Wireshark 学习笔记

1. 简介

2. Wireshark 功能模块

下图给出了 wireshark 功能模块：

a) GTK1/2

处理用户的输入输出显示，源码在 gtk 目录.

b) Core

核心模块，通过函数调用将其他模块连接在一起，源码在根目录

c) Epan

wireshark Packetage Analyzing，包分析引擎，源码在 epan 目录

- Protocol-Tree：保存数据包的协议信息，wireshark 的协议结构采用树形结构，解析协议报文时只需要从根节点通过函数句柄依次调用各层解析函数即可。
- Dissectors：在 epan/dissector 目录下，各种协议解码器，支持 700+种协议解析，对于每种协议，解码器都能识别出协议字段（field），并显示出字段值（field value）由于网络协议种类很多，为了使协议和协议间层次关系明显，对数据流里的各个层次的协议能够逐层处理，wireshark 系统采用了协议树的方式。
- Plugins：一些协议解码器以插件形式实现，源码在 plugins 目录
- Display-Filters：显示过滤引擎，源码在 epan/dfilter 目录

d) Capture

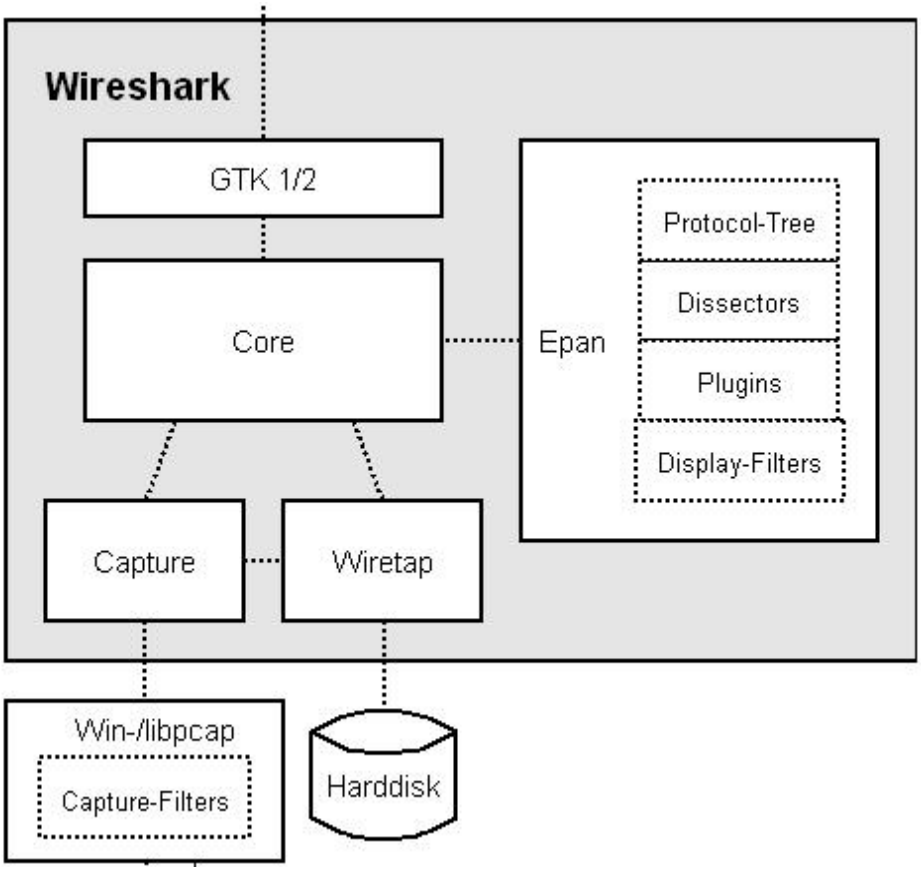
捕包引擎，利用 libpcap/WinPcap 从底层抓取网络数据包，libpcap/WinPcap 提供了通用的抓包接口，能从不同类型的网络接口(包括以太网，令牌环网，ATM 网等)获取数据包。

e) Wiretap

从文件中读取数据包，支持多种文件格式，源码在 wiretap 目录

f) Win-/libpcap

Wireshark 抓包时依赖的库文件



wireshark 功能模块

3. wireshark 流程分析

1) 初始化

Wireshark 的初始化包括一些全局变量的初始化、协议分析引擎的初始化和 Gtk 相关初始化，显示 Ethereal 主窗口，等待用户进一步操作。重点就是 Epan 模块的初始化。

Epan 初始化：

- tvbuff 初始化：全局变量 tvbuff_mem_chunk 指向用 memchunk 分配的固定大小的空闲内存块，每个内存块是 tvbuff_t 结构，从空闲内存块中取出后，用来保存原始数据包。
- 协议初始化：
  - 全局变量：
    - proto_names
    - proto_short_names

- proto_filter_names

  以上三个全局变量主要用来判断新注册的协议名是否重复，如果重复，给出提示信息，在协议解析过程中并没有使用。
- 协议注册：
  - 注册协议：将三个参数分别注册给 proto_names、proto_short_names、proto_filter_names 三个全局变量中，
  - 注册字段,需要在 wireshark 协议树显示的报文内容字段。
  - 协议解析表
- Handoff 注册
  - 将协议与父协议节点关联起来

- Packet（包）初始化
  - 全局变量：
    - frame_handle：协议解析从 frame 开始，层层解析，直到所有的协议都解析完为止。frame_handle 保存了 frame 协议的 handle。
    - data_handle：有的协议无法从 frame 开始，那么就从 data 开始。原理同 frame。
- 读配置文件 preference
- 读 capture filter 和 display filter 文件，分别保存在全局变量 capture_filter 和 display_filter 中。
- 读 disabled protocols 文件，保存全局变量 global_disabled_protos 和 disabled_protos 中
- 初始化全局变量 cfile
  - Cfile 是个重要的变量，数据类型为 capture file，它保存了数据包的所有信息，
- 取得命令行启动时，参数列表，并进行相应的处理

2) 处理流程

Wireshark 初始化完成以后进入实际处理阶段，主程序创建抓包进程，捕包进程和主程序是通过 PIPE 进行传递数据的，主程序把抓取的数据写入临时文件，通过函数 add_packet_to_packet_list 将数据包加入包列表。处理时，主程序从列表中选取一个数据包，提取该数据包中的数据填写在数据结构中，最后调用协议解析函数 epan_dissect_run 进行处理，从 epan_dissect_run 开始，是实际的协议解析过程，
下面以 HTTP 协议报文为例，流程如下：

a) 解析 frame 层

调用函数 dissect_frame 对 frame 层进行解析，并在协议树上填充相应字段信息。函数最后会判断是否有上层协议封装，如果有则调用函数 dissector_try_port 在协议树上查找对应的解析函数，这里函数 dissector_try_port 根据 pinfo->fd->lnk_t 查找对应的上层协议处理函数，pinfo->fd->lnk_t 值为 1，上层封装协议为以太网协议，全局结构体指针变量 dissector_handle 当前的协议解析引擎句柄置为 dissect_eth_maybefcs，至此，frame 层解析结束。

b) 解析以太网层

函数 call_dissector_work 根据 dissector_handle 调用 frame 上层协议解析函数

dissect_eth_maybefcs 对以太网层进行解析，并在协议树上填充相应字段，包括目的 MAC 地址和以太网上层协议类型等信息。函数最后会判断是否有上层协议封装，如果有则调用函数 dissector_try_port 在协议树上查找对应的解析函数，这里函数 dissector_try_port 根据 etype 查找对应的上层协议处理函数，以太网字段 etype 为 0800 的报文是 ip 报文，上层封装协议为 IP 协议，全局结构体指针变量 dissector_handle 当前的协议解析引擎句柄置为 dissect_ip，至此，以太网层解析结束。

c) 解析 IP 层

函数 call_dissector_work 根据 dissector_handle 调用以太网上层协议解析函数 dissect_ip 对以太网层进行解析，并在协议树上填充相应字段，包括版本号，源地址，目的地址等信息。函数最后会判断是否有上层协议封装，如果有则调用函数 dissector_try_port 在协议树上查找对应的解析函数，这里函数 dissector_try_port 根据 nxt (nxt = iph->ip_p)查找对应的上层协议处理函数，以太网字段 nxt 为 06 的报文是 TCP 报文，上层封装协议为 TCP 协议，全局结构体指针变量 dissector_handle 当前的协议解析引擎句柄置为 dissect_tcp，至此，IP 层解析结束。

d) 解析 TCP 层

函数 call_dissector_work 根据 dissector_handle 调用以太网上层协议解析函数 dissect_tcp 对 TCP 层进行解析，包括对 TCP 头的解析和选项字段的解析，并在协议树上填充相应字段，包括源端口，目的端口，标志位等信息。函数最后会判断是否有上层协议封装，如果有则调用函数 dissector_try_port 在协议树上查找对应的解析函数，这里函数 dissector_try_port 根据 port 查找对应的上层协议处理函数，将源端口和目的端口分别赋值给 low_port 和 high_port，根据 low_port 和 high_port 分别匹配上层协议解析函数，port 为 80 的报文是 HTTP 报文，上层封装协议为 HTTP 协议，全局结构体指针变量 dissector_handle 当前的协议解析引擎句柄置为 dissect_http，至此，TCP 层解析结束。

e) 解析 HTTP 层

至此 wireshark 进入应用层协议检测阶段，wireshark 解析 dissect_http 函数中注册的字段，并提取相应的字段值添加到协议树中，应用层的具体解析流程将在下面介绍。HTTP 协议具体函数调用过程参见：


wireshark_http_fun_list.TXT

重要的数据结构

```
struct _epan_dissect_t {
    tvbuff_t   *tvb;//用来保存原始数据包
    proto_tree     *tree;//协议树结构
    packet_info    pi;// 包括各种关于数据包和协议显示的相关信息
};

/** Each proto_tree, proto_item is one of these. */
typedef struct _proto_node {
```

```
    struct _proto_node *first_child;//协议树节点的第一个子节点指针
    struct _proto_node *last_child; //协议树节点的最后一个子节点指针
    struct _proto_node *next; //协议树节点的下一个节点指针
    struct _proto_node *parent;//父节点指针
    field_info    *finfo;//保存当前协议要显示的地段
    tree_data_t *tree_data;//协议树信息
} proto_node;


typedef struct _packet_info {
  const char *current_proto;//当前正在解析的协议名称
  column_info *cinfo;          //wireshark 显示的信息
  frame_data *fd;//现在分析的原始数据指针
  union wtap_pseudo_header *pseudo_header;//frame 类型信息
  GSList *data_src;       /*frame 层信息  */
  address dl_src;         /*  源 MAC */
  address dl_dst;         /*目的 MAC */
  address net_src;        /*  源 IP */
  address net_dst;        /*目的 IP */
  address src;            /*源 IP */
  address dst;            /*目的 IP */
  guint32 ethertype;          /*以太网类型字段*/
  guint32 ipproto;        /* IP 协议类型*/
  guint32 ipxptype;           /* IPX  包类型  */
  guint32 mpls_label;         /* MPLS 包标签*/
  circuit_type ctype;
  guint32 circuit_id;         /*环路 ID */
  const char *noreassembly_reason;   /*  重组失败原因*/
  gboolean fragmented;            /*为真表示未分片*/
  gboolean in_error_pkt;     /*错误包标志*/
  port_type ptype;       /*端口类型  */
  guint32 srcport;       /*源端口*/
  guint32 destport;           /*目的端口*/
  guint32 match_port;                /*进行解析函数匹配时的匹配端口*/
  const char *match_string;  /*调用子解析引擎时匹配的协议字段指针*/
  guint16 can_desegment;   /*  能否分段标志*/
  guint16 saved_can_desegment;
  int desegment_offset;       /*分段大小*/
#define DESEGMENT_ONE_MORE_SEGMENT 0x0fffffff
#define DESEGMENT_UNTIL_FIN             0x0ffffffe
  guint32 desegment_len;
  guint16 want_pdu_tracking;
  guint32 bytes_until_next_pdu;
  int         iplen;                      /*IP 包总长*/
```

```
    int         iphdrlen;                   /*IP 头长度*/
    int          p2p_dir;
    guint16 oxid;                           /* next 2 fields reqd to identify fibre */
    guint16 rxid;                           /* channel conversations */
    guint8    r_ctl;                        /* R_CTL field in Fibre Channel Protocol */
    guint8    sof_eof;
    guint16 src_idx;                        /* Source port index (Cisco MDS-specific) */
    guint16 dst_idx;                        /* Dest port index (Cisco MDS-specific) */
    guint16 vsan;                           /* Fibre channel/Cisco MDS-specific */
    /* Extra data for DCERPC handling and tracking of context ids */
    guint16 dcectxid;                       /* Context ID (DCERPC-specific) */
    int         dcetransporttype;
    guint16 dcetransportsalt;   /* fid: if transporttype==DCE_CN_TRANSPORT_SMBPIPE */
#define DECRYPT_GSSAPI_NORMAL     1
#define DECRYPT_GSSAPI_DCE      2
    guint16 decrypt_gssapi_tvb;
    tvbuff_t *gssapi_wrap_tvb;
    tvbuff_t *gssapi_encrypted_tvb;
    tvbuff_t *gssapi_decrypted_tvb;
    gboolean gssapi_data_encrypted;
    guint32 ppid;    /* SCTP PPI of current DATA chunk */
    guint32 ppids[MAX_NUMBER_OF_PPIDS]; /* The first NUMBER_OF_PPIDS PPIDS which are present * in the SCTP packet*/
    void        *private_data;     /* pointer to data passed from one dissector to another */
    /* TODO: Use emem_strbuf_t instead */
    GString *layer_names;       /* layers of each protocol */
    guint16 link_number;
    guint8    annex_a_used;
    guint16 profinet_type;       /* the type of PROFINET packet (0: not a PROFINET packet) */
    void *profinet_conv;        /* the PROFINET conversation data (NULL: not a PROFINET packet) */
    void *usb_conv_info;
    void *tcp_tree;           /* proto_tree for the tcp layer */

    const char *dcerpc_procedure_name;      /* Used by PIDL to store the name of the current dcerpc procedure */
    struct _sccp_msg_info_t* sccp_info;
    guint16 clnp_srcref;           /* clnp/cotp source reference (can't use srcport, this would confuse tpkt) */
    guint16 clnp_dstref;             /* clnp/cotp destination reference (can't use dstport, this would confuse tpkt) */
    guint16 zbee_cluster_id;    /* ZigBee cluster ID, an application-specific message identifier that
                                        * happens to be included in the transport (APS) layer header.
                                        */
    guint8 zbee_stack_vers;         int link_dir; /* 3GPP messages are sometime different UP link(UL)
```

```
or Downlink(DL)*/
} packet_info;
```