

Improved Network Security through a
Combined Ethernet Bridge, Firewall and IDS
A Design and Implementation

Vincent Berk

Institute for Security Technology Studies
Dartmouth College
Leiden Institute for Advanced Computer Science
Leiden University

August 7, 2001

Contents

1	Introduction	1
2	Technical Background	1
2.1	Routing and Bridging	1
2.2	An Example	4
2.3	Firewalls	5
2.4	Intrusion Detection	6
2.5	Attack Signatures	7
3	Design	8
3.1	Melting Fuse	8
4	Implementation	9
4.1	Bridge	9
4.2	Firewall	10
4.3	Intrusion Detection System	11
5	Testing	11
6	Conclusion	13
A	Source Code for Intrusion Detection Program	14
B	IP and TCP Protocol Header	25
B.1	IP Header	25
B.2	TCP Header	25
	Bibliography	

1 Introduction

Needless to say network security is a hot issue nowadays. Administrators are desperately trying to cope with the tremendous flow of new security updates and patches in an attempt to keep their networks safe and secure. Since every network is different, security tools need to be configured concisely and correctly, which can be a very difficult task. Due to this sensitive nature a lot of vulnerabilities are created because of misconfiguration. This invariably leads to a need for easy to install and configure network security devices.

In this technical report we introduce a system that can be installed everywhere on a network wire and operates without interaction of an administrator. It will pass along all network traffic searching for attack signatures and pre-attack probes. If those are detected, packets from the attacking Internet host will no longer be passed on, in an attempt to stop the attack prematurely. The concept is explained by using an example design, which will be implemented and tested.

Since this system will have no IP address it will be invisible on the Internet and thus not vulnerable to attack itself. Its easy of use should make it a valuable tool to improve general network security.

2 Technical Background

2.1 Routing and Bridging

To understand the difference between routing and bridging a good look at the OSI and TCP/IP Reference models can be very valuable. (**Figure 1**)

The *Application* layer describes the high-level protocols like *Telnet*, *FTP* and *SSH*. This can be seen as the language spoken, or the rules of engagement for those particular services. It describes the proper way to conduct data transfer between two given points.

The *Transport* layer gets packets from the *Application* layer and will determine the delivery method. This will usually be either *TCP* or *UDP*. This

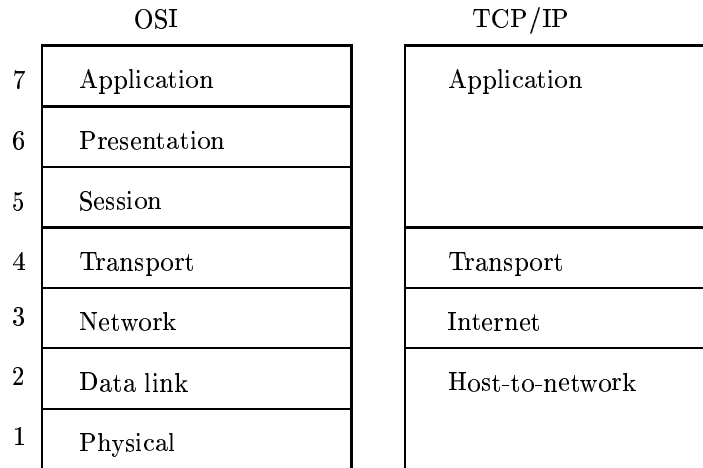


Figure 1: OSI and TCP/IP reference model.

layer defines how packets are delivered, for example the port number and whether or not there needs to be an acknowledgement.

The *Internet* layer defines how packets get from the source host to the destination host. Getting its information from the *Transport* layer it will try to deliver the packet to the given *IP address*. This is the layer where routing takes place.

At the *Host-to-Network* layer packets from the *Internet* layer will be delivered from one host to the next on the path to the final destination. For example, if the link is *Ethernet* then delivery between two hosts will be done using their *MAC* addresses. Bridging takes place at this level. (See [Tan96].)

If data is transferred between two hosts using TCP/IP both hosts will have an IP address. The packet is formed by the sending host, which will consult its routing table to find out where to send the packet first. Usually this routing table will include the definition for the local network and a default route through a *Gateway*. If the packet has to be sent to a machine on the local network (according to the routing table) the communication can be done immediately. If not, the sending host will send the packet to the default route, which will be the gateway. The gateway will once again look at the IP address of the receiver and decide where to send the packet next. The gateway is a router since it handles packets at the Internet layer, using the IP address.

Example of a routing table:

Destination	Gateway	Netmask	Interface
10.20.30.0	*	255.255.255.0	0
default	10.20.30.1	0.0.0.0	0

Whereas routers work at the Internet layer, forwarding packets based on their destination IP address, bridges work at the lowest layer; the Network layer. At the network layer a packet can only be sent between two machines if they are on the same network wire. Assuming Ethernet, this is done based upon the MAC (Media Access Control) addresses. Each interface has a unique MAC address. If a packet is sent out on an ethernet wire it'll be addressed to a MAC address. The interface with that MAC address will copy the packet from the wire and hand it over to the operating system (the remaining layers).

Before continuing with bridging, let's first take a good look at the relationship between IP addresses and MAC addresses. In order for an IP packet to arrive at its destination it has to travel through the network layer. The IP packet has to go out on the wire. It is thus passed on to the lowest layer of the protocol stack. Yet this layer only speaks with MAC addresses. Using a protocol called *ARP* or *Address Resolution Protocol* the correct MAC address is found for the destination IP address. This will be the MAC address of either the destination, or of the gateway that will handle the packet from there on. This will obviously depend on the routing tables. Using the MAC address the packet can now be sent to its destination (or to the gateway router for further handling.)

ARP is basically a broadcast protocol to find out which IP address matches with which MAC address. (Since a broadcast is handled by all machines on the wire.) An ARP request sends out a request to all machines asking who has the specified IP address. The machine with that IP address will respond to the broadcaster and give its MAC address. Now the requesting host can send the packets immediately to that MAC address. (See [Tan96], for an example see the next section.)

Once again, bridging is performed at the network layer. Basically, a bridge connects two physical network segments together and makes it one. This is done by copying packets from one segment to the other, as needed. The decision to copy a packet from one segment to the other is taken based on the destination MAC address. By looking at source MAC addresses a

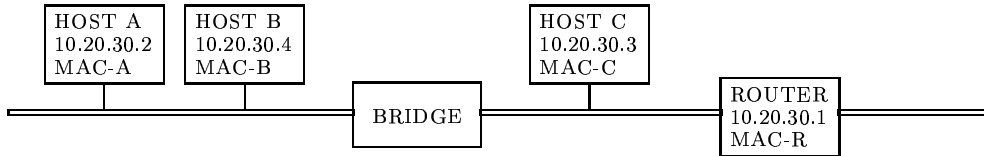


Figure 2: Network Layout

bridge is able to learn which machines are at which side. In fact it does not matter if there is an IP packet contained in the ethernet frame. Machines on either side of the bridge do not have to know about the bridge. Basically all machines on the two segments are considered to be on the same wire and will act according to that. (See IEEE 802.1d Ethernet Bridging).

In contrast to a bridge, a router, operating at one layer higher (the Internet layer) will have to decide what to do with a packet based on the destination IP address. Since not just two segments are considered but the whole Internet, it will not be possible for a router to listen which IP addresses are at which interface. Instead a router has an extensive routing table to decide which IP addresses can be contacted at which interface.

2.2 An Example

To clarify the difference we will now walk through an example. Given are two network segments connected by a bridge (see **Figure 2**). On segment **1** there are two hosts, host *A* (IP 10.20.30.2) and host *B* (IP 10.20.30.4). On segment **2** there is one host *C* (IP 10.20.30.3) and a gateway router with internal IP (IP 10.20.30.1).

The MAC addresses for A, B, C and Router are respectively **MAC-A**, **MAC-B**, **MAC-C** and **MAC-R**. The routing tables are given below. For the router, interface 0 is the internal one (connected to segment 2) and interface 1 is the external one, going to the Internet.

Routing table for A, B and C:

Destination	Gateway	Netmask	Interface
10.20.30.0	*	255.255.255.0	0
default	10.20.30.1	0.0.0.0	0

If *A* wants to send a packet to *C* it'll find in its routing tables that *C* is on the same network, thus the packet can be sent directly. *A* will do an

ARP broadcast asking for the MAC address of the interface holding *C*'s IP address. This broadcast will arrive at host *B*, which will drop it. The bridge however recognizes this as a broadcast and will copy it to segment 2, where host *C* will pick it up and reply to *A* with its MAC address. This packet is once again copied by the bridge, from segment 2 to segment 1. Now *A* can send the packet directly to *C* and, obviously, the bridge will copy that packet from segment 1 to segment 2.

Now for a packet from host *B* out to the Internet. Host *B* finds in its routing tables that the host it is looking for is outside of the local network. Thus the default rule will work and the packet has to be sent to the router. *B* will do an ARP request for the MAC address of the router(!). This is because the router will handle the packet from there. The ARP request will be a broadcast, copied to segment 2 by the bridge, and will be received by the router. The router will send its answer back to *B*, once again over the bridge. Now *B* can send the packet to the router which will handle it from there. All this will happen without any of the machines knowing about the bridge.

2.3 Firewalls

A firewall is a special type of router. Before sending the packets on, they are first inspected using a rulebase. (See **Table 1**.) Generally this rulebase will tell which connections are allowed. Filtering is primarily done on source and destination IP address and source and destination **Port** or **Service**. The port number basically describes what type of service is requested. Through these numbers a firewall can decide which packets are allowed through or not.

The two most common communication protocols in the transport layer are *TCP* or *Transmission Control Protocol* and *UDP* or *User Datagram Protocol*. Both these protocols use port numbers. Each port number represents a different service in the application layer. For example, port 22 is reserved for SSH and port 515 for line printer services. This means that in addition to the IP address a packet will also contain a port number to specify to which service the packet is to be delivered. If a specific service is not available for the Internet then the firewall can block that service port number.

Nowadays firewalls also have the ability to check if the actual content of a packet is valid and if the communication sequence is following the protocol (this is called *Statefull Filtering*). They will assure that a connection

SOURCE	DESTINATION	SERVICE	ACTION
Internet	WWW	HTTP	Accept
Internet	Mail	SMTP/IMAP/POP	Accept
Internet	DNS	DNS	Accept
Internet	Local Net	ANY	Deny
DMZ	Local Net	ANY	Deny
Local Net	Internet	ANY	Accept
Local Net	DMZ	ANY	Accept
ANY	ANY	ANY	Deny

Table 1: Example Firewall Rulebase

is initiated and closed properly. Obviously this protocol will differ depending on what type of communication is done. At the transport layer this is mainly TCP and UDP. Yet moving to the application layer there are many different services and subsequently many different protocols. Obviously, the more a firewall is instructed to check, the slower it gets. In general all protocols in the transport layer are verified and sometimes a few frequently used protocols in the application layer (such as HTTP and SSH). (For more information see [SH95].)

2.4 Intrusion Detection

Intrusion Detection is attempting to detect attack signatures in the packetflow of a network. In order to detect a malicious packet (or sequence of packets) the Intrusion Detection System or *IDS* has to know how to distinguish between good and bad. This is done through so called *Signatures*, describing how to detect a specific attack.

In general an IDS is a system listening on the main network wire of a network to see (and report) what is happening. If a signature is matched it will be reported to the administrator, who will have to decide on the proper course of action. In most cases the detection is a false alarm also called *False Positive*. It is the administrators task to decide what really is an attack or not.

A typical signature could contain an invalid combination of the TCP flags. (See Appendix for IP and TCP headers.) Traditionally the MS Windows NetBIOS system was vulnerable to an attack on port TCP 139. If a

packet with the URGENT bit (the Urgent Offset is valid) arrived at that port the operating system would freeze up. To detect this type of attack a signature would look for any TCP packet with the URGENT bit set heading for port 139 on any machine. This attack is called the WinNuke attack.

2.5 Attack Signatures

The general structure of an attack is to first probe for vulnerabilities and then exploiting those vulnerabilities. In the ideal case an intrusion detection system detects those probes. Probes are done by sending out connect packets to a range of network addresses and a range of port numbers. This way all the computers on a network can be mapped; ie. what ports are open on which machine. An open port means that the machine is listening and accepting connections for the service running behind that port. (See [Nor99].)

In order to find out which IP addresses are up and running in a network a ping sweep can be done. This is simply pinging all IP addresses in a certain network range and recording which IP's reply. The signature for this a set of ICMP echo requests coming from one host going to a range of hosts.

To find out which services are accepting connections on a host a portscan can be performed. Portscans come in many forms but are all aimed at a response difference between open and closed ports. TCP communication is always initiated using a threeway handshake. First the client sends a packet with the SYN flag set. If the service is unavailable (ie. there is no program on the server listening on that port) the server will reply with a reset packet; the RST flag will be set. If there is a server program listening, that program will respond to the client with a SYN-ACK packet. The client is then expected to return an acknowledgement, an ACK packet. To remain undetected, a portscan will break off the threeway handshake before the connection is established. This means that if the server responds with a SYN-ACK the attacker knows the port is open and replies with a RST packet. (For more portscanning methods see [Nor99].) One good way to detect portscanning is when a host sends packets to a wide range of ports on the network.

TCPDUMP transcript of the TCP threeway handshake.

```
11:11:12.093714 hostA.2136 > hostB.netbios-ssn:  
S 1076287380:1076287380(0) win 32120 <mss 1460,
```

```

        sackOK,timestamp 956902707[|tcp]> (DF)
11:11:12.095124 hostB.netbios-ssn > hostA.2136:
        S 3946479594:3946479594(0) ack 1076287381 win 17520 <mss 1460,
        nop,wscale 0,nop,nop,timestamp[|tcp]> (DF)
11:11:12.095153 hostA.2136 > hostB.netbios-ssn:
        . ack 1 win 32120 <nop,nop,timestamp 956902707 0> (DF)

```

Because different operating systems implement their TCP/IP stacks differently (the kernel software for handling network communication), various vulnerabilities exist purely based on implementation inconsistencies. An example was the WinNuke attack described earlier. Others include invalid combinations of TCP flags, such as FIN/PSH/URG, SYN/FIN, SYN/RST or no flags at all. Also these form clear detection signatures.

3 Design

3.1 Melting Fuse

The design proposed in this paper is to combine a firewall and an intrusion detection system on a bridge, instead of a router. Ultimately this device will have two network interfaces so that a network wire can be cut in half and be reconnected by the bridge. Since from both segments are processed by the bridge in the middle, none of the machines on the network wire need to be reconfigured. (There actually is nothing that *can* be reconfigured with this setup.)

Besides the required bridge, this machine can also run either a firewall, an intrusion detection system, or both. From now on the latter option is assumed; both a firewall and an IDS. By coupling the IDS and the firewall an active defense system can be built. As soon as the IDS detects an attack (or a pre-attack probe) the firewall can be instructed to block the incoming packets coming from the hostile machine. The system can be seen as a 'Melting Fuse' in a network wire.

Basically, the 'Melting Fuse' will allow all network traffic through (the bridge copies the packets.) As soon as hostile activity is from any IP address, the packets for that specific IP address will be blocked for a certain amount of time, ie. the fuse melted through.

Since this system is implemented on top of an ethernet bridge, it will not be visible at the internet layer. This basically means that the system will not have an IP address and thus cannot be addressed from anywhere else but the keyboard physically connected to the back of that machine. This renders the system completely immune to attacks from outside. Internet routed packets will not get to the machine since it cannot be addressed.

4 Implementation

4.1 Bridge

In order to implement this design a host running the Linux operating system is used. The bridge requires two network interfaces; eth0 and eth1. The kernel is specifically configured **not** to route packets. This will be the job of the bridge. For the bridge the *Linux Bridging Project* is used (see www.math.leidenuniv.nl/home/buytenh/bridge), which has basic firewalling capabilities built in. Also it has the option of connecting more than two interfaces together in a bridge.

Modern versions of the Linux Kernel (2.4.0, see www.kernel.org) have bridge support built in. It can be compiled in with **CONFIG_BRIDGE=y** option. Also, both ethernet interfaces should be recognized by the kernel. To make it work a configuration tool is needed; **brctl**, this is contained in the **bridge-utils** package. (www.openrock.net/bridge) Below is the sequence of commands to activate a bridge between two network interfaces:

```
ifconfig eth0 0.0.0.0 up promisc
ifconfig eth1 0.0.0.0 up promisc
brctl addbr br0
brctl addif br0 eth0
brctl addif br0 eth1
ifconfig br0 0.0.0.0 up
```

The **ifconfig** statements activate the network interfaces without an IP address (0.0.0.0) in *Promiscuous Mode*. When an interface is in promiscuous mode it will pass all packets on the wire to the operating system, whether they are addressed to its MAC address or not. This is also called *sniffing*. Next the bridge between the interfaces is created, called **br0**. Basically **br0** is itself a network interface that could be given an IP address, yet this would

defy its status as an invisible network device. As soon as these commands are given the bridge will start to learn which MAC addresses are at which interface. A similar set of commands can be given to deactivate the bridge.

4.2 Firewall

The Linux bridge comes with basic firewall capabilities. Yet a patch needs to be applied to the kernel bridging code to enable the firewall. The firewall can be controlled with the **ipchains** tool, which is designed to set up the firewall rules in the Linux kernel. Since the normal forwarding is disabled in the kernel (only routers use kernel packet forwarding) a new *Chain* has to be defined that is linked to the **br0** interface. Below are the commands to set up the firewall and two examples to add and remove rules from the rulebase:

```
# Creating a new firewall rulebase (chain) for the br0 interface:
ipchains -N br0

# Adding a rule:
ipchains -A br0 -s [SOURCE_ADDRESS] -i eth0 -j DENY -1
ipchains -A br0 -s [SOURCE_ADDRESS] -i eth1 -j DENY -1

# Removing a rule:
ipchains -D br0 -s [SOURCE_ADDRESS] -i eth0 -j DENY -1
ipchains -D br0 -s [SOURCE_ADDRESS] -i eth1 -j DENY -1
```

The rules are currently applied to both ethernet interfaces. **[SOURCE_ADDRESS]** can be a single host or a whole network range. Ultimately rules should also be created to block any packets going to the hostile host. This can be done as follows:

```
ipchains -A br0 -d [SOURCE_ADDRESS] -i eth0 -j DENY -1
ipchains -A br0 -d [SOURCE_ADDRESS] -i eth1 -j DENY -1
```

For further reference see the **ipchains** documentation. Note that filtering is done on IP addresses.

4.3 Intrusion Detection System

Intrusion Detection Systems are generally expensive and the source code is unavailable. Although many commercial IDS' could have been used we choose to implement our own. The core of the system is the *pcap* library *libpcap*, which allows packets to be captured from any given interface (<ftp://ftp.ee.lbl.gov/libpcap.tar.Z>). For each packet it captures, libpcap calls a callback-function with a pointer to the start of the packet in memory. The callback-function is user specified.

A distinction is made in stateless and statefull checks. Stateless checks only look at the current packet while statefull checks require a history of packets previously captured. The code presented in the appendix does stateless checking for packets with IP options set (these are usually source routed packets) and for several invalid combinations of TCP flags.

In order to do statefull checking a history is to be kept of previously captured packets. This is done with a hash table using the source IP address of the packet as the key. While inserting new packet summaries in the hash table, expired ones are removed. Packets expire with time.

For each packet the hash table is checked for the IP address of the current packet (the one being processed and inserted). If more than a certain amount of ping (ICMP echo request) packets was detected in a limited amount of time, the host is asumed to be doing either a ping sweep or a flood ping.

This code is not complete and does not represent a full scale intrusion detection system, yet it is a good example of the possibilities.

5 Testing

Given is a network (see **Figure 3**) setup with two ethernet segments connected by the bridge, firewall and IDS combination. On segment **1**, connected to bridge interface *eth0*, is a host with IP address 10.20.30.2 and hostname *A*. Segment **2** is connected to bridge interface *eth1* and has another host with IP address 10.20.30.3, with hostname *B*. In this test both a floodping and a scan with invalid TCP flags will be attempted from host *A* to host *B*.

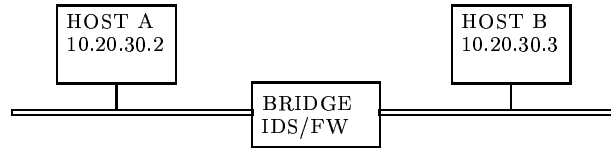


Figure 3: Network Layout

For the flooding the UNIX tool **ping** is used on host *A*:

```

hostA# ping -f 10.20.30.3
PING hostB (10.20.30.3): 56 data bytes
.....
--- hostB ping statistics ---
296 packets transmitted, 66 packets received, 77% packet loss
round-trip min/avg/max = 0.1/0.7/1.3 ms
hostA#
  
```

```

bridge# ipchains -N br0
bridge# ./project
Blocking host: 10.20.30.2   reason: ping sweep or flood ping detected
  
```

To scan *B* nmap (See www.insecure.org/nmap/) will be used. This tool allows to scan a remote host using Xmas packets, which have at least the FIN, PSH and URG TCP flags set. Below are the commands and the response of the IDS program: (Note that nmap reports **all** ports to be open on host *B*, this is because no replies with the XMAS scan also means that the port is opened.)

```

hostA# nmap -sX 10.20.30.3

Starting nmap V. 2.12 by Fyodor (fyodor@dhp.com, www.insecure.org/nmap/)
Interesting ports on hostB (10.20.30.3):
Port      State      Protocol  Service
1         open      tcp       tcpmux
2         open      tcp       compressnet
3         open      tcp       compressnet
4         open      tcp       unknown
5         open      tcp       rje
6         open      tcp       unknown
  
```

```

7      open      tcp      echo
8      open      tcp      unknown
9      open      tcp      discard
10     open      tcp      unknown
11     open      tcp      systat
12     open      tcp      unknown
.      .         .         .
.      .         .         .
.      .         .         .

```

```

Nmap run completed -- 1 IP address (1 host up) scanned in 3 seconds
hostA#

```

```

bridge# ipchains -N br0
bridge# ./project
Blocking host: 10.20.30.2   reason: TCP flags XMAS

```

On both occasions the IDS detected the activity and blocked the hostile host from crossing the bridge. Obviously this will also work when host *A* and host *B* or both were to be out on the Internet behind one or more router. (Of course as long as the packets pass through the bridge.) This is because both attacks use IP routed packets.

6 Conclusion

In this paper a system was proposed to greatly improve network security. By combining a firewall and an IDS on an ethernet bridge many common network attacks can be stopped in a fraction of a second. Although administrator interaction will remain necessary to prevent false detections from crippling their networks, no further maintenance is required. This product combines easy of use and quick installation with unlimited capabilities for those who have more time and skills.

A Source Code for Intrusion Detection Program

```
/*
 * Note:
 * This is an example intrusion detection system.
 * It is not meant to be complete but to serve as an example.
 *
 * Author: Vincent Berk (c) April 2000
 */

#include <pcap.h>
#include <stdio.h>
#include <stdlib.h>

#define EBUFLLEN 1500
#define CMDLEN 1024

#define ICMP 0x01
#define TCP 0x06
#define UDP 0x11

/* Amount of IP source hash entries and maximum age for stored packets */

#define IPHASH 0x3FF
#define PLAGE 3

/* Maximum amount of pings a host may do in PLAGE seconds */

#define MAXPINGS 20
#define ICMP_ECHO 8

/* TCP flags */
```

```
#define URG 0x20
#define ACK 0x10
#define PSH 0x08
#define RST 0x04
#define SYN 0x02
#define FIN 0x01
```

```
/* Structs */
```

```
struct IPv4_packet_summary
{
    /* IP header specific */
    struct timeval ts;
    unsigned int saddr;
    unsigned int daddr;
    unsigned int len;
    u_char protocol;

    /* TCP / UDP specific */
    unsigned short int sport;
    unsigned short int dport;

    /* For the packet list */
    struct IPv4_packet_summary *next;
};
```

```
/* This structure is used to keep track of hosts already blocked */
```

```
struct IPv4_blocked_address
{
    struct timeval ts;
    unsigned int saddr;
    struct IPv4_blocked_address *next;
};
```

```

/* Global data structures. */

struct IPv4_blocked_address *BL;          /* List of blocked IP's */
struct IPv4_packet_summary **PL;        /* List of recently captured packets */
char *cmd;                               /* Command line command. */
struct pcap_pkthdr pcap_header;        /* incoming packet header copy */

/* Code start */

/* Issues the command to block host using IPCHAINS */
void block_host ( const struct IPv4_packet_summary *ps, char *m )
{
    struct IPv4_blocked_address *b;

    /* Search entry for this IP address */
    b = BL;
    while ( b && (*b).saddr != (*ps).saddr )
        b = (*b).next;

    /* Insert new */
    /* AND BLOCK */
    if ( !b )
    {
        b = malloc ( sizeof ( struct IPv4_blocked_address ) );
        (*b).next = BL;
        BL = b;
        (*b).saddr = (*ps).saddr;

        /* Log and block */
        snprintf ( cmd, CMDLEN, "./blockhost %i.%i.%i.%i\n",
            ( (*b).saddr & 0xFF000000 ) >> 24,
            ( (*b).saddr & 0x00FF0000 ) >> 16,
            ( (*b).saddr & 0x0000FF00 ) >> 8, (*b).saddr & 0x000000FF );
        fprintf ( stderr, "Blocking host: %i.%i.%i.%i  reason: %s\n",

```

```

        ( (*b).saddr & 0xFF000000 ) >> 24,
        ( (*b).saddr & 0x00FF0000 ) >> 16,
        ( (*b).saddr & 0x0000FF00 ) >> 8, (*b).saddr & 0x000000FF, m );
    system ( cmd );
}

/* Copy timestamp */
(*b).ts.tv_sec = (*ps).ts.tv_sec;
(*b).ts.tv_usec = (*ps).ts.tv_usec;

}

/* Check if packet has IP options set. p[0] is first byte of IP packet. */
inline void no_IP_options ( const u_char *p, const struct IPv4_packet_summary *ps )
{
    if ( ( p[0] & 0x0F ) > 5 )
        block_host ( ps, "IP options set" );
}

/* Check if the packet has malicious TCP flags set. Asumed p[0] is IP packet
 * start, carrying TCP payload.
 */
inline void check_TCP_flags ( const u_char *p, const struct IPv4_packet_summary *ps )
{
    u_char f = p[33];
    /* We asume no IP options */

    if ( ( f & FIN ) && ( f & PSH ) && ( f & URG ) )
        block_host ( ps, "TCP flags XMAS" );
    else if ( ( f & SYN ) && ( f & FIN ) )
        block_host ( ps, "TCP flags SYN/FIN" );
    else if ( ( f & SYN ) && ( f & RST ) )
        block_host ( ps, "TCP flags SYN/RST" );
    else if ( ( f & 0x3F ) == 0 )
        block_host ( ps, "TCP flags NULL" );
}

```

```

/* Nicely formatted HEX dump of a packet */

void hexdump_packet ( const u_char *p, const struct pcap_pkthdr *pcap_header )
{
    int i;
    int l = (*pcap_header).len;

    for ( i = 0 ; i < l ; i += 2 )
    {
        if ( i % 16 == 0 && i != 0 )
            fprintf ( stdout, "\n" );
        fprintf ( stdout, " %02x", p[i] );
        if ( i + 1 < l )
            fprintf ( stdout, "%02x", p[i + 1] );
    }
}

```

```

/* Insert packet summary into the hashed packet stack and remove old.
 * update_PL is also responsible for freeing the memory used by the ps!
 */

```

```

void update_PL ( struct IPv4_packet_summary *ps )
{
    time_t rt;          /* Removal time. */
    int i;
    struct IPv4_packet_summary *c; /* current */
    struct IPv4_packet_summary *n; /* next */

    /* Hashed index in PL, source address. */
    i = (*ps).saddr & IPHASH;

    /* Remove old first. Take ps ts_sec as current 'time'.
     * Since packets are inserted in chronological order,
     * find first out-of-date one. */
    rt = (*ps).ts.tv_sec - PLAGE;

    /* Insert */
    (*ps).next = PL [i];
    PL [i] = ps;
}

```

```

/* Remove old entries. */
c = ps;
n = (*c).next;
/* Move to first ps to be freed and close the list. */
while ( n )
{
    /* If the next is to old, close the list and zap the rest. */
    if ( (*n).ts.tv_sec < rt )
    {
        (*c).next = NULL;
        /* Clean out the list. */
        while (n)

            {
                c = n;
                n = (*c).next;
                free (c);
            }
    }
    /* If not, move on ... */
    else
    {
        c = n;
        n = (*c).next;
    }
}
}

/* If the sender of the current packet send more than MAXPINGS
 * in the last PLAGE seconds we consider its activity malicious.
 * p[0] is IP packet
 */
void check_ping_sweep ( const u_char *p, const struct IPv4_packet_summary *ps )
{
    int pings;
    int i, s;
    struct IPv4_packet_summary *c;

```

```

/* Hashed index in PL, source address. */
pings = 0;
i = (*ps).saddr & IPHASH;
c = PL [i];
s = (*ps).saddr;

while ( c != NULL )
{
    if ( (*c).saddr == s && p[9] == ICMP && p[20] == ICMP_ECHO )
        pings += 1;
    c = (*c).next;
}

if ( pings > MAXPINGS ) block_host ( ps, "ping sweep or flood ping detected" );
}

/* Handler function for IPv4 packets. p[0] is IPv4 packet start. */

void IPv4_handler ( u_char *user, const struct pcap_pkthdr *pcap_header,
                  const u_char *p )
{
    struct IPv4_packet_summary *ps;
    struct IPv4_blocked_address *b;

    /* DEBUG: print some IP info and the packet. */
#ifdef DEBUG
    fprintf ( stdout, "\n\nPacket length: %i\n", (*pcap_header).len );
    fprintf ( stdout, "Source IP: %i.%i.%i.%i\n",
              (int) p[12], (int) p[13], (int) p[14], (int) p[15] );
    fprintf ( stdout, "Destination IP: %i.%i.%i.%i\n",
              (int) p[16], (int) p[17], (int) p[18], (int) p[19] );
    hexdump_packet ( p, pcap_header );
#endif

    /* Serious stuff. */

    /* Create the IP packet summary */

    ps = malloc ( sizeof ( struct IPv4_packet_summary ) );
    /* Is freed by either insert_PL or later in this function. */

```

```

(*ps).ts.tv_sec = (*pcap_header).ts.tv_sec;
(*ps).ts.tv_usec = (*pcap_header).ts.tv_usec;
(*ps).len = (*pcap_header).len;
(*ps).saddr = ( ( (int) p[12] ) << 24 ) + ( ( (int) p[13] ) << 16 ) +
              ( ( (int) p[14] ) << 8 ) + ( (int) p[15] );
(*ps).daddr = ( ( (int) p[16] ) << 24 ) + ( ( (int) p[17] ) << 16 ) +
              ( ( (int) p[18] ) << 8 ) + ( (int) p[19] );
(*ps).protocol = p[9];
(*ps).next = NULL;

if ( p[9] == UDP || p[9] == TCP )
{
    (*ps).sport = ( (int) p[20] << 8 ) + (int) p[21];
    (*ps).dport = ( (int) p[22] << 8 ) + (int) p[23];
}

/* Stateless checks. */

/* check IP_options
 * check TCP_flags
 */
no_IP_options ( p, ps );
if ( p[9] == TCP )
    check_TCP_flags ( p, ps );

/* Search if this IP is currently being blocked. */
b = BL;
while ( b && (*b).saddr != (*ps).saddr )
    b = (*b).next;

/* Attention: This if-statement frees the ps! */
if ( !b ) /* not blocked already */
{
    update_PL ( ps ); /* responsible for freeing of ps later on!!! */

    /* Statefull checks. */
    /* Also, this is where the treading should be done ... */
    check_ping_sweep ( p, ps );
}

```

```

    else
        free ( ps );
}

/* Deals with an ethernet packet. */

void ether_handler ( u_char *user, const struct pcap_pkthdr *h, const u_char *p )
{
    /* Copy for editing.*/
    /* pcap_header is a global, saves us lots of mallocing and freeing. */
    pcap_header.len = (*h).len;
    pcap_header.caplen = (*h).caplen;
    pcap_header.ts.tv_sec = (*h).ts.tv_sec;
    pcap_header.ts.tv_usec = (*h).ts.tv_usec;

    /* We want the whole packet. */
    if ( pcap_header.caplen != pcap_header.len )
        return;

    /* Is this a IPv4 packet? */
    if ( ( p[14] & 0xF0 ) == 0x40 )
    {
        /* Strip Ethernet header and trailer:
         * header + trailer = 14 + 4 bytes. */
        pcap_header.len -= 18;
        pcap_header.caplen -= 18;

        if ( pcap_header.caplen < 0 )    /* Nothing to do for IPv4_handler */
            return;

        /* IP handler handles the IPv4 packet. */
        IPv4_handler ( user, &pcap_header, p + 14 );
    }

    return;
}

```

```

int main ( int argc, char **argv )
{
    static pcap_t *pd;      /* pcap descriptor */
    char *ebuf;            /* error buffer */
    int pc;                /* Packet count */
    pcap_handler callback; /* Callback function for pcap_dispatch/loop */
    u_char *user;          /* our whatever thingy */
    int i;

    BL=NULL;    /* list of blocked addresses */

    /* List of packets, sorted by IPHASH */
    PL = malloc ( ( IPHASH +1 ) * sizeof ( struct IPv4_packet_summary* ) );
    for ( i = 0 ; i < IPHASH +1 ; i ++ )
        PL [i] = NULL;

    /* Message buffer and command line argument. */
    ebuf = malloc ( EBUFLen * sizeof ( char ) );
    cmd = malloc ( CMDLEN * sizeof ( char ) );

    /* Open device br0 */
    pd = pcap_open_live ( "br0", 1500, 1, 1000, ebuf );
    /* pd = pcap_open_offline ( "dump", ebuf );*/

    if ( pd == NULL )
    {
        fprintf ( stderr, "Could not open device br0, error: %s\n", ebuf );
        exit (1);
    }

    /* Do some packet dumping, function should never return */
    callback = ether_handler;
    pc = pcap_loop ( pd, -1, callback, user );

    pcap_close ( pd );

```

```
    return 0;  
}
```

B IP and TCP Protocol Header

B.1 IP Header

B.2 TCP Header

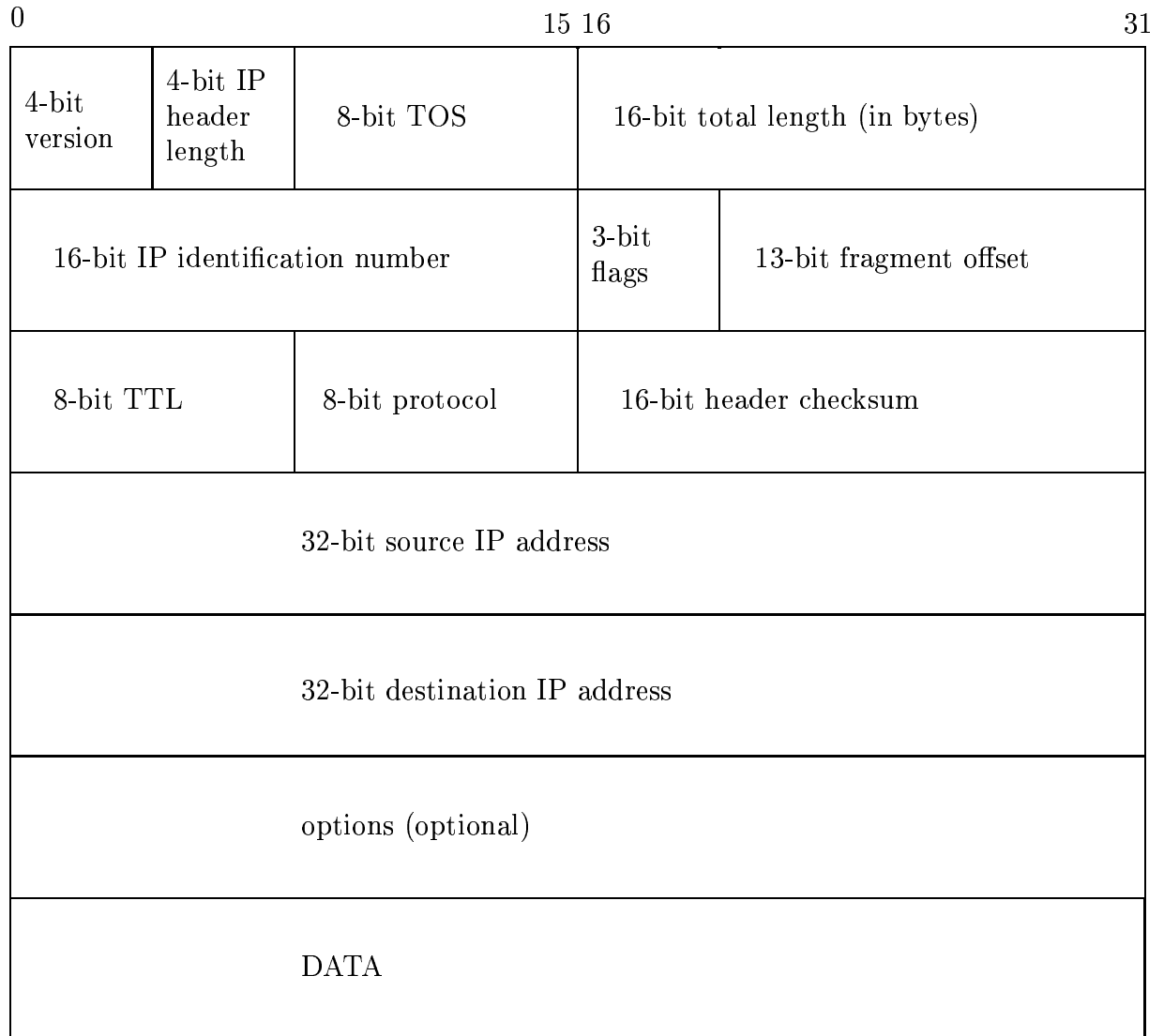


Figure 4: IP Protocol Header

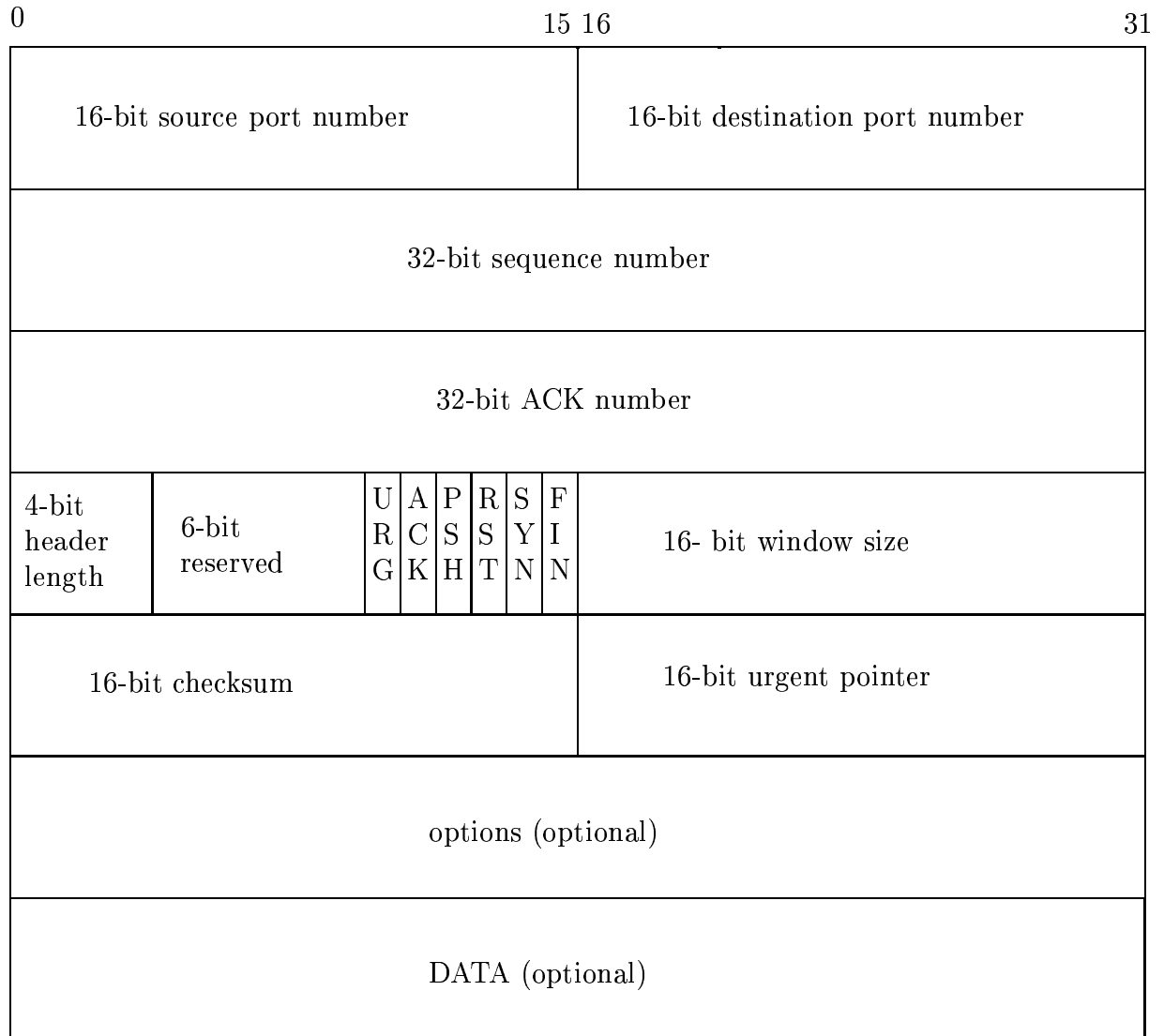


Figure 5: TCP Protocol Header

References

- [Bre99] Chris Brenton. *Mastering Network Security*. Sybex Network Press, 1999.
- [Nor99] Stephen Northcutt. *Network Intrusion Detection*. New Riders, 1999.
- [NSB00] Stephen Northcutt, Lance Spitzner, and Chris Brenton. Sans track 2: Firewalls, perimeter protection and virtual private networks. Course Material, october 2000. SANS Network Security 2000, Monterey California.
- [SH95] Karanjit Siyan and Chris Hare. *Internet Firewalls and Network Security*. New Riders, 1995.
- [Tan96] Andrew Tanenbaum. *Computer Networks*. Prentice Hall PTR, 1996.

This paper was typeset using \LaTeX 2 ϵ and \BIBTeX and a lot of tasty Ben&Jerry's!!!