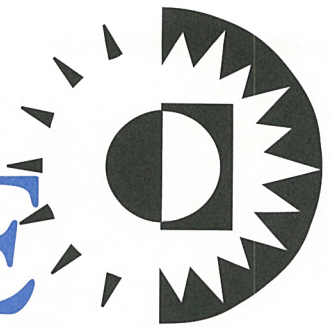


INSIDE SOLARIS™

Tips & techniques for users of SunSoft Solaris



IN THIS ISSUE

- 1**
Armoring Solaris
- 4**
Real applications and the WWW
- 6**
Shell toolbox 101, part1
- 9**
Configuring Network Interface Cards
- 12**
Shells
- 15**
Sun on a budget
- 15**
About our contributors

Visit our Web site at www.zdjournals.com/sun

Armoring Solaris

by Lance Spitzner

Firewalls are one of the fastest growing technical tools in the field of information security. However, a firewall is only as secure as the operating system it resides upon. This article will take a step-by-step look at how you can best armor your Solaris box, both Sparc and x86. We'll take a look at techniques like TCP wrappers, shown in **Figure A**, to protect your telnet sessions. These steps can apply to any situation; however, we'll be using Checkpoint Firewall 1 on Solaris 2.6 as an example.

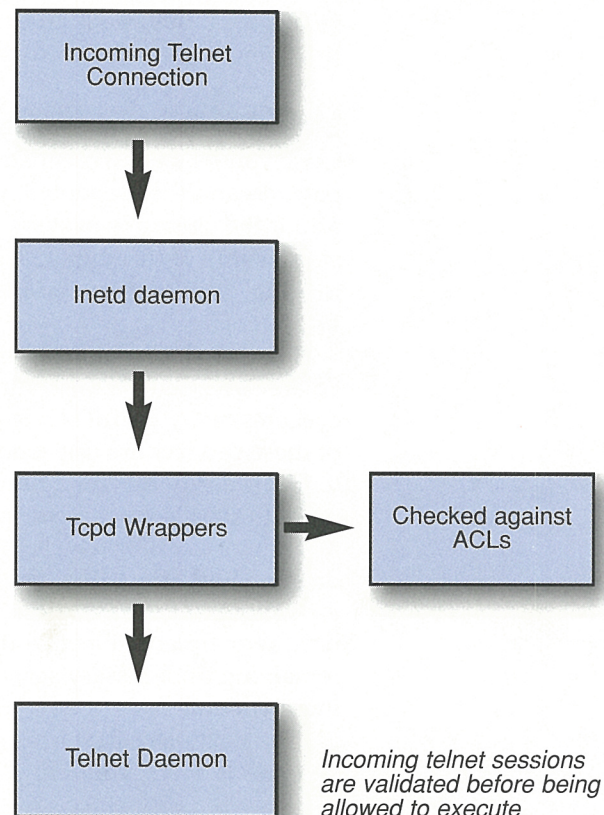
Installation

The best place to start in armoring your system is at the beginning: OS installation. Since this is your firewall, you can't trust any previous installations. You want to start with a clean installation, where you can guarantee the system integrity.

Place your system in an isolated network. At no time do you want to connect this box to an active network or the Internet, exposing the system to a possible compromise. To

get critical files and patches later, you'll need a second box that acts as a go-between. This second box will download files from the Internet, and then connect to your isolated, configuration "network" to transfer critical files.

Figure A



Once you've placed your future firewall box in an isolated network, you're ready to begin. The first step is selecting which OS package to load. We recommend End User with Online manual pages. The idea is to load the minimum installation, while maintaining maximum efficiency. The less software that resides on the box, the fewer potential security exploits or holes. Anything above the End User package, such as Developer, is adding useless, but potentially exploitable, software. Remember, this is your firewall; it shouldn't be running anything else.

The other option is to load the Core installation package, which is smaller and leaner than the End User installation. However, you must know exactly what you're doing, because the Core package may be missing critical executables or libraries. Usually, you'll have to add specific packages. We prefer the End User installation, because it tends to be more stable and flexible. Finally, add the manual pages. We find these to be a critical tool with little security risk.

Once the system has rebooted after the installation, be sure to install the recommended patch cluster and security patches. Also, be sure to use your go-between box to get the patches; the firewall box should always remain on an isolated network. Patches are critical to armoring a system and should always be updated. Bugtraq is an excellent source for following bugs and system patches.

Eliminating services

Once you've loaded the installation package, patches, and then rebooted, you're ready to armor the operating system. Armoring consists mainly of turning off services, adding logging, tweaking several files, and TCP Wrappers.

First, we'll turn off the services. By default, Solaris is a powerful operating system that executes many useful services. However, most of these services are unneeded and pose a potential security risk for a firewall. The first place to start is `/etc/inetd.conf`. This file specifies the services for which the `/usr/bin/inetd` daemon will listen. By default, `/etc/inetd.conf` is configured for 35 services, but you only need two: `ftp` and `telnet`. You eliminate the remaining unnecessary services by commenting them out.

The next place to start is `/etc/rc2.d` and `/etc/rc3.d`. Here, you'll find startup scripts

launched by the `init` process. Many of these processes aren't needed. To prevent a script from starting, replace the capital `S` with a lowercase `s`. That way, you can easily start the script again just by replacing the lowercase `s` with a capital `S`. The following scripts aren't needed and pose serious security threats to your system:

- `S73nfs.client`—Used for NFS mounting a system. A firewall should never mount another file system.
- `S47autofs`—Used for automounting. Once again, a firewall should never mount another file system.
- `S80lp`—Used for printing. Your firewall should never need to print.
- `S88sendmail`—The MTA daemon that listens for incoming E-mail. Your firewall box shouldn't be your E-mail server. However, your system will still be able to send mail, such as alerts.
- `S15nfs.server`—Used to share file systems, which is a bad idea for firewalls.
- `S76snmpdx`—`snmp` daemon.

Later, after you've completed both the OS and Firewall installation and configuration, you should turn off the following (other) `/etc/rc2.d` scripts:

- `S99dtlogin`—CDE daemon, starts CDE or OpenWindows by default.
- `S71rpc`—`portmapper` daemon.

Without these scripts, you can't launch a GUI interface. You most likely will want the GUI interface to help you with the installation and configuration. However, once you're done, there's no need for the GUI or either of the scripts. Both `rpc` and `OpenWindows` or `CDE` have many exploitable ports and services. To see how many services are running with both the GUI and `rpc` running, type the command

```
ps -aef | wc -l
```

Once you've finished with the installation and have turned off `S99dtlogin` and `S71rpc`, type the command again and compare how the number of services has decreased. The fewer services running, the better.

Logging and tweaking

Once you've eliminated as many services as possible, you want to enable logging. Most system logging occurs in `/var/adm`. You need to add two additional log files there: `suolog` and `loginlog`. `/var/adm/suolog` logs all `su` attempts, both successful and failed. This allows you to monitor attempts to gain root access to your system. `/var/adm/loginlog` logs consecutive failed login attempts. When a user attempts to log in five times, and all five attempts fail, this fact is logged. To enable the files, just touch the files `/var/adm/loginlog` and `/var/adm/suolog`. Ensure that both files are `chmod 640`, as they contain sensitive information.

Next comes tweaking. This involves various file administration tasks. The first thing you need to do is create the file `/etc/issue`. This file is an ASCII text banner that appears for all telnet logins. This legal warning will appear whenever someone attempts to log into your system.

We also want to create the file `/etc/ftppers`. Any account listed in this file can't ftp to the system. This restricts common system accounts, such as `root` or `bin`, from attempting ftp sessions. The easiest way to create this file is the command:

```
cut -f1 -d: > /etc/ftppers
```

Ensure that any accounts that need to ftp to the firewall are *not* in the file `/etc/ftppers`.

Finally, ensure that `root` can't telnet to the system. This forces users to log into the system as themselves and then `su` to the root. This is a system default, but always confirm this in the file `/etc/default/login`, where `console` is left un-commented.

TCP Wrappers

TCP Wrappers is a must for any firewall. No armored system should be without it. Created

by Wietse Venema, TCP Wrappers is a binary that wraps itself around `inetd` services, such as `telnet` or `ftp`. With TCP Wrappers, the system launches the wrapper for `inetd` connections, which logs all attempts and verifies the attempt against an access control list. If the connection is permitted, TCP Wrappers hands the connection to the proper binary, such as `telnet`. If the connection is rejected by the access control list, then the connection is dropped.

Many of you may be wondering why a firewall would need TCP Wrappers, since the firewall does all that for you. The answer is simple. First, in case the firewall is compromised or crashes, TCP Wrappers offer a second layer of defense. Second, and just as important, TCP Wrappers protect against Firewall misconfigurations. We've often seen firewalls misconfigured, especially in VPN situations, allowing unauthorized users telnet access to the firewall. Third, TCP Wrappers add a second layer of logging, verifying other system logs.

You can get TCP Wrappers from coast.cs.purdue.edu/pub/tools/unix. Once again, be sure to use your go-between system to retrieve and compile TCP Wrappers. We want to protect the armored Solaris box within its isolated network.

Once downloaded, be sure to review the README file first; it's an excellent introduction to TCP Wrappers. We recommend the following two options when compiling TCP Wrappers. First, go with `paranoid`, as this does a reverse lookup for all connections. Second, use the `advance` configuration, which is actually quite simple. This configuration keeps all the binaries in their original locations, which may be critical for future patches.

Implementing TCP Wrappers will involve editing several files (these examples are based on the `advance` configuration). First, once compiled, the `tcpd` binary will be installed in the

Listing A: Add the bottom two files to `/etc/inetd.conf`.

```
# Ftp and telnet are standard Internet services.
#
#ftp    stream tcp    nowait root    /usr/sbin/in.ftpd    in.ftpd
#telnet stream tcp    nowait root    /usr/sbin/in.telnetd in.telnetd
#
ftp     stream tcp    nowait root    /usr/local/bin/tcpd  in.ftpd
telnet stream tcp    nowait root    /usr/local/bin/tcpd  in.telnetd
.
```

/usr/local/bin directory. Second, the file /etc/inetd.conf must be configured for the services that are to be wrapped, as shown in [Listing A](#) on page 3. Notice how inetd first launches the wrapper, /usr/local/bin/tcpd, then the actual server daemon. Third, /etc/syslog.conf must be

Listing B: *Added to /etc/syslog.conf*

```
# Log all TCP Wrapper connections
#
local3.info
/var/adm/tcpdlog
```

Listing C: *Example of /etc/hosts.allow and /etc/hosts.deny*


```
cat /etc/hosts.allow
ALL: maggie,zeus,merlin: ALLOW

cat /etc/hosts.deny
ALL: ALL
```

edited for logging tcpd, as we've done in [Listing B](#). Last, the access control lists /etc/hosts.allow and /etc/hosts.deny, shown in [Listing C](#), must be added.

Once all the proper files have been edited and are in place, restart /usr/bin/inetd with kill-HUP. This will restart the daemon with TCP Wrappers in place. Be sure to verify both your ACLs and logging before finishing.

Conclusion

We've covered some of the more basic steps involved in armoring a Solaris box. The key to a secure system is having the minimal software installed, with protection in layers such as TCP Wrappers. There are many additional steps that can be taken, such as file permissions, backups, etc. Remember that no system is truly 100 percent secure. However, with the steps outlined above, you greatly reduce the security risks. 

CGI AND C

Real applications and the WWW

by Paul A. Watters

New programming languages come and go, but the responsibilities of application developers and programmers always remain the same. Applications have to be delivered within a specific time frame, on budget, and more often than not in today's distributed computing environment, they have to be accessible. Meeting these requirements shouldn't require rewriting our code every year or so! In this article, we'll look at ways of making available existing Solaris-based applications on the WWW that won't break the bank.

Recycle and reuse

Many programmers would associate the word *reuse* with C++ and other object-oriented languages. We're going to look at it in a slightly different context, although the lessons learned can apply equally to procedural and object-oriented applications. Although it might seem heretical, many applications don't need to be rewritten every time a new programming language comes along, especially if rewriting

isn't going to produce a tangible benefit (other than programmer satisfaction at using the latest technology, of course).

Many people would argue that existing code that's UNIX-based should be rewritten in Java, for example, so that it's readily portable and instantly useable on any platform that has a Java interpreter. Sharing applications and making them as portable as possible is a good thing. However, many existing applications simply weren't written with a single workstation in mind; client-server architectures are still very useful for sharing large amounts of data for which it isn't efficient to have many copies lying idle on hard disks around the lab or the office. In addition, if our applications involve any kind of mathematical operations, it's often faster to use a workstation to display the results of operations being performed by a multi-processor server that has no other overhead. With 64-bit server architectures on the way, we shouldn't be fooled into thinking that the humble PC will ever replace dedicated number-crunchers.

CGI & WWW options

But sticking with a client-server system for applications accessing large databases that require asynchronous updating, or just large amounts of data processing, doesn't mean that we need to sacrifice the benefits of being able to use applications through the WWW. We'll look at how to use the common gateway interface (CGI), which allows users with a WWW browser to run applications on a Solaris-based server. This method has the advantage of reducing code-development costs while taking advantage of existing applications and an existing set of protocols to exchange real-time data and respond to end-user requests.

Most WWW users encounter CGI when they use a search engine to retrieve URLs, telephone numbers, and map directions. However, the specification allows for quite general exchanges of data between client and server. The interface is especially suitable for programs written in C (when using the POST method of sending a request from an HTML document), since this sends data in a form which can be read as a standard input stream. Existing data input routines that parse streams and extract variable values can then be used to retrieve values entered by users from their Web page. The length in bytes of the input stream can be read from the environment by using

```
getenv("CONTENT_LENGTH")
```

After our program has parsed the input, set appropriate variables, and completed its assigned task, the results can be sent through an output stream in HTML form by

```
printf("Content-type: text/html\n\n");
```

after which, normally marked-up text and URLs will be displayed on the client's terminal.

As an alternative to writing our own code to read and write through the CGI, we can use one of the many freeware C libraries available on the WWW for converting streamed data into variables. One such library is Eugene Kim's CGIHTML, which is easy to use and compiles successfully in the Solaris environment. Point your browser to

www.eekim.com/software/cgihtml/

Case study

CGI and C form the best partnership when the number of parameters passed to the server is

small, but the computation required of the server is large. A good example might be the case of artificial neural networks, which are higher-order statistical models commonly used for modelling complex systems, such as the stock market. Although many neural network systems are available to run on PCs, either as C programs or as Java applets, the number of matrix operations that must be performed at each cycle becomes enormous when all the parameters associated with each data point presented to the network's "input layer" is computed.

For the case of the stock market, if we wanted to predict more than just a few hundred points, many PCs would take well over one hour to compute a result. However, if the same PCs were used as clients who issued requests through their Web browser to a server with several UltraSPARC processors, the prediction might only take in the order of seconds or minutes, while the user can still work on their client computer without slowing down computation speed.

An example neural network system for natural language processing, using this kind of client-server design is available at


www.comp.mq.edu.au/~pwatters/semantic.html

The purpose of this simulation is to retrieve a pronunciation for a word the user enters in a text box, along with two parameters that affect network performance. The simulation is performed on a Solaris system with two UltraSPARC processors, and takes only several seconds to produce online output for each network cycle, and postscript graphs when the simulation has completed. This is a substantial improvement over ports made recently to PC-based systems, where each simulation took several minutes to complete.

Further reading

If this case study has been convincing, by far the best place on the WWW to learn more about CGI programming is

www.cgi-resources.com/

In this article, only the relationship between C and CGI has been discussed, but any Solaris-based programming language that can read streams (such as shell scripts and Larry Wall's PERL) could be used, depending on the application's goals. 

Shell toolbox 101, part 1

by Jeff Forsythe, Sr.

The tools we'll introduce in this article each have relative usefulness on their own, but when put together, they make up a very nice editing toolbox for any Solaris programmer or system administrator. Before introducing them, however, let's discuss a convention for keeping your scripts straight.

Shell scripts

Just like C program binaries, shell scripts ought to be kept in bin directories. This lets you know they're executable. Shell scripts get out of control if you don't organize them well. When organizing, there are three types of scripts to consider: the script function, the script program, and the script library.

The first type, the *script function*, is a function written in your favorite shell script language. Functions are designed to be included in other shell functions or script programs. As a matter of convention, it's wise to store your script functions in a file with a `.fnc` (pronounced funk) extension instead of a `.sh`. This lets you know at a glance that it's a function rather than a script program. This differs slightly from an in-line function that's stored within a script program. A `.fnc`-type

function is designed to be used by more than one script program, whereas in-line scripts are designed to be called from within programs but have no outside reusability. In [Listing A](#), let's look at a hypothetical function called `get_response.fnc`.

Notice that the main function within this file is also called `get_response`. This is self-documenting and highly encouraged.

If you know what the name of the `.fnc` is, then you know the name of the function to call. The function calls the `DisplayPrompt`

function. `DisplayPrompt` has no intrinsic value outside of `get_response.fnc`. Therefore, it's an in-line function. Later, if you write another function or script program that needs the same code, then `DisplayPrompt` would be promoted to a `.fnc`.

A good rule of thumb is: If a function is used by more than one script, then put it in a `.fnc` file by itself and include it in the others. Otherwise, put it in-line and call it instead.

The second type of script is the shell *script program*. It's just that—a program written in the shell script language. This is what you normally think of when talking about shell scripts. The shell program may have commands, variables, in-line functions, include other scripts or `.fnc` functions, or have no functions at all, depending on its requirements. Scripts usually have the `.sh`, `.ksh`, or `.csh` extension, depending on which shell they were developed in. All examples given here are in the Bourne shell (`.sh`).

Finally, the third type of script is a shell *script library*. Like all libraries, the shell library is a collection. It contains such things as included functions, script programs, and internal or environment variables. It could even include in-line functions or other libraries. The purpose of the library is to make it easy to include all the necessary components of a given application. Library files should get a `.shlib`, `.kshlib`, or `.cshlib` extension to differentiate them from the ever-popular C program `.lib` libraries. Now that we have the basic concept of functions, programs and libraries, let's put them together to make something useful.

Building an editing library

In the rest of this article, and next month's issue, we'll build an editing library. It doesn't sound all that exciting, but when it's done, you'll be glad you did it. We'll need a few functions, some script programs, and perhaps some string and rubber bands to tie it all together. We'll build and use most of these tools individually. Then, in order to make your life easier, we'll put them all together in a script library. Since they all relate to editing, we'll

Listing A: Pseudo-code—
`get_response.fnc` pseudo-code

```
DisplayPrompt()
{
    Display prompt
}

get_response()
{
    Repeat
        DisplayPrompt
        Get a response
        Test for validity
    Until Valid response

    Return response
}

# END get_response.fnc
```

name it *edit.shlib*. After that, whenever we need to perform editing functions, all we need to do is include the library (*.edit.shlib*) and we have all our tools available to us in one swoop.

Edit library description

Our shell library called *edit.shlib* includes the entire collection of editing tools as listed below:

- *shell.sh*—Shell script editing front-end. Creates a documentation header and a framework for a script, and then calls the version control program (*vc.sh*).
- *file.sh*—Same as *shell.sh*, but for non-shell script files such as configuration files, etc.
- *vc.sh*—Version control shell. Performs version control based upon each user's individual configuration file, calls everyone's favorite editor (*vi*), and then calls *affects.sh* to see if the changes made affect other files. If so, you're asked if you'd like to edit the affected file(s), as well. This process is recursive, so an editing session could potentially include any number of files.
- *affects.sh*—Currently, vaporware (if you decide to build it, E-mail it to the author) lists the files that may be affected by the current editing change. It uses *header.fnc* to obtain information, meaning that this is only as good as the documentation in the header of the file. So, keep your documentation current!
- *header.fnc*—Displays the documentation header of the given file (created by *shell.sh* or *file.sh*).
- *logmsg.fnc*—Log message function. Logs the date, time, calling program, message type, and message to the given log.

This library or its individual elements (like any other library or script) can be included in other libraries, a *.profile*, a shell script, or at the command line. This is especially useful so that you can use the library contents from the command line. This particular library was built with that usage specifically in mind. For instance, if you're interested in the documentation of a particular file, you can use the header function to retrieve it. If you included the library in your *.profile*, all you need do is type *header filename* and the header information is returned to you. Otherwise, type: *. path/edit.shlib*, and then the header command. This is quite useful.

Background information

We'll look at each of the scripts individually, but the main emphasis here is to talk about them collectively as a library. Each tool was built over the last six years or so, as a need arose, but based in whole or in part on the previous work.

For instance, *shell.sh* is a neat little tool that was written about six years ago as a front end to *vi*. It asked a few questions, created a header, and then threw you into *vi* to create your file. In earlier versions, it checked to see if the file existed, asked if you wanted to edit or overwrite the file, and performed some other minor tasks. Since adding the version control program, this has become unnecessary. Version control was added as an add-on to *shell.sh*. Of course, *shell.sh* was *affected* by this addition, thereby creating the need to modify it. This brought up the idea for the *affects.sh* script. Your shell tools have undoubtedly developed the same way, or if you're new, they will. They just might not be in library form yet. Take the time and do that today; it will save you countless hours later.

The version control script is patterned after one found in an article in another publication about four years ago ("Sys Admin" Vol. 3, No. 5 Sept/Oct 1994). The methodology used in the original article's script and in *vc.sh* was borrowed from DEC's VMS file naming convention. The filenames on VMS end with a semicolon and a sequential number. It's no replacement for commercial version control packages, but it works well, it's very easy to implement, and, best of all, it's free!

Start building

First, build the *logmsg.fnc*, as shown in [Listing B](#) on page 8, because it has the greatest utilitarian function on its own, and is a basic building block for the rest of the library. You probably already have your own logging script that will readily adapt (if yours is better, E-mail a copy to the author). Just remember to modify the *logmsg* calls in the other scripts if you use your own. After *logmsg.fnc*, build *header.fnc*, which is found in [Listing C](#), also on page 8. The header of a file (as with all internal documentation) is extremely important. You can *never* over-document a program, no matter how simple it seems at the time. A year or two from now you'll wonder why a counter is initialized to 4 and not 0 or 1. Good header documentation

Listing B: logmsg.fnc Listing

```
#!/bin/sh
#####
# FUNCTION:      path/logmsg.fnc
# ARGUMENTS:    prog : $0
#               msg-type :
#               S  ⚭ START
#               M  ⚭ MESSAGE
#               E  ⚭ ERROR
#               F  ⚭ FINISH
#               "msg" : Message
#               log : log file
#
#####

logmsg()
{
echo "`date +%m/%d/%Y`!`date \
+%H:%M:%S`!$1!$2!$3" >> $4

exit 0
}

# END logmsg.fnc
```

will explain the 4 and header.fnc will make it easy to find out why. Yeah, you could just pg the file, but that gives you everything, not just the header. The header is also a nice thing to have in a hardcopy documentation folder.

The header in **Listing C** has been gutted in the interest of the publication. Only required information is included. The calling program must *include* the logmsg.fnc. Then, it can be called such as the example in **Listing D**.

The log provides you with invaluable information for many applications, including:

- Reporting programs (to search for errors) to determine if a program has finished
- Determining at what point a program is running or if it's hung as part of a locking mechanism (A nice topic for a future article—don't you think?)

Headers are enclosed within two lines of 57 # signs. That has been shortened to 38 here for publication reasons, but the script checks for 57. You may use any character to separate

Listing C: header.fnc Listing

```
#!/bin/sh
#####
# FUNCTION:      header.fnc
# USAGE:        header filename
# ARGUMENTS:    filename
#
#####

header()
{
if [ $# -ne 1 ]
then
echo "USAGE: header filename"
exit 1
fi


if [ ! -s ${1} ]
then
echo "Can't find ${1}"
exit 2
fi

IFS="
"
COUNT=0

for LINE in `cat ${1}`
do
echo ${LINE}

if [ "${LINE}" =
#####
#" ]
then
COUNT=`expr ${COUNT} + 1`
fi

if [ ${COUNT} -eq 2 ]
then
exit 0
fi
done
}
```


the header lines but do not use any asterisks (*) in any of your lines, unless required. Suffice it to say that a disaster occurred, long ago but never forgotten, that caused 57 pharmacy stores to close down, just because a pound sign was missing and a line of asterisks followed. The author is proud to be the one who found the problem and not the one who caused it. This gives your toolbox a start. Next month, we'll expand our toolbox even further. 

Listing D: *logmsg Include, Call and Output*

```
. ${FNCDIR}/logmsg.fnc

logmsg $0 F "FINISH" ${HOME}/logs/examp.log

08/12/1998|06:31:19|examp.sh|S|START
08/12/1998|06:32:04|examp.sh|M|Regular msg
08/12/1998|06:33:47|examp.sh|E|Error msg
08/12/1998|06:34:21|examp.sh|F|FINISH
```

GETTING YOUR INTERFACES TO TALK

Configuring Network Interface Cards

by Lance Spitzner

This article is the first of a two-part series. In this first article, we'll cover how to configure, troubleshoot, and modify system interfaces. The second article will cover static routing tables for systems with two or more interfaces. In both articles, we'll be focusing on TCP/IP in an Ethernet environment.

Interfaces

Network Interface Cards (NICs) are what allow your system to talk to the network. When they don't work, neither do you. Our goal is to get your interface up and properly running.

The first place to start is installing and testing the hardware. Once you've installed the hardware, SPARC systems can be tested at the EPROM level to verify the NICs. Use the manual that accompanies the interface card on how to test that specific card.

Solaris x86 is a little different, as there is no true EPROM, and the drivers are different. However, Solaris x86 2.6 is Plug and Play compatible, and we've had fairly good luck adding NICs.

Once you've confirmed at the hardware and driver level that everything works, the fun can begin. The place to start is the `ifconfig` command. This powerful command allows you to configure and modify your interfaces in real time. However, any modifications made with `ifconfig` aren't permanent. When the system reboots, it will default to its previous configuration. First, I'll show you how to make all modifications with the `ifconfig` com-

mand. The second half of this article will cover making these modifications permanent by modifying the proper configuration files.

`ifconfig`

I'll show you which interfaces are currently installed and active. Remember, just because you added the physical network interface card doesn't mean it's active. If you do an `ifconfig` before you've configured the device, the interface won't show up. Once configured, however, the typical output of the `ifconfig` command would look like this:

```
lo0: flags=849<UP,LOOPBACK,RUNNING,MULTICAST>
mtu 8232
    inet 127.0.0.1 netmask ffffffff
hme0: flags=863<UP,BROADCAST,NOTRAILERS,
RUNNING,MULTICAST> mtu 1500
    inet 192.168.1.132 netmask ffffff00
broadcast 192.168.1.255
    ether 8:0:20:9c:6b:2d
```

Here we see two interfaces, `lo0` and `hme0`. `lo0` is the standard loopback interface found on all systems. `hme0` is a 10/100Mbps interface. All `hme` interfaces are 10/100Mbps, all `le` interfaces are 10Mbps, all `qe` interfaces are quad 10Mbps, and `qfe` interfaces are quad 10/100Mbps.

There are three lines of information about the interface. The first line is about the TCP/IP stack. For the interface `hme0`, we see the system is up, running both broadcast and multicast, with an `mtu` (maximum transfer unit) of

1500 bytes—standard for an Ethernet LAN. *Notrailers* is a flag no longer used, but kept for backwards compatibility reasons.

The second line is about the IP addressing. Here we see the IP address, netmask, in hexadecimal format, and the broadcast address. The third line is the MAC address. Unlike most interfaces, Sun Microsystems' interfaces derive the MAC addressing from the NVRAM, not the interface itself. Thus, all the interfaces on a single SPARC system will have the same MAC address. This doesn't cause a problem in routing, since most NICs are always on a different network. Note that you must be the root to see the MAC address with the `ifconfig` command; any other user will only see the first two lines of information.

The first step in bringing up an interface is "plumbing" the interface. By plumbing, we're implementing the TCP/IP stack. We'll use the above interface, `hme0`, as an example. Let's say we had just physically added this network interface card and rebooted; so, now what? First, we plumb the device with the `plumb` command:

```
ifconfig hme1 plumb
```

This sets up the streams needed for TCP/IP to use the device. However, the stack hasn't been configured, as you can see below:

```
hme0: flags=842<BROADCAST,RUNNING,
MULTICAST> mtu 1500
    inet 0.0.0.0 netmask 0
    ether 8:0:20:9c:6b:2d
```

The next step is to configure the TCP/IP stack. We configure the stack by adding the IP address, netmask, and then telling the device it's up. All this can be done in one command, as seen below:

```
homer #ifconfig hme0 192.168.1.132
netmask 255.255.255.0 up
```

This single command configures the entire device. Notice the `up` command, which initializes the interface. The interface can be in one of two states, up or down. When an interface is down, the system doesn't attempt to transmit messages through that interface. A down interface will still show with the `ifconfig` command; however, it won't have the word *up* on the first line.

Virtual interfaces

A virtual interface is one or more logical interfaces assigned to an already existing interface. Solaris can have up to 255 virtual interfaces assigned to a single interface.

Once again, let's take the interface `hme0` as an example. We've already covered how to configure this device. However, let's say the device is on a VLAN (virtual LAN) with several networks sharing the same wire. We can configure the device `hme0` to answer to another IP address, say 172.20.15.4. To do so, the command would be the same as used for `hme0`, except the virtual interface is called `hme0:*`, where `*` is the number you assign to the virtual interface. For example, virtual interface one would be `hme0:1`. The command to configure it looks as follows:

```
ifconfig hme0:1 172.20.15.4
netmask 255.255.0.0 up
```

Once you've configured the virtual interface, you can compare `hme0` and `hme0:1` with the `ifconfig` command.

```
hme0: flags=843<UP,BROADCAST,
RUNNING,MULTICAST> mtu 1500
    inet 192.168.1.132
    netmask fffff00 broadcast 192.168.1.255
    ether 8:0:20:9c:6b:2d
hme0:1: flags=842<BROADCAST,
RUNNING,MULTICAST> mtu 1500
    inet 172.20.15.4
    netmask ffff0000 broadcast 172.20.255.255
```

Here you see the two devices, both of which are on the same physical device. Notice how the virtual interface `hme0:1` has no MAC address, as this is the same device as `hme0`. We can repeat this process all the way up to `hme0:255`. The operating system and most applications will treat these virtual devices as totally independent devices.

Configuration files

You now know how to configure your NICs. Unfortunately, any modifications, additions, or deletions you make with `ifconfig` are only temporary; you'll lose these configurations when you reboot. Now, I'll discuss what files you have to configure to make these changes permanent.

The place to start is the file `/etc/hostname.*` (where `*` is the name of the interface). In the case of `hme0`, the file name is `/etc/hostname.hme0`. The virtual interface `hme0:1` would have the

filename `/etc/hostname.hme0:1`. This file has a single entry—the name of the interface. This name is used in the `/etc/hosts` file to resolve name to IP address.

The file, `/etc/hostname.*`, is critical; this is what causes the device to be plumbed. During the boot process, the `/etc/rcS.d/rootusr.sh` file reads all the `/etc/hostname.*` files and plumbs the devices. Once plumbed, the devices are configured by reading the `/etc/hosts` and the `/etc/netmasks` file. By reading these two files, the device is configured for the proper IP and netmask, and brought to an up state.

Let's take the device, `hme0`, as an example. During the boot process, `/etc/rcS.d/rootusr.sh` looks for any `/etc/hostname.*` files. It finds `/etc/hostname.hme0`, which contains the following entry:

Homer

The shell file `/etc/rcS.d/rootusr.sh` looks in `/etc/hosts` and resolves the name `homer` with an IP address of `192.168.1.132`. The device, `hme0`, is now assigned this IP address. The script then looks at `/etc/netmasks` to find the netmask for that IP address. With this information, the startup script brings up interface, `hme0`, with an IP address of `192.168.1.132` and a netmask of `255.255.255.0`. It may seem redundant having the script review the netmask of a class C address. However, don't forget that, starting with 2.6, Solaris supports both classless routing and VLSM (Variable Length Subnet Masks), both of which we'll discuss in next month's issue.

As you've seen in this example, there are three files that must be modified for every interface. The first is `/etc/hostname.*`, which is the file you create to designate the interface's name. The second file is `/etc/hosts`, where you resolve the IP to the interface name. Last is `/etc/netmasks`, which is where you define the netmask of the IP address.

Troubleshooting

Once you've mastered the tricks to modifying your interfaces, troubleshooting should be easier. You should look for several things when troubleshooting an interface.

First, when working with an unfamiliar machine, often you don't know how many physical interfaces are on the machine. A quick way to tell is use `dmesg`; this will give you information on the physical hardware.


Look for `le0`, `qfe0`, `hme0`, or `qe0`. These are the names assigned to the physical devices.

If an interface isn't responding to the network, check to be sure it's the correct IP address and netmask. The `ifconfig` command is a quick and temporary way to change IP and netmask information for troubleshooting purposes.

Mtu is another possibility. Some systems may have problems communicating due to fragmented packets. Changing the mtu size may solve that problem. You'll notice that you didn't have to set the mtu size in the examples above; these are set to defaults automatically, such as 1500 for Ethernet interfaces.

If that fails, try bringing the face down, and then re-initializing it with the `up` command. If nothing else works, unplumb the device, and then plumb it again. Basically, this reinstalls the TCP/IP stack.

Conclusion

Network Interface Cards are critical to your systems networking capability. Understanding the configuration of your interface(s) ensures your system's productivity. Next month, we'll look at routing tables and ensuring that once your interfaces are configured and up, your packets will know where to go. 

What's Coming

Over the last few months we've received feedback about what you, our reader would like to see in *Inside Solaris*. Coming next month you'll find articles that cover:

- CGI scripts and data security
- Process and Thread Scheduling
- Continuation of our Shell toolbox

As we move into 1999 we plan to also feature articles that will focus on:

- Configuring PPP
- Alternate PPP software
- Network routing
- Jumpstart

If there are any other topics that you would like to see covered or are interested in becoming a writer yourself, please drop me an E-mail at gsuhm@tacticsus.com.

Garrett Suhm
Editor
Inside Solaris

Shells

by H-W Schlote

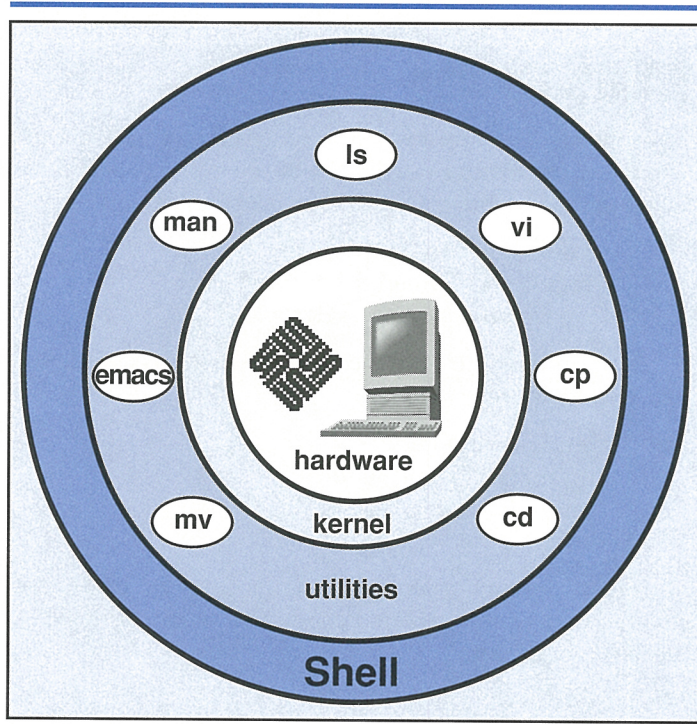
In the article “Shell toolbox 101, part 1” on page 6, we discussed building libraries of shell scripts. Let’s take a step back now and give some background on the various UNIX shells that are available. It can be useful to know what else is available, even though most users only know the one they use.

Background

According to Funk & Wagnalls New Encyclopedia:

SHELL or SEA SHELL—hard, largely calcareous skeleton of marine animals, especially the mollusks (see Mollusk). Because of their beauty, bright colors, variety in shape, and abundance on sea- and lake-shores, shells have long been used for ornamentation, tools, and coins. They are important today in the production of such items as “pearl” buttons, jewelry, and other decorative items. Shell-collecting is a popular hobby throughout the world.

Figure A



Here's the principal view of the shell's functionality.

So, why is the thing with the prompt where we type in commands called a *shell*? Because, it surrounds the UNIX core, as shown in **Figure A**. The shell is a program that listens to your terminal. It accepts and interprets the commands you type and turns them into requests to the underlying kernel, to perform the work you want.

There's some jewelry to find. In the dark, old days when I moved from DOS to UNIX, I couldn't understand how people could be satisfied without command line editing or the ability to move the cursor up and edit the last command. Even today, shells providing command line editing, even in this simple form, are exceptions. But, on the other hand, there's the one shell with a nearly complete set of emacs (the famous editor—originally written by Richard Stallman for PDP-10) commands, including searching backward.

If you compare the UNIX operating system with a car, then the steering wheel is part of the user interface. The shell can be thought of as the user interface to UNIX. Just as you can have servo steering in your car, one shell may offer a lot of conveniences, while others leave you with nearly no support at all.

This article covers just a few topics and examples about shells. It concentrates on interactive shells and makes no claim to be exhaustive. Maybe it will help you to decide which shell is the right one for you. If you want to go into detail, have a look at the references given at the end of this article.

Shell flavors

There are many different UNIX shells available, including *ash*, *bash*, *csh*, *choice*, *ksh*, *pdksh*, *smash*, *sh*, and *tcsh*. In general, they can be divided into two groups: the Bourne shell compatible ones, and the C shell compatible ones. I'll focus here on the five most popular shells:

- The *Bourne* shell (invoked as *sh*) is the oldest shell and often referred to as the standard shell.

- The C shell (invoked as `csh`) uses C syntax. Compared to the Bourne shell, it offers more features. But, using some of the simple features, like redirecting output and dividing `stdout` and `stderr`, isn't really straightforward. I'll show later how this can be done.
- The Korn shell (invoked as `ksh`) is a superset of the Bourne shell that lets you edit the command line. There are restrictions for the Korn shell, but lots of things can be configured. A public-domain version, `pksh`, exists also.
- The GNU extended C shell (invoked as `tcsh`) provides command line editing with cursor up/down functionality.
- The GNU Bourne-Again shell (invoked as `bash`) is the crown of all shells. It's a superset for the `ksh` and understands almost every `csh` command. `bash` is ultimately intended to be a conformant implementation of the IEEE POSIX Shell and Tools specification (IEEE Working Group 1003.2).

The disadvantage of the last two shells is that they consume a lot of resources. This is one of the reasons why many system administrators prefer not to install any of them, but stay with the shells provided by their hardware vendor. Normally, these are the first three shells mentioned (`sh`, `csh`, and `ksh`).

Output redirection

Shells know about standard file descriptors. 0 means `stdin`, 1 corresponds to `stdout`, and 2 is `stderr`. If you want to redirect output, you have to specify which output and do this by giving the number of standard file descriptor for `sh`, `ksh`, and `bash`. This results in

```
cmd 1> cmd.out
```

or shorter

```
cmd > cmd.out
```

to redirect `stdout` to file `cmd.out`. The second (short) form is the correct syntax for `csh`, too. Things differ if you want to redirect `stdout` to file `cmd.out` and `stderr` to `cmd.err`. I use this very often. Remember that `stderr` is unbuffered, whereas `stdout` is at least line buffered. So, if you don't separate `stdout` from `stderr`, all you have is a mess of a message file.

For `sh`, `ksh`, and `bash`, we simply have

```
cmd 1> cmd.out 2> cmd.err
```

Unfortunately, (t)csh only offers the choice to either redirect `stdout` alone or to redirect `stdout`, as well as `stderr`. Therefore, we have to use grouping to get the correct result:

```
(cmd > cmd.out) >& cmd.err
```

This means redirect `stdout` to `cmd.out`, and afterwards that is all that's left to `cmd.err`. This isn't really straightforward, but at least it works. If you build a C program and you start a program in a `csh` via a call to `exec`, you really run into trouble if you want to distinguish `stdout` and `stderr`.

On the other hand, redirecting both `stdout` and `stderr` to one file is very simple in `csh`, but results in some thinking or a look into the manpage or other documentation for Bourne shell compatibles. If you want to redirect both `stdout` and `stderr` to file `cmd.out` you type in `csh`:

```
cmd >& cmd.out
```

and

```
cmd > cmd.out 2>&1
```

in one of `sh`, `ksh`, or `bash`. The latter command reads redirect `stdout` to file `cmd.out` and redirect `stderr` to `stdout`. But, as mentioned before, `cmd.out` will most probably be garbled by strings from `stdout` and `stderr`.

Command line editing

Command line editing with cursor up/cursor down functionality is only provided by `tcsh` and `bash`. The standard shell, `sh`, doesn't provide any command line editing at all. The `ksh` provides the `fc` command. If you're used to it, you get along with it quite well. The `csh` offers a set of command substitutions from quick substitution to `sed` (the famous stream editor) syntax.

I must admit that I'm not able to remember all these fancy substitutions and recall command syntax. I prefer using the *cursor up key*, and the last command is there to be edited, and when I want to execute once again the last command containing string `foo`, I want to type `[Ctrl]R foo` like I do in `emacs`. This thing is offered by `bash` only. `tcsh` isn't bad. Most of the time, you'll use cursor up/down to re-execute commands anyway.

Adjusting the prompt

Unfortunately `sh`, `csh`, `ksh`, and `tcsh` don't offer a lot of possibilities to adjust the prompt. In general, the environment variable `PS1` holds the primary prompt string and can be changed to adjust the prompt to your personal choice. If environment variable `PROMPT_COMMAND` is set, the value is executed as a command prior to issuing each primary prompt for some shells. Thus, changing `PS1` wouldn't result in any change because normally variable `PROMPT_COMMAND` resets `PS1`. In this case, you have to unset variable `PROMPT_COMMAND` first.

Normally, UNIX users work on more than one machine. Consequently, the information the prompt should provide is which machine he is currently using and the current working directory. But, I don't want to see the name of my home directory in my prompt, if I'm somewhere below it.

In `bash`, I define `PS1` to be `"\h:\w\ $"` (the string inside the double quotes). This results in the hostname, followed by the current working directory. But, if the current working directory starts with my own home directory, the name of my home directory is substituted with `~`. `\$` results in `$` for normal users and `#` for the super-user.

Using the definition for `PS1` given above, my prompt looks like:

```
Gandalf: ~/Solaris.Inside/shells
```

If you want to have the current user included, you simply add `\u` somewhere to `PS1`. This is easy, isn't it?

For all other shells, you can define scripts to produce the same results. This leads to executing a script each time [ENTER] is pressed, and results in consuming a lot of system resources by just printing the shell prompt. So, if you want to use one of the more simple shells, you should stay with a simple prompt, like `%`.


Conclusion

In this article, I provided some remarks about popular shells. The main differences lie within interactive use. Programming syntax differs between these shells, too, but almost everything you can perform with one shell, you can perform with the others, also. It's just a matter of syntax. But, for reasons of efficiency and saving resources, I suggest using the `ksh` for programming.

Availability

The following shells are part of the Solaris operating system: `sh`, `csh`, `ksh`, `bash`, and `tcsh` can be obtained for example from sunsite.unc.edu.

Further reading

- *Introducing UNIX System V* by Rachel Morgan and Henry McGilton, McGraw-Hill
- *UNIX in a Nutshell* by Daniel Gilly and the staff of O'Reilly & Associates, O'Reilly & Associates 

Are you a good tipper?

Do you have any great Solaris tips that you've discovered? If so, send them our way! If we use your tip, it will appear on our weekly online ZDTips service. Visit www.zdtips.com to check out all our available tips. We may also publish it here in *Inside Solaris*. Your byline will appear along

with your E-mail and/or Web address. Send your tips to sun@zdtips.com, fax them to (716) 214-2386, or mail them to

The Editor, *Inside Solaris*
500 Canal View Boulevard
Rochester, NY 14623

Sun on a budget

Amazing things have happened to hardware in the Sun world in the last year. Sun has come out with all new workstations that really push the price/performance ratio to new heights. Both the Ultra 5 and Ultra 10 lines are both affordable and powerful. But these aren't your only options if you need to set up a workstation on a budget.

Old but not forgotten

Because of Sun's aggressive pricing, used Sun equipment has never been a better deal than now. Last-generation Sun workstations Such as Sparc

10's and Sparc 20's that had sold for tens of thousands of dollars new can be found for very reasonable prices on the used market.

They may not have the latest UltraSparc CPU's, but machines like the Sparc 20 have the ability to handle four processors as well as SCSI I/O (which the Ultra 5 and Ultra 10 workstations lack without add-on cards) and are built like tanks. These machines make perfect firewalls, news servers, and mail servers, as well as an inexpensive way to test for a Sun developer to acquire a workstation for home use.

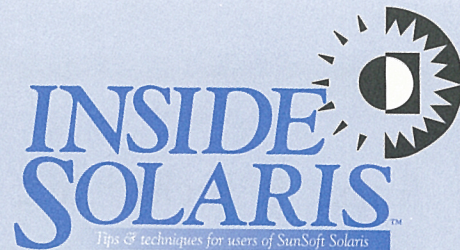
About our contributors

Lance Spitzner enjoys learning by blowing up his UNIX systems at home. Before this, he was an Officer in the Rapid Deployment Force, where he blew up things of a different nature. You can reach him at lsnitzner@enteract.com or www.enteract.com/~lsnitz.

Paul A. Watters is a research officer in the Department of Computing, Macquarie University, Australia. He has developed a number of numerically-intensive simulations using the Solaris development environment that are now accessible through the WWW. You can reach him at pwatters@mpce.mq.edu.au.

Jeff Forsythe, Sr. started programming in 1978. He's worked with Solaris and other UNIX flavors in retail, banking, manufacturing, and now the federal government at the US Bankruptcy Court. You can contact him at forsythe@tusco.net.

H-W Schlote was born in 1969 in Soltau, Germany. While studying physics at TU Braunschweig, he started programming in the UNIX environment for research projects. By studying information technology in addition to physics, he got a deeper understanding for operating systems, neural networks, fuzzy systems, and many other areas. After leaving the university with a degree in physics, he worked for about one year for a software firm in Braunschweig. He then moved to SUFFIX where he is now working as system analyst and project leader. He can be reached at H.Schlote@suffix.de.



Inside Solaris (ISSN 1081-3314) is published monthly by ZD Journals, 500 Canal View Boulevard, Rochester, NY 14623.

Customer Relations

US toll free(800) 223-8720
Outside of the US(716) 240-7301
Customer Relations fax(716) 214-2386

For subscriptions, fulfillment questions, and requests for group subscriptions, address your letters to

ZD Journals Customer Relations
500 Canal View Boulevard
Rochester, NY 14623

Or contact Customer Relations via Internet E-mail at zdjcr@zd.com.

Editorial

Editor.....Garrett Suhm
Assistant EditorJill Suhm
Copy Editors.....Rachel Krayer
Christy Flanders
Taryn Chase
Contributing Editors.....Jeff Forsythe, Sr.
H-W Schlote
Lance Spitzner
Paul A. Watters
Print DesignerLance Teitsworth

General ManagerJerry Weissberg
Editor-in-Chief.....Joan Hill
Editorial DirectorMichael Stephens
Managing Editor.....Kent Michels
Circulation Manager.....Jeff Marcus
Print Design Manager.....Charles V. Buechel
VP of Operations and Fulfillment.....Michael Springer

You may address tips, special requests, and other correspondence to

The Editor, *Inside Solaris*
500 Canal View Boulevard
Rochester, NY 14623

Editorial Department fax(716) 214-2387

Or contact us via Internet E-mail at sun@zdjournals.com.

Sorry, but due to the volume of mail we receive, we can't always promise a reply, although we do read every letter.

Postmaster

Periodicals postage paid in Louisville, KY.

Postmaster: Send address changes to

Inside Solaris
P.O. Box 35160
Louisville, KY 40232

Copyright

Copyright © 1998, ZD Inc. ZD Journals and the ZD Journals logo are trademarks of ZD Inc. *Inside Solaris* is an independently produced publication of ZD Journals. All rights reserved. Reproduction in whole or in part in any form or medium without express written permission of ZD Inc. is prohibited. ZD Journals reserves the right, with respect to submissions, to revise, republish, and authorize its readers to use the tips submitted for personal and commercial use.

Inside Solaris is a trademark of ZD Inc. Sun, Sun Microsystems, the Sun logo, SunSoft, the SunSoft logo, Solaris, SunOS, SunInstall, OpenBoot, OpenWindows, DeskSet, ONC, and NFS are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. Other brand and product names are trademarks or registered trademarks of their respective companies.

Price

Domestic\$99/yr (\$9.00 each)
Outside US\$119/yr (\$11.00 each)

Back Issues

To order back issues, call Customer Relations at (800) 223-8720. Back issues cost \$9.00 each, \$11.00 outside the US. You can pay with MasterCard, VISA, Discover, or American Express.

ZD Journals publishes a full range of journals designed to help you work more efficiently with your software. To subscribe to one or more of these journals, call Customer Relations at (800) 223-8720.

To see a list of our products, visit our Web site at www.zdjournals.com.

PERIODICALS MAIL

SunSoft Technical Support
(800) 786-7638

Please include account number from label with any correspondence.

This month we will review some of the more popular sources for used Sun equipment. In the coming months we will explore the used Sun market and show you the advantages (and disadvantages) of buying used equipment. For now, check out **Table A** for a sample of used Sun equip-

ment vendors. This list is not exhaustive, but it is meant to give a sample of what is available in the used market. If you have your own favorite used Sun equipment vendor, send me an E-mail at gsuhm@tacticsus.com and we will share these with our readers. 

Table A: Some vendors of used (and new) Sun equipment

Company	Description	Web Site/Phone#
Rave Computer Association Inc.	Reseller of used and refurbished Sparc, UltraSparc and Sparc Clones built by Rave	www.rave.com (800) 966-RAVE
Apcom Systems Inc.	New and used Sun equipment And Sparc clones. Online quote generator is available on their Web site	www.apcom.com (888) 422-0867
Integrated Solutions Group Inc.	Used Sparc and UltraSparc workstations, servers and peripherals	www.isgsun.com (425) 882-0400
Sun Data	Reseller of a variety of workstations and servers from various vendors. Flexible on-line quote generator	www.sundata.com (888) SUN-DATA
EIS Computers Inc.	Used Sparc and custom-built clones. Also OEM for Solaris X86 machines	www.eis.com (800) 351-4608
Minicomputer Exchange Inc.	Specialize in used Sun and Silicon Graphics hardware.	www.mce.com (888) 733-4400

