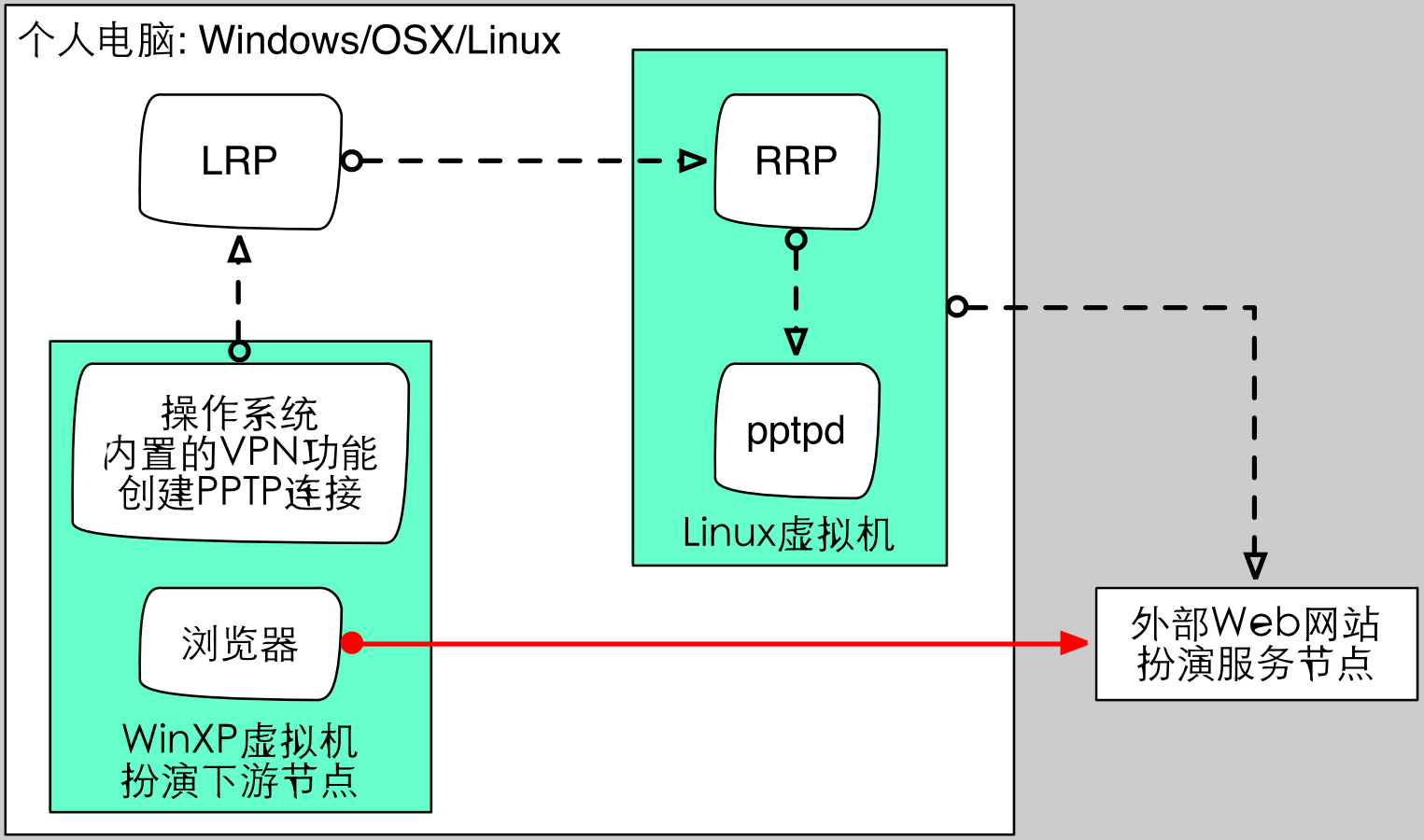
实验tcpTLV: TCP结构化消息

- 提交邮箱：buptne@gmail.com
- 邮件标题：tcpTLV-班级-学号-姓名
- 邮件正文：报告粘贴到正文（不要使用附件）
- 提交时间：2017年4月14日
- 报告内容：描述tcpTLV程序的设计思路

本次实验中RRP省略
LRP直接连接pptpd



成功效果:

1. WinXP虚拟机通过系统内置的VPN功能创建到LRP的PPTP连接,成功获取VPN地址.
2. WinXP虚拟机中的浏览器在VPN环境下可以访问外部Web网站

实验要求

- 在tcpRelay程序基础上编写程序LRP(与之前的LRP程序同名)
  - 向下游PPTP客户端提供1723服务端口
  - 收到下游TCP请求L1时发起到上游pptpd服务器的连接L2
  - 在pptpd的连接状态L2未决时需要暂存L1上的数据
  - L1或L2任意一端断开都会导致另一端也断开
  - 提供PPTP单个完整消息的接收
  - 提供对PPTP消息中的Call ID和Peer Call ID等字段的编解码
  - 当L2连接成功后在L1和L2之间中继转发数据(必须是完整的PPTP消息)
- 测试过程
  - 在LRP主机上运行TCP中继程序LRP
  - 在WinXP虚拟机上创建到LRP地址的PPTP-VPN并进行拨号
  - LRP在收到WinXP的PPTP请求后会连接到pptpd所在的Linux虚拟机
  - 在WinXP与Linux之间会进行正常的PPTP协议交互
  - LRP可以调试输出正确的Call ID和Peer Call ID字段值(通过Wireshark判断)
  - Wireshark上捕获到PPTP协议成功并发现GRE报文(验证PPTP消息转发正确)

Qt方案参考

- 在QTcpSocket的readyRead信号槽上处理
  - 对收到的数据进行PPTP消息解码
  - 如果消息解码成功则转发到对等连接(Link->peer)
  - 每个Link上有两个Call ID分别对应发送(callIdTx)和接收(callIdRx)
    - callIdTx：对应LRP在该连接Link上发送出的outCallRequest消息中的Call ID
    - callIdRx：对应LRP在该连接Link上接收到的outCallRequest消息中的Call ID
  - 下游一侧(对着PPTP客户端)
    - 下游发送到LRP的Call ID为该Link上的callIdRx
    - LRP发送到下游的Call ID为该Link上的callIdTx
  - 上游一侧(对着pptpd)
    - LRP发送到上游的Call ID为该Link上的callIdTx
    - 上游发送到LRP的Call ID为该Link上的callIdRx
- 后面代码示例中的LOCK和UNLOCK是为了互斥Pcap线程与TCP线程对Link资源的访问，这部分功能将在下一个试验中讲到。

```
static LinkHash     gLinkHash;
static LinkHashGre  gLinkHashGre;
static QMutex       gLinkHashMutex;
#define LOCK    do{ qWarning() << "lock";   gLinkHashMutex.lock();   }while(0)
#define UNLOCK  do{ qWarning() << "unlock"; gLinkHashMutex.unlock(); }while(0)

quint16 rawDecodeUint16(const char *data, int pos)
{
    quint16 result = (quint8)data[pos] * 256 + (quint8)data[pos + 1];
    return result;
}

quint32 rawDecodeUint32(const char *data, int pos)
{
    quint32 result = (quint8)data[pos]*256*256*256 + (quint8)data[pos+1]*256*256 + (quint8)data[pos+2]*256 + (quint8)data[pos+3];
    return result;
}

void rawEncodeUint16(char *data, int pos, quint16 value)
{
    data[pos + 0] = (value >> 8) & 0x00ff;
    data[pos + 1] = (value >> 0) & 0x00ff;
}

void rawEncodeUint32(char *data, int pos, quint32 value)
{
    data[pos + 0] = (value >>24) & 0x00ff;
    data[pos + 1] = (value >>16) & 0x00ff;
    data[pos + 2] = (value >> 8) & 0x00ff;
    data[pos + 3] = (value >> 0) & 0x00ff;
}
```

```
enum {
    pptpLen_msgLen                = 2,
    pptpLen_startCtrlConnRequest  = 156,
    pptpLen_startCtrlConnReply    = 156,
    pptpLen_outCallRequest        = 168,
    pptpLen_outCallReply          = 32,
    pptpLen_inCallRequest         = 220,
    pptpLen_inCallReply           = 24,
    pptpLen_inCallConnected       = 28,
    pptpLen_callClearRequest      = 16,
    pptpLen_callDisconnectNotify  = 148,
    pptpLen_wanErrorNotify        = 40,
    pptpLen_setLinkInfo           = 24
};
enum {
    pptpPos_pptpMsgType                  = 2,
    pptpPos_ctrlMsgType                  = 8,
    pptpPos_outCallRequest_callId        = 12,
    pptpPos_outCallRequest_CallSn        = 14,
    pptpPos_outCallReply_callId          = 12,
    pptpPos_outCallReply_peerCallId      = 14,
    pptpPos_inCallRequest_callId         = 12,
    pptpPos_inCallRequest_CallSn         = 14,
    pptpPos_inCallReply_callId           = 12,
    pptpPos_inCallReply_peerCallId       = 14,
    pptpPos_inCallConnected_peerCallId   = 12,
    pptpPos_callClearRequest_callId      = 12,
    pptpPos_callDisconnectNotify_callId  = 12,
    pptpPos_wanErrorNotify_peerCallId    = 12,
    pptpPos_setLinkInfo_peerCallId       = 12
};
```

```
enum {
    pptpType_startCtrlConnRequest  = 1,
    pptpType_startCtrlConnReply    = 2,
    pptpType_stopCtrlConnRequest   = 3,
    pptpType_stopCtrlConnReply     = 4,
    pptpType_echoRequest           = 5,
    pptpType_echoReply             = 6,
    pptpType_outCallRequest        = 7,
    pptpType_outCallReply          = 8,
    pptpType_inCallRequest         = 9,
    pptpType_inCallReply           = 10,
    pptpType_inCallConnected       = 11,
    pptpType_callClearRequest      = 12,
    pptpType_callDisconnectNotify  = 13,
    pptpType_wanErrorNotify        = 14,
    pptpType_setLinkInfo           = 15
};
```

```
struct Link {
    quint16 callIdRx;
    quint16 callIdTx;
    QByteArray dataRx;
    QByteArray dataTx;
    Link *peer;
    QTcpSocket *tcp;
};
typedef QHash<QTcpSocket*, Link*> LinkHash;
typedef QHash<QString, Link*> LinkHashGre;
```

```
void Server::tcpReadyRead()
{
    QTcpSocket *tcp = qobject_cast<QTcpSocket*>(sender());
    LOCK;
    LinkHash::iterator itr = gLinkHash.find(tcp);
    Link *link = itr.value();
    link->dataRx.append(tcp->readAll());
    qDebug() << tcpAddr(tcp) << "rx" << link->dataRx.size();
    pptpRecv(link);
    UNLOCK;
}
```

```
#define PPTP_CALL(type) case pptpType_##type: pptpRecv_##type(link); break

void Server::pptpRecv(Link *link)
{
    Link *peer = link->peer;
    bool peerSend = false;
    while (pptpLen_msgLen < link->dataRx.size()) {
        quint16 msgLen = rawDecodeUint16(link->dataRx.constData(), 0);
        qDebug() << "msgLen" << msgLen;
        if (link->dataRx.size() < msgLen) {
            break;
        }
        quint16 ctrlMsgType = rawDecodeUint16(link->dataRx.constData(), pptpPos_ctrlMsgType);
        switch (ctrlMsgType) {
        PPTP_CALL(startCtrlConnRequest);
        PPTP_CALL(startCtrlConnReply);
        PPTP_CALL(stopCtrlConnRequest);
        PPTP_CALL(stopCtrlConnReply);
        PPTP_CALL(echoRequest);
        PPTP_CALL(echoReply);
        PPTP_CALL(outCallRequest);
        PPTP_CALL(outCallReply);
        PPTP_CALL(inCallRequest);
        PPTP_CALL(inCallReply);
        PPTP_CALL(inCallConnected);
        PPTP_CALL(callClearRequest);
        PPTP_CALL(callDisconnectNotify);
        PPTP_CALL(wanErrorNotify);
        PPTP_CALL(setLinkInfo);
        }
        peerSend = true;
        peer->dataTx.append(link->dataRx.left(msgLen));
        link->dataRx.remove(0, msgLen);
    }
    if (peerSend) {
        qDebug() << tcpAddr(peer->tcp) << "tx" << peer->dataTx.size();
        tcpSend(peer);
    }
}
```

```
void Server::pptpRecv_outCallReply(Link *link)
{
    quint16 callId     = rawDecodeUint16(link->dataRx.constData(), pptpPos_outCallReply_callId);
    quint16 peerCallId = rawDecodeUint16(link->dataRx.constData(), pptpPos_outCallReply_peerCallId);
    qDebug() << "callId" << callId << "peerCallId" << peerCallId;
    link->callIdRx = callId;
    if (peerCallId != link->callIdTx) {
        qCritical() << "errPeerCallId" << peerCallId << "callIdTx" << link->callIdTx;
    }
    Link *peer = link->peer;
    peer->callIdTx = callId;
    rawEncodeUint16(link->dataRx.data(), pptpPos_outCallReply_peerCallId, peer->callIdRx);

    QString linkGre = link->tcp->peerAddress().toString() + ":" + QString::number(peerCallId);
    gLinkHashGre.insert(linkGre, link);

    QString peerGre = peer->tcp->peerAddress().toString() + ":" + QString::number(callId);
    gLinkHashGre.insert(peerGre, peer);
}

void Server::pptpRecv_outCallRequest(Link *link/*, const QByteArray &data*/)
{
    quint16 callId = rawDecodeUint16(link->dataRx.constData(), pptpPos_outCallRequest_callId);
    quint16 callSn = rawDecodeUint16(link->dataRx.constData(), pptpPos_outCallRequest_CallSn);
    qDebug() << "callId" << callId << "callSn" << callSn;
    link->callIdRx = callId;
    Link *peer = link->peer;
    peer->callIdTx = callId;
    rawEncodeUint16(link->dataRx.data(), pptpPos_outCallRequest_callId, peer->callIdTx);
}
```