January 1999 • $9.00
Vol. 5 No. 1

ZD JOURNALS

INSIDE SOLARIS

Tips & techniques for users of SunSoft Solaris

IN THIS ISSUE

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

CGI and data security

by Paul A. Watters

The CGI (Common Gateway Interface) is a popular method of operating on user data collected from an HTML <FORM> on a Web site. We can use CGI to process data by either an interpreted Perl or shell script, or a compiled program, which resides on the server. We can then return either a dynamically generated HTML page, or some other kind of data (for example, a graphics file) from the server-side application. However, the increased use of CGI programs has given rise to a number of recent security concerns, which potential users of custom-made applications and off-the-shelf scripts should be aware of.

CGI: Compiled or interpreted?

Applications that make use of the CGI interface have become increasingly popular on Solaris-based server systems, with which personal computers of many descriptions can interface using the WWW. A common example is a Web interface to a centralized database system. As shown in Figure A, a client

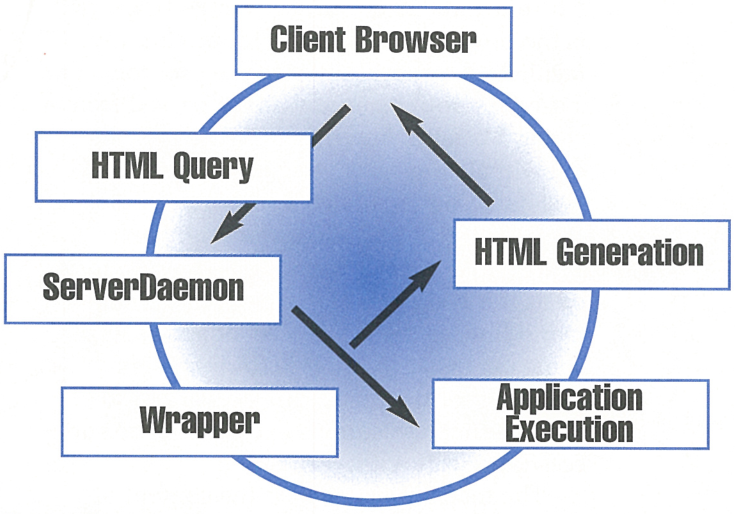


Figure A: *CGI process flow as initiated from a client PC*

ZIFF-DAVIS a SOFTBANK company

using a low-end PC can execute complex searches on a remote Solaris machine, accessed by HTML forms utilizing the CGI interface, without increasing overheads on their local machine. The results of the search, which might include graphics, text, and sounds, can then be returned in a dynamically generated Web page. This approach strongly supports the client-server model in a Web environment, providing access to up-to-date information at call.

CGI appeals to two main audiences: novice users and experienced users. Novice users often use freeware scripts downloaded from a resource site to include many advanced elements on their Web pages (like page counters, time/date displays, and guestbooks), without having to learn how to program.

Alternatively, experienced users, based in medium-to-large businesses, often use CGI to provide a modern interface to legacy systems, which might be expensive to rewrite in a modern compiled language (like C++). This approach not only saves money on expensive reprogramming, which must often be performed from the ground up, but enables a very fast turnaround time on providing a modern Web-based interface for clients.

There are certainly more general uses of CGI than we'll be looking at in this article. However, as we'll see, casual users are vulnerable to CGI exploits because they're not aware of some features of UNIX shell scripts, while experienced programmers might not be aware of the many exploits that are currently available over the Internet.

Query strings: Size matters

Currently, the major concern in Web security is the power of CGI scripts to unwittingly facilitate the execution of rogue programs on a server through a query string passed from a client. Mistakes in writing CGI scripts can allow malicious users to gain access to key system accounts. It's very important to learn the first law of CGI programming: parse everything! Every query string must be treated as if it were being executed on the command line, as that's the level of access which some exploits can provide for intruders. Planning for the worst-case scenario also ensures that less-serious exploits can be successfully dealt with.

The major concern with the content of query strings is that they can contain non-alphabetical characters like the tilde (~), which often have a special meaning in UNIX, and can be used in some cases to execute malicious commands. An example is using the `eval` statement from the Bourne shell and Perl scripting language, which will quite happily evaluate a query string with a semi-colon appearing somewhere within it. The semi-colon will, no doubt, be followed by a malicious command to mail the entire system's password file to a waiting hacker (although password shadowing can reduce the risk of non-root access to the password file). Using an additional code segment in shell scripts or Perl can usually fix this kind of problem—for example, all input can be parsed by a function to remove all non-alphabetical characters.

It isn't only the content of query string that we have to be conscious of; as with many things in life, size also counts. One problem with passing a query string is that the authors of some off-the-shelf scripts and compiled programs don't check the length of the query string, and usually declare the variable that stores the query string to be of constant size. For example

```
char query_string[1024]
```

allocates 1024 bytes to the variable `query_string` of type char, assuming that 1024 bytes will be the maximum argument size passed. We can see the folly of declaring variables in this way when there are a number of well-known exploits available on the Web that pass extraneous bytes after the declared 1024 to overwrite the address space declared for the `query_string` variable. This allows malicious users to execute arbitrary commands on the server in some cases.

The count.cgi program, for example, which is used to record and display the number of times a WWW page has been accessed, is a script widely used by non-programmers. However, failing to declare array sizes dynamically potentially allows the stack space of the program to be overwritten when a specific set of arguments is passed in the query string (CERT Advisory AA-97.27). Intruders can force the count.cgi program to execute arbitrary commands with the privileges of the server daemon. If the daemon is run as root (that is, where all child processes have root privileges), the results could be disastrous for the entire system, not just the Web interface or files in the document root directory. If you're still running an old version of this program, CERT advises you to update immediately:

www.fccc.edu/users/muquit/Count.html

Code solutions

The first technique that can be used to stop malicious use of a query of a large size is to dynamically allocate memory, depending on the size of the query string, which can fortunately be determined from an environment variable. Thus, if the size of the query string is passed by the variable CONTENT_LENGTH, we can simply declare a pointer to an array of type char (in C), and then set its size dynamically in the program, after CONTENT_LENGTH has been read from the environment:

```
query_string=(char *) calloc((size_t)
CONTENT_LENGTH, sizeof(char));
```

This means that the bounds of the array query_string can't be overwritten.

However, although this technique will stop a user from executing arbitrary commands on the server, it isn't enough to stop denial of service attacks that might be aimed at crashing the server and preventing access from paying clients, for example, rather than stealing passwords. In this case, several megabytes of data might be passed as a query string, eventually consuming all available physical memory and swap space. Even if enough memory is available to process a single query, the performance of the server will be dramatically reduced during processing. If two or three clients repeated this query consecutively, there's real scope for almost permanently denying service. One way of avoiding this kind of problem is to place a sensible limit on query strings, by rejecting input streams that are larger than the maximum required for any particular page on the server. A 1024-byte limit is generally sufficient for most Web pages, although it might need to be larger for complex query strings. For example, a simple check such as

```
if (CONTENT_LENGTH > 1024) then
  reject_query();
```

might be a useful remedy, if we have a standard error message defined in reject_query().

System solutions

Other CGI security solutions are aimed at the system level. For example, CGI.pm is a Perl library that parses and interprets strings composed of HTML queries in a standard, secure way. Using a standard programming library can save mistakes and simple errors that might not result in any observed changes in an application's behavior, but might open potential security holes. More information about CGI.pm can be found at:

stein.cshl.org/WWW/software/CGI/cgi_docs.html

Of course, the best solution in the long run might be to install a safer Web server—one that incorporates many existing CGI-related security strategies in its code. The apache Web server, for example, now has the suEXEC feature, which is designed to screen all CGI requests using a wrapper before any user-level scripts or programs are executed. SuEXEC also allows users to run CGI scripts and programs under usernames that are different from the standard *nobody* or *www* under which the daemon usually runs (or worse still, allowing child processes of the server daemon to run as *root*).

This feature considerably reduces the many security risks involved with allowing non-system users to run CGI programs for public use (using a wrapper is really mandatory on multi-user systems like Solaris). SuEXEC also refuses to run programs that don't meet a certain number of security-based criteria; for example, if extra arguments are passed, or if the directory with the CGI program is writeable by other users, then the query request is rejected. For servers where a high level of security is required, it's better not to run the server as root at all (for example, starting the daemon by a user like *nobody* on a high-numbered port). Even compiled code can potentially be exploited if the source code is freely available over the Internet—potential loopholes or coding errors might be exploited by an enthusiastic competitor or a bored hacker. This might seem like overkill, but when our businesses rely on providing reliable, Web-based information services, such precautions are mandatory.

Further reading

The definitive guide to CGI security can be found at:

www.w3.org/Security/Faq/www-security-faq.html

The best guide for general security issues for Solaris systems can be found at:

www.sunworld.com/common/security-faq.html

This FAQ contains a section "How do I make my Solaris Web server more secure?," which should be read by anyone considering running a Web server under Solaris. CERT advisories can be found at:

www.cert.org

CERT also maintains mailing lists so that you can be notified of any current advisories as soon as they're released.

SOLARIS UTILITIES

Shell toolbox 101, part 2

by Jeff Forsythe, Sr.

Last month, in part 1 of this article, you'll recall that we were building an editing library and we gave you the `logmsg` function and the `header` function to chew on. Hopefully you've put them to use already. This month, in part 2 of the article, we'll finish the library. So, let's dive right in.

Shell scripts

Our plan will add shell.sh, shown in **Listing A**, and file.sh to your toolbox next. They're similar in nature, but have a few differences. We'll demonstrate shell.sh and leave file.sh to you. Then, you'll build your edit.shlib because you can finally get some functionality out of the previous tools as a unit. You'll want to build vc.sh, shown in **Listing B** on page 6, quickly because it's the Easter Egg of the entire library. And finally, if you're so inclined, you'll want to build the affects.sh program (remember that this is vaporware for the author who would appreciate if you could email a copy of your affects.sh if you build one). Now that we have the plan down, let's get to work.

shell.sh

The next tool is the shell.sh program. It still could use some work, but has been so handy for six years and is such an integral part of the edit.shlib that it must be presented as-is.

The shell starts by obtaining all the pertinent information required to create the document header: Name, Author, Purpose, etc. Don't forget to change the `AUTHOR=` lines to default to your initials. The shell then creates a header and sends you to vc.sh. This program, like most of the others, is a simple, easy-to-follow utility. With minor modifications, it will work by itself. In fact, if you look closely at the weird way vc.sh is called, you'll probably guess that it was an after-thought (of about six years).

For your sake, spend the time to type in vc.sh rather than editing shell.sh to work by itself; it's worth it. Finally, you'll notice that the header contains starting and ending lines of 57 pound signs (#). This, you'll recall from last month, is used by header.fnc and must be exactly 57 in order to work. Now you can put shell.sh to work for the first time.

Creating internal documentation

For those of you who enjoy documenting after a program is written, or who wish to document your previously developed scripts, you can move the original shell to a temporary name, run shell.sh with the correct name, and then read in the temporary name you just created. This will create the internal documentation and get you where you need to be. Huh? To demonstrate, you should use this method to update logmsg.fnc and header.fnc (both were presented in part 1 of the December 1998 issue) to add the proper header documentation to them.

First, move logmsg.fnc to logmsg.nd (nd for no docs). Then, run shell.sh, logmsg.fnc, and answer the prompts. When you're ready to edit, just read in the logmsg.nd file (in vi, you type colon (:), the letter r for read, and the file name *:r logmsg.nd*). When you save and exit, you'll be saving logmsg.fnc with

Listing A: *shell.sh listing*

```
#!/bin/sh
############################################################
#
# SHELL:          shell.sh
# DATE WRITTEN: 08/24/92    JAF, Sr.
# DATE UPDATED: 09/18/98    JAF, Sr. v4.01
#                Fixed typo
# PURPOSE:       Used to create shell scripts
# USAGE:         shell.sh shell-name[.sh]
# FLAGS:         None
# ARGUMENTS:     shell-name
# RETURNS:       0 - Nominal
#                 1 - Invalid number of arguments
#                 2 - Attempting to create a
#                   file that exists
# CALLS:         ${FNCDIR}/logmsg.fnc
#                 ${SHELLDIR}/vc.sh
#                 ${SHELLDIR}/header.sh
# CALLED-BY:     N/A
# ERRATA:        None
# LIMITATIONS:   None
#
############################################################

. ${FNCDIR}/logmsg.fnc

logmsg shell.sh S Started ${LOGDIR}/shell.log

if [ $# -ne 1 ]
then
   echo "USAGE: shell.sh shell-name[.sh]"
   logmsg shell.sh E "Invalid number of
       arguments $#" ${LOGDIR}/shell.log
   logmsg shell.sh F Finished ${LOGDIR}
         /shell.log
   exit 1
fi

VERNAME="${1}"

if [ -s ${VERNAME} ]
then
   echo "${VERNAME} exists, use: vc.sh
          ${VERNAME} - exiting"
   exit 2
fi

if [ `echo ${VERNAME} | grep -c .sh` -eq 0 ]
then
        VERNAME="${VERNAME}.sh"
fi

SHELLNAME=${VERNAME}$$
```

```
echo "Purpose: \c"
read PURPOSE

AUTHOR="Inside Solaris Reader"
echo "Author's Initials [${AUTHOR}]: \c"
read AUTHOR

if [ "${AUTHOR}" = "" ]
then
        AUTHOR=" Inside Solaris Reader"
fi

echo "#!/bin/sh" >> ${SHELLNAME}
echo
     >> ${SHELLNAME}
echo "#" >> ${SHELLNAME}
echo "# SHELL:          ${VERNAME}" >>
   ${SHELLNAME}
echo "# DATE WRITTEN: `date +%m/%d/%Y`
   ${AUTHOR}" >> ${SHELLNAME}
echo "# DATE UPDATED:" >> ${SHELLNAME}
echo "# PURPOSE:        ${PURPOSE}" >>
   ${SHELLNAME}
echo "# USAGE:          ${VERNAME}" >>
   ${SHELLNAME}
echo "# FLAGS:          None" >> ${SHELLNAME}
echo "# ARGUMENTS:     None" >> ${SHELLNAME}
echo "# RETURNS:       0 - Nominal" >>
   ${SHELLNAME}
echo "#                 1 –
   Invalid number of arguments" >> ${SHELLNAME}
echo "# CALLS:          N/A" >> ${SHELLNAME}
echo "# CALLED-BY:     N/A" >> ${SHELLNAME}
echo "# ERRATA:         None" >> ${SHELLNAME}
echo "# LIMITATIONS:  None" >> ${SHELLNAME}
echo "#" >> ${SHELLNAME}
echo "########################################"
     >> ${SHELLNAME}
echo " " >> ${SHELLNAME}
echo " " >> ${SHELLNAME}

logmsg shell.sh M "vc.sh -tmp ${SHELLNAME}
   ${VERNAME}" ${LOGDIR}/shell.log

vc.sh -tmp ${SHELLNAME} ${VERNAME}

chmod +x ${VERNAME}

rm -f ${SHELLNAME}

logmsg shell.sh F Finished ${LOGDIR}/shell.log

exit 0
```

documentation. Now do the same thing with header.fnc. When you're done, you can remove the *.nd files (but make sure you don't have anything with this ending prior to this exercise).

The file.sh program is nothing short of a modified shell.sh. In order to create it, you run the shell.sh command with file.sh as the argument. Then, after answering the header prompts, you read in shell.sh and modify the header and the header that it creates. The purpose of file.sh is to do the same thing for non-shell script text files as shell.sh does for scripts—namely, to create a documentation header. In the interest of column space, this is left as an exercise for you.

Listing B: *vc.sh Listing*

```
#!/bin/sh
###########################################################################
#
# SHELL:          vc.sh
# DATE WRITTEN:   03/02/1998    JAF, Sr./rlh
# DATE UPDATED:   04/10/1998    JAF, Sr. v2.01
#                 Added -tmp flag
# PURPOSE:                Implement file version numbering.
#                 Taken from "Sys Admin" Vol. 3,
#              No. 5,
#                 Sept/Oct 1994 and HIGHLY MODIFIED!
# USAGE:          vc.sh [-tmp file] file [file...]
# FLAGS:          -tmp file: temporary file
# ARGUMENTS:              file [file ...]
# RETURNS:                0 - Nominal
#                 1 - Invalid number of arguments
#                 2 - File sums do not match
#                 3 - Error creating backup of
#                original file
# CALLS:          /usr/bin/vi
#                 ${FNCDIR}/logmsg.fnc
#                 ${UTILDIR}/mode.sh
# CALLED-BY:              N/A
# ERRATA:         None
# LIMITATIONS:            None
#
###########################################################################

. ${FNCDIR}/logmsg.fnc

# Check to ensure that at least one input
#     option was entered.
if [ $# -lt 1 ]
then
   echo "Usage:  vc.sh [-tmp file] file
      [file ...]"
   exit 1
else
   if [ "$1" = "-tmp" ]
   then
      if [ $# -lt 3 ]
      then
         echo "Usage: vc.sh [-tmp file] file
          [file...]"
         exit 1
      fi
      USETMP="TRUE"
      TMPFILE=$2
      shift
      shift
   else
      USETMP="FALSE"
   fi
fi

# Check for ${HOME}/.vc.cfg
if [ -s ${HOME}/.vc.cfg ]
then
        MAXVER=`grep MAXVER ${HOME}/.vc.cfg |
           awk '{print $2}'`
        VERDIR=`grep VERDIR ${HOME}/.vc.cfg |
           awk '{print $2}'`
else
        MAXVER=${MAXVER:5}
        VERDIR=${VERDIR:.}
fi

LOG=${VERDIR}/vc.log

# loop for each file in the argument list
for FILE in $*
do
   # Set DIR
   DIR=`dirname ${FILE}`

   # Set FILE
   FILE=`basename ${FILE}`

   # Set flag for ownership, group and mode
   MODEFLAG=FALSE

   # Get the last version present
   LASTVER="`ls ${VERDIR}/${FILE}\;*
       2>/dev/null | sed 's/.*;//g' |
       sort -rn | head -1`"

   if [ "${LASTVER}" != "" ]
   then
      # Check sum on original and highest
      # version
      if [ "`sum -r ${DIR}/${FILE} | awk
      '{ print $1 }'`" != \
                 "`sum -r ${VERDIR}/${FILE}\;
           ${LASTVER} | awk '{ print $1 }'`" ]
      then
                 banner "ERROR"
                 echo   "\n\n Highest version
                    and original file"
                 echo   "do not match.  This
                    could be a file out"
                 echo   "of sync or they
                    could be two different"
                 echo   "files from different
                    directories."
                 exit 2
      fi
      # Increment the version number.
      VER=`expr ${LASTVER} + 1`

      if [ ${VER} -gt ${MAXVER} ]
        # Do not exceed MAXVER copies.
      then
         cp ${VERDIR}/${FILE}\;1
           ${VERDIR}/${FILE}\;tmp

         # Start with 2 so that 2 replaces 1,
         # 3 replaces 2, etc.
         COUNT=2
         while [ ${COUNT} -le ${LASTVER} ]
         do
            LESSONE=`expr ${COUNT} - 1`

            mv ${VERDIR}/${FILE}\;${COUNT}
              ${VERDIR}/${FILE}\;${LESSONE}

            COUNT=`expr ${COUNT} + 1`
         done
```

Listing B: *continued*

```
        VER=${MAXVER}
        LASTVER=`expr ${LASTVER} - 1`
     fi

     ORIG="${VERDIR}/${FILE};${LASTVER}"
     NEWFILE="${VERDIR}/${FILE};${VER}"

     MODE=`ls -l ${DIR}/${FILE} ¦
        awk '{print $1}'`
     OWN=`ls -l ${DIR}/${FILE} ¦
        awk '{print $3}'`
     GRP=`ls -l ${DIR}/${FILE} ¦
        awk '{print $4}'`
     MODEFLAG=TRUE
  else
     # This is a new file - start the
     #version at 1.
     VER=1
     ORIG="${DIR}/${FILE}"
     NEWFILE="${VERDIR}/${FILE};${VER}"

     # Check to see if the file REALLY
     #EXISTS.
     if [ ! -f ${ORIG} ]
     then
        touch ${ORIG}
     else
        cp ${ORIG} ${VERDIR}/${ORIG}\;0
     fi
  fi

  # Copy the last edit file to the new
  # version for editing.
  if [ "${USETMP}" = "TRUE" ]
  then
     ERR=`cp ${TMPFILE} ${NEWFILE} 2>&1;
       echo $?`
  else
     ERR=`cp ${ORIG} ${NEWFILE} 2>&1;
       echo $?`
  fi

  # Check for copy error.
  if [ ${ERR} -ne 0 ]
  then
     ERR=`echo $ERR ¦ sed 's/.*: //g' 2>&1`
     echo "Your edit of ${NEWFILE} is aborted."
     echo "Attempting to access ${ORIG}
            has reported the following \c"
     echo "system error:\n\n\t${ERR}\n"
     echo "Aborting edit."
     exit 3
  fi

  echo "Loading ${NEWFILE} for edit ..."
  #     Load vi and edit file
  /usr/bin/vi + ${NEWFILE}
  if [ ${VER} -gt 1 ]
  then
     # Check to see if user actually
     # made changes.
     diff ${ORIG} ${NEWFILE} 2>&1 >/dev/null
     EDIT=$?
  else
     # This is the first edit - so don't run
     # diff command
     if [ -r ${NEWFILE} ]
     then
        EDIT=1
     fi
  fi

  if [ ${EDIT} -eq 1 ]   # The user DID make
    # changes
  then
     # Remove the pre-edit file and the
     # tmp copy.
     rm -f ${DIR}/${FILE} ${VERDIR}/
        ${FILE}\;tmp

     # Copy the newest file to the basename.
     cp ${NEWFILE} ${DIR}/${FILE}

     if [ "${MODEFLAG}" = "TRUE" ]
     then
        NEWMODE=`${UTILDIR}/mode.sh ${MODE}`
        chmod ${NEWMODE} ${DIR}/${FILE}
        chown ${OWN} ${DIR}/${FILE}
        chgrp ${GRP} ${DIR}/${FILE}
     fi


     logmsg $0 M "${LOGNAME} changed
       ${DIR}/${FILE}" ${LOG}
  else
     logmsg $0 M "${NEWFILE} was not edited.
          Restoring to ${ORIG}" ${LOG}
     # Remove the higher version because
     # NO changes were made.
     rm ${NEWFILE}

     # Move files back to original names.
     if [ -s ${VERDIR}/${FILE}\;tmp ]
     then
        # MAXVER -1, so that we can
        # move the files back up until
        # 1 becomes 2
        # and then we move the temp file
        # in as 1.
        COUNT=`expr ${MAXVER} ⫫ 1`
        while [ ${COUNT} -ge 1 ]
        do
           PLUSONE=`expr ${COUNT} + 1`


           mv ${VERDIR}/${FILE}\;${COUNT}
            ${VERDIR}/${FILE}\;${PLUSONE}


           COUNT=`expr ${COUNT} - 1`
        done
        mv ${VERDIR}/${FILE}\;tmp
          ${VERDIR}/${FILE}\;1
     fi
  fi
done


exit 0
```

Version control

That brings us to version control. How many times have you made changes to a script only to find out a day or two later that your changes don't work and you want to roll back to the previous version? With vc.sh in place and used properly, you'll have this option at your fingertips.

The version control script, vc.sh, first checks to see if there's a previous version control file. If there isn't, the file is copied to a backup and given a semicolon zero extension (filename;0). If the version control file *does* exist, a copy is made to the next higher number. The only exception to this rule is when the numbers exceed the maximum number of version control files. The maximum is set to five by default and can be controlled through an environment variable (`MAXVER`) or through the configuration file vc.cfg (which was created with file.sh).

If it exists, vc.cfg is kept in the user's home directory and is specific to that user. If the maximum number is reached, the lowest number (filename;1) is moved to a temporary name and each subsequent higher number is moved down to the next lower number (for example, filename;2 becomes filename;1). After the version control routine is completed, the editor is called.

Should the user exit without making any changes, the files are all reset to their original positions prior to vc.sh being called. That's why we keep filename;1 in a temporary location. If changes were made, then the temporary copy of filename;1 gets removed (if it exists).

Storing backup files

There are several schools of thought about how to store the backup files. We'll describe each thought, but for this article, we'll implement the third school of thought—storing the backup files in the same directory.

Backup directory

One school of thought says to keep the backup files in a special directory set up just for backup files, thus making them easier to back up separately from the nightly backup (the author's personal favorite—but slightly more difficult to implement).

The biggest drawback to this idea is that if you have files in multiple locations with the same name, they'd get mixed up with one another in the backup directory. However, vc.sh is aware of this possibility and performs a checksum on the latest file (highest number) and the file being edited. If they don't match, the user is warned that there's a problem. This method is also implemented as a means of checking for someone editing the file without the use of vc.sh.

Another problem with this method is that the backup directory's write permissions need to be opened up for everyone. This, obviously, isn't a good idea.

Separate file system

Another school says to create a separate file system that mimics the original under the mount point /vc. This option is an offshoot of the first in that it creates a mimic file system. For instance, if you were going to edit /etc/hosts, the backup file would be stored in /vc/etc/hosts;1. It doesn't take long to see the drawbacks in this idea.

First, the file system must get copied to the backup area. This takes up overhead unnecessarily and takes up a disk slice for the file system. The script is written using the `VERDIR` variable, which can be set in the .vc.cfg configuration file. This variable determines the location of the backup files.

Same directory

The final school says to keep the backup files in the same directory as the original. That's the easiest to implement and the one we'll show in this article.

If you want to perform a separate backup of the version control files, you search for files that end in a semicolon and a number. It's also nice to look only in /etc for hosts* rather than looking in /etc and one or more other locations for backups. This particular shell is an Easter Egg in disguise. Once you have it implemented, you'll know within a day or so why it's so useful.

Moving back to an older version after an editing session is as easy as remove and copy. Remove the highest numbered version and copy the next higher version to the edit name. If you don't like edit number 5 of the /etc/hosts file, remove number 5 and copy number 4 to /etc/hosts.

A very important point about vc.sh is that it uses the semicolon in the filename. The semicolon is a special character because it's also a statement separator in Solaris. So it's important to escape the semicolon when using it. That is to say, when you want to copy or remove a file with the semicolon, you need to use a backslash (\) prior to the semicolon. For example, rm /etc/hosts\;5 and cp /etc/hosts\;4 /etc/hosts. If you leave the backslash out of the `rm` statement (rm /etc/hosts;5), Solaris will try to remove /etc/hosts and then execute a program called 5. The `cp` statement would give an error because you're trying to copy /etc/hosts to nothing and then execute a program called 4 with a command line argument of /etc/hosts. Be careful when using the semicolon in filenames! We could use some other character, but the semicolon looks nice in an ls output.

Holding it all together

Last month, we promised you some string and rubber bands to hold this all together. The vc.sh uses a configuration file (the string) and calls another script called mode.sh (the rubber band) shown in Listing C. Listing D, on page 10, shows an example vc.cfg configuration file—which was created using file.sh. You can change the settings for each user and place this in their home directory. The VERDIR is the setting you can use to save the files in a location other than the current directory, as previously stated. Just give the full path to the verdir of your choice such as ${HOME}/verdir.

The `vc.sh` script also calls a program called `mode.sh`, which isn't included in the library but is necessary for vc.sh to work. It accepts the string mode of a file and returns the numeric mode. You can see the example in the header. This utility doesn't accept all string modes, but it works on almost every file.

Send your updates to the author. This script might well become a function after you see it's utility. Many programs can use `mode.sh` to set the mode of a file.

affects.sh

We discussed the vaporware affects.sh earlier. The idea is simple, but implementation is a challenge. If you're a weak-in-the-knees shell script programmer, read no further, skip to the next article. The affects.sh program uses the header function from header.fnc to find the CALLS and CALLED BY sections, and then attempts to determine if the changes you made to the current script have any affect on the scripts (if any) in these sections. If so, `affects.sh` notifies and asks if you want to edit the affected files. You could conceivably use this to test program source code as well if the location of the code is given.

You're a hard-core programmer if you implement this concept. Of course, this is a recursive procedure because editing changes made to the *affected* script may affect others. Therefore, affects.sh must trace the CALLED BY linage first and then the CALLS section next. As you can see, this can get really deep, fast.

Listing C: *mode.sh Listing*

```
#!/bin/sh
#####################################################################
#
# SHELL:          mode.sh
# DATE WRITTEN:   03/23/1998   JAF, Sr.
# DATE UPDATED:
# PURPOSE:        Translate string mode into
#                 numeric
# USAGE:          mode.sh
# FLAGS:          None
# ARGUMENTS:      String mode (ex. drwxrwxrwx,
#                 -rwsr-xr-x)
# RETURNS:        Numeric mode (ex. 777, 4755)
# CALLS:          N/A
# CALLED-BY:      N/A
# ERRATA:         None
# LIMITATIONS:    Doesn't handle all modes
#
#####################################################################

get_num()
{
   # Other modes can be added to the case
   # statement as needed
   # setuid bits are set to 11-17
   # because we couldn't use 1-7
   # and we needed to have a unique number
   # for them.  This
   # works out well when used below.
   case $1 in
        "---") NUM=0;;
        "--x") NUM=1;;
        "-w-") NUM=2;;
        "-wx") NUM=3;;
        "r--") NUM=4;;
        "r-x") NUM=5;;
        "rw-") NUM=6;;
        "rwx") NUM=7;;
        "--s") NUM=11;;
        "-ws") NUM=13;;
        "r-s") NUM=15;;
        "rws") NUM=17;;
   esac
}

MODE=$1

USER=`echo ${MODE} | cut -c2-4`
get_num ${USER}
if [ ${NUM} -gt 7 ]
```

Listing C: *continued*

```
then
        OUT=`expr 3000 + ${NUM} \* 100`
else
   OUT=`expr ${NUM} \* 100`
fi

GROUP=`echo ${MODE} | cut -c5-7`
get_num ${GROUP}
if [ ${NUM} -gt 7 ]
then
        OUT=`expr ${OUT} + 1900 + ${NUM} \* 10`
else
   OUT=`expr ${OUT} + ${NUM} \* 10`
fi

OTHER=`echo ${MODE} | cut -c8-10`
get_num ${OTHER}
OUT=`expr ${OUT} + ${NUM}`

echo ${OUT}
```

Listing D: *Example .vc.cfg configuration file*

```
# .vc.cfg
# configuration file for vc.sh for Inside Solaris Reader
MAXVER 5
VERDIR .
```

The hardest part will be determining if a change made to the current script has any affect on others. This can be accomplished by checking for an environment variable or file changes that are in both scripts, and by checking for differences in the flags or command line arguments of the scripts. Do you think that's all it will take? But isn't that enough to make you feel tingly all over? Version 1.0 should just list the files in the CALLS and CALLED BY sections and ask if the user wishes to edit them. Perhaps it will store them in a temporary file for later use. Then Version 2.0 will go full-blown. Feel free to code this script and email it to the author. Or, send it to *Inside Solaris*, as an article for an upcoming issue.

Summary

The edit.shlib is now complete, well almost. In Solaris, as in all UNIX development, everything is a work in-progress. As for the library, you now have many tools at your fingertips that put you miles ahead of your peers who don't read *Inside Solaris* (shameless plug). If you begin to use the library religiously, you'll see just how useful it can be.

SOLARIS TUNING

Improving server apps with scheduling under Solaris

by Abdur Chowdhury and Andy Spitzer

With system complexity growing each day, many machines are performing many different tasks simultaneously. This sharing of resources can affect the performance of your system. Understanding how the operating system schedules your process can help you architect better systems.

Solaris is a multi-tasking pre-emptive operating system. The operating system is responsible for scheduling when each process runs. When a process has reached the end of its allotted time slice, or blocks for some reason, Solaris saves its state and runs the next runnable process in the process queue. This gives the appearance of many different programs running simultaneously on a single processor. Solaris 2.6 and greater supports two usable process scheduling classes: real-time and time-sharing.

In this article, we'll describe the concept of process scheduling, the major differences between real-time and time-sharing scheduling, and the Solaris API to process scheduling ma-

DOWNLOAD
ftp.zdjournals. com/sun

nipulations. We'll also provide a simple example of its use to improve server performance.

Process scheduling

Process scheduling has one basic goal: to make the system more productive. To achieve this goal, a general-purpose operating system must balance the following different needs:

- Allot a fair share of CPU time to each process.
- Try to keep the CPU busy 100 percent of the time.
- Minimize response time for time sensitive processes.
- Minimize total job completion time.
- Maximize total number of jobs per time unit.

To achieve this balance, Solaris makes some assumptions about the type of processes that it's running. Therefore, the default scheduler may not always suit your application. In the next section, we'll present a brief background of several scheduling algorithms and their implementations.

Solaris uses a two-level thread implementation, where threads within a process are scheduled on a virtual processor known as an LWP (Light Weight Process). LWPs, in turn, are scheduled on the physical processors. When we talk about *process* scheduling below, we really mean LWP scheduling. Threads within a given process can be *bound* to an LWP, and it's the scheduling characteristics of the LWP that we describe below.

Scheduling algorithms

The research community has developed many optimal scheduling algorithms, but all algorithms are only optimal for certain workloads. There's no single solution that fits every system need. General-purpose operating systems face the problem of developing schedulers that are general enough to solve most needs, yet extensible enough to handle specific workloads.

There are many different algorithms for choosing the next process to run. Two of the most common algorithms, both used by the Solaris scheduler, are Round Robin (RR) and First-In-First-Out (FIFO).

FIFO runs each process until its completion then loads the next process in the queue. FIFO can have negative effects on systems where processes run for an extended time.

Round Robin allows each process at a given priority to run for a predetermined amount of time. When the process has run for the allotted time quantum, or if a higher priority process becomes runnable, the scheduler halts it and saves its state. It's then placed at the end of the process queue and the next process is started. Note that this may be the most optimal solution, if the time quantum is longer than the average runtime of a process. However, context switching between different processes has a price: We must also consider the overhead (context switching time) of changing processes.

With the need for different algorithms at different times, operating system developers created multilevel queue scheduling. Multilevel queue systems are simply several algorithms used simultaneously. Each queue is assigned a priority over the next queue. The scheduler simply starts at the highest priority queue, implements that queue's algorithm until no runnable processes remain, and then proceeds to the next priority queue. One queue could use FIFO while the other uses RR.

Solaris implements two scheduling classes—time-sharing and real-time. The time-sharing class includes interactive processes as well as time-sharing processes. These two types of processes share the same scheduling table as described by the man page for ts_dptbl(4). Processes in the time-sharing class have a changing priority based on factors such as if they previously exceeded their assigned time quantum, or if they're prevented from running for a long period of time by higher priority processes, or if they just sleep a lot. This change in priority, based on previous behavior, is what gives Solaris wonderful interactive response even under heavy loads. Batch-type jobs with high CPU demands tend to have lower priority because they keep exceeding their quantum. Interactive type jobs, like Web browsers, tend to have higher priority because most of the time they're blocked (sleeping), awaiting user input.

However, important processes, such as database servers or Web servers, may have the same characteristics as a batch job, and end up with a lower priority than the user playing Doom. To solve this dilemma, The

Solaris scheduler also provides a real-time scheduling class as of version 2.6. To provide soft real-time scheduling requirements, the scheduler must provide several services:

- Priority Scheduling—Scheduling based on process class.
- Real-Time processes won't change priority over their lifetime—A real-time process will always run before a time-sharing process.
- Processes must be interruptible during system calls and I/O operations—The kernel must be pre-emptable.

Solaris provides all these services. The Solaris kernel is entirely pre-emptable, so process scheduling doesn't need to wait for a safe time to switch between processes. Also, when implementing a real-time scheduler, the situation where a higher priority process is blocked waiting for a resource locked by lower priority process, called *priority inversion*, can occur. Solaris avoids priority inversions by implementing priority inheritance protocol. A process that owns a resource runs at the priority of the highest priority process that's awaiting that resource. This means that lower priority processes run at a higher priority until they release the lock or shared resource requested by a higher priority process. When the lock is released, the higher priority process becomes runnable and pre-empts the current process.

What does all this mean? It means the algorithm selected can affect the performance of your application, and choosing the right algorithm can improve your systems' performance. There are many different evaluation techniques such as:

- Deterministic modeling
- Queuing Models
- Simulation
- Implementation

The evaluation of your situation with the above methods is beyond the scope of this article. Those methods of evaluation may be needed for some systems and Jain as a good source to start your evaluation [Jain 91].

Solaris provides two scheduling algorithms for its real-time class—RR and FIFO. The FIFO processes will proceed until completion, unless they're pre-empted by a higher priority process or interrupted by a signal. Round-Robin processes will execute for a given time quantum if not preempted by a higher priority process or interrupted by a signal. By changing the scheduling class of your server applications, you can be guaranteed that they will run before other applications, thus improving responsiveness. Next, we'll briefly describe the API that Solaris provides for process scheduling manipulation and modification.

Process scheduling API under Solaris

Solaris, as a general purpose OS, must be configurable and even modifiable. To attain this goal, Sun provides several utilities to view the default scheduling classes and change the process priorities and quantum. We've listed several of these utilities here. Our main goal is to provide a method of viewing or changing a process' scheduling parameters. All of these utilities use a kernel call named *priocntlsys*. Below are two such utilities:

- `dispadmin(1M)`—Displays or changes process scheduler parameters while the system is running.
- `priocntl(1)`—Displays or sets scheduling parameters for specified process.

System administrators that want to run a process at a different priority or scheduling class can use the -e option to `priocntl(1)` to execute a command. Children that are created by that command will inherit the priority and scheduling class from the parent process. For example, to set your Web server process to the real-time class with priority 20:

```
priocntl -e -c RT -p 20 httpd
```

Solaris also provides mechanisms to replace the current real-time and time-sharing scheduling parameters with your own. The use of these functions is beyond the scope of this article, but we included them for completeness and for further reading. Please see the man pages for further information.

- `rt_dptbl(4)`—Real-time dispatcher parameter table
- `ts_dptbl(4)`—Time-sharing dispatcher parameter table

Developers who need to change the priority of their process dynamically will either use the `priocntl` function or use the POSIX real-time library calls like `sched_setscheduler`. Following are several such process-scheduling calls to control how the OS will schedule your process:

- `priocntl(2)`—Controls the scheduling of an active light-weight process LWP.
- `priocntlset(2)`—Changes the scheduling properties of an existing process.
- `sched_setscheduler(3R)`—Sets the scheduling policy and scheduling parameters of a process.
- `sched_setparam(3R)`—Sets the scheduling parameters of a process.

Improving responsiveness: Example server

To show the usage of the API and the effects of scheduling on your process, we've written a small example. The application reschedules itself as a real-time process if it has root privileges; without root privileges, it will run as a normal time-sharing process. After rescheduling itself, the application executes a tight counter loop, to use cpu cycles, then the process sleeps for approximately 10 microseconds. When the sleep is complete, we increment a counter. The value of the counter shows how many times the operation was accomplished. We check our counter every 10,000 microseconds, saving the results to a file. We've executed four experiments: time-sharing with no load, time-sharing with load, real-time round robin with no load, and finally real-time round robin with load. To simulate load on our system, we've written a program that loops in a tight `for` loop to load the system. See the Listings A and B, on page 14, for load.c and load-generator.c, respectively.

Figure A, also on page 14, shows a chart with four experiments. The y-axis is the number of operations accomplished in the allotted time. The x-axis is the number of the experiment; we ran 10 for each set. The time-sharing experiments show a degradation of operations from the no-load to the load experiments, as we'd expect. The real-time experiments don't show a degradation in performance for load or no-load. What these experiments do show is that real-time

Listing A: *We use load.c to schedule our processes for load testing.*

```
#define _REENTRENT
#define _POSIX_PTHREAD_SEMANTICS

#include <sys/times.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/uio.h>


#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <stropts.h>
#include <sched.h>
#include <errno.h>

#include <thread.h>
/*
     compile as:
     cc -g -o load load.c -lsocket -lnsl -lthread -lposix4
*/

/* gloabal's */
pthread_mutex_t    g_mutex ;
int g_counter ;

struct work
{
     int ms ;
} ;

void * worker(void *arg)
{
     struct work *dog = (struct work *)arg ;
     int x ;

     for (;;)
     {
         /* the poll is to simulate work
                 count to a 100000 then sleep for X ms */
         for (x=0;x<100000 ; x++) { ; }
         poll(0, NULL, dog->ms) ;
         thr_yield() ;
         /* lock the global counter */
         pthread_mutex_lock(&g_mutex) ;
         g_counter++ ;
         pthread_mutex_unlock(&g_mutex) ;
     }
     return NULL ;
}

void prio()
{
     /* Increase our priority to Real Time Status */
     {
         struct sched_param param = {0} ;
         int res ;

         param.sched_priority =
```

Listing A: *continued*

```
sched_get_priority_min(SCHED_RR) + 10 ;
        res = sched_setscheduler(0, SCHED_RR, &param) ;
        if (res < 0)
        {
                fprintf(stderr, "Cannot
sched_setscheduler(SCHED_RR), %d %s\n",
                        errno, strerror(errno)) ;
                fflush(stdout) ;
        }
    }
}


main()
{
    struct work dog1 = {10} ;
    FILE *output ;
    int x ;
```

```
      prio() ; /* Set up the priority for this process */

      thr_create(NULL, NULL, worker, (void *)&dog1, THR_BOUND,
NULL) ;

      output = fopen ( "data.txt", "w+" ) ;
      for (x=0; x < 10; x++)
      {
          poll(0, NULL, 10000) ;
          pthread_mutex_lock(&g_mutex) ;
          fprintf (output, "%d\n", g_counter ) ;
           fflush (output);
          g_counter=0 ;
          pthread_mutex_unlock(&g_mutex) ;
      }
      fclose ( output ) ;
}
```

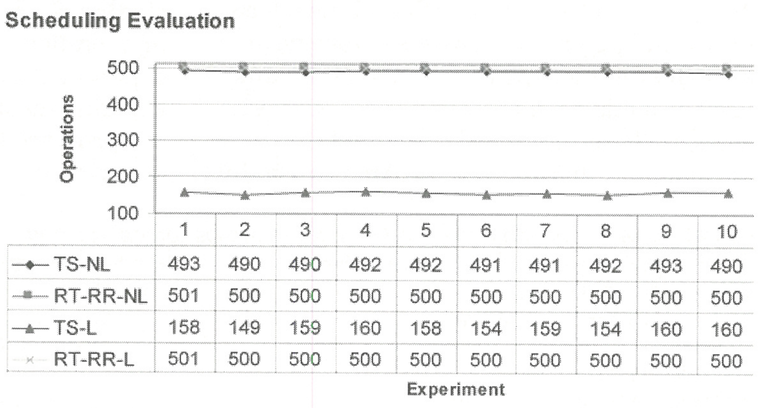
Listing B: *We use load-generator.c to create a CPU-intensive process.*

```
#include <stdio.h>

int main (int argc, char **argv)
{
    int x ;
    for (x=0; ; x++)
    {
        if (!x%100000) sleep ( 1 ) ;
    }

}
```

Figure A



	1	2	3	4	5	6	7	8	9	10
TS-NL	493	490	490	492	492	491	491	492	493	490
RT-RR-NL	501	500	500	500	500	500	500	500	500	500
TS-L	158	149	159	160	158	154	159	154	160	160
RT-RR-L	501	500	500	500	500	500	500	500	500	500

The results of our scheduling experiments evaluation.

scheduling can improve your applications performance when your system has other processes competing for the cpu.

As a side note, please be careful when rescheduling your process to the real-time class. Mistakes will likely result when you turn off the machine and reboot it, since all other process will run at a lesser priority and will never run if your program isn't responding. In this section, we've shown that Solaris has utilities and methods for controlling the scheduling of your processes.

Conclusion

Understanding your operating system's scheduling can help you improve your applications' performance. There are different types of scheduling algorithms, and picking the right one for your situation is important.

In this article, we presented several different algorithms and how they work, and we also presented a general overview of what Solaris provides to modify your applications scheduling. We presented an example program showing the benefits of rescheduling your process to the real-time class. (For additional reading see [Jain 91] Raj Jain, "The Art of Computer Systems Performance Analysis," John Wiley & Sons, Inc. 1991, ISBN 0-471-50336-3.) Solaris is a great general-purpose operating system, and understanding what it can do for you will improve the performance of your applications.

PatchDiag Tool

by Werner Klauser

The PatchDiag Tool, found at sunsolve.sun.com/patchdiag/ or on Sun's patch CD-ROM, is a diagnostic tool that enables system administrators to examine a profile of the patches installed on their Solaris system against the most current profiles available from Sun Microsystems with respect to:

- Latest revisions
- Recommended patches
- Security patches
- Other patches relevant to the software environment

The results of PatchDiag Tool can help a system administrator quickly determine and analyze the need for

About our contributors

Abdur Chowdhury is the staff scientist for Group Logic in Arlington, Virginia. He's also working on his Ph.D. in computer science on distributed systems. Also, he has authored many papers on process migration, fault tolerant routing protocols, and information retrieval topics. You can reach Abdur at abdur@grouplogics.com.

Jeff Forsythe, Sr. started programming in 1978. He's worked with Solaris and other UNIX flavors in retail, banking, manufacturing, and currently with the federal government. He welcomes your comments, code fixes, administrative scripts, etc. You can reach Jeff via email at forsythe@ tuscan.net.

Werner Klauser is an independent UNIX consultant working near Zurich, Switzerland. While not paragliding, enjoying his daughters, or roaring around on his Harley chopper, he can be reached by email at klauser@klauser.ch or on the Web at www.klauser.ch.

Andy Spitzer is a developer of Centigram's new Solaris-based enhanced telephony services. Andy has several patents in the telecommunications industry and has designed real-time and distributed systems. You can reach Andy at woof@centigram.com.

Rob Thomas is an aspiring blues guitarist earning his living as a Principal Engineer for Dimension Enterprises. He can be contacted through email at rthomas@dimension.net.

Paul A. Watters is a research officer in the Department of Computing at Marcquarie University, Australia. He has developed a number of numerically intensive simulations (for example, neural networks) using the Solaris development environment. You can reach Paul via email at pwatters@mpce.mq.edu.au.

INSIDE SOLARIS
Tips & techniques for users of SunSoft Solaris

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 email at zdjcr@zd.com.

Editorial

Editor	Garrett Suhm
Assistant Editor	Jill Suhm
Copy Editors	Rachel Krayer
	Christy Flanders
	Taryn Chase
Contributing Editors	Abdur Chowdhury
	Jeff Forsythe, Sr.
	Werner Klauser
	Andy Spitzer
	Rob Thomas
	Paul A. Watters
Print Designers	Ian Caspersson
	Lance Teitsworth

General Manager	Jerry Weissberg
Editor-in-Chief	Joan Hill
Editorial Director	Michael Stephens
Managing Editor	Kent Michels
Circulation Manager	Renee Costanza
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 email 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 92880
Rochester, NY 14692

Copyright

Copyright © 1999, 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.

SunSoft Technical Support

(800) 786-7638

PERIODICALS MAIL

C:7661905 00002096 04/00
[redacted]
CLINTON TOWNSHIP MI 48035-4218

Please include account number from label with any correspondence.

applying additional patches or newer revisions of existing patches. PatchDiag provides only patch profile information. It doesn't deliver or apply any patches. If a system administrator wants to acquire or apply additional patches to a system, he or she should use their normal Sun authorized service channel, or could access patches via the Web at **sunsolve.sun.com/private-cgi/patchpage.pl**.

Description

This tool is a Perl-compiled script compatible with all systems running a Solaris 2.3 or later environment and with all patches installed in Solaris patch packet format. As later versions of Solaris are released, this tool may not be supported any longer by earlier versions of Solaris.

Patches indicated as "Recommended" patches are the most important patches. They prevent the most critical system, user, or security-related bugs that have been reported and fixed to date. Patches not listed as recommended by PatchDiag should be used if needed. Some patches listed in this report can have certain platform-specific or application-specific dependencies and thus might not be applicable to your system. It's important to carefully review the README file of each patch to fully determine the compatibility of any patch with your system.

Usage

Invoke the PatchDiag Tool from a command line. PatchDiag with no options uses the default variables you specified during system installation. It produces a summary status report that lists:

- Installed patches
- Uninstalled recommended patches
- Uninstalled security patches `patchdiag -l` produces a more extensive audit report that includes all patches available pertinent to software installed on the system. It lists:
- Installed patches
- Uninstalled recommended patches
- Uninstalled security patches
- Other related uninstalled patches

Summary

Using this freely available tool informs you of patches that Sun recommends. Whether the patch belongs to the recommended or security patch category, it's very possible that you can find the patch that would alleviate your problem. This possibility is very interesting as the year 2000 and its potential problems approach. Just remember that applying pages to Solaris (or any operating system) can cause unexpected problems. Always make sure that your system is backed up and that you have a recovery plan in case you need to return to your pre-patch configuration. Even better, experiment on a development machine or a personal workstation before applying patches to your production server.

Coming up

- Network routing with Solaris
- Setting up PPP for dial-up connectivity
- Using Solaris Jumpstart to automate your Solaris installations