**系统安全技术 Project**

**——DDoS的模拟与测试**

**实验报告**

计算机科学与技术

14307130318 刘超颖

**1. 开发工具与环境**

使用Pycharm集成开发环境，以Python(2.7.10)作为主要开发语言，使用Wireshark做网络封包分析。运行环境为1台macOS Sierra 10.12.5(3 GHz Intel Core i7, 16GB)和3台Ubuntu Linux(64GB)服务器。

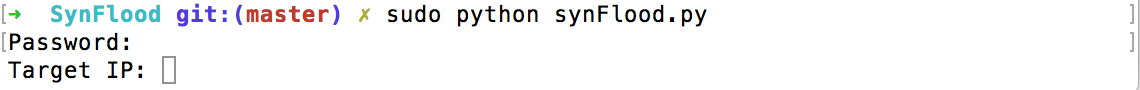
**2. 文件介绍**

|  |
| --- |
| .  |\_\_\_\_DDoS\_Effect.jpeg  |\_\_\_\_README.md  |\_\_\_\_Fraggle  | |\_\_\_\_fraggle.py  | |\_\_\_\_fraggle\_result.pcap  | |\_\_\_\_fraggle.jpeg  |\_\_\_\_HttpFlood  | |\_\_\_\_.DS\_Store  | |\_\_\_\_httpFlood.jpeg  | |\_\_\_\_httpFlood\_result.pcap  | |\_\_\_\_httpFlood.py  |\_\_\_\_ICMPFlood  | |\_\_\_\_icmpFlood.jpeg  | |\_\_\_\_icmpFlood\_result.pcap  | |\_\_\_\_icmpFlood.py  |\_\_\_\_PingofDeath  | |\_\_\_\_pingofDeath.py  | |\_\_\_\_pingofDeath\_result.pcap  | |\_\_\_\_pod.jpeg  |\_\_\_\_SlowLoris  | |\_\_\_\_slowLoris.jpeg  | |\_\_\_\_slowLoris.py  |\_\_\_\_Smurf  | |\_\_\_\_smurf.jpeg  | |\_\_\_\_smurf\_result.pcap  | |\_\_\_\_smurf.py  |\_\_\_\_SynFlood  | |\_\_\_\_portscan.jpeg  | |\_\_\_\_synflood.jpeg  | |\_\_\_\_synFlood.py  | |\_\_\_\_SynFlood\_result.pcap  |\_\_\_\_Trinoo  | |\_\_\_\_trinoo.py  | |\_\_\_\_trinoo\_result.pcap  | |\_\_\_\_trinoo.jpeg  |\_\_\_\_UdpFlood  | |\_\_\_\_udpflood.jpeg  | |\_\_\_\_udpFlood\_result.pcap  | |\_\_\_\_udpFlood.py  |\_\_\_\_UdpFlood\_2  | |\_\_\_\_udpFlood.py  |\_\_\_\_配库 |

我们实现了Fraggle, Http Flood, ICMP Flood, Ping of Death, Slowloris, Smurf, TCP SYN Flood和2种Udp Flood，.jpeg文件是通过Wireshark抓包实验得到的测试结果截图，.pcap文件是通过Wireshark抓包实验得到的结果。其中，少量DDoS没有.pcap或.jpeg文件，会在后文中说明原因。“配库”文件中存储的是一部分库文件配置的过程（如scapy, dumbnet等）。DDoS\_Effect.jpeg是一个通过DDoS占用本机CPU的效果样例。

**3. 使用说明**

根据命令行界面中提示的信息输入攻击对象IP地址、广播对象IP地址、端口号等信息。以下是TCP SYN Flood的一个例子：



由于涉及到数据包的发送，程序运行需要在sudo环境下执行。

**4. 设计过程**

**4.1 TCP SYN Flood**

在TCP三次握手中，假设一个用户向服务器发送了SYN报文后突然死机或掉线，那么服务器在发出SYN+ACK应答报文后是无法收到客户端的ACK报文的（第三次握手无法完成），这种情况下服务器端一般会重试（再次发送SYN+ACK给客户端）并等待一段时间后丢弃这个未完成的连接，这段时间的长度我们称为SYN Timeout，一般来说这个时间是分钟的数量级（大约为30秒-2分钟）；一个用户出现异常导致服务器的一个线程等待1分钟并不是什么很大的问题，但如果有一个恶意的攻击者大量模拟这种情况，服务器端将为了维护一个非常大的半连接列表而消耗非常多的资源——数以万计的半连接，即使是简单的保存并遍历也会消耗非常多的CPU时间和内存，何况还要不断对这个列表中的IP进行SYN+ACK的重试。实际上如果服务器的TCP/IP栈不够强大，最后的结果往往是堆栈溢出崩溃---即使服务器端的系统足够强大，服务器端也将忙于处理攻击者伪造的TCP连接请求而无暇理睬客户的正常请求（毕竟客户端的正常请求比率非常之小），此时从正常客户的角度看来，服务器失去响应，这种情况我们称作：服务器端受到了SYN Flood攻击。

在synFlood.py中，我们模拟了SYN Flood的过程。其中，random\_IP()函数实现了随机生成一个IP地址的功能，因为SYN Flood攻击中涉及IP欺骗，以这个随机生成的函数作为攻击数据包的发送地址是必要的。Checksum(msg)函数是一个计算校验和的函数，因为在生成某些数据包的包头过程中需要计算包内数据的校验和。接着，通过nmap扫描1-1024所有端口，找到主机上所有打开的端口，在之后的DDoS中集中向这些端口发起攻击。然后，生成IP数据包的包头[1]和TCP数据包的包头[2]，将这一数据包生成好之后，通过while 1的死循环不断向受攻击者发送虚假数据包，从而实现DDoS。

**4.2 UDP Flood**

UDPFlood是日渐猖厥的流量型DoS攻击，常见的情况是利用大量UDP小包冲击DNS服务器或Radius认证服务器、流媒体视频服务器。100k bps的UDPFlood经常将线路上的骨干设备例如防火墙打瘫，造成整个网段的瘫痪。由于UDP协议是一种无连接的服务，在UDPFLOOD攻击中，攻击者可发送大量伪造源IP地址的小UDP包。但是，由于UDP协议是无连接性的，所以只要开了一个UDP的端口提供相关服务的话，那么就可针对相关的服务进行攻击。

在实验中，我们实现了两种基于UDP Flood的DDoS，其中UdpFlood/udpFlood.py是通过直接使用random函数生成一个数据包实现的；而UdpFlood\_2/udpFlood.py继承了TCP SYN Flood的思路，通过代码生成一个虚假的IP地址，引入了IP欺骗。

**4.3 Trinoo**

Trinoo 攻击的攻击方法是向被攻击目标主机的随机端口发出全零的4字节UDP包，在处理这些超出其处理能力的垃圾数据包的过程中，被攻击主机的网络性能不断下降，直到不能提供正常服务，乃至崩溃。

我们发现，这一攻击模式和UDP Flood是极为相似的，基本上可以复用UDP Flood攻击中的代码，需要修改的地方在于将数据包里的数据赋为“00000000000000000000000000000000”。除此之外，由于它对IP 地址不做假，而是通过寻找肉机、端口代理实现隐藏，所以在实验中可以将random\_IP()删去。

**4.4 ICMP Flood**

ICMP flood通过高速发送大量的ICMP Echo Reply数据包，导致目标网络的带宽被耗尽，阻止合法的数据通过网络。ICMP Echo Reply数据包具有较高的优先级，在一般情况下，网络总是允许内部主机使用PING命令，这将导致系统不断地保留它的资源，直到无法再处理有效的网络流量。

在这个实验中，我们发现了scapy这一库[3]，通过使用这一库，我们将省去自己手动生成数据包包头的操作，而通过scapy内置的函数来伪造一个数据包。在ICMP Flood中，IP欺骗是必须的，我们依然运用和前三种DDoS中类似的random\_IP()函数进行虚假IP的生成。当我们引入scapy库进行数据包的伪造后，我们编写代码的工作量大大减少了。

**4.5 Ping of Death**

Ping of Death的方法是由攻击者故意发送大于65535字节的ip数据包给对方。TCP/IP的特征之一是碎裂：它允许单一IP包被分为几个更小的数据包。当许多操作系统收到一个特大号的IP包时候，它们不知道该做什么，因此，服务器会被冻结、宕机或重新启动。Ping of Death攻击是通过在最后分段中，改变其正确的偏移量和段长度的组合，使系统在接收到全部分段并重组报文时总的长度超过了65535字节，导致内存溢出，这时主机就会出现内存分配错误而导致TCP/IP堆栈崩溃，导致死机。

我们看出，Ping of Death攻击主要是利用发送长度超过65536字节的数据包来对主机进行攻击。由于ICMP Flood, Ping pf Death和之后会提到的Smurf都是基于ICMP包的攻击手段，所以我们基本上可以完全沿用之前的代码，只在构造包、构造攻击模式的两行代码处稍微改写一下，实现一个简单版的Ping of Death。

但事实上，目前的大部分系统（可能还有少量windows XP例外）应该已经不再存在“报文总长度超过65535字节导致内存溢出从而系统宕机”的问题了，Ping of Death这种攻击手段也已经不太现实了。在实验中，我们发送大量超过65535字节的数据包的操作，基本上可以看作是与之前ICMP Flood等价的一种行为。所以对Ping of Flood的实现其实是没有意义的，我们可以忽略这个DDoS的实现。

**4.6 Smurf**

Smurf攻击的主要方式为：攻击者会向接收站点中的一个广播地址发送一个IP ICMP ping(即“请回复我的消息”)。Ping数据包随后将被广播到接收站点的本地网络中的所有主机。该数据包包含一个虚假的源地址，这一IP地址即为该DDoS攻击对象的IP地址。每个收到此ping数据包的主机都会向这一虚假的源地址发送响应，从而导致被攻击主机收到大量的ping回复。如果收到的数据量过大，被攻击者就无法接收或区分真实流量。

这一攻击的主要思想是，攻击者将自身的IP伪造成被攻击者，然后引诱大量其他站点给被攻击者发送数据。在这一攻击方式中，主机不再需要随机伪造IP，而是直接将被攻击者的IP地址填入包头中。在使用scapy库之后，可以很轻易地在10行代码里实现一个简易的Smurf。

**4.7 Fraggle**

Fraggle攻击与Smurf攻击类似，只是利用UDP协议。攻击者掌握大量的广播地址，并向这些地址发送假冒的UDP包，通常这些包是直接到目标主机的7号端口——也就是Echo端口，而另一些情况下它却到了Chargen端口，攻击者可以制造一个在这两个端口之间的循环来产生网络阻塞。UDP端口7（ECHO）和端口19（Chargen）在收到UDP报文后，都会产生回应。在UDP的7号端口收到报文后，会回应收到的内容；在UDP的19号端口收到报文后，会产生一串字符流。它们都同ICMP一样，会产生大量无用的应答报文，占满网路带宽。

因此，我们向子网广播地址发送源IP地址为被攻击者的UDP包，端口号用7和19。按照理论，子网络启用了此功能的每个系统都会向受害者的主机做出响应，从而引发大量的包，导致受害网络的阻塞或受害主机的崩溃。如果将源端口改为19，目的端口为7，实现端口之间的循环，其危害性更大。

但事实上，在实验中，我们发现一般机器都不会打开这两个端口，即便在打开之后，Mac OS系统子网上也并没有启动这些功能。所以其唯一的效果就是产生一个ICMP不可达的消息，消耗一定的带宽。

**4.8 HTTP Flood**

在HTTP Flood DDoS攻击中，攻击者利用看似合法的HTTP GET或POST请求攻击Web服务器或应用程序。HTTP Flood不使用畸形数据包，欺骗或反射等技术，并且对带宽的需求比其他攻击更少。

由于HTTP Flood中不涉及欺骗、反射等技术，所以我们可以略去IP造假的过程，直接不断将“GET /%s HTTP/1.1\r\n\r\n”格式的数据包包头发送给被攻击服务器[4]。

除了数据包格式与TCP SYN Flood和UDP Flood略有不同外，其基本的flood思想与之前较为一致，此处不再赘述。

除此之外，我们试图在此处实现一个多线程攻击，使程序能并发地对服务器进行攻击，但目前为止其实现还有一些问题。

**4.9 Slowloris**

Slowloris也是基于HTTP协议的一种攻击手段。HTTP协议规定，HTTP Request以“\r\n\r\n”结尾表示客户端发送结束，服务端开始处理。那么，如果永远不发送“\r\n\r\n”会如何？Slowloris就是利用这一点来实现DDoS攻击的。攻击者在HTTP请求头中要求Web Server保持TCP连接不要断开，随后缓慢地每隔几分钟发送一个key-value格式的数据到服务端，如“a:b\r\n”，导致服务端认为HTTP头部没有接收完成而一直等待，从而使服务器的Web容器很快被攻击者占满而不再接受新的请求。

在实验中，我们通过

|  |
| --- |
| time.sleep(15) |

来实现Slowloris的“缓慢”特性。

除此之外，我们找到了一个日常使用的数据包包头列表“regular\_headers”[5]，并从中随机选取一个以“{}\r\n”的格式发出。

实验中的其他思想与之前的实验基本一致。

**5. 实验结果**

**5.1 TCP SYN Flood**

实验中使用服务器攻击主机。

在TCP SYN Flood中，我们抓到的包分为两个阶段。第一阶段为调用nmap.PortScanner()函数时抓到的包，反应了扫描端口阶段的操作；第二阶段为SYN Flood阶段，被攻击主机不断收到SYN数据包。

图1: TCP SYN Flood中扫描端口时抓的包

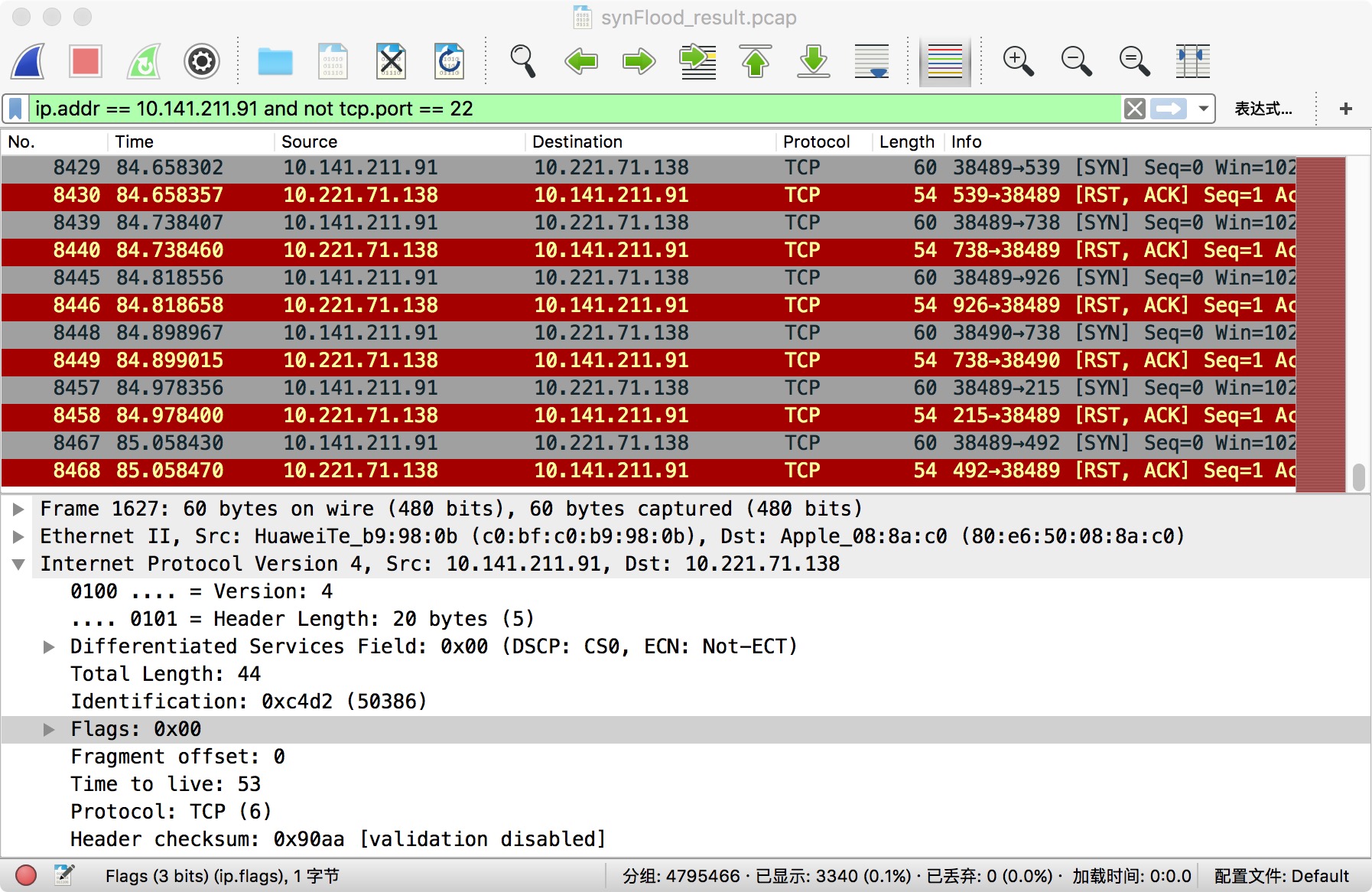
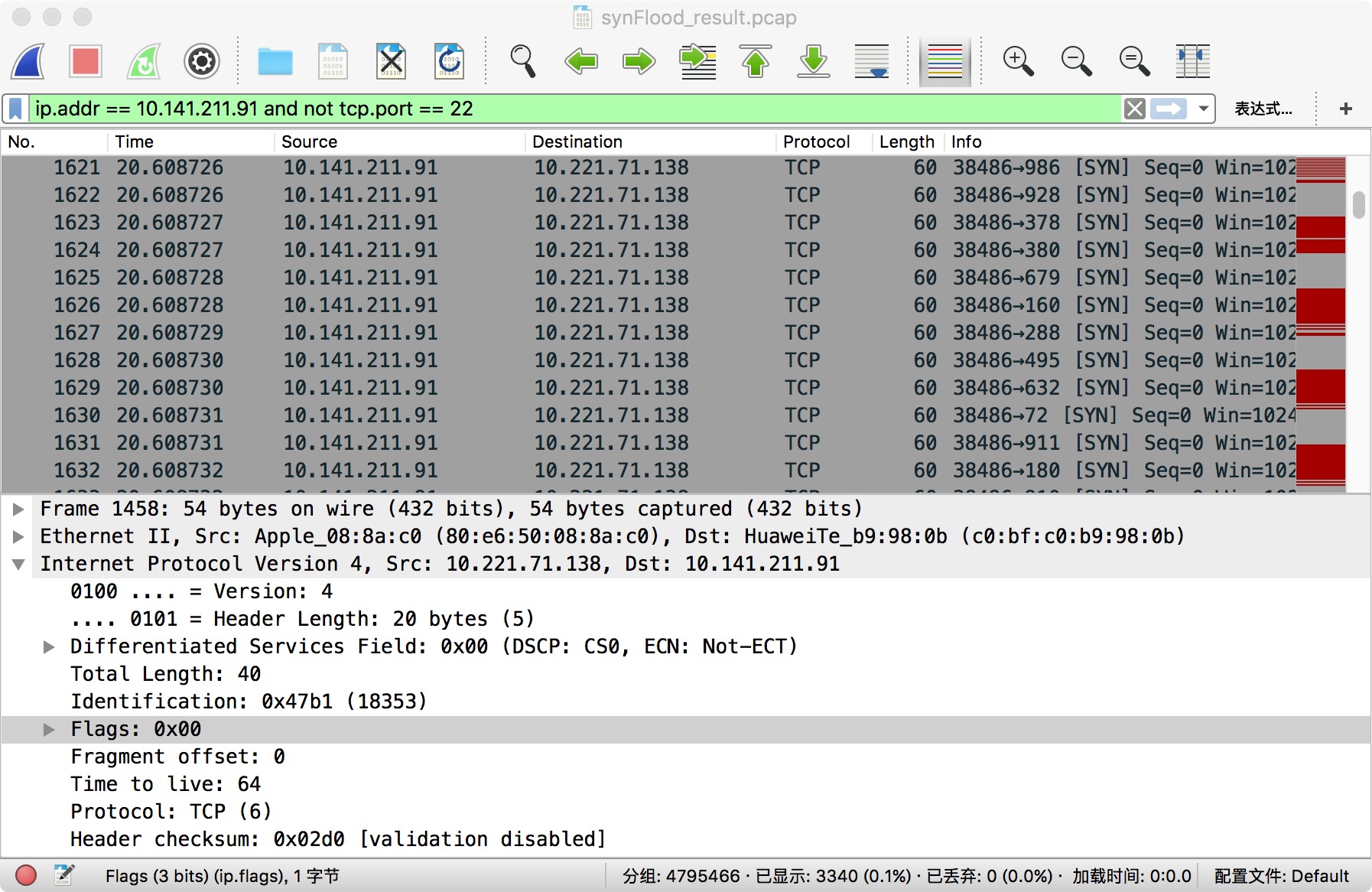


图2: SYN Flood阶段被攻击主机不断收到SYN数据包

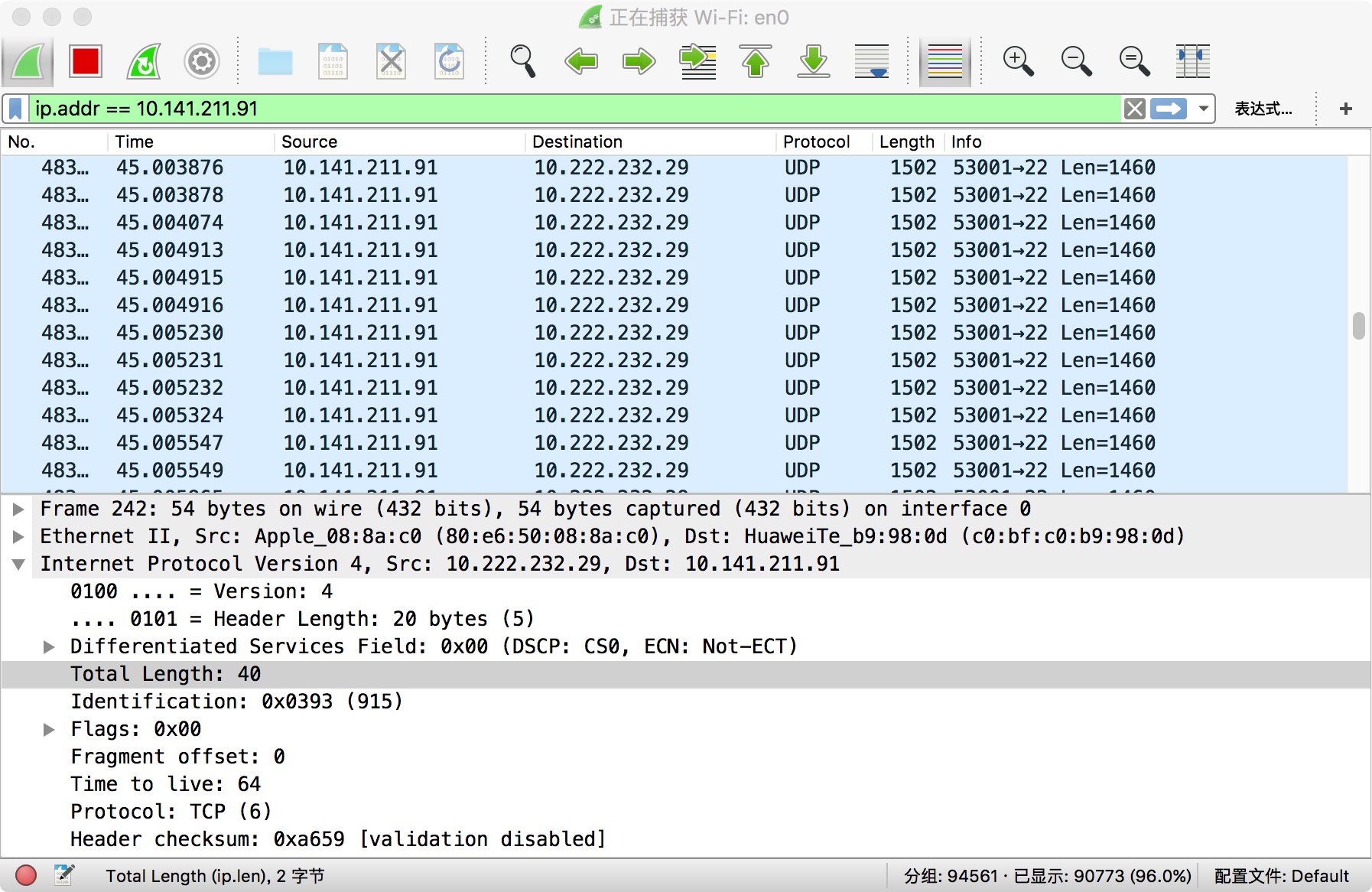


在后续的部分中，也可能涉及到这样的多阶段结果，我们之后就只截取主要的攻击结果进行展示。

**5.2 UDP Flood**

实验中使用服务器攻击主机。

图3: UDP Flood阶段被攻击主机不断收到UDP数据包



**5.3 Trinoo**

实验中使用服务器攻击主机。

图4: Trinoo阶段被攻击主机不断收到UDP数据包

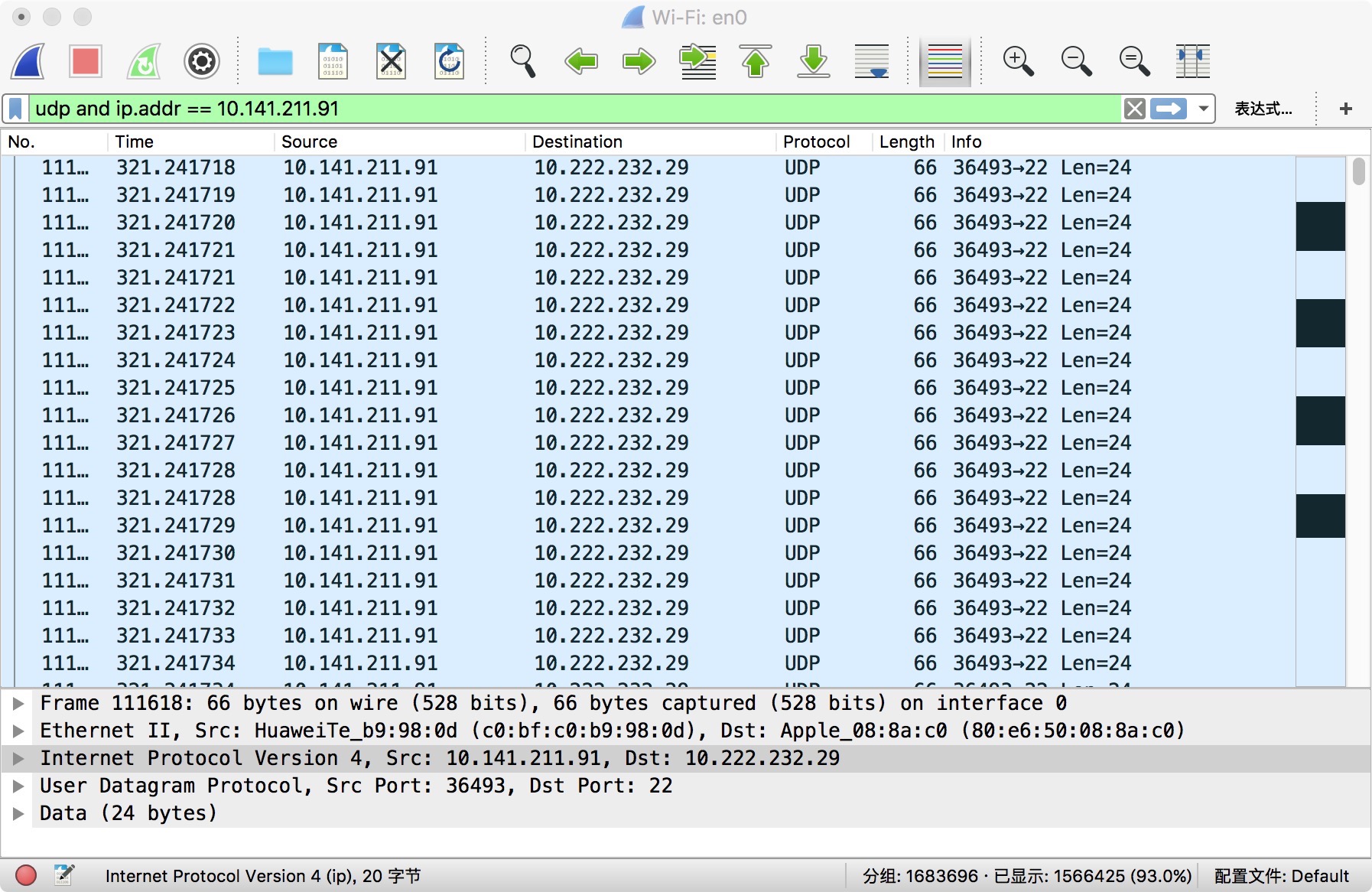
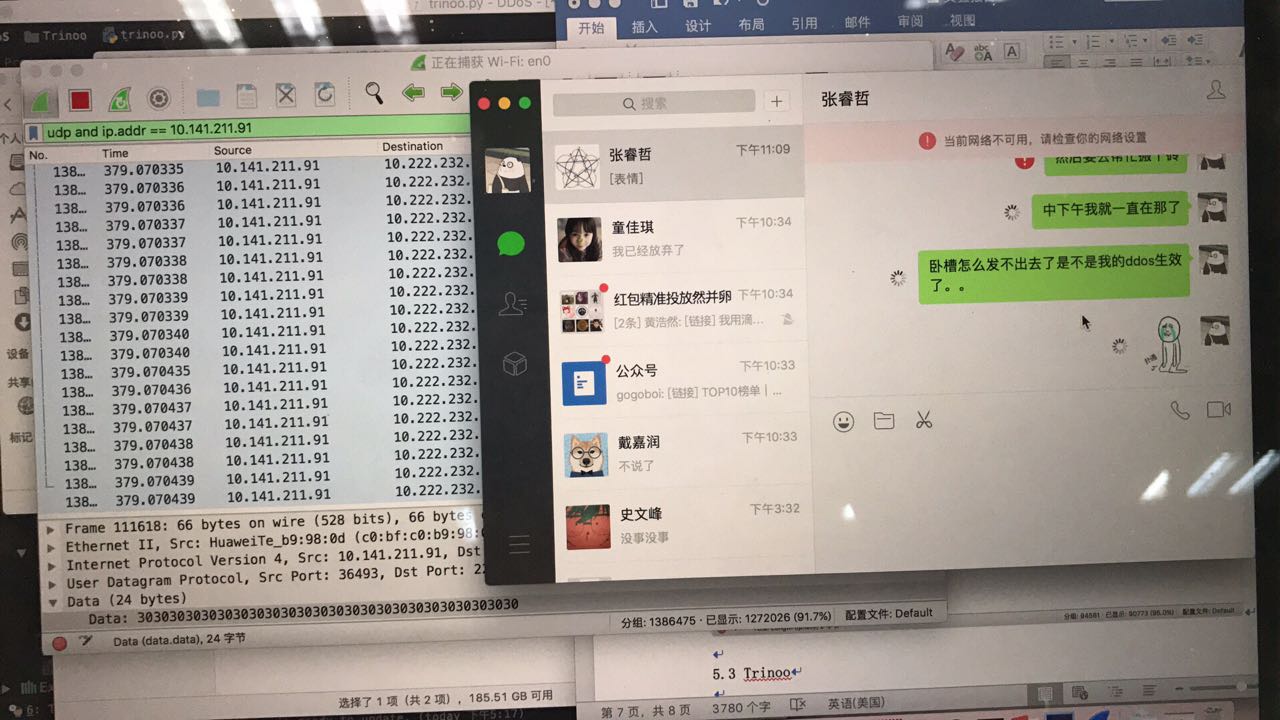


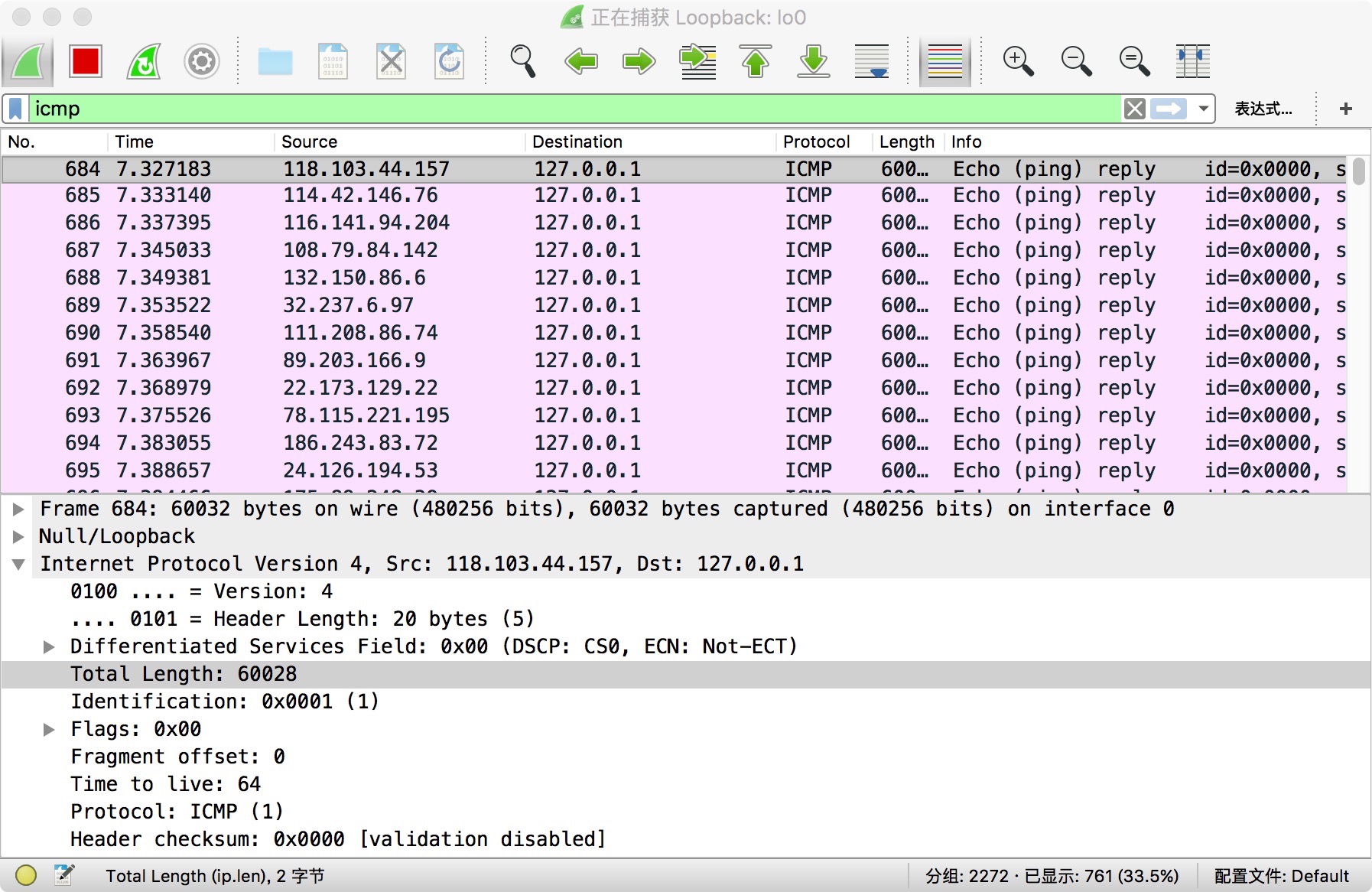
图5: Trinoo DDoS过程中本机网络情况变差



**5.4 ICMP Flood**

实验中使用服务器攻击主机。

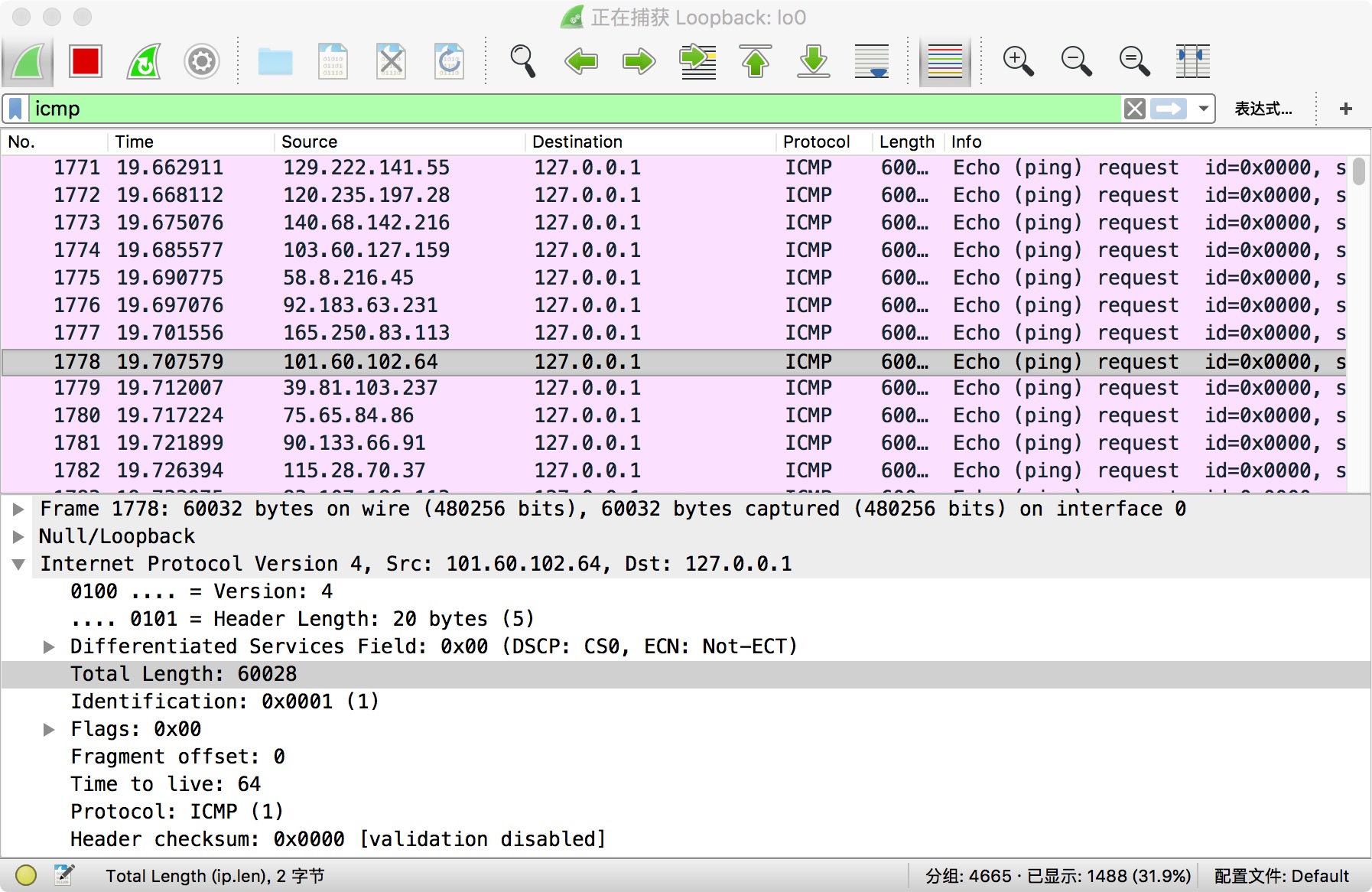
图6: Flood阶段被攻击主机不断收到ICMP Echo (ping) reply数据包



**5.5 Ping of Death**

实验中使用服务器攻击主机。

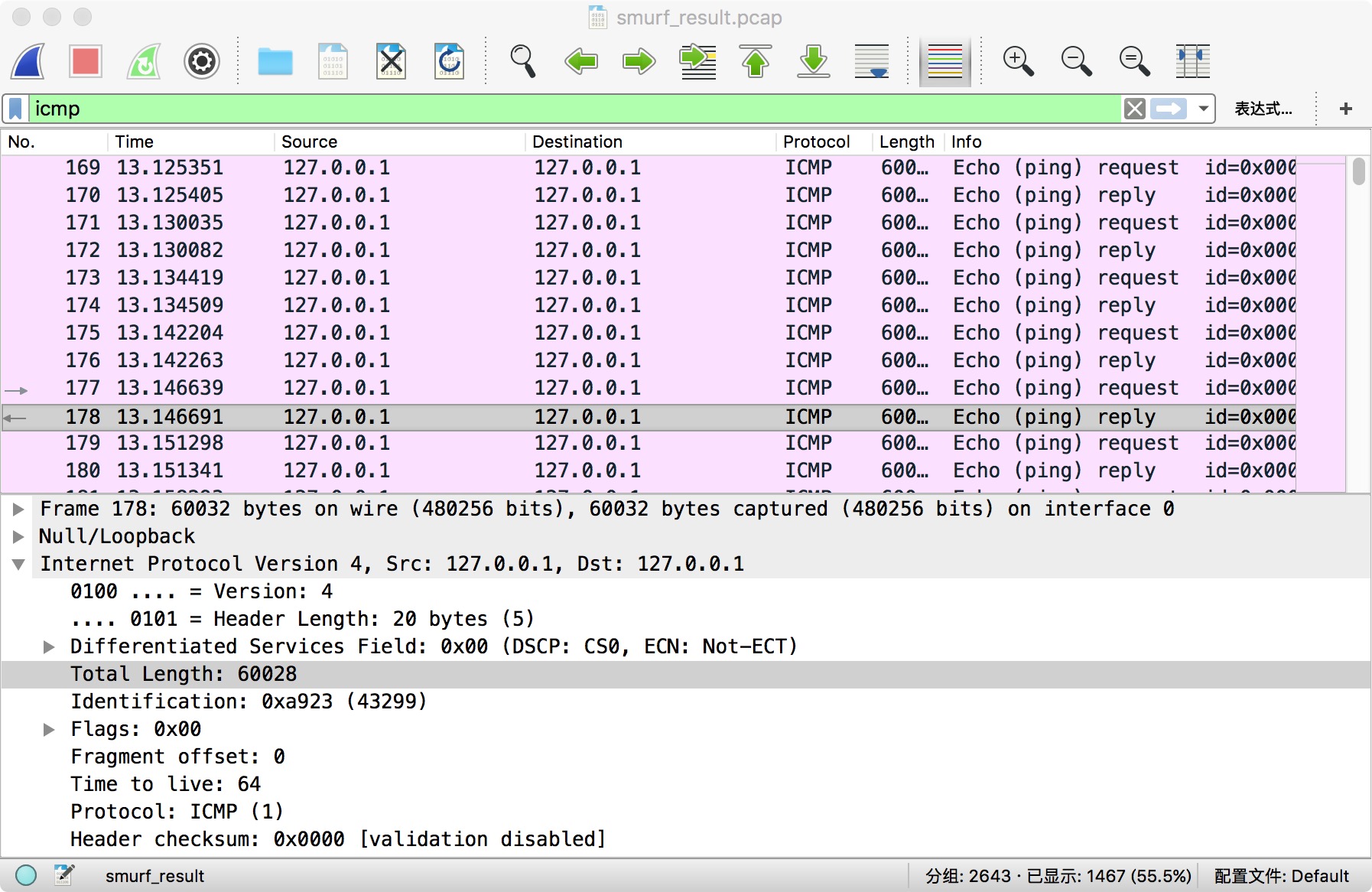
图7: 被攻击主机不断收到ICMP Echo (ping) request数据包

****

**5.6 Smurf**

由于网络环境限制，实验中使用主机攻击本机回环。

图8: 攻击主机不断发出ICMP Echo (ping) request数据包，被攻击收到ICMP Echo (ping) reply数据包



**5.7 Fraggle**

由于网络环境限制，实验中使用主机攻击本机回环。

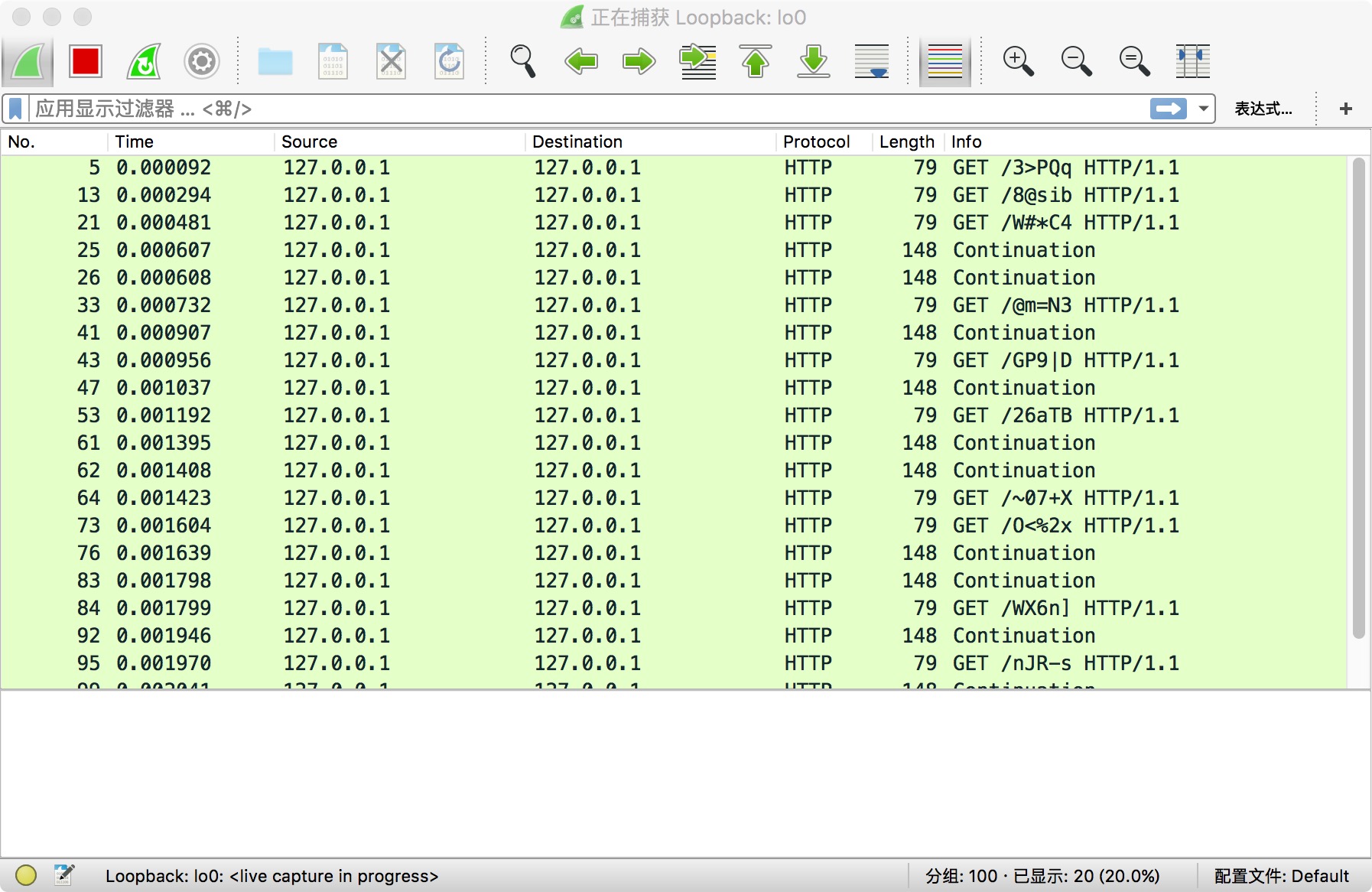
图9: 主机不断发出UDP数据包，但由于子网端口上并没有启动这些功能，所以其唯一的效果就是产生一个ICMP不可达的消息，消耗一定的带宽



**5.8 HTTP Flood**

由于网络环境限制，实验中使用主机攻击本机回环。

图10: 被攻击服务器不断收到“GET /%s HTTP/1.1\r\n\r\n”格式的数据包



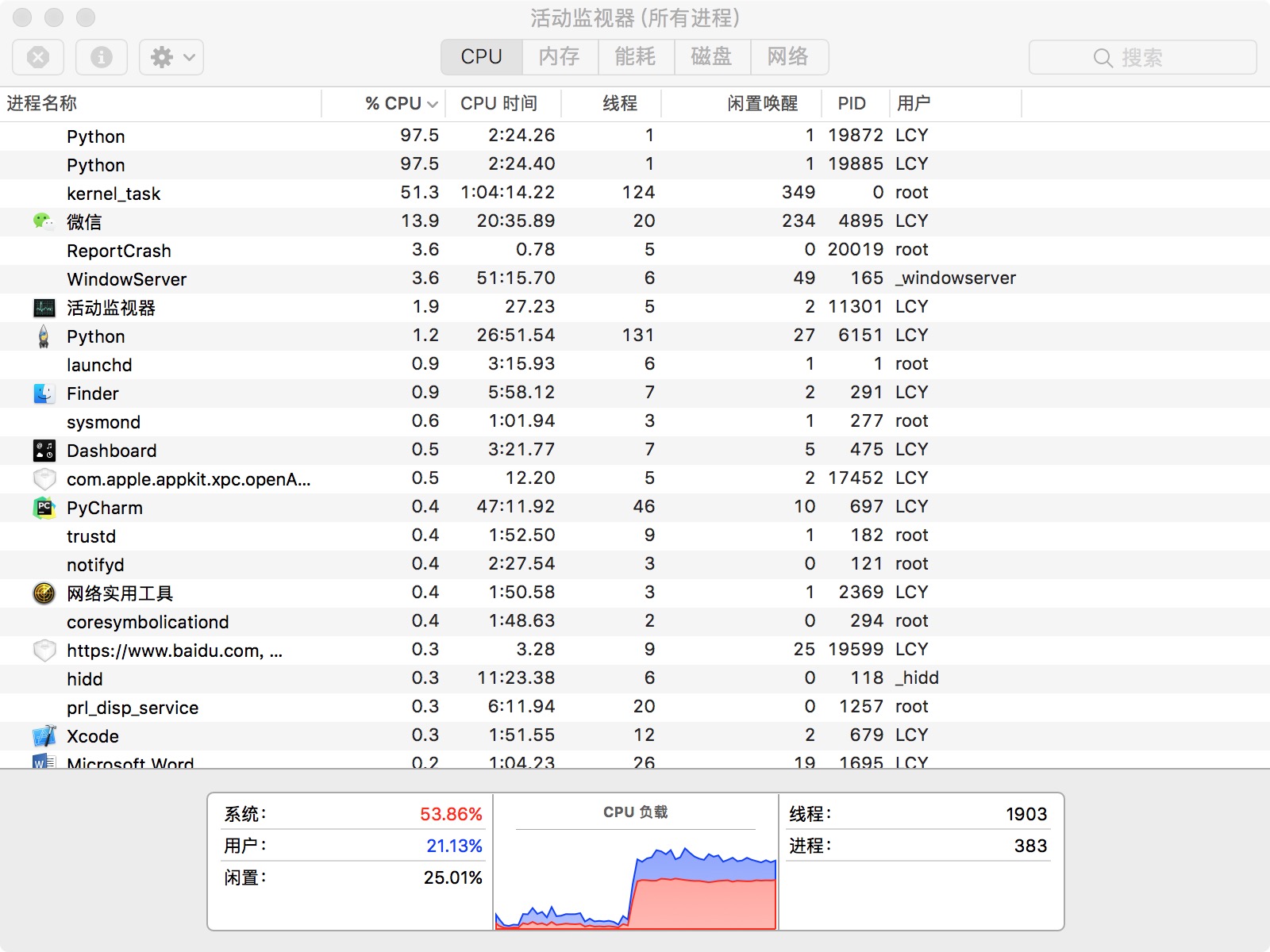
**5.9 Slowloris**

由于网络环境限制，实验中使用主机攻击本机回环。

攻击者在HTTP请求头中要求Web Server保持TCP连接不要断开，随后缓慢地每隔几分钟发送一个key-value格式的数据到服务端，如“a:b\r\n”。在Slowloris中，服务端认为HTTP头部没有接收完成而一直等待，所以并不会返回数据包，本机不能抓到相关的数据包

5.10 效果示例

图11: DDoS占用大量CPU



**6. 注意事项**

在环境配置中，scapy库的依赖项dumbnet配置有一定的难度，具体操作参见“配库”。

在HTTP Flood的编写过程中，很多网上的参考资料对于数据包包头的描述都是不正确的，普遍缺少了一个“\r\n”，这一点非常重要，涉及到实验中是否能够正确地抓到包。

在Slowloris实验过程中，由于服务端认为HTTP头部没有接收完成而一直等待，并不会返回数据包，本机也不能抓到相关的数据包，这应该属于正常现象。

以上DDoS操作都属于较为古老的DDoS方式，只能在局域网内供实验和学习使用。当下还存在的有效的DDoS基本上是直接对电路进行攻击，由于只是所限，很难实现，希望能进一步学习。

调用os.fork()后应该视为多线程攻击，但活动监视器中的线程数仍显示为1，不知道原因。

**7. 亮点和未来工作**

**7.1 亮点**

1）在实验环境下实现了多种DDoS，并且在实验环境下得到了较好的结果。

2）做了较为完整的实验来证实DDoS的确正常工作。

3）在实验中对DDoS攻击和防御手段有了更深刻的理解。

4）实现了少量攻击模式的选择，如：源IP段/端口、目标IP/端口等。

5）尝试了通过threading实现并发，虽然还存在一些问题，但在部分情况下能成功运行

**7.2 未来工作**

1）实现模拟平台，使用户能通过web页面或者GUI界面进行统一控制和操作

2）实现了更多攻击模式的选择

3）实现更多类型的DDoS

4）实现更新、更实用的DDoS

**8. 其他**

感谢15级数学科学学院的谭笑同学，提供了很多DDoS的思路和安全相关知识；感谢戴嘉润同学，在解决HTTP Flood包头数据格式错误方面提供了很大的帮助；感谢数据分析与安全实验室，为实验提供了服务器设备。

**Reference**

[1] http://www.360doc.com/content/12/1218/10/3405077\_254722699.shtml

[2] http://www.2cto.com/net/201303/193828.html

[3] http://www.secdev.org/projects/scapy/

[4] http://history.programmer.com.cn/12874/

[5] https://github.com/gkbrk/slowloris