awk-like construct for matching the first token of the line (indeed, of any and all lines of text this packet might complete or contain) against the regular expression /^RETR$/, which matches exactly the word ‘RETR.’

This approach seems to be fairly powerful. Ideally, we would like something that feels a lot like being able to apply awk scripts to arbitrary TCP streams flowing through the router.

8.2 RPC

Another common deficiency we’d like to remedy is the ability to filter RPC traffic, which passes between more or less randomly assigned ports. It is a relatively simple matter to periodically make requests to relevant portmappers to find port numbers, and thus keep a relatively current mapping of services to ports, and thereby be able to do filtering on a service basis. A more elegant, but trickier, solution, is to also glean port to service mappings from portmapper requests being made through the router.

9. Availability

Various versions of the packet filtering capabilities described herein are available on NSC bridge/router products, contact the author for further contact information.

10. References

[Ra] Ranum, Marcus J., “Thinking About Firewalls,” Proceedings of the Second International Conference on Systems and Network Security and Management (SANS-II), Apr. 1993.

[Cha] D. Brent Chapman, “Network (In)Security Through IP Packet Filtering,” Proceedings of the 3rd USENIX Security Symposium, Sept. 1992.

[Che] William Cheswick, “The Design of a Secure Internet Gateway,” Proceedings of the 3rd USENIX Security Symposium, Sept. 1992.

[Ci] Cisco Systems, Router Products Configuration and Reference, Software Release 9.1, Sept. 1992.

[3c] 3Com Corporation, NETBuilder® Family Bridge/Router Reference Guide, Software Version 7.0, Feb. 1992.

Finally, a short note on latency. The additional time through the router is negligible, 10s to 100s of microseconds. Again, certain applications will notice this, so there are efforts to improve this, but most applications will not notice even the slowest implementation.

8. Future directions

There is no shortage of things that would be useful, many of them are minor additions to the current suite of pattern elements available, or perhaps a new action or two. While this sort of thing is certainly part of future development, it's not architecturally interesting. Thus, the brief discussion below will focus on things which not only seem likely to be useful, but which are interesting extensions to the architecture.

8.1 Non-stateless filtering

The biggest problem with packet filtering as an access policy paradigm is the stateless nature of it. The current implementations of the packet filtering architecture under consideration remain stateless, that is, only information about the packet currently in flight can be used to determine whether or not to forward it, and to decide what actions to take. Unfortunately, applications are not in general stateless, and often we'd like to be able to gather and examine state at a level closer to that of the application.

The first significant item on the development program is, therefore, the ability to preserve and filter on TCP state. This will call for the addition of some additional patterns and actions:

- an action, perhaps **allocate_state(<type>)**, to allocate resources for maintaining state. The type field indicates the record type of the data stream. For example, many TCP protocols are essentially streams of newline terminated records whitespace separated fields. Thus, setting up to watch a new FTP connection might look like this:

```
tcp_connect_request
                    tcp_destination_port in (21)
                    allocate_state(nl_whitespace);
```

- a corresponding action to tear down existing state, freeing the resources. This must be used in conjunction with a garbage collector to delete state for connections which have silently vanished.
- a pattern to determine if state has been allocated for the connection the current packet is a part of (the connection determined by the 5-tuple: IP protocol, source host, source port, destination host, destination port).
- a pattern to determine if the current packet completes at least one record within the allocated state.
- some additional syntax for examining fields within completed records.
- perhaps some additional actions to force shutdowns of connections, perhaps an action to set RESET bits in TCP packets, for example.

A filter to disallow FTP GETs might look like this, then:

```
filter no_ftp_get
        not tcp_destination_port in (21) break;
        tcp_connect_request
                allocate_state(nl_whitespace);
        not state_allocated tcp_reset fail;
        not record_complete break;
        $1 ~ /^RETR$/ tcp_reset free_state
                counter_1;
        any fail;
end
```

The first line of the filter checks to see if this is a packet on an FTP control connection, if it's not, we exit this filter and carry on. The next line checks to see if this is a TCP connection request, that is, if someone is initiating an FTP connection. If so, we attempt to allocate state, and fall through to the next line of the filter. This next line checks to see, for connect requests and for every other packet heading for port 21 through this filter, if state has been successfully allocated. If it has not, the connection is reset, and the packet is dropped. Thus, if we cannot monitor the connection, we do not allow it. If the packet has made it this far, we check to see if it completes a record. In this case, whether the data portion carries a newline character. If not, we break out of the filter. If it does contain a newline, then we fall through again. At this point, things become slightly murky, as new syntax is introduced. Tentatively, something awk-like seems to be suitable for this sort of record stream, so the example uses an

the filter definition language acquires new capabilities, and as the tool itself is improved to give more readable output.

In addition to this assurance capability, there are active auditing tools designed to read and log the packets generated by the logging facilities of the filtering language. In brief, a filter may generate a UDP datagram addressed to a certain host, containing the packet being processed through the filter. The logging tool allows these packets to be captured and logged. At this point any sort of log reduction tools can be applied, for example we have seen variations on commercial tools which can detect the signature of the more common network probing tools. Of course, a set of awk scripts may be more appropriate to a site's needs, and are certainly cheaper.

We feel that these sorts of tools are the most important tools in access policy management. It's all very well to install boxes and software that come with glossy brochures explaining how this will make your networks secure, but if your only hard evidence of security is that / hasn't disappeared yet, you shouldn't be sleeping very well at night.

7. Performance

In general, turning on filtering capability in a router will degrade performance, depending on the architecture of the router, it will degrade performance more or less. On faster routers, which do more of the packet forwarding in hardware, forcing a lot of software for filtering to execute against each packet will degrade performance by a higher percentage than on a slower router. Nonetheless, it is the nature of the beast that packet filtering is done effectively in-kernel, there are no context switches. Further, any reasonable router will be architected to make it relatively easy for its software to examine packet headers. It is fairly safe to assume that any router-based packet filter will perform substantially better than a host-based access policy facility running on hardware of similar capability.

As for hard numbers, there really are a lot of variables, so it really is hard to pin them down. Since routers are packet-by-packet devices, we'll talk about packet rates, in terms of how large the packets need to be to be filtered at the full speed of the media[1]. This seems to be more useful a metric than pure packets per second, since it gives an indication of the sort of traffic that can be filtered fast enough to give full use of the media. That is to say, performance of the router is not an issue unless it is too slow to forward all the packets the media it's attached to can carry, and the latter quantity depends on the sizes of packets being moved. In general, we assume that for the lion's share of packets, the result of the filtering will be to pass the packet, and to take no substantial action. Thus, the majority of the work the packet filtering need do will be pattern patching. We consider a handful of cases:

- A Null access policy, which really just measures the overhead of turning on packet filtering without applying any filtering rules, seems to drop performance by somewhere between 20 and 50 percent, depending on the platform. This seems to be a substantial bite, but the fact is that modern routers are very fast. In the least heavily powered implementation (one 25Mhz CPU for 6 ethernet ports) it will still filter at wire rate for packets a few hundred bytes long.
- A simple access policy, in which 3 or 4 pattern elements have to be executed against each packet, will cost in the region of 30 percent of the remaining performance, that is, will cost 30 to 60 percent of the unfiltered performance. This will give you wire rates in the aforementioned slowest known implementation for packets at a slightly unrealistic 1 kilobyte.
- A fairly stringent access policy, in which 10 pattern elements are executed against a typical packet, will wind up costing roughly 40 to 80 percent of the unfiltered performance. This will not filter at wire rate, in the aforementioned most underpowered implementation.

Despite these apparent performance problems, performance is in fact not really that much of an issue. A production ethernet running close to capacity is not a happy ethernet. Furthermore, by no means all the packets on an ethernet segment will pass from one ethernet to another. The only really common applications which will tend to notice the performance hit of packet filtering are those running on high performance hosts, streaming data to another high-performance host. These applications will tend to be generating packets of a size close to the media's maximum transmission unit, bringing the packet rate down. In short, the throughput is at worst good enough. Of course, we're always seeking to improve it, there's no need to stop merely because the vast majority of applications will not notice the difference.

1. It should be noted that performance numbers herein, such as they are, were measured using a prototype implementation, both to show worst case numbers, and to avoid putting numbers which really belong to marketers in this paper.

6.1 On-router auditing

The on-router facilities consist of some detailed statistics gathering capabilities, and an action which generates a console alarm. Our statistics gathering infrastructure recognizes a simple counter object, which has 4 integers in it:

- a packet count
- a byte count
- a first user counter
- a second user counter

Every counter in the system is associated with a source/destination address pair, or a simply a source address, or simply a destination address, depending on how it was instantiated. Statistics actions to increment 'statistics' will, if necessary, instantiate a counter associated with a source/destination address pair. Statistics actions to increment 'sa_statistics' or 'da_statistics' will instantiate counters associated with a source or a destination address, respectively. In short, using a statistics action on a packet will cause the appropriate counter given the action and the addresses in the packet to be instantiated, if it has not already been so.

Let us imagine that we have a subnet we're trying to keep particularly safe, and we'd like to audit anything we don't approve of. We could attach a filter to the outgoing side of the interface attached to that subnet, and construct this filter with a filter element for each type of legitimate traffic with a **break** disposition. Thus, any packet which arrived at the bottom of the filter would be bad. We use something like the filter in shown in Figure 6.

Note that this filter is using the counters indexed by source address, so we don't know what the destination addresses were, but we do know the sources. Since the filter is applied on the outgoing side of the interface attached to the subnet we're watching, the source address will be the apparent address of the host generating the errant packet(s), presumably the host doing whatever probe was being done. Is we used simply **statistics**, **counter_1**, and **counter_2**, we would have a larger statistics table indexed by both the probing host, and the probee.

In addition, packet and byte counts are maintained on a per-filter, per disposition basis, so it is possible for example to find out how many packets were **fail**ed by a particular filter, and how many bytes were in them. In cases where the network administrator is forced to 'poke holes' in the filtering rules, the administrator can carefully design the hole to be implemented by a filter dedicated to that task, and can then periodically check counters associated with the filter. If the counts are persistently zero, the administrator could either close the hole, or follow up with the original requestor to see if it can be closed. We know of at least one site which automatically ages such filtering rules out if they go unused.

6.2 Host-supported auditing

The support for really industrial strength auditing and assurance is substantially better than the on-router facilities, not surprisingly. The effort required to manage an access policy audited at this level is also substantially greater, but it's a distinct step up from slapping some filters on your routers, and waiting for / to disappear on your file server.

In this direction are a several tools hosted on Unix workstations. To provide some degree of assurance, we have a tool to parse an existing set of filter definitions and the details of where each filter is applied, and to generate as output a set of english statements that describe the access policy implemented by those filters. Clearly, this is a tool which can never be complete, as

```
filter audit_one
        # < rules for what we explicitly allow >
        ..
        ip_protocol in (6) sa_counter_1;      # Count TCP packets in the 1st user counter
        ip_protocol in (17) sa_counter_2;     # Count UDP packets in the 2nd user counter
        any sa_statistics;                    # Track byte and packet counts for everything
        fail;                                 # drop everything that gets this far
end
```

Figure 6

```
filter otter
        ip_source_address in (1.2.3.4, 1.2.3.5, 2.0.0.0 .. 2.255.255.255) break;
                # We trust 2 hosts on net 1, and all of net 2
        ip_source_address in (3.0.0.0 .. 3.255.255.255) tcp_destination_port in (23) break;
                # We trust net 3 for telnet access
        tcp_connect_request ip_dest_address in (5.1.1.1) tcp_destination_port in (25)
                counter_1 break;
                # Connections to our mail host are ok, we count the connection
                # requests so we know how much mail we're receiving.
        # etc etc.
        any statistics fail;  # If we don't allow it, fail here and count it
end
```

Figure 4

A more interesting example appears in Figure 5, a filter which sanitizes the IP options present in a packet, by discarding out of hand anything with a source route in it, and by stripping any remaining options out. This is a general utility filter that might be called by other filters which decide that, while a packet may be ok, we don't trust it enough to let it possibly play games with IP options. In particular, the action **strip_any_options** appears here to strip out all the IP options present in the packet, illustrating the kind of thing one might want to do which is neither outright allowing the packet through, nor dropping it.

5.5 Infrastructure

Filters are named objects, and filter points contain the name of the filter to apply, a reference to the filter, not the filter itself. This has a number of useful implications. Filters can be re-used, by applying them at multiple filter points. For example, it is possible to write two simple filters, 'ok' which immediately passes any packet, and 'notok' which immediately drops any packet, let's say, and then build a simple access policy by specifying a large apply table attaching the filters 'ok' and 'notok' to appropriate source/destination pairs in the table. Filters with no protocol dependent components can usefully be applied in many places. For example, if policy says that a certain subnet shall be completely isolated except during working hours, it would be easy to write a filter named 'workinghours' which passed all packets between 9 and 5, Monday through Friday, and dropped all packets at other time. Applying this filter to the outgoing and incoming interface filter points for the appropriate interface on the router, for all protocols, would accomplish this. Filters can be redefined at run time. When a filter, let's say 'otterpaws,' is recompiled while the router is running, the name 'otterpaws' now refers to the new filter definition, so all filter points containing the name 'otterpaws' will immediately begin to use the new filter. This makes it reasonable to update the access policy on hardware that is in service (though of course one must be careful!)

6. Closing the loop

There are a couple of levels upon which filtering can be audited. The simple approach is to use the on-router facilities, and the more industrial strength approach uses the logging actions, together with host based tools for capturing logged packets and analyzing these captured packet traces.

```
filter victor
        ip_option_present 137 fail;          # Dump anything with a strict source route
        ip_option_present 131 fail;          # and anything with a loose source route
        any strip_any_options continue;      # Axe any options leftover and carry on
end
```

Figure 5

5. The filtering language

A filter consists of several parts. First, it has a name, by which it can be referenced in filter points, and by other filters. Then, it has a body, consisting of any number of so-called *filter elements*. Finally, it ends with the keyword ‘end.’ A filter element is more or less a single statement in the program the filter represents, and has itself three parts. A filter elements begins with 0 or more *pattern elements*, continues with 0 or more *actions*, and concludes with a *disposition*. Optionally, a filter element may consist of a call to another named filter.

5.1 Pattern Elements

The language has a selection of pattern elements to use. There are, of course, the obvious ones such as checking source and destination host addresses and ports for inclusion in some range, or more generally, some set. More interesting are predicates such as ‘is this a TCP connection request’ (or response), ‘does the hardware source address from which we received this packet match the hardware address to which we would send a packet going the other way?’, and various numbers that can be checked for inclusion in arbitrary sets, such as ‘what time of day is it?’ ‘what day of the week is it?’, ‘is the 39th byte after the end of the IP header in such-and-such a set?’, ‘what is the next hop we’re going to send this packet to?’ and so on.

The goal is to provide as rich a set of pattern elements as possible, to provide as much flexibility in the access policy as possible. In general, any reasonable question one can ask about the packet in hand can be asked. The set of pattern elements can be expected to become richer over time, since adding new functionality at this level is relatively simple.

5.2 Actions

If a packet matches a set of pattern elements, actions may be taken. These actions do not include the obvious ‘drop it’ or ‘forward it’, since this resides in the disposition. Actions are used to react to the fact of the packet, to carry out auditing functions, to gather statistics, to modify the packet or the route it will take, and so forth. For example, actions exist of the form ‘generate a console alarm at priority <n>’ and ‘wrap this packet up in a UDP datagram and send to host <H>’, ‘re-write the source IP address to <I>’. It is in the actions that any access policy work beyond the simple ‘drop it’ and ‘forward it’ is done. There may be as many actions as the user likes in a filter element.

Again, the underlying architecture makes it easy to add other actions not terribly related to access policy issues, such as IP options processing, and modifying IP source and destination addresses to implement a simple network address translator.

5.3 The disposition

Any filter element which matches will, after having the actions applied, will proceed as indicated by the disposition. The disposition may be one of the following keywords:

- **succeed** which will cause all filter processing to stop immediately, and the packet to be forwarded.
- **fail** which will cause all filter processing to stop immediately, and the packet to be dropped.
- **continue**, the default, will cause filter processing to continue at the next filter element in the current filter, if any, or the first filter element of the next filter to be executed, if any, or the packet to be forwarded, if there is no further processing to be done.
- **break** which will cause processing of the current filter to cease, and filter processing to carry on with the next filter to be applied, if any, or the packet to be forwarded if there is no additional processing.

5.4 Examples

Here we give a few simple examples of what filters look like, to give a feel for the facility, and to illustrate the various kinds of logic one can apply.

The filter which appears in Figure 4 applies certain tests to detect packets we like, and drops all the rest. Note that since it uses the **break** disposition, further filter processing is done. Thus, this filter might be applied on the incoming side of the interface from the global internet, and implement that component of an access policy. Subsequent filters, say outgoing interface filters, may implement other policy, such as time-of-day based access to certain services

```
set udp 17;              # Define IP protocol number for UDP
set tcp 6;               # Define IP protocol number for TCP
set smtp 25;             # Define TCP port number for SMTP


filter example
        tcp_destination_port in (smtp) succeed;
        ip_protocol in (udp) succeed;
        ip_source_address in (1.2.3.4) ip_protocol in (tcp) succeed;
        fail;
end
```

Figure 3

The filter depicted in Figure 3 could be applied to all packets with any source address and destination address matching 10.0.0.0 masked with 255.0.0.0 (see the apply table described below).

4. Architecture

At the heart of the packet filtering strategy is a language for defining rules to apply to individual packets. Since the architecture is tied to routers/bridges, and these are by their nature packet oriented, the language semantics are all tied to the packet-by-packet nature of the underlying system. This is admittedly less useful that, say, a stream oriented system, but this deficiency is being addressed in current development, to be discussed below. By using a description language, and compiling it into byte code for execution against packets, we avoid artificial limits on ruleset size, as we are bounded only by available memory. A compiled language also gives us a platform upon which to build additional functionality with relative ease, as has been done for several years now, and as continues to be done. This language provides a large suite of pattern matching elements, to classify packets into the various lines of ones access policy, and also provides a useful suite of actions, to implement the appropriate reaction as specified by the access policy. The language allows for the definition of arbitrarily many named objects called ‘filters’ herein, each filter containing as many collections of patterns and actions as you like. A filter is the low-level model of a group of the textual lines from the imaginary access policy described above.

A *filter* in this architecture is a named block of byte code, a program which may be executed against a packet in flight through the router. Much of the power of this architecture derives from the facilities for selecting which filters to apply to which packets, since executing the byte code for an entire access policy against every packet would probably not be very efficient. Thus, there are *filter points*, which are logical points in the path of the packet as it flows through the router, where filters are attached (by name). Filter points are defined on a per-network-protocol basis (that is, IP has a set of filter points, IPX has another set, and so on), though filters themselves are not protocol dependent. The filter points are as follows:

- Media interfaces have both and incoming and an outgoing filter point, for each protocol
- Each protocol has a so-called first and a last filter point. A packet flowing through the box has the first filter executed against it, if any, then the incoming filter for the interface upon which is arrived, then the outgoing filter for the interface it’s going to be sent out, and finally the last filter for the protocol.
- In the middle, between the interface filters, there is an apply table, which is a table of host pairs (possibly wildcarded), which may define another filter to apply to the packet based on source and destination host or network. This closely resembles the access lists described in Implementation 1, above. However, rather than allowing some rules to be attached to a line of the access list, the apply table allows a named filter, of whatever complexity the user chooses to write, to be attached to each line.

This plethora of filter points allows filters themselves to be generally small and to the point, so they can be executed reasonably quickly. The apply table in the middle is typically where the bulk of the access policy is implemented, if it is complex. Incoming and outgoing interface filters allow special rules to be applied for networks directly attached to the router.

3.1 Implementation 1

A popular packet filtering facility in use today allows one to create lists of (possibly wildcarded) host pairs, and allows some simple access rules to be applied on a per-line basis. Lists can then be attached to interfaces, and packets leaving on an interface have the access policy described in the list applied. See Figure 1 below, this access list permits any TCP connection from anywhere to the class A network 10.0 if the destination port is 25, it permits any UDP datagram from anywhere to net 10, it permits any TCP connection from host 1.2.3.4 to network 10, and it denies everything else headed in to net 10.

This has the advantage of simplicity, producing a set of access lists from an access policy is relatively easy if the access policy happens to resemble an access list. Since most of an access policy tends to, this is useful. It is, therefore, quite usable but lacks in flexibility. If the user wants to do anything other than allow/disallow packets based on the somewhat limited criteria allowed by the definition of a row in an access list, that is just too bad. There is also a paucity of auditability, and no real scope for integrating with a smarter host to handle things which the access lists cannot.

3.2 Implementation 2

Another facility allows one to create a list of, again possibly wildcarded host pairs, and to attach to each pair a simple forward or discard disposition, and optionally to attach a more complex filter to each such line of access policy. These optional filters consist of a list of offsets into the packet, starting from a header specified in the access policy line (for example, the TCP header), and operations to apply to the value extracted from that offset. This is very flexible, since in principle you can extract anything you like from the packet, and filter based on that. It does require an unfortunate familiarity with the details and layout of packet headers.

See Figure 2 below, this implements the same policy as the previous one, use the ability to extract arbitrary data from the packet in the 2nd line to define a special filter to extract the destination port number and to compare it the hexadecimal number 19, which is decimal 25, or the SMTP port.

This implementation of packet filtering combines the simplicity of the previous one with a flexible and powerful ability to filter based on virtually anything in the packet.

3.3 Our Implementation

For reference, we include our implementation of the little access policy used in the above examples.

```
access-list 101 permit tcp 0.0.0.0 255.255.255.255 10.0.0.0 0.255.255.255 eq 25
access-list 101 permit udp 0.0.0.0 255.255.255.255 10.0.0.0 0.255.255.255
access-list 101 permit tcp 1.2.3.4 0.0.0.0 10.0.0.0 0.255.255.255
access-list deny 0.0.0.0 255.255.255.255 10.0.0.0 0.255.255.255 ip
```

Figure 1

```
add -ip FilterAddrs 0.0.0.0/0.0.0.0 > 10.0.0.0/255.0.0.0 forward tcp 1
add !1 -ip Filters %2:%19
add -ip FilterAddrs 0.0.0.0/0.0.0.0 > 10.0.0.0/255.0.0.0 forward udp
add -ip FilterAddrs 1.2.3.4/255.255.255.255 > 10.0.0.0/255.0.0.0 forward tcp
add -ip FilterAddrs 0.0.0.0/0.0.0.0 > 10.0.0.0/255.0.0.0 discard
```

Figure 2

Given that policy is to be implemented at a level one step up from individual hosts, it is clear that the lion's share of the policy implementation must reside in the routers and bridges connecting the network segments together, and that is indeed the focus of this paper. If policy is to be implemented on a segment-by-segment fashion, performance issues will arise, especially in this era of ever increasing media speeds. Any architecture for implementing policy must be able to process traffic with high throughput and low latency, as close to the native speed of the router or bridge as possible, in fact. Since access policies are potentially complex, an architecture for implementing it must not have arbitrary limits on numbers of rules that can be applied, it is not acceptable to implement half of ones policy, after all. Access policies are also subject to change, so the architecture must be easy to update, without service disruption. An access policy is not much good without some sort of auditing mechanism, to determine whether or not the policy is being implemented as the implementors hope, so a really useful architecture should include mechanisms for audit, closing the loop as it were. Lastly, it should be relatively easy to translate an human readable, natural language, description of the access policy into the machine implemented form.

That is, what we want from facility for implementing an access policy is:

- Speed, it should be fast enough to do some checking on every packet passing between access policy domains. Speed requirements are therefore highly dependent on configuration, but certainly being able to run at ethernet speeds with moderately large packets, say a few thousand packets per second, would be a desirable minimum.
- Flexible, it should allow the implementation of arbitrarily complex access policy rules. Any reasonable criterion for access should be checkable. There should be no built in limit on the number of rules to apply to each packet.
- There should be audit mechanisms. At a minimum, some rudimentary counters which can be somehow used to track access policy violations and usage, and as usual, the more the merrier. Ideally, the ability to count anything based on any criterion, and the ability to log anything based on any criterion.
- It should be easy to use. Any mechanism which is too difficult to use will lead to, at best, poorly maintained access policies, and at worst, no access policy at all.

In addition, from the point of view of the implementor, we'd like to see:

- Easily extensible, so new functionality can be easily added as new needs are seen in the field.

In this paper we describe a system which we feel meets these criteria, and outline future directions which will help it to meet them better. We hope the system will continue to meet them as 'fast' comes to mean faster, and as new protocols and associated threats arise.

In general, an access policy is a list of pairs of hosts, and the protocols allowed to flow between each pair. This is of course simplistic, perhaps this definition should be extended to allow one to describe a whole group of hosts as one of the endpoints. Perhaps we'd also like to audit certain attempts to use an unauthorized protocol between certain host pairs, or audit successful uses of certain permitted protocols. Perhaps we'd like to modify the data stream for certain protocols flows. It should be clear, after a little thought, that the notion of access policy is not one we can define completely. Thus, tools should not tie themselves to anything more specific than a general paradigm. Herein, we imagine an access policy as a collection of lines of text, each line describing a certain class of data (for example, TCP packets from host A to host B) and a reaction to take to such data (e.g. drop it and log an access violation to host D). The idea here is that the description of the data and the reaction to it should be as open-ended as possible, a perfect architecture for implementing access policy would permit anything whatsoever to appear in these imaginary lines of text.

The goal of our architecture is to provide as flexible a set of tools as possible, to this end. Since access policies tend to classify data according to what host or group of hosts the data originates from, and what host or group thereof it is flowing too, our architecture has certain optimizations to make it easier to describe such rules, and to execute such rules rapidly.

3. Background

Herein, we look at some implementations of packet filtering, and try to point out where their weaknesses are as a general access policy facility. These are not necessarily current implementations, since packet filtering technology is evolving, but should give some flavor of what's in use today. Performance is in general adequate, so we will focus on our other criteria.

An Architecture for Advanced Packet Filtering

Andrew Molitor

Network Systems Corporation
amolitor@network.com
7600 Boone Ave.
Brooklyn Park, MN, 55428

ABSTRACT

Packet filtering in routers has been underrated as anything but an adjunct to other network security measures. This paper presents an architecture, and an implementation of it, for packet filtering that addresses many of the perceived problems with packet filtering. Starting from a short discussion of what constitutes a network access policy, the paper makes a case for extremely flexible packet filtering as an integral part of an access policy. After briefly examining a couple of commonly used packet filtering implementations, the paper goes on to describe a more flexible architecture for packet filtering, and gives some examples of how the implementations of this architecture can be used. After a discussion of how the architecture and the implementations better support auditing and assurance procedures for a network access policy, the paper finishes with a description of some of the more architecturally interesting planned future development.

1. Introduction

In recent history, packet filtering has come to be seen as inadequate for 'real security' [Ra], which has led to the proliferation of non-router, non-packet-filter based approaches to managing network access policies [Ra, Che]. In truth, many implementations of packet filtering are limited, and it is certainly also true that not every aspect of every security policy can be handled by anything resembling that which we think of as packet filtering. See [Cha] for the definitive discussion. The goal of this paper is to outline an architecture which can provide high quality packet filtering (and more generally, access policy implementation and management), which can grow as the needs of access policies become more well defined, and which can integrate reasonably well with more traditional host based access policy mechanisms. We also take this opportunity to talk about access policy strategies, and how the architecture described herein works with our preferred viewpoint.

2. Access Policies

Network security means more than simply keeping the hackers out, it means controlling access between all the segments of your network appropriately. It seems certain that trying to secure ones computing systems at the host level is a lost cause, hosts are complex beasts, and often must run problematic applications in order to be useful. Too often, the tendency is to then give up on the entire internal network, and focus on building a hard shell around the soft chewy center [Che]. This approach neglects the middle ground between host-level and enterprise-level granularity, and this middle ground is both fertile and the subject of this paper. It is possible, and desirable, to implement a security or access policy at the network level, defining policy in terms of which subnets or segments can talk what protocols to what other segments. In this model, the Internet is just another (particularly untrustworthy) segment. As usual, it is possible to take points of view, 'that which is not expressly forbidden is permitted' and 'that which is not expressly permitted is forbidden.' If the global IP network is present as a segment, it seems to be agreed that the latter policy is probably the better one.

The following paper was originally published in the
Proceedings of the Fifth USENIX UNIX Security Symposium
Salt Lake City, Utah, June 1995.

An Architecture for Advanced Packet Filtering

Andrew Molitor
Network Systems Corporation
Broklyn Park, MN

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org