

网络与系统安全

课程设计指导手册

《基于 TCP 流重组的软件行为分析》

(v6)

华中科技大学网络空间安全学院

二零二五年二月

目 录

1	课程简介	1
1.1	课设目的	1
1.2	开发环境和分析工具	1
1.3	课设内容	1
2	TCP 流重组参考	3
2.1	libpcap 基础.....	3
2.2	报文协议解析	7
2.3	FTP 文件传输	11
2.4	TCP 数据流重组	13
3	软件行为分析参考	15
3.1	逆向分析基础与工具使用	15
3.2	常见 Linux 程序恶意代码与漏洞表征.....	18
3.3	二进制程序模拟执行与调试工具	23
4	拓展思考	26
4.1	在线重组	26
4.2	乱序重组	26
4.3	自动化动态漏洞检测	26
4.4	基于虚拟机的启发式恶意代码检测	26
4.5	系统保护绕过	26

1 课程简介

1.1 课设目的

恶意软件通常会伪装成合法软件，并通过网络通信进行传播。TCP 协议为应用程序提供了可靠传输机制，是网络通信使用的主要传输层协议。通过对网络中传输的 TCP 流进行重组，还原出所传输的软件，并对软件的行为进行分析，以检测和识别出恶意软件的传播，进而阻止通信的进行，从而将恶意软件防范的关口从主机前移至网络，能够有效降低成本提高效率。

本课程设计是学生学习了《软件安全》、《计算机网络安全》课程后的集中实践，目的是通过对所学理论知识在相关场景中的综合应用和系统设计，掌握网络流量安全检测的基本方法和技术。具体目标包括：

- 掌握使用 libpcap 函数库对网络通信报文进行解析的方法；
- 掌握 TCP 协议的可靠传输机制及流重组的原理；
- 掌握 FTP 协议的基本原理与工作过程；
- 掌握 Linux 应用程序逆向分析方法
- 掌握 Linux 应用程序常见漏洞和恶意行为基本原理
- 掌握 Linux 应用程序动态分析与调试方法

1.2 开发环境和分析工具

1.2.1 操作系统

建议使用 Ubuntu 22.04。

1.2.2 相关工具

gcc 编译器；

libpcap 库；

QEmu 仿真器

IDA 或者 Ghirda 逆向分析工具

1.3 课设内容

本课程设计提供一个样本软件以及使用 FTP 传输该样本软件时截包保存的 pcap 文

件，要求学生[独立](#)完成以下工作：

1.3.1 TCP 流重组

设计开发一个程序：

（1）使用 libpcap 库对 pcap 文件进行报文解析，提取出 TCP 报文段，并分析出其中的 FTP 会话的控制连接报文；

（2）对 FTP 控制连接中的消息进行分析，提取该会话中传输的文件名称以及数据连接的参数；

（3）找到 FTP 数据连接的报文段，提取报文段的 TCP 数据部分，按照 TCP 可靠传输的字节流机制重组还原出所传输的文件；

（4）将重组还原的文件，与样本软件进行对比，验证所设计程序 TCP 流重组的准确性。

1.3.2 软件行为分析

（1）使用 IDA Pro 或者 Ghirda 等二进制逆向分析工具手动分析样本软件中的恶意行为和漏洞，要求至少找到存在漏洞的函数名称、地址、漏洞类型、漏洞成因以及存在恶意功能的函数名称、地址和恶意功能的具体行为危害信息。

（2）编写 IDA Pro 或者 Ghirda 等二进制逆向分析工具插件自动化检测样本软件中的恶意行为和漏洞（自动化输出（1）中的信息）

（3）使用 IDA Pro 或者 Ghirda 等二进制逆向分析工具手动分析出恶意行为和漏洞的触发条件，并在基于 QEMU 的虚拟环境中进行动态验证。（人工分析即可不需要实现工具自动化）

2 TCP 流重组参考

2.1 libpcap 基础

libpcap (Packet Capture Library) 是一个用于捕获网络数据包的 C 函数库，广泛应用于网络嗅探、数据抓取和协议分析等场景。它提供了跨平台的接口，用于捕获经过指定网络接口的数据包，并支持数据包过滤和流量统计等功能。

2.1.1 libpcap 安装

在大多数 Linux 发行版中，可以直接通过包管理器安装 libpcap 库。例如，在 Ubuntu 系统中，可以使用以下命令安装：

```
sudo apt-get install libpcap-dev
```

安装完成后，需要在编写程序时包含<pcap.h>头文件，并在编译时链接-lpcap 库。

2.1.2 实时捕获数据

2.1.2.1 查找网络设备

pcap_lookupdev()函数用于查找可用的网络设备。示例如下：

```
char errbuf[PCAP_ERRBUF_SIZE];
const char *device = pcap_lookupdev(errbuf);
if (device == NULL) {
    fprintf(stderr, "Error: %s\n", errbuf);
    return 1;
}
printf("Device: %s\n", device);
```

2.1.2.2 打开网络设备

pcap_open_live()函数用于打开指定的网络设备，参数包括设备名称、捕获数据包的最大字节数、是否启用混杂模式、超时时间等。示例如下：

```
pcap_t *handle = pcap_open_live(device, 65536, 1, 1000, errbuf);
if (handle == NULL) {
    fprintf(stderr, "Error opening device: %s\n", errbuf);
    return 1;
}
```

2.1.2.3 设置过滤器

`pcap_compile()`和 `pcap_setfilter()`函数用于编译和设置数据包过滤规则，这个步骤是可选的。示例如下：

```
struct bpf_program fp;
char filter_exp[] = "tcp port 80";
if (pcap_compile(handle, &fp, filter_exp, 0, net) == -1) {
    fprintf(stderr, "Error: %s\n", pcap_geterr(handle));
    return 1;
}
if (pcap_setfilter(handle, &fp) == -1) {
    fprintf(stderr, "Error: %s\n", pcap_geterr(handle));
    return 1;
}
```

2.1.2.4 捕获数据

`pcap_next()`函数用于捕获单个数据包，可以通过循环调用持续捕获。示例如下：

```
const unsigned char *packet;
struct pcap_pkthdr header;
packet = pcap_next(handle, &header);
if (packet != NULL) {
    printf("Packet captured: %d bytes\n", header.len);
}
```

2.1.2.5 关闭网络设备

`pcap_close()`函数用于关闭网络设备并释放资源。示例如下：

```
pcap_close(handle);
```

2.1.2.6 简单示例

以下是一个简单的 `libpcap` 实时抓包程序示例，用于捕获并打印网络接口上的数据包信息，注意源文件中包含`<pcap.h>`头文件。

```
#include <pcap.h>
#include <stdio.h>

int main() {
    char errbuf[PCAP_ERRBUF_SIZE];
    const char *device = pcap_lookupdev(errbuf);
    if (device == NULL) {
        fprintf(stderr, "Error: %s\n", errbuf);
    }
}
```

```

        return 1;
    }

    pcap_t *handle = pcap_open_live(device, 65536, 1, 1000, errbuf);
    if (handle == NULL) {
        fprintf(stderr, "Error opening device: %s\n", errbuf);
        return 1;
    }

    const unsigned char *packet;
    struct pcap_pkthdr header;
    while ((packet = pcap_next(handle, &header)) != NULL) {
        printf("Packet captured: %d bytes\n", header.len);
    }

    pcap_close(handle);
    return 0;
}

```

使用以下命令编译上述程序，注意需要链接-lpcap 库。

```
gcc -o simple_sniffer simple_sniffer.c -lpcap
```

运行程序时需要使用 sudo 以获取网络接口的访问权限。

```
sudo ./simple_sniffer
```

2.1.3 离线报文分析

除了实时捕获数据包，libpcap 还支持离线分析，即从 wireshark、tcpdump 等软件保存的 pcap 文件中读取和处理数据包。本课程的任务是从离线文件中重组传输的文件，下面对 libpcap 的离线处理流程进行介绍。

2.1.3.1 打开离线数据文件

libpcap 提供了 pcap_open_offline() 函数，用于打开已保存的 pcap 文件进行离线分析，其函数原型为：

```
pcap_t *pcap_open_offline(const char *fname, char *errbuf);
```

其参数为：

fname 指定要打开的文件名，该文件应该是以 tcpdump/libpcap 格式保存的数据包文件；

errbuf 是一个字符串缓冲区，用于存储错误信息。如果函数调用失败，错误信息会被写入这个缓冲区。

函数调用成功时返回一个 `pcap_t` 类型的指针，这是用于后续操作的捕获描述符；失败时返回 `NULL`，并且错误信息会被写入 `errbuf`。

2.1.3.2 读取和处理数据包

打开离线文件后，可以使用 `pcap_next()` 或 `pcap_loop()` 函数读取数据包。`pcap_next()` 前面章节已经介绍。以下主要介绍 `pcap_loop()`，它通过回调函数逐个处理数据包，直到满足指定条件，其函数原型为：

```
int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user);
```

其参数为：

`p` 是由 `pcap_open_live()` 或 `pcap_open_offline()` 返回的 `pcap_t` 捕获描述符；

`cnt` 用于设置所捕获数据包的个数，如果为 -1 或 0，表示无限循环，直到达到文件末尾或调用 `pcap_breakloop()` 停止；

`callback` 是回调函数指针，用于处理每个数据包；

`user` 是留给用户使用的，当 `callback` 被调用的时候这个值会传递给 `callback` 的第一个参数（也叫 `user`）。

其中 `callback` 回调函数的原型为：

```
void pcap_callback(u_char *user, const struct pcap_pkthdr *header, const u_char *packet);
```

`callback` 函数的第一个参数 `user` 指针是可以由用户使用的，如果你想给 `callback` 传递自己参数，那就只能通过 `pcap_loop` 的最后一个参数 `user` 来实现了

`callback` 函数的第二个参数 `header` 是一个结构体指针，这个结构体是由 `pcap_loop` 填充的，包含一些当前数据包的信息，其定义如下：

```
struct pcap_pkthdr {
    struct timeval ts;    /* 数据包捕获的时间戳 */
    bpf_u_int32 caplen;   /* 数据包捕获长度，可能小于数据包实际长度（例如由 snaplen 限制） */
    bpf_u_int32 len;      /* 数据包实际长度 */
};
```

`callback` 函数的最后一个参数 `packet` 指向一块内存空间，这个空间中存放的就是当前数据包的二进制数据，是我们后续要分析处理的对象。

2.1.3.3 简单示例

以下是一个完整的代码示例，展示如何从 `pcap` 文件中读取每个数据包并打印其长度：

```
#include <stdio.h>
```



```

#include <pcap.h>

// 回调函数，用于处理每个数据包
void packet_handler(u_char *user, const struct pcap_pkthdr *header, const u_char *packet)
{
    printf("Packet length: %d bytes\n", header->len);
    // 在这里可以进一步处理数据包
}

int main(int argc, char **argv)
{
    char errbuf[PCAP_ERRBUF_SIZE];

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <capture file>\n", argv[0]);
        return 1;
    }

    pcap_t *handle = pcap_open_offline(argv[1], errbuf); // 打开离线文件
    if (handle == NULL) {
        fprintf(stderr, "Error opening capture file: %s\n", errbuf);
        return 1;
    }

    printf("Processing packets from file: %s\n", argv[1]);
    pcap_loop(handle, 0, packet_handler, NULL); // 处理所有数据包

    pcap_close(handle);
    return 0;
}

```

2.2 报文协议解析

libpcap 捕获的网络数据包是以二进制数据形式存储的。为了理解这些数据包的内容，需要对其进行协议解析，即根据网络协议的规范，将二进制数据解析为可读的格式，从而提取出有用的信息。如前所述，`callback()`函数通过最后一个参数 `packet` 得到 `pcap_loop()`函数从 `pcap` 文件中读取的数据包的二进制数据。这个二进制数据，是从数据链路层的以太帧首部开始的，其与 TCP/IP 协议栈各层协议封装的对应关系如下所示：

02 42 a8 2e dc ae 02 42 0a 00 02 07 08 00 45 00 00 3c f8 4d 40 00 40 06 2a 60 0a 00

以太网首部	IP 首部	TCP 首部	应用层消息
-------	-------	--------	-------

协议解析就要在二进制数据中找到对应协议的首部位置，并按照首部的格式对数据进行对应提取。在 c 语言编程中，主要通过定义相关首部的数据结构，将数据结构指针指向二进制数据的对应字节，通过强制类型转换实现对数据格式的指定和内容的访问。

2.2.1 解析以太网首部

libpcap 并未提供各类网络协议的首部结构定义，我们要进行网络协议解析，需要自行定义协议数据结构。

以太网首部包含目的 MAC 地址、源 MAC 地址和协议类型等信息，我们将其数据结构定义为：

```
//以太网首部
typedef struct ethhdr {
    __u8 h_dest[6];        //目的 MAC 地址
    __u8 h_source[6];     //源 MAC 地址
    __u16 h_type;          //帧类型
} EthHdr_t;
```

以下是一段解析以太网首部的代码示例：

```
EthHdr_t *eth = (EthHdr_t *) packet;

printf("** Ether Header **\n");
printf("Dest   MAC: %02X:%02X:%02X:%02X:%02X:%02X\n", eth->h_dest[0] , eth->h_dest[1] , eth->h_dest[2] , eth->h_dest[3] , eth->h_dest[4] , eth->h_dest[5]);
printf("Source MAC: %02X:%02X:%02X:%02X:%02X:%02X\n", eth->h_source[0] , eth->h_source[1] , eth->h_source[2] , eth->h_source[3] , eth->h_source[4] , eth->h_source[5]);
printf("Frame Type: 0x%04X(%s)\n\n", ntohs(eth->h_type), ((ntohs(eth->h_type) == 0x0800) ? "IP" : "Other"));
```

其中 h_type 字段指明了以太网封装的网络层报文类型，取值为 0x0800 时，表示网络层报文为 IP 报文。

2.2.2 解析 IP 首部

如果以太网首部的协议类型为 IPv4（0x0800），则其后紧接的就是 IP 首部。IP 首部包含 20 字节的固定部分和不定长的选项部分，我们按照 IP 协议的规范，将其数据结构定义如下：

```
//IP 数据报首部
```

```

typedef struct iphdr {
#ifdef __BIG_ENDIAN__
    __u8 version:4, ihl:4; //版本,报头长度
#else
    __u8 ihl:4, version:4; //版本,报头长度
#endif
    __u8 tos;           //服务类型
    __u16 tot_len;       //总长度
    __u16 id;           //标识
    __u16 frag_off;      //标志+片偏移
    __u8 ttl;           //生存周期
    __u8 protocol;       //协议类型
    __u16 check;         //头部校验和
    __u32 saddr;         //源 IP 地址
    __u32 daddr;         //目的 IP 地址
    __u8 option[0];      //选项
} IPHdr_t;

```

需要注意的是：

在网络编程和数据通信中，字节序（Byte Order）是一个非常重要的概念。它决定了多字节数据（如整数、浮点数等）在内存中的存储顺序。字节序有主机字节序和网络字节序的区别。

主机字节序是指计算机系统内部存储多字节数据的顺序，不同的架构可能采用不同的字节序：小端字节序（Little-Endian）或是大端字节序（Big-Endian）。

而网络字节序则是一种标准化的字节序，用于在网络中传输数据。为确保不同架构的设备之间能够正确解析数据，网络字节序统一采用大端字节序。

我们定义网络协议数据结构的时候，应按照网络字节序定义，考虑到是在主机内解析首部，部分按位定义的字段，需要考虑区分不同的主机字节序来定义。

而在使用这些字段时，要注意 2 字节、4 字节的字段，在数据包的二进制数据块中是网络字节序存放的，在主机内访问，需要进行字节序转换，可以调用系统库提供的以下函数实现自动转换：

```

uint32_t ntohl(uint32_t netlong);    //网络字节序到主机字节序（32 位）
uint32_t htonl(uint32_t hostlong);   //主机字节序到网络字节序（32 位）。

uint16_t ntohs(uint16_t netshort);   //网络字节序到主机字节序（16 位）。
uint16_t htons(uint16_t hostshort);  //主机字节序到网络字节序（16 位）。

```

以下是一段解析 IP 首部的代码示例：

```

IPHdr_t *iph = (IPHdr_t *) (packet + sizeof(EthHdr_t));

```

```

printf("*** IP Header ***\n");
printf("Version : %d\n", iph->version);
printf("Headerlen: %d (%d bytes)\n", iph->ihl, iph->ihl * 4);
printf("Total len: %d\n", ntohs(iph->tot_len));
printf("Source IP: %08X\n", ntohl(iph->saddr));
printf("Dest IP: %08X\n", ntohl(iph->daddr));
printf("Protocol : %d", iph->protocol);

```

2.2.3 解析 TCP 首部

如果 IP 头的协议字段为 TCP（6），则其后紧接的就是 TCP 首部。TCP 首部包含 20 字节的固定部分和不定长的选项部分，我们按照 TCP 协议的规范，将其数据结构定义如下：

```

//TCP 报文段首部
typedef struct tcphdr {
    __u16 source;      //源端口
    __u16 dest;       //目的端口
    __u32 seq;        //序号
    __u32 ack_seq;    //确认号
#ifdef __BIG_ENDIAN__
    __u8 doff:4, res1:4; //数据偏移,保留
    __u8 cwr:1, ece:1, urg:1, ack:1, psh:1, rst:1, syn:1, fin:1; //标志位
#else
    __u8 res1:4, doff:4; //数据偏移,保留
    __u8 fin:1, syn:1, rst:1, psh:1, ack:1, urg:1, ece:1, cwr:1; //标志位
#endif
    __u16 window;     //窗口
    __u16 check;      //检验和
    __u16 urg_ptr;     //紧急指针
    __u8 option[0];   //选项
} TCPhdr_t;

```

由于 IP 首部的长度不固定，我们不能使用数据结构的大小来计算 TCP 首部在 IP 首部之后的位置，而应该使用 IP 首部的首部长度字段来计算。注意 IP 首部长度是以 4 字节为单位的，计算时应乘以 4。以下是一段解析 TCP 首部的代码示例：

```

TCPhdr_t *tcph = (TCPhdr_t *) (packet + sizeof(EthHdr_t) + iph->ihl * 4);

printf("***** TCP Header *****\n");
printf("Source Port: %d\n", ntohs(tcph->source));
printf("Dest Port: %d\n", ntohs(tcph->dest));
printf("Data Offset: %d (%d bytes)\n", tcph->doff, tcph->doff * 4);
printf("SequenceNum: %u\n", ntohl(tcph->seq));

```

```
DBG("Ack Number : %u\n", ntohl(tcp->ack_seq));
```

至此，我们已经解析了数据链路层、网络层、运输层的协议，可以得到 TCP 报文段所传输的应用层消息了。同样，由于 TCP 首部的长度也不固定，我们也不能使用数据结构的大小来计算应用层消息在 TCP 首部后的位置，而应该使用 TCP 首部的数据偏移字段来计算。同样，TCP 数据偏移字段也是以 4 字节为单位的，计算时应乘以 4。例如：

```
__u8 *msg = packet + sizeof(EthHdr_t) + iph->ihl * 4 + tcp->doff * 4;
```

2.3 FTP 文件传输

FTP（File Transfer Protocol，文件传输协议）是一种用于在网络上进行文件传输的应用层协议。FTP 基于客户端/服务器架构，允许用户通过客户端软件连接到 FTP 服务器，进行文件的上传和下载操作。FTP 使用运输层的 TCP 协议进行传输，本身不加密，用户名、密码和文件数据以明文形式传输，我们可以通过分析 FTP 流量，提取控制命令信息以及所传输的文件。

FTP 协议使用两个连接进行通信：

控制连接用于传输 FTP 命令和响应，默认端口为 21；

数据连接用于传输文件内容和目录列表等数据，端口号根据传输模式动态分配。

2.3.1 FTP 控制连接

FTP 控制连接用于传输客户端的命令以及服务器对命令的响应，以 ASCII 码的字符协议传输。协议定义了一系列标准命令，用于执行用户认证、文件操作和目录管理。常见的命令包括：

```
USER: 用户登录;  
PASS: 用户密码;  
LIST: 列出目录内容;  
RETR: 下载文件;  
STOR: 上传文件;  
MKD: 创建目录;  
RMD: 删除目录;  
DELE: 删除文件。
```

每个命令都有对应的响应码，这些响应码遵循 RFC 959 标准：

```
1xx: 信息性响应;  
2xx: 成功响应;  
3xx: 需要更多信息;  
4xx: 客户端错误;  
5xx: 服务器错误。
```

2.3.2 FTP 的数据连接

FTP 的数据连接用于传输文件内容和目录列表等二进制数据，有两种建立模式：主动模式（PORT）和被动模式（PASV）。

2.3.2.1 主动模式（PORT）

在主动模式下，FTP 客户端主动打开一个端口，等待 FTP 服务器的连接。具体步骤如下：

客户端通过控制连接向服务器发送 PORT 命令，携带逗号分隔的 6 个不大于 255 的数字，格式为：

```
PORT a1,a2,a3,a4,a5,a6
```

其中 a1、a2、a3、a4 为客户端 IP 地址的点分十进制各段数值， $(a5 \times 256 + a6)$ 则是客户端指定的用于接收数据连接的本地监听端口，例如：

```
PORT 192,168,1,100,10,1
```

表明客户端 IP 地址为 192.168.1.100，监听端口为 2561。

服务器收到 PORT 命令后，发送 200 响应码确认：

```
200 PORT command successful
```

服务器收到 PORT 命令后，会通过 TCP 连接到客户端指定的端口新建一个数据连接，通过数据连接发送文件内容或目录列表。

数据传输完成后，数据连接关闭。

控制连接则在整个会话期间保持打开，用于传输 FTP 命令和响应。

2.3.2.2 被动模式（PASV）

在被动模式下，FTP 服务器打开一个端口，等待客户端的连接。具体步骤如下：

客户端通过控制连接向服务器发送 PASV 命令，请求服务器进入被动模式：

```
PASV
```

服务器收到 PASV 命令后，会打开一个随机的高端口号，并通过控制连接以 227 响应码告知客户端进入被动模式等待客户端连接，响应消息中也携带逗号分隔的 6 个不大于 255 的数字，格式为：

```
227 Entering Passive Mode (a1,a2,a3,a4,a5,a6)
```

其中 a1、a2、a3、a4 为服务器 IP 地址的点分十进制各段数值， $(a5 \times 256 + a6)$ 则是服务器指定的用于接收数据连接的本地监听端口，例如：

```
227 Entering Passive Mode (192,168,1,202,10,2)
```

表明服务器 IP 地址为 192.168.1.202，监听端口为 2562。

客户端收到服务器的响应后，会通过 TCP 连接到服务器指定的端口，新建一个数据连接，通过该连接发送文件或目录列表。

数据传输完成后，数据连接关闭。

控制连接在整个会话期间保持打开，用于传输 FTP 命令和响应。

注意文件是上传还是下载，与数据连接是主动模式还是被动模式没有关系，因为 TCP 是全双工的协议，无论从哪个方向发起的连接，都可以在两个方向上根据需要传输数据。

我们要做的是，解析 FTP 控制连接中的命令与响应，一是通过 PORT 命令或者 227 响应的信息，找到数据连接的 IP 端口信息，在后续的报文中找到数据连接的报文；二是检测到 RETR 或者 STOR 命令，提取所传输文件的文件名。

至此，我们就可以进一步对数据连接进行 TCP 流重组，将报文段中的文件数据块存到文件中，还原出所传输的文件。

2.4 TCP 数据流重组

2.4.1 基本原理

TCP 数据流重组是指将分散的 TCP 数据包按照其在传输过程中的顺序重新组合成完整的数据流。TCP 协议通过序列号（Sequence Number）来确保数据包的顺序和完整性。

每个 TCP 数据包首部都包含一个序列号字段（Sequence Number），用于标识数据包在数据流中的位置。

在网络传输中，数据包可能会因路由问题、网络拥塞等原因乱序或丢失。TCP 协议通过重传机制确保数据的完整性，而接收端则根据序列号将数据包排序保证数据顺序正确。

TCP 流重组完成的条件是：一个 TCP 会话的所有数据包的序列号连续，且第一个数据包的 SYN 标志为 1，最后一个数据包的 FIN 标志为 1。

2.4.2 设计思路

一种实现思路，是为一个 TCP 会话建立一个数据包链表，根据当前处理数据包的序列号，将数据包复制缓存一份（因为 libpcap 的 callback() 函数一旦返回，就无法再得到曾经处理过的数据包），并插入到会话的链表中。如果链表中已存在相同序列号的数据包，则忽略数据包。插入一个数据包后，可以遍历链表，检查该会话的数据包是否完整，若达到重组条件，按照序列号顺序依次将各个数据包的数据写入文件，还原出传输的文件。

为避免缓存整个连接的数据过于占用内存，可以参照 TCP 协议的接收窗口机制，采用流式重组，收到一定数量的按序数据包，即可将它们写入文件并释放掉，再继续处理后续的数据包。

具体如何做，就请同学们自行设计算法并实现。

3 软件行为分析参考

3.1 逆向分析基础与工具使用

3.1.1 逆向分析基础

正向的软件开发编译的基本步骤包括编辑->汇编->链接。因此，对于我们逆向分析软件的功能和行为，需要对其进行反汇编和反编译。

反汇编器的作用是将二进制代码通过反汇编算法翻译成能够看懂的汇编代码。反编译器的作用则是在反汇编程序的基础上，进一步转换成高级程序语言，例如 C 语言。他们的作用是分析并使用便于人类理解阅读的方式呈现出经过编译的二进制文件的代码结构。

通过反汇编和反编译过程，我们可以纵观整个程序的功能、包含了什么字符串以及哪一段代码引用了这些字符串、哪些程序外部的操作系统函数被调用了。

反汇编器的作用是以底层汇编器的形式描述程序代码，所以用 C++、Dephi、Visual Basic 或者其他高级编程语言写的软件被编译成原生机器代码后，反汇编器会以 x86 或 x64 的形式向我们展示对应汇编代码。

反编译是尽最大可能还原初始高级语言代码。以 C++ 为例，其中的数据结构、类型声明和代码结构想要从编译后的代码中再重新获得，其复杂性很高。正因如此，目前反编译工具较少，本课程设计可以使用 IDA Pro 或者开源的 Ghirda 进行。

3.1.2 逆向分析工具的使用（以 IDA Pro 为例）

IDA Pro（Interactive DisAssembler）是一款广泛使用的反汇编和反编译逆向工程工具，主要用于分析二进制可执行文件。

3.1.2.1 程序加载

对于已知格式的应用程序，如下图所示，IDA Pro 可以自动化判断其指令集、架构和内存布局等信息，对于未知格式的需要用户指定。本实验给予同学们为已知格式的 i386 的 Linux 应用程序。

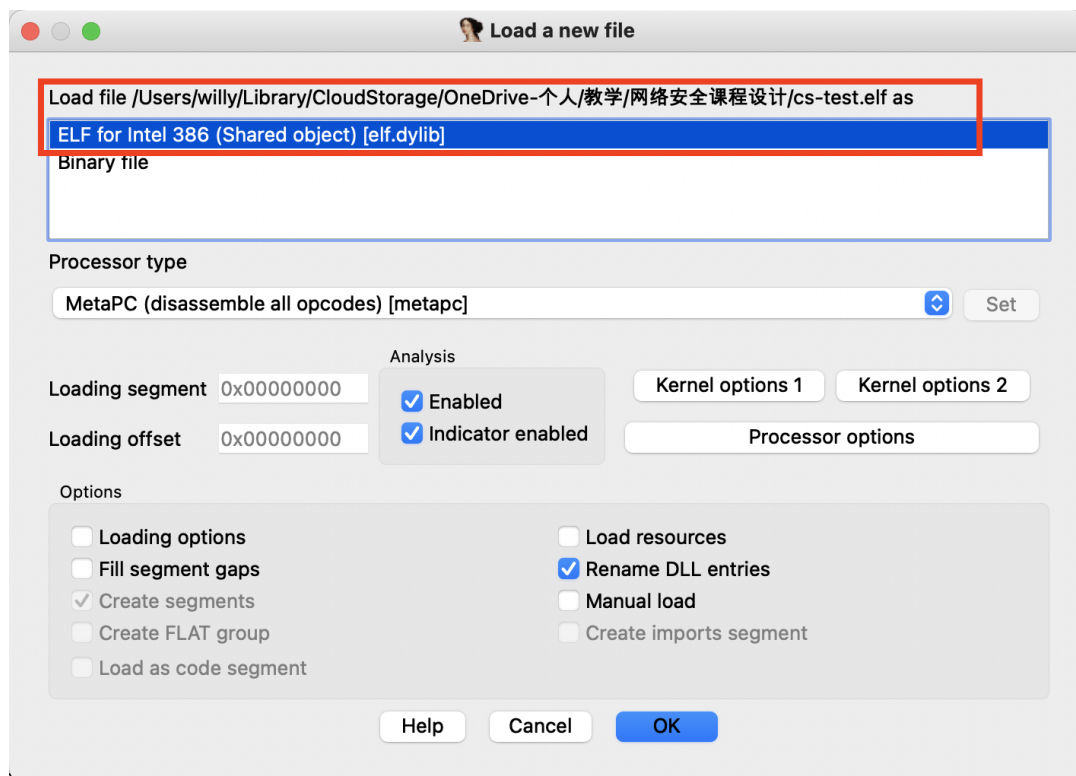


图 3.1 应用程序加载

加载后，IDA Pro 会自动化定位到 main 函数的反汇编代码，左侧栏可以选择反汇编函数，空格可以显示控制流图，按 F5 可以显示反编译 C 伪代码，如下图所示。

```

text:000018FD ; int __cdecl main(int argc, const char **argv, const char **envp)
text:000018FD public main
text:000018FD main proc near ; DATA XREF: .got:main_ptr!o
text:000018FD
text:000018FD argc = dword ptr 4
text:000018FD argv = dword ptr 8
text:000018FD envp = dword ptr 0Ch
text:000018FD
text:000018FD ; __unwind {
text:000018FD
text:00001901 endbr32
text:00001901 lea ecx, [esp+argc]
text:00001905 and esp, 0FFFFFFF0h
text:00001908 push dword ptr [ecx-4]
text:0000190B push ebp
text:0000190C mov ebp, esp
text:0000190E push ebx
text:0000190F push ecx
text:00001910 sub esp, 1A0h
text:00001916 call __x86_get_pc_thunk_bx
text:0000191B add ebx, 265Dh
text:00001921 mov eax, large gs:14h
text:00001927 mov [ebp-0Ch], eax
text:0000192A xor eax, eax
text:0000192C mov dword ptr [ebp-1A0h], 10h
text:00001936 sub esp, 4
text:00001939 push 0
text:0000193B push 1
text:0000193D push 2
text:0000193F call sub_1330
text:00001944 add esp, 10h
text:00001947 mov [ebp-19Ch], eax
text:0000194D cmp dword ptr [ebp-19Ch], 0FFFFFFFh
text:00001954 jnz short loc_1972
text:00001956 sub esp, 0Ch
text:00001959 lea eax, (aSocketFailed - 3F78h)[ebx] ; "socket failed"
text:0000195F push eax
text:00001960 call sub_1240
text:00001965 add esp, 10h
text:00001968 sub esp, 0Ch
text:0000196B push 1
text:0000196D call sub_12A0
text:00001972

```

图 3.2 Main 函数反汇编代码

3.1.2.2 控制和数据依赖人工逆向分析方法

本实验中，需要分析程序的数据和控制依赖从而判断漏洞和恶意行为的触发条件，例如，在 demo.c 的源码中存在如下代码片段，则表示当 array 数据的第二和第三个字节分别为 A 和 B 的时候 malware_function 函数会被调用。

```
...  
if (array[1] == 'A' && array[2] == 'B') {  
    malware_function();  
}  
...
```

对应编译后的二进制代码反汇编后，在 IDA Pro 中我们同样可以发现此逻辑如下图所示，从而得出其触发条件。

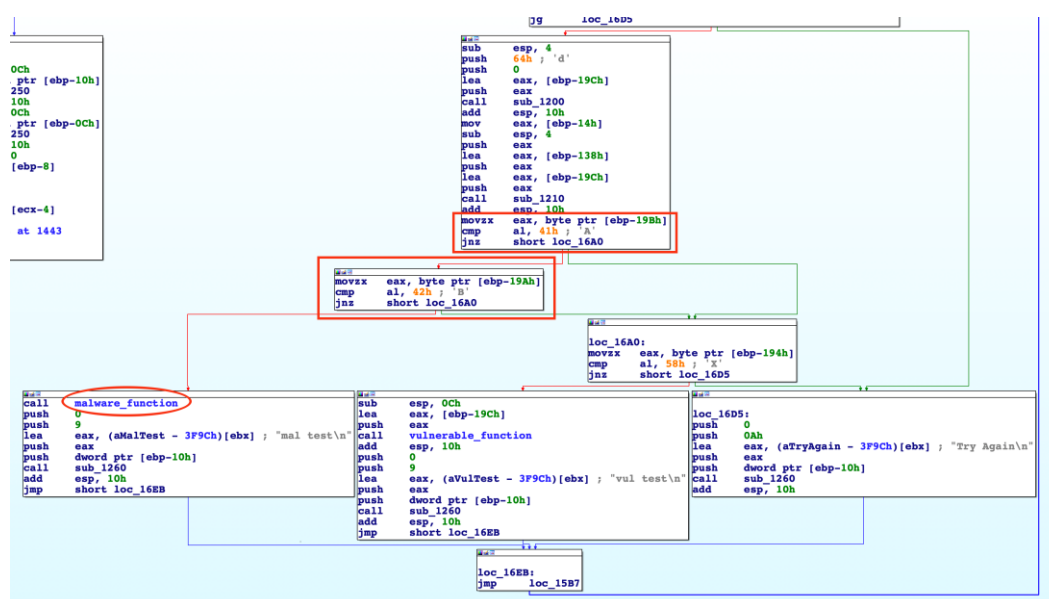


图 3.3

3.1.2.3 插件开发自动化分析

IDA Pro 等反汇编和反编译工具同时支持插件开发，可以利用插件开发实现自动化的恶意代码和漏洞辅助检测，如下所示，可以编写如下 Python 插件代码实现敏感函数查找，输出目标函数是否存在以及名称和地址。

插件编写方法也可以询问大模型和参考网上教程，例如 <https://bbs.csdn.net/topics/618733410>

```
import idutils  
import idc  
  
def detect_func():  
    # 搜索所有函数调用  
    for func_ea in idutils.Functions():
```

```

func_name = idc.get_func_name(func_ea)

# 检查是否使用了拷贝类型的敏感函数
for head in idutils.Heads(func_ea, idc.get_func_attr(func_ea, idc.FUNC
ATTR_END)):
    mnemonic = idc.GetMnem(head)
    if mnemonic in ['call']:
        called_func = idc.GetOpnd(head, 0)
        if called_func in ['strcpy', 'sprintf', 'gets', 'memcpy']:
            print(f"目标函数调用: {func_name} 中的 {called_func} 在地址 {hex(head)}")

# 执行检测
detect_func ()

```

3.2 常见 Linux 程序恶意代码与漏洞表征

本实验要求通过 3.1 中介绍的反汇编和反编译工具编写插件实现对样本代码中存在的恶意代码行为和漏洞进行自动化检测，本节主要介绍 Linux 常见恶意代码和漏洞的源码表现形式以及标准的任务一和任务二答案模板，覆盖现场检查中的样本中的漏洞和行为 90% 以上。

注意：任务要求中的漏洞类型或者是恶意代码功能类型和本章四级目录标题保持一致。

3.2.1 恶意代码

在 Linux 应用程序中，恶意行为通常是指未经授权或恶意地访问、篡改、删除或损坏数据，或者干扰系统的正常功能。

3.2.1.1 系统文件删除与修改

恶意软件可能会删除或者修改系统关键文件包括系统日志、密码文件等，例如：

```

#include <stdio.h>
#include <stdlib.h>

int main() {
    // 删除系统文件 /etc/passwd
    if (remove("/etc/passwd") == 0) {
        printf("File deleted successfully\n");
    } else {
        printf("Error deleting file\n");
    }
}

```

```
}  
return 0;  
}
```

答案说明:

(1) 函数名称和函数调用地址: 删除是 `remove` 函数的调用地址, 修改是 `fwrite/fopen` 等文件操作函数, 重命名是 `rename` 函数, 修改文件权限是 `chmod` 函数, 函数不唯一的时候需要每个名称和地址都写出来, 注意只写恶意的, 如果是正常文件操作 (非系统文件) 则不要列出, 如果列出则为误报反而要扣分;

(2) 功能描述要说清楚对应系统函数的功能是什么, 特别是功能中的参数是什么, 即修改了什么系统文件、怎么修改了、修改了什么内容或者权限。

3.2.1.2 开启后门

恶意软件开启额外后门进行隐藏数据传输等行为, 例如,

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main() {  
    // 启动一个恶意的监听端口 (例如 12345)  
    system("nc -l -p 12345 -e /bin/bash &");  
    return 0;  
}
```

答案说明:

(1) 函数名称和函数调用地址: `system` 函数以及其调用地址 (注意只列恶意功能的 `system` 函数调用地址, 不要列正常功能的);

(2) 功能描述要说清楚 `system` 函数的参数是什么, 并对参数进行解释, 比如开启了多少号端口。

3.2.1.3 创建僵尸子进程

恶意程序可能会启动大量的进程以消耗系统资源, 造成 Denial of Service (DoS) 攻击。

```
#include <stdio.h>  
#include <stdlib.h>  
  
int main() {  
    while(1) {  
        // 启动一个新的子进程  
        if (fork() == 0) {
```

```

        // 子进程立即退出，成为僵尸进程
        exit(0);
    }
}
return 0;
}

```

答案说明：

(1) 函数名称和函数调用地址：`fork` 函数以及其调用地址，注意只列恶意功能的 `fork` 函数调用地址（创建数目巨大如大于 10 个以上，且 `fork` 的进程没有实体功能直接退出），不要列正常功能的；

(2) 功能描述要说清楚为什么这个 `fork` 函数是僵尸进程不是正常进程。

3.2.1.4 禁用系统保护功能

恶意软件可以通过系统函数或者修改系统文件来禁用系统保护功能包括 PIE、SELinux、ASLR 等。

```

void disable_selinux() {
    // 禁用 SELinux printf("[+] Disabling SELinux...\n");
    system("setenforce 0");
}

```

答案说明：

(1) 函数名称和函数调用地址：`system` 或其他函数（自行查询）以及其调用地址；

(2) 功能描述要说清楚禁用了什么系统保护、如何实现了禁用。

3.2.2 软件漏洞

3.2.2.1 栈溢出

攻击者可能通过缓冲区溢出漏洞来执行恶意代码，获得系统控制权。如下 `strcpy` 没有做边界检查，会导致缓冲区溢出，可以用来覆盖返回地址，控制程序流程。

```

#include <stdio.h>
#include <string.h>

void vulnerable_function(char *input) {
    char buffer[10];
    strcpy(buffer, input); // 缓冲区溢出漏洞
}

```

答案说明：

(1) 漏洞成因地址：`strcpy` 或常见其他拷贝函数（自行查询）以及其调用地址；

(2) 漏洞成因描述要说清楚目的和源操作都是什么、分别大小多少、为什么溢出了。

3.2.2.2 整数溢出

整数溢出发生在一个整数运算结果超出了该整数类型的最大或最小值，导致数据被错误地表示。通常，整数溢出会导致逻辑错误或安全漏洞。代码中 `a` 被初始化为 `unsigned int` 类型的最大值 `UINT_MAX`（通常是 4294967295）。然后，通过 `a = a + 1` 操作让 `a` 超过了 `unsigned int` 的最大值，导致溢出，`a` 的值变成了 0。

```
#include <stdio.h>
#include <limits.h> // 定义了整数类型的最大最小值

void integer_overflow_example() {
    unsigned int a = UINT_MAX; // 将 a 初始化为 unsigned int 的最大值
    printf("a = %u\n", a);
    // 模拟整数溢出: a 加 1 会导致溢出
    a = a + 1; // 整数溢出, a 变为 0
    printf("After overflow, a = %u\n", a);
}
```

答案说明（整数溢出漏洞不要求实现脚本自动化分析）：

- (1) 漏洞地址：溢出的计算指令的地址比如如上示例中 `a+1` 指令的地址；
- (2) 漏洞成因描述要说清楚哪个整数存在溢出，为什么可能存在整数溢出。

3.2.2.3 堆溢出

堆溢出通常发生在程序分配内存时，写入超出分配区域的数据，从而覆盖堆上其他数据。这可能导致程序崩溃，甚至允许攻击者执行任意代码。如下程序使用 `malloc(20)` 动态分配了 20 字节的内存。然后通过 `strcpy` 函数将一个长度超过 20 字节的字符串写入 `buffer`。`strcpy` 不会检查目标缓冲区的大小，因此它会将超出分配内存大小的数据写入堆上可能相邻的内存区域，导致堆溢出。堆溢出可能会覆盖堆中的其他数据结构，进而导致程序崩溃或被攻击者利用执行任意代码。

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

void heap_overflow_example() {
    char *buffer = (char *)malloc(20); // 分配 20 字节的内存
    if (buffer == NULL) {
        printf("Memory allocation failed!\n");
    }
}
```

```

        return;
    }

    // 模拟堆溢出: 我们写入超过 20 字节的数据, 导致溢出
    strcpy(buffer, "This is a very long string that exceeds the buffer size
");

    printf("Buffer content: %s\n", buffer);

    free(buffer); // 释放内存
}

int main() {
    heap_overflow_example();
    return 0;
}

```

答案说明:

- (1) 漏洞地址: 调用拷贝函数的地址;
- (2) 漏洞成因描述要说清楚对堆指针以及拷贝函数的源数据大小、堆大小、为什么溢出。

3.2.2.4 使用后重放 (Use After Free)

访问已释放内存可能导致程序崩溃、数据损坏或在某些情况下被攻击者利用来执行恶意代码。例如,如下程序首先通过 `malloc` 动态分配了一块内存,并将数据写入该内存。然后,通过 `free(ptr)` 释放了这块内存。尽管内存已经被释放,但程序仍然试图访问这块内存,即 `printf("Data: %s\n", ptr)`, 产生 Use After Free (UAF) 漏洞。

```

#include <stdio.h>
#include <stdlib.h>

void example_uaf() {
    char *ptr = malloc(100); // 动态分配内存
    if (ptr == NULL) {
        return;
    }

    // 填充数据
    snprintf(ptr, 100, "Hello, world!");

    // 释放内存
    free(ptr);
}

```



```
// UAF 漏洞：访问已经释放的内存
printf("Data: %s\n", ptr); // 这里访问已经被释放的内存
}
```

答案说明：

- (1) 漏洞地址：malloc 的地址、free 的地址和 use 的地址都要列出来；
- (2) 漏洞描述要说清楚 use after free 的具体条件，比如示例中解释堆指针 ptr 指针在 free 之后又在 printf 中使用。

3.2.2.5 重放释放 (Double Free)

DF (Double Free) 漏洞是指程序错误地对同一块内存进行多次释放。这会导致内存管理错误，进而可能被攻击者利用来执行恶意代码。

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

void example_df() {
    char *ptr = malloc(100); // 动态分配内存
    if (ptr == NULL) {
        return;
    }

    // 填充数据
    snprintf(ptr, 100, "Hello, world!");

    // 第一次释放内存
    free(ptr);

    // 第二次释放同一块内存 (Double Free)
    free(ptr); // 错误：再次释放已经释放的内存
}
```

答案说明：

- (1) 漏洞地址：两次 free 的地址；
- (2) 漏洞成因描述要说清楚什么条件下会出现第一次 free 和第二次 free。

3.3 二进制程序模拟执行与调试工具

本实验建议安装和使用 QEMU 虚拟化模拟执行工具对样本软件进行测试和调试。

3.3.1 环境依赖安装

由于样本程序为 32 位，为确保其他系统环境下可执行 32 位程序要首先安装 32 位的 glibc 库，参考指令如下：

```
sudo apt update
sudo apt install libc6:i386
```

3.3.2 QEMU 安装

在大多数 Linux 发行版中，可以直接通过包管理器安装 QEMU。例如，在 Ubuntu 系统中，可以使用以下命令安装用户态模式 QEMU，系统模式 QEMU 安装方法可以自行查阅：

```
sudo apt install -y qemu qemu-user qemu-user-static
```

3.3.3 QEMU 模拟执行待测样本程序

获得样本程序后，动态运行样本前先利用给样本程序赋予可执行的权限。然后在命令行中使用如下指令用 qemu 用户态模拟执行该程序，系统模式模拟执行该程序的方法和指令请自行查阅。其中 in_asm 和-strace 参数可以生成执行的系统函数和指令等信息，输出文件到 trace.log 文件中，方便同学们进行查阅和验证

```
qemu-i386 -strace -d in_asm -D trace.log demo
```

如下图所示，为日志的部分输出可以看到执行后的汇编指令和系统函数。

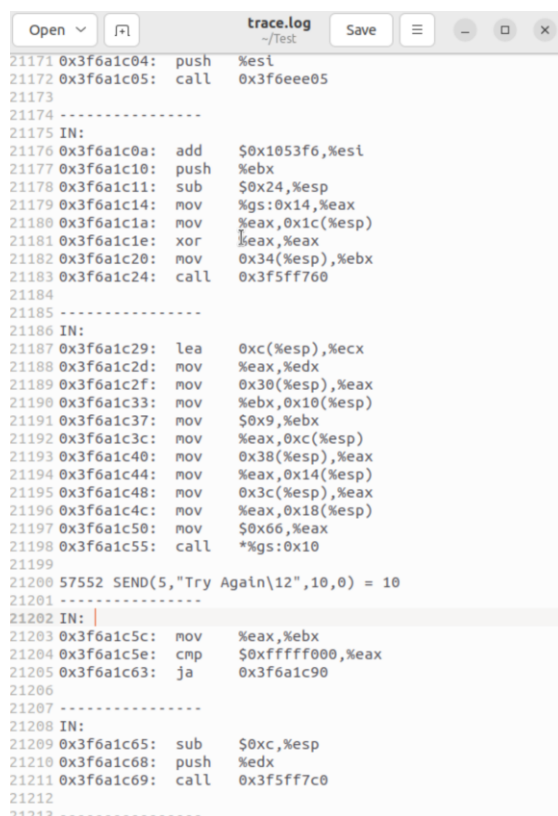
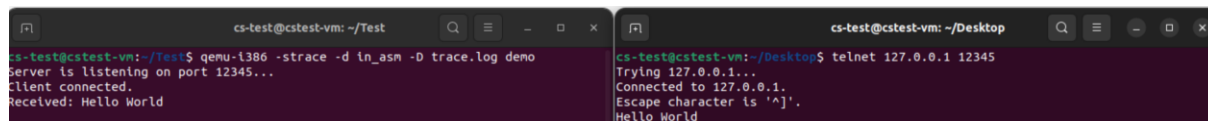


图 3.4

3.3.4 模拟执行的样本程序交互

模拟执行待测程序后，另外打开一个终端 `telnet 127.0.0.1 12345` 连接上即可进行交互测试，例如下图运行模拟执行 `demo` 程序后和远程终端的交互截图，本实验要求输入不同的内容，观察样本软件的执行情况，从而触发全面恶意行为和漏洞。



```
cs-test@ctest-vm: ~/Test
cs-test@ctest-vm:~/Test$ qemu-i386 -strace -d in_asm -D trace.log demo
Server is listening on port 12345...
Client connected.
Received: Hello World

cs-test@ctest-vm: ~/Desktop
cs-test@ctest-vm:~/Desktop$ telnet 127.0.0.1 12345
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^'.
Hello World
```

4 拓展思考

4.1 在线重组

本课程设计当前采用的离线方式，是针对提供的截包 pcap 文件进行静态的 TCP 流重组，这种方式在实际场景中存在检测滞后响应不及时的问题，如果调整为在线方式实时重组则会有很大改进。那么在线方式应该如何实现呢？同学们可以查阅资料，思考并尝试。

4.2 乱序重组

本课程设计当前提供的截包 pcap 文件，是在无干扰的理想网络环境中截包得到的，TCP 数据流是按序完整的，重组比较简单。现实网络场景中，TCP 报文段可能出现丢失、乱序等异常情况。要在这些异常情况下正确的重组得到传输的文件，应该如何去改进我们的程序呢？同学们可以查阅资料，思考并尝试。

4.3 自动化动态漏洞检测

本课程设计主要采用静态逆向分析工具对样本软件进行漏洞检测与分析，然而静态分析方法受限于检测规则设置以及静态逆向分析工具本身的能力等问题，难以避免存在较多漏报和误报。因此，在真实漏洞检测中大多采用动态检测方法例如模糊测试技术，请同学们自行查阅资料，思考并尝试利用动态检测方法进行漏洞检测并生成漏洞利用输入。

4.4 基于虚拟机的启发式恶意代码检测

本课程设计主要采用基于固定的恶意代码特征静态恶意行为自动化检测方法，然后在虚拟环境中进行人工验证，然而，真实场景中，恶意行为变化多样种类繁多且可能会采用加密和压缩技术，请同学们自行查阅资料，思考并尝试设计更加通用的启发式检测规则，并在虚拟机环境中如 QEMU 进行自动化恶意行为检测。

4.5 系统保护绕过

本课程设计中软件程序编译的时候关闭了栈保护，因此栈溢出漏洞可以直接利用，但实际应用程序编译时候都会打开许多默认的程序保护功能，请同学们自行查阅资料，思考如何利用待测程序中的恶意功能动态关闭程序保护再进行漏洞利用的方法。