红超的吾记之谈的博客
http://blog.sina.com.cn/wzhnsc [订阅] [手机订阅]

个人资料

红超的吾记之谈

微博

加好友 发纸条
写留言 加关注

博客地图 world map

博客等级:19
博客积分:1544
博客访问:891,413
关注人气:176
获赠金笔:25
赠出金笔:0
荣誉徽章:

相关博文

- 盘点娱乐圈最反感直播带货的八大I 代军哥哥
- cctv5直播辽篮VS北控，郭艾伦领衔; 晚池
- 赵露思生日写真宛如清新浪漫的小· 苗禹哥
- CBA第11轮本土最佳阵容，高效郭3 晚池
- 王霏霏外滩源街拍演绎摩登酷飒时| 苗禹哥
- CBA青岛男篮vs苏州男篮名单出炉, 晚池
- 赵本山徒弟胖丫与许晴比美，经历3 代军哥哥
- cctv5直播国足PK阿曼男足等3场世:

正文 字体大小:大 中 小

使用 OpenSSL API 建立安全连接 - 双向认证

(2012-12-04 15:36:02) 转载

标签: 杂谈 分类: 开源项目之OpenSSL

使用 OpenSSL API 进行安全编程

一、概念:

1.什么是 SSL?

SSL 是一个缩写，全称是 Secure Sockets Layer。
它是支持在 Internet 上进行安全通信的标准，并且将数据密码术集成到了协议之中。
数据在离开您的计算机之前就已经被加密，然后只有到达它预定的目标后才被解密。
证书和密码学算法支持了这一切的运转，使用 OpenSSL，您将有机会切身体会它们。

可以将 SSL 和安全连接用于 Internet 上任何类型的协议，
不管是 HTTP、POP3，还是 FTP。
还可以用 SSL 来保护 Telnet 会话。
虽然可以用 SSL 保护任何连接，但是不必对每一类连接都使用 SSL。
如果连接传输敏感信息，则应使用 SSL。

2.什么是 OpenSSL?

OpenSSL 不仅仅是 SSL。
它可以实现消息摘要、文件的加密和解密、数字证书、数字签名和随机数字。
关于 OpenSSL 库的内容非常多，远不是一篇文章可以容纳的。

OpenSSL 不只是 API，它还是一个命令行工具。
命令行工具可以完成与 API 同样的工作，而且更进一步，可以测试 SSL 服务器和客户机。
它还让开发人员对 OpenSSL 的能力有一个认识。

首先需要的是最新版本的 OpenSSL 。
不过，为了安全起见，我建议您下载最新的源代码并自己编译它。

二进制版本通常是由第三方而不是由 OpenSSL 的开发人员来编译和发行的。
一些 Linux 的发行版本附带了 OpenSSL 的二进制版本，
对于学习如何使用 OpenSSL 库来说，这足够了；

如果想使用 OpenSSL 来生成证书请求和数字证书，那么必须创建一个配置文件。
在 OpenSSL 程序包的 apps 文件夹中，有一个名为 openssl.cnf 的可用模板文件。
不过，该模板文件有一些非常好的注释，
而且如果在 Internet 上搜索，您可以找到很多讨论修改该文件的教程。

3.什么是数字证书?

数字证书的最简单形式就是 不对称加密密钥(asymmetric cryptography key)。
目前关于数字证书的标准中都有一些标识信息，在密钥中也都包含了这些信息。
一个典型的数字证书包含:
所有者的名字(如果这个证书是在一个 Web 服务器上使用的，那么名字就是完整的域名)以及联系信息，
还有一个有效日期范围，
以及一个安全性签名，用来验证这个证书没有被篡改。

数字证书可以使用 OpenSSL 命令行工具或其他用于此目的的工具简单地创建。
但是任何人创建的数字证书都有一个信任的问题。
数字证书不仅仅是一个加密密钥，它还是一个在线凭证。
证书会向那些试图与您进行通信的人证明您的身份。
为了显示信任关系，数字证书可以由认证权威(CA - Certificate Authority)机构进行签名。

认证权威在数字安全性领域充当一个可信的第三方。
由于在在线领域中证明某个实体的身份非常困难，认证权威就接管了这个挑战。
它们为那些购买证书或对证书进行签名的用户的身份提供证明。
因此，要信任一个证书，用户只需要信任证书权威即可。
用户通过拥有并使用 CA 的信任证书来表明自己对认证权威的信任。
Verisign 和 Thawte 是非常知名的认证权威。

如果一个证书的安全性曾经受到过威胁，那么这个证书就会被丢弃 -- 也就是说，将其声明为无效。
当一个证书被声明为无效时，CA 不可能将其通知所有拥有该证书拷贝的人。
相反，CA 会发布一个 证书撤销列表(CRL - Certificate Revocation List)。
浏览器和其他使用数字证书的程序都可以验证这个证书已经被其属主或 CA 撤销了。

证书的撤销也可以使用 OCSP 协议进行检查。
OCSP 代表 Online Certificate Status Protocol(在线证书状态协议)，
它是在 RFC 2560 中定义的。
OpenSSL 既有 OCSP 的功能，又有 CRL 的功能，但是对这些功能的介绍已经超出了本文的范围。
目前数字证书所采用的标准是 X.509，这是在 RFC 3280 中定义的。

OpenSSL 有一个专门用于数字证书的库。
这个库的源代码就位于 crypto/x509 和 crypto/x509v3 目录中。
源代码为数字证书的处理定义了几个与 X.509 证书有关的 OpenSSL 结构：
结构　　功能
X509　　包含所有有关数字证书的数据。
X509_ALGOR　　提供该证书设计所针对的算法。
X509_VAL　　该证书有效的时间跨度。
X509_PUBKEY　　证书的公钥算法，通常是 RSA 或 DSA 。
X509_SIG　　证书的 hash 签名。
X509_NAME_ENTRY 证书所包含的数据的各个项。
X509_NAME　　包含名字项的堆栈。
这些只是其中涉及的几个结构。
在 OpenSSL 中使用的大部分 X.509 结构您自己在应用程序中几乎都不会用到。
在这些结构之上，有一些用来处理数字证书的函数。
这些函数得名于它们所适用的结构。
例如：一个名字以 X509_NAME 开始的函数，通常会应用于一个 X509_NAME 结构。

4.什么是开展业务之前的握手？
安全连接要求在连接建立后进行握手。
在握手过程中，服务器向客户机发送一个证书，然后，客户机根据一组可信任证书来核实该证书。
它还将检查证书，以确保它没有过期。
要检验证书是可信任的，需要在连接建立之前提前加载一个可信任证书库。

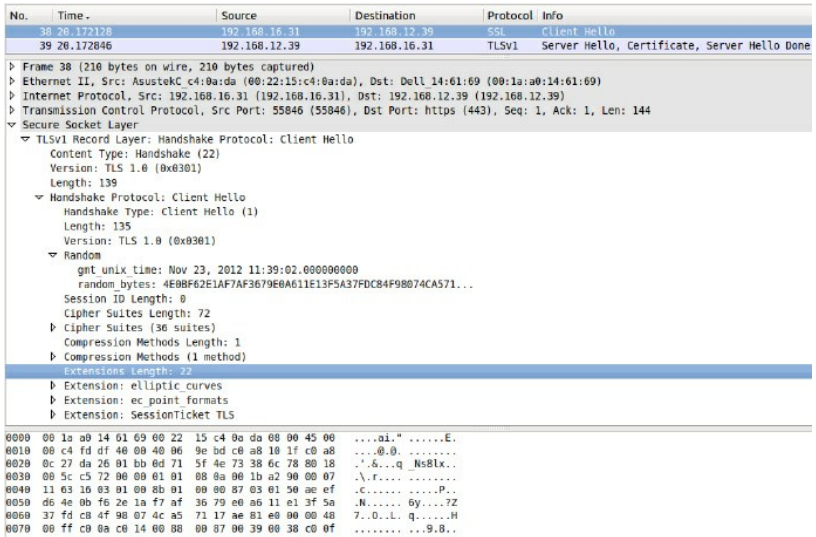
只有在服务器发出请求时，客户机才会向服务器发送一个证书。
该过程叫做客户机认证。
使用证书，在客户机和服务器之间传递密码参数，以建立安全连接。
尽管握手是在建立连接之后才进行的，但是客户机或服务器可以在任何时刻请求进行一次新的握手。

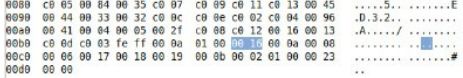
握手过程：
安全且可信的握手意味着事务的双方都相信它们正在做的事情对双方都是有益的。
不安全的握手标记着只有一方会对事务有着正确的理解。
1.在一个连接上开始握手通常是从客户端向服务器说“Hello”开始的。
　helllo 消息(规范就是如此说的)包含了：
　客户端 SSL 协议的版本号，
　加密算法的种类，
　产生的随机数，
　以及其他服务器和客户端之间通讯所需要的各种信息。
2.服务端会使用自己的 hello 消息进行响应，其中包含了：
　SSL 协议的版本号，

加密算法的种类，
随机数，
以及其他相关信息，
同时服务器还将向客户端传送自己的证书。
注：如果客户端还要为这个连接进行认证，那么服务器还会发送一个请求，索取客户端的证书。

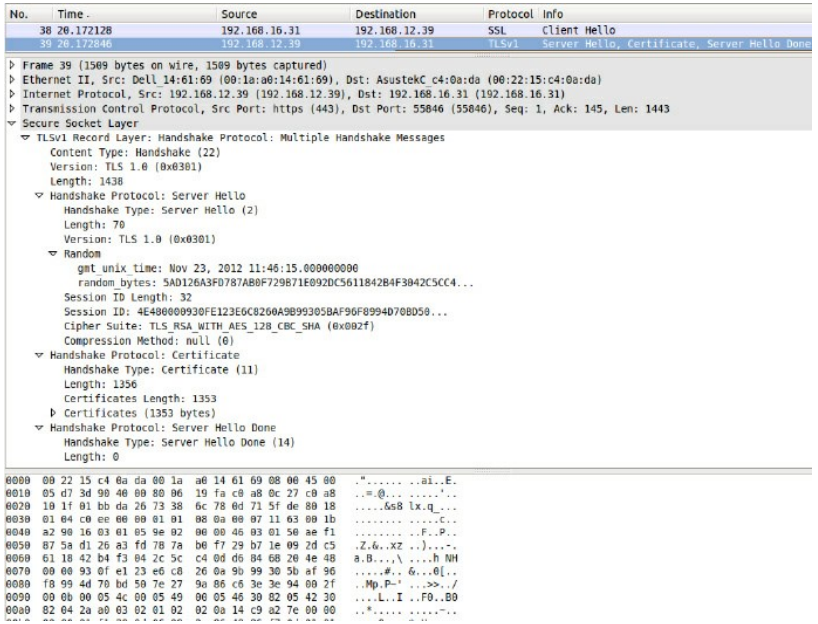
3. 客户端接收到服务端的 hello 消息之后，数字证书就要进行验证了。
   服务端证书的合法性检查包括：
   证书是否过期，
   发行服务器证书的 CA 是否可靠，
   发行者证书的公钥能否正确解开服务器证书的"发行者的数字签名"，
   服务端证书上的域名是否和服务器的实际域名相匹配。
   如果合法性验证没有通过，通讯将断开；
   如果合法性验证通过，将继续进行第 4 步。
4. 客户端随机产生一个用于后面通讯的"对称密钥"，
   然后用服务器的公钥(服务器的公钥从步骤 2 中的服务器的证书中获得)对其加密，
   然后将加密后的"预主密码"传给服务器。
5. 如果服务器要求客户的身份认证(在握手过程中为可选)，
   用户可以建立一个随机数然后对其用自己的私钥进行数据签名(hash值)，
   将这个含有签名的随机数、
   客户端自己的证书
   以及加密过的"预主密码"
   一起传给服务器。
6. 如果服务器要求客户的身份认证，服务器必须检验客户证书和签名随机数的合法性，
   客户端证书的合法性检查包括：
   客户的证书使用日期是否有效，
   为客户提供证书的 CA 是否可靠，
   发行 CA 的公钥能否正确解开客户证书的发行 CA 的数字签名，
   检查客户的证书是否在证书废止列表(CRL)中。
   如果检验没有通过，通讯立刻中断；
   如果检验通过，服务器将用自己的私钥解开加密的"预主密码"，
   然后会通过一个算法使用 "预主密码" 来创建 主通讯密码 。
   注：客户端也将通过同样的算法产生相同的主通讯密码。
7. 服务器和客户端用相同的主密码即"通话密码"，
   一个对称密钥用于 SSL 协议的安全数据通讯的加解密通讯。
   同时在 SSL 通讯过程中还要完成数据通讯的完整性，防止数据通讯中的任何变化。
8. 客户端向服务器端发出信息，指明后面的数据通讯将使用的步骤 7 中的主密码为对称密钥，
   同时通知服务器客户端的握手过程结束。
9. 服务器向客户端发出信息，指明后面的数据通讯将使用的步骤 7 中的主密码为对称密钥，
   同时通知客户端服务器端的握手过程结束。
10. SSL 的握手部分结束，SSL 安全通道的数据通讯开始，
    客户和服务器开始使用相同的对称密钥进行数据通讯，同时进行通讯完整性的检验。

客户端说：Hello





服务端说：Hello



二、开发：

1.头文件：

本教程所使用的头文件只有三个：(它们都位于 include/openssl 子目录中)

ssl.h

ssl 的主要数据结构定义在 ssl.h 中。

bio.h

不管连接是安全的还是不安全的，

OpenSSL 都使用了一个名为 BIO 的抽象库来处理包括文件和套接字在内的各种类型的通信。

err.h

2.初始化：

要初始化 OpenSSL 库，只需要三行代码行即可：

```
// 初始化 SSL 算法库函数，调用 SSL 系列函数之前必须调用此函数！
SSL_library_init();
// 加载 SSL 错误消息
SSL_load_error_strings();
// 加载 BIO 抽象库的错误信息
ERR_load_BIO_strings();
// 加载所有 加密 和 散列 函数
OpenSSL_add_all_algorithms();
```

3.1.建立非安全连接：

```
// 首先，我将向您展示如何建立一个标准的套接字连接。
// 相对于使用 BSD 套接字库，该操作需要的代码行更少一些。
// 在建立连接(无论安全与否)之前，要创建一个指向 BIO 对象的指针。
// 这类似于在标准 C 中为文件流创建 FILE 指针。
BIO* bio = NULL;

// 创建新的连接需要调用 BIO_new_connect 。
// 您可以在同一个调用中同时指定主机名和端口号。
// 也可以将其拆分为两个单独的调用：
```

```
// 一个是创建连接并设置主机名的 BIO_new_connect 调用，
// 一个是设置端口号的 BIO_set_conn_port（或者 BIO_set_conn_int_port ）调用。
// 一旦 BIO 的主机名和端口号都已指定，该指针会尝试打开连接。
// 如果创建 BIO 对象时遇到问题，指针将会是 NULL。
// 下面这行代码使用指定的主机名和端口创建了一个新的 BIO 对象。
// 参数字符串可以是 www.ibm.com:80
bio = BIO_new_connect( "hostname:port" );
if ( NULL == bio )
{
    // 处理故障
}

// 为了确保连接成功，必须执行 BIO_do_connect 调用。
// 如果出错，则返回 0 或 -1。
if ( 0 >= BIO_do_connect( bio ) )
{
    // 处理连接失败
}
```

3.2.1.为安全连接进行设置：

```
// 为安全连接进行设置要多几行代码。
// 同时需要有另一个类型为 SSL_CTX 的指针。该结构保存了一些 SSL 信息。
// 您也可以利用它通过 BIO 库建立 SSL 连接。
// 可以通过使用 SSL 方法函数调用 SSL_CTX_new 来创建这个结构，
// 该方法函数的参数通常是 TLSv1_client_method() (以前是 SSLv23_client_method) 。
SSL_CTX* ctx = SSL_CTX_new( TLSv1_client_method() );

// 还需要另一个 SSL 类型的指针来保持 SSL 连接结构(这是短时间就能完成的一些连接所必需的)。
// 以后还可以用该 SSL 指针来检查连接信息或设置其他 SSL 参数。
SSL* ssl = NULL;
```

3.2.2.加载可信任证书：

```
// 在创建上下文结构之后，必须加载一个可信任证书库。这是成功验证每个证书所必需的。
// 如果不能确认证书是可信任的，那么 OpenSSL 会将证书标记为无效(但连接仍可以继续)。
// OpenSSL 附带了一组可信任证书。它们位于源文件目录树的 certs/demo 目录中。
// 不过，每个证书都是一个独立的文件，也就是说，需要单独加载每一个证书。
// 在 certs 目录下，还有一个 expired 存放过期证书的子目录，试图加载这些证书将会出错。
// 在数字证书进行信任验证之前，
// 必须为在为安全连接设置时创建的 OpenSSL SSL_CTX 对象提供一个默认的信任证书，
// 这可以使用几种方法来提供，
// 但是最简单的方法是将这个证书保存为一个 PEM 文件，
// 并使用
// SSL_CTX_load_verify_locations( ctx, file, path );
// 将其加载到 OpenSSL 中。
// 该函数有三个参数：
// ctx  - 上下文指针( SSL_CTX_new 函数返回 );
// file - 包含一个或多个 PEM 格式的证书的文件的路径(必需);
// path - 到一个或多个 PEM 格式文件的路径，不过文件名必须使用特定的格式(可为 NULL);
// 如果指定成功，则返回 1 ，如果遇到问题，则返回 0 。
// 尽管当信任证书在一个目录中有多个单独的文件时更容易添加或更新，
// 但是您不太可能会如此频繁地更新信任证书，因此不必担心这个问题。
if ( !SSL_CTX_load_verify_locations( ctx,
                                     "/path/to/trusted.pem", // CA 的信任证书
                                     NULL ) )
{
    // 在这里处理失败的加载
}

// 如果打算使用目录存储可信任库，那么必须要以特定的方式命名文件。
// OpenSSL 附带了一个名为 c_rehash 的工具(在 tools 目录中的一个脚本文件)，
// 它可以将文件夹配置为可用于 SSL_CTX_load_verify_locations 的路径参数。
if ( !SSL_CTX_load_verify_locations( ctx,
                                     NULL,
                                     "/path/to/certfolder" ) )
```

```
  {
       // 在这里处理错误
  }
```

注:3.2.3.*. 是建立安全连接双向认证的,如果不需要验证客户端,可以不用添加这些函数调用!

3.2.3.1.加载客户端证书

```
  // 加载客户端证书 - 用法与 加载可信任证书 一样
  if ( !SSL_CTX_use_certificate_file( ctx,
 "/path/to/clientcert.pem",
 SSL_FILETYPE_PEM ) )
  {
       // 在这里处理错误
  }
```

3.2.3.2.加载客户端私钥文件:

```
  // 加载客户端私钥文件 - 用法与 加载可信任证书 一样
  if ( !SSL_CTX_use_PrivateKey_file( ctx,
"private.key",
SSL_FILETYPE_PEM ) )
  {
       // 在这里处理错误
  }
```

3.2.3.3.检查私钥是否和证书匹配:

```
  // 验证私钥是否与证书一致
  if ( !SSL_CTX_check_private_key( ctx ) )
  {
       // 在这里处理错误
  }
```

3.2.4.创建连接:

```
  // 将指向 SSL 上下文的指针作为惟一参数,使用 BIO_new_ssl_connect 创建 BIO 对象。
  bio = BIO_new_ssl_connect( ctx );

  // 还需要获得指向 SSL 结构的指针,在本文中,只将该指针用于 SSL_set_mode 函数。
  BIO_get_ssl( bio, &ssl );

  // 而这个函数是用来设置 SSL_MODE_AUTO_RETRY 标记的。
  // 使用这个选项进行设置,如果服务器突然希望进行一次新的握手,那么 OpenSSL 可以在后台处理
它。
  // 如果没有这个选项,当服务器希望进行一次新的握手时,进行读或写操作都将返回一个错误,
  // 同时还会在该过程中设置 retry 标记。
  SSL_set_mode( ssl, SSL_MODE_AUTO_RETRY );
```

3.2.5.打开安全连接:

```
  // 设置 SSL 上下文结构之后,就可以创建连接了。
  // 主机名是使用 BIO_set_conn_hostname 函数设置的。
  // 主机名和端口的指定格式与前面的相同,也可以是 192.168.12.39:443。
  // 该函数还可以建立与服务端的连接。
  BIO_set_conn_hostname( bio, "hostname:port" );

  // 为了确认已经成功打开连接,必须执行对 BIO_do_connect 的调用。
  // 该调用还将执行握手来建立安全连接。
  if ( 0 >= BIO_do_connect( bio ) )
  {
       // 处理失败的连接
  }
```

3.2.6.检查证书是否有效:

```
  // 连接建立后,必须检查证书,以确定它是否有效。
  // 实际上,OpenSSL 为我们完成了这项任务。
  // 如果证书有致命的问题(例如,哈希值无效),那么将无法建立连接。
  // 但是,如果证书的问题并不是致命的(当它已经过期或者尚不合法时),那么仍可以继续使用连接。
  // 可以将 SSL 结构作为惟一参数,
```

```
  // 调用 SSL_get_verify_result 来查明证书是否通过了 OpenSSL 的检验。
  // 如果证书通过了包括信任检查在内的 OpenSSL 的内部检查，则返回 X509_V_OK。
  // 如果有地方出了问题，则返回一个错误代码，在 OpenSSL 文档的 verify 部分中都进行了介绍。
  // 注:该错误代码被记录在命令行工具的 verify 选项下。
  // 应该注意的是，验证失败并不意味着连接不能使用。
  // 是否应该使用连接取决于验证结果和安全方面的考虑。
  // 例如，失败的信任验证可能只是意味着没有可信任的证书。
  // 连接仍然可用，只是需要从思想上提高安全意识。
  // OpenSSL 在对证书进行验证时，有一些安全性检查并没有执行，
  // 包括证书的失效检查和对证书中通用名的有效性验证。
  if ( X509_V_OK != SSL_get_verify_result( ssl ) )
  {
       //处理失败验证
  }
```

3.2.7.检索证书:

```
  // 如果您希望向用户显示证书的内容，或者要根据主机名或证书权威对证书进行验证，
  // 那么就需要检索证书的内容。
  // 要在验证测试结果之后再检索证书，请调用 SSL_get_peer_certificate() 。
  // 它返回一个指向该证书的 X509 指针，如果证书不存在，就返回 NULL 。
  X509* peerCertificate = NULL;
  if ( X509_V_OK == SSL_get_verify_result(ssl) )
  {
       peerCertificate = SSL_get_peer_certificate( ssl );
  }
  else
  {
       // 在这里处理检验错误
  }
```

3.2.8.验证证书:

```
  // 在握手时所提供的服务器的证书应该有一个名字与该服务器的主机名匹配。
  // 如果没有，那么这个证书就应该标记为值得怀疑的。
  // 内部验证过程已经对证书进行信任和有效期的验证;
  // 如果这个证书已经超期，或者包含一个不可信的签名，那么这个证书就会被标记为无效的。
  // 由于这不是 SSL 标准的一部分，因此 OpenSSL 并不需要根据主机名对该证书的名字进行检查。
  // 证书的"名字"实际上是证书中的 Common Name 字段。
  // 这个字段应该从证书中进行检索，并根据主机名进行验证。
  // 如果二者不能匹配，就只有怀疑这个证书无效了。
  // 有些公司(例如 Yahoo)就在不同的主机上使用相同的证书，
  // 即使证书中的 Common Name 只是用于一个主机的。
  // 为了确保这个证书是来自于相同的公司，可以进行更深入的检查，但是这完全取决于项目的安全性需
要。
  // 从证书中检索通用名需要两个步骤:
  // 1)从证书结构检索 X509_NAME 对象。
  //   使用 X509_get_subject_name() 从证书中检索 X509_NAME 结构。
  //   这会返回一个指向 X509_NAME 的对象。
  // 2)然后从 X509_NAME 对象检索名字。
  //   请使用 X509_NAME_get_text_by_NID() 来检索通用名，并保存到一个字符串中。
  char commonName [512];
  X509_NAME* name = X509_get_subject_name( peerCertificate );
  X509_NAME_get_text_by_NID( name,
                             NID_commonName, // openssl-0.9.8o 的宏值为 13
                             commonName,
                             512 );
  // 使用标准的 C 字符串函数或您习惯使用的字符串库对通用名和主机名进行比较。
  // 对不匹配的处理，完全取决于项目的需要或用户的决策。
  // 如果要更深入地进行检查，我建议使用一个单独的字符串库来降低复杂性。
  // 以下只是简单比较，如果必要的话，可以更彻底地检查证书
  if ( 0 != stricmp( commonName,
                     hostname ) )
  {
       // 在这里处理一个可疑的证书
  }
```

3.2.9.清除 SSL 上下文：

```
// 必须在结束应用程序之前的某个时刻释放 SSL 上下文结构。
// 可以调用 SSL_CTX_free 来释放该结构。
SSL_CTX_free( ctx );
```

4.与服务器进行通信：

不管 BIO 对象是套接字还是文件，对其进行的读和写操作都是通过以下两个函数来完成的：
BIO_read 和 BIO_write 。

```
// BIO_read 将尝试从服务器读取一定数目的字节。
// 它返回读取的字节数、0 或者 -1。
// 在受阻塞的连接中，该函数返回 0，表示连接已经关闭，而 -1 则表示连接出现错误。
// 在非阻塞连接的情况下，返回 0 表示没有可以获得的数据，返回 -1 表示连接出错。
// 可以调用 BIO_should_retry 来确定是否可能重复出现该错误。
int x = BIO_read( bio, buf, len );
if ( 0 == x )
{
     // 处理已关闭的连接
}
else if ( 0 > x  )
{
     if ( !BIO_should_retry( bio ) )
     {
          // 在这里处理读取失败
     }

     // 做某事以尝试重试
}

// BIO_write 会试着将字节写入套接字。
// 它将返回实际写入的字节数、0 或者 -1。
// 同 BIO_read ，0 或 -1 不一定表示错误。
// BIO_should_retry 是找出问题的途径。
// 如果需要重试写操作，它必须使用和前一次完全相同的参数。
if ( 0 >= BIO_write( bio, buf, len ) )
{
     if ( !BIO_should_retry( bio ) )
     {
          // 在这里处理写入失败
     }

     // 做某事以尝试重试
}
```

5.关闭连接：

关闭连接也很简单。
您可以使用以下两种方式之一来关闭连接：BIO_reset 或 BIO_free_all 。
如果您还需要重新使用对象，那么请使用第一种方式。
如果您不再重新使用它，则可以使用第二种方式。

```
// BIO_reset 关闭连接并重新设置 BIO 对象的内部状态，以便可以重新使用连接。
// 如果要在整个应用程序中使用同一对象，比如使用一台安全的聊天客户机，那么这样做是有益的。
// 该函数没有返回值。
BIO_reset( bio );

// BIO_free_all 所做正如其所言：
// 它释放内部结构体，并释放所有相关联的内存，其中包括关闭相关联的套接字。
// 注:如果将 BIO 嵌入于一个类中，那么应该在类的析构函数中使用这个调用。
BIO_free_all(bio);
```

6.错误检测：

首先，您需要得到错误代码本身；
     ERR_get_error 可以完成这项任务；
然后，需要将错误代码转换为错误字符串，

它是一个指向由 SSL_load_error_strings 或
ERR_load_BIO_strings
加载到内存中的永久字符串的指针。
可以在一个嵌套调用中完成这项操作。

ERR_reason_error_string 返回一个静态字符串的指针，
然后可以将字符串显示在屏幕上、写入文件，
或者以任何您希望的方式进行处理。

ERR_lib_error_string 指出错误发生在哪个库中。

ERR_func_error_string 返回导致错误的 OpenSSL 函数。

```
// 例如:打印出最后一个错误
printf( "Error: %s\n",
        ERR_reason_error_string( ERR_get_error() ) );
```

您还可以让库给出预先格式化了的错误字符串。
可以调用 ERR_error_string 来得到该字符串。
该函数将错误代码和一个预分配的缓冲区作为参数。
而这个缓冲区必须是 256 字节长。
如果参数为 NULL，
则 OpenSSL 会将字符串写入到一个长度为 256 字节的静态缓冲区中，并返回指向该缓冲区的指针。
否则，它将返回您给出的指针。
如果您选择的是静态缓冲区选项，那么在下一次调用 ERR_error_string 时，该缓冲区会被覆盖。

```
printf( "%s\n",
        ERR_error_string( ERR_get_error(),
                          NULL ) );
```

您还可以将整个错误队列转储到文件或 BIO 中。
可以通过 ERR_print_errors 或 ERR_print_errors_fp 来实现这项操作。
队列是以可读格式被转储的。
第一个函数将队列发送到 BIO ，

```
ERR_print_errors( BIO* );
```

第二个函数将队列发送到 FILE 。

```
ERR_print_errors_fp( FILE* );
```

字符串格式如下(引自 OpenSSL 文档):

```
[pid]:error:[error code]:[library name]:[function name]:[reason string]:[file name]:
[line]:[optional text message]
```

其中，

[pid] 是进程 ID，
[error code] 是一个 8 位十六进制代码，
[file name] 是 OpenSSL 库中的源代码文件，
[line] 是源文件中的行号。

以下为 访问 HTTPS 的双向认证示例代码:

```
#include "stdio.h"
#include "string.h"

// 使用 OpenSSL API 进行安全编程所需的三个头文件
// debug openssl 可以在 openssl-0.9.8o/Makefile 中的 CFLAG 变量里添加 -g -O0
// 本程序需要的 openssl 库是 libcrypto.a 和 libssl.a ，还需要 -l dl
// ssl 的主要数据结构定义在 ssl.h 中。
#include <openssl/ssl.h>
// BIO 抽像库来处理包括文件和套接字在内的各种类型的通信。
#include <openssl/bio.h>
//
#include <openssl/err.h>

int
main()
```

```
{
    int iResult = 0; // 零为成功,负数为错误码

    BIO*        bio = NULL;
    SSL*        ssl = NULL;
    SSL_CTX*    ctx = NULL;
    X509*       pX509 = NULL;
    SSL_METHOD* sslMethod = NULL;

    char        commonName[512] = { 0 };
    X509_NAME*  pX509_NAME = NULL;

    // http://192.168.12.39/testssl 对应的 HTTP 请求协议
    // 其中的 /testssl/ 最后的 / 是必须的
    char szHttpReq[] = "GET /testssl/ HTTP/1.1\r\n"
                       "Host:192.168.12.39\r\n"
                       "Connection: keep-alive\r\n\r\n";

    // 一次收到的字节数
    unsigned int uiRecBytes = 0;

    // 接收服务端信息的字节缓冲区
    unsigned char baRecBuffer[1024];
    // 最终收完后的服务端信息缓冲区
    unsigned char* pbRecFinish = NULL;
    // 临时备份缓冲区
    unsigned char* pbRecBak = NULL;
    // 临时备份缓冲区字节大小
    unsigned int uiRecBakLen = 0;

    // 按行打印收到的信息
    char* pMsgline = NULL;

    // 初始化 OpenSSL 库,只需以下四行代码
    // 初始化 SSL 算法库函数,调用 SSL 系列函数之前必须调用此函数!
    SSL_library_init();
    // 加载 BIO 抽像库错误信息
    ERR_load_BIO_strings();
    // 加载 SSL 抽像库错误信息
    SSL_load_error_strings();
    // 加载所有加密和散列函数
    OpenSSL_add_all_algorithms();

    do // 非循环,只是为了减少分支缩进
    {
        // 设置客户端使用的 SSL 协议算法
        sslMethod = TLSv1_client_method();
        if ( NULL == sslMethod )
        {
            printf( "TLSv1_client_method err: %s\n",
                    ERR_error_string( ERR_get_error(),
                                      NULL ) );
            iResult = -1;
            break;
        }

        // 创建 SSL 上下文
        ctx = SSL_CTX_new( sslMethod );
        if ( NULL == ctx )
        {
            printf( "SSL_CTX_new err: %s\n",
                    ERR_error_string( ERR_get_error(),
                                      NULL ) );
            iResult = -2;
```

```
    break;
}

// 加载可信任的 CA 证书
if( 0 == SSL_CTX_load_verify_locations( ctx,
                                        "certnew.pem",
                                        NULL ) )
{
    printf( "SSL_CTX_load_verify_locations err: %s\n",
            ERR_error_string( ERR_get_error(),
                              NULL ) );
    iResult = -3;
    break;
}

// 加载客户端证书
if ( 0 == SSL_CTX_use_certificate_file( ctx,
                                        "test1.pem",
                                        SSL_FILETYPE_PEM ) )
{
    printf( "SSL_CTX_use_certificate_file err: %s\n",
            ERR_error_string( ERR_get_error(),
                              NULL ) );
    iResult = -4;
    break;
}

// 加载客户端私钥文件
if ( 0 == SSL_CTX_use_PrivateKey_file( ctx,
                                       "private.key",
                                       SSL_FILETYPE_PEM ) )
{
    printf( "SSL_CTX_use_PrivateKey_file err: %s\n",
            ERR_error_string( ERR_get_error(),
                              NULL ) );
    iResult = -5;
    break;
}

// 验证私钥是否与证书一致
if ( 0 == SSL_CTX_check_private_key( ctx ) )
{
    printf( "SSL_CTX_check_private_key err: %s\n",
            ERR_error_string( ERR_get_error(),
                              NULL ) );
    iResult = -6;
    break;
}

// 创建 BIO 对象
bio = BIO_new_ssl_connect( ctx );
if ( NULL == bio )
{
    printf( "BIO_new_ssl_connect err: %s\n",
            ERR_error_string( ERR_get_error(),
                              NULL ) );
    iResult = -7;
    break;
}

// 获得指向 SSL 结构的指针
BIO_get_ssl( bio, &ssl );

// SSL_MODE_AUTO_RETRY - 如果服务端希望进行一次新的握手，OpenSSL 后台处理它。
```

```
// 没有 SSL_MODE_AUTO_RETRY 此选项，则新握手返回错误，且设置 retry 标记。
SSL_set_mode( ssl,
              SSL_MODE_AUTO_RETRY );

// 建立与服务端的连接
BIO_set_conn_hostname( bio, "192.168.12.39:443" );

// 为了确认成功打开连接，需执行 BIO_do_connect 函数
// 该调用还将执行握手来建立安全连接
if ( 0 >= BIO_do_connect( bio ) )
{
    printf( "BIO_do_connect err: %s\n",
            ERR_error_string( ERR_get_error(),
                              NULL ) );
    iResult = -8;
    break;
}

// 连接建立后，必须检查证书，以确定它是否有效。
// 实际上，OpenSSL 为我们完成了这项任务。
// 如果证书有致命的问题(如：哈希值无效)，那么将无法建立连接。
// 如果证书无致命的问题(如：已过期或尚不合法时)，那么可以继续使用连接。
// 调用 SSL_get_verify_result 来查明证书是否通了 OpenSSL 的检验。
// 如果证书通过了包括信任检查在内的 OpenSSL 的内部检查，则返回 X509_V_OK 。
// 如果有地方出了问题，则返回一个错误码，该代码被记录在命令行工具的 verify 选项下。
if ( X509_V_OK != SSL_get_verify_result( ssl ) )
{
    printf( "SSL_get_verify_result err: %s\n",
            ERR_error_string( ERR_get_error(),
                              NULL ) );
    iResult = -9;
    break;
}

// 如果您希望向用户显示证书的内容，或者要根据主机名或证书权威对证书进行验证，
// 那么就需要检索证书的内容。
// 要在验证测试结果之后再检索证书，请调用 SSL_get_peer_certificate()。
// 它返回一个指向该证书的 X509 指针，如果证书不存在，就返回 NULL 。
pX509 = SSL_get_peer_certificate( ssl );
if ( NULL == pX509 )
{
    printf( "SSL_get_peer_certificate err: %s\n",
            ERR_error_string( ERR_get_error(),
                              NULL ) );
    iResult = -10;
    break;
}

// 使用 X509_get_subject_name() 从证书中检索 X509_NAME 结构。
// 这会返回一个指向 X509_NAME 的对象。
pX509_NAME = X509_get_subject_name( pX509 );
if ( NULL == pX509_NAME )
{
    printf( "X509_get_subject_name err: %s\n",
            ERR_error_string( ERR_get_error(),
                              NULL ) );
    iResult = -10;
    break;
}

X509_NAME_get_text_by_NID( pX509_NAME,
                           NID_commonName,
                           commonName,
                           512 );
```

```
if ( 0 != strcasecmp( commonName,
                      "192.168.12.39" ) )
{
    printf( "Certificate's name 192.168.12.39 != %s\n",
            commonName );
    iResult = -11;
    break;
}

// 通过 SSL 发送 HTTP 请求
// BIO_write 会试着将字节写入套接字。
// 它将返回实际写入的字节数、0 或者 -1。
// 同 BIO_read，0 或 -1 不一定表示错误。
// BIO_should_retry 是找出问题的途径。
// 如果需要重试写操作，它必须使用和前一次完全相同的参数。
if ( 0 >= BIO_write( bio,
                     szHttpReq,
                     strlen( szHttpReq ) ) )
{
    printf( "Sent HTTP request failed by BIO_write\n" );
    iResult = -12;
    break;
}

// 开始接收服务端信息数据
// BIO_read 将尝试从服务器读取一定数目的字节。
// 它返回读取的字节数、 0 或者 -1。
// 在受阻塞的连接中，该函数返回 0，表示连接已经关闭，而 -1 则表示连接出现错误。
// 在非阻塞连接的情况下，返回 0 表示没有可以获得的数据，返回 -1 表示连接出错。
// 可以调用 BIO_should_retry 来确定是否可能重复出现该错误。
while ( 0 < ( uiRecBytes = BIO_read( bio,
                                     (void*)baRecBuffer,
                                     sizeof( baRecBuffer ) ) ) )
{
    // 最终收完后的服务端信息缓冲区指针不为空说明要先备份后扩展
    uiRecBakLen = 0; // 每次都清空以防计算错误
    if ( NULL != pbRecFinish )
    {
        uiRecBakLen = strlen( (char*)pbRecFinish );

        pbRecBak = (unsigned char*)malloc( uiRecBakLen );

        // 分配失败
        if ( NULL == pbRecBak )
        {
            printf( "pbRecBak call malloc to allocate mem fail\n" );
            free( pbRecFinish );
            pbRecFinish = NULL;

            uiRecBakLen = 0; // 每次都清空以防计算错误

            break;
        }

        // 结尾的零没有一起拷贝!!!
        memcpy( pbRecBak, pbRecFinish, uiRecBakLen );

        free( pbRecFinish );
        pbRecFinish = NULL;
    }

    // 备份缓冲区和接收缓冲区都没有结尾的零，所以要多分一个字节
    pbRecFinish = (unsigned char*)malloc( uiRecBakLen + uiRecBytes + 1 );
```

```
        // 分配失败
        if ( ( NULL == pbRecFinish ) &&
             ( NULL != pbRecBak ) )
        {
            printf( "pbRecFinish call malloc to allocate mem fail\n" );
            free( pbRecBak );
            pbRecBak = NULL;

            uiRecBakLen = 0; // 每次都清空以防计算错误

            break;
        }

        // 全清零，此操作比数组快，且为字符串最后补零了
        memset( pbRecFinish, 0, uiRecBakLen + uiRecBytes + 1 );

        // 先恢复备份的
        if ( NULL != pbRecBak )
        {
            memcpy( pbRecFinish, pbRecBak, uiRecBakLen );

            free( pbRecBak );
            pbRecBak = NULL;
        }

        // 再拷贝入新收的
        memcpy( pbRecFinish + uiRecBakLen, baRecBuffer, uiRecBytes );

        // 未收满，说明没有数据可收了
        if ( sizeof( baRecBuffer ) > uiRecBytes )
        {
            break;
        }
    }

    if ( NULL != pbRecFinish )
    {
        printf( "%s\n", strtok( (char*)pbRecFinish,
                                "\r\n" ) );
        while( NULL != ( pMsgline = strtok( NULL, "\r\n" ) ) )
        {
            printf( "%s\n", pMsgline );
        }

        free( pbRecFinish );
        pbRecFinish = NULL;
    }

    break; // 流程到此结束
} while ( 0 );

if ( NULL != pbRecFinish )
{
    // 释放申请的内存
    free( pbRecFinish );
    pbRecFinish = NULL;
}

if ( NULL == bio )
{
    // 释放内部结构体相关的内存
    BIO_free_all( bio );
}
```

```
    if ( NULL != ctx )
    {
        // 清除 SSL 上下文
        SSL_CTX_free( ctx );
    }

    return iResult;
}
```


前一篇:李开复 - 互联网的九个产品精神

后一篇:定制 OpenSSL 访问 HTTPS 验证客户端时使用 USBKey 中的私钥做签名

< 前一篇
李开复 - 互联网的九个产品精神

后一篇 >
定制 OpenSSL 访问 HTTPS 验证客户端时使用 USBKey 中的私钥做签名