
实验三 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：

- 1) 虚拟专用网络；
- 2) TUN/TAP 和 IP 隧道；
- 3) 路由；
- 4) 公钥加密，PKI 和 X.509 证书；
- 5) TLS/SSL 编程；
- 6) 身份认证。

2 实验环境

2.1 系统平台

操作系统：Windows 10 家庭中文版 20H2

处理器：Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz

内存 RAM：16.0 GB (15.8 GB 可用)

系统类型：64 位操作系统，基于 x64 的处理器

2.2 软件工具

VMware® Workstation 16 Pro 16.1.1 build-17801498

SEEDUbuntu 16.04
Docker 1.10.3 build 20f81dd
OpenSSL 1.0.2g

2.3 网络拓扑与配置

客户端穿过 Internet 环境和 VPN 网关建立隧道连接,VPN 网关和内网主机之间形成同一局域网,具体拓扑结果如下图所示:

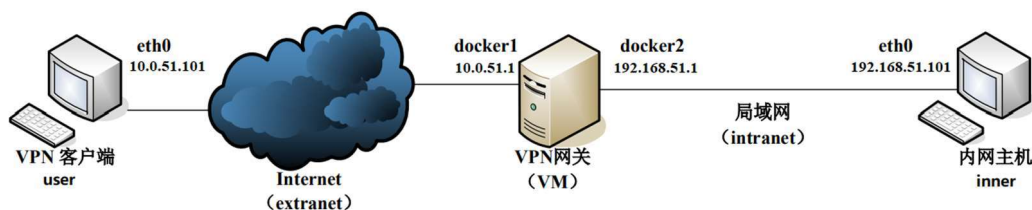


图 2-1 网络拓扑与配置图

同时,利用 TUN 来建立一个主机到网关的传输隧道,具体数据传输过程如下图所示:

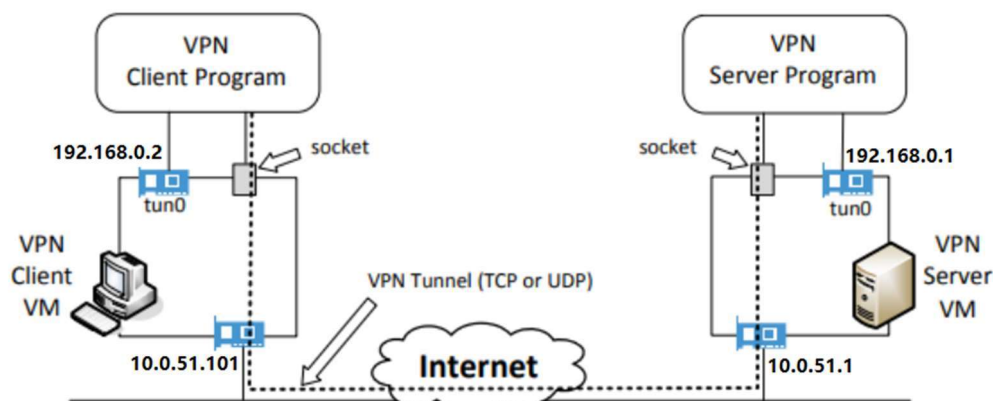


图 2-2 TUN 隧道传输图

3 实验内容

熟悉 TLS/SSL PN 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。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 MniVPN。

4 实验步骤及结果分析

4.1 环境配置

在实际环境中，VPN 客户端（user）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 inner 是内部局域网的计算机。Internet 主机 user 上的用户希望通过 VPN 隧道与内部局域网的主机 inner 通信。为模拟此设置，我们使用 docker 网络“intranet”将 inner 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，inner 不能直接从 Internet 访问，即不能直接从 user 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
docker network create --subnet=10.0.51.0/24 --gateway=10.0.51.1 --opt "com.docker.network.bridge.name"="docker1" extranet
# 在VM上创建docker网络 intranet
docker network create --subnet=192.168.51.0/24 --gateway=192.168.51.1 --opt "com.docker.network.bridge.name"="docker2" intranet
# 在VM上创建并运行容器 user
docker run -it --name=user --hostname=user --net=extranet --ip=10.0.51.101 --privileged "seedubuntu" /bin/bash
# 在VM上创建并运行容器 inner
docker run -it --name=inner --hostname=inner --net=intranet --ip=192.168.51.101 --privileged "seedubuntu" /bin/bash
# 在容器 user 和 inner 内分别删除默认路由
route del default
```

4.2 使用 TUN/TAP 创建一个主机到主机的隧道

使用如下命令编译 VPN 客户端和服务端程序：

```
gcc -o vpnclient vpnclient.c
gcc -o vpnserver vpnserver.c
```

4.2.1 运行 VPN 服务端

启动 VPN 服务程序 vpnserver 并配置虚拟 TUN 网络接口，命令如下：

```
# 在VM上启动VPN服务
sudo ./vpnserver
# 在VM上的另一个终端配置tun0虚拟IP地址并激活接口
sudo ifconfig tun0 192.168.0.1/24 up
```

为使 SEEDUbuntu 发挥网关的作用，启动其 IP 转发功能；同时为防止其 iptables 的规则会阻断转发报文，清空 iptables 规则，命令如下：

```
# 在VM上启用IP转发
sudo sysctl net.ipv4.ip_forward=1
# 在VM上清除iptables规则
sudo iptables -F
```

4.2.2 运行 VPN 客户端

首先将客户端程序 vpnclient 拷贝至 user 中，然后启动该程序并配置虚拟 TUN 网络接口，命令如下：

```
# 在VM上拷贝VPN客户端
docker cp vpnclient user:/vpnclient
# 在容器user中启动VPN客户端
./vpnclient 10.0.51.1
# 在容器user的另一个终端配置tun0虚拟IP地址并激活接口
ifconfig tun0 192.168.0.2/24 up
```

4.2.3 在 user 上设置路由

在上述两步骤完成后将成功建立通讯隧道。为访问专用网络并指示通过隧道的预期流量，在 user 上将所有需要进入专用网络 intranet 的数据包都定向到 tun0 接口，从该接口可通过 VPN 隧道转发数据包。命令如下：

```
# 在user上创建隧道路由
route add -net 192.168.51.0/24 tun0
```

4.2.4 在 inner 上设置路由

为保证当 inner 回复从 user 发送的数据包时，该数据包经由 VPN 服务器后可以成功被送到 VPN 隧道的另一端，在 user 中使用“ping 192.168.51.101”命令后用 Wireshark 进行抓包，以判读源 IP 地址信息，如下：

No.	Time	Source	Destination	Protocol	Length	Info
1	2021-05-05 07:43:04.013815986	192.168.0.2	192.168.51.101	ICMP	98	Echo (ping) request id=0x000e, seq=1/256, ttl=63 (request in 2)
2	2021-05-05 07:43:04.013848924	192.168.51.101	192.168.0.2	ICMP	98	Echo (ping) reply id=0x000e, seq=1/256, ttl=64 (request in 1)
3	2021-05-05 07:43:05.046515730	192.168.0.2	192.168.51.101	ICMP	98	Echo (ping) request id=0x000e, seq=2/512, ttl=63 (reply in 4)
4	2021-05-05 07:43:05.046599827	192.168.51.101	192.168.0.2	ICMP	98	Echo (ping) reply id=0x000e, seq=2/512, ttl=64 (request in 3)
5	2021-05-05 07:43:06.071139475	192.168.0.2	192.168.51.101	ICMP	98	Echo (ping) request id=0x000e, seq=3/768, ttl=63 (reply in 6)
6	2021-05-05 07:43:06.071422531	192.168.51.101	192.168.0.2	ICMP	98	Echo (ping) reply id=0x000e, seq=3/768, ttl=64 (request in 5)
7	2021-05-05 07:43:07.094147050	192.168.0.2	192.168.51.101	ICMP	98	Echo (ping) request id=0x000e, seq=4/1024, ttl=63 (reply in 8)
8	2021-05-05 07:43:07.094479731	192.168.51.101	192.168.0.2	ICMP	98	Echo (ping) reply id=0x000e, seq=4/1024, ttl=64 (request in 7)
9	2021-05-05 07:43:08.118076037	192.168.0.2	192.168.51.101	ICMP	98	Echo (ping) request id=0x000e, seq=5/1280, ttl=63 (reply in 10)
10	2021-05-05 07:43:08.119288982	192.168.51.101	192.168.0.2	ICMP	98	Echo (ping) reply id=0x000e, seq=5/1280, ttl=64 (request in 9)
11	2021-05-05 07:43:09.173297939	02:42:b8:84:ca:82	02:42:c8:a8:33:65	ARP	42	Who has 192.168.51.101? Tell 192.168.51.1
12	2021-05-05 07:43:09.173294781	02:42:c8:a8:33:65	02:42:b8:84:ca:82	ARP	42	Who has 192.168.51.1? Tell 192.168.51.101
13	2021-05-05 07:43:09.17342899	02:42:b8:84:ca:82	02:42:c8:a8:33:65	ARP	42	192.168.51.1 is at 02:42:b8:84:ca:82
14	2021-05-05 07:43:09.17346876	02:42:c8:a8:33:65	02:42:b8:84:ca:82	ARP	42	192.168.51.101 is at 02:42:c8:a8:33:65

图 4-1 Wireshark 抓包图

观察到成功回复包，这是由于当路由表中不存在相关信息时，会直接向网关（即 VPN 服务器）询问，此时根据 VPN 服务器的路由转发表，正在运行的 VPN 服务端会将数据包进行处理并发回查询源地址。

4.2.5 测试 VPN 隧道

完成上述设置后，使用 ping 命令测试通过隧道从 user 访问 inner，如下：

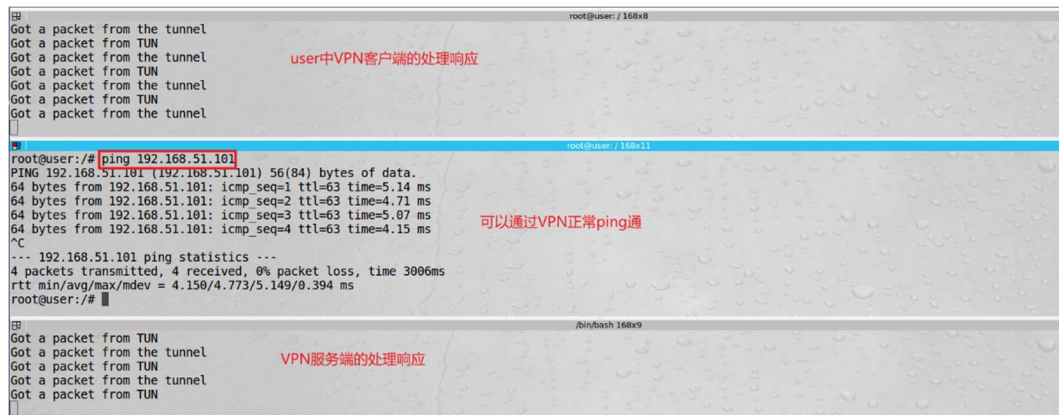
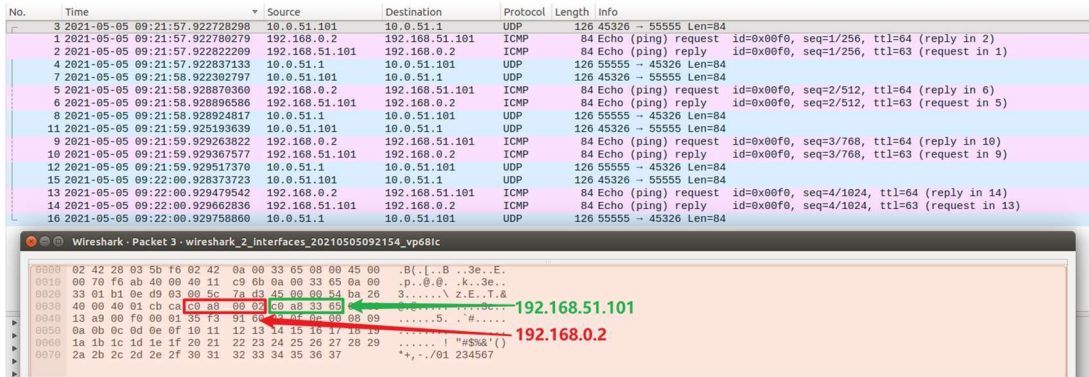


图 4-2 ping 命令测试隧道图

对 user 所在的 tun0 和 docker1 网卡进行抓包，得到如下结果：



No.	Time	Source	Destination	Protocol	Length	Info
3	2021-05-05 09:21:57.922728298	10.0.51.1	10.0.51.1	UDP	126	45326 → 55555 Len=84
1	2021-05-05 09:21:57.922780279	192.168.0.2	192.168.51.101	ICMP	84	Echo (ping) request id=0x00f0, seq=1/256, ttl=64 (reply in 2)
2	2021-05-05 09:21:57.922822209	192.168.51.101	192.168.0.2	ICMP	84	Echo (ping) reply id=0x00f0, seq=1/256, ttl=63 (request in 1)
4	2021-05-05 09:21:57.922837133	10.0.51.1	10.0.51.1	UDP	126	55555 → 45326 Len=84
7	2021-05-05 09:21:58.922302797	10.0.51.1	10.0.51.1	UDP	126	45326 → 55555 Len=84
5	2021-05-05 09:21:58.928870360	192.168.0.2	192.168.51.101	ICMP	84	Echo (ping) request id=0x00f0, seq=2/512, ttl=64 (reply in 6)
6	2021-05-05 09:21:58.928896586	192.168.51.101	192.168.0.2	ICMP	84	Echo (ping) reply id=0x00f0, seq=2/512, ttl=63 (request in 5)
8	2021-05-05 09:21:58.928924817	10.0.51.1	10.0.51.1	UDP	126	55555 → 45326 Len=84
11	2021-05-05 09:21:59.925193930	10.0.51.1	10.0.51.1	UDP	126	45326 → 55555 Len=84
9	2021-05-05 09:21:59.929263822	192.168.0.2	192.168.51.101	ICMP	84	Echo (ping) request id=0x00f0, seq=3/768, ttl=64 (reply in 10)
10	2021-05-05 09:21:59.929367577	192.168.51.101	192.168.0.2	ICMP	84	Echo (ping) reply id=0x00f0, seq=3/768, ttl=63 (request in 9)
12	2021-05-05 09:21:59.929517370	10.0.51.1	10.0.51.1	UDP	126	55555 → 45326 Len=84
15	2021-05-05 09:22:00.928373723	10.0.51.1	10.0.51.1	UDP	126	45326 → 55555 Len=84
13	2021-05-05 09:22:00.929479542	192.168.0.2	192.168.51.101	ICMP	84	Echo (ping) request id=0x00f0, seq=4/1024, ttl=64 (reply in 14)
14	2021-05-05 09:22:00.929662836	192.168.51.101	192.168.0.2	ICMP	84	Echo (ping) reply id=0x00f0, seq=4/1024, ttl=63 (request in 13)
16	2021-05-05 09:22:00.929758860	10.0.51.1	10.0.51.1	UDP	126	55555 → 45326 Len=84

图 4-3 tun0 抓包结果图

观察到隧道的端口信息被封装在 UDP 数据包内，途中粉色的 ICMP 包即为经过隧道传输的数据包，其余则是非隧道流量。

4.2.6 Tunnel 断开测试

具体见第五节思考题部分。

4.3 加密隧道

该部分开始对 MiniVPN 进行验证，首先使用如下命令编译并将其拷贝至 user 中以便客户端使用。命令如下：

```
make && docker cp ../MiniVPN user:/workspace
```

同时，由于服务器证书中的名称是“lyg.com”，客户端使用的主机名应该与这个通用名称相匹配，否则 TLS 连接将失败（这个检查在客户端程序中实现）。因此使用如下命令在 user 的“/etc/hosts”文件中添加该域名信息，以便将“lyg.com”这个域名映射到对应服务器的 IP 地址：

```
sudo echo "10.0.51.1 lyg.com" >> /etc/hosts
```

随后在 VM 中运行 TLS VPN 服务端，命令如下：

```
sudo ./svpnserver
```

接着在 user 上运行 TLS VPN 客户端，其中服务器监听端口为 4433，使用用户名 seed 和密码 dees 登录，客户端隧道端口 IP 的最后一位为 2。命令如下：

```
sudo ./svpnclient lyg.com 4433 seed dees 2
```

使用 Wireshark 进行抓包，得到如下结果：



图 4-4 报文加密结果图

观察到证书认证完成后所传输的数据已被加密。

4.4 认证 VPN 服务器

4.4.1 为服务器生成证书

首先复制 “/usr/lib/ssl/openssl.cnf” 文件至服务器证书所在目录下，并查看该文件，如下图：

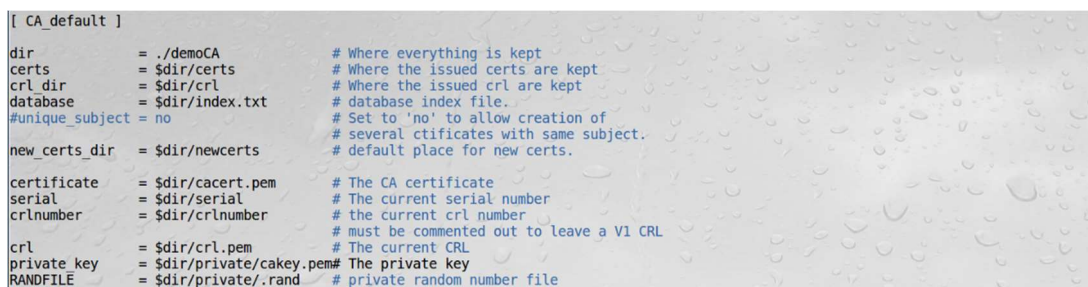


图 4-5 openssl.cnf 图

根据上述信息创建相应的文件及目录，并对这些文件进行配置，具体为：保持 “index.txt” 文件为空，在 serial 文件中输入一个字符串格式的数字（如 1000）。命令如下：



图 4-6 创建对应文件与文件夹

接着为 CA 创建一个自签名的证书，CA 私钥存储于文件“ca-lyg-key.pem”中，CA 公钥证书存储于文件 “ca-lyg-crt.pem” 中。具体命令如下：

```
openssl req -new -x509 -keyout ca-lyg-key.pem -out ca-lyg-crt.pem -config openssl.cnf
```

该 CA 签名得到的证书具体配置信息如下图所示：

```
[05/08/21]seed@VM:~/../cert_server$ openssl req -new -x509 -keyout ca-key-lyg.pem -out ca-lyg-crt.pem -config openssl.cnf
Generating a 2048 bit RSA private key
.....+++
writing new private key to 'ca-key-lyg.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:CN
State or Province Name (full name) [Some-State]:Hubei
Locality Name (eg, city) []:Wuhan
Organization Name (eg, company) [Internet Widgits Pty Ltd]:HUST
Organizational Unit Name (eg, section) []:HUST
Common Name (e.g. server FQDN or YOUR name) []:YigeLIU
Email Address []:u201814851@mail.hust.edu.cn
[05/08/21]seed@VM:~/../cert_server$
```

图 4-7 CA 证书配置信息图

对于服务端，首先获得一个 RSA 密钥对，该密钥对存储与“server-lyg-key.pem”文件中。同时，利用创建的 CA 证书为该密钥文件生成证书。命令如下：

```
openssl genrsa -des3 -out server-lyg-key.pem 1024
openssl req -new -key server-lyg-key.pem -out server-lyg-csr.pem -config openssl.cnf
```

最后，使用创建的 CA 来生成证书，命令及结果如下：

```
openssl ca -in server-lyg-csr.pem -out server-lyg-crt.pem -cert ca-lyg-crt.pem -keyfile ca-lyg-key.pem -config openssl.cnf
```

```
[05/20/21]seed@VM:~/../cert_server$ openssl ca -in server-lyg-csr.pem -out server-lyg-crt.pem -cert ca-lyg-crt.pem -keyfile ca-lyg-key.pem -config openssl.cnf
Using configuration from openssl.cnf
Enter pass phrase for ca-lyg-key.pem:
Check that the request matches the signature
Signature ok
Certificate Details:
  Serial Number: 4096 (0x1000)
  Validity
    Not Before: May 19 23:47:03 2021 GMT
    Not After : May 19 23:47:03 2022 GMT
  Subject
    countryName           = CN
    stateOrProvinceName   = Hubei
    organizationName      = HUST
    organizationalUnitName = HUST
    commonName             = lyg.com
    emailAddress           = u201814851@mail.hust.edu.cn
  X509v3 extensions:
    X509v3 Basic Constraints:
      CA:FALSE
    Netscape Comment:
      OpenSSL Generated Certificate
    X509v3 Subject Key Identifier:
      EB:42:30:B4:18:5B:3E:31:E9:4C:C3:F2:DA:B8:27:F0:E8:36:03:58
    X509v3 Authority Key Identifier:
      keyid:7E:58:9B:D4:35:06:27:CA:68:97:13:93:D6:31:99:CB:9B:62:37:A1
Certificate is to be certified until May 19 23:47:03 2022 GMT (365 days)
Sign the certificate? [y/n]:y

1 out of 1 certificate requests certified, commit? [y/n]:y
Write out database with 1 new entries
Data Base Updated
[05/20/21]seed@VM:~/../cert_server$
```

图 4-8 自签名证书结果图

当完成对服务器证书的生成后，给客户端创建一个用于 VPN 服务器身份认证的目录“ca_client”，并将给服务器颁发证书的 CA 证书拷贝至该目录下，如下图：

```
[05/08/21]seed@VM:~/../VPNS$ mkdir ca_client
[05/08/21]seed@VM:~/../VPNS$ cp cert_server/ca-lyg-crt.pem ca_client/
[05/08/21]seed@VM:~/../VPNS$
```

图 4-9 客户端验证证书转存图

随后使用从该证书“subject”字段生成的哈希值创建指向它的符号链接，命令及结果如下：

```
[05/21/21]seed@VM:~/.../MiniVPN$ c_rehash ca_client
Doing ca client
[05/21/21]seed@VM:~/.../MiniVPN$ ll ca_client/
total 4
lrwxrwxrwx 1 seed seed 14 May 21 05:50 a36e721b.0 -> ca-lyg-crt.pem
lrwxrwxrwx 1 seed seed 14 May 21 05:50 c43e383f.0 -> ca-lyg-crt.pem
-rw-rw-r-- 1 seed seed 1415 May 20 07:49 ca-lyg-crt.pem
[05/21/21]seed@VM:~/.../MiniVPN$
```

图 4-10 符号链接生成图

4.4.2 服务器身份认证代码

```
SSL* setupTLSclient(const char* hostname) {
    // Step 0: OpenSSL library initialization
    // This step is no longer needed as of version 1.1.0.
    SSL_library_init();
    SSL_load_error_strings();
    SSL_load_error_strings();
    SSL_load_error_strings();

    SSL_METHOD *meth;
    SSL_CTX *ctx;
    SSL *ssl;

    meth = (SSL_METHOD *)TLSv1_2_method();
    ctx = SSL_CTX_new(meth);

    SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER, verify_callback);
    if (SSL_CTX_load_verify_locations(ctx, NULL, CA_DIR) < 1) {
        printf("[ERROR] Error setting the verify locations. \n");
        exit(0);
    }
    ssl = SSL_new(ctx);

    X509_VERIFY_PARAM *vpm = SSL_get0_param(ssl);
    X509_VERIFY_PARAM_set1_host(vpm, hostname, 0);

    return ssl;
}
```

客户端采用“SSL_CTX_load_verify_locations()”函数利用目录“ca_client”下的给函数签名的证书完成对服务器的认证，该函数会在指定目录下以证书 hash 值的方式查找目标证书并利用该证书数据进行验证。当发起 SSL 连接时会根据指定的 SSL 信息完成验证过程。

4.4.3 测试

运行 VPN 客户端和服务端，利用回调函数输出 SSL 验证过程中使用的证书信息，观察到成功验证服务端，如下：

```
root@user:/workspace/tls# sudo ./tlsclient lyg.com 4433 seed dees 2
[INFO] Set up TLS client successfully!
[INFO] Set up TCP client successfully!
subject= /C=CN/ST=Hubei/L=Wuhan/O=HUST/OU=HUST/CN=YigeLIU/emailAddress=u201814851@mail.hust.edu.cn
[INFO] Verification passed.
subject= /C=CN/ST=Hubei/O=HUST/OU=HUST/CN=lyg.com/emailAddress=u201814851@mail.hust.edu.cn
[INFO] Verification passed.
[INFO] SSL connection is successful
[INFO] SSL connection using AES256-GCM-SHA384
```

图 4-11 验证服务器结果图

4.5 认证 VPN 客户端

4.5.1 使用 shadow 文件身份认证

当服务器对客户端进行身份认证时，会使用客户端发来的用户名与密码信息，将其与服务器上的“/etc/shadow”文件中的用户名和密码进行对比，具体为：

1. 首先使用“getspnam(user_name)”函数来得到用户名对应的加密后的身份信息结构体，当用户不存在时会返回 NULL；
2. 接着使用“crypt()”函数对客户端发来的密码进行同类型加密，然后再与“/etc/shadow”文件中的结果进行对比，来判断用户名所对应的密码是否正确。

其实现代码如下：

```
int login(char *user, char *passwd) {
    struct spwd *pw;
    char *epasswd;
    pw = getspnam(user);
    if (pw == NULL) {
        printf("[ERROR] Password is NULL.\n");
        return 0;
    }

    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)) {
        printf("[ERROR] Incorrect password.\n");
        return 0;
    }
    return 1;
}
```

4.5.2 测试

运行 VPN 和客户端，利用代码中打印出的提示信息跟踪认证过程，观察到成功认证客户端，如下：



```
[05/21/21]seed@VM:~/.../tls$ sudo ./tlsserver
[sudo] password for seed:
Enter PEM pass phrase:
net.ipv4.ip_forward = 1
[INFO] SSL connection established!
Login name: seed
Passwd      : $6$wDRrWCQz$IsBXp9.9wz9SGrF.nbihoN5w.zQx02sht4cTY8qI7YKh00wN/sfYvDeCAcEo2QYzCfpZoaEVJ8sbCT7hkxXY/
[INFO] Login successfully!
```

图 4-12 验证客户端结果图

4.6 支持多个客户端

4.6.1 利用 pipe 和多进程实现对多客户端的支持

由于多客户端经过服务器向内网主机发送报文时，来自不同客户端的报文并不会影响最终的转发处理，因此对有客户端发出的报文不需要进行额外处理。但是对于发回客户端的报文，由于每个客户端和服务器间都建立了不同的 SSL 连接，因此当报文经过服务器后必须要经过对应的 SSL 将数据重新发回对应的客户端，才能完成报文的成功转发。具体情况如下图所示：

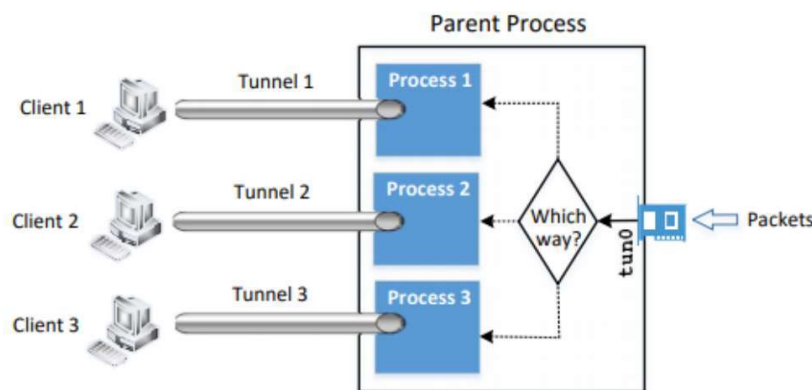


图 4-13 多客户端数据处理图

综上，为使服务端能够支持多个客户端的通讯连接，使用 pipe 技术来完成对不同客户端对应消息的定向转发，具体实现为：

- 1) 当客户端发起连接后，首先创建一个子进程，然后根据 tun 隧道中的虚拟 IP 地址（192.18.53.x）最后一字节 x 的值在“./pipe”文件夹下创建对应名为 x 的 pipe 文件；
- 2) 利用先前建立好的 tun 监听线程对 tun 进行监听，当收到报文时，仅对发回客户端的报文进行处理，而对客户端向外发送的报文仅做转发处理。随后根据报文中的目标 IP 地址最后一字节信息找到对应的 pipe 文件，使用 write() 函数向其中写入监听到的完整数据信息，该步骤依靠 listen_tun() 函数完成，代码如下：

```
void *listen_tun(void *tunfd) {
    int fd = *((int *)tunfd);
    char buff[2000];
    while (1) {
        int len = read(fd, buff, 2000);
        if (len > 19 && buff[0] == 0x45) {
            printf("Receive TUN: len = %d | ip.des = 192.168.53.%d\n", len, (int)buff[19]);
            char pipe_file[10];
            sprintf(pipe_file, "./pipe/%d", (int)buff[19]);
            int fd = open(pipe_file, O_WRONLY);
            if (fd == -1) {
                printf("[WARN] File %s is not exist.\n", pipe_file);
            } else {
                write(fd, buff, len);
            }
        }
    }
}
```

图 4-14 listen_tun() 函数代码图

- 3) 对创建好的 pipe 文件，利用 PIPEDATA 结构体将对应 pipe 文件描述符和其

SSL 连接进行封装，以便在 SSL 连接和 pipe 文件间建立关系，方便后续转发的使用，代码如下：

```
pthread_t listen_pipe_thread;
PIPEDATA pipeData;
pipeData.pipe_file = pipe_file;
pipeData.ssl = ssl;
pthread_create(&listen_pipe_thread, NULL, listen_pipe, (void *)&pipeData);
sendOut(ssl, sock, tunfd);
pthread_cancel(listen_pipe_thread);
remove(pipe_file);
```

图 4-15 pipe 文件与 SSL 封装代码图

该代码中的 sendOut()函数的主要作用是将从客户端向外发送写入 tun 中，以便从服务器的虚拟 tun 接口将数据转发到目的 IP 地址，该函数的代码具体如下：

```
void sendOut(SSL *ssl, int sock, int tunfd) {
    int len;
    do {
        char buf[1024];
        len = SSL_read(ssl, buf, sizeof(buf) - 1);
        write(tunfd, buf, len);
        buf[len] = '\0';
        printf("Received SSL: %d\n", len);
    } while (len > 0);
    printf("[INFO] SSL shutdown.\n");
}
```

图 4-16 sendOut()函数代码图

- 4) 利用封装的结构体获取 pipe 文件描述符，并创建对指定 pipe 的监听函数，当 pipe 中有数据时则读出并使用对应的 SSL 连接向指定的客户端发送，该步骤依靠 listen_pipe()函数完成，代码如下：

```
void *listen_pipe(void *threadData) {
    PIPEDATA *ptd = (PIPEDATA*)threadData;
    int pipefd = open(ptd->pipe_file, O_RDONLY);

    char buff[2000];
    int len;
    do {
        len = read(pipefd, buff, 2000);
        SSL_write(ptd->ssl, buff, len);
    } while (len > 0);
    printf("%s read 0 byte. Close connection and remove file.\n", ptd->pipe_file);
    remove(ptd->pipe_file);
}
```

图 4-17 listen_pipe()函数代码图

4.6.2 测试

为支持多客户端的测试，新创建一个容器 user2 来运行第二个客户端，命令如下：

```
# 在VM上创建并运行容器 user2
docker run -it --name=user2 --hostname=user2 --net=extranet --ip=10.0.51.102 --privileged "seedubuntu" /bin/bash
```

随后在 user 和 user2 上启动客户端，观察到两个客户端均可以通过 VPN 正常连接内网 inner，如下图：

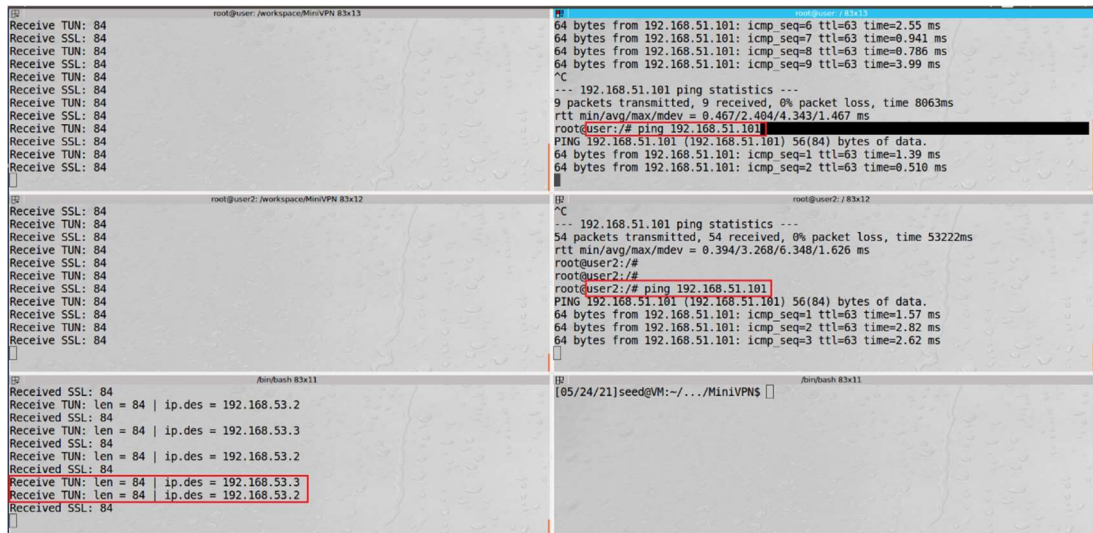


图 4-18 多客户端测试结果图

同时使用 Wireshark 进行抓包，观察到两个客户端的数据在不同的隧道内进行传输，如下图：

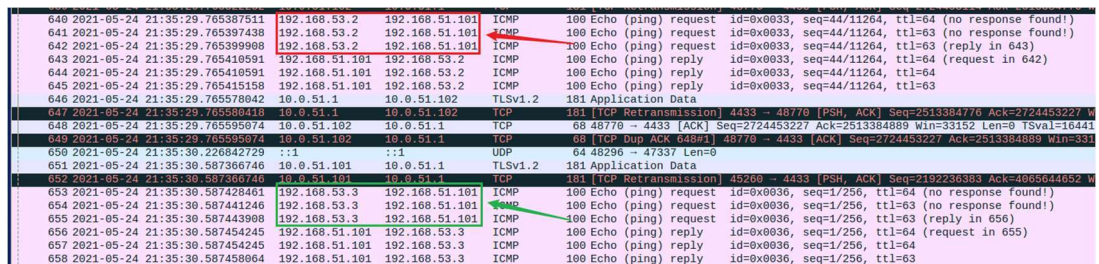


图 4-19 Wireshark 抓包图

此时尝试断开一个客户端，观察到此时服务端会监测到连接的断开，并输出相应的信息，如下图：



图 4-20 断开连接时服务器提示信息图

查看 pipe 文件信息，观察到创建了与隧道 IP 最后一字节相同的 pipe 文件，如下图：



5 实验思考

5.1 Tunnel 断开测试

首先使用如下命令开启 inner 的 telnet 服务:

```
service openbsd-inetd start
```

然后使用如下命令查看 VPN 程序的进程号, 并使用 kill 命令结束对应进程以断开隧道, 如下:

```
ps -aux | grep vpn
```

```
[05/06/21]seed@VM:~/.../vpn$ ps -aux | grep vpn
root   6416  0.0  0.0  2208  60 ?        Ss   06:18   0:00 /home/seed/workspace/vpn/vpnserver
root   6568  0.0  0.0  4936  172 ?        Ss   06:18   0:00 /workspace/vpnclient 10.0.51.1
seed    6631  0.0  0.0  7732 1844 pts/2    S+   06:32   0:00 grep --color=auto vpn
[sudo] password for seed:
[05/06/21]seed@VM:~/.../vpn$
```

图 5-1 断开连接图

需要将隧道 VPN 的服务端与客户端均断开, 这是由于当 VPN 建立时服务端启动并等待连接, 当得到一个客户端的连接请求时再完成对服务进程的初始化, 随后进入接受数据包并完成相应处理的循环中。如果仅断开客户端, 则由于服务端仍处于等待转发数据的循环中, 仍对原有连接端口进行监听, 不能正常响应客户端的连接请求, 因此无法正常连接。如果仅断开服务端, 则当服务端再次启动时, 由于客户端仍处于原有隧道的处理循环中, 连接发送端口不变, 不能完成连接请求的发送, 因此无法完成连接。具体如下代码所示:

<pre>87 int main(int argc, char *argv[]) 88 { 89 int tunfd, sockfd; 90 91 daemon(1, 1); 92 93 tunfd = createTunDevice(); 94 sockfd = connectToUDPServer(argv[1]); 95 96 // Enter the main Loop 97 while (1) { 98 fd_set readFDSet; 99 100 FD_ZERO(&readFDSet); 101 FD_SET(sockfd, &readFDSet); 102 FD_SET(tunfd, &readFDSet); 103 select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL); 104 105 if (FD_ISSET(tunfd, &readFDSet)) 106 tunSelected(tunfd, sockfd); 107 if (FD_ISSET(sockfd, &readFDSet)) 108 socketSelected(tunfd, sockfd); 109 } 110 }</pre>	客户端	<pre>94 int main(int argc, char *argv[]) 95 { 96 int tunfd, sockfd; 97 98 daemon(1, 1); 99 100 tunfd = createTunDevice(); 101 sockfd = initUDPServer(); 102 103 // Enter the main Loop 104 while (1) { 105 fd_set readFDSet; 106 107 FD_ZERO(&readFDSet); 108 FD_SET(sockfd, &readFDSet); 109 FD_SET(tunfd, &readFDSet); 110 select(FD_SETSIZE, &readFDSet, NULL, NULL, NULL); 111 112 if (FD_ISSET(tunfd, &readFDSet)) 113 tunSelected(tunfd, sockfd); 114 if (FD_ISSET(sockfd, &readFDSet)) 115 socketSelected(tunfd, sockfd); 116 } 117 }</pre>	服务端
---	------------	--	------------

图 5-2 客户端与服务端实现代码图

此时在 telnet 中输入, 没有任何响应 (包括输入回显)。随后再次重新连接隧道, 如下图:

```
[05/06/21]seed@VM:~/.../vpn$ ps -aux | grep vpn
root   6683  0.0  0.0  2208  60 ?        Ss   06:34   0:00 /home/seed/workspace/vpn/vpnserver
root   6862  0.0  0.0  4936  172 ?        Ss   06:35   0:00 /workspace/vpnclient 10.0.51.1
seed    8058  0.0  0.0  7732 1836 pts/2    S+   07:31   0:00 grep --color=auto vpn
[05/06/21]seed@VM:~/.../vpn$
```

图 5-3 重连图

观察到隧道连接后 telnet 端显示出在断开时输入的指令，如下图：

```
root@user:/# telnet 192.168.51.101
Trying 192.168.51.101...
Connected to 192.168.51.101.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
inner login: seed
Password:
Last login: Thu May 6 06:11:38 CST 2021 on pts/0
sh: 1: cannot create /run/motd.dynamic.new: Directory nonexistent
[05/06/21]seed@inner:~$ pwd
/home/seed
[05/06/21]seed@inner:~$ pwd
```

图 5-4 重连结果图

同时使用 Wireshark 抓包，得到如下结果：

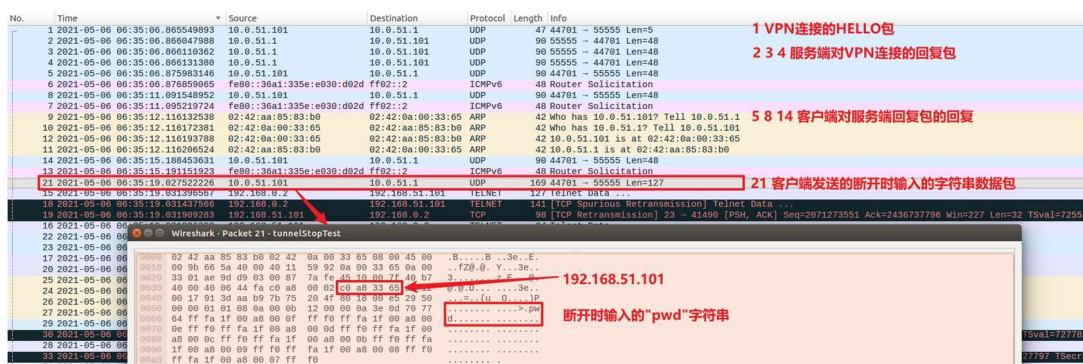


图 5-5 Wireshark 抓取的重连数据包图

观察到当重新建立隧道连接时，客户端会首先向服务端发送连接请求，随后收到服务端的响应并发送响应数据包。同时，由于 telnet 服务的特性，当连接重新建立后，会发送在断开连接时未发送出去的数据。

5.2 超时重传

对上述 telnet 所提供的连接断开重连后能够成功回显输入的现象进行解释。这是由于 telnet 服务使用的是 TCP 连接，当客户端与服务器下线后，TCP 数据包无法发送出去，因此触发超时重传机制。而当再次开启服务端与客户端后，由于服务端另一侧此时仍处于超时重传机制中，因此会发送对之前没有收到数据包的 ACK。当客户端接收到三个冗余 ACK 后触发快速重传，重新发送之前没有发送出的报文，得到相应回复并根据回复完成指令回显。