81

AIX update

July 2002

In this issue

AIX Update

Published by

Xephon
27-35 London Road
Newbury
Berkshire RG14 1JL
England
Telephone: 01635 38342
From USA: 01144 1635 38342
E-mail: trevore@xephon.com

North American office

Xephon
PO Box 350100
Westminster, CO 80035-0100
USA
Telephone: 303 410 9344

Subscriptions and back-issues

A year's subscription to *AIX Update*, comprising twelve monthly issues, costs £180.00 in the UK; $275.00 in the USA and Canada; £186.00 in Europe; £192.00 in Australasia and Japan; and £190.50 elsewhere. In all cases the price includes postage. Individual issues, starting with the November 1999 issue, are available separately to subscribers for £16.00 ($24.00) each including postage.

AIX Update on-line

Code from *AIX Update*, and complete issues in Acrobat PDF format, can be downloaded from our Web site at http://www.xephon.com/aix; you will need to supply a word from the printed issue.

Editors

Trevor Eddolls

Disclaimer

Readers are cautioned that, although the information in this journal is presented in good faith, neither Xephon nor the organizations or individuals that supplied information in this journal give any warranty or make any representations as to the accuracy of the material it contains. Neither Xephon nor the contributing organizations or individuals accept any liability of any kind howsoever arising out of the use of such material. Readers should satisfy themselves as to the correctness and relevance to their circumstances of all advice, information, code, JCL, scripts, and other contents of this journal before making any use of it.

Contributions

When Xephon is given copyright, articles published in *AIX Update* are paid for at the rate of £170 ($260) per 1000 words and £100 ($160) per 100 lines of code for the first 200 lines of original material. The remaining code is paid for at the rate of £50 ($80) per 100 lines. In addition, there is a flat fee of £30 ($50) per article. To find out more about contributing an article, without any obligation, please download a copy of our *Notes for Contributors* from www.xephon.com/nfc.

Printed in England.

A review of recent AIX security exposures

INTRODUCTION

This article provides a review of the principal security vulnerabilities that have been shown to affect AIX between January 2001 and March 2002.

The review considers two components of AIX security. First to be considered are the vulnerabilities which directly affect the AIX operating system itself, and are therefore relevant to all readers of this journal. Second, we assess the vulnerabilities that have been shown to affect general Unix security (such as BIND and SSH) that can have an impact on AIX in some circumstances. Because these are sometimes tangential to AIX, their security is often overlooked, but they can provide a backdoor into a system. The exploits and vulnerabilities are reviewed in chronological order of discovery, beginning with the most recent.

The review concludes with an analysis of the types of exposure discovered, the lessons we have learned, and suggestions for the future.

In this article the terms hacker, cracker, and malicious user are used interchangeably to indicate an intruder with malicious intent, rather than using the purist definition of 'hacker' as a coder.

AIX VULNERABILITIES

The following exposures directly affect the AIX operating system and its immediate applications.

Double free bug in zlib compression library

In March 2002 it became clear that there was a bug in the zlib compression library. AIX 5.1 ships with open source-originated zlib that is used with the Redhat Package Manager (rpm) to install applications that are included in the AIX-Linux Affinity Toolkit. zlib (libz.a) is a shared library in AIX. AIX 5.1 is vulnerable to this exposure, but AIX 4.3.*x* does not ship with zlib. An updated rpm.rte install image for AIX 5.1 should be installed. This can be obtained from ftp://ftp.software.ibm.com/aix/freeSoftware/aixtoolbox/INSTALLP/ppc/.

Users who obtained zlib from the AIX-Linux Affinity Toolkit CD can obtain an updated zlib RPM from ftp://ftp.software.ibm.com/aix/freeSoftware/aixtoolbox/RPMS/ppc/zlib/

AIX loadable authentication modules

In December 2001 it became clear that there was an authentication vulnerability in AIX 5L loadable authentication modules. This revolved around a password checking flaw under integrated login in AIX. A remote user could log into a system with an invalid password if the system were configured to use loadable authentication modules, as used in integrated login methods. Using this exposure, a remote user could log into the system with root-level privileges. This vulnerability does not occur when using NIS or local password files (the default authentication mechanism).

IBM has provided a patch for AIX 5.1L (APAR IY26302).

Buffer overflow in System V-derived login

December 2001 saw another weakness begin circulating on the Internet, which exposed a vulnerability in applications using a System V-derived login for authentication to a system. Malicious hackers could exploit a flaw in the remotely exploitable buffer overflow in login derived from System V to gain root access to the server. This exposure affects AIX Versions 4.3 and 5.1.

Several implementations of login that are derived from System V allow a user to specify arguments such as environment variables to the process. A number of buffers are used to store these arguments. A flaw exists in the code used to check the number of arguments accepted. This allows the buffers to be overflowed.

On most systems, login is not suid (set user ID), so it runs as the user who called it. However, if login is called by an application that runs with greater privileges than those of the user (ie telnetd or rlogind), then the hacker can exploit this vulnerability to acquire these enhanced privileges. Furthermore, in the case of telnetd or rlogind, root access can be achieved remotely.

It is possible to limit your exposure to these vulnerabilities by using a router or firewall to restrict access to port 23/TCP (telnet) and port 513/TCP (rlogin). However, remember that this will not protect you against attackers within your network.

You could also consider restricting access to login. Do not use any programs that use a vulnerable login for authentication. Remember that some SSH applications can be configured to use login for authentication. If this configuration is selected, you will still be vulnerable.

Both AIX 4.3 and 5.1 are susceptible to this exposure. An emergency fix (efix), called 'tsmlogin_efix.tar.Z', is available for download from ftp://aix.software.ibm.com/aix/efixes/security

IBM has provided patches for AIX 5.1 (APAR IY26221), while the APAR for AIX 4.3 is pending.

Format string exposure in CDE ToolTalk

In September 2001 a format string vulnerability was found in the CDE ToolTalk database server (rpc.ttdbserver). This vulnerability affects AIX 4.3 and 5.1 and is revealed when a user makes an RPC open request to ttdbserverd that specifies a directory where the .ind or .var file are non-existent.

An intrusion can be performed remotely by a malicious hacker through the creation of relevant files by invoking a create RPC call. The user then deletes the .ind or .var file and makes an RPC open call to ttdbserver for these recently deleted files. If the pathname provided contains format characters, syslog() will interpret these, and allow the attacker to write an arbitrary memory address. With the relevant code, a malicious remote user can gain root privileges on the system, thereby compromising the integrity of the system and the attached network.

IBM has provided patches for AIX 4.3 and AIX 5.1 (APAR IY23846). Aside from the patches, there is no workaround that can minimize the vulnerability, other than disabling the CDE ToolTalk RPC database server.

Buffer overflow in lpd

In September 2001 it became clear that the AIX Line Printer daemon (lpd) contained several buffer overflow vulnerabilities that have the potential to allow a malicious remote user to gain root access. Furthermore, even if the hacker were unable to gain root access, they could still cause a system crash through denial of service (DoS). This exposure applies to AIX 4.3 and 5.1.

It should be noted that two of the three vulnerabilities found require the attacker's system to be listed in */etc/hosts.lpd*, or */etc/hosts.equiv*. The third requires that the malicious user has control over the Domain Name Server (DNS).

IBM provided fixes for AIX 4.3 (APAR IY23037) and AIX 5.1 (APAR IY23041). No workarounds are recommended other than disabling the line printer daemon until the APAR is installed.

Note: this exposure highlights the need to disable all unused daemon services. I would strongly recommend that you do this as a standard part of good security practice.

Buffer overflow in the libi18n library

During June 2001, it became clear that the libi18n library in AIX contains a function that is vulnerable to a buffer overflow through the LANG environment variable. This exposure affects AIX 4.3.*x* and 5.1.

The libi18n library located in the */usr/ccs/lib* directory ships as standard with AIX. An ordinary user has the ability to set the LANG environment variable to any value they choose. When this variable is set to a suitably formatted string and a program is run which uses the vulnerable library, the program will terminate abnormally. If this program is also setuid root (ie aixterm), a malicious local user could spawn a root shell and gain root privileges on the attacked system, compromising the system and its local network.

IBM provided the following fixes for AIX 4.3.*x* (APAR IY20867) and AIX 5.1.0 (APAR IY21309).

One work-around for this exposure is simply to make 'aixterm' non-suid. You need to be root to do this. Although ordinary users can still use the program, there may be some side-effects.

Root shell spawning through diagrpt

In addition, in June 2001 it was revealed that a malicious user could obtain AIX root privileges by exploiting a vulnerability in **diagrpt**. **diagrpt** is a diagnostic reporting command that ships with AIX. However, this command is shipped as suid, and is therefore executable by an ordinary user.

In this way an ordinary user is able to set the DIAGDATADIR environment variable to a directory of their choice. The user can place a shell program in this directory – this can be executed when the user runs the **diagrpt** command. The suid bit for **diagrpt** will run the shell program as root, and this program will force the spawning of a new shell with root privileges.

In this way, a malicious local user can use an exploit code to gain root privileges on the system. This exploit affects AIX 4.3.*x* and 5.1.

Of course, you can also negate this vulnerability by making the **diagrpt** command non-suid. You must be root to do this. However, ordinary users will not be able to use the command if it is non-suid.

Buffer overflow in (x)ntp

In April 2001 a buffer overflow vulnerability was recorded in (x)ntp, which affects AIX 4.3.*x* and 5.1. The Network Time Protocol daemon, (x)ntp, is shipped as standard with AIX. The buffer overflow vulnerability in (x)ntp allows a local or remote user to obtain root access or cause a denial of service (DoS). However, it should be noted that obtaining root access through the exploitation of this vulnerability is difficult, because it requires knowledge of the hardware-dependent stack registers and addresses required for different architectures.

A hack intended for use on Intel (not SP) architectures to gain root access was released on the Internet in April. However, it has been shown to cause ntp daemon problems when run. One result is likely to be a denial of service. Therefore, if the malicious user is unable to gain root access, they could still cause a system crash through denial of service using this vulnerability.

Format string in the AIX(r) locale subsystem

The **catopen()** call functionality in AIX allows a user-specified locale file to be used for displaying messages.

This call uses the NLSPATH environment variable to specify an alternative locale file instead of one of the system locale files. By constructing a valid locale file that contains special format characters and setting the NLSPATH environment variable to point to its path, a malicious local user can use

his locale file to obtain root privileges. Consequently, any executable with the setuid or setgid bit set is potentially vulnerable to root access.

This vulnerability affects IBM AIX 3.2.*x*, 4.1.*x*, 4.2.*x*, and 4.3.x. A patch has been provided for AIX 4.3.*x* (APAR IY13753).

EXTERNAL VULNERABILITIES

The following exposures have been shown to impact on AIX in some instances. Because these exposures are not directly related to AIX they are often overlooked. We have certainly seen these vulnerabilities used by crackers to provide a backdoor into AIX systems.

Secure Shell daemon vulnerability

In December 2001 a series of exposures were revealed in systems running implementations of the Secure Shell (SSH) protocol. Some of these had been previously noted by CERT, etc, but inadequate protective measures had been taken.

AIX does not ship with OpenSSH; however, it should be noted that OpenSSH is available for installation on AIX through the Linux Affinity Toolkit so there have been some exposures in AIX shops.

Openssh is a package that provides a secure (encrypted, authenticated) replacement to the R-commands (rlogin, rcp, and rsh). The SSH protocol enables a secure communications channel from a client to a server. CERT has recorded an increase in scanning and exploitation of vulnerabilities related to SSH.

As mentioned earlier, the vulnerabilities are specific to sites that have downloaded OpenSSH from the Linux Affinity Web site prior to December 2001 or the Linux Affinity Toolkit available on CD number VU157447. This represents a very small number of sites, but if you have deployed or explored Linux at your shop, it is worth checking your level of exposure.

The Web site has been updated and the updated software can be obtained from http://www6.software.ibm.com/dl/aixtbx/aixtbx-p.

This site contains Linux Affinity applications containing cryptographic algorithms, and new users of this site will be required to register.

If you cannot disable the service, you can limit your exposure to these vulnerabilities by using a router or firewall to restrict access to port 22/TCP (SSH). Use TCP wrappers or a program that provides similar functionality, or use the key-based IP restriction offered by your implementation. Nevertheless, remember that this does not protect you against attackers from within your network.

Telnetd derived from BSD source

In July 2001 a security exposure was discovered in systems running versions of telnetd derived from a BSD source.

The telnetd program is a server for the Telnet remote virtual terminal protocol. There is a remotely exploitable buffer overflow in Telnet daemons derived from BSD source code. During the processing of the Telnet protocol options, the results of the telrcv function are stored in a fixed-size buffer. It is assumed that the results are smaller than the buffer and no bounds checking is performed. This vulnerability can crash the server, or be used to gain root access.

For protection you should restrict access to the Telnet service (typically port 23/tcp) using a firewall or packet-filtering technology. Until a patch can be applied, you may wish to block access to the Telnet service from outside your network perimeter. This will limit your exposure to attacks. However, blocking port 23/tcp at a network perimeter would still allow attackers within the perimeter of your network to exploit the vulnerability. It is important to understand your network's configuration and service requirements before deciding what changes are appropriate.

IBM has provided fixes for AIX 4.3.3 (APAR IY22029) and AIX 5.1 (APAR IY22021).

Vulnerabilities in BIND

Since 1997 CERT has published twelve documents describing vulnerabilities in BIND with information and advice on upgrading and preventing compromises. Unfortunately, many system and network administrators still have not upgraded their versions of BIND, making them susceptible to a number of vulnerabilities. Prior vulnerabilities in BIND have been widely exploited by intruders.

The vulnerabilities in the various versions of BIND have been a consistent theme in security alerts. 2001 saw four BIND-related security alerts.

Domain Name System (DNS) Servers running various versions of ISC BIND (including both 4.9.*x* prior to 4.9.8 and 8.2.*x* prior to 8.2.3; 9.*x* is not affected) and derivatives. Because the normal operation of most services on the Internet depends on the proper operation of DNS servers, other services could be impacted if these vulnerabilities are exploited. AIX 4.3.*x* is affected by these vulnerabilities. DNS can be completely disrupted on affected servers.

The CERT/CC has recently learned of four vulnerabilities spanning multiple versions of the Internet Software Consortium's (ISC) Berkeley Internet Name Domain (BIND) server. BIND is an implementation of the Domain Name System (DNS) that is maintained by the ISC. Because the majority of name servers in operation today run BIND, these vulnerabilities present a serious threat to the Internet infrastructure.

ISC BIND 8 contains buffer overflow in transaction signature (TSIG) handling code

During the processing of a transaction signature (TSIG), BIND 8 checks for the presence of TSIGs that fail to include a valid key. If such a TSIG is found, BIND skips normal processing of the request and jumps directly to code designed to send an error response. Because the error-handling code initializes variables differently than in normal processing, it invalidates the assumptions that later function calls make about the size of the request buffer.

Once these assumptions are invalidated, the code that adds a new (valid) signature to the responses may overflow the request buffer and overwrite adjacent memory on the stack or the heap. When combined with other buffer overflow exploitation techniques, an attacker can gain unauthorized privileged access to the system, allowing the execution of arbitrary code.

This vulnerability may allow an attacker to execute code with the same privileges as the BIND server. Because BIND is typically run by a superuser account, the execution would occur with superuser privileges.

ISC BIND 4 contains buffer overflow in nslookupComplain()

The vulnerable buffer is a locally-defined character array used to build an

error message intended for syslog. Attackers attempting to exploit this vulnerability could do so by sending a specially formatted DNS query to affected BIND 4 servers. If properly constructed, this query could be used to disrupt the normal operation of the DNS server process, resulting in either denial of service or the execution of arbitrary code.

ISC BIND 4 contains input validation error in nslookupComplain()

The vulnerable buffer is a locally-defined character array used to build an error message intended for syslog. Attackers attempting to exploit this vulnerability could do so by sending a specially formatted DNS query to affected BIND 4 servers. If properly constructed, this query could be used to disrupt the normal operation of the DNS server process, resulting in the execution of arbitrary code.

This vulnerability was patched by the ISC in an earlier version of BIND 4, most likely BIND 4.9.5-P1. However, there is strong evidence to suggest that some third-party vendors who redistribute BIND 4 have not included these changes in their BIND packages. Therefore, the CERT/CC recommends that all users of BIND 4 or its derivatives base their distributions on BIND 4.9.8.

This vulnerability may allow an attacker to execute code with the privileges of the BIND server. Because BIND is typically run by a superuser account, the execution would occur with superuser privileges.

Queries to ISC BIND servers may disclose environment variables

This vulnerability is an information leak in the query processing code of both BIND 4 and BIND 8 that allows a remote attacker to access the program stack, possibly exposing program and/or environment variables. Sending a specially formatted query to vulnerable BIND servers triggers this vulnerability.

This vulnerability may allow attackers to read information from the program stack, possibly exposing environment variables.

This vulnerability can disrupt the proper operation of the BIND server and may allow an attacker to execute code with the privileges of the BIND server. Because BIND is typically run by a superuser account, the execution would occur with superuser privileges.

Apply a patch

The ISC has released BIND Versions 4.9.8 and 8.2.3 to address these security issues. CERT recommends that users of BIND 4.9.*x* or 8.2.*x* upgrade to BIND 4.9.8, BIND 8.2.3, or BIND 9.1.

Because BIND 4 is no longer actively maintained, the ISC recommends that users affected by this vulnerability upgrade to either BIND 8.2.3 or BIND 9.1. Upgrading to one of these versions will also provide functionality enhancements that are not related to security.

The BIND 4.9.8 and 8.2.3 distributions can be downloaded from: ftp://ftp.isc.org/isc/bind/src/

The BIND 9.1 distribution can be downloaded from ftp://ftp.isc.org/isc/bind9/

The IBM fix for AIX 4.3.3 is APAR IY16182.

Use strong cryptography to authenticate services

Services and transactions that rely exclusively on the DNS system for authentication are inherently weak. We encourage organizations to use strong cryptography to authenticate services and transactions where possible. One common use of strong cryptography is the use of SSL in authenticating and encrypting electronic commerce transactions over the Web. In addition to this use, we encourage organizations to use SSL, PGP, S/MIME, SSH, and other forms of strong cryptography to distribute executable content, secure electronic mail, distribute important information, and protect the confidentiality of all kinds of data traversing the Internet.

Use split horizon DNS to minimize impact

It may also be possible to minimize the impact of the exploitation of these vulnerabilities by configuring your DNS environment to separate DNS servers used for the public dissemination of information about your hosts from the DNS servers used by your internal hosts to connect to other hosts on the Internet. Frequently, different security policies can be applied to these servers such that even if one server is compromised the other server will continue to function normally. Split horizon DNS configuration may also have other security benefits.

Vulnerabilities in the Simple Network Management Protocol (SNMP)

The AIX operating system is susceptible to the vulnerabilities tested for by the PROTOS test suite for all levels of AIX 4.3.*x* prior to level 4.3.3.51, and AIX 5.1 before level 5.1.0.10. APARs were developed and made available last year. For 4.3.*x*, the relevant APAR is IY17630; for 5.1, the APAR is IY20943.

AUGUST 2001 HACKER ATTACKS

During August 2001, it became clear that a hacker group was targeting AIX systems. The group was cracking AIX systems and then defacing the Web sites associated with those systems. The code being used to crack the systems was primarily written by a Polish hacker group called the 'Last Stages of Delirium'. The Web defacements caused considerable embarrassment, but the important issue was that all the vulnerabilities in AIX that had been exploited had been fixed by IBM, some as far back as 1996. The APARs for all of the vulnerabilities were available and in some cases had been available for half a decade!

The vulnerabilities exploited include those associated with chatmpvc, mkatmpvc, rmatmpvc, digest, dtaction, dtprintinfo, ftpd, nslookup, pdnsd, rpc.ttdbserverd, piobe, piomkapqd, portmir, sdrd, setsenv, and telnetd. The following IBM patches have been available:

- AIX 4.1 (APAR IX74457)
- AIX 4.2 (APAR IY08288, IX85555, IX81441, IY06547, and IX76272)
- AIX 4.3 (APAR IY02944, IY08128, IX74599, IY06367, IX79909, IX81442, IY12638, IY07832, IX80724, IY08812, IY02120, and IY22029)
- AIX 4.3.2 (APAR IX85556)
- AIX 4.3.3 (APAR IY04477)
- AIX 5.1 (APAR IY22021).

Affected users need to upgrade the level of their AIX operating systems and/or apply the APARs listed above to protect their systems.

OBTAINING AIX PATCHES

Prior to 2002 users were able to obtain AIX APARs using IBM's Electronic Fix Distribution (using the FixDist program). However, FixDist was withdrawn on 31 December 2001 and replaced with Web-based fix distribution support available from http://techsupport.services.ibm.com/rs6k/fixdb.html.

These new Web-based services allow users to:

- Download recommended maintenance levels – currently AIX 5100-01 Recommended Maintenance Package is available. This package provides a collection of updates containing fixes for problems reported after AIX 5.1.0 was made available. Download and installation tips are included with this service.
- Download selective fixes – the latest operating system fixes can be downloaded by individual fileset or by groups of filesets. A new interface allows users to select all filesets within categories such as monitoring tools or performance tools, or alternatively it is possible to select each individual fileset you wish to download.
- Download selective fixes – users are able to download fixes by APAR or PTF number. There are also troubleshooting databases that can be useful for debugging problems.

More information can be obtained about accessing fixes using the Internet from http://techsupport.services.ibm.com/rs6k/fixes.html.

Users can also visit IBM Server Support to obtain fixes electronically or on physical media from http://www.ibm.com/server/support.

Security fixes are periodically bundled into a cumulative APAR. For more information on these cumulative APARs, including last update and a list of individual fixes, send an e-mail to aixserv@austin.ibm.com with the words 'subscribe Security_APARs' in the subject line.

Remember that fixes are no longer provided for AIX versions prior to 4.3 because IBM no longer supports these. IBM recommends that users running pre-4.3 releases upgrade to 4.3.3 at the latest maintenance level, or to 5.1.

ISSUES RAISED

The most worrying issue that has surfaced as a result of the security vulnerabilities seen in the last year is the number of AIX sites that have not kept up to date with security patches. This was made obvious by the Web defacements in August that exploited vulnerabilities dating back to AIX 4.1 in 1996. It was also seen in the exploitation of the Secure Shell daemon vulnerability in December 2001 which already had patches provided. If you are using older versions of AIX (especially pre-4.3) you should ensure they are patched to the relevant level or move to a higher AIX level.

As a general practice, we recommend disabling all services that are not explicitly required. You may wish to disable the SSH access if there is not a patch available from your vendor.

Our research indicates that an above-average number of the enterprises in the SME (Small-Medium Enterprise) sector have been affected by these exposures in the last year. This is unsurprising, because of the increased likelihood that there will be no or few security staff in smaller and/or mid-size enterprises. Keeping up-to-date with the IBM, CERT, and Bugtraq warnings is less likely to be undertaken in these situations. However, even large AIX shops cannot let their guard down. The volume of hacker activity is increasing.

The prestige associated with cracking large corporations is still a factor with malicious hackers, but with the increase in the number of 'scriptkiddies' this may be less of an issue. They seem less concerned with who they hack, which helps explain the increase in hacker attacks against SMEs and even home users.

A common entry point for attackers is through vulnerabilities in network services. It only takes a brief scan of the vulnerabilities discussed above to see that new exposures are discovered almost weekly; so, what is secure today may not be secure tomorrow.

As mentioned earlier, it is best to turn off all services that are not needed. The best time to do this is before the server is introduced, although few of us have that luxury, but, if you do, you should turn off as much as possible. Then, you should be able to turn back on services when there is proof that they are required. It is much easier to turn on what is needed before going live than it is to try to ascertain what needs to be turned off when the server is in production.

You can then focus your security efforts on making configuration changes and applying the latest security patches on the remaining services. Take, for example, a Web server. Among other things, this server consists of the network, the operating system, and the Web server software (application). If you focus only on securing the Web server software piece, you leave the operating system and the network doors vulnerable. An entry gained through any door is an entry gained.

Securing AIX is not a one-time process. Even though you have taken steps to turn off unnecessary services and protect the remaining ones, you still need to maintain constant vigilance. New hacking techniques are being deployed almost daily, and changes to the system, such as software upgrades, potentially open new holes. Apply security fixes to your system as soon as possible. It is also good practice to get a 'feel' for the normal behaviour of your system so that it will be easier to spot any anomalies if they occur.

In response to the issues of vulnerability raised by this review, future editions of *AIX Update* will provide a regular assessment of current AIX security vulnerabilities. This will provide an additional reminder to enterprises about the nature of security threats and the response required.

AKNOWLEDGEMENTS

This work has brought together the research of AIX Security in Austin, Texas, Bugtraq, CERT, CIAC, The SANS Institute, TESO, and many others. Security is a community effort that relies on the work of individuals and organizations to bring information to the attention of all users.

User-defined backend programs for spooling

Problem: your application generates reports and dumps them to a print queue without allowing you to customize or modify them. Solution: create a user-defined backend program that allows you to capture the print stream and modify and redirect it as needed while utilizing the existing spool daemon. This article should give you a good start on developing a user-defined backend program of your own written in Korn shell (you

might be able to call other interpreted language scripts in Bourne Shell, C Shell, or a C program, but I have not tried it).

SUMMARY OF HOW NORMAL PRINTING WORKS

An **lp**, **qprt**, **enq**, **lpr**, or similar command creates a print request to the print spooling subsystem by invoking the **enq** command. The **enq** command checks the */etc/qconfig* file for information about the queue and puts a request in the proper queue. When the qdaemon sees the request, it hands the print off to the appropriate backend program, which is usually **piobe**, **rembak**, or a user-defined backend. Think of **piobe** as being the local printer backend, **rembak** as the remote printer backend, and a user-defined backend as calling a custom script.

An example of an */etc/qconfig* entry defining an external print server with a laser printer is:

```
pat:
        device = @ps_pat
        up = TRUE
        host = ps_pat
        s_statfilter = /usr/lib/lpd/aixshort
        l_statfilter = /usr/lib/lpd/aixlong
        rq = ps
@ps_pat:
        backend = /usr/lib/lpd/rembak
```

SUMMARY OF A USER-DEFINED BACKEND PROGRAM

As with a standard queue, the print queue is generated with one of the print commands listed above, processed by **enq**, and then by the qdaemon, except the qdaemon executes the user-defined backend program by redirecting the print stream as standard input to the shell script.

SET UP A SMALL USER-DEFINED BACKEND PROGRAM

This user-defined backend program takes in one print file and sends it to two printers.

You can add a print queue.

Create file */tmp/laser.sh* with the following two lines, where queue1 and queue2 are two existing working queues that print to real printers:

```
eval "cat $*"|lp -c -dqueue1
eval "cat $*"|lp -c -dqueue2
```

Standard input contains parameters about the print queue and queue entry and also the print stream. The $* variable lists the values of all parameters passed to the shell as standard input and sends them to the **lp** print command. In my case, my postscript report was actually multiple parameters in the standard input so all parameters had to be passed. The **-c** flag makes a copy of the print file so that it does not disappear before it actually prints.

Via smit:

```
smitty spooler
Select "Add a Print Queue"
Select "other                User Defined Backend"
(these are 3 required fields)
* Name of QUEUE to add                      [twoqueue]
* Name of QUEUE DEVICE to add               [twoqueued]
* BACKEND PROGRAM pathname                  [/tmp/twoqueue.sh]
```

Enter the above to create the queue and now browse the */etc/qconfig* file to see the new entries created for the new queue:

```
twoqueue:
        device = twoqueued
        up = TRUE
twoqueued:
        backend = /user/scripts/twoqueue.sh
```

MORE THINGS YOU CAN DO WITH A USER-DEFINED BACKEND PROGRAM

Other things you can do with a user-defined backend program include:

- This script is coded to run multiple instances simultaneously so make sure that temporary files etc are named differently even though a print queue would process only one at a time. The script is stored on an NFS shared directory and can be run from multiple systems at the same time, which requires uniquely named temp files.
- Write the standard input print stream to a temporary file to make it easier to work with. Rename the print file after the data it contains.
- Take data in the report and assign it to variables so you can build conditional statements based on the data in the report.

- Write the variables out to a log so you can see what your queue has processed.
- Change or add data on the report.
- Kick off other scripts from within the backend script.
- Perform audits of which reports have or have not been printed.
- A million and one other uses.

This particular script prepares a postscript file to be passed to the AS/400 system via an NFS file share. The print stream is redirected to a file and the file name is renamed after data in the report – based upon some conditional logic. The big picture is that, once on the AS/400, an Adobe Distiller program converts the postscript to Adobe Acrobat PDF format and the AS/400 files it in a COLD storage system, which is not covered in this article. Rather than print the complete lengthy script, I'm including lines which are of more general interest. The big question is, what do you want to do with your report?

```
#!/usr/bin/ksh
# user define backend program that takes in a postscript report.
# Bump the sequential number of the report right away so it can handle
# multiple prints all issued at the same time.
read i < /user/scripts/appl.cnt
let i=i+1
echo $i > /user/scripts/appl.cnt
# Echo the number of parameters to a log file
echo $# >> /tmp/flags.out

# Echo the username who submitted the job
echo $USER >> /tmp/flags.out

# Echo users PTYPE environment variable.
echo $PTYPE >> /tmp/flags.out
# echo pid number
echo $$ >> /tmp/flags.out

# Echo base of file name for archive.
# echo $USER.$PTYPE >> /tmp/flags.out
# Echo file name of file being printed.
#  It is always the last argument.
# eval "echo \$$# >> /tmp/flags.out"
# First create unique number for each instance.
# Always cat the file which is the last parameter in the
# standard input"
# Cat to the temp file
```

```
eval "cat $* > /tmp/appl.$i"
sleep 1
# grab the account number out of the report.
#  Set a variable equal to the acct number of the patient
ACCT='grep '597 -124' /tmp/appl.$i|head -1|cut -c3-15'
# assign date and time the report was requested
PRINTDT='grep '990 -150' /tmp/appl.$i|head -1|cut -f 1 -d ")"|cut -f 2
-d "("'
PRINTDT=${PRINTDT}
echo $PRINTDT
month='echo $PRINTDT|cut -c 4-6'
# convert month name to month number.
case ${month} in Jan)     month=01;;
          Feb) month=02;;
          Mar) month=03;;
          Apr) month=04;;
          May) month=05;;
          Jun) month=06;;
          Jul) month=07;;
          Aug) month=08;;
          Sep) month=09;;
          Oct) month=10;;
          Nov) month=11;;
          Dec) month=12;;
esac

# Kick off another script do audit of all reports.
/user/scripts/LOS_compare.sh

# ********************************************************************
# The following line commented out from a day when it was thought all
# fields might be contained in the filename.  By renaming the file
#  after the variables, the variables can be passed to other systems
#  that might receive the print file.
# Total filename length limited to 50 char. including directory path
#  and ext.
# ********************************************************************
# Take all of the variables and combine them into one long file name
# and store it temporarily in a file.
#echo $ADMITDT$PRINTDT$FROMRANGE$TORANGE$ACCT$MEDREC$REPORT|sed -e 's/
/_/g'|sed -e 's/\///g'>/tmp/appl.$i.newfilename
# sleep 1
echo $i>>/tmp/flags.out
echo $REPORT >>/tmp/flags.out
cat /tmp/appl.$i.newfilename>>/tmp/flags.out

# check length of /tmp/appl.$i.newfilename to make sure it is 23 char
#  long
# If not 23 long then  record name of report and page SA later once a
#  day.
if [ 'cat /tmp/appl.$i.newfilename|wc -c' -ne 24 ]
```



```
then
cat /tmp/appl.$i.newfilename>>/user/scripts/wellness_page_list
hostname>>/user/scripts/wellness_page_list
exit 0
fi
#
# You can take your print file and send it to other systems via an nfs
# mount. Check to see if the nfs mount to the AS/400 is up. If up cp
# file to nfs directory else sleep for a while more then try it again.
# with nfsstat, if wait times for nfs calls are a long time then sleep
# also checks to see if directory 001AAR exists to see if nfsmount is
# present. Don't use the df command to see if a nfs mount is their or a
#  ls command.  The command would hang while the system tries to
#  establish the mount and you might get unexpected results. Its better
#  to use the nfsstat command because it comes back right away
#  regardless of the mount status.
until [ 'nfsstat -m /tmp/appl/E|grep All|tail -1|cut -d "," -f3|cut -d
"(" -f2|sed -e 's/ms)//g'' -lt 1000  -a -d /tmp/appl/E/001AAR ]
do
echo "nfs is down">>/tmp/flags.out
sleep 60
done
echo "nfs is up">>/tmp/flags.out
cp /tmp/appl.$i /tmp/appl/E/001$REPORT/in/'cat /tmp/
appl.$i.newfilename'.eps
chmod 777 /tmp/appl/E/001$REPORT/in/'cat /tmp/appl.$i.newfilename'.eps
# cp /tmp/appl.$i /tmp/appl/E/001$REPORT/in/'cat /tmp/
appl.$i.newfilename'.eps
#lp -dcvsix /tmp/appl/E/001$REPORT/in/'cat /tmp/
appl.$i.newfilename'.eps
rm /tmp/appl.$i.newfilename
rm /tmp/appl.$i
```

I have found no good reference to user-defined backend programs although the IBM Red Book *Printing for Fun and Profit Under AIX V4*, Document Number GG24-3570-01, is a good general printing reference.

Joe Grathoff
Senior Systems Analyst (USA)

Shell functions

USE AND SYNTAX OF FUNCTIONS

A function is similar to a shell script in that you designate a set of AIX commands to be executed when called up by a single name. They can be

used on the command line, which is not really suitable for long lists of commands, but the most common usage is in shell scripts.

There are several advantages to using shell functions.

- Functions can be designed to contain logical groupings of commands which can then be used to achieve a specific purpose within a script – almost like a series of easily understood mini-scripts, and less complex than the whole .
- When a function has been defined, it is loaded into memory, so that when it is called it runs faster than had you used the same commands in series – functions do not run as separate processes when called and exist in the current shell.
- Functions are ideal for organizing long scripts into modular sections, which is a significant advantage when it comes to debugging and testing.
- Groups of commands that are to be executed several times in the script can be called by a single function command. Once the function has been defined, it can be called from any location in the script, thus reducing the amount of coding required.

Functions defined on the command line last only for your login session, and disappear when you log out. Similarly, functions defined in shell scripts are only accessible when the shell script is being executed. If you want to use functions on a permanent basis they can either be placed in your *.profile* file, or in a *.kshrc* file defined by the **ENV** variable. Be aware that if you define large numbers of functions in these files this may result in slowing down your logging in, or running shell scripts, since your environment file is read each time.

You can define a function using one of two methods:

```
function function_name
{
     command_1
     command_2
     . . .
}
```

or:

```
function_name()
{
     command_1
     command_2
     . . .
}
```

There is no difference between the two and both formats are equally acceptable. The two curly braces define the body of the function and it is possible to put the first **{** on the same line as the function name, but it is sometimes easier to determine where a function starts if the curly brace is on a different line from the function name; the closing curly brace must be the first character on the line. Indentation of commands within the function is also recommended for readability.

Whichever format you use, you should be consistent throughout your scripts. We will use the second format.

COMMAND LINE FUNCTIONS

You will find that there is almost no necessity to define functions on the command line; you may have defined them in your profile file, or elsewhere, and thus call them from the command line, but there is rarely a need to actually define a function in such circumstances. In this section we use command line functions only to illustrate simple examples.

Suppose, for example, you wished to define your own special version of **ls**, which uses the **-x** and **-F** options to give a multi-column listing with directories and executable files appropriately marked. This can be achieved with:

```
ls( )
{
/bin/ls -x -F
}
```

As you type this function definition at your terminal you will be prompted with the secondary prompt, usually >. The conversation will look like:

```
$ ls( )
> {
> /bin/ls -x -F
> }
$
```

You will automatically revert to the usual primary prompt, **$**, after you have typed the closing **}** to terminate the definition of your function. When you enter the **ls** command without arguments (we shall later deal with arguments to functions), the new version will be executed, giving listings as required.

You must be very careful when naming your functions, especially when you are creating one to produce a modified version of an existing command. To illustrate this, redefine the **ls()** function, and instead of using the full pathname, **/bin/ls**, replace it with **ls**. When you now try to run this function, the system will eventually display a message stating that there cannot be more than nine levels of recursion, or some such similar message, depending on the current operating system release.

This is because this new version of **ls** is trying to call **ls** (that is, trying to call *itself*), which is trying to call itself, and so on. This can be avoided by using the full pathname of the command within the body of the function, or by using a totally different name for the function.

You should also be aware that you cannot create functions having the same name as built-in shell commands. You will get an error message telling you that the operation is not allowed since your function name is a shell built-in.

If you wish to change a function, or remove it completely, you should use the command:

```
unset -f function_name
```

You can also see what functions have been defined in your current shell by running the **functions** command, which is an alias for **typeset -f** (see below). This will display all the functions that have been set and the commands that are to be run within the body of each function, which may give you a large amount of output.

FUNCTIONS IN SHELL SCRIPTS

Care must be taken when functions are placed in shell scripts. Suppose, for example, we had placed the **ls()** function in a simple shell script:

```
$ vi testls
```



```
print The current directory listing is:
ls

ls()
{
 . . .
}
```

When you execute this shell script you will notice that the normal **ls** command is executed, and not your modified version. This is because the shell acts as an interpreter, and not a compiler. It interprets one line at a time, and when it reaches the **ls** command, it has not yet seen the function modifying it and so it executes a normal **ls** command. To overcome this, functions must be defined within shell scripts before they are intended to be used; most programmers create a section near the start of their scripts to contain all function definitions and thus ensure that they are defined before use.

It is also good programming practice to name your functions in such a way that when you call them there is absolutely no confusion to a casual reader that you are calling a function and not some obscure system command with which the reader may be unfamiliar. One method you can use is to ensure that all your function names start with **f_** since there is little likelihood of encountering an AIX command starting with these characters. We will use this convention.

As mentioned earlier, using functions within your scripts greatly improves your ability to debug them. For example, suppose you had a script that was loosely structured with the following functions (ignore the syntax):

```
# Functions section
f_function_1()
f_function_2()
f_function_3()

# Main section
f_function_1
f_function_2
f_function_3
```

If you have designed your script so that each function is a separate entity independent of other functions, you can comment out selected functions from the main section so that your script runs only those that you suspect have bugs. You could comment out **f_function_1** and **f_function_3** if

you suspected that the bug lay in **f_function_2**, for example. Or, if you are not sure where the bug is, you could run them in sequence, each time commenting out all the functions you didn't want to run. This greatly speeds up the debugging process, particularly if it is a very large script.

The use of functions to cut down on the amount of code used is nowhere near as important as the ability to speed up debugging. As you will discover, most scripts are developed in a fairly *ad hoc* manner with little foresight and planning, and usually it is only when you are more than half way through the script that you wish you had designed some of your functions so that you could reuse the code!

There is a syntactical use of the function within shell scripts, which is to be avoided. Suppose, for example, that you have a loop within your script which calls a predefined function. Suppose also that your intention within this called function is to perform a test which, if true, should cause the program to break out of the enclosing loop. In other words, the **break** command is contained within the body of the function.

When the shell script is executed and the function is called, even if the test is found to be true, you will not be able to break out of the enclosing loop. As you will see later, you can only break out of an enclosing loop if the **break** command is actually contained within the loop itself.

ARGUMENTS TO FUNCTIONS

A useful feature available with functions is that it is possible to pass arguments to them. For example, suppose you have a script which occasionally displays different messages on the screen when certain conditions are met. The function may look something like:

```
messages( )
{
.
print $1
print $2 >> logfile
.
}
```

In the above example do not confuse the **$1** and **$2** in the function with the command line arguments **$1** and **$2** which are passed to scripts. The function arguments are local copies for the function, so that, if you want

to pass a script's command line arguments to a function, you can either set them to variable names first, and then pass in the variables as arguments to the function, or you can use them as arguments when you call the function. For example:

```
messages $1 $2
```

or:

```
messages $*
```

In our example above, whenever the script requires a message to be displayed, the function is called by a command embedded somewhere in the body of the script. For example:

```
messages "/tmp has exceeded its filesystem threshhold"
```

When the function executes, the string "/tmp has exceeded its filesystem threshhold" becomes **$1**, so that the command **print $1** will display the message on the screen. Even though the function expects two command line arguments, and you are only passing one to it, it will set **$2** to null, so that in this particular case it just adds a blank line to the log file.

You can call the function with multiple arguments:

```
messages "/tmp has exceeded its filesystem threshhold" $var
```

In this case the contents of the variable **var** are passed to the function and become **$2**. In the above example we have enclosed the first argument in double quotes to ensure that the message itself is passed as a single argument to the function. We could also have enclosed **$var** in quotes but we know that we are only passing a file pathname to the function so it is not necessary. In general though, as you would for script arguments, always enclose your function arguments in quotes if you want to pass spaces as part of the argument.

In our earlier example of the **ls()** function we had no arguments within the function itself so that the function would only display entries in our current directory. If we wanted to pass it one or more arguments, such as a number of directories and/or files, then we would modify the function to:

```
$ ls( )
> {
> /bin/ls -x -F $*
```

```
> }
$
```

and call it with, for example:

```
ls /tmp /var
```

FUNCTIONS AND THE TYPESET COMMAND

We mentioned earlier that the **functions** command, which is an alias of **typeset -f**, can be used to display all function names within the current shell, along with their inclusive text. Since function names are stored in the history file, **typeset** does not display them if the history file does not exist. You can also display a single function by using:

```
typeset -f function_name
```

or:

```
functions function_name
```

You can use an alternative form of **typeset** to display just the function names only:

```
typeset +f
```

Ordinarily, functions are unset when the shell executes a shell script. For example, if from the command line you define the function **ls()** and then create a simple script which just runs **ls** and nothing more, when you execute this new script it will not run the function **ls()** but will instead run the standard **ls**.

It is possible, however, to export function definitions so that they remain in effect across shell programs that are not separate invocations of **ksh**. This can be done using the **-x** option:

```
typeset -fx function_name
```

If you now run the command **typeset -fx ls** and again run the simple script, you will see that it now runs your function **ls()** and not the standard **ls** command. If you enter **ksh** to create a sub-shell and then try to run your function **ls()**, or the simple script that runs the function, you will see that the function definitions have not been exported to this separate invocation of the shell and the standard **ls** command will be run instead.

Functions that must be defined across separate invocations of the shell,

for example when you are in a windows environment, or when you simply execute the **ksh** command as above, should be defined in the **ENV** variable file, which is usually **.kshrc**. There is no need to export the function since **.kshrc** is executed every time a sub-shell is created, and the function will always be available in the new sub-shell.

And just to confuse you completely, the function will still not be accessible in any shell script which is executed in this sub-shell! To ensure that the function is always available, both in sub-shells and in scripts, it should be exported within the **.kshrc** file by using the **typeset -fx** command.

Options to typeset

In addition to using **typeset** for functions, there are a number of other useful options to the command that allow you to set, unset, and display the values of variables, parameters, and their attributes. With no command line arguments, **typeset** displays the names of all variables and parameters in effect, as well as their types (**export**, **function**, **integer**, **read-only**, etc).

The various options for **typeset** are:

- **-u** – converts all alphabetic characters assigned to the variable to upper case. Numbers and other characters remain the same.
- **-l** – converts all alphabetic characters assigned to the variable to lower case. Numbers and other characters remain the same.
- **-i[n]** – declares the variable to be an integer with the base of **n**; if **n** is not specified the default is base **10**. Declaring a variable as an integer makes arithmetic faster. Use **2** for binary, **8** for octal, and **16** for hex.
- **-L[n]** – left justifies the variable. The optional **n** specifies the field width, otherwise it is determined by the width of the value on its first assignment. When the parameter is assigned, it is filled on the right with blanks, or it is truncated. The **L** and **R** options are mutually exclusive. For example:

```
$ typeset -L8 var=00123456789
$ print $var
00123456
```

- **-LZ[n]** – strips off leading **0**s or leading blanks, counts the number

of characters from the left up to **n**, and then left-justifies. For example:

```
$ typeset -LZ8 var=00123456789
$ print $var
12345678
```

- **-R[n]** – right-justifies. If **n** is specified, it truncates after counting **n** characters from the right. For example:

```
$ typeset -R8 var=00123456789
$ print $var
23456789
```

- **-RZ[n]** – right-justifies and fills with leading zeros if the length is less than **n**. For example:

```
$ typeset -RZ8 var=1234
$ print $var
00001234
```

- **-r** – marks the variable as read-only.
- **-x** – marks the variable as exported.

The **-** in front of an option sets the attribute. You can unset the attribute by using a **+** before the option. If **typeset** is used within a function, a local variable is created. When the function exits, the variable's original value and attributes are restored if it previously existed.

Although in the above examples we have used the format **typeset options variable=xxxx**, you do not have to set the variable's value at the same time as you declare it. You can define more than one variable at a time using, for example:

```
typeset -i qty1 qty2
```

which will declare both of these variables to be type integer, but will not give them a value.

BLOCKS OF CODE

When you look at the syntax of a function definition, you will note that we are merely surrounding a sequence of commands with the curly braces. If we were to remove the function name and parentheses, and just use the

braces to surround the commands, we would be left with a block of code that would behave just like a function without a name. The commands would be executed as they would in a function, but we would not be able to call this block from elsewhere in a script.

This block of code does not appear to have a great deal of use as it currently stands, but there may be occasions when you want to redirect the output from a number of commands to the same location, or alternatively you may wish to receive the input from a file. You could of course redirect input/output individually for each command, or you could use a construction requiring less code.

As a simple example let's assume that we are performing a system back-up and we want to redirect the output from all our commands to a log file, and any error messages to a separate error log. We can do this with the construction:

```
{
date
print "Starting backup"
mksysb . . .
date
print "Backup completed"
} > logfile 2> errorlog
```

In simple scripts you might use coding such as that shown above, but if this back-up example were to be part of a much larger script, where we performed a series of tests first, and then perhaps a validation at the end of the back-up, then the block of code would probably be called as a function, say **f_start_backup**, and we would redirect the output of the function with:

```
f_start_backup > logfile
```

You should, of course, indent the commands within a code block to make it easier to read and determine the start and end of the code block, but be aware that you cannot use constructions like:

```
{ date ; print "Starting backup" ; . . } > logfile
```

This is because the last brace of the code block must be the first character on a line. Instead, the secondary prompt will be displayed if you run such a construction from the command line, effectively asking you to terminate

the command with the closing curly brace. If you run it in a script it will give you a syntax error.

You should be aware that you cannot use code blocks surrounded by braces within a function itself since this will generate a syntax error. This is because the shell expects a closing curly brace to define the function boundary after it has encountered the initial opening brace, and not another opening brace starting a code block.

Code blocks in sub-shells

There is another construction which groups commands into blocks of code, but operates in a slightly different way from blocks contained within curly braces. You may remember the **$(. . .)** construction that is used in command substitution, where the commands within the brackets are executed as a block with the resulting output usually allocated to a variable. The brackets in this construction serve as delimiters to a code block in much the same way as braces, and where you use the braces construction in a script you can, in most cases, use brackets instead. There are, however, three main differences between the two constructions.

The first is a relatively trivial syntactical difference. If you use brackets you can use a construction (not permissible with braces) such as:

```
( date ; print "Starting backup" ; . . ) > logfile
```

The second involves a much more significant difference. When you use brackets around the group of code, the commands are executed in a sub-shell, unlike using braces where the commands are executed in the current shell. Under most circumstances this will not be a problem, but if you modify a variable in the block of code which runs in the sub-shell, its new value will not be available elsewhere in the script, and under these circumstances you must use the braces construction. Also, when you use sub-shells you can run into problems when trapping signals.

The last difference relates to the fact that when you surround your code with brackets, then the commands are run in a separate process, which is usually less efficient than using code blocks surrounded by braces.

Tonto Kowalski
Guru (UAE)

Should I use ‘large file enabled’ file systems?

Do you run a database on your AIX system? If you do and you haven’t checked whether your file systems supporting the database are ‘large file enabled’, then read on (otherwise it could be a very long night for you some time in the not too distant future!).

Let me start by saying what ‘large file enabled’ means, then I will talk about why you would use it, how to check whether you are actually using it, and what could happen if you don’t use it.

WHAT DOES ‘LARGE FILE ENABLED’ MEAN?

‘Large file enabled’ basically means that files within a file system can be larger than 2GB. The default is that files cannot be larger than 2GB.

WHY WOULD YOU WANT YOUR FILE SYSTEM TO BE LARGE FILE ENABLED?

If you are running any application (and I count databases here as applications) that has files which will grow to be more than 2GB in size, then you need to large file enable the file system containing these files. Once a file hits the 2GB limit you will get an error message back, even if you still have plenty of space in your file system.

TO CHECK WHETHER YOUR FILE SYSTEM IS LARGE FILE ENABLED

You can check whether a file system (<*fs-name*>) is large file enabled by issuing the following command:

```
#lsfs -q /<fs-name>
```

Your output will look something like this:

```
Name            Nodename   Mount Pt           VFS   Size    Options
Auto Accounting
/dev/hd4        —          /                  jfs   40960   —
yes  no
(lv size: 40960, fs size: 40960, frag size: 4096, nbpi: 2048, compress:
```

```
no, bf : false, ag: 4)
```

In this particular case, the file system is not large file enabled (bf: false); if it were, you would see bf: true.

TO CREATE A FILE SYSTEM THAT IS LARGE FILE ENABLED

You can specify large file enabled when you create a file system as follows:

```
#crfs -v jfs -g <volume-group>
-a size=<size-of-filesystem-in-512-blocks>
-m <mount-point>
-a bf=true
-a frag=512
```

the '-a bf=true' bit is the large file enable bit.

WHAT CAN I DO IF I HIT THIS PROBLEM?

If you don't have large file enabled file systems and a file within the file system hits the 2GB limit, at that point you have only one option – create a new, temporary, file system and copy the data to it; delete the old file system; create another new file system, this one being large file enabled; and copy the data back to that. This supposes that you have enough space to create the temporary file system. There is no way to *alter* a file system to make it large file enabled; you must create it!

CONCLUSION

If you are running a database on AIX, you should give serious consideration to changing all the file systems that support the database to be large file enabled (or make sure that no individual file within the file systems will ever be larger than 2GB!).

C Leonard
Freelance Consultant (UK)

Have you come across any undocumented features in AIX 5L? Why not share your discovery with others? Send your findings to us at any of the addresses shown on page 2.

Input for shell scripts

The three most common ways that shell scripts can obtain input are:

- Command substitution
- Interactive input using the **read** command
- Command line arguments.

COMMAND SUBSTITUTION

Although we have included command substitution in this article, which is intended to introduce you to the ways in which shell scripts get their input, in reality, command substitution is most frequently used to assign the output of a command, or series of commands, to a shell variable. Indeed, all the examples we have used in scripts so far have been of this type. Because it can also be used for the input for shell scripts, it seemed like a good opportunity to introduce this topic.

There are two different types of syntax used in command substitution, and the one you adopt is purely a matter of personal preference, but, as we shall see, one particular type of construction leaves you with limited options.

Using backquotes

Backquotes (also called *grave accent marks*) are metacharacters that indicate to the shell that it is to perform command substitution. Any command enclosed in backquotes will be executed, and the place it occupies on the command line, including the backquotes, will be replaced with the output generated by the command.

If, for example, you entered the command:

```
print you are currently in the `pwd` directory
```

then before running **print**, the shell executes **pwd** and replaces the sequence of characters **`pwd`** with the output generated by **pwd**. Then **print** is run with the resultant list of arguments.

You must exercise some care when the output from any command substitution (whether using backquotes or otherwise) is used as the input

to another command, particular when the output generated contains spaces and the command accepting the input expects a single argument. This is much the same as using variable values as input for commands, and you must present this input as a single argument by enclosing all the spaces inside double quote marks since they do not quote a backquote.

For example, suppose that you wanted to look for the current date in a number of files. If you mistakenly entered the following command:

```
grep `date` file1 file2 file3
```

the shell would perform the substitution, and **grep** would be run with arguments such as the following:

```
Mon, May, 3, 08:55:09, 2002
```

This would produce a series of messages saying, for example, **can't open Mon**, etc. The command you need to enter is:

```
grep "`date`" file1 file2 file3
```

so that the total output of the **date** command would be passed as a single argument to **grep**.

Using the $(. . .) construction

There are two major disadvantages to using backquotes for command substitution:

1 When reading scripts, and depending on the type of font being used, it is sometimes difficult to distinguish a backquote from a standard single quote, often leading to confusion when trying to interpret or debug a script.

2 Backquotes cannot be nested since the shell interprets everything between the first and second backquotes it encounters on the command line as one command (or pipeline), everything between the third and fourth backquotes as another command, and so on. It is exceedingly unlikely that nested backquotes will produce the result you intended, and is more likely to generate many error messages.

There is another form of command substitution that we have already briefly encountered in *Examples of shell scripts, AIX Update*, Issue 78, April 2002, and which is more versatile and less confusing to the reader; this uses the construction **$(. . . .)**. Between the brackets you can include

a single command, with or without arguments, or a pipeline, and all commands are executed as part of the substitution. Because of the above disadvantages, we will use this type of construction instead of backquotes.

You have already seen examples in one of the sample scripts in the earlier article, but the following is an example that includes nesting. Forget the syntax of the probably unrecognizable **test** command for the time being since it will be explained in a future article.

```
DIR=$( [[ $(pwd) = "/" ]] && print root)
```

The innermost substitution is executed first, and the shell gradually works its way out until all substitutions have been completed.

The above example is a fairly complex command substitution used to assign a value to a variable and in your shell programming career you will probably start off with far simpler examples, getting progressively more complicated as you gain experience with the Korn shell.

Generating argument lists

Command substitution can be used to generate a list of arguments for a program. For example, suppose that you wanted to count the number of lines in a lot of files and you also wanted to perform other operations on this group of files as well. To make dealing with the group easier, you could create a file containing the names of each of the files.

To make the example simple, we will only use three files:

```
$ vi file_list

/etc/sendmail.cf
/etc/passwd
/etc/inittab
```

Create this text file and then run the following command:

```
time wc -l $(< file_list)
```

We have introduced the **time** command to give you some idea how long this command takes to run. The reason for this is that we could have used **cat file_list** instead of **<filelist**, but the latter construction is run by the shell in a more efficient manner than **cat**.

Now try running the following command and note the difference in times; ideally you should run both commands several times so as to give an

average, since there may be additional system activity when you run one command but not when you run the other. This gap would be even greater with large numbers of files, or with larger files.

```
time wc -l $(cat file_list)
```

In this particular example you could have used filename generation characters to specify the files. However, if the files that you want a command to operate on are not named in a consistent way, or are spread across a number of directories, it may not be possible to specify them with patterns.

INTERACTIVE INPUT USING READ

The **read** command is used for interactive input for shell scripts and waits until you enter a line of text. It then reads the line and sets a shell variable to the text that you entered. The syntax for the command is:

```
read variable
```

When you enter a command such as this, nothing will happen until you type some text and press **<CR>**. Then, whatever you type will be assigned to *variable*. The **read** command actually separates its input line into words delimited by the Internal Field Separator (**IFS**) variable, which by default can be either spaces or tabs. In the above example, *variable* will be set to the characters of the first word.

When a **read** command is executed in a shell script, execution of the script halts until the user types something and presses the *Enter* key. In shell scripts, the **read** command will usually be preceded by a **print** statement, which displays a message to let the user know that he will be expected to enter some text. For any script containing a **read** statement, speed of execution is obviously not an important factor since at some stage you will be waiting for a user to enter data.

Try the following example:

```
$ vi input

print Please enter a word
read word
print $word
```

Execute the shell script and, after displaying the first message, the program

will wait until you enter text. Once you have done so, the next command is then executed and the word you entered will be displayed. If instead of typing text you had pressed **<CR>**, then the variable word would have been assigned a null string.

You can read several words of input into several variables at the same time with a single **read** statement. Here is an example:

```
$ vi input2

print Please enter two filenames:
read file1 file2
print File1: $file1
print File2: $file2
```

When you execute this shell script, you must enter the two filenames on the same line separated by spaces or tabs. The **read** command takes whatever you have entered and assigns it to the two variables. If there are two variables and you have entered only one word, the second variable will be assigned a null string. If you were to enter more words than there are variables, then the first word would be assigned to the first variable, and all the remaining words would be assigned to the second variable.

Accepting read input from a file

The following simple example shows how **read** can accept input from a file. First create a file containing a single line using

```
print "fred bloggs" > namefile
```

Then enter:

```
read name < namefile
```

and test that this has worked by entering **print $name**. If there is more than one line in the file then **name** will be assigned all the characters on the line up to the carriage return at the end of the line. As we shall see later, we can use **while** loops to read all the lines in the file and perform operations against each line.

Using read with fredmail

Let us now rename the **fredmail** file we previously created in the earlier article, and modify it so that it asks the user to enter the name of the recipient, and sends the mail to that user:

```
$ vi mailto

#!/bin/ksh
# Get name of recipient and subject of message
print 'Message to: \c'
read whom
print 'Subject: \c'
read subject

# Add the subject to the memo.
print "Subject: $subject" > memo_tmp
# The cat command reads whatever the user types and
# appends it to memo_tmp. End the text with CTRL D.
print 'Enter message and finish with ^D on a blank line'
cat >> memo_tmp

# Append the date, the message, and a blank line to the log file
date >> logfile_$whom
cat memo_tmp >> logfile_$whom
print '\n' >> logfile_$whom

# mail the message
mail $whom < memo_tmp
# Remove the message now that a copy has been saved
rm memo_tmp
```

This version of **mailto** will prompt you to enter the user name of the person you want the mail sent to, and stores it in the variable **whom**. It next reads the subject, and then prompts you to enter the text. A copy is stored in a log file, and a separate log will be created for each user. The name of the file created is **logfile_$whom**. Finally, the mail is sent to the user.

You will notice that the argument to the first two **print** commands is the escape sequence **\c**. This will leave the cursor on the same line as the message, and you can enter your response there; some programmers prefer this neat and tidy approach. Note that the whole argument to **print** has been enclosed in quotes because the \ must be quoted to remove its normal metacharacter meaning and we want to pass this pair of characters to **print** so that they can be interpreted as an escape sequence. Otherwise the shell will replace the pair of characters **\c** with the single character **c**. You could achieve the same result by using **print -n**, which omits the final newline.

If you look at the line that prints the subject to the temporary file, you will see that **Subject: $subject** is enclosed in double quotes. This particular **print** command would also work without the double quotes, until such time as a subject containing a metacharacter was entered, for example, a

*****. Without double quotes, **print** would then first display the word **Subject:**, followed by the list of all the files in the current directory because of the filename expansion character, *****. The double quotes result in the execution of the command **print "Subject: *"**, and the ***** loses its meaning as a metacharacter and merely becomes an ordinary character.

If you had instead used single quotes, then the **print** command would have appended the characters **Subject: $subject** to the memo, which is not what you intended.

You can practise using the **mailto** command to send messages to yourself. View them with the **mail** command. Remember to remove the log file when you have finished.

COMMAND LINE ARGUMENTS

Another way that you can give information to a shell script is through command line arguments. Although the **read** command can always be used to obtain input interactively, it is not always the most convenient method. Often it is easier for the user to enter whatever data the command needs on the same line as the command itself.

Consider the command line:

```
cp file1 file2
```

There are two arguments on this command line, which are **file1** and **file2**. Similarly:

```
ls -l /usr/bin
```

also has two arguments, namely **-l** and */usr/bin*.

The command line:

```
ps -ef | grep root
```

contains two commands, **ps -ef** and **grep root**, each of which has one argument. The **|** is not an argument and will not be passed to either command.

Within a shell script you can have access to the arguments that appear on the command line used to invoke the script. The command line arguments are a set of variables called positional parameters, their notations being **$1**,

$2, **$3**, etc. The value of **$1** is the first command line argument, **$2** is the second, and so on.

If you consider the example:

```
script one two
```

then any occurrences of **$1** in the script will be replaced with the string **one**, and **$2** will be replaced with **two**. If you are going to access these variables in multiple places in your script, then it is good practice to set them to variable names at the start of the script. This maintains consistency and helps to avoid confusion since, as you will see later, **$1** in the main body of a script is not the same as **$1** when used in a function.

Here is a shell script that displays the first three arguments with which it was called. Before printing the arguments, the script displays its own name; **$0** is always equal to the name of the shell script.

```
$ vi param

print \$0 is: $0
print \$1 is: $1
print \$2 is: $2
print \$3 is: $3
```

When you enter the command with three command line arguments, you will see them displayed as, for example:

```
$ param hi there folks

$0 is: param
$1 is: hi
$2 is: there
$3 is: folks
```

If you run **param** with two command line arguments instead of three, **$1** and **$2** will be assigned the first two arguments, and **$3** (as well as all subsequent positional parameters) will be assigned the null string. Similarly, if you run **param** without any arguments, then all the positional parameters will be assigned the empty string. On the other hand, if you invoke **param** with five arguments, **$1** to **$5** will be set to the five arguments entered, and **$6** onwards will be assigned the null string; **param** does not reference **$4** and **$5**, but that does not stop values from being assigned to them.

In the Korn shell, if you use the expression **$10** to try to get a tenth command line argument, the shell will interpret the **$10** as **${1}0**; in other

words, the first command line argument followed by zero. Instead you can access such parameters by using **${10}**, **${11}**, etc.

You also need to be aware that you cannot assign new values to the positional parameters in the way you assign new values to ordinary variables. For example, the following command will not assign a new value to **$1**:

```
$ 1=newvalue
/bin/ksh: 1=newvalue: not found
```

As you are aware, if you want a string that contains spaces to be treated as a single argument, you must precede the space with a \, or enclose the string in single or double quote marks. In the following example, note that the entire quoted string is assigned to **$1**:

```
$ param "hi there" folks

$0 is: param
$1 is: hi there
$2 is: folks
$3 is:
```

Here is another example that illustrates the use of variable substitution, but from a different perspective:

```
$ greeting=hello
$ param $greeting


$0 is: param
$1 is: hello
$2 is:
$3 is:
```

If you set **greeting** to **hello there**, and you want this to be treated as a single argument to **param**, then you must surround **$greeting** with double quotes when you use it as an argument to **param**.

Making mailto accept an argument

Let us now modify the **mailto** script from the renamed **fredmail**, so that it accepts a command line argument containing the recipient's name. This modification will make it behave more like the **mail** command.

The new **mailto** file should be modified to the following:

```
$ vi mailto
```

```
#!/bin/ksh
RECIPIENT=$1
LOG=logfile_$RECIPIENT
LOGDIR=/usr/local/log
TMPDIR=/var/tmp
MEMOFILE=memo_$$
# Get subject of message
print 'Subject: \c'
read subject

# Add the subject to the memo.
print "Subject: $subject" > $TMPDIR/$MEMOFILE
# The cat command reads whatever the user types and
# appends it to MEMOFILE. End the text with CTRL D.
print 'Enter message and end with ^D on a blank line'
cat >> $TMPDIR/$MEMOFILE

# Append the date, the message, and a blank line to the log file
date >> $LOGDIR/$LOG
cat $TMPDIR/$MEMOFILE >> $LOGDIR/$LOG
print '\n' >> $LOGDIR/$LOG

# mail the message
mail $RECIPIENT < $TMPDIR/$MEMOFILE
# Remove the message now that a copy has been saved
# and it has been mailed
rm $TMPDIR/$MEMOFILE
```

You will see that we have introduced a number of variables at the beginning of the script. **RECIPIENT** defines the name of the user who will receive the memo (see method of execution below), **LOG** defines the name of our log file (which becomes a unique name because we are appending the recipient's name to it), **LOGDIR** defines the location of our log file, **TMPDIR** is set to a temporary directory name (more on that later), and **MEMOFILE** is the name of the file containing our temporary memo (remember that the copy of the memo we send is located in the log file).

In general, any string of characters which are to be used multiple times within a script, whether they be a directory pathname, filename, or otherwise, should always be defined as early as possible in the script. Should the value have to be changed at a later date then all you need to do is make a single change.

One major flaw with the script is that we do not test that there is a single command line argument, and that it is a valid username, before setting the **RECIPIENT** variable. The reason for this is that you have not yet learned the appropriate commands to do so; all will be revealed in a future article!

You will also discover that a significant task in creating any well-written script will involve code which does extensive error testing to ensure that your script is as resilient as possible. Quite often this can be greater than the other code you need to achieve your ends.

If two people ran the old version of **mailto** at the same time, they did not interfere with each other, because each time the script was run, the temporary file was most likely created in their home directories. However, if two people were to run the new version with the temporary file still being named **memo_tmp**, and located in the directory set by the **TMPDIR** variable, then both processes would try to create the same file in the same directory, at the same time.

To avoid this problem, we make sure that, each time **mailto** is run, a different filename is used. As you will recall, **$$** is the variable that is equal to the process ID of the current shell process. If the temporary file name is of the form **memo_$$**, then each time **mailto** is invoked it will be run by a separate shell process with a unique process ID, and a different temporary file will be created for each user. If two users run the script at the same time, the two processes will have different IDs, and two different temporary files will be created.

You might argue that there is very little chance of two users running the script at the same time, and on a system with a small user base this might be so. But many scripts which start off with a single user, or just a small number of users, have a habit of increasing in popularity, particularly if they perform a function not previously available, and so are introduced to a much larger user base. When this happens all sorts of things start to go wrong when the script is simultaneously executed by multiple users, and as a general principle all your scripts should be written with this in mind; it is far easier to accommodate multiple users in the design stage than to make changes at a later date.

If the old version of **mailto** were terminated abnormally, the temporary file, which the code tries to remove at the end of **mailto**, would not be removed. Programs that create temporary files should usually create them in a directory that is periodically cleaned up by system administrators, and **/var/tmp** should be one such directory. Another alternative is to create the temporary file in any suitable directory and then introduce code to remove it when the command is terminated abnormally.

To run this new version of **mailto**, you must specify the recipient's name on the command line:

```
mailto fred
```

What happens if more than one argument is specified on the command line? The **mail** command in the script only sends the memo to **$1**, so that only the first person listed on the command line will receive the message. You will later learn how to make a shell script loop over the list of command line arguments, and perform a sequence of operations, such as mailing a file, once for each argument.

All the command line arguments – $*

The variable **$*** is equal to the list of all the positional parameters. The following shell script illustrates this:

```
$ vi all

print The parameters:
print $1 $2 $3 $4 $5 $6 $7 $8 $9
print '$*':
print $*
```

When you run this command you will see that **$*** does not include **$0**, the name of the script.

Although there are only nine positional parameters in this script, if you enter more than nine arguments on the command line, **$*** will include them all. You can try this by running **all** with more than nine arguments. You can also modify this or any script to process more than 9 command line arguments using the **${10}** type notation.

As you may recall, we created a shell script named **word_count**, which counts the number of occurrences of each word in a file. We can modify this script so that it can operate on any file, not just the one named in the original script. This can be done by changing the first line to:

```
tr -A "[A-Z]" "[a-z]" < $1 |
```

This now allows input from the file specified by **$1**, rather than by redirecting the input from **mytext**.

We can further improve on **word_count** by making it operate on more than one file. You may be tempted to change the first **tr** command to read:


```
tr -A "[A-Z]" "[a-z]" < $1 $2 $3 $4 $5 $6 $7 $8 $9 |
```

or:

```
tr -A "[A-Z]" "[a-z]" < $* |
```

Neither of these commands will work because the standard input can be redirected from one file only. This can be overcome by using the **cat** command to join all the files together. For example:

```
cat $1 $2 $3 $4 $5 $6 $7 $8 $9 |
tr -A "[A-Z]" "[a-z]" |
```

or:

```
cat $* |
tr -A "[A-Z]" "[a-z]" |
```

You should use **$***, rather than listing the positional parameters, if you want this script to be able to deal with more than nine filenames. You should also note that this version of **word_count** will produce combined totals of the occurrences of words in all files, and not in individual files listed one after the other.

The number of command line arguments – $#

$# is another automatically maintained variable, and is equal to the number of command line arguments. It does not include the name of the script. The following shell script simply prints the number of arguments with which it was invoked:

```
$ vi count

print $#
```

Try this script by entering:

```
$ count
Ø

$ count hi there
2

$ count 'hi there'
1
```

AIX news

Micro Focus has announced the general availability of the only 64-bit COBOL development environment for AIX 5. Micro Focus Server Express is now capable of providing businesses maximum performance from both the processor and updated operating system architecture.

For further information contact:
Micro Focus, Old Bath Road, Newbury, Berk, RG14 1QN, UK.
Tel: (01635) 32646.
URL: http://www.microfocus.com/press/releases/20020514.asp.

* * *

IBM has announced Version 2 of its Workstation APL2 V2 application development and data exploration tools for use on AIX, Linux, Solaris, and Windows. New features include support for Linux on PC-compatible systems, namespaces for application encapsulation and reuse, interface to Tcl/Tk command language for platform-independent GUI development, and APL2 Runtime Library for distribution of APL2 applications.

The programming language is used by both developers and interactive end users for application development and problem solving, targeting both commercial and scientific applications in areas such as commercial data processing, system design and prototyping, engineering and scientific computation, artificial intelligence, and the teaching of mathematics and other subjects. It supports what-if modelling, exploratory programming, interactive computing, decision support, and data analysis. It manages large quantities of data and enables development of applications that can be deployed on both host and workstation systems.

The latest version combines the features of previous workstation APL2 products for AIX, Solaris, and Windows with support for Linux on PC-compatible systems.

For further information contact your local IBM representative.
URL: http://www.ibm.com/software.

* * *

ValiCert has released Version 3.5 of SecureTransport for AIX 5.1. The product is a secure document and data delivery system.

The new release provides a new browser-based ActiveX client for Internet Explorer users, increasing the efficiency and reliability of their file transfer while lowering customer support and software distribution costs. There are new, customizable end-user interfaces, and an improved authentication framework.

The product now offers customizable HTML templates, which maintain visual continuity for existing Web portals, preserve corporate branding, and can be customized to include application-specific user interfaces.

For further information contact:
Valicert, 1215 Terra Bella Avenue, Mountain View, CA 94043, USA.
Tel: (650) 567 5400.
URL: http://www.valicert.com/products/secure_transport.html.