



# 网络抓包器 WireWhale 的设计与实现

基于 Python3 的 WireShark 的模仿实现

作者：曹昊天 & 蔡星浩

组织：SJTU F1703601

时间：December 29,

2019版本：1.00



*Victory won't come to us unless we go to it. — M. Moore*

# 目 录

---

<b>1 项目功能概述</b>	<b>1</b>
1.1 本项目实现的基本功能 . . . . .	1
1.2 本项目实现的特色功能 . . . . .	1
1.3 我们的联系方式 . . . . .	1
<b>2 项目实现概述</b>	<b>2</b>
2.1 项目开发环境 . . . . .	2
2.2 第三方库列表 . . . . .	3
<b>3 主要算法与数据结构</b>	<b>4</b>
3.1 文件结构 . . . . .	4
3.2 抓包核心功能 . . . . .	4
3.3 TCP 流重组 . . . . .	12
<b>4 项目功能演示</b>	<b>13</b>
4.1 项目基本功能 . . . . .	13
4.2 项目特色功能 . . . . .	21
<b>5 遇到的问题与解决方法</b>	<b>31</b>
5.1 曹昊天 . . . . .	31
5.2 蔡星浩 . . . . .	31
<b>6 个人体会与建议</b>	<b>32</b>
6.1 曹昊天 . . . . .	32
6.2 蔡星浩 . . . . .	32

# 第1章 项目功能概述

---

## 1.1 本项目实现的基本功能

1. 具有用户友好的交互界面
2. 通过指定需要侦听的网卡
3. 侦听进出本主机的数据包，并解析数据包的内容（包含 ARP、IP、ICMP、TCP、UDP 等报文中各字段的内容，且数据部分具有可读性）
4. 实现 TCP、UDP 数据包的全部数据显示——即 IP 包的分片重组
5. 实现包过滤功能
6. 实现数据包的查询
7. 实现数据包的保存与读取
8. 实现了文件重组功能

## 1.2 本项目实现的特色功能

1. 高度可定制化界面，允许用户修改字体、背景图片、各模块窗口大小等
2. 可引导用户进行流量包的伪造
3. 可对系统相关程序进行流量监测，并绘制相应图像
4. 实现可视化的流量包统计与分析
5. 内嵌帮助文档，方便用户查询使用方法
6. 实现抓包表格自动滚屏与表头自适应功能

## 1.3 我们的联系方式

### 1.3.1 项目 Github 地址

- Github 网址: <https://github.com/caohaotiantian/WireWhale>

### 1.3.2 曹昊天

- 手机号码: 18986335909
- 微信号: caohaotiantian
- QQ 号: 952550419
- 邮件: [952550419@qq.com](mailto:952550419@qq.com)

### 1.3.3 蔡星浩

- 手机号码: 18671951397
- 微信号: cxh943789196
- QQ 号: 943789196
- 邮件: [943789196@qq.com](mailto:943789196@qq.com)

## 第 2 章 项目实现概述

---

本项目基于 Python3 实现，故在 Windows、Linux 以及 Mac OS 等主流操作系统上均能正常运行，具有较强的可移植性和跨平台性。

软件的构建仿照著名网络协议分析软件 WireShark 进行开发，故对 WireShark 较为熟悉的用户能够使用过程中找到些许熟悉的感觉，这也方便了用户对我们产品的快速上手，一定程度上降低了学习成本

### 2.1 项目开发环境

#### 2.1.1 操作系统环境

##### Windows 规格

版本	Windows 10 家庭中文版
版本号	1909
安装日期	2019/12/10
操作系统版本	18363.535

图 2.1: 操作系统环境

#### 2.1.2 硬件环境

处理器	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz
已安装的 RAM	16.0 GB (15.8 GB 可用)

图 2.2: 硬件环境

### 2.1.3 开发软件环境

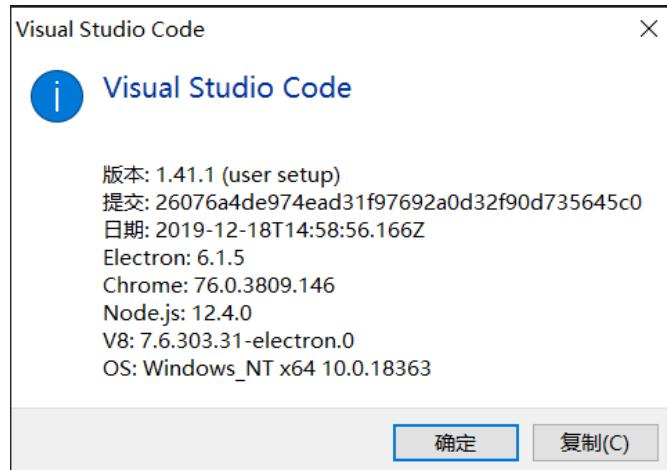


图 2.3: 开发软件环境

## 2.2 第三方库列表

第三方库名称	版本号	作用简述
scapy	2.4.3	使用户能够发送，嗅探和剖析并伪造网络数据包，可以用于构建探测，扫描或攻击网络的工具
matplotlib	3.1.1	具有强大的绘制图像的能力，用于将统计数据可视化
pyqt	5.9.2	用于构建可视化界面

表 2.1: 第三方库列表

# 第3章 主要算法与数据结构

---

## 3.1 文件结构

- capture\_core.py —— 抓包核心功能实现
- flow\_monitor.py —— 流量监测系统的实现
- forged\_packet.py —— 伪造包的实现
- main.py —— 主程序入口
- main\_ui.py —— 主要界面的内容
- tools.py —— 工具集，获取网卡列表，上传下载速度等
- monitor\_system.py —— 流量监测系统的界面

## 3.2 抓包核心功能

1、判断操作系统类型，设置刷新时间

```
platform, netcards = get_nic_list()
flush_time = 2000
if platform == 'Windows':
    keys = list(netcards.keys())
elif platform == 'Linux':
    keys = list(netcards)
```

图 3.1: 判断操作系统类型

2、设置 arp 参数字典

```
# arp字典
arp_dict = {
    1: "who-has",
    2: "is-at",
    3: "RARP-req",
    4: "RARP-resp",
    5: "Dyn-RARP-req",
    6: "Dyn-RARP-resp",
    7: "Dyn-RARP-err",
    8: "InARP-req",
    9: "InARP-resp"
}
```

图 3.2: 设置 arp 参数字典

## 3、设置 icmpv6 参数字典

```
# icmpv6 code字典
icmpv6_code = {
    1: {
        0: "No route to destination",
        1: "Communication with source: str administratively prohibited",
        2: "Beyond scope of source address",
        3: "Address unreachable",
        4: "Port unreachable"
    },
    3: {
        0: "hop limit exceeded in transit",
        1: "fragment reassembly time exceeded"
    },
    4: {
        0: "erroneous header field encountered",
        1: "unrecognized Next Header type encountered",
        2: "unrecognized IPv6 option encountered"
    },
}
```

图 3.3: 设置 icmpv6 参数字典

## 4、设置端口字典

```
# 端口字典
ports = {
    80: "HTTP",
    1900: "SSDP",
    53: "DNS",
    123: "NTP",
    21: "FTP",
    20: "FTP_Data",
    22: "SSH"
}
```

图 3.4: 设置端口字典

## 5、HTTPS 解析

```
# HTTPS解析
content_type = {
    '14': "Change Cipher Spec",
    '15': "Alert Message",
    '16': "Handshake Protocol",
    '17': "Application Data"
}
version = {'00': "SSLv3", '01': "TLSv1.0", '02': "TLSv1.1", '03': "TLSv1.2"}
```

图 3.5: HTTPS 解析

## 6、设置数据包背景颜色

```
# 数据包背景颜色字典
color_dict = {
    "TCP": "#e7e6ff",
    "TCPv6": "#e7e6ff",
    "UDP": "#daefff",
    "UDPV6": "#daefff",
    "ARP": "#faf0d7",
    "SSDP": "#ffe3e5",
    "SSDPv6": "#ffe3e5",
    "HTTP": "#caffbe",
    "HTTPv6": "#caffbe",
    "SSLv3": "#FFFFCC",
    "TLSv1.0": "#FFFFCC",
    "TLSv1.1": "#c797ff",
    "TLSv1.2": "#bfbdff",
    "ICMP": "#fce0ff",
    "ICMPv6": "#fce0ff",
    "NTP": "#daefff",
    "NTPv6": "#daefff",
    "DNS": "#CCFF99",
    "DNSv6": "#CCFF99"
}
```

图 3.6: 设置数据包背景颜色

## 7、抓包后台类 Core()

## a、设置相关参数

```
class Core():
    """ 抓包后台类 """
    # 抓到的包编号从1开始
    packet_id = 1
    # 开始标志
    start_flag = False
    # 暂停标志
    pause_flag = False
    # 停止标志
    stop_flag = False
    # 保存标志
    save_flag = False
    # 窗口
    main_window = None
    # 开始时间戳
    start_timestamp = 0.0
    # 临时文件路径
    temp_file = None
    # 计数器
    counter = {"ipv4": 0, "ipv6": 0, "tcp": 0, "udp": 0, "icmp": 0, "igmp": 0, "arp": 0}
    # filter表达式
    filters = {"src_ip": [], "des_ip": [], "sport": [], "dport": [], "prot": []}
    # filter标志位
    filter_flag = False
    # 显示指示序号
    index = 0
```

图 3.7: 设置相关参数

## b、初始化

```
def __init__(self, mainwindow):
    """
    初始化，若不设置netcard则为捕捉所有网卡的数据包
    :param mainwindow: 传入主窗口
    """
```

图 3.8: 初始化

## c、设置过滤规则

```
>     def rule_create(self, expression):
>         rules = {"src_ip": [], "des_ip": [], "sport": [], "dport": [], "prot": []}
>         if expression.find("src_ip==") != -1: ...
>             if expression.find("des_ip==") != -1: ...
>                 if expression.find("sport==") != -1: ...
>                     if expression.find("dport==") != -1: ...
>                         if expression.find("prot==") != -1: ...
>
>         return rules
```

图 3.9: 设置过滤规则

## d、处理抓到的数据包

```
def process_packet(self, packet, packet_id):
    """
    处理抓到的数据包
    :param packet: 需要处理分类的包
    """

    try:
        protocol = None
        if self.packet_id == 1:
            self.start_timestamp = packet.time
        packet_time = packet.time - self.start_timestamp
        # 第二层
        ether_type = packet.payload.name
        version_addr = ""
        # IPv4
        if ether_type == "IP":
            source = packet[IP].src
            destination = packet[IP].dst
            self.counter["ipv4"] += 1
        # IPv6
        elif ether_type == "IPv6":
            source = packet[IPv6].src
            destination = packet[IPv6].dst
            version_addr = "v6"
            self.counter["ipv6"] += 1
        # ARP
    
```

图 3.10: 处理抓到的数据包

## e、处理点击列表中的项，获取相关信息

```
def on_click_item(self, this_id):
    """
    处理点击列表中的项
    :param this_id: 包对应的packet_id，在packet_list里获取该packet
    """

    try:
        if not this_id or this_id < 1:
            return
        previous_packet_time, packet = self.read_packet(this_id - 1)
        # 详细信息列表，用于添加进GUI
        first_return = []
        second_return = []
        # 第一层：Frame
        first_layer = []
    
```

图 3.11: 处理点击列表中的项，获取相关信息

## f、递归处理更详细的信息

```

def get_next_layer(self, packet):
    """
    递归处理下一层信息
    :param packet: 处理来自上一层packet的payload
    """

    # 第二层: Ethernet
    first_return = []
    second_return = []
    next_layer = []
    try:
        protocol = packet.name
        packet_class = packet.__class__
        if protocol == "NoPayload": ...
        elif protocol == "Ethernet": ...
        # 第三层: 网络层
        # IPv4
        elif protocol == "IP" or protocol == "IP in ICMP": ...
        # IPv6
        elif protocol == "IPv6" or protocol == "IPv6 in ICMPv6": ...
        elif protocol == "ARP": ...
        # 第四层: 传输层
        elif protocol == "TCP" or protocol == "TCP in ICMP": ...
        elif protocol == "UDP" or protocol == "UDP in ICMP": ...
        elif protocol == "ICMP" or protocol == "ICMP in ICMP": ...
        elif len(protocol) >= 6 and protocol[0:6] == "ICMPv6": ...
        # 第五层: 应用层
        # TLS
        else: ...
        if next_layer: ...
        first_temp, second_temp = self.get_next_layer(packet.payload)
        first_return += first_temp
        second_return += second_temp
    except:
        # 未知数据包
        first_return.clear()
        second_return.clear()

```

图 3.12: 递归处理更详细的信息

## g、刷新相关数据

```

def flow_count(self, netcard=None):
    """
    刷新下载速度、上传速度、发包速度和收包速度
    """

    if netcard and platform == 'Windows':
        # 反转键值对
        my_dict = dict(zip(netcards.values(), netcards.keys()))
        netcard = my_dict[netcard]
    while not stop_capturing_thread.is_set():
        recv_bytes, sent_bytes, recv_pak, sent_pak = get_formal_rate(
            get_rate(netcard))
        if not self.pause_flag:
            self.main_window.comNum.setText('下载速度: ' + recv_bytes)
            self.main_window.baudNum.setText('上传速度: ' + sent_bytes)
            self.main_window.getSpeed.setText('收包速度: ' + recv_pak)
            self.main_window.sendSpeed.setText('发包速度: ' + sent_pak)
        self.main_window.comNum.setText('下载速度: 0 B/s')
        self.main_window.baudNum.setText('上传速度: 0 B/s')
        self.main_window.getSpeed.setText('收包速度: 0 pak/s')
        self.main_window.sendSpeed.setText('发包速度: 0 pak/s')

```

图 3.13: 刷新相关数据

## h、抓包

```
def capture_packet(self, netcard, filters):
    """
    抓取数据包
    """

    stop_capturing_thread.clear()
    # 第一个参数可以传入文件对象或者文件名字
    writer = PcapWriter(self.temp_file, append=True, sync=True)
    thread = Thread(target=self.flow_count, daemon=True, args=(netcard, ))
    thread.start()
    # sniff中的store=False 表示不保存在内存中，防止内存使用过高
    sniff(
        iface=netcard,
        prn=(lambda x: self.collect(x, writer)),
        filter=None,
        stop_filter=(lambda x: stop_capturing_thread.is_set()),
        store=False)
    # 执行完成关闭writer
    writer.close()
```

图 3.14: 抓包

```
def start_capture(self, netcard=None, filters=None):
    """
    开启新线程进行抓包
    :param netcard: 选择的网卡, "any"为全选
    :param filters: 过滤器条件
    """

    # 如果已开始抓包，则不能进行操作
    if self.start_flag:
        return
    # 如果已经停止且未保存数据包，则提示是否保存数据包
    if self.stop_flag:
        if not self.save_flag and self.packet_id > 1:
            result = QMessageBox.question(
                None,
                "提示",
                "是否保存已抓取的数据包？",
                QMessageBox.Yes,
                QMessageBox.Cancel,
```

图 3.15: 抓包

## i、保存文件

```
def save_captured_to_pcap(self):
    """
    将抓到的数据包保存为pcap格式的文件
    """
    if self.packet_id == 1:
        QMessageBox.warning(None, "警告", "没有可保存的数据包!")
        return
    # 选择保存名称
    filename, _ = QFileDialog.getSaveFileName(
        parent=None,
        caption="保存文件",
        directory=os.getcwd(),
        filter="Pcap Files (*.pcap);;All Files (*)",
    )
    if filename == "":
        QMessageBox.warning(None, "警告", "保存失败!")
        return
    # 如果没有设置后缀名（保险起见，默认是有后缀的）
    if filename.find(".pcap") == -1:
        # 默认文件格式为 pcap
        filename = filename + ".pcap"
    shutil.copy(self.temp_file, filename)
    os.chmod(filename, 0o400 | 0o200 | 0o040 | 0o004)
    QMessageBox.information(None, "提示", "保存成功!")
    self.save_flag = True
```

图 3.16: 保存文件

## j、打开文件

```
def open_pcap_file(self):
    """
    打开pcap格式的文件
    """
    if self.stop_flag and not self.save_flag:
        reply = QMessageBox.question(
            None,
            "提示",
            "是否保存已抓取的数据包？",
            QMessageBox.Yes,
            QMessageBox.Cancel,
        )
        if reply == QMessageBox.Yes:
            self.save_captured_to_pcap()
    filename, _ = QFileDialog.getOpenFileName(
        parent=None,
        caption="打开文件",
        directory=os.getcwd(),
        filter="Pcap Files (*.pcap);;All Files (*)",
    )
    if filename == "":
        return
    self.main_window.info_tree.clear()
    self.main_window.treeWidget.clear()
    self.main_window.set_hex_text("")
    # 如果没有设置后缀名（保险起见，默认是有后缀的）
    if filename.find(".pcap") == -1:
        # 默认文件格式为 pcap
        filename = filename + ".pcap"
```

图 3.17: 打开文件

```
def read_packet(self, location):
    ...
    读取硬盘中的pcap数据
    :param location: 数据包位置
    :return: 返回参数列表[上一个数据包的时间, 数据包]
    ...
    # 数据包时间是否为纳秒级
    nano = False
    # 打开文件
    f = open(self.temp_file, "rb")
    # 获取Pcap格式 magic
    head = f.read(24)
    magic = head[:4]
    linktype = head[20:]
    if magic == b"\xa1\xb2\xc3\xd4":  # big endian
        endian = ">"
        nano = False
    elif magic == b"\xd4\xc3\xb2\xa1":  # little endian
        endian = "<"
        nano = False
    elif magic == b"\xa1\xb2\x3c\x4d":  # big endian, nanosecond-precision
        endian = ">"
        nano = True
    elif magic == b"\x4d\x3c\xb2\xa1":  # little endian, nanosecond-precision
        endian = "<"
        nano = True
    ...
```

图 3.18: 打开文件

### 3.3 TCP 流重组

采用了以 SYN 标识数据包和 FIN 标识数据包作为接受 TCP 流的首位，在获取到要追踪的数据包的信息后，以套接字作为标识，扫描整个已抓包的列表，将符合套接字的包筛选出来，随后在一个列表中按数据起始序号 SEQ 为序将抓到的包进行排列，并丢弃掉重发送的包。当已抓包扫描完毕或者抓到的有效数据长度等于连接释放和建立时的序号之差时，结束扫描，将已有的数据按字段序号进行拼接。

# 第4章 项目功能演示

## 4.1 项目基本功能

### 4.1.1 用户友好的交互界面

我们为 WireWhale 编写了用户友好的交互界面，我们在细节处别出心裁，设定了一系列小彩蛋

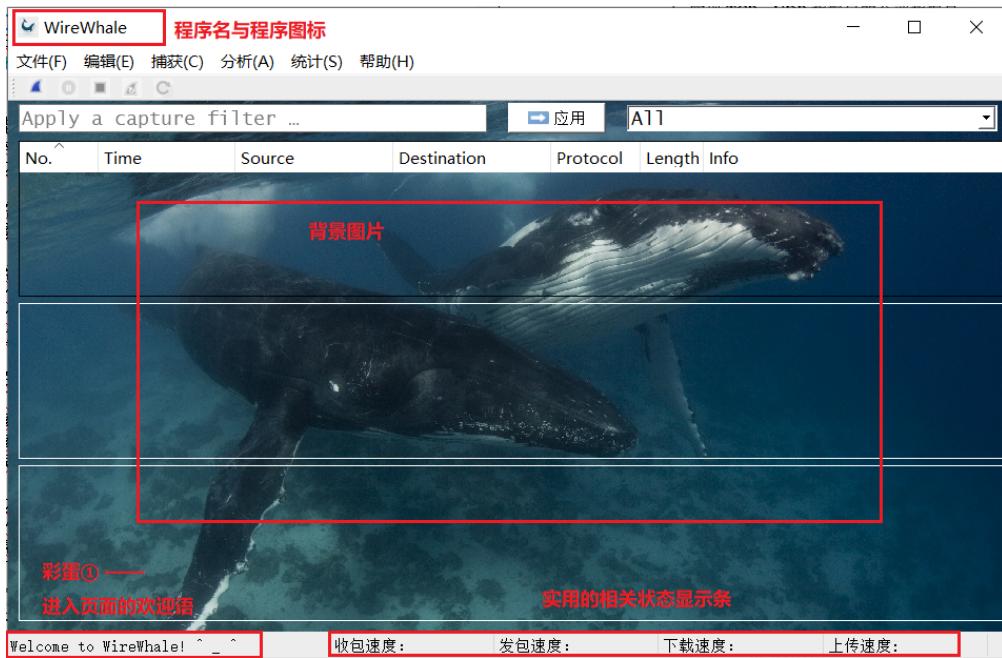


图 4.1: 初入程序的界面

- 进入界面时可在左下角看到欢迎语
- 窗体下方可根据执行功能的不同显示不同的提示信息
- 窗体背景为深海蓝鲸，对应我们的软件名称 WireWhale
- 程序图标是一只小蓝鲸，对应我们的软件名称 WireWhale

### 4.1.2 指定需要侦听的网卡

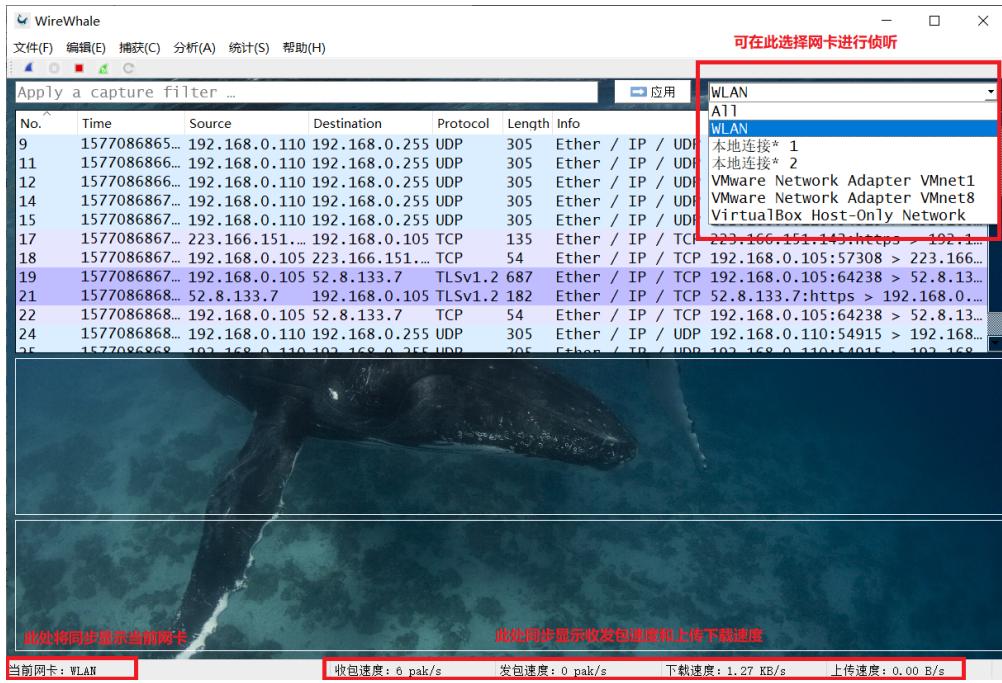


图 4.2: 选择需要侦听的网卡

- 可在界面右上角选择网卡进行侦听
- 窗体左下角同步显示当前网卡
- 窗体下方同步显示当前的收包速度、发包速度、上传速度、下载速度

### 4.1.3 侦听数据包并解析内容

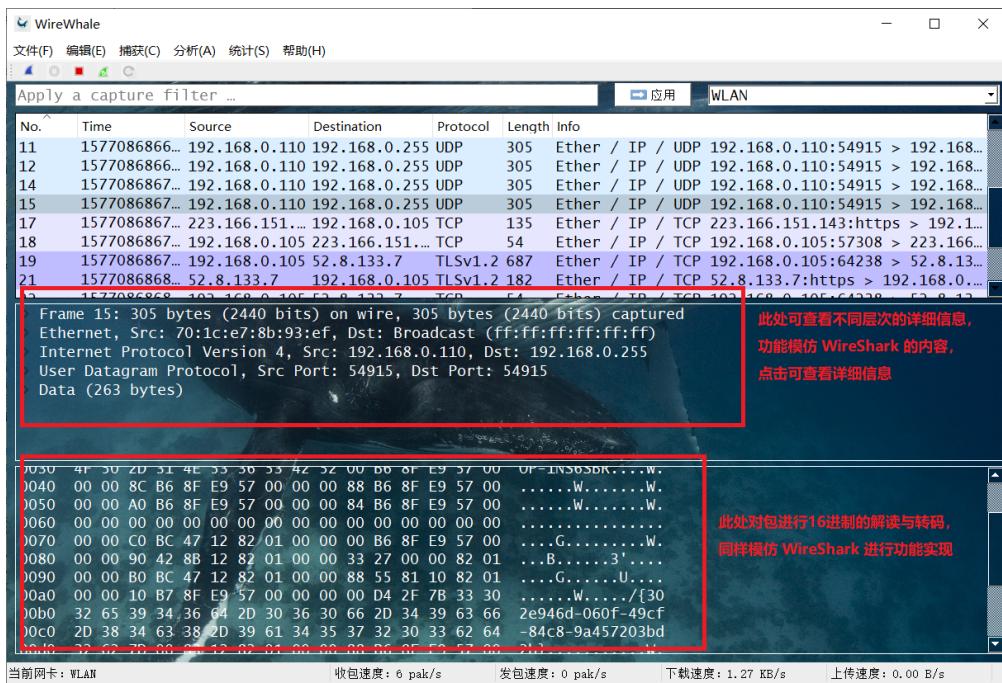


图 4.3: 侦听数据包并解析内容

- 在已侦听包的列表中选取一个即可立即在下方显示相关信息
- 中间窗体仿照 Wireshark 的功能，可分层查看不同的详细信息
- 下方窗体可看到对数据包的解析内容，同时以 16 进制和编码结果并排显示

#### 4.1.4 TCP 数据包的重组

- 在停止抓包后，右键点击 TCP 数据包，选中 TCP 流重组选项

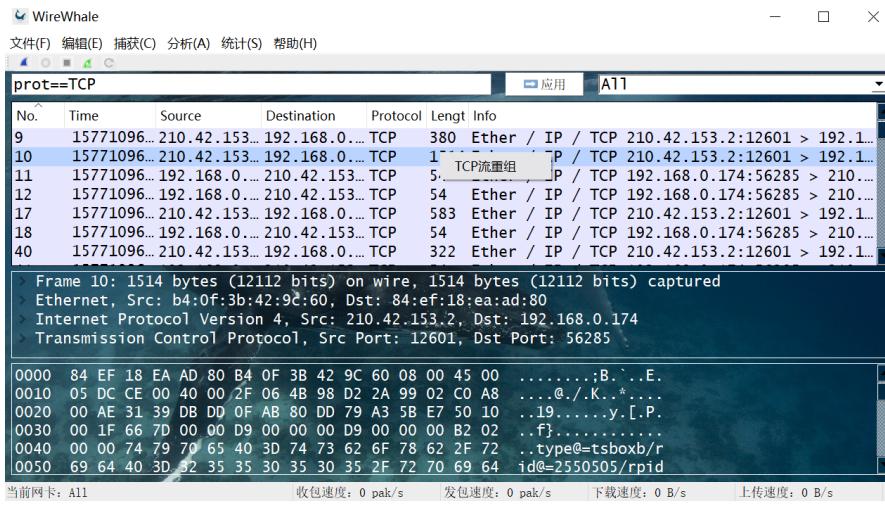


图 4.4: 选择重组包

- 随后显示 TCP 数据流的重组结果，若无重组数据则显示 No data in this packet

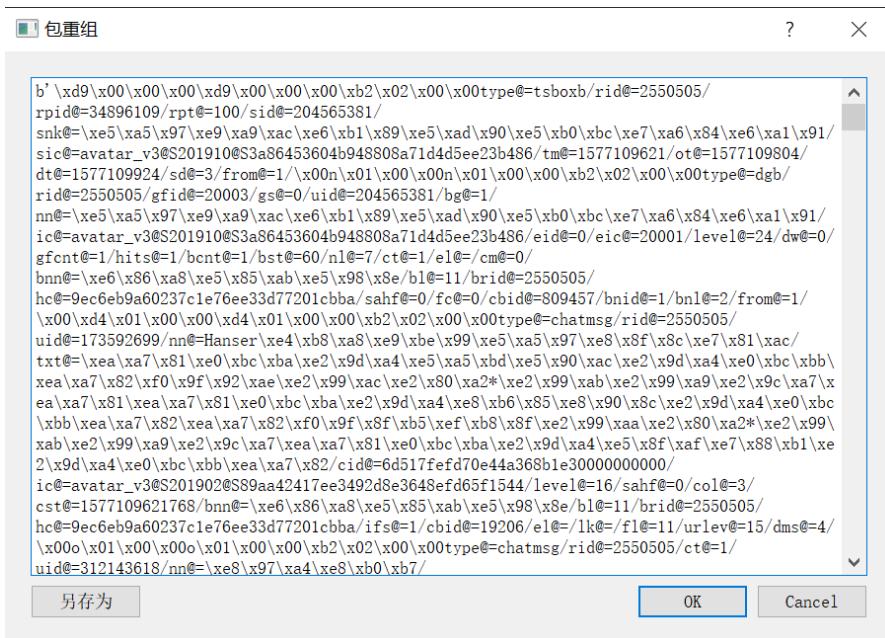


图 4.5: TCP 流重组结果

- 重组的数据可以选择保存为文件

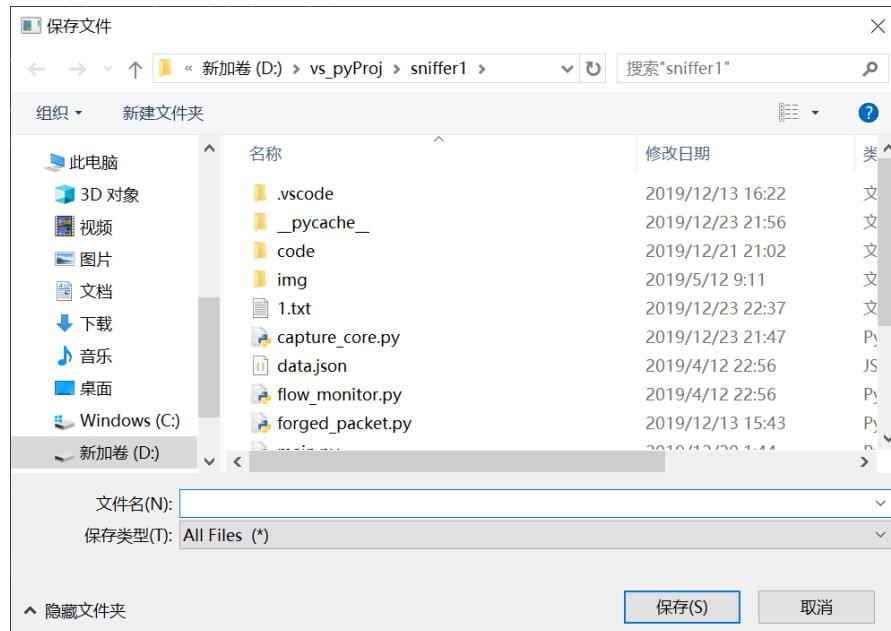


图 4.6: 指定保存路径

#### 4.1.5 包过滤功能

- 在抓包进行时和停止抓包后都可以在 filter 输入框中输入 filter 表达式进行包过滤，其表达式规则为：

*src\_ip ==< ip1 > | < ip2 > |...| < ipn >*

*dst\_ip ==< ip1 > | < ip2 > |...| < ipn >*

*sport ==< p1 > | < p2 > |...| < pn >*

*dport ==< p1 > | < p2 > |...| < pn >*

*prot ==< prot1 > | < prot2 > |...| < pron >*

- 其中每个选项间以“**&**”相连，不需要的项可以缺省。点击“应用”后 filter 改变 a

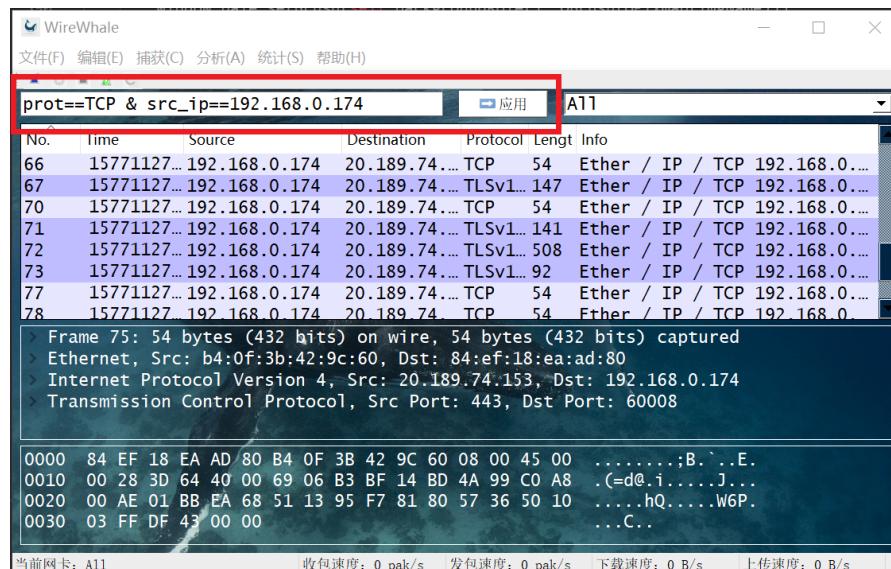


图 4.7: filter 应用

## 4.1.6 数据包的查询

- 数据查找只需要向 filter 中插入 keywords 选项即可，以括号作为识别边界

*keywords == (< content >)*

- 和其他 filter 选项也可用' & '来进行连接

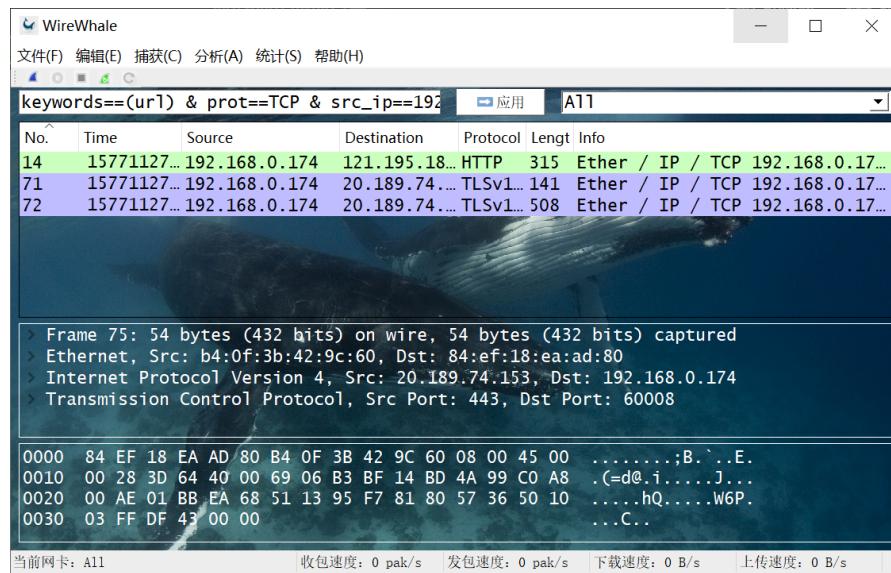


图 4.8: 数据包查询

## 4.1.7 数据包的保存与读取

### 4.1.7.1 数据包的保存

- 通过点击菜单栏中的选项进行手动保存

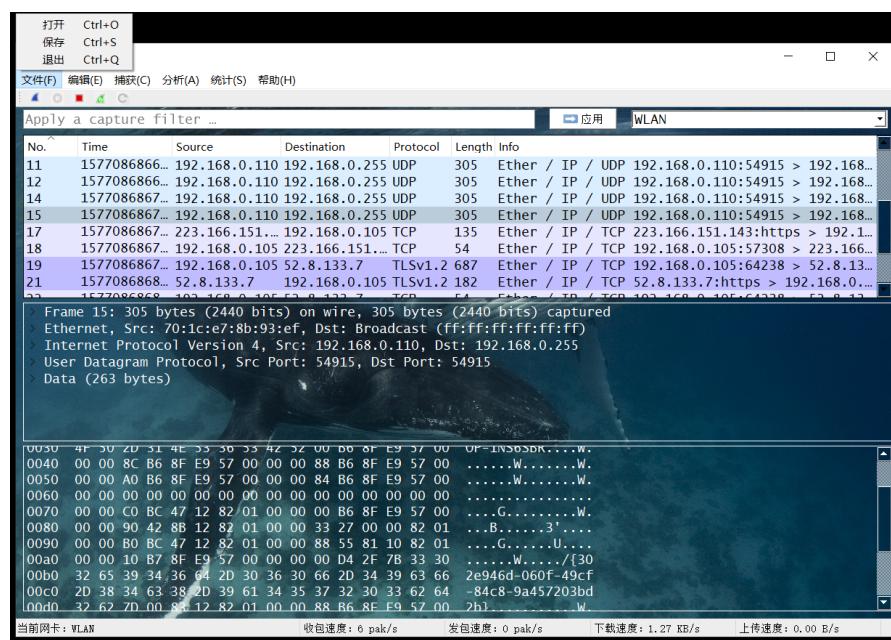


图 4.9: 手动保存

- 关闭程序时提示用户保存

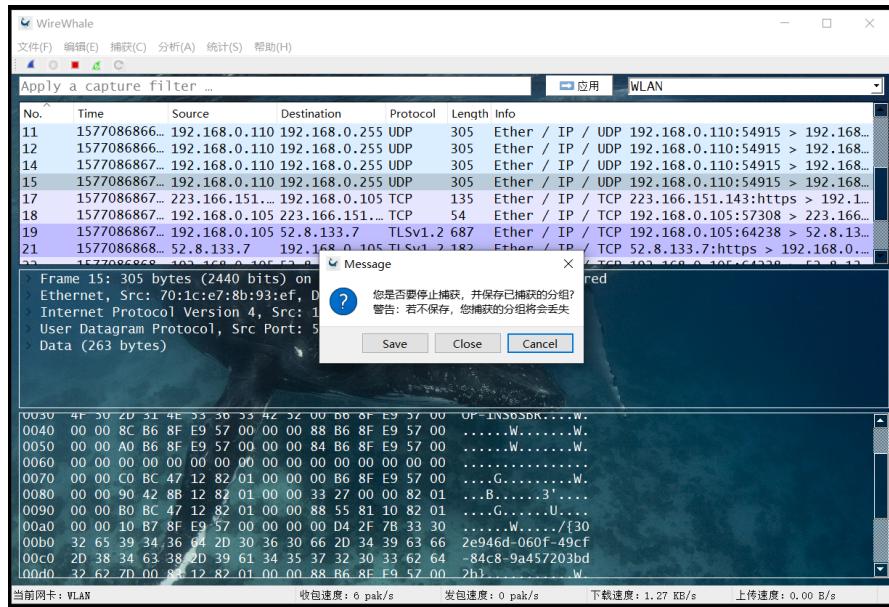


图 4.10: 提示保存

- 选择保存路径

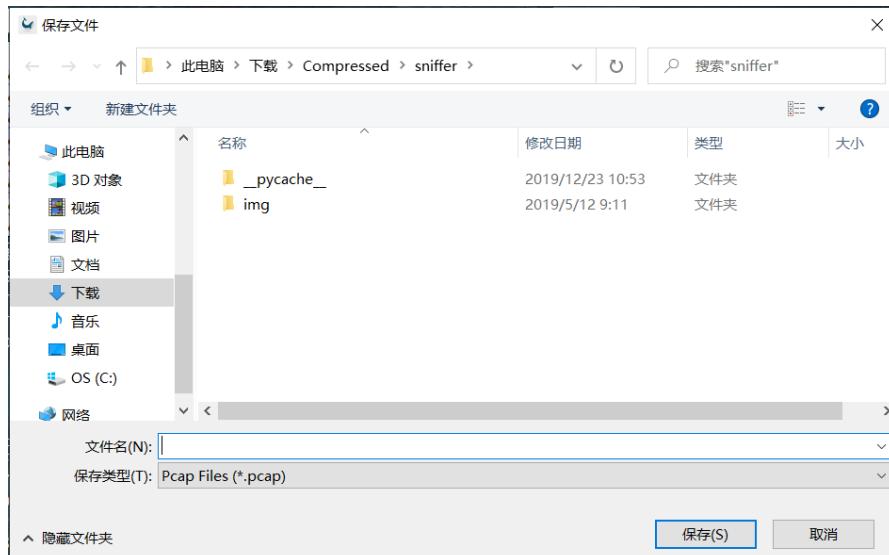


图 4.11: 指定保存路径

- 报错提示机制

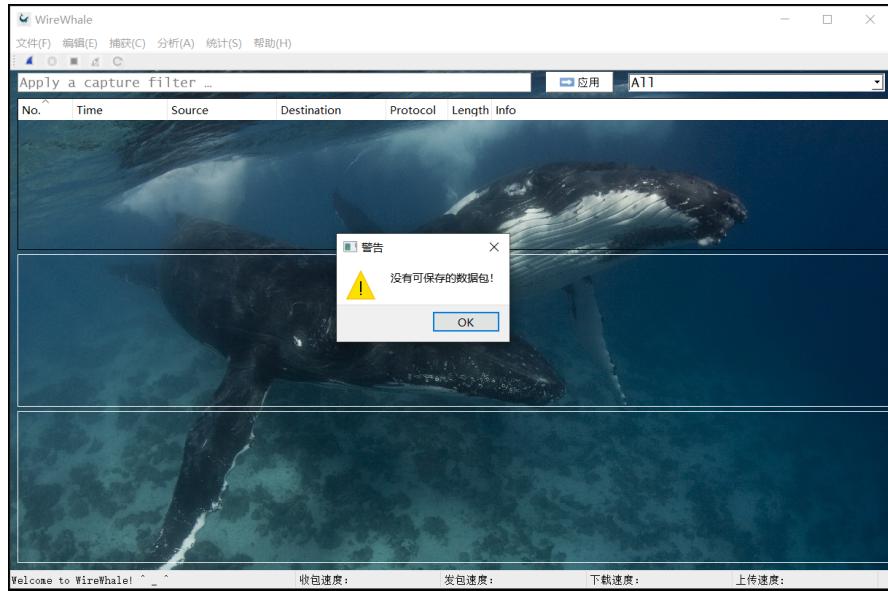


图 4.12: 空包保存报错

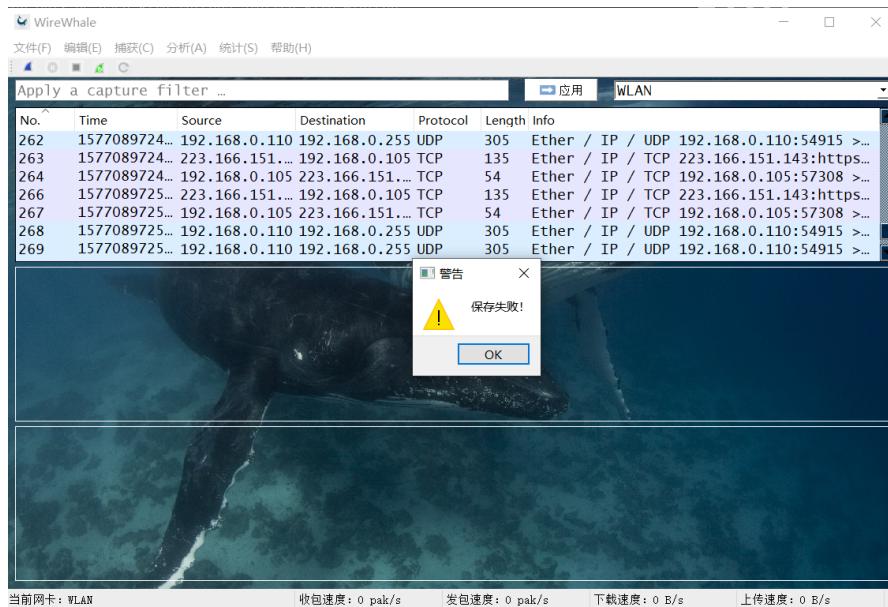


图 4.13: 保存失败

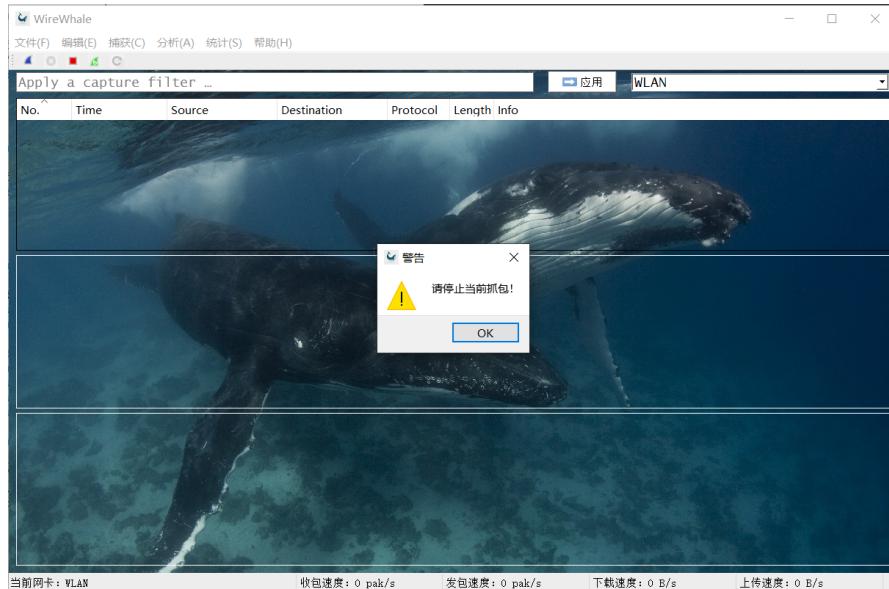


图 4.14: 保存前需停止抓包

#### 4.1.7.2 数据包的读取

- 菜单栏选择打开

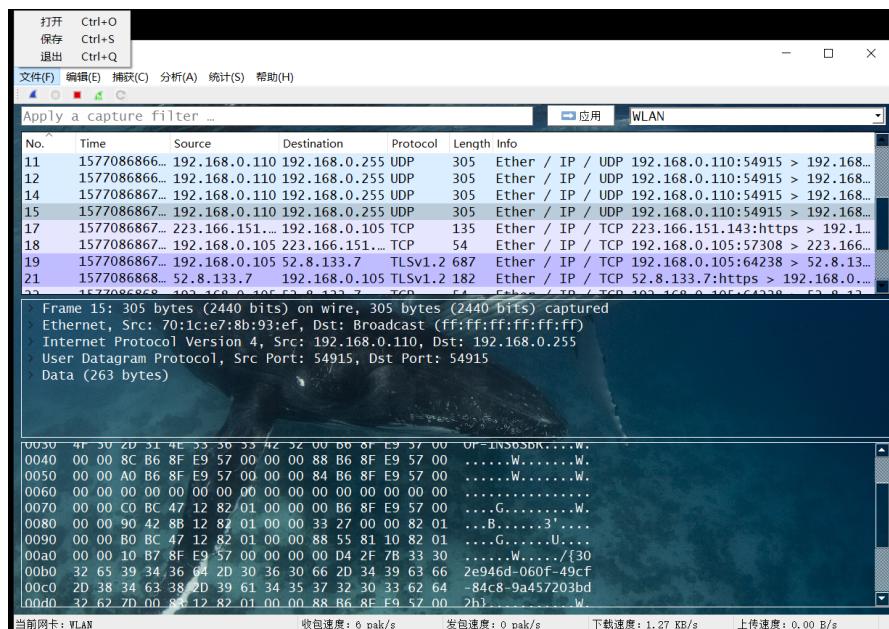


图 4.15: 菜单栏选择打开

- 选择打开文件路径

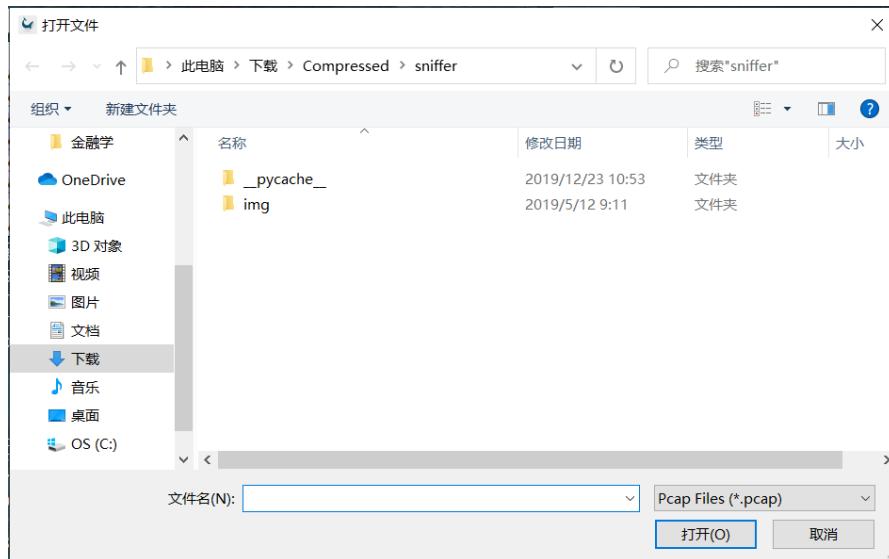


图 4.16: 选择打开文件路径

- 打开后的界面

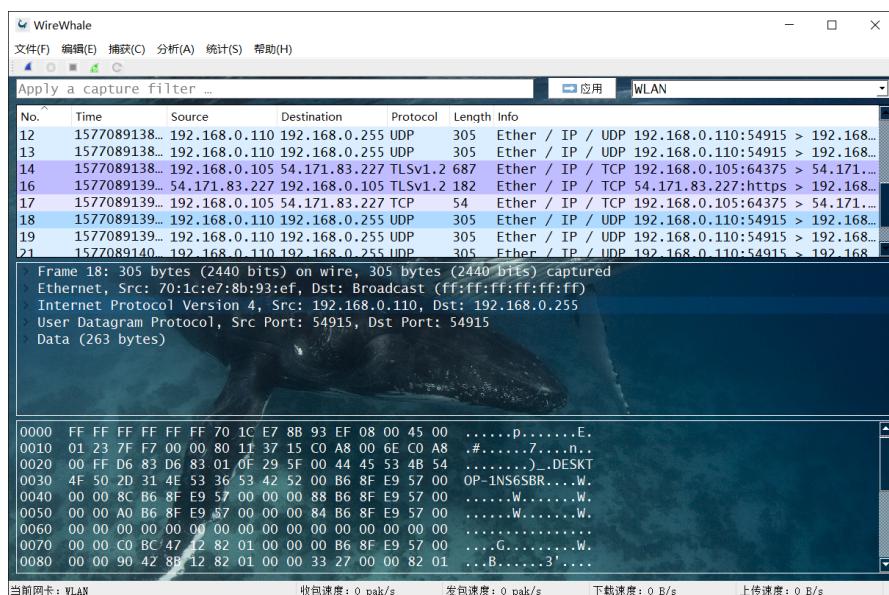


图 4.17: 打开后的界面

## 4.2 项目特色功能

### 4.2.1 高度可定制化界面

#### 4.2.1.1 修改字体

点击编辑可找到修改主窗口字体的选项

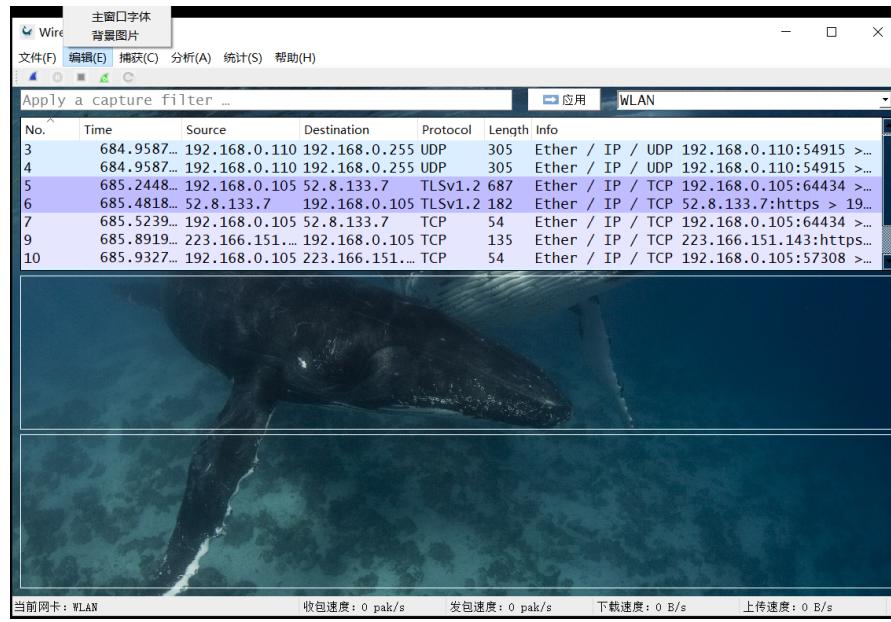


图 4.18: 在“编辑”中定制字体

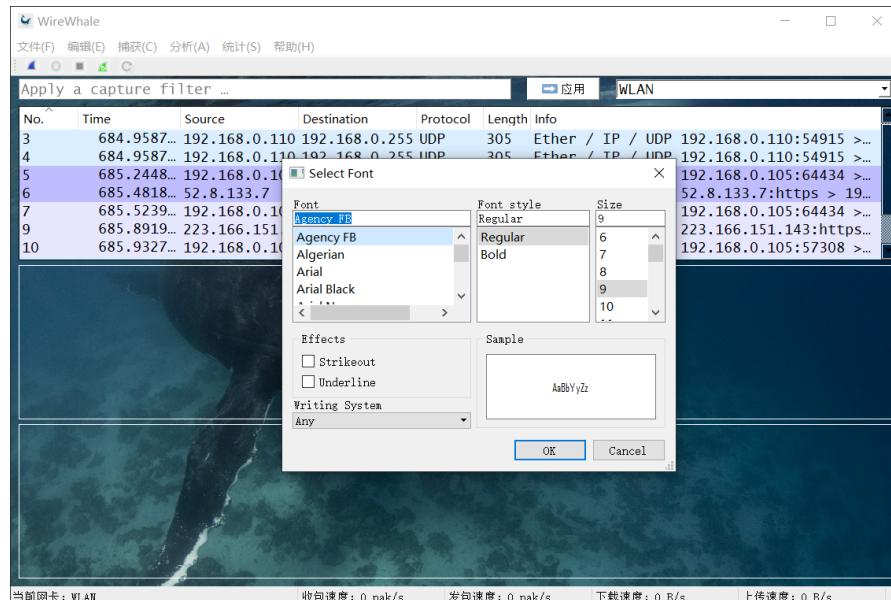


图 4.19: 改变字体

### 4.2.1.2 修改背景图片

点击编辑可找到修改背景的选项

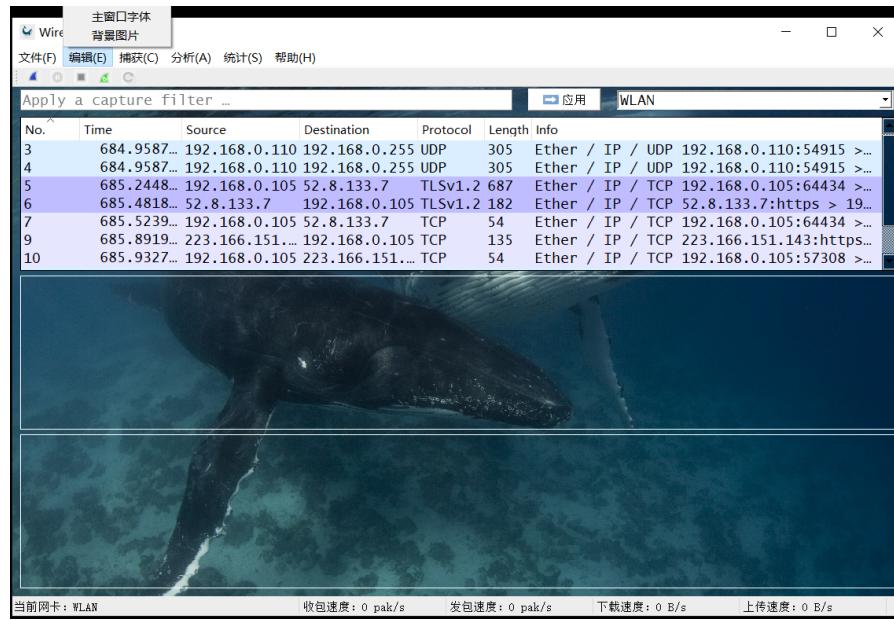


图 4.20: 在“编辑”中定制背景

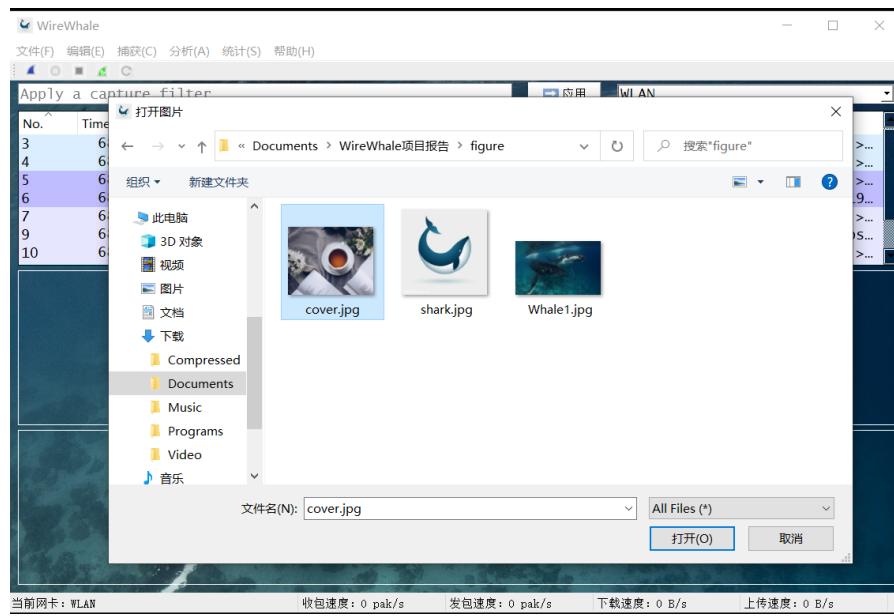


图 4.21: 改变背景

### 4.2.1.3 根据需要调整各模块窗口大小

- 可通过拖动窗口边沿调整各模块大小

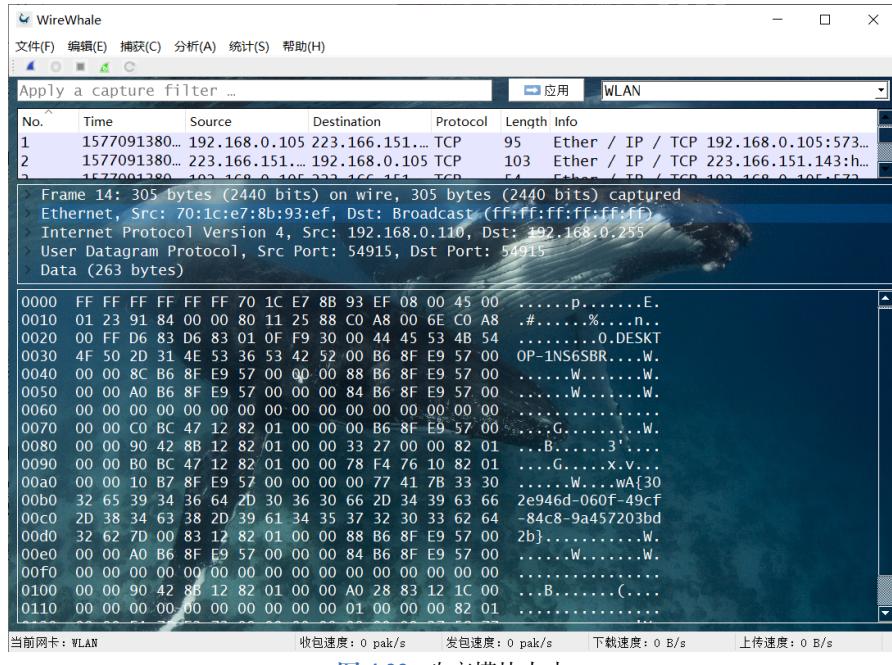


图 4.22: 改变模块大小

- 当模块边沿拖动至边界时将自动隐藏临近模块以提供更大的空间

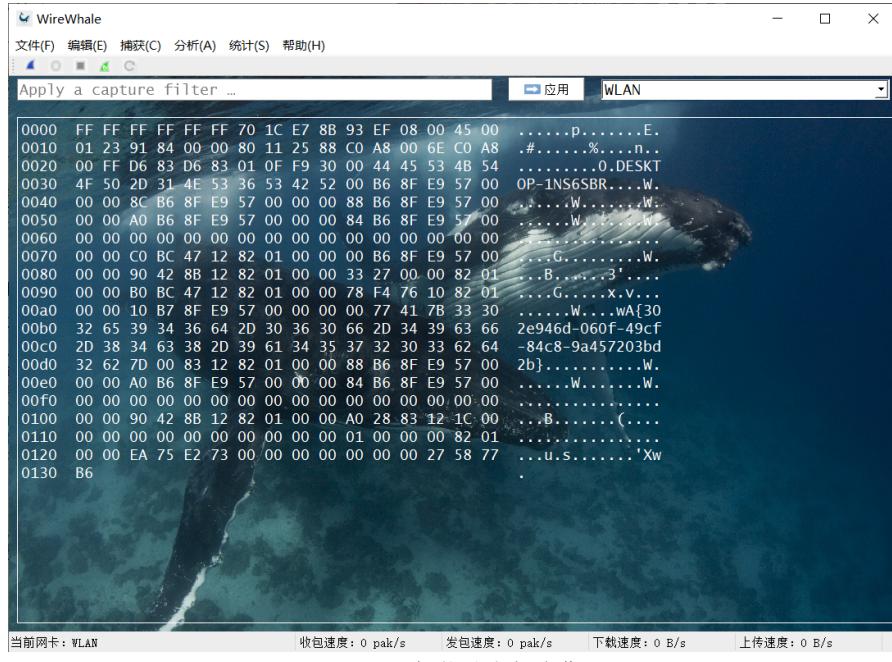


图 4.23: 智能贴靠与隐藏

## 4.2.2 用户友好的流量包伪造向导

- 在菜单栏的“分析”中选择伪造包

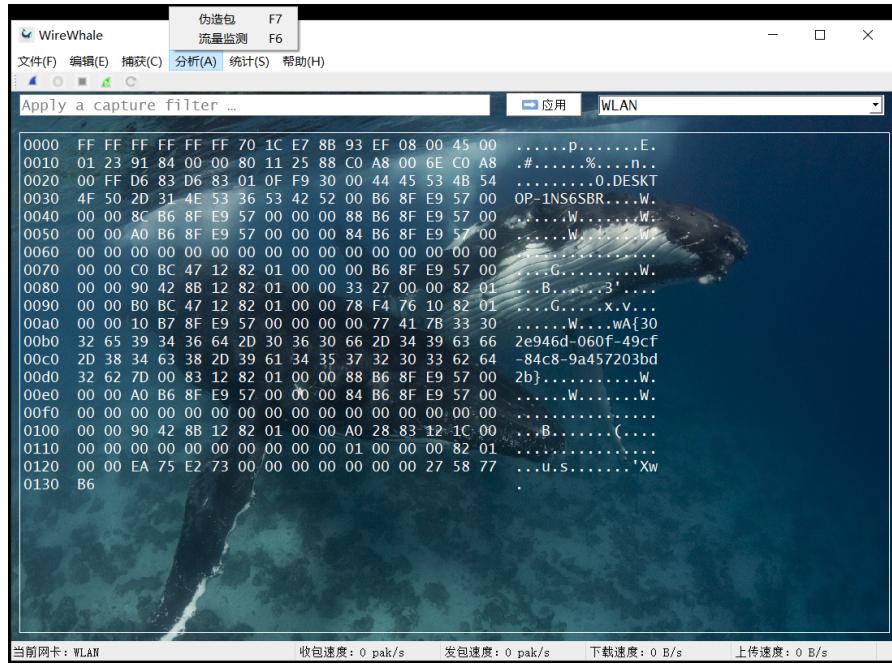


图 4.24: 选择伪造包

- 依照步骤填写相关内容即可完成流量包的伪造



图 4.25: 第一步

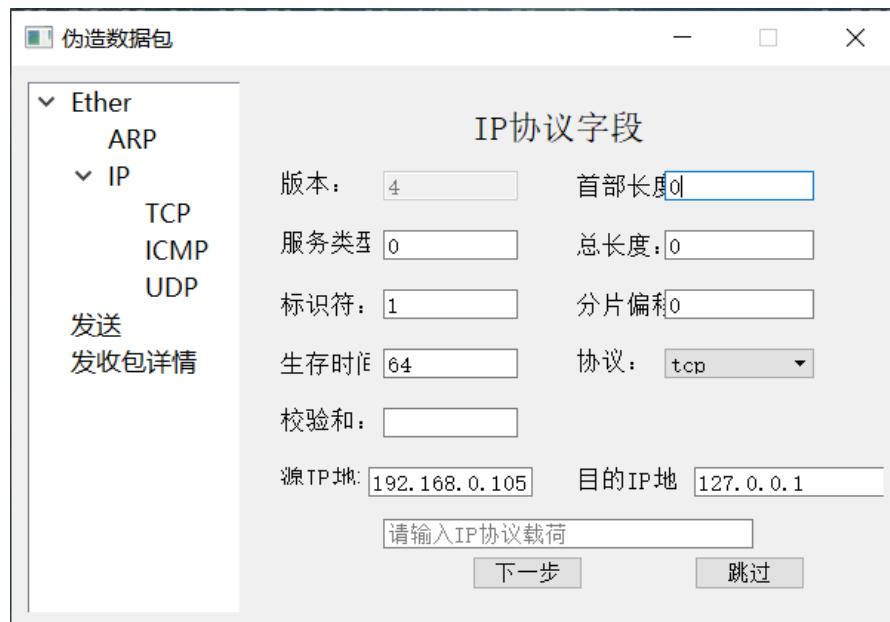


图 4.26: 第二步

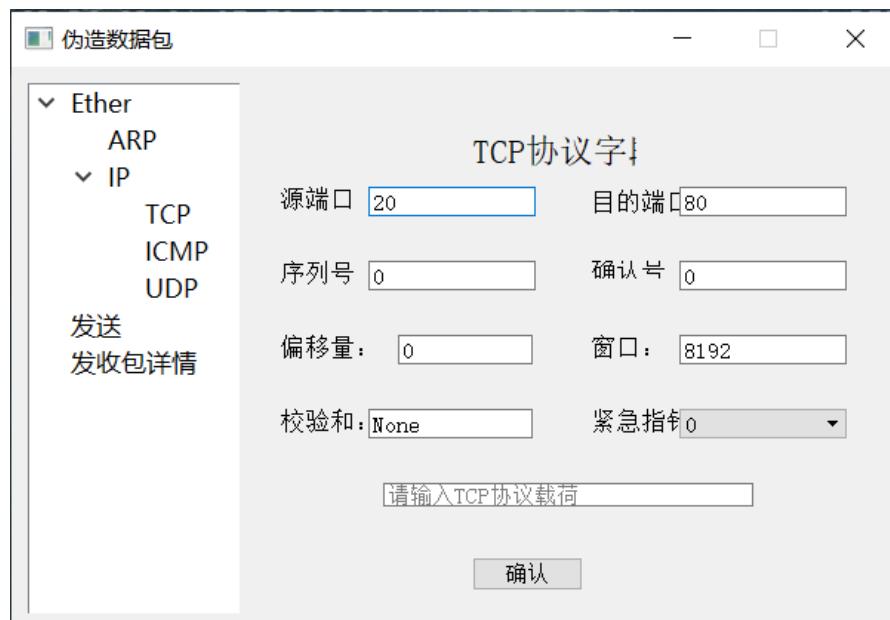


图 4.27: 第三步

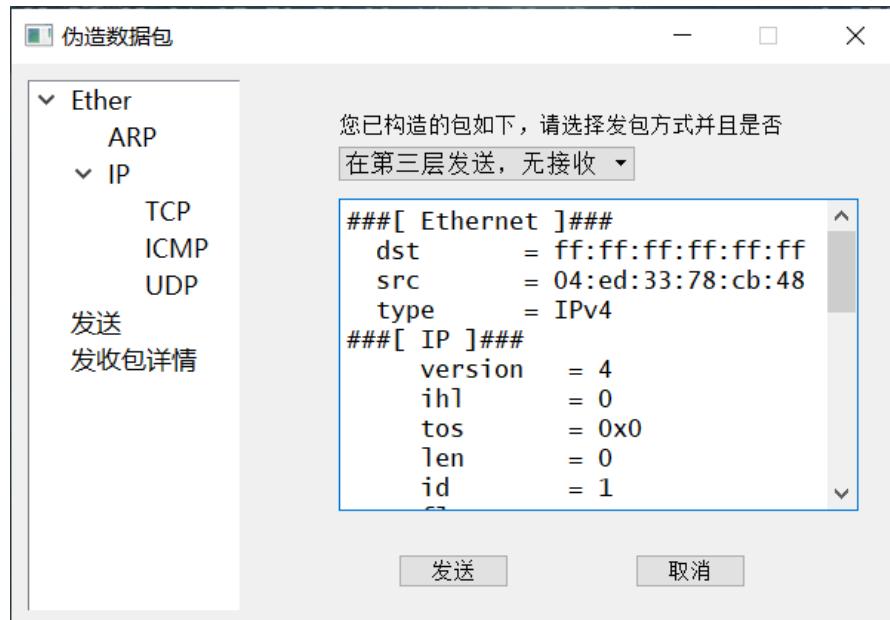


图 4.28: 第四步

### 4.2.3 可可视化的流量监测

- 在菜单栏的“分析”中选择流量监测，可设置流量预警线，当流量超过预警线时自动报警

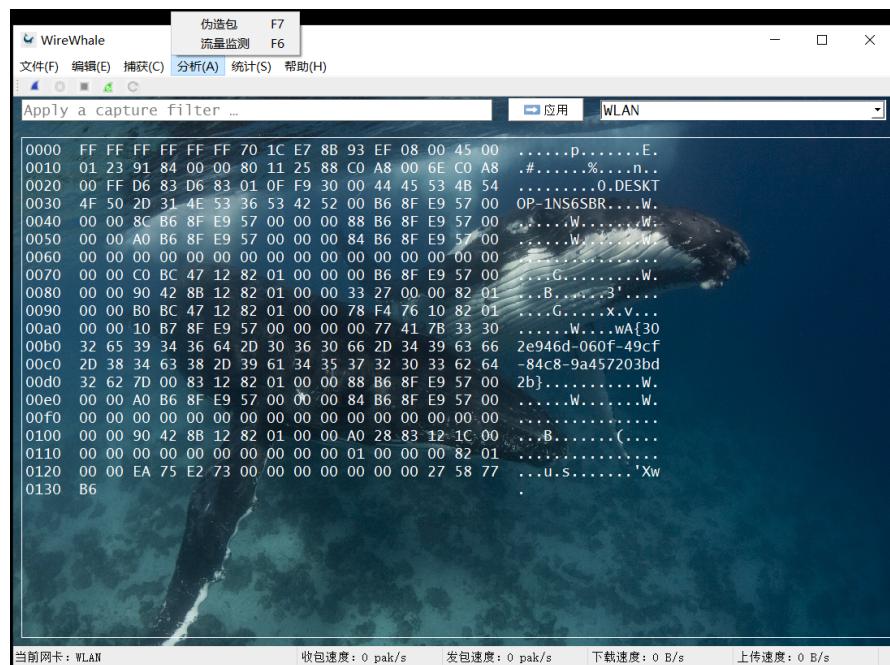


图 4.29: 选择流量监测

- 在流量监测界面可看到当前系统进程，选择系统进程可进行实时监测，此处我们选择 Edge 浏览器进行监测，同时进行网页浏览。红色为流量预警线

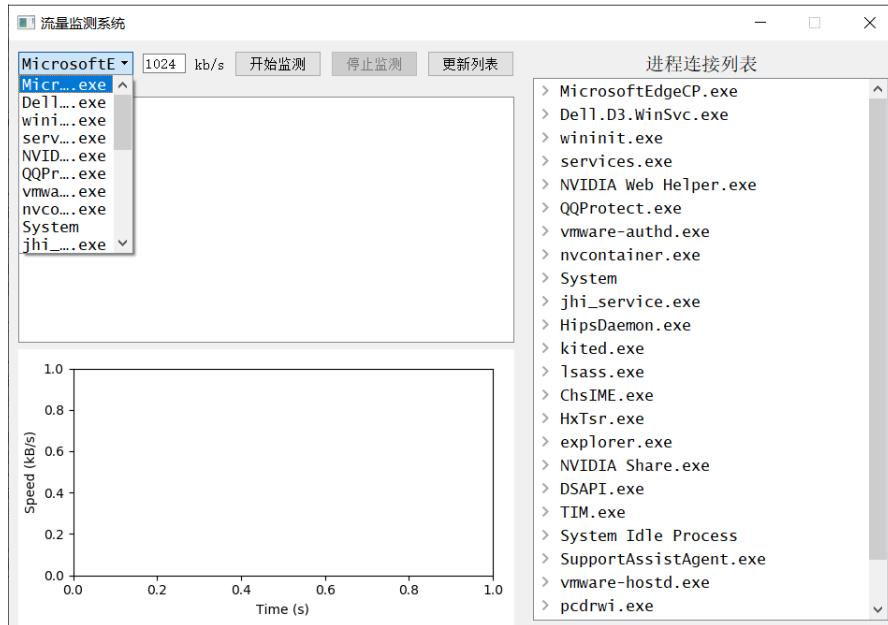


图 4.30: 选择进程开始监测

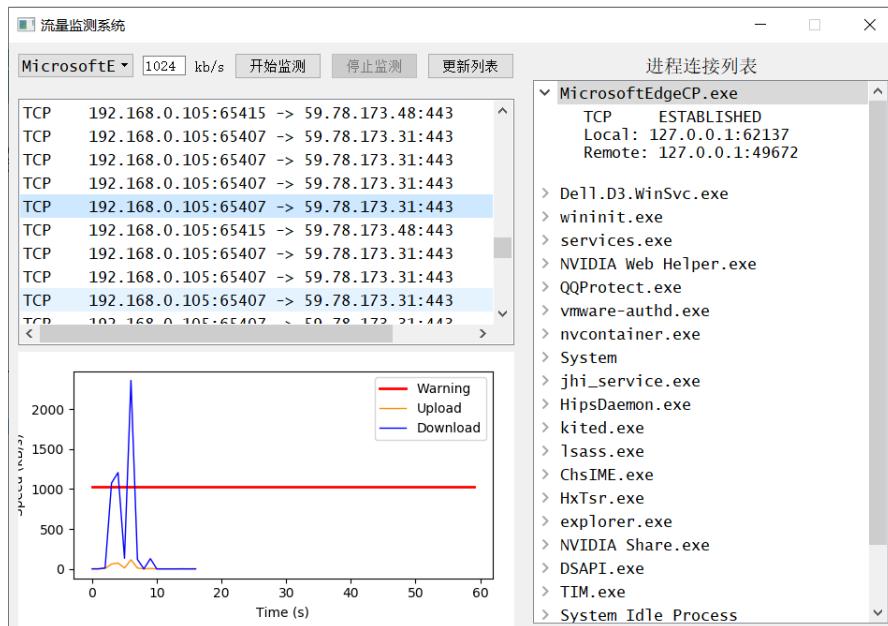


图 4.31: 实时监测结果

#### 4.2.4 可可视化的统计分析

- 在菜单栏的“统计”中选择对应的分析选项

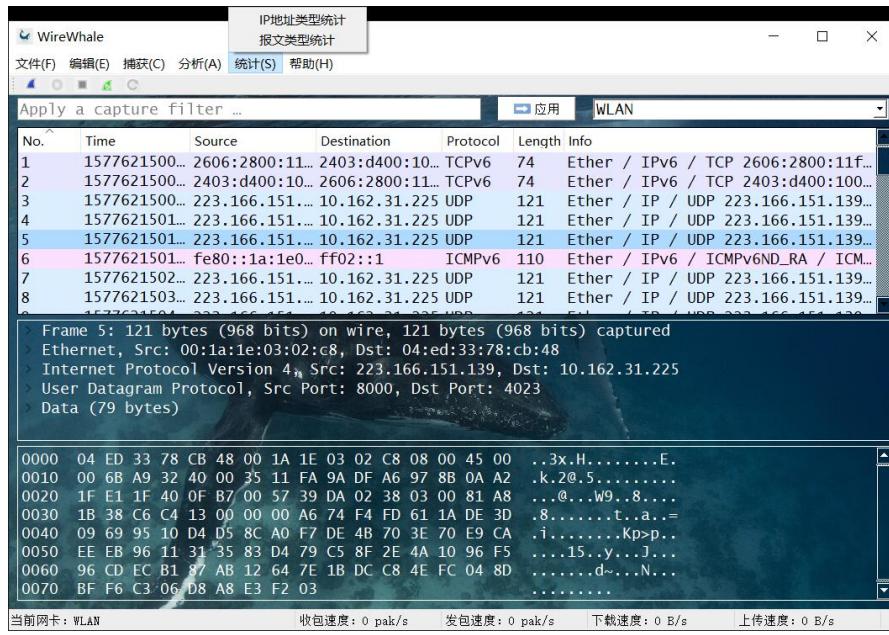


图 4.32: 选择相关选项进行分析

- 接下来我们便可以得到对所抓数据包的可视化分析结果，同时可以对所得的图表进行相关操作，辅助分析

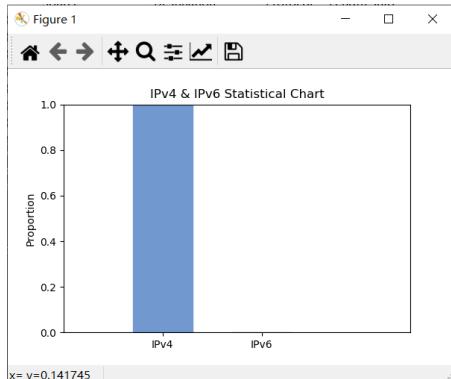


图 4.33: IP 地址类型统计

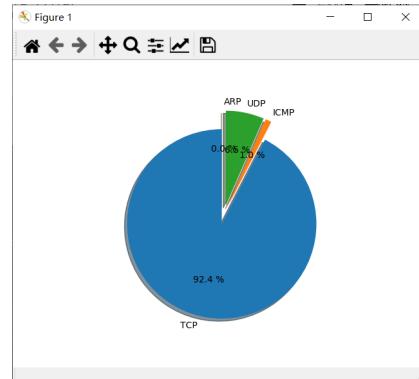


图 4.34: 报文类型统计

#### 4.2.5 内嵌帮助文档

- 点击“帮助”菜单栏可看到“帮助文档”和“关于”选项

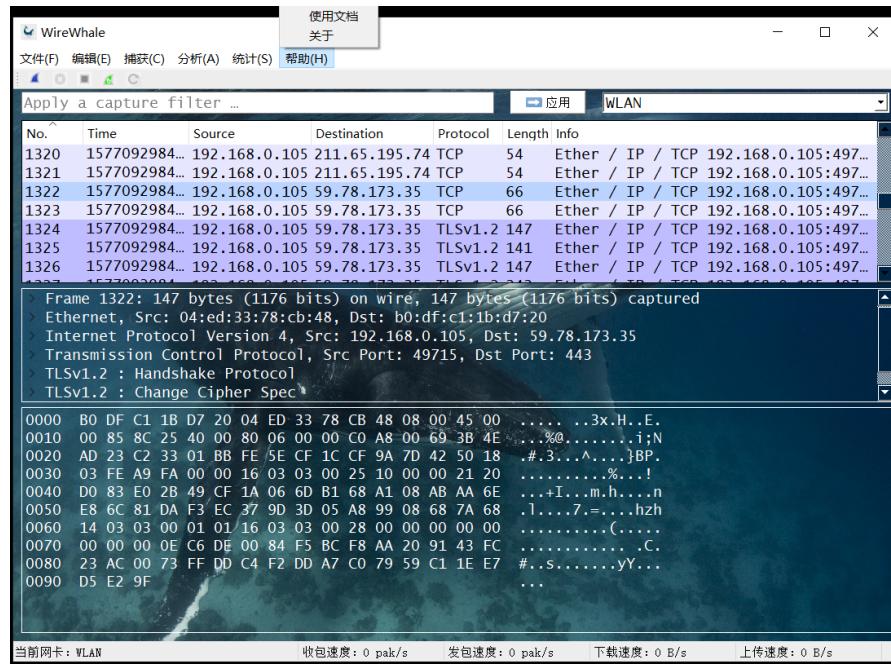


图 4.35: 选择帮助

- 选择“关于”可看到对本软件的功能概述，选择“帮助文档”则会打开本文档

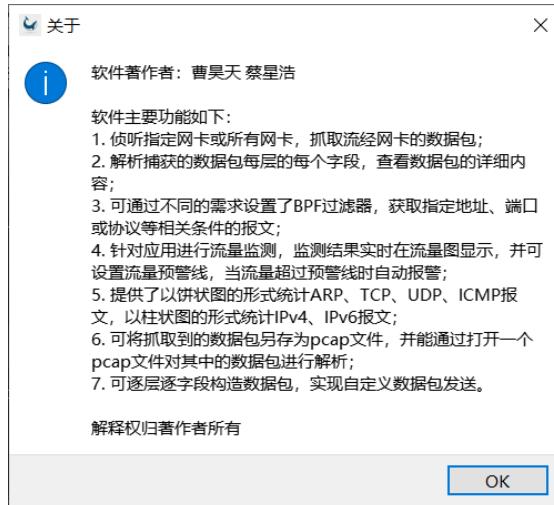


图 4.36: 关于

# 第5章 遇到的问题与解决方法

---

## 5.1 曹昊天

此次的项目开发中，我先分别完成了各功能模块的开发，再进行组合与界面设计。

在本次项目的开发过程中，我遇到的最大的问题就是 scapy 库细节信息的匮乏。不得不承认的是，scapy 库的功能的确十分强大，这点在我阅读官方文档是便有所体会。但当我深入开发时我发现，官方文档给出的介绍内容并不能满足我的开发所需，相关细节即使是上网搜索也无处可寻。比如在指定网卡这一功能模块的实现过程中，官方文档给出的只有以字符串的形式返回系统网卡信息的函数，我曾想通过对返回的字符串做信息提取后再使用相关接口进行操作的办法达到目的，但这种做法无疑牺牲了程序的执行效率，最后我决定阅读 scapy 库的源码，通过对源代码的阅读，理解相关函数的参数定义和实现过程。通过这种方式我找到了更好的实现方式，是对系统网卡的识别与选择变得更为流畅。

另外，由于先前没有足够的界面程序开发经验，我在本次的项目实践中边学边干，使用 PyQt5 进行软件界面的开发，当中也遇到了许多问题，比如线程的冲突、事件信号的传递等。不过好在网络这一方面的资料较为充足，我成功地完成了本次的开发工作。

## 5.2 蔡星浩

本项目中，我先对已有的 UI 进行改写和增添，再将 UI 进行事件绑定并实现逻辑功能。

由于这次是双人合作开发，比起单人开发有着一些额外的问题，例如事先未能完全约定好各个接口的使用模式，未能将项目的整个架构先加以细化后构建出框架图，这导致两人进行同时开发的效率较低。为此我们采用先后开发的模式，由曹昊天的同学进行软件基本框架的编写，在完成基础 UI 和功能后再由我添加其他功能选项，但这样依然遇到了一些问题。在我开始工作后，发现各个 UI 控件的控制命令以及事件绑定分布在代码的各个地方，这使得我花了一段时间进行代码的完整阅读和功能定位，之后在进行具体功能的编写。即使是这样，调试也花费了不少的时间。

另外，由于 Scapy 的参考资料极其缺乏，就连官方文档也没有给全大部分函数和参数的意义用法，于是只能通过阅读源码和英文缩写来进行试探，以了解其功能并使用，这对项目的进度造成了一定程度的延缓。

在编写 filter 的过程中，由于无法使用 sniff 函数自带的 filter 功能，于是自己编写了一组语法来接收过滤规则。一开始我使用接收到的表达式字符串直接进行判断过滤，但这样效率太低，每次都要重新从表达式中提取含义，造成资源浪费。于是将规则单独分离开，将规则设置为 capture\_core 类中的字典类型成员变量，再专门编写一个由表达式产生规则的方法用于产生规则，这样便能只在 filter 更新时重新计算规则，在过滤包时就可直接判断了。

为了准确的重组 TCP 数据段，我采用了以 SYN 标识数据包和 FIN 标识数据包作为接受 TCP 流的首位，在获取到要追踪的数据包的信息后，以套接字作为标识，扫描整个已抓包的列表，将符合套接字的包筛选出来，随后在一个列表中按数据起始序号 SEQ 为序将抓到的包进行排列，并丢弃掉重发送的包。当已抓包扫描完毕或者抓到的有效数据长度等于连接释放和建立时的序号之差时，结束扫描，将已有的数据按字段序号进行拼接。

# 第6章 个人体会与建议

---

## 6.1 曹昊天

### 6.1.1 个人体会

在本次的项目中，我负责完成了软件界面的开发，倾听网卡的选择、数据包内容的解析与呈现、软件定制化功能的增加、IP地址类型分析与报文类型分析的可视化、流量监控与预警等功能的实现。

这些开发过程使我对多种网络协议有了更为深刻的理解与体会，也使我对一个软件开发过程有了更为深入的了解。给我感触最深的有两个方面：一是在合作开发的过程中开发者们需要实现协商软件内容的架构与功能模块的逻辑，这样可以在合作开发时节约大量的时间；二是我们在开发时不能只是对前人已经造好的轮子进行简单的拼装，更要深入理解这些功能模块的实现过程，这样才能有所收获。

经过本次开发，我深刻认识到了打好理论基础的重要性。老师在课堂上讲述的理论知识使我在进行开发的过程中对自己所要做的内容有着明确的方向，不仅大大提高了我在开发过程中的效率，也使得我的理论知识更为巩固。在这里感谢老师与助教的辛苦付出！

### 6.1.2 课程建议

希望老师能够增加一些项目实践的内容，丰富我们的项目选择，且可以提早一些布置下来。我本意是想开发一个集抓包、分析、嗅探、扫描、伪造、攻击于一体的综合性软件，但是由于后半学期时间上不是特别充裕，故只是主要实现了抓包功能，辅以简单的伪造包和分析功能，这对我而言也是个小小的遗憾。之后有时间我也会继续对这个项目进行维护

## 6.2 蔡星浩

### 6.2.1 个人体会

在本次项目中，我负责的主要的是拓展功能的添加，包括包过滤、数据包查询、TCP流数据包的文件重组以及数据包的保存与读取。

这一次我选择了双人合作项目，这是我第一次与他人一起完成一个完整的软件项目，比起单人开发，双人开发想要达到高的开发效率，工程开始前的规划和约定是必不可少的，这次项目我深深的体会到了这一点。我觉得自己需要一些关于软件工程开发规范的学习。这次的项目模块化的效果不算是特别好，有不少地方需要修改模块内部的东西，这也是应该极力避免的。

另外，自己编写包过滤和TCP数据流重组等功能时，更加深刻的理解了所学的各个协议各个报文字段的含义，了解到了这些协议原理的必要之处。

通过这一次的项目经历，我也更认识到自己理论知识其实还不够完善，有许多想要实现的其他功能因为自身水平和时间限制而没能完成，有些已有的功能也没有做到特别完善。因此，在之后我还会继续精进自己在这方面的学习，并将这个项目做得更加完善化。

### 6.2.2 课程建议

这门课程我觉得实践占据更主要的部分会更好，可以把大作业的部分放到学期开始就进行，贯穿整个学期来进行开发，这样可以增添更多内容，也能让同学们在开发中更好的掌握知识，项目的可选范围也可以再拓宽一些。这次开发我感到了时间的不充裕，因为已是后半学期，大作业集中，完成的效果也会有所欠缺，如果将开发时间放宽到整个学期，也能让同学们在前半学期也能得到实践上的锻炼。