Proceedings of the 1st Conference on Network Administration

Santa Clara, California, USA, April 7-10, 1999

Just Type Make! Managing Internet Firewalls Using Make and Other Publicly Available Utilities

Sally Hambridge, Charles Smothers, Tod Oace, and Jeff Sedayao

Intel Corporation

Just Type Make! - Managing Internet Firewalls Using Make and other Publicly Available Utilities

Sally Hambridge
Intel Corporation
Charles Smothers
Intel Corporation
Tod Oace
Intel Corporation
Jeff Sedayao
Intel Corporation

Abstract

Managing Internet firewalls that can failover between each other is quite a challenge. When those firewalls are geographically dispersed and have a small number of people to be maintain them, it becomes even more challenging. Intel Corporation has a small staff that manages several geographically dispersed Internet firewalls with failover requirements. These firewalls use a standard screened subnet architecture [1] with packet filtering inner and outer firewall routers and a number of bastion hosts between them. These bastion hosts provide services with load balancing and disaster recovery for relaying SMTP mail, answering DNS queries, and proxying web requests. To manage this complex system of firewalls, Intel's Internet Connectivity Engineering staff have come up with a way to model all of the interrelated firewall as one distributed system. Host and router configurations are considered source to that system and compilation and installation of that source is driven by the Make [2] utility. Packet filtering Access Control Lists (ACLs) are built by a Makefile. The Makefile assembles the ACLs and executes an Expect [3] script that installs them. We configure bastion hosts by configuring Make to drive **rdist**, which run over the secure shell (SSH) [4]. In this way, only updated files are pushed out to the bastion hosts and passwords and other configuration information do not go in the clear. Our experiences with Make and these publicly available utilities are quite good - allowing us to manage a large distributed set of firewall devices. Using a Make driven approach requires much discipline, however, to avoid the distribution of bad configurations. Future plans include ACL optimization and sanity tests before and after bastion host configuration pushes.

1. Introduction

Managing a single Internet firewall complex can be difficult - there can be multiple routers with long packet filtering access control lists (ACLs) and bastion hosts performing different functions that all must work together seamlessly, efficiently, yet securely. Managing multiple Internet firewalls which interact with each other and provide failover capability between each other while maintaining both security and some comparable level of performance becomes even more challenging. The Internet Connectivity Engineering Group at Intel has created a way to manage multiple interacting Internet firewall complexes spread across the world using the familiar utility – Make utility. This paper describes how Intel integrated make with other publicly available tools to administer multiple firewall complexes across the world.

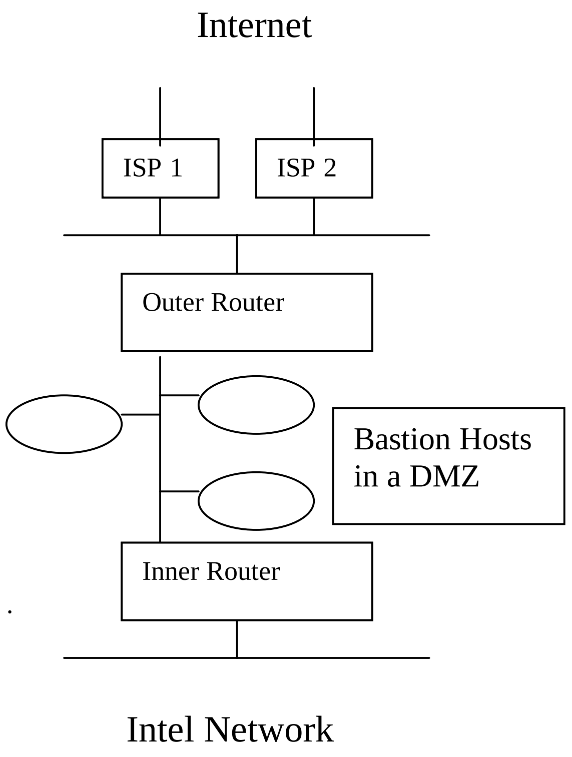
The first section of this paper talks about the firewall environment at Intel. It describes the key features of the Intel Internet firewall environment and the challenges that that led us to create our Make-based configuration tool. The second section covers how we overcame those challenges by implementing a Make driven configuration tool. The architecture of the Make tool and the publicly available utilities in our implementation are described here. The third section of the paper goes over our mostly positive experiences with our make driven update process, followed by a final section on future work that we are planning.

2. Intel's Internet Firewall Environment

It is impossible to understand how and why we created our Make driven update tools without understanding Intel's Internet firewall environment. In this section, we provide context for our Make tools by describing the Internet connectivity at Intel. We also talk about the key motivating factors that led us to create these tools.

The standard Intel Internet gateway uses the screened subnet architecture [1], as shown in Figure 1. Multiple bastion hosts lie between a external outer router and an internal inner ("choke") router. These bastion hosts provide a variety of functions, such as web service to the Internet, SMTP mail relaying, proxy services (web, telnet, FTP, AOL, etc.), and performance monitoring. Outside of the external router is a segment for ISP access. Multiple ISPs have a presence (routers) on this segment and exchange traffic with the bastion hosts through the external router. Both the external router and internal "choke" router do extensive packet filtering. The packet filters prevent generic access to the bastion hosts from the Internet. The packet filters allow access to the bastion servers from hosts within Intel, but do not allow generic access from the bastion hosts into Intel. This is our implementation of "defense in depth." A single compromise of a bastion host does not mean that there is complete entry into the heart of Intel

Figure 1: Intel’s Firewall Architecture



There are multiple Internet gateways at Intel, each with two or more ISPs, spread across the world. Our intention was to minimize the amount of traffic on Intel's internal network and get good performance for Internet applications. We also wanted to have multiple gateways for failover. If one Internet gateway failed, traffic should be able to flow in and out through another gateway.

Intel's firewall environment, as described above, presents a number of challenges to the staff maintaining and engineering the gateways. The first set of challenges involved the cisco routers. We had to find some way to maintain the Cisco Router packet filtering access control lists (ACLs) used on the inner and outer routers to permit failover. If one site's Internet connectivity went away, the packet filtering ACLs at other sites need to be able to allow traffic for that bastion hosts at the first sites to flow in and out. Even if there was failover, the router ACLs need to be consistent between sites. If a bastion host has a particular kind of access at its site, it should still have that access if traffic needs to fail over to another gateway. In addition, we have large numbers of cisco ACL entries. While these entries have to be consistent between gateways for failover, we have to arrange the entries to offer good performance. To make things even more challenging, if we had a problem with the ACLs we rolled out, we need some easy way to backup the changes.

The second set of challenges involves the bastion hosts. These hosts contain all kind of tables and configuration files like anti-spam lists, sendmail configuration files, and performance monitoring information. We need this information to be consistent between sites, in order to make maintenance easier and to have consistent access policies and DNS databases. It would not make any sense for one Internet gateway to have very strict anti-spam rules and another to have very loose policies. As with router ACLs, any changes in configurations need to be easy to back out, in case there are problems. Since the bastion hosts are on a segment exposed to the Internet, we are extremely concerned about problems with eavesdropping and spoofing. Any kind of updates to these hosts needs to verify the identity of originating hosts and be secure from eavesdropping and spoofing.

The final set of challenges involved the staff maintaining the Internet gateways. There are 7 Internet gateways spread around Intel but fewer people than that responsible for engineering and maintaining them. We needed an easy way to update many hosts and routers. Any personnel intensive update method would not be viable in this environment.

3. Solving our Maintenance Challenges using Make

The first step toward solving our maintenance challenges was coming up with a conceptual model for our Internet gateways. We knew that they were not stand-alone systems - instead they were interconnected and interrelated. Each gateway is an interconnected part of a single integrated system. Traffic from one gateway had to be able to travel through another. Mail servers must have consistent configurations to relay mail and block spam effectively. DNS servers must be consisent in order to provide DNS information correctly. Once we began thinking of the gateways as a single integrated system, we started looking at system and router configurations as source code with various compilation dependencies that is fed into that single yet distributed machine. With that model in mind, what better utility to manage source code compilation and installations than the classic utility Make!

Once we had a model for managing our configurations, we had to implement configuration management for the routers and for our bastion hosts. A key part was figuring out how to break down the configurations into units small enough to manipulate and build Makefile dependencies with them. The rest of this section describes how we split up configurations and used Make as a way of building configurations and installing them. The first part shows how router ACLs are managed with Make. The second part deals with managing server configurations. The final part describes how we manage our configuration files on the systems that build and install configurations.

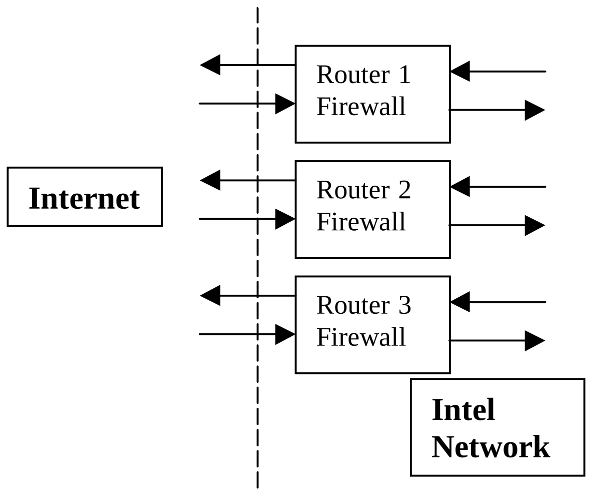
3.1 Managing Router ACLs using Make

Since a key requirement is that traffic from any gateway within Intel must be able to go into and out of any other gateway for failover, being able to manage the interchangability of router ACLs is critical. Imagine creating ACLs for a firewall complex. You have to deal with the services and well known ports for traffic coming in and for traffic going out. In addition, a gateway may have some special applications running in it to support external customers. Now imagine a second gateway with mostly the same and then merging the two sets of access lists. Doing this by hand for two firewall complexes is difficult. Doing this by hand for eight firewall complexes would be totally unmanageable.

We simplified this task by breaking down ACLs into files containing the permissions going out to each segment and coming in from each segment. By segment, we mean a subnet or network that forms a DMZ network between the inner and outer routers. The firewall architecture shown in Figure 1 shows a single segment with bastion hosts on it. In our architecture, the inner firewall router and outer firewall router each have one interface onto each segment.

We created a naming scheme based on the directionality and the router name. Thus *Intel-to-router1-seg1* contains router ACLs pertaining to traffic from Intel to segment 1 on router 1. *router1-seg1-to-Intel* contains router ACLs for segment 1 on router 1 to Intel. *router1-seg1-to-Net* contains ACLs for segment 1 on router 1 to the Internet while *Net-to-router1-seg1* contains ACLs for the Internet to segment 1 on router 1. For simplicity's sake, all files are named for the outer firewall router in each complex. As a way of illustrating how this can work, let's say that there are three geographically dispersed firewalls that need to failover between each other, as shown in Figure 2. The arrows depict traffic going in and out of each firewall to the Internet or to Intel's Internal network.

Figure 2: Three firewall complexes with failover



In this example, we have named each firewall by its outer router name. Each firewall could contain any number of segments (which are not explicitly shown here). If the Router 1 firewall should lose its connectivity to the Internet, then its traffic to and from the Internet should be able to travel through the Router 2 or Router 3 firewalls. If the Router 2 firewall lost its Internet connectivity, then Router 2 traffic should be able to failover through the Router 1 and 3 firewalls. The same should happen for Router 3.

For each segment, we group the ACLs together that apply to traffic going in a particular direction. For example, all of the ACL entries applying to traffic from the segment into Intel are grouped together. All of the ACL entries applying to traffic from Intel to the segment are put in another group. Each group is put into a separate file. For each firewall, we can concatenate all of the files together into one large file containing all the permissions for that firewall. We also concatenate the files from any other firewall complex needing failover capability through the first gateway. For our example, assuming that each firewall complex had only one segment, we would concatenate together the following files for the outer firewall routers: Net-to-router1-seg1, Net-to-router2-seg1, Net-to-router3-seg1, router-seg1-to-Net, router2-seg1-to-Net, and router3-seg1-to-Net. The inner router at router1's firewall complex would include all of these files but also router1-seg1-to-Intel and Intel-to-router1-seg1. The inner router at router2's firewall complex would have all of the outer firewall files but also router2-seg1-to-Intel and Intel-to-router2-seg1. The inner router at router3's firewall would use a similar configuration except using router3-seg1.

Routers check packet filtering ACLs one line at a time and stop checking when they find a match. Having the lines most matched at the top speeds the traffic through ACLs and through the router. Because of this, we want the large file of concatenated ACLs to be in a different order depending on the gateway. For the router1 firewall, we'd like the files for router1 to be first in the list. For the router2 firewall, the files for router2 should be at the top of the concatenated ACL file, and similarly for the router3 firewall. We've done this by defining sets of macros for each firewall gateway which contain the files going in or going out. INFROMNET_GW1 contains Net-to-router1-seg1, and if there is any other segment on router2 (such as seg2), Net-to-router2-seg2. OUTTONET_GW1 contains router1-seg1-to-Net and if there is any other segment on router2 (such as seg2), router2-seg2-to-Net.

We then define a macro in the Makefile containing the macros of all the files for all the routers needing failover through that firewall:

```
GW1_OUTER = ${OUTTONET_GW1} \
        ${INFROMNET_GW1} \
        ${OUTTONET_GW2} \
        ${INFROMNET_GW2} \
        ${OUTTONET_GW3} \
        ${INFROMNET_GW3}
GW2_OUTER = ${OUTFROMNET_GW2}\
        ${INFROMNET_GW2} \
        ${OUTTONET_GW1} \
        ${INFROMNET_GW1} \
        ${OUTTONET_GW3} \
        ${INFROMNET_GW3}
```

If there are special permissions in GW1 which don't need failover, it is easy enough to add the file name with those permissions to the gateway macro. Here we add a special set of permissions for a segment 0 on router1:

```
GW1_OUTER = Net-to-router1-seg0 \
        ${OUTTONET_GW1} \
        ${INFROMNET_GW1} \
        ${OUTTONET_GW2} \
        ${INFROMNET{GW2} \
        ${OUTTONET_GW3} \
        ${INFROMNET_GW3}
```

If there are files to be included in every gateway, it is easy enough to define a "generic" macro which contains those files:

```
GENERIC_IN = generic-in monitor-in
GENERIC_OUT = generic-out monitor-out
```

The actual concatenation is defined by the target *router-access*. That target takes as its dependencies the gateway macro and the generic ones. It cats them together, throws them through some processing the C preprocessor and some custom filters (we will discuss this in more detail in section 4) and concludes by appending "end" at the bottom of the concatenated files.

```
Router1-access: ${GW1_OUTER} ${INTELNETS}
        cat  ${GW1_OUTER} | ${CPP} \
              ${ROUTER1DEFS} | \
              intel_nets_convert ${INTELNETS} \
              > router-access
         echo ‘end’ >> router1-access
```

Finally, the target *router1-access-install* takes router-access as a dependency, but before it re-loads the ACLs, it also runs an Expect [3] script which pulls the current ACLs off the router and stores them with their associated counters. Cisco routers maintain a count of how many packets have matched each line of the ACL. The counters are reset each time new ACLs are loaded. In order to acquire data about the use and efficiency of our access-lists, we download the current set of ACLs before we upload the new set. Both of these operations are done via Expect scripts which log into the router interactively, then either write out the current set of ACLs to the server and load the ACLs from the server to the router.

```
router1-acl-stats:
        /usr/local/expect/get_acl_stats router1
router1-access-install: router1-access, router1-acl-stats
        cp router1-access /tftpboot
        /usr/local/expect/load_up router1
        rm /tftpboot/router1-access
```

Recording our ACL usage has been useful for simplifying ACL maintenance. With data on which statements are used and which are not, it is easier to identify usage which is no longer needed. It's amazing that users will gladly follow a process to allow them access but develop amnesia when they need to follow a process to end that same access. Intel employees have also requested access to external applications for all of Intel and then have not publicized the availability, so these are never used as well. We remove access when the lines show no activity for three months.

The flexibility of Make allows us to use many utilities and programs as part of our process. In the example just shown, we used shell commands, expect, and PERL (within the intel-nets-convert program which we will discuss later) in order to build and compile router ACLs.

3.2 Bastion Host Management using Make

Intel's Internet gateways contain bastion host services such as DNS, SMTP mail relaying, and Internet access proxying. Maintenance of these services shares many of the same requirements as Intel's firewall routers. The services are mission critical and call for a high level of care when changes are made. In addition they also back each other up during outages and provide load balancing, thus requiring consistent configuration among hosts. Make drives our bastion host configuration too, as we will describe in this section.

One of the core ideas we utilize at Intel is called "Copy Exactly!" Once a product or facility is designed for a specific purpose, it should be deployed identically everywhere. It should not be redesigned and rebuilt every time it is deployed. Adhering to this ideal helps our chip fabrication plants come up to speed rapidly and produce very high yields. In the Information Technology (IT) space, particularly in the network and server space, it helps us create rapidly deployable services with lower maintenance. More specifically, it helps us successfully maintain 43 (and growing) geographically diverse bastion host servers with a staff of three engineers, and even leaves them time to work on lots of other things.

The key to our success is in doing things very efficiently and very carefully. We use a central secured server which holds several configuration file distribution trees. One tree contains the base operating system for all of our bastion hosts. Additional distribution trees such as *dns* and *sendmail* exist for each type of bastion host application. All distribution trees are as similar to each other as possible. The directory layouts are similar, and the commands to check and install changes are the same, thanks to a commonly included *Makefile.inc* file. Finally, access to the distribution server is granted to only those who need access. Our Makefile automatically generates what configurations are necessary, logs in to systems that need to change, and loads the configurations.

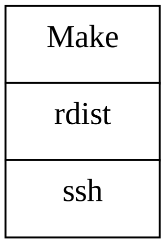
With our centralized build in place as a reference, we needed a method to keep all our servers in sync with the reference. Ease of use was very important. However, even more important, was to manage the servers securely. Building a system that could meet both of these requirements was a key challenge. We were already accustomed to managing the servers by logging in, hand editing, and setting the permissions and ownership on critical files. Keeping this familiar model would make the management task easier for us and any future administrators. We needed a tool that could clone entire directory trees between several machines. By using **rdist**, we could build and develop the master directories on the central management server. Then, we would clone the reference directory tree onto the remote servers.

This solved the ease of use problem. Now, we needed to determine how to use rdist without compromising the security of the remote server. In order for remote management to work, each remote server needed to be configured to allow our central management server to gain root access. It must do it in such a way that could not easily be exploited by anyone other than the central management server. We needed strong authentication, to avoid such things as IP spoofing. We needed strong encryption to avoid things like password snooping. Fortunately, rdist supports the use of alternatives to the *rsh* program. We found that **secure shell** (SSH) [3] worked well this purpose. It can use host-based public/private key authentication, so that only our central management system, with the right private key can gain access. It also uses encrypted sessions to prevent someone from sniffing out a cleartext password, or connection hijacking.

Additionally, rdist allows for minor differences between servers. During the design of our system, we considered using rsync, which also supports ssh, but it lacked this last feature.

Architecturally, our bastion host distribution scheme is shown in Figure 3:

Figure 3: Bastion Host Distribution Architecture



Make controls rdist, which is implemented to run over SSH. The combination of Make, rdist, and SSH work very well to allow the centralized management of remote UNIX servers.

When used to manage the remote servers, Make causes several operations to occur. It builds the distribution files, which our network installation process uses. It warns us about which files would be replaced or added on each server being targeted. And finally, it pushes the changes out.

Our network installation process allows us to build new servers very easily. A boot floppy is all that is needed to create or rebuild any of our managed servers. To accomplish this, there must already be several tarred and gzipped *build* files present on the central management server. These *build* files contain the latest server configurations and must be rebuilt whenever a change is made in the corresponding configuration directory tree. During the process of engineering a new set of configurations, we try to minimize the time required to build these files. Putting each set of configurations into its own directory and having a Makefile at the top of that directory forces only those directory trees containing changes to have their corresponding build files remade.

To minimize the number of files pushed out to any given host, rdist compares the sizes and timestamps of the target files. Only those files needing to be updated will cause a file to be replaced on the target server.

When we make a change, we carefully change a central distribution tree and then run a distribution check to verify what we are about to change. The check is done with a "make check" command. This check does two very important things. It allows one to verify that right things were made in the distribution tree. It also points out any other changes that have been made either to the distribution tree or to the bastion hosts. It is possible a colleague is working on one of the bastion hosts or someone is making changes that should not be made. Whatever the case, all changes to be made should be understood before they are implemented. We implement this check by invoking rdist with an option that just prints what must change.

When changes are well understood, they are installed with a "make install" command. This command is almost the same as the "make" check command, except that the Makefile calls upon rdist to actually distribute changes rather than just verify and report changes.

Sometimes special things need to happen before distribution, such as running M4 to generate a new sendmail.cf file. And sometimes things need to be distributed in a special way, such as when sending different versions of password files to different sites. These customizations are easily accommodated in the distribution tree's Makefile and Distfile.

We gain enormous flexibility by using make and rdist, and yet we're still able to maintain a simple and consistent interface throughout our entire distribution mechanism. This makes our distribution system powerful while being easy to use.

3.3 Managing Router and Bastion Host Configuration Files

The way we manage configuration files lets us back out change if we need to. Back-outs to changes are implemented in the same way as normal changes. A previous version of a file is re-installed into a distribution tree, then it is distributed just like any other file would be.

We often use RCS to assist us in coming up with the previous versions of files. RCS gives us past versions of files as well as their history information. RCS directories and files may be embedded into the distribution trees where they are easy to get at and use. They are automatically ignored by the distribution mechanism, in particular by using

```
except_pat( .*,v RCS)
```

in the rdist's Distfile. This minimizes the amount of files copied and keeps configuration distribution as fast as possible.

4. Experiences with the Make Approach

For the most part, using Make to control and manage our configurations has allowed us to overcome the maintenance challenges of our environment. We manage all of our bastion host configuration and router configurations from a few central systems. Management of configurations has become tremendously faster and more consistent. We no longer have to guess what files we need to install on systems after something has changed. When files need to be installed, we no longer have to copy individual files to individual systems. Instead, a command like "make install" automatically figures out what files have been changed, installs those files, and runs whatever utilities are necessary to configure the end system. With our router management, we no longer have to individually log into routers and type in configuration commands or download new configurations.

One of the greatest advantages to this system of ACL maintenance is that the individual files can be put under a revision control system such as SCCS or RCS. (We use RCS). Each change made can be noted in comments and also noted in the version control log. Keeping this history allows us quickly to answer questions such as when various changes were or what was changed on a particular date. It also allows us to compare older versions of the files so that changes can be tracked easily.

We have implemented macros to save typing. Intel is not fortunate enough to have a Class A. Instead, we have many Class Bs and Class Cs. Additionally, we are making increased use of private IP space [5]. We have several gateways that provide secure communications between Intel and other companies. This means that our access-lists from Intel into the DMZs cannot use "permit any" (0.0.0.0 255.255.255.255) because that would allow transit traffic from our business partners through our Internet connections. To prevent this, we have a list of about twenty networks which we have to allow into our DMZs. Instead of coding these into each ACL and having to change each ACL when a new network comes up, we maintain a list of these networks that are incorporated into the ACLs via a macro. The list is in a file called *intel-nets*. The macro expands the variable INTEL_NETS by running a PERL program that takes the appropriate line of the ACL and replaces the INTEL_NETS variable with each network in the intel-nets file.

Our Make driven maintenance has allowed us to scale up the number of firewall complexes at Intel. Our group originally only managed one firewall complex. Now we manage eight and are adding even more. Only with a tool like Make could we manage to scale the number of firewalls with the constraints we have.

The design has proven to be quite robust. We have a commitment to provide 99.95% connectivity to www.intel.com. We can do this because our ACLs allow traffic to fail-over from one gateway to another. We can also lose an entire Internet gateway (both providers or firewall routers) and still get Intel's traffic out to the Internet and back from the Internet in a way we're sure is secure. Yes, ACLs are long, but since they're optimized, most traffic gets through the routers without encountering long processing times.

Our Make based distribution system is a very powerful tool. It can distribute problem solving router and bastion host configurations all across the world. It can also distribute problem-creating configurations all across the world! In any case, this tool requires great discipline.

One problem we occasionally face is when one of our engineers is making changes on a bastion host, but not to the distribution tree on the central server. There can be some very good reasons for this, such as when new software is being tested. It is still a problem because it makes it hard to maintain consistency between the distribution tree and the actual files on the servers. The solution here is always to have staff members check, confirm, and resolve any problems before actually distribute changes. If a bastion host needs to be different for an extended period of time, it should be pulled out of the distribution tree(s) and possibly added to new distribution trees. The powerful nature of this distribution system makes it important to maintain consistency from day to day so each new change receives as much focus as possible.

Being able to quickly and easily push out changes can be a problem. Incorrect changes can be pushed out everywhere on all bastion hosts and routers, resulting in havoc. Before any change of significance is widely implemented, it is important...CRUCIAL...to make sure that it works beforehand. Disciplined preparation is required before you "just type make."

Changes, especially new versions of packages, should be pilot tested on one bastion host before they are rolled out to all bastion hosts. A new distribution tree should be created when installing a new package. A pilot server can be pulled out of the old package tree's distribution list and added to the new tree's distribution. As the pilot progresses, servers can be migrated from the old to the new tree.

Currently, educational use and charity use of the secure shell protocol is allowed free of charge. However, commercial use requires that you license each server. Have your lawyers visit http://www.ssh.fi/sshprotocols2/licensing/ssh2_non-commerical_licensing.html. Additionally, SSH can use any of several encryption ciphers, including idea, des, 3des, rc4, and blowfish. It can also use RSA public-private key pairs. There may be additional restrictions on the use of these ciphers and authentication methods depending on location.

We have occasionally encountered problems with Make, in that the Makefile itself will have the "dreaded spaces rather than tabs" problem. Sometimes we will find a space at the end of a line, but these are usually easy to diagnose. We have also changed the Makefile without touching any of the files and been told that targets are up-to-date. A little "touch" here and there solves that particular error.

5. Future Plans

We plan a number of extensions to our make driven update system, for both router configuration and host configuration. With router configuration, we currently have a way to optimize our router access lists, but this technique is manual. We would like a way to do this automatically – feeding the ACL usage data into a program and having optimized ACLs emerge, along with a list of ACL entries that can be deleted. We would like to be able to handle cisco's named access list feature. Currently, our scripts cannot deal with named access lists, only numbered. Also, we would like to generate the entire router configuration (not just the ACLs) from a template. That would allow us much greater standardization of router configurations and make it faster to bring up new gateways.

Another item that we wish to improve is the way we change our router configurations. Currently, our scripts log into the same way as a network administrator would – telnet to the router, get enable privileges, make changes, and then log off. Clearly this approach has problems as telnet sessions traverse DMZs. Using enhanced security features like Kerberos should improve the security of our router change processes.

With host configuration, we mentioned how it was easy to push out an error-ridden configuration to all of our bastion hosts. We would like to implement some sanity checks that would check configurations for common errors before they were pushed out and then run immediate checks after they are installed. This is particularly important when maintaining DNS tables. This way, we would notice if we pushed out a change that somehow damaged key DNS records at Intel.

6. Conclusions

Our experiences show that Make can be an effective way to maintain and administer firewalls. Intel Corporation manages a number of geographically dispersed firewall complexes that failover between themselves, including both router packet filtering access lists and bastion host configuration, using Make driven configuration. While this approach requires discipline to use (it is just as easy to push out a bad configuration as well as a good configuration), "just typing Make" solves many of the challenges of administering distributed yet interacting firewalls.

References

[1] Chapman, D. Brent and Elizabeth Zwicky. *Building Internet Firewalls*. O'Reilly and Associates, Inc., Sebastopol, CA 1995. pp. 66.

[2] Oram, Andrew and Steve Talbot. *Managing Projects with Make*. O'Reilly and Associates, Inc., Sebastopol, CA 1991.

[3] Libes, Don. *Exploring Expect*. O'Reilly and Associates, Inc. ., Sebastopol, CA 1995.

[4] SSH Home page. http://www.ssh.fi/

[5] Rekhter, Y., B. Moskowitz, D. Karrenberg, G.J. de Groot, E. Lear. "Address Allocation for Private Internets." RFC 1918, February 1996.