MARCH 1989

WRL
Research Report 89/4

Simple and Flexible Datagram Access Controls for Unix-based Gateways

Jeffrey C. Mogul

Western Research Laboratory 100 Hamilton Avenue Palo Alto, California 94301 USA

The Western Research Laboratory (WRL) is a computer systems research group that was founded by Digital Equipment Corporation in 1982. Our focus is computer science research relevant to the design and application of high performance scientific computers. We test our ideas by designing, building, and using real systems. The systems we build are research prototypes; they are not intended to become products.

There is a second research laboratory located in Palo Alto, the Systems Research Center (SRC). Other Digital research groups are located in Paris (PRL) and in Cambridge, Massachusetts (CRL).

Our research is directed towards mainstream high-performance computer systems. Our prototypes are intended to foreshadow the future computing environments used by many Digital customers. The long-term goal of WRL is to aid and accelerate the development of high-performance uni- and multi-processors. The research projects within WRL will address various aspects of high-performance computing.

We believe that significant advances in computer systems do not come from any single technological advance. Technologies, both hardware and software, do not all advance at the same pace. System design is the art of composing systems which use each level of technology in an appropriate balance. A major advance in overall system performance will require reexamination of all aspects of the system.

We do work in the design, fabrication and packaging of hardware; language processing and scaling issues in system software design; and the exploration of new applications areas that are opening up with the advent of higher performance systems. Researchers at WRL cooperate closely and move freely among the various levels of system design. This allows us to explore a wide range of tradeoffs to meet system goals.

We publish the results of our work in a variety of journals, conferences, research reports, and technical notes. This document is a research report. Research reports are normally accounts of completed research and may include material from earlier technical notes. We use technical notes for rapid distribution of technical material; usually this represents research in progress.

Research reports and technical notes may be ordered from us. You may mail your order to:

Technical Report Distribution
DEC Western Research Laboratory, UCO-4
100 Hamilton Avenue
Palo Alto, California 94301 USA

Reports and notes may also be ordered by electronic mail. Use one of the following addresses:

Digital E-net:	`DECWRL::WRL-TECHREPORTS`
DARPA Internet:	`WRL-Techreports@decwrl.dec.com`
CSnet:	`WRL-Techreports@decwrl.dec.com`
UUCP:	`decwrl!wrl-techreports`

To obtain more details on ordering by electronic mail, send a message to one of these addresses with the word ‘‘`help`’’ in the Subject line; you will receive detailed instructions.

Simple and Flexible Datagram Access Controls for Unix-based Gateways

Jeffrey C. Mogul

March, 1989

Western Research Laboratory 100 Hamilton Avenue Palo Alto, California 94301 USA

Abstract

Internetworks that connect multiple organizations create potential security problems that cannot be solved simply by internal administrative procedures. Organizations would like to restrict inter-organization access to specific restricted hosts and applications, in order to limit the potential for damage and to reduce the number of systems that must be secured against attack. One way to restrict access is to prevent certain packets from entering or leaving an organization through its gateways. This paper describes simple, flexible, and moderately efficient mechanisms for screening the packets that flow through a Unix-based gateway.

This research report is a preprint of a paper
to appear at the *Summer 1989 USENIX Technical Conference.*

1. Introduction

Internetworking has greatly improved communication between administratively distinct organizations, linking businesses, schools, and government agencies to their common benefit. Unfortunately, internetworks that connect multiple organizations create potential security problems that cannot be solved by the mechanisms used within organizations, such as restricting physical access. In particular, interconnection at the datagram level is an ‘‘all or none’’ mechanism, allowing outsiders access to all the hosts and applications of an organization on the internetwork. To avoid penetration, every host within an organization must be made secure, no small feat when it involves tens of thousands of poorly-managed workstations.

We would like to be able to restrict inter-organization access to specific hosts and applications. Doing so limits the potential for damage, and reduces the number of systems that must be secured against attack. One approach is to place an application-level gateway between the organization and the internetwork, eliminating packet-level access but supporting a small set of approved and presumably bullet-proof applications. Typically, these include electronic mail, name service, and perhaps remote terminal access.

This approach is painful because it requires writing application gateway software for each approved application, and because it may be too draconian for some organizations. It also significantly reduces performance.

A more flexible way to control access is to prevent certain packets from entering or leaving an organization through its gateways. This allows greater flexibility than an application-level gateway, although as with any power tool it also requires greater vigilance. Various commercial gateway systems provide such a mechanism [2, 27] and it has also been treated in the literature [10].

Although it is not a good idea to use a Unix® system as an internetwork gateway, the popularity of 4.2BSD Unix (and its successors and derivatives), coupled with bureaucratic inertia, has led numerous organizations to use Unix-based gateways. The mechanisms described in this paper allow users of Unix-based gateways to impose packet-level access controls without major changes to existing software. Perhaps more important, by separating mechanism from policy [17] so that arbitrarily precise access control policies may be developed as ordinary Unix user processes, these mechanisms support fine-tuning and experimentation that would be difficult using commercial gateway products.

Section 2 of this paper sets out the background and goals of this project, and describes previous work in the field. Section 3 presents the design and implementation of a simple, flexible, and efficient modification to the Unix kernel. Sections 4 and 5 then describe two different user-level daemons that make use of this modified kernel to provide packet-level access control. Section 6 discusses how the new mechanism may be used for purposes besides access control.

2. Background and Goals

The Internet Protocol (IP) suite [15] is today the dominant means of connecting disparate organizations into an internetwork. Virtually all of the practical and experimental work on datagram access controls has been done using IP protocols. Although the examples in this paper do assume the use of IP, much of the mechanism (especially the kernel support) should be applicable to any similar protocol suite, including OSI [30].

All datagrams in an IP internetwork carry an IP header [24], which includes the source and destination host addresses. In general, this is all that an IP gateway may assume about a datagram, so one might choose to restrict access on a host-by-host basis. (Since IP network numbers can be extracted from IP host addresses, and since network numbers can often be identified with specific organizations, an IP-level mechanism might also restrict datagrams based on source or destination network number.)

In fact, however, almost all information is carried by transport protocols layered above IP. Primarily, these include TCP [25] for reliable byte-stream applications (mail, file transfer, remote terminal access), and UDP [23] for request-response protocols (name service, routing, the NFS file access protocol [28]). Certain control information is carried by the ICMP protocol [26]. One may wish to require or restrict the use of such higher-level protocols, which can be done based on information in the IP header.

Further, the TCP and UDP protocols incorporate the concept of a ‘‘port,’’ identifying an endpoint of a communication path, and these protocols support the concept of ‘‘well-known’’ ports. For example, a TCP remote terminal access connection will always be addressed to well-known port 23 at the server host. In some cases, it may be useful to require or restrict access to specific ports; the access-control mechanism in this case would have to examine the higher-level header, since the source and destination ports do not appear in the IP header.

2.1. Policy and Mechanism

There is a wide range of access control policies from which to choose. One goal of an access control mechanism should be to allow each organization to choose its own policy, and to change its policy (perhaps quite frequently). In other words, it pays to separate the *mechanism* for forwarding packets from the *policy* that decides what should be forwarded. Although the underlying concept has been known for a long time, the term *policy/mechanism separation* was invented by the designers of the Hydra system [17], who established it as a ‘‘basic design principle ... of a kernel composed (almost) entirely of mechanisms.’’ Policies were embodied in user-level processes, thus improving flexibility while keeping the kernel simpler and presumably more reliable.

The Unix kernel, on the other hand, contains most of the policy functions of the operating system. There are a few exceptions; for example, in 4.2BSD Unix, the disk quota mechanism in the kernel receives quota information from user-level processes, and the network routing table is maintained by a user-level process. Still, in the Unix model the kernel makes the decisions.

The system described in this paper adheres to the Hydra model: a simple, general mechanism inside the kernel ‘‘asks’’ a user-level process to pass judgement on every packet that is to be forwarded. The kernel makes no access control decisions; rather, it provides the packet header to the user process, which then tells the kernel whether or not to forward the packet. The kernel mechanism is simple (it took about one day to code and test) and robust (a failure of the policy module should not result in a system crash, or indeed in any consequential failure save a temporary suspension of packet forwarding). Further details on the kernel mechanism are given in section 3.

Once this kernel mechanism is in place, it is easy to experiment with policy modules implemented as normal Unix user processes. Section 4 describes an implementation based on a daemon process that checks each packet against the criteria specified in a configuration file. This program is able to filter based on arbitrary criteria, including transport-level header information.

An alternative design is sketched in section 5. In this design, a user-level process would implement a *visa* protocol [10]. In visa protocols, there is no configuration file at the gateway; rather, the source host is required to attach a cryptographically-secured mark to the datagram header, proving to the gateway that this datagram is authorized to be forwarded. Authorization is done by an ‘‘access control server’’ distinct from the gateway; thus, visa protocols employ an additional level of policy/mechanism separation.

One subtle (but explicit) aspect of the IP architecture is that gateways are stateless packet switches, not required to maintain any history of previous packets [4]. The policy modules described in this paper can accommodate a certain amount of gateway state (see section 4.3), but may not support a protocol requiring significant gateway state. This is because so little information is passed between the gateway function in the Unix kernel and the decision-making function in the user process; neither function has access to the history of the other. This is not a serious problem for the IP protocol.

2.2. Related Work

It has long been recognized that an organization may protect itself against unwanted network connections by blocking them in its gateways. One simple approach is to remove from a gateway’s routing tables routes to specific networks, thus making it impossible for a ‘‘local’’ host to send packets to them. (Most protocols require at least some bidirectional packet flow even for unidirectional data flow, so breaking the route in only one direction is usually sufficient.) This approach, of course, does not work when the point is to permit access to some local hosts but not others.

Most (perhaps all) commercially-available gateway systems now provide the ability to screen packets based not only on destinations, but on sources or source-destination pairs. For example, the Proteon p4200 gateway [27] allows the manager to specify access based on pairs of IP addresses; each address is combined with a specified mask before comparison, so that the pairs may refer to networks or subnets instead of specific hosts.

Gateway products from cisco Systems [2] support a more complicated screening scheme, allowing finer control over source or destination addresses. For example, one could deny access to all but one host on a particular network. The cisco gateways also allow discrimination based on IP protocol type, and TCP or UDP port numbers.

Unlike access controls based on gateway-resident configuration tables, *Visa* protocols [10] control the path between specific pairs of hosts. Moreover, visa protocols provide some protection against forged datagram headers, by proving that the source address of a packet is genuine. Visas thus protect against malefactors within the local organization, as well as those outside, and because policy decisions are made by a server distinct from the gateway, one can employ an arbitrarily complex policy without fear of overwhelming the gateway. In spite of the advantages of visa protocols, they require explicit support from host implementations and increase the per-packet effort at gateways, so they are only in experimental use [9].

The kernel-resident mechanism described in section 3 owes an intellectual debt to the ‘‘packet filter’’ mechanism [21] used to give user processes efficient access to arbitrary datagrams. In the packet filter, the kernel applies user-specified criteria to received packets before demultiplexing the packets to the appropriate process. This is the reverse of the situation described in this paper, where decisions are made in user processes and consumed by the kernel, but the principle of doing the more complex job at user level remains the same.

3. Kernel support

The mechanism described in this section is called the *gateway screen*. It has been experimentally implemented in the context of the Ultrix™ 3.0 operating system. Since the IP forwarding code in Ultrix is nearly identical to that in 4.3BSD Unix, this mechanism should port to nearly any 4.2BSD-derived kernel with only minor modifications. Porting it to unrelated Unix kernels, or to other operating systems, may require changes in various details. The presentation in this section assumes the use of a 4.2BSD-derived kernel.

3.1. Overview

When an IP packet is received by the kernel, it is first processed by the ***ip_intr()*** procedure, which determines if the packet is meant for ‘‘this host.’’ If not, then the packet is passed to the ***ip_forward()*** procedure, which determines whether and how to forward the packet to its ultimate destination. The ***ip_intr()*** procedure runs as an interrupt handler, not in the context of a specific process.

The gateway screen intercepts packets just before they are passed to ***ip_forward()***. These potentially-forwardable packets are placed on a queue, and any processes waiting for this queue are awakened. (A process waits for this event by executing a particular system call.) When any such process wakes up, it removes a pending packet from this queue. The kernel extracts the datagram header from the packet, wraps this with some control information, and passes the result out to the user-level process. The system call then completes, allowing the user process to run and decide if the packet in question should be forwarded. The user process, through a subsequent system call, informs the kernel of its decision, and the kernel either drops the packet, or passes it on to ***ip_forward()***.

The kernel effectively acts as the ‘‘client’’ of a ‘‘server process’’ implemented at user level. The user process, however, makes the calls into the kernel, not vice-versa. One may view this as a relationship structured entirely of ‘‘up-calls’’ [3], with no ‘‘down-calls’’ at all. This structure requires the solution to certain problems (starvation, matching of requests with responses) not present in a more conventional client-server interaction, but it makes use of the synchronization, protection, and control mechanisms already provided by the normal system call implementation.

We can now look at the implementation in greater detail.

3.2. Interrupt-level functions

Not much processing is required at interrupt level. The ***ip_intr()*** procedure is modified to pass packets to the ***gw_forwardscreen()*** procedure, instead of directly to ***ip_forward()***. The ***gw_forwardscreen()*** procedure allocates a small control block to contain information about the packet, records the address of the packet buffer (‘‘mbuf chain’’) in this control block, puts the control block on a queue of ‘‘pending’’ packets, and issues a ***wakeup()*** call. The control block includes fields for the packet arrival time and a unique transaction identifier, which are set in this procedure.

Since the kernel has little control over how fast the user-level process responds to requests, it would be unwise to allocate control blocks directly out of kernel dynamic memory, for fear of using it up. Instead, the gateway screen maintains a limited pool (currently 32 items) of preallocated control blocks; this not only avoids using up memory, but makes allocation much faster.

Since this limits the number of pending packets, we must have some policy to apply to packets arriving when our private free list is empty. Two policies suggest themselves: accept the new packet and drop an old one, or drop all new packets until some old ones have been processed. The former policy is more expensive to implement, and the latter policy conforms to the assumptions of various congestion-avoidance and congestion-control mechanisms[1]. Since dropping the incoming packet is both easier and ‘‘better,’’ that is what is done. (Even when a packet is dropped, a ***wakeup()*** is still done, in case there are server processes sleeping.)

3.3. Programming interface

Before we look at the implementation of the system call interface, it is helpful to examine the alternatives for communicating information across the user-kernel boundary. There are two ways to do this in Unix: either the information is moved as the parameter (by-value or by-reference) of a system call, or it is moved via the I/O mechanisms over a file descriptor (I/O is done with system calls, of course, but these calls leave little room for improvisation).

Note that for each packet, we need to communicate information out of the kernel, let the user process run, and then communicate the decision back into the kernel. We do not necessarily have to make two system calls per packet; the trick is to pass one decision into the kernel on the same system call that passes the following packet’s information out of the kernel. If we want to use this trick, then we cannot use the normal I/O mechanisms (e.g., ***read/write*** or ***send/recv***).

Perhaps the cleanest approach would be to invent a new system call with one ‘‘in’’ parameter and one ‘‘out’’ parameter. Adding a new system call to the Unix kernel, however, requires changing a number of files within the kernel, as well as the system call interface library, and this seemed too painful.

---

[1] Nagle implies as much with his statement that, when facing buffer exhaustion, a gateway should ‘‘drop the packet at the end of the longest queue, since it is the one that would be transmitted last.’’ [22] Jacobson’s work [12] implies that gateways should give preference to the earlier packets in a multiple-packet window, since they are more likely to be retransmitted immediately once congestion is detected.

The approach actually taken was to define a few new *ioctl* requests. Since an ioctl parameter can be ‘‘value-result’’ (both ‘‘in’’ and ‘‘out’’), we can use the ‘‘one system-call’’ trick, and since adding a new ioctl request requires no new code aside from where the request is dispatched, it is easy to make this change. The disadvantages are that an ioctl parameter can carry at most 127 bytes of data, and that the entire parameter is copied in both directions. The size limit is not actually a problem (the largest possible IP header is 60 bytes, and the largest reasonable TCP header is 24 bytes). The extra data copying is a slight disadvantage (compared to the ‘‘new system call’’ approach) but is less costly than having to do twice as many system calls.

The ***screen_data*** structure passed between kernel and user space for the SIOCSCREEN ioctl request contains a prefix of the packet (including at least the packet headers), a timestamp indicating when the packet was received, and a transaction identifier to be used in matching requests with responses. This is necessary because there is no ‘‘connection’’ (such as a file descriptor) established between the kernel and the server process, so there is no other way for the kernel to associate the decision passed in on one system call with the information passed out on a previous one. Figure 1 shows the layout of the ***screen_data*** structure.

```
/*
 * Some fields of this struct are "OUT" fields (kernel write,
 * user read), and some are "IN" (user write, kernel read).
 */

struct screen_data_hdr {
    short sdh_count;    /* length of entire record */   /* OUT */
    short sdh_dlen;     /* bytes of packet header */    /* OUT */
    u_long sdh_xid;     /* transaction ID */            /* OUT */
    struct timeval
        sdh_arrival;    /* time this pkt arrived */     /* OUT */
    short sdh_family;   /* address family */            /* OUT */
    int sdh_action;     /* disposition for this pkt */  /* IN */
                        /*      see defs below      */
};

/* Possible dispositions of the packet */
#define SCREEN_ACCEPT   0x0001  /* Accept this packet */
#define SCREEN_DROP     0x0000  /* Don't accept this packet */
#define SCREEN_NOTIFY   0x0002  /* Notify the sender of failure */
#define SCREEN_NONOTIFY 0x0000  /* Don't notify the sender */

/* Screening information + the actual packet  */

#define SCREEN_MAXLEN 120       /* length of struct screen_data */
#define SCREEN_DLEN (SCREEN_MAXLEN - sizeof(struct screen_data_hdr))

struct screen_data {
    struct screen_data_hdr sd_hdr;                      /* IN/OUT */
    char sd_data[SCREEN_DLEN];  /* pkt headers */       /* OUT */
};
```

Figure 1: Layout of the ***screen_data*** structure

The lack of a connection creates some complexity in the kernel implementation, but it avoids the greater complexity of creating and managing connections, and in particular avoids the need to garbage-collect connections belonging to dead processes.

The structure passed between kernel and user space also contains one field that is set by the user process, to indicate whether or not the packet should be forwarded, and if the packet is rejected, whether or not to notify the source host. (The ICMP protocol provides a means for a gateway to notify a host that a packet has been dropped; unfortunately, there is no ‘‘Access Violation’’ message, so we must make do with an approximation such as ‘‘Host Unreachable.’’[2])

In addition to SIOCSCREEN, new ioctls are defined to turn screening on or off (SIOCSCREENON) and to get statistics information (SIOCSCREENSTATS). Only the SIOCSCREENSTATS request is available to unprivileged processes; processes must be running as the super-user to execute the other requests.

Figure 2 shows the complete source of an simple daemon program that ‘‘decides’’ to reject all packets.

3.4. Process-context functions

When the user process issues the SIOCSCREEN request (over a file descriptor returned by the ***socket()*** system call), control in the kernel passes to the ***ifioctl***() procedure. This is the only place besides ***ip_intr()*** that must be modified to support the gateway screen. This procedure simply transfers control to a procedure called ***screen_control()*** if any of the gateway screen ioctl requests are made.

When viewed as a complete system call, the SIOCSCREEN request starts by copying a decision into the kernel, sleeps waiting for a new packet, and then copies new information out of the kernel. Since there is no connection between what happens before and after the sleep, it is easier to describe a complete cycle starting with the sleep rather than starting with the system call.

Each packet in the pending queue is either ‘‘claimed’’ or ‘‘unclaimed.’’ A claimed packet is marked with the process ID of a process that has been given this packet to act on. When the process awakens from its sleep, it checks the queue of pending packets to see if there are any that have not yet been claimed. The pending-packet queue is managed first-in, first-out to avoid unfairly delaying any packet. If no unclaimed packets exist, the process returns to sleep. Otherwise, the packet is marked with the current process ID. A prefix of the packet, and other information from the control block, is copied into a ***screen_data*** structure, and the ioctl request completes in the normal way; since the kernel ‘‘knows’’ that the ioctl parameter is value-result, the ***screen_data*** structure is copied out to the user process.

---

[2] The ‘‘Blacker’’ system [6], which provides military-style security by interposing cryptographic hardware between secure hosts and an insecure backbone network, does define an appropriate ICMP code, but no commonly-used host software recognizes it.

```
#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <net/ip_screen.h>

main()
{
    int s;
    struct screen_data scdata;

    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }

    scdata.sdh_xid = 0; /* start with garbage transaction id */
    while (1) {
        if (ioctl(s, SIOCSCREEN, (caddr_t)&scdata) < 0) {
            perror("ioctl (SIOCSCREEN)");
            exit(1);
        }
        printf("dropping pkt, transaction %d\n", scdata.sdh_xid);
        scdata.sdh_action = SCREEN_DROP;
    }
}
```

Figure 2: Example program that rejects all packets

Once the user process has made its decision, it sets the appropriate bits in the ***screen_data*** structure and reissues the SIOCSCREEN request. Again, the kernel knows to copy the ***screen_data*** structure into the kernel, and control is passed to the ***screen_control()*** procedure. At this point, the kernel must match the decision in this SIOSCREEN request to one of the packets on the pending-packet queue.

Since the ***screen_data*** structure contains the unique transaction identifier stored in the packet control block, the kernel simply searches the pending queue for a packet with the right transaction identifier. If none are found, then the user process has made a mistake (or is making its first system call) and no further action is taken. If, during the search, packets are found that are claimed by the current process, but that do not have the right transaction identifier, then the user process has failed to follow first-in, first-out order; these packets are removed from the queue and simply dropped.

Assuming that a matching packet is found, if the decision encoded in the ***screen_data*** structure is positive, then the packet is passed to ***ip_forward()***, as if it had come directly from ***ip_intr()***[3]. The ***ip_forward()*** procedure may of course decide not to forward the packet (for example, because no route exists) but this decision is made independent of access control.

---

[3] There is a subtle difference: normally, when ***ip_intr()*** calls ***ip_forward()*** it does so at an elevated interrupt priority level (IPL), but the gateway screen calls ***ip_forward()*** at low IPL. This does not seem to be a problem, but one must be careful to ensure that ***ip_forward()*** does not assume it is called at any particular IPL.

If the user process instead decided against forwarding the packet, it is simply dropped. If the user process requested that the sender be notified, the ***icmp_error()*** procedure is called with the appropriate arguments, causing an ICMP Host Unreachable message to be sent.

At this point, the cycle is complete. The packet control block is put back on the private free list, and the user process is put back to sleep (after checking the pending-packet queue to make sure that no additional packets are waiting).

Earlier it was pointed out that there are a limited number of packet control blocks. Since it is possible for a user process to claim a packet and then die without providing a decision, the pending-packet queue could fill up with junk. Therefore, if when the pending-packet queue is searched and no unclaimed packets are found, and if the private free list is empty, we make the assumption that something is wrong. All packets older than a threshold age (for example, 5 seconds) are simply removed from the queue and dropped. If their ‘‘owning’’ processes are actually still alive and subsequently do render a decision, this causes no further problems. Thus, if a large number of processes do fail, the worst outcome is that the gateway stops forwarding packets for several seconds (provided that additional screening processes exist or are restarted).

3.5. Protocol-independence

None of the code within the kernel implementation of the screening module makes any assumption about the content of the packets being handled (including the layout of the packet headers). The only protocol-dependent actions required are the calls to either forward a packet or to send an error notification. These procedures are called indirectly, through pointers stored in the control block that were provided by the protocol-specific code that called ***gw_forwardscreen()***. To add screening for a new protocol family, one need only supply protocol-specific forwarding and error functions, and insert a call to ***gw_forwardscreen()*** in the appropriate place.

Since it is necessary for the user process to know which kind of packet it is processing, the protocol-specific module also provides a type code (the ‘‘address family’’: AF_INET for IP packets) that is stored in the protocol control block and passed to the user process (see figure 1). A process can ‘‘request’’ to receive packets of only one family by setting the `sdh_family` field when it passes a ***screen_data*** structure to the kernel.

3.6. Performance

In order to get an idea of the performance of the kernel portion of the implementation, we can look at the limiting case of a minimal user-level daemon: for example, the program in figure 2, changed to accept all packets without examining them.

The most interesting characterization of performance is the increment in delay over an unmodified Unix-based gateway. (It is much easier to measure round-trip delays rather than one-way delays; we must assume that the underlying one-way delay is about half of the total.) This increment is easily measured using the ICMP Echo protocol, which is especially convenient because all the processing in the ‘‘echo server’’ host is done in the Ultrix kernel, reducing the variance in delay.

The performance in this case depends mostly on the cost of transferring data and control between kernel and user contexts; that is, system call overhead dominates the cost of code within the gateway screen implementation. The entire pending-packet queue is searched once per system call, so to some extent the length of that queue affects performance. Thus, since the cost of that search is linear in the queue length, two measurements suffice to define the performance of the kernel implementation: the incremental delay when the pending-packet queue is empty, and the incremental delay when that queue is artificially kept nearly full.

Table 1 shows the measured round-trip and calculated one-way delays for several gateway implementations: an unmodified Ultrix kernel, the gateway screen with an empty queue, and the gateway screen with artificial garbage entries in the queue. The experimental setup consisted of a gateway, based on a MicroVax™ 3500 (about 2.7 times as fast as a Vax™-11/780), connecting two Ethernets [8], with an echo client host on one Ethernet and an echo server host on the other Ethernet. Packets contained 56 bytes of data, in addition to 42 bytes of Ethernet, IP, and ICMP headers. The measurements reflect average delays over a large number of trials.

Time in milliseconds			
Version	Round trip	One way	Added delay
No screen	8	4	
Empty queue	10	5	1
Full queue	14	7	3

Table 1: Performance with minimal user-level daemon

One measurement was also made with the client and server on the same Ethernet, with no intervening gateway; this gave a round-trip time of 3 milliseconds, or a one-way time of 1.5 milliseconds.

The ‘‘Added delay’’ column in table 1 shows the increment in one-way delay over an unmodified Ultrix-based gateway. For this hardware, the gateway screen delays each packet by about 1 millisecond, which is consistent with the cost of doing a system call.

The ‘‘Full queue’’ case in the table reflects a situation where the length of the pending-packet queue is artificially maintained at 500 packets. The additional delay imposed by this queue was about 2 milliseconds, or about 4 microseconds per entry. Normally, this queue is limited to 32 packets, but at that length the effect (estimated to be 160 microseconds per packet) is too small to be measured. Note that as long as the packet rate remains below overload (about 200 packets/second) the queue will remain nearly empty.

One other measure of an implementation is its size. Aside from a few lines of code added for linkage from other modules, the gateway screen implementation consists of 436 lines of heavily commented code (and 140 lines of header file). Compiled for the Vax, this results in 1512 bytes of object code, and less than 1 Kbyte of data storage is required at run time.

3.7. Further work

Although the ioctl-based programming interface makes it quite easy to integrate the gateway screen into the Unix kernel, it is not as efficient as one might like. The screening function could be embodied in a new system call that copied the packet header information only in one direction.

It is also possible to reduce the system call count even further, by batching several packets and decisions together for one system call. Batching has been shown to be profitable in a similar application [21]. When the load is low, the pending-packet queue will seldom hold more than one packet, and the batch size will be 1, but at high loads, several packets may arrive before the user-level daemon process can be scheduled (packet interrupts having higher priority than user processes). Thus, as the system approaches overload, batch size increases, and the system call overhead per packet decreases; this is precisely the behavior one wants.

```
gateway_screen(packet_data, packet_count, decision, decision_count)
struct screen_data *packet_data;        /* pointer to buffer */
int *packet_count;      /* result returned by reference */
struct screen_data_hdr *decision;       /* pointer to buffer */
int decision_count;
```

Figure 3: Proposed new system call

Figure 3 shows how the programming interface might appear for a new system call supporting batched operation. The ***packet_data*** and ***decision*** parameters are vectors of one or more ***screen_data*** and ***screen_data_hdr*** structures, respectively, with their lengths specified by the ***packet_count*** and ***decision_count*** arguments, respectively.

Since for most access control policies, the forwarding decision for a packet is independent of any previously received packets, the gateway screen is a natural application for parallel processing. On a multiprocessor, one could run several copies of the user-level daemon process, each on its own processor. The current kernel implementation, since it uses raised interrupt priority level for synchronization, would have to be modified for use in a symmetric multiprocessing kernel; explicit locks would be needed for the pending-packet queue and the private free list. Contention should be relatively low, especially if the items on the pending-packet queue are individually locked when being manipulated, since locked items can simply be ignored when searching that list.

If the kernel kept a cache of recent decisions, as is done in the user-level program described in section 4.2, it could potentially avoid most of the transfers to user-level code, and so significantly improve performance. The trick is to choose a cache-match function; an incoming packet will almost never match a previously received packet in its entirety, so only a few selected fields can be used in the matching function. Not only would the choice of these fields be protocol-specific (and would probably involve several layers of protocol header), but it might also be policy-specific; the user-level policy process would want to specify the matching function. It does not appear feasible to implement a general-purpose mechanism in the kernel, although it might pay to provide a caching function for a few heavily used protocols, such as TCP.

4. A table-driven policy daemon

This section describes the design and implementation of ***screend***, a table-driven policy daemon to make datagram access control decisions, to be used with the kernel mechanism described in section 3. This program uses only the most vanilla features of Unix (besides the gateway screen mechanism, of course) and so should be portable to any Unix-like system.

4.1. User interface

To use ***screend***, one starts by generating a configuration file. This is a text file that describes the kinds of packets that should be accepted or rejected. The daemon program is then started, parses the configuration file, and then enters an infinite loop making packet-forwarding decisions according to the criteria in the configuration-file. If one wishes to change the criteria, one simply edits the configuration file and restarts the daemon process. (In principle, the daemon could notice that the file has been changed, but this might add unnecessary code to the performance-critical inner loop).

The complete grammar for the configuration file is rather involved, and is given in appendix I. To understand this section, one must understand some of the concepts that may be expressed by this grammar.

The main purpose of the configuration file is to specify the action (accept or reject) taken when a particular kind of packet arrives. Packets are identified by their IP source and destination address, the next level protocols they use (for example, TCP or UDP), and the source and destination ports, or ICMP type codes, if these apply[4].

A packet can thus be precisely specified by listing its source and destination addresses, its protocol type, and if applicable, its source and destination ports or ICMP type code. One can also leave any of these fields unspecified, or partially specified. For example, one may specify that packets from a particular host to any host at all, using any protocol at all, should be be rejected; this is one way to isolate that host from the internetwork. One may also partially specify an address by specifying not a host address but a network number, or perhaps a subnetwork number. Finally, one can specify that certain fields should not match for the entire specification to match. Most fields can be specified either numerically or symbolically.

Specifications are evaluated in the order that they appear in the configuration file. Thus, specific exceptions to a more general rule should appear earlier in the file. Also, note that to specify both directions on a path, one needs two rules. The actions taken, in addition to accepting or rejecting a packet, can include notifying the sender upon rejection, and logging the packet (useful for detecting breakins). Figure 4 shows some examples (these examples would not make sense for any single gateway, since the host names are chosen at random).

---

[4] ICMP type codes are interesting because some ICMP messages can be harmful (for example, routing Redirect packets) while others are harmless (for example, Echo packets). Well, mostly harmless: a villain could use ‘‘harmless’’ packets to flood a victim’s host.

```
from host xx.lcs.mit.edu tcp port 3
        to host score.stanford.edu tcp port telnet reject;

between host sri-nic.arpa and any accept;

from net milnet to subnet 36.48.0.0 proto vmtp reject;

from net-not cmu-net tcp port reserved
        to net cmu-net tcp port rlogin reject notify;

from any icmp type echo to any accept log;
```

Figure 4: Example configuration file

There are two other kinds of rules that can be in the configuration file. First, one can specify a default action (normally rejection). Second, one can specify the network masks for arbitrary networks. This allows subnet specifications for non-local networks. Normally, a gateway is not supposed to know about the subnet structure of a distant network [20], but this information might be useful in deciding whether to forward a packet that originates someplace in an organization with multiple network numbers.

4.2. Implementation

Most of the complexity in the implementation ***screend*** is in the code that parses the configuration file and builds the internal data structures. The parser itself is a straightforward application of ***lex*** [16] and ***yacc*** [13]. All translations of symbolic values (such as host names) to numeric values are done during parsing; this is necessary for performance, although it means that if a host changes its address, the internal databases may reflect stale specifications.

The main loop of the daemon accepts a packet header from the kernel, extracts certain fields from the header, and matches the extracted fields against the internal representation of the configuration file. Since matching is the most time-consuming part of the inner loop, the choice of internal representation clearly affects performance.

Several different representations were examined, including hash tables, decision trees, and various combinations. The most difficult problem is that while the incoming packets contain specific addresses, the specifications in the configuration file may contain partially specified addresses, and several specifications may match various fields of the packet. Moreover, it is important to provide a deterministic way of selecting between possibly several specifications that all match the packet (the ‘‘in order of appearance rule’’ is arbitrary but easy to comprehend). None of the complicated data structures seemed to have much of an advantage, and all involve complicated implementations.

A simple array of specifications, searched linearly, is easy to implement and provides a deterministic evaluation order. On the other hand, if the configuration file contains many rules, this array may be quite long, and since the individual match evaluations are rather costly, the performance of this approach could be quite bad.

The solution is to combine an array representation of the entire database with a cache recording recent decisions. Since the decisions recorded in the cache are with respect to specific packet headers (actually, only the important fields of these headers), rather than the partial specifications of the configuration file, matching is quite efficient. Since most delay-sensitive applications tend to involve repeated packet exchanges, a small cache with least-recently-used (LRU) replacement should yield high hit rates. (Applications that are sensitive to connection setup time, or that exchange packets infrequently, may suffer low cache hit rates; the delays, however, should be an order of magnitude lower than the propagation delay in a cross-country network. If necessary, specifications pertaining to such delay-sensitive applications can be listed early in the configuration file, to shorten the database search.)

The match function for cache lookups is quite simple. Once the relevant fields of the packet (source and destination addresses, protocol type, and source or destination ports or ICMP type if applicable) are extracted, they are placed into a compact structure that can be compared word-by-word to a cache of previous such records. The cache also includes the decision made for the first occurrence of this pattern. The entries in the cache can never become invalid, since the configuration file does not allow specifications that depend upon any time-varying quantity.

If the cache does not hold a matching record, the entire database must be searched. In this case, the matching function is more complex; it must take into account partial specifications. In particular, each IP address appearing in the packet may have to be converted to a subnet number or network number before comparing with a specification. Since these conversion operations are moderately expensive, they are postponed until necessary, and then done only once per packet.
A s bFe0nuib$r tc oAk uP dhd u‘net m‘s An a has‘ 0a l% u lP

Table 2 shows the measured round-trip and calculated one-way delays for the best case (100% cache hit rate) and for 0% cache hit rates with moderate-sized (10 entry) and large (100 entry) databases. The table also includes some measurements from table 1, for comparison; the experimental setup used identical hardware and methodology.

Time in milliseconds			
Version	Round trip	One way	Added delay
No screen	8	4	
Minimal daemon	10	5	1
screend 100% cache hits	10	5	1
screend 0% cache hits 10 entries	11	5.5	1.5
screend 0% cache hits 100 entries	17	8.5	4.5

Table 2: Performance of ***screend*** system

The results for the case of 100% cache hits demonstrate that ***screend*** is quite efficient when its cache is properly sized. Even when the cache is too small, if the number of distinct rules in the configuration file is small, the additional delay (around 0.5 millisecond per packet) is nearly insignificant. The results for the larger configuration database show that it is important to choose both a simple set of rules, and an efficient evaluation order, to avoid noticeable performance degradation. Even in this case, the excess delay is comparable to other delays in a large internetwork.

Varying the packet size, while changing the total round-trip delays, has no significant effect on the incremental delay imposed by the ***screend*** program. This is as expected, since the kernel provides only a prefix of each packet to the ***screend*** program.

The ***screend*** program sources consist of about 2600 lines of commented code. The resulting binary file, compiled for the Vax, is about 48 Kbytes of object code (including library functions), and when running requires on the order of 150 Kbytes of memory, depending on the size of the configuration file.

Actual experience with ***screend*** has been quite successful. The system is reliable, easy to configure, and performs well. The ability to log improper packets has made it possible to discover previously obscured problems, including software bugs and ‘‘back door’’ routes that might be security holes.

4.5. Problems and further work

The access control model followed by ***screend*** assumes that application-specific controls (as opposed to host-specific controls) can be based on the TCP or UDP port number of at least one of the end points. This is valid for many TCP and UDP applications, such as remote terminal access or name service, but is not true in all cases. For example, the Berkeley ‘‘talk’’ daemon, used for online conversations between users on different hosts, listens for connection requests on a well-known port number, but builds the actual connection using arbitrary port numbers. The ‘‘talk’’ daemon could be modified to coexist with ***screend***, but this is not always convenient.

Another example where this assumption breaks down, although not directly relevant to the IP-based ***screend***, is found in the ‘‘socket’’ mechanism in the Pup byte-stream protocol (BSP) [1]. Although the initial BSP connection packet is directed at a well-known socket number, the server listening on that socket immediately creates a new socket for the actual connection. Thus, subsequent packets do not travel to a well-known socket number.

Although the well-known port mechanism is followed by existing TCP and UDP applications, it has certain disadvantages related to number management. As a response, it has been proposed that this mechanism be replaced by a ‘‘port service multiplexer,’’ a mechanism using port names instead of port numbers [18]. This would make the port-matching support of ***screend*** nearly useless. The Sun RPC system, used with NFS, includes a ‘‘port mapper’’ mechanism that follows the same model [29]. Fortunately, because the RPC packets follow a fixed format, one might extend ***screend*** to parse the RPC headers and filter on the RPC program number.

IP multicasting is another issue not addressed by ***screend***. A standard for IP multicast addressing has only recently been developed [5], and as yet there is little support for or experience with multicasting. Identifying multicast IP addresses is quite simple, so modifying ***screend*** to support the concept of multicast groups should not be difficult.

One attractive extension to ***screend*** would be to use it for datagram accounting. Since the marginal cost of a datagram on a LAN is quite small, datagrams flowing within an organization might be paid for through pooled overhead charges, but some long-distance carriers charge per datagram and an organization might want to charge these costs back to specific sources. The ***screend*** program could easily be extended to record accounting information, for all packets or for those following specific paths. Access control might be used not for security but to restrict long-distance access to those hosts that have promised to pay.

One user has asked that ***screend*** be enhanced to log the beginning and end of TCP connections (that is, logging packets with the SYN and FIN flags set). This is made difficult by the cache-based implementation of ***screend***, because these flags are not part of the packet addressing information. It should be possible to modify the logging mechanism to parse these flags in those packets for which SYN/FIN logging is requested.

Remote network management has recently been an area of active development. Although ***screend*** is table-driven, it is not difficult to imagine a version that is managed via a protocol such as CMIP [11].

Finally, the current implementation of ***screend*** does not parse IP header options, which is necessary to defend against use of source routing to avoid simplistic address checks. It would not be difficult to add option parsing, without changing the basic design of the program. Currently, ***screend*** simply rejects any packets with IP header options.

5. Implementing Visa mechanisms

The implementation described in section 4 uses a configuration file that is stored at the gateway. A *visa* protocol [10] is a different kind of mechanism for implementing datagram access controls. This section sketches the design of visa-protocol support based on the gateway screen mechanism of section 3. This design has not been implemented; an entirely kernel-resident implementation of the visa protocol already exists [9].

5.1. Overview of visa protocols

In a visa protocol, each inter-organization datagram carries an unforgeable mark that proves to a gateway that transmission of the datagram is properly authorized. These marks are called *visas*, by analogy to the stamp made on a passport that allows a bearer to cross a border; visas are unforgeable because they are created using cryptographic mechanisms and secret keys.

In the visa protocols, a gateway has no policy function at all; it simply checks the visa on a packet for validity. Policy decisions are made by a separate ‘‘access control server’’ (ACS), which holds a secret key in common with the gateways of its organization. Thus, the principle of policy/mechanism separation is extended even further.

In use, a source host first applies to an ACS for permission to send a packet to a given destination host. The ACS makes a decision based on an arbitrary policy, and it may decide not to issue a visa. If it does issue a visa, there are several alternative protocols that can be followed.

In the first, or ‘‘stateful gateway,’’ policy, the ACS creates a new *visa key* to be used for this connection, and transmits the key to both the source host and the appropriate gateways. The source host then creates a new visa for each packet by computing a function based on the packet contents and the key, and attaches the visa to the packet before transmitting it to the gateway for forwarding. The gateway, which has stored its copy of the key in a table, checks to make sure that the visa attached to the packet was computed using the same key.

In the second, or ‘‘stateless gateway,’’ policy, the ACS accepts some information from the source host, including a secret session key chosen by the source host. The ACS creates a digitally-signed [7] and encrypted copy of this information, which is returned to the source host. The source host then attaches this visa to each packet, along with a ‘‘signature’’ value computed by a function based on the packet contents and the session key, then transmits the packet to a gateway. The gateway decrypts the visa (since it shares the secret encrypt encryption key with the ACS) and extracts the session key; this allows the gateway to validate the visa.

5.2. Implementing visa protocols

The main loop of a visa-protocol daemon is similar to that of ***screend***, except that instead of matching the packet header against the configuration database, the daemon must validate the visa contained in the packet header. The algorithms to do this validation are described in detail in [9], and can be treated as ‘‘black boxes’’ for the purposes of this paper.

The main difference between the ***screend*** program and a visa protocol daemon is that ***screend*** is entirely synchronous, whereas a visa daemon must process events coming from several sources. The main source of events is, of course, packet information arriving via the SIOCSCREEN ioctl, but a visa daemon must also receive and process messages from an ACS. In the stateful-gateway visa protocol, these messages contain the visa keys, and arrive fairly often. In the stateless-gateway protocol, the messages are used to distribute the secret key shared between the ACS and the gateways; this happens infrequently, when it is determined that the key may have been compromised and a new one is needed.

4.2BSD Unix has several mechanisms for supporting asynchronous event streams. The most elegant is the ***select()*** system call, which allows a process to wait for an event on any of several I/O streams (file descriptors). The SIOCSCREEN ioctl, however, does not fit this model, since an ‘‘event’’ (the arrival of a new packet) occurs half-way through the request.

The other mechanism is the use of signals (asynchronous software interrupts). A process can request that it receive a signal whenever a message arrives over a specified I/O stream. Since communication with the ACS can be done over such a stream, and because a signal interrupts the execution of the SIOSCREEN ioctl at a ‘‘safe’’ point, the daemon process can accept messages from the ACS while waiting for incoming packets.

It is possible to implement an entirely synchronous visa-protocol daemon by checking for ACS messages whenever the SIOCSCREEN ioctl completes, but before processing the packet. This may lead to extra work in the inner loop, and delays in packet processing, but it is simple to implement. To avoid long delays in ACS-to-gateway transactions, which might cause timeouts, the daemon should use an intermediary process to handle ACS communications.

6. Other Applications

The gateway screen mechanism may be used for other purposes besides access control. For example, the ***screend*** program could be used solely to log certain kinds of events, such as connections to a service for which clients are charged, or apparently misrouted datagrams.

A similar program might be used to collect statistical information about the flows through a gateway. For example, one might want to know the rate of traffic flowing along a particular set of routes, to support capacity planning. Or, one might want to collect packet traces for use in simulating routing cache policies or packet-header compression algorithms. The gateway screen provides a hook by which a variety of statistics and trace-gathering programs may get access to the complete stream of packet headers.

7. Summary and Conclusions

Gateway-based datagram access controls provide a powerful tool for protecting an organization attached to an internetwork. Ideally, such controls should be incorporated in a dedicated gateway system, but Unix-based gateways are feasible and often fill the need. The kernel-resident gateway screen mechanism described in section 3 provides a simple, flexible, and fairly efficient means of adding access controls to a Unix-based gateway.

The ***screend*** program described in section 4 implements a flexible, powerful access control policy, without significant performance degradation. Other access control policies, such as the visa mechanism discussed in section 5, should be equally easy to implement in a similar way. Although the performance of a Unix-based gateway may not be equal to a that of a dedicated system, the use of a general purpose operating system together with the gateway screen mechanism makes it easy to experiment with a wide range of access control policies.

8. Acknowledgements

Deborah Estrin got me interested in the issues of datagram access controls; Richard Johnsson, Brian Reid, and Paul Vixie got me interested in devising a specific solution. I also thank Greg Satz and John Shriver for providing information on the capabilities of existing gateway products, Chris Kent for proofreading, and the USENIX referees for pointing out several shortcomings of the original draft.

9. References

[1] David R. Boggs, John F. Shoch, Edward A. Taft, and Robert M. Metcalfe. Pup: An Internetwork Architecture. *IEEE Transactions on Communications* COM-28(4):612-624, April, 1980.

[2] *Gateway System Manual.* cisco Systems, Inc., Menlo Park, CA, 1988.

[3] David D. Clark. The Structuring of Systems Using Upcalls. In *Proc. 10th Symposium on Operating Systems Principles*, pages 171-180. Orcas Island, WA, December, 1985.

[4] David D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 106-114. Stanford, CA, August, 1988.

[5] Stephen E. Deering. *Host Extensions for IP Multicasting*. RFC 1054, Network Information Center, SRI International, May, 1988.

[6] —. Blacker Front End Interface Control Document. *DDN Protocol Handbook, Volume 1.* DDN Network Information Center, SRI International, Menlo Park, CA, 1985.

[7] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory* IT-22(11):644-654, November, 1976.

[8] *The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications (Version 2.0).* Digital Equipment Corporation, Intel, Xerox, 1982.

[9] Deborah Estrin, Jeffrey C. Mogul, and Gene Tsudik. Visa Protocols for Controlling Inter-Organization Datagram Flow. *IEEE Journal on Selected Areas in Communication* , 1989. In press (Special Issue on Secure Communications).

[10] Deborah Estrin and Gene Tsudik. Visa Scheme for Inter-Organization Network Security. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 174-183. IEEE, April, 1987.

[11] —. Common Management Information Protocol (CMIP). ISO 9595/2.

[12] Van Jacobson. Congestion Avoidance and Control. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, pages 314-329. Stanford, CA, August, 1988.

[13] S. C. Johnson. *Yacc -- Yet Another Compiler-Compiler*. Technical Report 32, Bell Laboratories, Murray Hill, New Jersey, July, 1975.

[14] Christopher A. Kent and Jeffrey C. Mogul. Fragmentation Considered Harmful. In *Proc. SIGCOMM '87 Workshop on Frontiers in Computer Communications Technology*, pages 390-401. Stowe, VT, August, 1987.

[15] B. Leiner, J. Postel, R. Cole, and D. Mills. The DARPA Internet protocol suite. In *Proc. INFOCOMM 85*. Washington, DC, March, 1985. Also in *IEEE Communications Magazine*, March, 1985.

[16] M. E. Lesk. *Lex -- A Lexical Analyzer Generator*. Technical Report 39, Bell Laboratories, Murray Hill, New Jersey, October, 1975.

[17] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proc. 5th Symposium on Operating Systems Principles*, pages 132-140. ACM, November, 1975.

[18] Mark Lottor. *TCP Port Service Multiplexer (TCPMUX)*. RFC 1078, Network Information Center, SRI International, November, 1988.

[19] Jeffrey C. Mogul, Christopher A. Kent, Craig Partridge, and Keith McCloghrie. *IP MTU Discovery Options*. RFC 1063, Network Information Center, SRI International, July, 1988.

[20] Jeffrey Mogul and Jon Postel. *Internet Standard Subnetting Procedure*. RFC 950, Network Information Center, SRI International, August, 1985.

[21] Jeffrey C. Mogul, Richard F. Rashid, Michael J. Accetta. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proc. 11th Symposium on Operating Systems Principles*, pages 39-51. Austin, Texas, November, 1987.

[22] John Nagle. On Packet Switches With Infinite Storage. *IEEE Transactions on Communications* COM-35(4):435-438, April, 1987.

[23] Jon Postel. *User Datagram Protocol*. RFC 768, Network Information Center, SRI International, August, 1980.

[24] Jon Postel. *Internet Protocol*. RFC 791, Network Information Center, SRI International, September, 1981.

[25] Jon Postel. *Transmission Control Protocol*. RFC 793, Network Information Center, SRI International, September, 1981.

[26] Jon Postel. *Internet Control Message Protocol*. RFC 792, Network Information Center, SRI International, September, 1981.

[27] Proteon, Inc. *ProNET p4200 Gateway Software User's Guide.* Westboro, MA, 1988.

[28] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conference*, pages 119-130. Portland, OR, June, 1985.

[29] *Network Programming.* Sun Microsystems, Inc., Mountain View, CA, 1988.

[30] Hubert Zimmermann. OSI Reference Model -- The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications* COM-28:425-432, April, 1980.

Unix is a registered trademark of AT&T.
Ultrix, MicroVax, and Vax are trademarks of Digital Equipment Corporation.

Appendix I. Grammar for the screend configuration file

This is an informal guide to the grammar of the ***screend*** configuration file. It is meant for readers who are familiar with the basic concepts of the IP protocol family.

I.1. Lexical structure

- **Comments** can either be ‘‘C-style’’ comments, delimited by ‘‘`/*`’’ and ‘‘`*/`’’, or ‘‘csh-style’’ comments begun with ‘‘#’’ and terminated by the end of a line. Comments do not nest.
- **Case** is significant in reserved words (all are lower-case). This is actually a benefit, because if a host name happens to conflict with a reserved word, you can use the host name in upper-case.
- **Host names** begin with alphabetics but may contain digits, ‘-’, ‘.’, and ‘_’. The same is true of network, subnet, and netmask names. All can also be entered in dotted quad notation (for example, ‘‘10.1.0.11’’).
- **Numbers** may be in decimal or in hex (0x0 notation). Octal notation is not allowed because nobody uses it in this context. (Actually, hex is almost as useless).
- **Protocol names** and **port names** (for TCP or UDP) are as in `/etc/protocols` and `/etc/services`, respectively. These can also be given as numbers (host byte order).
- **ICMP type codes** must be chosen from this list, or given as numbers:

```
echoreply
unreachable
sourcequench
redirect
echo
timeexceeded
parameterproblem
timestamp
timestampreply
informationrequest
informationrreply
addressmaskrequest
addressmaskreply
```

- **All white space** is the same (including newlines).

I.2. Syntax

General syntax rules:

1. The configuration file consists of ‘‘specifications’’ terminated by semicolons.
2. There are three kinds of specifications:
    a. **default action specification**: There should only be one of these (the last one is the one that counts); it specifies what action to take if no action specification matches a packet.

    b. **subnet mask specifications**: specifies the subnet mask used for a given network.

    c. **action specifications**: specifies a class of packets and the action to take when such a packet is received.

3. Specifications can appear in any order, but the evaluation order of action specifications is the order in which they appear in the file.

In BNF, this is:

<*configuration-file*> ::= { <*specification*> | <*configuration-file*> <*specification*> }
<*specification*> ::= { <*default-action*> | <*subnet-spec*> | <*action-spec*> }

The syntax for a default action specification is:

<*default-action*> ::= **default** {**accept** | **reject**} [**notify**] [**log**] **;**

Note that ‘‘default accept notify;’’ is legal but the ‘‘notify’’ in this case is a no-op. If not specified, the default action is ‘‘reject’’.

The syntax for subnet mask specifications is:

<*subnet-spec*> ::= **for** <*network*> **netmask is** <*maskval*> **;**

The <*network*> is either a network name or a dotted-quad address, such as ‘‘36.0.0.0’’. ‘‘36’’ is *not* a reasonable value. <*Maskval*> is either a name (treated as a hostname) or a dotted-quad address, such as ‘‘255.255.255.0’’ (bits are *on* for the network and subnet fields.)

The syntax for action specifications is:

<*action-spec*> ::= **from** <*object*> **to** <*object*> {**accept** | **reject**} [**notify**] [**log**] **;**

Such a specification says that packets flowing this way between this pair of ‘‘objects’’ (defined below) should either be accepted or rejected. If ‘‘notify’’ is specified, when a packet is rejected an ICMP error message is returned to the source. If ‘‘log’’ is specified, this packet and its disposition are logged.

Conceptually, for each packet the action specifications are searched in the order they appear in the configuration file, until one matches. The specified action is then performed. If no specification matches, the default action is performed.

To simplify the configuration file, the syntax

<*action-spec*> ::= **between** <*object*> **and** <*object*> {**accept** | **reject**} [**notify**] [**log**] **;**

may be used to indicate that the same action should be performed on packets flowing in either direction between the specified pair of ‘‘objects.’’ Note that this is simply syntactic sugar; it has the same effect as specifying the two unidirectional rules, with the ‘‘forward’’ direction listed first.

An ‘‘object’’ is a specification of the source or destination of a packet. The syntax for object specifications is somewhat complex, since certain fields are optional:

<*object*> ::= { <*address-spec*> | <*port-spec*> | <*address-spec*> <*port-spec*> }

If the <*address-spec*> is not given, ‘‘any host’’ is assumed. If the <*port-spec*> is not given, ‘‘any protocol and port’’ is assumed.

<*address-spec*> ::= { <*net-spec*> | <*subnet-spec*> | <*host-spec*> | **any** }

<*net-spec*> ::= { **net** <*name-or-addr*> | **net-not** <*name-or-addr*> }
<*subnet-spec*> ::= { **subnet** <*name-or-addr*> | **subnet-not** <*name-or-addr*> }

<*host-spec*> ::= { **host** <*name-or-addr*> | **host-not** <*name-or-addr*> }

The ‘‘-not’’ convention means that the object specification matches if the specified field does *not* have the specified value. For example, ‘‘`from host-not sri-nic.arpa to host any reject`’’ means that packets *not* from `sri-nic.arpa` are dropped. The ‘‘subnet’’ and ‘‘subnet-not’’ forms match against the entire address under the subnet mask (for example, if the netmask for net 36 is 255.255.0.0, then ‘‘`subnet 36.8.0.0`’’ matches a packet address of 36.8.0.1).

<*name-or-addr*> ::= { <*name*> | <*dotted-quad*> | **any** }

<*port-spec*> ::= { **proto** <*proto-name-or-number*>
| **icmp type** <*type-name-or-number*> | **icmp type-not** <*type-name-or-number*>
| **tcp port** <*port-name-or-number*> | **tcp port-not** <*port-name-or-number*>
| **udp port** <*port-name-or-number*> | **udp port-not** <*port-name-or-number*> }

<*proto-name-or-number*> ::= { <*name*> | <*number*> }
<*type-name-or-number*> ::= { <*name*> | <*number*> | **any** | **infotype** }
<*port-name-or-number*> ::= { <*name*> | <*number*> | **any** | **reserved** }

‘‘Reserved’’ ports are those reserved by 4.2BSD Unix for privileged processes. ‘‘Infotype’’ ICMP packets are those that are purely ‘‘informational’’: echo, timestamp, information, and addressmask requests, and the corresponding replies.

Table of Contents

List of Figures

List of Tables