的各项安全参数对数据包进行安全封装。

进入处理有别于外出处理，以下两种情况分别对待：①如果收到的数据包内没有包含IPSec头，说明是个普通IP包：检索SPD，根据检索结果决定将包丢弃，或者传递给传输层做进一步的处理。②IP包中含有IPSec头：提取IPSec头部信息，组成SAID，检索SAD，根据SA中指明的各项安全参数对数据包进行验证、解封装。接着，根据从解封了的原始IP包中提取的选择符检索SPD，验证SA的使用是否得当。

1. IKE

IPSec默认的自动密钥管理协议是IKE。IKE是Oakley和SKEME协议的一种混合，并在由ISAKMP规定的框架内运作。

IKE用于动态建立SA。IKE代表IPSec对SA进行协商，最终确定可以称为“保护套件”的安全参数——包括加密算法、散列算法、验证方法和Diffie-Hellman组。之后对安全关联库SAD进行填充。IKE是一个用户级进程，启动后作为后台守护进程运行，在需要使用IKE服务前它一直处于不活动状态。

可以通过两种方式来请求IKE服务：

1. 内核的安全策略模块要求自动建立SA时；
2. 远程IKE实体需要协商SA时。

IKE使用了两个阶段的ISAKMP。在第一阶段，通信各方彼此间建立一个已经通过身份验证和安全保护的通道，即建立IKE安全联盟。在第二阶段，利用这个既定的安全联盟，为IPSec协商具体的安全联盟。

IKE共定义了5种交换：2个阶段1次交换，1个阶段2次交换，2个额外的交换。阶段一的两种交换：对身份进行保护的“主模式”交换，根据基本ISAKMP文档制订的“野蛮模式”交换。阶段二使用“快速模式”交换。IKE自己定义了两种交换：①为通信各方间协商一个新的Diffie-Hellman组类型的“新组模式”交换；②在IKE通信双方间传送错误及状态消息的ISAKMP信息交换。

IKE保证了动态建立安全联盟和建立过程的安全。IKE的实现是IPSec协议实现的重要组成部分，实现极为复杂；但它也很可能成为整个系统的瓶颈。优化IKE程序、优化密钥算法是实现IPSec的核心问题之一。

#### IPSec工作模式

IPSec协议既可以用来保护IP上层协议，也可以用来保护一个完整的IP数据包。这两方面的保护分别由IPSec的传输模式和隧道模式来提供。数据包格式如图6-10所示。



图6-10 两种IPSec模式的数据包结构

只有在要求端到端的安全保障时，才能使用IPSec的传输模式。路由器主要通过检查网络层来做出路由决定，而且路由器不会、也不应该改变网络层头之外的其他东西。如果通过路由器为数据包流插入传输模式的IPSec头，便违反了这一规则。

在隧道模式中，把要保护的整个IP包封装到另一个IP数据包里，增加了一个新IP头。通信终点由内部IP头指定，而加密终点，也就是隧道终点，则由外部新IP头指定。