网络安全实验三

VPN 实验指导手册

华中科技大学网络空间安全学院

二零二二年五月

VPN 实验

1. 实验目的

虚拟专用网络（VPN）用于创建计算机通信的专用的通信域，或为专用网络到不安全的网络（如 Internet）的安全扩展。VPN 是一种被广泛使用的安全技术。在 IPSec 或 TLS/SSL（传输层安全性/安全套接字层）上构建 VPN 是两种根本不同的方法。本实验中，我们重点关注基于 TLS/SSL 的 VPN。这种类型的 VPN 通常被称为 TLS/SSL VPN。

本实验的学习目标是让学生掌握 VPN 的网络和安全技术。为实现这一目标，要求学生实现简单的 TLS/SSL VPN。虽然这个 VPN 很简单，但它包含了 VPN 的所有基本元素。TLS/SSL VPN 的设计和实现体现了许多安全原则，包括以下内容：

- 虚拟专用网络
- TUN/TAP 和 IP 隧道
- 路由
- 公钥加密，PKI 和 X.509 证书
- TLS/SSL 编程
- 身份认证

2. 实验环境

VMware Workstation 虚拟机。

Ubuntu 16.04 操作系统（SEEDUbuntu16.04）。

本次实验需要使用 openssl 软件，该软件包含头文件，库函数和命令。该软件包已经安装在我们上述 VM 镜像中。

3. 实验要求

熟悉 TLS/SSLVPN 的原理。

了解 TUN/TAP 设备工作原理。

使用本实验提供的虚拟机完成实验内容。

通过实验课的上机实验，演示给实验指导教师检查，并提交详细的实验报告。

4. 实验内容

本次实验，学生需要为 Linux 操作系统实现一个简单的 VPN。我们将其称为

miniVPN。

4.1 任务 1：配置环境

我们将在计算机（客户端）和网关之间创建 VPN 隧道，允许计算机通过网关安全地访问专用网络。我们使用虚拟机本身作为 VPN 服务器网关（VM），并创建两个容器分别作为 VPN 客户端（HostU）和专用网络中的主机（HostV）。网络设置如图 1 所示。

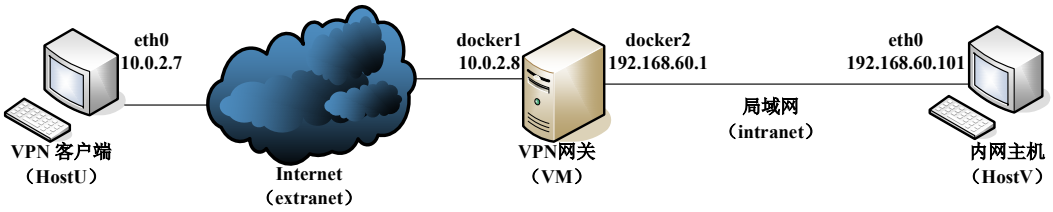

图 1 配置虚拟机实验环境

在实际环境中，VPN 客户端（HostU）和 VPN 服务器网关的外网口通过 Internet 连接。简单起见，我们在本实验中将这两台机器直接连接到同一 docker 网络“extranet”，模拟 Internet。第三台机器 HostV 是内部局域网的计算机。Internet 主机 HostU 上的用户希望通过 VPN 隧道与内部局域网的主机 HostV 通信。为模拟此设置，我们使用 docker 网络“intranet”将 HostV 与 VPN 服务器网关的内网口连接，模拟内部局域网。在这种设置中，HostV 不能直接从 Internet 访问，即不能直接从 HostU 访问。为实现上述的网络环境配置，我们需要执行以下操作：

```
在 VM 上创建 docker 网络 extranet
root@VM:/$ sudo docker network create --subnet=10.0.2.0/24 --
gateway=10.0.2.8 --opt "com.docker.network.bridge.name"="docker1"
extranet
在 VM 上创建 docker 网络 intranet
root@VM:/$ sudo docker network create --subnet=192.168.60.0/24 --
gateway=192.168.60.1 --opt "com.docker.network.bridge.name"="docker2"
intranet

在 VM 上新开一个终端，创建并运行容器 HostU
root@VM:/$ sudo docker run -it --name=HostU --hostname=HostU --
net=extranet --ip=10.0.2.7 --privileged "seedubuntu" /bin/bash
在 VM 上新开一个终端，创建并运行容器 HostV
root@VM:/$ sudo docker run -it --name=HostV --hostname=HostV --
net=intranet --ip=192.168.60.101 --privileged "seedubuntu" /bin/bash

在容器 HostU 和 HostV 内分别删除掉默认路由
root@HostU:/# route del default
root@HostV:/# route del default
```

4.2 任务 2：使用 TUN/TAP 创建一个主机到主机的隧道

TLS/SSL VPN 中使用了 TUN/TAP 技术，其在现代操作系统中已经广泛使用。TUN 和 TAP 是虚拟网络内核驱动程序；它们可以实现完全由软件支持的网络设备。TAP 模拟以太网设备，处理的是以太网帧等二层数据包；TUN 模拟网络层设备，处理的是 IP 等三层数据包。我们可以用 TAP/TUN 创建虚拟网络接口。

用户空间程序通常访问 TUN/TAP 虚拟网络接口。操作系统通过 TUN/TAP 网络接口将数据包传送到用户空间程序。另一方面，程序通过 TUN/TAP 网络接口发送的数据包被注入操作系统网络栈；在操作系统看来，数据包是通过虚拟网络接口的外部源进来的。

当程序从 TUN/TAP 接口读取数据时，计算机发送到此接口的 IP 数据包将被传送给程序；另一方面，程序发送到接口的 IP 数据包将被传送到计算机中，就好像这些数据包通过这个虚拟网络接口从外部来的一样。程序可以使用标准的 read()和 write()系统调用来接收或发送数据包到虚拟接口。

我们已经创建了一个示例 VPN 客户端程序（vpnclient）和一个服务器程序（vpnserver），这两个程序都会上传到群共享文件。

vpnclient 和 vpnserver 程序是 VPN 隧道的两端，它们使用 TCP 或 UDP 协议通过图 3 中描述的套接字相互通信。为简单起见，在我们的示例代码中，我们选择使用 UDP。客户端和服务器之间的虚线描述了 VPN 隧道的路径。VPN 客户端和服务器程序通过 TUN 接口连接到主机系统，做以下两件事：

（1）从主机系统获取 IP 数据包，因此数据包可以通过隧道发送；

（2）从隧道获取 IP 数据包，然后将其转发到主机系统，主机系统将数据包转发到其最终目的地。以下过程介绍了如何使用 vpnclient 和 vpnserver 程序创建 VPN 隧道。

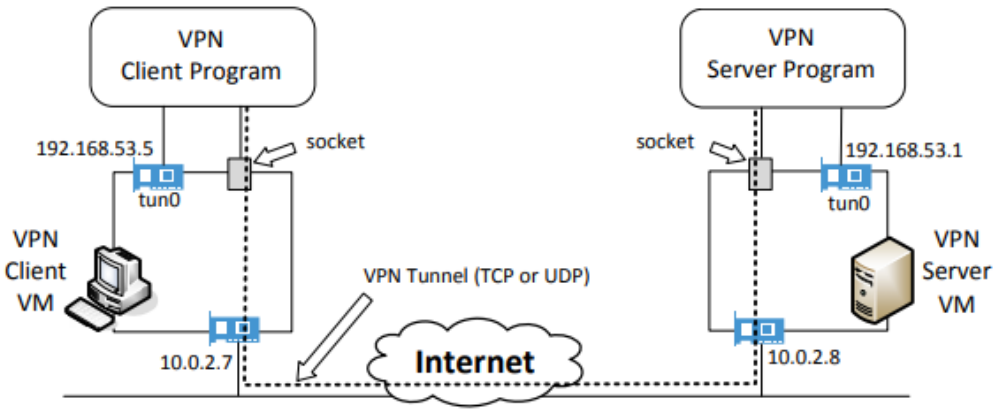

图 3 VPN 的客户端和服务端

第 1 步：运行 VPN 服务器

我们首先在服务器 VM 上运行 VPN 服务器程序 vpnserver。程序运行后，系统中将出现一个虚拟 TUN 网络接口（我们可以使用“ifconfig -a”命令看到它；在大多数情况下，接口的名称将是 tun0，但它们可以是 tunX，其中 X 是一个数

字）。此新接口尚未配置，因此我们需要通过为其提供 IP 地址来配置它。我们使用 192.168.53.1 作为此接口的 IP 地址。

运行以下命令。第一个命令将启动服务器程序，第二个命令将一个 IP 地址分配给 tun0 接口，然后激活它。老版程序中，第一个命令将阻塞并等待连接，因此我们需要在另一个终端运行第二个命令。新版程序改进为守护进程运行，可以在同一终端继续操作，由于程序有打印调试信息，新开一个终端运行第二个命令仍是较好的选择。

```
在 VM 上启动 VPN 服务
root@VM:/$ sudo ./vpnserver

在 VM 上的另一个终端配置 tun0 虚拟 IP 地址并激活接口
root@VM:/$ sudo ifconfig tun0 192.168.53.1/24 up
```

除非特别配置，否则计算机将仅充当主机，而不充当网关。VPN Server 需要在内网和隧道之间转发报文，因此需要作为网关。我们需要为计算机启用 IP 转发，使其行为类似于网关。由于 VM 上的 iptables 规则可能阻断转发报文，我们还需要清除 iptables 规则。

```
在 VM 上启用 IP 转发
root@VM:/$ sudo sysctl net.ipv4.ip_forward=1

在 VM 上清除 iptables 规则
root@VM:/$ sudo iptables -F
```

第 2 步：运行 VPN 客户端

我们现在在容器 HostU 上运行 VPN 客户端程序。我们首先要在 VM 上将编译好的 VPN 客户端拷贝到容器 HostU 中。

```
在 VM 上拷贝 VPN 客户端
root@VM:/$ sudo docker cp vpnclient HostU:/vpnclient
```

然后在 HostU 中运行以下命令。第一个命令将启动 VPN 客户端以连接到 10.0.2.8 上运行的 VPN 服务器程序。老版程序此命令也会阻塞，因此我们需要在另一个终端配置 VPN 客户端程序创建的 tun0 接口。新版程序改进为守护进程运行，可以在同一终端继续操作，由于程序有打印调试信息，新开一个终端运行第二个命令仍是较好的选择。我们将 IP 地址 192.168.53.5 分配给 tun0 接口。

```
在容器 HostU 中启动 VPN 客户端
root@HostU:/# ./vpnclient 10.0.2.8

在容器 HostU 中配置 tun0 虚拟 IP 地址并激活接口
root@HostU:/# ifconfig tun0 192.168.53.5/24 up
```

第 3 步：在 HostU 和 VPN 服务器上设置路由

完成上述两个步骤后，隧道将会建立。在我们使用隧道之前，我们需要在 HostU 和 VPN 服务器上设置路由，以指示通过隧道的预期流量。在 HostU 上，我们需要将所有进入专用网络（192.168.60.0/24）的数据包定向到 tun0 接口，从该接口可以通过 VPN 隧道转发数据包。如果没有此设置，我们将无法访问专用网络。我们可以使用 route 命令添加路由条目。以下示例显示如何将 192.168.60.0/24 数据包路由到接口 tun0。

```
在 HostU 上创建隧道路由
root@HostU:/# route add -net 192.168.60.0/24 tun0
```

在客户端和服务器计算机上，我们还需要设置路由条目，以便将进入 192.168.53.0/24 网络的所有流量定向到 tun0 接口。当我们将 192.168.53.X 分配给 tun0 接口时，通常会自动添加此条目。如果由于某些原因未添加，我们可以使用 route 命令添加它。

第 4 步：在 HostV 上设置路由

当 HostV 回复从 HostU 发送的数据包时，它需要将数据包路由到 VPN 服务器虚拟机，从那里，它可以被送入 VPN 隧道到另一端。

你需要找出要添加的条目，然后使用 route 命令添加路由条目。提示：当 HostV 从 HostU（通过隧道）接收到数据包时，你需要知道数据包中的源 IP 是什么；在回复数据包中，源 IP 成为目标 IP，将由路由表使用。因此，你需要确定从 HostU 到 HostV 的数据包的源 IP。在此步骤中，你需要弄清楚这一点并正确设置路由。

第 5 步：测试 VPN 隧道

完成所有设置后，我们可以通过隧道从 HostU 访问 HostV。请使用 ping 和 telnet 进行以下测试，并报告你的结果。使用 wireshark 捕获 HostU 上所有接口上的网络流量，并确定哪些数据包是隧道流量的一部分，哪些数据包不是隧道流量

```
在 HostU 上测试隧道通信
root@HostU:/# ping 192.168.60.101
root@HostU:/# telnet 192.168.60.101
```

第 6 步：Tunnel 断开测试

在 HostU 上，telnet 到 HostV。在保持 telnet 连接存活的同时，断开 VPN 隧道。然后我们在 telnet 窗口中输入内容，并报告观察到的内容。然后我们重新连接 VPN 隧道。需要注意的是，重新连接 VPN 隧道，需要将 vpnserver 和 vpnclient 都退出后再重复操作，请思考原因是什么。正确重连后，telnet 连接会发生什么？会被断开还是继续？请描述并解释你的观察结果。

4.3 任务 3：加密隧道

此时，我们已经创建了一个 IP 隧道，但是我们的隧道没有受到保护。只有在我们保障了这个隧道的安全之后，才能将其称为 VPN 隧道。这就是我们在这项任务中要实现的目标。为了保护这条隧道，我们需要实现两个目标，即机密性和完整性。使用加密来实现机密性，即，通过隧道的内容将被加密。完整性目标确保没有人可以篡改隧道中的流量或发起重放攻击。使用消息验证代码（MAC）可以实现完整性。可以使用传输层协议（TLS）实现这两个目标。

TLS 通常建立在 TCP 之上。任务 2 中的示例 VPN 客户端和服务器程序使用 UDP，因此我们首先需要使用 TCP 通道替换示例代码中的 UDP 通道，然后在隧道的两端之间建立 TLS 会话。TLS 客户端和服务器程序示例（tlsclient 和 tlsserver）。tgz 压缩文件中包含的 README 文件中提供了有关如何编译和运行代码的说明。在演示中，你需要使用 Wireshark 捕获 VPN 隧道内的流量，并显示流量确实已加密。

4.4 任务 4：认证 VPN 服务器

在建立 VPN 之前，VPN 客户端必须对 VPN 服务器进行身份认证，确保服务器不是假冒的服务器。另一方面，VPN 服务器必须认证客户端（即用户），确保用户具有访问专用网络的权限。在这个任务中，我们实现服务器认证。客户端身份认证在下一个任务中。

认证服务器的典型方法是使用公钥证书。VPN 服务器需要首先从证书颁发机构（CA）（例如 Verisign）获取公钥证书。当客户端连接到 VPN 服务器时，服务器将使用证书来证明它是客户端预期的服务器。Web 中的 HTTPS 协议使用这种方式来认证 Web 服务器，确保客户端正在与预期的 Web 服务器通信，而不是伪造的 Web 服务器。

在本次实验中，MiniVPN 应该使用上述方法来认证 VPN 服务器。学生可以从头开始实施身份验证协议（如 TLS/SSL），幸运的是，openssl 已经为我们完成

了大部分工作。我们只需要正确配置我们的 TLS 会话，openssl 就可以自动为我们进行身份验证。

服务器身份认证有三个重要步骤：（1）验证服务器证书是否有效（2）验证服务器是证书的所有者（3）验证服务器是否是目标服务器（例如，如果用户打算访问 example.com，我们需要确保服务器确实是 example.com，而不是另一个站点）。请指出程序中代码的哪些行执行上述验证。在演示中，需要演示有关第三次验证的两种不同情况:“服务器是目标服务器”的服务器身份认证成功情况，以及“服务器不是预期服务器”的服务器身份认证失败的情况。

注意：MiniVPN 程序应该能够与不同机器上的 VPN 服务器通信，因此无法在程序中对 VPN 服务器的主机名进行指定，需要从命令行键入主机名。此名称代表用户的标志，因此应在验证中使用。此名称还应用于查找服务器的 IP 地址。5.2 节提供了一个示例程序，展示了如何获取给定主机名的 IP 地址。

示例 TLS 客户端和服务器程序：服务器身份认证在示例程序中实现。部分身份认证需要颁发服务器证书的 CA 证书。在./ca 客户端文件夹中放置了两个 CA 证书：一个是颁发服务器证书的 CA（服务器的主机名是 vpnlabserver.com），另一个是颁发 Google 证书的 CA。因此，示例 TLS 客户端程序可以与我们自己的服务器以及 Google 的 HTTPS 服务器通信：

```
$ ./tlsclient vpnlabserver.com 4433
$ ./tlsclient www.google.com 443
```

应该注意的是，学生不允许使用示例代码中的 vpnlabserver.com 作为他们的 VPN 服务器名称；为了区分学生的工作，要求学生生成自己的 CA，并在服务器证书名称中包含自己的姓名。生成证书的方法可以参考 5.3 节。

为了能够使用我们的客户端与 HTTPS 服务器通信，我们需要获取其 CA 证书，将证书保存在./ca 客户端文件夹中，并使用从其 subject 字段生成的哈希值创建指向它的符号链接（或重命名）。例如，为了使客户端能够与 Google 通信，从名为“GeoTrust Global CA”的根 CA 获取证书，我们从 Firefox 浏览器获取此根 CA 证书（GeoTrustGlobalCA.pem），并运行以下命令以获取它的哈希然后创建符号链接：

```
$ openssl x509 -in GeoTrustGlobalCA.pem -noout -subject_hash
2c543cd1
$ ln -s GeoTrustGlobalCA.pem 2c543cd1.0
$ ls -l
lrwxrwxrwx 1 ... 2c543cd1.0 -> GeoTrustGlobalCA.pem
lrwxrwxrwx 1 ... 9b58639a.0 -> cacert.pem
-rw-r--r-- 1 ... cacert.pem
-rw-r--r-- 1 ... GeoTrustGlobalCA.pem
```

4.5 任务 5：认证 VPN 客户端

访问专用网络内的计算机是一项权限，仅授予授权用户，而不是授予所有人。

因此，只允许授权用户与 VPN 服务器建立 VPN 隧道。在此任务中，授权用户是在 VPN 服务器上拥有有效帐户的用户。因此，我们将使用标准密码身份验证来验证用户身份。基本上，当用户尝试与 VPN 服务器建立 VPN 隧道时，将要求用户提供用户名和密码。服务器将检查其影子文件（/etc/shadow）；如果找到匹配的记录，则对用户进行认证，并建立 VPN 隧道。如果没有匹配，服务器将断开与用户的连接，因此不会建立隧道。有关如何使用 shadow 文件对用户进行身份验证的示例代码，请参见 5.5 节。

4.6 任务 6：支持多个客户端

在真实应用中，一个 VPN 服务器通常支持多个 VPN 隧道。也就是说，VPN 服务器允许多个客户端同时连接到它，每个客户端都有自己的 VPN 隧道（从而有自己的 TLS 会话）。MiniVPN 应该支持多个客户端。

在典型的实现中，VPN 服务器进程（父进程）将为每个隧道创建一个子进程（参见图 4）。当数据包来自隧道时，其相应的子进程将获取数据包，并将其转发到 TUN 接口。无论是否支持多个客户端，此方向都相同，另一个方向变得具有挑战性。当数据包到达 TUN 接口（来自专用网络）时，父进程将获取数据包，现在需要确定该数据包应该到达哪个隧道。你需要考虑如何实施这种决策逻辑。

一旦做出决定并选择了隧道，父进程就需要将数据包发送到附加了所选隧道的子进程。这需要 IPC（进程间通信）。典型的方法是使用管道。在 5.6 节中提供了一个示例程序，以演示如何使用 IPC 管道。

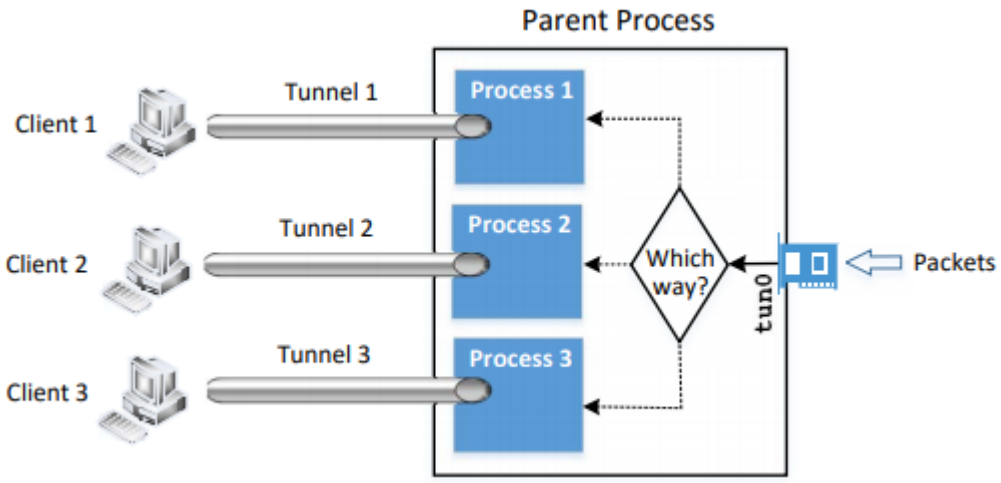

图 4 支持多个 VPN 客户端

子进程需要监视此管道接口，并在有数据时从中读取数据。由于子进程还需要注意来自套接字接口的数据，因此它们需要同时监视多个接口。第 5.7 节展示了如何实现这一目标。

5.8 节展示了一次一次完整的通过 VPN 进行 telnet 通信的过程。

5. 实验指南

5.1 在 Wireshark 中显示 TLS 流量

Wireshark 根据端口号识别 TLS/SSL 流量。它知道 443 是 HTTPS 的默认端口号，但我们的 VPN 服务器侦听不同的非标准端口号。我们需要让 Wireshark 知道这一点；否则，Wireshark 不会将我们的流量标记为 SSL/TLS 流量。以下是我们可以做的：转到 Wireshark 的 Edit 菜单，然后单击 Preferences，Protocols，HTTP，然后找到“SSL / TLS Ports”条目。添加 SSL 服务器端口。例如，我们可以将条目的内容更改为 443，4433，其中 4433 是我们的 SSL 服务器使用的端口。

显示解密的流量。上面显示的方法只能让 Wireshark 将流量识别为 TLS/SSL 流量；Wireshark 无法解密加密的流量。出于调试目的，我们希望看到解密的流量，Wireshark 提供了这样的功能。我们需要做的就是向 Wireshark 提供服务器的私钥，Wireshark 将自动从 TLS/SSL 握手协议派生会话密钥，并使用这些密钥解密流量。要向 Wireshark 提供服务器的私钥，请执行以下操作：

```
Click Edit -> Preferences -> Protocols -> SSL
Find the "RSA key list", and click the Edit button
Provide the required information about the server, see this example:
    IP Address: 10.0.2.65
    Port: 4433
    Protocol: ssl
    Key File: /home/seed/vpn/server-key.pem (privat key file)
    Password: deesdees
```

5.2 根据主机名获得 IP 地址

给定主机名，我们可以获取此名称的 IP 地址。在我们的示例 tlsclient 程序中，我们使用 gethostbyname()函数来获取 IP 地址。但是，此功能已过时，因为它不支持 IPV6，应用程序应该使用 getaddrinfo()代替。以下示例显示如何使用此函数获取 IP 地址。

```
#include <stdio.h>
#include <stdlib.h>
#include <netdb.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>

struct addrinfo hints, *result;

int main() {
    hints.ai_family = AF_INET; // AF_INET means IPv4 only addresses

    int error = getaddrinfo("www.example.com", NULL, &hints,
&result);
    if (error) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(error));
        exit(1);
    }

    // The result may contain a list of IP address; we take the first
one.
    struct sockaddr_in* ip = (struct sockaddr_in *) result->ai_addr;
    printf("IP Address: %s\n", (char *)inet_ntoa(ip->sin_addr));

    freeaddrinfo(result);
    return 0;
}
```

5.3 生成证书

使用 OpenSSL 来创建证书必须有一个配置文件。配置文件通常具有扩展名.cnf。它由三个 OpenSSL 命令使用：ca，req 和 x509。可以上网查找命令的具体使用方法，还可以从/usr/lib/ssl/openssl.cnf 获取配置文件的副本。将该文件直接复制到当前文件夹后，需要根据配置文件中的说明创建多个子目录（查看[CA default]部分）：

dir = ./demoCA # Where everything is kept
certs = $dir/certs # Where the issued certs are kept
crl_dir = $dir/crl # Where the issued crl are kept
new_certs_dir = $dir/newcerts # default place for new certs.
database = $dir/index.txt # database index file.
serial = $dir/serial # The current serial number

对于 index.txt 文件，只需创建一个空文件。对于序列号文件，在文件中输入

字符串格式的单个数字（例如 1000）。完成 openssl.cnf 的配置，就可以为 OpenSSL 涉及的三方：证书颁发机构（CA），服务器和客户端创建证书。

证书颁发机构（CA）。学生可以创建自己的 CA，然后使用此 CA 为服务器和用户颁发证书。我们将为 CA 创建一个自签名证书，这意味着此 CA 完全可信，其证书将作为根证书。学生可以运行以下命令为 CA 生成自签名证书：

$ openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf

系统会提示输入信息和密码。不要丢失此密码，因为每次使用此 CA 签署另一个证书时都必须键入密码。还需要填写一些信息，例如国家名称，通用名称等。命令的输出存储在两个文件中：ca.key 和 ca.crt。ca.key 文件包含 CA 的私钥，而 ca.crt 包含公钥证书。

服务器。现在我们有了自己的可信 CA，我们可以请求 CA 为服务器签发公钥证书。首先，我们需要创建一对公钥和私钥，在服务器端执行以下命令获得 RSA 密钥对，还需要我们自己提供一个密码来保护密钥，密钥将会存储在 server,key 文件里面。命令如下：

$ openssl genrsa -des3 -out server.key 1024

一旦拥有密钥文件，就可以生成证书签名请求（CSR）。CSR 将发送给 CA，CA 将为密钥生成证书（通常在确保 CSR 中的身份信息与服务器的真实身份匹配之后）。

$ openssl req -new -key server.key -out server.csr -config openssl.cnf

客户端。客户端可以按照以下相似的命令来生成 RSA 密钥对和 CSR：

$ openssl genrsa -des3 -out client.key 1024

$ openssl req -new -key client.key -out client.csr -config openssl.cnf

生成证书。CSR 文件需要有 CA 的签名才能形成证书。在实际应用中，CSR 文件通常会发送给受信任的 CA 进行签名。在本实验中，我们将使用我们自己的可信 CA 来生成证书：

$ openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key -config openssl.cnf

$ openssl ca -in client.csr -out client.crt -cert ca.crt -keyfile ca.key -config openssl.cnf

如果 OpenSSL 拒绝生成证书，可能是因为请求中的名称与 CA 的名称不匹配。匹配规则在配置文件中指定（查看[策略匹配]部分）。可以更改请求的名称以符合策略，也可以更改策略。配置文件还包含另一个策略（称为任意策略），该策略的限制较少。可以通过更改以下行来选择该策略：

"policy = policy_match" change to "policy = policy_anything"

5.4 SSL 套接字编程概要

本节简要介绍如何使用 openssl 的 API 实现一个简单的 SSL 客户端和服务端。

(1)初始化 SSL 库

```
SSL_library_init(); /* 为 SSL 加载加密和哈希算法 */
SSL_load_error_strings(); /* 为了更友好的报错，加载错误码的描述字符串 */
```

SSL_library_init 注册了所有在 SSL APIs 中的加密算法和哈希算法。SSL_library_init 正常情况只会返回 1。

SSL 应用程序需要调用 SSL_load_error_strings，该函数为 SSL 接口和 Crypto 加密接口加载错误描述字符串。

(2) 创建 SSL 上下文 SSL_CTX

```
SSL_METHOD *meth;
SSL_CTX *ctx;

meth = SSLv23_server_method();
ctx = SSL_CTX_new(meth);
```

首先选择 SSL/TLS 协议版本号；通过下面的 API 创建一个 SSL_METHOD 结构；SSL_METHOD 结构之后会用于通过 SSL_CTX_new()创建 SSL_CTX 结构。

对于每个 SSL/TSL 来说，有三种 APIs 可以用来创建一个 SSL_METHOD：一个可以用于服务端和客户端，一个只能用于服务端，另外一个只能由于客户端。SSLv2、SSLv3 以及 TLSv1 有着和协议名一致的接口函数名。注意并没有 SSLv23 这个协议号，SSLv23_method 将会选择 SSLv2、SSLv3 或者 TLSv1 来匹配对端的版本号。

协议号	通用接口	服务端专用	客户端专用
SSLv2	SSLv2_method()	SSLv2_server_ method()	SSLv2_client_ method()
SSLv3	SSLv3_method()	SSLv3_server_ method()	SSLv3_client_ method()
TLSv1	TLSv1_method()	TLSv1_server_ method()	TLSv1_client_ method()
SSLv23	SSLv23_method()	SSLv23_server_ method()	SSLv23_client_ method()

(3)设置证书及验证方式

```
SSL_CTX_use_certificate_file(SSL_CTX *ctx, const char *file, int
type); /* 加载用户证书 */

SSL_CTX_use_PrivateKey_file(SSL_CTX *ctx, const char *file, int
type); /* 加载用户私钥 */

SSL_CTX_load_verify_location(SSL_CTX *ctx, const char *Cafile, const
char *Capath); /* 指定 CA 证书及路径 */

int SSL_CTX_set_verify(SSL_CTX *ctx, int mode, int(*verify_callback),
int(X509_STORE_CTX *)); /* 指定证书验证方式 */
```

必须安装以及可选安装的证书如下：

服务端：服务器自己的证书(必须的)，CA 证书（可选的）

客户端：CA 证书（强制的），客户端自己的证书(可选的)

为了验证证书，首先需要加载 CA 证书（因为对端证书需要用 CA 证书来验证），SSL_CTX_load_verify_locations 的第一个参数 ctx，指向 一个用于加载 CA 证书的 SSL_CTX 结构，第二个参数和第三个参数 CAfile、CApatch，用于指向 CA 证书的路径；当查找 CA 证书时，Openssl 库先通过 CAfile 查找，找不到再通过 CApath。如果 CAfile 已经指定了证书(证书必须存在于 SSL 应用程序的相同路径下)，CApath 可以指定为 NULL。

如果要验证对端证书成功，对端的证书必须通过 CA 证书直接或者间接的方式的签名（存在一个正确的证书链）。SSL_CTX_set_verify 可以用于设置在 SSL_CTX 结构中的验证标记，第三个参数 verify_callback 可以设置一个指定验证过程的回调函数。（回调参数如果设置为 NULL 意味着用内建默认的验证方式）。SSL_CTX_set_verify 中的第二个参数可以指定以下宏：

SSL_VERIFY_NONE

SSL_VERIFY_PEER

SSL_VERIFY_FAIL_IF_NO_PEER_CERT

SSL_VERIFY_CLIENT_ONCE

其中 SSL_VERIFY_PEER 可以指定于客户端和服务端，用于启动验证。

(4)建立 SSL 套接字

```
SSL *SSl_new(SSL_CTX *ctx); // 申请一个 SSL 套接字
int SSL_set_fd(SSL *ssl, int fd);)  // 绑定读写套接字
int SSL_set_rfd(SSL *ssl, int fd);  // 绑定只读套接字
int SSL_set_wfd(SSL *ssl, int fd); // 绑定只写套接字

int SSL_connect(SSL *ssl);   //客户端发起连接

int SSL_accept(SSL *ssl); //服务端接受连接
```

(5)SSL 通信

```
int SSL_read(SSL *ssl, void *buf, int num);
int SSL_write(SSL *ssl, const void *buf, int num);
```

(6)关闭 SSL 套接字

```
int SSL_shutdown(SSL *ssl); // 关闭 SSL 套接字
void SSl_free(SSL *ssl); // 释放 SSL 套接字
void SSL_CTX_free(SSL_CTX *ctx);  // 释放 SSL 会话环境
```

5.5 使用 shadow 文件身份认证

以下程序显示如何使用存储在 shadow 文件中的帐户信息对用户进行身认证。该程序使用 getspnam()从 shadow 文件中获取给定用户的帐户信息，包括散列密码。然后，它使用 crypt()来散列给定的密码，并查看结果是否与从 shadow 文件中获取的值匹配。如果是，则用户名和密码匹配，并且验证成功。

```
#include <stdio.h>
#include <string.h>
#include <shadow.h>
#include <crypt.h>

int login(char *user, char *passwd)
{
    struct spwd *pw;
    char *epasswd;
    pw = getspnam(user);
    if (pw == NULL) {
        return -1;
    }

    printf("Login name: %s\n", pw->sp_namp);
    printf("Passwd : %s\n", pw->sp_pwdp);

    epasswd = crypt(passwd, pw->sp_pwdp);
    if (strcmp(epasswd, pw->sp_pwdp)) {
        return -1;
    }
    return 1;
}
void main(int argc, char** argv)
{
    if (argc < 3) {
        printf("Please provide a user name and a password\n");
        return;
    }

    int r = login(argv[1], argv[2]);
    printf("Result: %d\n", r);
}
```

我们可以编译上面的代码并使用用户名和密码运行它。应该注意，从 shadow

文件读取时需要 root 权限。请参阅以下命令以进行编译和执行。

```
$ gcc login.c -lcrypt
$ sudo ./a.out seed dees
```

需要注意的是，我们在上面的编译中使用了-lcrypt，而在编译 TLS 程序时使用了-lcrypto。crypt 和 crypto 是两个不同的库，所以这不是一个错字。

5.6 使用 Pipe 进程间通信

以下程序显示父进程如何使用管道将数据发送到其子进程。在 Line①父进程使用 Line pipe 中的 pipe()创建管道。每个管道有两端：输入端的文件描述符是 fd [0]，输出端的文件描述符是 fd [1]。

创建管道后，使用 fork()生成子进程。父进程和子进程都具有与管道关联的文件描述符。他们可以使用管道发送数据，这是双向的。但是，我们只使用此管道将数据从父进程发送到子进程，而父进程不会从管道中读取任何内容，因此我们关闭父进程中的输入端 fd [0]。类似地，子进程不会通过管道发送任何内容，因此它会关闭输出端 fd [1]。此时，我们已经建立了从父进程到子进程的单向管道。要通过管道发送数据，父进程写入 fd [1]（参见 Line②）;为了从管道接收数据，子进程从 fd [0]读取（参见 Line③）。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main(void)
{
    int fd[2], nbytes;
    pid_t pid;
    char string[] = "Hello, world!\n";
    char readbuffer[80];

    pipe(fd);  ①

    if((pid = fork()) == -1) {
        perror("fork");
        exit(1);
    }
    if(pid>0) { //parent process
        close(fd[0]); // Close the input end of the pipe.

        // Write data to the pipe.
        write(fd[1], string, (strlen(string)+1));  ②
```

```
        exit(0);
    } else { //child process
        close(fd[1]); // Close the output end of the pipe.

        // Read data from the pipe.
        nbytes = read(fd[0], readbuffer, sizeof(readbuffer));  ③
        printf("Child process received string: %s", readbuffer);
    }
    return(0);
}
```

5.7 使用 select 监听多输入接口

我们的 VPN 程序需要监控多个接口，包括 TUN 接口，套接字接口，有时还有管道接口。所有这些接口都由文件描述符表示，因此我们需要监视它们以查看是否有来自它们的数据。一种方法是继续轮询它们，并查看每个接口上是否有数据。这种方法的性能是不可取的，因为当没有数据时，进程必须在空闲循环中继续运行。另一种方法是从界面读取。默认情况下，read 是阻塞的，即如果没有数据，则将暂停进程。当数据可用时，该过程将被解除阻塞，并且其执行将继续。这样，在没有数据时不会浪费 CPU 时间。

基于读取的阻塞机制适用于一个接口。如果进程正在等待多个接口，则它不能仅阻塞其中一个接口。它必须完全阻止所有这些。 Linux 有一个名为 select() 的系统调用，它允许程序同时监视多个文件描述符。要使用 select()，我们需要使用 FD SET 宏将所有要监视的文件描述符存储在一个集合中（参见下面代码中的行①和②）。然后我□将□集合□予select()系统调用（Line③），□将阻止□□程，直到数据在集合中的一个文件描述符上可用。然后，我们可以使用 FD ISSET 宏来确定哪个文件描述符已接收数据。在下面的代码示例中，我们使用 select()来监视 TUN 和套接字文件描述符。

```
fd_set readFDSet;
int ret, sockfd, tunfd;

FD_ZERO(&readFDSet);
FD_SET(sockfd, &readFDSet);  ①
FD_SET(tunfd, &readFDSet);  ②
ret = select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL); ③

if (FD_ISSET(sockfd, &readFDSet){
    // Read data from sockfd, and do something.
}
if (FD_ISSET(tunfd, &readFDSet){
    // Read data from tunfd, and do something.
}
```

5.8 示例：在 VPN 中使用 telnet 的一次完整通信过程

为帮助大家理解应用程序中的数据包如何通过我们的 MiniVPN 流向目的地，我们用两张图来说明当用户从主机上运行“telnet 10.0.20.100”时的完整数据包流路径，该主机是主机到网关的 VPN 中的 A 点。VPN 的另一端位于网关上，该网关连接到我们的 telnet 服务器 10.0.20.100 所在的 10.0.20.0/24 网络。

图 5 显示了数据包如何从 telnet 客户端流向服务器。图 6 显示了数据包如何从 telnet 服务器返回到客户端。我们只描述图 5 中的路径，一旦你理解了图 5 中的路径，图 6 的返回路径就很好理解了。

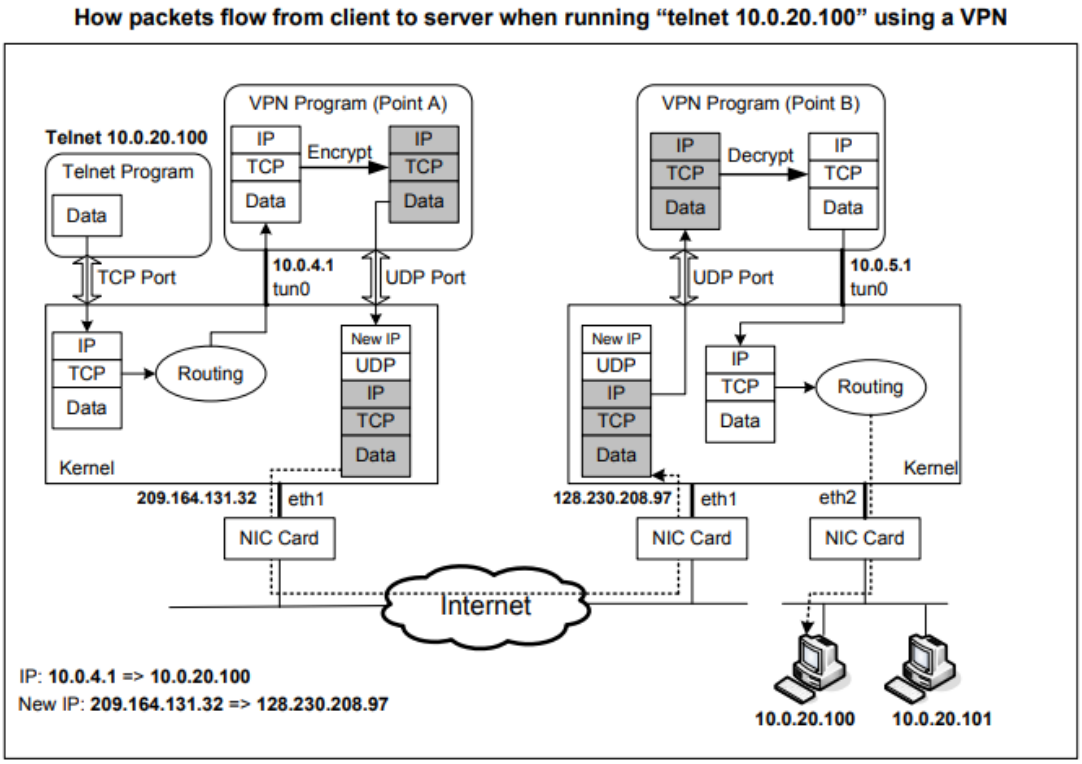

图 5 telnet 数据包通过主机到网关的 VPN 从客户端到服务器的流经路径

1) telnet 程序发出数据
2) 内核使用 10.0.20.100 作为目的地址构造 IP 数据包
3) 内核需要确定使用 eth1 或者 tun0 作为网络接口来发送数据包，这需要学生提前设置好路由表，让内核使用 tun0 来路由数据包，内核相应的会使用 tun0 的 IP 地址作为源 IP 地址。
4) IP 数据包通过虚拟接口 tun0 来到了我们的 VPN（A 点）程序，之后数据包会被加密再通过 UDP 端口发送给内核，这次不再通过 tun0 接口，因为 VPN 程序使用 UDP 作为通信隧道。
5) 内核把加密后的 IP 数据包看作 UDP 数据部分，构架新的 IP 包。新的目的 IP 地址会变成我们设置的 VPN 的另一端的 IP 地址，图中目的 IP 为 128.230.208.97。
6) 再次强调正确配置路由表，才能保证新的数据包会被路由到 eth1 接口，该数据包的源 IP 设置为 209.164.131.32
7) 该数据包会在 Internet 传递，负载着被加密的 telnet 数据包，这就是为什么我们称之为隧道。
8) 通过 eth1 接口，数据包到达网关 128.230.208.97。
9) UDP 数据包的的数据部分会通过 UDP 端口传递给正在等待的 VPN 程序（B 点）。
10) VPN 程序将解密有效负载，然后通过虚拟网络接口 tun0 将解密后的有效负载（原始 telnet 数据包）传递给内核。
11) 由于它来自网络接口，内核会将其视为 IP 数据包，查看其目标 IP 地址，并决定将其路由到何处。请记住，此数据包的目标 IP 地址是 10.0.20.100。如果你的路由表设置正确，则应该通过 eth2 路由数据包，因为这是连接

到 10.0.20.0/24 网络的接口。

12) telnet 数据包将会传递给最终的目的地址 10.0.20.100

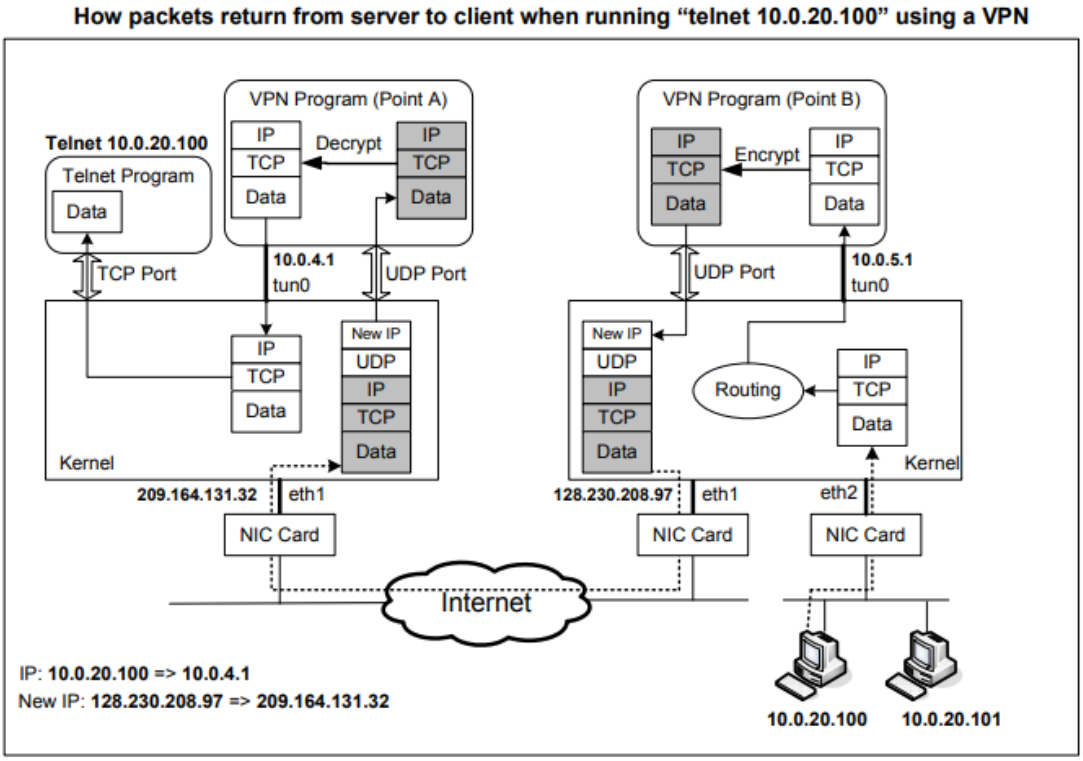

图 6 telnet 数据包从服务器到客户端的流经路径

6. 实验报告

学生需要提交完整详细的实验报告，包含系统设计和实现的所有过程，还要包括如何测试系统功能和系统安全的过程，并为验收的老师或助教演示系统。