Randy Cook (SCSA) is a Senior Engineer with BayMountain (www.baymountain.com) a local IT services company. Randy was the co-author and technical editor of the *Sun Certified System Administrator for Solaris 8.0 Study Guide* (ISBN: 0-07-212369-9), and Syngress Publishing's *Hack Proofing Sun Solaris 8.0* (ISBN: 1-928994-34-2) and has written technical articles for industry publications. He has also hosted a syndicated radio program, *Technically News*, which provided news and information for IT professionals.

Chapter 7

Configuring Solaris as a Secure Router and Firewall

Best Damn Topics in this Chapter:

- Configuring Solaris as a Secure Router
- Routing IP Version 6
- IPV 6 Hosts
- Configuring Solaris as a Firewall

Introduction

With its foundations in Berkley Software Distribution (BSD) UNIX, Solaris—much like its predecessors—is a multifaceted operating system. It is perfectly suited to running on a 124-processor E15000 that acts as the foundation of a multinational banking firm or reproducing seismographs of the earthquakes along the San Andreas fault over the last 10,000 years within the period of a few minutes, but its performance and reliability as a secure router, secure gateway, and firewall are equally valuable. Although it will not outperform a hardware-based solution such as a Cisco router or a NetScreen firewall, it does offer reliable, stable service. Solaris is the operating system of choice for many commercial packages that provide firewall services.

Our first exposure to using Solaris for such a task was at a small Internet service provider (ISP) in eastern North Carolina. In the first year of operation, the ISP had anticipated no more than 1000 clients from the small coastal town. The end of the year came—with a total of 7000 clients, new service offerings in five additional towns along the Carolina coast, and lots of problems. Not only was this growth not anticipated; worse yet, it wasn't budgeted. Faced with the problem of an internal network and server pool both in need of access control, we faced the dilemma of making do with what we had. This type of dilemma often inspires the kind of panic that proves the resourcefulness of systems administrators.

In this chapter, we first examine the use of Solaris as a secure router and gateway. Next, we look at using Solaris as an Internet firewall, and we discuss using host-based firewalls on Solaris. Finally, we talk about guarding Internet access. We highlight the reasons for using Solaris for these types of tasks and talk about some of the security implications involved with using the OS in each scenario. We also examine implementations of these types and discuss some of the steps required in implementation.

Configuring Solaris as a Secure Router

To differentiate between a host and router, let's first define the functions of each. A *host* is typically a system with any number of interfaces that may or may not be connected on the same network. A host does not allow traffic to enter in one interface and out another. A typical server in a high-availability configuration has multiple interfaces, with each interface connected to a different network segment to prevent a single point of failure.

A *router* is a system with a minimum of two interfaces connected to at least two segments of different networks. The router allows traffic to reach its destination by entering one interface and passing out through another. An *interface* is loosely defined as a physical connection that allows other systems to communicate with the system via Ethernet, serial port, point-to-point link, or some other method. We will not get into a discussion of how the decision is made for the traffic to reach its destination; that issue is outside the scope of this chapter. A good reference on traffic routing and TCP/IP is *TCP/IP Illustrated, Volume 1: The Protocols*, by W. Richard Stevens.

Reasoning and Rationale

Let's attempt to answer the inevitable question, "Why use Solaris?" There are numerous platforms and designs available to use as a low-cost router, all of which are viable solutions. Some key factors in selecting one over others are the availability of hardware, the amount of time allotted to

thing. However, if the system's intention is to function as a multihomed host in a high-availability configuration, this configuration can have unexpected results.

To get a better understanding of why Solaris automatically routes traffic when two interfaces are present, let's look at some of the code in the S69inet script. We'll look only at the code pertinent to our discussion. On line 93, we have the following block:

```
if [ "$_INIT_NET_STRATEGY" = "dhcp" ] && [ -n "`/sbin/dhcpinfo Router`" ]; then
        defrouters=`/sbin/dhcpinfo Router`
elif [ -f /etc/defaultrouter ]; then
        defrouters='/usr/bin/grep -v \^\# /etc/defaultrouter | \
            /usr/bin/awk '{print $1}''
        if [ -n "$defrouters" ]; then
```

This code first checks DHCP for routing information. If the system does not return routing information from the program dhcpinfo, it next checks for the existence of the file /etc/defaultrouter, which is used for static default route entries. The last line in the block checks the variable *$defrouters* for a nonzero value. If the variable length is greater than zero, some further checking of routing information is performed. If the check on the last line of the block yields a nonzero value, the system sets the default routes contained in /etc/defaultrouter on line 124. Otherwise, it flushes the routing table. If neither of the first two tests is true, the script sets the *$defrouters* variable to a null value.

The decision of whether to run the system as an IPv4 router is made on line 186. The script first checks for the existence of the /etc/notrouter file. Following this check, the script checks the configured interfaces to count the number that were configured via DHCP. The script then checks for a number of interfaces greater than two (loopback plus one interface) or if any point-to-point interfaces are configured. Finally, the script checks to see if the /etc/gateways file exists. If:

- The /etc/notrouter file does not exist, the number of interfaces configured by DHCP is equal to zero, and the number of interfaces configured, including the loopback device, is greater than two
- There are one or more point-to-point connections, or
- The /etc/gateways file exists

the script executes ndd to manipulate the IP kernel module and sets the *ip_forwarding* variable to 1. The script then launches in.routed and forces in.routed to supply routing information. The in.rdisc daemon is started next, launched in router mode. Otherwise, *ip_forwarding* is set to 0, in.rdisc is launched in solicitation mode to discover routers on the network, and in.routed is launched in quiet mode.

Configuring for Routing

A default installation of Solaris with more than two interfaces (including the loopback interface) that aren't configured by DHCP will route traffic by default. This process, of course, depends on the system having not been altered by administrative staff. In some situations, however, it might

be impossible to reinstall an operating system on a machine that will be routing traffic. In this situation, we need to be able to configure the system to route traffic manually.

Let's walk through a check of an already configured and functioning system to ensure that it's ready to route traffic. First, we make a list of items to check and, if necessary, alter. We'll do this in step-by-step fashion, in order to pay due attention to detail and ensure that we don't miss a step that could result in failure of our objective. Following the step-by-step account, we briefly discuss each step and any possible caveats.

A Seven-Point Checklist

Here's our checklist:

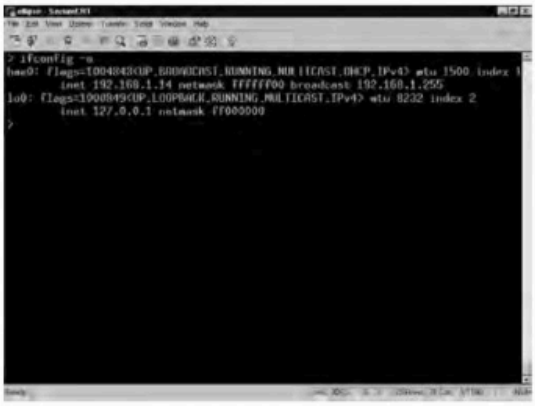
1. Check for interfaces configured via DHCP.
2. Ensure that each interface to be configured has a corresponding hostname.interface file in /etc and that the contents of the files are valid.
3. Check the /etc/rcS.d/S30network.sh file (inode link to /etc/init.d/network) for signs of alteration.
4. Check the /etc/rc2.d/S69inet file (inode link to /etc/init.d/inetinit).
5. Check for the /etc/notrouter file, and if it exists, remove it.
6. After the system has booted, poll /dev/ip for the status of the ip_forwarding variable.
7. Test the system in an isolated environment to ensure traffic routing.

Each step is covered in more detail in the following sections.

Step 1: Check for Interfaces Configured via DHCP

In the first step, we verify that all the interfaces are being configured with static information. As previously mentioned, a system using interfaces configured by DHCP will not be configured as a router. The easiest way to check for this configuration is by using the *ifconfig* command on a running system and then examining the output. Using the *all* flag with *ifconfig* typically displays the pertinent information, as we see in Figure 7.1.

Figure 7.1 A hme0 Interface That Has Been Configured with DHCP

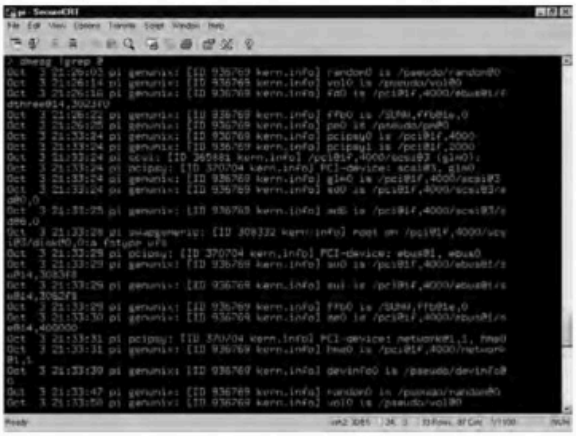


Step 2: Ensure Each Interface Has Corresponding File

Ensure that each interface to be configured has a corresponding hostname.interface file in /etc and that the contents of the files are valid. It is necessary to have a hostname.interface file for each interface to be configured when the system is bootstrapped. A nonexistent hostname.interface file will result in a nonexistent interface. Similarly, an incorrectly formatted hostname.interface file will result in an incorrectly configured interface.

For each interface to be configured by the system, create a hostname.interface file. For example, if there were two 100Mbit interfaces on a system, there would have to be a hostname.hme0 and hostname.hme1 file in the /etc directory. The device names can be discovered by reviewing the output of command, as we see in Figure 7.2.

Figure 7.2 Browsing dmesg for Interfaces Detected during Bootstrap



Ensure that the hostname.interface files contain one of two things: an IP address or a host name with an entry in the /etc/hosts file. In a standard configuration, the host name is placed in the hostname.interface file with an entry for the host name in the /etc/hosts file. When the system boots, it resolves this host name against the /etc/hosts file and configures the interface with the corresponding IP address. Although it's possible to place an IP address directly in the hostname.interface file, it is recommended that, for consistency, you follow the standard procedure. The file must contain either an IP address or a host name; it can't contain both.

Steps 3 and 4: Check the /etc/rcS.d/S30network.sh and /etc/rc2.d/S69inet Files

Check the /etc/rcS.d/S30network.sh file (inode link to /etc/init.d/network) for signs of alteration. In addition, check the /etc/rc2.d/S69inet file (inode link to /etc/init.d/inetinit). It is common practice for a systems administrator to alter boot scripts in order to create a more secure system. This practice can lead to problems for those who inherit such a system, however, because problems can occur that are not immediately traceable. Two scripts commonly modified are the /etc/rcS.d/S30network.sh and /etc/rc2.d/S69inet scripts.

Often, documents that discuss the hardening of systems instruct administrators to alter these files and change or comment out sections of code to create a more secure configuration. Some automated system-hardening tools alter these scripts as well. These scenarios can result in unpredictable behavior and abundant frustration when a system's mission and configuration change.

In the third and fourth steps, we verify the integrity of these two files. We can do this via one of three methods. The first method, and the most unreliable one, is to visually inspect the file for signs of alteration by using an editor and examining the change time of the file. The second and more reliable method is to compare the file against a known unaltered copy of the file. The third and most secure method is to compare the file md5 sum of the file against the known sum in the Sun Fingerprints Database. When in doubt, restore from the CD-ROM.

Step 5: Check for the /etc/notrouter File and, If It Exists, Remove It

Check for the /etc/notrouter file; if it exists, remove it. The /etc/notrouter file is used to keep the system from being configured as a router. A typical system on which this file will exist is a correctly configured multihomed host. This file is not created by default, nor is there a configuration option in the install process to create it. Therefore, a freshly installed system will not have this file and so this situation won't be a concern. However, you should manually check previously installed hosts. If this file exists, remove it.

Step 6: Poll /dev/ip for the Status of the ip_forwarding Variable

After the system bootstrap, poll /dev/ip for the status of the *ip_forwarding* variable. The system will not route traffic if IP forwarding is not turned on. Therefore, after you've taken the previous configuration steps, reboot the system, and the *ip_forwarding* variable of the IP kernel module will be polled to ensure that the system is prepared to route traffic. The result is a Boolean. If the variable returns 1, the configuration was successful and the system is ready to route traffic. If the variable returns 0, there was an error somewhere in procedure and the system will not route traffic.

Step 7: Test the System

Test the system in an isolated environment to ensure traffic routing. In the final step, testing should be conducted to ensure that the system is functional. A private, isolated segment of network should be created to test the router's functionality and ensure proper configuration, reliability, and performance.

Security Optimization

A number of parameters associated with TCP/IP on a Solaris system can be modified to provide enhanced security. The configuration of ARP, IP, TCP, UDP, and ICMP in their default state might not provide the greatest level of security. For the sake of brevity, we don't delve deeply into this topic nor discuss it in brief. This would not do the topic justice. This topic has been covered comprehensively in a document, "Solaris Operating Environment Network Settings for Security," by Keith Watson and Alex Noordergraaf of Sun Microsystems Blueprints. Their documents are available from Sun Blueprints at www.sun.com/blueprints.

Security Implications

You don't have a hope of security or integrity for your network without first having a secure router. Therefore, the implementation of a system as a router must be secure by design. This consideration must be made at the very beginning of system design and observed diligently through deployment and afterward in maintenance. The intricacies of designing a secure router are covered in detail elsewhere in this book. Here we give some general guidelines to enhance security. From these guidelines, we'll repeat the minimalism mantra.

Minimal Installation

A secure router should include a minimal, functional installation of the operating system. However, this is more a management issue than a security issue. Simply put, smaller software installations make machines that are more easily managed and monitored for intrusion.

A system with a smaller installation is more easily managed because only the necessary pieces are in place. What constitutes *necessary* is the software to achieve your mission and business needs. A system with a minimal installation also removes a number of unnecessary services and makes it easier to monitor the system for intrusion.

There are two camps on the types of software that should be installed on a system. One side is against having a C compiler on the system; the other is for it. Neither side is right or wrong, but both have valid lines of reasoning to take into account.

The side against an accessible local C compiler fears a local user compiling exploits or other programs and using the system for unauthorized activities. Such violations could lead to a local user gaining elevated privileges or unauthorized network access. The other side of the argument believes that having a C compiler on the local system is a necessary utility. Without a C compiler, they believe, it's impossible to build programs from source.

We're happy to announce that we're proud members of both camps. We're against local users having unlimited free reign of a system through some goody built with a C compiler, but not against having the C compiler. This risk can be eliminated through proper permissions and access control such as RBAC or simple access control lists (ACLs).

Minimal Services

A router needs very little in terms of services. Since the system has one purpose, there isn't a necessity for things such as NFS, NIS, RPC, and sendmail. By eliminating these services, you enhance overall system performance.

Additionally, eliminating these services closes entry points for possible intruders. By limiting the channels that allow an intruder potential access to the system, we've mitigated the risk of opening a system to future compromise by a new vulnerability. Shutting down all services or using the system solely as a router isn't always possible. This is, however, the recommended practice.

Many of these services are started via the Internet daemon (inetd). Commenting out the services is a good practice. Commenting out the services and not starting inetd at all is the best methodology. The inetd is started in the /etc/rc2.d/S69inet script.

Another good practice is checking the rc directories in /etc for programs that might be started. For example, the rc3.d directory starts a number of services that, in addition to being

Routing IP Version 6

Beginning with versions distributed from February 2000 and later, Solaris 8 is IP version 6 capable. It is not possible to configure Solaris 8 as a solely IPv6 system from the installation menu. It is possible, however, to configure an interface to communicate with any IPv6 host on the network and still retain IPv4 communications. This process is known as *running a dual stack*. A Solaris system can be configured to run strictly IPv6 by removing the hostname.interface file, although this configuration could cause problems when communicating with IPv4 hosts that do not currently support IPv6. This makes it possible for Solaris to function in any IPv6 environment as a host, gateway, or router.

In this section, we discuss setting up a Solaris IPv6 router. We talk about the file configurations necessary to make IPv6 functional. We also discuss the programs necessary to IPv6. However, we do not discuss the protocol, since there are better documents that do so. It is recommended that a user interested in setting up IPv6 for the first time reference the appropriate RFCs.

Configuration Files

Putting everything in place to make IPv6 functional on a Solaris 8 system is relatively easy. One prerequisite is having the system to route traffic configured for regular IPv4 traffic. Once we have completed the steps for configuring an IPv4 router, we can proceed with the setup of an IPv6. In this section, we talk about the files necessary to get an IPv6 router working. These files include the hostname6.interface file, the ndpd.conf file, and the ipnodes file.

The hostname6.interface File

This file is similar to the previously discussed hostname.interface for IPv4. The syntax of items contained in the hostname6.interface file is different from that of the IPv4 version, however.

Previously, the only thing needed in this file was either an IP address or a host name with an entry in the /etc/hosts directory. Now additional parameters must be entered in the hostname6.interface file. These parameters are parsed by the S30network.sh script in /etc/rcS.d when the system boots and are then passed to ifconfig. In the following example, we see a hostname6.interface entry for our IPv6 router:

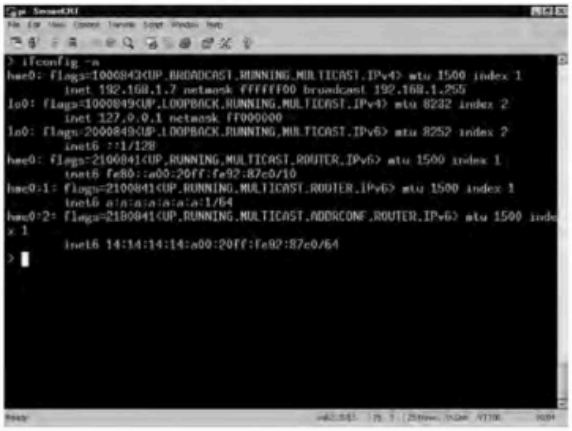
```
addif sturgeon.mydomain.com/64 up
```

The first parameter we see is *addif*. The *addif* parameter is an extension of the Solaris *ifconfig* command, which tells *ifconfig* to add the address to the next available interface. Since we are seeing this file in the /etc/hostname6.hme0 file, *ifconfig* searches the interface table for the next available virtual interface on the hme0 device. The address resolving to sturgeon.mydomain.com will be configured to this interface. At the end of the line, we see the *up* command, which makes the interface network accessible. As we can see in Figure 7.3, this address was configured to the hme0:1 device.

As we can see, the address is now configured with the ROUTER flag and is ready to handle traffic from other hosts. However, additional configuration steps have been taken prior the inter-

face being brought up. We'll talk about these steps shortly, in addition to the configuration steps necessary for *ifconfig* to resolve the address for sturgeon.

Figure 7.3 A Configured IPv6 Address Attached to the hme0:1 Interface after a Reboot



One subtle point we have not mentioned is that we're configuring this interface with a static address. There is a good reason to do so. With IPv6, it's possible to autoconfigure hosts when they boot. These systems poll the network during bootstrap to get information necessary to communicate with the rest of the network. If we do this with a router, we're forced to remember that the link-local address in.ndpd assigns to the interface at bootstrap. This address is usually easily remembered because it's typically composed of our network information and the Media Access Control (MAC) address of the interface. Whether or not we configure Solaris 8 with a static IPv6 address, the link-local address is configured by design.

In most cases, it is much easier to remember an address we've specifically assigned to the system. If there is ever a problem on the network, we'll know the address we have given to the router. This knowledge makes the router a little more accessible, a little easier to remember, and a little easier to name with a host name. This process does not take into account DNS, which will be mentioned later.

The ndpd.conf File

The ndpd.conf file is the configuration file for the in.ndpd program, or the Internet Network Discovery Protocol Daemon. This configuration file is supposed to reside in the /etc/inet directory and is read by the daemon when it is launched by the S69inet script when the system enters run-level 2, typically during the bootstrap process. It is worth mentioning that the ndpd.conf file does not exist by default. To understand why this configuration file is significant, we should talk about the in.ndpd program and the purpose it serves.

The in.ndpd program, when implemented on a router, must be configured to act as a router for the IPv6 network. This configuration involves making some entries in ndpd.conf to make the daemon the known router for the network. When other systems bootstrap and send a request for

routing information via Neighbor Discovery Protocol, in.ndpd responds as the router for the network.

Minimal configuration of ndpd.conf that provides IPv6 functionality on a Solaris system consists of the following two entries:

```
ifdefault AdvSendAdvertisements true
prefix 0A:0A:0A:0A:0A:0A:0A:0/64 hme0
```

To understand these entries, let's examine them in a little more detail. On the first line, we see the *ifdefault* command. The *ifdefault* and *if* commands are used to set interface configuration parameters. The *ifdefault* command must precede any *if* commands, because *ifdefault* is used to specify any default operations of the interface.

The next variable we see is the *AdvSendAdvertisements* parameter. This parameter designates whether or not the system will function as an IPv6 router. By default, this option is set to *false* on systems, which causes in.ndpd to run in host mode. When *AdvSendAdvertisements* is set to *true*, in.ndpd initiates itself as a router on the interface on which it is being configured to operate, sending periodic router advertisements via multicast and responding to router solicitations.

On the next line, we see the *prefix* entry. The *prefix* command controls the configuration variables for each prefix, or network. There is also a *prefixdefault* variable, which is similar to the *prefix* variable, except that the *prefixdefault* variable specifies configuration parameters for all prefixes. The *prefixdefault* variables must precede any prefix variables in ndpd.conf.

Next on the prefix line we see the network address. This is the 128-bit address, divided into eight blocks of 16 bits. At the end of the address we have the netmask. It is worth mentioning that this is a classless interdomain routing address block, also known as CIDR. We should also mention that this address is strictly for educational purposes and should not be used. At the end of the string, we have the name of the physical network interface.

Additional configuration options are supported in this ndpd.conf file. The preceding configurations will get the daemon functioning as the IPv6 router for the 0A:0A:0A:0A:0A:0A:0A:0 network. For more information on other supported options, see the ndpd.conf(4) man page.

The ipnodes File

With IPv4, Solaris uses the /etc/inet/hosts file to resolve known hosts. This process is controlled by the nsswitch.conf file in the /etc directory. When a process from the local system attempts to connect by host name to another system via IPv4, the nsswitch.conf forces the process to check the /etc/inet/hosts for name resolution. With IPv6, Solaris now uses the /etc/inet/ipnodes file to resolve known hosts. This is controlled by the ipnodes entry in nsswitch.conf. The ipnodes configuration file structure is similar to that of the hosts file. In Figure 7.4, we see two entries in the ipnodes file of sturgeon.

On the first line, we see the entry for our router, sturgeon.mydomain.com. Much like the hosts file, this entry assigns the pictured address to the host name and gives it a canonical name of *sturgeon*. Following this entry, we see an entry for one of the nodes on the network, barracuda.mydomain.com. This address allows us to reach the system barracuda without the necessity for DNS.