Global Impact Analysis of Dynamic Library Dependencies

Yizhan Sun and Dr. Alva L. Couch – Tufts University

ABSTRACT

Sowhat is an administrative tool that performs global impact analysis of dynamic library dependencies for Solaris systems. Sowhat runs in two phases. It first builds a database of dependencies offline in the background, and then answers user queries and generates reports in real time based upon stored knowledge. Using sowhat, one can find problems with library bindings in large program repositories before these problems annoy potential users.

"I can stands what I can stands, but I can't stands no more!" – Popeye

Introduction

We manage a large Solaris network containing several large shared program repositories, each containing up to 1000 programs apiece. Several times in the last two years we have inadvertently and unpredictably broken user programs and network services during routine upgrades of dynamic libraries. Library dependencies that load libraries from NFS-mounted partitions have also been responsible for random boot-time failures of daemons, because a daemon cannot access its libraries when it needs to load them. Lastly, we can never be sure that a dynamic library is not in use by any program, so we can never delete a dynamic library safely.

In this paper we describe a very simple technique for avoiding such anomalies through "in vivo" global analysis of library dependencies. A Perl script constructs a library dependency database from a user's-eye-view of a system and generates an 'inverted' report, for each dynamic library, of *all* of the programs that will attempt to load it. Using this report, one can easily spot problems such as those mentioned above.

Related work

Our program is related to many other tools that *almost* – but not quite – entirely fail in detecting library dependencies in a heterogeneous and open computing environment. Package managers such as RPM [1] and Depot [2, 8, 9, 11] allow point-of-installation dependency analysis based upon a "closed-world" assumption. As long as either has complete control of the environment, dependencies will likely be satisfied correctly. Combine these with other software installation techniques such as direct compilation, however, and they will fail to spot possible problems in the resulting stew. Neither RPM nor Depot has full knowledge of the user environment in which the software that they install will be utilized.

Change detectors such as tripwire [14] and aide [7] can detect changes to a filesystem but cannot analyze the potential effects. We need to know not only the names of files that changed, but also which programs could be affected by libraries that have changed, and even perhaps which programs might be broken by a particular 'make install'.

Pre-existing tools most applicable to our problem act to control the user's environment carefully so that conflicts should not occur. The Soft [5] environment control system manages library bindings by careful control of the user's environment variables. Soft was preceded by much other work on creating software modules that can be invoked by users on demand [4, 6]. These inspired our own software module mechanism that sowhat understands and analyses.

Others have faced the same problems with libraries and opted to control the dynamic linking environment carefully in order to avoid the need for approaches like sowhat. Vendor-supplied software has been hampered in Linux by the large number of differing distributions of what is essentially the same core operating system. Differences in distributions can often break software, so that a product that works properly in one distribution may not work in another.

The Linux Standard Base (LSB) project [13] seeks to provide a dynamic linking environment within Linux in which vendor-provided software is guaranteed to execute properly. The goal of LSB is to identify a set of core standards that must be shared among distributions in order to guarantee that a product that works properly in one of them will work in all compliant distributions. These standards include requirements for the content of dynamic libraries, as well as standards for locations of system files used by library functions.

With these standards in hand, the LSB provides tools with which one can certify both environments and programs to be compliant with the standard. Linux distributions can be examined by an automatic certification utility that checks link order, versions of libraries, and locations of relevant system files. A distribution may have more libraries than the standard specifies, but the libraries specified in the standard must be first to be scanned during linking and must contain the appropriate versions of library subroutines. Another certification utility checks that the binary code for linux applications only calls library functions protected by the standard. Since the LSB tools solely analyze the contents of binary files, they can check closed-source executables for compliance.

The LSB's library version probing is a much deeper library analysis than our tool performs, though not beyond the scope of future work. For example, sowhat relies upon the names of libraries to indicate versions, while LSB scans them for embedded version strings, so that it can accurately determine the content and versions of renamed or even misnamed libraries.

Dynamic Libraries and Global Analysis

The cause of our problems is that in a modern UNIX environment, the file containing an executable program is seldom the only component required in order to execute the program. Each executable binary file contains references to one or more dynamic libraries that are linked into the program after it is invoked and before it begins execution (through the dynamic linker ld.so). The typical reason for this way of segmenting programs is to share runtime memory; one in-memory copy of a library may be shared among several executables running at the same time. But if the libraries needed at runtime by a specific program are changed or deleted, the referring program may change in behavior or fail to run at all.

While it is possible (through the command ldd) to examine which libraries are loaded by any specific program, no general mechanism except sowhat exists for examining which programs load a specific library. This is because while the former question involves examining one executable, the latter involves examining *all possible executables* that could load the library. The former question is *local* in scope, while the latter is *global*. In practice, this means that without a tool such as sowhat, one can never safely delete any library in the system without the risk of breaking an unknown program. If user uptime is more important than disk space, this means one can never delete or upgrade any library at all because the impact of such a change is unknown. One can only add libraries. This leads to 'library rot' much like 'filesystem rot' [3], in which libraries gradually fill up with useless files that one cannot delete with any assurance of lack of impact.

Analyzing Dependencies

The basic function of sowhat is very straightforward. One runs it as a normal user. Sowhat parses the user's PATH environment variable to create a list of programs to scan. For each program, it parses the output of the Solaris utility 'ldd' [12] to generate an index of the libraries the program will load. It inverts this index so that it can list programs that call each library and stores the inverted index in a database from which it can generate reports. One can easily limit the report to specific libraries of interest by listing them on the command line.

How *sowhat* Works

Sowhat is currently written to analyze a Solaris 2.x environment. In this environment, the dynamic linker ld.so utilizes hard-coded library paths encoded into the executable program, as well as search requests that tell ld.so to scan a library path for a library matching a given name pattern and version. This scan checks all directories listed in the environment variable LD_LIBRARY_PATH. The function of ld.so is mimicked by the diagnostic program ldd, which reports the full pathnames of libraries that will satisfy each search request when ld.so is invoked prior to program execution.

To function properly, sowhat has to intimately understand the possible responses of the ldd command. The command ldd lists dynamic dependencies of executable files or shared objects. For example, for the executable file /local/bin/g++, ldd lists the path names of all shared objects that will be loaded whenever /local/bin/g++ is loaded, e.g.:

```
libc.so.1 =>  /usr/lib/libc.so.1
libdl.so.1 => /usr/lib/libdl.so.1
```

These records are *relative*; g++ asks for the first version of the library in the current library path matching the pattern libdl.so.1. This matches /usr/lib/libdl.so.1 .

The output of ldd can look quite different for vendor-supplied software. Consider the output of ldd for the tnshut command supplied with Sun's TotalNet software:

```
libdl.so.1 =>     /usr/lib/libdl.so.1
libsocket.so.1 =>/usr/lib/libsocket.so.1
libnsl.so.1 =>    /usr/lib/libnsl.so.1
libintl.so.1 =>   /usr/lib/libintl.so.1
libc.so.1 =>      /usr/lib/libc.so.1
libmp.so.2 =>     /usr/lib/libmp.so.2
/usr/platform/SUNW,
       Ultra-250/lib/libc_psr.so.1
```

The last library has an *absolute* binding. It must exist in exactly that place in the filesystem or the program will not function. In this case there is a good reason for the absolute binding as the existence of the library in question is dependent upon the sub-architecture of the particular machine. If it is not present, the system in question has the wrong architecture to run the software!

In order to find failures, sowhat must also understand the meaning of the various and sundry error messages provided by ldd. These include:

1. No match to library pattern:
   ```
   libucb.so.1 => (file not found)
   ```
2. Correct version not found:
   ```
   libm.so.1 (SUNW_1.1) =>
           (version not found)
   ```

In both cases, sowhat will record the actual library name as NotHere. If you then ask sowhat to list the programs that use the library NotHere, it will list all programs that cannot run due to missing libraries.

Environmental Awareness

Alas, any such script must be cognizant of the detailed structure of the user's environment – a matter of local operating policy. In particular, sowhat must be able to reconstruct any environment available to a user in order to test for problems within each one.

At Tufts EECS, we utilize a simple derivative of software module management [4, 5, 6] to allow users to dynamically add modules to their environment, modifying both their PATH and LD_LIBRARY_PATH as needed. Users need only type the shell command:

```
use packname
```

(where packname is the name of the package) in order to modify their environments for a specific package. We use this mechanism to allow users access to a variety of software that is not accessible except by specific request, such as vendor software for computer-aided design.

The 'use' command above invokes a package-specific startup script to modify the user's environment appropriately so that the desired package will work properly. For example, use new executes /local/env/new.cshrc:

```
set path = ($path[1-3] \
  /local/new/bin $path[4-])
setenv MANPATH \
  /local/new/man:${MANPATH}
setenv LD_LIBRARY_PATH \
  /local/new/lib:${LD_LIBRARY_PATH}
```

which has the effect of including the beta software testing tree /local/new in the user's PATH, MANPATH, and LD_LIBRARY_PATH.

The way the use command works is quite straightforward. The tcsh alias:

```
alias use \
  'set packages = ( \!* ) ; \
   source /local/lib/use'
```

sources the script /local/lib/use with $packages set to the appropriate package name:

```
#! /usr/bin/tcsh
if ($?packages) then
  foreach pkg ($packages)
    if (-f /local/env/$pkg.cshrc) then
      echo Using package $pkg : \
        setting up your environment
      source /local/env/$pkg.cshrc
    else
      echo There is no package $pkg : \
        please check your spelling.
    endif
  end
endif
unset packages
```

This script in turn sources a setup script from /local/env (starting with the package name) that sets PATH and LD_LIBRARY_PATH appropriately for the new module.

Analyzing Customizations

Using packages of this kind provides not only a way to avoid user confusion about commands most users do not need; it also provides a marvelous hiding place for library binding errors. To find library problems, one must analyze each package environment that the user can construct. Starting from each user's default environment, *Sowhat* constructs each custom environment available to users, one by one, and then analyzes effects of any additional libraries or executables. It does not test the effect of executing more than one 'use' at a time, though this would be helpful if not ridiculously time-consuming.

Results

Analyzing all packages available in a large system is a very time-consuming process. It takes between 10 and 20 minutes to analyze the configuration of a typical user on a Sun Enterprise-250 server, depending upon system load. Typically we invoke *sowhat* for data collection in background or overnight runs and store the results for later perusal and comparison. Runs of sowhat can be incremental or restarted from a previous failure. Sowhat is also capable of running in a differential mode in which it compares its recorded data against the system to detect potentially damaging changes.

Sowhat has educated us about our practices and the state of our program repositories in a way we could never have seen without it. It provides a previously unavailable window into our systems that informs us not only of potential problems, but also gives us a general overview of the health of our program repositories and the impact of our management practices.

Observed Problems

It was remarkable to us just how many things were wrong with our repositories. Using sowhat we detected the following kinds of problems:

1. Binary program invalid.
   a. Wrong subarchitecture.
   b. Wrong exec format.
2. Missing library.
   a. Nonexistent library.
      i. Due to absolute library pathname.
      ii. In all library path members.
   b. Incompatible library version.
      i. Due to absolute library pathname.
      ii. In all library path members.
   c. Incompatible library subarchitecture.
      i. Due to absolute library pathname.
      ii. In all library path members.

Rarely, the program we analyzed was not a Solaris program at all. Unbelievably, we found several Linux x86 programs installed in our Solaris repository!

A more subtle and serious error was that several programs in the correct exec format were compiled for a different hardware subarchitecture, e.g., 64-bit code on a 32-bit machine. These crept into our repository due to addition of 64-bit servers, while no one noticed that they could not be used on most of the workstations.

The remaining errors we found are the ones we were looking for. Several programs had outlasted their libraries by several years; we deleted libraries because we were completely unaware of the program's need for them. Other programs were compiled to hard-coded library locations, and these locations had been moved by operating system updates. Still others needed an older or newer version of the library than was available. Finally, some programs were made available on a machine having an inappropriate subarchitecture, which showed up in sowhat as an incompatible *library* subarchitecture.

Avoiding errors

Sowhat is very useful for analyzing the results of messy repository maintenance, but is equally useful in preventing the messes before they happen. Perhaps the nicest thing about sowhat is that one can ask it about the future impact of any change upon the user environment.

If one wishes to change or delete a library, sowhat can tell which programs will change in function or break based upon this change. One can then test these programs after the change to insure that they still work appropriately. If there are no such programs then the library may be deleted with no impact upon users.

If one wishes to delete a program, sowhat will suggest libraries that can be deleted along with it. These are the libraries that only the doomed program uses. So libraries never need to be kept around 'just in case' some program uses them. This greatly simplifies maintenance of repositories because they no longer need fill up with libraries that no program uses, just because it is unsafe to delete them without knowing which programs do.

Sowhat's differential mode not only notifies one of the effects of intentional library replacement, but also the effects of unintended or malicious changes. Unlike Tripwire [14] and Aide [7] it can detect not only a malicious change, but also identify its potential sphere of effect.

After operating system upgrades, sowhat can tell you which library bindings changed for which programs. This allows you to test those programs for possible problems created by the upgrade. One can also run it in 'differential mode' to compare the *user environments* on two hosts sharing the same command repository.

When we first ran sowhat, on one of our machines named andante, out of 9780 executables, we found 12 programs with missing libraries. Out of 61 packages, eight packages did not work for a variety of reasons. Some of the numbers generated by sowhat are a bit staggering: if we change /usr/lib/libc.so.1, 2237 executables will be affected!

By running sowhat on several different machines, we can determine the differences and inconsistencies in their user environments. For example, we discovered that libm.so.1(SUNW_1.1) is missing on andante, but present on other machines. On some machines we observed ''execution failed'' or ''Exec format error'' messages that were not seen on others. This is due to sub-architecture differences between the machines.

Lessons Learned

The main lesson that we learned from sowhat is that one cannot judge the impact of changing a dynamic library without some form of global analysis. Our tool does this analysis sufficiently well that one can rather accurately predict the sphere of possible effects of any potential change. Before we started using sowhat, we had already experienced several service failures due to library replacements, notably libz.so, which is used by a surprising variety of open source tools. This kind of potential effect was invisible to us before we wrote *sowhat*.

Alas, sowhat has several limitations. The first is that to analyze the user environment, sowhat must also know how that environment can change based upon user needs. This is a site-specific system property. In turn, to perform a complete analysis on a given site, sowhat must be updated to understand the mutations that can occur in the user environment at that site.

By far, the largest blind spot in sowhat is that it does not detect *conflicts* between user-invoked packages. It is quite possible that, by a specific sequence of environmental modifications, a user can produce a broken environment. The environment-modifying scripts can in principle do *anything*. There is no guarantee that one will not undo the good of another, and it is impractical to check all sequences of executions of these scripts. This is not a limit of sowhat, but one imposed by our environment and operating policy.

Future Work

Sowhat is a fairly closed-ended tool with a specific function that it performs quite well. We are unlikely to expand upon its basic functionality. However, we have already been begged to port this utility to Linux and this port is likely to become available in the near future. Other architectures are less likely as porting targets.

Several open questions may be attacked by future tools with differing scopes. It would be nice, e.g., to automate the process of comparing user environments and checking for homogeneity between various machines. This would be especially useful if it operated also in a *heterogeneous* environment, e.g., comparing commands and versions available to Solaris and Linux users.

Availability

Sowhat is freely available from http://www.eecs.tufts.edu/~couch/sowhat. It is a Perl script that requires access to a MySql database through the DBI and DBD::Mysql interfaces in order to record its results.

Acknowledgements

Many people contributed to the ideas in this project. The original idea came from a conversation with Steve Moshier about the difficulty of uninstalling software in a UNIX environment. Andy Davidoff and Michael Gilfix helped to define the problem, offered valuable insights, and provided valuable comments on the paper as it was being written.

Author Biographies

Alva L. Couch was born in Winston-Salem, North Carolina where he attended the North Carolina School of the Arts as a high school major in bassoon and contrabassoon performance. He received an S.B. in Architecture from M. I. T. in 1978, after which he worked for four years as a systems analyst and administrator at Harvard Medical School. Returning to school, he received an M.S. in Mathematics from Tufts in 1987, and a Ph.D. in Mathematics from Tufts in 1988. He became a member of the faculty of Tufts Department of Computer Science in the fall of 1988, and is currently an Associate Professor of Electrical Engineering and Computer Science at Tufts. Prof. Couch is the author of several software systems for visualization and system administration, including Seecube (1987), Seeplex (1990), Slink (1996) Distr (1997), and Babble (2000). In 1996 he also received the Leibner Award for excellence in teaching and advising from Tufts. He has assisted in maintaining the Tufts computer systems for Computer Science teaching and research since 1985, when he was a Ph.D. student. He can be reached by surface mail at the Department of Electrical Engineering and Computer Science, 161 College Avenue, Tufts University, Medford, MA 02155. He can be reached via electronic mail as couch@eecs.tufts.edu. His work phone is (617)627-3674.

Yizhan Sun is a Masters student at Tufts University who expects to graduate in December of 2001. She also holds an MS in Physics from Boston College. Aside from being a teaching assistant at Tufts, she was a system administrator in the Center for Connected Learning at Tufts from May 2000 to Aug 2000. In her spare time, she enjoys reading and swimming. She can be reached by email to ysun@eecs.tufts.edu, or by postal mail to 455 Boston TPKE Apt 4, Shrewsbury, MA, 01545.

References

[1] Bailey, E., *Maximum RPM*, Red Hat Press, 1997.

[2] Colyer, Wallace, and Walter Wong, "Depot: a Tool for Managing Software Environments," *Proc. LISA-VI*, Usenix Assoc., 1992.

[3] Couch, A., "SLINK: Simple, Effective Filesystem Maintenance Abstractions for Community-Based Administration," *Proc. Lisa-X*, Usenix Assoc, 1996.

[4] Elling, R., and M. Long, "Usersetup: A System for Custom Configuration of User Environments, or Helping Users Help Themselves," *Proc. LISA-VI*, Usenix Assoc., 1992.

[5] Evard, R. and R. Leslie, "Soft: A Software Environment Abstraction Mechanism" *Proc. LISA-VIII*, Usenix Assoc., 1994.

[6] Furlani, J. L., "Modules: Providing a Flexible User Environment," *Proc. LISA-VI*, Usenix Assoc., 1992.

[7] Lehti, Rama, "AIDE – Advanced Intrusion Detection Environment," http://www.cs.tut.fi/~rammer/aide.html.

[8] Manheimer, Kenneth, Barry Warsaw, Stephen Clark, and Walter Rowe, "The Depot: A Framework for Sharing Software Installation Across Organizational and UNIX Platform Boundaries," *Proc. LISA-IV*, Usenix Assoc., 1990.

[9] Rouillard, John P., and Richard B. Martin, "Depot-Lite: A Mechanism for Managing Software," *Proc. LISA-VIII*, Usenix Assoc., 1994.

[10] Sellens, John, "Software Maintenance in a Campus Environment: The Xhier Approach," *Proc. LISA-V*, Usenix Assoc., 1991.

[11] Wong, Walter C., "Local Disk Depot – Customizing the Software Environment," *Proc. LISA-VII*, Usenix Assoc., 1993.

[12] LDD man page, "man ldd," Sun Microsystems Inc.

[13] The Linux Standard Base Project, "The Linux Standard Base," http://www.linuxbase.org.

[14] Tripwire, Inc, "The Tripwire Security Scanner," http://www.tripwire.com.